UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LUCAS JOSÉ MERENCIA

INTEGRAÇÃO ENTRE TECNOLOGIAS DA PLATAFORMA JAVA EE COM O USO DE CDI

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2013

LUCAS JOSÉ MERENCIA

INTEGRAÇÃO ENTRE TECNOLOGIAS DA PLATAFORMA JAVA EE COM O USO DE CDI

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Everton Coimbra de Araújo.

Co-orientador: Prof. MEng. Juliano Rodrigo Lamb.

MEDIANEIRA

2013

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

UTFPR UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Integração entre tecnologias da plataforma Java EE com uso de CDI

Por

Lucas José Merencia

Este Trabalho de Diplomação (TD) foi apresentado às 13:50 horas do dia 23 de Agosto de 2013 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Manutenção Industrial, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado com louvor e mérito.

Prof. Dr. Everton Coimbra de Araújo
UTFPR – *Campus* Medianeira
(Orientador)

Prof. MEng. Juliano Rodrigo Lamb
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Ricardo Sobjak
UTFPR – *Campus* Medianeira
(Convidado)

Prof. MEng. Juliano Rodrigo Lamb
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

AGRADECIMENTOS

Primeiramente gostaria de agradecer a toda minha família, em especial aos meus pais, Genésio e Marli Merencia, pelo incentivo, por todos os ensinamentos para a vida e por me aconselharem em todos os momentos e decisões difíceis com que me deparei. A educação que recebi de vocês serve de base para tudo o que eu conseguir conquistar, incluindo esta graduação. Saibam que sem vocês nada disso seria possível.

Com grande apreço, gostaria de agradecer ao meu orientador professor Dr. Everton Coimbra de Araújo, não somente pela orientação neste trabalho, mas também por seu esplêndido trabalho com educador, sempre buscando desenvolver o melhor de seus alunos. Suas aulas e sua orientação me proporcionaram imenso conhecimento na área de informática e posso dizer com toda convicção que foi uma honra ser seu orientando.

Gostaria de agradecer também ao professor MEng. Juliano Rodrigo Lamb, pela sua dedicação e auxílio durante minha graduação e por ter aceitado ser meu co-orientador. Seus conhecimentos em engenharia de *software* disseminados em suas aulas e seu tempo concedido às minhas dúvidas, foram de grande importância em minha formação.

Obrigado também a todos o professores da área de informática da UTFPR que mesmos não citados aqui, colaboraram direta ou indiretamente em minha formação.

Ao meu grande amigo Giovani Guizzo, que me auxiliou não somente em assuntos acadêmicos mas também em assuntos pessoais. Nossas parcerias durante o curso renderam grandes frutos além de proporcionaram grande conhecimento na área de computação. Muito obrigado!

Gostaria de agradecer também ao meu amigo Carlos Alexandro Becker, que mesmo não tendo estudando na mesma turma, tornou-se um grande amigo com quem tive diversos debates e trocas de ideias que me auxiliaram muito durante a graduação.

Por ultimo e não menos importante, gostaria de agradecer a minha noiva, companheira e amiga Layenne Fernanda Albring Prado, pelo seu carinho, apoio e compreensão. Obrigado por tudo!

RESUMO

MERENCIA, Lucas José. INTEGRAÇÃO ENTRE TECNOLOGIAS DA PLATAFORMA JAVA EE COM O USO DE CDI. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas). Universidade Tecnológica Federal do Paraná. Medianeira 2013.

Durante o desenvolvimento de um aplicação utilizando a plataforma Java EE o desenvolvedor pode usufruir das diversas especificações disponíveis nesta plataforma. Uma delas é o CDI, que provê um conjunto de serviços que quando combinados facilitam o desenvolvimento deste tipo de aplicação. Inicialmente o CDI foi projetado para uso com objetos *Stateful*, porém esta tecnologia provê usos mais amplos, permitindo a interação entre vários tipos de componentes de uma maneira flexível. Desta maneira é possível utilizá-lo como uma camada de integração entre as diversas tecnologias da plataforma Java EE unificando escopos e ciclos de vidas. O presente trabalho tem como finalidade demonstrar as principais características e recursos do CDI e como aplicá-los para prover uma camada de integração e unificação entre algumas das principais tecnologias da plataforma Java EE.

Palavras-Chave: Injeção de Dependências, Modularização, Aplicações Empresariais.

ABSTRACT

MERENCIA, Lucas José. INTEGRAÇÃO ENTRE TECNOLOGIAS DA PLATAFORMA JAVA EE COM O USO DE CDI. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas). Universidade Tecnológica Federal do Paraná. Medianeira 2013.

During the development of an application which uses the Java EE platform, the developer can take advantage of various specifications available on this platform. One of them is CDI, which provides a set of services that when combined may facilitate the development of this kind of application. CDI was initially designed for use with stateful objects, but this technology provides broader uses, allowing the interaction between various components in a flexible manner. Thus, it is possible to use it as an integration layer between the various technologies which the Java EE platform provides, by unifying life cycles and scopes. The present paper aims to demonstrate the main characteristics and features of CDI and how to apply them to provide a layer of integration and unification of some key technologies of the Java EE platform.

Keywords: Dependency Injection, Modularization, Enterprise Applications.

LISTA DE SIGLAS

API	*Application Programming Interface*
CDI	*Context and Dependency Injection*
CORBA	*Common Object Request Broker Architecture*
EJB	*Enterprise JavaBeans*
EL	*Expression Language*
FTP	*File Transfer Protocol*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
Java EE	*Java Enterprise Edition*
JAX-RS	*Java API for RESTful Web Services*
JAX-WS	*Java API for XML Web Services*
JCP	*Java Community Process*
JDBC	*Java Database Connectivity*
JMS	*Java Message Service*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
JSF	*JavaServer Faces*
JSON	*JavaScript Object Notation*
JSP	*JavaServer* Pages
JSR	*Java Specification Request*
MDB	*Message-Driven Bean*
ORM	*Object-Relational Mapping*

RMI	*Remote Method Invocation*
SOAP	*Simple Object Access Protocol*
SPI	*Service Provider Interface*
SQL	*Structured Query Language*
WSDL	*Web Services Description Language*
XML	*eXtensible Markup Language*

LISTA DE FIGURAS

Figura 1 - História do Java EE..15
Figura 2 - Diagrama de Componentes Carrinho de Compras sem CDI. 33
Figura 3 - Diagrama de Componentes Carrinho de Compras com CDI..................................36
Figura 4 - Diagrama de sequência para persistência de um novo produto. 38
Figura 5 - Diagrama de classe download de lista de produtos. 45
Figura 6 - Arquivo CSV gerado. 49

LISTA DE QUADROS

Quadro 1 - Escopos dos *Managed Beans* 18
Quadro 2 - Injetando um *Bean.* 24
Quadro 3 - Exemplo de *Named Bean.* 25
Quadro 4 - *Qualifier Premium.* 26
Quadro 5 - Injetando um *Customer Premium.* 26
Quadro 6 - Escopos dos *Named Beans.* 27
Quadro 7 - Arquivo beans.xml. 28
Quadro 8 - Implementação do EJB ProductController. 34
Quadro 9 - Implementação do *ManagedBean* ProductManagedBean. 34
Quadro 10 - Componente JSF para listagem de produtos. 35
Quadro 12 - Componente JSF para listagem de produtos referenciando um EJB. 37
Quadro 13 - Método produtor de EntityManager. 39
Quadro 14 - InterceptorBinding Transactional. 40
Quadro 15 - Interceptador TransactionInterceptor. 40
Quadro 16 - Declaração do Interceptor no bean.xml. 41
Quadro 17 - Implementação da Interface DAO. 42
Quadro 18 - Implementação de DAO para a entidade Product. 42
Quadro 19 - Injeção da classe ProductDao em uma classe de negócio. 43
Quadro 20 - Implementação do *Qualifier* ProductQualifier. 46
Quadro 21 - Product Dao Qualificado. 47
Quadro 22 - Servlet de download da lista de produtos. 48
Quadro 23 - Declaração do Servlet no web.xml. 48

SUMÁRIO

1.1 OBJETIVO GERAL 11

1.2 OBJETIVOS ESPECÍFICOS 11

1.3 JUSTIFICATIVA 11

1.4 ESTRUTURA DO TRABALHO 13

2.1 JAVA EE 14

2.2 SERVLETS 16

2.3 JAVA SERVER FACES 16

2.4 ENTERPRISE JAVA BEANS 18

2.4.1 Tipos de EJBs 19

2.4.2 O Container EJB 19

2.5 JAVA PERSISTENCE API 21

2.6 CONTEXT AND DEPENDENCY INJECTION 21

2.6.1 Beans 23

2.6.2 Injetando *Beans* 24

2.6.3 *Named Beans* 24

2.6.4 *Qualifiers* 25

2.6.5 Escopos dos *Beans* 26

3.1 WELD 28

3.2 GLASSFISH 29

3.3 NETBEANS 29

3.4 ECLIPSELINK 30

3.5 UML 30

3.6 ASTAH COMMUNITY 30

4.1 INTEGRAÇÃO JSF E EJB COM CDI 32
4.1.1 Implementação sem CDI 32
4.1.2 Implementação com CDI 35
4.2 CRIANDO UMA CAMADA DE PERSISTÊNCIA COM JPA E CDI 37
4.3 UTILIZANDO CDI COM SERVLETS 44
5.1 CONCLUSÃO 50
5.2 TRABALHOS FUTUROS 51

1 INTRODUÇÃO

No desenvolvimento de sistemas, inúmeras vezes são necessárias a utilização de várias tecnologias para que os requisitos levantados na análise sejam implementados. A integração destas tecnologias pode ser complexa, pois podem trabalhar com contextos e ciclos de vida distintos. Desta maneira, a codificação da infraestrutura do *software* pode se tornar trabalhosa, pois essas tecnologias operam de maneiras diferentes e com isso poderá ser gerada uma grande quantidade de código para adaptação entre as mesmas.

Tendo em vista os sistemas corporativos e sistemas *Web*, o ambiente Java fornece uma série de tecnologias e especificações unificadas em uma única plataforma atendendo as necessidades desta categoria de *software*. Esta plataforma é denominada *Java Enterprise Edition* (Java EE). Segundo Jendrock *et. al.* (2010, p. 4), o objetivo mais importante da Java EE é simplificar o desenvolvimento, fornecendo uma estrutura comum entre os varios tipos de componentes Java EE.

Uma das especificações Java EE estabelecida pela *Java Community Process*[1] (JCP) é a Java *Specification Request*[2] (JSR) 299, que define o *Contexts and Dependency Injection* (CDI). Segundo Jendrock *et. al.* (2010, p. 30), esta especificação define um conjunto de serviços contextuais, fornecidos pelo container Java EE, que facilitam o uso de *Enterprise JavaBeans* (EJBs) com páginas J*avaServer Faces* (JSF) em aplicações *Web*.

Este trabalho visa a utilização do CDI para aperfeiçoar sistemas que utilizam algumas das principais tecnologias da plataforma Java EE, como EJBs e componentes JSF. Desta maneira, os recursos oferecidos pelo CDI podem fazer a integração entre estas tecnologias abstraindo o código estrutural, que até então pode estar sendo implementado de uma maneira não muito eficiente.

---

[1] *Java Community Process* (JCP) é um processo para o desenvolvimento de especificações técnicas de padrões para a tecnologia Java. (JCP, 2012).
[2] Java *Specification Request* (JSR) é uma requisição com o propósito de definir uma especificação Java. Segundo Goncalces (2010, p. 3), quando uma nova especificação for necessária é criada uma nova JSR e é formado um grupo de peritos, composto por representantes de companhias, organizações, universidades ou indivíduos particulares. Sendo que este grupo será responsável pelo desenvolvimento da nova JSR.

