



Edição 151 :: R\$ 14,90

Autorizações dinâmicas com Spring Security

Criando uma infraestrutura que permita
a definição de permissões

Java 9: Money API na prática

Trabalhando com a nova API e os
principais recursos do Java 8

Apache Cassandra e Java EE

Modelando dados para realizar
buscas eficientes no Cassandra

 DEVMEDIA



MVC BASEADO EM AÇÕES

Conheça a JSR 371,
novidade da Java EE 8

MVP

R\$ 1.000.000,00
INVESTIDOS EM CONTEÚDO
NOS ÚLTIMOS 12 MESES.

APlique esse investimento
na sua carreira...

E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 151 • 2016 • ISSN 1676-8361

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

Distribuição

FC Comercial e Distribuidora S.A

Rua Teodoro da Silva, 907, Grajaú - RJ

CEP 20563-900, (21) 3879-7766 - (21) 2577-6362

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Java, o logotipo da xícara de café Java e todas as marcas e logotipos baseados em/ou referentes a Java são marcas comerciais ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

FÓRUM DEVMEDIA

**O lugar perfeito para você ficar por dentro
de tudo o que acontece nas tecnologias do
mercado atual**



No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

ACESSE AGORA
www.devmedia.com.br/forum

Sumário

Artigo no estilo Curso

06 – Como usar o Apache Cassandra em aplicações Java EE – Parte 3

[Marlon Patrick]

16 – Money API: programação avançada com os novos recursos do Java 8

[Alexandre Gama]

Conteúdo sobre Novidades

Destaque - Vanguarda

26 – MVC baseado em ações no Java EE

[Marlon Silva Carvalho]

Conteúdo do tipo Mentoring

Destaque - Mentoring

37 – Autorizações dinâmicas com Spring Security

[Emanuel Mineda Carneiro]



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

REVISTAS DIGITAIS



Imagine poder ter acesso a todas as edições das revistas Java Magazine, .NET Magazine, SQL Magazine, Mobile Magazine, Engenharia de Software, ClubeDelphi e Easy Java.

São mais de 4.000 artigos publicados!

Uma verdadeira biblioteca online perfeita para seus estudos!



Para mais informações :

<http://www.devmmedia.com.br/mvp>

 **DEVMEDIA**

Como usar o Apache Cassandra em aplicações Java EE - Parte 3

Conheça neste artigo como modelar dados para realizar buscas eficientes no Cassandra

ESTE ARTIGO FAZ PARTE DE UM CURSO

Como já mencionado nas partes anteriores desse artigo, o movimento NoSQL ganhou força nos últimos anos devido a uma nova gama de requisitos que os bancos de dados relacionais não conseguiram atender de forma adequada, por exemplo: clusterização, alta disponibilidade, capacidade de lidar com volumes gigantescos de dados, tolerância a falhas, execução em nuvem, etc.

A fim de nos aprofundarmos mais nesse assunto, iniciamos essa série de artigos sobre o Apache Cassandra por se tratar de um dos bancos de dados NoSQL mais reconhecidos da indústria. Na primeira parte do artigo, demonstramos conceitualmente a arquitetura do Cassandra, possibilitando um alicerce teórico aos leitores. Na segunda etapa, focamos na parte prática da aplicação de exemplo, onde foram demonstrados diversos aspectos do Cassandra, destacando-se principalmente a integração desse banco com uma aplicação Java EE e a implementação de comandos básicos através de um driver Java (o driver DataStax).

Agora, no terceiro e último artigo da série, iremos finalizar a implementação da aplicação de exemplo acrescentando conceitos de modelagem de dados e desenvolvendo consultas para tirar o máximo proveito dessa modelagem. Ao terminar dessa série, o leitor terá um entendimento mais claro de como uma aplicação que usa o Cassandra guia sua modelagem baseado em consultas, qual a importância dessa abordagem e como isso impacta no desenvolvimento.

Fique por dentro

Este artigo é útil porque apresenta ao leitor estratégias de modelagem de dados no Apache Cassandra, bem como a implementação de consultas para utilizar os melhores recursos desse banco de dados. Esse é um bom início para quem deseja criar soluções que adotem o NoSQL e também para quem deseja fazer uma transição da modelagem de dados tradicional para o formato proposto por essa solução não relacional. Para isso, o artigo demonstra a implementação de uma aplicação Java EE, utilizando tecnologias como WildFly 9, PrimeFaces 5.3, Cassandra 2.2, entre outras.

Para atingirmos esses objetivos, a aplicação WebShelf continuará evoluindo e agregando novas funcionalidades, indo desde buscas mais complexas com CQL, até operações de escrita mais elaboradas utilizando o driver DataStax.

Prateleira de Livros

No decorrer da segunda parte do artigo implementamos as funcionalidades básicas do WebShelf como login, cadastro de usuários e cadastro de livros. Com isso, podemos, agora, focar na principal funcionalidade desse sistema: a criação de uma prateleira de livros online. Através dela o usuário poderá gerenciar todos os seus livros, classificando-os como livros já lidos, livros que está lendo no momento e livros que deseja ler em breve.

Elaborando a página responsiva da prateleira com PrimeFaces

A tela da prateleira de livros será constituída de três partes, para que os livros possam ser exibidos nas seguintes seções: Já Li, Estou Lendo e Pretendo Ler. Além disso, essa página terá um campo para pesquisa de novos livros, como pode ser observado no trecho de código do arquivo *home.xhtml*, apresentado na **Listagem 1**.

Listagem 1.

Código da tela da prateleira de livros (private/home.xhtml).

```
<ui:define name="mainContent">
<h:form prependId="false">
<p:defaultCommand target="searchButton"/>

<div class="ui-fluid">
    <!-- seção com a funcionalidade de pesquisa de livros -->
    <columnClasses="ui-grid-col-1,ui-grid-col-10,ui-grid-col-1"
        styleClass="ui-panelgrid-blank">
        <p:outputLabel for="searchQuery" value="Pesquisar"/>
        <p:inputText id="searchQuery" placeholder="Título, Autor, ISBN"
            value="#{shelfController.searchQuery}" />
        <p:commandButton id="searchButton" value="Pesquisar"
            action="#{shelfController.searchBooks}"
            onclick="if($('#searchQuery').val().trim()==''){return false;}" />
    </p:panelGrid>
    <br />
    <!-- seção com a prateleira Já Li -->
    <numVisible="10" headerText="Já Li" responsive="true"
        itemStyle="text-align: center; margin-right: 10px;">
        <p:panelGrid layout="grid" columns="1"
            styleClass="ui-panelgrid-blank">
            <p:graphicImage value="#{shelfController.shelfImage}" width="150"
                height="200">
                <f:param name="bookisbnImage" value="#{item.book.isbn}" />
            </p:graphicImage>
            <h:outputText value="#{item.book.title}" />
            <h:outputText value="#{item.book.author}" />
            <p:splitButton value="Remover" process="@this" update="@form"
                actionListener="#{shelfController.remove(item.book)}"/>
            <p:menuitem value="Planejo Ler" process="@this" update="@form"
                actionListener="#{shelfController.addToRead(item.book,false)}"/>
            <p:menuitem value="Estou Lendo" process="@this" update="@form"
                actionListener="#{shelfController.addReading(item.book,false)}"/>
        </p:splitButton>
    </p:panelGrid>
    </p:carousel>
    <!-- as seções Estou Lendo e Pretendo Ler ficam daqui em diante -->
</div>
</h:form>
</ui:define>
</ui:composition>
```

A tela de visualização da prateleira pode ser dividida basicamente em quatro partes. Na primeira parte está o campo de pesquisa de novos livros, sendo composta, portanto, de um **p:inputText** que receberá a informação a ser pesquisada. Como indicado pelo placeholder, o usuário poderá digitar o ISBN, título ou autor. Essa área também contém um **p:commandButton**, que será responsável por executar a ação de pesquisa propriamente dita. Tal botão só é acionado caso o campo de pesquisa não esteja vazio. Por fim, temos o componente **p:defaultCommand**, que é usado para direcionar o evento padrão do *Enter* para o botão de pesquisa.

Nas três partes seguintes os livros são exibidos nas seções já mencionadas ("Já Li", "Estou Lendo" e "Pretendo Ler"), todas implementadas respeitando o mesmo modelo e com os mesmos

componentes. Por conta disso, a **Listagem 1** demonstra apenas a seção "Já Li".

Para mostrar os livros essas três seções utilizam o componente **p:carousel**, o qual apresenta um número limitado de livros na tela. Caso o usuário deseje visualizar mais, poderá usar os botões de paginação. Um detalhe importante a observar é que esse componente agora tem um campo **responsive**, o qual setamos como **true**, e como o próprio nome sinaliza, irá gerá-lo de maneira responsiva.

Dentro do **p:carousel** há, ainda, um **p:panelGrid**, que exibe algumas informações do livro, como título, autor e uma imagem ilustrativa. Essa imagem é gerada através da tag **p:graphicImage**, que invoca o método **ShelfController.getShelfImage()** passando o ISBN do livro para obter o stream da figura e então renderizá-la na página. Outro componente utilizado é o **p:splitButton**, que gera um botão com uma opção visível e outras opções dentro de um menu. A opção visível em todos os casos é a ação de remover o livro da prateleira e as opções no menu são as ações de mover o livro de uma seção para outra. Ademais, note que toda a página foi construída usando as features de responsividade do PrimeFaces.

Desenvolvendo o controller da prateleira

Seguindo uma arquitetura MVC, implementaremos nesta seção o controller referente à prateleira online, apresentado nas **Listagens 2 e 3**. Esse controller irá intermediar a UI com a camada de negócio provendo vários métodos, a saber: remover livro (**remove()**), adicionar livro a uma determinada seção (**addToRead()**, **addReading()** e **addRead()**), carregar a imagem do livro (**getShelfImage()** e **getSearchBooksImage()**), entre outros.

Como podemos observar, a classe **ShelfController** é um bean CDI com escopo de sessão, como indicado pela anotação **@SessionScoped**. Nessa classe temos o método **getShelf()**, que é responsável por carregar a prateleira de livros na primeira vez que o usuário acessar a página. Isso é feito com o auxílio do método **ShelfBean.findShelfByUser()**, o qual recebe um usuário como parâmetro e então retorna um objeto **Shelf** devidamente preenchido com os dados obtidos do Cassandra. Após esse carregamento inicial, os livros ficam disponíveis em memória até o usuário efetuar logout ou a sessão expirar.

Em seguida, temos o método **searchBooks()**, que é responsável por obter o texto digitado pelo usuário no campo de pesquisa e então consultar os livros correspondentes através do método **BookBean.findByKeywords()**.

Nota

Para uma aplicação do mundo real essa abordagem pode gerar problemas por consumir muita memória, podendo ser mais indicado um cache com tempo de expiração menor do que a sessão. Contudo, saiba que não há uma receita pronta para isso, sendo necessário, portanto, encontrar a configuração que melhor se adequa à cada situação. Isso pode levar ao cenário, por exemplo, em que pode não ser preciso fazer cache.

