

Microserviços e EJB

Escale sua aplicação, não a complexidade



Sumário

- [ISBN](#)
- [Sobre o livro](#)
- [Agradecimentos](#)
- [Início rápido](#)
 - [1 Microserviços e Java EE](#)
 - [2 Serviços REST com JAX-RS](#)
- [Conhecendo de forma mais profunda](#)
 - [3 Sendo reapresentados aos EJBs](#)
 - [4 Os Sessions Beans por estado](#)
 - [5 Os Sessions Beans por interface](#)
 - [6 Integrando EJB e JPA](#)
 - [7 Gerenciando transações com EJBs](#)
 - [8 Lidando com mais de uma transação](#)
 - [9 Testando nossos serviços](#)
 - [10 Criando WebServices com JAX-RS](#)
 - [11 Server-Sent Events ou SSE \(JAX-RS 2.1, Java EE 8\)](#)
 - [12 Métodos assíncronos e paralelismo para aumento de performance](#)
 - [13 Utilizando a Concurrency Utilities for Java EE](#)
 - [14 Conclusão](#)

ISBN

Impresso e PDF: 978-65-86110-38-8

EPUB: 978-65-86110-37-1

MOBI: 978-65-86110-39-5

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

Sobre o livro

Público-alvo

Este livro foi escrito para você que busca saber como desenvolver o *back-end* da sua aplicação com Java, seja no modelo tradicional ou de microsserviço.

Provavelmente você já passou por situações em que sua aplicação tinha comportamentos que você não entendia e teve que buscar alguma solução na internet, que até funcionou, mas você também não sabe muito bem por quê. O pior é que, às vezes, nem mesmo quem passou a solução sabe explicar, pois também pegou de outra pessoa; e nesse cenário vem o "faz assim que funciona".

Aqui vamos fazer o oposto disso: no lugar de jogar a solução de qualquer jeito, vamos construí-la juntos e entender cada parte. Porém, como sei que em algumas situações precisamos de uma solução mais rápida, seja pela urgência do projeto, ou mesmo porque já sabemos o conceito, mas esquecemos a sintaxe, temos o livro dividido em duas partes: a primeira é mais direta ao ponto, e a segunda detalha mais a solução.

Como este livro está organizado

Parte 1: Início rápido

Pensando em situações em que temos pouco tempo para nos aprofundar em um aprendizado mais denso, foi criada uma parte inicial, com dois capítulos, nos quais é apresentado como criar rapidamente uma aplicação com microsserviços usando EJBs, JAX-RS e JPA. Essa primeira parte poderia ser considerada um "mini-book" sobre microsserviços com *back-end* em Java.

Principais temas:

- Criação de serviços REST com JAX-RS e JPA
- Utilização do Thorntail (Java EE MicroProfile)
- Gestão automática de transações do Java EE

Parte 2: Conhecendo de forma mais profunda

Na segunda parte, passamos para um aprendizado mais tradicional. A cada novo capítulo, veremos novos conteúdos e consolidaremos o conhecimento anterior. Aqui temos um ritmo mais cadenciado, onde o conteúdo é um pouco mais extenso e mais completo.

Principais temas:

- Session Beans: Stateless, Stateful, Singleton, Remotos, Locais, Sem interface
- Como usar JPA com gerenciamento automático de transações
- Como fazer um ajuste fino nas transações
- Como lidar com várias transações na mesma requisição
- Implementando testes de integração com Arquillian
- Aprendendo mais a fundo a construção de WebServices REST
- Server-Sent Events (SSE) e Reactive Client API (JAX-RS 2.1, Java EE 8)
- Trabalhando com serviços assíncronos e paralelismo
- Utilizando a Concurrency Utilities for Java EE

Em resumo, o livro visa ser uma referência das principais ferramentas com que lidamos no dia a dia, mas nem tudo precisa ser visto antes de iniciar nossos projetos. Inicie pela primeira parte, e vá pinçando os conteúdos que precisa aprender nos demais capítulos do livro. Desejo a você uma boa leitura.

Pré-requisitos

Para aproveitar bem este livro é importante que você já tenha o conhecimento da linguagem Java e também entenda como funciona uma aplicação web; basicamente requisição e resposta.

Versões das principais ferramentas utilizadas

- Este livro se baseia no **Java EE 8** mas serve também para o **Java EE 7** já que ambas usam **EJB 3.2**.
- O código geral está usando o **Java 8**.
- O servidor de aplicação utilizado foi o **Wildfly 19**.
- Na primeira parte do livro usamos o **Thorntail 2.6.0.Final**.
- A IDE sugerida é o **Netbeans 12**, mas **qualquer IDE Java** servirá, pois os projetos usam o Maven.
- Como dito anteriormente, os projetos usam **Maven 3.6.3**.
- E para os testes de integração usamos o **Arquillian 1.6.0**.

Agradecimentos

Agradeço primeiramente a Deus, pois sem Ele nada seria possível. Agradeço também especialmente à minha esposa Dantiele Cordeiro de Freitas Queiróz, que sempre me apoiou e procurou formas de me ajudar a continuar escrevendo mesmo em momentos em que não tinha de onde tirar tempo para fazer isso. E também ao meu filhinho Davi Cordeiro Queiróz, que mesmo sendo bebê me dá ânimo para buscar fazer o melhor.

Agradeço também aos meus pais, José Cordeiro de Souza e Cleunice dos Santos Cordeiro, e à minha irmã Giselly Santos Cordeiro, que sempre me deram força e são referenciais para a minha vida, sem sombra de dúvidas.

Também agradeço aos meus sogros, Vanderley e Iraci, que sempre entenderam as vezes em que deixamos de ir às pescarias por estarmos estudando, trabalhando, escrevendo...

E, por fim, agradeço também à equipe da Casa do Código, em especial à Vivian, pela paciência com esse projeto que demorou mais que o previsto inicialmente, mas que espero que agora surta frutos para quem o ler.

Início rápido

Esta primeira parte do livro foi pensada para oferecer um ponto de partida, caso você precise aprender rapidamente como fazer um *back-end* Java. Vamos aproveitar e focar mais nas tecnologias utilizadas no desenvolvimento de microsserviços, que na verdade são um subconjunto do que veremos no restante do livro.

As explicações aqui serão mais introdutórias com relação à segunda parte, então teremos referências dos capítulos específicos nos quais o assunto será desenvolvido com mais detalhes. Dessa forma, após este primeiro contato você poderá ir para lá, para aprender com mais profundidade.

CAPÍTULO 1

Microsserviços e Java EE

Neste primeiro capítulo veremos como utilizar as principais tecnologias do Java EE, tais como JAX-RS, JPA e EJB para a construção de microsserviços. Veremos que o processo é mais simples do que pode parecer em um primeiro momento, e caso tenhamos dúvidas se as tecnologias do Java EE são adequadas para a construção de microsserviços, aqui vamos responder aos principais questionamentos que podem vir à nossa mente.

1.1 Por que microsserviços?

Logo mais veremos **como** fazer, mas agora vamos entender só um pouco do **porquê** de pensar em microsserviços. Esse tipo de arquitetura foi uma evolução natural da nossa forma de desenvolver. Não precisamos discutir que o desenvolvimento monolítico é algo ruim. Ele faz com que nossas aplicações virem algo parecido com

aquele jogo de puxar varetas, em que temos que ter cuidado para ao mover uma peça, se não vamos estragar tudo.

Passamos a criar níveis de contenção enquanto desenvolvemos, usando interfaces em vez de classes para que a mudança de uma implementação não mexa nas demais varetas da aplicação, que devem conhecer somente a interface. Mas, além de separar esses níveis, passamos também a separar a aplicação em camadas, e a configuração mais famosa foi a conhecida como MVC (*Model, View, Controller*, ou Modelo, Visualização e Controle).

A ideia dessa configuração foi deixar a lógica de negócio limpa de qualquer tecnologia de visualização, e isso ganhou ainda mais espaço com o desenvolvimento de aplicativos móveis. Assim podemos ter uma mesma lógica sendo utilizada por um cliente móvel, outro desktop, um outro web, e assim por diante.

Na fase seguinte, começamos a usar arquiteturas orientadas a serviços (SOA, ou *Service-Oriented Architecture*). Nesse cenário, passamos a quebrar nosso modelo baseado em assunto. Então, se na nossa aplicação precisamos lidar com dados de RH e financeiros, não basta deixar tudo isso no modelo para ser reutilizável em diversos clientes. Separamos a parte de RH e de finanças em serviços específicos para que possam ser reutilizados em outros sistemas. Dessa maneira, nossas aplicações passaram a juntar serviços prontos que estavam à nossa disposição.

Os microsserviços são uma evolução desse modelo de desenvolvimento orientado a serviços, porém ainda mais focada no baixo acoplamento, para que seja possível a implantação (*deploy*) independente de cada serviço. Não pensamos mais em um *WebService* para cada serviço, estando todos eles implantados no mesmo servidor. Não que isso seja errado, mas a ideia dos microsserviços é aumentar a abstração entre os serviços, assim cada microsserviço é pensado para ser implantado independentemente de outros.

Nesse contexto, cresceu bastante a adoção de contêineres Docker, pois facilitam a implantação. Vale lembrar que o conceito de contêiner não é nada novo, o próprio Tomcat é um *servlet container* muito usado há muito tempo. Com ele, podemos desenvolver nossa aplicação, e distribuir (enviar para o cliente ou disponibilizar para download) um arquivo `war` que será colocado nesse contêiner. Porém, o Docker vai além: permite disponibilizarmos uma imagem que contém inclusive o Tomcat ou qualquer outro *runtime* necessário para a execução da nossa aplicação. A vantagem é que quem administra precisa conhecer somente o Docker, e dentro de cada imagem pode ter um Tomcat, ou MySQL, ou mesmo executar uma aplicação em Rails ou .Net.

Com base nessa evolução, vemos que apenas dividir em *front-end* e *back-end* não é suficiente. Utilizando como exemplo um sistema de *e-commerce*, em vez de entregarmos um grande *back-end* com todas as funcionalidades necessárias, podemos entregar diversos serviços independentes, mas que são compostos para chegarmos às mesmas funcionalidades. Por exemplo, um serviço de consulta de produtos e outro do carrinho de compras e fechamento de pedido.

A vantagem dessa abordagem é que cada serviço pode ser gerenciado de forma independente. O serviço de pesquisa provavelmente vai ser mais demandado, pois geralmente navegamos mais nos produtos do que efetivamente compramos, por isso podemos colocar mais servidores para balancear a carga desse serviço. Já a parte do carrinho de compras e fechamento de pedido pode não ser tão demandada, mas talvez precise de uma confiabilidade maior, pois perder uma requisição na pesquisa não é tão grave quanto perder um item do carrinho de compras do cliente.

Quando desenvolvemos tudo em um único bloco, podemos ter que gastar muito para escalar a aplicação como um todo, sendo que isso só seria necessário na parte da pesquisa de produtos; e também para aumentar a confiabilidade do sistema inteiro, quando isso só

seria necessário na parte do carrinho de compras e fechamento do pedido.

A tecnologia utilizada para o desenvolvimento de microsserviços não é o mais importante

Perceba que o que foi falado até aqui sobre o desenvolvimento de microsserviços tem muito mais a ver com pensarmos nossa solução de forma fragmentada, do que com qual framework de persistência eu devo usar para conseguir isso. Provavelmente, qualquer framework de persistência vai servir.

O desenvolvimento de microsserviços está mais relacionado com como disponibilizamos nossos serviços do que com como implementamos esses serviços.

1.2 Thorntail ou Spring Boot?

Apesar de termos visto que o desenvolvimento de microsserviços tem mais relação com a forma de entregar do que com a tecnologia em si, este é um livro prático, então vale a pena discutirmos um pouco sobre a tecnologia.

Assim como no desenvolvimento de aplicações "tradicionais" ao pensarmos em solução full-stack temos o Java EE ou Spring; quando desenvolvemos microsserviços temos as variações dessas opções: Thorntail (Java EE) ou Spring Boot (Spring).

O objetivo aqui não é comparar essas opções, apenas dizer que ambas cumprem o mesmo papel, cada uma à sua maneira (e para falar a verdade são bem parecidas).

Voltando para o campo dos microsserviços, como a ideia é servir cada parte da nossa aplicação de forma independente, a abordagem mais comum é empacotar cada serviço (ou

microsserviço) como um "executável" independente. Assim, em vez de termos diversos arquivos `war` para colocar em um mesmo servidor de aplicação, temos diversos `jar` executáveis, sendo que cada um já sobe o serviço e o servidor junto. Esses arquivos com tudo necessário para executar nossa aplicação são chamados de *uber-jar* ou *fat-jar*.

Para alguém que está familiarizado com o desenvolvimento Java, pode parecer sem sentido subir diversos servidores, um para cada `war`, já que economizaríamos recursos se subíssemos um único servidor e colocássemos nele os serviços com as mesmas características. Por exemplo, os que precisam de muita escalabilidade em um servidor, os que precisam de mais confiabilidade em outro etc.

Porém, com microsserviços, podemos ter um serviço feito em Java, outro em PHP, um terceiro em .Net e assim por diante. Em vez de tentar gerenciar cada um desses serviços em seus respectivos servidores, o que poderia aumentar exponencialmente a complexidade do ambiente, acabou virando um padrão de mercado a utilização de contêineres Docker, como dito anteriormente.

Logo, como cada serviço nosso vai rodar dentro de um contêiner, que é como uma máquina virtual independente, precisamos que esse serviço seja executável (*bootable*). Para facilitar tudo isso, em vez de termos que colocar dentro da imagem do Docker um servidor de aplicações, nosso `war`, e ainda fazer as configurações necessárias, é que surgiu o Spring Boot, que oferece a solução para construir um `war` executável para a *stack* do Spring. E depois surgiu o Thorntail (antigo WildFly Swarm), que faz o equivalente para a *stack* Java EE.

Tanto o Spring Boot quanto o Thorntail possuem o mesmo "jeitão": temos geradores de projetos em que selecionamos as tecnologias que queremos utilizar, e eles nos entregam um ponto de partida rápido para nosso desenvolvimento. Não temos que nos preocupar

com ficar configurando o ambiente antes de ter uma aplicação rodando.

Além disso, em ambos os casos o projeto gerado vai usar o Maven (ou no caso do Spring Boot pode ser o Gradle, que é uma alternativa ao Maven baseada em Groovy) para gerar o `war` como é de costume, mas também será gerado o `jar` executável, que já inclui o servidor de aplicação todo configurado. No caso do Spring Boot, é usado um Tomcat, e no caso do Thorntail, é um Wildfly, que é o servidor que usaremos no decorrer deste livro. Nesse último caso, porém, o Wildfly utilizado sobe apenas os módulos necessários para nossa aplicação, e não todo o servidor de aplicação.

Sendo assim, como ambas as soluções são equivalentes, o que conta no final é a *stack* que vamos utilizar. Como este livro é baseado em Java EE, utilizaremos o Thorntail.

1.3 Criando primeiro serviço com JAX-RS

Como dito na seção anterior, assim como o Spring Boot, o Thorntail possui um gerador de projetos no endereço:

<https://thorntail.io/generator>.

Ali nós especificamos o *Group ID* e o *Artifact ID* do nosso projeto Maven e selecionamos as tecnologias que vamos utilizar na nossa aplicação. Para começar, vamos usar apenas o JAX-RS, que - como veremos a seguir - é uma API que permite criar serviços REST de uma maneira bem simples. Para isso, podemos tanto começar a digitar "JAX-RS" e selecionar a opção do combo que será apresentado, ou usar o modo completo que exibe todas as tecnologias disponíveis para selecionarmos através de *checkboxes*.

Para o exemplo do livro, especificaremos o *Group ID* como `br.com.casadocodigo` e o *Artifact ID* como `javacred-corretora`, além de selecionar o JAX-RS.

Feito isso, basta clicar na opção de gerar o projeto e ele será baixado para seu computador. Agora é só abri-lo na sua IDE favorita, que provavelmente já identificará que se trata de um projeto Maven e ele já poderá ser executado. Para executar nossa aplicação, basta usarmos o comando `mvn thorntail:run`, como pode ser visto na própria página de geração do projeto, e nosso "hello world" está pronto.

Utilize sua IDE para deixar esse comando de uma forma simples para que você apenas use a opção "Run" e a aplicação já execute.

Agora vamos de fato criar nosso serviço, que vai retornar a cotação de uma determinada quantidade de um ativo como moeda ou papel da bolsa de valores. Para isso, vamos criar uma classe, dentro do `src/main/java` da aplicação que baixamos do gerador, como o seguinte conteúdo:

TODO O CÓDIGO ESTÁ DISPONÍVEL NO GITHUB

É interessante irmos construindo nosso próprio código com a ajuda da nossa IDE. Obviamente, digitar cada linha de código apresentada no livro tornaria o processo bastante cansativo, mas lembre-se de que isso ajuda na aprendizagem, e que getters e setters, construtores, imports e o próprio completamento de código são algo que a IDE faz facilmente por nós.

Mesmo assim, caso queira acessar a versão acabada do arquivo, basta acessar o projeto no GitHub:

<https://github.com/gscordeiro/microservicos-ejbs/javacred-corretora/>.

```

package br.com.casadocodigo.javacred.corretora.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Path("/cotacao")
@Produces(MediaType.APPLICATION_JSON)
public class CotacaoBean {

    @GET
    @Path("/{codigo}")
    public double buscaCotacao(
        @PathParam("codigo") String codigoAtivo,
        @QueryParam("quantidade") double quantidade){

        System.out.println("Buscar cotação no banco de dados...");
        double cotacao = new Random().nextDouble();

        return cotacao * quantidade;
    }
}

```

Com esse simples código, agora temos um serviço que devolve a cotação, que na verdade é um número randômico. Para testar basta executar o projeto e acessar no navegador:

<http://localhost:8080/cotacao/PETR4?quantidade=100>.

Aqui usamos apenas o básico do JAX-RS para criar algo até mesmo parecido com uma *servlet*, mas, por mais que não pareça agora, esse serviço já devolve um JSON, então já temos algo melhor do que uma *servlet*.

Por mais simples que o exemplo seja, o importante é que pudemos perceber que fazer um serviço REST em Java não é algo complexo. Os detalhes do porquê de cada anotação nós veremos no capítulo **10. Criando WebServices com JAX-RS**, pois o intuito desses dois

primeiros capítulos é oferecer um início rápido para nosso aprendizado.

1.4 Adicionando persistência de dados

Como na seção anterior já conseguimos um serviço REST funcionando, agora é hora de deixar o exemplo mais parecido com a vida real. Caso nosso serviço seja apenas um calculador ou combinador de outros serviços, pode ser que o JAX-RS seja a única coisa necessária, ou talvez apenas o JAX-RS combinado com CDI, pois este vai permitir a injeção de dependências de uma forma mais sofisticada.

Mas é muito comum que tenhamos também algum acesso a dados nos nossos serviços, então para isso teremos que adicionar mais algumas dependências no nosso projeto.

Quando formos gerar nosso projeto usando o gerador do Thorntail, até podemos já selecionar essas tecnologias para que nosso projeto já venha pré-configurado. Mas agora vamos adicioná-las diretamente no `pom.xml` e veremos que também é bastante simples.

As tecnologias extras de que precisaremos serão: JPA, para acesso a banco de dados; EJB, para que nossos *beans* tenham gerenciamento automático de transação; e Datasources, que criam o caminho para o banco de dados que será usado pela JPA.

Como vamos continuar somente com a classe `CotacaoBean`, não precisaremos de CDI para fazer a injeção das dependências, mas no caso de uma aplicação mais complexa também o adicionaríamos.

Para configurarmos nossa aplicação, deixaremos o seu `pom.xml` da seguinte forma:


```

...
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>jaxrs</artifactId>
</dependency>
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>jpa</artifactId>
</dependency>
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>datasources</artifactId>
</dependency>
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>ejb</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.18</version>
</dependency>
...

```

A dependência `jaxrs` já estava lá, apenas adicionamos as outras três. A necessidade da dependência da `jpa` é a mais óbvia, afinal precisaremos trabalhar com persistência. Já a dependência de `datasources` se faz necessária porque queremos configurar um `datasource` diferente do `ExampleDS` que vem pré-configurado em todo servidor Wildfly.

A dependência de `ejb` é simplesmente porque precisamos que nosso *bean* seja transacional para podermos persistir informações no banco de dados, e não vamos querer fazer isso manualmente. Se fôssemos adicionar a dependência de CDI, bastaria colocar mais uma entrada com o `artifactId` igual a `cdi`.

Por fim, temos a dependência do *driver* do MySQL que será usado na nossa camada de persistência.

Agora que temos as dependências configuradas, vamos voltar à nossa classe `CotacaoBean` e atualizá-la para ficar assim:

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.servlet.http.HttpServletRequest;
import javax.ws.rs.core.Context;
... //mantém os outros imports

@Stateless
@Path("/cotacao")
@Produces(MediaType.APPLICATION_JSON)
public class CotacaoBean {

    @PersistenceContext
    private EntityManager entityManager;

    @GET
    @Path("/{codigo}")
    public double buscaCotacao(
        @PathParam("codigo") String codigoAtivo,
        @QueryParam("quantidade") double quantidade,
        @Context HttpServletRequest request){

        System.out.println("Buscar cotação no banco de dados...");
        double cotacao = new Random().nextDouble();

        LogConsulta logConsulta = new LogConsulta(codigoAtivo,
            cotacao, quantidade, request.getRemoteAddr());

        entityManager.persist(logConsulta);

        return cotacao * quantidade;
    }
}
```

De diferente, no nosso código, temos a anotação `@Stateless` na nossa classe, o que faz com que agora ela gerencie transações automaticamente. Também injetamos o `EntityManager` com

@PersistenceContext , e após a "busca" da cotação, salvamos uma espécie de log dessa consulta através da classe LogConsulta .

Antes de analisarmos o código da classe LogConsulta , que é uma entidade bem simples, perceba o parâmetro extra no método buscaCotacao , que serve para acessarmos o objeto que representa a requisição do usuário. Geralmente, esse tipo de informação de controle pode ser injetado usando a anotação @Context . Tendo em mãos o HttpServletRequest , podemos recuperar o endereço IP do usuário que está acessando nosso serviço para então salvar isso no nosso log.

Agora sim, vamos à classe LogConsulta :

```
@Entity
public class LogConsulta {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String codigoAtivo;
    private double valorAtivo;
    private double quantidade;
    private Date data;
    private String ip;

    @Deprecated
    public LogConsulta(){}

    public LogConsulta(String codigoAtivo, double valorAtivo,
        double quantidade, String ip) {

        this.codigoAtivo = codigoAtivo;
        this.valorAtivo = valorAtivo;
        this.quantidade = quantidade;
        this.data = new Date();
        this.ip = ip;
    }

    //getters and setters...
}
```

Aqui criamos uma entidade simples, com um construtor padrão que anotamos com `@Deprecated` apenas para desencorajar seu uso, uma vez que temos um construtor que de fato recebe os valores necessários para criar o objeto de forma íntegra. Por mais que não venhamos a usá-lo diretamente, esse construtor padrão é muitas vezes usado por partes do Java EE, como o JAX-RS etc.

Colocamos também uma chave primária com autoincremento do tipo `IDENTITY`. Assim, no MySQL e no SQLServer será usada a propriedade própria para isso, e em bancos que não têm isso de forma nativa, a JPA cria uma *sequence* para fazer o mesmo trabalho.

Visto o código da nossa entidade, o próximo passo é a configuração da persistência.

Configurando o arquivo `persistence.xml`

Se tivéssemos gerado nossa aplicação já com a dependência da JPA, nossa aplicação já teria o arquivo `persistence.xml`, que é o que configura a camada de persistência, mas como adicionamos essa dependência manualmente, vamos criar esse arquivo no caminho `src/main/resources/META-INF/persistence.xml`, e com o seguinte conteúdo (disponível no GitHub):

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  version="2.1">
  <persistence-unit name="default">
    <properties>
      <property name="hibernate.hbm2ddl.auto"
        value="update" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL55Dialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

```
        <property name="hibernate.use_sql_comments"
            value="true" />
    </properties>
</persistence-unit>
</persistence>
```

As configurações mais importantes aqui são duas:

`hibernate.hbm2ddl.auto` , que serve para dizermos à JPA para gerar a estrutura do banco de dados com base nas nossas classes, e a `hibernate.dialect` , em que indicamos o tipo de dialeto que queremos, que no caso é o do MySQL55. As demais propriedades são apenas para gerar SQL formatado no console.

Agora que já temos a JPA configurada, falta criarmos o datasource apontando para nosso banco MySQL.

Criando um datasource

Para configurar coisas de ambiente no nosso projeto Thorntail, criamos o arquivo `src/main/resources/project-defaults.yml` . Esse é um tipo de arquivo hierárquico, como se fosse um XML, mas sem a necessidade de ficarmos abrindo e fechando tags. Tudo é feito por indentação, como podemos ver (também disponível no GitHub):

```
thorntail:
  datasources:
    data-sources:
      JavacredCorretoraDS:
        driver-name: mysql
        connection-url:
          jdbc:mysql://localhost/javacred_corretora?
            createDatabaseIfNotExist=true&useSSL=false
        user-name: root
        password: root
```

Aqui eu quebrei a propriedade `connection-url` em três linhas para facilitar a leitura, mas deixe tudo em uma única linha no seu projeto. Com esse arquivo, terminamos as alterações necessárias para nosso projeto ter suporte à persistência. Nesse último arquivo,

definimos um datasource chamado `JavacredCorretoraDS`, mas como é o único DS no nosso ambiente, nem precisamos especificá-lo no nosso `persistence.xml`. Também já assumimos que o MySQL está instalado e configurado conforme foi colocado nesse arquivo.

Com isso pronto, basta executarmos novamente nosso projeto e acessar a tabela `LogConsulta` do banco `javacred_corretora` pelo cliente do MySQL, e veremos os logs sendo incluídos lá.

Devolvendo um objeto complexo

Agora que nossa persistência está funcionando, podemos alterar o retorno do nosso método para devolver o objeto `LogConsulta` em vez de um simples `double`. Como dito antes, por mais que fosse apenas um `double`, a resposta do nosso serviço já era em formato JSON. Então, apenas trocando o retorno para o `LogConsulta` já teremos sua versão JSON, afinal essa conversão de Java para JSON é nativa no Java EE.

```
...
@GET
@Path("/{codigo}")
public LogConsulta buscaCotacao(
    @PathParam("codigo") String codigoAtivo,
    @QueryParam("quantidade") double quantidade,
    @Context HttpServletRequest request){

    System.out.println("Buscando cotação no banco de dados...");
    double cotacao = new Random().nextDouble();
    LogConsulta logConsulta = new LogConsulta(codigoAtivo,
        cotacao, quantidade, request.getRemoteAddr());

    entityManager.persist(logConsulta);

    return logConsulta;
}
...
```

Basta executarmos novamente a chamada pelo navegador e teremos um objeto JSON complexo em vez de uma simples resposta numérica. Para visualizar melhor, instale em seu navegador alguma extensão (plugin) que formate o JSON, caso ele não tenha suporte nativo.

1.5 EJBs e Java EE no geral não são muito complexos para microsserviços?

Se formos recapitular o que foi preciso fazer neste capítulo, temos um arquivo `xml` e um `ym1` de configuração que são chatinhos, especialmente o `xml`. Podíamos até ter gerado o projeto com JPA para que o arquivo viesse criado desde o início e também deixado de criar um datasource dedicado, o que praticamente eliminaria qualquer configuração, mas seria algo meio artificial, na minha opinião.

Infelizmente, esse tipo de configuração não é um privilégio do Java EE, tampouco do Java em si. Seja qual for a tecnologia utilizada, em algum momento teremos que configurar pelo menos o lugar onde os dados serão persistidos.

Creio que a análise mais importante em relação à complexidade ou não do Java EE esteja mais relacionada com os arquivos Java propriamente ditos, pois se estivéssemos usando Spring com JPA, apenas para comparar, também teríamos um `persistence.xml`. Então, olhando para a classe `CotacaoBean`, vimos que o que a tornou um EJB foi uma simples anotação: `@Stateless`, não sendo necessário qualquer configuração para isso.

Novamente: Thorntail ou Spring Boot?

Se formos comparar o Java EE para microsserviços com Thorntail com a o Spring Boot, veremos que o jeitão de fazer as coisas é

muito equivalente. Tirando a questão de gosto pessoal, veremos que hoje não há tanta diferença assim no modelo de programação dessas duas *stacks*.

A diferença maior pode ser encontrada no resultado final do projeto: o `(uber|fat)-jar`. Por mais que o Wildfly seja todo modular, e que somente os módulos necessários sejam adicionados ao nosso projeto, ele ainda ocupa mais espaço em disco e um pouco mais de memória que o Tomcat, que vem por padrão no Spring Boot. Mas essa diferença não é tão grande assim, além disso, quando adicionamos a imagem do Docker por cima, a diferença praticamente some. De qualquer forma, é bom procurar comparativos com as últimas versões de cada um, pois isso pode mudar com o tempo.

Para fechar o capítulo

Neste primeiro capítulo, vimos como criar um primeiro projeto de uma forma bem prática com o *Thorntail generator* e já saímos com nosso *hello world* REST funcionando. Fomos avançando na implementação de um serviço simples e depois na configuração de um ambiente que tem tudo de que vamos precisar em um projeto real, que basicamente é o JAX-RS mais o controle de transações e persistência de dados.

No próximo capítulo, vamos avançar um pouco mais nossa aplicação, vendo como criar clientes para consumir os serviços REST, mas também tópicos mais avançados de escalabilidade através do uso de cache e requisições (`GET` e `POST`) condicionais.

CAPÍTULO 2

Serviços REST com JAX-RS

No capítulo anterior, já vimos o básico de JAX-RS ao usar as anotações `@Path` e `@GET`. Isso foi o suficiente para um *hello world*, mas não para implementarmos uma primeira aplicação.

Neste capítulo, vamos avançar um pouco mais com o intuito de termos o necessário para o início rápido de um *back-end* Java real, feito com serviços REST.

Com isso, teremos uma boa base para quanto chegarmos ao capítulo **10. Criando WebServices com JAX-RS**, no qual veremos mais detalhes sobre WebServices e como expor nossos serviços REST e construir nossos clientes via JavaScript, por exemplo.

2.1 Escalando serviços REST através do uso de Cache

Como já dito algumas vezes, microserviços e serviços REST não são sinônimos, mas é muito difícil dissociarmos uma coisa da outra. E é fato que esse tipo de arquitetura tem muito da sua razão de existir pensando na escalabilidade. Como também já vimos, por termos partes específicas da nossa aplicação separadas por assunto, é mais fácil escalar somente a parte que realmente precisa disso. E, pensando na escalabilidade, não podemos deixar de pensar no uso de cache.

Para desenvolver essa funcionalidade, vamos continuar com o exemplo do capítulo passado, que é um serviço responsável por devolver a cotação de ativos como moedas, papéis da bolsa etc. Agora imagine a quantidade de requisições para se obter a cotação do dólar comercial que um serviço como o nosso recebe por dia. E o

pior, ou melhor para o nosso caso, é que uma boa parte dessas requisições receberá como resultado praticamente o mesmo valor. Digo o melhor para o nosso caso, pois é aí que pode ser aplicado um sistema de cache com muita eficiência.

Outro exemplo em que o uso de cache seria muito eficiente é em sites de notícias, pois as notícias mudam ou são acrescentadas em uma velocidade muito menor que a quantidade de requisições que são recebidas, o que permite fazer um cache delas para aliviar o serviço.

Voltando ao nosso contexto, talvez você já tenha notado que diversos sites com conteúdo de economia apresentam as cotações de moeda e bolsa de valores dizendo que eles são exibidos com até quinze minutos de atraso, e que, se quisermos um valor em tempo real, podemos fazer uma assinatura e pagar por isso.

Se pensarmos um pouco, o que esses portais financeiros fazem é exatamente o que também vamos fazer: colocar um cache de quinze minutos para as cotações de câmbio e papéis da bolsa. Para oferecer a versão sem esse atraso, basta usarmos a versão vista no capítulo passado juntamente com algum mecanismo de autenticação que pode ser feito com o uso de **JAAS (Java Authentication and Authorization Service)**.

O resultado esperado com o uso do cache é que, se nosso portal recebesse em média um milhão de consultas por dia somente para a cotação do dólar comercial, usando o cache de quinze minutos diminuiríamos isso para 96 requisições por dia (4 vezes por hora, 24 horas por dia). Com isso, podemos baratear nossa infraestrutura alocando menos instâncias de servidores mais caros, que efetivamente processam algo, e deixando a maior carga para camadas mais baratas que funcionam como um simples proxy, que faria o cache dos resultados.

Mas agora que a vantagem ficou clara, vamos de fato ao desenvolvimento. Vale lembrar que aqui estamos usando o projeto

javacred-corretora , mas tudo o que estamos vendo de REST neste capítulo se aplica também para os serviços REST que veremos no capítulo 10.

Para evoluir nosso exemplo, em vez de mudá-lo como fizemos no capítulo anterior, vamos criar uma nova versão do nosso serviço de cotação. Assim podemos manter um histórico de como o código foi evoluindo, mas principalmente, desenvolveremos mantendo a retrocompatibilidade. Para isso, vamos criar uma nova classe e colocar um /v2 no caminho do serviço. Assim a versão anterior continua funcionando.

```
import javax.ws.rs.core.CacheControl;
import javax.ws.rs.core.Response;
import java.util.concurrent.TimeUnit;
...
@Stateless
@Path("/cotacao/v2")
@Produces(MediaType.APPLICATION_JSON)
public class CotacaoBeanV2 {

    @PersistenceContext
    private EntityManager entityManager;

    @GET
    @Path("/{codigo}")
    public Response buscaCotacao(
        @PathParam("codigo") String codigoAtivo,
        @QueryParam("quantidade") double quantidade,
        @Context HttpServletRequest request){

        System.out.println("Busca fake no banco de dados...");
        double cotacao = new Random().nextDouble();

        LogConsulta logConsulta = new LogConsulta(codigoAtivo,
            cotacao, quantidade, request.getRemoteAddr());

        entityManager.persist(logConsulta);

        CacheControl cc = new CacheControl();
```

```
//delay padrão de serviços gratuitos
cc.setMaxAge(((int) TimeUnit.MINUTES.toSeconds(15)));

    return Response.ok(logConsulta).cacheControl(cc).build();
}
}
```

Este serviço não tem muita coisa diferente do que estamos acostumados a usar. A novidade é o uso da classe `CacheControl`, e que, em vez de devolver o objeto `LogConsulta`, devolvemos um `Response`. Com esse tipo de retorno conseguimos controlar com maior precisão os tipos de resposta que vamos criar, qual código HTTP usaremos e assim por diante.

Com relação ao `CacheControl`, usamos apenas seu `maxAge`, que é o tempo que o cache deve durar, especificado em segundos. Deixamos nosso cache público por padrão, assim permitiremos que o cache seja feito por algum intermediário, como um proxy. Se tivéssemos especificado que o cache é privado, somente o cliente final poderia guardar seu valor.

Agora faça o teste no seu navegador, chame o serviço passando algum código de ativo cuja cotação deseja buscar. Depois baixe o servidor e execute novamente a consulta pelo navegador e perceba que o resultado continua lá. Infelizmente nem todo navegador se comporta como deveria, então tente usar o Firefox, pois ele faz o serviço corretamente. Mas fique tranquilo, apesar de termos testado no navegador, na prática faremos um cliente programático que não terá essa falta de educação que alguns navegadores têm, e que acabam não tratando o cache da forma correta.

Aqui não veremos as diversas formas de se criar um cliente REST, pois isso será tratado no capítulo 10, mas a forma que veremos por enquanto já vai nos atender.

Antes de mexer no código Java, precisamos colocar as seguintes dependências no `pom.xml` para podermos desenvolver nossos testes:

```

<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-client</artifactId>
  <version>3.9.1.Final</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jackson2-provider</artifactId>
  <version>3.9.1.Final</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>

```

Essas são as dependências do **RestEasy**, que é a implementação do JAX-RS que vem dentro do Wildfly. Adicionadas as dependências, podemos ir para os testes Java que deverão ficar no diretório `src/test/java` do nosso projeto. A descrição do funcionamento está logo após o código.

```

import org.junit.*;
import javax.ws.rs.client.*;
import javax.ws.rs.core.*;
import org.jboss.resteasy.client.jaxrs.cache.BrowserCacheFeature;
...

```

```

@Ignore
public class LogConsultaTest {

    @Test
    public void testaCache() throws ExecutionException,
                                   InterruptedException {

        //criando os objetos básicos do cliente JAX-RS
        Client client = ClientBuilder.newClient();
    }
}

```

```

//função de cache
client.register(new BrowserCacheFeature());

WebTarget javacred = client.target("http://localhost:8080");

WebTarget cotacaoBean = javacred.path("cotacao/v2");
Response response = null;

//controle para validar se o cache ainda está válido

//Tempo do cache no servidor
Duration tempoCache = Duration.ofMinutes(15);
//Duration tempoCache = Duration.ofSeconds(10); //para teste
Instant inicio = Instant.now();
Double cotacaoCache = null;

for(int i = 0; i < 6; i++){

    //executa o GET no nosso serviço
    response = cotacaoBean
        .path("{codigo}").resolveTemplate("codigo", "BBAS3")
        .queryParam("quantidade", 200)
        .request(MediaType.APPLICATION_JSON)
        .get();

    LogConsulta logConsulta = response
        .readEntity(LogConsulta.class);

    //verifica quanto tempo se passou até aqui
    Duration tempoDecorrido =
        Duration.between(inicio, Instant.now());

    if(tempoDecorrido.compareTo(tempoCache) > 0){
        System.out.println("Cache expirado!");
        inicio = Instant.now();
        cotacaoCache = null;
    }

    //se ainda não tem valor do cache, ele é atribuído aqui
    if (cotacaoCache == null){
        cotacaoCache = logConsulta.getValorAtivo();
    }
}

```

```

    }

    Assert.assertEquals(cotacaoCache,
        logConsulta.getValorAtivo(), 0.001);

    System.out.println(cotacaoCache + " = " +
        logConsulta.getValorAtivo());

    System.out.println("=====SLEEP 5=====");
    Thread.sleep(5_000);
}

response.close();
}
}

```

Normalmente não colocamos `System.out.println()` em testes, até porque eles são executados automaticamente e geralmente nem estamos olhando para ele. Porém, aqui vamos usar esses testes de uma maneira mais didática. Vamos executá-los manualmente e verificar seu comportamento.

Como estamos executando o Thorntail via Maven, se não colocarmos `@Ignore` em cima da nossa classe, o teste será executado antes de o servidor subir, e então teremos um erro. Dessa forma, vamos poder executar os testes manualmente depois que o servidor estiver executando. No capítulo **9. Testando nossos serviços** veremos formas mais completas de lidar com testes, e fazer com que o próprio teste execute o servidor. Mas por ora vamos manter `@Ignore` nos testes do projeto `javacred-corretora`.

Já dentro do código de testes, logo após criarmos o `client`, adicionamos nele a funcionalidade de fazer cache como se fosse o navegador. Perceba que essa é uma funcionalidade do RestEasy, e não algo padrão do Java EE. Mas se estivermos usando outra implementação, provavelmente ela terá uma funcionalidade equivalente, então basta buscarmos na documentação.

Caso nossa aplicação necessite rodar em mais de um servidor, podemos parametrizar o nome da classe que faz o cache via banco de dados ou arquivo de configuração e usar o método `register` passando um `Class` em vez de um `Object`.

Já a chamada do serviço propriamente dito executa dentro de um laço, pois queremos chamar várias vezes o mesmo serviço e ter a certeza de que o cache está funcionando. Não adiantaria executar várias vezes o mesmo código se, por exemplo, ele estivesse dentro de um método `main` (sem laço), pois ao final de cada execução todos os objetos seriam removidos da memória, inclusive o que guarda o cache. Por isso, o laço é importante.

Perceba que dentro do nosso teste colocamos um controle para saber se o cache do servidor expirou ou não, então é importante colocar o mesmo valor no servidor e no teste. Nesse controle, além de contar o `tempoDecorrido` para saber se o cache ainda é válido, usamos a variável `cotacaoCache` para guardar a primeira resposta do servidor até que o cache expire novamente.

Como iniciamos o tempo do nosso cache com **15 minutos**, podemos executar nosso projeto, subindo o servidor e, logo em seguida, executar nosso teste. Assim que perceber pelo console que a primeira execução ocorreu com sucesso, podemos baixar o servidor e perceber que o teste continuará executando até o final. Afinal, o cache está maior que o tempo de execução do nosso teste.

Para testar mais um pouco, podemos diminuir o tempo do cache no servidor e no teste de **15 minutos** para **10 segundos** e executar novamente o servidor e o teste. Perceba que o teste continuará funcionando, mas que no servidor teremos agora 3 execuções da consulta de cotação, pois nosso teste executa por 30 segundos (6 x 5), e nosso cache dura 10 segundos.

Da mesma forma, ao analisarmos o retorno de cada chamada ao servidor no console do cliente, vemos que sempre temos duas requisições com o mesmo valor de cotação, ainda que nosso

servidor sempre gere uma nova cotação randomicamente. Isso porque a segunda resposta é o cache da primeira.

2.2 Revalidando o cache com GETs condicionais

Na seção anterior, vimos que o cache é uma forma muito boa de economizar muitas requisições ao nosso servidor; isso sem falar do cache da camada de persistência, que é um nível a mais de proteção contra consumo desnecessário de recursos. Agora veremos que podemos revalidar um cache que já expirou, prorrogando um cache que a princípio era inválido.

No exemplo da seção passada, usamos uma cotação, que pode ser simples de recuperar, mas que deveria usar cache pelo alto número de requisições. Além disso, uma cotação não muda. O que ocorre é que temos uma nova cotação para aquele ativo, tanto que ao final do dia temos o valor da última cotação como o valor de fechamento. Mas podemos consultar a variação da cotação no decorrer do dia.

Para termos um cenário de revalidação de cache, precisamos de um objeto que pode ou não ter mudado, e queremos evitar ter que buscar esse objeto toda hora no banco se muitas vezes o resultado será o mesmo. Veja que novamente podemos falar da camada de persistência para nos ajudar com isso, mas aqui vamos resolver o problema somente com JAX-RS.

Como a cotação não se aplica ao que precisamos para esse cenário, vamos evoluir o exemplo da nossa corretora para agora termos o envio de ordens de compra e venda de ativos. No cenário de uma corretora real, geralmente podemos alterar nossa ordem antes de ela ser executada. Esse intervalo pode ser bem curto ou pode ser de semanas, dependendo dos critérios (preço mínimo, fracionamento) que colocamos na ordem que enviamos à corretora. Essa explicação é somente uma contextualização da mutabilidade

de uma ordem de compra ou venda, mas não vamos implementar essas regras de negócio no exemplo pois não é nosso objetivo aqui.

O controle se a `Ordem` mudou ou não será feito usando a data da última alteração. Existe também outra abordagem que é através do *hash* do objeto, analisando se o seu conteúdo foi alterado, mas vamos ver isso mais adiante neste capítulo, quando formos construir `POSTs` condicionais.

Vamos criar a classe `Ordem` com a propriedade `ultimaModificacao`, e criar uma consulta para buscar a última vez que uma determinada ordem foi modificada:

```
@Entity
@NamedQuery(name = BUSCAR_DATA_ULTIMA_ALTERACAO, query =
    "select o.ultimaModificacao from Ordem o where o.id = :id")
public class Ordem {

    public static final String BUSCAR_DATA_ULTIMA_ALTERACAO =
        "Ordem.buscarDataUltimaAlteracao";

    public enum Tipo {COMPRA, VENDA}

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private Tipo tipo;
    private String codigoAtivo;
    private double quantidade;
    private double valorAtivo;
    private double valorTotal;
    private Date ultimaModificacao;
    private String ip;

    @Deprecated
    public Ordem(){}

    public Ordem(Tipo tipo, String codigoAtivo, double quantidade,
        double valorAtivo, String ip) {
        this.tipo = tipo;
        this.codigoAtivo = codigoAtivo;
```

```

        this.quantidade = quantidade;
        this.valorAtivo = valorAtivo;
        this.valorTotal = valorAtivo * quantidade;
        this.ip = ip;
        this.ultimaModificacao = new Date();
    }

    @PrePersist
    @PreUpdate
    public void updateBean(){
        ultimaModificacao = new Date();
    }

    //getters and setters

}

```

Não tem nada de muito diferente nessa classe em relação a qualquer outra entidade, o importante para nós é a propriedade `ultimaModificacao` que é atualizada sempre que o objeto for escrito no banco (`insert` ou `update`). Com isso, sabemos exatamente a última vez em que nossa `Ordem` foi atualizada, que é a condição para revalidarmos o cache. Vamos então criar uma nova classe para lidar com essa *entity*:

```

import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Stateless
@Path("/ordem")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrdemBean {

    @PersistenceContext
    private EntityManager em;

    @GET
    @Path("/{id}")
    public Response buscarOrdem(@PathParam("id") Integer id,

```

```

@Context Request request, @Context HttpHeaders httpHeaders){

    //escrevendo cabeçalhos no console
    System.out.println(httpHeaders.getRequestHeaders());

    Date ultimaAlteracaoOrdem = buscarUltimaAlteracaoOrdem(id);

    if(ultimaAlteracaoOrdem == null){
        return Response.noContent().build();
    }

    Response.ResponseBuilder builder = request
        .evaluatePreconditions(ultimaAlteracaoOrdem);

    //se foi alterado depois da última consulta,
    //gerar nova resposta
    if(builder == null){
        Ordem contrato = em.find(Ordem.class, id);
        ultimaAlteracaoOrdem = contrato.getUltimaModificacao();
        builder = Response.ok(contrato);
    }

    builder.lastModified(ultimaAlteracaoOrdem);

    CacheControl cc = new CacheControl();
    //troque MINUTES por SECONDS para facilitar os testes
    cc.setMaxAge(((int) TimeUnit.MINUTES.toSeconds(10)));
    builder.cacheControl(cc);
    return builder.build();
}

private Date buscarUltimaAlteracaoOrdem(Integer id) {
    try{
        return em.createNamedQuery(
            Ordem.BUSCAR_DATA_ULTIMA_ALTERACAO, Date.class)
            .setParameter("id", id).getSingleResult();
    }catch (NoResultException e){
        return null;
    }
}

```

```
}  
}
```

Com esse método, introduzimos o uso do

`request.evaluatePreconditions(Date)` . Como dito antes, podemos fazer essa validação com base na data da modificação do registro, ou com base em um hash que representa seu conteúdo, logo temos versões do método `evaluatePreconditions` para essas situações. Aqui estamos usando a versão que recebe a data.

Para que funcione corretamente, o cliente deve mandar a data que ele conhece da última modificação no parâmetro HTTP `If-Modified-Since` . Então a leitura é bem simples. Na prática, o cliente pede para fazer o `GET` *"se houve modificação a partir"* da data informada por ele nesse parâmetro.

E como ele fica sabendo a data da última modificação? Isso é informado por meio do `builder.lastModified(Date)` , que para o cliente vai no parâmetro de cabeçalho `Last-Modified` . Mas fique tranquilo porque interface `javax.ws.rs.core.HttpHeaders` tem todas essas constantes para nós, e ainda assim geralmente a API do JAX-RS manipula isso automaticamente, então são poucas as vezes em que precisamos fazer uso direto. No entanto, mesmo que essa conversa entre cliente e servidor seja simplificada para nós, é importante sabermos como ocorre o diálogo para entendermos o funcionamento do `GET` condicional.

Basicamente, se o resultado estiver no cache, funciona igual vimos antes, a requisição nem chega à nossa classe. Mas se o cache estiver "vencido", o nosso método é chamado e então buscamos no banco de dados a data da última atualização da `Ordem` por meio do método `buscarUltimaAlteracaoOrdem(Integer id)` e comparamos com o valor da última atualização conhecida pelo cliente. Se o cliente estiver com a última versão, o `evaluatePreconditions(Date)` devolve uma instância de `Response.ResponseBuilder` .

Seguindo no código, vemos que sendo esse valor diferente de nulo, apenas "recarimbamos" a requisição com os dados do cache e devolvemos a resposta. Isso devolverá para o cliente uma resposta com o código HTTP 304 que significa que não há alterações no servidor. O melhor é que nesse cenário não fizemos qualquer processamento no servidor, a não ser uma consulta para buscar a última versão. Sequer buscamos o `Ordem` no banco de dados.

USO DE CACHE NA CAMADA DE PERSISTÊNCIA

Estamos vendo como evitar processamento desnecessário no nosso servidor por meio do uso de cache. Vimos que conseguimos evitar a busca da `Ordem` no banco de dados, trazendo apenas a data da última atualização. Mas ainda temos uma dimensão a mais para otimizar nossa aplicação e torná-la mais escalável, que é usar o cache de segundo nível e cache de consulta da JPA. Nesse caso, conseguiríamos evitar até mesmo a consulta no banco para trazer a data da última alteração. Neste livro não estamos abordando muito de JPA, mas na própria Casa do Código temos outros livros sobre JPA, inclusive um meu chamado: *Aplicações Java para a web com JSF e JPA*. Apesar de tratar também de JSF, tem diversos capítulos dedicados apenas a JPA, que independem do framework MVC utilizado.

Caso o `evaluatePreconditions(Date)` devolva nulo, significa que a data passada pelo cliente está desatualizada em relação à nossa última alteração, e então temos que fazer o processamento necessário para devolver a resposta atualizada para o cliente. No caso, apenas temos uma consulta no banco para buscar a `Ordem`, mas podemos ter casos em que a montagem completa dessa resposta pode envolver um processamento mais pesado. Imagine se tivéssemos que retornar uma dívida com correção monetária atualizada até aquele exato momento. Esse tipo de cálculo será visto no capítulo **12. Métodos assíncronos e paralelismo para aumento de**

performance, e muitas vezes não é algo simples de fazer. Então essa nossa abordagem evita esse tipo de processamento desnecessário.

Mas o que muda no nosso cliente?

2.3 Cliente usando GET condicional

O cliente do `GET` condicional acaba não mudando em relação ao cliente do cache que vimos antes. A diferença é que agora perceberemos que a API do JAX-RS vai automaticamente manipular as informações de cabeçalho para que essa comunicação ocorra do jeito certo.

Vamos novamente criar uma classe no nosso diretório de testes do projeto.

```
@Ignore
public class OrdemBeanTest {

    @Test
    public void testarGetCondicional()
        throws InterruptedException {

        //criando os objetos básicos para um cliente JAX-RS
        Client client = ClientBuilder.newClient();

        client.register(new BrowserCacheFeature());

        WebTarget javacred = client.target("http://localhost:8080");

        WebTarget ordemBean = javacred.path("ordem");
        Response response = null;

        for(int i = 0; i < 6; i++){

            //executando o GET no nosso serviço
```

```

        response = ordemBean.path("{id}")
            .resolveTemplate("id", 1)
            .request(MediaType.APPLICATION_JSON)
            .get();

        OrdemTO ordemTO = response.readEntity(OrdemTO.class);

        System.out.println(new Date());
        System.out.println(response.getStatus());
        System.out.println(ordemTO);

        //escrevendo cabeçalhos no console
        System.out.println(response.getHeaders());

        System.out.println("=====SLEEP 5=====");
        Thread.sleep(5_000);
    }

    response.close();
}
}

```

A dinâmica do teste aqui, como já foi dito, é a mesma do teste que fizemos para testar o cache, porém sem aqueles controles de tempo. Isso porque nosso foco aqui não é controlar com base no tempo, e sim na mudança do objeto.

Além da classe de testes, temos que criar a classe `OrdemTO`, que vai representar a entidade `Ordem` no lado do cliente. Por mais que aqui estejamos usando o mesmo projeto, temos que ter em mente que uma das vantagens do uso do REST é a maior liberdade que temos na definição dos tipos no cliente e no servidor em relação ao uso de clientes EJB, por exemplo. Então, para explorar isso, vamos definir esse novo tipo no diretório de testes do nosso projeto:

```

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
...
@JsonIgnoreProperties(ignoreUnknown = true)
public class OrdemTO {

```



```

public enum Tipo {COMPRA, VENDA}

private Integer id;
private Tipo tipo;
private String codigoAtivo;
private double quantidade;
private double valorAtivo;
private double valorTotal;
private Date ultimaModificacao;
private String ip;

@Deprecated
public OrdemTO(){}

public OrdemTO(Tipo tipo, String codigoAtivo,
    double quantidade, double valorAtivo, String ip) {

    //construtor usando fields
}

//getters e setters
}

```

A anotação `@JsonIgnoreProperties` é para que as propriedades que existem no servidor, e consequentemente no JSON, e que não existem na nossa classe sejam simplesmente ignoradas.

Voltando ao código do nosso teste perceba que tanto no código do cliente quanto no código do servidor nós escrevemos no console o cabeçalho da requisição. Isso facilita que vejamos a conversa que é estabelecida através das entradas `Last-Modified`, enviada pelo servidor, e `If-Modified-Since`, enviada pelo cliente.

O ciclo de execução entre cliente e servidor também é similar entre este exemplo e o da cotação, em que o cliente executa por 30 segundos, fazendo uma consulta a cada 5 segundos e o servidor mantém o cache por 10 segundos (troque `MINUTES` por `SECONDS` no servidor para facilitar os testes). Com isso, a cada duas consultas do

cliente, uma vai ao servidor e a segunda recebe o valor do cache. Até aqui nada muda.

Porém, como `Ordem` é uma entidade nova, sem nenhum registro no banco de dados, e nós só veremos como fazer um `POST` para inserir registros na próxima seção, vamos incluir mais o seguinte trecho apenas para garantir que teremos ao menos um registro para nosso teste funcionar:

```
@Path("/ordem")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrdemBean {

    @PersistenceContext
    private EntityManager em;

    @PostConstruct
    public void configuracaoInicial(){
        if(em.find(Ordem.class, 1) == null){

            em.persist(new Ordem(Ordem.Tipo.COMPRA, "BBAS3",
                                26.71, 100, "localhost"));
        }
    }

    ...
}
```

Esse trecho checa se existe um registro com `id = 1`, e caso não exista, insere um registro no banco. Essa estratégia dá certo porque estamos usando chaves do tipo `IDENTITY` do MySQL, e assim as chaves são sequenciais dentro da mesma tabela.

Agora podemos executar o `testarGetCondicional()` e analisar o comportamento do servidor. No exemplo da consulta da cotação, quando o cache expirava, o método era executado de novo. Agora veremos que o método `evaluatePreconditions(Date)` nos permite não

executar novamente toda a lógica do método, que no caso é apenas uma consulta, mas que poderia ser algo bem mais complexo, e por isso vale a pena evitar.

Diferentemente do exemplo da consulta de cotação, analisando o console do servidor no exemplo da busca da `Ordem` vemos que a busca ocorre uma única vez. Nas outras duas requisições em que o cache expirou, o servidor apenas checa se houve mudança através da consulta da última alteração, e como não houve, não é preciso processar tudo de novo.

2.4 Server-Sent Events (SSE) como alternativa ao cache

Acabamos de ver o quanto o uso de cache pode economizar recursos do servidor, diminuindo a quantidade de processamento desnecessário para entregar um dado que já foi entregue antes. Porém, existem outras técnicas como o uso de eventos enviados pelo servidor aos clientes, seja por meio de SSE ou mesmo WebSockets.

Claro que tudo que vimos aqui não se perde pelo simples fato de que nem sempre os clientes estarão preparados para trabalhar com SSE, e nesse caso precisamos projetar nossos serviços para escalarem mesmo assim - e o uso do cache continua sendo nossa principal ferramenta nesse caso. Porém, como o intuito deste livro é servir de referência para a maior variedade de cenários possível, vamos discutir sobre o uso de SSE, e comentar sua diferença para os WebSockets no capítulo 11, na seção *Server-Sent Events ou SSE (JAX-RS 2.1, Java EE 8)*.

2.5 Enviando dados (POST) através do cliente REST

Até agora basicamente consultamos dados do servidor. Agora vamos realizar o envio. Mas antes, vamos recordar que o protocolo HTTP possui alguns métodos, sendo que os mais comuns são `GET` para recuperar dados, `POST` para inserir dados, `PUT` para atualizar dados e `DELETE` para remover dados, mas sem dúvida os mais usados acabam sendo os dois primeiros.

Os exemplos feitos até agora, nos quais recebíamos dados, utilizavam o método `GET`, como podemos ver por meio da chamada do método `get()`. E se voltarmos aos exemplos anteriores veremos o mesmo. Agora vamos utilizar o método `POST`, mas a lógica para os métodos `PUT` e `DELETE` é basicamente a mesma.

Antes de partirmos para o código do cliente, precisamos criar o código no servidor. Ele será um código simples, que apenas persiste a `Ordem` recebida.

```
@Stateless
@Path("/ordem")
public class OrdemBean {
    ...

    @POST
    public Ordem salvar(Ordem contrato){

        return em.merge(contrato);
    }

    //demais métodos permanecem iguais
}
```

E agora vamos enviar os dados de forma bem parecida com os códigos anteriores.

```
@Test
public void testarPost(){
```

```

Client client = ClientBuilder.newClient();

OrdemTO novaOrdem = new OrdemTO(OrdemTO.Tipo.COMPRA,
                                "PETR4", 100, 14.3, "test");

WebTarget ordemBean =
    client.target("http://localhost:8080/ordem");
Response response = ordemBean.request()
    .post(Entity.json(novaOrdem)); //POST!

OrdemTO ordemSalva = response.readEntity(OrdemTO.class);

Assert.assertNotNull(ordemSalva.getId());
Assert.assertEquals(OrdemTO.Tipo.COMPRA, ordemSalva.getTipo());
Assert.assertEquals("PETR4", ordemSalva.getCodigoAtivo());
Assert.assertEquals(100, ordemSalva.getQuantidade(), 0.001);

System.out.println(ordemSalva);
}

```

Pronto, com esse código simples já chamamos o método `salvar` do `OrdemBean`. Como ele é o único método que aceita `POST` no serviço, não precisamos especificar nenhum caminho através do método `path()` como nos exemplos anteriores, mas caso tenhamos mais de um é só especificar o caminho do método.

2.6 Criando uma interface Java do lado cliente para servir de proxy

Apesar de já conseguirmos trabalhar com REST e JSON de uma forma bem completa, ainda estamos fazendo código que manipula a API do JAX-RS de forma explícita. Não seria bom se tivéssemos uma classe Java que tivesse a mesma assinatura do meu serviço no servidor, e a complexidade para fazer isso funcionar pela rede

ficasse por trás dos panos? O bom é que podemos fazer isso usando o RESTEasy.

Antes de mais nada, precisaremos criar a interface Java com as mesmas anotações de caminho do serviço real, para que o RESTEasy faça o mapeamento de forma automática para nós. Como estamos trabalhando com o serviço `OrdemBean`, vamos criar uma interface chamada `OrdemBeanProxy`, no diretório de testes, que tem representado o código do nosso cliente, com o seguinte conteúdo:

```
@Path("/ordem")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public interface OrdemBeanProxy {

    @GET
    @Path("/{id}")
    OrdemTO buscar(@PathParam("id") Integer id);

    @POST
    OrdemTO salvar(OrdemTO ordem);

}
```

Mesmo que o verdadeiro `OrdemBean` possua mais métodos, só colocamos aqui os que precisaremos utilizar. Repare que o nome da interface é diferente da classe original, assim como os tipos dos parâmetros, que aqui estamos usando `OrdemTO`. A ideia é deixar explícito que não há qualquer relação de compilação entre o lado servidor e o cliente. A "mágica" é feita pelas anotações que especificam o mesmo caminho do serviço real.

Agora para usar, vamos criar uma nova classe de testes, que chamei de `OrdemBeanProxyTest`:

```
@Ignore
public class OrdemBeanProxyTest {
```

```

private OrdemBeanProxy ordemBean;

@Before
public void configuraProxy(){
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target("http://localhost:8080");
    ResteasyWebTarget rtarget = (ResteasyWebTarget) target;

    ordemBean = rtarget.proxy(OrdemBeanProxy.class);
}

@Test
public void testaBuscaComProxy(){

    OrdemTO ordem = ordemBean.buscar(1);

    Assert.assertNotNull(ordem);

    System.out.println(ordem);
}

@Test
public void testaPostComProxy(){

    OrdemTO novaOrdem = new OrdemTO(OrdemTO.Tipo.VENDA,
        "BBAS3", 200, 26.71, "test");

    OrdemTO ordemSalva = ordemBean.salvar(novaOrdem);

    Assert.assertNotNull(ordemSalva.getId());
    Assert.assertEquals(OrdemTO.Tipo.VENDA,
        ordemSalva.getTipo());
    Assert.assertEquals("BBAS3", ordemSalva.getCodigoAtivo());
    Assert.assertEquals(200, ordemSalva.getQuantidade(), 0.001);

    System.out.println(ordemSalva);
}
}

```

Iniciando a análise do nosso código pelo método `configuraProxy()`, vemos que iniciamos o `WebTarget` da mesma forma que fizemos nos

testes anteriores. Vale lembrar aqui que esse `WebTarget` é uma interface padrão da JAX-RS. Depois, fazemos um *cast* para acessar a interface do `RESTEasy` que está por trás da `WebTarget`. Fazendo um paralelo com a JPA, é como se acessássemos a `Session` do Hibernate por trás da `EntityManager`.

Estando com a `ResteasyWebTarget` na mão, podemos usar o método `proxy` passando a interface que tem as anotações do JAX-RS e então recebemos a instância do proxy que delegará todas as chamadas de método para o serviço REST. Isso é feito com base nas anotações que colocamos na interface que será nosso proxy.

Depois disso, podemos ver no código dos testes que usamos o nosso proxy como um objeto Java comum, deixando nosso código muito mais limpo, sem aquelas coisas típicas do JAX-RS.

2.7 Enviando dados através de um POST condicional

Logo antes neste capítulo vimos como fazer um `GET` condicional, em que cliente e servidor trocavam informações que possibilitam evitar processamentos desnecessários. Faremos o mesmo agora, só que na atualização de um dado.

A lógica vai ser bem parecida com a do `GET` condicional, porém em vez de fazer a validação no dado enviado pelo cliente através da data de alteração (já que estamos justamente tentando alterar o valor), faremos essa validação ou checagem com base no *hash* do objeto.

Então, antes de mais nada precisaremos criar esse atributo na classe `Ordem` e também criar uma consulta parecida com a que fizemos para buscar a última alteração. E o mais importante é que

antes de salvar o objeto no banco de dados precisamos alterar o seu *hash* para refletir o novo valor.

```
@Entity
@NamedQueries({
    @NamedQuery(name = BUSCAR_DATA_ULTIMA_ALTERACAO, ...),
    @NamedQuery(name = BUSCAR_HASH, query =
        "select o.hash from Ordem o where o.id = :id")
})
public class Ordem {
    ...
    public static final String BUSCAR_HASH = "Ordem.buscarHash";
    ...
    private int hash; //getter e setter

    @PrePersist
    @PreUpdate
    public void updateBean(){
        //atualizar o hash antes de salvar
        hash = this.hashCode();
        ultimaModificacao = new Date();
    }

    //gerado pela IDE,
    //o importante é lembrar quais propriedades usamos
    @Override
    public int hashCode() {
        int result;
        long temp;
        result = id != null ? id.hashCode() : 0;
        //IMPORTANTE! use o ordinal ou a "String" da ENUM
        result = 31 * result + (tipo != null ? tipo.ordinal() : 0);
        result = 31 * result + (codigoAtivo != null ?
            codigoAtivo.hashCode() : 0);
        temp = Double.doubleToLongBits(quantidade);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(valorAtivo);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(valorTotal);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        result = 31 * result + (ip != null ? ip.hashCode() : 0);
    }
}
```

```

        return result;
    }
}

```

Esse método `hashCode` é aquele gerado pela IDE mesmo, não precisamos nos preocupar muito com sua implementação. O que é **muito importante** no nosso exemplo é que precisamos usar as mesmas propriedades na implementação do `hashCode` da classe `OrdemTO` que fica no diretório de testes do nosso projeto. Isso porque, a partir do cliente, vamos mandar o *hash* do objeto no cabeçalho da requisição, então no servidor o resultado do *hash* tem que ser o mesmo para um objeto com os mesmos valores. Outro ponto muito importante é, na comparação do `enum`, usar seu ordinal ou sua representação como `String`, nunca a instância, pois como temos uma definição no cliente e uma no servidor, as instâncias serão sempre diferentes, e consequentemente o seu `hash()` também.

Ajustada a classe `Ordem`, vamos criar uma nova versão do método `salvar` na `OrdemBean` para preservar as duas versões no nosso projeto. Na prática, teríamos somente essa segunda versão.