1.1 OBJETIVO GERAL

Aplicar a especificação do CDI para prover a integração de componentes da plataforma Java EE, visando a aperfeiçoar o desenvolvimento de sistemas baseados nesta plataforma, por meio de 3 exemplos.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos do projeto são:

- Apresentar técnicas de desenvolvimento de *software* modularizado e reutilizável, entre os componentes Java EE.
- Realizar a integração de tecnologias Java EE como JSF, EJBs e *Servlets* utilizando um *framework* que implemente as especificações do CDI.
- Utilizar CDI para abstrair o código estrutural de sistemas, utilizando injeção de dependências.

1.3 JUSTIFICATIVA

As preocupações no desenvolvimento de *software* não se limitam à programação e tecnologias utilizadas. Destas preocupações podem-se destacar a manutenibilidade e reutilizabilidade do código, classes, módulos e componentes coesos e desacoplados. Enfim, todos os artefatos que auxiliam na otimização do *software*.

Tendo em vista os desafios encontrados na utilização de tais fatores, a utilização das tecnologias disponibilizadas pela plataforma Java EE pode ser uma alternativa viável, pois elas são documentadas na forma de especificações. Estas especificações definem como um determinado problema do desenvolvimento do *software* será solucionado na tecnologia Java.

Quando se desenvolve um *software* utilizando as diversas tecnologias englobadas na plataforma Java EE, tem-se algumas opções para implementar uma determinada camada do sistema. Na camada de negócio, uma das tecnologias amplamente utilizadas são os EJBs (*Enterprise Java Beans*). Segundo Bond *et. al.* (2003, p. 33), a lógica de negócio poderá ser encapsulada em EJBs. É possível utilizar outros componentes ou objetos Java puros para isto, porém os EJBs fornecem uma maneira conveniente de encapsular e compartilhar a lógica de

negócio comum, e assim tirar proveito dos serviços oferecidos pelos containers EJB ou servidores de aplicação.

Já na camada de visualização, ou na interface com o usuário, pode-se optar pela utilização de *Servlets*, *Java Server Pages* (JSP) ou ainda JSF. Segundo Araújo (2010, p. 51), *Servlets* são classes Java que são executadas no servidor de aplicação, que por sua vez são requisitados pelos clientes, realizam algum processo e ao término deste processo, retornam uma resposta ao cliente. Os *Servlets* podem atender requisições *Hypertext Transfer Protocol* (HTTP) que retornam páginas *HyperText Markup Language* (HTML), contudo não se limitam a isto, pois podem, por exemplo, atender requisições *File Transfer Protocol* (FTP).

As páginas JSP permitem incorporar em páginas HTML, *tags* JSP conhecidas como *Scriptlets,* que possibilitam a utilização de código Java dentro destas páginas. Segundo Araújo (2010, p. 89-91), as páginas JSP são na realidade *Servlets*, e estas, diferente das páginas HTML que exibem informações estáticas, permitem o dinamismo. Já as páginas JSF abstraem o código Java utilizado na camada de apresentação. Araújo (2010, p. 143) relata que JSF é uma especificação criada pela JCP que oferece recursos para viabilizar uma produtividade maior para aplicações *Web*.

Estas tecnologias são apenas algumas das disponibilizadas pela plataforma Java EE para o desenvolvimento de aplicações corporativas, que servem como facilitadores para os desenvolvedores. Sabe-se que cada uma destas tem seu modo de utilização e podem atender a necessidades distintas em uma aplicação. Contudo, estas tecnologias necessitam de integração, a qual algumas vezes pode ser dificultada, pois implementam seus contextos e ciclos de vidas, cada uma a seu modo e desta maneira, por exemplo, o contexto de um EJB pode não existir no ciclo de vida de uma página JSF e vice-versa.

Uma outra tecnologia contida na plataforma Java EE é o CDI, e esta propõem a unificação dos contextos utilizados pelas diversas tecnologias da plataforma. Desta forma, torna-se plausível a utilização do CDI como uma camada de abstração que provê a comunicação entre os demais recursos disponíveis. Abstraindo a camada de integração espera-se que o acoplamento entre as classes da aplicação seja reduzido, aumentando assim a manutenibilidade e reutilizabilidade do código.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho é composto por um total de cinco capítulos. No primeiro capítulo é feita uma introdução ao assunto abordado, os objetivos a que se destina e também uma justificativa. No segundo capítulo é apresentada a fundamentação teórica, onde primeiramente são abordadas algumas das principais tecnologias da plataforma Java EE e suas definições, juntamente com os conceitos necessários sobre CDI. No terceiro capítulo são apresentados os materiais é métodos utilizados no desenvolvimento deste trabalho. O quarto capítulo consiste no desenvolvimento do trabalho, sendo abordadas as aplicabilidades do CDI acompanhadas de um exemplo prático e como fazer sua implementação. E por fim, no quinto capítulo, são apresentadas as conclusões sobre a utilização de CDI e ainda as ideias para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordadas as tecnologias e conceitos necessários para a compreensão e embasamento do objetivo proposto.

2.1 JAVA EE

Segundo Goncalves (2010, p. 2-3), a plataforma Java EE foi inicialmente chamada de J2EE, desenvolvida pela Sun e liberada em 1999 contendo 10 Java *Specification Requests* (JSRs). Inspirado pelo *Common Object Request Broker Architecture*[3] (CORBA), foi criada tendo em vista sistemas distribuídos. Em 2003 o J2EE incluía 20 especificações e teve adicionado suporte à *Web Services*. Nos anos subsequentes, a plataforma foi vista como um modelo de componentes amplo, com dificuldade na realização de testes, distribuição e execução. Entretanto, no segundo trimestre de 2006, o Java EE 5 (JSR 244) foi liberado e se mostrou com uma melhoria notável.

Segundo Goncalves (2010, p. 5), a versão 6 da plataforma Java EE (JSR 316) contém outras 33 especificações. Esta versão veio com um conjunto de inovações, tais como: o Java API *for RESTful Web Services*[4] (JAX-RS); simplificação de *Application Programming Interface* (APIs), como EJB 3.1 e melhorias de outras como *Java Persistence* API (JPA) 2.0. Contudo as principais modificações foram as relacionadas com portabilidade, à depreciação de algumas especificações e a criação de subconjuntos da plataforma, por meio de perfis. Recentemente (28 de Maio de 2013) foi lançada a versão 7 da plataforma Java EE, nesta versão além de melhorias em APIs já existentes foram adicionadas novas funcionalidades visando o aumento de produtividade com tecnologias com HTML5 e WebSockets. A Figura 1 apresenta a trajetória resumida do progresso da plataforma Java EE (até a versão 6).

---

[3] O *Common Object Request Broker Architecture* (CORBA) é um modelo de arquitetura de objetos distribuídos da *Object Management Group* (OMG). O CORBA é um *middleware* aberto composto de objetos distribuídos, que define um padrão de desenvolvimento de aplicações distribuídas para ambientes heterogêneos, provendo a interoperabilidade entre os objetos de forma transparente, reusabilidade e portabilidade (SERRA, 2004).

[4] Segundo a JCP (2011), JAX-RS é uma especificação definida pela JSR 311, esta descreve uma API que provê suporte a criação de RESTful (Transferência de Estado Representacional) *Web Services* na Plataforma Java.

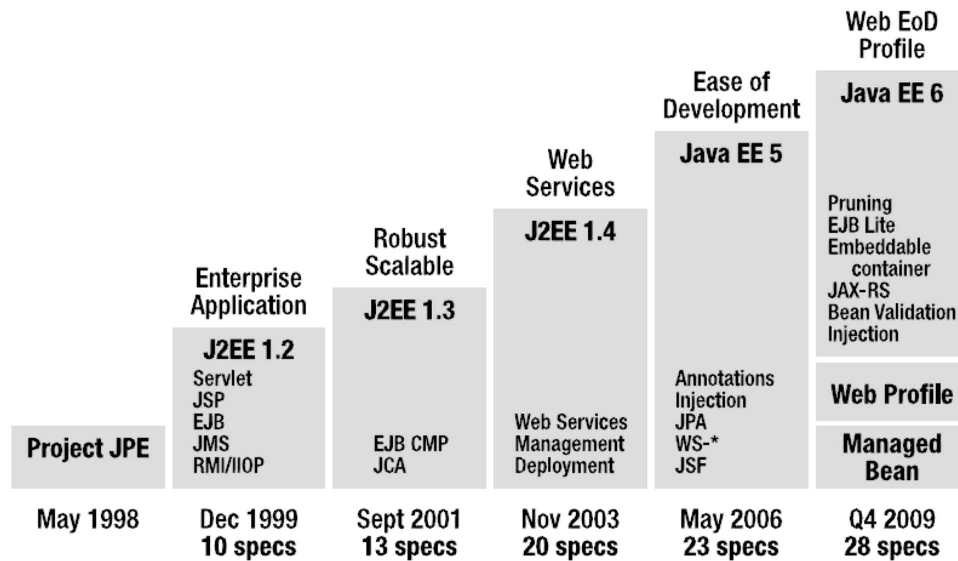

Figura 1 - História do Java EE.
Fonte: GONCALVES (2010, p. 2).

Segundo Goncalves (2010, p. 4), Java EE é um conjunto de especificações implementado por diferentes containers. GONCALVES (2010, p. 4) explica que os containers são ambientes de execução de aplicações Java EE que proveem determinados serviços para os componentes, como: gerenciamento de ciclo de vida; injeção de dependência; gerenciamento de transações; entre outros.

As aplicações Java EE normalmente são baseadas em componentes que são desenvolvidos e adequados às especificações. Segundo Jendrock, *et al.* (2010, p. 13), quando se trabalha com aplicações baseadas em componentes para a plataforma Java EE, o desenvolvimento é facilitado, pois a lógica de negócio é organizada em componentes reutilizáveis.

Para Goncalves (2010, p. 5 - 6), a plataforma Java EE define quatro tipos de componentes que suas implementações devem suportar:

- ***Applets*****:** É uma interface gráfica que é executada no navegador. Este tipo de componente é caracterizado por utilizar a API *Swing* para desenvolver suas interfaces;
- **Aplicações:** São programas executados no cliente. Normalmente são interfaces gráficas ou processamentos em lote que têm acesso a todos os serviços dos componentes da camada intermediária das aplicações Java EE, estes serviços normalmente são referentes à camada de negócio;

- **Aplicações *Web*:** São aplicações executadas no container *Web*, que atendem requisições HTTP de clientes *Web* (como navegadores) e podem, por exemplo, responder a estas com HTML, *eXtensible Markup Language* (XML) ou ainda *JavaScript Object Notation* (JSON);
- **Aplicações Corporativas:** São aplicações executadas em um container EJB, os quais são componentes para tratar a lógica de negócio que podem ser acessados localmente e remotamente por meio de *Remote Method Invocation* (RMI), HTTP, *Simple Object Access Protocol* (SOAP) e *Web Services* RESTful;

2.2 SERVLETS

Segundo Araújo (2010, p. 51), *Servlets* são classes Java que são executadas em um Container *Web*. Um *Servlet* é requisitado por um cliente, recebendo algumas informações enviadas por este para que possa realizar algum processamento e ao termino desse processamento é retornada uma resposta ao cliente. Para Heffelfinger (2010, p. 39), um *Servlet* é usado para estender as capacidades do servidor onde a aplicação é hospedada, atendendo a requisições e gerando uma resposta.

Heffelfinger (2010, p. 39) aponta que a classe base para todos os *Servlets* é `GenericServlet`. Esta classe define um *Servlet* genérico. Entretanto, o tipo mais comum de *Servlet* é o *HTTP Servlet*, que atende as requisições e gera as respostas para o protocolo HTTP. Este tipo de *Servlet* estende a classe `HttpServlet`, que é uma subclasse de `GenericServlet`**.** Contudo, segundo Araújo (2010, p. 51), é importante salientar que um *Servlet* não atende somente requisições relacionadas às páginas *Web*, pois, por exemplo, é possível implementar um *Servlet* que atenda a requisições FTP.

2.3 JAVA SERVER FACES

Segundo Jendrock, *et al.* (2010, p. 73), a tecnologia JSF (*Java Server Faces*) é um *framework* de componentes para desenvolvimento de aplicações baseadas na *Web*. Este *framework* simplifica a construção e a manutenção de páginas *Web*, pois provê um modelo de programação bem definido e várias bibliotecas de componentes e *tags*.