Como usar o Apache Cassandra em aplicações Java EE - Parte 3

Listagem 2. Código do controller da prateleira – Primeira parte (webshelf-web).

```
@Named  
@SessionScoped  
public class ShelfController implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    @Inject  
    private ShelfBean shelfBean;  
  
    @Inject  
    private BookBean bookBean;  
  
    @Inject  
    @LoggedInUser  
    private User loggedInUser;  
  
    private Shelf shelf;  
    private String searchQuery;  
    private Map<Long, Book> searchBooks;  
  
    public Shelf getShelf() {  
        if (shelf == null) {  
            shelf = shelfBean.findShelfByUser(new User(loggedInUser.getLogin()));  
        }  
        return shelf;  
    }  
  
    public String searchBooks() {  
        searchBooks = bookBean.findByKeywords(searchQuery);  
        return "searchBooks?faces-redirect=true";  
    }  
  
    public void remove(Book book) {  
        this.shelf.remove(book);  
        RequestContext.getCurrentInstance().showMessageInDialog  
        (new FacesMessage("Livro removido!"));  
    }  
  
    public void addToRead(Book book, Boolean showMessage) {  
        this.shelf.addToRead(book);  
  
        if(showMessage){  
            RequestContext.getCurrentInstance()  
            .showMessageInDialog(new FacesMessage("Livro adicionado na sessão  
Pretendo Ler!"));  
        }  
    }  
  
    public void addReading(Book book, Boolean showMessage) {  
        this.shelf.addReading(book);  
        if(showMessage){  
            RequestContext.getCurrentInstance()  
            .showMessageInDialog(new FacesMessage("Livro adicionado na sessão  
Estou Lendo!"));  
        }  
    }  
  
    public void addRead(Book book, Boolean showMessage) {  
        this.shelf.addRead(book);  
        if(showMessage){  
            RequestContext.getCurrentInstance().showMessageInDialog  
            (new FacesMessage("Livro adicionado na sessão Já Li!"));  
        }  
    }  
}
```

Os livros retornados na consulta são armazenados no atributo de mesmo nome do método (**searchBook**) e após isso o usuário é redirecionado pelo JSF para a tela de consulta de livros através da string de navegação que o **searchBooks()** retorna.

Já o método **remove()**, como próprio o nome diz, recebe um livro como parâmetro e o remove da prateleira através do método **Shelf.remove()**. Logo após, ele adiciona uma mensagem na tela através do objeto **RequestContext** do PrimeFaces.

Os métodos **addXxx()**, por sua vez, fazem basicamente a mesma coisa: adicionam livros na prateleira através de chamadas aos métodos correspondentes na classe **Shelf**. Após adicionar o livro na sua devida prateleira, esse método pode ou não exibir uma mensagem de sucesso baseado no parâmetro **showMessage**.

Por fim, conforme apresentado na **Listagem 3**, a classe **ShelfController** possui os métodos **getShelfImage()** e **getSearchBooksImage()**.

Listagem 3. Código de ShelfController, gerando imagens – Segunda parte.

```
public StreamedContent getShelfImage() {  
    FacesContext context = FacesContext.getCurrentInstance();  
  
    if (PhaseId.RENDER_RESPONSE.equals(context.getCurrentPhaseId())) {  
        return new DefaultStreamedContent();  
    }  
  
    String bookImageld = context.getExternalContext().getRequestParameterMap().  
    get("bookisbnImage");  
  
    if (bookImageld == null) {  
        return null;  
    }  
  
    ShelfItem shelfItem = shelf.findItemByisbn(Long.valueOf(bookImageld));  
  
    if (shelfItem == null) {  
        return null;  
    }  
  
    return new ByteArrayContent(shelfItem.getBook().getImage(), "image/jpeg");  
}  
  
public StreamedContent getSearchBooksImage() {  
  
    FacesContext context = FacesContext.getCurrentInstance();  
  
    if (PhaseId.RENDER_RESPONSE.equals(context.getCurrentPhaseId())) {  
        return new DefaultStreamedContent();  
    }  
  
    String bookImageld = context.getExternalContext().getRequestParameterMap().  
    get("bookisbnImage");  
  
    if (bookImageld == null) {  
        return null;  
    }  
  
    Book book = searchBooks.get(Long.valueOf(bookImageld));  
  
    if (book == null) {  
        return null;  
    }  
  
    return new ByteArrayContent(book.getImage(), "image/jpeg");  
}
```

Esses métodos são responsáveis por carregar a imagem do livro e gerar um objeto **StreamedContent**. A classe **StreamedContent** faz parte do PrimeFaces e pode ser utilizada, por exemplo, para exibir uma imagem na página XHTML baseada num **InputStream** da imagem, não sendo necessário passar um caminho para um arquivo. A diferença entre esses métodos é que **getShelfImage()** obtém a imagem de livros já inclusos na prateleira, enquanto **getSearchBooksImage()** obtém a imagem dos livros retornados na pesquisa.

Criando as classes de modelo da Prateleira

Antes de partir para os métodos de negócio, precisamos criar as classes de domínio para a prateleira online: **Shelf** e **ShelfItem**.

A classe **ShelfItem** representa os livros na prateleira e pode ser visualizada na **Listagem 4**. Ela é composta, basicamente, dos campos status (lido, lendo, a ler), data de cadastro, que simboliza a data/hora em que o livro entrou naquela seção, e livro. Como podemos verificar, trata-se de um simples POJO com algumas anotações da Bean Validation API.

Listagem 4. Código da classe ShelfItem (webshelf-business).

```
package br.com.devmedia.webshelf.model;

import java.time.LocalDateTime;
import javax.validation.constraints.NotNull;

public class ShelfItem {

    @NotNull(message = "Status: não pode estar vazio.")
    private ShelfItemStatus status;

    @NotNull(message = "Data de cadastro: não pode estar vazio.")
    private LocalDateTime insertionDate;

    @NotNull(message = "Livro: não pode estar vazio.")
    private Book book;

    public ShelfItem() {
    }

    public ShelfItem(Book book, ShelfItemStatus status) {
        this.status = status;
        this.book = book;
    }

    //getters and setters

    //hashCode e equals baseado no atributo book
}
```

A classe **Shelf**, por sua vez, representa a prateleira de livros como um todo, sendo formada pelo campo usuário e três coleções que representam os livros em cada uma das seções: lido, lendo e a ler. Além disso, também tem um **ShelfBean** para invocar alguns métodos de negócio em suas operações, conforme demonstra a **Listagem 5**.

Nessa classe temos alguns Maps responsáveis por armazenar os itens da prateleira (**ShelfItem**) onde a chave desses Maps é o ISBN

do livro. A proposta disso é facilitar as pesquisas nessas coleções, já que o ISBN representa uma informação que não se repete. Dessa forma, evita-se iterar por toda a lista sempre que precisar buscar um item. Outra coisa a se notar é que essas coleções de itens são expostas de maneira que não podem ser modificadas fora da classe, pois estão sendo retornadas encapsuladas por meio de coleções imutáveis (**Collections.unmodifiableCollection**). Isso é importante para garantirmos a integridade da representação da prateleira em memória (coleções) e o que está gravado no banco de dados, pois como será visto na **Listagem 6**, a classe **Shelf** propaga as mudanças dessas coleções para o banco de dados, e caso elas pudessem ser modificadas fora da classe **Shelf**, então não haveria a garantia de que as modificações seriam replicadas para o Cassandra.

Listagem 5. Código da classe Shelf (webshelf-business).

```
public class Shelf implements Serializable{

    private static final long serialVersionUID = 1L;
    private User user;
    private Map<Long, ShelfItem> itemsToRead;
    private Map<Long, ShelfItem> itemsReading;
    private Map<Long, ShelfItem> itemsRead;
    private ShelfBean shelfBean;

    public User getUser() {
        return user;
    }

    public Collection<ShelfItem> getItemsToRead() {
        return Collections.unmodifiableCollection(itemsToRead.values());
    }

    public Collection<ShelfItem> getItemsReading() {
        return Collections.unmodifiableCollection(itemsReading.values());
    }

    public Collection<ShelfItem> getItemsRead() {
        return Collections.unmodifiableCollection(itemsRead.values());
    }
}
```

Continuando a explicação da classe **Shelf**, a mesma ainda possui alguns métodos para adicionar livros nas suas prateleiras, conforme pode ser visto na **Listagem 6**.

O que os três métodos **addXxx()** fazem é criar um novo objeto **ShelfItem** contendo o livro passado como parâmetro, o status (lido, lendo ou a ler) desse livro e a data/hora de sua inclusão na prateleira indicada. Em seguida, o livro é buscado nas outras seções da prateleira e caso seja encontrado, é então removido da respectiva coleção.

Ao final, o livro é adicionado na coleção que representa sua nova prateleira e então a classe **ShelfBean** é invocada para realizar as mesmas modificações no Cassandra, já que até então os ajustes foram apenas em memória. Assim, o livro é gravado no banco de dados na sua nova prateleira e removido da prateleira antiga, caso exista.

Como usar o Apache Cassandra em aplicações Java EE - Parte 3

Listagem 6. Código da classe Shelf – adicionando livros (webshelf-business).

```
public void addToRead(Book book) {
    ShelfItem item = new ShelfItem(book, ShelfItemStatus.TO_READ);
    item.setInsertionDate(LocalDateTime.now());

    ShelfItem oldItem = itemsReading.remove(book.getIsbn());
    if(oldItem == null){
        oldItem = itemsRead.remove(book.getIsbn());
    }

    if(itemsToRead.putIfAbsent(book.getIsbn(), item) == null){
        shelfBean.asyncAddShelfItem(user, item, oldItem);
    }
}

public void addReading(Book book) {
    ShelfItem item = new ShelfItem(book, ShelfItemStatus.READING);
    item.setInsertionDate(LocalDateTime.now());

    ShelfItem oldItem = itemsToRead.remove(book.getIsbn());
    if(oldItem == null){
        oldItem = itemsRead.remove(book.getIsbn());
    }

    if(itemsReading.putIfAbsent(book.getIsbn(), item) == null){
        shelfBean.asyncAddShelfItem(user, item, oldItem);
    }
}

public void addRead(Book book) {
    ShelfItem item = new ShelfItem(book, ShelfItemStatus.READ);
    item.setInsertionDate(LocalDateTime.now());

    ShelfItem oldItem = itemsReading.remove(book.getIsbn());
    if(oldItem == null){
        oldItem = itemsToRead.remove(book.getIsbn());
    }

    if(itemsRead.putIfAbsent(book.getIsbn(), item) == null){
        shelfBean.asyncAddShelfItem(user, item, oldItem);
    }
}
```

Para finalizar o código da classe **Shelf**, ainda nos resta mostrar os métodos **remove()** e **findItemByIsbn()**, apresentados na **Listagem 7**.

O método **remove()** recebe um livro como parâmetro que é buscado em cada uma das coleções da classe **Shelf**, e caso seja encontrado em qualquer uma delas, é removido. De maneira semelhante aos métodos **addXxx()**, o método **remove()** altera a coleção em memória e depois repercute essa modificação no Cassandra de maneira assíncrona através da invocação do método **ShelfBean.asyncRemoveShelfItem()**.

Já o método **findItemByIsbn()** é utilizado para encontrar um **ShelfItem** baseado no ISBN de um livro. Esse método recebe um ISBN (**Long**) como parâmetro e realiza uma consulta nas três coleções da classe **Shelf** (**itemsToRead**, **itemsReading** e **itemsRead**) e caso o livro seja encontrado, então o **ShelfItem** que encapsula esse livro é retornado. O método **findItemByIsbn()** é usado no método **ShelfController.getShelfImage()**.

Listagem 7. Código da classe Shelf - removendo livros (webshelf-business).

```
public void remove(Book book) {
    ShelfItem item = itemsToRead.remove(book.getIsbn());

    if(item == null){
        item = itemsReading.remove(book.getIsbn());
    }

    if(item == null){
        item = itemsRead.remove(book.getIsbn());
    }

    if(item != null){
        shelfBean.asyncRemoveShelfItem(user, item);
    }
}

public ShelfItem findItemByIsbn(Long isbn){
    ShelfItem item = itemsRead.get(isbn);

    if(item == null){
        item = itemsToRead.get(isbn);
    }

    if(item == null){
        item = itemsReading.get(isbn);
    }

    return item;
}
```

Modelando a prateleira no Cassandra

Agora que já vimos como foram projetadas as classes **Shelf** e **ShelfItem**, precisamos conhecer a modelagem de dados da prateleira no Cassandra.