```

import javax.ws.rs.core.EntityTag;
...
@Path("/v2")
@POST
public Response salvar(Ordem ordem, @Context Request request,
                        @Context HttpHeaders httpHeaders){

    if(ordem.getId() != null){

        Integer hash = buscarHashOrdem(ordem.getId());
        System.out.println(ordem);
        System.out.println("Hash ordem param: " + ordem.hashCode());
        System.out.println("Hash ordem select: " + hash);

        System.out.println(httpHeaders.getRequestHeaders());

        EntityTag etag = new EntityTag(String.valueOf(hash), true);
    }
}

```

```

Response.ResponseBuilder builder =
    request.evaluatePreconditions(etag);

    if(builder != null){
        return builder.build();
    }
    else{
        em.merge(ordem);
        return Response.noContent().build();
    }
}

em.persist(ordem);

return Response.ok(ordem).build();
}

private Integer buscarHashOrdem(Integer id) {
    try {
        return em.createNamedQuery(Ordem.BUSCAR_HASH, Integer.class)
            .setParameter("id", id).getSingleResult();

    }catch (NoResultException e){
        return null;
    }
}
...

```

O funcionamento desse método não difere muito do que fizemos o `GET` condicional, a principal diferença é que antes a checagem se o dado estava válido ou não era feita através de uma data. Agora a faremos a partir de um *hash*.

Porém, não comparamos esse *hash* diretamente, fazemos isso através da classe `EntityTag`, quem além de receber uma `String`, opcionalmente pode receber um `boolean` para indicar se é uma "tag fraca". No caso, dissemos que sim, e isso significa que não estamos considerando 100% do objeto para a composição do *hash*, e sim somente as propriedades que consideramos relevantes para nosso

negócio. Como a implementação do método `hashCode` precisa ser igual no servidor e no cliente, nas duas pontas precisamos usar `EntityTag` com o mesmo valor para a propriedade `weak`. Caso contrário, a checagem vai falhar.

Agora que sabemos que a validação é feita através do hash, podemos analisar o método como um todo. A lógica é que, se nosso objeto é novo, persistimos e devolvemos o seu conteúdo na resposta através de `Response.ok(ordem).build()`, que gerado um código HTTP `200`. Como é um objeto novo, no cliente não teremos o seu `id`, então também usamos o `ok(ordem)` porque ele devolve o conteúdo do objeto `ordem` na resposta. Como nosso serviço devolve JSON, será devolvida a representação JSON da nossa `ordem` recém-persistida.

Agora se o objeto já existe e vai ser atualizado, então precisamos analisar o seu conteúdo. Buscamos no banco de dados o *hash* do objeto persistido e comparamos com o *hash* do objeto passado como parâmetro. Se o método `evaluatePreconditions(EntityTag)` devolver um valor diferente de nulo, significa que os *hashes* dos objetos são iguais, logo podemos considerar que nada foi mudado, e então simplesmente devolvemos a resposta com `builder.build()`. Isso gera uma resposta HTTP com código `412`, que significa que a validação não passou.

Mas como assim? Parece não fazer muito sentido essa resposta, então mesmo antes de vermos o código do cliente, vamos entender o diálogo entre esse cliente e o servidor. Quando o cliente envia a requisição, ele envia o `EntityTag` em uma entrada de cabeçalho chamada `If-None-Match`. Ou seja, a requisição de *update* só deve ser processada se o *hash* do objeto passado como parâmetro não combinar com o *hash* do objeto que já existe no servidor. Como nesse primeiro caso eles combinam, o servidor não executa nada e responde que a **condição** `If-None-Match` (de os *hashes* serem diferentes) não foi satisfeita. Por isso, a resposta `412`.

Já se o resultado do `evaluatePreconditions(EntityTag)` for nulo, significa que os *hashes* são diferentes, logo a condição de só processar se forem diferentes passou, e com isso o `POST` será processado, resultando em um `em.merge(contrato)`. A resposta nesse caso então geralmente é um `Response.noContent().build()`, que devolve um código HTTP `204`, que significa que a requisição foi executada com sucesso (assim como o conhecido código `200`), mas nenhum conteúdo foi retornado pelo servidor.

Nesse caso, não precisamos devolver nada porque o cliente enviou toda a informação que foi inserida ao servidor, então em tese ele já tem a última versão completa com ele. Mas caso queira que o cliente receba, por exemplo, a data da alteração que foi gerada no servidor ou outra informação qualquer, podemos fazer uma resposta igual à que usamos logo após o `em.persist(contrato)`, usando `Response.ok(ordem).build()`.

Construindo o cliente do POST condicional

Com tudo isso, já construímos e entendemos bem o código do servidor, e até parte do código do cliente. Vamos criar então mais dois testes na `OrdemBeanTest`:

```
public void testarPostCondicional(boolean alterarOrdem){
    Client client = ClientBuilder.newClient();
    client.register(new BrowserCacheFeature());

    WebTarget javacred = client.target("http://localhost:8080");

    WebTarget ordemBean = javacred.path("ordem");

    Response response = ordemBean.path("{id}")
        .resolveTemplate("id", 1)
        .request(MediaType.APPLICATION_JSON)
        .get();

    //aqui temos a ordem com os dados que vieram do servidor
    OrdemTO ordemTO = response.readEntity(OrdemTO.class);
```

```
...  
  
}
```

Nessa primeira parte do código, recuperamos a `OrdemTO` com `id = 1`. Até aqui nada de novo, mas como não sabemos a ordem em que esse teste será executado, pode ser que ainda não tenhamos nenhuma `Ordem` no banco de dados, por isso o código continua da seguinte maneira:

```
public void testarPostCondicional(boolean alterarOrdem){  
  
    ...  
  
    boolean objetoNovo = false;  
    if(ordemTO == null){  
        //se objeto não existe vamos criar  
        ordemTO = new OrdemTO(OrdemTO.Tipo.COMPRA, "PETR4",  
                                100, 14.3, "test");  
        objetoNovo = true;  
    }  
    else if(alterarOrdem){  
        //se teste pedir, vamos alterá-lo  
        ordemTO.setQuantidade(ordemTO.getQuantidade() + 100);  
    }  
  
    //vamos criar a tag lembrando de  
    // também definir como weak = true  
    EntityTag etag = new EntityTag(  
        String.valueOf(ordemTO.hashCode()), true);  
  
    //enviamos o POST condicional  
    response = ordemBean.path("/v2").request()  
        .header(HttpHeaders.IF_NONE_MATCH, etag)  
        .post(Entity.json(ordemTO));  
  
    System.out.println(response.getStatus());  
    System.out.println(response.getHeaders());  
    System.out.println(response);  
}
```

```

        if(objetoNovo){
            Assert.assertEquals(200, response.getStatus());
        }
        else if(alterarOrdem){
            Assert.assertEquals(204, response.getStatus());
        }
        else{
            Assert.assertEquals(412, response.getStatus());
        }

        response.close();
    }
}

```

Continuando, depois de vermos se o objeto é novo ou não, checamos o parâmetro para ver se é ou não para fazermos uma alteração, pois isso vai influenciar no resultado do nosso teste.

Depois disso é que vem o código mais importante, que é a montagem do `EntityTag` com `weak=true` e passá-lo como cabeçalho com a condição `IF_NONE_MATCH` nesse trecho:

```
header(HttpHeaders.IF_NONE_MATCH, etag) .
```

E agora os métodos de teste propriamente ditos:

```

@Test
public void testarPostCondicionalComAlteracao(){
    testarPostCondicional(true);
}

```

```

@Test
public void testarPostCondicionalSemAlteracao(){
    testarPostCondicional(false);
}

```

Ao executar o primeiro teste, caso ainda não houvesse no banco de dados uma `Ordem` com `id = 1`, teríamos uma inserção e o código HTTP da resposta seria `200`. Passando desse caso em que `objetoNovo` é `true`, temos o teste em que a mudança deve acontecer, como o `testarPostCondicionalComAlteracao()` e em que a

mudança não deve acontecer, como o

`testarPostCondicionalSemAlteracao()`. No primeiro caso, o código HTTP da resposta será `204`, e no segundo caso será `412`. De certa forma, nosso teste serve como uma documentação do funcionamento do `POST` condicional.

Para finalizar, perceba a impressão do cabeçalho da requisição no console do cliente e do servidor. Especialmente o valor `If-None-Match=W/"194491767"`. O número aí é só o *hash* do objeto que foi enviado no meu exemplo, então vai variar de acordo com os valores das propriedades do contrato. O mais importante observarmos é o nome do cabeçalho, que já tínhamos visto antes, e que seu valor inicia com `w/`, significando que estamos usando uma `EntityTag weak`. Se esquecermos de colocar essa informação no cliente ou no servidor, também não dará certo.

2.8 Como o desenvolvimento de microsserviços se relaciona com o restante do livro

O desenvolvimento de microsserviços está muito mais relacionado com a forma como projetamos a distribuição de nossas soluções do que com a forma como escrevemos código dentro de cada serviço.

Podemos continuar usando projeto criados com Thorntail para não precisarmos baixar um servidor, por exemplo. No entanto, tudo que formos fazer dentro do serviço em si, que é o nosso código, será igual ao que fazemos em uma aplicação padrão. Então, se estamos desenvolvendo um microsserviço e precisamos saber mais sobre como funcionam as transações gerenciadas automaticamente, o conteúdo do **capítulo 7. Gerenciando transações com EJBs** vai servir da mesma forma, por mais que o exemplo de lá seja de uma aplicação "tradicional".

O mesmo se aplica aos demais capítulos. Para saber mais sobre como construir serviços REST podemos ir ao **capítulo 10. Criando WebServices com JAX-RS**. E lá veremos formas de clientes que não vimos aqui, como consumir um serviço via JavaScript ou criar um cliente reativo, algo introduzido no Java EE 8.

Espero que você não pare por aqui, que aproveite a leitura e vá se aprofundando no conhecimento conforme avance pelos capítulos. A partir do próximo capítulo iniciaremos um novo projeto com Wildfly e passaremos a explorar muito mais do que apenas criar um recurso REST. *So... enjoy it!*

Para fechar o capítulo

Neste capítulo nos aprofundamos mais em como podemos desenvolver WebServices REST com JAX-RS. Como foi dito mais de uma vez, essa primeira parte, que é composta pelos dois primeiros capítulos, não esgota as formas de se criar um *back-end* Java com serviços REST, mas é um ponto de partida rápido para você produzir algo mesmo antes de ver tudo o conteúdo deste livro.

Conhecendo de forma mais profunda

Como tivemos uma primeira parte focada em entregar o conteúdo de forma mais rápida, focamos mais no "como" do que no "porquê", agora veremos o conteúdo de uma forma mais completa.

Claro que a ideia continua sendo ver conteúdo prático e útil, mas o que diferencia alguém que sabe um conteúdo de alguém que sabe **bem** o mesmo conteúdo é que o primeiro muitas vezes responde "faz assim que funciona", e o segundo sabe não só o como, mas o porquê de cada coisa, e sabe discorrer de uma forma mais completa sobre o tema. Nosso foco é nos tornarmos esse segundo tipo de desenvolvedor.

Além de passar os conteúdos, o intuito dessa parte também é ser uma referência para futuras consultas. Afinal quem nunca se deparou com uma situação onde você sabe o que precisa fazer, entende os conceitos, mas esqueceu a sintaxe? Da próxima vez que isso acontecer consulte o índice do livro e divirta-se.

CAPÍTULO 3

Sendo reapresentados aos EJBs

Como comentado na abertura dessa segunda parte do livro, a partir deste capítulo vamos entrar mais nos "porquês", enquanto antes nos focamos mais no "como". Então, principalmente nos primeiros capítulos dessa parte, teremos um pouco mais de texto; mas prometo não encher linguiça. De toda forma, caso queira ir direto para o código, siga para seção 3.4.

3.1 Por que usar EJB hoje em dia?

Neste livro estamos estudando a última versão da especificação EJB, que é a 3.2. Os EJBs, Enterprise JavaBeans, são uma parte crucial do Java EE. De forma resumida, são classes Java simples, sendo que cada método seu tem controle automático de transações provido pelo servidor de aplicações. Às vezes nem percebemos, mas lidar com transações é algo que precisamos fazer toda vez que usamos um banco de dados, e praticamente qualquer aplicação que formos fazer precisa acessar banco de dados.

A uma primeira vista, pode parecer que o controle automático de transações é só não ter que dar *begin* e *commit* no meio do nosso código, o que poderíamos fazer com um interceptador, e já eliminaríamos a possível complexidade extra em se usar EJBs. Mas algo em que não pensamos quando falamos de transações é quando fazemos integrações, e precisamos garantir a atomicidade de uma operação que envolve mais de um sistema. A integração em si usando serviços REST é bem simples, mas e a atomicidade? Bom, isso nos EJBs vem de fábrica, o que por si só já responderia à pergunta título desta seção.

E outra vantagem que podemos citar, além do controle de transação, é o fato de ser uma especificação, e não um framework; logo, se não estivermos felizes com a quantidade de bugs ou performance de uma implementação, podemos mudar para outra sem impactar, ou impactando muito pouco no nosso código.

Spring e DeltaSpike como alternativas aos EJBs

No mundo Java a mais famosa alternativa aos EJBs é o Spring, que é um framework que oferece as mesmas funcionalidades do Java EE, só que à sua maneira, e dentre essas funcionalidades uma das principais é o controle de transações. Podemos também controlar as transações via interceptadores do CDI, ou usar alguma biblioteca como a DeltaSpike da Apache (<http://deltaspike.apache.org>), em especial o módulo JPA que implementa esses interceptadores para nós.

Na prática, seja com EJBs, Spring ou usando DeltaSpike, pouca coisa mudará no nosso código, especialmente se for uma aplicação mais simples. O que muda basicamente é que para executarmos EJBs precisamos de um tipo de servidor mais pesado que o necessário para executar Spring ou DeltaSpike, o que era um problema até alguns anos atrás quando os servidores de aplicação eram um monólito que subia diversos serviços mesmo quando não precisávamos deles.

MicroProfiles e Thorntail

Como acabamos de ver nos capítulos anteriores, hoje podemos subir o Thorntail, uma versão do Wildfly bastante enxuta, que consegue competir em pé de igualdade com alternativas como o Spring Boot e qualquer outra solução como uma aplicação com DeltaSpike no Tomcat.

E não é só o Thorntail/Wildfly que roda dessa forma mais leve. Na verdade, depois que o Java EE criou o conceito de *profiles*, com o qual seria possível a criação de um servidor com um subconjunto das diversas funcionalidades do Java EE, a coisa começou a melhorar muito. E o resultado principal para nós foi a criação da iniciativa MicroProfile (<https://microprofile.io>), que definiu um subconjunto mínimo com as funcionalidades mais comuns hoje em dia. Com isso, acaba aquilo do Java EE ser uma bazuca para muitas vezes matar uma formiga. Então, além das opções já comentadas, temos ainda o Open Liberty (<https://openliberty.io>), o Payara (<https://www.payara.fish>) e o TomEE (<http://tomee.apache.org>).

Voltando à pergunta inicial

Então voltando à pergunta desta seção, se precisamos trabalhar com banco de dados, teremos que trabalhar com transações, e isso um EJB faz de forma nativa com uma codificação tão simples como colocar uma anotação em uma classe. Não é papo de pescador, fizemos isso no capítulo anterior e nem percebemos que estávamos

usando um EJB. Além disso, quando temos situações mais complexas, o EJB está mais bem preparado do que soluções como a do DeltaSpike por exemplo, cuja solução começa a complicar junto com a complexidade do nosso problema. Afinal, o EJB foi criado originalmente para tratar de coisas complexas, e veio sendo simplificado para as tarefas do dia a dia.

E por último, podemos citar o fato de um EJB ser altamente monitorável sem precisarmos codificar nada a mais para isso, permitindo-nos coletar estatísticas de diversas maneiras, assim como coletamos estatísticas do Hibernate. Por exemplo, podemos contar as execuções, o tempo de execução etc.

3.2 O que vamos usar para criar nossa aplicação?

No capítulo passado criamos uma aplicação, basicamente o que chamamos de um microsserviço, que rodava com o servidor embarcado. Agora vamos desenvolver usando a abordagem "tradicional", que é criar uma aplicação e colocá-la para rodar em um servidor.

Usaremos o Wildfly 19, mas você pode usar versões mais novas caso estejam disponíveis, só procure saber se existe algum ajuste necessário para uma aplicação feita na versão em que estamos rodar na nova versão.

Além do servidor, utilizaremos uma IDE, que dentre as opções de mercado, temos como mais populares o Eclipse, o IntelliJ IDEA e o Netbeans. Todos podem ser utilizados e farão muito bem o serviço, até pelo fato de estarmos utilizando Maven, que nos deixa mais independentes da IDE.

Caso você ainda não tenha sua IDE preferida, dê uma olhada no Netbeans, pois entre as opções gratuitas, ele é o que consegue trabalhar melhor com EJB, JPA, JSF etc. sem a necessidade de plugins adicionais. Mas apesar de não precisar de plugins para essas tecnologias, na última versão disponível enquanto escrevo (11.3), é preciso adicionar manualmente o plugin para o WildFly. Em versões anteriores (10.x) isso era mais automático, e pode voltar a ser. Então veja como isso está na versão que você está usando.

A diferença básica entre as IDEs estará no caminho para criar um determinado recurso, mas para minimizar as diferenças, e também para exercitarmos e aprendermos como as coisas funcionam, utilizaremos o mínimo possível de *wizards*, pois esses, sim, são bem diferentes entre as IDEs.

3.3 Fazer o código na mão, utilizar Wizards, ou o JBoss Forge?

A maior diferença entre as IDEs está no trabalho que cada uma se propõe a fazer pelo desenvolvedor. Esse trabalho feito por nós é bom, pois economiza nosso tempo, porém só é interessante utilizar esse recurso se soubermos o que ele faz.

Não acrescenta muito ao conhecimento de um desenvolvedor saber clicar com o botão direito e gerar uma aplicação completa, com dezenas de telas de cadastro funcionando, se ele não tiver a menor ideia de como as peças se encaixam para formar aquela aplicação.

Depois de aprender como os EJBs funcionam, poderemos utilizar os wizards que as IDEs nos oferecem sem qualquer problema. Mas veremos que criar um EJB costumam ser tão simples, que será até difícil encontrarmos situações onde o auxílio de um *wizard* será realmente necessário.

Agora, se você busca algo como os scripts de geração de recursos do *Rails* ou *Grails*, porém voltado para o Java EE, melhor que usar o *wizard* da IDE pode ser usar o **JBoss Forge** (<https://forge.jboss.org>). Ele possui scripts semelhantes aos citados, inclusive com geração de *scaffold* usando as tecnologias do Java EE.

Possui ainda diversos plugins que acrescentam mais funcionalidades e ainda pode ser instalado nas principais IDEs de mercado. Por mais que você acabe usando a assistência da IDE para fazer esse tipo de tarefa, os assistentes ou *wizards* serão os mesmos em qualquer IDE que você utilizar, e o resultado também será o mesmo se rodar os scripts por linha de comando.

Pode não parecer muito, mas assim como o Maven, isso deixa nosso desenvolvimento mais independente de IDE, permitindo que cada pessoa em um mesmo projeto use sua IDE preferida sem maiores problemas.

3.4 Baixando o servidor e criando o projeto

Para iniciar nosso projeto, vamos baixar a última versão do Wildfly e descompactar em um diretório da nossa escolha. O download pode ser feito na página <http://wildfly.org/downloads>. Depois de descompactado, vamos à nossa IDE iniciar nosso projeto.

Agora vamos criar nosso projeto, que será o sistema de uma financeira. Nosso objetivo inicial será fazer simulações de financiamento, mas no decorrer do livro vamos adicionando novas funcionalidades.

Para começar, basta clicar com o botão direito ou ir até `File > New Project` e escolher a opção `Java with Maven > Web Application`, como vemos na imagem a seguir.

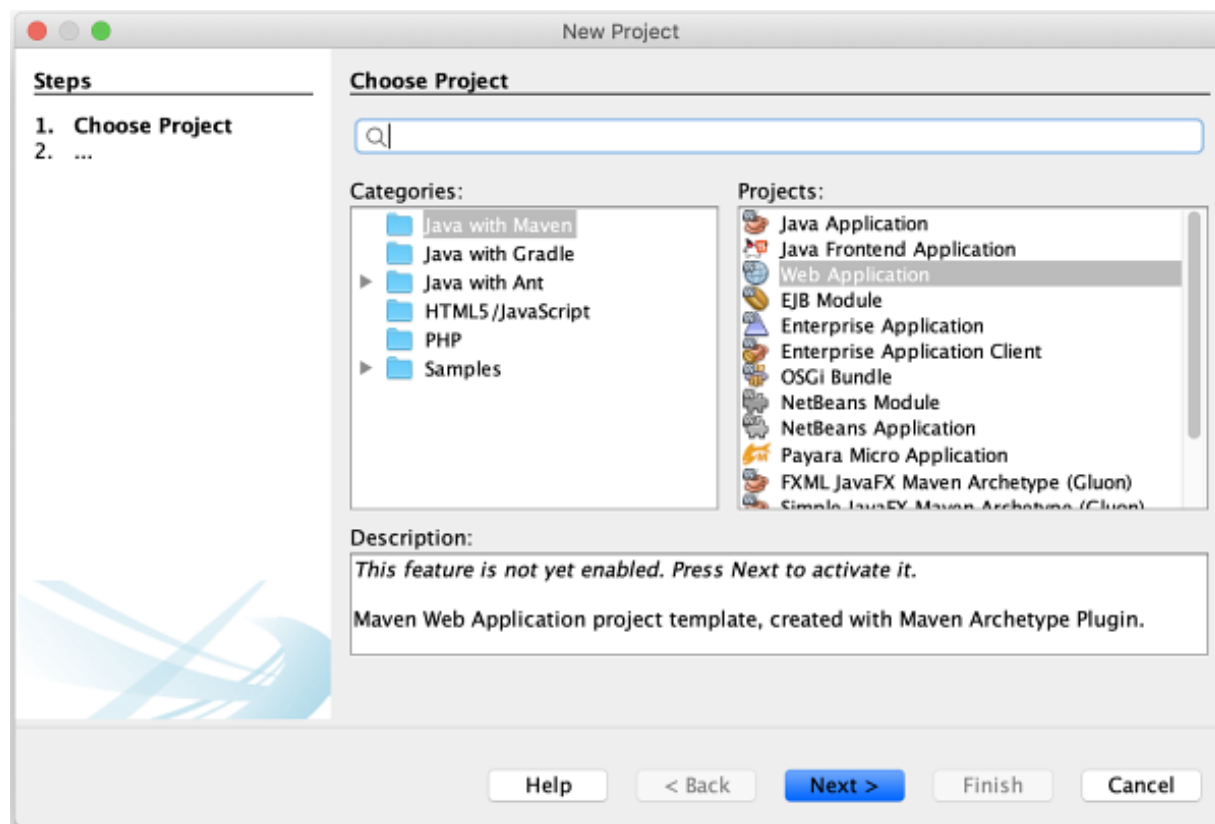


Figura 3.1: Criando um projeto Web com Maven 1/2

Como nossa financeira é basicamente uma instituição de crédito, chamaremos nossa aplicação de "javacred". Esse mesmo nome já será copiado para o campo `Artifact Id`, e precisaremos especificar apenas o `Group Id`, que costuma ser o pacote básico da nossa empresa, por exemplo. No livro usaremos `br.com.casadocodigo`. A versão poderemos deixar a que foi sugerida, já que estamos em uma versão bem embrionária. E, por fim, o pacote da aplicação costuma ser a junção do `Group Id`, que é a identificação da nossa empresa, com o `Artifact Id`, a identificação da nossa aplicação. O resultado será algo como o da imagem a seguir.

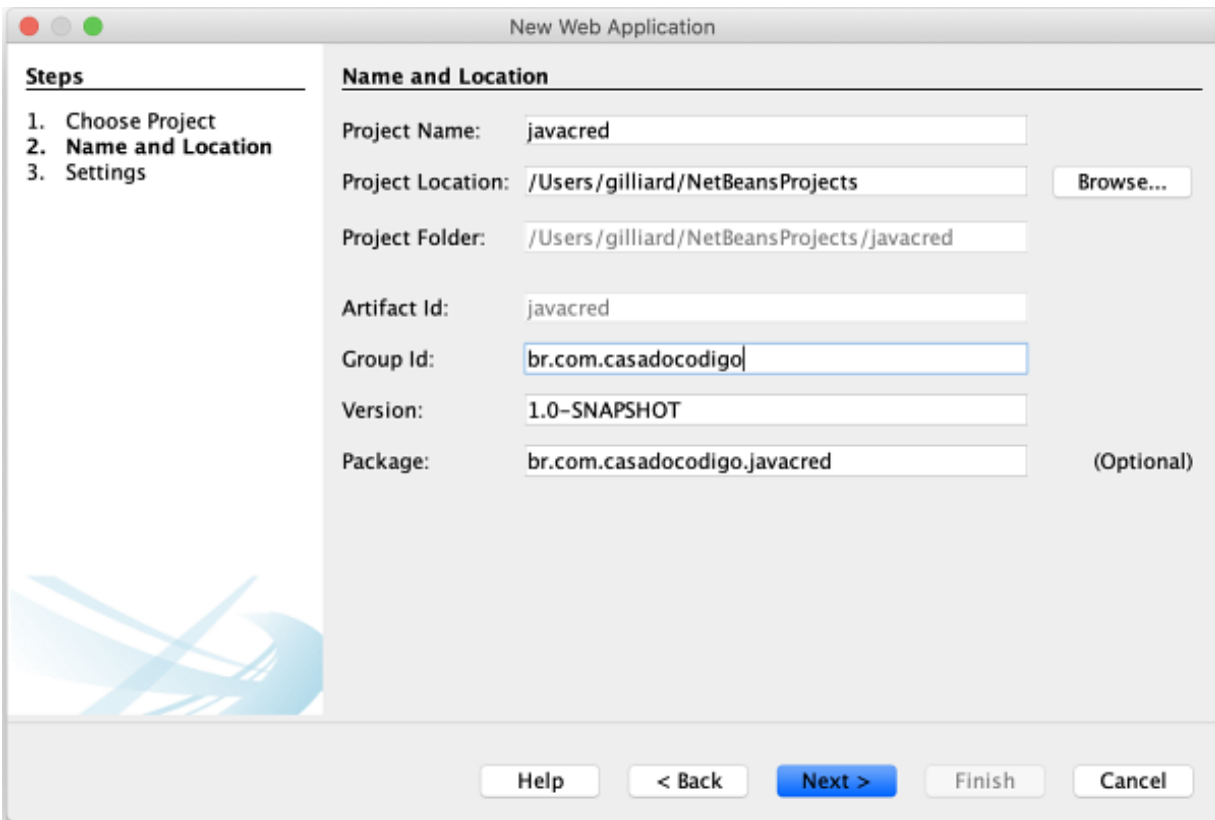


Figura 3.2: Criando um projeto Web com Maven 2/2

Agora usaremos a opção **Próximo** e iremos para a configuração do servidor. Quando a tela seguinte for apresentada, clique em **Adicionar**.

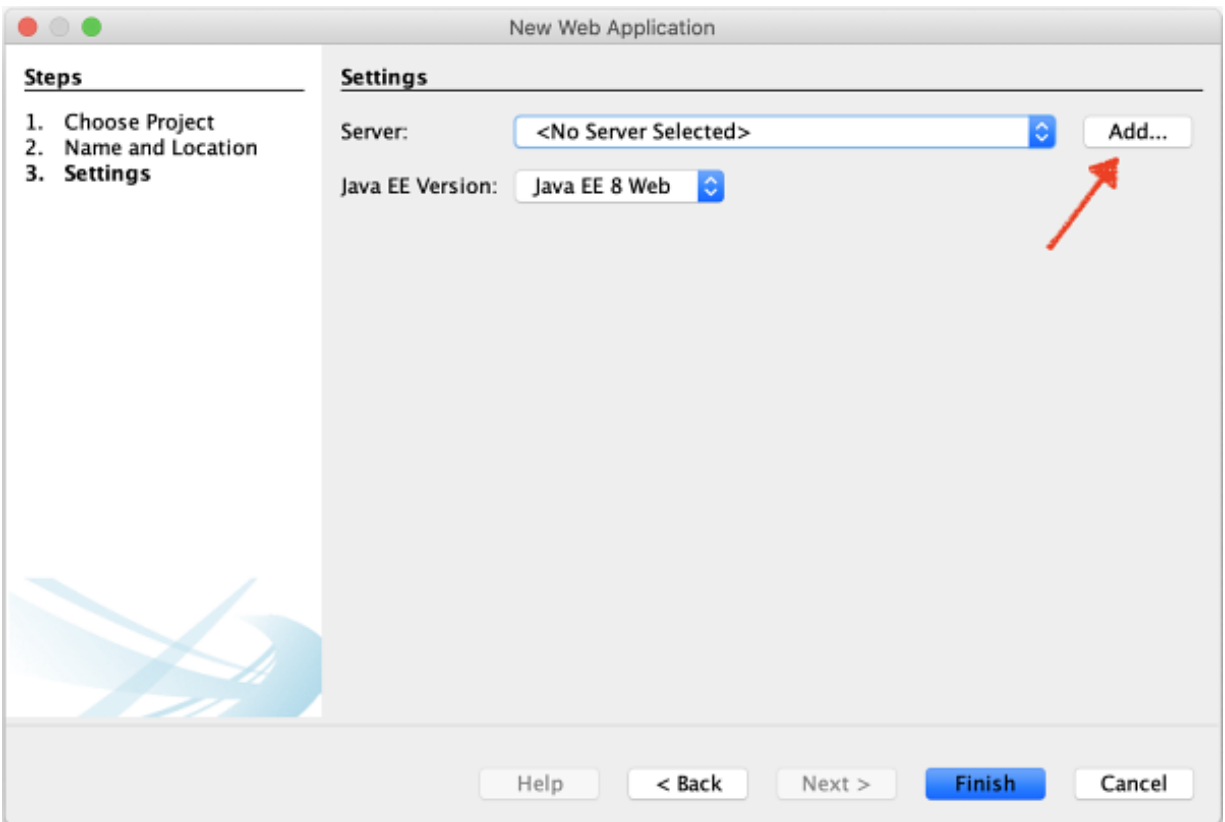


Figura 3.3: Configurando o servidor da aplicação 1/4

Na tela seguinte, escolha o servidor Wildfly. Lembrando que no Netbeans 12 o plugin tem que ser adicionado manualmente. Baixe-o pelo link a seguir (caso não encontre versão mais nova):

<http://plugins.netbeans.org/plugin/76472/wildfly-application-server>.

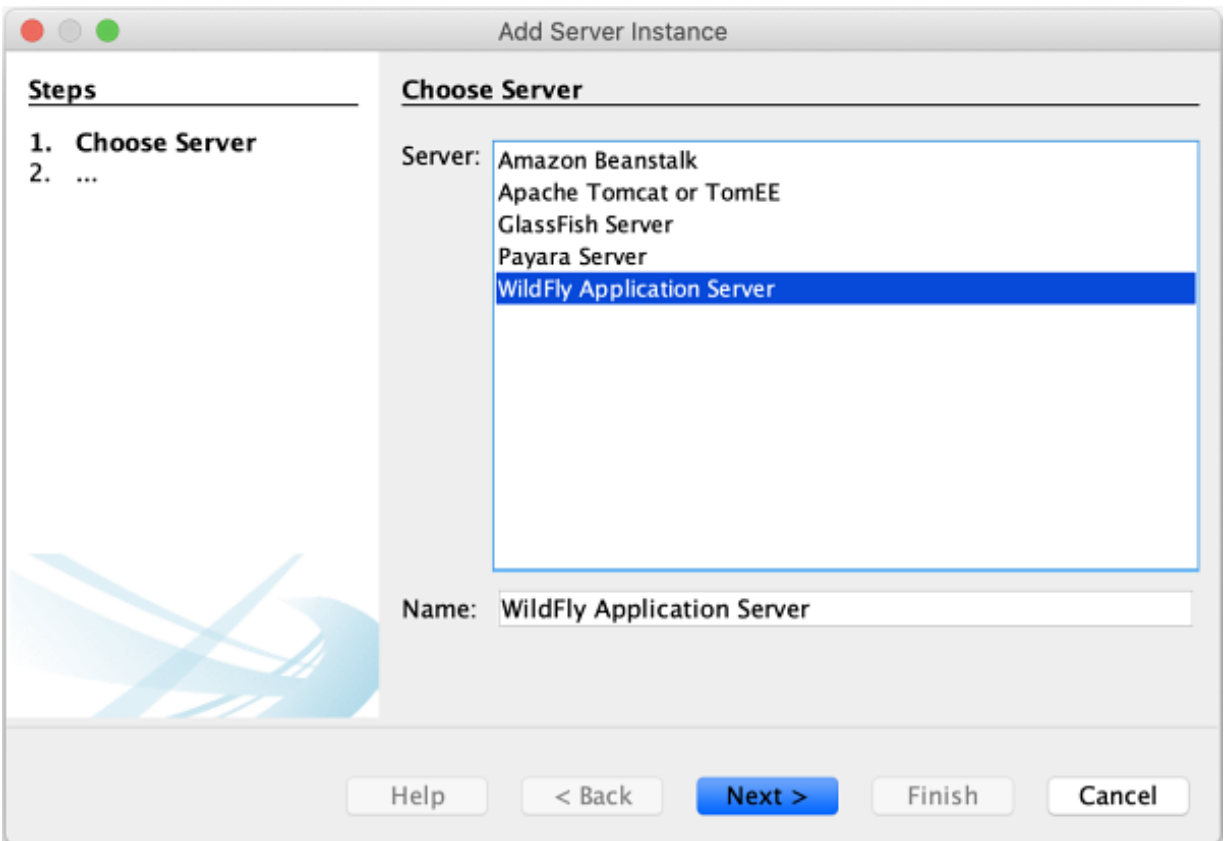


Figura 3.4: Configurando o servidor da aplicação 2/4

Na tela que será mostrada em seguida, indicaremos o diretório onde descompactamos nosso wildfly no primeiro campo. Em seguida podemos ir direto para a opção Finalizar .

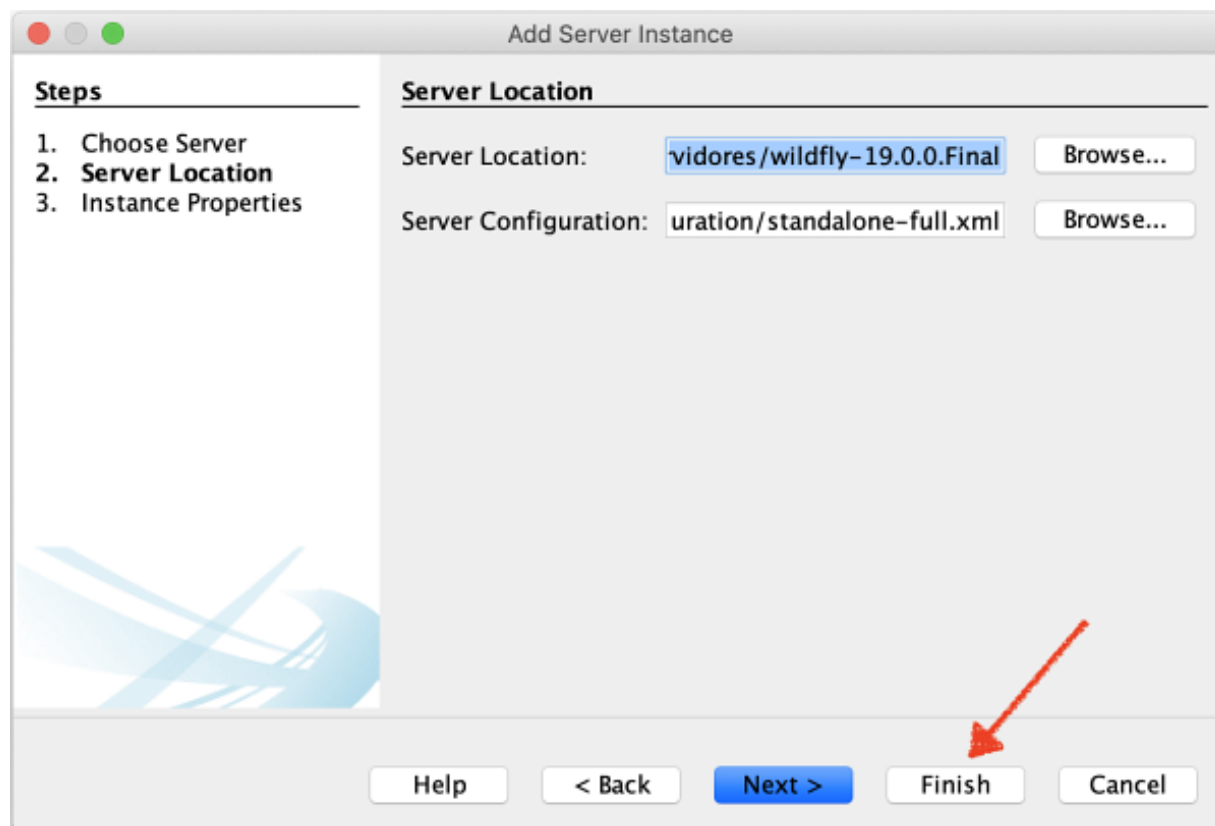


Figura 3.5: Configurando o servidor da aplicação 3/4

Agora que o servidor está pronto, podemos selecioná-lo na lista de servidores e finalizar a criação do nosso projeto.

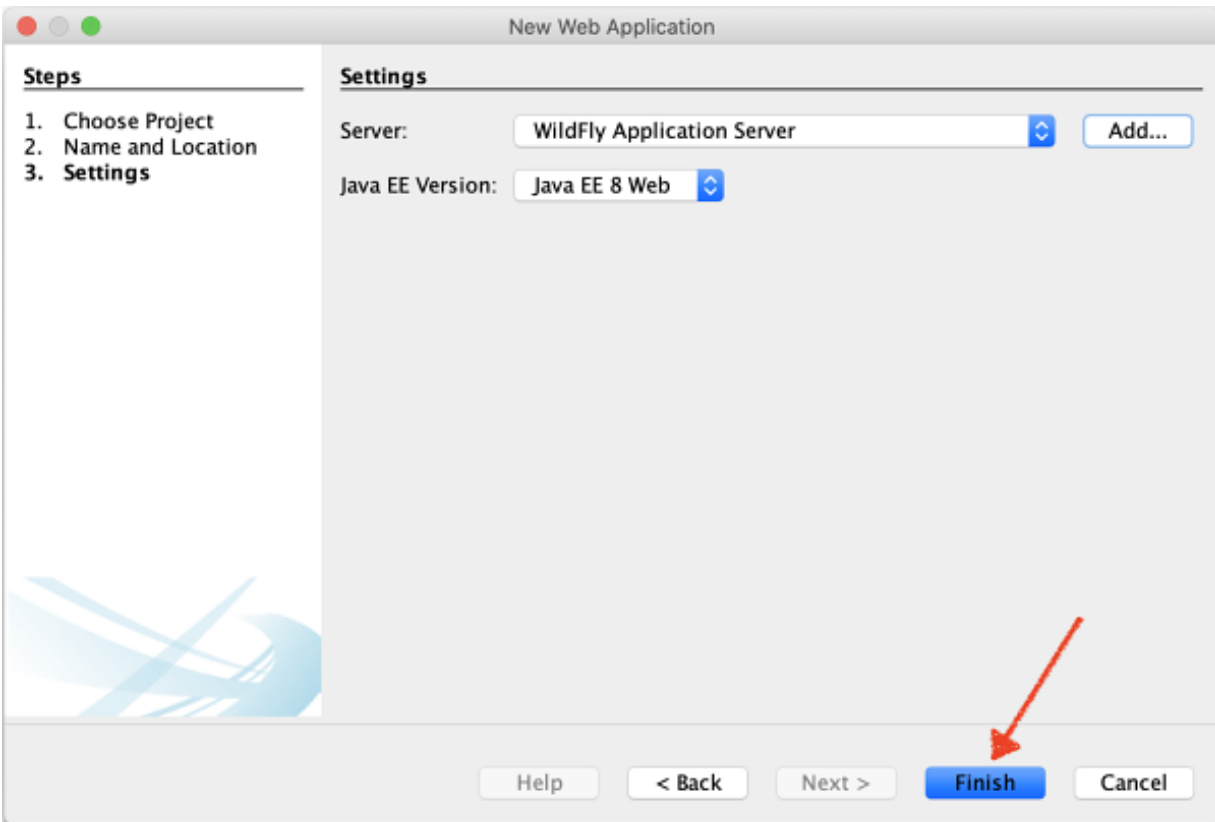


Figura 3.6: Configurando o servidor da aplicação 4/4

Após a criação do projeto, a IDE gera um arquivo `pom.xml` que fica na raiz do nosso projeto. Porém, cada IDE apresenta o projeto de uma forma diferente: por exemplo, o Netbeans mostra os arquivos da raiz do projeto dentro de uma pasta (que não existe no sistema de arquivos) chamada `Arquivos do Projeto`. Independentemente de onde esteja, vamos editar nosso `pom.xml` deixando somente com o necessário.

Para ficar mais fácil, você pode copiar o conteúdo do arquivo do repositório do projeto no GitHub:

<https://github.com/gscordeiro/microservicos-ejbs/javacred/>.

O conteúdo do arquivo deve ficar assim:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>br.com.casadocodigo</groupId>
<artifactId>javacred</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<dependencies>

    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>8.0</version>
        <scope>provided</scope>
    </dependency>

</dependencies>

<build>
    <finalName>${artifactId}</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>

```

Como vamos usar o Java EE 8, deixamos somente a pendência Maven referente a ele. Como o Java EE é uma coleção de outras especificações, apenas com essa configuração já teremos

disponíveis CDI (injeção de dependências), JPA (persistência), JSF (framework MVC), EJB e por aí vai. Além disso, deixamos o `maven-compiler-plugin` e o ajustamos para usar Java 8.

Esse passo de nivelamento e limpeza é importante para termos certeza que independente da IDE utilizada o resultado será o mesmo.

Vamos agora criar uma classe Java simples. A forma mais fácil é clicar com o botão direito em cima do projeto e escolher `New > Java Class`. No pacote, podemos especificar `br.com.casadocodigo.javacred.ejb` e no nome da classe colocar `CalculadoraFinanciamento`.

Sim, é basicamente o mesmo exemplo que vimos no capítulo anterior, pelo menos no início, pois no nosso exemplo é o mais parecido que temos com o *Hello World*.

Nossa classe vai receber como parâmetros um valor e uma quantidade de meses, e como resultado vai devolver o valor de cada parcela desse possível financiamento.

```
package br.com.casadocodigo.javacred.ejb;  
  
public class CalculadoraFinanciamento {  
  
    public double simulaFinanciamento(double valorEmprestimo,  
    int meses){  
        double taxaJuros = 0.01; // 1% ao mês  
        double totalJuros = meses * taxaJuros; // juros simples  
        double valorDivida = valorEmprestimo * (1 + totalJuros);  
        return valorDivida / meses;  
    }  
}
```

Agora temos que testar nosso código. Nesse momento ainda não se trata de teste automatizado, pois isso é matéria para o capítulo 9. **Testando nossos serviços.** Por enquanto, usaremos teste como

sinônimo de colocar o código para executar. Nesse caso, executaremos nosso código via navegador.

A IMPORTÂNCIA DOS TESTES AUTOMATIZADOS

Testes automatizados são importantes, use-os sempre que possível. E se você achar que não é possível, procure um pouco mais, talvez não esteja usando a melhor ferramenta para o seu caso. Estamos aqui abrindo mão de fazê-los nesse momento apenas por uma questão de didática, uma vez que os testes precisam de uma explicação própria; e serão vistos no capítulo 9. Porém, no dia a dia do desenvolvimento, os testes serão feitos juntamente com nosso código (antes, durante, depois...).

Para que possamos chamar nossa classe pelo navegador, vamos precisar de uma *servlet*, que é uma das formas mais simples (e antigas) de executar algo na web utilizando Java. Então vamos criar uma nova classe Java com o seguinte conteúdo.

```
package br.com.casadocodigo.javacred.servlet;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;

@WebServlet("/calculadora-financiamento")
public class CalculadoraFinanciamentoServlet extends
HttpServlet {

}
```

Aqui simplesmente declaramos uma servlet e, por meio da anotação `@WebServlet`, já definimos o seu mapeamento para `/calculadora-financiamento`.

Utilize bastante o completador de código da sua IDE. Assim, em vez de digitar todo o `import` e também o `@WebServlet`, basta você digitar, por exemplo, `@WeS`, que o Netbeans ou o Eclipse vai buscar por

anotações cujo nome combine com esse padrão, e vai encontrar a que queremos. Ao combinarmos maiúsculas e minúsculas, é como estivéssemos buscando assim: `@We*S*`. Se houver mais de uma opção, selecione a que usamos aqui, que o `import` será colocado automaticamente e o nome da anotação será completado.

Além de autocompletarmos nossa anotação, faremos o mesmo com o método `doGet` que utilizaremos. Uma servlet é uma classe que responde aos métodos HTTP, e como queremos simplesmente digitar no browser `http://localhost:8080/javacred/calculadora-financiamento`, dar um `enter` e já testar nosso código, precisamos do método `GET` para isso.

Então, coloque o cursor dentro da classe

`CalculadoraFinanciamentoServlet`, escreva por exemplo `"doG"` e aperte `control` (command no Mac) + espaço. Depois disso o método deve aparecer com a assinatura correta, e os `imports` necessários serão acrescentados. Para testar nosso código, faltará somente chamar nossa calculadora ali dentro.

```
...
protected void doGet(HttpServletRequest req,
HttpServletRequest resp) throws ServletException, IOException {

    CalculadoraFinanciamento calculadora =
        new CalculadoraFinanciamento();
    double parcela =
        calculadora.simulaFinanciamento(10_000.0, 10);

    resp.getWriter().printf("valor da parcela %.2f", parcela);
}
...
```

Para executar, basta clicar com o botão direito em cima do nosso projeto e selecionar `Run`. Automaticamente será aberto o browser na URL raiz do nosso projeto. Basta informar a URL que mostramos anteriormente e veremos o resultado, uma parcela de 1.100 (mil e cem).

Vimos que o resultado corresponde às expectativas, mas até agora não passa de uma classe Java simples. Porém, como o objetivo do nosso estudo é aprender a usar os EJBs, temos que transformar essa classe em um EJB.

Com isso, nossa classe ganhará "superpoderes", como controle automático de transações, estatísticas de utilização, e até acesso pela rede. Essa última característica é liberada com o uso de EJBs remotos, que serão vistos na seção **5.1. EJBs Remotos**.

3.5 Criando um EJB simples

Depois de um pouco de explicação, vamos transformar nossa classe Java simples, `CalculadoraFinanciamento`, em um EJB. A mudança é extremamente simples, basta colocarmos a anotação `@Stateless` na classe `CalculadoraFinanciamento`, usando sempre o completador de código para não nos esquecermos do `import`.

```
package br.com.casadocodigo.javacred.ejbs;
```

```
import javax.ejb.Stateless;
```

```
@Stateless
public class CalculadoraFinanciamento {
    ...
}
```

Bastou essa anotação para transformar uma classe simples em um EJB. Então se você já ouviu alguém dizendo que fazer um EJB é difícil, agora sabe que não é bem assim. Essa cultura de achar que EJB é complicado vem de muitos anos atrás, anterior ao lançamento da versão 3.0, que simplificou totalmente seu uso. E olhe que essa versão é de maio de 2006, ou seja, não é de hoje que usar EJBs se tornou tarefa simples. E para melhorar, estamos estudando a versão 3.2, que já é a segunda leva de simplificações em cima da 3.0.

Agora falta alterar nossa servlet para usar o EJB, e não a classe simples. É interessante observar que, se deixarmos a servlet como está, ela continuará funcionando, pois o código do nosso EJB está bem trivial. Porém, ao usar o operador `new`, não estamos utilizando um EJB, e sim uma classe simples. Mas qual a diferença?

Como dito há pouco, um EJB possui um gerenciamento feito pelo servidor, dando a ele serviços extras que uma classe simples não tem. Nesse primeiro momento, não utilizaremos essas funcionalidades extras, até por isso o exemplo continua funcionando mesmo antes de adequarmos a servlet. Mas veremos que muito em breve esses serviços serão essenciais.

Assim como a mudança da calculadora para EJB, a mudança da servlet também é fácil. Basta deixarmos de criar a instância manualmente e pedirmos para que ela seja passada para nossa classe, ou injetada, pelo próprio servidor, através da anotação `@EJB`.

O USO DE `@EJB` VS. USO DE `@INJECT`

Nos capítulos seguintes, veremos que é sempre vantajoso utilizar `@Inject` para injetar nossos EJBs, mas como estamos bem no início do projeto e ainda nem habilitamos o suporte a CDI, vamos usar `@EJB` apenas por uma sequência didática.

...

`@EJB`

`private` CalculadoraFinanciamento calculadora;

`@Override`

`protected void doGet`(HttpServletRequest req,
HttpServletRequestResponse resp) `throws ServletException, IOException` {

`double` parcela =
calculadora.simulaFinanciamento(10000.0, 10);

resp.getWriter().printf("valor da parcela %.2f", parcela);

```
}  
...
```

Executando novamente a servlet, veremos que continua funcionando. Na maioria das IDEs, não é necessário parar o servidor a cada alteração. Ela é publicada automaticamente, bastando salvá-la. Isso faz com que a aplicação toda seja atualizada, sem precisar subir novamente o servidor, e sem ter que subir novamente outras aplicações que possam estar no mesmo servidor.

Agora se for fazer diversas implementações, pode ser interessante baixar o servidor para que ele não fique sendo atualizado em segundo plano a cada pequena alteração, algo que pode fazer com que seu ambiente fique mais lento enquanto a atualização é feita. Algo que costumo fazer é desabilitar a atualização automática e fazê-la explicitamente quando terminar de fazer minhas alterações, assim não preciso baixar o servidor enquanto desenvolvo.

Independente do método que você tenha utilizado, observe o resultado no console do servidor após subirmos nossa aplicação depois de termos transformado nossa calculadora em EJB.

Perceba a seguinte saída no console:

```
java:global/javacred/CalculadoraFinanciamento!  
  br.com.casadocodigo.javacred.ejbs.CalculadoraFinanciamento  
java:app/javacred/CalculadoraFinanciamento!  
  br.com.casadocodigo.javacred.ejbs.CalculadoraFinanciamento  
java:module/CalculadoraFinanciamento!  
  br.com.casadocodigo.javacred.ejbs.CalculadoraFinanciamento  
java:global/javacred/CalculadoraFinanciamento  
java:app/javacred/CalculadoraFinanciamento  
java:module/CalculadoraFinanciamento
```

Isso mostra que agora nosso servidor conhece "mais de perto" a nossa calculadora, e que ela está disponível dentro do servidor com esses nomes. Esses são os nomes JNDI da nossa calculadora. JNDI (*Java Naming and Directory Interface*) é o serviço de nomes

do Java. É como se fosse um serviço de DNS, que vincula endereços a objetos, como EJBs.

É nesse endereço que teremos alterações se dermos nomes diferentes ao nosso projeto ou às nossas classes, então precisamos ficar atentos. Mais à frente utilizaremos esses endereços, mas por enquanto basta observarmos que o tratamento que nosso servidor deu à nossa classe mudou.

Mas qual será o resultado dessa mudança? O que teremos de benefício fazendo isso? Bom, este é o assunto do livro como um todo, mas como já foi dito nesse capítulo, além do gerenciamento automático de transações, ganhamos também a coleta de estatísticas de cada EJB da nossa aplicação.

Para fechar o capítulo

O intuito deste capítulo foi mostrar um pouco de como criamos e acessamos um EJB. Não nos aprofundamos muito em nenhum aspecto, pois cada discussão terá seu local específico no decorrer do livro. O objetivo foi apenas mostrar que criar e acessar EJBs é mais simples do que muitos de nós imaginamos, e agora podemos começar a aprofundar mais o nosso conhecimento.

CAPÍTULO 4

Os Sessions Beans por estado

Quando aprendemos a andar de bicicleta temos primeiro aula de mecânica, engrenagens, marchas, ou simplesmente subimos em cima e tentamos sair andando? Como andar de bicicleta é um bom parâmetro de aprendizado, afinal, dizem que nunca mais esquecemos, seguimos a mesma ideia até aqui. Então em vez de começar por um capítulo como este, que apresenta todos os tipos de EJBs, simplesmente saímos usando no capítulo anterior. Agora que já demos uma voltinha, vamos entender mais das partes que compõem nosso novo brinquedo.

Para classificar os EJBs, precisamos examinar dois grandes grupos, um é o dos *Session Beans* e o outro o dos *Message-Driven Beans* ou MDBs. O primeiro grupo pode ser separado em duas dimensões, a primeira é a do estado, que diz como o EJB mantém seus dados. Nessa dimensão podemos ter beans *Stateless*, *Stateful* ou *Singleton*. A outra dimensão diz respeito à visibilidade do nosso bean, ou o tipo de interface que ele tem. Nesse caso, podemos classificar as interfaces como *No-interface*, *Local* e *Remote*.

O segundo grupo, dos MDBs, não possui subdivisões, e apesar de ser um assunto bastante interessante, não abordaremos neste livro pois é conteúdo suficiente para um livro próprio.

Neste capítulo, vamos ver a primeira dimensão dos *Session Beans*, que, como acabamos de ver, refere-se ao seu estado; e no próximo veremos os tipos de interface.

Parece muita classificação, no entanto "não entre em pânico". A ideia não é decorar nada disso, e sim entendermos como e quando usar cada opção. O fato de termos várias combinações não quer dizer que no dia a dia usamos todas com a mesma frequência. É bem possível que a maioria dos seus EJBs seja sem estado e sem

interface, mas conhecer os tipos facilita a decisão correta quando saímos do trivial.

4.1 EJB Stateless

Como o nome já nos adianta, beans *Stateless* são aqueles que não mantêm estado. Ou seja, quando usamos um *Stateless Session Bean - SLSB* não podemos contar que uma informação guardada nele estará disponível quando precisarmos dela novamente.

Muitos poderiam pensar que o melhor seria nem conseguir armazenar qualquer dado nesses beans, recebendo uma exceção quando a aplicação subisse e percebesse a presença de propriedades no escopo da classe desse tipo de EJB, mas isso não ocorre. Em vez de uma checagem em tempo de execução, temos que saber que isso não dará certo, e então por nossa própria conta não guardar nenhuma informação nesses beans.

Vamos voltar um pouco no nosso exemplo da `CalculadoraFinanciamento` que é apresentado novamente só para relembrarmos como deixamos no capítulo passado.

```
import javax.ejb.Stateless;

@Stateless
public class CalculadoraFinanciamento {
    ...
}

@WebServlet("/calculadora-financiamento")
public class CalculadoraFinanciamentoServlet extends HttpServlet{

    @EJB
    private CalculadoraFinanciamento calculadora;
    ...
}
```

Como podemos ver, nossa calculadora é um EJB Stateless, e definimos isso apenas colocando a anotação `@javax.ejb.Stateless`. E para utilizar esse bean, só precisamos injetá-lo na nossa servlet, via `@javax.ejb.EJB`. A construção disso é bem simples, e até já passamos por isso no capítulo anterior, mas a diferença está na relação do nosso bean, a calculadora, com o seu cliente, a servlet. Vamos analisar novamente o código do método `doGet`, onde o EJB é utilizado.

```
...
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException {

    double parcela = calculadora.simulaFinanciamento(10000.0, 10);

    resp.getWriter().printf("valor da parcela %.2f", parcela);
}
...
```

Quando referenciamos a variável `calculadora`, parece que estamos lidando com um objeto qualquer, mas a verdade não é bem assim.

Um bean Stateless na prática é um link entre o cliente e alguma instância da `CalculadoraFinanciamento`. Vejamos a figura a seguir.

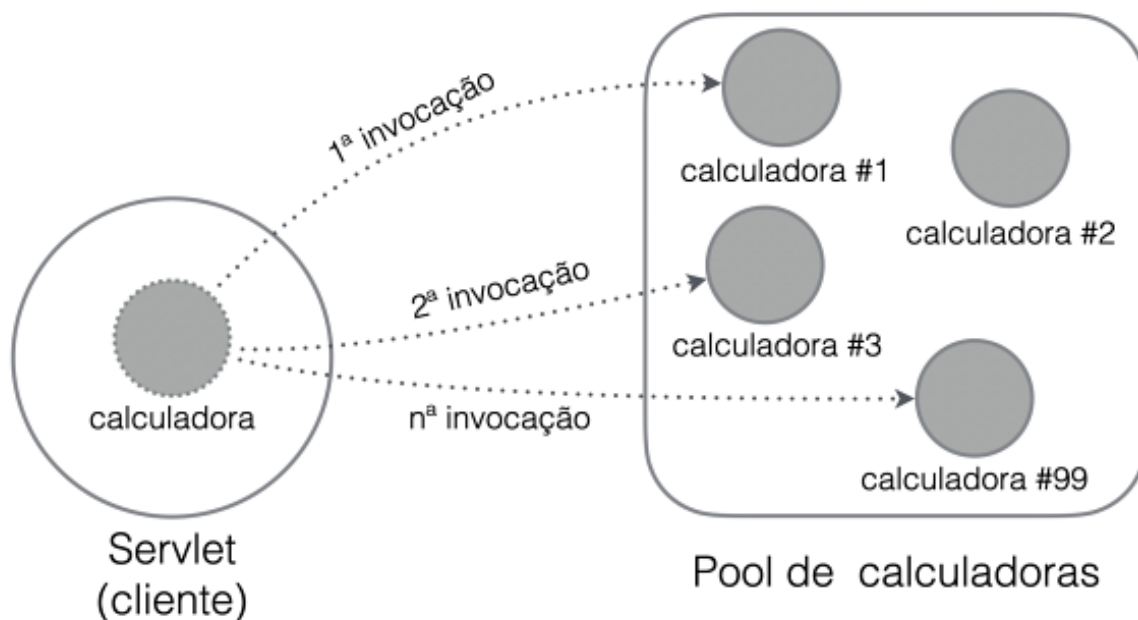


Figura 4.1: Ligação entre o cliente e o EJB Stateless

Observando a ilustração, vemos que, a cada invocação de método na nossa calculadora, uma instância diferente do nosso EJB pode responder. Isso não quer dizer que sempre será uma diferente. De fato, se fizermos um pequeno teste, podemos até ter a mesma instância respondendo algumas vezes, mas o servidor é quem escolhe, dentro do *pool*, uma disponível e a usa para processar a requisição do cliente.

Mesmo que estejamos dentro de um laço de repetição em um método do cliente, a cada iteração podemos ter uma instância diferente do nosso EJB. A disponibilização de uma instância não é feita no momento da injeção ou do `lookup`, e sim no momento do uso. Assim, a mesma instância, por voltar para o *pool* após o uso, pode ser utilizada por diferentes clientes. Em resumo, não há qualquer vínculo entre cliente e os EJBs Stateless.

Isso dá ao servidor uma incrível flexibilidade. É possível criar novas instâncias quando a demanda aumenta, e remover essas instâncias quando a demanda for baixa; é uma elasticidade natural do servidor. Algo parecido não seria possível se houvesse uma relação rígida

entre o cliente e o EJB, pois o servidor não poderia simplesmente destruir um EJB quando bem entendesse, uma vez que este poderia estar sendo utilizado por algum cliente. É uma gerência mais complexa, que veremos quando estivermos falando de EJBs Stateful, na seção a seguir.

Alguns assuntos como o tratamento de exceções ficarão melhor tratados em capítulos posteriores. Nos capítulos 7 e 8 veremos sobre transações, e isso tem relação direta com o tratamento de exceções. E para esse tema fluir bem, precisaremos primeiro ver um pouco de JPA, como lidamos com o `EntityManager` dentro de um EJB, algo que será visto no capítulo 6. Além disso, a grande maioria dessas questões se aplica a todos os tipos de EJBs que estamos estudando, mais um motivo para não vermos isso agora, pois nesse momento nos interessam mais suas diferenças.

4.2 EJB Stateful

Já os EJBs Stateful possuem os mesmos atributos dos Stateless, como controle de transação (que é o que usamos mais), porém ele não é um ponteiro para um *pool*, e sim uma instância que fica dentro do cliente, assim como todo objeto java que usamos no dia a dia. A principal diferença de um objeto java comum é que, pelo fato de o EJB Stateful poder rodar em outra máquina (se for remoto), não tem como o *garbage collector* removê-lo de forma automática em todas as situações. Mas teremos tópico específicos para tratar da remoção desses beans. Além disso, veremos na seção 6.5 veremos que esse tipo de bean é uma forma muito elegante de tratarmos a `LazyInitializationException`.

Agora vamos pensar em um exemplo onde nosso bean Stateful seria útil. Sabemos que as financeiras são consultadas para simulações de financiamento, dentre outros serviços. É o que ocorre quando uma concessionária de carros sai consultando diversas

opções para encontrar a melhor condição de pagamento. Aqui no nosso exemplo disponibilizaremos uma interface (não gráfica) de simulações de quitação de dívida, algo que indiretamente é usado nos processos de portabilidade de financiamento entre bancos e financeiras, pois na prática a dívida antiga é quitada e uma nova é criada.

Além disso não sabemos como nosso serviço será consumido, se dentro de uma aplicação web, desktop ou móvel, e precisamos garantir que seu funcionamento seja consistente em qualquer dessas situações. Por esse motivo não temos como deixar que o estado seja gerenciado, por exemplo, no controlador, pois essa camada não é desenvolvida pela nossa aplicação, e sim na aplicação cliente. Motivo pelo qual nosso bean precisará ele mesmo gerenciar seu estado, e portanto ser Stateful.

Então, partindo para a implementação, criaremos um novo EJB, chamado `NegociacaoBean`, como podemos ver a seguir.

```
@Stateful
public class NegociacaoBean {

    private Cliente cliente;
    private double saldoDevedor;
    private List<PropostaQuitacao> propostas = new ArrayList<>();

    ...
}
```

Aqui temos novas classes, como a `Cliente` e a `PropostaQuitacao`. Essas classes são simples, e serão mais bem exploradas ao trabalharmos com a camada de persistência. Agora precisamos delas apenas de forma a suportar a lógica do nosso EJB. Então vamos a elas.

```
public class Cliente implements Serializable {
    private String nome;
    private Set<Contrato> contratos;
    // getters e setters
}
```

```

}

public class Contrato implements Serializable {

    private Double saldo;
    private Cliente cliente;
    // getters e setters
}

```

Ao analisar a classe `Cliente`, já precisamos de uma classe `Contrato`, então as duas já foram apresentadas juntas. Além dessas, temos ainda a classe `PropostaQuitacao`, que é dependência do nosso bean.

```

public class PropostaQuitacao implements Serializable {

    private Date dataProposta;
    private Integer quantidadeContratos;
    private Double saldoOriginal;
    private Double valorProposto;
    //essa classe veremos logo mais
    private FormaPagamento formaPagamento;

    //construtor usando os campos
}

```

Além das propriedades apresentadas, teremos um construtor que recebe as propriedades como parâmetro. A exceção será a `dataProposta`, que receberá a data atual. Procure na sua IDE qual o atalho para gerar esse tipo de construtor assim como para gerar os *getters* e *setters*.

Vale lembrar que o código completo está no GitHub:
<https://github.com/gscordeiro/microservicos-ejbs/javacred>.