As principais facilidades providas pela utilização de JSF são descritas por Jendrock *et al.* (2010, p. 73). Os autores explicam que a tecnologia JSF consiste em:

- Uma API para: representar componentes e gerenciar seus estados; tratamento de eventos do lado do servidor; validação e formatação de dados; definição de navegação entre páginas; suporte à internacionalização e acessibilidade; e prove extensibilidade para todas estas características.
- Bibliotecas de *tags* para adicionar componentes em paginas *Web* e conectar estes componentes com objetos que estão no lado do servidor.

Segundo Araújo (2010, p. 143 - 147), a utilização de JSF simplifica uma das maiores dificuldades ligada ao desenvolvimento *Web*, pois utilizando esta tecnologia delega-se a responsabilidade de apresentação do conteúdo apenas às páginas. Estas páginas utilizam as bibliotecas de *tags* e incorporam os componentes JSF, o que permite uma maior produtividade no desenvolvimento de aplicações *Web*.

Para fazer a comunicação entre cliente e servidor, os componentes JSF podem ser conectados a objetos no servidor conhecidos como *Managed Beans*. Segundo Araújo (2010, p. 150 - 151), a responsabilidade de um *Managed Bean* é intermediar a comunicação entre as páginas e o modelo, sendo que os componentes se conectam com estes por meio de *Expression Language* (EL), relacionando-se com o atributo *value* das *tags* presentes nas páginas.

> *Managed Bean* pode ser visto como um objeto Java criado, inicializado e disponibilizado para a aplicação. É preciso que o usuário saiba que o JSF faz uso da inicialização tardia do modelo, isso significa que o *bean*, em escopo especifico, é criado e inicializado não quando o escopo é criado, mas sim, por demanda, ou seja, quando o *bean* é requisitado pela primeira vez. (ARAÚJO, 2010, p. 151).

Segundo Heffelfinger (2010, p. 216), um *Managed Bean* sempre possui um escopo que define o seu tempo de vida em uma aplicação. Na versão 2.0 do JSF os escopos podem ser definidos por *annotations*. Se não for definido nenhum escopo em um *Managed Bean*, por padrão o escopo utilizado é o `@RequestScoped`. O Quadro 1 lista e descreve todos os possíveis escopos desta versão, explicados por HEFFELFINGER.

Annotation* de escopo de *Managed Bean	**Descrição**
`@ApplicationScoped`	A mesma instância de um *Managed Bean* é disponível para todos os clientes da aplicação. Se um cliente modificar o valor de um *Managed Bean* de aplicação, a alteração é refletida em todos os clientes.

`@SessionScoped`	Uma instância de cada *Managed Bean* de sessão é atribuída para cada cliente da aplicação. Um *Managed Bean* de sessão pode ser usado para armazenar os dados específicos do cliente nas requisições.
`@RequestScoped`	Um *Managed Bean* com escopo de requisição tem seu ciclo de vida definido apenas durante a requisição HTTP.
`@ViewScoped`	*Managed Beans* com escopo de *view* (visão) são associados com uma página, eles são destruídos quando o usuário navegar para outra página.
`@NoneScoped`	*Managed Beans* sem escopo são instanciados quando acessados por outro *Managed Bean*, normalmente como uma propriedade gerenciada.
`@CustomScoped`	JSF 2.0 introduz a possibilidade de criação de escopos personalizados, a *annotation @CustomScoped* deve ser resolvida para um mapa no escopo da sessão.

Quadro 1 - Escopos dos *Managed Beans*
Fonte: Adaptado de HEFFELFINGER (2010, p. 216).

2.4 ENTERPRISE JAVA BEANS

Segundo Heffelfinger (2010, p. 333), os EJBs (*Enterprise Java Beans*) são componentes, que ficam no lado servidor e encapsulam a lógica de negócio. Além disso os EJBs simplificam o desenvolvimento de aplicações, pois, por exemplo, o gerenciamento de transações com o banco de dados pode ser tratado automaticamente pelo container EJB.

Goncalves (2010, p. 180) explica que utilizando EJBs os desenvolvedores podem se concentrar na implementação da lógica de negócio, enquanto o container EJB fica responsável pela execução e disponibilização dos módulos. Dessa maneira, um módulo EJB pode ser desenvolvido uma única vez e implantado em qualquer container que de suporte a especificação.

Nas subseções seguintes serão detalhados os tipos de EJBs suportados pela plataforma Java EE, sendo abordado também o container EJB, ambiente de execução dos EJBs.

2.4.1 Tipos de EJBs

Como as aplicações corporativas podem ser complexas, normalmente é necessário que algumas partes específicas da aplicação possuam contextos diferentes de outras. Por exemplo, um carrinho de compras deverá manter o estado durante toda a interação com o usuário, já um cadastro de produto não, pois em um cadastro os dados informados apenas são persistidos no banco de dados e recuperados quando necessário. Goncalves (2010, p. 181) explica que a plataforma Java EE define três tipos de EJBs e ressalta que, *Session Beans* são, efetivamente os componentes EJBs responsáveis por encapsular a lógica de negócio, o que os torna a parte mais importante da tecnologia EJB. Um *Session Bean* pode ser:

- *Stateless:* O *Session Bean* não mantém nenhum estado de comunicação entre requisições, e qualquer instância pode ser usada por qualquer cliente;
- *Stateful:* O *Session Bean* mantém o estado da comunicação. Este estado deve ser mantido entre requisições de um usuário;
- *Singleton:* Um único *Session Bean* é compartilhado entre todos os clientes e suporta o acesso concorrente.

Na tecnologia EJB, além dos *Session Beans*, há outro tipo de *Bean*: os *Message-Driven Beans* (MDBs). Segundo Jendrock, *et al.* (2010, p. 25), este tipo de *Bean* combina as características dos *Session Beans* com *Message Listeners*, possibilitando assim que componentes corporativos recebam mensagens assíncronas. Geralmente, estes *Beans* atuam como *listeners* de mensagens JMS (*Java Message Service*).

2.4.2 O Container EJB

Como os EJBs são componentes do lado servidor, é necessário que sejam executados em um container. Segundo Goncalves (2010, p. 206), este ambiente de execução oferece funcionalidades centrais comuns a muitas aplicações Java EE, tais como:

- **Comunicação remota com o cliente**: um cliente EJB pode invocar métodos remotamente por meio de protocolos padrões;
- **Injeção de dependências**: o container pode injetar vários recursos em um EJB, como por exemplo outros EJBs;

- **Gerenciamento de estado**: para *beans Statefull*, o container gerencia transparentemente seu estado. Sendo possível manter o estado de um cliente em particular, como se estivesse desenvolvendo uma aplicação desktop;
- **Agregação**: para MDBs e *beans Stateless*, o container cria grupos de instâncias (*pool*) que podem ser compartilhados por múltiplos clientes. Uma vez invocado, um EJB retorna ao *pool* para ser reutilizado, ao invés de ser destruído.
- **Ciclo de vida do componente**: o container é responsável por gerenciar o ciclo de vida de cada componente;
- **Sistema de mensagens**: o container permite que MDBs escutem destinatários e consumam mensagens sem excesso de conexão JMS;
- **Gerenciamento de transação**: com o gerenciamento de transações, um EJB pode usar *annotations* para informar ao container sobre a política de transações que ele deve usar, ou seja, o container é responsável pelo *commit* ou pelo *rollback* da transação;
- **Segurança**: o controle de acesso em níveis de classes ou de métodos pode ser especificado nos EJBs para reforçar a autenticação de funções e de usuário;
- **Suporte a concorrência**: com exceção dos *Singletons*[5], onde alguma declaração de concorrência é necessária, todos os outros tipos de EJBs têm segurança de segmento por natureza. Com isso pode-se construir aplicações de alto desempenho sem se preocupar com questões de concorrência de acesso;
- **Interceptadores**: Podem ser definidos comportamentos em interceptadores e reutilizados em pontos distintos da aplicação, os quais serão invocados automaticamente pelo container.
- **Invocação assíncrona de métodos**: com EJB 3.1 é possível utilizar chamadas assíncronas sem o envolvimento do uso de mensagens.

Gonçalves (2010, p. 206) afirma que: “Depois que o EJB é distribuído, o container cuida dessas funcionalidades, deixando o desenvolvedor se concentrar na lógica funcional, enquanto se beneficia desses serviços sem acrescentar nenhum código a nível de sistema”. Desta maneira, os esforços aplicados no desenvolvimento podem ser direcionados unicamente para atender os reais problemas dos clientes.

---

[5] *Singleton* é um padrão de projetos do catálogo GoF, Este padrão garante que uma classe tenha somente uma instância e fornece um ponte global de acesso para a mesma (GAMMA, HELM, *et al.*, 2000, p. 130).

2.5 JAVA PERSISTENCE API

A JPA (Java *Persistence* API) foi criada para solucionar as dificuldades do mapeamento objeto relacional do Java. Segundo Goncalves (2010, p. 50) a JPA pode ser utilizada para acessar e manipular dados relacionais a partir de EJBs, componentes *Web* e até mesmo em aplicações Java SE.

Esta tecnologia provê uma camada de abstração sobre os JDBCs (Java *Database Connectivity*), o que a torna independente da *Structured Query Language* (SQL). Para Goncalves (2010, p. 50), os principais componentes da JPA são:

- O *Object-Relational Mapping* (ORM), que é o mecanismo para mapeamento de objetos para dados relacionais;
- Uma API gerenciadora de entidades para realizar operações relacionadas com a base de dados, como criar, ler, atualizar e excluir dados;
- A *Java Persistence Query Language* (JPQL), que permite a criação de *queries* com uma linguagem de consultas orientada a objetos;
- Mecanismos de transação e bloqueio de acesso concorrente a dados;
- *Callbacks* e *Listeners* para adição de lógica funcional no ciclo de vida de um objeto persistente.

2.6 CONTEXT AND DEPENDENCY INJECTION

Segundo Jendrock, *et al.* (2010, p. 305), *Contexts and Dependency Injection* (CDI) é um grupo de serviços que, quando usados em conjunto, facilitam o desenvolvimento de aplicações *Web* que utilizam a tecnologia EJB com a tecnologia JSF. Inicialmente projetado para uso com objetos *Stateful*, o CDI provê usos mais amplos, permitindo a interação entre vários tipos de componentes de uma maneira flexível, mas *typesafe*.

Jendrock *et al.* (2010, p. 305) afirmam que CDI é especificado pela JSR 299, e inclui as especificações:

- JSR 330, *Dependency Injection for Java*;
- JSR 316, especificação que define os *Managed Beans*.

Segundo Jendrock *et al.* (2010, p. 306) os principais serviços fornecidos pelo CDI são:

- **Contextos:** É a capacidade de vincular o ciclo de vida e interações de componentes *Stateful* com ciclos de vida bem definidos, porém com contextos extensíveis.
- **Injeção de dependência:** É a capacidade de injetar componentes em uma aplicação de forma *typesafe*, incluindo a habilidade de escolher qual implementação de uma determinada *interface* será injetada.

Jendrock *et al.* (2010, p. 306) abordam também os serviços adicionais proporcionados pela especificação do CDI. Estes adicionais ampliam as capacidades da tecnologia, pois muitos destes agregam grande valor à especificação e trazem muitos facilitadores para os desenvolvedores. Estes adicionais são:

- Integração com *Expression Language (EL)*, que possibilita a utilização de qualquer componente, diretamente em páginas JSF ou JSP;
- A capacidade de decorar componentes injetados, ou seja, CDI implementa o padrão de projetos *Decorator*[6], que pode ser utilizado junto com a injeção de dependência;
- A capacidade de associar interceptadores com componentes usando *typesafe*, assim pode-se definir interceptadores e utilizar *annotations* para indicar o que deverá ser interceptado por eles;
- Um modelo de notificação de eventos, sendo que, este modelo implementa o padrão de projetos *Observer*[7];
- Adiciona o escopo de conversação (`@ConversationScoped`) aos três escopos definidos na especificação Java *Servlet*;
- Um *Service Provider Interface* (SPI) que possibilita a integração de *frameworks* de terceiros de forma limpa no ambiente Java EE.