No que se refere à prateleira do WebShelf, a principal regra é que os livros mais recentes de cada seção devem ser listados primeiro. Outra regra é que um livro só pode aparecer apenas uma vez na prateleira, seja em qual seção for. Também não há necessidade de pesquisar os livros que já estejam na prateleira. Eles serão simplesmente listados na ordem explicitada. Por fim, a prateleira terá que exibir a imagem de cada livro, bem como outros dados como autor e título.

De acordo com essas regras, gerou-se a tabela da **Listagem 8**.

Listagem 8. Criação da tabela Shelfitem.

```
CREATE TABLE IF NOT EXISTS webshelf.shelfitem (
    user_login text,
    status text,
    insertion_date timestamp,
    book_isbn bigint,
    book_title text,
    book_author text,
    book_country text,
    book_publisher text,
    book_image blob,
    PRIMARY KEY ((user_login, status), insertion_date, book_isbn)
) WITH CLUSTERING ORDER BY (insertion_date DESC, book_isbn ASC);
```

user_login	status	insertion_date	book_isbn	book_author
marlon	READ	Tue Nov 17 00:10:32 BRT 2015	353536456786876	Paulo Silveira
marlon	READ	Mon Nov 16 23:49:43 BRT 2015	123456789	Martin Fowler
marlon	READ	Mon Nov 16 23:41:06 BRT 2015	4567890123	Sujoy Archarya
marlon	READ	Mon Nov 16 23:39:36 BRT 2015	6789012345	Martin Fowler

Figura 1. Ordenação baseada em clustering columns

Como a única pesquisa que se dará nessa tabela é para preencher as coleções da classe **Shelf**, será preciso consultar os itens da prateleira de um usuário de acordo com o status desses itens. Por exemplo, consultar todos os livros do usuário João que estejam com o status lido (**ShelfItemStatus.READ**). Por isso, a partition key da tabela é composta dos campos login do usuário e status.

Nota

A partition key no Cassandra corresponde ao primeiro campo da primary key. Caso você deseje que a partition key seja composta de mais de um campo, poderá colocar esses campos entre parênteses no início da declaração da primary key.

Para garantir a ordenação dos livros conforme a data de cadastro, colocamos o campo *insertion_date* como sendo a primeira clustering column. Além disso, para que a ordenação seja decrescente (livros mais recentes primeiro), foi especificado explicitamente esse tipo de ordenação através da instrução **WITH CLUSTERING ORDER BY**. Isso foi necessário porque a ordenação padrão é ascendente, portanto, caso se queira usar uma ordenação decrescente, é preciso instruir o Cassandra para isso.

Saiba que usar a ordenação das clustering columns garante que a consulta seja executada mais rapidamente, já que essa ordenação está pronta no disco, ou seja, não é necessário que o Cassandra faça qualquer arrumação dos dados em memória. Simplesmente esses dados são exibidos da mesma forma como estão gravados. Sendo assim, em nenhum momento você precisará se preocupar com a cláusula **ORDER BY**, pois os dados estarão sempre organizados conforme as clustering columns. A **Figura 1** exibe o resultado de uma consulta nessa tabela, e através dela é possível observar a ordenação natural pela coluna *insertion_date*.

Completando a primary key da tabela *shelfitem*, o campo *book_isbn* foi incluído porque identifica unicamente um livro. No entanto, com a inclusão da coluna *insertion_date* não há como garantir a unicidade de um livro na prateleira via Cassandra, o que deverá ser feito pela aplicação. Todavia, entende-se que essa

é uma regra que pode ser quebrada sem provocar um grande problema no negócio do WebShelf.

No mais, a tabela também possui os campos de livro. Assim, será possível exibi-los na prateleira sem a necessidade de executar mais consultas, isto é, numa única consulta teremos todos os dados que precisamos (lembre-se da desnormalização).

Desenvolvendo a lógica de negócio da prateleira

A classe **ShelfBean** é responsável pela lógica de negócios da prateleira em conjunto com a classe **Shelf**. Sua principal função é propagar as alterações da prateleira que foram executadas em memória para o banco de dados. O código pode ser visto na **Listagem 9**.

Para iniciarmos a explanação da classe **ShelfBean**, temos o método **asyncRemoveShelfItem()**, que é responsável por deletar um livro da prateleira de um usuário. Observe que a exclusão é feita utilizando todos os campos que compõem a primary key da tabela *shelfitem* como filtro, pois de outra forma resultaria em erro no Cassandra. Isso acontece porque diferentemente do select, que obriga apenas a inclusão da partition key, o delete, requer toda a primary key.

O método **asyncAddShelfItem()**, por sua vez, inclui um livro na prateleira do usuário. Caso esse livro já exista na base de dados, então ele é deletado antes da inclusão para evitar que esteja em duas seções diferentes ao mesmo tempo. Para que essa operação seja atômica, executamos os comandos de deleção e inserção em batch.

Note que tanto o **asyncRemoveShelfItem()** quanto o **asyncAddShelfItem()** serão executados assincronamente, devido à presença da anotação **@Asynchronous** da especificação EJB. Contudo, caso não houvesse a possibilidade de usar EJB, você poderia usar a execução assíncrona do próprio driver, através dos métodos **executeAsync()** do objeto **Session**.

Em seguida, temos o método **findShelfByUser()**, o qual irá consultar todos os itens da prateleira de um usuário e criar um objeto **Shelf** com esses itens devidamente separados em cada seção.

Como usar o Apache Cassandra em aplicações Java EE - Parte 3

Para criar o objeto **Shelf**, além das coleções de itens é necessário passar um **ShelfBean** como parâmetro no construtor da classe. De acordo com a especificação EJB, não é possível usar o **this** nesse cenário, pois para prover as funcionalidades oferecidas pelo EJB, você precisa de um objeto que possa ser interceptado pelo servidor. Assim, o método **SessionContext.getBusinessObject()**

fornecer esse proxy que nós precisamos. Se você passar o **this** como parâmetro, perceberá que as chamadas dos métodos assíncronos não serão assíncronas, pois o objeto será tratado como um objeto comum e não como um EJB.

Continuando a explicação da classe, o método **findItemsByUserAndStatus()** usa a classe **LinkedHashMap** para que os itens

Listagem 9. Código do bean da prateleira, ShelfBean (webshelf-business)

```
@Stateless
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ShelfBean {

    @Resource
    private SessionContext sessionContext;

    @Inject
    private CassandraCluster cassandra;

    @Asynchronous
    public void asyncRemoveShelfItem(User user, ShelfItem item){
        Date oldDate = Date.from(item.getInsertionDate())
            .atZone(ZoneId.systemDefault()).toInstant();

        BoundStatement deleteShelfByUserStatus = cassandra.boundDelete
            ShelfItemByUserStatus();
        deleteShelfByUserStatus.bind(user.getLogin(), item.getStatus().name(),
            oldDate, item.getBook().getLsbn());

        cassandra.execute(deleteShelfByUserStatus);
    }

    @Asynchronous
    public void asyncAddShelfItem(User user, ShelfItem newItem, ShelfItem oldItem){
        BatchStatement batch = new BatchStatement();

        if(oldItem != null){
            Date oldDate = Date.from(oldItem.getInsertionDate())
                .atZone(ZoneId.systemDefault()).toInstant();

            BoundStatement deleteShelfByUserStatus = cassandra.boundDeleteShelf
                ItemByUserStatus();
            batch.add(deleteShelfByUserStatus.bind(user.getLogin(),
                oldItem.getStatus().name(), oldDate, oldItem.getBook().getLsbn()));
        }

        BoundStatement insertShelfItem = cassandra.boundInsertShelfItem();
        insertShelfItem.setString(0, user.getLogin());
        insertShelfItem.setString(1, newItem.getStatus().name());
        insertShelfItem.setDate(2, Date.from(newItem.getInsertionDate())
            .atZone(ZoneId.systemDefault()).toInstant());
        insertShelfItem.setLong(3, newItem.getBook().getLsbn());
        insertShelfItem.setString(4, newItem.getBook().getTitle());
        insertShelfItem.setString(5, newItem.getBook().getAuthor());
        insertShelfItem.setString(6, newItem.getBook().getCountry());
        insertShelfItem.setString(7, newItem.getBook().getPublisher());
        insertShelfItem.setBytes(8, newItem.getBook().getImageBuffer());

        batch.add(insertShelfItem);

        cassandra.execute(batch);
    }

    public Shelf findShelfByUser(User user) {
        Map<Long, ShelfItem> itemsToRead = findItemsByUserAndStatus(user.getLogin(),
            ShelfItemStatus.TO_READ);
        Map<Long, ShelfItem> itemsReading = findItemsByUserAndStatus
            (user.getLogin(), ShelfItemStatus.READING);
    }
}
```

```
Map<Long, ShelfItem> itemsRead = findItemsByUserAndStatus(user.getLogin(),
    ShelfItemStatus.READ);
ShelfBean shelfBean = sessionContext.getBusinessObject(ShelfBean.class);

return new Shelf(user, itemsToRead, itemsReading, itemsRead, shelfBean);
}

private Map<Long, ShelfItem> findItemsByUserAndStatus(String userLogin,
    ShelfItemStatus status) {
    Map<Long, ShelfItem> items = new LinkedHashMap<>();

    BoundStatement selectShelfByUserStatus = cassandra.boundSelectShelfByUser
        Status().bind(userLogin, status.name());

    ResultSet resultSet = cassandra.execute(selectShelfByUserStatus);

    for (Row row : resultSet) {
        ShelfItem item = rowToItem(row);
        items.put(item.getBook().getLsbn(), item);
    }

    return items;
}

private Map<Long, ShelfItem> findItemsByUserAndStatus(String userLogin,
    ShelfItemStatus status) {
    Map<Long, ShelfItem> items = new LinkedHashMap<>();

    BoundStatement selectShelfByUserStatus = cassandra.boundSelectShelfByUser
        Status().bind(userLogin, status.name());

    ResultSet resultSet = cassandra.execute(selectShelfByUserStatus);

    for (Row row : resultSet) {
        ShelfItem item = rowToItem(row);
        items.put(item.getBook().getLsbn(), item);
    }

    return items;
}

private ShelfItem rowToItem(Row row) {
    Book book = new Book();
    book.setLsbn(row.getLong("book_lsbn"));
    book.setTitle(row.getString("book_title"));
    book.setAuthor(row.getString("book_author"));
    book.setCountry(row.getString("book_country"));
    book.setPublisher(row.getString("book_publisher"));
    book.setImage(row.getBytes("book_image").array());

    ShelfItem item = new ShelfItem();
    item.setStatus(ShelfItemStatus.valueOf(row.getString("status")));
    item.setInsertionDate(LocalDateTime.ofInstant(row.getDate("insertion_date")
        .toInstant(), ZoneId.systemDefault()));
    item.setBook(book);

    return item;
}
}
```

sejam mantidos na ordem em que são adicionados ao **Map**. Assim, a ordenação que vem do próprio Cassandra será preservada. O mesmo não aconteceria com implementações mais conhecidas de **Map** como **HashMap**.

Por fim, o método **rowToItem()** demonstra como obter os dados de um **ResultSet** do driver Cassandra, convertendo o objeto **Row** num objeto **ShelfItem**.

Implementando CQL para o gerenciamento da prateleira

Para fechar a funcionalidade de gestão da prateleira, vamos preparar os comandos CQL necessários para a classe **ShelfBean** cumprir seus objetivos. Dessa maneira, será preciso fazer mais algumas modificações na classe **CassandraCluster**, como demonstrado na **Listagem 10**.