Agora vamos para implementação da lógica do nosso `NegociacaoBean`.

```

@Stateful
public class NegociacaoBean {

```

```
//....

public void iniciaNegociacao(Cliente cliente){

    this.cliente = cliente;

    this.saldoDevedor = cliente.getContratos().stream()
        .mapToDouble(Contrato::getSaldo).sum();

}
}
```

O primeiro método apresentado é bem simples, e simplesmente soma o saldo de todos os contratos de um determinado cliente. O bean tem essa estrutura, pois ele será instanciado ao abrirmos a tela da negociação, e nesse momento o atendente do setor de cobrança ainda não iniciou a negociação com um cliente específico. Imagine que ele chegou ao serviço e vai correr uma lista de clientes para contatar naquele dia. Entre todas essas ligações a tela permanecerá aberta e, a cada novo contato, o método `iniciaNegociacao` será chamado para um novo `Cliente`.

Porém, o método que realmente faz a negociação ainda não foi visto, e este sim fará uso da classe `FormaPagamento` usada pela classe `PropostaQuitacao`. Ela não foi esquecida, apenas vamos analisá-la junto com o método de negócio propriamente dito, assim fará mais sentido.

```
@Stateful
public class NegociacaoBean {

    //...

    public void registrarProposta(FormaPagamento formaPagto){

        double desconto = saldoDevedor * formaPagto.getDesconto();

        int qtdeDeContratos = cliente.getContratos().size();
        double valorProposto = saldoDevedor - desconto;
    }
}
```

```

        propostas.add(new PropostaQuitacao(qtdeDeContratos,
                                            saldoDevedor,
                                            valorProposto,
                                            formaPagamento));
    }
}

```

Aqui vemos que a própria `FormaPagamento` possui o percentual de desconto. Até poderíamos ter um cadastro de diferentes formas de pagamento, cada qual com seu percentual de desconto vinculado. Para simplificar nossa aplicação, usaremos as três formas de pagamento já descritas: à vista, parcelado até seis vezes, e parcelado em mais de seis vezes. O código fica da seguinte forma:

```

public enum FormaPagamento {

    VISTA(30), ATE_6_VEZES(20), MAIS_6_VEZES(10);

    private double desconto;

    private FormaPagamento(double percentual) {
        this.desconto = percentual/100;
    }

    public double getDesconto() {
        return desconto;
    }
}

```

Como só temos essas três formas de pagamento, foi utilizada uma enumeração. Apesar de usarmos bastante `enum` no cotidiano, esquecemos muitas vezes que elas são classes Java completas, podendo ter atributos como qualquer outra. Por isso, criamos o desconto como um atributo, e o alimentamos via construtor, que é chamado em cada uma das instâncias da nossa `enum`: `VISTA`, `ATE_6_VEZES` e `MAIS_6_VEZES`.

Analisando em conjunto a `enum FormaPagamento` com o método `registrarProposta`, vemos que a lógica é bem simples; só tivemos

muito código porque tivemos que criar várias classes que dão sentido ao nosso exemplo.

Agora só falta mais uma classe para nosso exemplo rodar, e assim como fizemos no capítulo anterior, criaremos uma *servlet* para testar nosso EJB. Ainda chegará a hora de criarmos uma tela, mas aí entram novos conceitos como escopos e interface Web, por isso é mais produtivo fazermos um cliente mais simples agora.

```
import static
    br.com.casadocodigo.javacred.entidades.FormaPagamento.*;

@WebServlet("/negociacao-bean")
public class NegociacaoBeanServlet extends HttpServlet {

    @EJB
    private NegociacaoBean negociacao;

    protected void doGet(HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException {

        Cliente cliente = buscarClientePorCPF(1234567890);

        negociacao.iniciaNegociacao(cliente);

        negociacao.registrarProposta(VISTA);
        negociacao.registrarProposta(ATE_6_VEZES);
        negociacao.registrarProposta(MAIS_6_VEZES);

        negociacao.getPropostas().forEach(System.out::println);

        resp.getWriter().printf("Teste finalizado");
    }
}
```

Aqui fizemos um código parecido com o que vimos no capítulo passado, mas a grande diferença é que um EJB Stateful é basicamente igual um objeto java comum, que estamos

acostumados a usar, e que a cada invocação de método não esquece seu estado; obviamente que com a adição de características como as transações automáticas e demais gerenciamentos feitos pelo contêiner.

Mas nosso exemplo ainda não está completo, falta o conteúdo do método `buscarClientePorCPF`. Esse é na verdade um método "falso", pois não faz busca alguma, ele apenas cria um `Cliente` qualquer para podermos executar nosso exemplo. Claro que em uma aplicação real buscaremos isso no banco de dados. Essa simplificação aqui é só para não adiantarmos conteúdo desnecessariamente e perdermos o foco.

```
private Cliente buscarClientePorCPF(int cpf) {
    Set<Contrato> contratos = new HashSet<>();

    Contrato contrato1 = new Contrato();
    contrato1.setValor(100_000.0);
    contrato1.setSaldo(40_000.0);
    contratos.add(contrato1);

    Contrato contrato2 = new Contrato();
    contrato2.setValor(100_000.0);
    contrato2.setSaldo(60_000.0);
    contratos.add(contrato2);

    Cliente cliente = new Cliente();
    cliente.setContratos(contratos);

    return cliente;
}
```

Finalmente, nosso exemplo está completo, e podemos executá-lo pela url `http://localhost:8080/javacred/negociacao-bean` para testar. Fazendo isso, o resultado apresentado no console apresentará as seguintes informações.

```
[saldoOriginal=100000.0, valorProposto=70000.0,
  formaPagamento=VISTA]
[saldoOriginal=100000.0, valorProposto=80000.0,
```



```
formaPagamento=ATE_6_VEZES]  
[saldoOriginal=100000.0, valorProposto=90000.0,  
formaPagamento=MAIS_6_VEZES]
```

Depois de testar nossa regra de negócio básica, podemos explorar com mais detalhes a diferença entre Stateless e Stateful. Para isso, troque a anotação `@Stateful` por `@Stateless` na classe

`NegociacaoBean`, e rode novamente o exemplo. Se o servidor ainda estiver executando, apenas republique nossa aplicação (no eclipse basta clicar com o botão direito e usar a opção `Full Publish`).

Ao executar você perceberá um `NullPointerException` dentro do método `registrarProposta` do nosso `NegociacaoBean`. Mais precisamente na linha onde recuperamos os contratos do cliente. Isso acontecerá porque o objeto `cliente`, que foi atribuído na chamada do método `iniciaNegociacao(Cliente)`, estará nulo. Como já foi dito quando estudamos os EJBs Stateless, esse tipo de bean não guarda estado, e a cada chamada, um novo objeto do *pool* nos é devolvido. Assim, o cliente que acabou de ser passado ao EJB, não pode ser recuperado em uma chamada de método posterior. Por esse motivo é que utilizamos um bean Stateful nesse exemplo. Depois de testar, volte nosso bean para Stateful, se não os demais exemplos não funcionarão corretamente.

4.3 Removendo um EJB Stateful

Como vimos, um *SFSB* é interessante quando queremos que um objeto no servidor fique sob controle do cliente, seja este remoto ou dentro do próprio servidor. Porém, como o objeto fica sempre disponível, diferentemente do Stateless, que é destruído a qualquer momento pelo servidor, precisamos informar quando o bean pode ser removido.

ALERTA DE SPOILER: REMOÇÃO AUTOMÁTICA À FRENTE

Talvez essa questão da remoção dos SFSB seja o "calcanhar de Aquiles" dessa tecnologia. Ficamos apreensivos de esquecer da remoção de alguma forma e termos problemas depois. Mas apesar de vermos as diversas formas de controlar a remoção desses beans, mais à frente neste capítulo veremos o que será usado na grande maioria dos casos, que é a remoção automática via CDI. Então se começou a ficar preocupado só de entrar nesse assunto de remoção manual de objetos (algo nada comum em uma tecnologia com *garbage collector*), fique tranquilo.

Para informar isso ao servidor, precisamos marca o objeto como pronto para ser removido, e para tanto chamamos algum método do bean anotado com `@Remove`. Tal método até pode conter alguma regra de negócio, como um método "salvar" ou "concluirTransacao" que salva tudo no banco de dados, mas o mais comum é fazer isso através do método anotado com `@PreDestroy` que veremos logo mais. Assim sendo, a principal função desse método anotado com `@Remove` é dizer ao servidor que o bean pode ser removido. Pode ser um método vazio, ou apenas com informação de log, por exemplo.

No nosso caso, utilizaremos apenas com uma linha que escreve no console que o método foi chamado. O ideal nesses casos é utilizar uma API de log, mas para não introduzir mais um ingrediente nessa nossa receita, faremos isso via escrita simples com o `System.out` mesmo.

Para fazer isso, voltemos à nossa *servlet*, onde utilizamos a nossa instância de `NegociacaoBean`, e vamos incluir a seguinte linha de código:

```
@WebServlet("/negociacao-bean")
public class NegociacaoBeanServlet extends HttpServlet {
```

```

...
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException {

    //o restante do código permanece igual

    negociacao.getPropostas().forEach(System.out::println);

    negociacao.remove(); //linha adicionada

}
}

```

Esse método `remove()` ainda não existe, então vamos adicioná-lo na classe `NegociacaoBean`. Nas principais IDEs, na própria linha com o erro de compilação pela falta do método teremos sugestões de como resolver o problema, e entre eles acrescentar o método que falta na nossa classe. Feito isso, o método será gerado. Agora sim, dentro deste método que foi criado pela IDE, é que vamos colocar o seguinte código:

```

@Remove
public void remove() {
    //linha criada utilizando "systr" (no eclipse)
    //ou "sout" (no netbeans) e depois Ctrl/Command + espaço
    System.out.println("NegociacaoBean.remove()");
}

```

Além do corpo do método, adicionamos a anotação `@Remove`, do pacote `javax.ejb`, e é ela que marca o método de forma que o servidor entenda que após sua execução, o bean pode ser destruído. Fazer isso é muito importante, pois se usarmos beans `Stateful` e não os liberarmos, estaremos entulhando um monte de objetos no servidor, objetos esses que nunca serão destruídos; e com o passar do tempo, é bem provável que a aplicação apresente problemas de consumo excessivo de recursos, como memória e IO.

Para comprovar a remoção do bean, podemos fazer o teste a seguir, alterando nossa *servlet* de testes.

```
@WebServlet("/negociacao-bean")
public class NegociacaoBeanServlet extends HttpServlet {

    ...
    protected void doGet(HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException {

        //o restante do código permanece igual

        negociacao.getPropostas().forEach(System.out::println);

        negociacao.remove();

        //mando imprimir novamente
        negociacao.getPropostas().forEach(System.out::println);

    }
}
```

Dessa maneira veremos que nosso código executa bem até o momento em que tentamos imprimir novamente as propostas após termos chamado o método `remove()` da `negociacao`. Nesse caso, será lançada a exceção `javax.ejb.NoSuchEJBException`, informando que o nosso EJB não pôde mais ser encontrado, pois foi removido.

Mas então se este é o último método do nosso EJB que conseguimos chamar antes de ele ser removido da memória, não seria nele que deveríamos fazer a liberação de recursos? Antes disso, o que é mesmo uma liberação de recursos?

Liberação de recursos com `@PreDestroy`

Os recursos mais comuns que precisam ser liberados são conexões como o bando de dados ou acesso a arquivos que precisam ser fechados. Pelo menos era assim antigamente. Hoje, com o

fechamento automático de recursos adicionado desde o Java 7, conhecido como *try-with-resources*, esse tipo de recurso é fechado automaticamente quando o bloco onde eles são usados termina.

E isso se estivermos falando de Java SE. Como este livro é sobre Java EE, a gestão de recursos é ainda mais automatizada. Não precisamos lidar diretamente com conexões com o banco de dados, e mesmo os recursos que encapsulam a conexão, como um `EntityManager`, também são fechados automaticamente quando utilizamos EJBs. Mas então, por que estamos falando disso se tudo é automático? Porque um recurso muito importante que precisa ser fechado é o EJB Stateful como acabamos de ver.

Quando um bean, seja um EJB ou mesmo um managed bean do JSF, faz uso de um EJB Stateful, este último precisa ser liberado. Veremos que usando `@Inject` a partir do Java EE 7 isso não é mais necessário, mas se estivermos usando `@EJB` com Java EE 5 ou 6, essa remoção deve ser feita manualmente através da invocação de seu método anotado com `@Remove`.

Dessa maneira, o melhor lugar para colocar esse fechamento é em um método anotado com `@PreDestroy`, pois esse é o método que sempre é chamado antes do bean ser destruído. Novamente o bean em questão não precisa ser um EJB. Até mesmo *managed beans* do JSF ou do CDI têm o método anotado com `@PreDestroy` chamado antes de serem removidos da memória.

Isso se torna ainda mais importante porque em casos de remoção via *timeout* como veremos a seguir, ou na remoção automática quando usamos `@Inject`, o método anotado com `@Remove` não é chamado pelo servidor, pois é um método usado somente para uma remoção manual.

Na verdade, usando Java EE 7 (ou superior) e `@Inject`, só precisaríamos da remoção manual em caso de invocações remotas como a que estamos usando na `NegociacaoMain`. Então se estivermos usando um *SFSB* apenas dentro da aplicação, nem haverá um

método anotado com `@Remove`. Já o método anotado com `@PreDestroy` é sempre chamado para que possamos fazer a liberação de recursos, independente de como o servidor ficou sabendo que podia remover o bean (se automática ou manualmente).

4.4 Removendo um bean Stateful via timeout

Outra forma de removermos nossos beans Stateful é através de *timeout*. Dessa maneira, assim que o bean ficar ocioso, o tempo do timeout começa a contar, e quando ele é alcançado o bean é removido. Sua declaração é bastante simples, basta adicionarmos a anotação `javax.ejb.StatefulTimeout` no nosso EJB Stateful.

```
@Stateful
@StatefulTimeout(30)
public class NegociacaoBean {
    ...
}
// ou mais especificamente

@Stateful
@StatefulTimeout(value=30, unit=TimeUnit.MINUTES)
public class NegociacaoBean {
    ...
}
```

A anotação `StatefulTimeout` por padrão marca o tempo em minutos, mas podemos usar todas as medidas disponíveis na enumeração `java.util.concurrent.TimeUnit` para especificar o *timeout*. Dentre os valores possíveis temos dias, horas, minutos, segundos e medidas ainda menores.

Temos ainda dois valores especiais para esse timeout:

- **Zero**: indica que assim que ficar ocioso, o bean pode ser removido.
- **-1 (menos um)**: indica que o bean nunca será removido por timeout.

É importante notarmos que, se o cliente desse bean tentar utilizá-lo depois de passado esse timeout, ele terá a mesma exceção que é lançada quando tentamos acessar um EJB removido programaticamente como vimos na seção anterior, a

`javax.ejb.NoSuchEJBException`.

Da mesma forma que ocorre quando removemos o bean manualmente, antes de ele ser removido por timeout, o método anotado com `@PreDestroy` será chamado para que possamos liberar recursos.

A remoção via *timeout* é um método a mais para termos a certeza de que os beans não ficarão ocupando espaço no servidor, mas é interessante termos como plano original a gestão correta do ciclo de vida dos beans, para não ficarmos dependentes do *timeout*. Tendo esse cuidado é provável que nossa aplicação fique mais acertada. Não porque o *timeout* pode falhar, mas por procurarmos cuidar melhor da nossa aplicação. Mas é claro que esse cuidado todo faz sentido se estivermos usando versões anteriores ao Java EE 7, onde poderíamos esquecer de remover corretamente algum SFSB. A partir do Java EE 7 basta usarmos `@Inject` para injetar nossos EJBs que a gestão do ciclo de vida dos SFSB será automática.

4.5 Lidando com Stateful Session Beans removidos e `NoSuchEJBException`

Como acabamos de ver na seção anterior, pode acontecer de nosso *SFSB* ter sido removido por um timeout e ao tentar utilizá-lo teremos a `NoSuchEJBException`. O que faremos nesse caso?

Mas antes de responder, vamos abrir um pouco mais o leque de situações onde podemos enfrentar um caso como esse. Quando chegarmos na seção **7.4 EJB Stateful e o rollback de transações** compreenderemos melhor o processo como um todo, mas agora basta sabermos que uma exceção que não seja da aplicação (por padrão qualquer uma filha de `RuntimeException`) também causa a remoção do *SFSB*, e nesse caso ao tentar utilizá-lo após essa exceção teremos o mesmo problema.

Para resolver esse caso, considerando que estamos utilizando injeção de dependências para conseguir nosso EJB (preferencialmente via `@Inject`, mas também pode ser via `@EJB`), acabamos com uma solução pouco intuitiva. Isso porque quando trabalhamos com injeção de dependências não precisamos fazer `lookup`, mas é isso que vamos precisar fazer.

Quando o *SFSB* injetado pelo servidor é removido (por timeout, exceção etc.), o servidor não injeta outro novo em seu lugar, pelo menos não até a versão de Java EE disponível enquanto escrevo. É como se a primeira instância nos fosse dada de graça, mas se precisarmos de outra teremos que buscar via `lookup` da mesma forma que um cliente remoto precisa fazer. A diferença é que podemos usar nomes internos dos EJBs, que são mais curtos. Algo como `java:module/<nome do ejb>` já é o suficiente.

No caso da aplicação remota que já precisa fazer `lookup`, caso o *SFSB* seja removido, faremos um novo `lookup`. Mas aqui, como já precisamos fazer essa operação desde a primeira vez, não soa tão estranho.

Por fim, temos uma outra forma indireta de fazer um `lookup` se estivermos usando CDI:

```
@Inject //poderia ser @EJB, mas @Inject é melhor
private UmStatefulSessionBean sfsb;
```

```
@Inject
private Instance<UmStatefulSessionBean> sfsbInstance;
```



```
try{
    sfsb.metodoQueLancaExcecao();

} catch (RuntimeException e) {
    //aqui o sfsb já foi removido por ter sido lançada uma
    //RuntimeException, então fazemos um novo "lookup"
    sfsb = sfsbInstance.get();
}
```

Esse código está aqui como referência e não vai fazer parte do nosso projeto, é apenas um exemplo. Como já citado antes, na seção **7.4 EJB Stateful e o rollback de transações** veremos novamente esse assunto de uma forma mais completa. O foco em incluir esse tópico aqui é para agrupar os diversos assuntos relacionados aos *SFSB* no mesmo local.

Por enquanto, o mais importante é vermos o uso da interface `javax.enterprise.inject.Instance` que nos possibilita criar uma espécie de ponteiro para chegar até à instância que seria injetada. É como se fosse um lookup *lazy* do EJB, mas usando a API da CDI. E também não se preocupe sobre a remoção do EJB com exceção, pois a partir da seção **7.3 Exceções de sistema e exceções de aplicação** isso ficará bem contextualizado.

4.6 Remoção automática de beans Stateful associados a um contexto CDI

A versão 3.2 da especificação de EJBs, que é a presente no Java EE 7 e no Java EE 8, trouxe essa esperada funcionalidade para nós desenvolvedores. Ainda não estudamos sobre CDI, mas vale a pena adiantarmos essa informação aqui pois estamos vendo as formas de controlar os Stateful beans, e essa é sem dúvidas a forma mais simples de alcançar esse objetivo.

Na verdade, isso já foi bastante dito neste capítulo, mas realmente para que tenhamos uma gestão automática do ciclo de vida dos *SFSB* basta que em vez de injetá-los com `@EJB`, façamos isso usando `@Inject`.

Se o *SFSB* só for utilizado via injeção de dependências (`@Inject`, não `@JEB`), e nunca fazemos lookup para recuperá-lo, o mesmo nem precisa de um método anotado com `@Remove`, pois sempre será removido automaticamente. Na prática, teremos isso na maior parte dos nossos *SFSB*, pois uma minoria deles precisa de acesso remoto, onde o lookup é obrigatório. E por mais que tenhamos visto casos onde faremos lookup para buscar novamente beans removidos, isso também está longe de ser a regra, e no próximo capítulo quando trabalharmos com Web ficará mais claro.

Mas agora vamos considerar o seguinte, se é tão simples usar essa gestão automática, por que estamos vendo também as formas menos automatizadas? Pelo simples fato de que nem sempre temos a última versão de todas as especificações à disposição no nosso projeto. Podemos ter só Java EE 5 ou 6 ao nosso dispor. Sem contar nas situações como acesso remoto onde precisamos remover manualmente o bean. Agora se tivermos a possibilidade de só utilizar a gestão automática, com certeza nossa vida será muito mais simples.

4.7 EJB Singleton

Os EJBs do tipo Singleton são uma implementação do padrão de projeto que tem o mesmo nome, que consiste em disponibilizar uma única instância de um determinado objeto. Esses EJBs são como os *Stateful beans*, mantendo seu estado entre diferentes requisições, porém compartilhados com qualquer cliente que tentar utilizá-lo. São úteis quando queremos que a mesma informação esteja visível para

todos os clientes. Como exemplo, podemos pensar em um bean que lista todas as agências da nossa financeira.

```
import javax.ejb.Singleton;

@Singleton
public class AgenciasBean {

    private List<Agencia> agencias;

    public List<Agencia> getAgencias(){
        return agencias;
    }

    @PostConstruct
    public void buscarListaAgencias(){
        //vai no banco e carrega a propriedade agencias
    }
}
```

Nesse exemplo da listagem de agências, essa lista, que costuma ser praticamente estática, será buscada no banco apenas uma vez, e será reutilizada em qualquer chamada de qualquer cliente que utilizar o `AgenciasBean`.

Por padrão, o bean é criado na primeira vez que alguém tentar utilizá-lo; mas podemos mudar isso para forçar que seja criado assim que a aplicação terminar de inicializar. Isso pode ser útil por exemplo se tivermos um processo muito custoso para construir o objeto.

Vamos supor que a lista de agências precisa ser buscada em diversas fontes de dados e que isso demora bastante. Por mais que seja um processo executado somente uma vez durante a vida da aplicação, o primeiro usuário que tentar acessar essa funcionalidade vai sofrer um pouquinho.

Imaginando que nossa aplicação deverá sofrer manutenções programadas para as madrugadas, seria interessante já a subir com

essa lista carregada, pois há uma probabilidade menor de causar lentidão no sistema nesse horário. Para isso, basta utilizarmos a anotação `@javax.ejb.Startup`.

```
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Singleton
@Startup
public class AgenciasBean {
    // o resto continua igual
}
```

Agora sim, ao subir a aplicação, mesmo sem nenhuma utilização, o bean será carregado e seu método anotado com `@PostConstruct` será chamado.

Para fechar o capítulo

Embora não tenhamos evoluído muito nosso exemplo da financeira nesse capítulo, a ideia é que ele e o próximo sirvam como um guia rápido para que, quando for necessário, relembremos onde usar cada um dos tipos de EJB. Enquanto aqui vimos os *Session Beans* por estado, no próximo os veremos classificados por tipo de interface.

CAPÍTULO 5

Os Sessions Beans por interface

Agora que já vimos a classificação dos *Session Beans* quanto ao seu estado (*Stateless*, *Stateful* ou *Singleton*), veremos a classificação quanto à visibilidade: *remote*, *local* e *no-interface*, sendo que o último nada mais é do que uma simplificação dos EJBs locais.

5.1 EJBs Remotos

Iniciaremos pelos beans remotos pelo fato de eles serem menos utilizados; então é legal já entendermos o que eles são para depois não ficarmos com receio de usá-los. Um bean remoto é um EJB que pode ser chamado de fora da mesma JVM, podendo estar na mesma máquina, via rede ou mesmo internet; parecido com chamar um *WebService*. Mas aqui por ser conversa de Java com Java, temos algumas vantagens de que trataremos logo mais. E antes do exemplo em si, vamos a um questionamento muito recorrente.

EJB remoto, jar contendo a implementação, ou *WebService*?

Primeiro vamos dividir nossa análise em opções: execução local via utilização de um jar, ou execução remota via EJB remoto ou *WebService* (WS). Nessa primeira análise não discutiremos as diferenças entre o EJB remoto e o WS, trataremos do que eles têm em comum.

Então, por que usar uma solução remota sendo que poderíamos colocar a lógica dentro de um jar e usar diretamente a implementação? A vantagem da solução remota está na terceirização do problema para um serviço. Dessa maneira, se a

regra de negócio desse serviço precisar mudar, basta alterar a implementação do EJB remoto ou WS e os clientes já usariam a nova implementação. Se utilizássemos um arquivo jar, ao mudar uma regra de negócio precisaríamos percorrer todas as aplicações que fazem uso desse serviço para garantir que todos estivessem executando na última versão.

Apenas como um exemplo bem simples, imagine que esse serviço ou esse jar tenha o cálculo de impostos devidos, e que, por uma mudança na lei que entrou em vigor imediatamente, esse cálculo precisa ser ajustado. Contudo, em vez de alterar em um só lugar, tivemos que alterar em dezenas, e um ou outro ficou para trás. O problema é que, ao executar o código velho, o imposto calculado ficará errado, o que pode acarretar prejuízos à empresa, seja pelo pagamento excessivo de impostos, seja por pagar a menos e ser multada pelos órgãos de fiscalização.

Então quase sempre é melhor optarmos por uma solução remota quando temos regra de negócio envolvida. Arquivos jar são para utilitários, como formatadores de arquivos, coisas que realmente vão executar localmente. Ou então quando a solução tem um requisito de performance tão severo que a inclusão da passagem pela rede inviabilize o uso de uma solução remota, seja EJB remoto ou WS. Mas lembre-se de que nesse caso teremos a dificuldade de atualizar os arquivos jar de todos os clientes quando tivermos uma versão nova da implementação.

Mas e a escolha entre EJB remoto e WS, como podemos fazer? Existe o cenário simples, que é quando temos implementação e cliente em tecnologias diferentes: por exemplo, um serviço escrito em Java sendo consumido pelo PHP ou Ruby. Nesse caso usaríamos WS, sem dúvidas. Mas e quando ambos são escritos em Java, existe alguma vantagem em usar EJB remoto ou eu deveria usar logo WS e ficar acessível para quem sabe um futuro cliente em outra linguagem?

A vantagem dos EJBs remotos existe se estivermos fazendo conversarem dois servidores de aplicação, um com o cliente do EJB e outro com o próprio EJB, pois nesse caso os servidores conseguem fazer um controle de transação distribuído, bastando para tanto que os *datasources* seja do tipo *XA*. Na prática ganhamos a tranquilidade de não termos que nos preocupar com o controle de uma possível divergência entre o cliente e o servidor de um serviço.

Por exemplo, imagine que nosso sistema `javacred` tem a opção de empréstimo consignado (com desconto em folha) e que temos um outro sistema que cuida da folha de pagamento de algumas empresas. Quando um empréstimo consignado é requerido, o nosso sistema se comunica com o de folha de pagamento para ver se as parcelas cabem no salário do solicitante e, então, através de um serviço (EJB) do sistema de folha de pagamento, agenda os débitos para os próximos pagamentos daquela pessoa. Esse agendamento devolve um protocolo que é salvo junto do registro de empréstimo dentro do `javacred`.

Agora vamos imaginar que, nessa última operação, ocorre um erro ao salvar o registro do empréstimo porque o banco de dados do `javacred` está indisponível, porém o agendamento dos débitos já foi feito no outro sistema. O mais comum nesse tipo de situação é ter que desfazer a operação que deu certo, mas e se também ocorrer algum erro no ato de desfazer essa operação? Se estivermos usando EJBs remotos e *XA-Datasources*, esse controle é feito automaticamente.

Ainda veremos detalhes de como as transações funcionam no capítulo **7. Gerenciando transações com EJBs**, mas a vantagem do uso de EJBs em vez de WS é que, usando os primeiros, a transação pode ser distribuída, remota, nos dispensando de ficar cuidando manualmente da integridade.

5.2 Desenvolvendo um EJB remoto

Como vimos, o EJB remoto é uma alternativa aos WebServices, e a sua principal vantagem está relacionada às vantagens dos EJBs que foram tratadas no capítulo passado e também no controle de atomicidade quando há a conversa entre serviços usando *XA-Datasources*.

Apesar do cenário onde os EJBs remotos se mostram mais vantajosos, seja na comunicação entre servidores, onde podemos ter as transações distribuídas citadas logo antes, a configuração da comunicação entre servidores é bem específica em cada servidor de aplicação. Dessa maneira compensa mais consultar a documentação do servidor que estiver utilizando tanto como cliente quanto como servidor do serviço oferecido via EJB remoto.

Então para não dispersarmos nossa atenção com diversas configurações proprietárias, faremos um cliente desktop (classe *main*), que será o suficiente para compreendermos o funcionamento dos EJBs remotos. Nesse cliente vamos fazer uso da

`CalculadoraFinanciamento` para fazer uma simples simulação passando um valor e uma quantidade de parcelas.

Mas, para termos um EJB remoto, precisamos que nosso bean implemente uma interface remota, como a seguinte.

```
import javax.ejb.Remote;

@Remote
public interface CalculadoraFinanciamentoRemota {

    double simulaFinanciamento(double valorEmprestimo, int meses);

}

import javax.ejb.Stateless;
@Stateless
public class CalculadoraFinanciamento implements
```



```
        CalculadoraFinanciamentoRemota {  
    ...  
}
```

Geralmente colocamos a anotação `@Remote` na interface que estamos definindo, mas podemos ter algumas variações. A primeira delas é baseada na possibilidade de nossa interface não ter qualquer anotação, mais comum quando reutilizamos uma interface preexistente. Vamos considerar que essa interface esteja dentro de um `jar`, e não no nosso código-fonte; e que não possamos ou não queiramos editá-la para colocar a anotação. Nesse caso podemos fazer a seguinte modificação no nosso exemplo.

```
//interface sem anotação  
public interface CalculadoraFinanciamentoRemota {  
  
    double simulaFinanciamento(double valorEmprestimo, int meses);  
  
}  
...  
  
import javax.ejb.Remote;  
import javax.ejb.Stateless;  
  
@Remote(CalculadoraFinanciamentoRemota.class)  
@Stateless  
public class CalculadoraFinanciamento implements  
    CalculadoraFinanciamentoRemota {  
    ...  
}
```

Nesse trecho de código nós informamos que a interface `CalculadoraFinanciamentoRemota` será a interface remota do nosso bean colocando a anotação `@Remote` nele próprio. Há também a possibilidade, a partir do EJB 3.2, de apenas anotar nossa classe com `@Remote`, e todas as interfaces que ela implementar, se não especificarem nada diferente, serão interfaces remotas desse bean, como no trecho a seguir.

```
//interfaces sem anotação
public interface CalculadoraFinanciamentoRemota {...}
public interface ServicoConcessaoEmprestimos {...}
```

...

```
@Remote
@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota,
    ServicoConcessaoEmprestimos {...}
```

A interface `ServicoConcessaoEmprestimos` não faz parte do nosso projeto, está aqui apenas como um exemplo a mais da declaração de uma interface remota. No exemplo anterior, tanto a

`CalculadoraFinanciamentoRemota` quanto a `ServicoConcessaoEmprestimos` serão expostas como interfaces remotas do nosso bean

`CalculadoraFinanciamento`. Porém, como veremos a seguir, se alguma das interfaces implementadas tiver explicitamente a marcação de interface remota, esteja a anotação nela própria ou no bean, somente ela será exposta como interface de negócio.

```
@Remote
public interface CalculadoraFinanciamentoRemota {...}
//interface sem anotação
public interface ServicoConcessaoEmprestimos {...}
```

...

```
@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota,
    ServicoConcessaoEmprestimos {...}
```

Agora somente a interface `CalculadoraFinanciamentoRemota` será interface remota do nosso bean. O mesmo efeito é conseguido com o código a seguir.

```
//interfaces sem anotação
public interface CalculadoraFinanciamentoRemota {...}
```

```
public interface ServicoConcessaoEmprestimos {...}
```

```
...
```

```
@Remote(CalculadoraFinanciamentoRemota.class)
```

```
@Stateless
```

```
public class CalculadoraFinanciamento implements
```

```
    CalculadoraFinanciamentoRemota,
```

```
    ServicoConcessaoEmprestimos {...}
```

Apesar de nossos exemplos serem com um bean remoto Stateless, nada impede que o mesmo seja remoto e Stateful.

A IMPORTÂNCIA DE IMPLEMENTARMOS SERIALIZABLE QUANDO POSSÍVEL

Sempre que possível, é bom implementar a interface `Serializable` nas nossas classes que armazenam informação. Apesar de aqui estarmos utilizando tipos básicos no nosso bean remoto, para passar tipos complexos pela rede é preciso que eles sejam serializáveis. Também precisaremos que nossos objetos sejam serializáveis quando formos armazená-los em escopos web, como veremos em capítulos adiante.

Em alguns casos, quando não fazemos nossas classes serializáveis, receberemos uma exceção que indica a falta dessa capacidade (geralmente em escopos da Web com CDI), mas teremos casos, como no uso de EJBs remotos, em que poderemos ter erros cuja mensagem em nada dá a entender que o que falta é ser serializável. Portanto, é bom criar o hábito de fazer serializável quando possível pois evitamos casos como este último onde perderíamos tempo procurando outras soluções.

5.3 Criando um cliente de EJB remoto

Para acessar nosso bean por meio de uma classe `main` vamos criar um novo projeto, que vai abrigar essa classe. Seguiremos o mesmo processo usado na criação do nosso projeto principal, mas em vez de um projeto Web, usaremos `Java with Maven > Java Application`, e o chamaremos de `javacred-desktop`.

Esse projeto precisará de duas coisas no seu *classpath*. A primeira é a interface `CalculadoraFinanciamentoRemota`, e a outra é o `jar` que nos dá acesso às classes que possibilitam a comunicação entre essa nova aplicação e o Wildfly.

A primeira dependência nós resolvemos acessando as classes do projeto `javacred`. No entanto, aquele é um projeto Web, e no final gera um arquivo `war` e por isso não temos como adicioná-lo diretamente como dependência dentro do `pom.xml` do `javacred-desktop`. Vamos fazer uma alteração no `pom.xml` do projeto Web para que, além do `war`, seja gerado um `jar` com as classes daquele projeto.

Para tanto, faremos a seguinte alteração no `pom.xml` do projeto `javacred`:

```
...
<build>
  <finalName>${artifactId}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <!-- parte nova -->
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
```

```

        <version>2.6</version>
        <configuration>
            <attachClasses>true</attachClasses>
        </configuration>
    </plugin>
</plugins>
</build>
...

```

Adicionando esse trecho da segunda tag `<plugin>`, logo abaixo da configuração para usar o Java 8, fazemos com que seja gerado esse `jar` extra que ficará ao lado do `war` no nosso repositório. Como exemplo, se nosso `war` se chama `javacred-1.0-SNAPSHOT.war`, nosso `jar` vai se chamar `javacred-1.0-SNAPSHOT-classes.jar`, e para usá-lo precisaremos colocar o seguinte trecho no `pom.xml` do projeto `javacred-desktop`:

```

<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>javacred</artifactId>
    <version>1.0-SNAPSHOT</version>
    <classifier>classes</classifier>
</dependency>

```

Perceba que o sufixo `classes` do `jar` não faz parte da versão, tanto é que, se olharmos no repositório, o `jar` e o `war` ficam na mesma pasta (da mesma versão). Em vez disso, ele se refere a um classificador, e usamos a tag `classifier` para indicá-lo.

Feito isso, nosso projeto `javacred-desktop` já consegue acessar as classes do projeto `javacred`, mas faremos uso apenas da interface `CalculadoraFinanciamentoRemota`. Em um caso real, talvez o ideal fosse criarmos um `jar` que tem apenas as classes visíveis remotamente e o colocarmos como dependência tanto no projeto Web quando no desktop, mas para nosso código de aprendizado como fizemos já está bom.

Para completar a configuração do novo projeto, precisamos adicionar a dependência do cliente do Wildfly. Se estivéssemos

utilizando outro servidor de aplicações teríamos que fazer a mesma coisa, apenas a dependência em si é que seria diferente.

Portanto vamos adicionar mais uma dependência no `pom.xml` do projeto `javacred-desktop` :

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>19.0.0.Final</version>
  <type>pom</type>
</dependency>
```

A versão dessa dependência deve coincidir com a versão do nosso servidor.

Depois de tanta configuração de dependência, vamos mexer novamente com um pouco de código. Vamos criar uma classe com um método `main` . No exemplo a seguir, o nome dado à classe foi `FinanciamentoMain` . Basta criar uma classe Java qualquer, e digitar `psvm` (**public static void main**) dentro do corpo dela e apertar `Control/Command + Espaço` , que o método `main` é criado; isso se você estiver no Netbeans. Se estiver no Eclipse o atalho é digitar `main` e `Control/Command + Espaço` .

```
import javax.naming.Context;
import javax.naming.InitialContext;

public class FinanciamentoMain {

    public static void main(String[] args) throws Exception {

        Context context = new InitialContext();
    }
}
```

Perceba que criamos uma instância da interface `Context` , por meio da implementação `InitialContext` . Ambas são do pacote `javax.naming` , que seria como o DNS do Java.

Com isso, conseguiremos chegar até o servidor, que deve estar executando, e então pedir para ele a instância de `CalculadoraFinanciamentoRemota` que está lá dentro.

Conforme instanciamos o `InitialContext`, percebemos que a IDE nos avisa que precisamos tratar algumas exceções. Para simplificar nosso exemplo, colocamos `throws Exception` na declaração do método `main`. Assim esta e outras exceções que também podem ser lançadas no `lookup` do objeto já ficam tratadas (ou mais precisamente não tratadas, mas apenas repassadas).

O restante do código é simples, basta buscarmos o objeto dentro do servidor e então chamar seu método.

```
Context context = new InitialContext();

String nomeJndi = <nome JNDI>;
CalculadoraFinanciamentoRemota calculadora =
    (CalculadoraFinanciamentoRemota) context.lookup(nomeJndi);

double parcela = calculadora.simulaFinanciamento(20_000.0, 10);
System.out.println(parcela);
```

O detalhe dessa implementação está por conta do conteúdo da variável `nomeJndi`. O formato é basicamente este: `ejb:<nome do EAR>/<nome do Jar ou WAR>/<nome do EJB>!<interface usada para acessar o EJB>`.

Como não estamos utilizando um EAR, o espaço para especificar seu nome fica em branco, logo nosso nome JNDI vai começar com `ejb:/`. No nosso caso a interface que estamos utilizando é a `CalculadoraFinanciamentoRemota`, mas como precisamos especificar seu nome completo, fizemos da seguinte maneira.

```
String nomeJndi = "ejb:/javacred/CalculadoraFinanciamento!" +
    CalculadoraFinanciamentoRemota.class.getName()
```

Até fizemos o que era necessário do ponto de vista da programação Java, mas ainda faltam alguns detalhes relacionados à comunicação

entre nossa aplicação desktop com o servidor. Em um exemplo simples, esses dois processos executarão na nossa própria máquina, mas em um cenário real, o EJB estará em um servidor e a versão desktop estará na máquina do usuário; logo, precisamos dizer como a aplicação desktop faz para se comunicar com o servidor, passando pelo menos o IP e a porta.

Precisaremos de dois arquivos `properties`, sendo que o primeiro se chama `jndi.properties`. Este é um arquivo padrão, cujo conteúdo faz um link entre a instância de `InitialContext` com as classes contidas no `jar` do servidor que adicionamos no nosso projeto. No nosso caso, foi a dependência `wildfly-ejb-client-bom` que colocamos no `pom.xml` da aplicação desktop, se fosse o Glassfish o processo seria basicamente o mesmo, mas com sua respectiva dependência, e assim por diante nos demais servidores de aplicação. O conteúdo do arquivo precisa apenas uma linha como podemos ver a seguir.

```
java.naming.factory.url.pkgs = org.jboss.ejb.client.naming
```

Esse arquivo pode também ser substituído por uma configuração programática, que ficaria da seguinte maneira:

```
Properties properties = new Properties();
properties.put(Context.URL_PKG_PREFIXES,
    "org.jboss.ejb.client.naming");
Context context = new InitialContext(properties);
```

A única diferença é que, quando formos instanciar o `InitialContext`, temos que passar essa propriedade. Já utilizando o arquivo de propriedade, podemos apenas dar `new` sem qualquer parâmetro como fizemos antes, e a configuração é buscada automaticamente no arquivo `jndi.properties`. Lembre-se de não misturar as duas formas de configurar. Se criar o arquivo `properties`, utilize o construtor sem argumentos. Agora se preferir passar a configuração via construtor, não precisa criar o arquivo.

Com isso, quando instanciamos o `InitialContext` temos na realidade uma instância de uma classe que sabe em qual servidor vai se

conectar. No nosso caso é o Wildfly, e dessa maneira nem é preciso fazer uma chamada ao servidor para validar se o endereço passado no `lookup` está com a sintaxe correta, e a comunicação efetiva com o servidor só ocorre quando um método do nosso EJB é invocado. Isso vale para os EJBs Stateless como o nosso, no caso dos Stateful a ida no servidor ainda é necessária, mas vamos ver uma coisa de cada vez.

O segundo arquivo será específico do Wildfly, e se chama `jboss-ejb-client.properties`, e tem detalhes de como nossa aplicação e o servidor deverão se comunicar, como IP, porta, mecanismo de segurança utilizado (se houver). No nosso caso, o conteúdo ficou da seguinte forma:

```
remote.connections = default
remote.connection.default.host = localhost
remote.connection.default.port = 8080
```

Esses dois arquivos devem ficar na raiz do nosso *classpath*. Como estamos usando o Maven, ele ficará no diretório `src/main/resources`. Você pode baixar os dois arquivos do exemplo, assim como o restante do projeto no GitHub:

<https://github.com/gscordeiro/microservicos-ejbs/javacred-desktop>.

Agora ao executar nossa main, varemos que o resultado da parcela é exibido no console. Agora nosso EJB já funciona tanto na Web, através da nossa *servlet*, quanto no desktop.

5.4 EJBs Locais

Enquanto os EJBs remotos são aqueles executados fora da nossa aplicação, os EJBs locais são aqueles que nossa própria aplicação consome e são os mais comuns no dia a dia da maioria dos desenvolvedores.

Em aplicações reais, quase sempre temos mais de um serviço sendo oferecidos, mesmo que não haja a necessidade de expô-los remotamente. Um exemplo pode ser o pagamento de um boleto via site de um banco ou da nossa financeira. É tecnicamente possível disponibilizar esse serviço da mesma forma como fizemos com a simulação do financiamento. Mas por alterar o saldo do cliente, pode ser uma decisão de negócio não expor isso para outras aplicações, e então em vez de remoto, teremos um bean local.

Para definir um EJB local, o processo é praticamente o mesmo do remoto, trocando apenas a anotação utilizada na interface.

```
import javax.ejb.Local;

@Local
public interface ServicoPagamento {

    void efetuarPagamento(int identificacaoCobranca,
        double valor, Conta contaDebito);

}

import javax.ejb.Stateless;
@Stateless
public class ServicoPagamentoBean implements ServicoPagamento {
    ...
}
```

A implementação ainda não é importante, precisamos apenas notar que a forma de implementar é a mesma do bean remoto, mudando apenas a anotação da interface. Assim como fizemos no capítulo anterior, para utilizar nosso bean, o injetamos em quem precisar através da anotação `@EJB`, pelo menos até começarmos a ver um pouquinho de CDI.

```
import javax.ejb.EJB;

...
@EJB
```

```
private ServicoPagamento servicoPagamento;  
...
```

Como acabamos de ver, ao utilizar o bean, utilizamos sua interface, e não a classe que a implementa; é a mesma regra dos beans remotos. E assim também como no caso dos beans remotos, podemos utilizar beans locais tanto Stateless quanto Stateful, não há restrição na combinação de tipo de estado com tipo de visibilidade.

Veremos na seção seguinte os EJBs sem interface, mas quando nosso bean tem uma interface, caso que vimos até aqui, o mesmo não pode ser acessado de outra forma a não ser pela sua interface. Por exemplo, o código a seguir não funcionaria.

```
@EJB  
private ServicoPagamentoBean servicoPagamento;
```

Isso simplesmente não funciona porque, no exemplo em questão, a classe `ServicoPagamentoBean` implementa a interface local `ServicoPagamento`, então o bean só pode ser acessado através da interface.

Assim como ocorre com os EJBs remotos, temos uma boa variação de combinações para atribuir uma interface local a um bean, especialmente após a flexibilização das regras, introduzida na versão 3.2 da especificação. A lógica é a mesma dos beans remotos.

```
//interface sem anotação  
public interface ServicoPagamento {...}  
  
@Local(ServicoPagamento.class)  
@Stateless  
public class ServicoPagamentoBean implements  
    ServicoPagamento {...}
```

O código apresentado também expõe a `ServicoPagamento` como interface de negócio do nosso bean, assim como o trecho a seguir.

```
//interfaces sem anotação
public interface ServicoPagamento {...}
public interface ServicoConcessaoEmprestimos {...}

//poderia ter a anotação @Local aqui, que não faria diferença
@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

Por padrão, desde a versão 3.1 da especificação, mesmo sem qualquer aparição da anotação `@Local`, quando temos um EJB como o nosso `ServicoPagamentoBean`, as interfaces que ele implementa passam a ser automaticamente consideradas interfaces locais. Teríamos exatamente o mesmo efeito se além de `@Stateless` o bean estivesse anotado com `@Local`; mas essa possibilidade só veio na versão 3.2 da especificação.

Porém, assim como temos com os beans remotos, caso alguma das interfaces envolvidas esteja explicitamente marcada como local, as demais não são expostas como interface de negócio do bean. É o caso dos próximos exemplos, onde somente a `ServicoPagamento` será interface de negócio de bean.

Exemplo 1

```
@Local
public interface ServicoPagamento {...}
//interface sem anotação
public interface ServicoConcessaoEmprestimos {...}

@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

Estamos utilizando a interface `ServicoConcessaoEmprestimos` tanto no bean `CalculadoraFinanciamento` quanto no `ServicoPagamentoBean`, pois no mundo real é relativamente comum a aquisição de um empréstimo logo após uma simulação ou mesmo adquirir um financiamento para pagar outro, uma espécie de refinanciamento.

Além disso, utilizar as duas interfaces nos dá o cenário que precisamos para explorar as novidades das versões 3.1 e 3.2 dos EJBs.

Exemplo 2

```
//interfaces sem anotação
public interface ServicoPagamento {...}
public interface ServicoConcessaoEmprestimos {...}

@Local(ServicoPagamento.class)
@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

5.5 EJBs sem interface

O Java EE como um todo tem evoluído para simplificar o desenvolvimento, tornando-se simples de utilizar mesmo em aplicações menores. Até mesmo pelo nome "Enterprise", o Java EE costumava ser pensado para aplicações grandes e complexas, mas a maioria não é assim tão grande ou tão complexa. Foi aí que surgiu a expressão "utilizar uma bazuca para matar uma mosca". Os EJBs sem interface foram introduzidos na versão 3.1 da especificação de EJBs, logo, para utilizá-los é preciso ter um servidor Java EE 6, pelo menos.

Como dito antes, beans locais são muito mais comuns que os remotos. A todo o tempo criamos beans locais, e apesar de ser uma boa prática desenvolvermos direcionados pelas interfaces e não pela implementação, há casos simples - e outros nem tão simples assim - onde dificilmente teremos uma mudança na estratégia da implementação.

Uma grande vantagem da utilização da interface é poder trocar sua implementação como um todo sem afetar seus clientes. No caso de um EJB remoto, cujo cliente é uma aplicação de fora, é justo que esta fique mais isolada das mudanças internas dentro da nossa aplicação. Porém, quando lidamos com beans locais, naturalmente eles compartilham conhecimento da lógica de negócio, ajudando para alcançar um objetivo maior; nesse caso, isolar as classes via interface em alguns casos pode dar mais trabalho do que trazer benefícios. Em aplicações mais complexas, onde temos subsistemas ou módulos envolvidos, a utilização de interfaces nos beans locais ainda faz bastante sentido, mas em aplicações pequenas e médias, o processo de criar interface para qualquer bean começa a parecer um pouco burocrático.

Dessa forma, os beans sem interface são simplesmente classes Java simples, marcadas diretamente com `@Stateless` OU `@Stateful`. Podemos converter o exemplo do serviço de pagamento para um bean sem interface da seguinte maneira.

```
import javax.ejb.Stateless;
@Stateless
public class ServicoPagamentoBean {
    ...
}

...
@EJB
private ServicoPagamentoBean servicoPagamento;
```

Com isso, temos as mesmas vantagens de um EJB completo, com a simplicidade de criar uma classe Java simples e colocar uma anotação em cima dela. Só temos que ter o cuidado de continuar utilizando o bean através da injeção de dependências, pois se utilizarmos o operador `new`, estaremos com uma instância Java simples, sem controle de transação, sem contexto de segurança, sem coleta de estatísticas, sem interceptadores, e tudo mais que podemos fazer com EJBs.

Para finalizar o tópico sobre beans sem interface temos uma situação onde podemos ter uma classe que implementa uma interface qualquer, mas não queremos que essa interface seja utilizada para expor o EJB, queremos que ele seja considerado um bean sem interface, e para isso utilizamos a anotação

```
@javax.ejb.LocalBean .
```

```
//interface legada qualquer  
public interface Observavel {...}
```

```
...
```

```
import javax.ejb.LocalBean;  
import javax.ejb.Stateless;
```

```
@Stateless  
@LocalBean  
public class ServicoPagamentoBean implements Observavel{  
    ...  
}
```

Os EJBs sem interface, como já foi dito, foram introduzidos na versão 3.1 da especificação, e na versão 3.2 foi introduzida uma flexibilização no modelo de EJBs locais **com** interface. Enquanto, até a versão 3.1, obrigatoriamente tínhamos que colocar a anotação `@Local` na interface ou no bean para que ela fosse considerada a forma de expor o EJB, a partir da versão 3.2, se houver uma interface não anotada com `@Local` nem com `@Remote`, ela será considerada uma interface local, como já vimos na seção anterior.

Dessa maneira, como estamos utilizando um servidor Java EE 8, que entende EJB 3.2 (mesma versão do Java EE 7), se não fosse a anotação `@LocalBean`, não conseguiríamos injetar diretamente uma instância de `ServicoPagamentoBean`, como um bean sem interface, pois ele acabaria sendo exposto com a interface legada `Observavel`, cujos métodos não convém imaginar aqui. Porém, como usamos a anotação `@LocalBean`, essa e qualquer outra interface seria ignorada

e o bean é exposto diretamente (sem interface), para podermos usar da seguinte maneira.

```
@EJB
```

```
private ServicoPagamentoBean servicoPagamento;
```


USANDO EJBS ESCRITOS NA LINGUAGEM GROOVY

Groovy é uma linguagem de programação que executa na JVM e tem sintaxe muito parecida com Java, porém tem todos os aspectos positivos das linguagens de tipagem dinâmica. Pela proximidade com a linguagem Java, sua curva de aprendizado é muito menor para quem já programa em Java, e é uma boa alternativa para ser utilizada em equipes que não têm muito tempo para dominar uma nova linguagem.

Esse tópico se torna relevante quando falamos de Groovy, porque assim como em Java todos os objetos são descendentes de `Object`, em Groovy, além dessa regra do Java que também é aplicada, todo objeto implementa automaticamente a interface `GroovyObject`, que possui vários métodos relacionados às características dinâmicas da linguagem. Por se tratar de uma interface, um EJB escrito em Groovy fica no mesmo caso do exemplo que acabamos de ver, onde o bean `ServicoPagamentoBean` implementava a interface `Observavel`.

Sempre que formos utilizar EJBS 3.2 escritos em Groovy, é interessante colocar a anotação `@LocalBean`, mesmo que no código-fonte não haja explicitamente a implementação de qualquer interface; pois, assim como o compilador Java coloca o "extends Object", o compilador Groovy coloca o "implements GroovyObject" automaticamente, então precisamos estar atentos.

Versões mais novas do Wildfly já ignoram as interfaces do pacote `groovy.lang`, dentre elas a `GroovyObject`, mas outros servidores podem se comportar de maneira diferente.

5.6 Revisão geral sobre exposição de interfaces dos EJBs

Nesta seção vamos relembrar as combinações que vimos até aqui, e também colocar uma possibilidade que nem foi mencionada antes pois é menos comum, que é a possibilidade de expor um bean com uma interface - local ou remota - que ele não implementa, mas que tem assinatura compatível.

Exemplo remoto 1: caso básico do bean remoto

```
@Remote
public interface CalculadoraFinanciamentoRemota {...}

@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota {...}
```

Exemplo remoto 2: anotação da interface remota no bean

```
//interface sem anotação
public interface CalculadoraFinanciamentoRemota {...}

@Remote(CalculadoraFinanciamentoRemota.class)
@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota {...}
```

No caso do exemplo remoto 2, a utilização mais comum é quando não queremos ou não podemos alterar a interface para colocar a anotação nela.

Exemplo remoto 3: expõe duas interfaces de negócio remotas

```
//interfaces sem anotação
public interface CalculadoraFinanciamentoRemota {...}
public interface ServicoConcessaoEmprestimos {...}

@Remote
```

```
@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota,
    ServicoConcessaoEmprestimos {...}
```

Anotar o bean com `@Remote` sem especificar qual das interfaces é a remota significa que todas serão remotas a menos que especifiquem o contrário.

Exemplo remoto 4: expõe só a interface de negócio remota

`CalculadoraFinanciamentoRemota`

```
@Remote
public interface CalculadoraFinanciamentoRemota {...}
//interface sem anotação
public interface ServicoConcessaoEmprestimos {...}
```

```
@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota,
    ServicoConcessaoEmprestimos {...}
```

Exemplo remoto 4.1: expõe só a interface de negócio remota

`CalculadoraFinanciamentoRemota`

```
//interfaces sem anotação
public interface CalculadoraFinanciamentoRemota {...}
public interface ServicoConcessaoEmprestimos {...}
```

```
@Remote(CalculadoraFinanciamentoRemota.class)
@Stateless
public class CalculadoraFinanciamento implements
    CalculadoraFinanciamentoRemota,
    ServicoConcessaoEmprestimos {...}
```

Nesse caso, como foi especificada a `CalculadoraFinanciamentoRemota` dentro da anotação `@Remote`, somente ela será remota, as demais serão locais a menos que estejam anotadas de forma diferente.

Exemplo local 1: caso básico do bean local

```
@Local
public interface ServicoPagamento {...}

@Stateless
public class ServicoPagamentoBean
    implements ServicoPagamento {...}
```

Exemplo local 2: anotação da interface local no bean

```
public interface ServicoPagamento {...}

@Local(ServicoPagamento.class)
@Stateless
public class ServicoPagamentoBean
    implements ServicoPagamento {...}
```

É o mesmo caso da diferenciação do exemplo 1 e 2 do EJB remoto. Usamos o mostrado no exemplo 2 quando não queremos ou não podemos anotar a interface `ServicoPagamento` com `@Local`.

Exemplo local 3: expõe duas interfaces de negócio locais

```
//interfaces sem anotação
public interface ServicoPagamento {...}
public interface ServicoConcessaoEmprestimos {...}

@Local
@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

Esse exemplo é correspondente ao exemplo 3 dos EJBs remotos. Anotando o próprio EJB com `@Local`, a menos que as interfaces por ele implementadas especifiquem algo diferente, elas serão locais.

Exemplo local 3.1: expõe duas interfaces de negócio locais (sem correspondente nas interfaces remotas)

```
//interfaces sem anotação
public interface ServicoPagamento {...}
public interface ServicoConcessaoEmprestimos {...}

@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

A partir da versão 3.1 da especificação dos EJBs, caso não especifiquemos nada em uma interface, ela já é considerada uma interface `@Local` .

Exemplo local 4: expõe só a interface de negócio local **ServicoPagamento**

```
@Local
public interface ServicoPagamento {...}
//interface sem anotação
public interface ServicoConcessaoEmprestimos {...}

@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

Já nesse caso como uma das interfaces está anotada com `@Local` , somente ela será exposta como interface do EJB.

Exemplo local 4.1: expõe só a interface de negócio local **ServicoPagamento**

```
//interfaces sem anotação
public interface ServicoPagamento {...}
public interface ServicoConcessaoEmprestimos {...}

@Local(ServicoPagamento.class)
@Stateless
public class ServicoPagamentoBean implements
    ServicoPagamento, ServicoConcessaoEmprestimos {...}
```

É o mesmo que o caso anterior, expõe como interface de negócio apenas a `ServicoPagamento` , pois ela foi especificada como `@Local` no

próprio bean. Uma interface sem qualquer anotação só se torna uma interface de negócio caso não haja nenhuma outra especificação feita explicitamente.

Exemplo sem interface 1: caso básico do bean sem interface

```
@Stateless  
public class ServicoPagamentoBean {...}
```

Exemplo sem interface 2: usando `@LocalBean` para que as interfaces implementadas não sejam automaticamente consideradas interfaces de negócio locais, como ocorre no Exemplo local 3.1.

```
//interface legada qualquer  
public interface Observavel {...}
```

```
@LocalBean  
@Stateless  
public class ServicoPagamentoBean implements Observavel {...}
```

No caso do bean sem interface, basta que não adicionemos qualquer interface. Isso parece bem óbvio, mas há casos onde queremos um bean sem interface que já implementa uma interface qualquer, que não é de negócio. Nesse caso, para forçarmos que ele seja exposto sem qualquer interface, precisamos usar a anotação `@LocalBean`.

Para fechar o capítulo

Após passarmos pelo capítulo anterior e por este podemos dizer que vimos os tipos de EJBs, e agora quando vemos um parafuso na nossa frente não vamos tentar martelá-lo, pois sabemos que há várias ferramentas que podemos usar.

Nesses dois capítulos não esgotamos o assunto relacionado a cada tipo de EJB, apenas os apresentamos juntos e evidenciamos suas diferenças e semelhanças. Teremos nos próximos capítulos a oportunidade de aprofundarmos um pouco mais nosso conhecimento em cada tipo visto aqui, e a vantagem é que não teremos a barreira do primeiro contato.

Por exemplo, quando formos estudar as transações, partiremos do princípio de que esse primeiro contato já aconteceu, então pode ser bom reler a seção referente ao assunto que vai ser estudado nos capítulos mais específicos. E é claro, mesmo depois de ler o livro todo, este capítulo e o anterior podem ser bons para retornar sempre que precisar de uma lembrada rápida sobre alguns dos assuntos.

Partindo para nosso próximo capítulo, veremos como utilizar a JPA junto com o EJB, e aí sim nosso projeto vai evoluir um pouco mais. Será um capítulo muito interessante mesmo para quem já trabalha com JPA fora do contexto Java EE, pois muita mudança sutil acontece nesse ambiente, e muitas vezes nos frustramos pois fora do Java EE sabíamos resolver um determinado problema, e dentro dele patinamos como se fosse o primeiro contato.

Mas fique tranquilo, tudo é questão de entender a diferença e depois disso fica fácil usar JPA dentro ou fora de um contexto transacional.

CAPÍTULO 6

Integrando EJB e JPA

Muito do que precisamos fazer no dia a dia dos projetos está relacionado com o que veremos neste capítulo. Mesmo quando já estamos habituados a trabalhar com JPA, quando precisamos utilizá-la em conjunto com EJBs surgem algumas dúvidas: como evitar problemas de inicialização *lazy*; ou quando as transações sofrem *commit* ou *rollback*; dentre outras. Passar por esses pontos, e acabar com essas dúvidas, é o objetivo deste e do próximo capítulo.

6.1 Iniciando com a JPA

A JPA é a API padrão de persistência do Java (Java Persistence API), e por mais que este não seja um livro sobre essa API, veremos o mínimo para podermos entender bem como integrar EJBs e JPA. Para saber mais sobre JPA, eu indico o livro *Aplicações Java para a web com JSF e JPA* da Casa do Código, onde eu comento com detalhes como utilizar JPA e como ela funciona; também mostro como fazer o mesmo com JSF, e integrar bem essas duas tecnologias. Mas focando no escopo de estudo deste capítulo, vejamos o modelo a seguir:

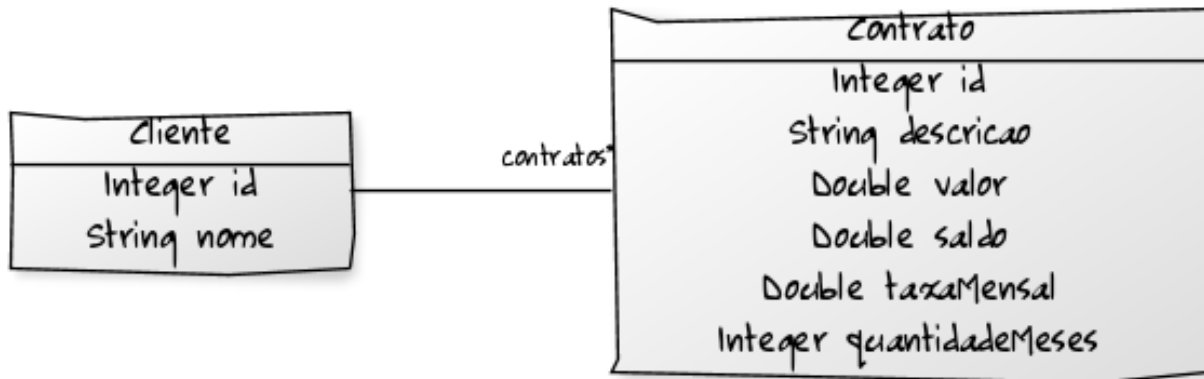


Figura 6.1: Diagrama de classes: Cliente e seus contratos

Apesar de não ser puramente orientado a objetos colocar o atributo `id`, como sabemos que essas classes serão mapeadas para entidades no banco de dados, já deixamos isso claro desde já.

Quando falamos em JPA muitas vezes pensamos na linguagem de pesquisa orientada a objetos e na interface `EntityManager`, mas na verdade o objetivo dela é permitir o mapeamento padronizado do mundo OO para o mundo relacional dos bancos de dados. Apesar de existirem implementações que dão um suporte limitado a bancos não relacionais, o foco aqui é o mapeamento Objeto-Relacional, ou *ORM* (Object-Relational Mapping).

Mapear o mundo OO para o relacional não é algo trivial, pois, apenas como exemplo, enquanto no primeiro sempre teremos objetos completos sendo referenciados, no segundo temos apenas chaves. Por isso já criamos nossas classes com o atributo `id` que fará esse papel no mundo relacional.

Apesar de diversas outras diferenças, com o passar dos anos foram amadurecendo padrões de como superar esse abismo entre os dois mundos, e cada framework fez isso de um jeito. Foi aí que veio a JPA definindo anotações, linguagem de consulta, e API padrão para podermos trabalhar com nossos objetos como se eles estivessem sempre executando no mundo OO, e delegando para a camada de persistência o mapeamento que transformará isso em comandos do mundo relacional.

Em síntese, ao utilizar *ORM* trabalhamos pensando e trafegando somente objetos. Nossos métodos só recebem e devolvem objetos, nunca chaves ou propriedades soltas como acontece no mundo relacional. Se em um projeto OO percebermos "chaves", "códigos" e coisas do tipo nas assinaturas dos nossos métodos, e não se tratar de uma busca pela chave, tem uma chance muito boa de ter algo errado no nosso modelo.

Iniciando nosso exemplo, utilizaremos as anotações da JPA para o Mapeamento Objeto-Relacional, como podemos ver nos trechos de código a seguir.