Considerando a essência do CDI e os serviços adicionais oferecidos por ele, Jendrock *et al.* (2010, p. 306) enfatizam que, uma das principais vantagens proporcionada pela tecnologia é o baixo acoplamento. Utilizando CDI, o cliente é desassociado do servidor, para isso são criados tipos bem definidos e qualificadores, e estes, aplicados de tal maneira que, a implementação do servidor pode variar sem a necessidade de reescrever código no cliente. A

---

[6] *Decorator* é um padrão de projetos do catalogo GoF, este padrão agrega responsabilidades adicionais a um objeto, fornecendo uma alternativa flexível ao uso de subclasses para extensão de funcionalidades (GAMMA, HELM, *et al.*, 2000, p. 170).

[7] *Observer* é um padrão de projetos do catalogo GoF, que define uma dependência um para muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente (GAMMA, HELM, *et al.*, 2000, p. 274).

tecnologia desacopla os contextos e ciclos de vida dos componentes, permitindo a utilização de componentes *Stateful* como serviços que trabalham puramente com troca de mensagens, sendo que estes componentes possuem ciclos de vida gerenciados automaticamente pelo container de aplicação.

Para prover este baixo acoplamento, o CDI também aborda de maneira eficaz: (a) o tratamento de eventos, o que desacopla os produtores dos responsáveis pelo tratamento dos eventos; (b) desacopla implementações adicionais que atendem à preocupações distintas, porém devem ser tratadas juntamente com uma determinada implementação, por meio de interceptadores que podem executar diversos processamentos, sem que a implementação principal tenha conhecimento dos mesmos.

A utilização de CDI elimina completamente os *lookups* baseados em *strings* e com isso o compilador pode detectar erros de digitação (JENDROCK, *et al.*, 2010, p. 306). Antes do CDI, os erros de *lookup* eram constatados apenas em tempo de execução, diferente do que ocorre quando utiliza-se o CDI, pois o compilador não encontrará o componente em tempo de compilação. O autor ainda afirma que com o uso de anotações, é possível especificar quase tudo, eliminando grande parte dos XMLs descritores de implementação, o que facilita a compreensão da estrutura de dependências em tempo de desenvolvimento.

2.6.1 Beans

CDI redefine o conceito de *bean* para além de sua utilização em outras tecnologias Java, como os *JavaBeans* e *Enterprise JavaBeans*. No CDI um *bean* é a origem dos objetos contextuais que definem o estado ou a lógica da aplicação (JENDROCK, *et al.*, 2010, p. 307).

Segundo King *et al.* (2012) um bean abrange os seguintes atributos:

- Um conjunto (não vazio) de tipos de *beans*;
- Um conjunto (não vazio) de qualificadores (*qualifiers*);
- Um escopo;
- Opcionalmente, um nome EL do *bean*;
- Um conjunto de vinculações com interceptadores;
- Uma implementação do bean;

2.6.2 Injetando *Beans*

Heffelfinger (2010, p. 308) explica que o capacidade de injeção de dependências tem sido adotada pela tecnologia Java por certo tempo, e desde o Java EE 5 é possível injetar recursos e alguns tipos de objetos em objetos gerenciados pelo container. O CDI estende esta capacidade de injeção para uma gama maior de objetos, tornando possível a injeção de objetos que até então não eram injetáveis pelo container.

Ao utilizar CDI todos os *beans* da aplicação tornam-se disponíveis para injeção, estes *beans* podem ser injetados em qualquer classe que esteja dento do contexto do container, ou seja, que possua o ciclo de vida gerenciado pelo container Java EE.

Para que um *bean* seja injetado basta anotar a variável que receberá a instância do *bean* com a *annotation* `@Inject`. O Quadro 2 apresenta na linha 7 como é indicado para o container onde efetuar a injeção de um *bean*.

```
1   package controller;
2
3   import bean.Greeting;
4   import javax.inject.Inject;
5
6   public class GreetingController {
7       @Inject
8       Greeting greeting;
9
10      public String hello(String name){
11          return greeting.hello(name);
12      }
13  }
```

Quadro 2 - Injetando um *Bean*.
Fonte: Autoria Própria.

Heffelfinger (2010, p. 375) explica que ao utilizar a *annotation* `@Inject` em um objeto gerenciado pelo container, uma instância de classe requerida será obtida e atribuída ao campo automaticamente. Ou seja, ao encontrar a *annotation* `@Inject` o container executa uma verificação e obtêm uma instância do *bean* correto para injeção.

2.6.3 *Named Beans*

Segundo Heffelfinger (2010, p. 371), CDI fornece a capacidade de nomear os *beans* utilizando a *annotation* `@Named` que, quando aplicada em uma classe, informa ao container

que as instâncias desta classe serão *Named Beans*. A utilização de *Named Beans* simplificam a injeção de *beans* em outras classes que dependem destes. Outra vantagem da utilização desta *annotation*, é que o *bean* anotado pode ser referenciado facilmente em páginas JSF por meio de EL. Desta maneira a anotação `@Named` possibilita, por exemplo, a invocação de métodos de um EJB em uma página JSF. É válido salientar que a *annotation* `@Named` não é o que torna uma classe um *bean*, ela apenas torna possível referenciar um *bean* a partir de EL. O Quadro 3 apresenta o exemplo de um *Named Bean.*

```
4
5   @Named
6   public class Customer {
7       /* ... */
8   }
```

Quadro 3 - Exemplo de *Named Bean.*
Fonte: Adaptado de HEFFELFINGER (2010, p. 372).

A classe `Customer`, apresentada no Quadro 3, é um *Named Bean* que pode ser referenciado em códigos não Java, como em uma página JSP ou em um componente JSF. Salvo a capacidade de ser referenciado a partir de EL, um *Named Bean* é exatamente igual a um *bean* convencional do CDI.

2.6.4 *Qualifiers*

Segundo Heffelfinger (2010, p. 375), muitas vezes o *bean* injetado é uma interface ou uma superclasse, contudo em algumas situações, é necessário injetar uma subclasse ou uma implementação específica de uma interface. Isto não é possível utilizando somente a *annotation* `@Inject`. Para atender a esta necessidade o CDI fornece os *qualifiers*.

Um *qualifier* é uma *annotation* criada pelo desenvolvedor, que por sua vez, é anotada com a *annotation* `@Qualifier`. A *annotation* criada pode ser utilizada para anotar uma subclasse específica ou ainda uma implementação de uma interface que será qualificada. Além disso o campo onde o objeto será injetado também precisa ser anotado com o *qualifier.*

Supõe-se que a aplicação possui um tipo especial de cliente. Alguns clientes podem ser considerados clientes *premium* e com isso, a classe `Customer` (define um cliente normal), é estendida pela classe `PremiumCustomer` (define um cliente *premium*). Para diferenciar as classes é criado um *qualifier*, e com isso a subclasse pode ser qualificada. O Quadro 4 apresenta o *qualifier Premium* responsável por qualificar a subclasse criada.

```
 8
 9  @Qualifier
10  @Retention(RetentionPolicy.RUNTIME)
11  @Target({ElementType.FIELD, ElementType.METHOD,
12      ElementType.PARAMETER, ElementType.TYPE})
13  public @interface Premium {
14  }
```

Quadro 4 - ***Qualifier Premium.***
Fonte: Adaptado de HEFFELFINGER (2010, p. 376).

A classe `PremiumCustomer` é anotada com o *qualifier* `@Premium`, assim, quando for necessário injetar um objeto desta classe, deve-se qualificar o campo utilizando este *qualifier*. O Quadro 5 apresenta a injeção de um `Customer` qualificado como *Premium*. Desta maneira, a instância do objeto injetado no campo `customer` será da classe `PremiumCustomer`. Se o campo `customer` não estivesse anotado com o *qualifier*, seria injetado nele uma instância da classe `Customer`.

```
14      @Inject
15      @Premium
        private Customer customer;
```

Quadro 5 - Injetando um ***Customer Premium.***
Fonte: Autoria Própria.

2.6.5 Escopos dos *Beans*

Segundo Heffelfinger (2010, p. 379) assim como os *Managed Beans* do JSF, os *beans* do CDI possuem escopo. Isto significa que os *beans* do CDI são objetos contextuais. Quando um *bean* é necessário, por ser injetado ou por ser referenciado em uma página JSF, o CDI procura por uma instância do *bean* no escopo que ele pertence e injeta-o no código dependente. Se nenhuma instância for encontrada, uma é criada e armazenada no escopo apropriado para ser usada futuramente. O Quadro 6 apresenta todos os escopos disponibilizados pelo CDI.

Escopo	***Annotation***	**Descrição**
Request	`@RequestScoped`	As instâncias dos *beans* são compartilhadas durante toda a duração de uma única requisição. Esta requisição pode se referir a uma requisição HTTP, a invocação de um método EJB, uma chamada de um *Web Service* e até mesmo o envio de uma mensagem JMS a um MDB.

Conversation	`@ConversationScoped`	O escopo de conversação pode se estender por várias requisições, mas é tipicamente menor do que o escopo de sessão. Neste escopo as instâncias são compartilhadas através das várias requisições.
Session	`@SessionScoped`	As instâncias dos *beans* de sessão atendem a todas as requisições em uma sessão HTTP. Cada usuário de um aplicativo obtém sua própria instância de um *bean* de sessão.
Application	`@ApplicationScoped`	As instâncias destes *beans* existem durante todo ciclo de vida da aplicação. Um *bean* neste escopo é compartilhado entre todas as sessões de usuários.
Dependent	`@Dependent`	Os *beans* de escopo dependente não são compartilhados. Com isso toda vez que um destes *beans* for injetado, é criada uma nova instância.

Quadro 6 - Escopos dos *Named Beans*.
Fonte: Adaptado de HEFFELFINGER (2010, p. 379).

Nos escopos do CDI são adicionados dois novos escopos aos disponibilizados pelo JSF. Entretanto, no CDI não estão disponíveis os escopos: `ViewScoped`; `NoneScoped`; e `CustomScoped`. Segundo Heffelfinger (2010, p. 380), os escopos do CDI possuem algumas particularidades:

- O escopo de requisição, não se limita à requisições HTTP;
- O escopo de conversação não existe no JSF. As classes que acessam um *bean* do escopo de conversação necessitam de uma instância da classe injetada. No momento que for necessário iniciar a conversação, o método `begin()` é invocado e para finaliza-la é necessário invocar o método `end()`;
- O escopo de sessão do CDI é exatamente igual ao seu homólogo do JSF: seu ciclo de vida está ligado diretamente a uma sessão HTTP;
- O escopo de aplicação é também exatamente igual ao escopo equivalente no JSF: uma única instância de um determinado *bean* existe durante toda execução da aplicação;
- O escopo de dependência não existe no JSF. Cada vez que um *bean* deste escopo for injetado uma nova instância é criada.

3 MATERIAL E MÉTODOS

Como as JSR definem especificações, é possível existir várias implementações para uma única especificação. Neste capítulo são apresentadas as implementações utilizadas, bem como suas configurações quando forem necessárias.