Listagem 10. Instruções CQL para gerenciar a prateleira (CassandraCluster).

```
public BoundStatement boundInsertShelfItem(){  
    return prepare("insert into webshelf.shelfitem (user_login,status,  
    insertion_date,book_isbn,book_title,book_author,book_country,  
    book_publisher,book_image) values(?,?,?,?,?,?,?,?,?,?);");  
}  
  
public BoundStatement boundDeleteShelfItemByUserStatus(){  
    return prepare("delete from webshelf.shelfitem where user_login=?  
    and status=? and insertion_date=? and book_isbn=?");  
}  
  
public BoundStatement boundSelectShelfByUserStatus(){  
    return prepare("select * from webshelf.shelfitem where user_login=?  
    and status=?");  
}
```

Pesquisa de livros

Para finalizarmos o desenvolvimento da aplicação, falta apenas disponibilizar a funcionalidade de pesquisa de livros, para que o usuário possa encontrar os livros que deseja e então adicioná-los à sua prateleira.

Elaborando a página responsiva de pesquisa de livros com o PrimeFaces

A tela de pesquisa de livros terá um input para que o usuário digite sua busca e um botão para que a consulta seja executada. Além disso, abaixo do campo de pesquisa será exibido um link cujo destino é a tela de cadastro de livros, que será útil nos casos em que o usuário não encontrar o livro desejado. O resultado da consulta será exibido logo após esses componentes, mostrando os detalhes de todos os itens retornados. Nesse espaço, para cada livro haverá opções para adicioná-lo em qualquer uma das seções na prateleira. A **Listagem 11** apresenta o código dessa tela.

Como pode-se notar, a primeira parte da tela, compreendida pelo primeiro **p:panelGrid**, é exatamente igual à tela da prateleira. Temos um campo de pesquisa, um botão de pesquisa e um **p:defaultCommand** para direcionar o *Enter* para esse botão.

Em seguida, temos a área que especifica o link para o Cadastro de Livros. Nesse ponto, há apenas um **p:panelGrid** usando o layout grid para viabilizar a responsividade, um **h:outputText** exibindo

Listagem 11. Tela de pesquisa de livros (private/searchBooks.xhtml).

```
<ui:define name="mainContent">  
    <h:form preprendId="false">  
  
        <div class="ui-fluid">  
            <p:panelGrid columns="3" layout="grid"  
                columnClasses="ui-grid-col-1,ui-grid-col-10,ui-grid-col-1"  
                styleClass="ui-panelgrid-blank">  
                <p:outputLabel for="searchQuery" value="Pesquise" />  
                <p:inputText id="searchQuery" placeholder="Título, Autor, ISBN"  
                    value="#{shelfController.searchQuery}" />  
                <p:commandButton id="searchButton" value="Pesquisar"  
                    action="#{shelfController.searchBooks}"  
                    onclick="if($('#searchQuery').val().trim()==''){return false;}" />  
            </p:panelGrid>  
  
            <p:panelGrid columns="1" layout="grid"  
                columnClasses="ui-grid-col-12" styleClass="ui-panelgrid-blank"  
                style="text-align: center;">  
                <h:panelGroup>  
                    <h:outputText value="Não encontrou o livro que procurava?" />  
                    <h:link value="Cadastre-o!" outcome="insertBook" />  
                </h:panelGroup>  
            </p:panelGrid>  
  
            <p:dataScroller value="#{shelfController.searchBooks}" var="book"  
                chunkSize="10" />  
  
            <f:facet name="loader">  
                <div class="ui-grid ui-grid-responsive">  
                    <div class="ui-grid-row">  
                        <div class="ui-grid-col-6">  
                            <div class="ui-grid-col-1" style="text-align: center;">  
                                <p:commandButton type="button" value="Mais"  
                                    icon="ui-icon-circle-triangle-s" />  
                            </div>  
                        </div>  
                    </div>  
                </div>  
            </f:facet>  
  
            <p:panelGrid columns="2" layout="grid"  
                columnClasses="ui-grid-col-1,ui-grid-col-1"  
                styleClass="ui-panelgrid-blank">  
                <p:graphicImage value="#{shelfController.searchBooksImage}"  
                    height="200" width="150" />  
                <f:param name="bookisbnImage" value="#{book.isbn}" />  
            </p:graphicImage>  
  
            <h:panelGrid columns="1" cellpadding="5">  
                <h:outputText value="#{book.title}" />  
                <h:outputText value="#{book.author}" />  
                <h:outputText value="#{book.publisher}" />  
                <h:outputText value="#{book.country}" />  
                <h:outputText value="#{book.isbn}" />  
            </h:panelGrid>  
        </p:panelGrid>  
  
        <p:panelGrid columns="1" layout="grid"  
            columnClasses="ui-grid-col-2" styleClass="ui-panelgrid-blank">  
            <p:splitButton value="Planejo Ler"  
                actionListener="#{shelfController.addToRead(book,true)}">  
                <p:menuitem value="Estou Lendo"  
                    actionListener="#{shelfController.addReading(book,true)}" />  
                <p:menuitem value="Já Li"  
                    actionListener="#{shelfController.addRead(book,true)}" />  
            </p:splitButton>  
        </p:panelGrid>  
    </p:dataScroller>  
    </div>  
    </h:form>  
</ui:define>
```

Como usar o Apache Cassandra em aplicações Java EE - Parte 3

uma mensagem na página e um **h:link** que redireciona o usuário para cadastrar os livros não encontrados na pesquisa.

Logo após, temos o bloco de código que lista os livros encontrados na pesquisa usando o componente do PrimeFaces **p:dataScroller**. Esse componente foi configurado para listar inicialmente 10 itens (**chunkSize="10"**) e caso o usuário clique no botão *Mais* (**<f:facet name="loader">**), ele irá carregar mais 10 livros, até que não haja mais nenhum item a ser exibido.

Por fim, através do componente **p:splitButton** o usuário poderá adicionar o livro nas seções *Já Li*, *Estou Lendo* ou *Pretendo Ler*. Ao realizar qualquer uma dessas ações uma mensagem de sucesso será informada através do objeto **RequestContext**, obtido no **ShelfController**.

Desenvolvendo a lógica de negócio da pesquisa de livros

Agora que a tela de pesquisa de livros está pronta, implementaremos a consulta propriamente dita na classe **BookBean**. Conforme já mencionado, essa busca deverá pesquisar livros por ISBN, título e autor. A **Listagem 12** apresenta essas modificações.

O primeiro método dessa listagem, **findByKeywords()**, obtém os dados digitadas pelo usuário no campo de pesquisa e então consulta as três tabelas de livros no Cassandra: *book_by_isbn* (caso a **String** de consulta possa ser convertida em um número), *book_by_title* e *book_by_author*. Qualquer resultado encontrado nessas tabelas é armazenado numa coleção e listado para o usuário.

A consulta é realizada nas três tabelas de livros porque cada uma delas nos possibilita fazer um filtro diferente. A tabela *book_by_isbn* nos permite fazer a consulta por ISBN. Já a tabela *book_by_title* propicia a consulta por título, e a tabela *book_by_author* viabiliza a consulta por autor. Essa abordagem tem relação com a recomendação de modelarmos nossas tabelas baseadas em padrões de consulta, o que é comumente utilizado no Cassandra. Normalmente, cada padrão de consulta gera uma tabela diferente. Por isso, nesse caso da pesquisa de livros, temos três padrões e, portanto, três tabelas. Para uma explicação mais detalhada a respeito, recomenda-se a leitura da seção “Modelando o cadastro de livros no Cassandra”, apresentada na segunda parte desse artigo.

Preparando comandos CQL para a pesquisa de livros

Para terminar a implementação da aplicação, a **Listagem 13** demonstra as instruções CQL utilizadas na pesquisa de livros.

Como pudemos confirmar no decorrer desta série, o Apache Cassandra pode e deve ser utilizado em conjunto com aplicações Java EE. Note que não há nenhuma restrição que invalide essa combinação. Na verdade, existem vários benefícios, já que fomos capazes de observar que ambas as tecnologias se complementam no âmbito de uma aplicação distribuída.

No que se refere ao aprendizado do Cassandra, conforme mencionado nas outras partes desse artigo, é importante termos um embasamento teórico a respeito da arquitetura desse banco para conseguirmos nos desenvolver com mais segurança. Além disso, precisamos nos aprofundar ainda mais na questão prática, a fim de adquirirmos proficiência no uso dessa solução NoSQL.

Listagem 12. Implementando a consulta de livros na classe BookBean.

```
public Map<Long, Book> findByKeywords(String keywords){  
    Map<Long, Book> books = new LinkedHashMap<Long, Book>();  
  
    if(Utils.isIntegerNumber(keywords)){  
        findByISBN(keywords, books);  
    }  
  
    findByTitle(keywords, books);  
    findByAuthor(keywords, books);  
    return books;  
}  
  
private void findByAuthor(String keywords, Map<Long, Book> books) {  
    BoundStatement selectBookByAuthor = cassandra.boundSelectBookByAuthor().bind(keywords);  
  
    ResultSet resultSet = cassandra.execute(selectBookByAuthor);  
    for (Row row : resultSet) {  
        Book book = rowToBook(row);  
        books.put(book.getIsbn(), book);  
    }  
}  
  
private void findByTitle(String keywords, Map<Long, Book> books) {  
    BoundStatement selectBookByTitle = cassandra.boundSelectBookByTitle().bind(keywords);  
  
    ResultSet resultSet = cassandra.execute(selectBookByTitle);  
    for (Row row : resultSet) {  
        Book book = rowToBook(row);  
        books.put(book.getIsbn(), book);  
    }  
}  
  
private void findByISBN(String keywords, Map<Long, Book> books) {  
    BoundStatement selectBookByISBN = cassandra.boundSelectBookByISBN().bind(Long.valueOf(keywords));  
  
    ResultSet resultSet = cassandra.execute(selectBookByISBN);  
    Row row = resultSet.one();  
    if(row != null){  
        Book book = rowToBook(row);  
        books.put(book.getIsbn(), book);  
    }  
}  
  
private Book rowToBook(Row row) {  
    Book book = new Book();  
    book.setIsbn(row.getLong("isbn"));  
    book.setTitle(row.getString("title"));  
    book.setAuthor(row.getString("author"));  
    book.setCountry(row.getString("country"));  
    book.setPublisher(row.getString("publisher"));  
    book.setImage(row.getBytes("image").array());  
    return book;  
}
```

Listagem 13. Comandos CQL para a pesquisa de livros (CassandraCluster).

```
public BoundStatement boundSelectBookByISBN(){  
    return prepare("select * from webshelf.book_by_isbn where isbn=?");  
}  
  
public BoundStatement boundSelectBookByTitle(){  
    return prepare("select * from webshelf.book_by_title where title=?");  
}  
  
public BoundStatement boundSelectBookByAuthor(){  
    return prepare("select * from webshelf.book_by_author where author=?");  
}
```

Outro aspecto que vale relembrar é a necessidade de entendermos quando o Cassandra é ou não recomendado, pois não podemos pensar que o mesmo resolve todos os problemas. Para isso, aconselhamos o estudo de sua documentação, bem como a apreciação de artigos relacionados na DevMedia e em cases e palestras de outros grandes especialistas.

Com relação às principais features do Cassandra, deixamos aqui um breve resumo do que pudemos aprender nesse artigo:

- **Escalabilidade Horizontal:** escalar horizontalmente através da adição de novas máquinas ao cluster de maneira simples, sem necessidade de paradas, reinicializações ou qualquer tipo de indisponibilidade;
- **Masterless:** todos os nós no cluster podem ler e escrever;
- **Alta disponibilidade:** através do suporte à replicação de dados e de sua natureza masterless, é possível eliminar pontos únicos de falha e ter uptime constante;
- **Tolerância a falhas:** caso algum nó do cluster tenha problemas, o mesmo pode ser restaurado ou trocado sem necessidade de paralisar os demais;
- **Multi-data center/Cloud:** suporte à replicação através de vários data centers, inclusive geograficamente distribuídos, bem como à múltiplas zonas na cloud, tanto para escrita quanto para leitura;
- **Consistência de dados tunável:** o desenvolvedor pode escolher entre consistência forte ou eventual em uma variedade de opções.