```
import javax.persistence.*;

@Entity
public class Cliente {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String nome;

    @OneToMany(mappedBy="cliente")
    private Set<Contrato> contratos = new LinkedHashSet<>();

    private boolean preferencial;

    //getters e setters
}

@Entity
public class Contrato {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String descricao;
    private Double valor;
    private Integer quantidadeMeses;
    private Double taxaMensal;
    private Double saldo;
```

```

//construtor padrão
public Contrato() {
}

//construtor que recebe o relacionamento com cliente
public Contrato(Cliente cliente) {
    this.cliente = cliente;
    this.valor = 0.0;
}

@ManyToOne
private Cliente cliente;

//getters e setters
}

```

No mapeamento com a JPA, o básico é termos uma anotação `@Entity` na classe que terá uma representação no banco de dados, e uma propriedade anotada com `@Id`, que será a chave primária. Aqui colocamos também

`@GeneratedValue(strategy=GenerationType.IDENTITY)` para indicar que queremos algo como o `autoincrement` do MySQL ou o `identity` do SQL Server. O bom de uma ferramenta de persistência é que ela faz a tradução para a linguagem específica do banco, então nem precisamos saber como se chama essa funcionalidade no banco que vamos usar. E caso o banco não tenha suporte a algo assim diretamente, será criada uma *sequence* e ligada à nossa chave para que o funcionamento seja o mesmo.

Além das anotações da entidade e da chave, temos também o mapeamento dos relacionamentos com `@ManyToOne` e `@OneToMany`, que são os dois lados de uma mesma relação. Só temos que lembrar que sempre que tivermos relacionamentos bidirecionais precisamos que um dos lados especifique o *mappedBy*.

O lado que tiver essa especificação acaba informando que o outro lado é que o "dono" da relação no banco de dados, ou seja, o outro lado é que terá a chave estrangeira. Em casos como o nosso de

@OneToMany e @ManyToOne sempre teremos o primeiro "passando a bola" para o segundo. Afinal, não temos como guardar chaves estrangeiras para uma lista, já que no modelo relacional não existem coleções. Agora quando tivermos dois @ManyToOne ou dois @OneToOne, precisamos escolher qual lado será o "dono" do relacionamento.

Além de anotar as classes, precisamos do arquivo de configuração persistence.xml que deve ficar dentro da pasta META-INF da raiz do nosso projeto. Como estamos utilizando Maven, e esse não é um artefato Java, o diretório usado será o src/main/resources; então o caminho completo será src/main/resources/META-INF/persistence.xml, e seu conteúdo pode ser igual ao apresentado a seguir. Não se preocupe em digitar essas coisas, peça para sua IDE criar o arquivo e pelo menos a estrutura virá pronta, ou então pegue a versão pronta em <http://github.com/gscordeiro/javacred/javacred>.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  version="2.1">
  <persistence-unit name="javacred">
    <jta-data-source>
      java:/datasources/JavacredDS
    </jta-data-source>
    <properties>

      <property
name="javax.persistence.schema-generation.database.action"
value="drop-and-create" />

      <!-- alternativa ao drop-and-create
<property name="hibernate.hbm2ddl.auto" value="update" />
-->

      <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL55Dialect" />
```

```
<!-- parâmetros para vermos os SQLs no console -->
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
<property name="hibernate.use_sql_comments"
    value="true" />
</properties>
</persistence-unit>
</persistence>
```

Nesse arquivo nós definimos a configuração básica da JPA. Nós especificamos qual *datasource* será utilizado, referenciando seu caminho JNDI. Fizemos isso através da tag `jta-data-souce`, que também já indica que utilizaremos a API de Transações do Java (Java Transaction API), que é quem manipulará as transações em nosso lugar.

Logo em seguida especificamos qual o comportamento da JPA em relação à geração do banco de dados. Como estamos em ambiente de desenvolvimento por enquanto, utilizamos a opção `drop-and-create`, mas temos ainda as opções `create`, `drop` e `none` na JPA.

Abaixo, ainda foi deixada comentada a configuração proprietária do Hibernate, que é a implementação de JPA presente no WildFly, que tem a opção `update`. Essa opção é mais amigável ao desenvolvimento que a `drop-and-create` pois ela não destrói o banco a cada execução, preservando os dados dos nossos testes anteriores.

A configuração seguinte também é específica do Hibernate. Nela nós dizemos qual dialeto deve ser usado para "conversar" com o banco de dados. Por exemplo no caso do MySQL, banco que usaremos, existe mais de um dialeto, devido à compatibilidade com versões anteriores e também com os diferentes *engines* suportados pelo banco. No nosso caso escolhemos utilizar a versão 5 e o *InnoDB* como *engine* do banco.

Por fim, colocamos três propriedades que servem apenas para vermos no console as consultas sendo executadas no banco. Essa funcionalidade deve ser desligada em ambiente de produção, pois gera um gasto a mais para escrever tudo o que vai ser feito a todo instante.

Terminado o arquivo `persistence.xml`, precisamos configurar o *datasource* no servidor de aplicações. Uma forma simples de fazer isso em tempo de desenvolvimento no Wildfly, é através de um arquivo cujo nome termine com `-ds.xml` e que fique na pasta `WEB-INF`, na raiz do contexto web. Como estamos utilizando Maven, essa pasta é a `src/main/webapp`, e o caminho completo do nosso arquivo será `src/main/webapp/WEB-INF/javacred-ds.xml`. O nome `javacred` foi escolhido por ser igual ao da nossa aplicação, mas não há qualquer relação entre as coisas. Para declararmos o *datasource* basta estar na pasta correta e ter o sufixo correto, o nome do meio pouco importa. No nosso exemplo o conteúdo do arquivo ficou assim:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://www.jboss.org/ironjacamar/schema">
  <datasource jndi-name="java:/datasources/JavacredDS"
              pool-name="JavacredPool">
    <connection-url>
      jdbc:mysql://localhost/javacred?createDatabaseIfNotExist=true
    </connection-url>
    <driver>mysql-connector-java-8.0.18-bin.jar</driver>
    <security>
      <user-name>root</user-name>
      <password>root</password>
    </security>
  </datasource>
</datasources>
```

O mais importante a repararmos é a propriedade `jndi-name` da tag `datasource`, onde especificamos o nome JNDI do nosso *datasource*, que está sendo referenciado no arquivo `persistence.xml` que acabamos de ver.

A url de conexão, usuário e senha são propriedades comuns, porém o driver requer nossa atenção. No WildFly o único driver instalado de fábrica é o `h2`. Como vamos utilizar o MySQL, basta jogarmos o jar do driver dentro da pasta `standalone/deployments` do servidor e o driver é disponibilizado para nós, porém com nome igual ao do jar. Isso também é mais comum em tempo de desenvolvimento, mas ao colocarmos em produção, nem o arquivo `-ds.xml` nem o deploy simplificado do driver costumam ser utilizados.

CRIANDO DATASOURCES EM PRODUÇÃO

Essa forma simplificada que vimos só serve para desenvolvimento, e mesmo assim em alguma versão futura do Wildfly ela não estará mais disponível. Então precisamos também saber como configurar nosso datasource da forma "tradicional".

Para isso, vamos criar um arquivo chamado `datasource.cli` com o seguinte conteúdo:

```
embed-server --server-config=standalone-full.xml
batch

module add --name=com.mysql \
    --resources=<pasta do download>/mysql-connector-java-5.1.13-
bin.jar \
    --dependencies=javax.api,javax.transaction.api

/subsystem=datasources/jdbc-driver=mysql\
    :add(driver-name=mysql,driver-module-name=com.mysql)

data-source add --jndi-name=java:/datasources/JavacredDS --
name=JavacredPool \
    --connection-url=jdbc:mysql://localhost/javacred?
createDatabaseIfNotExist=true \
    --driver-name=mysql --user-name=root --password=root

run-batch
```

Esse script inicia conectando no servidor que vai estar offline (será subida uma versão embarcada) e especificamos que o arquivo alvo das nossas configurações é o `standalone-full.xml`. E em seguida iniciamos uma configuração em `batch`, que funciona como uma transação, ou faz tudo ou não faz nada.

Logo depois adicionamos o módulo do MySQL no servidor, e aqui temos que colocar o caminho onde baixamos o jar onde está `<pasta do download>`. O próximo passo é configurar o driver,

que chamamos apenas de `mysql`. O driver é a referência em tempo de execução daquele módulo que criamos no passo anterior, tanto que faz uso dele.

E por fim criamos o `datasource` propriamente dito. Agora basta executar o seguinte comando: `./jboss-cli.sh --file=<path>/datasource.cli`. Se estivermos usando Windows, usamos `jboss-cli.bat`. Com essa configuração teremos nosso `datasource` configurado corretamente.

Depois de tudo ajustado, basta executarmos a aplicação no servidor que, ao subir o *datasource*, o banco será automaticamente criado, devido à presença do parâmetro `createDatabaseIfNotExist=true` na url de conexão. Além disso, quando a aplicação subir, a JPA vai gerar automaticamente as tabelas dentro do banco, pois configuramos a propriedade `javax.persistence.schema-generation.database.action` com o valor `drop-and-create`, então se as tabelas já existirem elas serão apagadas, e depois serão criadas novamente.

Olhando o resultado no banco de dados veremos algo assim:

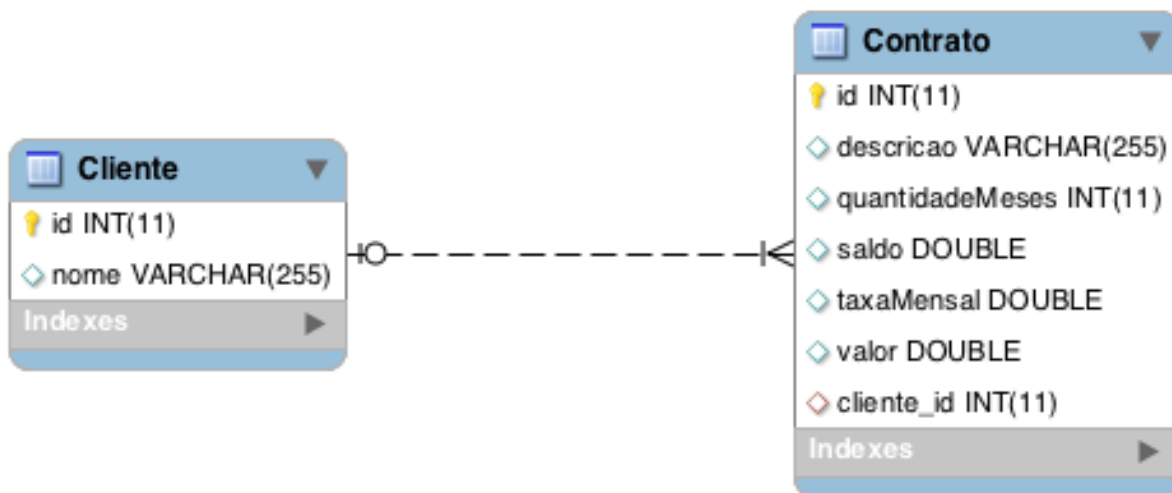


Figura 6.2: Diagrama Entidade-Relacionamento: Cliente e seus contratos

6.2 Primeiras diferenças entre JPA puro e seu uso com EJBs

Tanto ao usarmos JPA puro (Java SE) quanto ao utilizarmos dentro do servidor de aplicações, juntamente com EJBs (Java EE), temos sempre um mesmo ponto de partida, que é a instância de `javax.persistence.EntityManager`. Porém, a forma de conseguir essa instância é bem diferente nos dois casos. Aqui vamos apresentar as duas maneiras, mas no nosso exemplo vamos adicionar apenas a forma *"enterprise"* (Java EE) de fazer, que é a que usamos com EJBs.

```
import javax.persistence.*;

//---- obtendo o EntityManager no Java SE ----

//definido no persistence.xml
String persistenceUnit = "javacred";
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(persistenceUnit);

EntityManager em = emf.createEntityManager();

//---- obtendo o EntityManager no Java EE ----

@PersistenceContext
EntityManager em;
```

Quando utilizamos JPA dentro do Java EE, ou seja, com um servidor de aplicações fazendo boa parte do trabalho por nós, o `EntityManagerFactory` é mantido pelo servidor, e basta utilizarmos a anotação `@PersistenceContext` que uma instância de `EntityManager` é injetada. Enquanto no modo Java SE precisamos do nome da unidade de persistência para criar a factory manualmente, no Java EE só precisamos desse nome se nossa aplicação trabalhar com mais de uma unidade de persistência. Nesse caso podemos

especificar de qual queremos injetar usando a propriedade `unitName` da anotação, por exemplo: `@PersistenceContext(unitName="javacred")` .

Vamos dar início então à nossa classe que vai lidar com a regra de negócio relacionada à entidade `Cliente` . Aqui é importante salientar que o esquema apresentado não é a única forma de implementarmos. Criaremos aqui uma "classe de serviços" para o `Cliente` , mas há quem prefira separar as regras de negócio por caso de uso, e há também a própria documentação do Java EE que coloca os "métodos de negócio" junto com o managed bean que responde aos estímulos da tela (Controlador do MVC).

Como em tudo na vida, temos diversas opções, cada uma com pontos positivos e negativos. Mas o foco agora é mostrar como funcionam os EJBs em conjunto com JPA; logo, em vez de pensar no que é mais adequado, até porque cada time ou projeto vai pedir um estilo, nossa opção será pelo que é mais didático, então criaremos a classe `ClienteBean` com o seguinte código.

```
import javax.ejb.Stateless;
import javax.persistence.*;

@Stateless
public class ClienteBean {

    @PersistenceContext
    private EntityManager em;

    public List<Cliente> listarTodos(){
        return em.createQuery("select c from Cliente c",
            Cliente.class).getResultList();
    }

    public void salvar(Cliente cliente){
        em.persist(cliente);
    }
}
```

Aqui já podemos ver a simplicidade que é usar JPA com EJB. Temos uma Stateless Session Bean com um `EntityManager` que foi injetado com a anotação `@PersistenceContext`. Depois disso é só usar esse `EntityManager`. E o melhor é que, como os EJBs são transacionais por natureza, como veremos com mais detalhes no próximo capítulo, no método `salvar(Cliente)` basta chamar `em.persist(cliente)` e o objeto já é corretamente salvo no banco de dados com gerenciamento automático de transação.

Se não estivéssemos utilizando um EJB, precisaríamos gerenciar a transação manualmente para poder salvar um novo cliente. O código ficaria parecido com o seguinte.

```
import javax.persistence.*;

public class ClienteBean {

    private EntityManager em;

    public ClienteBean() {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("javacred");
        em = emf.createEntityManager();
    }

    public List<Cliente> listarTodos(){
        return em.createQuery("select c from Cliente c",
            Cliente.class).getResultList();
    }

    public void salvar(Cliente cliente){
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            em.persist(cliente);
            tx.commit();
        } catch (Exception e) {
            if(tx.isActive())
                tx.rollback();
        }
    }
}
```

```
}  
}
```

A única ressalva nesse código é que geralmente o `EntityManagerFactory` não é instanciado assim, e sim recuperado de alguma forma global, pois basta uma instância dele para toda a aplicação. Já o restante do código é o que seria necessário fazer manualmente. Isso dentro do bean, porque fora dele ainda precisaríamos de algum mecanismo para fechar o `EntityManager` que foi aberto. Isso será visto quando tratarmos do padrão *OpenEntityManagerInView*.

Como podemos ver, ao utilizar EJBs junto com a JPA, as coisas parecem acontecer de forma "mágica", ou automáticas demais. E este provavelmente é o motivo pelo qual muitos de nós desenvolvem um certo temor em utilizar tal combinação.

É natural confiarmos mais naquilo que conhecemos, e desconfiar do que parece mágica. Mas devemos nos lembrar que quando a fotografia foi inventada também parecia mágica ou feitiçaria para quem não entendia. Aqui estamos falando de um desses avanços que muitos preferem continuar olhando como magia, mas se estamos estudando o assunto é porque queremos fazer parte daqueles que entendem o que acontece atrás das cortinas.

Depois que entendemos, as coisas passam a ficar tão simples, que lidamos com elas como se tivessem sido implementadas por nós mesmos.

6.3 Usar o padrão DAO, Service Bean + Controller, ou tudo junto?

Já comentamos rapidamente que na própria documentação do Java EE temos um único objeto que lida desde com ações de banco de dados como com os estímulos vindos da tela, que costuma ser o

"Controller" do MVC. Particularmente eu evito um pouco essa abordagem porque podemos ficar com nosso objeto de negócio muito tendencioso à nossa tecnologia de visualização. Isso porque na maioria das vezes os *controllers* têm uma relação meio promiscua com a *view*. Mesmo que não tenhamos nenhum *import* da tecnologia de visualização, só o "jeitão" da camada de visualização já pode influenciar.

Por exemplo, se nosso framework de visualização é Action-Based como o Spring MVC, nosso controlador tende a devolver objetos que representam a próxima *view* a ser apresentada e também os objetos que deverão estar disponíveis para a renderização da resposta, como uma instância de `ModelAndView`. E se for Component-Based como o JSF, a tendência é devolver uma `String` que indica a próxima *view*, e os objetos usados na renderização vão estar disponíveis via *Expression Language (EL)*.

Apesar da separação proposta pelo MVC já prever essa possibilidade há mais de uma década, o uso efetivo de diversas visualizações se popularizou com as *views* Web e Mobile nativas, e microsserviços no *back-end*. Logo, vejo a possibilidade de termos mais de um *client* como um motivo suficiente para não contaminar nosso *back-end*, e por isso não uso tudo junto.

Agora, a separação entre Service (como um objeto que guarda a lógica de negócio relacionada àquele objeto ou entidade) e um objeto que só trata da camada de persistência, eu não vejo como necessária hoje em dia. Obviamente essa é uma preferência pessoal, e você deve decidir isso junto com sua equipe. Mas obviamente por ser uma preferência, não significa que não mereça uma explicação.

O padrão DAO (Data Access Object) surgiu em uma época onde tínhamos muito código de acesso a dados, e diferentes implementações para diferentes bancos de dados (SGBDs). Então se nossa aplicação precisava rodar tanto em MySQL quanto em

PostgreSQL, tínhamos diferentes implementações de DAO, uma para cada banco.

Com a JPA, esse problema foi resolvido. Temos até *hints* que podemos colocar nas nossas `Query` de JPA para que ela se comporte melhor em *runtime*. Então a menos que você desenvolver diferentes implementações na camada de persistência, não vejo por que usar DAO.

Outros argumentos são em relação à organização e testabilidade do código. Quanto à primeira, perceberemos com o decorrer do desenvolvimento de diversas aplicações que a implementação de serviços mais simples são em sua grande maioria códigos de acesso a dados. E caso estivéssemos usando um DAO teríamos a ocorrência de um *anti-pattern* que é ter um objeto que só delega, ou seja, nosso serviço só repassa as chamadas para o DAO. Então na prática não temos mais organização, e sim muita delegação pura.

E por fim, quanto à testabilidade, o que temos é que com um objeto separado, podemos mais facilmente trocar a camada real de persistência por objetos que imitam esse comportamento para serem usados em cenários de testes. Isso não deixou de ser verdade, o que mudou é que dificilmente fazemos essa substituição manualmente hoje em dia, e mesmo nos projetos mais simples fazemos uso de frameworks ou bibliotecas que auxiliam a construção desses objetos, como o Mockito.

Com o uso dessas ferramentas, a facilidade de trocar um objeto inteiro de persistência por um que imita esse comportamento se estende a fazer o mesmo com um determinado método, como veremos no capítulo 9, onde trataremos de testes. Então o que permanece é a necessidade de separarmos o acesso a dados em métodos específicos do nosso serviço, mas não precisamos ter um objeto separado com todos esses métodos. Assim evitamos ter aqueles serviços simples que só delegam para o DAO, e ao mesmo tempo quando precisarmos trocar uma consulta no banco pelo retorno de uma lista fixa no nosso teste, faremos isso da mesma

forma como se tivéssemos um DAO. Isso graças à evolução das ferramentas de construção de *mock* da atualidade.

6.4 Usando JSF para construir as telas

Já criamos nosso EJB que vai persistir e buscar os `Clientes`, agora é a vez de criarmos uma tela onde possamos salvar e listá-los. Faremos um uso bem simples de JSF, apenas o suficiente para entendermos algumas particularidades do uso de JPA com EJBs. Vamos iniciar pelo formulário de inserção de `Clientes`, que colocamos na raiz do contexto web e chamamos de `cliente.xhtml`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">

  <f:view transient="true">
    <h:body>
      <h:form>
        <h:panelGrid columns="2">
          Nome <h:inputText
            value="#{clienteController.cliente.nome}"/>
          Preferencial? <h:selectBooleanCheckbox
            value="#{clienteController.cliente.preferencial}"/>
        </h:panelGrid>
        <h:commandButton value="Salvar"
          action="#{clienteController.salvar()}" />
      </h:form>
    </h:body>
  </f:view>
</html>
```


Boa parte do código é declaração do XHTML, o código que interessa mesmo é o compreendido dentro da tag `h:body`. Também não tente digitar tudo isso, crie a página XHTML a partir da sua IDE, usando o completador de código, ou então pegue o código no GitHub. Como podemos ver, nossa tela tem apenas um campo para receber o nome do `cliente` e um botão para salvar. A tela faz uso de um managed bean chamado `clienteController`, cujo código também é simples, e podemos ver a seguir.

```
import javax.faces.view.ViewScoped;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@ViewScoped
public class ClienteController implements Serializable{

    private static final long serialVersionUID = 1L;

    @Inject
    private ClienteBean clienteBean;

    private List<Cliente> clientes;

    private Cliente cliente = new Cliente();

    public List<Cliente> getClientes() {
        if (clientes == null) {
            clientes = clienteBean.listarTodos();
        }
        return clientes;
    }

    public String salvar(){
        clienteBean.salvar(cliente);
        return "clientes";
    }

    public Cliente getCliente() {
        return cliente;
    }
}
```

```
}
```

```
}
```

Agora já é possível executar a aplicação no servidor e acessar o formulário pelo endereço `http://localhost:8080/javacred/cliente.jsf`, e salvar alguns objetos no banco. Vá analisando o console, onde aparecerá a criação do banco e os inserts.

Voltando ao código, usei essa nomenclatura pois tanto os managed beans quanto os EJBs costumam ter o mesmo sufixo "Bean". Então, para deixar bem claro o papel dessa classe, usamos o sufixo "Controller", fazendo referência a essa camada do modelo MVC. O código da classe `ClienteController` é bem simples, mas precisamos nos atentar para alguns pontos importantes. O primeiro está nas anotações da própria classe. Por mais que existam anotações parecidas nos pacotes do JSF, é bom usar as anotações do CDI sempre que possível.

CDI É UMA ESPECIFICAÇÃO DISCRETA QUE FAZ A COLA DENTRO DO JAVA EE

Mesmo quem não é religioso sabe o que é uma bíblia: um livro que na verdade é uma coleção de livros. E esses livros têm suas próprias histórias com início, meio e fim. Alguns livros têm histórias que ocorrem ao mesmo tempo, outros contam histórias que são continuação do anterior, e ainda existem livros com histórias praticamente isoladas. E mesmo sendo um só livro (composto de diversos outros), alguns usam somente parte dele sem nenhum problema, como usar apenas o novo testamento.

Mas por que essa analogia? Porque com o Java EE temos algo parecido. É uma especificação composta de diversas outras, e essas especificações que compõe o Java EE tem seu próprio início, meio e fim. É o caso, só para exemplificar, do JSF e do EJB. Cada um tem seu próprio modelo de "bean", com anotações que criam esses beans e anotações para injetá-los. No entanto a CDI vem para fazer a junção desses mundos que podem viver isolados, mas que para funcionarem bem juntos precisam de alguém "de fora" para fazer o papel de cola. Ao usar as anotações da CDI em vez das do JSF ou do EJB, temos um resultado melhor. Por isso não vamos usar `@ManagedBean` do JSF, e sim `@Named` do CDI. Igualmente, não vamos usar `@EJB` do EJB e sim `@Inject` do CDI.

Como podemos ver na `ClienteController` e no código do formulário, classes anotadas com `@Named` ficam expostas para as expressões `#{}` com o mesmo nome da classe (sem considerar o pacote), porém com a primeira letra em minúsculo.

Outro ponto muito importante é a injeção do nosso EJB via `@Inject` em vez de `@EJB`. Apesar de a segunda alternativa funcionar, a primeira é muito mais versátil, nos dando acesso a funcionalidades mais modernas da CDI, como interceptadores (EJB também tem,

mas é mais limitado) e principalmente a remoção automática do EJB Stateful, como já vimos. Por mais que o nosso EJB nesse exemplo seja Stateless, é sempre uma boa prática utilizar `@Inject`.

ADICIONANDO SUPORTE A CDI NA NOSSA APLICAÇÃO

Para que nossos exemplos funcionem, precisaremos adicionar o suporte a CDI na nossa aplicação. O processo para se fazer isso é simples, basta adicionar um arquivo vazio chamado `beans.xml` na pasta `src/main/webapp/WEB-INF`; é o mesmo lugar onde fica o `web.xml` da aplicação.

Os demais pontos nós vamos analisar conforme passarmos pela próxima tela, que a responsável por fazer a listagem dos `Clientes` e também está na raiz do contexto web com o nome de `clientes.xhtml`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

```
<f:view transient="true">
```

```
    <h:dataTable value="#{clienteController.clientes}"
        var="cliente" border="1">
        <h:column>
            <f:facet name="header">ID</f:facet>
            #{cliente.id}
        </h:column>
        <h:column>
            <f:facet name="header">Nome</f:facet>
            #{cliente.nome}
        </h:column>
        <h:column>
```

```

        <f:facet name="header">Preferencial</f:facet>
        #{cliente.preferencial ? 'Sim' : 'Não'}
    </h:column>
</h:dataTable>

</f:view>

</html>

```

Novamente temos as declarações relacionadas ao XHTML, mas o que nos importa é o código a partir da tag `h:dataTable`. Essa tag é a que nos permite mostrar tabelas, como a listagem de clientes. Ela funciona como se tivéssemos escrito isto: `for(Cliente cliente : clienteController.getClientes())`. Então dentro da tag usamos a variável `cliente` para acessar cada propriedade deste.

Agora podemos analisar a declaração da `h:dataTable` em conjunto com o método `getClientes()` da classe `ClienteController`. Devemos nos atentar para nunca disparar o método de pesquisa dentro de um método que vai na propriedade `value` da `h:dataTable`, pois esse método será chamado uma vez para cada item da tabela. Por esse motivo é que utilizamos uma lista auxiliar, para buscarmos no banco somente na primeira invocação do método. Nunca podemos fazer um método como o `getClientes()` desta forma:

```

...
@Inject private ClienteBean clienteBean;

public List<Cliente> getClientes() {
    return clienteBean.listarTodos();
}
...

```

Pois a menos que tenhamos um cache de consulta ativo, teremos diversas consultas iguais no banco, deixando a aplicação bastante lenta.

Outra coisa que precisamos observar é que, após termos criado a página `"clientes.xhtml"`, ao executar a aplicação e inserir um novo

cliente, o JSF automaticamente passou a exibir a listagem que acabamos de criar.

Para entendermos o porquê disso, voltemos ao código do formulário, e analisemos o `h:commandButton` que chama o método `salvar()`. Já que o método chamado retornou uma `String`, ela é utilizada para buscar a próxima tela a ser exibida. Como no exemplo retornamos `"clientes"`, a tela `"clientes.xhtml"`, que está na mesma pasta da página atual foi chamada. Como antes não era encontrada nenhuma página com o mesmo nome da `String` retornada, o JSF deixava a mesma tela para o usuário. Esse é o princípio básico da regra de navegação do JSF, e funciona por convenção sem qualquer configuração necessária.

6.5 Lidando com `LazyInitializationException` ou equivalente

Agora que nossa listagem de clientes está funcionando, vamos listar também os contratos de cada cliente. Para que possamos fazer isso, teremos que complementar um pouco mais o nosso cadastro, mas nada muito diferente do que já vimos. O primeiro ponto é criarmos o método que adiciona um contrato à lista do cliente:

```
@Named
@ViewScoped
public class ClienteController implements Serializable{

    //todo o resto continua igual

    public void adicionarContrato(){
        cliente.getContratos().add(new Contrato(cliente));
    }

}
```

E agora precisamos adicionar no código da página tanto a chamada para esse método quanto a listagem dos contratos. Para a listagem, vamos usar o mesmo componente `h:dataTable` que usamos para listar os clientes, mas agora listando os contratos do cliente que estamos cadastrando. E como estamos inserindo os dados, e não listando, colocaremos campos de entrada de dados (`h:inputText`) na nossa tabela.

```
<f:view transient="false">
  <h:head/>
  <h:body>
    <h:form>
      <h:panelGrid columns="2">
        Nome <h:inputText
          value="#{clienteController.cliente.nome}"/>
      </h:panelGrid>

      <h:commandButton value="Adicionar Contrato"
        action="#{clienteController.adicionarContrato()}">
        <f:ajax render="contratos"/>
      </h:commandButton/>

      <h:dataTable id="contratos"
        value="#{clienteController.cliente.contratos}"
        var="contrato" border="1">
        <h:column>
          <f:facet name="header">
            Descrição Contrato
          </f:facet>
          <h:inputText value="#{contrato.descricao}"/>
        </h:column>
      </h:dataTable>

      <h:commandButton value="Salvar"
        action="#{clienteController.salvar()}">
      </h:commandButton/>
    </h:form>
  </h:body>
</f:view>
```

Podemos ver que agora há um botão "Adicionar Contrato" , que chama o método que acabamos de colocar no nosso controlador. Logo a seguir podemos ver a tabela que lista o contrato recém-adicionado. Ele aparecerá na tela como um campo em branco, para escrevermos a descrição do contrato. E no final, temos o botão salvar da mesma maneira que tínhamos antes.

Agora, a cada vez que executar a ação "Adicionar Contrato" , o JSF vai processar nosso método no servidor e devolver para a mesma tela, para que possamos adicionar mais contratos ou salvar. No entanto, para que essa ação funcione, temos que mudar o valor da tag `f:view` para que a propriedade `transient` fique igual a `false` . Podemos ver isso na primeira linha da listagem anterior. Na verdade, esse é o valor padrão dessa propriedade, de modo que podemos simplesmente retirar a tag `f:view` , que o funcionamento será o mesmo.

Anteriormente usamos o valor `transient="true"` para que pudéssemos deixar a tela aberta no browser e ir salvando clientes mesmo que, entre um e outro, parássemos e reiniciássemos o servidor. Em resumo, com `transient="true"` o JSF se comporta praticamente igual aos frameworks MVC baseados em ação. Agora com essa propriedade configurada para `false` , precisamos pedir para o JSF gerar novamente a tela toda vez que pararmos o servidor, caso contrário teremos um erro dizendo que aquela tela não pôde ser restaurada:

```
Exception handling request to /javacred/cliente.jsf:
javax.servlet.ServletException: viewId:/cliente.jsf -
    A exibição de /cliente.jsf não pôde ser restaurada.
...
Caused by: javax.faces.application.ViewExpiredException:
    viewId:/cliente.jsf - A exibição de /cliente.jsf
    não pôde ser restaurada.
```

Mas por que precisamos mudar isso, então? Porque agora precisamos que o JSF mantenha o estado da tela entre as requisições. Caso contrário não conseguiríamos adicionar mais que

um contrato, ou então teríamos que manter campos ocultos ou outras formas de controle manual, parecido com o que temos que fazer em alguns frameworks MVC baseados em ação.

A explicação mais detalhada disso é um pouco mais longa, mas para nosso exemplo o que vimos até aqui é o suficiente. Para saber mais detalhes recomendo a leitura do meu livro *Aplicações Java para a web com JSF e JPA*.

Outro pontos que vale nossa atenção é a inclusão da tag `h:head`, que não precisa ter qualquer conteúdo, mas fica ali para o JSF injetar dentro dela os JavaScripts necessários para o funcionamento do ajax, que pode ser visto através da tag `f:ajax` dentro do nosso `h:commandButton`. Como podemos ver, basta dizer a parte da tela que queremos atualizar depois da nossa requisição e o JSF faz o trabalho. Para isso, temos que combinar o conteúdo da propriedade `render` da tag `f:ajax` com o `id` do elemento da tela que queremos atualizar.

Temos também a opção de não enviar todo o formulário quando vamos fazer uma requisição parcial como essa. No nosso caso, o ganho não é tão grande pois nosso formulário só tem mais um campo, mas para fazer isso em casos mais específicos basta utilizarmos a propriedade `execute` também da tag `f:ajax`.

Com tudo certo, agora podemos executar nosso exemplo. Adicione quantos contratos quiser, informando sua descrição, e depois clique em salvar.

A APARÊNCIA DO EXEMPLO E O USO DE UM MODELO SIMPLIFICADO

Ao executar nosso exemplo nesse ponto, vemos que a aparência não está nem perto de algo apresentável ao usuário, e também que estamos usando apenas a `descricao` do `Contrato`, mesmo sabendo que a classe tem mais propriedades.

Não se preocupe com isso agora, vamos focar em entender o funcionamento das ferramentas. Com o passar dos exemplos vamos ter a oportunidade de desenvolver a regra de negócio e melhorar a aparência da aplicação.

Logo após salvar, veremos que o JSF apresentará a listagem das pessoas já cadastradas, mas ainda não estamos listando seus respectivos contratos. Para isso, vamos acrescentar mais uma coluna na tabela da `clientes.xhtml`. O código é o seguinte:

...

```
<h:dataTable value="#{clienteController.clientes}"
    var="cliente" border="1">
    <h:column>
        <f:facet name="header">ID</f:facet>
        #{cliente.id}
    </h:column>
    <h:column>
        <f:facet name="header">Nome</f:facet>
        #{cliente.nome}
    </h:column>
    <h:column>
        <f:facet name="header">Preferencial</f:facet>
        #{cliente.preferencial ? 'Sim' : 'Não'}
    </h:column>
    <h:column>
        <f:facet name="header">Contratos</f:facet>
        <h:dataTable value="#{cliente.contratos}"
            var="contrato" border="1">
            <h:column>
```

```

        <f:facet name="header">ID</f:facet>
        #{contrato.id}
    </h:column>
    <h:column>
        <f:facet name="header">Descrição</f:facet>
        #{contrato.descricao}
    </h:column>
</h:dataTable>
</h:column>
</h:dataTable>
...

```

Agora temos uma listagem dentro de outra e, ao cadastrar um novo Cliente , ou ao executar diretamente a tela `clientes.xhtml` , teremos a seguinte exceção.

```

Error Rendering View[/clientes.xhtml]:
org.hibernate.LazyInitializationException:
    failed to lazily initialize
a collection of role:
    br.com.casadocodigo.javacred.entidades.Cliente.contratos,
    could not initialize proxy - no Session

```

Aqui enfrentamos a conhecida exceção `LazyInitializationException` , ou simplesmente `LIE` . O nome da exceção muda se utilizarmos outra implementação de JPA. No entanto, como o Hibernate - implementação que estamos utilizando - é a mais conhecida, o nome dessa exceção é igualmente difundido.

A saída no console foi formatada de forma diferente para uma melhor visualização, mas é claro que ocorreu uma `LazyInitializationException` , e isso porque não foi possível iniciar a propriedade `contratos` da nossa classe `Cliente` . Isso ocorre pelo fato de o `EntityManager` já estar fechado quando seria necessário para buscar a lista de contratos no banco de dados. A mensagem da exceção no console nos dá essa dica através da expressão `"no Session"` .

Aqui precisamos nos lembrar de que a JPA é uma especificação, e que no Hibernate a interface correspondente a `EntityManager` chama-se `Session`, por isso, na mensagem aparece "no `Session`" em vez de "no `EntityManager`". O importante é que nesse caso podemos considerar que significam a mesma coisa.

Agora que nos deparamos com a `LIE`, temos que entender por que ela ocorre e como lidar com ela. Analisando o próprio nome da exceção, vemos que o problema ocorre quando a JPA tenta inicializar uma propriedade *lazy*, no nosso exemplo, uma lista. Propriedades *lazy* são buscadas no banco de dados somente quando necessário. Isso é bom pois evita carregamento indesejado de objetos, mas em alguns casos podemos ter essa exceção.

Para exemplificar melhor, imagine se a JPA trouxesse do banco de dados todos os objetos relacionados a uma entidade qualquer. Nesse caso se tivéssemos que listar todos os clientes para colocar apenas seus nomes em um *combobox*, teríamos o carregamento involuntário de todos os contratos desses clientes, e como efeito cascata, todos os contratos trariam todas as parcelas, e assim sucessivamente. Dependendo dos objetos trazidos, e o quão relacionados eles são, poderíamos trazer quase o banco de dados todo para a memória! Logo, o carregamento *lazy* é nosso aliado, só precisamos entendê-lo e tirar proveito dele.

Nosso desafio nesta seção não é apenas resolver a questão do carregamento *lazy* como podemos fazer em uma aplicação web comum. Aqui estamos usando EJBs, e isso muda um pouco a forma de trabalhar; e é nesse ponto que algumas pessoas se enroscam.

Quando usamos JPA em aplicações web, há um padrão extremamente conhecido chamado *Open EntityManager In View*, que nada mais é que um nome novo para o padrão *Open Session In View*, que já era utilizado no Hibernate antes mesmo do surgimento da JPA (pois como já sabemos, no Hibernate o objeto equivalente ao `EntityManager` é a `Session`).

Quando utilizamos EJBs juntamente com o contexto de persistência, precisamos entender mais sobre o ciclo de vida do `EntityManager`. Vamos começar vendo como resolver esse problema em um ambiente onde não usamos EJBs, uma vez que as coisas se tornam mais explícitas por termos que escrever todas as linhas de código; e depois então veremos a resolução no ambiente EJB. Para isso vamos relembrar como está nossa classe `Cliente`.

```
import javax.persistence.*;

@Entity
public class Cliente {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String nome;

    @OneToMany(mappedBy="cliente")
    private Set<Contrato> contratos;

    private boolean preferencial;

    //getters e setters
}
```

O relacionamento de `Cliente` com `Contrato` é através do atributo `contratos`, que é uma lista. É um relacionamento um-para-muitos. O `mappedBy` serve apenas para dizer que é um relacionamento bidirecional, e que `Contrato` terá um atributo `cliente`, que é o que faz o relacionamento bidirecional acontecer. Aqui está implícito que o carregamento dos `contratos` é *lazy*, pois sempre que tivermos uma lista, seja no um-para-muitos ou no muitos-para-muitos, por padrão o relacionamento será *lazy*.

Agora que revimos e entendemos melhor o que é um relacionamento *lazy*, vamos analisar como faríamos uma consulta padrão, sem EJBs, e sem usar o padrão *Open EntityManager In View* (`OEMIV`). Esse e os demais códigos são apenas para mostrar

como as coisas funcionam, não vamos colocá-los no nosso projeto. Quando chegar a hora de voltar a incrementar o código do nosso projeto isso será explicitado.

```
//nesse exemplo ClienteBean não é um EJB
public class ClienteBean {

    public List<Cliente> listarTodos(){

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("javacred");
        EntityManager em = emf.createEntityManager();

        List<Cliente> clientes = em.createQuery(
            "select c from Cliente c", Cliente.class)
            .getResultList();
        em.close();

        return clientes;
    }
    ...
}
```

Novamente, a ideia não é levar esse trecho de código a ferro e fogo, pois geralmente recuperamos o `EntityManager` de algum lugar para não termos que criar um novo a cada método, mas o mais importante é analisarmos que, no caso que estamos estudando, que é o uso da JPA sem EJB e sem o padrão *OEMIV*, acabamos geralmente criando o `EntityManager` (EM) no começo do método, e o destruindo logo após fazer a operação de banco de dados, que no caso é uma consulta. A imagem a seguir ilustra esse caso.

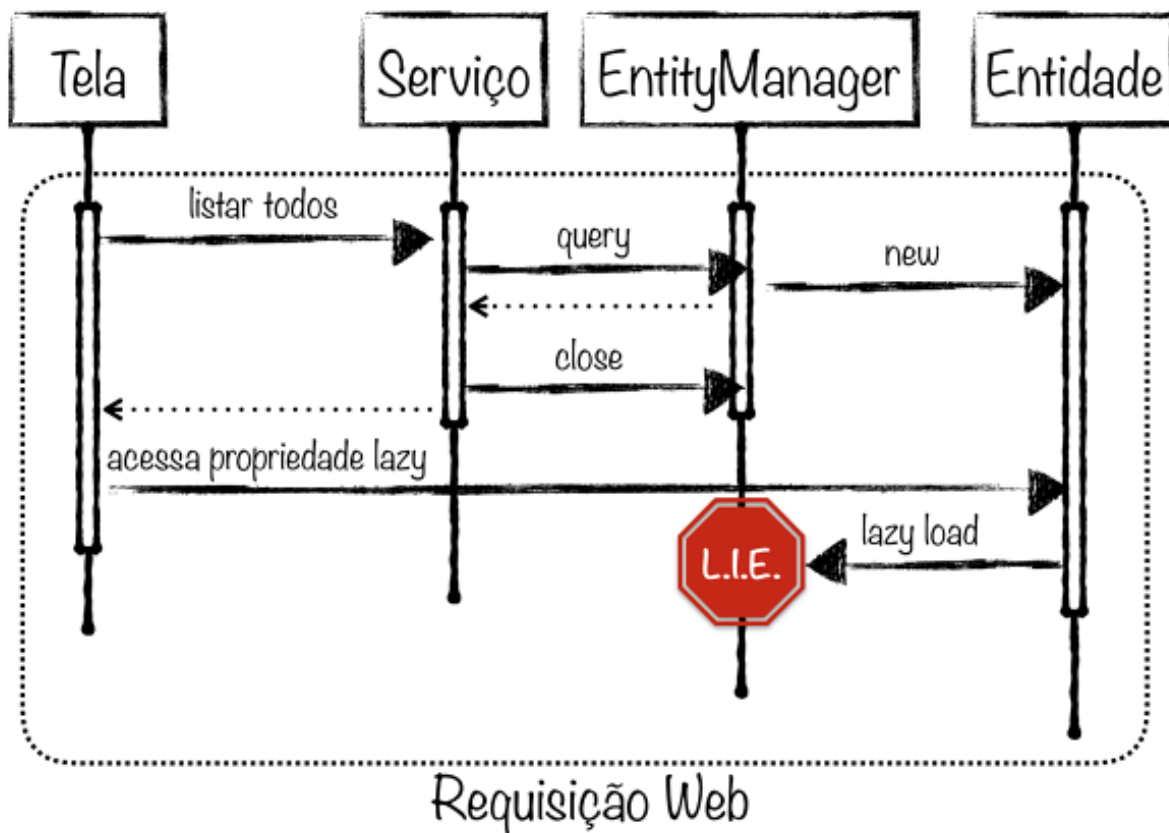


Figura 6.3: Ilustração da ocorrência da LIE sem o uso de EJBs

Fechar o EM é importante porque ele tem dentro de si uma conexão com o banco de dados. Se não fecharmos, a aplicação travará pois em algum momento o banco começará a negar novas conexões.

Nesse ambiente, para reproduzirmos a ocorrência da exceção, não precisamos sequer executar a tela. O código Java na sequência fará o mesmo que a tela faria.

```

ClienteBean clienteBean = new ClienteBean();
List<Cliente> clientes = clienteBean.listarTodos();
//nesse ponto o EntityManager já está fechado
for(Cliente cliente : clientes){

    Set<Contrato> contratosDoCliente = cliente.getContratos();
    //contratosDoCliente é uma lista "virtual", um ponteiro,
    //ainda não carregada do banco
}

```

```
        for(Contrato contrato : contratosDoCliente){ //LIE
            System.out.println(contrato.getDescricao());
        }
    }
    //só aqui poderíamos fechar o EntityManager p/ evitar exceções
```

Nesse código simples, teremos a exceção de inicialização *lazy*, pois após o retorno do `clienteBean.listarTodos()`, o `EntityManager` que fez a pesquisa já foi fechado. Podemos ver isso na penúltima linha do método `listarTodos()` da listagem anterior. Mas o que isso tem a ver?

Os `clientes` retornados não possuem a lista de contratos carregada do banco, pois essa lista é *lazy*, em vez disso possuem apenas uma espécie de ponteiro para o `EntityManager` que trouxe os clientes do banco. Esse ponteiro serve para que quando a lista de contratos for necessária, o `EM` faça uma nova consulta no banco para então retornar esses contratos. O interessante é que como já vimos, o `EM` já foi fechado, e a lista de `contratosDoCliente` só foi usada no `for(Contrato contrato : contratosDoCliente)`. Como essa lista não está carregada, e o objeto que poderia carregá-la já foi fechado, não resta alternativa a não ser lançar uma exceção.

Como funciona o padrão Open EntityManager In View

A implementação desse padrão é bem simples, consiste em utilizar um mesmo `EntityManager` durante toda uma requisição web, e fechá-lo apenas após a tela ter sido totalmente renderizada. Voltando à listagem anterior, o equivalente seria fechar o `EM` após o `for(Contrato contrato : contratosDoCliente)`, que é quando acaba a utilização da lista *lazy*.

O esquema básico da solução é o que podemos ver na imagem a seguir.

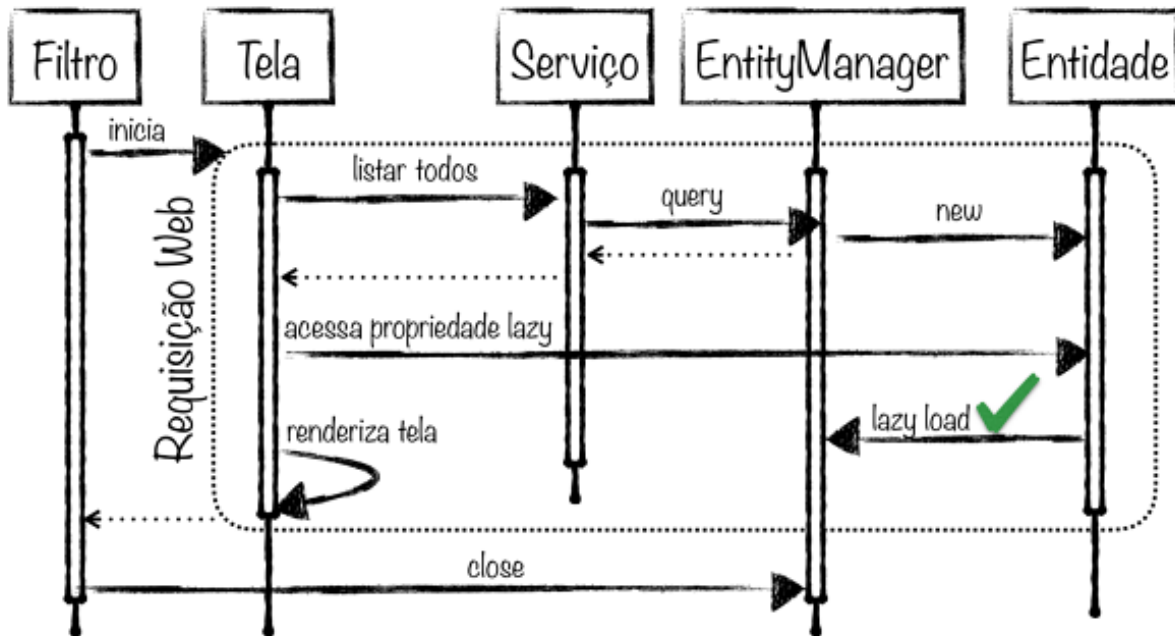


Figura 6.4: Esquema do padrão Open EntityManager In View (OEMIV)

Agora vejamos como seria a implementação desse padrão. De forma resumida, para implementar o padrão OEMIV temos uma classe utilitária, por exemplo `JpaUtil`, com um método `getEntityManager()` que devolve sempre a mesma instância quando for chamado dentro da mesma requisição. Para isso utiliza-se um objeto `ThreadLocal`, que é um mapa vinculado à `Thread` corrente. Como em um servidor cada requisição é processada em uma `Thread`, bastava usar o `ThreadLocal` que conseguiríamos recuperar o mesmo objeto independentemente de quem chamasse o `JpaUtil.getEntityManager()`. Uma implementação possível seria assim:

```

public class JpaUtil {

    private static ThreadLocal<EntityManager> threadLocal =
        new ThreadLocal<EntityManager>();

    private static EntityManagerFactory emf;

    public static EntityManager getEntityManager() {
        if (emf == null) {

```

```

        emf = Persistence
            .createEntityManagerFactory("javacred");
    }
    EntityManager m = threadLocal.get();
    if (m == null) {
        m = emf.createEntityManager();
        threadLocal.set(m);
    }
    return m;
}
...
}

```

Agora podemos chamar dezenas de vezes o `getEntityManager()`, dentro de diferentes beans, que sempre a mesma instância seria devolvida. E o melhor é que, por termos sempre o mesmo objeto à mão, podemos fechá-lo quando a requisição terminar, e a forma mais simples de fazer isso é via um filtro. Podemos ter um filtro parecido com o seguinte.

```

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;

@WebFilter(urlPatterns={"/*"})
public class OpenEntityManagerInView implements Filter{

    @Override
    public void doFilter(ServletRequest req,
        ServletResponse resp, FilterChain chain)
        throws IOException, ServletException {

        chain.doFilter(req, resp); //processa a requisição

        //quando tudo já foi feito, pode fechar o EM
        EntityManager em = JpaUtil.getEntityManager();
        em.close();
    }
    ...
}

```

A ideia desse trecho é dar uma visão geral, essa implementação é muito simples e possui algumas falhas, como criar uma instância de `EM` apenas para fechá-la caso nenhuma operação de banco de dados tenha sido realizada durante a requisição. Porém, o que importa é compreender como isso evita a `LazyInitializationException` (`LIE`) ou exceção equivalente caso a implementação de JPA utilizada não seja o Hibernate.

Como fazer ao utilizar EJBs?

Apesar de este padrão funcionar bem, não podemos utilizá-lo quando utilizamos EJBs, pois é o servidor quem decide quando instanciar e quando destruir o `EntityManager`. Como estamos usando um EJB Stateless, tanto o EJB quanto o `EntityManager` são fechados ao final da transação. O funcionamento é como o visto na imagem a seguir.

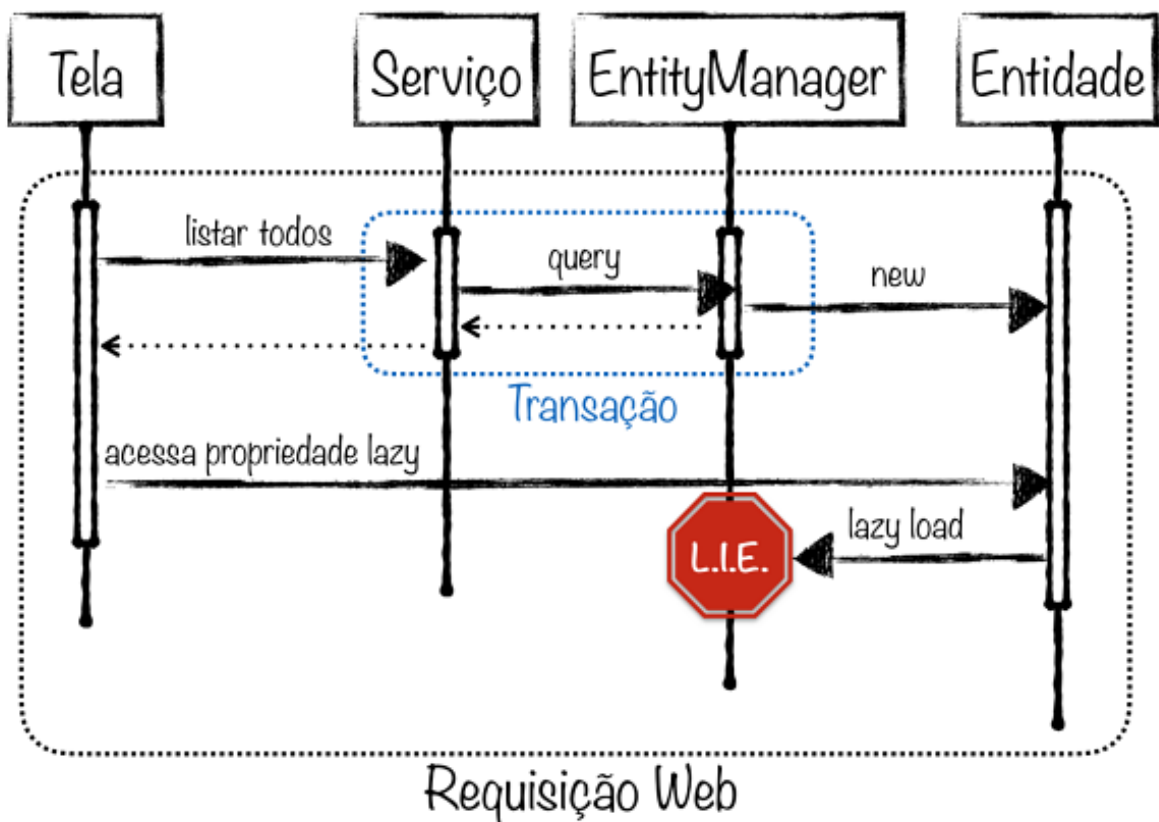


Figura 6.5: Ocorrência da LIE com EJBs Stateless

A diferença é que, em vez de chamar manualmente o método `close()` do `EntityManager`, tanto este quanto o EJB são removidos automaticamente com o fim da transação.

Agora vamos voltar ao nosso exemplo para resolver essa exceção utilizando EJBs. A vantagem agora é que depois de termos visto como resolveríamos manualmente, as coisas ficarão mais simples de entender ao utilizarmos as ferramentas que os EJBs nos dão.

Uma das soluções seria fazer uma consulta um pouco diferente. Então vamos relembrar como está nossa consulta de clientes:

```
@Stateless
public class ClienteBean {

    @PersistenceContext
```

```

private EntityManager em;

public List<Cliente> listarTodos(){
    return em.createQuery("select c from Cliente c",
        Cliente.class).getResultList();
}
...
}

```

Em vez de usar "select c from Cliente c", poderíamos utilizar "select distinct c from Cliente c join fetch c.contratos". O **join fetch** diz que, por mais que o relacionamento seja *lazy*, nessa consulta específica gostaríamos que a lista `c.contratos` fosse trazida do banco junto com o `Contrato` em si.

Essa é uma ótima opção quando temos uma consulta envolvida. Mas como nem sempre a vida é simples, há casos onde poderíamos recuperar o `Cliente` direto pela chave, e então sua lista de contratos continuaria *lazy* e teríamos `LIE` do mesmo jeito. Então qual a outra opção?

Se você pensou em desabilitar o carregamento *lazy* da lista no próprio mapeamento, transformando-a em *eager*, deixe esse pensamento de lado. Muito, mas muito raramente teremos uma situação onde isso realmente será a melhor solução. Como vimos antes, ao desativar o carregamento *lazy*, teríamos o carregamento de todos os contratos de um cliente quando quisermos carregar esse cliente apenas para mostrar seu nome.

Em vez de transformar a lista em *eager*, podemos escrever uma consulta específica como vimos, ou então a opção de utilizar um bean `Stateful` como veremos a seguir.

6.6 Evitando `LazyInitializationException` com EJB `Stateful`

Como já vimos, o padrão *Open EntityManager In View* é amplamente utilizado para resolver essa exceção em aplicações web que não utilizam EJB. A boa notícia é que, com EJBs, essa funcionalidade vem de fábrica, e para isso basta que o `ClienteBean` seja Stateful. Apesar de simples, muitos não utilizam a solução pelo fato de o EJB Stateful precisar ser removido da memória, algo que para desenvolvedores Java é incomum. Além disso há muitos mantras repetidos indiscriminadamente como "nunca use Stateful beans". Mantras são bons para quem não sabe o que faz. Como esse não é nosso caso, vamos ser mais espertos e entender por que essa pode ser a melhor ferramenta para resolver o `LIE` em boa parte dos casos.

MAS USAR STATEFUL SESSION BEANS NÃO DÁ SETE ANOS DE AZAR?

Como acabou de ser dito, temos muitos mantras repetidos dizendo para não usarmos objetos Stateful, que nossos objetos devem estar disponíveis somente uma requisição do usuário para aumentar a escalabilidade da nossa aplicação.

Isso é a mais pura verdade, o que muitas vezes não nos damos conta é que se um bean dura uma requisição, é porque ele é Stateful! Se fosse Stateless não durava nada. Parece estranho, mas veremos isso logo mais.

Já conhecemos bem esse tipo de EJB, e vimos que a partir do Java EE 7, a remoção pode ser automática, basta usarmos CDI para injetar nossos EJBs. Tudo isso pareceu mais abstrato e teórico no capítulo passado, mas agora temos um caso prático disso. Vamos avançando com calma para podermos compreender bem o que ocorre e chegarmos à solução da imagem a seguir.

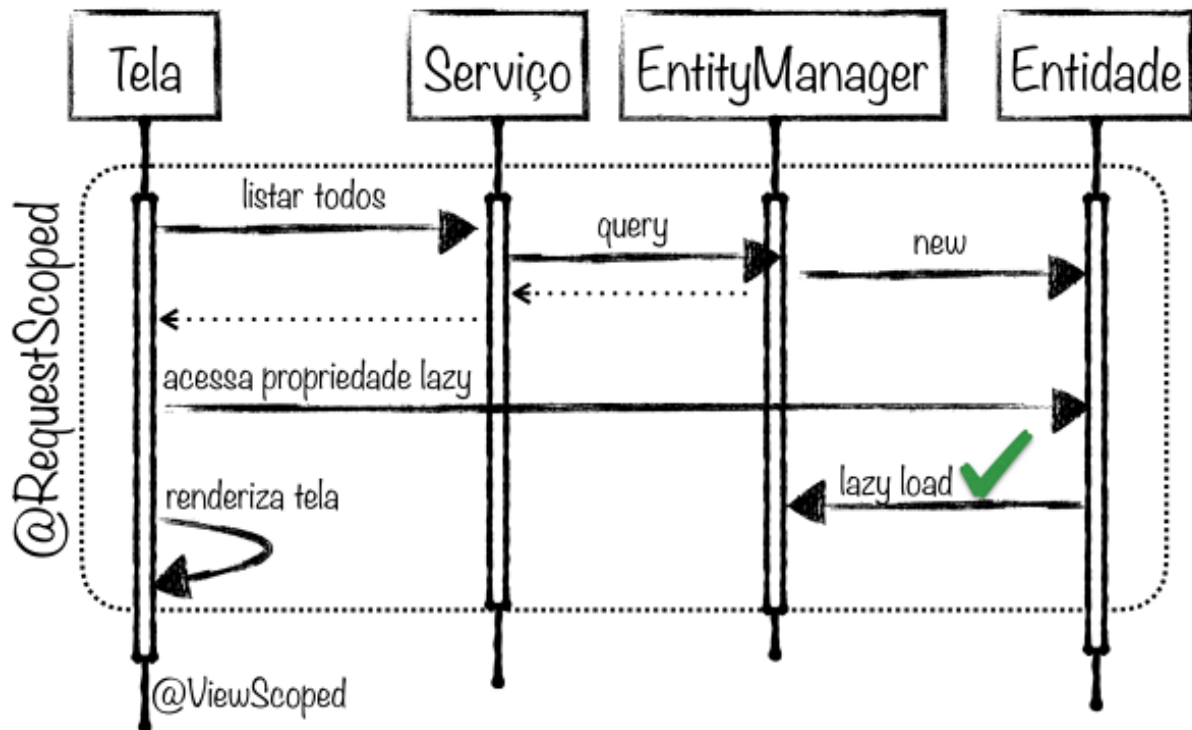


Figura 6.6: Solução da LIE com EJB Stateful com escopo de requisição

Na nossa solução final teremos um EJB Stateful com vida curta, `@RequestScoped`, o que já nos deixa mais tranquilos. Além disso, usaremos nossa tela e controlador com o escopo `@ViewScoped`, mas essa parte é uma sugestão do nosso exemplo, na prática pode ser tudo como `@RequestScoped`.

Agora que vimos a ideia geral, vamos à implementação dessa solução. Como podemos ver a seguir, no nosso `ClienteBean`, injetamos uma instância de `EntityManager`, através da anotação `@PersistenceContext` que é fechada automaticamente pelo servidor.

```

@Stateless
public class ClienteBean {

    @PersistenceContext
    private EntityManager em;
  
```

```
...  
}
```

Até aqui nada novo, isso já havia sido falado antes, mas o que não ficou claro é em que momento esse `EM` é aberto ou fechado.

Quando escrevemos todo o código manualmente isso fica claro, mas quando usamos algo automatizado, não. Quanto tempo um `EM` vive vai depender da propriedade `type` da anotação

`@PersistenceContext`. Como não especificamos nada, o padrão é utilizado, ficando implicitamente dessa forma:

`@PersistenceContext(type=PersistenceContextType.TRANSACTION)`. Lendo isso novamente vemos que por padrão uma instância de `EntityManager` vai viver durante uma única transação.

Agora precisamos lembrar o que vimos no começo deste capítulo, na seção **6.2. Primeiras diferenças entre JPA puro e seu uso com EJBs**, onde no código sem EJB precisávamos controlar as transações manualmente, já no código com EJB não precisamos fazer isso. Isso porque dentro de um EJB, por padrão todos os métodos já executam dentro de uma transação.

Se todo método executa com uma transação, significa que ela é iniciada pelo servidor de aplicação antes de começar o método e é finalizada logo após o final do método, seja com um `commit` ou um `rollback`. Se a transação é finalizada logo após o método terminar, e a `EntityManager` vive somente durante a transação, então temos na prática o mesmo que tínhamos no exemplo onde fechávamos a `EM` antes do `return` do método. Por mais que as coisas sejam feitas automaticamente, entendendo como funciona vemos que é parecido com o que fazemos manualmente.

Naquele caso, quando passamos pela `LIE`, vimos que a solução é manter a instância de `EM` aberta por toda a requisição do usuário. Se olharmos a classe `ClienteController`, vemos que ela está com o escopo de tela (`@ViewScoped`), que é um escopo bom para a tela, mas maior do que o necessário para o nosso EJB. Esse escopo serve para que o estado do controlador se mantenha enquanto estivermos

na mesma tela, ainda que façamos diversas requisições. É exatamente o caso de adicionarmos diversos contratos para nosso cliente. São várias requisições, uma para cada novo contrato, e os dados permanecem porque utilizamos o escopo de tela.

Mas como sabemos, devemos manter nossos objetos vivos no menor escopo possível que resolva nosso problema. No caso do EJB, esse escopo é o de requisição, ou `@RequestScoped`. Então precisamos que o `EM` fique vivo durante esse escopo, mas como?

Antes de mais nada, se quisermos que nosso EJB tenha uma vida maior do que a simples invocação de um método, ele deve ser `Stateful`. A mudança é simples, como podemos ver a seguir.

```
import javax.faces.view.ViewScoped;

@Named
@ViewScoped
public class ClienteController implements Serializable{

    @Inject
    private ClienteBean clienteBean;

    ...
}

import javax.enterprise.context.RequestScoped;

@Stateful
@RequestScoped
public class ClienteBean {

    @PersistenceContext(type=PersistenceContextType.EXTENDED)
    private EntityManager em;

    ...
}
```

Apenas mudando o tipo do EJB para `@Stateful`, faremos com que o `ClienteController` segure o `ClienteBean` vivo durante todo o seu

escopo. Porém, como seu escopo é *view*, se não especificarmos nenhum escopo para o EJB, ele seguirá o escopo de onde ele foi injetado, logo ele também terá escopo de tela. Para especificar um escopo mais apropriado, mudamos também o escopo do nosso EJB para `@RequestScoped`.

Entretanto, isso não é o suficiente para o `EntityManager` também ficar vivo enquanto o EJB onde ele está estiver vivo. Se não mudarmos o tipo do `@PersistenceContext`, o EM continuará vivo somente durante uma transação. Para resolver isso, precisamos mudar o tipo para `type=PersistenceContextType.EXTENDED`. Com essa diretiva injetamos um EM que acompanha a vida do EJB onde ele está declarado, que no nosso caso vive durante uma requisição (`@RequestScoped` da classe `ClienteBean`). Exatamente o mesmo funcionamento que teríamos com o padrão *Open EntityManager In View*.

Como o tipo `EXTENDED` diz ao `EntityManager` que ele deverá viver durante o escopo do seu EJB, só podemos especificar esse tipo em EJBs Stateful. O motivo é simples, os EJBs Stateless não possuem escopo/estado para segurar o EM. Com essa alteração podemos executar novamente nosso exemplo e ver que agora funciona.