3.1 WELD

O Weld é a implementação de referência do CDI e atualmente encontra-se na versão 2.0.3, a qual foi liberada dia 22 de Julho de 2013. Segundo King *et al.* (2012), para a execução do Weld é necessário um ambiente de execução que o suporte, como:

- JBoss AS 6.0.0
- Glassfish 3.0
- Apache Tomcat 6.0.x
- Jetty 6.1.x

A vantagem da utilização de containers Java EE completos, como JBoss AS e Glassfish é que, em suas versões mais atuais, possuem o suporte para CDI embutidos, ou seja, o Weld está integrado ao container. Isto faz com que não seja necessária nenhuma configuração adicional para que o CDI possa ser utilizado. Entretanto, para que a aplicação utilize os recursos oferecidos pelo CDI, é necessário habilitá-los, bastando a aplicação possua o arquivo `beans.xml` na pasta `WEB-INF`. O Quadro 7 apresenta o conteúdo deste arquivo.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://java.sun.com/xml/ns/javaee"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5                http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
6  </beans>
```

Quadro 7 - Arquivo beans.xml.
Fonte: Autoria própria.

O arquivo `beans.xml` como apresentado no Quadro 7, não requer nenhuma configuração adicional para que as injeções de dependências do CDI funcionem. Entretanto se for necessário utilizar alguns recursos complexos, como interceptadores, será necessário

declará-los dentro da *tag* `<beans>`. Um exemplo de declaração de um interceptador no arquivo `beans.xml` é apresentada na sessão 4.2.1 deste trabalho.

3.2 GLASSFISH

É um servidor de aplicação que implementa as especificações do Java EE. “Apesar de ser um servidor de aplicação bastante novo, o Glassfish já é usado por um grande número de desenvolvedores e corporações.” (GONCALVES, 2010, p. 38). Segundo a JCP (2011) o Glassfish é a implementação de referencia da JSR 316, especificação do Java EE 6.

Por ser a implementação de referência do Java EE 6, o Glassfish contém todas as implementações das demais especificações da plataforma Java EE como: *Servlets*, *Web Services*, JSP, JSF, EJB, entre outros.

Segundo Goncalves (2010, p. 38-39), após a Sun e o grupo JServ doarem o Tomcat à Apache, a Sun por várias vezes reutilizou o Tomcat em seus produtos e em 2005 criou o projeto Glassfish. Seu principal objetivo era produzir um servidor de aplicações Java EE completamente certificado. Sua primeira versão foi liberada em maio de 2006.

“O Glassfish v2 foi liberado em setembro de 2007 e, desde então tem tido várias atualizações. Está é a versão mais aplamente distribuida, até hoje.” (GONCALVES, 2010). Em março de 2010, logo após a aquisição da Sun Microsystems pela Oracle, novas versões foram planejadas para os anos seguintes.

Apesar do Glassfish estar na versão 4.0, foi utilizado o Glassfish 3.1.2 para a implementação dos exemplos deste trabalho.

3.3 NETBEANS

O Netbeans é um ambiente de desenvolvimento integrado (*Integrated Development Environment* - IDE), *open-source* e oferece suporte a várias linguagens de programação, como Java, C, C++ e PHP. Foi adquirido pela Sun Microsystems e em junho de 2000 teve seu código disponibilizado para a comunidade. A empresa se manteve patrocinadora do projeto até janeiro de 2010, quando se tornou uma subsidiária da Oracle.

O Netbeans oferece ferramentas para a criação de todos os componentes Java EE, como JSF, EJB, *Servlets*, dentre outros. Atualmente encontra-se na versão 7.3.1, a qual foi utilizada neste trabalho.

3.4 ECLIPSELINK

O EclipseLink é uma implementação da JPA 2.0. Segundo GONCALVES (2010, p. 51), ele oferece uma poderosa e flexivel estrutura para armazenamento de objetos em uma base de dados relacional.

Goncalves (2010, p. 51) explica que o EclipseLink surgiu do produto TopLink da Oracle, doado à Fundação Eclipse em 2006. O EclipseLink é a implementação de referência da JPA 2.0 e sua versão atual é 2.5.0 (liberada em Maio de 2013), a qual foi utilizada neste trabalho.

3.5 UML

Unified Modeling Language (UML) é uma linguagem de modelagem, apoiada por metamodelo, que ajuda na descrição e no projeto de sistemas de *software* principalmente construídos utilizando o paradigma orientado a objetos (FOWLER, 2005, p. 25). Ela nasceu da unificação das muitas linguagens gráficas de modelagem orientadas a objeto do final dos anos 80 e inicio dos 90.

A UML 2 fornece para o processo de modelagem de *software* 13 diagramas, entretanto, por não ser do escopo do presente trabalho, serão utilizados somente os diagramas necessários para a exemplificação dos exemplos apresentados, como por exemplo, diagramas de classe, componentes e sequência. Nos exemplos apresentados neste trabalho não são documentadas todas as etapas de modelagem, sendo que os diagramas são apresentados com a finalidade de servirem como um modelo para o desenvolvimento do exemplo.

3.6 ASTAH COMMUNITY

Para a modelagem foi utilizado o *Astah Community 6.7.0*. Essa é uma ferramenta gratuita, que fornece os recursos necessários para a elaboração dos principais diagramas propostos pela UML 2.0. A ferramenta possui também versões pagas que possuem funcionalidades adicionais, como por exemplo, fluxogramas, mapas mentais, diagramas de

fluxo de dados, dentre outros. Entretanto, estas funcionalidades não são relevantes para o escopo deste trabalho.

A versão *community,* além de componentes prontos, fornece suporte à criação de classes e interfaces, que podem ser reutilizadas em vários diagramas. Com isso, a diagramação é agilizada e a refatoração facilitada.

4 RESULTADOS E DISCUSSÕES

Neste capítulo serão apresentadas algumas das aplicabilidades do CDI e como esta tecnologia pode agir como uma camada de integração entre alguns dos principais recursos da plataforma Java EE.

Nas próximas sessões serão apresentados exemplos de utilização do CDI provendo uma camada de integração entre outras tecnologias da plataforma Java EE. É importante salientar que os exemplos apresentados são somente para abordar o funcionamento do CDI e como aplicá-lo.

4.1 INTEGRAÇÃO JSF E EJB COM CDI

Quando se trabalha com a arquitetura Java EE, é muito comum utilizar os recursos disponibilizados pelo container para criar camadas distintas do sistema, por exemplo, é possível utilizar EJBs para encapsular as regras de negócio e JSF para prover uma camada de apresentação. Entretanto é necessário que haja a comunicação entre estas camadas. Esta comunicação pode ser feita utilizando ou não o CDI.

Para exemplificar a diferença entre a integrações de EJB e JSF com e sem a utilização de CDI, são propostos dois exemplos de carrinhos de compras. Estes exemplos são apresentados nas próximas subseções.

4.1.1 Implementação sem CDI

A Figura 2 apresenta o diagrama de componentes que exemplifica como os componentes JSF se relacionam com os EJBs sem a utilização de CDI.

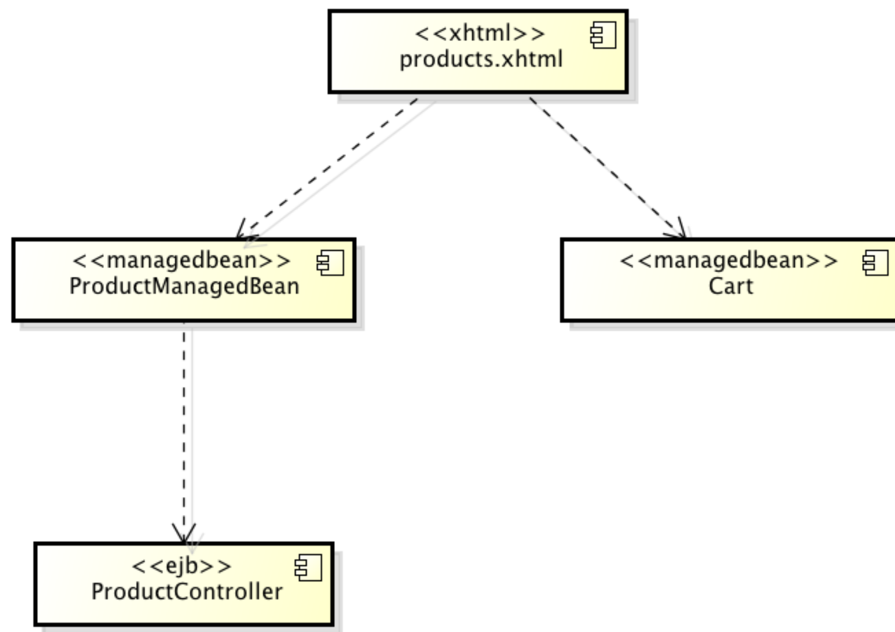

Figura 2 - Diagrama de Componentes Carrinho de Compras sem CDI.
Fonte: Autoria própria.

A página `products.xhtml` exibe os produtos disponíveis, obtidos normalmente do componente de negócio, neste caso, o `ProductController`. Como este componente de negocio é um EJB, não é possível referenciá-la diretamente na página de listagem de produtos. Para relacionar a página JSF ao EJB é utilizado um *Managed Bean* como mediador, pois o mesmo pode ser referenciado em páginas JSF, e pode, ainda, consumir recursos do container Java EE, como por exemplo, um EJB.

Há também o componente `Cart` que pode ser definido como um modelo não persistente, que armazena os produtos selecionados pelo usuário. Entretanto, ele é também um *Managed Bean*, pois deve existir no escopo de sessão e será necessário referenciá-lo na página JSF.

O Quadro 8 apresenta uma implementação limitada de um EJB *Stateless*, que serve apenas como exemplo, pois os produtos cadastrados estão definidos estaticamente no código. Esta situação não ocorre em aplicações reais. Por exemplo, tendo como base o método `getProducts()`, responsável por obter os produtos cadastrados, este método acessa a camada de persistência, o que executa uma consulta no banco de dados para obter os produtos cadastrados. Entretanto, como o exemplo visa apresentar a utilização do CDI, a lista de produtos está sendo definida estaticamente no código-fonte.

```
package controller;

import bean.Product;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Stateless;

@Stateless
public class ProductController {

    public List<Product> getProducts() {
        /*
         * Normalmente aqui é acessada a camada DAO,
         * que realiza uma consulta no banco de dados e retorna os produtos
         * cadastrados.
         */
        List<Product> products = new ArrayList<Product>();
        products.add(new Product("Livro Java EE", 110.90));
        products.add(new Product("Livro Java SE", 100.50));
        products.add(new Product("Livro Desing Patterns", 150.30));
        products.add(new Product("Livro Ruby On Rails", 99.90));
        products.add(new Product("Livro Ruby", 70.00));
        return products;
    }
}
```

Quadro 8 - Implementação do EJB ProductController.
Fonte: Autoria própria.

Um *Managed Bean* pode utilizar a especificação EJB e obter por meio de injeção de dependências uma instância de um EJB. Para isso é necessário utilizar a anotação `@EJB`. Desta forma, o *Managed Bean* pode fazer chamadas para o EJB e retornar os valores obtidos para a página JSF, e assim, conectando indiretamente um EJB a uma página JSF. O Quadro 9 apresenta a `ProductManagedBean`, classe que integra o EJB `ProductController` à página JSF.

```
package managedbean;

import bean.Product;
import controller.ProductController;
import java.util.List;
import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class ProductManagedBean {

    @EJB
    ProductController controller;

    public List<Product> getProducts() {
        return controller.getProducts();
    }
}
```

Quadro 9 - Implementação do *ManagedBean* ProductManagedBean.
Fonte: Autoria própria.

Na página JSF o *Managed Bean* é referenciado e no caso do exemplo apresentado, lista os produtos que são obtidos pelo EJB. O Quadro 10 apresenta o componente JSF responsável pela listagem de produtos.