Por fim, podemos dizer que o leitor que acompanhou os três artigos dessa série possui agora um conhecimento básico sobre o Cassandra a ponto de iniciar a utilização dessa tecnologia em projetos do mundo real.

Autor



Marlon Patrick

marlon.patrick@mpwtecnologia.com - marlonpatrick.info



É bacharel em Ciência da Computação e atua como Consultor Java no Grupo Máquina de Vendas com aplicações de missão crítica. Tem oito anos de experiência com tecnologias Java e é certificado SCJP 5.

Links:

DataStax - Apache Cassandra Product Guide.

<http://docs.datastax.com/en/cassandra/2.2/index.html>

Advanced data modeling with Apache Cassandra.

<http://pt.slideshare.net/patrickmcfadin/advanced-data-modeling-with-apache-cassandra>

Understanding How CQL3 Maps to Cassandra's Internal Data Structure.

<https://www.youtube.com/watch?v=UP74jC1kM3w>

Cassandra Modeling Kata.

<https://github.com/allegro/cassandra-modeling-kata>

Documentação do PrimeFaces.

<http://www.primefaces.org/documentation>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Money API: programação avançada utilizando os recursos do Java 8

Saiba como utilizar um dos destaques do Java 9 em conjunto com os principais recursos do Java 8

Em qualquer sistema, seja ele financeiro, jurídico ou contábil, é comum trabalharmos com valores monetários para representar os preços de ações, de produtos e cálculos financeiros complexos, por exemplo. Para isso, ao longo dos anos a comunidade adotou como padrão de mercado os tipos **Double**, **Float** ou **BigDecimal**. Contudo, visto que não são tipos voltados para essa necessidade, muitos desenvolvedores passaram por diferentes problemas durante a implementação, como:

- Imprecisão no arredondamento;
- Conversão de moedas manual e limitada;
- Passos desnecessários para operações simples, como soma e subtração;
- Dificuldade na realização de operações mais avançadas, como ordenações e agrupamentos, bem como estatísticas básicas (ex.: média, valores mínimo e máximo, entre outras).

Essa forma padronizada que usamos até hoje se deu pela inexistência de uma API formal do Java para manipular valores em um sistema. Se você já precisou usar o tipo **BigDecimal** ou o **Double** para representar preço em alguma aplicação, provavelmente você já passou pelos problemas supracitados, correto?

Como esse problema de manipulação de valores é recorrente, o Java 9 está trazendo uma opção nativa para o tratamento de dados monetários, a Money API, cuja especificação está sendo desenvolvida sob a JSR 354.

Uma introdução sobre esta API foi feita na edição 145 da Java Magazine. Este artigo, por sua vez, cobrirá os assuntos não abordados e mais avançados. Ainda assim, caso não tenha acompanhado a edição anterior, não se preocupe, pois analisaremos cada exemplo detalhadamente.

Fique por dentro

Neste artigo você vai aprender a desenvolver sistemas que manipulam valores monetários sem precisar lidar com os tipos **Double**, **Float** ou **BigDecimal**, comumente utilizados para suportar esse tipo de requisito. Isso é possível porque o Java 9 será lançado com a Money API, uma nova especificação que possibilitará uma forma mais simples, elegante e profissional de lidar com valores e moedas.

Além de analisar a Money API, apresentaremos um resumo das principais novidades da API do Java 8, que foi largamente utilizada na construção dessa especificação. A partir disso, veremos diversos exemplos e códigos que demonstrarão problemas do mundo real, e como esta nova API facilita a resolução de cada um.

Ademais, é válido ressaltar que o foco do nosso estudo, obviamente, estará nos recursos da Money API, mas para melhorar o aprendizado, vamos implementar o código dos exemplos em classes simples usando o framework JUnit, que permitirá que criemos métodos de teste que facilitarão a visualização dos resultados esperados.

Revisando as novidades do Java a partir da versão 8

Antes de iniciarmos nosso estudo sobre a Money API, é importante relembrarmos algumas características que foram adicionadas no Java 8. Precisamos deste conhecimento porque a Money API utiliza amplamente diversos destes novos recursos em sua própria API e por esse motivo não é possível utilizá-la em versões anteriores do Java.

Sabendo disso, vamos revisar, de forma breve, as seguintes novidades: Lambda Expression, Predicate, Functions, Suppliers, Optionals e Stream. Após o estudo dessas features estaremos prontos para aprender sobre os recursos avançados da tecnologia foco deste artigo.

Expressões lambda

Uma das principais mudanças adicionadas na versão 8 do Java é a possibilidade de trabalharmos com expressões Lambda. De forma resumida, uma expressão lambda é uma função que apresenta basicamente duas características principais: podem possuir ou não parâmetros em sua assinatura e seus tipos podem ser explícitos ou podem ser omitidos, permitindo que o compilador consiga inferir os tipos desejados.

As expressões lambda foram inseridas no Java para que a linguagem pudesse ganhar algumas características de linguagens funcionais, como Scala e Clojure. O grande destaque das expressões lambda está na simplicidade de escrita e leitura de funções, que até então eram extremamente verbosas.

Na **Listagem 1** temos um exemplo simples e clássico do uso de threads. Neste código, através de uma classe anônima, criamos uma thread da forma padrão, para impressão de uma frase.

Listagem 1. Código que implementa threads da forma padrão para imprimir uma mensagem, usando classe anônima.

```
@Test
public void imprimeMensagemUtilizandoThreads() throws Exception {
    Runnable runnable = new Runnable() {

        @Override
        public void run() {
            System.out.println("Thread de forma clássica");
        }
    };

    new Thread(runnable).start();
}
```

Note que o método envolve apenas a interface **Runnable** e a classe **Thread**. Contudo, mesmo para um código básico, foi necessária a escrita de muitas linhas.

Na **Listagem 2** apresentamos o mesmo exemplo, mas desta vez declarando uma expressão lambda.

Listagem 2. Código que utiliza threads e expressões lambda para imprimir uma mensagem, sem utilizar classe anônima.

```
@Test
public void imprimeMensagemUtilizandoThreadsComLambda() throws Exception {
    Runnable runnable = () -> System.out.println("Thread utilizando Lambda!");

    new Thread(runnable).start();
}
```

Este método possui o mesmo comportamento que o apresentado na listagem anterior, mas com um código mais limpo e de fácil leitura e compreensão. Ao analisá-lo, percebemos a presença dos parênteses após o operador de igualdade, o que representa a declaração da função que está sendo criada.

No código dessa listagem, o compilador entende que o método **System.out.println()** deverá ser executado dentro do método **run()** da interface **Runnable**, usando o recurso de expressão lambda.

Lembra que o método **run()** da interface **Runnable** não recebe parâmetro e não retorna nenhum valor (**void**)? Note que a função que criamos respeita exatamente essa assinatura. Os parênteses vazios sinalizam a criação de uma função que não possui parâmetros e também não retorna valor; consequentemente, nossa função também será **void**.

Outra característica importante das expressões lambda é que podemos passar uma expressão lambda como parâmetro para um método, reduzindo ainda mais o tamanho do código e tornando-o mais legível e expressivo. Como exemplo, na **Listagem 3** temos o mesmo método implementado com a classe **Thread**, mas que passa a função lambda diretamente como parâmetro para o construtor dessa classe.

Listagem 3. Código mais simples e elegante com threads e expressões lambda.

```
@Test
public void imprimeMensagemUtilizandoLambdaPorParametro() throws Exception {
    new Thread(() -> System.out.println("Mensagem utilizando lambda de forma
        mais simples e elegante")).start();
}
```

Observe que o nosso código agora possui apenas uma linha para a impressão da mensagem, sem contar com as linhas que declaram o nome do método e a anotação **@Test**.

Predicate

A interface **Predicate<T>** é nova no Java 8 e seu principal objetivo é verificar se o objeto informado como parâmetro atende ao requisito do método **boolean test(T t)**. Podemos, inclusive, passar uma expressão lambda para a interface, contanto que essa expressão seja uma interface funcional. Na **Listagem 4** é apresentado um exemplo utilizando a interface **Predicate<T>**.

Listagem 4. Código que utiliza Predicate para validação do tamanho do texto.

```
public class InterfacePredicateTest {

    @Test
    public void deveIndericarQueOTamanhoDoTextoEMaiorQueZero() throws Exception {
        Predicate<String> predicate = (s) -> s.length() > 0;

        boolean textoMaiorQueZero = predicate.test("Alexandre Gama");

        assertTrue(textoMaiorQueZero);
    }
}
```

Como você pode notar, passamos para a interface **Predicate<T>** uma expressão lambda que verifica se determinada **String** recebida por parâmetro é maior que zero. Este teste só foi possível porque a interface **Predicate<T>** é uma Functional Interface, ou seja, possui somente um método abstrato: **boolean test(T t)**.

Functions

A interface **Functions<T, R>** também possui um objetivo simples: dado um valor do tipo **T**, retornar um valor do tipo **R**, assim como em uma função matemática, utilizando o método **apply(T t)** para fazer essa transformação. Como você pode perceber, o método **apply()** funciona como um conversor, podendo ser usado para converter um valor do tipo **String** em um valor do tipo **Integer**, por exemplo.

Na **Listagem 5** é exibido um exemplo de uso da interface **Function<T, R>**.

Listagem 5. Conversão de uma String em um Integer utilizando a interface Function.

```
@Test  
public void converteUmaStringEmInteiro() throws Exception {  
    Function<String, Integer> converterParaInteger = Integer::valueOf;  
    Integer convertido = converterParaInteger.apply("10");  
    assertEquals(10, convertido, 0);  
}
```

Note que passamos para a interface **Function<T, R>** a expressão lambda **Integer::valueOf**. Isso só foi possível porque a interface **Function<T, R>** também é uma interface funcional e você pode verificar que a assinatura do método estático **valueOf()** possui a mesma assinatura do método **apply(T t)**.

Uma característica importante que a interface **Function<T, R>** apresenta é a possibilidade de encadear várias outras chamadas de funções, como no padrão Builder. A **Listagem 6** demonstra um exemplo de conversão de uma **String** em um **Integer** e em seguida converte o valor do inteiro em seu respectivo valor negativo, utilizando a chamada encadeada de outra função.

Listagem 6. Código de conversão de uma String em um inteiro negativo utilizando a interface Function.

```
@Test  
public void converteStringEmInteiroERetornarOValorNegativo() throws Exception {  
    Function<String, Integer> converterParaInteger = Integer::valueOf;  
    Function<String, Integer> converterParaNegativo = converterParaInteger  
        .andThen(Math::negateExact);  
  
    Integer convertido = converterParaNegativo.apply("10");  
  
    assertEquals(-10, convertido, 0);  
}
```

Observe que neste código também chamamos o método **andThen()**, presente na interface **Function**, que possibilita o encadeamento de mais chamadas a outros métodos desta interface.

Optional

A classe **Optional<T>** segue o padrão de projeto Null Object, que tem como objetivo fazer o tratamento de objetos que podem ser nulos, evitando o famoso **NullPointerException**. Como verificaremos, o uso dessa classe é simples: precisamos passar para o

Optional o objeto que gostaríamos de manipular. Com esse objeto em mãos, a classe **Optional** consegue indicar para o usuário se o objeto a ser manipulado é nulo ou não.

Na **Listagem 7** é exposto um exemplo simples, que cria um objeto nulo e depois o manipula com a classe **Optional**.