E quanto ao EJB Stateful, não corremos o risco de um vazamento de memória por não o removermos? Se estivermos usando EJB 3.2, e injetarmos o EJB com `@Inject` em vez de `@EJB`, como fizemos, não precisamos nos preocupar. Para ter certeza, acrescente o seguinte trecho de código no `ClienteBean`.

```
@Stateful
@RequestScoped
public class ClienteBean {

    ...

    @PreDestroy
    public void removendo(){
        System.out.println("Fique tranquilo, "
            + "sou um EJB Stateful ")
    }
}
```

```
        + "mas já estou sendo removido da memória!");  
    }  
}
```

Agora ao executar novamente o exemplo veremos a saída no console. Isso porque como o EJB foi injetado pelo CDI, e o `ClienteBean` está vinculado a um escopo CDI (`@javax.enterprise.context.RequestScoped`). Logo, quando o escopo for removido - quando a requisição terminar, os EJBs Stateful vinculados a ele serão automaticamente removidos também.

Para fechar o capítulo

A implementação que fizemos neste capítulo foi simples, mais simples do que implementar manualmente um `JpaUtil` que mantém a instância de `EntityManager` na `ThreadLocal`, e ainda um filtro para fechar a `EM` após o processamento da operação do usuário. O que deu trabalho foi entender o ciclo de vida do `EntityManager` em relação à transação ou ao EJB Stateful. Porém é algo que precisamos entender uma única vez.

Desde o capítulo anterior, vimos que os EJBs Stateful são muitas vezes olhados com desconfiança, principalmente pelo fato de antigamente termos que removê-los manualmente. Mas graças à evolução da especificação, na versão atual isso não é mais necessário se utilizarmos CDI para injetar os EJBs. Além disso, não é incomum a confusão entre o termo Stateless com o escopo de requisição; mas como vimos, para que um EJB tenha vida de uma requisição ele precisa ser Stateful.

Pudemos evoluir mais nosso exemplo, explorando a parte de banco de dados com JPA, que é uma das partes mais importante de entendermos. Vimos que os EJBs facilitam muito nosso trabalho com o gerenciamento automático de transações e gestão do ciclo de vida do `EntityManager`. Também vimos como mudar o padrão no que se refere a esse ciclo de vida.

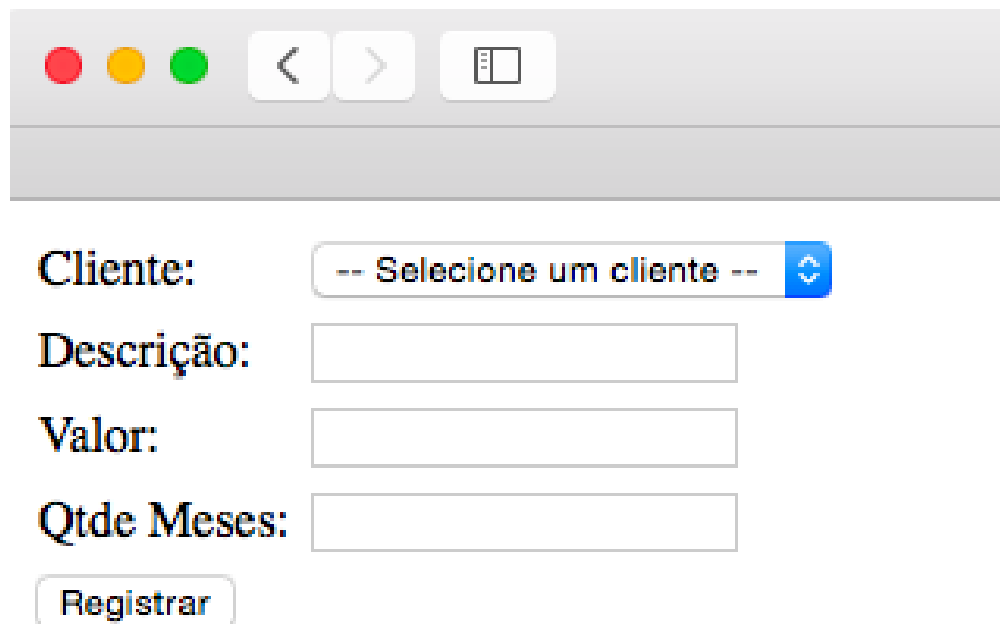
No entanto não nos aprofundamos muito na gestão das transações: quando é feito um *commit* ou um *rollback*? E se não quisermos que um método tenha transação, ou ainda se quisermos abrir uma subtransação em um determinado caso? Isso tudo será matéria de estudo nos próximos capítulos.

CAPÍTULO 7

Gerenciando transações com EJBs

No capítulo anterior nós superamos o que provavelmente é a parte mais difícil do nosso estudo. Compreendemos como funciona o ciclo de vida do `EntityManager` e como gerenciá-lo com os EJBs Stateless e Stateful. Mas neste capítulo refinaremos um pouco mais o controle. Vamos ver como completar ou abortar uma transação (*commit* ou *rollback*), como criar novas transações, não ter transações ou até suspender uma transação em curso.

Como de costume, para vermos esse novo conteúdo precisaremos expandir um pouco mais o nosso projeto. Criaremos agora efetivamente a tela onde solicitaremos um empréstimo. Para isso precisaremos criar algumas classes auxiliares, como conversores JSF. Mas vamos iniciar pela tela de cadastro de contratos, que ficará como essa:



Cliente: -- Selecione um cliente --

Descrição:

Valor:

Qtde Meses:

Figura 7.1: Tela de cadastro de contratos

O código dessa tela está em seguida, mas não se esqueça de que pode conferir ou até copiar os próximos códigos direto do GitHub (<http://github.com/gscordeiro/javacred/javacred>) para não correr o risco de cometer algum erro de digitação.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">

    <f:view transient="true">

        <h:messages/>

        <c:set var="contrato"
            value="#{emprestimoController.contrato}"/>
        <c:set var="clientes"
            value="#{emprestimoController.clientes}"/>

    <h:body>
        <h:form>
            <h:panelGrid columns="2">
                Cliente:
                <h:selectOneMenu value="#{contrato.cliente}">

                    <f:selectItem
                        itemLabel="-- Selecione um cliente --"
                        noSelectionOption="true"/>

                    <f:selectItems value="#{clientes}" var="c"
                        itemValue="#{c}" itemLabel="#{c.nome}"/>

                </h:selectOneMenu>

                Descrição:
                <h:inputText value="#{contrato.descricao}"/>
            </h:panelGrid>
        </h:form>
    </h:body>
</html>
```

```

        Valor:
        <h:inputText value="#{contrato.valor}"/>

        Qtde Meses:
        <h:inputText value="#{contrato.quantidadeMeses}"/>

    </h:panelGrid>

    <h:commandButton value="Registrar"
        action="#{emprestimoController.contratar()}" />
</h:form>
</h:body>
</f:view>
</html>

```

Como podemos ver, temos uma caixa de seleção de usuários, que é feita utilizando a tag `h:selectOneMenu` do JSF. Nessa mesma tag especificamos o conversor que veremos logo mais, e dentro dela colocamos os itens. O primeiro deles é uma linha que não pode ser selecionada e tem a informação para o usuário selecionar um cliente. E na tag `f:selectItems` temos a lista de clientes: `value="#{clientes}"`. Além disso, fizemos uso da tag `c:set` para darmos nomes mais curtos às nossas variáveis, e o código ficar mais legível no livro. Os demais campos são entradas simples como já vimos, e por fim temos o botão `Registrar`. O código do controlador que responde por essa tela é o seguinte:

```

@Named
@RequestScoped
public class EmprestimoController {

    private Contrato contrato;
    private List<Cliente> clientes;

    @Inject
    private ClienteBean clienteBean;

    @PostConstruct
    public void init(){

```

```

        contrato = new Contrato();
        clientes = clienteBean.listarTodos();
    }

    public void contratar(){
        try {

            String msg = String
                .format("Contrato de {0} meses registrado para {1}",
                    contrato.getQuantidadeMeses(),
                    contrato.getCliente());

            System.out.println(msg);
        } catch (Exception e) {
            FacesContext fc = FacesContext.getCurrentInstance();
            FacesMessage msg = new FacesMessage(e.getMessage());
            fc.addMessage(null, msg);
        }
    }

    public Contrato getContrato() {
        return contrato;
    }

    public void setContrato(Contrato contrato) {
        this.contrato = contrato;
    }

    public List<Cliente> getClientes() {
        return clientes;
    }
}

```

Por enquanto, nosso código não faz nada além de escrever no console uma mensagem que usa algumas propriedades do `contrato`. A ideia é apenas mostrar que o objeto já está completo, faltando apenas fazer a regra de negócio correspondente.

Para executar nosso exemplo, vamos criar um conversor do JSF. Um conversor serve para possamos transformar um objeto Java em uma `String` e vice-versa. Apesar de todo objeto Java possuir um método `toString()` que o exibe em forma textual, geralmente a informação devolvida por ele contém dados do objeto que auxilia no processo de debug, mas aqui precisamos de algo mais específico.

Um conversor é usado quando o objeto vai ser inserido no HTML, por exemplo, nos itens de um select (combo box), e aqui o mais comum é retornarmos a chave primária ou alguma outra informação que seja útil para que o objeto seja novamente recuperado. Então o texto devolvido pelo método `getAsString(...)` vai ser usado como entrada para o método `getAsObject(...)` devolver novamente o objeto Java.

Vejamos então como esse conversor pode ser:

```
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;
...

@FacesConverter(forClass = Cliente.class, managed = true)
public class ClienteConverter implements Converter<Cliente> {

    @Inject
    private ClienteBean clienteBean;

    @Override
    public Cliente getAsObject(FacesContext fc,
        UIComponent c, String s){

        if(s == null || s.isEmpty()) return null;

        return clienteBean.buscarPorId(Integer.valueOf(s));
    }

    @Override
    public String getAsString(FacesContext fc,
        UIComponent c, Cliente cliente){
```

```

        if(cliente == null) return null;

        return String.valueOf(cliente.getId());
    }
}

```

Podemos ver que essa conversão foi feita com base no `id` do `Cliente`, e depois, a partir deste, recuperamos o cliente através do `ClienteBean.buscarPorId(id)`.

Aqui utilizamos a versão `managed` do conversor do JSF 2.3 presente no Java EE 8. Mas por razões de compatibilidade, o JSF 2.3 só ativa suas novas funcionalidade que poderiam conflitar com o JSF 2.2 por meio da anotação `@FacesConfig(version = FacesConfig.Version.JSF_2_3)`. Essa anotação recebe a versão que o JSF deverá utilizar para sua compatibilidade, mas por padrão, quando ela está presente, já se entende que vamos usar a versão 2.3, pois se fosse para ficar na 2.2 bastaria não usar essa anotação.

Porém, essa anotação não fica em qualquer lugar, precisa estar em um bean CDI. Para isso vamos criar uma espécie de configurador da nossa aplicação, igual ao exemplo que vimos na seção **4.7. EJB Singleton**. A diferença é que lá vimos um exemplo hipotético de carregamento de uma lista de agências praticamente estática, e aqui vemos o princípio aplicado a um configurador parecido com o que fizemos para o contexto **JAX-RS** (REST) da nossa aplicação.

Criaremos então a seguinte classe.

```

import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.faces.annotation.FacesConfig;

@Singleton
@Startup
//a versão 2.3 é a padrão,
//então podemos ou não deixar explícito

```

```
@FacesConfig //(version = FacesConfig.Version.JSF_2_3)
public class StartupConfig {
}
```

Até poderíamos fazer de outra forma, sem ser um EJB (que também é um bean CDI), mas como vamos precisar de uma classe que execute no momento da subida da aplicação em capítulos posteriores, vamos deixar assim por enquanto.

Um último detalhe: para implementar o conversor criamos um novo método no `ClienteBean` :

```
@Stateful
@RequestScoped
public class ClienteBean {

    ...

    public Cliente buscarPorId(Integer id){
        return em.find(Cliente.class, id);
    }

    //... todo o resto permanece igual

}
```

Agora sim nosso exemplo está pronto. Podemos executar a tela de cadastro de contratos e observar a saída no console do servidor. Esse código não é o fim, mas o meio para estudarmos nossas transações. Poderíamos até ir direto ao ponto, mas ficaríamos sem ter uma forma simples de testar na nossa aplicação. Novamente, lembro que veremos testes automatizados no capítulo 9, até aqui quando falamos de testes nos referimos a uma forma simples de executar nosso código.

7.1 Executando commit em uma transação

Esta seção não é diferente do que temos visto durante todo o livro. Veremos mais uma vez que ao executarmos um método dentro de um EJB, ele automaticamente está dentro de uma transação, e essa transação automaticamente sofre um `commit` após o término do método.

Porém, como nosso projeto está caminhando para algo que nos permita estudar as outras operações com transação, vamos fazer um exemplo de `commit` um pouco mais elaborado. Vamos envolver mais de um EJB em uma mesma transação e então confirmá-la.

O que faremos agora será coordenar duas operações distintas quando o usuário pede um empréstimo. E colocaremos esse código no lugar onde até agora só escrevíamos uma mensagem no console.

As operações em questão são o registro do próprio contrato no banco de dados, através de um novo EJB chamado `ContratoBean`, e a análise desse empréstimo por um serviço de análise financeira.

O código do `ContratoBean` é bastante simples como podemos ver a seguir:

```
@Stateless
public class ContratoBean {

    @PersistenceContext
    private EntityManager em;

    public Contrato salvar(Contrato contrato){
        contrato.setSaldo(contrato.getValor());
        return em.merge(contrato);
    }
}
```

Apenas criamos um EJB simples que salva no banco de dados um `Contrato`. Repare que a operação que usamos para a persistência foi o `merge`, porque usaremos o método `salvar(Contrato)` tanto para inserir quanto para atualizar contratos.

O próximo serviço que precisaremos criar é o `ServicoAnaliseFinanceira` (SAF) que é quem realiza a análise. Por enquanto, a única operação que faremos é o registro do empréstimo:

```
@Stateless
public class ServicoAnaliseFinanceira {

    @PersistenceContext
    private EntityManager em;

    public void analisar(String tomador, Double valor){

        RegistroEmprestimo registro = new RegistroEmprestimo(
            tomador, valor, Tipo.CONTRATACAO);

        em.persist(registro);
    }
}
```

Aqui novamente temos um EJB bem simples que também só salva um valor no banco de dados. A diferença aqui é que esse método vai evoluir para fazer mais coisas, não será um mero salvador de `RegistroEmprestimo` para sempre, tanto que ele nem recebe esse objeto por parâmetro como o `ContratoBean.salvar(Contrato)` faz com o `Contrato`.

Outro detalhe é que, como o `RegistroEmprestimo` funciona como o log, ele nunca será alterado, somente inserido, e por isso usamos o método `persist` da `EntityManager` em vez do `merge` que usamos antes.

Agora falta definirmos a entidade `RegistroEmprestimo`, como podemos ver a seguir:

```
@Entity
public class RegistroEmprestimo {

    public static enum Tipo {CONTRATACAO, QUITACAO}
```

```

@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer id;
private String tomador;
private Double valor;
@Temporal(TemporalType.DATE)
private Date data = new Date();
private Tipo tipo;

public RegistroEmprestimo() {
}

public RegistroEmprestimo(
    String tomador, Double valor, Tipo tipo){
    this.tomador = tomador;
    this.valor = valor;
    this.tipo = tipo;
}

//getters e setters
}

```

Nessa entidade, como em todas as outras, temos os atributos, a chave primária e também definimos um `enum` para classificar o tipo de registro. Vale a pena observar o `@Temporal(TemporalType.DATE)` em cima do nosso atributo do tipo `Date`. Isso serve para que o registro guarde apenas a data, e não a data e hora como é o padrão. Isso será útil conforme nosso código evoluir.

Agora que já temos a entidade e os dois beans, vamos criar um terceiro bean que coordenará o processo de registrar o empréstimo como um todo. Esse EJB é o `EmprestimoBean`, que encapsulará a persistência do `Contrato` no banco de dados e o seu posterior envio para o serviço de análise financeira.

```

@Stateless
public class EmprestimoBean {

    @Inject
    private ContratoBean contratoBean;
}

```

```

@Inject
private ServicoAnaliseFinanceira saf;

public void registrarEmprestimo(Contrato contrato){

    contratoBean.salvar(contrato);

    saf.analisar(contrato.getCliente().getNome(),
        contrato.getValor());
}
}

```

A partir desse método `registrarEmprestimo(Contrato)` vamos ver como podemos mudar o comportamento das nossas transações. Aqui, como dito antes, vamos fazer o *commit* tanto do `salvar` quanto do `analisar`, mas no futuro isso vai mudar.

E agora voltaremos ao controlador `EmprestimoController` para trocar a mensagem que antes somente falava que o empréstimo foi registado pela chamada do método `registrarEmprestimo(Contrato)`, que agora de fato registra o empréstimo.

```

@Named
@RequestScoped
public class EmprestimoController {

    ...

    @Inject
    private EmprestimoBean emprestimoBean;

    public void contratar(){
        try {

            emprestimoBean.registrarEmprestimo(contrato);

        } catch (Exception e) {
            FacesContext fc = FacesContext.getCurrentInstance();
            FacesMessage msg = new FacesMessage(e.getMessage());

```

```

        fc.addMessage(null, msg);
    }
}

...

}

```

Executando o exemplo novamente, temos o cadastro do `Contrato` e do `RegistroEmprestimo` no banco de dados. A imagem a seguir nos mostra o processo como um todo.

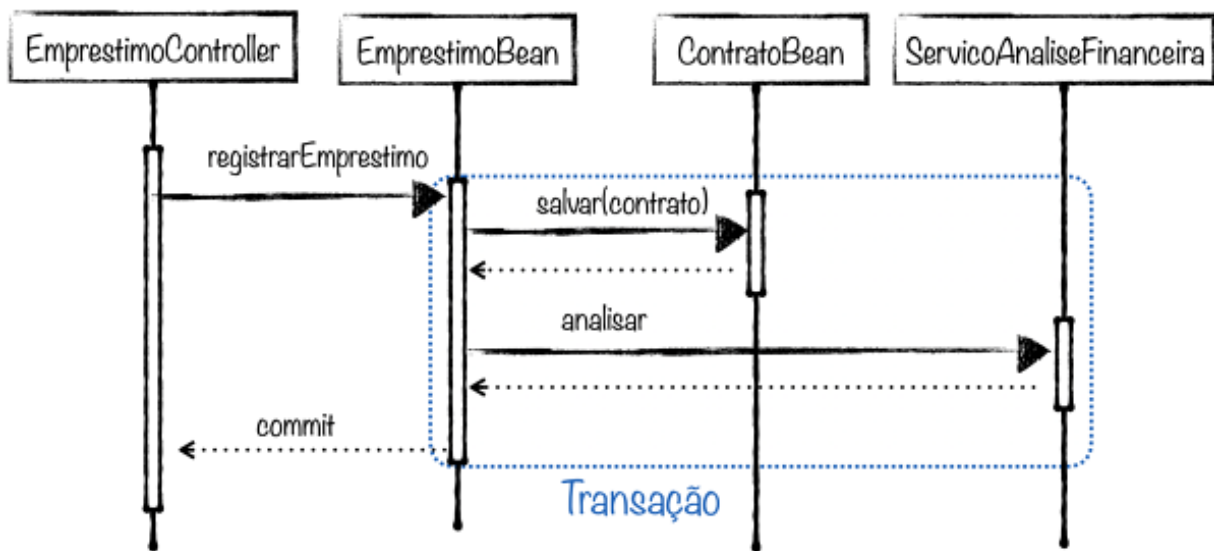


Figura 7.2: Diagrama de sequência da solicitação de empréstimo

Como podemos ver, assim que o método `registrarEmprestimo` do `EmprestimoBean` é invocado, é iniciada uma transação, e como não mudamos o comportamento padrão, todos os outros métodos dos outros EJBs envolvidos estarão inseridos nessa mesma transação. Nesse caso, ou todas as operações são completadas com sucesso, ou nenhuma delas será.

Todo o processo que fizemos foi apenas mais uma forma de vermos uma transação executar com sucesso. Mas como será o caminho do *rollback*? Afinal, nem sempre as coisas saem como esperado, e

pode ser necessário cancelar a operação. E esse é o assunto da próxima seção.

7.2 Executando rollback em uma transação

Quando utilizamos uma forma automatizada de fazer alguma tarefa, como o controle de transações pelos EJBs, temos a sensação de ter perdido o controle, mas sempre há uma forma de fazer o que faríamos manualmente e ainda tirar proveito do trabalho que é feito por nós. Geralmente quando pensamos que não há como fazer algo, simplesmente ainda não descobrimos como. O controle de transações com EJBs se encaixa perfeitamente nessa questão.

Vamos agora fazer uma alteração no nosso exemplo, deixando o `ServicoAnaliseFinanceira` mais criterioso. Agora se o tomador solicitar um segundo empréstimo no mesmo dia, será lançada uma exceção. Para isso, nosso código fica assim:

```
@Stateless
public class ServicoAnaliseFinanceira {

    @PersistenceContext
    private EntityManager em;

    public void analisar(String tomador, Double valor){

        List<RegistroEmprestimo> emprestimosDeHoje =
            buscarEmprestimos(tomador, new Date());

        if(!emprestimosDeHoje.isEmpty()){

            String msg = "Tomador já solicitou empréstimo hoje!";
            throw new JavacredException(msg);

        }
    }
}
```

```

        RegistroEmprestimo emprestimo =
new RegistroEmprestimo(tomador, valor);
        em.persist(emprestimo);

    }

    public List<RegistroEmprestimo>
    buscarEmprestimos(String tomador, Date data){

        String query = "select e from RegistroEmprestimo e "
+ "where e.tomador = :tomador and data = :data";

        return em.createQuery(query, RegistroEmprestimo.class)
            .setParameter("tomador", tomador)
            .setParameter("data", data)
            .getResultList();
    }
}

```

Podemos notar que fizemos uma simples busca com base no tomador do empréstimo, que aqui é feito apenas com seu nome, e também com base na data, que através da anotação

`@Temporal(TemporalType.DATE)` foi configurada para considerar apenas o dia, desconsiderando as horas. Caso essa consulta retorne algum resultado, lançamos a exceção `JavacredException`, cujo código é o seguinte.

```

public class JavacredException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public JavacredException(String msg) {
        super(msg);
    }
}

```

Agora se tentarmos cadastrar um segundo empréstimo para um mesmo tomador no mesmo dia, veremos que a operação não será realizada. É interessante notar que propositalmente colocamos a chamada para persistir o contrato antes do registro no SAF. Isso

significa que, mesmo que a primeira operação tenha sido completada, por estar tudo dentro de uma mesma transação, o *rollback* será completo. Se olharmos no banco de dados logo após tentarmos um segundo empréstimo, veremos que nenhum novo registro foi gerado.

Conseguimos efetuar o *rollback* que queríamos, mas o que exatamente provocou isso? Qualquer exceção que fosse lançada provocaria o mesmo efeito? O segredo está no fato de nossa `JavaCredException` ser uma `RuntimeException`. Logo, toda vez que uma `RuntimeException` for lançada dentro de um método transacional, toda a transação será marcada como *rollback only*. Em outras palavras, ainda que tenhamos uma proteção *try-catch* no nosso código, e consigamos executar diversas linhas de código após o lançamento da exceção, a transação corrente está fadada ao *rollback*.

Outra forma de obter o mesmo resultado seria através do seguinte código:

```
import javax.annotation.Resource;
import javax.ejb.EJBContext;
...

@Stateless
public class ServicoAnaliseFinanceira {

    @PersistenceContext
    private EntityManager em;

    @Resource
    private EJBContext ejbContext;

    public void analisar(String tomador, Double valor){

        List<RegistroEmprestimo> emprestimosDeHoje =
            buscarEmprestimos(tomador, new Date());

        if(!emprestimosDeHoje.isEmpty()){
```

```

        ejbContext.setRollbackOnly();

    }

    RegistroEmprestimo emprestimo =
new RegistroEmprestimo(tomador, valor);

    em.persist(emprestimo);

}

...
}

```

A diferença aqui é que, pelo fato de não lançarmos a exceção, não teremos a mensagem para o usuário avisando que a operação não foi realizada porque o tomador do empréstimo já tinha feito outro na mesma data. Mas se você precisar marcar uma transação para rollback e não quiser lançar uma exceção, essa é a forma de se fazer. No entanto, o mais comum é lançar uma exceção para que o restante da aplicação tenha um feedback de que algo não correu como esperado. Ainda assim, na próxima seção veremos casos onde usar o `setRollbackOnly()` é mais útil.

Se quisermos saber se a transação corrente está marcada para rollback, independente de ter sido pela chamada do método `setRollbackOnly()` ou pelo lançamento de um `RuntimeException`, basta consultarmos o método `getRollbackOnly()` do objeto `EJBContext`.

EXECUTANDO O EXEMPLO VS EXECUTAR TESTES AUTOMATIZADOS

Conforme vamos desenvolvendo nosso exemplo, temos que voltar à nossa tela de cadastro de contratos para testar se a regra de negócio está ficando conforme projetamos. Nesse ponto podem surgir dúvidas se estamos testando certo, ou pela boa prática de programação não deveríamos estar executando testes automatizados para verificar o funcionamento.

Com certeza, a melhor opção é termos um teste automatizado para testar as funcionalidades do sistema em vez de ficar navegando nas telas para reproduzir o comportamento esperado. No entanto, por questões puramente didáticas estudaremos os testes no capítulo 9. Essa separação visa tanto deixar os capítulos mais coesos para uma futura releitura, quando vamos querer lembrar um ou outro ponto específico, quanto evitar que quem ainda não é familiarizado com testes de integração tenha que entender o tema de testes ao mesmo tempo que aprende o funcionamento das transações.

Caso você tenha familiaridade com o tema de testes de integração e prefira executar testes automatizados em vez de voltar à tela conforme avançamos neste capítulo, dê uma espiada no capítulo 9 e pegue de lá os testes de integração necessários para o nosso exemplo. Mas caso prefira continuar a leitura desse capítulo, fique tranquilo que não perderá nada e os testes serão bem vistos no capítulo 9.

7.3 Exceções de sistema e exceções de aplicação

Acabamos de ver que lançar uma `RuntimeException` é a forma mais natural de indicarmos que desejamos descartar a transação

corrente. Mas e se quisermos poder tratar uma exceção via *try-catch* para então decidir se vamos ou não descartá-la? Pode parecer banal a essa altura do campeonato falar de *try-catch*, mas veremos que as vezes as coisas não são tão óbvias.

Para ficar mais claro vamos novamente ao nosso exemplo. Dessa vez vamos fazer um *try-catch* para dar uma segunda chance ao cliente mesmo após a negativa do `ServicoAnaliseFinanceira`.

```
@Stateless
public class EmprestimoBean {

    @Inject
    private ContratoBean contratoBean;

    @Inject
    private ServicoAnaliseFinanceira saf;

    public void registrarEmprestimo(Contrato contrato){

        contratoBean.salvar(contrato);

        try {
            saf.analisar(contrato.getCliente().getNome(),
                contrato.getValor());
        } catch (JavacredException e) {

            if(contrato.getCliente().isPreferencial()){

                System.out.println(
                    "Como é um cliente preferencial, "
                    + "vou ignorar o SAF e manter o contrato!");
            }
            else {
                System.out.println(
                    "Como NÃO é um cliente preferencial, "
                    + "vou observar o SAF e descartar o contrato!");
            }
        }
    }
}
```

```

        throw e;
    }

    }
}

```

Vamos olhar bem o código do método `registrarEmprestimo(contrato)`, pois é onde se encontra a mudança que fizemos. Agora colocamos um *try-catch* em volta da chamada do método `analisar(tomador, valor)` para tentar proteger o restante da aplicação dos efeitos da exceção. Nossa intenção é que o *rollback* ocorra somente se essa for a vontade do método invocador do que lançou a exceção. No nosso exemplo quem escolheria se o *rollback* ia de fato ocorrer seria o método `EmprestimoBean.registrarEmprestimo(contrato)`, e não o `ServicoAnaliseFinanceira.analisar(tomador, valor)`.

O resultado dessa alteração é podermos ignorar a análise do SAF caso o cliente em questão seja um cliente preferencial. Mas será que basta segurar a propagação da exceção para evitar o *rollback*? Vamos executar nosso exemplo, selecionando algum cliente preferencial que tenhamos cadastrado e verificar se conseguimos.

Como podemos perceber, ao cadastrar um segundo empréstimo de um cliente preferencial a aplicação até escreve no console que vai manter o contrato, mas ao olhar no banco, vemos que ele não foi persistido. Isso acontece porque não conseguimos conter o resultado de uma exceção com *try-catch* como imaginávamos. Basta que a exceção seja lançada para que a transação seja marcada como *rollback only*. Mas obviamente se ainda não conseguimos, é porque falta sabermos algo.

Já vimos que a `JavaCredException` é uma `RuntimeException`. Dentro do contexto transacional dos EJBs esse tipo de exceção é considerado uma **exceção de sistema**. Agora se tivéssemos uma *checked exception*, que é aquela que não descende de `RuntimeException` e que o compilador nos obriga a tratar, teríamos uma **exceção de**

aplicação, que é o que precisamos para fazer nosso código funcionar.

Mas antes de seguir ao exemplo, temos que entender o porquê dessa diferenciação. *Runtime exceptions*, ou exceções de sistema, não nos abrigam a tratá-la com um *try-catch*, logo o Java EE considera que provavelmente quando ela acontecer não haverá um tratamento, e para evitar que a aplicação fique em um estado inconsistente, a transação em curso é marcada como *rollback only* imediatamente, e depois que isso é feito não há como desmarcar.

Por outro lado, *checked exceptions* - ou exceções de aplicação - como é chamado pelo Java EE, por padrão nos obrigam a tratá-las. Dessa maneira o raciocínio é simples, se a exceção vai ser tratada pela própria aplicação (já que o compilador obriga), o Java EE considera que o estado da aplicação estará consistente depois de sua ocorrência, e não marca a transação para sofrer *rollback*.

Agora que entendemos a diferenciação, sabemos que precisamos transformar a `JavacredException` em uma *application exception* para conseguirmos assumir o controle sobre o *rollback*. Como já foi dito, a forma principal de se fazer isso é mudar sua classe mãe de `RuntimeException` para `Exception` e consequentemente colocar a cláusula `throws JavacredException` no método

`ServicoAnaliseFinanceira.analisar(tomador, valor)`. Mas em vez disso, vamos utilizar uma forma que acarrete menos alteração no nosso código, que é apenas adicionar a anotação `@ApplicationException` na nossa exceção.

Bastaria marcar a exceção `JavacredException` com essa anotação, mas como é interessante mantermos ambos os comportamentos dentro da nossa aplicação, não alteraremos essa exceção, e sim criaremos uma outra para esse fim.

```
import javax.ejb.ApplicationException;

@ApplicationException
public class JavacredApplicationException
```



```

    extends JavacredException {

        private static final long serialVersionUID = 1L;

        public JavacredApplicationException(String msg) {
            super(msg);
        }
    }
}

```

Bem simples, não? Para transformá-la em uma *application exception* basta anotá-la com `@ApplicationException`. Fizemos a nova exceção como filha da antiga pois assim o *try-catch* que tratava a exceção original não precisará ser alterado. A ideia é alterar o mínimo possível para obtermos nosso resultado. Feito isso, vamos lançar a nova exceção dentro do `ServicoAnaliseFinanceira` e executar novamente nosso exemplo para perceber o que ocorre, mas temos que nos lembrar de fazer o teste com clientes preferenciais e não preferenciais. O código do SAF fica assim:

```

...
public void analisar(String tomador, Double valor){

    ...
    if(!emprestimosDeHoje.isEmpty()){

        String msg = "Tomador já solicitou empréstimo hoje!";
        throw new JavacredApplicationException(msg);

    }

    ...

}
//as partes suprimidas continuam iguais

```

Agora, ao executar o exemplo novamente, percebemos que em ambos os casos o `Contrato` está sendo salvo. Ainda que o cliente não seja preferencial e a aplicação escreva no console que vai acatar o que disse o serviço de análise financeiro e descartar o

contrato, esse contrato é salvo no banco assim como ocorre com o cliente preferencial.

Isso acontece porque não adianta relançarmos a exceção com um `throw` e como fizemos, já que agora essa exceção é inofensiva do ponto de vista do *rollback*. Na verdade esse relançamento da exceção é apenas informativo, para que o resto da aplicação saiba o que ocorreu e possa, por exemplo, mostrar a mensagem na tela do usuário.

Existe uma forma bem simples de resolver isso que é através de uma propriedade *rollback* da própria anotação que transformou nessa exceção de sistema para uma de aplicação, ficando assim: `@ApplicationException(rollback=true)`. Dessa maneira, apesar de ser uma exceção de aplicação, aconteceria um *rollback* sempre que ela ocorresse. Porém, para exercitar mais nossos conhecimentos, vamos utilizar aquela outra forma de marcar uma transação como *rollback only* que vimos na seção passada:

`EJBContext.setRollbackOnly()`. Vamos alterar mais uma vez nosso `EmprestimoBean`, para dessa vez deixar como gostaríamos.

```
@Stateless
public class EmprestimoBean {

    @Inject
    private ContratoBean contratoBean;

    @Inject
    private ServicoAnaliseFinanceira saf;

    @Resource
    private EJBContext ejbContext;

    public void registrarEmprestimo(Contrato contrato){

        contratoBean.salvar(contrato);

        try {
            saf.analisar(contrato.getCliente().getNome()),
```

```

        contrato.getValor());

    } catch (JavacredException e) {

        if(contrato.getCliente().isPreferencial()){

            System.out.println(
                "Como é um cliente preferencial, "
                + "vou ignorar o SAF e manter o contrato!");

        }
        else {
            System.out.println(
                "Como NÃO é um cliente preferencial, "
                + "vou observar o SAF e descartar o contrato!");

            ejbContext.setRollbackOnly();

            throw e;
        }
    }
}
}

```

A diferença aqui foi a injeção do `EJBContext` e a chamada do método `setRollbackOnly()` onde quisemos realmente provocar o *rollback*. Agora sim podemos dizer que nossa missão foi cumprida, conseguimos transferir para o coordenador de todas as transações, que é o `EmprestimoBean` a decisão de quando ocorrerá *rollback*.

7.4 EJB Stateful e o rollback de transações

O assunto que trataremos agora já foi abordado de passagem na seção **4.5. Lidando com Stateful Session Beans removidos e NoSuchEJBException**, mas como foi comentado naquela

oportunidade, somente depois de termos entendido a relação dos EJBs com as exceções é que o assunto ficará mais claro.

Como temos visto neste capítulo, há uma relação entre as exceções e as transações, ficando bem claro isso na distinção de tratamento entre exceções de sistema (*runtime exceptions*) e exceções de aplicação (*checked exceptions*). Vimos que quando aquela é lançada ocorre o *rollback*, diferente do que ocorre quando uma destas é lançada.

Isso, como já dito, é porque se parte do princípio de que, pelo fato de a aplicação ser obrigada a tratar as *checked exceptions*, quando elas ocorrerem o estado da aplicação continuará consistente. Já como não se sabe se haverá ou não tratamento das *runtime exceptions*, o estado da aplicação depois da sua ocorrência é incerto. Mas acabamos de ver isso aplicado para as transações, o que tem isso a ver com os EJBs Stateful?

O princípio é o mesmo, se depois de uma exceção de sistema (*runtime*) a transação pode ficar em estado incerto, o EJB também pode estar com estado inconsistente. Por exemplo, um objeto referenciado pelo EJB pode ter sido atualizado, mas a transação sofreu *rollback*, logo, o valor desse objeto não vai estar consistente com o valor do banco de dados.

Falando com outras palavras, se tanto a transação quanto o EJB pode ficar com estado inconsistente após uma exceção de sistema, ambos serão descartados. O descarte da transação é através do *rollback*, enquanto o do EJB é através da sua remoção. E aí ao tentarmos utilizá-lo novamente teremos a

`javax.ejb.NoSuchEJBException` já estudada na seção 4.5.

Como aqui estamos falando em EJB com estado inconsistente, nem precisamos dizer que isso só se aplica a EJBs Stateful, já que os Stateless não podem ficar com estado inconsistente por não ter estado.

Na referida seção nós vimos como lidar com essa exceção, que é fazendo um `lookup` ou usando `@Inject Instance<TipoDoEJB>`. Então agora vamos nos ater em como fazer o *rollback* de uma transação quando estamos em um *SFSB*, já que não podemos lançar uma exceção de sistema sem sacrificar o EJB junto com a transação.

Testando o rollback via exceção com SFSB

Para exercitar isso que foi dito, vamos voltar ao nosso primeiro exemplo de SFSB, o `NegociacaoBean` que é utilizado pela classe `NegociacaoMain` do projeto `javacred-desktop`. O fato de esse EJB ser também remoto e estar sendo chamado de outra aplicação não muda em nada o que vamos testar. O importante é ele ser *Stateful*.

Relembrando nosso exemplo, tínhamos um método que representava o registro de uma proposta de quitação de débitos. Mas no exemplo visto até então só era negociável o método de pagamento, sendo que cada um já possuía um desconto fixo. Agora vamos criar um novo método que permite o devedor fazer uma proposta de valor para quitar o saldo devedor, além do método de pagamento. Esse método pode ser visto a seguir.

```
@Stateful
@StatefulTimeout(value=30, unit=TimeUnit.SECONDS)
public class NegociacaoBean implements NegociacaoRemota {

    ... resto continua igual...

    @Override
    public void registrarProposta(
        FormaPagamento formaPagamento, double valorProposto){

        double desconto =
            saldoDevedor * formaPagamento.getDesconto();
        double valorComDesconto = saldoDevedor - desconto;
```

```

        if(valorProposto < valorComDesconto * 0.85){
            throw new NegociacaoException(
                "Valor proposto menor que o mínimo");
        }

        int qtdeDeContratos = cliente.getContratos().size();
        propostas.add(new PropostaQuitacao(qtdeDeContratos,
            saldoDevedor, valorProposto, formaPagamento));
    }
}

```

Para que isso funcione precisaremos adicionar a assinatura desse método na interface `NegociacaoRemota`. A própria IDE geralmente oferece a opção de declarar esse método na interface automaticamente. E como essa interface é compartilhada entre os projetos, execute `mvn install` a partir do projeto `javacred` para que fique visível no projeto `javacred-desktop`.

Analisando o código vemos que a lógica consiste em dar até no máximo 15% (quinze por cento) adicional de desconto além do desconto definido no modo de pagamento. Se o valor proposto pelo devedor foi abaixo disso, será lançada uma `NegociacaoException`. A definição dessa exceção por enquanto ficará assim:

```

public class NegociacaoException extends RuntimeException {

    public NegociacaoException(String msg) {
        super(msg);
    }
}

```

E para testar esse código, faremos o seguinte ajuste na classe `NegociacaoMain`:

```

...
negociacao.registrarProposta(FormaPagamento.VISTA);
//código novo começa aqui
negociacao.registrarProposta(FormaPagamento.VISTA, 60_000.0);
try {

```

```

    negociacao.registrarProposta(FormaPagamento.VISTA, 59_000.0);
} catch (Exception e) {
    System.err.println(e.getMessage());
}
//código novo termina aqui
negociacao.registrarProposta(FormaPagamento.ATE_6_VEZES);
negociacao.registrarProposta(FormaPagamento.MAIS_6_VEZES);
...

```

Adicionamos apenas o trecho indicado pelos comentários no exemplo que já tínhamos. Novamente, recordando o exemplo, o total de contratos desse cliente era 100.000,00 (cem mil), e o pagamento a vista dá 30% (trinta por cento) de desconto. Sendo assim, a proposta da primeira chamada do método `registrarProposta` terá um valor proposto de 70.000,00 (setenta mil).

Já na segunda chamada desse método começamos a usar sua nova versão, onde, além de dizer que a proposta é à vista, o cliente oferece um total de 60.000,00 (sessenta mil) para quitar. Como é aceito um desconto extra de até 15% (quinze por cento) em cima do valor com desconto (70.000,00), o valor mínimo aceito seria 59.500,00 (cinquenta e nove mil e quinhentos). Uma vez que o valor proposto foi acima disso, a linha executará sem problemas.

Já a terceira chamada do `registrarProposta` está envolta por um `try-catch`, porque ali ocorrerá a exceção uma vez que a proposta de 59.000 (cinquenta e nove mil) é 500 (quinhentos) abaixo do mínimo aceito. Apesar da exceção que será lançada, como temos um bloco de proteção, em tese as linhas seguintes deveriam executar também. Para ver se é isso mesmo, vamos executar o exemplo e analisar a saída:

```

NegociacaoException: Valor proposto menor que o mínimo
Exception in thread "main" javax.ejb.NoSuchEJBException: ...

```

A saída foi simplificada para ter uma melhor leitura no livro, mas podemos entender o que houve. Após acontecer a exceção que é escrita no console dentro do `catch`, vemos a `NoSuchEJBException`, que já foi estudada antes. Como já sabemos agora, ela ocorreu porque

lançamos uma exceção de sistema (runtime) e com isso tanto a transação quanto o EJB foi descartado. Porém agora já sabemos o que precisamos fazer.

O caminho mais ortodoxo seria transformar nossa exceção em *checked*, removendo a herança de `RuntimeException`. O inconveniente de fazer isso é que temos que mudar a assinatura dos métodos na nossa interface, implementação e onde usamos esse método. Na nossa aplicação real isso pode até ser interessante, pois fica mais explícito. Mas como aqui precisamos mudar o código para comparar o funcionamento, cada mudança ficaria custosa demais.

A opção que seguiremos é novamente anotar nossa exceção com `@ApplicationException`:

```
@ApplicationException
public class NegociacaoException extends RuntimeException {
    ...
}
```

Agora executando novamente nossa `NegociacaoMain` veremos que funciona como esperado. Ao menos à primeira vista:

```
Valor proposto menor que o mínimo
[saldoOriginal=100000.0, valorProposto=70000.0,
 formaPagamento=VISTA]
[saldoOriginal=100000.0, valorProposto=60000.0,
 formaPagamento=VISTA]
[saldoOriginal=100000.0, valorProposto=80000.0,
 formaPagamento=ATE_6_VEZES]
[saldoOriginal=100000.0, valorProposto=90000.0,
 formaPagamento=MAIS_6_VEZES]
```

Novamente a saída no console foi alterada para ficar mais fácil de ler. Percebemos que a exceção ocorreu, mas não impediu que o código continuasse, pois agora é uma exceção de aplicação e por isso o EJB não é removido. Porém, se relembrarmos o que estudamos até aqui, a transação também não é descartada. Como

aqui estamos lançando a exceção antes de efetuar o registro da proposta, vemos que a proposta de 59.000 (cinquenta e nove mil) não consta na lista de propostas, mas isso não quer dizer que a transação tenha sido abortada.

Se analisarmos bem nosso exemplo, vemos que nem existe persistência das propostas em banco de dados. E ainda que tivesse, como a exceção foi lançada antes, a operação de persistência ainda não teria acontecido. Com isso temos a impressão de que o exemplo está perfeito, mas não está. Para conseguir o que queremos precisamos manter o SFSB vivo, porém descartar a transação atual.

Já vimos nas seções anteriores como fazer isso através do método `EJBContext.setRollbackOnly()` , mas aqui vamos usar a outra opção que apenas comentamos antes. Vamos apenas colocar na nossa exceção a informação de que apesar de ser uma exceção de aplicação, queremos que ela marque a transação corrente para *rollback only*. Para isso basta o seguinte ajuste:

```
@ApplicationException(rollback = true)
public class NegociacaoException extends RuntimeException {
    ...
}
```

Com isso temos exatamente o comportamento esperado, descartar a transação, mas não o EJB. Se você quiser praticar mais, um bom exercício extra seria evoluir o exemplo adicionando persistência para as propostas e invertendo a ordem do lançamento da exceção para só ocorrer após a chamada do método

```
EntityManager.persist(proposta) .
```

Para fechar o capítulo

Acredito que este capítulo não tenha sido tão complexo como os três anteriores, até porque eles concentram as partes mais elementares para aprendermos a lidar com os EJBs. Sendo assim, cada detalhe fica muito importante.

Já neste capítulo estamos em velocidade de cruzeiro, então o processo é mais leve. Aqui vimos o comportamento básico das transações quando usamos EJBs. E mais uma vez é importante dizer que esse será o comportamento mesmo que estejamos em um microserviço. Independente do "porte" da aplicação, o modelo é o mesmo no Java EE.

CAPÍTULO 8

Lidando com mais de uma transação

No capítulo anterior vimos como controlar o *rollback* dentro da própria aplicação utilizando uma `ApplicationException`. Agora veremos que podemos ter mais de uma transação acontecendo dentro de um processo como o que ocorre no `EmprestimoBean`. Assim, o cadastro do contrato e a análise de crédito poderiam acontecer cada um dentro da sua própria transação, de modo que o *rollback* de uma não influencie na outra.

No entanto, em vez de mudar o que acabamos de fazer, vamos implementar uma nova funcionalidade que nossa financeira disponibiliza a seus clientes, que é quitar um empréstimo. Com isso teremos em nosso código um exemplo de como controlar a transação via exceção, e outro mostrando como fazer isso com transações independentes.

Nosso ponto de partida será a listagem dos contratos que são gerados a partir do nosso exemplo anterior. Para isso vamos ajustar NOSSO `EmprestimoController` para listar os contratos.

```
@Named
@RequestScoped
public class EmprestimoController {

    private List<Contrato> contratos;

    public List<Contrato> getContratos() {
        if(contratos == null){
            contratos = emprestimoBean.listarContratos();
        }
        return contratos;
    }

    //resto continua igual
```

```
}
```

Fizemos a busca dos contratos dentro do `if` para garantir que a listagem seja carregada somente uma vez, e somente quando for necessária. Não fizemos isso no método `@PostConstruct public void init()`, pois quando fôssemos cadastrar um novo `Contrato`, acabaríamos listando todos os contratos sem necessidade. É o inverso do caso da lista de clientes, que só é utilizada ao cadastrar um novo contrato, mas não quando vamos listá-los.

Fora esse pequeno ajuste, criamos o método `listarContratos()` dentro do `EmprestimoBean`, que por sua vez delega a chamada para o `ContratoBean`.

```
@Stateless
public class EmprestimoBean {

    //resto continua igual

    public List<Contrato> listarContratos(){
        return contratoBean.listarTodos();
    }
}

@Stateless
public class ContratoBean {

    //resto continua igual

    public List<Contrato> listarTodos(){
        return em.createQuery("select c from Contrato c",
            Contrato.class).getResultList();
    }
}
```

A simples delegação de um método para outro, quando utilizado em excesso, se torna uma má prática de projeto. No entanto, como estamos utilizando o `EmprestimoBean` para fazer todas as tarefas referentes aos nossos contratos de empréstimo, escondendo do

controlador detalhes como a existência do `ServicoAnaliseFinanceira`, utilizar essa delegação apenas mantém a linha que já vínhamos seguindo.

Com nossas classes ajustadas, precisaremos construir uma nova tela, que chamaremos de `emprestimos.xhtml`, e seu conteúdo será algo como o seguinte:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">

    <f:view transient="true">

        <h:messages/>

        <h:body>
            <h:form>

                <h:dataTable value="#{emprestimoController.contratos}"
                    var="contrato" border="1">
                    <h:column>
                        <f:facet name="header">ID</f:facet>
                        #{contrato.id}
                    </h:column>
                    <h:column>
                        <f:facet name="header">Cliente</f:facet>
                        #{contrato.cliente.nome}
                    </h:column>
                    <h:column>
                        <f:facet name="header">Descricao</f:facet>
                        #{contrato.descricao}
                    </h:column>
                    <h:column>
                        <f:facet name="header">Valor</f:facet>

```

```

        <h:outputText value="#{contrato.valor}" >
            <f:convertNumber type="currency" />
        </h:outputText>
    </h:column>
    <h:column>
        <f:facet name="header">
            Saldo Devedor
        </f:facet>
        <h:outputText value="#{contrato.saldo}" >
            <f:convertNumber type="currency" />
        </h:outputText>
    </h:column>
    <h:column>
        <f:facet name="header">Qtde. Meses</f:facet>
        #{contrato.quantidadeMeses}
    </h:column>
</h:dataTable>

</h:form>
</h:body>
</f:view>
</html>

```

O código é bem parecido com o utilizado para listar os clientes, logo não tem nada que já não saibamos como funciona. Agora podemos testar nossa listagem abrindo nosso navegador no endereço

<http://localhost:8080/javacred/emprestimos.jsf> .

A listagem funciona, mas ainda não é o nosso objetivo. Agora vamos utilizá-la para disponibilizar a opção de quitar um empréstimo. O processo será praticamente o mesmo utilizado na solicitação. Vamos utilizar tanto o `ContratoBean` , que alterará o status do contrato, quando o `ServicoAnaliseFinanceira` , para registrar a quitação desse `Contrato` . Afinal, não basta saber se alguém adquiriu um empréstimo, é preciso saber se ele está sendo pago.

Para isso, vamos adicionar a opção de quitar um contrato na nossa tela de listagem que acabamos de fazer. Basta acionar a seguinte coluna dentro do `h:dataTable` :

```

<h:column>
    <f:facet name="header">Ações</f:facet>
    <h:commandButton value="Quitar"
        action="#{emprestimoController.quitar(contrato)}"/>
</h:column>

```

Devemos notar que nossa tabela está dentro de um `h:form`. Se não fosse isso, não conseguiríamos ter um `h:commandButton` funcionando. Feito isso, vamos implementar os demais métodos do nosso exemplo, a começar pelo controlador:

```

@Named
@RequestScoped
public class EmprestimoController {

    ...
    public void quitar(Contrato contrato){

        try {

            emprestimoBean.quitar(contrato);

        } catch (Exception e) {
            FacesContext fc = FacesContext.getCurrentInstance();
            FacesMessage msg = new FacesMessage(e.getMessage());
            fc.addMessage(null, msg);

            //limpa a lista para buscar de novo no banco
            //e não parecer que a quitação deu certo quando não deu
            this.contratos = null;
        }
    }
}

```

Esse código é como o `contratar()`, que apenas delega para o `EmprestimoBean` e faz o tratamento de tela, que é adicionar a mensagem para o usuário quando ocorrer exceção e cuidar para que após isso não fiquemos com um valor em memória diferente do

que está no banco. Por isso acrescentamos a linha que limpa a lista da memória, forçando buscar no banco de dados novamente.

Os demais códigos também são bastante simples:

```
@Stateless
public class EmprestimoBean {

    //o resto continua igual

    public void quitar(Contrato contrato) {

        contratoBean.quitar(contrato);

        saf.registrarQuitacao(contrato.getClient().getNome(),
            contrato.getValor());

    }
}
```

```
@Stateless
public class ContratoBean {

    //o resto continua igual

    public void quitar(Contrato contrato) {
        contrato.setSaldo(0.0);
        em.merge(contrato);

    }
}
```

O método `quitar(Contrato)` do `EmprestimoBean` inicia chamando o método de mesmo nome do `ContratoBean`; é tudo bem simples. Agora na implementação da `ServicoAnaliseFinanceira` é que vamos colocar a lógica parecida com a que vimos na análise do crédito do tomador do empréstimo.

Naquela situação, o SAF lançava exceção quando entendia que o usuário não deveria conseguir o empréstimo, e essa lógica nada mais era que olhar se houve outro empréstimo do mesmo tomador no mesmo dia. Agora como trata-se de uma quitação, não teríamos porque negar, então vamos simular algum tipo de falha na comunicação dos serviços. Estando tudo dentro da mesma aplicação parece não ser algo muito real, mas quando tivermos aplicações separadas, isso é algo mais comum de acontecer, principalmente se essas aplicações estiverem em servidores diferentes. Sem mais delongas, vamos ao código.

```
@Stateless
public class ServicoAnaliseFinanceira {

    //o resto continua igual

    public void registrarQuitacao(String tomador, Double valor){

        //aleatoriamente, o sistema vai falhar
        if(new Random().nextBoolean()){
            String msg = "SAF temporariamente indisponível.";
            throw new JavacredException(msg);
        }

        RegistroEmprestimo registro = new RegistroEmprestimo(
            tomador, valor, Tipo.QUITACAO);
        em.persist(registro);
    }
}
```

Novamente, para não complicar o código desnecessariamente, fizemos uma lógica bem simples, pois nosso objetivo é aprender a lidar com transações. Por isso fizemos um serviço de registro de quitação que vai falhar de forma aleatória.

Um ponto importante que devemos observar é o uso da exceção `JavacredException`, que é uma exceção de sistema, e por isso quando

ela é lançada, a transação vai sofrer `rollback`. Logo, ao tentarmos quitar nossos empréstimos através da tela que acabamos de construir, veremos que quando o `SAF` falha, o saldo do contrato não é zerado, pois, como já vimos na seção anterior, por padrão tudo acontece na mesma transação e, falhando um método, todos falham.

Apenas para relembrarmos, a solução que adotamos no capítulo passado foi trocar a exceção por uma `@ApplicationException`, e dessa maneira conseguirmos manualmente controlar quando o lançamento da exceção deveria resultar em um `rollback`. Agora que nosso exemplo mudou vamos utilizar uma abordagem diferente, e faremos isso através do uso de múltiplas transações.

Se observarmos nosso exemplo, o momento onde a operação se divide em duas partes é dentro do método

`EmprestimoBean.quitar(Contrato)`. Ali dentro temos a lógica de efetivamente quitar o `Contrato`, zerando seu saldo, e a de notificar o `SAF`. Porém, como podemos ver, essa segunda tarefa não deveria invalidar a primeira, pois se adotamos a ideia que o registro do empréstimo no `SAF` é opcional, como quando um cliente preferencial faz mais de um empréstimo no mesmo dia, não tem por que a falha desse serviço impedir a quitação do contrato.

Na prática, a forma de implementarmos isso é fazendo o método que queremos "blindar" execute em uma transação independente. Assim, depois que passarmos pela linha que chama

`contratoBean.quitar(contrato)`, essa transação separada sofrerá um *commit*, mesmo que a transação original sofra um *rollback*. A implementação disso é bem simples, basta anotarmos o método que queremos que rode em uma transação própria:

```
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
```

```
@Stateless
public class ContratoBean {
```

```
//o resto continua igual

@Transactional(TransactionalType.REQUIRES_NEW)
public void quitar(Contrato contrato) {
    contrato.setSaldo(0.0);
    em.merge(contrato);
}
}
```

Através da anotação `@Transactional` conseguimos mudar o comportamento padrão das transações. Logo mais veremos as opções disponíveis, mas por agora o que nos importa é que especificando `REQUIRES_NEW` nós informamos que esse método requer uma nova transação para ser executado. Dessa forma dizemos ao servidor de aplicações que queremos justamente o comportamento que descrevemos logo antes: "Servidor, execute esse método em uma nova transação para que se as demais falharem, o resultado deste método não seja afetado".

Agora podemos voltar à nossa tela de listagem e executar a operação de quitar um contrato e observar o resultado tanto na tela quanto no banco de dados. Veremos que mesmo que apareça a mensagem que o Serviço de Análise Financeira (SAF) está indisponível, o contrato continuará sendo quitado. A diferença é que nesse caso não teremos um registro novo na tabela da classe `RegistroEmprestimo`.

Conseguimos agora dominar duas formas distintas de assumir o controle do que vai ser confirmado, e o que será desfeito durante uma ou mais transações. Obviamente, as duas formas não são excludentes, podem ser misturadas livremente conforme a necessidade. Aqui optamos por separá-las em dois exemplos distintos por questões didáticas, e para que tenhamos os dois exemplos como referência para voltarmos e fazer testes quando estivermos lidando com essas questões em uma aplicação real.

Nesta seção vimos apenas a opção `REQUIRES_NEW` da anotação `@TransactionAttribute`, mas na seção seguinte veremos quais outras estão disponíveis.

8.1 As opções de transações com `@TransactionAttribute`

Já vimos como o uso da anotação `@TransactionAttribute` pode modificar o padrão de como um método do nosso EJB se comporta em relação às transações. Essa anotação pode ser utilizada em cada método que quisermos mudar o comportamento, ou então pode ser colocada na própria classe do EJB, caso em que todos seus métodos assumem o comportamento especificado. Mas anotar a classe não nos impede de anotar também os métodos. Por exemplo, se especificarmos `REQUIRES_NEW` no bean inteiro, podemos voltar um ou outro método para o padrão ou ainda especificar um terceiro comportamento.

No exemplo da seção anterior utilizamos a `enum` `TransactionAttributeType` para especificar o comportamento do nosso método. No entanto, vimos apenas dois comportamentos até agora: o padrão, quando não especificamos nada; e a criação forçada de uma nova transação. Agora vamos analisar as outras possibilidades. Para isso, basta analisarmos o código simplificado dessa `enum`:

```
public enum TransactionAttributeType {  
    MANDATORY,  
    NEVER,  
    NOT_SUPPORTED,  
    REQUIRED,  
    REQUIRES_NEW,  
    SUPPORTS  
}
```

Cada entrada dessa `enum` representa um comportamento diferente. Para entendermos melhor, temos sempre que pensar o que deve ser feito com a transação atual, se existir alguma, quando vamos entrar em um novo método.

Todo método tem o comportamento de um desses tipos, mesmo que não especifiquemos nada nele e nem na classe onde ele está declarado. Nesse caso o método terá o comportamento padrão:

`REQUIRED`. Porém, antes de explicar como cada tipo funciona, vamos analisar a dinâmica de como alteramos o tipo padrão de transação de um método.

Dinâmica básica da anotação `@TransactionAttribute`

Independente do tipo de comportamento que queremos atribuir, podemos definir que a transação será desse tipo anotando um método específico, ou então anotando o EJB inteiro. Nesse último caso, todos os métodos terão o mesmo comportamento até que algum especifique algo diferente.

Para ficar mais claro, veremos um trecho de código que não faz parte do nosso projeto, mas vai servir para entendermos a dinâmica para todos os tipos de transações.

```
@Stateless
public class Bean1 {

    @TransactionAttribute(TransactionAttributeType.MANDATORY)
    public void metodo1() {
        ...
    }

    public void metodo2() {
        ...
    }
}
```

Nesse caso, o `metodo1()` especifica o comportamento `MANDATORY`, enquanto o `metodo2()` especifica o padrão: `REQUIRED`. Usamos `MANDATORY` aqui apenas porque na ordem alfabética ele é o primeiro, mas a forma de atribuição é a mesma para todos os tipos.

Vejamos agora mais um exemplo para auxiliar nossa compreensão:

```
@Stateless
@Transactional(TransactionAttributeType.MANDATORY)
public class Bean2 {

    public void metodoA() {
        ...
    }

    public void metodoB() {
        ...
    }

    @Transactional(TransactionAttributeType.NEVER)
    public void metodoC() {
        ...
    }

    @Transactional(TransactionAttributeType.REQUIRED)
    public void metodoD() {
        ...
    }
}
```

Nesse novo exemplo, anotamos o bean inteiro com `MANDATORY`, logo, todos os seus métodos automaticamente serão desse tipo a menos que especifiquem outro comportamento. Então os métodos `metodoA()` e `metodoB()` serão `MANDATORY`; enquanto `metodoC()` e `metodoD()` serão `NEVER` e `REQUIRED` respectivamente.

É importante notarmos que anotando o bean, mudamos o comportamento padrão de todos os seus métodos. Então no exemplo do `Bean2`, quando quisemos o tipo `REQUIRED`, tivemos que especificá-lo explicitamente.

Apenas reforçando, nesses trechos usamos tipos quaisquer apenas para exemplificar. Com exceção do tipo `REQUIRED` que é o padrão quando não especificamos nada no bean, todos os demais tipos estão aí como exemplo, a mudança de um tipo pelo outro não altera a dinâmica. Com isso bem claro, podemos passar ao estudo específico de cada comportamento.

MANDATORY

Quando especificamos esse comportamento no nosso método, estamos dizendo que é obrigatória a presença de uma transação para executar nosso método, mas que se ela não vier pronta, a exceção `javax.transaction.TransactionRequiredException` será lançada.

Isso pode ser utilizado se quisermos um controle mais refinado de quem vai invocar nosso método. Se por exemplo estivermos trabalhando usando MVC, onde a camada controladora seria por exemplo um *managed bean* JSF ou CDI, e nossa regra de negócio fosse um EJB, poderíamos com a transação do tipo `MANDATORY` especificar que nosso método não deve ser chamado diretamente pela camada de controle, e sim por outro EJB.

Isso porque a camada de controle, seja JSF ou CDI, normalmente não é transacional, deixando a criação da transação para o EJB que será chamado. Dessa maneira se algum controlador tentar chamar diretamente nosso método de negócio, a exceção acima será lançada.

Outra forma de fazer esse controle específico seria deixar o método de negócio com visibilidade de pacote em vez de público, mas aí todos nossos EJBs precisariam estar no mesmo pacote, algo que não costuma ser verdade em uma aplicação maior onde temos módulos com pacotes próprios. Nesse caso, o uso de `@TransactionAttribute(TransactionAttributeType.MANDATORY)` pode ser uma boa alternativa.

NEVER

Métodos com esse comportamento formam um par com os `MANDATORY`, pois são os dois tipos que nos obrigam a seguir o comportamento especificado ou lançam uma exceção caso contrário. Isso quer dizer que em um método com `@TransactionAttribute(TransactionAttributeType.NEVER)` nunca podemos ter uma transação em curso ou a exceção `java.rmi.RemoteException` será lançada. O mais comum, no entanto, quando não queremos uma transação, é o tipo que veremos em seguida: `NOT_SUPPORTED`.

NOT_SUPPORTED

Ao definirmos que o nosso método não suporta transações usando `NOT_SUPPORTED`, se quem estiver chamando o método de negócio for um controlador que não é transacional, simplesmente nada será feito, e o método do EJB executará sem transação. Até aqui é igual ao `NEVER`. Porém, se já houver uma transação em curso e o método com `NOT_SUPPORTED` for chamado, em vez de lançar exceção, simplesmente a transação em curso será suspensa, e nosso método executará fora da transação. Dessa forma, assim como o `NEVER` temos a segurança de executar fora de uma transação, a diferença é que o comportamento do `NOT_SUPPORTED` é mais amigável.

Um dos motivos para se usar um método fora de transação é se nele nós modificamos uma entidade da JPA que esteja - ou que possa estar - gerenciada, mas que não desejamos persistir a alteração no banco.

Mas essa certamente não é a melhor forma de fazer isso, já que o objeto modificado pode continuar transitando durante a requisição até chegar em um método transacional, e aí a alteração será persistida. Se a ideia é modificar o objeto sem salvar, o melhor é usar o método `EntityManager.detach(entity)` passando o objeto que queremos modificar. Isso porém também gera efeitos colaterais, pois outros métodos transacionais podem contar que aquele objeto está gerenciado, e aí alterações que precisariam ir para o banco precisariam explicitamente do método `EntityManager.merge(entity)` para isso acontecer. Uma alternativa ao método `detach` seria clonar

o objeto e modificar apenas o clone não gerenciado. Isso pode ser útil se utilizarmos consultas baseadas em exemplos do Hibernate. Apesar de esse ser um uso possível, já é algo bem específico.

REQUIRED

O comportamento especificado por `REQUIRED` é o padrão. Assim, como vimos nos exemplos anteriores, se não especificarmos nada diferente, este será o comportamento do nosso método. Esse comportamento é a versão mais amigável do `MANDATORY`, pois enquanto aquele lança uma exceção se não houver transação em curso, este apenas a criará.

Até aqui vimos dois pares de comportamento, que podem ser representados na matriz a seguir.

	Lança Exceção	Sem Exceção
Com Transação	MANDATORY	REQUIRED
Sem Transação	NEVER	NOT_SUPPORTED

Figura 8.1: Matriz de transações (com transação X sem transação - com exceção X sem exceção)

Como podemos ver, na primeira coluna temos os tipos que lançam exceção se a transação não chegar como esperado, e na segunda temos os tipos que simplesmente resultarão no resultado esperado, sem lançar exceção. Olhando as linhas, temos na primeira os tipos que executam nosso método em um contexto transacional, e na segunda os que executam nosso método fora de uma transação.