```
14            <h:dataTable value="${productManagedBean.products}" var="product">
15                <h:column>
16                    <f:facet name="header">
17                        <h:outputLabel value="Descrição"/>
18                    </f:facet>
19                    <h:outputLabel value="${product.description}"/>
20                </h:column>
21                <h:column>
22                    <f:facet name="header">
23                        <h:outputLabel value="Volor"/>
24                    </f:facet>
25                    <h:outputLabel value="${product.value}"/>
26                </h:column>
27                <h:column>
28                    <h:commandButton value="Adicionar ao Carrinho"
29                            action="${cartManagedBean.addProduct(product)}"/>
30                </h:column>
31            </h:dataTable>
```

Quadro 10 - Componente JSF para listagem de produtos.
Fonte: Autoria Própria.

Em grande parte dos casos esta estrutura pode ser aplicada para a utilização de recursos disponibilizados por EJB em componentes JSF. O container Java EE tem a capacidade de injetar um EJB em um *Managed Bean*, o que de certa forma, estaria disponibilizando o recurso aos componentes JSF.

4.1.2 Implementação com CDI

Seguindo a premissa de facilitar a utilização de EJB com JSF, o exemplo apresentado na sessão 4.1.1 foi reelaborado para prover a integração entre as tecnologias através dos serviços disponibilizados pelo CDI.

A Figura 3 apresenta o diagrama de componentes que exemplifica como os componentes JSF se relacionam aos EJBs utilizando CDI.

Inicialmente, analisando a nova estrutura de componentes, pode-se notar que o *Managed Bean* que fazia a mediação entre os componentes JSF da tela e os EJBs pode ser removido.

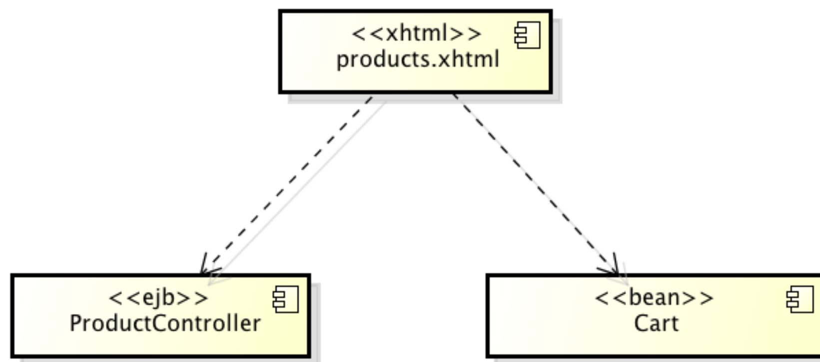

Figura 3 - Diagrama de Componentes Carrinho de Compras com CDI.
Fonte: Autoria própria.

Com base na estrutura de componentes apresentada, para que seja feita a integração, é necessário efetuar algumas alterações nas classes Java e também na página. Primeiramente, o EJB deverá ser anotado com `@Named`. Esta anotação possibilitará que o mesmo seja referenciado em componentes JSF por meio de EL. O Quadro 11 apresenta o EJB utilizando esta anotação.

```
package controller;

import ...

@Named
@Stateless
public class ProductController {
    public List<Product> getProducts(){
        /*
         * Normalmente aqui é acessada a camada DAO,
         * que realiza uma consulta no banco de dados e retorna os produtos
         * cadastrados.
         */
        List<Product> products = new ArrayList<Product>();
        products.add(new Product("Livro Java EE", 110.90));
        products.add(new Product("Livro Java SE", 100.50));
        products.add(new Product("Livro Desing Patterns", 150.30));
        products.add(new Product("Livro Ruby On Rails", 99.90));
        products.add(new Product("Livro Ruby", 70.00));
        return products;
    }
}
```

Quadro 11 - EJB ProductController anotado com `@Named`.
Fonte: Autoria própria.

Após adicionar a anotação, já é possível referenciá-lo em páginas JSF. O Quadro 12 apresenta o componente JSF que lista os produtos cadastrados, utilizando o EJB diretamente no JSF.

```
14            <h:dataTable value="${productController.products}" var="product">
15                <h:column>
16                    <f:facet name="header">
17                        <h:outputLabel value="Descrição"/>
18                    </f:facet>
19                    <h:outputLabel value="${product.description}"/>
20                </h:column>
21                <h:column>
22                    <f:facet name="header">
23                        <h:outputLabel value="Valor"/>
24                    </f:facet>
25                    <h:outputLabel value="${product.value}"/>
26                </h:column>
27                <h:column>
28                    <h:commandButton value="Adicionar ao Carrinho"
29                                     action="${cart.addProduct(product)}"/>
30                </h:column>
31            </h:dataTable>
```

Quadro 12 - Componente JSF para listagem de produtos referenciando um EJB.
Fonte: Autoria Própria.

Levando em conta os exemplos apresentados, pode-se perceber que a utilização de CDI simplifica a comunicação entre JSF e EJB. Ainda que a utilização de EJBs com Java EE 6 simplifique o *lookup* dos mesmos, quando se utiliza CDI não é necessário implementar a estrutura responsável por mediar esta comunicação, neste caso, o *ManagedBean*. Como o container de aplicação irá mediar esta comunicação, é possível dizer que a integração entre as tecnologias será menos suscetível a falhas, pois esta é feita pelo container que é estável e desenvolvido conforme as especificações.

É válido salientar que nem sempre a utilização de um EJB diretamente em uma página JSF será a melhor opção. Em casos específicos, poderá ser necessário utilizar classes mediadoras, nestes casos não para integrar as tecnologias, mas onde realmente será necessário implementar alguma lógica entre a camada de interface (páginas JSF) e a camada de negócio (EJBs).

4.2 CRIANDO UMA CAMADA DE PERSISTÊNCIA COM JPA E CDI

Em aplicações que necessitem gravar dados, uma das principais e primeiras camadas a serem desenvolvidas é a camada de persistência. Normalmente, em aplicações Java, para a criação desta camada é utilizada a especificação JPA, que possibilita o mapeamento objeto relacional abstraindo a conversão de objetos em tabelas do banco de dados.

Para demonstrar a utilização de JPA com CDI, é proposta a implementação de um modelo de camada de persistência utilizando alguns recursos disponibilizados pelo CDI, que poderá colaborar na criação de classes bem definidas, coesas e com acoplamento minimizado.

A Figura 4 apresenta um diagrama de sequência que demonstra o processo de inserção de um novo produto no banco de dados utilizando a camada de persistência proposta.

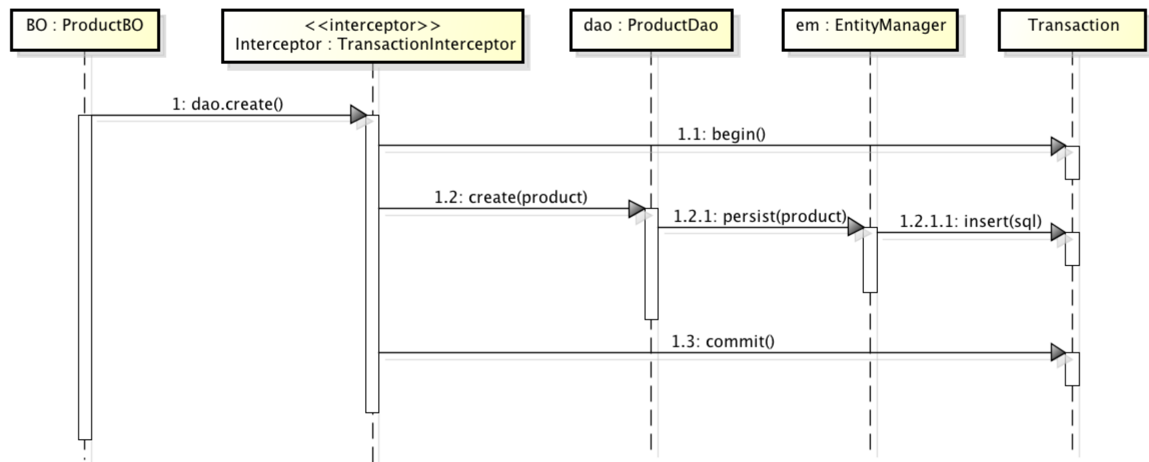

Figura 4 - Diagrama de sequência para persistência de um novo produto.
Fonte: Autoria Própria.

A classe `ProductBO` é responsável por encapsular todas as regras de negócio referente à `Product`. Para persistir um novo objeto ela faz uma chamada à classe `ProductDao`, e esta abstrai a camada de persistência. Em suma ela será a classe que utilizará a implementação do JPA e gravará as informações no banco de dados.

Para gravar os dados no banco de dados, a classe `ProductDao` obtém uma instância de um `EntityManager` e invoca o método `persist()`. Com isso, o JPA converterá a entidade a ser salva em um comando de inserção e executará o mesmo no banco de dados. Entretanto, para que ocorra efetivamente a gravação, é necessário que seja iniciada uma transação, executada a instrução de inserção e finalizada a transação. Para que estas ações sejam abstratas para a classe que invoca o método `create()` uma solução, é iniciar e finalizar uma transação neste método. Contudo, desta forma o método não seria coeso, pois estaria executando outras tarefas além de criar um novo registro.

Uma solução mais sofisticada é a utilização e um dos recursos do CDI que possibilita a criação de interceptadores. A classe `TransactionInterceptor` utiliza este recurso e é responsável por abrir e fechar uma transação para todos os métodos que executam operações transacionais no banco de dados, como por exemplo o método `create()`.

Um ponto importante é que a plataforma Java EE possui recursos para gerenciamento de transações. Entretanto, será criado outro meio para automatizar este

gerenciamento, que poderá ser aplicado em containers que não implementem totalmente a especificação do Java EE.

Quando se utiliza JPA para persistir um objeto, é utilizado um `EntityManager`, entretanto este não está disponível para injeção no contexto do CDI. Para torná-lo injetável, é necessário criar um método produtor que crie e retorne uma instancia da classe desejada. O Quadro 13 apresenta uma classe com um método produtor que cria as instâncias de `EntityManager` disponibilizando-o para injeção com CDI.

```
9   public class EntityManagerProductor {
10
11      @Produces
12      @RequestScoped
13      public EntityManager getEntityManager() {
14          EntityManagerFactory factory;
15          factory = Persistence.createEntityManagerFactory("JpaPU");
16
17          EntityManager em = factory.createEntityManager();
18
19          return em;
20      }
21  }
```

Quadro 13 - Método produtor de EntityManager.
Fonte: Autoria Própria.

Basicamente, um método produtor deve retornar uma instância da classe desejada, deverá ser anotado com `@Produces` e com a anotação referente ao contexto que o *bean* será aplicado. Neste caso, por utilizar a *annotation* `@RequestScoped`, será criado um novo `EntityManager` para cada requisição feita ao servidor. É importante salientar que a utilização de um método produtor não se limita apenas à injeção de classes de bibliotecas externas, mas também para classes próprias que possuem alguma particularidade no momento da instanciação.

Para que os métodos transacionais possam ser interceptados e para que a manipulação da transação seja abstraída, é necessário criar um *interceptor*. No CDI a criação de um interceptador pode ser dividida nas seguintes etapas:

- Criação de uma *annotation* que identificará os métodos interceptados;
- Criação da classe interceptadora;
- Declaração do Interceptador no arquivo de configuração `beans.xml`.

Primeiramente é necessário criar uma *annotation* `InterceptorBinding`, neste caso a *annotation* `@Transactional`, que será responsável por relacionar um interceptador

aos métodos interceptados. O Quadro 14 apresenta o `InterceptorBinding` responsável por informar quais métodos deverão ser interceptados por um determinado interceptador.

```
1  package dao.aop;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7  import javax.interceptor.InterceptorBinding;
8
9  @InterceptorBinding
10 @Target({ElementType.METHOD, ElementType.TYPE})
11 @Retention(RetentionPolicy.RUNTIME)
12 public @interface Transactional {}
```

Quadro 14 - InterceptorBinding Transactional.
Fonte: Autoria Própria.

Após a criação do `InterceptorBinding` é necessário criar a classe interceptadora. Esta classe será a responsável por abrir e fechar as transações do `EntityManager`.

O Quadro 15 apresenta a classe que irá interceptar os métodos transacionais. Para informar ao CDI que esta classe será um interceptador, é necessário anotá-la com `@Interceptor` e com `@Transactional`. Com isso será especificado que a classe `TransactionInterceptor` será um interceptador para métodos anotados com `@Transactional`.