Listagem 6. Código de conversão de uma String em um inteiro negativo utilizando a interface Function.

```
@Test  
public void converteStringEmInteiroERetornarOValorNegativo() throws Exception {  
    Function<String, Integer> converterParaInteger = Integer::valueOf;  
    Function<String, Integer> converterParaNegativo = converterParaInteger  
        .andThen(Math::negateExact);  
  
    Integer convertido = converterParaNegativo.apply("10");  
  
    assertEquals(-10, convertido, 0);  
}
```

Listagem 7. Código que valida se o objeto do Optional é nulo.

```
@Test  
public void deveriaIndicarQueOValorDoObjetoENulo() throws Exception {  
    Pessoa pessoa = null;  
    Optional<Pessoa> optional = Optional.ofNullable(pessoa);  
    assertFalse(optional.isPresent());  
}
```

Observe que passamos como parâmetro para o método **ofNullable()** o objeto nulo **pessoa**. Feito isso, para verificar se existe um objeto não nulo, foi usado o método **isPresent()**. Caso o retorno deste seja **true**, o objeto **pessoa** não é nulo e poderemos retornar a própria instância de **pessoa** através do método **get()** da classe **Optional**, como mostra a **Listagem 8**.

Listagem 8. Código que indica que o objeto do Optional não é nulo.

```
@Test  
public void indicaQueOValorDoObjetoNaoENulo() throws Exception {  
    Pessoa pessoa = new Pessoa("Alexandre Gama");  
    Optional<Pessoa> optional = Optional.ofNullable(pessoa);  
  
    Pessoa alexandre = null;  
    if (optional.isPresent()) {  
        alexandre = optional.get();  
    }  
  
    assertEquals("Alexandre Gama", alexandre.getNome());  
}
```

Note também que se você tentar invocar o método **get()** e o objeto for nulo, uma exceção será lançada. O código da **Listagem 9** mostra esse caso.

É possível ainda informar ao **Optional** que, caso ele esteja guardando um objeto nulo, retorne um objeto qualquer de nossa escolha. Dessa forma evitamos o **if** com o método **isPresent()**, que utilizamos na **Listagem 8**. Na **Listagem 10** é apresentado um código com essa proposta, sem a chamada ao método **isPresent()**.

Note que chamamos o método **orElse()** passando como argumento a criação de um novo objeto **Pessoa**.

Listagem 9. Código que lança uma exceção ao tentar acessar um objeto que é nulo.

```
@Test(expected = NoSuchElementException.class)
public void lançaExecçãoAoTentarAcessarUmObjetoNulo() throws Exception {
    Pessoa alexandre = null;
    Optional<Pessoa> optional = Optional.ofNullable(alexandre);
    optional.get(); //A exceção ocorrerá nessa linha
}
```

Listagem 10. Código que retorna um novo objeto caso o Optional possua um objeto nulo.

```
@Test
public void retornaNovoObjetoAoAcessarObjetoNulo() throws Exception {
    Pessoa alexandre = null;
    Optional<Pessoa> optional = Optional.ofNullable(alexandre);
    Pessoa pessoa = optional.orElse(new Pessoa("Alexandre"));
    assertEquals("Alexandre", pessoa.getNome());
}
```

Este método indica ao **Optional** que ele deve retornar o novo objeto **pessoa** passado via parâmetro caso esteja guardando um objeto nulo, o que neste exemplo é uma afirmação verdadeira. Note ainda que desta forma a semântica do método fica mais simples, já que pedimos o objeto para o **Optional** e, “caso ele não tenha” – método **orElse()** – retorne um novo objeto. É comum que esta operação seja feita com um **if**, como mostrado na **Listagem 8**, mas a classe **Optional** facilita esse código adicionando o método **orElse()** tornando o **if** desnecessário.

Stream

A Stream API está diretamente relacionada à API de Collections e visa facilitar a manipulação dos objetos de uma determinada coleção. De modo simples, podemos pensar em uma stream como uma coleção de objetos que podem sofrer operações encadeadas e retornar resultados baseados nessas operações.

Uma stream permite que façamos dois tipos de operações sobre os dados de uma coleção: a primeira é chamada de intermediária, e como o próprio nome indica, é uma operação que é executada e gera como resultado outra stream, que pode ser utilizada para chamar uma nova operação intermediária ou terminal. Como exemplo, imagine que criamos uma stream a partir de uma coleção de Strings e depois filtramos todas as palavras dessa stream que começam com a letra “A”. Como essa operação cria uma nova stream, dizemos que ela é intermediária. Já o segundo tipo de operação, chamado de terminal, tem como objetivo colocar os dados resultantes dos processamentos realizados em uma coleção. O retorno dessa segunda operação pode ser uma coleção do tipo **List**, **Map** ou **Set** e encerra a stream.

Na **Listagem 11** apresentamos o código que cria uma coleção de nomes do tipo **String** e depois, através da API de Streams, filtra somente aqueles que iniciam com a letra A.

Note que utilizamos o método **stream()** na lista de nomes para indicar que queremos trabalhar com streams. Logo após, executamos a operação de filtro pelo método **filter()**. Caso você seja mais curioso, poderá notar que o método **filter()** recebe justamente um

Predicate, interface que já comentamos anteriormente. Como o método **startsWith()** da classe **String** possui a mesma assinatura do método **test(T t)** da interface **Predicate**, utilizamos uma expressão lambda para indicar que o filtro só deverá ser feito para os nomes iniciados com a letra A. Por fim, invocamos o método **collect()** e passamos a classe utilitária **Collectors** para indicar que o retorno do resultado deve ser um **List<T>**.

Listagem 11. Código que filtra os nomes com base na primeira letra.

```
@Test
public void filtraNomesPelaLetraA() throws Exception {
    List<String> nomes = Arrays.asList("Alexandre", "Bruna", "Fernando", "Augusto",
                                         "Sergio", "Paulo");

    List<String> nomesComLetraA = nomes
        .stream()
        .filter((s) -> s.startsWith("A"))
        .collect(Collectors.toList());

    assertEquals("Alexandre", nomesComLetraA.get(0));
    assertEquals("Augusto", nomesComLetraA.get(1));
}
```

Nos próximos exemplos vamos testar as seguintes operações em uma coleção:

- Ordenação de nomes;
- Conversão em maiúscula de todas as Strings de uma coleção.

Na **Listagem 12** temos o segundo exemplo de código que utiliza a API de Stream, mas dessa vez filtrando os nomes pela primeira letra (“A”) e em seguida ordenando-os de forma ascendente.

Listagem 12. Filtro e ordenação de um array com stream.

```
@Test
public void deveriaOrdenarOsNomesEfiltrarPelaPrimeiraLetra() throws Exception {
    List<String> nomes = Arrays.asList("Alexandre", "Adalberto", "Antonio", "Aecio",
                                         "Acacio", "Bruna");

    List<String> nomesComLetraA = nomes
        .stream()
        .filter((s) -> s.startsWith("A"))
        .sorted()
        .collect(Collectors.toList());

    System.out.println(nomesComLetraA);
    assertEquals("Acacio", nomesComLetraA.get(0));
    assertEquals("Adalberto", nomesComLetraA.get(1));
    assertEquals("Aecio", nomesComLetraA.get(2));
    assertEquals("Alexandre", nomesComLetraA.get(3));
    assertEquals("Antonio", nomesComLetraA.get(4));
}
```

Para este caso, conforme esperado, utilizamos novamente o método **filter()**. Como esse método é uma operação intermediária e retorna uma nova stream, contendo o resultado do filtro, podemos executar mais uma operação intermediária, neste caso o método **sorted()**, o que possibilitará a ordenação dos nomes da stream filtrada.

Money API: programação avançada utilizando os recursos do Java 8

Por fim, chamamos a operação terminal `collect()`, para transformar a stream resultante da execução de `sorted()` em uma coleção.

Vejamos agora a implementação do terceiro exemplo, que transformará todas as letras dos nomes de uma coleção em letras maiúsculas. Para resolver esse problema usaremos o método `map()`, que permite que façamos esse tipo de conversão em uma coleção inteira. Sem esse método, precisaríamos implementar um laço para percorrer todos os elementos da coleção e fazer a alteração em cada nome.

Outro caso de uso comum do método `map()` é a conversão de apenas alguns objetos de uma coleção. Como exemplo, imagine algo próximo ao caso anterior, mas que precisamos colocar em maiúsculo somente as Strings que começam com a letra A. Para resolver essa questão, aplicamos o filtro para obter os nomes que iniciam com a letra A e depois invocamos o método `map()` para converter esses nomes para maiúscula.

O código a seguir mostra a assinatura do método `map()`:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Observe que ele recebe como parâmetro a interface `Function<T, R>`, que foi estudada anteriormente e tem o objetivo de converter objetos. Outro detalhe importante na assinatura desse método é o seu retorno, que é do tipo `Stream`, permitindo assim invocar mais métodos na coleção.

Dito isso, vejamos um exemplo que mostra como realizar a conversão das letras de minúsculo para maiúsculo. No código da **Listagem 13** temos dois nomes na coleção (Alexandre e Gustavo) e vamos modificá-los para que fiquem em letra maiúscula. Para isso, basta chamarmos o método `map()` e passar como parâmetro a função `toUpperCase()` da classe `String`.

Listagem 13. Código que converte os nomes da coleção para que fiquem com todas as letras em maiúsculo.

```
@Test
public void transformaOsTextosEmMaiusculo() {
    List<String> nomes = Arrays.asList("Alexandre", "Gustavo");

    List<String> nomesEmMaiusculo = nomes.stream().map(String::toUpperCase).
        collect(Collectors.toList());

    assertEquals("ALEXANDRE", nomesEmMaiusculo.get(0));
    assertEquals("GUSTAVO", nomesEmMaiusculo.get(1));
}
```

Money API e os novos recursos do Java 8

Finalizada a revisão de algumas das principais APIs do Java 8, podemos concentrar o nosso foco na manipulação avançada da Money API. Para isso, serão apresentados diversos exemplos, como ordenação de valores, filtros avançados com moedas diferentes, agrupamento e geração de estatísticas e finalizaremos o estudo da API examinando o tão esperado recurso de conversão de moedas.

Adicionando a Money API ao projeto

Para adicionar a Money API ao nosso projeto podemos seguir dois caminhos: fazer o download do JAR manualmente ou usar um gerenciador de dependências, como Maven ou Gradle. Para o artigo, optamos pela solução que envolve o Maven. Em relação à IDE, adotaremos o Eclipse, mas os exemplos podem ser implementados em qualquer IDE, como NetBeans ou IntelliJ.

Na **Listagem 14** é apresentado o código que deve ser adicionado ao `pom.xml` para indicar ao Maven a dependência que deve ser baixada. Repare que referenciamos um projeto de nome Moneta. Isso porque o Moneta é a implementação de referência da Money API.

Listagem 14. Trecho do `pom.xml` para download da dependência da implementação Moneta.

```
<dependency>
    <groupId>org.javamoney</groupId>
    <artifactId>moneta</artifactId>
    <version>1.0</version>
</dependency>
```

Alterado esse arquivo, execute o comando `mvn eclipse:eclipse` no terminal para que o download seja realizado. Logo após, volte ao Eclipse e atualize o projeto para que o JAR da implementação da Money API seja exibido na árvore de dependências.

Ordenação de valores monetários

Iniciaremos a nossa série de exemplos criando uma coleção de preços em real e em seguida a ordenaremos de forma ascendente, utilizando para tanto a Money API e os recursos de streams do Java 8. Assim, crie a classe `OrdenacaoDePrecosTest` com o método `ordenaPrecosDeFormaAscendente()` conforme apresenta a **Listagem 15**.

Listagem 15. Ordenação de preços utilizando a Money API.

```
public class OrdenacaoDePrecosTest {

    @Test
    public void ordenaPrecosDeFormaAscendente() throws Exception {
        MonetaryAmount trintaReais = Money.of(30, real);
        MonetaryAmount vinteReais = Money.of(20, real);
        MonetaryAmount cincoReais = Money.of(5, real);
        MonetaryAmount dezReais = Money.of(10, real);

        List<MonetaryAmount> precos = Arrays.asList(trintaReais, vinteReais,
            cincoReais, dezReais);

        List<MonetaryAmount> precosOrdenados = precos
            .stream()
            .sorted(MonetaryFunctions.sortNumber())
            .collect(Collectors.toList());

        assertEquals(cincoReais, precosOrdenados.get(0));
        assertEquals(dezReais, precosOrdenados.get(1));
        assertEquals(vinteReais, precosOrdenados.get(2));
        assertEquals(trintaReais, precosOrdenados.get(3));
    }
}
```

Observe que criamos uma lista de preços usando a interface **MonetaryAmount** – uma das principais interfaces da Money API – e em seguida realizamos a ordenação dos preços através do método **sorted()**, invocado após a chamada ao método **stream()**.