Agora veremos mais dois tipos que saem um pouco desse modelo de pares.

REQUIRES_NEW

O comportamento do `REQUIRES_NEW` se parece com o `REQUIRED`, pois ele não vai lançar exceção caso ainda não haja uma transação em curso, e também vai fazer com que nosso método execute em um contexto transacional. A diferença é que o `REQUIRED` aproveita a transação em curso caso exista, já como o próprio nome do `REQUIRES_NEW` sugere, ele criará uma nova transação independente para o nosso método.

A aplicação para esse tipo de comportamento é o mesmo para quando desejamos uma subtransação, mas aqui não há essa hierarquia; a nova transação é independente da primeira. Isso nos permite desfazer uma transação (*rollback*) sem interferir na outra. É o comportamento que vimos no início deste capítulo, onde queremos ter transações independentes.

SUPPORTS

Utilizando este tipo de comportamento, dizemos que nosso código é indiferente ao contexto transacional: se não houver transação, o método executará sem transação; se houver uma, nosso método executará no contexto dela. É o único tipo que não faz qualquer alteração no contexto transacional em curso, o que vier, está bom para nosso código.

Então agora podemos resumir o comportamento de cada tipo de configuração de transação na seguinte tabela:

@TransactionAttribute	Tx do cliente	Tx resultante
MANDATORY	sem tx	Exception
MANDATORY	tx 1	tx 1
NEVER	sem tx	sem tx
NEVER	tx 1	Exception
NOT_SUPPORTED	sem tx	sem tx
NOT_SUPPORTED	tx 1	sem tx
REQUIRED	sem tx	tx 1
REQUIRED	tx 1	tx 1
REQUIRES_NEW	sem tx	tx 1
REQUIRES_NEW	tx 1	tx 2
SUPPORTS	sem tx	sem tx
SUPPORTS	tx 1	tx 1

Figura 8.2: Comportamento de cada combinação de @TransactionAttribute

Essa imagem é uma adaptação de uma tabela encontrada na documentação do Java EE, e nela podemos ter um resumo visual do que vimos nos títulos anteriores. Na primeira coluna temos a configuração de @TransactionAttribute, na segunda temos a informação se já existia ou não uma transação em curso antes de chegar ao nosso EJB, e na terceira coluna temos o comportamento resultante dentro do nosso EJB: vai executar dentro de transação ou fora? Vai lançar exceção? A transação que vai estar em execução é a mesma que veio do cliente ou é uma nova?

Para fechar o capítulo

Como em cada capítulo que passamos, neste aprendemos mais sobre como resolver questões que podem aparecer no nosso dia a dia, ainda que com menor frequência. É praticamente certo que teremos nas nossas aplicações muito mais cenários onde

precisaremos de uma única transação executando com o comportamento padrão do que cenários que precisam de transações coordenadas de alguma forma. A diferença é que agora estamos preparados para quando essa necessidade surgir.

No próximo capítulo vamos conhecer o Arquillian, que é uma ferramenta de testes de integração que nos permite testar de forma automatizada o comportamento de tudo que vimos até aqui.

Poderemos por exemplo testar dentro do servidor, e sem ter que navegar na tela para isso, se no caso onde a transação deveria sofrer um *rollback*, isso realmente está acontecendo: checando a chamada do método de persistência e ainda assim ver que nada foi para o banco de dados. Com o Arquillian veremos que isso é possível.

CAPÍTULO 9

Testando nossos serviços

Sabemos que precisamos testar o que desenvolvemos, o problema é que muitas vezes esses testes são feitos navegando na tela, clicando nas coisas e assim por diante. Com isso temos testes que muitas vezes não seguem uma fórmula, e podemos deixar passar coisas importantes. Então não basta testar, temos que usar testes automatizados.

Apesar de termos os testes de unidade, que devem testar uma classe de forma isolada, e os de integração onde vamos testar a comunicação entre as classes, vamos iniciar já no segundo caso. O motivo para não abordarmos primeiramente o teste de unidade propriamente dito é que a diferença entre este e o de integração é semântica, e não técnica, pois em ambos os casos a ferramentas e técnicas utilizadas serão as mesmas, não havendo qualquer prejuízo para nosso aprendizado avançarmos um pouco mais em direção ao que precisamos construir já no início.

Ainda assim, dentro do contexto de testes de integração podemos criar uma subdivisão apenas para padronizar nossa comunicação, onde teremos os testes de integração "puro", que visa testar somente a integração das nossas classes entre si, e o teste de integração com o contêiner, que realmente instala parte de nossa aplicação no servidor e executa o teste lá dentro. Esse teste é diferente do teste de sistema pois, nesse último, testamos o todo, estilo caixa preta. Então nesse tipo de testes usaríamos inclusive os de interface gráfica para automatizar o teste que seria feito pelo usuário.

Por uma questão lógica, vamos iniciar pelo mais simples, que é o teste de integração das nossas próprias classes.

9.1 Criando testes de integração com Mockito

Uma parte fundamental em qualquer tipo de testes é ter um cenário definido para testarmos. Como em um teste de integração temos várias classes envolvidas, pode ser um pouco mais complexo configurar um cenário onde cada classe tem que se comportar de uma determinada maneira. Nesses casos é muito útil utilizarmos *mocks*, que são classes que se comportam de uma maneira predefinida.

No nosso exemplo usaremos o Mockito (<http://mockito.org>), que é um framework para construção de *mocks* bem fácil de usar. Para isso precisaremos adicioná-lo (juntamente com uma dependência dele) no `pom.xml` da nossa aplicação:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.3.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy</artifactId>
  <version>1.10.9</version>
  <scope>test</scope>
</dependency>
```

A documentação do Mockito é muito boa, com muitos exemplos, então aqui vamos apenas usar o básico, e ir comentando conforme as funcionalidades aparecem. Se quiser saber mais, a documentação do framework é suficiente para isso.

Para fazer nossos testes temos que ter o cenário que pretendemos testar claro na nossa mente. Basicamente vamos testar o caso do cliente comum e do cliente preferencial quando eles pedem mais de um empréstimo no mesmo dia. Pela nossa regra de negócio, o

cliente comum terá o segundo empréstimo negado e o preferencial consegue fazer quantos empréstimos quiser.

Teste do Serviço de Análise Financeira

Vamos então iniciar os testes pela menor peça que compõe essa solução, que é o `ServicoAnaliseFinanceira`. Esse objeto é que faz a análise se o crédito deve ou não ser liberado com base no número de empréstimos no dia. Mas ele não considera se o cliente é preferencial ou não. Aqui basicamente vamos lançar uma exceção se o cliente já fez um pedido de empréstimo no dia.

Usaremos o JUnit e o Mockito para nossos testes. A seguir o código do teste que verifica o caso onde já houve um empréstimo no dia:

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

public class ServicoAnaliseFinanceiraTest {

    @Test(expected = JavacredApplicationException.class)
    public void deve_lancar_excecao_se_ja_tem_emprestimo()
        throws Exception {

        List<RegistroEmprestimo> emprestimos = new ArrayList<>();
        emprestimos.add(new RegistroEmprestimo());

        ServicoAnaliseFinanceira saf =
            spy(new ServicoAnaliseFinanceira());

        doReturn(emprestimos).when(saf)
            .buscarEmprestimos(anyString(), any(Date.class));

        saf.analisar("Fulano", 10_000.0);

    }
}
```

Nesse teste criamos uma lista de um único `RegistroEmprestimo` para ser retornado pelo nosso método de busca, e assim simular que foi

encontrado pelo menos um registro no banco de dados.

No capítulo 6. *Integrando EJB e JPA*, já comentamos sobre separar ou não o acesso a dados em uma classe separada como um DAO. Um dos argumentos a favor do uso de DAOs é que seria mais fácil de testar as classes de negócio colocando um *mock* no lugar do DAO. No entanto, no nosso teste que acabamos de implementar vimos que é perfeitamente possível substituir o acesso a dados através de *mocks* mesmo sem o uso de DAO, desde que separemos o acesso a dados em um método específico.

Então não precisamos separar o acesso a dados em outra classe, mas com certeza precisamos separar em outro método. O que não podemos é no meio do método de negócio fazer uso direto do `EntityManager`. Se você não consegue controlar o impulso de fazer isso, talvez seja melhor usar um DAO.

Mas só é possível fazer essa mescla de objeto real com *mock* porque o Mockito nos disponibiliza o método `spy(Object)`, onde passamos um objeto real que pode ser "*mockado*" parcialmente. Se não configurarmos nenhum comportamento, o método real será chamado, mas podemos sobrescrever esse comportamento por um *mock* dentro dos nossos testes.

Foi o que fizemos na configuração `doReturn(emprestimos)...`. Se não tivermos feito essa configuração, quando o método `analisar()` fosse chamado, ele usaria o método `buscarEmprestimos()` e teríamos uma `NullPointerException` pois não existe um `EntityManager` para fazer a busca no banco. Por isso, configuramos o Mockito para retornar a lista `emprestimos` quando for chamado o método `buscarEmprestimos` com qualquer `String` e qualquer `Date` como parâmetros.

Aqui temos dois pontos a analisar. Primeiro é que usamos essa sintaxe `doReturn()` porque nosso método é `void`. O mais comum é vermos outro tipo de assinatura que veremos mais à frente. E o outro ponto é o uso de `anyString()` e `any(Date.class)` que são *matchers* do Mockito. Temos métodos no estilo `anyXXX()` para os

tipos mais básicos do `java.lang`, e o método `any(Class)` para as demais classes. Usamos esses *matchers* porque se usarmos diretamente um valor ao configurar o comportamento do nosso método, esse comportamento só será aplicado se na execução forem passados exatamente os mesmos objetos usados na configuração; e no nosso caso o objeto passado em si pouco importa.

Atenção: se por acaso você já usou o Mockito 1.x, tome cuidado pois agora no Mockito 2.x os *matchers* não aceitam `null`. Então se eu passar uma `String` nula, o `anyString()` não vai fazer o *match*.

Com tudo pronto, ao executarmos nosso teste vemos que a barra fica verde pois foi lançada a exceção `JavacredApplicationException` conforme o esperado pelo nosso teste.

Vamos agora para o segundo teste, que é validar o comportamento quando nenhum `RegistroEmprestimo` é encontrado:

```
...
@Test
public void nao_deve_lancar_excecao_se_nao_tem_emprestimo()
    throws Exception {

    ServicoAnaliseFinanceira saf =
        spy(new ServicoAnaliseFinanceira());
    EntityManager entityManager = mock(EntityManager.class);
    saf.setEntityManager(entityManager);

    List<RegistroEmprestimo> emprestimos = new ArrayList<>();

    doReturn(emprestimos).when(saf)
        .buscarEmprestimos(anyString(), any(Date.class));

    saf.analisar("Fulano", 10_000.0);

    ArgumentCaptor<RegistroEmprestimo>
        registroEmprestimoCaptor = ArgumentCaptor.forClass(
            RegistroEmprestimo.class);
```

```

verify(entityManager)
    .persist(registroEmprestimoCaptor.capture());

RegistroEmprestimo emprestimoRegistrado =
    registroEmprestimoCaptor.getValue();

assertEquals("Fulano", emprestimoRegistrado.getTomador());
assertEquals(10_000.0, emprestimoRegistrado.getValor(),
    0.001);
}
...

```

Aqui além de usar o `ServicoAnaliseFinanceira` no modo híbrido através do `spy()`, também criamos um *mock* puro para o `EntityManager` e passamos esse *mock* para o objeto que vamos testar.

Perceba que temos que fazer um pequeno ajuste nas nossas classes para que elas fiquem mais testáveis. Por exemplo, o método `setEntityManager` geralmente não é criado, e simplesmente anotamos a propriedade com `@PersistenceContext`. Mas para que possamos passar essa dependência de forma simples nos nossos testes, temos que alterar nossa classe `ServicoAnaliseFinanceira` para ficar assim:

```

@Stateless
public class ServicoAnaliseFinanceira {

    private EntityManager entityManager;

    @PersistenceContext
    public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
    ...
}

```

Veja que apenas mudamos a anotação da propriedade para o *setter*. Conforme nossos exemplos avançam, teremos mais mudanças desse tipo para fazer.

Mas voltando ao exemplo de teste, vemos que configuramos nossa "busca no banco" para retornar uma lista vazia, isso significa que o empréstimo deverá ser registrado, e dessa forma uma instância de `RegistroEmprestimo` deverá ser passada para o método `persist()` do nosso `EntityManager`.

Poderíamos verificar se o método foi chamado da seguinte maneira: `verify(entityManager).persist(any(RegistroEmprestimo.class))`, mas aqui usamos o `ArgumentCaptor` para conseguirmos analisar qual objeto foi passado como parâmetro. Seu uso é como o que podemos ver no teste, após o instanciar, usamos o método `capture()` no `verify()` para podermos analisar o conteúdo do objeto.

Logo em seguida recuperamos o objeto que foi passado para o `EntityManager` com o método `getValue()` e então o usamos nas asserções. Pode ser um código novo, mas é simples de entender depois que conhecemos a sintaxe.

Testando o EmprestimoBean

Agora que já sabemos que o `ServicoAnaliseFinanceira` está funcionando, vamos testar a `EmprestimoBean`, que uma classe mais complexa e que usa a anterior internamente.

Vamos novamente começar testando pelo caso onde o empréstimo não deve acontecer. Para iniciar, vamos configurar o comportamento que esperamos das classes que são dependência da `EmprestimoBean`:

```
public class EmprestimoBeanTest {
    @Test
    public void
        cliente_nao_preferencial_deve_ter_emprestimo_negado() {

        //Análise financeira deve ser "negada"
```

```

    ServicoAnaliseFinanceira saf =
        mock(ServicoAnaliseFinanceira.class);
    doThrow(JavacredApplicationException.class)
        .when(saf).analisar(anyString(), anyDouble());

    EJBContext ejbContext = mock(EJBContext.class);

    ContratoBean contratoBean = mock(ContratoBean.class);

    EmprestimoBean emprestimoBean = new EmprestimoBean();

    emprestimoBean.config(contratoBean, saf);
    emprestimoBean.setEjbContext(ejbContext);
    ...
}
}

```

O primeiro comportamento que configuramos foi o do `ServicoAnaliseFinanceira`. Como seu comportamento interno já é testado no seu próprio teste, aqui só configuramos o *mock* para lançar a exceção `JavacredApplicationException` sempre que o método `analisar()` for chamado. É muito importante manter o isolamento entre os testes, por isso aqui usamos um *mock* do `SAF`, e não uma instância real, pois caso tenhamos um erro na implementação dessa classe, os testes dela é que devem falhar, não os do `EmprestimoBean`.

Em seguida, configuramos um *mock* de `ContratoBean`. E por fim configuramos nosso `EmprestimoBean` com as dependências "*mockadas*". Aqui novamente teremos que ajustar nossa classe para que ela fique mais testável. Então o código da classe `EmprestimoBean` fica parecido com o seguinte:

```

@Stateless
public class EmprestimoBean {
    ...
    @Resource
    public void setEjbContext(EJBContext ejbContext) {
        this.ejbContext = ejbContext;
    }
}

```

```

@Inject
public void config(ContratoBean contratoBean,
                  ServicoAnaliseFinanceira saf){
    this.contratoBean = contratoBean;
    this.saf = saf;
}
...
}

```

Tiramos as anotações `@Resource` e `@Inject` das propriedades e colocamos nos *setters*. Porém, para economizar *setters*, usamos um único método anotado com `@Inject` com todas as dependências providas pelo CDI. Infelizmente ainda não é possível injetar as dependências providas por `@Resource` junto com as providas por `@Inject` de forma automática. Seria necessário criar um método provedor do CDI para isso, mas para não adicionarmos complexidade desnecessária, deixamos dessa forma. Em versões futuras do Java EE isso deve ser resolvido.

Vamos então continuar o desenvolvimento do nosso teste para de fato testar o objeto `EmprestimoBean` que acabamos de configurar.

```

@Test
public void
    cliente_nao_preferencial_deve_ter_emprestimo_negado() {
    ...
    Cliente cliente = new Cliente("Fulano", false);
    Contrato contrato = new Contrato(cliente);

    try {

        emprestimoBean.registrarEmprestimo(contrato);
        fail("Aqui deve ocorrer a exceção");
    }
    catch (JavacredApplicationException e){

    }

    verify(contratoBean).salvar(contrato);
}

```

```
        verify(ejbContext).setRollbackOnly();
    }
}
```

Como configuramos o `ServicoAnaliseFinanceira` para lançar exceção, sabemos que a linha onde colocamos o `fail()` não deve ser executada. Logo em seguida verificamos se o método

`contratoBean.salvar(contrato)` foi invocado com o mesmo `Contrato` que foi criado anteriormente. Repare que aqui checamos se foi passado o mesmo objeto. Se quiséssemos testar se foi passado qualquer `Contrato`, usaríamos `any(Contrato.class)`. E como sabemos que foi passada a mesma instância, não precisamos fazer qualquer asserção usando `ArgumentCaptor` como fizemos antes. Depois verificamos se o método `ejbContext.setRollbackOnly()` foi chamado, isso porque o cliente do contrato não é preferencial.

Agora vamos testar o cenário onde o cliente que tenta fazer o empréstimo é preferencial:

```
@Test
public void
    cliente_preferencial_deve_ter_emprestimo_aprovado() {

    //Análise financeira deve ser "negada"
    ServicoAnaliseFinanceira saf =
        mock(ServicoAnaliseFinanceira.class);
    doThrow(JavacredApplicationException.class)
        .when(saf).analisar(anyString(), anyDouble());

    EJbContext ejbContext = mock(EJbContext.class);

    ContratoBean contratoBean = mock(ContratoBean.class);

    EmprestimoBean emprestimoBean = new EmprestimoBean();
    emprestimoBean.config(contratoBean, saf);
    emprestimoBean.setEjbContext(ejbContext);
    ...
}
```

Iniciamos a configuração exatamente igual ao teste anterior. O `SAF` vai lançar a exceção informando que por ele o empréstimo não deve ser aprovado. Mas as diferenças na configuração vêm no restante do teste:

```
@Test
public void cliente_preferencial_deve_ter_emprestimo_aprovado() {
    ...
    Cliente cliente = new Cliente("Beltrano", true);
    Contrato contrato = new Contrato(cliente);

    emprestimoBean.registrarEmprestimo(contrato);

    verify(contratoBean).salvar(contrato);

    verify(ejbContext, never()).setRollbackOnly();
}
```

Ao analisar o resultado, vemos que agora ao `registrarEmprestimo` a exceção não é lançada. Além disso é invocado o método `salvar` do `ContratoBean`, e agora o método `setRollbackOnly()` nunca deve ser chamado, o que verificamos usando `never()` como segunda argumento do `verify()`.

Como dito nos capítulos anteriores, no desenvolvimento de uma aplicação o melhor seria ir desenvolvendo os testes para verificar o comportamento em vez de só testar através das telas como fizemos. Mas se fizéssemos antes, adicionaríamos todo esse código junto com o conteúdo dos outros capítulos, o que tiraria nosso foco do que era mais importante naquele momento.

A partir de agora vamos continuar desenvolvendo nossos testes, mas dessa vez usaremos efetivamente o servidor de aplicações para não termos que ficar simulando o comportamento do `EJBContext`, por exemplo.

9.2 Criando testes de integração com o contêiner

No último teste que implementamos, por ser um pouco mais complexo, tivemos que fazer várias verificações para ter certeza de que métodos como `EJBContext.setRollbackOnly()` foram chamados nos casos certos etc. Mesmo assim, como acabamos de aprender como lidar com as transações nos capítulos passados, pode ficar aquela pulga atrás da orelha que nos faz pensar se o resultado final seria mesmo o esperado; que era inserir no banco de dados somente o primeiro contrato caso o cliente não seja preferencial, e mais de um caso seja preferencial.

Para deixar ainda mais claro o tipo de problema que queremos resolver, vamos considerar o `ServicoAnaliseFinanceiraTest` onde fazemos verificamos se o método `EntityManager.persist()` foi chamado passando o `RegistroEmprestimo` como parâmetro. À primeira vista parece que essa verificação nos assegura que ao final da execução o `RegistroEmprestimo` estará no banco de dados. Mas e se em outro método, na mesma transação, for chamado o `EJBContext.setRollbackOnly()` ? Será que precisaríamos então que o método de análise de crédito executasse em uma transação separada para ter um isolamento desse tipo de possibilidade?

Apesar de estarmos falando de transações, essa dificuldade de garantir o funcionamento somente usando *mocks* também se aplica para a maioria das situações envolvendo banco de dados. Por exemplo, de que serve substituir um objeto que faz consultas por um *mock* se a consulta pode estar com a lógica ou até mesmo a sintaxe errada e o teste usando *mock* executaria com sucesso?

Nesse tipo de situação não tem como termos certeza do resultado final senão subindo o ambiente e executando o código. Ou seja, vamos ter que desenvolver um teste que rode de verdade dentro do servidor de aplicações.

9.3 Utilizando o Arquillian para testes de integração

Por mais que o `EmprestimoBeanTest` que criamos possa ser considerado um teste de integração por verificar como o `EmprestimoBean` funciona em conjunto com outros objetos, pelo fato de usarmos ferramentas básicas de teste de unidade, como o JUnit e o Mockito, que serve para de alguma forma isolar o objeto que estamos testando dos demais, muitas vezes esse tipo de testes também é equivocadamente chamado de teste de unidade.

Lembrando o que foi dito no início do capítulo, um teste de unidade testa a menor unidade do sistema, que no caso de desenvolvimento Orientado a Objetos é um objeto, logo uma só classe. E aqui desde o início fizemos teste de integração. Também no início do capítulo subdividimos o teste de integração entre o "puro", que usa as mesmas ferramentas do teste unitário (e daí alguns podem confundir a nomenclatura), e o teste de integração que envolve o ambiente de *runtime*, ou os contêineres. E aqui estou usando o termo "contêiner" em sentido amplo, podendo ser tanto o servidor de aplicações como o Wildfly, um *servlet container* como o Tomcat, o motor de CDI como o Weld, ou mesmo o que hoje é mais comumente associado ou termo "contêiner" que seria uma imagem do Docker.

Novamente, apenas para simplificar, costuma-se chamar os primeiros testes de unitário (seja unitários propriamente ditos ou de integração "puros" como os que fizemos) mais pela velocidade com que eles executam, pois não precisam subir nenhum contêiner. Então quando falarmos de testes de integração de agora em diante, estaremos falando desses últimos, que realmente executam o teste dentro do mesmo ambiente que nosso código será executado em produção.

Nos nossos testes de integração vamos utilizar o Arquillian (<http://arquillian.org>), um projeto da JBoss para executar nossos

testes dentro do Wildfly (ou outro servidor que quisermos usar), e assim conseguirmos tirar a pulga de trás da orelha como comentamos no início deste capítulo; tento certeza que ao final da execução do nosso código o resultado é o esperado.

9.4 Configurando o Arquillian na nossa aplicação

O Arquillian é uma ferramenta bastante completa, possibilitando incontáveis combinações de configuração de ambientes. Como nosso objetivo aqui é mais testar se o que entendemos no capítulo anterior está correto do que fazer um estudo aprofundado do Arquillian, vamos configurar apenas um cenário onde o Arquillian usa a instalação que já temos do Wildfly para executar os testes, mas é possível inclusive configurar nossos testes para baixar automaticamente o servidor de aplicações e executar os testes nele. Podemos inclusive rodar os mesmos testes no Wildfly e no Glassfish para verificar se estamos mantendo a compatibilidade etc.

Basicamente, o Arquillian é uma dependência (ou algumas) que precisamos colocar no `pom.xml` da nossa aplicação. Como o Arquillian possui diversos módulos, usamos a tag `dependencyManagement` para "instalar" a base do Arquillian, e depois vamos colocando os módulos que precisamos, e assim todos seguem a versão da base instalada. Então vamos adicionar o seguinte trecho no `pom.xml` do projeto `javacred`.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian</groupId>
      <artifactId>arquillian-bom</artifactId>
      <version>1.6.0.Final</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
```

```
    </dependencies>
</dependencyManagement>
```

Aqui estamos usando essa versão, mas obviamente é uma boa ideia buscar a mais recente quando estiver desenvolvendo sua aplicação. Vale lembrar que, como em todo caso em que atualizamos a versão de algo, precisamos tomar cuidado pois algo mostrado aqui pode ser diferente na versão que você for utilizar.

Agora que já temos a base instalada, podemos adicionar o seguinte módulo:

```
<dependency>
  <groupId>org.jboss.arquillian.junit</groupId>
  <artifactId>arquillian-junit-container</artifactId>
  <scope>test</scope>
</dependency>
```

Perceba que não precisamos definir a versão, pois esse módulo segue a versão definida no `dependencyManagement`. Porém, além do suporte ao JUnit do Arquillian, vamos ter que adicionar também a dependência de um ambiente onde nosso teste vai executar, que no caso será o Wildfly Embedded:

```
<dependency>
  <groupId>org.wildfly.arquillian</groupId>
  <artifactId>
    wildfly-arquillian-container-embedded
  </artifactId>
  <version>2.2.0.Final</version>
  <scope>test</scope>
</dependency>
```

Aqui, como não é uma dependência do Arquillian, e sim algo feito para ser executado pelo Arquillian, temos que especificar a versão. Ocorre que a versão *embedded* do Wildfly não é tão encapsulada quanto a versão equivalente de outros servidores de aplicação onde só definimos a dependência no `pom.xml` e o servidor já sai funcionando. No caso do Wildfly precisamos especificar um lugar para ele usar a "instalação" do servidor. Podemos fazer isso no

próprio `pom.xml` , fazendo o próprio Maven baixar e descompactar o zip do Wildfly e executar o teste usando ele. Mas para simplificar, vamos usar a própria instalação que temos.

Para configurar essas informações de contexto, como onde buscar a instalação do Wildfly, usamos o arquivo `arquillian.xml` que fica na raiz dos nossos testes: `src/test/resources` :

```
<arquillian xmlns="http://jboss.org/schema/arquillian"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="
                http://jboss.org/schema/arquillian
                http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
    <container qualifier="wildfly" default="true">
        <configuration>
            <property name="jbossHome">
                /Users/.../wildfly-19.0.0.Final
            </property>
            <property name="modulePath">
                /Users/.../wildfly-19.0.0.Final/modules
            </property>
        </configuration>
    </container>
</arquillian>
```

Apenas fechei a tag `property` na outra linha, e abreviei o caminho do servidor pela legibilidade do código no livro. O que importa é que o Wildfly Embedded precisa de duas configurações: `jbossHome` e `modulePath` . No caso nós especificamos o caminho de onde o servidor foi baixado, e o diretório de módulos. Apesar do que parece, como estamos usando o modo *embedded* do Wildfly, ele não vai subir nosso servidor como se rodássemos o comando `./standalone.sh` (ou `.bat` no Windows), então se tem aplicações dentro da pasta `deployments` ou configurações no `standalone.xml` não precisamos nos preocupar, é como se fosse uma forma de "economizar" um download apenas.

Por fim, precisamos adicionar uma configuração no plugin `maven-surefire-plugin` no nosso `pom.xml` :

```

<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.17</version>
  <configuration>
    <systemPropertyVariables>
      <java.util.logging.manager>
        org.jboss.logmanager.LogManager
      </java.util.logging.manager>
    </systemPropertyVariables>
  </configuration>
</plugin>

```

Essa configuração serve apenas para passarmos como parâmetro quem será o *logging manager* do servidor.

Agora vamos criar nosso primeiro teste usando o Arquillian.

9.5 Criando um teste "hello world"

Para começar, vamos fazer um EJB que não faz nada de especial nem tem qualquer dependência para vermos o mínimo necessário.

```

@Stateless
public class CalculadoraSimplesBean {

    public double calculaPercentual(double valor,
        double percentual) {
        return valor * percentual;
    }
}

```

Como é possível ver, nosso EJB faz uma simples multiplicação para chegar a um percentual. Agora vamos construir o teste:

```

import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;

```

```

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(Arquillian.class)
public class CalculadoraSimplesTest {

    @Deployment
    public static JavaArchive createDeployment() {
        JavaArchive jar = ShrinkWrap.create(JavaArchive.class)
            .addClass(CalculadoraSimplesBean.class)
            //não obrigatório, apenas para deixar CDI instalado
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
        System.out.println(jar.toString(true));
        return jar;
    }

    @Inject
    CalculadoraSimplesBean calculadora;

    @Test
    public void deve_calcular_vinte_porcento() {
        Assert.assertEquals(2_000.0,
            calculadora.calculaPercentual(10_000.0, 20.0/100),
            0.0001);
    }
}

```

Como sempre, deixei os *imports* visíveis para não termos dúvidas que a IDE automaticamente fez o correto.

Agora vamos por partes. O primeiro ponto do nosso teste é que ele deve ser gerenciado pelo Arquillian como podemos ver na anotação `@RunWith(Arquillian.class)` da nossa classe.

Depois temos o segundo ponto, que é a construção do arquivo que será instalado no nosso servidor para a execução do teste. Isso é

feito no método estático que devolve uma instância de `Archive` (aqui usamos a subclasse `JarArchive`). Aqui usamos o `ShrinkWrap` para criar um EJB-JAR que tem apenas a classe `CalculadoraSimplesBean`. Adicionalmente colocamos um `beans.xml` em branco para instalar o CDI no pacote. Isso não é obrigatório, mas é bom deixar aqui pois fica de referência para quando formos criar outros testes, em outros projetos. Em seguida mandamos escrever a estrutura desse jar no console apenas para vermos o que tem dentro dele, e em seguida retornamos o arquivo. Esse método deve ser anotado com `@Deployment` e funciona como um `@BeforeClass` do JUnit.

Por fim, temos o código de teste propriamente dito, que consiste em injetarmos nosso EJB e em seguida fazer o teste verificando se nosso bean sabe fazer o cálculo de percentual conforme esperado.

Veja que injetamos nosso EJB como se nosso próprio teste também fosse um EJB. E como estamos executando o servidor de verdade, poderemos nos próximos testes validar se as transações estão funcionando como queríamos etc. Mas por enquanto vamos apenas executar nosso teste como um teste comum do JUnit e ver que a barra verde apareceu.

Analisando o console, veremos algo parecido com isso:

```
831aafaa-fdcf-4a4a-9376-0076e9e08a6d.jar:  
/br/  
/br/com/  
/br/com/casadocodigo/  
/br/com/casadocodigo/javacred/  
/br/com/casadocodigo/javacred/ejbs/  
/br/com/casadocodigo/javacred/ejbs/CalculadoraSimplesBean.class
```

Significa que foi gerado um nome qualquer para o nosso jar (até podemos especificar se quisermos), e em seguida a estrutura completa desse jar. Isso é interessante para termos a exata noção se o arquivo está sendo gerado exatamente como queríamos.

Agora que passamos do *hello world*, podemos testar nosso exemplo de verdade.

9.6 Criando um teste real

O objetivo deste capítulo é fazermos um teste de integração do `EmprestimoBean`. Então vamos criar a classe e o método que cria o arquivo para *deploy*.

```
@RunWith(Arquillian.class)
public class EmprestimoBeanIntegrationTest {

    @Deployment
    public static Archive createDeployment() {
        Archive archive = ShrinkWrap.create(WebArchive.class)
            .addClasses(EmprestimoBean.class)
            //não obrigatório, apenas para deixar CDI instalado
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");

        System.out.println(archive.toString(true));
        return archive;
    }
}
```

Não temos nada de novo ainda, apenas mudamos o formato do arquivo de um EJB-JAR para um WAR. Fora isso, em vez de usar o método `addClass`, usamos a versão dele no plural para podermos passar várias classes (desculpe o *spoiler*, uma classe não será suficiente aqui).

E agora vamos ao código de testes:

```
...
@Inject
EmprestimoBean emprestimoBean;
```



```

@Test
public void
    cliente_nao_preferencial_deve_ter_2o_emprestimo_rejeitado(){

    Cliente cliente = new Cliente("Fulano", false);
    emprestimoBean.registrarEmprestimo(new Contrato(cliente));
    try{
        emprestimoBean.registrarEmprestimo(new Contrato(cliente));
        Assert.fail("Não pode haver um segundo empréstimo");
    }
    catch (JavacredApplicationException e){

    }
}

```

Vamos relembrar da regra do nosso empréstimo: clientes preferenciais podem fazer quantos empréstimos quiserem, mas os que não são preferenciais podem fazer apenas um por dia. Nesse teste, não pode haver o segundo empréstimo pois criamos o cliente como não preferencial. Então chamamos o método `registrarEmprestimo(Contrato)` uma primeira vez, e não devemos ter problemas. Já a segunda vez, fizemos a chamada dentro do `try-catch` pois o esperado é que haja uma exceção, e se essa exceção não acontecer, o teste deve falhar.

Agora executando o teste temos o seguinte erro:

```

...
Caused by: java.lang.NoClassDefFoundError:
br/com/casadocodigo/javacred/exceptions/
    JavacredApplicationException
...

```

Isso acontece porque colocamos apenas o `EmprestimoBean` no nosso arquivo, mas ele tem diversas dependências. Conforme formos adicionando as classes pedidas, novos erros como esse vão aparecer até que todas as classes sejam adicionadas. Para ganhar tempo, segue a versão que tem todas as classes necessárias:

```

@Deployment
public static Archive createDeployment() {
    Archive archive = ShrinkWrap.create(WebArchive.class)
        .addClasses(EmprestimoBean.class,
            ContratoBean.class,
            AjustadorContratoBean.class,
            ServicoAnaliseFinanceira.class,
            JavacredApplicationException.class,
            JavacredException.class,
            Contrato.class, Cliente.class, Indice.class,
            IndiceValor.class, RegistroEmprestimo.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");

    System.out.println(archive.toString(true));
    return archive;
}

```

Com esse código, passamos a ter todas as classes necessárias para nosso exemplo, mas ao executar recebemos mais um erro:

```

...
Can't find a persistence unit named null in deployment ...

```

Essa mensagem aparece lá no meio da *stack trace*. E isso acontece porque dentro do `ContratoBean`, que é dependência do `EmprestimoBean`, temos a injeção por `EntityManager` via `@PersistenceContext`, e para isso funcionar precisamos de um `persistence.xml`.

Como aqui estamos executando um teste, vamos criar um `persistence` de teste no caminho `src/test/resources/test-persistence.xml` com o seguinte conteúdo:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="test">

```

```

    <jta-data-source>
        java:jboss/datasources/ExampleDS
    </jta-data-source>
    <properties>
        <property name="hibernate.hbm2ddl.auto"
            value="create-drop"/>
        <property name="hibernate.show_sql" value="true"/>
    </properties>
</persistence-unit>
</persistence>

```

Aqui usamos aquele *datasource* que já vem instalado por padrão no Wildfly só para não termos que configurar um DS para o teste. E configuramos como `create-drop` porque não precisamos guardar dados de testes.

Para adicionar esse persistence ao nosso WAR precisamos ajustar a criação do nosso arquivo da seguinte maneira:

```

@Deployment
public static Archive createDeployment() {
    Archive archive = ShrinkWrap.create(WebArchive.class)
        .addClasses(EmprestimoBean.class,
            ContratoBean.class,
            AjustadorContratoBean.class,
            ServicoAnaliseFinanceira.class,
            JavacredApplicationException.class,
            JavacredException.class,
            Contrato.class, Cliente.class, Indice.class,
            IndiceValor.class, RegistroEmprestimo.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    .addAsResource("test-persistence.xml",
        "META-INF/persistence.xml");

    System.out.println(archive.toString(true));
    return archive;
}

```

O ShrinkWrap sabe o local certo de colocar o arquivo `persistence.xml` dependendo do tipo de arquivo que escolhemos (jar

ou war, pode exemplo). E aqui ao adicionar o xml, o renomeamos dentro do arquivo de deploy.

Executando agora o teste, vemos a barra verde. Alterne entre `JarArchive` e `WebArchive` e analise o console para ver como o `ShrinkWrap` muda a estrutura do arquivo. Apesar de nosso teste estar passando, não temos nenhuma asserção nele, o que não é aceitável. Se relembrarmos da regra da nossa aplicação, como o caso sendo testado é de um cliente não preferencial, ao final teremos apenas uma instância de `Contrato` persistida no banco de dados, então vamos usar isso para validar nosso teste. Porém, se lembrarmos como estão nossas classes, quem faz essa gestão de `Contratos` no banco de dados é o `ContratoBean`. Então vamos ter que usá-lo diretamente no nosso teste:

...

```
@Inject
EmprestimoBean emprestimoBean;

@Inject
ContratoBean contratoBean;

@Test
public void
    cliente_nao_preferencial_deve_ter_2o_emprestimo_rejeitado() {

    Assert.assertEquals(0, contratoBean.listarTodos().size());

    Cliente cliente = new Cliente("Fulano", false);
    emprestimoBean.registrarEmprestimo(new Contrato(cliente));
    try{
        emprestimoBean.registrarEmprestimo(new Contrato(cliente));
        Assert.fail("Não pode haver um segundo empréstimo");
    }
    catch (JavacredApplicationException e){

    }
}
```

```
Assert.assertEquals("O segundo empréstimo deve ser negado!",  
    1, contratoBean.listarTodos().size());  
  
}
```

Adicionamos as asserções para validar que antes de executar nosso código não existe nenhum registro de `Contrato` no banco, e depois, por mais que tenhamos tentado registrar dois empréstimos, só existe um registro, pois o segundo foi negado uma vez que o cliente não é preferencial. Mesmo que antes tenhamos feito a simulação desse funcionamento usando *mocks*, agora sabemos que o funcionamento em produção será o esperado.

Ainda assim, falta testar o caso do cliente preferencial, então vamos adicionar o seguinte teste:

```
@Test  
public void  
    cliente_preferencial_deve_ter_2o_emprestimo_aprovado() {  
  
    Assert.assertEquals(0, contratoBean.listarTodos().size());  
  
    Cliente cliente = new Cliente("Beltrano", true);  
    emprestimoBean.registrarEmprestimo(new Contrato(cliente));  
    emprestimoBean.registrarEmprestimo(new Contrato(cliente));  
  
    Assert.assertEquals(  
        "Os dois empréstimos devem ser concedidos",  
        2, contratoBean.listarTodos().size());  
}
```

Agora executando os testes da nossa classe, percebemos que o novo teste passa, mas o anterior falha já na primeira asserção, pois espera que antes do teste não haja nenhum `Contrato`, mas vem dois, resultado do teste com o cliente preferencial.

Esse é o tipo de erro que executando cada teste individualmente não veríamos, pois ocorre justamente de um teste influenciar no outro. Por isso, em qualquer ferramenta que usarmos para construir nossos testes, temos que deixá-los totalmente independentes. Para

tanto vamos usar um método de configuração de ambiente, que sempre limpará os registros tanto de `Contrato` quanto de `Cliente`.

```
@PersistenceContext
EntityManager em;

@Inject
UserTransaction utx;

@Before
public void limparDados() throws Exception {
    utx.begin();
    em.createQuery("delete from Contrato ")
        .executeUpdate();
    em.createQuery("delete from RegistroEmprestimo")
        .executeUpdate();
    utx.commit();
}
```

Nesse trecho, injetamos o `EntityManager` para poder executar os comandos de limpeza dos dados, e também o `UserTransaction`, pois só conseguimos escrever no banco de dados se estivermos dentro de uma transação, e nosso teste não é transacional como um EJB. Agora podemos executar novamente nossos testes e veremos que o funcionamento está de acordo com o esperado.

9.7 Usando mocks nos testes com Arquillian

Apesar de termos visto que alguns testes, especialmente quando envolve banco de dados, não serem tão efetivos quando usamos *mocks*, estes continuam sendo muito úteis para isolar a classe que estamos testando.

No nosso exemplo do teste de integração do `EmprestimoBean` não é tão interessante substituir o `ContratoBean` por um *mock*, pois ele é quem faz o contato com o banco de dados, e validar o que vai para

o banco é justamente o que buscamos com nosso teste de integração. Agora o `ServicoAnaliseFinanceira` poderia sim ser substituído por um *mock* mesmo dentro do nosso teste de integração. Isso é interessante não somente para isolar o máximo possível nosso objeto sendo testado, mas também para nos permitir criar cenários de teste mais precisos.

Assim sendo, vamos criar mais um caso de teste que agora vai usar o que já vimos de *mock*, só que dentro do teste de integração. Como o código vai ser um pouco longo vamos quebrar em duas partes:

```
@Inject
ServicoAnaliseFinanceira servicoAnaliseFinanceira;

@Test
public void
cliente_nao_preferencial_deve_ter_2o_emprestimo_rejeitado_mock(){

    Assert.assertEquals(0, contratoBean.listarTodos().size());

    //configura os objetos reais para serem inspecionáveis
    ContratoBean contratoBean = spy(this.contratoBean);

    //cria mock para substituir o ServicoAnaliseFinanceira real
    ServicoAnaliseFinanceira servicoAnaliseFinanceira =
        mock(ServicoAnaliseFinanceira.class);

    doThrow(JavacredApplicationException.class)
        .when(servicoAnaliseFinanceira)
        .analisar(anyString(), anyDouble());

    //configura o emprestimoBean com os objetos
    //inspecionáveis e mock
    emprestimoBean
        .config(contratoBean, servicoAnaliseFinanceira);

    ...
}
```

Até aqui nós configuramos nosso `EmprestimoBean` com o *mock* do `ServicoAnaliseFinanceira`, que é basicamente a mesma configuração que usamos na primeira parte desse capítulo quando testamos apenas usando *mocks*. A diferença ficou por conta do `ContratoBean`, onde estamos usando o objeto real, e portanto precisamos injetá-los também como dependência da nossa classe de testes.

O Mockito nos dá a opção de fazer um `spy` em objetos reais, assim podemos tanto inspecioná-los, quanto sobrescrever algum método desse objeto da mesma maneira que configuramos os métodos de um *mock*. No nosso exemplo não sobrescreveremos nada, apenas usamos o `spy` para podermos fazer asserções mais abaixo no nosso teste.

Então vamos agora à segunda metade do teste:

```
@Test
public void
cliente_nao_preferencial_deve_ter_2o_emprestimo_rejeitado_mock(){
    ...

    Cliente cliente = new Cliente("Ciclano", false);
    Contrato contrato = new Contrato(cliente);
    try{
        emprestimoBean.registrarEmprestimo(contrato);
        Assert.fail("Não pode haver nenhum empréstimo");
    }
    catch (JavacredApplicationException e){
    }

    verify(contratoBean).salvar(contrato);

    Assert.assertEquals("O empréstimo deve ser negado!", 0,
        contratoBean.listarTodos().size());

    //volta a implementação padrão
    //para não influenciar os outros testes
    emprestimoBean.config(this.contratoBean,
        this.servicoAnaliseFinanceira);
}
```


Agora vamos fazer uso da configuração que fizemos antes. Vamos invocar o `registrarEmprestimo` e começar nossas asserções. O interessante de inspecionarmos os objetos reais, é que podemos verificar que o método `salvar` do `ContratoBean` foi chamado, mas mesmo assim ao olhar no banco de dados, vemos que nada foi persistido porque foi dado *rollback* na transação através da invocação de `EJBContext.setRollbackOnly()`. Dessa maneira podemos concluir que a combinação de *mocks* com objetos reais inspecionáveis é o melhor dos dois mundos.

Como já temos o código do teste completo, vamos executar novamente nossa classe de testes e ver o resultado:

```
java.lang.NoClassDefFoundError: org/mockito/Mockito
```

Aqui temos o erro dizendo que a classe `Mockito` não foi encontrada. Isso porque ele não foi incluído no nosso pacote que é gerado lá no método anotado com `@Deployment`. No entanto, não adianta adicionarmos a classe `Mockito` como fizemos com nossas classes de negócio, pois nem sabemos quantas dependências podem haver por baixo dos panos, afinal, terceirizamos esse trabalho para o Maven ao adicionar o Mockito como dependência do nosso projeto.

A solução é fazer o mesmo com o nosso pacote que será instalado no momento do teste. Vamos fazer programaticamente como se estivéssemos criando um `pom.xml` dentro do projeto a ser instalado. Mas primeiro temos que adicionar mais uma dependência no `pom.xml` do nosso projeto:

```
<dependency>
  <groupId>org.jboss.shrinkwrap.resolver</groupId>
  <artifactId>shrinkwrap-resolver-impl-maven</artifactId>
  <scope>test</scope>
</dependency>
```

Isso nos dará acesso à classe que resolve as dependências do Maven programaticamente. Em seguida, ajustaremos o método `createDeployment()` da nossa classe de testes da seguinte forma:

```

import org.jboss.shrinkwrap.resolver.api.maven.Maven;
...

@Deployment
public static Archive createDeployment() {
    Archive archive = ShrinkWrap.create(WebArchive.class)
        .addClasses(EmprestimoBean.class,
            ContratoBean.class,
            AjustadorContratoBean.class,
            ServicoAnaliseFinanceira.class,
            JavacredApplicationException.class,
            JavacredException.class,
            Contrato.class, Cliente.class, Indice.class,
            IndiceValor.class, RegistroEmprestimo.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
        .addAsResource("test-persistence.xml",
            "META-INF/persistence.xml")
        .addAsLibraries( //adiciona dependências via Maven
            Maven.resolver()
                .resolve("org.mockito:mockito-core:3.3.3")
                .withTransitivity().asFile()
        );

    System.out.println(archive.toString(true));
    return archive;
}

```

Adicionamos aqui a chamada do método `addAsLibraries(File[])`, passando os arquivos necessários que foram resolvidos pelo Maven. Esse método só existe no `WebArchive`, pois não tem como colocarmos jars dentro de outro jar. Então até aqui era possível usar tanto `WebArchive` quanto `JarArchive` na hora de gerar nosso pacote. A partir de agora só podemos ter arquivos war. Feita essa configuração, nosso teste usando *mocks* vai funcionar.

Para fechar o capítulo

Não precisamos discutir a importância de testar nosso código, e apesar de termos feito alguns testes usando *mocks* para simular o

comportamento esperado de outros objetos, em algumas situações isso não é suficiente.

Neste capítulo vimos ambos os casos, quando vale a pena usar *mocks* e quando isso não basta, e então temos que de fato o melhor é colocar tudo para executar no ambiente real e ver o funcionamento.

Vimos como utilizar o Arquillian para essa tarefa. Vimos na verdade somente o seu básico, pois existem diversas outras formas de utilizá-lo para cenários de testes mais severos. Em alguma situação pode ser útil executar em mais de um servidor de aplicações para garantir a compatibilidade, ou ainda fazer uso de módulos de teste de sistema para testar inclusive nossas telas. Mas como nosso objetivo aqui é aprender a desenvolver o *back-end* usando EJBs, o que vimos neste capítulo foi o suficiente.

No próximo capítulo veremos como transformar nossos EJBs em *web services* REST. Não precisamos ter de fato um EJB por trás dos nossos WSs; até porque cada tipo, REST ou SOAP, possui sua especificação (JSR) independente. Mas essa é a forma mais simples de implementar um serviço real que acessa banco de dados, por exemplo. O que importa é que, independente do tipo de WS que formos usar na nossa aplicação, veremos que é mais simples de implementar do que parece quando visto de longe.

CAPÍTULO 10

Criando WebServices com JAX-RS

Até aqui vimos que criar um EJB é tão simples quanto criar qualquer classe java, bastando anotá-la para termos um bean transacional. Só precisamos definir interfaces quando queremos expor nossos beans como serviços remotos. Já foi comentado que a grande maioria dos beans de uma aplicação costumam ser locais, mas quando precisarmos de um bean remoto já sabemos como fazer também.

Neste capítulo veremos como criar *WebServices* com o Java EE, o que num primeiro momento pode ser visto como uma complementação aos EJBs remotos, pois permite a integração entre aplicações não escritas em Java com os nossos EJBs; mas em alguns casos pode ser a única forma que usaremos para expor nossos serviços. É só lembrarmos da primeira parte do livro onde focamos em microserviços.

A utilização de um grande número de serviços, expostos como WebServices, é uma característica muito comum hoje em aplicações que fazem a camada por trás de aplicações móveis, e também no cenário de microserviços. Por mais que a tecnologia utilizada não defina a arquitetura em si, é muito mais comum usar WebServices, em especial os REST para a construção de microserviços que usar SOAP ou EJBs remotos para isso.

10.1 JAX-RS: Transformando um EJB em um serviço REST que devolve JSON

Agora nós vamos fazer algumas alterações no nosso EJB `CalculadoraFinanciamento`, que é Stateless para que ele fique

disponível como um serviço REST. É algo que fizemos na primeira parte do livro, naquele projeto separado que chamamos de `javacred-corretora`, mas lá apenas demos uma olhada rápida, agora vamos estudar com mais calma.

No nosso código Java teremos apenas que adicionar algumas anotações, ficando da seguinte maneira:

```
import javax.ejb.Stateless;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Stateless
@Path("/financiamento")
@Produces(MediaType.APPLICATION_JSON)
public class CalculadoraFinanciamento
    implements CalculadoraFinanciamentoRemota{

    @GET @Path("/simular")
    public double simulaFinanciamento(
        @QueryParam("valor") double valorEmprestimo,
        @QueryParam("meses") int meses){

        //o resto fica igual...
    }
    ...
}
```

As anotações adicionadas foram `@Path` e `@Produces` na nossa classe, sendo a segunda opcional; e no nosso método as anotações `@GET`, `@Path` e `@QueryParam`. A anotação `@Path` serve para especificar o caminho para se invocar o método. A anotação `@GET` especifica que esta será a forma de acesso desse método, então podemos facilmente invocá-lo no próprio browser para testar. E por fim a anotação `@QueryParam` especifica como os parâmetros serão

informados. No nosso exemplo, o seguinte teste poderia ser feito pelo browser: `<contexto>/financiamento/simular?valor=10000&meses=10` .

O nosso serviço já está pronto, mas agora vem o passo de configurar o contexto REST da nossa aplicação. Nas aplicações que utilizam JAX-RS precisamos ter uma classe que representa a nossa API REST. A JAX-RS é uma especificação para criação de serviços REST, assim como a JPA é uma especificação para mapeamento objeto-relacional.

O intuito dessa classe é registrar nossos serviços, dentre outras coisas. Mas caso não façamos nada, o RESTEasy vai procurar por serviços dentro da nossa aplicação e registrá-los automaticamente. O RESTEasy é a implementação da JAX-RS que vem dentro do Wildfly, assim como o Hibernate é a implementação da JPA que vem nesse servidor.

A implementação dessa classe é bem simples como podemos ver:

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/rest")
public class JavacredApplication extends Application {

}
```

Basicamente estendemos a classe abstrata `Application` e anotamos com `@ApplicationPath` , onde especificamos a raiz do nosso contexto REST. Assim, se no exemplo anterior foi mostrado o caminho `<contexto>/financiamento/simular?valor=10000&meses=10` , agora podemos especificar o que vem a ser o `<contexto>` . Ele é composto do contexto da nossa aplicação, que rodando localmente deve ser algo como `http://localhost:8080/javacred` , e acrescenta o contexto dos serviços REST, que definimos como `/rest` . Dessa maneira, para testar o nosso serviço, precisaremos acessar o seguinte caminho: `http://localhost:8080/javacred/rest/financiamento/simular?`

valor=10000&meses=10 , mas para isso precisaremos de um último passo.

Até agora não precisamos nem mesmo criar o arquivo `web.xml` , já que é opcional em aplicações web a partir do Servlet 3.0. Porém, para o funcionamento do RESTEasy vamos precisar de um `web.xml` , mesmo que seja vazio como o seguinte.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

</web-app>
```

Agora sim podemos testar e ver o resultado no próprio navegador. Vai aparecer somente um número pois é o que nosso serviço retorna, mas essa já é uma resposta JSON válida; e isso vai ficar mais visível no nosso próximo exemplo.

Criando nosso segundo serviço REST

Parece repetitivo continuar nesse tópico, mas as vezes a primeira impressão não é a melhor pois temos configurações globais a fazer. Um exemplo foi a classe `JavacredApplication` , e o arquivo `web.xml` que, por mais simples que sejam, precisaram ser criados. Além disso, agora já conhecemos o "jeitão" de fazer, logo, vamos mais rápido.

Vamos então criar um serviço que devolve os detalhes de um determinado contrato. Para isso vamos utilizar o identificador desse contrato, que no nosso caso é a própria chave primária, mas poderia ser um número com dígito verificador ou qualquer outra regra de negócio. Da mesma maneira seria possível buscar todos os contratos de um determinado cliente e assim por diante, mas aí é só implementar como faremos aqui com os detalhes do contrato.

Alteraremos a classe `ContratoBean`, e seu código ficará da seguinte forma:

```
import javax.ws.rs.PathParam;
...

@Stateless
@Path("/contrato")
@Produces(MediaType.APPLICATION_JSON)
public class ContratoBean {

    @PersistenceContext
    private EntityManager em;

    @GET
    @Path("/{id}")
    public Contrato buscarContrato(@PathParam("id") Integer id){
        return em.find(Contrato.class, id);
    }

    //resto continua igual

}
```

Agora para testar basta buscar algum contrato que exista no nosso banco de dados. Por exemplo podemos ver os detalhes do contrato com identificador `1` pelo segundo caminho

`http://localhost:8080/javacred/rest/contrato/1`. A primeira diferença aqui é que não especificamos um caminho para o nosso método `buscarContrato`, mas como ele é o único método `GET` do nosso serviço, não tem problema. Outra é que em vez de passarmos o parâmetro no estilo *query string*, o informamos como parte do caminho do serviço, e para isso utilizamos a anotação `@PathParam`.

Dessa vez ao acessar nosso serviço pelo navegador podemos ver um JSON mais complexo como resposta. Para facilitar a visualização é interessante instalar no navegador algum complemento que exiba o conteúdo JSON formatado. Há opções

diferentes para cada navegador, escolha a que mais lhe agradar, e o resultado será algo parecido com o seguinte.

```
{
  id: 1,
  descricao: "Parcelamento da Loja XYZ",
  valor: 300,
  quantidadeMeses: 6,
  taxaMensal: null,
  saldo: 300,
  cliente: {
    id: 2,
    nome: "Fulano",
    preferencial: false,
    contratos: [ ]
  }
}
```

Figura 10.1: Exemplo de resposta JSON

Apesar do sucesso no retorno do contrato no formato JSON, se observarmos o console do nosso servidor perceberemos que algo não esperado aconteceu: uma exceção de carregamento *lazy* dos contratos do cliente do contrato que estamos buscando. Por se tratar de uma lista, ela deve ser *lazy* mesmo, sendo carregada somente se necessário. Já vimos como tratar isso na seção 6.5, mas como não podemos ter um bean *Stateful* servindo serviços REST, a melhor forma seria utilizar um *join fetch* que também foi visto na seção citada.

Seria essa a alternativa se isso não causasse um *loop* infinito no nosso JSON. Afinal estamos mostrando um contrato, que é de um

cliente, que tem contratos, que tem aquele cliente, e por aí vai. Dessa forma, em vez de simular essa falha, vamos simplesmente ignorar a lista de contratos do nosso cliente ao gerar a resposta JSON. Para isso vamos fazer o seguinte na classe `Cliente`.

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlTransient;
...

@Entity
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente implements Serializable {

    ...

    @XmlTransient
    @OneToMany(mappedBy="cliente", cascade=CascadeType.ALL)
    private Set<Contrato> contratos = new LinkedHashSet<>();

    ...

}
```

Acrescentando essas anotações nós dissemos que queremos que a lista de contratos do cliente seja ignorada quando for gerada uma saída via WebServices. Mas veja que estranho, utilizamos anotações de XML sendo que até agora não vimos saída em XML, apenas em JSON. Mas fique tranquilo que logo veremos como escolher dinamicamente o tipo de saída, por ora basta sabermos que, independente da renderização da saída, se JSON ou XML, usaremos essas anotações para configurar uma única vez e funcionará para ambos os casos.

Para facilitar a memorização disso, que não é muito intuitivo, pense que os serviços utilizando XML já eram populares antes dos serviços que utilizam REST, por isso a configuração do primeiro é "reaproveitada" para o segundo.

Analizando as anotações que usamos, temos a `@XmlTransient` que é quem efetivamente faz o trabalho que queremos, que é ignorar a propriedade ao gerar a saída - que como estamos vendo, não precisa ser necessariamente no formato XML. Já a anotação `@XmlAccessorType(XmlAccessType.FIELD)` serve apenas para dizer que colocaremos a anotação `@XmlTransient` no "campo"/propriedade/atributo `contratos`. Se não especificarmos isso, as anotações de configuração serão buscadas nos métodos *get*, e não no "campo".

Esse último aspecto também ocorre na JPA, pois podemos anotar nossos atributos ou os *getters*, mas a diferença é que a JPA tem a capacidade de detectar onde estão as anotações sem sermos obrigados a especificar como tivemos que fazer aqui.

Agora sim podemos executar novamente nosso serviço e perceber que não há mais nenhuma exceção no console e que também não aparece mais a propriedade `contratos` no JSON retornado como podemos ver a seguir.

```
{
  id: 1,
  descricao: "Parcelamento da Loja XYZ",
  valor: 300,
  quantidadeMeses: 6,
  taxaMensal: null,
  saldo: 300,
  cliente: {
    id: 2,
    nome: "Fulano",
    preferencial: false
  }
}
```

Figura 10.2: Exemplo de resposta JSON ignorando uma propriedade

10.2 Retornando um XML através do serviço REST

Na seção anterior acabamos utilizando anotações que aparentemente só deveríamos utilizar quando estivéssemos trabalhando com XML, mas estávamos retornando JSON. Isso ficou especificado através da anotação

`@Produces(MediaType.APPLICATION_JSON)` que até agora sempre temos colocado nos nossos serviços.

Se nos recordarmos do primeiro serviço que implementamos veremos que essa anotação é opcional, e se não a especificarmos, por padrão o retorno será no formato HTML. Porém como queremos

um retorno XML, alteraremos de `MediaType.APPLICATION_JSON` para `MediaType.APPLICATION_XML`, simples assim.

Depois de fazer essa alteração e executar novamente o nosso serviço receberemos o seguinte retorno: `Could not find MessageBodyWriter for response object of type: br.com.casadocodigo.javacred.entidades.Contrato` of media type: `application/xml`. Para resolver isso basta uma simples anotação no objeto que será a raiz do nosso XML. Nesse caso é a entidade `Contrato`, como a própria mensagem de erro já informou. Essa anotação é a `@javax.xml.bind.annotation.XmlRootElement`. Só isso, como podemos ver no código a seguir.

```
@Entity
@XmlRootElement
public class Contrato implements Serializable {
    ...
}
```

Executando nosso serviço novamente teremos a seguinte saída.

```
- <contrato>
  - <cliente>
    <id>2</id>
    <nome>Fulano</nome>
    <preferencial>false</preferencial>
  </cliente>
  <descricao>Parcelamento da Loja XYZ</descricao>
  <id>1</id>
  <quantidadeMeses>6</quantidadeMeses>
  <saldo>300.0</saldo>
  <valor>300.0</valor>
</contrato>
```

Figura 10.3: Exemplo de resposta XML

Ok, problema resolvido. Mas e agora, quando utilizamos XML e quando utilizamos JSON? Como vimos, o trabalho de produzir um ou outro retorno é basicamente o mesmo, o que provavelmente vai influenciar mais na escolha é a capacidade do cliente de consumir nosso serviço. Clientes modernos têm a facilidade de consumir JSON, que é uma resposta menor, mas clientes mais antigos - ou mesmo mais novos onde o padrão escolhido foi o XML - podem não entender o JSON; o que fazer nesses casos?

10.3 Retornando diferentes formatos com o mesmo serviço REST

Uma boa ideia é não fixarmos nosso tipo de resposta como fizemos até agora. Podemos dar autonomia para cada cliente consumir o

serviço com o padrão que preferir, visto que o trabalho de produzir um ou outro é o mesmo para nós. Para tanto vamos fazer uma configuração no nosso arquivo `web.xml`, que até agora deixamos em branco. O código do arquivo ficará como o seguinte:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <context-param>
    <param-name>resteasy.media.type.mappings</param-name>
    <param-value>
      html: text/html,
      json: application/json,
      xml: application/xml,
      txt: text/plain
    </param-value>
  </context-param>

</web-app>
```

Com essa configuração nós mapeamos extensões virtuais para cada tipo de retorno que estamos prontos a responder. Além disso precisamos abrir as opções presentes na anotação `@Produces` do nosso serviço. Ou adicionando os tipos possíveis, ou então removendo a anotação e deixando a opção totalmente aberta. No nosso exemplo vamos executar essa segunda opção, removendo a anotação.

Agora podemos utilizar o seguinte caminho para ter a mesma resposta REST que vimos anteriormente:

`http://localhost:8080/javacred/rest/contrato/1.json`. Para executarmos o mesmo serviço e termos um XML como resposta, basta acessarmos dessa maneira:

`http://localhost:8080/javacred/rest/contrato/1.xml`, e para devolver o

resultado do `toString()` do objeto, ainda podemos acessar da seguinte maneira: `http://localhost:8080/javacred/rest/contrato/1.txt` .

10.4 Criando um cliente de serviço REST

Na primeira parte deste livro já vimos sobre serviços REST. No entanto, como o objetivo da primeira parte era um início rápido, alguns detalhes não foram ditos, alguns exemplos de uso da API do JAX-RS não foram mostrados etc. Então agora vamos ver novamente alguns itens para podermos detalhar mais esses pontos e também vermos alguns tipos de clientes que não vimos naquela parte.

Serviços REST tem a facilidade de serem menos formais que os SOAP, e com isso conseguimos testar até mesmo pelo navegador, como se faz com as *servlets* (algo não muito comum de se fazer hoje em dia). No entanto, essa simplicidade faz com que não tenhamos ferramentas padrão como o `wsimport` , afinal, não há um equivalente ao `wsdl` que define a exata estrutura do nosso serviço.

Se quisermos algo parecido precisamos de ferramentas auxiliares como o Swagger (<https://swagger.io>) e o Swagger Codegen (<https://swagger.io/tools/swagger-codegen>), que são as mais comuns entre as ferramentas do tipo; porém, diferentemente do `wsdl` que é "nativo", para documentar um serviço REST precisamos de passos adicionais que variam dependendo da ferramenta utilizada, e por isso vamos deixar fora do escopo deste livro.

Mesmo sem um `wsdl` para usar de base, veremos que isso não dificulta nosso trabalho, pois não faltam opções de formas para construir nossos clientes conforme veremos a seguir.

Cliente chamando diretamente a URL como fazemos no navegador

A opção mais simples possível é fazer programaticamente o que fizemos via navegador. Para isso voltaremos ao serviço mais simples, que é a `CalculadoraFinanciamento`. Relembrando nossos exemplos anteriores, para chamarmos o serviço de simulação de financiamento precisávamos chamar o seguinte caminho no navegador: `http://localhost:8080/javacred/rest/financiamento/simular?valor=12000&meses=10` para simular um valor de R\$ 12.000,00 em dez meses. Executando isso no navegador temos o resultado `1320.0000000000002`, que em reais seria R\$ 1.320,00 por parcela, já que nossa taxa fixa é 1% (um por cento) ao mês.

Agora vamos fazer o mesmo no nosso código Java e ir estruturando aos poucos. Para isso, criaremos um caso de teste com JUnit, que ficará no projeto `javacred-desktop`, onde estamos colocando os demais clientes da nossa aplicação. Antes de passar para o código propriamente dito, precisamos adicionar as seguintes dependências no `pom.xml` do `javacred-desktop`:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-client</artifactId>
    <version>3.5.1.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson2-provider</artifactId>
    <version>3.5.1.Final</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Precisamos dessa configuração porque o projeto `javacred-desktop` não é um projeto que executa no servidor como o `javacred`, e com isso ele não tem automaticamente disponível a implementação da especificação JAX-RS para trabalharmos com webservices REST.

Então adicionamos manualmente o RestEasy, que como já foi dito antes, é a implementação que vem no Wildfly. Além, é claro do próprio JUnit para podermos construir nossos testes.