```
1  package dao.aop;
2
3  import javax.inject.Inject;
4  import javax.interceptor.AroundInvoke;
5  import javax.interceptor.Interceptor;
6  import javax.interceptor.InvocationContext;
7  import javax.persistence.EntityManager;
8
9  @Transactional
10 @Interceptor
11 public class TransactionInterceptor {
12
13     @Inject
14     private EntityManager em;
15
16     @AroundInvoke
17     public Object manageTransaction(InvocationContext context)
18             throws Exception {
19         em.getTransaction().begin();
20         Object result = context.proceed();
21         em.getTransaction().commit();
22         return result;
23     }
24 }
```

Quadro 15 - Interceptador TransactionInterceptor.
Fonte: Autoria Própria.

Além das *annotations* utilizadas na classe, para definir um interceptador, é necessário criar um método e anotá-lo com `@AroundInvoke`. Este método deverá receber como parâmetro um `InvocationContext` e retorna um `Object`. Desta forma, quando um método transacional for invocado ele será interceptado e o método anotado com `@AroundInvoke` interceptará a execução do método transacional.

Para que os métodos transacionais possuam uma transação aberta ao serem invocados, é necessário obter uma instância de um `EntityManager`, o que pode ser feito via injeção de dependências, bastando utilizar a *annotation* `@Inject` na declaração da variável que armazenará o objeto requerido. Com este `EntityManager` é possível obter e manipular uma transação com o banco de dados, basicamente basta obtê-la a partir do método `getTransaction()` e invocar o método `begin()`, para que uma transação seja iniciada. Tendo a transação aberta, o método que foi interceptado pode ser executado. Para isso é necessário invocar o método `proceed()` do `InvocationContext`. Quando a execução do método finalizar, no interceptador é necessário finalizar a transação invocando o método `commit()`, que pode ser feito da mesma forma que o método `begin()`.

Para finalizar a criação do interceptador, é necessário registrá-lo no CDI. Para isso basta declará-lo no arquivo `beans.xml` conforme apresentado no Quadro 16.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://java.sun.com/xml/ns/javaee"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://jav
5      <interceptors>
6          <class>dao.aop.TransactionInterceptor</class>
7      </interceptors>
8  </beans>
```

Quadro 16 - Declaração do Interceptor no bean.xml.
Fonte: Autoria Própria.

Com o interceptador finalizado e podendo injetar um `EntityManager`, é possível criar uma camada de persistência utilizando o padrão *Data Access Object*[8] (DAO) para criar esta camada. Primeiramente é necessário definir uma interface que contêm os métodos principais para a criação de uma classe DAO (Quadro 17).

[8] Segundo Araújo, *et al.* (2013, p.95), DAO é um padrão de integração definido no catálogo Java EE e é responsável por encapsular o acesso e a manipulação de dados em uma camada de persistência. Desta forma, o principal e único objetivo do padrão DAO é a disponibilização de interfaces bem definidas para a manipulação de dados persistentes.

```java
1    package dao;
2
3    import java.util.List;
4
     public interface Dao<T> {
         public T create(T entity);
         public T read(Long id);
         public T update(T entity);
         public void delete(T entity);
         public List<T> findAll();
11   }
```

Quadro 17 - Implementação da Interface DAO.
Fonte: Autoria Própria.

Após criar a interface, basta criar uma implementação da interface Dao para cada entidade, e se necessário, adicionar os métodos específicos de cada um. O Quadro 18 apresenta uma implementação de Dao, que será responsável por gerenciar as operações referentes à entidade Product.

```java
1    package dao.impl;
2
3    import ...
11
12   public class ProductDao implements Dao<Product>{
13
14       @Inject
         private EntityManager em;
16
17       @Override
18       @Transactional
         public Product create(Product entity) {
20         em.persist(entity);
21         return entity;
22       }
23
24       @Override
         public Product read(Long id) {...}
28
29       @Override
30       @Transactional
         public Product update(Product entity) {...}
35
36       @Override
         public void delete(Product entity) {...}
40
41       @Override
         public List<Product> findAll() {...}
47
48   }
```

Quadro 18 - Implementação de DAO para a entidade Product.
Fonte: Autoria Própria.

Na classe `ProductDao` é utilizado um `EntityManager`, que é injetado pelo CDI da mesma forma como é feito no interceptador. O objeto injetado será o mesmo que o utilizado no interceptador, desde que seja na mesma requisição. Este comportamento é definido no método produtor de `EntityManager`, onde foi definido o escopo de requisição.

O método `create()` fará a inserção do novo registro no banco de dados, e para isso, em sua implementação, apenas é invocado o método `persist()` e retornado a entidade atualizada. Entretanto, o método está anotado com `@Transactional`, e por isso será interceptado. Assim, durante sua execução do método, a transação com o banco de dados estará aberta.

Com toda esta estrutura criada, a classe `ProductDao` poderá ser utilizada nas classes de negócio, como por exemplo, em um *Bussines Object*[9]. O Quadro 19 apresenta um exemplo básico de utilização da classe DAO criada, sendo que sua instância é fornecida pelo CDI através de injeção de dependência.

```
1   package ejb;
2
3   import ...
10
11  @Stateful
12  @Named
13  public class ProductBO implements Serializable{
14
15      @Inject
16      private ProductDao dao;
17
18      public void create(Product product) throws Exception{
19          if(validate(product)){
20              dao.create(product);
21          }else{
22              throw new Exception("Invalid Product");
23          }
24      }
25
26      public List<Product> getAll(){...}
29
30      public void remove(Product product){...}
33
34      public boolean validate(Product product){...}
40  }
```

Quadro 19 - Injeção da classe ProductDao em uma classe de negócio.
Fonte: Autoria Própria.

[9] Bussines Object é um padrão definido no catálogo J2EE, Segundo Deepak, *et al.* (2004, p. 334) este padrão define um modelo de objeto utilizado para separar os dados e a lógica de negócios da lógica de persistência. Estes objetos mantêm os dados de negócios principais e implementam o comportamento que é comum a toda aplicação ou domínio.

No exemplo apresentado, é possível identificar que a utilização de CDI pode simplificar a injeção de dependências de classes que possuem construção complexa. Por exemplo, através de um *Builder*[10] ou *Factory Method*[11], e até mesmo instâncias de classes externas, como é o caso do `EntityManager`.

A utilização de interceptadores pode melhorar consideravelmente a qualidade do código. No caso do gerenciamento das transações para métodos que executem operações transacionais, os métodos em si se tornam desacoplados da forma como a transação é manipulada, sendo que toda a responsabilidade é delegada ao interceptador. Outro ponto interessante é que a modularização do código é favorecida, pois é possível criar um interceptador único e coeso e reaproveitá-lo em vários pontos do sistema. Além disso, os métodos possuirão um propósito único e o código implementado no método será destinado unicamente a atender este propósito. Com isso a extensão do método diminuirá, o que facilitará a leitura e consequentemente a compreensão do código-fonte.

4.3 UTILIZANDO CDI COM SERVLETS

Quando se desenvolve um sistema com Java, normalmente não é indicado tentar resolver todas as situações utilizando a mesma API ou *framework*, pois cada especificação Java é destinada a um fim específico. Por exemplo, se for necessário criar um *Web Service* utilizando arquivos *Web Services Description Language* (WSDL), é indicado que seja usado JAX-WS[12]. Entretanto, se for necessário criar um RESTFul, é indicada a utilização JAX-RS. Resumidamente, para a grande maioria das situações já existe uma especificação destinada a atendê-las.

Todavia, em algumas situações é necessário implementar uma solução para um problema especifico, onde é necessário utilizar tecnologias de baixo nível, como *Servlets*. Por exemplo, suponha que seja necessário gerar um arquivo no servidor e efetuar o *download* do mesmo. Neste caso, por ser uma situação específica, pode ser que não exista uma tecnologia Java bem definida, padronizada, estável e destinada unicamente a esta funcionalidade. Com

[10] *Builder* é um padrão de projeto definido no catálogo GoF. Segundo Gamma, *et al.* (2000, p. 104), este padrão tem como objetivo separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações.

[11] *Factory Method* é um padrão de projeto definido no catálogo GoF. Segundo Gamma, *et al.* (2000, p. 112), o *Factory Method* define uma interface para criar um objeto, mas deixa as subclasses decidirem que classe instanciar.

[12] Segundo Goncalves (2010, p. 461), o JAX-WS define um conjunto de APIs e anotações que permitem a construção e consumo de *Web Services* com java através de SOAP, mascarando a complexidade do protocolo.

isso, uma das opções do desenvolvedor é a criação de um *Servlet* que será responsável por tal funcionalidade.

Quando se utiliza CDI, todos os seus recursos são disponibilizados para todas as tecnologias Java EE, consequentemente todos os *beans* estarão disponíveis para injeção em todas estas tecnologias. Sendo assim, os *beans* podem ser utilizados em tecnologias de baixo nível, como por exemplo, em um *Servlet*.

Para demonstrar a utilização de CDI em um *Servlet*, é proposta a criação de um recurso que disponibilizará para download um arquivo *Comma-Separated Values* (CSV), contendo todos os produtos cadastrados na aplicação e seus valores. Este artefato pode ser utilizado por lojas virtuais que queiram disponibilizar sua lista de produtos para *download*. A Figura 5 apresenta o diagrama de classes que detalha a estrutura de relação das classes envolvidas no processo de download da lista de produtos.

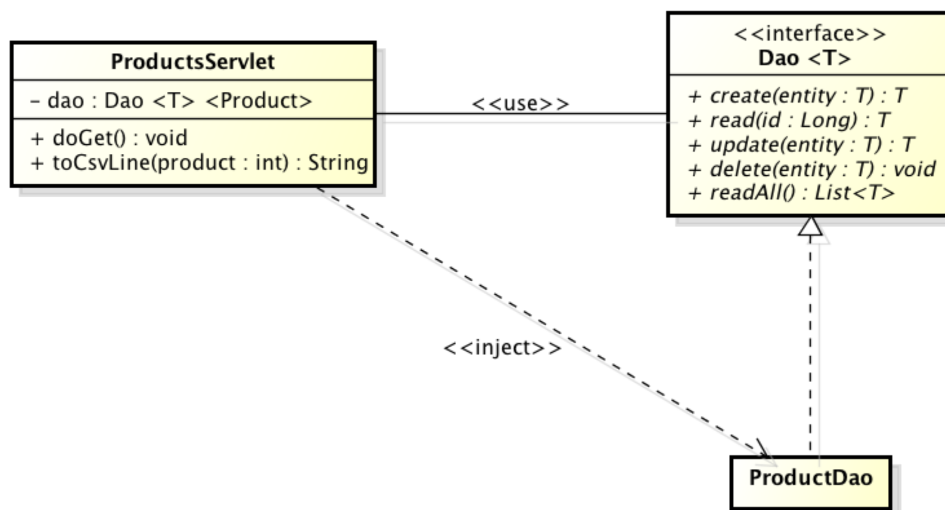

Figura 5 - Diagrama de classe download de lista de produtos.
Fonte: Autoria própria.

O *Servlet* `ProductServlet` é responsável por receber as requisições que solicitam o *download* da lista de produtos e retornar o arquivo CSV. Ao receber uma solicitação, ele executa uma busca na camada de persistência por meio de um objeto que implementa a interface `Dao`, neste caso, utilizando uma instância de `ProductDao` que é obtida utilizando a injeção de dependências. Após obter os dados dos produtos, os mesmos são convertidos para o formato CSV e retornados como resposta à chamada HTTP.

Ao utilizar interfaces, podem existir diversas implementações para a mesma. Levando em consideração que podem existir também várias entidades persistentes, é provável e esperado que exista no mínimo uma implementação da interface `Dao` para cada entidade.