O método **sorted()** recebe um **Comparator** como parâmetro que deve indicar a forma de comparação para realizar a ordenação. A novidade desse código está no uso da classe utilitária **MonetaryFunctions** – da Money API – que contém diversos métodos auxiliares que serão analisados a seguir. Para este exemplo usamos o método auxiliar **sortNumber()**, que efetua a ordenação por valor.

Como os métodos da interface **Collection** são imutáveis, sua invocação não altera o estado do objeto em questão (neste caso, a lista de preços) e desta forma precisamos invocar o método **collect()** passando como parâmetro o método **toList()** da interface **Collectors**. O resultado será uma nova lista do tipo **MonetaryAmount** já com os preços devidamente ordenados.

Avançando um pouco mais no nosso estudo, vamos demonstrar agora como ordenar uma lista de preços contendo valores em Real e em Dólar. Para essa ordenação, vamos primeiro agrupar os valores em suas respectivas moedas e então ordenar os preços pelo valor. Para facilitar a compreensão, imagine a seguinte lista de preços: R\$ 10, \$ 8, R\$ 4, \$ 5, R\$ 40, \$ 3. O resultado esperado deve ser o seguinte: R\$ 4, R\$ 10, R\$ 40, \$ 3, \$ 5, \$ 8.

Na **Listagem 16** temos o código completo da resolução desse problema.

Para esse exemplo, note que usamos o **sorted()** duas vezes. Na primeira, passamos como parâmetro o método **sortNumber()** – da classe utilitária **MonetaryFunctions** – cuja finalidade é ordenar a stream pelos valores. Feito isso, chamamos o **sorted()** mais uma vez, mas agora passando como parâmetro o método **sortCurrencyUnit()**, responsável pelo agrupamento dos valores em suas respectivas moedas.

Listagem 16. Agrupamento de moedas e ordenação de preços.

```
@Test
public void ordenaPrecosDeMoedasDiferentesDeFormaAscendente() throws
Exception {
    MonetaryAmount quatroReais = Money.of(4, real);
    MonetaryAmount oitoDolares = Money.of(8, dollar);
    MonetaryAmount dezReais = Money.of(10, real);
    MonetaryAmount cincoDolares = Money.of(5, dollar);
    MonetaryAmount quarentaReais = Money.of(40, real);
    MonetaryAmount tresDolares = Money.of(3, dollar);

    List<MonetaryAmount> precos = Arrays.asList(quatroReais, oitoDolares,
dezReais, cincoDolares, quarentaReais, tresDolares);
    List<MonetaryAmount> precosOrdenados = precos.stream();
    sorted(sortNumber()).sorted(sortCurrencyUnit()).collect(toList());

    assertEquals(tresReais, precosOrdenados.get(0));
    assertEquals(dezReais, precosOrdenados.get(1));
    assertEquals(quarentaReais, precosOrdenados.get(2));

    assertEquals(tresDolares, precosOrdenados.get(3));
    assertEquals(cincoDolares, precosOrdenados.get(4));
    assertEquals(oitoDolares, precosOrdenados.get(5));
}
```

Perceba que, no resultado apresentado, os valores foram ordenados do menor para o maior, ou seja, de forma ascendente. Caso seja necessário, também é possível fazer a ordenação dos valores de forma descendente. Para isso, deve-se utilizar os métodos **sortCurrencyUnitDesc()** e **sortNumberDesc()**.

Nesse momento vale destacar uma questão: como a API realiza a ordenação das moedas? Simples, utilizando os nomes destas. No caso, para Dólar e Real, os nomes utilizados são USD e BRL. Sendo assim, no resultado da **Listagem 16**, o Dólar veio depois do Real. Se tivéssemos invocado o método **sortCurrencyUnit()** antes de **sortNumber()**, o resultado seria \$ 3, R\$ 4, \$ 5, \$ 8, R\$ 10, R\$ 40, que indica que a ordenação usando os nomes das moedas é ignorada antes da ordenação de valores.

Filtros e reduções utilizando stream

Quando temos uma lista de preços, é comum a necessidade de realizar operações de filtro nessa lista como, por exemplo:

- Encontrar os preços mínimo e máximo;
- Criar uma lista com preços acima de um valor;
- Criar uma lista com os preços entre dois valores;
- Criar uma lista com preços acima ou igual a um valor e que não mostre preços duplicados.

Diante disso, vamos demonstrar como utilizar a Money API e a API de Streams para criar soluções que resolvam cada uma destas situações.

Obtendo os preços mínimo e máximo

Com o intuito de solucionar o primeiro exemplo, a **Listagem 17** apresenta o código da classe **FiltrosDePrecosTest** e um método para identificar os preços mínimo e máximo de uma lista.

Para filtrarmos pelo preço mínimo, invocamos o método **min()** da classe **MonetaryFunctions**. Em seguida, reduzimos o resultado através do método **reduce()**. Esta chamada ao método **reduce()** é necessária porque, como o próprio nome diz, reduzirá os valores

Listagem 17. Filtro para obter os preços mínimo e máximo de uma lista.

```
public class FiltrosDePrecosTest {

    @Test
    public void filtraPeloPrecoMinimoEMaximoDeUmaListaDePrecos() throws
Exception {
        MonetaryAmount vinteReais = Money.of(20, real);
        MonetaryAmount dezReais = Money.of(10, real);
        MonetaryAmount noventaReais = Money.of(90, real);
        MonetaryAmount cincoReais = Money.of(5, real);

        List<MonetaryAmount> precos = Arrays.asList(vinteReais, dezReais,
noventaReais, cincoReais);

        MonetaryAmount precoMinimo = precos.stream().reduce(min()).get();
        MonetaryAmount precoMaximo = precos.stream().reduce(max()).get();

        assertEquals(cincoReais, precoMinimo);
        assertEquals(noventaReais, precoMaximo);
    }
}
```

Money API: programação avançada utilizando os recursos do Java 8

da coleção a apenas um valor: o menor preço da lista. Note como ficou fácil usar métodos utilitários da Money API. Neste exemplo, caso não estivéssemos trabalhando com essa API, precisaríamos varrer a lista de preços e comparar um a um para identificar o preço mínimo. Para retornar o valor máximo da coleção, basta utilizar o método `max()`.

Um detalhe importante nesse código é que o método `reduce()` não retorna o valor propriamente dito e sim um valor encapsulado na classe **Optional**. E como vimos anteriormente, é preciso invocar o `get()` dessa classe para obter o valor encapsulado. Assim, ao final da chamada a `reduce()` invocamos o método `get()` para ter acesso ao preço mínimo.

Obtendo uma lista com preços acima de um valor

Agora vamos escrever um método que será responsável por criar uma nova lista, a partir de uma lista de preços, levando em consideração apenas os valores que são maiores que 30 reais. Para isso, a seguinte lista será usada: R\$ 20, R\$ 10, R\$ 90, R\$ 5, R\$ 40. A Listagem 18 mostra o código completo.

Listagem 18. Filtro e criação de nova lista a partir de um preço.

```
@Test
public void filtraECriaSublistaAPartirDe30Reais() throws Exception {
    MonetaryAmount vinteReais = Money.of(20, real);
    MonetaryAmount dezReais = Money.of(10, real);
    MonetaryAmount noventaReais = Money.of(90, real);
    MonetaryAmount cincoReais = Money.of(5, real);
    MonetaryAmount quarentaReais = Money.of(40, real);

    List<MonetaryAmount> precos = Arrays.asList(vinteReais, dezReais,
        noventaReais, cincoReais, quarentaReais);

    List<MonetaryAmount> novosPrecos = precos
        .stream()
        .filter(isGreaterThanOrEqualTo(of(30, real)))
        .collect(Collectors.toList());

    assertEquals(2, novosPrecos.size());
    assertEquals(quarentaReais, novosPrecos.get(0));
    assertEquals(noventaReais, novosPrecos.get(1));
}
```

Neste caso, note que usamos o método `filter()` passando como parâmetro outro método, o `isGreaterThanOrEqualTo()`, da classe **MonetaryFunctions**, que recebe como parâmetro o valor R\$ 30,00. Deste modo, o resultado esperado é: R\$ 90, R\$ 40.

Obtendo uma lista com preços entre dois valores

Para solucionar esse problema, vamos utilizar um novo filtro na coleção inicial. Também da classe **MonetaryAmount**, o método `isBetween()` possibilita selecionar os preços da lista que estão entre os valores mínimo e máximo especificados. No caso do exemplo, 10 e 50 reais, que devem ser declarados como sendo do tipo **MonetaryAmount**. Veja o código na Listagem 19.

Listagem 19. Filtro e criação de nova lista de preços que contenha valores entre 10 e 50 reais.

```
@Test
public void filtraValoresEntre2Precos() throws Exception {
    MonetaryAmount vinteReais = Money.of(20, real);
    MonetaryAmount dezReais = Money.of(10, real);
    MonetaryAmount noventaReais = Money.of(90, real);
    MonetaryAmount cincoReais = Money.of(5, real);

    List<MonetaryAmount> precos = Arrays.asList(vinteReais, dezReais,
        noventaReais, cincoReais);

    List<MonetaryAmount> novosPrecos = precos
        .stream()
        .filter(isBetween(of(10, real), of(50, real)))
        .collect(Collectors.toList());

    assertEquals(2, novosPrecos.size());
    assertEquals(vinteReais, novosPrecos.get(0));
    assertEquals(dezReais, novosPrecos.get(1));
}
```

Obtendo uma lista com preços em Real, ordenada, acima ou igual a um valor e que não tenha valores duplicados

Implementaremos agora o quarto e último exemplo com filtros. Para isso, definiremos uma lista com dois tipos de moeda, Real e Dólar, e que terá diversos valores, sendo alguns duplicados. A partir desse conjunto de dados, o resultado deverá ser uma coleção que contenha:

- Somente valores em Real;
- Somente um valor igual a R\$ 5 e valores acima de R\$ 5;
- Somente valores únicos, não duplicados;
- Valores ordenados de forma ascendente.

Para testar nosso código, a seguinte lista de preços será usada: R\$ 20, R\$ 10, R\$ 10, R\$ 90, R\$ 5, R\$ 5, \$ 60, \$ 15, \$ 15 e o resultado esperado é: R\$ 5, R\$ 10, R\$ 20, R\$ 90. Na Listagem 20 apresentamos o código que soluciona o problema.

Como você pode observar, o código é semelhante aos anteriores, mas foram usados dois novos métodos: o `isGreaterThanOrEqualTo()` e o `distinct()`. O primeiro, como o próprio nome indica, é responsável por verificar se o valor da coleção é maior ou igual ao valor passado como parâmetro. Já o segundo método tem o objetivo de retornar um novo stream que não contenha valores duplicados.

Agrupamento e estatística de valores

Outra funcionalidade interessante que podemos utilizar em nossas aplicações é o agrupamento de valores a partir de determinados filtros. Por exemplo: imagine que nossa coleção seja formada por diferentes valores nas moedas Real e Dólar e precisamos agrupar todos os valores da moeda Real em uma coleção e todos os valores em Dólar em outra. Este requisito é bastante comum em telas que apresentam o resumo de valores ou em relatórios.

A Money API permite esse agrupamento através de mais um método utilitário: o **groupByCurrencyUnit()**.