Com as dependências resolvidas, passemos para o código da nossa classe, que é bem simples como podemos ver a seguir.

```
import org.junit.Assert;
import org.junit.Test;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;

public class SimuladorFinanciamentoTest {

    @Test
    public void testaSimulacaoFinanciamento(){
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080/"
            + "javacred/rest/financiamento/"
            + "simular?valor=12000&meses=10");
        Response response = target.request().get();
        Double resultado = response.readEntity(Double.class);
        response.close();

        Assert.assertEquals(1320.0, resultado, 0.001);
    }
}
```

Executando esse teste veremos que temos o mesmo resultado que tivemos ao executar pelo navegador. Como podemos ver através dos *imports*, além do JUnit, só utilizamos classes padrão JAX-RS. O "jeitão" vai ser sempre esse: primeiro obtemos um `Client`, que seria o equivalente ao nosso navegador. Em seguida, configuramos o caminho através do `WebTarget`. A partir desse caminho nós executamos a requisição utilizando o método GET, e isso nos devolve uma resposta. Com a resposta em mãos, utilizamos o

método `readEntity()` para transformá-la em um tipo Java, e então podemos utilizar como quisermos.

É importante observar a linha onde fechamos a resposta, pois esse objeto não implementa a interface `AutoCloseable` que nos permitirá deixar essa responsabilidade por conta do *try-resource* do Java 7 em diante.

Montando nossa requisição utilizando parâmetros

Ok, o código funcionou, mas nós colocamos os parâmetros no próprio caminho do nosso serviço. Na prática, vamos querer passar isso como parâmetros de verdade, por isso vamos alterar nosso exemplo para o seguinte:

```
@Test
public void testaSimulacaoFinanciamentoComParametros(){
    Client client = ClientBuilder.newClient();
    WebTarget javacred =
        client.target("http://localhost:8080/javacred/rest/");

    WebTarget simuladorFinanciamento =
        javacred.path("financiamento");

    Response response = simuladorFinanciamento.path("simular")
        .queryParam("valor", 12000.0)
        .queryParam("meses", 10)
        .request().get();

    Double resultado = response.readEntity(Double.class);
    response.close();

    Assert.assertEquals(1320.0, resultado, 0.001);
}
```

Após executarmos o código o resultado é o mesmo, mas agora nós decompomos nosso `WebTarget` em objetos que representam diferentes partes do nosso caminho. O primeiro aponta para a raiz do nosso contexto REST, o segundo vai até nosso serviço, e para

isso faz uso do método `path()` , que adiciona uma parte ao caminho que já estava definido antes. O terceiro aponta para o método do nosso serviço, e na sequência já passa os parâmetros via `queryParam(nome, valor)` . O restante é igual ao exemplo anterior. Essa construção é mais parecida com a de uma `Query` da JPA que depois recebe os parâmetros. Nosso código melhorou nessa segunda versão, mas ainda há espaço para melhorias.

Porém, em vez de avançar mais nesse cliente, vamos construir um cliente para o serviço `ContratoBean` , que recebe e devolve objetos complexos. Assim já vemos como fazer clientes que recebem e devolvem objetos do nosso modelo, e não só tipos básicos do Java; e ao mesmo tempo fazemos um cliente mais polido.

O primeiro método que vamos testar desse serviço é o que devolve um `Contrato` a partir do seu identificador. A diferença é que aqui em vez de informarmos o parâmetro no estilo *query param*, que utiliza `?nome=valor` , vamos utilizar o parâmetro passado no caminho, que acaba ficando visualmente mais agradável. Por exemplo, para acessar os dados do `Contrato` com identificador `1` , utilizamos o seguinte caminho: `http://localhost:8080/javacred/rest/contrato/1` , e não `http://localhost:8080/javacred/rest/contrato?id=1` .

Para tanto criaremos o seguinte cliente, em uma classe separada:

```
package br.com.casadocodigo.javacred.restclient;

import org.junit.Assert;
import org.junit.Test;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class ContratoBeanTest {
```

```

@Test
public void deveBuscarContratoPorId() {

    Client client = ClientBuilder.newClient();
    WebTarget javacred =
        client.target("http://localhost:8080/javacred/rest");

    WebTarget contratoBean = javacred.path("contrato");

    Response response = contratoBean.path("{id}")
        .resolveTemplate("id", 1)
        .request(MediaType.APPLICATION_JSON)
        .get();

    String json = response.readEntity(String.class);

    Assert.assertEquals(200, response.getStatus());

    System.out.println(json);

    response.close();

}
}

```

E para refrescar nossa memória, segue a parte do `ContratoBean` que estamos utilizando:

```

@Stateless
@Path("/contrato")
public class ContratoBean {
    ...

    @GET
    @Path("/{identificador}")
    public Contrato buscarContrato(@PathParam("identificador")
        Integer identificador){
        ...
    }
}

```

```
    //outros métodos que lembraremos a seguir  
}
```

Assim como no exemplo anterior nós separamos o caminho base do contexto REST da aplicação do caminho do serviço, e depois a partir dele acessamos o método, também utilizando o método `path`. A diferença aqui é que para visualizar um contrato, não precisamos de um caminho específico para o método `buscarContrato`, apenas especificamos o identificador, que no nosso exemplo foi o "id" `1`. Para tanto, em vez de utilizarmos o método `queryParams`, utilizamos o `resolveTemplate` como vimos no exemplo. Além disso, pelo fato do nosso serviço não fixar um tipo de troca de informação, especificamos que utilizaremos *JSON* na nossa requisição. É o equivalente a usar `".json"` no final da URL como vimos anteriormente neste capítulo.

Vale registrar que podemos misturar parâmetros do tipo `@PathParam` com `@QueryParam`, e respectivamente utilizar os métodos `resolveTemplate` e `queryParams` no mesmo cliente.

Analisando o resultado do nosso teste no console, vemos que temos o JSON que é a resposta do serviço. Agora veremos diversas formas de interagir com essa resposta.

Manipulando o JSON retornado

O exemplo anterior devolve uma `String` que é o mesmo JSON que antes víamos no navegador. Para manipular essa resposta utilizaremos a biblioteca *Jackson*, que já foi adicionada ao nosso projeto logo antes.

Agora vamos analisar a resposta do nosso serviço. Consideremos que o contrato com "id" igual a `1` tenha o seguinte conteúdo.

```
{
  id: 1,
  descricao: "Parcelamento da Loja XYZ",
  valor: 300,
  quantidadeMeses: 6,
  taxaMensal: null,
  saldo: 300,
  cliente: {
    id: 2,
    nome: "Fulano",
    preferencial: false
  }
}
```

Figura 10.4: Conteúdo do Contrato com ID igual a 1 (um)

PARA FACILITAR O AMBIENTE DE TESTES

Para tentar garantir que teremos um `Contrato` com o `id` igual a `1`, podemos usar a classe `StartupConfig` que criamos no início do capítulo 7. O que vamos fazer é um código de inicialização que verifica se há no banco um contrato com esse `id` e caso não haja inserimos um. Até então a classe estava vazia, agora terá o seguinte conteúdo.

```
@PersistenceContext
private EntityManager em;

@Inject
private ContratoBean contratoBean;

@PostConstruct
public void startup(){

    if(contratoBean.buscarContrato(1) == null){

        em.createNativeQuery("truncate table Contrato")
            .executeUpdate();

        Contrato contrato = new Contrato(
            new Cliente("Fulano", true));
        contrato.setDescricao("Contrato teste");
        contrato.setValor(12_000.0);
        contratoBean.salvar(contrato);

    }
}
```

Executamos um `truncate` na tabela pois só assim o incremento do `id` é zerado. Caso o banco utilizado não suporte chave do tipo *identity*, essa abordagem pode não funcionar pois estará sendo utilizada uma *sequence* ou algo equivalente. Mas como esse é apenas um código para auxiliar nos nossos testes, não precisamos esgotar todas as possibilidades.

Em linguagens como o Groovy, o suporte a JSON é nativo. Como aqui estamos usando Java, como podemos acessar a descrição do contrato ou o nome do cliente? Apesar de haver diversas opções, usaremos o *Jackson* como dito há pouco. Indo agora ao nosso exemplo, adicionando mais um teste na classe `ContratoBeanTest`, onde receberemos uma resposta JSON e a leremos de forma estruturada:

```
...
import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.map.ObjectMapper;
...

@Test
public void deveRecuperarDescricaoENome() throws IOException {

    Client client = ClientBuilder.newClient();
    WebTarget javacred = client
        .target("http://localhost:8080/javacred/rest");

    WebTarget contratoBean = javacred.path("contrato");

    Response response = contratoBean.path("{id}")
        .resolveTemplate("id", 1)
        .request(MediaType.APPLICATION_JSON)
        .get();

    String json = response.readEntity(String.class);

    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode rootNode = objectMapper.readTree(json);

    String descricao = rootNode.path("descricao").asText();
    String nomeCliente = rootNode.path("cliente")
        .path("nome").asText();

    Assert.assertNotNull(descricao);
    System.out.println(descricao);
}
```

```
Assert.assertNotNull(nomeCliente);
System.out.println(nomeCliente);

response.close();
}
```

Primeiro utilizamos a classe `ObjectMapper` para mapear o texto que representa o JSON para o objeto `JsonNode` que é o JSON estruturado em forma de objeto, e a partir dele podemos navegar no seu conteúdo. A forma de navegar utilizando o `JsonNode` é bem parecido com o DOM do XML. Há como iterar coleções, modificar o objeto e muito mais; mas não precisaremos fazer isso no nosso exemplo pois nosso intuito nesse teste foi somente navegar pelos elementos do JSON, como a descrição do contrato e o nome do cliente.

Quando percebermos que queremos algo mais complexo como percorrer uma lista ou ler vários atributos, passaremos para uma abordagem como a do próximo tópico.

Transformando a resposta JSON em objetos Java

Assim como nosso serviço transforma um objeto Java em um documento JSON, faremos o caminho inverso, transformando o JSON em um objeto Java. Como no nosso projeto cliente tem uma referência para o projeto onde estão os serviços, seria até possível utilizar as mesmas classes `Contrato` e `Cliente` para montar esses objetos. Porém, como muitas vezes não teremos essas classes disponíveis, vamos criar outras classes no projeto `javacred-desktop` para mostrar que não precisa ser a mesma classe do servidor para funcionar. Faremos aqui o mesmo que fizemos no projeto `javacred-corretora` com o `OrdemTO`.

Com base na estrutura do nosso JSON, criamos duas novas classes, chamadas `MeuContrato` e `MeuCliente`, só para ficar bem diferenciadas das mesmas classes existentes no servidor. A estrutura é basicamente a mesma do servidor:

```

public class MeuContrato {

    private Integer id;
    private String descricao;
    private Double valor;
    private Integer quantidadeMeses;
    private Double taxaMensal;
    private Double saldo;

    private MeuCliente cliente;

    //getters, setters e toString gerados pela IDE
}

public class MeuCliente {
    private Integer id;
    private String nome;
    private boolean preferencial;

    public MeuCliente() {
    }

    public MeuCliente(String nome) {
        this.nome = nome;
    }

    //getters, setters e toString gerados pela IDE
}

```

Basta agora criar um novo teste para em vez de recuperar a resposta no formato de `String`, fazer isso com o tipo `MeuContrato`.

```

@Test
public void deveConverterRespostaEmObjetosJava(){

    Client client = ClientBuilder.newClient();
    WebTarget javacred = client
        .target("http://localhost:8080/javacred/rest");

    WebTarget contratoBean = javacred.path("contrato");
}

```

```

Response response = contratoBean.path("{id}")
    .resolveTemplate("id", 1)
    .request(MediaType.APPLICATION_JSON)
    .get();

MeuContrato contrato = response
    .readEntity(MeuContrato.class);
String descricao = contrato.getDescricao();
String nomeCliente = contrato.getCliente().getNome();

Assert.assertNotNull(descricao);
System.out.println(descricao);

Assert.assertNotNull(nomeCliente);
System.out.println(nomeCliente);

response.close();
}

```

Podemos agora acessar normalmente os atributos a partir das classes Java, muito melhor que ficar acessando propriedade por propriedade através do método `path()` da classe `JsonNode`. Mas ainda há bastante para vermos sobre clientes REST.

Recebendo coleções como retorno de serviços REST

Já vimos que nos nossos clientes REST, para receber um determinado tipo de objeto basta utilizar o método `readEntity()` passando a classe que representa o tipo que será retornado. Mas e quando temos um tipo que usa *generics* como uma coleção? Para isso a própria API da especificação JAX-RS tem o tipo `javax.ws.rs.core.GenericType`. Sua utilização é bem simples, como podemos ver a seguir.

```

...
import javax.ws.rs.core.GenericType;
...
@Test
public void deveLidarComColecoes() throws Exception {

```

```

Client client = ClientBuilder.newClient();
WebTarget javacred = client
    .target("http://localhost:8080/javacred/rest");

WebTarget contratoBean = javacred.path("contrato");

Response response = contratoBean.path("todos")
    .request(MediaType.APPLICATION_JSON).get();

List<MeuContrato> contratos = response
    .readEntity(new GenericType<List<MeuContrato>>() {});
response.close();

contratos.forEach(Assert::assertNotNull);
contratos.forEach(System.out::println);
}

```

Como dito antes, a utilização é bem simples, e apesar de poder parecer um pouco estranha, é a forma de se lidar com tipos genéricos desde o Java 5.

10.5 Criando um cliente JS para um serviço REST

Algo bastante interessante é que além dos clientes que já fizemos em Java, podemos criar clientes diretamente em JavaScript. Essa, porém, não é um funcionalidade da especificação JAX-RS, e sim algo da implementação dela que estamos usando, o RESTEasy.

Ferramentas atuais como JSF e outros frameworks web possuem suporte nativo a AJAX, que nada mais é do que a capacidade de, via JavaScript, invocar lógicas de negócio que estão no servidor. Porém, em um framework web as operações realizadas com AJAX costumam ter como objetivo gerar um resultado HTML para atualizar uma parte da tela, já o que veremos aqui é um pouco diferente.

Voltaremos à tela de cadastro de contratos que foi vista no início do capítulo 7. Lá nós apenas cadastrávamos os contratos, agora poderemos fazer uma busca do contrato pelo seu número, e em seguida poderemos alterá-lo. Essa sem dúvidas é uma implementação bastante simples, mas em vez de utilizar o nosso framework MVC, faremos isso via *JS* acessando diretamente o `ContratoBean`, que é o EJB que contém as regras de negócio envolvendo contratos. Na próxima seção teremos algumas considerações sobre a questão de *design* envolvida aqui, mas por enquanto seguiremos nosso exemplo.

O método de busca aqui não tem nada complexo, o número do contrato será o próprio ID (chave primária), e como nosso intuito aqui não é explorar as possibilidades de interface com o usuário, o campo de buscar será uma caixa de entrada também *JS* (`prompt`). Para isso voltemos à tela `emprestimo.xhtml`, que é onde se dá o cadastro do contrato e adicionaremos o seguinte trecho de código:

```
<h:head>
  <script lang="javascript" src="./rest-js"/>
  <h:outputScript library="js" name="jquery-3.4.1.min.js" />
  <h:outputScript>
    function buscar(){

      var id = prompt('Informe o ID do contrato')

      var contrato =
        ContratoBean.buscarContrato({identificador:id})

      $('#id').val(contrato.id)
      $('#descricao').val(contrato.descricao)
      $('#valor').val(contrato.valor)
      $('#qtdeMeses').val(contrato.quantidadeMeses)
      $('#taxa').val(contrato.taxaMensal)
      $('#cliente').val(contrato.cliente.id)

    }
  </h:outputScript>
</h:head>
```

Aqui estamos utilizando a API do RESTEasy e também o JQuery. Perceba que para acessar o método do `ContratoBean` apenas o fizemos como se chamássemos um método estático em uma classe Java. E como nesse método definimos um parâmetro nomeado, ao chamá-lo nós especificamos esse nome. Para ficar mais simples segue a definição do método apenas como recapitulação:

```
...
public class ContratoBean {
    ...

    @GET
    @Path("/{identificador}")
    public Contrato buscarContrato(@PathParam("identificador")
        Integer identificador){
        ...
    }

    ...
}
```

Porém, para esse código JS funcionar, temos que adicionar a biblioteca JQuery, que é um processo bem simples. No nosso exemplo apenas colocamos o arquivo `jquery-3.4.1.min.js` dentro da pasta `src/main/webapp/resources/js`. Como estamos usando Maven, na prática a raiz do nosso contexto web é a pasta `webapp`. E dentro da raiz do contexto web, temos a pasta `resources`, que é onde o JSF busca os arquivos de recursos (js, css, imagens...), e dentro dela criamos a pasta `js`, mas que poderia ter qualquer nome. O nome dessa última pasta deve ser o mesmo utilizado na propriedade `library` na tag `<h:outputScript library="js" ...>`.

Mas o mais importante mesmo é a API do RESTEasy, que adicionamos através do trecho `<script lang="javascript" src="./rest-js"/>`. Esse caminho especificado no `src` do nosso JS nada mais é que o padrão da *Servlet* que precisamos configurar no `web.xml` para isso funcionar:

```

<servlet>
  <servlet-name>RETEasy JSAPI</servlet-name>
  <servlet-class>
    org.jboss.resteasy.jsapi.JSAPIServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>RETEasy JSAPI</servlet-name>
  <url-pattern>/rest-js</url-pattern>
</servlet-mapping>

```

Essa *servlet* no entanto não vem por padrão no RETEasy, e sim como uma dependência extra. Para adicioná-la ao nosso projeto precisamos da seguinte dependência no `pom.xml` do projeto `javacred`.

```

<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jsapi</artifactId>
  <version>3.11.2.Final</version>
</dependency>

```

Com isso, nosso código JS estará apto para chamar o método de negócio, mas quando for preencher a tela com os dados de retorno teremos um detalhe. Por padrão, quando não especificamos manualmente os IDs para cada componente JSF, são gerados IDs sequenciais. Mas agora como vamos querer manipulá-los, precisaremos adicionar esses IDs:

```

...
<h:body>
  <h:form prependId="false">

    <h:inputHidden id="id" value="#{contrato.id}"/>
    <input type="button" value="Buscar" onclick="buscar()"/>

    <h:panelGrid columns="2">
      Cliente:
      <h:selectOneMenu id="cliente"

```



```

value="#{contrato.cliente}"
converter="#{clienteConverter}">

    <f:selectItem
itemLabel="-- Selecione um cliente --"
noSelectionOption="true"/>

    <f:selectItems value="#{clientes}" var="c"
itemValue="#{c}"
itemLabel="#{c.nome}"/>
</h:selectOneMenu>

Descrição:
<h:inputText id="descricao"
value="#{contrato.descricao}"/>

Valor:
<h:inputText id="valor" value="#{contrato.valor}"/>

Qtde Meses:
<h:inputText id="qtdeMeses"
value="#{contrato.quantidadeMeses}"/>

Taxa Mensal:
<h:inputText id="taxa"
value="#{contrato.taxaMensal}"/>

</h:panelGrid>

<h:commandButton value="Registrar"
action="#{emprestimoController.contratar()}" />
</h:form>
</h:body>
...

```

Além de especificar os IDs, apenas por questão de estética, pedimos para o JSF não concatenar o ID do formulário nos IDs dos elementos dele, e fizemos isso através do atributo `prependId="false"` na tag `<h:form>`.

Continuando no nosso código, logo após a abertura do formulário adicionamos duas coisas: a primeira é um campo oculto para guardar a chave primária do nosso `Contrato`, pois sem isso em vez de atualizar um contrato existente, estaríamos criando um novo, e a segunda é o botão que faz a chamada para a função JS que criamos há pouco.

Com isso, agora temos uma nova funcionalidade, que é a busca de contratos totalmente feita utilizando JavaScript. Mas agora que terminamos a parte de código, voltaremos à questão de *design* que deixamos em aberto para não desviar nossa atenção do exemplo.

10.6 Acessar diretamente a camada de negócios a partir da visualização é uma boa?

Antes de mais nada, vale lembrar que mesmo utilizando clientes puramente JS como o React, há separação (ou deve haver) entre a visualização e o controle. Então, independentemente da tecnologia utilizada, essa é mais uma decisão de *design*, assim como diversas outras que precisamos tomar a todo momento quando desenvolvemos nossas aplicações.

Já tivemos uma questão como essa na seção **6.3. Usar o padrão DAO, Service Bean + Controller, ou tudo junto?**, quando discutimos o uso direto de JPA em relação ao uso de DAOs, e também quando discutimos o padrão *Open Session In View* na seção **6.5. Lidando com LazyInitializationException ou equivalente**.

Então como nas ocasiões anteriores já falamos um pouco sobre a simplicidade em relação a algo mais "ortodoxo", vamos ao foco da questão: pular a camada de controle faz sentido? Novamente a resposta pode desafiar as cartilhas mais tradicionais, e mais uma

vez o que vale é a maturidade que você ou seu time tem. Não é bom flexibilizar demais se correr o risco de virar bagunça.

Minha sugestão é que se você não conseguir explicar de uma maneira simples e objetiva quando usar de um jeito ou de outro, defina uma regra única, por exemplo sempre passar pelo controlador.

Agora se você percebe que esse tipo de serviço - como buscar um contrato pelo código - não tem nada que deveria ser feito pelo controlador (ele funcionaria como mero despachante), não há pecado nenhum em encurtar a etapa.

Só não se esqueça que se você trabalha com um time, procure sempre pesar junto com ele qual a forma mais intuitiva, já pensando na manutenção. Muitas vezes esse fator nos faz dar um passo atrás e usar algo mais "quadrado".

Para fechar o capítulo

Neste capítulo vimos como trabalhar com WebServices REST utilizando JAX-RS do Java EE. Apesar da facilidade de se trabalhar com EJBs remotos, muitas vezes precisamos de algo que funcione em diferentes linguagens, ou simplesmente queremos expor nossos serviços de uma forma diferente, como REST.

Independentemente do motivo, vimos que podemos exibir nossos serviços de mais de uma maneira: EJB, como já tínhamos visto, REST, e até mesmo implementar clientes diretamente em JavaScript.

No próximo capítulo veremos uma nova forma que o Java EE 8 introduziu para nos permitir usar *Server-Sent Events* e criar clientes reativos.

CAPÍTULO 11

Server-Sent Events ou SSE (JAX-RS 2.1, Java EE 8)

Apesar de estarmos lidando com Java EE 8, nem tanta coisa assim mudou em relação ao Java EE 7. Por exemplo, a versão de EJB permanece sendo a 3.2, mas a versão do JAX-RS mudou, dando-nos basicamente duas principais novidades, o SSE, que veremos agora, e a possibilidade de criação de clientes reativos, que veremos na próxima seção.

O SSE, ou envio de mensagens pelo servidor, é uma alternativa ao *pooling*, que é quando ficamos em um laço fazendo uma requisição a cada intervalo de tempo para ver se temos novidades. Vamos tomar como exemplo a funcionalidade de consultar a cotação de uma determinada moeda.

No projeto `javacred-corretora`, que usamos nos capítulos 1 e 2, vimos algo parecido, mas lá não usamos SSE. Criamos um mecanismo de cache para proteger nosso serviço do excesso de requisições "desnecessárias", uma vez que o dado não tinha mudado.

Agora vamos criar um exemplo onde nosso cliente pode se registrar para receber as cotações de uma determinada moeda e não precisa ficar consultando de tempo em tempo. Com isso não precisaremos aqui fazer uma gestão de cache, pois obrigatoriamente nossos clientes precisarão saber lidar com os eventos enviados pelo servidor.

QUAL A DIFERENÇA ENTRE SSE E WEBSOCKET?

Tanto *Server-Sent Events* quanto *WebSockets* permitem que o servidor envie mensagens para o cliente, a diferença é que no caso dos *WebSockets* a comunicação é de mão dupla, como o que usaríamos para implementar um chat, por exemplo; enquanto o SSE é uma comunicação somente do servidor para o cliente.

Além disso, *WebSocket* é um protocolo próprio rodando em cima do TCP, já o SSE roda em cima do HTTP, que é um protocolo mais conhecido, e então, além de mais fácil de desenvolver (não precisamos tratar métodos `onOpen`, `onMessage` etc.), é mais fácil de colocar em produção, pois é bem conhecido por *firewalls*, *proxies*, *load balancers* etc.

Para iniciar nosso exemplo vamos criar um novo serviço, `GerenciadorCotacoesBean`, com o seguinte código:

```
import javax.ejb.Singleton;
import javax.ws.rs.sse.Sse;
import javax.ws.rs.sse.SseBroadcaster;
import javax.ws.rs.sse.SseEventSink;
...

@Singleton
@Path("/acompanhar-cotacao")
public class GerenciadorCotacoesBean {

    @Context
    private Sse sse;

    private SseBroadcaster sseBroadcaster;

    @PostConstruct
    public init() {
        //cria o SseBroadcaster a partir do Sse
        this.sseBroadcaster = sse.newBroadcaster();
    }
}
```

```

    }

    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS)
    public void registrar(@Context SseEventSink eventSink) {
        sseBroadcaster.register(eventSink);
    }

    ...
}

```

Até aqui fizemos o código necessário para o registro do nosso cliente. O que temos de novo aqui são os objetos do pacote `javax.ws.rs.sse`. O primeiro é o `Sse`, que é o ponto de partida para criar os outros objetos que vamos usar depois, como o `SseBroadcaster`, que é quem sabe fazer o *broadcast* para todos os clientes registrados. A instância do `SseBroadcaster` é gerenciada pela aplicação, e não pelo JAX-RS, por isso estamos aqui usando um bean `@Singleton`. Temos também aqui o `SseEventSink`, que é o canal para envio de mensagens para os clientes. Para ter acesso a esse objeto o injetamos como parâmetro do nosso método de "registro".

O próximo passo é adicionar na nossa classe os métodos que enviam os eventos, e para isso vamos usar o serviço de timer do EJB:

```

...
public class GerenciadorCotacoesBean {

    ...

    //a cada 5 segundos
    @Schedule(hour = "*", minute = "*", second = "*/5")
    public void geradorCotacoes(){

        Cotacao cotacao = geraCotacao();

        OutboundSseEvent event = sse.newEventBuilder()
            .data(cotacao)
            .mediaType(MediaType.APPLICATION_JSON_TYPE)

```

```

        .build();
        sseBroadcaster.broadcast(event);

    }

    private Cotacao geraCotacao(){
        Random r = new Random();
        Cotacao.Moeda[] moedas = Cotacao.Moeda.values();
        Cotacao.Moeda moeda = moedas[r.nextInt(moedas.length)];
        return new Cotacao(1, r.nextDouble() * 10,
            moeda, Cotacao.Moeda.REAL);
    }
}

```

Criamos um timer que roda a cada 5 (cinco) segundos, gera uma `Cotacao`, em seguida cria um evento (`OutboundSseEvent`) e então faz seu *broadcast*. Para criar esse evento usamos o *builder* do objeto `Sse` para que pudéssemos especificar coisas como o corpo e o *media type*. Mas para casos mais simples podemos criar diretamente um evento com um corpo texto (`String`) via o método `Sse.newEvent(String data)`.

A geração da `Cotacao` simplesmente seleciona randomicamente a moeda e o valor da cotação, deixando fixa somente a moeda destino que é o Real. Assim a cotação pode ser de qualquer moeda para a nossa (inclusive o próprio Real, mas como é um código de exemplo não tem problema).

O código da classe `Cotacao` é o seguinte:

```

@XmlRootElement
public class Cotacao {

    public enum Moeda {DOLAR, EURO, REAL}

    private Date data;
    private Integer quantidade;
    private Double valor;
    private Moeda moedaCotada;
    private Moeda moedaValor;
}

```

```

public Cotacao(){}
public Cotacao(Integer quantidade, Double valor,
                Moeda moedaCotada, Moeda moedaValor) {

    this.data = new Date();
    this.quantidade = quantidade;
    this.valor = valor;
    this.moedaCotada = moedaCotada;
    this.moedaValor = moedaValor;
}

//getters e setters
}

```

É uma classe bastante simples, que serve somente como estrutura de dados para devolvermos ao nosso cliente. E por falar nele, falta agora somente a implementação do cliente, que vamos criar no projeto `javacred-desktop` apenas para ficar junto dos demais. Vamos criar uma classe que pode se chamar `CotacaoMoedasSSETest` e colocar o seguinte dentro dela:

```

@Test
public void testeSimplesSSE(){

    Client client = ClientBuilder.newClient();
    WebTarget target = client.target(
        "http://localhost:8080/javacred/rest/acompanhar-cotacao");

    try (SseEventSource source = SseEventSource
        .target(target).build()) {

        source.register(System.out::println);
        source.open();
        Thread.sleep(20_000);

    } catch (InterruptedException e) {

    }

}

```


O processo da criação do cliente é similar aos demais clientes REST, porém usamos o método `SseEventSource.target(WebTarget)` para obter o builder para o objeto `SseEventSource` propriamente dito. Como esse objeto também implementa `AutoCloseable`, podemos usar o *try-resource* do Java 7 em diante, o que nos dispensa de chamar o método `close()` depois do `sleep()`.

Estando com o `SseEventSource`, nós registramos o consumidor de eventos (método que recebe `InboundSseEvent`), que no caso foi o próprio `System.out.println()`. Aqui usamos direto a sintaxe do Java 8. Sem ela, o código ficaria bem verboso, pois precisaríamos de uma *inner class*.

Depois que registramos o consumidor de eventos, podemos abrir o canal para receber as mensagens ou eventos. Repare logo em seguida no uso do `sleep()`, que é necessário para que nosso consumidor tenha tempo para ir recebendo nossas mensagens, que no nosso exemplo são geradas a cada 5 (cinco) segundos. Isso é necessário porque o `open()` não é bloqueante, então se simplesmente deixarmos o método seguir, a execução termina e não recebemos qualquer mensagem.

Como já vimos o suficiente da parte servidora aqui, vou deixar apenas no GitHub uma versão aprimorada do código onde o cliente pode escolher a `Moeda` cuja cotação ele quer acompanhar. Assim temos uma referência boa de como implementar algo mais elaborado, e ao mesmo tempo mantemos o conteúdo do livro um pouco mais enxuto.

Por fim, vamos melhorar apenas nosso cliente, para demonstrar outros consumidores que podemos registrar, além do que trata o "caminho feliz":

```
@Test
public void testaAcompanhamentoEventos()
    throws InterruptedException {

    Client client = ClientBuilder.newClient();
```

```

WebTarget target = client.target(
    "http://localhost:8080/javacred/rest/acompanhar-cotacao");

SseEventSource eventSource = SseEventSource
    .target(target).build();

System.out.println("Teste executado na thread: " +
    Thread.currentThread().getName());

eventSource.register(this::processaEvento,
    this::processaExcecao,
    this::execucaoFinalizada);

eventSource.open();
Thread.sleep(20_000);
eventSource.close();
}

public void processaEvento(InboundSseEvent sseEvent){
    System.out.println("Evento recebido na thread: " +
        Thread.currentThread().getName());
    System.out.println("Dado JSON: " + sseEvent.readData());
    System.out.println("JSON em Objeto: " +
        sseEvent.readData(Cotacao.class,
            MediaType.APPLICATION_JSON_TYPE));
}

public void processaExcecao(Throwable t){
    System.err.println("Erro: " + t);
}

public void execucaoFinalizada(){
    System.out.println("Execucao finalizada.");
}

```

Aqui mais uma vez usamos a sintaxe do Java 8, se não o código ficaria ilegível (um verdadeiro "callback hell"), com três *inner classes* para podermos registrar cada um dos consumidores de eventos: o primeiro que processa a mensagem ou evento recebido com sucesso; o segundo que processo os erros ou exceções, e por fim o

terceiro que simplesmente executa uma lógica de finalização (desconexão, liberação de recursos etc.).

Além disso, dessa vez foi colocada uma saída de texto apenas para mostrar que os consumidores de eventos executam numa `Thread` separa da linha principal, de forma não bloqueante e, por isso, como já vimos antes, fazemos uso do `sleep()` para nosso exemplo funcionar.

Vale observar as duas formas de consumir o evento que utilizamos dentro do método `processaEvento(InboundSseEvent)`. Na primeira, apenas usamos o método `InboundSseEvent.readData()` que devolve uma `String`, que no caso é o texto JSON que produzimos no servidor. Na segunda forma, usamos o método `readData(Class, MediaType)`, que assim como nos outros clientes REST permite que transformemos diretamente nosso JSON em um objeto Java.

Já vimos nos exemplos anteriores que não há qualquer dependência binária entre a classe usada para dar origem ao JSON e a classe usada para montar o objeto Java do lado do cliente, tanto que criamos classes diferentes para demonstrar isso (`MeuContrato` e `MeuCliente`). Aqui, apenas para não termos que criar uma classe "MinhaCotacao", utilizamos a própria classe `Cotacao` que está disponível no *class path* do projeto `javacred-desktop`, mas poderíamos fazer uma nova classe sem mudar o resultado.

11.1 Reactive Client API (JAX-RS 2.1, Java EE 8)

Outra novidade do JAX-RS 2.1 (presente no Java EE 8), foi a possibilidade de criarmos clientes reativos. Esse tipo de cliente é fruto na verdade da programação reativa, que é uma forma de desenvolvermos de forma não bloqueante. Então em vez da programação tradicional, onde chamamos um método e esperamos

uma resposta, pedimos uma execução e registramos um *callback* que vai tratar a resposta quando ela estiver pronta.

Essa abordagem nos dá um ganho em escalabilidade, uma vez que menos recursos podem servir a uma quantidade maior de requisições agora que não estarão mais parados esperando outro trecho de código terminar seu serviço. O lado negativo é que pode levar ao "callback hell" citado no final da seção anterior; mas aqui a coisa fica muito mais feia se for feita sem cuidado.

Para colocar um pouco de ordem na casa, foram definidos alguns padrões de como fazer esse tipo de processamento, e isso foi introduzido no Java 8 e agora pode ser usado no Java EE 8, do qual o JAX-RS 2.1 faz parte. Mas é claro que não precisamos rodar dentro de um servidor Java EE para isso, podemos usar em aplicações JAX-RS puras desde que usemos essa última versão.

Todo o desenvolvimento é feito com base na interface

`java.util.concurrent.CompletionStage` que permite o encadeamento de chamadas assíncronas no Java. Antes disso, tínhamos apenas a interface `Future`, do mesmo pacote, que permitia que tivéssemos um ponteiro para uma execução que retornaria um valor no futuro, quando terminasse. Mas se quiséssemos usar esse valor a ser retornado junto com outra execução assíncrona, tínhamos que esperar a primeira execução terminar, o que nos limitava bastante.

Pode parecer complicado, mas vamos usar um exemplo que vai facilitar bastante nosso entendimento. Na seção anterior vimos um serviço que nos permitia acompanhar a cotação de uma determinada moeda. Agora vamos criar um novo serviço que também vai trabalhar com cotação de moedas, mas de forma assíncrona para entendermos o uso dos clientes reativos.

OBSERVAÇÃO ACERCA DO DESENVOLVIMENTO ASSÍNCRONO NESSE E NOS PRÓXIMOS CAPÍTULOS

Os dois próximos capítulos do livro são focados no uso de métodos assíncronos e paralelismo para ganho de performance, então aqui não vamos detalhar algumas coisas que serão melhor apresentadas a seguir.

O intuito aqui é manter a relação dos clientes reativos com o capítulo que trata de WebServices, e não duplicar conteúdo.

Nesse novo exemplo vamos poder consultar a cotação de Dólar para Real, e a cotação de Euro para Dólar, mas para conseguir a cotação de Euro para Real teremos que combinar os dois serviços, e faremos isso da forma tradicional e da forma reativa para vermos a diferença. Mas vamos começar pelo nosso serviço:

```
import javax.enterprise.concurrent.ManagedExecutorService;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
...

@Path("/cotacao")
@Produces(MediaType.APPLICATION_JSON)
public class CotacaoMoedasBean {

    @Resource
    private ManagedExecutorService mes;

    @GET
    @Path("/dolar/{quantidade}")
    public void cotarDolar(@PathParam("quantidade") Integer qtde,
        @Suspended AsyncResponse resp){

        cotar(Cotacao.Moeda.DOLAR, Cotacao.Moeda.REAL, qtde, resp);

    }
}
```

```

@GET
@Path("/euro/{quantidade}")
public void cotarLibra(@PathParam("quantidade") Integer qtde,
    @Suspended AsyncResponse resp){

    cotar(Cotacao.Moeda.EURO, Cotacao.Moeda.DOLAR, qtde, resp);

}

private void cotar(Cotacao.Moeda cotar, Cotacao.Moeda destino,
    Integer quantidade, AsyncResponse response){

    String threadPrincipal = Thread.currentThread().getName();

    System.out.printf("Thread principal >> %s em %s :: %s \n",
        cotar, destino, threadPrincipal);

    mes.execute(() -> {
        try {
            String threadFilha = Thread.currentThread().getName();

            System.out.printf("Thread filha >> %s em %s :: %s \n",
                cotar, destino, threadFilha);

            TimeUnit.SECONDS.sleep(3);

            Double valorUnitario = new Random().nextDouble() * 10;
            Cotacao cotacao = new Cotacao(quantidade,
                valorUnitario * quantidade, cotar, destino);

            System.out.printf("Thread filha << %s em %s :: %s \n",
                cotar, destino, threadFilha);

            response.resume(Response.ok(cotacao).build());

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });

    System.out.printf("Thread principal << %s em %s :: %s \n",

```

```

        cotar, destino, threadPrincipal);
    }
}

```

Começamos nosso serviço injetando o `ManagedExecutorService`, que veremos de forma bem detalhada no capítulo seguinte, mas por ora basta sabermos que ele possibilita executar um código assíncrono dentro do servidor. Em programação desktop, seria o equivalente a lançar uma `Thread` manualmente, mas como não podemos fazer isso diretamente no servidor, usamos esse objeto para fazer o mesmo através dele.

Fora isso, o que temos de novidade é o `@Suspended AsyncResponse`, que nos permite preparar uma resposta para o cliente sem ficar bloqueado esperando algum processamento.

Dentro do método `cotar()` temos a execução assíncrona que fica esperando 3 (três) segundos antes de retornar uma cotação aleatória. Novamente usamos a sintaxe do Java 8 para não termos que criar uma `Runnable` anônima, que deixaria o código mais "feinho".

O que vimos até aqui nós já tínhamos na versão anterior de JAX-RS, a diferença vai estar na criação do cliente. Mesmo assim vamos começar pelo cliente mais trivial, criando uma classe chamada `CotacaoMoedasBeanTest` e colocando o seguinte dentro dela:

```

@Test
public void testaCotacaoDolar() throws Exception {

    Client client = ClientBuilder.newClient();
    WebTarget cotacaoMoedasBean = client.target(
        "http://localhost:8080/javacred/rest/cotacao/dolar")
        .path("{quantidade}")
        .resolveTemplate("quantidade", 50);

    Future<Response> response = cotacaoMoedasBean
        .request(MediaType.APPLICATION_JSON)

```

```

        .async()
        .get();

String json = response.get().readEntity(String.class);
System.out.println(json);

}

```

A diferença desse trecho de código para o que já fazíamos em relação aos outros clientes é que aqui antes do `get()` usamos o método `async()`, e assim em vez de receber um `Response`, recebemos um `Future<Response>`. Até aqui também já podíamos fazer no JAX-RS 2.0, e o funcionamento do `Future` é o mesmo desde o Java 5: ao chamar o método `get()` ficamos bloqueados até que o processamento em segundo plano termine. Assim, o que era assíncrono passa a ser síncrono.

Só não vamos confundir o primeiro `get()` que se refere ao método GET do HTTP, com o segundo `get()` que é um método de `Future`.

Feito o primeiro exemplo, vamos agora recuperar a cotação do Euro em Dólar, e então pegar a cotação do Dólar em Real para podermos ter a cotação do Euro em Real.

```

@Test
public void testaCotacaoEuroEmReal() throws Exception {

    Client client = ClientBuilder.newClient();
    WebTarget cotacaoMoedasBean = client.target(
        "http://localhost:8080/javacred/rest/cotacao");

    Future<Response> response = cotacaoMoedasBean
        .path("euro/{quantidade}")
        .resolveTemplate("quantidade", 50)
        .request(MediaType.APPLICATION_JSON)
        .async()
        .get();

    Cotacao euroDolar = response.get().readEntity(Cotacao.class);
    System.out.println(euroDolar);
}

```



```

response = cotacaoMoedasBean
    .path("dolar/{quantidade}")
    .resolveTemplate("quantidade", 50)
    .request(MediaType.APPLICATION_JSON)
    .async()
    .get();

Cotacao dolarReal = response.get().readEntity(Cotacao.class);
System.out.println(dolarReal);

Double euroEmReal =
    euroDolar.getValor() * dolarReal.getValor();

System.out.println(euroEmReal);

}

```

Esse novo exemplo é também uma chamada de cliente REST como diversas que já fizemos, então não tem anda de novo. Por mais que alteremos a ordem de alguns métodos, como cada chamada para pegar a cotação tem um `sleep()` de 3 (três) segundos, o tempo final do nosso método terá um pouco mais de 6 (seis) segundos. Isso porque com esse método tradicional de programação assíncrona, usando `Future`, acabamos tendo que sincronizar o código usando `get()`.

11.2 A diferença entre Future e CompletionStage

Agora vamos ver como fazer um cliente um pouco diferente. Começaremos buscando somente uma cotação de forma reativa, mas sabemos que o final pretendido é encadear as duas chamadas que acabamos de fazer de forma sequencial.

```

@Test
public void testaCotacaoDolarRx() throws Exception {

```

```

Client client = ClientBuilder.newClient();
WebTarget cotacaoMoedasBean = client.target(
    "http://localhost:8080/javacred/rest/cotacao");
    .path("{quantidade}")
    .resolveTemplate("quantidade", 44);

CompletionStage<Response> stage = cotacaoMoedasBean
    .request(MediaType.APPLICATION_JSON)
    .rx()
    .get();

stage.thenApply(resp -> resp.readEntity(String.class))
    .thenAccept(System.out::println);

Thread.sleep(5_000);
}

```

Como podemos ver, a diferença é que em vez de usar `async()`, que devolve `Future` na hora de montar a requisição, aqui usamos `rx()`, que devolve `CompletionStage`, que é a interface do Java 8 que permite encadear as chamadas assíncronas.

Quando vamos consumir esse tipo de objeto, em vez de um método `get()` bloqueante como temos no `Future`, aqui registramos *callbacks* que serão chamados quando o método terminar. No exemplo usamos uma função que transforma nosso `Response` em `String` através do `thenApply()`. Usamos esse método para registrar funções que transformam um tipo em outro, e passam o retorno para o próximo método da chamada encadeada.

Aqui está a beleza dessa API, encadeamos chamadas sem ficarmos bloqueados. Em seguida, registramos um consumidor que vai receber a `String` que foi devolvida pela função anterior no encadeamento, e vai escrever seu valor no console através do método `println()`.

Uma prova de que nada aqui é bloqueante é a presença do `sleep()` no final do código, que é para dar tempo da execução em segundo

plano acabar. Sem isso o teste termina e não recebemos nenhum resultado do serviço por causa do *delay* de 3 (três) segundos que colocamos antes de devolver a cotação.

Finalmente vamos poder combinar a chamada dos dois serviços de cotação sem ter que esperar um terminar para chamar o outro. Prova disso é que vamos criar um novo teste que deve executar em 5 (cinco) segundos, se não falha:

```
@Test(timeout = 5_000)
public void testaCotacaoEuroEmRealRx() throws Exception {

    Client client = ClientBuilder.newClient();
    WebTarget cotacaoMoedasBean = client
        .target("http://localhost:8080/javacred/rest/cotacao");

    CompletionStage<Double> euroDolarStage = cotacaoMoedasBean
        .path("euro/{quantidade}")
        .resolveTemplate("quantidade", 50)
        .request(MediaType.APPLICATION_JSON)
        .rx()
        .get()
        .thenApply(resp -> resp.readEntity(Cotacao.class)
            .getValor());

    CompletionStage<Double> dolarRealStage = cotacaoMoedasBean
        .path("dolar/{quantidade}")
        .resolveTemplate("quantidade", 50)
        .request(MediaType.APPLICATION_JSON)
        .rx()
        .get()
        .thenApply(resp -> resp.readEntity(Cotacao.class)
            .getValor());

    CompletionStage<Double> combinadoStage = euroDolarStage
        .thenCombine(dolarRealStage, (euroDolar, dolarReal) -> {

            Double euroEmReal = euroDolar * dolarReal;
```

```

        return euroEmReal;
    });

    combinadoStage.thenAccept(System.out::println);

    Thread.sleep(4_000);
}

```

À primeira vista, esse código pode assustar um pouco, mas vamos passar pouco a pouco e veremos que não tem nada muito complexo. Iniciamos nosso código configurando nossas chamadas aos dois serviços de forma independente. A diferença é que, em vez de ficar com um `CompletionStage<Response>`, já aplicamos a função de transformação que devolve somente o valor da `Cotacao`.

Essa função de transformação é bem parecida com a anterior que transformava o `Response` em uma `String` que continha o JSON. A diferença é que usamos o `Response.readEntity(Cotacao.class)` para já receber um objeto dessa classe, e assim devolver seu valor já em formato `Double`.

Aqui, como na seção anterior, usamos a mesma classe `Cotacao` do projeto `javacred` que está no *class path* de `javacred-desktop`, mas poderíamos criar uma classe nova se quiséssemos.

Depois dessa configuração, ficamos com duas instâncias de `CompletionStage<Double>`, e usamos `thenCombine` para combinar uma na outra. A sintaxe pode parecer um pouco diferente, mas basicamente chamamos o método a partir de um dos `CompletionStage`, passando como parâmetro o outro e mais uma função que combina o resultado de ambos. Como os dois retornam `Double`, temos uma função que recebe dois `Double`s (`euroDolar` e `dolarReal`) e devolve um.

No código escrito no livro é mais difícil, mas na IDE você conseguirá examinar o tipo de cada parâmetro e ficará mais claro.

Com o `CompletionStage` resultante dessa combinação, usamos o método que já vimos antes, `thenAccept`, para consumir o resultado, e tudo isso será feito em pouco mais de três segundos, pois não haverá mais sincronização entre as chamadas. Por isso mesmo, mais uma vez temos o `sleep` para garantir que o teste ficará esperando os serviços retornarem.

E como dissemos antes, tudo isso executa mais rápido do que seria se as chamadas fossem sincronizadas, pois aí nosso teste falharia já que não teria como executar em menos de cinco segundos.

11.3 Como usar o `CompletionStage`?

Aqui fizemos uso de alguns métodos, mas temos outras opções. Então segue um resumo, mas, para saber de verdade tudo que é possível, consulte o Javadoc da interface.

`thenApply`

Usamos para encadear uma função que vai transformar o objeto passado no encadeamento de um tipo para outro.

`thenAccept`

Usado para consumir o resultado. Seu retorno é `void`, então diferente da função passada no `thenApply`, não dá para sair encadeando a chamada de um `thenAccept` em outro.

`thenRun`

Serve para executar um método, mas além de não devolver nada assim como o `thenAccept`, também não recebe nada. É como aquele *listener* que é chamado no final da execução do `SseEventSource` que vimos na seção passada. Podemos usar para alguma lógica de finalização.

Podemos também combinar `CompletionStages` como fizemos.

thenCombine

Método que usamos para combinar chamadas independentes. Ou seja, no nosso exemplo tanto faz se primeiro terminaria a consulta da cotação do Euro em Dólar ou do Dólar em Real, só queríamos que os resultados de ambos fossem combinados independentemente de qual terminasse antes. Terceirizamos esse trabalho.

thenCompose

Usamos para encadear chamadas que devem ser sequenciadas, assim o resultado de um vai ser entrada para o outro.

Além disso temos variações desses métodos que executam essas transformações de forma também assíncrona, como `thenApplyAsync`, `thenAcceptAsync`, `thenRunAsync`, `thenCombineAsync` e `thenComposeAsync`. Se não especificarmos nada adicional, essa execução será feita em uma outra `Thread` padrão. Mas podemos passar uma instância de `Executor` como parâmetro adicional, e então o trecho de código será executado através dele.

No próximo capítulo, o uso dos `Executor` será detalhado, mas já tivemos um primeiro contato, injetando `ManagedExecutorService` no `CotacaoMoedasBean`.

Para fechar o capítulo

Acabamos de ver como o Java EE 8 evoluiu com a inclusão de SSE e com a introdução da API reativa para construirmos clientes REST.

No próximo capítulo veremos como lidar com métodos assíncronos e paralelismo com Java EE dentro da nossa aplicação. Veremos que é um meio simples e muito eficiente de ganhar performance ao utilizar melhor os recursos de hardware dos servidores atuais.

CAPÍTULO 12

Métodos assíncronos e paralelismo para aumento de performance

Apesar de serem coisas diferentes, esses dois temas foram colocados juntos pois a implementação de um está ligada à do outro. Por exemplo, se temos um processamento que demora muito, conseguimos entregar uma usabilidade melhor para o usuário dizendo para ele que a tarefa está sendo processada e que em instantes estará pronta. Assim o usuário pode continuar trabalhando normalmente pelo restante do sistema, e enviamos um e-mail ou o avisamos de alguma outra forma quando a tarefa terminar.

É mais comum tratarmos diretamente com isso em aplicações Desktop, onde precisamos abrir esse trabalho em uma `Thread` separada, se não a tela do usuário congela. Em aplicações Web isso fica mais sutil, pois, quando o usuário submete, é comum ele esperar um pouco com a tela processando (seja via AJAX ou não), e na maioria das vezes esse tempo é o suficiente para terminar o processamento.

Ok, com essa introdução entendemos a necessidade de usarmos métodos assíncronos para o usuário não pensar que o sistema *bugou* enquanto na verdade ele está processando. Mas onde entra o paralelismo e o aumento de performance?

A resposta é que também utilizaremos ferramentas parecidas com a que usamos para dar maior fluidez na apresentação do sistema para o usuário, que é por meio de processamento paralelo. Este não é um capítulo de *tunning* visando performance, mas sim um capítulo onde veremos que conseguimos ganhar muito em performance apenas pensando de maneira inteligente em como as tarefas mais pesadas da nossa aplicação serão executadas. No capítulo anterior vimos como o paralelismo ajuda na conversa de um serviço com

seu cliente; aqui o foco será mais na comunicação dentro da própria aplicação.

12.1 O uso de `@Asynchronous` para não bloquear o usuário

Assim como foi comentado na introdução do capítulo, vamos iniciar vendo uma situação em que a operação que o usuário deseja executar pode demorar muito e deixá-lo esperando. A operação no caso será atualizar o saldo de todos os empréstimos, aplicando os índices de correções em cada contrato. Para simplificar, vamos trabalhar com um único índice e adicionar valores fictícios para seu valor a cada mês.

Porém, até agora temos alguns poucos contratos no nosso banco de dados, e para que possamos sentir alguma diferença é importante que tenhamos um volume grande de contratos, e por isso vamos implementar um gerador também bastante simples.

```
import java.time.YearMonth;
...
@Path("/gera-contratos")
public class GeradorContratosController {

    @Inject
    private ContratoBean contratoBean;

    @GET
    public void geraContratos(){

        long inicio = System.currentTimeMillis();

        //valor fictício de IGPM a 10%
        Indice igpm = new Indice("IGPM");
        igpm.setValores(Arrays.asList(
            //Date/Time API Java 8
```



```

        new IndiceValor(igpm, YearMonth.now(), 0.1)));

    //Aqui vamos gerar 10 mil contratos
    for (int i = 0; i < 10_000; i++) {
        Contrato contrato = new Contrato();
        contrato.setValor(1_000_000.0);
        contrato.setIndiceCorrecao(igpm);
        contratoBean.salvar(contrato);
    }

    long fim = System.currentTimeMillis();
    System.out.printf("Contratos gerados com sucesso. "
        + "Tempo %d ms\n", fim-inicio);
}
}

```

Criamos um valor fictício de IGPM a 10% para servir como nosso índice, e então criamos 10 mil contratos com saldo de um milhão e aplicamos o índice anterior. Mas como ainda não vimos o tipo `Indice`, precisamos criá-lo para nosso código anterior funcionar.

```

@Entity
public class Indice {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(unique = true)
    private String nome;

    @OneToMany(mappedBy = "indice", cascade={PERSIST, MERGE})
    private List<IndiceValor> valores;

    public Indice(){

    }

    public Indice(String nome) {
        this.nome = nome;
    }
}

```

```
//getters e setters  
}
```

Essa é uma entidade bem simples usando JPA como já vimos algumas vezes. Além do índice, temos o tipo `IndiceValor` que guarda efetivamente o valor do índice para cada mês.

```
@Entity  
public class IndiceValor {  
  
    @Id  
    @GeneratedValue(strategy= GenerationType.IDENTITY)  
    private Integer id;  
    private YearMonth anoMes;  
    private Double valor;  
  
    @ManyToOne  
    private Indice indice;  
  
    public IndiceValor(){}  
  
    public IndiceValor(Indice indice, YearMonth anoMes, Double valor) {  
        this.indice = indice;  
        this.anoMes = anoMes;  
        this.valor = valor;  
    }  
    //getters e setters  
}
```

E depois de criar mais essa entidade simples, temos que relacionar o `Indice` ao **NOSSO** `Contrato`:

```
public class Contrato implements Serializable {  
    ...  
  
    @ManyToOne(cascade=CascadeType.MERGE)  
    private Indice indiceCorrecao;  
  
    //getter e setter
```

```
...  
}
```

Agora basta chamar pelo navegador a url

`http://localhost:8080/javacred/rest/gera-contratos` e teremos dez mil contratos no banco. Perceba que, dessa vez, em vez de criarmos uma servlet para fazer esses trabalhos simples, utilizamos um serviço REST, que, além de poder ser tão simples quanto uma servlet na forma de consumir, nos permite tudo que vimos no capítulo anterior. Veja também que, para simplificar nosso exemplo, acabamos deixando nosso `Contrato` ser criado sem um cliente, sem descrição etc. Em um caso real provavelmente todos os campos do `Contrato` seriam obrigatórios.

Se você já executou essa operação, percebeu que ela demorou um pouco para terminar. Certamente esse tempo não é algo que podemos considerar aceitável para o usuário. Então ao criar o mecanismo de atualização do saldo dos contratos, vamos tratar isso para que o usuário não fique esperando.

Para esse exemplo vamos criar uma nova tela e um novo controlador para ela, além de adicionar alguns métodos no

`ContratoBean` .

Vamos começar pelo controlador, que terá a responsabilidade de chamar a atualização dos contratos e mostrar para o usuário quanto tempo demorou a execução.

```
import javax.faces.view.ViewScoped;  
...  
@Named  
@ViewScoped  
public class AtualizadorSaldoController implements Serializable {  
  
    @Inject  
    private ContratoBean contratoBean;  
  
    private Long numeroContratos;
```

```

private Map<String, Benchmark> benchmarks;

private List<Indice> indices;

@PostConstruct
public void init(){
    numeroContratos = contratoBean.contaContratos();
    indices = contratoBean.buscarIndices();
    benchmarks = new HashMap<>();
}

public void aplicaCorrecaoSincrona() throws Exception{
    Benchmark benchmark =
        new Benchmark(numeroContratos, "síncrono");
    benchmarks.put("sync", benchmark);

    contratoBean.corrigeContratos(indices);

    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage(benchmark.getMessage()));
}

public Long getNumeroContratos() {
    return numeroContratos;
}

public Map<String, Benchmark> getBenchmarks() {
    return benchmarks;
}
}

```

Podemos ver que o código é bem simples, apenas contamos o número de contratos e criamos o método `aplicaCorrecaoSincrona()`, que pede para o `ContratoBean` ajustar os contratos passando os `indices`. Além disso, guardamos a informação do tempo que a operação levou para depois comparar com a versão melhorada.

Apesar de poder ser um pequeno *spoiler* do que vem pela frente, precisamos buscar os índices antes de corrigir os contratos porque não queremos que nosso método de correção monetária faça essa

busca antes de ajustar os contratos. Agora pode até parecer sem sentido já que a busca e a atualização são feitas dentro do mesmo bean, mas quando nosso exemplo avançar mais, veremos que fazer uma busca antes da correção estragaria a performance do nosso código. Obviamente isso poderia ser tratado com um cache de consulta, mas além de esse tema não ser nosso objeto de estudo, fazer buscas no banco dentro de métodos de cálculo não é algo muito esperto, o melhor é receber por parâmetro tudo que é necessário e depois retornar o resultado.

Além disso, criamos a classe `Benchmark` para nos ajudar a organizar os resultados. O código da classe é bem simples, como podemos ver a seguir:

```
public class Benchmark {

    private long numeroContratos;
    private String nome;
    private Long inicio, fim;

    public Benchmark(long numeroContratos, String nome) {
        this.numeroContratos = numeroContratos;
        this.nome = nome;
        this.inicio = System.currentTimeMillis();
    }

    public void stop(){
        this.fim = System.currentTimeMillis();
    }

    public long getDuracao(){
        return fim - inicio;
    }

    public String getMessage(){
        if(fim == null) stop();

        return String.format(
            "%d contratos atualizados (%s) em %d ms",
```

```

        numeroContratos, nome, getDuracao());
    }
}

```

Como essa classe `Benchmark` é somente um auxílio para nosso código, não vale a pena comentar muito seu conteúdo, até porque ele é autoexplicativo.

Mas como está, nosso código ainda não funciona. Temos ainda que implementar a tela e complementar nosso `ContratoBean`. Como estamos mexendo com código Java, vamos primeiro ver este último. Uma dica para facilitar a implementação na sua IDE é utilizar a correção automática para criar esses métodos que ainda não existem: `contaContratos()`, `buscarIndices()` e `corrigeContratos(List<Indice> indices)` na classe `ContratoBean`.

No Eclipse, por exemplo, basta colocar o cursor na linha do erro e pressionar `Control/Command + 1`, que é exibida uma lista de alternativas para corrigir o problema. Em outras IDEs você pode clicar no ícone de erro que fica à esquerda da linha que não está compilando e as mesmas opções serão apresentadas.

```

import org.jboss.ejb3.annotation.TransactionTimeout;
...
public class ContratoBean {

    @TransactionTimeout(value = 5, unit = TimeUnit.MINUTES)
    public void corrigeContratos(List<Indice> indices) {

        for(Contrato contrato : listarTodos()){
            corrigeContrato(contrato, indices);
        }

    }

    private void corrigeContrato(Contrato contrato,
        List<Indice> indices){
        double percentual =
            calculaPercentualCorrecao(contrato, indices);
        Double novoSaldo =

```

```

        contrato.getSaldo() * (1 + percentual);
        contrato.setSaldo(novoSaldo);
    }

    private double calculaPercentualCorrecao(Contrato contrato,
        List<Indice> indices){
        return 0.1;
    }

    public Long contaContratos() {
        return em.createQuery(
            "select count(c) from Contrato c", Long.class)
            .getSingleResult();
    }

    public List<Indice> buscarIndices(){
        return em.createQuery(
            "select i from Indice i join fetch i.valores",
            Indice.class).getResultList();
    }

    ...
}

```

O restante do código da classe `ContratoBean` permanece igual. E agora temos que adicionar a seguinte dependência no `pom.xml`.

```

<dependency>
    <groupId>org.jboss.ejb3</groupId>
    <artifactId>jboss-ejb3-ext-api</artifactId>
    <version>2.3.0.Final</version>
    <scope>provided</scope>
</dependency>

```

Perceba que precisamos de um novo *import* para ajustar o *timeout* da nossa transação, que nesse caso vai poder demorar até 5 minutos. Essa anotação é específica para JBoss/Wildfly, pois é algo que depende de cada servidor. Se sua aplicação executará em outro servidor ou em diversos servidores, você pode usar a anotação do outro servidor, ou o mais indicado seria deixar essa configuração

para tempo de deploy, o que geralmente é feito via XML. Nesse caso você também terá que buscar a configuração para cada servidor que for utilizar. Aqui manteremos apenas essa configuração pois colocar cada possibilidade inflaria demais este capítulo, e encontrar as diferentes configurações na documentação do servidor ou mesmo buscando no Google não é difícil.

Outro ponto interessante é que, ao buscar nossos índices, já carregamos também seus valores, pois sem isso de nada adiantaria fazer a busca no `@PostConstruct` do `AtualizadorSaldoController`, uma vez que as listas são *lazy* por padrão.

E, por fim, temos o método `calculaPercentualCorrecao(Contrato contrato, List<Indice> indices)` que por enquanto devolve um valor fixo para simplificar nossa implementação.

Com a parte servidora pronta, vamos criar a tela onde vamos executar nosso método. Criaremos uma tela JSF bem simples sem qualquer preocupação com layout, pois o interesse aqui é na parte servidora mesmo.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">

  <h:head>
  </h:head>

  <h:body>
    <h:messages/>

    <h:form>
      <c:set var="controller"
        value="#{atualizadorSaldoController}"/>

      Quantidade de contratos: #{controller.numeroContratos}
    <br/>
```



```

        <h:commandButton value="Atualizar saldo dos contratos"
            action="#{controller.aplicaCorrecaoSincrona()}" />
        ultima execução:
            #{controller.benchmarks['sync'].duracao} ms;
    <br/>

</h:form>
</h:body>
</html>

```

Pronto, agora podemos executar a nossa tela no endereço `http://localhost:8080/javacred/ajustaContratos.jsf` e executar o ajuste dos contratos e esperar o resultado. Novamente temos uma péssima experiência para o usuário, pois a tela demora bastante para devolver o resultado. Mas agora já temos a base pronta para resolvermos isso.

Iniciando uma execução assíncrona

Para iniciar uma implementação mais interessante para o usuário, vamos adicionar um novo botão na tela, com seu respectivo contador de tempo. E no lado servidor vamos criar novos métodos no controlador e no `ContratoBean`. Desta vez começaremos pela tela:

```

...
<h:commandButton
    value="Atualizar saldo dos contratos assincrono"
    action="#{controller.aplicaCorrecaoAssincrona()}" />
ultima execução:
    #{controller.benchmarks['async'].duracao} ms;
...

```

Apenas adicionamos essas linhas abaixo do último `
` que tínhamos na versão anterior; e agora vamos ao controlador.

```

...
public class AtualizadorSaldoController implements Serializable{
    ...

```

```

    public void aplicaCorrecaoAssincrona() throws Exception{
        Benchmark benchmark =
            new Benchmark(numeroContratos, "assíncrono");
        benchmarks.put("async", benchmark);

        Future<Double> resultado =
            contratoBean.corrigeContratosAsync(indices);
        benchmark.setResultadoAsync(resultado);

        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage(benchmark.getMessage()));
    }
}

```

No controlador a mudança foi também bastante simples, praticamente duplicamos o código anterior mudando apenas a mensagem para o usuário e chamando um novo método do ContratoBean, que é o `aplicaCorrecaoAssincrona(List<Indice> indices)`. Agora vejamos a implementação desse novo método, pois é lá que está toda a "mágica".