Quando utilizada a *annotation* `@Inject` para informar ao CDI que é necessário injetar uma instância na declaração de uma variável do tipo de uma interface, se existir várias implementações para a mesma, o CDI não poderá resolver qual implementação utilizar. Para solucionar este problema, é possível utilizar *qualifiers* que poderão ser utilizados para informar qual implementação será injetada. Portanto foi criado o *qualifier* `ProductQualifier`, que será utilizado para designar implementações específicas para produtos quando o mesmo for utilizado juntamente com a *annotation* `@Inject`. O Quadro 20 apresenta o *annotation* que o define.

```
1   package qualifier;
2
3   import java.lang.annotation.ElementType;
4   import java.lang.annotation.Retention;
5   import java.lang.annotation.RetentionPolicy;
6   import java.lang.annotation.Target;
7   import javax.inject.Qualifier;
8
9   @Qualifier
10  @Retention(RetentionPolicy.RUNTIME)
11  @Target({ElementType.TYPE, ElementType.FIELD,
12      ElementType.METHOD, ElementType.PARAMETER})
13  public @interface ProductQualifier {
14
15  }
```

Quadro 20 - Implementação do *Qualifier* ProductQualifier.
Fonte: Autoria Própria.

Dispondo do *qualifier* que especificará quais serão as implementações específicas para produtos, é necessário anotar as classes implementadoras com o *qualifier* criado. Neste caso, a classe anotada é a `ProductDao` e é a mesma classe utilizada no caso de uso apresentado na sessão 4.2 deste trabalho, porém qualificada com a *annotation* `@ProductQualifier`. O Quadro 21 apresenta a implementação DAO para as entidades `Product`.

```
1   package dao.impl;
2
3   import ...
12
13  @ProductQualifier
14  public class ProductDao implements Dao<Product>, Serializable{
15
16      @Inject
17      private EntityManager em;
18
19      @Override
20      @Transactional
21      public Product create(Product entity) {...}
25
26      @Override
27      public Product read(Long id) {...}
30
31      @Override
32      @Transactional
33      public Product update(Product entity) {...}
37
38      @Override
39      public void delete(Product entity) {
40          em.remove(entity);
41      }
42
43      @Override
44      public List<Product> findAll() {
45          TypedQuery<Product> query = em.createQuery("SELECT p FROM Product p",
46                  Product.class);
47          return query.getResultList();
48      }
49  }
```

Quadro 21 - Product Dao Qualificado.
Fonte: Autoria Própria.

Definir uma implementação de uma interface com um qualificador, possibilita que a instância da classe utilizada pela injeção de dependência do CDI seja definida em tempo de codificação. Por exemplo, o Quadro 22 apresenta o *Servlet* `ProductsServlet`, responsável pela geração e download da lista de produtos. Nele é injetada uma instância de uma classe que implementa a interface `Dao`. O *qualifier* `ProductQualifier` define qual o tipo do objeto a ser injetado.

```
15  public class ProductsServlet extends HttpServlet{
16
17      @Inject
18      @ProductQualifier
19      Dao<Product> dao;
20
21      @Override
22      protected void doGet(HttpServletRequest req, HttpServletResponse resp)
23              throws ServletException, IOException {
24
25          resp.setHeader("Content-Disposition",
26                  "attachment;filename=products.csv");
27
28          PrintWriter writer = resp.getWriter();
29          List<Product> products = dao.findAll();
30          for (Product product : products ){
31              writer.print(toCsvLine(product));
32          }
33          writer.close();
34      }
35
36      private String toCsvLine(Product product){...}
46
47  }
```

Quadro 22 - Servlet de download da lista de produtos.
Fonte: Autoria Própria.

O Quadro 22 mostra na linha 19 a variável `dao`, anotada com `@Inject` e com `@ProductQualifier`. Com isso o CDI irá buscar pelas implementações de interface, verificar qual delas está anotada com `@ProductQualifier` e injetar uma instância após resolver a dependência. Vale salientar que o *Servlet* não fica acoplado à implementação, pois o que define qual objeto será instanciado é o *qualifier*. Desta forma é possível alterar qual objeto será injetado de duas maneiras: (a) alterando o qualificador no *Servlet*; e (b) a implementação anotada com o qualificador.

Para finalizar a criação do *Servlet* de download da lista de produtos, é necessário declará-lo no arquivo `web.xml`. Este arquivo é responsável pelas configurações do projeto Java WEB. O Quadro 23 apresenta esta declaração. Conforme definido na *tag* `url-pattern`, quando a aplicação receber uma requisição na URL "/download" o *Servlet* `ProductsServlet` tratará a mesma e retornará a lista de produtos.

```
14      <servlet>
15          <servlet-name>ProductsServlet</servlet-name>
16          <servlet-class>servlet.ProductsServlet</servlet-class>
17      </servlet>
18
19      <servlet-mapping>
20          <servlet-name>ProductsServlet</servlet-name>
21          <url-pattern>/download</url-pattern>
22      </servlet-mapping>
```

Quadro 23 - Declaração do Servlet no web.xml.
Fonte: Autoria Própria.

Os arquivos CSVs podem ser abertos em *software* de planilhas, o que facilita a visualização dos dados. A Figura 6 apresenta o conteúdo de um arquivo CSV gerado utilizando o *Servlet* apresentado no exemplo.

	A	B	C
1	1	Coca-Cola	4.5
2	2	Carne KG	19.9
3	3	P4	23.0
4	4	Batata	4.95
5			

Figura 6 - Arquivo CSV gerado.
Fonte: Autoria Própria.

A utilização de CDI pode aperfeiçoar o desacoplamento entre classes. Tendo como base o exemplo apresentado, é possível utilizar uma instância específica, entretanto, sem que haja a necessidade de acoplar duas implementações. Este é o caso do *Servlet* `ProductsServlet`, pois este depende diretamente de uma instância específica de uma implementação da interface Dao, neste caso, da implementação responsável pela persistência dos objetos da entidade `Products`.

Desacoplando as implementações, a modularidade da aplicação é aperfeiçoada, pois as mesmas serão acopladas somente ao que for necessário e caso seja feita alguma alteração em uma determinada classe da aplicação, as outras não serão afetadas. Utilizando CDI para prover esta modularidade, é possível alterar o objeto injetado como dependência, apenas trocando o qualificador, ou então, trocando a classe implementadora anotada pelo mesmo.

5 CONSIDERAÇÕES FINAIS

Neste capítulo serão apresentadas as conclusões sobre os benefícios da utilização do CDI para prover a integração entre as demais tecnologias da plataforma Java EE. Também neste capítulo são apresentadas as propostas de trabalhos futuros com CDI.

Este trabalho apresentou três exemplos práticos da aplicabilidade do CDI. No primeiro exemplo foi apresentada uma integração de JSF e EJB utilizando CDI, possibilitando assim a injeção de EJBs diretamente em páginas JSF através de EL. No segundo exemplo foi apresentada a utilização de CDI para a criação de uma camada de persistência com JPA e interceptadores. Já no terceiro exemplo, foi apresentada a integração de CDI com *Servlets* com a utilização de *qualifiers*.

5.1 CONCLUSÃO

Pode-se dizer que o CDI aproxima estas duas tecnologias e sua utilização unifica os escopos dos *beans*. Basicamente ela agrupa as definições dos *Managed Beans* com injeção de dependências, o que o torna flexível para utilização em diversas situações que simplificam e reduzem o código estrutural.

Além da unificação dos escopos e da injeção de dependência, o CDI provê alguns recursos extras que podem ser de grande valor para melhoria do código. Um destes recursos são os interceptadores, que podem encapsular lógicas e aplicá-las como comportamentos. Com isso é possível implementar um comportamento em um interceptador e reaproveitá-lo em diversos pontos do sistema, interceptando métodos que requeiram este comportamento. Este recurso deixa os métodos interceptados desacoplados da implementação dos comportamentos e com isso, a implementação de um método interceptado terá somente o código necessário para a resolução do propósito para qual foi criado, o que pode facilitar a leitura e o entendimento do código. Além disso, ao utilizar os interceptadores a modularidade do código é valorizada, pois um determinado comportamento é definido em um único ponto do sistema, e ao ser alterado, não requer que os métodos que são interceptados sejam adaptados a nova implementação, e com isso, facilita a manutenção do sistema.

De certa forma, um ponto negativo da utilização do CDI é a necessidade de um profundo conhecimento em como o mesmo funciona, pois a utilização de injeção de dependências combinada com o uso de interceptadores poderá causar confusão em

desenvolvedores que não estão acostumados à utilização destes recursos. Entretanto, esta combinação se mostra muito flexível, pois pode ser aplicada a diversas tecnologias da plataforma Java EE.

5.2 TRABALHOS FUTUROS

Propõe-se como trabalho futuro a criação de um estudo de caso de uma arquitetura completa e real baseada na utilização de CDI e integrando as tecnologias da plataforma Java EE em sistemas de grande porte, de modo a comprovar experimentalmente os resultados obtidos nos exemplos deste trabalho. Pretende-se ainda realizar um estudo sobre os recursos adicionais do CDI e suas aplicabilidades em desenvolvimento de *software* orientado a aspectos e desenvolvimento de *software* orientado a eventos.

REFERÊNCIAS BIBLIOGRÁFICAS

ARAÚJO, Everton Coimbra de. **Desenvolvimento para WEB com Java.** Florianópolis: Visual Books Editora, 2010. 244 p.

ARAÚJO, Everton Coimbra de; GUIZZO, Giovani; LAMB, Juliano Rodrigo; MERENCIA, Lucas José. **Padrões de Projeto em Aplicações WEB**. 1. ed. Florianópolis: Editora Visual Books Ltda., 2013. 142 p.

BOND, Martin; HAYWOOD, Dan; LAW, Debbie; LONGSHAW, Andy; ROXBURGH, Peter. **Aprenda J2EE em 21 dias**. São Paulo: Pearson Education do Brasil, 2003. 962 p.

DEEPAK, Alur; MALKS, Dan; CRUPI, John. **Core J2EE Patterns:** Best Practices and Design Strategies. 2nd Edition. ed. Upper Saddle River: Prentice Hall PTR, 2004. 528 p.

FOWLER, Martin. **UML Essencial:** Um breve guia para linguagem padrõ de modelagem de objetos. 3ª Edição. ed. Porto Alegre: Bookman, 2005. 160 p.

GAMMA, Erich; HELM, Richard; JOHSON, Ralph; VLISSIDES, Jhon. **Padrões de Projeto:** Soluções Reutilizáveis de software orientado a objetos. Porto Alegre: bookman, 2000. 350 p.

GONCALVES, Antonio. **Beginning Java™ EE 6 Platform with GlassFish™ 3**. Second Edition. ed. New York: Springer Science+Business Media, 2010. 536 p.

HEFFELFINGER, David. **Java EE 6 with GlassFish 3 Application Server**. Birmingham: Packt Publishing Ltd, 2010. 488 p.

JCP. JSR 316: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification. **Jaca Comunity Process**, 2009. Disponivel em: <http://jcp.org/en/jsr/detail?id=316>. Acesso em: 9 Julho 2012.

JCP. JSR 311: JAX-RS: The JavaTM API for RESTful Web Services. **Java Community Process**, 2011. Disponivel em: <http://jcp.org/en/jsr/detail?id=311>. Acesso em: 9 Julho 2012.

JCP. **Java Community Process**, 2012. Disponivel em: <http://www.jcp.org/en/home/index>. Acesso em: 27 ago. 2013.

JENDROCK, Eric; EVANS, Ian; GOLLAPUDI, Devika; HAASE, Kim; SIRVATHSA, Chinmayee. **The Java EE 6 Tutorial:** Basic Concepts. 4th. ed. Boston: Pearson Education, Inc., 2010. 600 p.

KING, Gavin; MUIR, Pete; ALLEN, Dan; ALLEN, David; BENAGLIA, Nicola. Weld - Implementação de Referência da JSR-299. **Seam Framework - Community**

Documentation, 2012. Disponivel em: <http://docs.jboss.org/weld/reference/1.1.5.Final/pt-BR/html_single>. Acesso em: 18 Junho 2012.

ORACLE. Oracle GlassFish Server 3.1-3.1.1 Reference Manual. **Oracle Documentation**, 2011. Disponivel em: <http://docs.oracle.com/cd/E18930_01/html/821-2433/asadmin-1m.html>. Acesso em: 9 Julho 2012.

SERRA, Ana Paula Gonçalves. O modelo de arquitetura CORBA e suas aplicações. **Integração** , São Paulo, v. I, n. 37, p. 157-163, 2004.