```

import javax.ejb.Asynchronous;
...
public class ContratoBean {

    @Asynchronous
    public void corrigeContratosAsync(List<Indice> indices) {

        corrigeContratos(indices);

        //enviar notificação/e-mail
        //avisando que o processo terminou.
    }
}

```

Tudo que tivemos de mudança real foi o acréscimo da anotação `@Asynchronous` no novo método. E para não termos dúvida de que a forma de aplicar a correção não mudou, esse novo método apenas chama o antigo.

Executando nossa aplicação novamente, teremos uma tela como a seguinte:

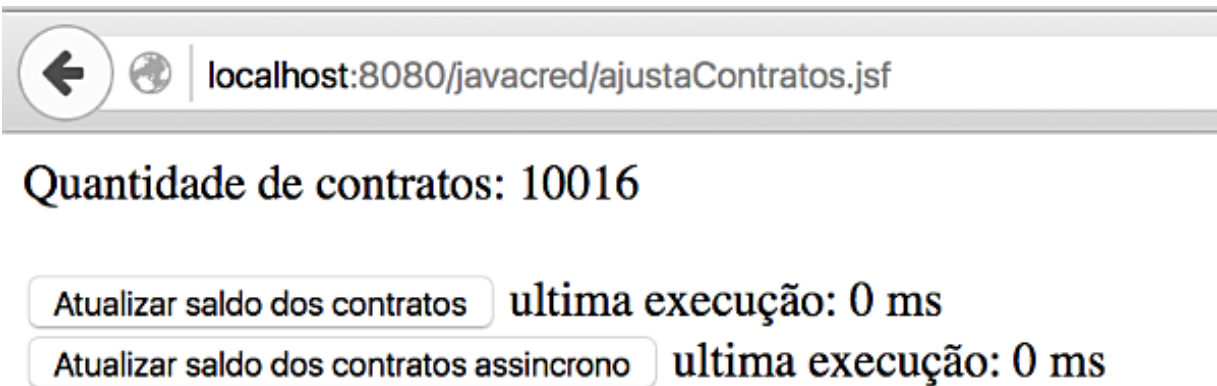


Figura 12.1: Tela de ajuste de contratos

Agora teste executar o método de ajuste síncrono e espere o seu término e em seguida faça o mesmo com o método assíncrono e veja a diferença. Perceba pelo console do servidor que, assim como a mensagem para o usuário informa, a operação ainda está acontecendo em segundo plano, mas isso é o suficiente para o usuário continuar utilizando outras funcionalidades do sistema sem ficar travado.

E, ao final do processo assíncrono, podemos notificar o usuário seja por e-mail ou através de alguma notificação no próprio sistema, algo parecido com as notificações do Facebook, por exemplo. A importância dessa mudança é muito mais para nos fazer pensar do que efetivamente técnica; pois, como vimos, a implementação difere da padrão em apenas uma anotação.

O que devemos prestar atenção é no fato de que a escolha correta da abordagem evitou que buscássemos uma otimização prematura. E esse processo ainda não acabou. Nas seções seguintes veremos como melhorar ainda mais esse processo sem ter que pensar em otimização do ajuste do contrato, apenas utilizando uma melhor estrutura para o cálculo.

OTIMIZAÇÃO PREMATURA VS. NENHUMA OTIMIZAÇÃO

A ideia não é de forma alguma dizer que não devemos otimizar nossos algoritmos, apenas que não devemos fazer isso antes do necessário, pois podemos cair em uma má prática de desenvolvimento, que é a otimização prematura.

Imagine que, para evitar que o usuário ficasse um minuto parado na tela, nós refatorássemos todo o processamento - algo que no mundo real será realmente complexo - para no final ter um ganho de 50% (cinquenta por cento). Seria uma boa otimização, afinal reduzimos o tempo pela metade. Mas ainda assim o usuário ficaria 30 segundos parado, o que ainda seria ruim.

Apenas mudando a estratégia para algo assíncrono damos ao usuário a sensação de ser algo imediato, deixando-o continuar seu trabalho, o que é um resultado bem melhor, e seguramente custa uma mínima fração do esforço de otimização do algoritmo em si.

Como ainda estamos no primeiro estágio da transformação do nosso código, pode parecer um tanto simplista aceitar essa conclusão, mas com o desenvolver deste capítulo isso ficará mais claro.

12.2 Recuperando o retorno de uma execução em segundo plano

A solução anterior resolveu o problema de o usuário conseguir executar algo grande sem ficar bloqueado até que a operação termine, mas no exemplo nós não fizemos uso de nenhum retorno

dos métodos. Aparentemente temos a solução quando o método é `void`, mas como fazemos para recuperar o retorno de um método que executa em segundo plano?

Aqui utilizaremos a interface `java.util.concurrent.Future`, algo disponível desde o Java 5 e que nos permite recuperar um valor que estará disponível no futuro, quando o processamento de outra `Thread` terminar.

Um ponto que vale a pena observarmos é que a especificação de EJB não foi atualizada no Java EE 8, sendo a mesma versão 3.2 do Java EE 7. Com isso não temos aqui as novidades do Java 8 que foram incorporadas na JAX-RS 2.1, como o suporte a programação reativa através da interface `CompletionStage` que vimos no capítulo passado. Pode ser que tenhamos algo nesse sentido no Jakarta EE 9.

Voltando ao nosso exemplo, vamos mudá-lo para agora ao final do ajuste de todos os contratos devolver o novo saldo geral. Então em vez de `void` teremos um retorno `double` no método síncrono. Em seguida veremos como fazer isso na versão assíncrona.

Na classe `ContratoBean` a seguinte alteração é necessária:

```
...
@Transactional(timeout = 5, unit = TimeUnit.MINUTES)
public double corrigeContratos(List<Indice> indices) {

    double saldoGeral = 0.0;
    for(Contrato contrato : listarTodos()){
        double percentual = corrigeContrato(contrato, indices);
        Double novoSaldo =
            contrato.getSaldo() * (1 + percentual);
        contrato.setSaldo(novoSaldo);
        saldoGeral += novoSaldo;
    }

    return saldoGeral;
}
```

```
}  
...
```

Nada diferente do que faríamos mesmo sem o código aqui. Agora vamos adicionar os seguintes trechos na classe a seguir:

```
...  
public class AtualizadorSaldoController implements Serializable{  
    ...  
    public void aplicaCorrecaoSincrona(){  
        ...  
  
        double resultado =  
            contratoBean.corrigeContratos(indices);  
        benchmark.setResultado(resultado);  
  
        ...  
    }  
    ...  
}  
  
public class Benchmark {  
    ...  
    private double resultado;  
  
    public void setResultado(double resultado){  
        this.resultado = resultado;  
        this.stop();  
    }  
  
    //getter  
    ...  
}
```

Com essas alterações agora temos o resultado da nossa correção monetária guardada dentro do `Benchmark`. E por fim vem a mudança na tela:

```
...  
ultima execução: #{controller.benchmarks['sync'].duracao} ms;
```

```
saldo geral:
<h:outputText value="#{controller.benchmarks['sync'].resultado}">
    <f:convertNumber type="currency"/>
</h:outputText>
...
```

Aqui apenas colocamos o texto com o saldo geral após o tempo da última execução, e para formatar como moeda usamos o conversor `f:convertNumber` do JSF.

Agora podemos executar o exemplo e ver que funcionou. Mas essa era a parte óbvia. A parte interessante é como faremos com a execução em segundo plano. Para isso, vamos novamente no `ContratoBean` para vermos como usar o tipo `Future` mencionado anteriormente.

```
import javax.ejb.AsyncResult;
...
@Asynchronous
public Future<Double> corrigeContratosAsync(List<Indice> indices) {

    double saldoGeral = corrigeContratos(indices);

    return new AsyncResult<Double>(saldoGeral);
}
```

Bem simples, não? Já existe o tipo `AsyncResult` na especificação de EJBs para tratar esse tipo de situação. Esse tipo é uma implementação para o Java EE da interface `Future` que é do Java SE. Visto isso, só precisamos saber como usar o tipo `Future`, que, apesar de já termos visto no capítulo anterior, aqui estudaremos com mais cuidado.

Essa interface tem os seguintes métodos:

```
public interface Future<V> {
    boolean    cancel(boolean interromperSeExecutando);
    V get();
    V get(long timeout, TimeUnit unit);
    boolean    isCancelled();
}
```

```
        boolean    isDone();  
    }
```

Uma explicação rápida de cada método é importante para prosseguirmos:

boolean cancel(boolean)

Pedimos o cancelamento da execução. Como mandamos executar em segundo plano, pode ser que a execução ainda não tenha iniciado após algum tempo. Se ainda não tiver iniciado, será cancelado, mas se já tiver iniciado, só será cancelado se o parâmetro informado for `true`. Se a execução foi cancelada, retorna `true`, se terminou normalmente ou já estava em execução e o parâmetro informado foi `false`, o retorno será `false`.

V get()

Se invocarmos esse método, a execução atual fica esperando até a que está em segundo plano terminar e retornar o valor. Se não quisermos esse comportamento devemos primeiro ver se a execução já terminou via `isDone()` antes de chamar `get()`. Caso a execução tenha terminado por exceção, cancelamento ou outro motivo que não seja seu término normal, será lançada uma exceção com o tipo específico.

V get(long, TimeUnit)

O funcionamento é basicamente o mesmo do método anterior, mas define um tempo máximo para esperar o término do método em segundo plano. Se não terminar nesse tempo será lançada uma `java.util.concurrent.TimeoutException`.

boolean isCancelled()

Serve para checar se a execução foi ou não cancelada.

isDone()

Usamos esse método para saber se a operação já parou de executar, mesmo que seja com erro ou por cancelamento. Se não está mais executando podemos chamar os métodos `get` sem ficarmos bloqueados.

Sabendo como funciona o `Future`, estamos prontos para adaptar o restante do exemplo. Basicamente teremos que cuidar para não usar o método `get()` sem que o `isDone()` retorne `true`.

Vamos começar pelo controlador.

```
...
public class AtualizadorSaldoController implements Serializable{
    ...
    public void aplicaCorrecaoAssincrona(){
        ...
        Future<Double> resultado = contratoBean
            .corrigeContratosAsync(indices);
        benchmark.setResultadoAsync(resultado);
        ...
    }
    ...
}
...
public class Benchmark {
    ...
    Future<Double> resultadoAsync;
    public void setResultadoAsync(Future<Double> resultadoAsync){
        this.resultadoAsync = resultadoAsync;
        this.stop();
    }
    //getter de resultadoAsync
    ...
}
```

Nada de diferente do que tivemos no método que devolvia `double` direto, mas a grande diferença virá na tela. Apesar de nosso foco aqui não ser *front-end* nem JSF, esse e outros trechos de tela servirão para demonstrar como fazemos uso do `Future` no lado cliente do nosso serviço.

```

<h:panelGroup id="processamentoAssincrono">

    ultima execução:
        #{controller.benchmarks['async'].duracao} ms;

    saldo geral:
    <c:set var="saldoGeralAssincrono"
        value="#{controller.benchmarks['sync'].resultadoAsync}"/>

    <h:outputText value="#{saldoGeralAssincrono.get()}"
        rendered="#{saldoGeralAssincrono.done}">
        <f:convertNumber type="currency"/>
    </h:outputText>

    <h:outputText value="não calculado"
        rendered="#{empty saldoGeralAssincrono}"/>

    <h:commandLink value="calculando... (clique para atualizar)"
        rendered="#{not empty saldoGeralAssincrono
            and !saldoGeralAssincrono.done}">
        <f:ajax render="processamentoAssincrono"/>
    </h:commandLink>

</h:panelGroup>

```

Assim como na outra tela, nós colocamos o resultado logo após o texto do tempo de execução, porém aqui colocamos três saídas diferentes, uma para quando ainda não realizamos o cálculo, outra para quando ele está pronto, e uma terceira para quando ele está em progresso. Na verdade a única saída realmente diferente é essa última, pois no caso da execução síncrona o correto seria fazer as duas primeiras saídas, mas para simplificar simplesmente ignoramos a saída de quando a execução ainda não foi feita, deixando apenas o valor zerado.

Agora indo realmente para a saída diferente, nós fizemos uma checagem para garantir que o `saldoGeralAssincrono` não está nulo através da EL `not empty`. Checamos também se já não tinha terminado através do método `isDone()`, que na EL fica simplesmente

`!saldoGeralAssincrono.done` . Note que em uma EL o prefixo `is` é tão desnecessário quanto o `get` ou `set` , pois são todos prefixos padrão *JavaBean*, sendo entendidos pela EL.

Além da checagem nós exibimos um link de ação em vez de um simples texto, assim o usuário pode clicar nele para, via AJAX, atualizar todo o `h:panelGroup` e consequentemente recalcular qual saída é a correta naquele momento.

Isso também poderia ser feito através de *polling*, que é de tempos em tempos executar uma ação independente de uma ação do usuário. É possível fazer isso com JavaScript puro ou então utilizando componentes prontos de bibliotecas de componentes como o PrimeFaces. Nos exemplos posteriores veremos como fazer isso, mas por enquanto deixaremos assim, e ao final teremos os dois exemplos na mesma página.

Se executarmos novamente nossa página, veremos que podemos clicar diversas vezes no link de ação até finalmente terminar o processamento. Apesar de a função desse link ser atualizar o trecho da tela que mostra o resultado, isso mostra que o usuário não fica mais bloqueado esperando o resultado. Por ser um link de ação, ele poderia chamar qualquer outra regra de negócio que executaria paralelamente à correção monetária em curso.

Por mais que a experiência do usuário tenha melhorado por ele não ficar mais "travado", vemos que a execução demora bastante para terminar, e não podemos deixar nosso sistema desse jeito. Por isso, o próximo estágio é paralelizar o processo.

12.3 Executando código em paralelo com @Asynchronous

Relembrando algo que foi dito no início deste capítulo, podemos usar a mesma estratégia de execução em segundo plano para ganhar em performance, e não apenas para "destravar" a tela do usuário como vimos antes.

Se pensarmos um pouco, ao executar uma tarefa "pesada" em segundo plano - como fizemos na seção *O uso de @Asynchronous para não bloquear o usuário* - estamos executando a tal tarefa pesada e a renderização da tela de resposta em paralelo. Agora, pensando em ganho de performance, o que precisamos fazer é quebrar esse processamento pesado em partes que possam ser executadas em paralelo também.

Já vimos que ao chamar um método anotado com `@Asynchronous` fazemos com que ele execute em uma `Thread` separada. Então o que vamos fazer agora é pedir para que cada contrato seja atualizado por um método próprio, que vai tratar contratos individualmente e que seja anotado com `@Asynchronous`. Assim vamos ter o que buscamos, que é uma paralelização de trabalho que tira proveito dos múltiplos núcleos de processamento dos computadores atuais, e principalmente dos servidores, que costumam ter ainda mais poder de processamento paralelo.

Para isso vamos criar um novo EJB, que fará o ajuste de cada contrato de forma assíncrona:

```
@Stateless
public class AjustadorContratoBean {

    @PersistenceContext
    private EntityManager em;

    @Asynchronous
    public Future<Double> ajustaContratoPercentualAsync(
        Contrato contrato,
        double percentual) {

        System.out.println("executando... ")
```

```

        + Thread.currentThread());

    Double novoSaldo =
        contrato.getSaldo() * (1 + percentual);
    contrato.setSaldo(novoSaldo);

    em.merge(contrato);
    return new AsyncResult<Double>(novoSaldo);
}
}

```

Esse é um código bastante simples. Nele, como sempre, o mais importante não é só colocar o código para executar, e sim entendê-lo. Afinal, se não fosse assim bastaria pegar um trecho de código pronto na internet que nem sabemos como funciona. No entanto, para entender de verdade essa implementação, precisaremos ver em conjunto o código do `ContratoBean` logo mais.

Iniciando nossa análise do método `ajustaContratoPercentualAsync` vemos novamente um método assíncrono que retorna `AsyncResult`, nada de novo aqui. O ponto de atenção está no trecho `em.merge(contrato)`, mas, como dito antes, para fazer mais sentido precisaremos analisar o código do `ContratoBean` que chama nosso método assíncrono:

```

...
@Inject
private AjustadorContratoBean ajustadorContratoBean;
...
public double corrigeContratosPar(List<Indice> indices) {

    int i = 0;
    List<Future<Double>> resultatos = new LinkedList<>();

    for(Contrato contrato : listarTodos()){

        System.out.printf(
            ">>>>> enfileirando: %d -> Thread: %s \n",
            ++i, Thread.currentThread());
    }
}

```

```

        double percentual =
            calculaPercentualCorrecao(contrato, indices);

        em.detach(contrato);

        resultados.add(ajustadorContratoBean
            .ajustaContratoPercentualAsync(contrato, percentual));
    }

    double saldoGeral = 0.0;

    for (Future<Double> resultado : resultados) {

        try {
            saldoGeral += resultado.get();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }

    return saldoGeral;
}
...

```

Novamente quase todo o código aqui não é novo. Criamos o método `corrigeContratosPar` que delega o cálculo para o `AjustadorContratoBean`. O nome do nosso método ficou meio feio porque aqui estamos economizando no tamanho dos nomes para que a visualização no livro fique melhor.

Analisando nosso código podemos ver que delegamos o cálculo do ajuste de cada contrato ao novo EJB que acabamos de criar e guardamos o resultado que é do tipo `Future<Double>` em uma lista para podermos recuperar o resultado depois que o cálculo de cada contrato tiver sido enfileirado.

Inclusive colocamos uma saída direta no console apenas para facilitar nossa visualização do processo enquanto executamos o exemplo. Nessa saída mostramos exatamente esse enfileiramento,

enquanto no `AjustadorContratoBean` colocamos uma saída parecida que serve para vermos a execução. Isso é interessante para percebermos as diferentes `Threads` funcionando.

12.4 Paralelizando a persistência no banco de dados

Apesar de esse código também ser simples, temos novamente um pequeno trecho em que devemos prestar atenção, que é o `em.detach(contrato)`. Ele faz par com o `em.merge(contrato)` que destacamos no código do `AjustadorContratoBean`. Porém, para entender a relação entre esses dois trechos, e por que são importantes, precisamos lembrar como funciona o contexto de persistência em um EJB. Isso foi bastante discutido nos capítulos 7 e 8; mas vamos a uma recapitulação voltada para o nosso caso.

Em regra temos uma instância diferente de `EntityManager` em cada transação que é executada, e essa transação é propagada para todos os métodos chamados a partir de um método transacional de um EJB. Como por padrão os métodos de um EJB já são transacionais, o método `corrigeContratosPar` executa dentro de uma transação, e essa mesma transação seria propagada para a execução do método `ajustaContratoPercentual` do `AjustadorContratoBean`, pois seu método foi chamado a partir do método `corrigeContratosPar`. Como seria a mesma transação, apesar de serem classes diferentes, a instância de `EntityManager` injetada seria a mesma.

Porém, como vimos no capítulo 8. **Lidando com mais de uma transação**, podemos mudar esse comportamento para que o método chamado em sequência execute dentro de sua própria

transação, e fazemos isso através da anotação

`@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)` . Apesar de não termos colocado essa anotação em momento algum aqui, **quando utilizamos `@Asynchronous` em um método, implicitamente estamos dizendo que executará em uma transação separada, como ocorre ao utilizarmos `REQUIRES_NEW`.**

Então significa que se não fizermos `em.detach(contrato)` e `em.merge(contrato)` as alterações feitas no contrato não serão persistidas? Errado, elas serão persistidas, mas não na transação nova criada para cada contrato, pois dentro dela o objeto `Contrato` passado como parâmetro vai estar *detached*, ou seja, as mudanças não são persistidas automaticamente. Para isso acontecer é preciso que o `EntityManager` associado a essa nova transação faça um `merge` nessa entidade.

Mas então onde seria salvo esse objeto se não fizéssemos isso? O contrato seria atualizado na "transação mãe", que envolve o método `ContratoBean.corrigeContratosPar` . Mas se deixássemos isso acontecer, todo o trabalho que foi feito em paralelo teria a operação de banco feita na `Thread` original, o que eliminaria praticamente todo o ganho de performance nesse caso. Agora, se no caso da sua aplicação não há uma operação de banco em questão, e sim processamento puro, cujo resultado é passado para algum lugar via *Web Services*, por exemplo, então não haveria problema. Porém, em boa parte dos sistemas, a operação de banco consome uma parte significativa do trabalho total.

E para finalizar essa análise, o ato da transação original salvar automaticamente o contrato alterado na transação nova ocorre porque estamos utilizando EJBs locais, logo os objetos são passados por referência e a transação original "percebe" a alteração feita. Por esse motivo nós utilizamos `em.detach(contrato)` para que o `EntityManager` se "esqueça" daquela instância e deixe o controle para o `EntityManager` associado à nova transação. Obviamente, para que tudo isso fique mais claro, durante a execução do exemplo

experimente comentar esses dois trechos (`detach` e `merge`) para ver o resultado. Fazendo isso perceberemos um resultado parecido com o da seção anterior, onde a execução era sequencial.

Bom, a parte mais complicada já passou, agora é novamente alterar nosso controlador e tela para podermos chamar esse novo método e comparar o seu resultado com o da execução sem paralelização e também sem executar em segundo plano. Então vamos começar pelo `AtualizadorSaldoController`.

```
...
public void aplicaCorrecaoParalela(){
    Benchmark benchmark =
        new Benchmark(numeroContratos, "paralelo");
    benchmarks.put("par", benchmark);

    double resultado = contratoBean.corrigeContratosPar(indices);
    benchmark.setResultado(resultado);

    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage(benchmark.getMessage()));
}
...
```

E agora vamos à tela `ajustaContratos.xhtml`, onde criaremos um botão que chamará nosso novo método, e também exibiremos o tempo de execução e o resultado.

```
...
<br/>
<h:commandButton value="Atualizar saldo dos contratos paralelo"
    action="#{controller.aplicaCorrecaoParalela()}" />
ultima execução: #{controller.benchmarks['par'].duracao} ms;
saldo geral:
<h:outputText value="#{controller.benchmarks['par'].resultado}">
    <f:convertNumber type="currency" />
</h:outputText>
...
```

Agora nossa tela deve estar parecida com a seguinte:

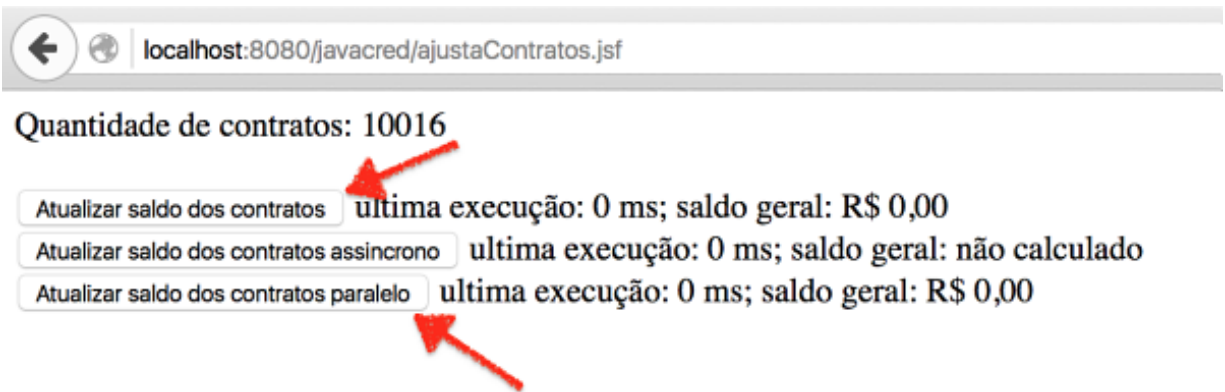


Figura 12.2: Tela com botão para execução em paralelo

Vamos então executar e comparar os tempos dos dois botões marcados na imagem. O resultado foi como esperado? Você saberia explicar o porquê?

Temos alguns pontos a analisar. O primeiro é que, se repararmos na saída no console, vemos que, antes de atualizar nosso contrato, há sempre uma consulta no banco que carrega esse mesmo contrato. Esse comportamento parece estranho, mas é esperado. Como estamos usando um segundo `EntityManager` para salvar as alterações feitas em uma entidade que foi trazida do banco por um `EntityManager` diferente, a única forma de saber como o objeto estava no banco antes das alterações é através de uma consulta. Isso serve inclusive para casos onde esse objeto nem está no banco ainda, e em vez de uma atualização, o método `merge` precisará fazer uma inserção.

Ok, esse é o comportamento esperado quando utilizamos JPA, mas e se quisermos assumir a responsabilidade e mandar: "pode fazer *update* direto que eu garanto que esse objeto já está no banco e que ninguém mais está mexendo nele agora"? Nesse caso precisaremos descer de nível, e em vez de usar a JPA, que nem possui método `update`, precisaremos acessar a API nativa do Hibernate, que é a implementação de JPA utilizada no Wildfly. Para isso vamos precisar deixar a API do Hibernate acessível em tempo

de desenvolvimento, adicionando o seguinte trecho ao nosso `pom.xml` :

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.15.Final</version>
    <scope>provided</scope>
</dependency>
```

Perceba que o escopo utilizado é o `provided` , pois essa biblioteca já está dentro do servidor, queremos apenas acessá-la pela nossa IDE. A versão aqui usada é a mesma que vem instalada no Wildfly 19.

Com isso pronto, podemos mudar a linha do `AjustadorContratoBean` que faz `em.merge(contrato)` da seguinte maneira:

```
import org.hibernate.Session;
...
@Stateless
public class AjustadorContratoBean {
    ...
    @Asynchronous
    public Future<Double> ajustaContratoPercentualAsync(
        Contrato contrato,
        double percentual) {

        ...

        //em.merge(contrato); //Sai essa linha
        em.unwrap(Session.class).update(contrato); //E entra essa!

        return new AsyncResult<Double>(novoSaldo);
    }
}
```

Através do método `unwrap` nós recuperamos a instância de `Session` da API do Hibernate que fica por trás do `EntityManager` , e então podemos usar o método `update` . A diferença é que agora nossa

aplicação não está mais portátil entre diferentes implementações de JPA. Mas, caso precisemos mudar, é só buscar o método específico da API da outra implementação que faz a mesma coisa e, em último caso, voltar a linha com `merge` que acabamos de comentar.

Outro ponto sobre a execução paralela é que ela não deu todo o ganho de performance que esperávamos. Mas antes que você fique preocupado, não há nenhum problema ou parte faltando na nossa solução, o que ocorre é comum por estarmos executando um código de exemplo muito simples e também por estarmos executando tudo na nossa máquina local. Como o processamento em si é uma multiplicação extremamente simples, o custo de abrir e delegar isso para uma `Thread` e depois recuperar o valor sai mais caro que simplesmente executar a multiplicação. Além disso estamos executando tanto o servidor de aplicação quanto o banco de dados na mesma máquina, logo não há o envolvimento da camada de rede, que no mundo real existe e faz com que nossa execução paralela tenha ainda mais vantagem.

Então para simular esse tempo de rede em uma aplicação real, e também uma lógica um pouco mais complexa, vamos colocar um pequeno atraso de cinco milissegundos em ambas implementações e ver o que ocorre.

```
...
public class ContratoBean {
    ...
    @Transactional(timeout = 5, unit = TimeUnit.MINUTES)
    public double corrigeContratos(List<Indice> indices) {

        double saldoGeral = 0.0;
        for(Contrato contrato : listarTodos()){
            Double novoSaldo =
                corrigeContrato(contrato, indices);
            contrato.setSaldo(novoSaldo);
            saldoGeral += novoSaldo;
        }
    }
}
```

```

        try {
            Thread.sleep(5); //código novo
        } catch (InterruptedException e) {
        }
    }

    return saldoGeral;
}
...
}

```

Apenas adicionamos um `Thread.sleep(5)` para que esses cinco milissegundos simulem um cálculo real e também a persistência desse cálculo em um banco de dados remoto. Agora vamos fazer o mesmo com o cálculo em paralelo.

```

...
public class AjustadorContratoBean {
    ...
    @Asynchronous
    public Future<Double> ajustaContratoPercentualAsync(
        Contrato contrato,
        double percentual) {

        System.out.println("executando... "
            + Thread.currentThread());

        Double novoSaldo =
            contrato.getSaldo() * (1 + percentual);
        contrato.setSaldo(novoSaldo);

        try {
            Thread.sleep(5); //código novo
        } catch (InterruptedException e) {}

        em.unwrap(Session.class).update(contrato);
        return new AsyncResult<Double>(novoSaldo);
    }
}

```

Agora executando novamente os dois códigos, vemos que a diferença já se mostra, pois está mais parecido com um caso do mundo real. Com isso conseguimos o mais importante deste capítulo, que era ver como de uma forma simples podemos executar algo em segundo plano para não fazer o usuário ficar esperando, e como essa mesma técnica pode ser aplicada para fazer nossas tarefas mais custosas executarem melhor, tirando proveito do paralelismo. E quando você conseguir aplicar essa técnica em um ambiente real, vai ver que a tendência é que os resultados sejam ainda melhores.

Apenas como exemplo, fazendo em um projeto real o que fizemos aqui, vi uma tarefa que levava mais de dois dias passar a ser executada em pouco mais de duas horas. Pode parecer muito, mas pense, se no nosso exemplo conseguimos ver diferença ao forçar um tempo de execução de cinco milissegundos, imagine se cada unidade dessa levasse um minuto. E imagine o tempo ocioso de diversos núcleos de processamento do nosso servidor enquanto uma tarefa tão grande é executada sequencialmente.

Como dito antes, muitas das vezes para ganhar performance temos mais que **pensar** em como fazer direito do que gastar uma enorme quantidade de energia tentando ganhar um milissegundo aqui e outro ali. No caso que comentei, pouco adiantaria reduzir uns milissegundos (ou mesmo segundos) em cada operação. Ou ainda que fosse um ganho maior, na ordem de 50% (cinquenta por cento), ainda assim demoraria mais de um dia para executar. Então temos sempre que nos lembrar de que desenvolvimento de software é um trabalho intelectual, e não braçal.

Está bom, mas ainda não acabou

Ok, conseguimos fazer o que queríamos no princípio do capítulo, mas ainda temos mais para refinar. Ainda teremos uma seção para ver como implementar o que fizemos via `@Asynchronous` utilizando a *Concurrency Utilities for Java EE*, introduzida no Java EE 7. Mas antes ainda falta fazer nossa execução paralela funcionar de forma

também assíncrona. O processo é exatamente o mesmo que fizemos antes quando desenvolvemos a primeira versão assíncrona, então não colocarei aqui o código Java, mas ele está disponível no GitHub com os demais códigos da nossa aplicação. Porém, para adquirir fluência, sugiro fazer o ajuste por conta própria e depois apenas conferir se ficou parecido com a solução do repositório.

A diferença do que vamos fazer agora estará na tela, pois usaremos algo melhor do que fazer o usuário ficar clicando para ver o resultado sem saber se o processamento terminou ou não. Inclusive quando fizemos isso deixei uma observação sobre a melhoria no próximo exemplo do tipo. Pois bem, a hora de melhorar chegou. Antes de mais nada, vamos adicionar o *Prime Faces* no projeto, que é uma biblioteca de componentes JSF muito utilizada. Para isso basta baixar a última versão do jar na página do projeto (<http://www.primefaces.org/downloads>). Aqui utilizamos a versão 8.0 community. Ou então, como estamos usando o Maven, basta adicionar o seguinte trecho de código no `pom.xml`, lembrando que a versão completa está no GitHub.

```
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>8.0</version>
</dependency>
```

Agora, já na nossa tela, vamos adicionar o *Prime Faces* nas *namespaces* da nossa página.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
    xmlns:p="http://primefaces.org/ui">
```

...

</html>

Agora podemos utilizar os componentes do *Prime Faces* através do prefixo `p`. Então vamos adicionar o seguinte trecho no final do código da nossa tela:

...

<h:commandButton

value="Atualizar saldo dos contratos paralelo assincrono"

action="#{controller.aplicaCorrecaoParalelaAssincrona()}" />

<h:panelGroup id="processamentoParaleloAssincrono">

ultima execução:

#{controller.benchmarks['parAsync'].duracao} ms;

saldo geral:

<c:set var="saldoParaleloAssincrono"

value="#{controller.benchmarks['parAsync'].resultadoAsync}" />

<h:outputText value="#{saldoParaleloAssincrono.get()}"

rendered="#{saldoParaleloAssincrono.done}">

<f:convertNumber type="currency" />

</h:outputText>

<h:outputText value="não calculado"

rendered="#{empty saldoParaleloAssincrono}" />

<h:panelGroup rendered="#{not empty saldoParaleloAssincrono
and !saldoParaleloAssincrono.done}">

<h:outputText value="calculando..." />

<h:graphicImage library="images" name="processando.gif" />

</h:panelGroup>

<p:poll update="processamentoParaleloAssincrono"

stop="#{empty saldoParaleloAssincrono or
saldoParaleloAssincrono.done}" />

</h:panelGroup>

...

Apesar de o código Java como um todo não ser replicado aqui, só como contexto, o método de ação chamado no nosso *controller* é o `aplicaCorrecaoParalelaAssincrona()`. Ele chama a execução assíncrona e guarda o resultado na variável `saldoParaleloAssincrono` (por meio do `c:set`) que, assim como no exemplo anterior, é do tipo `Future<Double>`.

Mesmo esse sendo um código grande, é praticamente igual ao do botão "Atualizar saldo dos contratos assincrono". A diferença é que criamos algumas variáveis para encurtar o código das `EL`s. Então no geral a explicação é a mesma que foi dada quando vimos aquele botão, e está bem ligada ao funcionamento do tipo `Future` que usamos aqui através da variável `saldoParaleloAssincrono`.

Uma pequena melhoria que fizemos foi colocar um gif para indicar o processamento em segundo plano. E para que esse gif e o texto ficassem unidos, utilizamos um `h:panelGroup` em volta deles. A propriedade `library` indica a pasta abaixo da pasta *resources* onde se encontra a imagem. Logo, o caminho completo dessa imagem seria `src/main/webapp/resources/images/processamento.gif`.

A verdadeira diferença está no `p:poll`, que usamos para de tempos em tempos atualizar o `h:panelGroup` com id `processamentoParaleloAssincrono` que envolve praticamente todo esse código. Por padrão, essa atualização é feita a cada dois segundos, e não alteramos esse comportamento.

Além do componente que será atualizado, temos a condição de parada desse *polling*, que está especificada na expressão do atributo `stop`. Ali definimos que, se o cálculo ainda não foi disparado (`empty saldoParaleloAssincrono`), ou se ele já terminou (`saldoParaleloAssincrono.done`), o componente deve ficar parado. Dessa maneira ele só fica atualizando quando precisa, que é durante a execução.

O resultado final ficará parecido com o da seguinte imagem:

- 10016 contratos sendo atualizados em paralelo e assincronamente (5 ms)

Quantidade de contratos: 10016

Atualizar saldo dos contratos	ultima execução: 0 ms; saldo geral: R\$ 0,00
Atualizar saldo dos contratos assíncrono	ultima execução: 0 ms; saldo geral: não calculado
Atualizar saldo dos contratos paralelo	ultima execução: 0 ms; saldo geral: R\$ 0,00
Atualizar saldo dos contratos paralelo assíncrono	ultima execução: 5 ms; saldo geral: calculando...




Figura 12.3: Processamento assíncrono com feedback visual para o usuário

Para fechar o capítulo

Finalizada essa pequena melhoria, podemos avançar para o funcionamento paralelo utilizando os `Executors`. Mas se da forma como fizemos já alcançamos o resultado, qual seria a vantagem de ver outra forma? Apenas acadêmica? Na verdade, não.

Vamos imaginar a situação onde temos um processamento tão pesado que quando iniciamos sua execução em paralelo o servidor "senta". Obviamente, como nosso exemplo é bem simples, isso não será um problema, mas no dia a dia teremos casos assim. Isso acontece porque por padrão temos 10 (dez) `Threads` para atender a esse tipo de execução em paralelo, mas, como dissemos, pode ser que nosso ambiente não tenha recursos suficientes para executar todas essas operações ao mesmo tempo.

Uma alternativa seria configurar o servidor para que tenha menos execuções assíncronas em paralelo. Funcionaria, mas afetaria todo o servidor, então se temos outra aplicação executando no mesmo ambiente, poderíamos acabar limitando a execução paralela dessa outra aplicação, algo que não gostaríamos de fazer.

Por esse motivo no próximo capítulo vamos ver como trabalhar com a *Concurrency Utilities for Java EE*, que nos dá a opção de fazer o ajuste fino desse tipo de coisa.

12.5 Pós-créditos

Durante este capítulo vimos várias formas de fazer a atualização monetária dos nossos contratos, mas a implementação que usamos simplesmente sempre devolvia 10% (ou 0.1):

```
private double calculaPercentualCorrecao(Contrato contrato,
    List<Indice> indices){
    return 0.1;
}
```

Caso quiséssemos uma implementação mais realista, precisaríamos cadastrar no banco o `IndiceValor` mês a mês do nosso índice, e colocar no nosso contrato uma data de atualização relativamente antiga, para que o índice pudesse ser aplicado para essa correção.

Por exemplo, o `GeradorContratosController` poderia ficar assim:

```
...
Indice igpm = new Indice("IGPM");
igpm.setValores(Arrays.asList(
    new IndiceValor(igpm, YearMonth.now().minusMonths(2), 0.01),
    new IndiceValor(igpm, YearMonth.now().minusMonths(1), 0.02),
    new IndiceValor(igpm, YearMonth.now(), 0.03)
));

for (int i = 0; i < 10_000; i++) {
    Contrato contrato = new Contrato();
    contrato.setValor(1_000_000.0);
    contrato.setIndiceCorrecao(igpm);
    contrato.setUltimaAtualizacao(
        YearMonth.now().minusMonths(2));
    contratoBean.salvar(contrato);
}
...
```

E agora a implementação do `calculaPercentualCorrecao`, usando Java 8, poderia ser assim:

```

private double calculaPercentualCorrecao(Contrato contrato,
    List<Indice> indices){

    //filtra o índice a ser aplicado nesse contrato (ex IGPM)
    Indice indiceAplicar = indices.stream()
        .filter(i -> i.equals(contrato.getIndiceCorrecao()))
        .findFirst().get();

    //filtra os IndiceValor depois da
    //última atualização do contrato
    Stream<IndiceValor> valoresAplicar =
        indiceAplicar.getValores().stream().filter(v ->
            v.getAnoMes()
                .isAfter(contrato.getUltimaAtualizacao()));

    //faz a soma de percentual
    //(para multiplicar precisamos somar 1)
    Double somaIndice = valoresAplicar
        .mapToDouble(v -> v.getValor() + 1)
        .reduce(1, (i1, i2) -> i1 * i2);

    //devolvemos o resultado removendo o 1 adicionado na soma
    return somaIndice - 1;
}

```

Obviamente esse código é somente uma curiosidade, por isso ele está nesta seção. A única coisa que fizemos aqui foi somar os percentuais de todos os `IndiceValor` do `Indice` usado no nosso contrato, que sejam posteriores à última atualização. Por exemplo, se a última atualização do nosso contrato foi em agosto, e hoje estamos em novembro, teremos que aplicar o índice de setembro e de outubro, já que provavelmente em um caso real ainda não teríamos o índice de novembro disponível.

Outro pequeno detalhe é a soma de percentuais, que se faz somando 1 e depois subtraindo esse mesmo 1. Por exemplo, se tivermos 10% + 20% o resultado será 32%. Ou, seguindo nossa conta: $(1.1 * 1.2) - 1 = 0.32$. Note que 10% é expresso como 1.1 já é $0.1 + 1$.

CAPÍTULO 13

Utilizando a Concurrency Utilities for Java EE

Uma introdução do Java EE 7 foi a possibilidade de trabalharmos com `Threads` no Java EE praticamente da mesma maneira que fazemos no Java (SE) 7. Vimos também que no Java EE 8 houve um avanço na criação de clientes reativos usando a nova API disponível no Java 8; porém, como aqui estudaremos a *Concurrency Utilities for Java EE*, trabalharemos com a API do Java 7, já que não houve mudanças nessa API no Java EE 8.

Então vamos ver (ou relembrar) como fazemos para lidar com `Threads` no Java SE para usarmos aqui no EE também.

Basicamente temos duas interfaces para usar: a `java.lang.Runnable`, que geralmente usamos quando vamos executar algo que não tem retorno (ou seja `void`); e a `java.util.concurrent.Callable<V>`, que usa *generics* para dizer que executa uma operação e retorna o tipo `V`.

O primeiro tipo existe no Java desde sempre, e a forma de executá-lo, que também existe desde sempre, é através de uma `Thread`. Algo parecido com o seguinte:

```
public class ClassePreguicosa implements Runnable {
    public void run(){
        System.out.println(
            "eu ia fazer algo, mas deu preguiça...");
    }
}
...
Runnable preguicosa = new ClassePreguicosa();
new Thread(preguicosa).start();
```

Porém, esse jeitão mais simples de tratar as coisas, dando `new` diretamente na `Thread`, mudou muito a partir do Java 5, versão onde também foi incluída a interface `Callable`. A partir do Java 5 passamos a ter a interface `ExecutorService`, do então novo pacote

`java.util.concurrent` , que pelo nome já nos dá a entender que tem várias coisas para nos ajudar com processamento paralelo.

A partir de então, podemos usar a classe utilitária `Executors` - do mesmo pacote - para termos diferentes maneiras de criar uma instância de `ExecutorService` .

Por exemplo, temos a opção de criar uma instância que internamente terá uma única `Thread` através do método `newSingleThreadExecutor()` . Podemos ainda usar o método `newCachedThreadPool()` para criar um `ExecutorService` que terá um número variável de `Threads` internas, e como o nome dá a entender, elas são mantidas (em cache) temporariamente para serem reutilizadas, e se ficarem inativas por mais de 60 (sessenta) segundos, serão destruídas automaticamente. Em compensação, se for enviado um número muito grande de processamento de uma vez, várias `Threads` serão criadas e colocadas em execução ao mesmo tempo, logo é uma boa opção para serviços leves e esporádicos.

Se precisar de um uso mais pesado, podemos usar o método `newFixedThreadPool(int qtdeThreads)` , onde estará disponível um número fixo de `Threads` e haverá uma fila para controlar o trabalho excedente, assim conseguimos executar um trabalho pesado sem comprometer o ambiente com centenas (milhares?) de `Threads` sendo colocadas para trabalhar ao mesmo tempo.

Por fim, no Java 8 foi adicionado o método `newWorkStealingPool()` que, baseado na quantidade de processadores a que o Java tiver acesso, vai tentar manter um nível de paralelismo compatível. Por exemplo, se o Java pode usar 4 (quatro) processadores, ele vai gerenciar quantas `Threads` e filas achar necessário para tentar manter 4 (quatro) `Threads` ativas por vez.

Depois que temos a instância de `ExecutorService` mais adequada para nosso caso, basta submetermos o trabalho para ela, mais ou menos assim:

```

public class ClassePreguicosa implements Runnable {
    public void run(){
        System.out.println(
            "eu ia fazer algo, mas deu preguiça...");
    }
}

public class CalculadoraPreguicosa implements Callable<Integer> {
    public Integer call(){
        System.out.println(
            "eu ia calcular, mas deu preguiça...");
        return 42;
    }
}

...
Runnable runnable = new ClassePreguicosa();
Callable<Integer> callable = new CalculadoraPreguicosa();

ExecutorService executorService =
    Executors.newWorkStealingPool();
Future<?> runnableResult = executorService.submit(runnable);
Future<Integer> callableResult =
    executorService.submit(callable);
executorService.shutdown();

```

Como o `Runnable` é `void`, o `Future` retornado serve apenas para acompanharmos o progresso da execução da tarefa. Se invocarmos o método `get()` dele receberemos `null` quando a execução terminar. E ao final não podemos esquecer de finalizar nossa `ExecutorService` através do `shutdown()`, pois somente assim os recursos serão liberados.

Esse foi um bom *overview* que servirá como bagagem para nosso trabalho daqui para a frente. A vantagem é que a decisão de como instanciar o `ExecutorService` fica para o servidor, além é claro do habitual controle do ciclo de vida dos objetos que são gerenciados, ou seja, também não precisamos chamar o `shutdown()`. Para isso,

precisamos injetar o `ExecutorService` (na verdade, um subtipo mais específico) através de `@Resource` .

Então voltemos à classe `ContratoBean` para fazer uma nova forma de calcular adicionando o seguinte código.

```
...
@Resource
private ManagedExecutorService executorService;

//estilo 1
public double corrigeContratosViaMES1(List<Indice> indices) {

    List<Future<Double>> resultatos = new LinkedList<>();

    for(Contrato contrato : listarTodos()){
        double percentual =
            calculaPercentualCorrecao(contrato, indices);

        resultatos.add(
            executorService.submit(
                new AjustadorContratoCallable(
                    contrato,
                    percentual,
                    ajustadorContratoBean)
            )
        );
    }

    double saldoGeral = 0.0;

    try {

        for (Future<Double> resultado : resultatos) {
            saldoGeral += resultado.get();
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



```
        return saldoGeral;
    }
    ...
```

Nesse trecho de código podemos ver que na verdade injetamos o tipo `ManagedExecutorService`, que é subtipo de `ExecutorService`. Então tudo o que vimos sobre ele continua válido, a diferença é que esse subtipo é gerenciado pelo servidor.

RETOMANDO O MANAGEDEXECUTORSERVICE VISTO QUANDO ESTUDAMOS REACTIVE CLIENT API

Você deve se lembrar que quando estudamos Reactive Client API há dois capítulos passamos pelo `ManagedExecutorService` sem explicar muita coisa, pois veríamos com mais detalhes mais adiante.

Pois bem, acabamos de entender melhor o funcionamento desse objeto através do estudo (ou revisão) do seu tipo mais simples (`ExecutorService`) disponível no Java SE.

O uso dele é exatamente como vimos no `ExecutorService`, usando o método `submit(Callable)` para disparar a execução. E agora, vamos ao código da classe `AjustadorContratoCallable`.

```
public class AjustadorContratoCallable
    implements Callable<Double> {

    private Contrato contrato;
    private double percentual;
    private AjustadorContratoBean ajustadorContratoBean;

    public AjustadorContratoCallable(Contrato contrato,
        double percentual,
        AjustadorContratoBean ajustadorContratoBean) {
```

```

        this.contrato = contrato;
        this.percentual = percentual;
        this.ajustadorContratoBean = ajustadorContratoBean;
    }

    @Override
    public Double call() throws Exception {

        System.out.println("executando >> "
            + Thread.currentThread());

        return ajustadorContratoBean
            .ajustaContratoPercentualDireto(
                contrato, percentual);
    }
}

```

Aqui simplesmente criamos uma implementação de `Callable` que será usada para fazer o ajuste do contrato usando o `AjustadorContratoBean`.

Como queremos fazer o cálculo paralelo, mas também persistir de forma paralela como vimos antes, precisaremos fazer uso do EJB dentro da nossa `Callable`. Porém, como vamos dar `new` nessa classe, não conseguimos usar injeção dentro dela. Teríamos que fazer um *lookup* ou receber o EJB pronto pelo construtor, sendo essa segunda a abordagem que usamos. Com o EJB em mãos podemos chamar seu método `ajustaContratoPercentualDireto`.

Apesar de já termos visto o código do `AjustadorContratoBean`, esse método que acabamos de ver é novo. Até então o único método que tínhamos lá dentro era o `ajustaContratoPercentualAsync`, mas agora teremos o seguinte conteúdo.

```

...
@Asynchronous
public Future<Double> ajustaContratoPercentualAsync(
    Contrato contrato, double percentual) {

```

```

        Double novoSaldo = ajustaContratoPercentualDireto(
            contrato, percentual);
        return new AsyncResult<Double>(novoSaldo);
    }

    //método novo, sem @Asynchronous
    public Double ajustaContratoPercentualDireto(
        Contrato contrato, double percentual) {

        System.out.println("executando... "
            + Thread.currentThread());

        Double novoSaldo = contrato.getSaldo() * (1 + percentual);
        contrato.setSaldo(novoSaldo);

        try {
            Thread.sleep(5);
        } catch (InterruptedException e) {}

        em.unwrap(Session.class).update(contrato);
        return novoSaldo;
    }
    ...

```

A mudança foi simples, basicamente tivemos dois métodos, sendo que um calcula diretamente enquanto o outro (async) chama o primeiro (direto) para fazer a operação de forma assíncrona, mas com o cálculo exatamente igual. A importância disso é que nossa instância de `Callable` já vai executar em um ambiente com paralelismo via `ManagedExecutorService`, então o cálculo em si pode ser feito diretamente como fizemos agora.

Agora que vimos a classe `AjustadorContratoCallable` já temos todo o código de que precisamos para nosso exemplo. Vai faltar só uma configuração, mas, aproveitando que estamos vendo bastante código, antes de configurar, vamos olhar como seria um segundo estilo de chamar várias tarefas sem ter que chamar o método `submit` para cada uma.

```

...
@Resource
private ManagedExecutorService executorService;

//estilo 2
public double corrigeContratosViaMES2(List<Indice> indices) {

    List<Callable<Double>> tasks = new LinkedList<>();

    for(Contrato contrato : listarTodos()){
        double percentual =
            calculaPercentualCorrecao(contrato, indices);

        tasks.add(new AjustadorContratoCallable(
            contrato,
            percentual,
            ajustadorContratoBean));
    }

    double saldoGeral = 0.0;

    try {

        List<Future<Double>> resultatos =
            executorService.invokeAll(tasks);

        for (Future<Double> resultado : resultatos) {
            saldoGeral += resultado.get();
        }

    } catch (Exception e) {
        e.printStackTrace();
    }

    return saldoGeral;
}
...

```

Como podemos ver, por nosso código ser bastante simples, não muda muita coisa. Porém, essa forma nos dá a possibilidade de, por exemplo, termos algum método especializado em descobrir o que

precisa ser feito e já devolver a lista de tarefas a serem executadas. Assim bastaria invocar sua execução através do método `invokeAll(Collection<? extends Callable<T>> tasks)`, e ele já devolve a lista de resultados.

Essa lista possui a mesma ordem na qual as tarefas foram passadas, ou seja, o primeiro elemento da lista retornada corresponde ao primeiro elemento da lista de tarefas e assim sucessivamente. O restante não muda.

Para completar esse exemplo vamos ao código da tela:

```
...
<h:commandButton
    value="Atualizar saldo dos contratos paralelo (via MES)"
    action="#{controller.aplicaCorrecaoViaMES()}" />

ultima execução: #{controller.benchmarks['mes'].duracao} ms;
saldo geral:
<h:outputText value="#{controller.benchmarks['mes'].resultado}">
    <f:convertNumber type="currency" />
</h:outputText>
...
```

Novamente seguimos o mesmo padrão onde um botão chama a ação e logo depois é exibido o saldo atualizado pela operação disparada. E para finalizar os códigos dessa seção, vejamos como fica o controlador.

```
...
public void aplicaCorrecaoViaMES(){

    Benchmark benchmark = new Benchmark(numeroContratos, "MES");
    benchmarks.put("mes", benchmark);

    double resultado =
        contratoBean.corrigeContratosViaMES2(indices);
    benchmark.setResultado(resultado);

    FacesContext.getCurrentInstance().addMessage(null,
```

```

        new FacesMessage(benchmark.getMessage()));
    }
    ...

```

Mais uma vez seguimos o mesmo padrão dos exemplos anteriores, que é usar a classe `Benchmark` para guardar o tempo e o resultado da operação.

Com isso pronto, podemos em fim executar o exemplo completo e analisar o resultado.

Quantidade de contratos: 10016

Atualizar saldo dos contratos	ultima execução: 0 ms; saldo geral: \$0.00
Atualizar saldo dos contratos assíncrono	ultima execução: 0 ms; saldo geral: não calculado
Atualizar saldo dos contratos paralelo	ultima execução: 0 ms; saldo geral: \$0.00
Atualizar saldo dos contratos paralelo assíncrono	ultima execução: 0 ms; saldo geral: não calculado
Atualizar saldo dos contratos paralelo (via MES)	ultima execução: 0 ms; saldo geral: \$0.00

Figura 13.1: Versão final da tela com todos os botões de teste

Ótimo, conseguimos fazer tudo funcionar como gostaríamos; mas na prática apenas trocamos o `@Asynchronous` pelo uso de um `ManagedExecutorService`. Ainda temos aquele problema que comentamos no final do capítulo anterior, que é a possibilidade de termos processamentos muito pesados que precisariam executar com um nível de paralelismo menor. E é claro, podemos também ter o oposto, um servidor bastante parrudo que é utilizado basicamente para processamentos pesados, e que para tirar proveito, precisaríamos de um processamento ainda mais paralelizado.

Uma vez que já dominamos o funcionamento dos `Executors`, vamos partir para o ajuste fino. Mas antes de iniciar qualquer customização, precisamos conhecer o padrão. O Wildfly vem de fábrica com o `ManagedExecutorService` (MES) "default" configurado para trabalhar com uma quantidade de `Threads` simultâneas que está relacionada com a quantidade de *threads* que o processador da máquina consegue trabalhar, assim como o padrão do `@Asynchronous`.

O tamanho da fila nesse `MES` "default" é indefinido, o que significa que a fila é ilimitada, mas poderíamos ajustar, por exemplo, os atributos `"core-threads"` e `queue-length` para valores que consideremos mais apropriados. Só que isso afetaria todos os beans que utilizassem esse `MES`, ainda que de outra aplicação, já que o recurso é do servidor e não da aplicação. Por isso, o mais interessante talvez seja criar um novo `ManagedExecutorService` para que possamos usar nesses casos específicos.

EXECUTANDO O EXEMPLO EM VERSÕES ANTERIORES AO WILDFLY 10

Caso executemos o exemplo em versões anteriores ao Wildfly 10, assim como não temos a configuração automática do *pool* de EJBs Stateless visto no capítulo 4, também não temos uma fila configurada no `ManagedExecutorService` "default", o que ao executar nosso exemplo provavelmente nos daria o seguinte resultado:

```
java.util.concurrent.RejectedExecutionException:
Task org.glassfish.enterprise.concurrent.internal
.ManagedFutureTask@3d863293 rejected from org.glassfish.enterprise
.concurrent.internal.ManagedThreadPoolExecutor@3ba4783a
[Running, pool size = 25, active threads = 25,
  queued tasks = 0, completed tasks = 811]
```

Com essa exceção, o servidor de aplicação está nos dizendo que ele rejeitou execuções pois ele só suporta 25 execuções simultâneas, e acabamos mandando uma carga acima disso. Esses números podem ser ligeiramente diferentes, mas o que importa é a observação da propriedade `queued tasks = 0`.

Para resolver o problema no próprio MES "default", podemos especificar um tamanho de fila com o seguinte comando *CLI*:

```
/subsystem=ee/managed-executor-service=default
:write-attribute(name=queue-length,value=2147483647)
```

No caso, especificamos o valor igual ao `Integer.MAX_VALUE`, para definirmos uma fila ilimitada como temos por padrão a partir do Wildfly 10.

13.1 Criando um novo Managed Executor Service

Como dito antes, em vez de mexer no `MES` "default", o que afetaria qualquer aplicação que estivesse executando na mesma instância do servidor, vamos criar um novo, e então o configuramos de acordo com as necessidades do nosso bean. Vamos chamá-lo de "javacred-contratos", mas obviamente o nome não importa tanto, o mais importante é que, para diferenciar nosso `MES` do "default", vamos colocar nesse novo `MES` uma quantidade de *threads* igual a 5 (cinco) e o tamanho da fila igual a 20.000 (vinte mil).

OS NÚMEROS PROVAVELMENTE SERÃO DIFERENTES NA SUA APLICAÇÃO

Esses valores usados aqui são apenas para fins didáticos. Por exemplo, a quantidade de *threads* padrão em um servidor de 16 cores, onde cada um consegue lidar com 2 *threads* seria de 32 (trinta e duas) *threads*. Para nosso caso pode ser muito, pois cada execução é muito pesada e queremos deixar uma porção menor do servidor fazendo esse trabalho, e por isso vamos criar um `MES` com apenas 5 (cinco). Seria como um QOS (*Quality of Service*) dentro do nosso servidor.

Na sua aplicação real, procure um valor que faça sentido. Ah, e provavelmente não precisaremos mexer no tamanho da fila se estamos usando pelo menos a versão 10 do Wildfly (ou outro servidor que já tenha a fila ilimitada). Novamente estamos alterando aqui por fins didáticos.

Como em outras situações, vamos usar a interface de linha de comando para configurar:

```
/subsystem=ee/managed-executor-service=javacred-contratos:add(  
jndi-name=java\:jboss\ee\concurrency\executor\javacred\contratos,  
core-threads=5, queue-length=20000)
```

Apesar de o comando CLI funcionar muito bem, a parte chata aqui é termos que fazer o *scape* dos caracteres `:` e `/` dentro do caminho

jndi . Podemos também editar o arquivo de configuração standalone-full.xml e deixar da seguinte maneira.

```
...
<managed-executor-services>
  <managed-executor-service name="default" ... />
  <managed-executor-service name="javacred-contratos"
    jndi-name=
      "java:jboss/ee/concurrency/executor/javacred/contratos"
    core-threads="5" queue-length="20000"/>
</managed-executor-services>
...
```

É só colocar uma nova entrada de `<managed-executor-service>` com as configurações de que precisamos. Por fim, podemos utilizar ainda a interface gráfica, que pode variar a depender da versão do Wildfly utilizada. Como essa é a parte que geralmente mais muda entre as versões, recomendo procurar na documentação como fazer via interface de administração gráfica.

Agora sim, independentemente da forma que usamos para configurar, conseguimos usar um `ManagedExecutorService` customizado. Só precisamos lembrar de mudar qual *MES* estamos injetando. Então vamos fazer o seguinte ajuste no nosso código.

```
...
//sai essa versão
@Resource
private ManagedExecutorService executorService;

//entra essa, com a indicação do caminho JNDI que criamos
@Resource(lookup=
  "java:jboss/ee/concurrency/executor/javacred/contratos")
private ManagedExecutorService executorService;
...
```

Como podemos ver, apenas colocamos o caminho *JNDI* que definimos ao criar o novo *MES* e sair usando. Mas como não poderia deixar de ser, sempre que vemos como melhorar o nosso exemplo, como acabamos de fazer ao criar um *executor*

customizado, vemos em seguida que temos outras formas de tratar o mesmo problema. Afinal de contas, isso é computação, e geralmente ou não se tem a solução de algo, ou se tem várias.

13.2 Criando um Executor Service programaticamente

Apesar de já termos conseguido customizar um `ManagedExecutorService` para nosso exemplo específico, ainda tivemos que criar algo no servidor e usar na nossa aplicação. Não que isso seja necessariamente um problema, mas é mais flexível fazermos nós mesmos a criação programática de um objeto. Geralmente fazer isso em um ambiente Java EE não é uma boa ideia, pois como já sabemos não adianta dar `new` na classe de um EJB, já que ele precisa ser gerenciado pelo contêiner para ter seus "superpoderes".

Quando trabalhamos com `Threads` no Java EE é ainda pior, pois é algo desaconselhado a menos que façamos via os *"ManagedAlgumaCoisa"*. O que faremos a seguir parece até fugir dessa regra, mas veremos que não é bem assim.

Mesmo quando vimos os `ExecutorServices` no Java SE, não precisamos criar nenhum manualmente porque usamos a classe utilitária `Executors` para fazer isso por nós. Mas podemos fazer isso por meio da implementação de `ExecutorService` chamada `ThreadPoolExecutor`, que usaremos agora. Para não ficar muito teórico vamos primeiro ver como instanciá-la e depois procuramos entender melhor o que foi feito. Esse código deverá ser colocado no `ContratoBean`.

```
...
@Resource(name="concurrent/tf/DefaultThreadFactory")
private ManagedThreadFactory managedThreadFactory;

private ExecutorService _executorService;
```

...

```
private ExecutorService getExecutor() {

    if (_executorService == null) {
        int corePoolSize = 5;
        int maximumPoolSize = 5;
        long keepAliveTime = 10;
        int tamanhoFila = 20_000;

        _executorService = new ThreadPoolExecutor(
            corePoolSize,
            maximumPoolSize,
            keepAliveTime,
            TimeUnit.MILLISECONDS,
            new ArrayBlockingQueue<Runnable>(tamanhoFila),
            managedThreadFactory);
    }

    return _executorService;
}
```

Agora em vez de usar diretamente o `_executorService`, basta usarmos o `getExecutor()` que teremos um `ExecutorService` customizado programaticamente. Repare que foi usado um nome esquisito na propriedade para evitar que a usemos diretamente. Fizemos também a partir de um *getter* em vez de inicializar via `@PostConstruct` porque como estamos criando `Threads`, é mais interessante fazer só se realmente precisar.

O nome que usamos no `@Resource` é um padrão da Java EE, então podemos utilizá-lo para recuperar o `ManagedThreadFactory`. E esse objeto é o motivo por que nossa solução não foge à regra de "não sair instanciando coisas dentro do Java EE", e em vez disso usar o que o servidor disponibiliza para nós. Como a classe `ThreadPoolExecutor` aceita uma `ThreadFactory` no construtor, passamos a versão gerenciada pelo contêiner, e então no final das contas não

vamos instanciar `Threads` diretamente, pois nosso `Executor` usará a `ManagedThreadFactory` para fazer o serviço.

Os valores usados na instanciação da `ThreadPoolExecutor` são apenas de exemplo. Em resumo, como estamos utilizando uma fila para guardar as execuções futuras, podemos usar o mesmo valor em `corePoolSize` e em `maximumPoolSize`, pois as execuções que estarão esperando ficarão na fila. Da mesma maneira, o tempo para a `Thread` ficar viva antes de ser removida faz mais sentido quando não temos uma fila e usamos os dois primeiros atributos para criar uma flutuação entre um mínimo e um máximo de `Threads`. Nesse caso o `keepAliveTime` indicaria quanto uma `Thread` poderia ficar ociosa antes de ser destruída caso o número total dentro do *pool* estivesse maior que o `corePoolSize`.

Agora para testar podemos pegar algum dos métodos que usavam o `MES` para usar essa forma programática. Por exemplo, podemos fazer isso no método `corrigeContratosViaMES1` que, em vez de usar diretamente a propriedade `executorService`, passa a usar o método `getExecutor()`.

Para fechar o capítulo

Somando este e o capítulo anterior, tivemos um estudo longo sobre processamento assíncrono no Java EE. Vimos que não devemos deixar o usuário esperando por uma execução longa. Devemos executá-la em segundo plano e deixar o usuário livre para continuar trabalhando. Vimos também que execuções em segundo plano como essa para deixar o usuário trabalhar nada mais são do que executar a tarefa em uma `Thread` diferente, e então usamos diferentes maneiras de executar algo em múltiplas `Threads`, mas dessa vez para ganhar performance.

Pudemos perceber que temos mais de uma forma de fazer coisas parecidas, e obviamente isso não é exclusividade desse assunto. O interessante aqui é que vimos como evoluir o nível de controle que temos sobre o paralelismo da nossa aplicação. Apesar do volume

total de código que vimos nesses dois capítulos, boa parte foi código de apoio para nossos comparativos, e não código necessário em uma situação real. Para fazer o que realmente precisaremos fazer no dia a dia, a implementação costuma ser bem simples.

Porém, é obvio que implementação simples não significa solução simples. Muitos desenvolvedores usam uma ou outra forma que foi apresentada aqui mas não compreendem tudo que foi discutido nesses capítulos, como o comportamento das transações quando temos execução paralela, só para exemplificar. Então se você teve a dedicação de passar por cada tópico e executar os exemplos para perceber seu funcionamento, parabéns, provavelmente você sabe mais sobre esse assunto que a grande maioria dos desenvolvedores.

Além de concluir nosso capítulo, concluimos também a segunda parte deste livro, onde as ferramentas mais usuais foram apresentadas com uma profundidade suficiente para nos virarmos muito bem na maioria das situações que aparecerem diante de nós.

CAPÍTULO 14

Conclusão

Como foi dito na introdução deste livro, a ideia era de que sua primeira parte servisse como um guia para início rápido, enquanto a segunda parte fosse tanto para uma leitura seletiva de cada conteúdo conforme a necessidade, quanto para uma leitura sequencial. De toda forma, ao final você deve ter adquirido um conhecimento bem interessante sobre os temas abordados.

Espero que, apesar de longa, a leitura tenha sido proveitosa e que tenha servido para acabar com alguns mitos envolvendo EJBs. Espero ainda que o que foi aprendido aqui ajude você a ser um(a) profissional mais completo(a), e que tenha sempre muito sucesso na sua carreira.

Procure sempre entender bem as ferramentas que você usa no dia a dia. Assim, provavelmente você se destacará. Infelizmente hoje a maioria das pessoas busca apenas o "fast food" do conhecimento, sabendo apenas fazer algo um pouco mais avançado que um *hello world*, sem dominar de fato o funcionamento das tecnologias usadas.

Apesar de todo o cuidado dos envolvidos na finalização dessa obra, caso você encontre algo que precise ser ajustado, abra um ticket no GitHub, submeta uma errata na página do livro na Casa do Código, ou me mande um e-mail. E qualquer dúvida entre em contato comigo: **gscordeiro@gmail.com**.

Muito obrigado pela leitura, e que Deus abençoe muito sua vida e sua carreira.