Para demonstrar essa opção, dada uma lista de preços com duas moedas diferentes, vamos implementar um agrupamento por cada moeda, gerando duas novas listas, como mostra a **Listagem 21**.

Listagem 20. Filtro e obtenção de nova coleção ordenada, sem valores duplicados e que apresenta somente os preços em Real.

```
@Test
public void filtraPorRealRemovendoPrecosDuplicadosEOrdenando() throws
Exception {
    MonetaryAmount vinteReais = Money.of(20, real);
    MonetaryAmount dezReais = Money.of(10, real);
    MonetaryAmount dezReaisDuplicado = Money.of(10, real);
    MonetaryAmount noventaReais = Money.of(90, real);
    MonetaryAmount cincoReais = Money.of(5, real);
    MonetaryAmount cincoReaisDuplicado = Money.of(5, real);
    MonetaryAmount sessentaDolares = Money.of(60, dolar);
    MonetaryAmount quinzeDolares = Money.of(15, dolar);
    MonetaryAmount quinzeDolaresDuplicado = Money.of(15, dolar);

    List<MonetaryAmount> precos = Arrays.asList(vinteReais, dezReais,
    noventaReais, cincoReais, cincoReaisDuplicado, dezReaisDuplicado,
    sessentaDolares, quinzeDolares, quinzeDolaresDuplicado);

    List<MonetaryAmount> novosPrecos = precos
        .stream()
        .filter(isCurrency(real))
        .filter(isGreaterThanOrEqualTo(Money.of(5, real)))
        .sorted(sortNumber())
        .distinct()
        .collect(toList());

    assertEquals(4, novosPrecos.size());
    assertEquals(cincoReais, novosPrecos.get(0));
    assertEquals(dezReais, novosPrecos.get(1));
    assertEquals(vinteReais, novosPrecos.get(2));
    assertEquals(noventaReais, novosPrecos.get(3));
}
```

Listagem 21. Agrupamento de preços por moeda.

```
@Test
public void agrupaOsPrecosPorMoeda() throws Exception {
    MonetaryAmount dezReais = Money.of(10, real);
    MonetaryAmount vinteReais = Money.of(20, real);
    MonetaryAmount trintaReais = Money.of(30, real);
    MonetaryAmount cincoDolares = Money.of(5, dolar);
    MonetaryAmount quinzeDolares = Money.of(15, dolar);

    List<MonetaryAmount> precos = Arrays.asList(dezReais, cincoDolares,
    vinteReais, trintaReais, quinzeDolares);

    Map<CurrencyUnit, List<MonetaryAmount>> precosAgrupados = precos
        .stream()
        .collect(groupByCurrencyUnit());

    assertEquals(3, precosAgrupados.get(real).size());
    assertEquals(2, precosAgrupados.get(dolar).size());

    assertEquals(dezReais, precosAgrupados.get(real).get(0));
    assertEquals(vinteReais, precosAgrupados.get(real).get(1));
    assertEquals(trintaReais, precosAgrupados.get(real).get(2));

    assertEquals(cincoDolares, precosAgrupados.get(dolar).get(0));
    assertEquals(quinzeDolares, precosAgrupados.get(dolar).get(1));
}
```

Nesse código usamos novamente o **collect()**, mas dessa vez passando como parâmetro o método **groupByCurrencyUnit()** da classe **MonetaryFunctions**. O interessante a ser observado é que o retorno do método **collect()** é um mapa, do tipo **Map<CurrencyUnit, List<MonetaryAmount>>**. Isto é necessário porque na lista temos dois tipos de moeda, Real e Dólar, e para cada moeda precisamos de uma lista com os seus respectivos valores.

Outra funcionalidade muito útil da Money API é a possibilidade de obtermos estatísticas de forma simples e rápida a partir de uma coleção de valores. Para expor esse recurso: imagine que, dada uma lista de preços, precisamos dos seguintes dados:

1. Preço mínimo da lista por moeda;
2. Preço máximo da lista por moeda;
3. Soma total dos preços por moeda;
4. Quantidade de preços na lista por moeda;
5. Média aritmética dos valores da lista por moeda.

Conforme já demonstrado no artigo, podemos usar os métodos utilitários de **MonetaryFunctions** para retornar cada valor supracitado. No entanto, apesar da facilidade de fazer o agrupamento usando esses métodos, temos também a classe **GroupMonetarySummaryStatistics**, que fornece opções que sumarizam os valores de uma coleção e retornam todos os valores agrupados conforme a lista dos cinco dados supracitados.

Na **Listagem 22** é apresentado o código que resolve o problema proposto.

Listagem 22. Agrupamento de valores e geração de estatísticas.

```
@Test
public void agrupaMostrandoAEstatisticaDaColecao() throws Exception {
    CurrencyUnit real = Monetary.getCurrency("BRL");
    CurrencyUnit dolar = Monetary.getCurrency("USD");

    MonetaryAmount dezReais = Money.of(10, real);
    MonetaryAmount vinteReais = Money.of(20, real);
    MonetaryAmount trintaReais = Money.of(30, real);
    MonetaryAmount cincoDolares = Money.of(5, dolar);
    MonetaryAmount quinzeDolares = Money.of(15, dolar);

    List<MonetaryAmount> precos = Arrays.asList(dezReais, cincoDolares,
    vinteReais, trintaReais, quinzeDolares);

    GroupMonetarySummaryStatistics estatistica = precos
        .stream()
        .collect(groupBySummarizingMonetary());

    Map<CurrencyUnit, MonetarySummaryStatistics> map = estatistica.get();

    MonetarySummaryStatistics estatisticasEmReal = map.get(real);
    MonetarySummaryStatistics estatisticasEmDolar = map.get(dolar);

    assertEquals(3, estatisticasEmReal.getCount());
    assertEquals(2, estatisticasEmDolar.getCount());
    assertEquals(dezReais, estatisticasEmReal.getMin());
    assertEquals(trintaReais, estatisticasEmReal.getMax());

    assertEquals(Money.of(20, real), estatisticasEmReal.getAverage());
    assertEquals(Money.of(60, real), estatisticasEmReal.getSum());
}
```

Note que invocamos o método `collect()` passando como parâmetro o método utilitário `groupBySummarizingMonetary()`. O retorno dessa chamada é um objeto do tipo `GroupMonetarySummaryStatistics` que é responsável por agrupar em um mapa os valores mínimo, máximo, total, média e soma. A esse conjunto de dados agrupados é dado o nome de estatísticas, o que caracteriza o sufixo `Statistics` no final do nome da classe.

Como você pode observar, os tipos do mapa retornado pelo método `get()` da classe `GroupMonetarySummaryStatistics` são `CurrencyUnit` e `MonetarySummaryStatistics`. E como o tipo `CurrencyUnit` representa a moeda da coleção e é a chave do mapa, podemos retornar, para cada moeda, um objeto do tipo `MonetarySummaryStatistics` que contém as estatísticas relacionadas à aquela moeda.

No exemplo da [Listagem 22](#), a primeira invocação ao método `get()` foi para obter as estatísticas dos preços em Real e a segunda invocação, para obter as estatísticas dos preços em Dólar. Note que cada chamada ao método `get()` retornou um objeto do tipo `MonetarySummaryStatistics`, que contém os seguintes métodos: `getCount()`, `getMin()`, `getMax()`, `getAverage()` e `getSum()`. Ao final dessa listagem simplesmente invocamos esses métodos.

Conversão de moedas

Com os recursos atuais, você já parou para pensar em como poderia solucionar um problema de conversão de moedas (de dólar para real, por exemplo)? Pensando em facilitar a implementação desse requisito, a Money API nos trouxe um excelente recurso.

Atualmente, independentemente da Money API, dois órgãos fornecem informações sobre a cotação atual de cada moeda no mundo. Esses órgãos, também chamados de providers, são o ECB (*European Central Bank*) e o IMF (*International Monetary Fund*) – veja a seção [Links](#).

A Money API, para viabilizar a conversão, acessa um desses providers para obter o valor da cotação a ser utilizada. Esse acesso se dá pela interface `ExchangeRateProvider`, que possui cinco formas diferentes de trabalhar com os providers supracitados:

1. **ECB** – Realiza a conversão de acordo com o European Central Bank;
2. **IMF** – Realiza a conversão de acordo com as informações mais recentes do International Monetary Fund;
3. **IMF_HIST** – Realiza a conversão de acordo com uma data específica no IMF, a data de hoje ou uma data do passado;
4. **ECB_HIST90** – Realiza a conversão de acordo com os últimos 90 dias no ECB;
5. **ECB_HIST** – Realiza a conversão de acordo com uma data específica no ECB, que seja a partir de 1999.

Com o intuito de demonstrar esse recurso, criaremos a classe `ConversaoDeMoedaTest` com um método para converter 1 dólar em real (vide [Listagem 23](#)).

Neste exemplo, através da interface `ExchangeRateProvider` selecionamos o provider que vamos utilizar; neste caso, o ECB. Feito isso, indicamos ao próprio provider qual será a moeda

de conversão, que no nosso caso é o Real, e especificamos, em seguida, o valor a ser convertido, que será 1 Dólar, através da classe `Money`. Após definir a moeda e o valor, usamos o provider para obter a interface `CurrencyConversion`, responsável pela conversão de uma moeda. Para finalizar, invocamos o método `apply()` do conversor para fazer a conversão do nosso valor de 1 Dólar e retornar um `MonetaryAmount` já com o valor convertido para o Real.

Listagem 23. Conversão de 1 dólar em real.

```
public class ConversaoDeMoedaTest {  
    @Test  
    public void converteUmDolarEmReal() throws Exception {  
        CurrencyUnit real = Monetary.getCurrency(new Locale("pt", "BR"));  
        CurrencyUnit dolar = Monetary.getCurrency(new Locale("en", "US"));  
  
        ExchangeRateProvider provider = MonetaryConversions.  
            getExchangeRateProvider("ECB");  
  
        MonetaryAmount umDolar = Money.of(1, dolar);  
  
        CurrencyConversion conversor = provider.getCurrencyConversion(real);  
        MonetaryAmount dolarParaReal = conversor.apply(umDolar);  
  
        System.out.println(dolarParaReal);  
    }  
}
```

Internamente, a implementação da Money API possui o endereço externo dos providers. Assim, para acessar a cotação pelo provider ECB, por exemplo, a API utiliza o endereço do ECB indicado na seção [Links](#).

Neste momento o leitor pode estar se perguntando: o que acontecerá se a aplicação não tiver acesso à internet no momento de fazer a conversão, já que o acesso à conversão é externo? A Money API lida com essa situação de uma forma simples. A cada chamada ao método de conversão a API cria um arquivo de cache para cada provider e caso esse arquivo já tenha sido criado, será atualizado com a informação mais atual. Em suma, quando o método de conversão é invocado são criados três arquivos (`IMFRateProvider.dat`, `ECBCurrentRateProvider.dat` e `ECBHistoric90RateProvider.dat`) no diretório oculto `resourceCache` para serem utilizados como cache.

Nota

Ao executar o método de conversão da [Listagem 23](#), mesmo que você tenha optado pelo provider ECB, a API também criará o arquivo relacionado ao provider IMF.

Na [Listagem 24](#) é apresentado um pequeno trecho do conteúdo do arquivo `ECBCurrentRateProvider.dat`.

Observe que o arquivo é de fácil leitura e compreensão. Cada tag `Cube` com o atributo `currency` indica a moeda e seu respectivo valor naquele dia. Note ainda que a data da cotação é sinalizada pelo atributo `time`. Neste exemplo, temos o valor 2016-01-05, dia em que foi feita a invocação ao método de conversão.

Listagem 24. XML com os valores do provider ECB para conversão.

```
<gesmes:Envelope xmlns:gesmes="http://www.gesmes.org/xml/2002-08-01"
  xmlns="http://www.ecb.int/vocabulary/2002-08-01/eurofxref">
  <gesmes:subject>Reference rates</gesmes:subject>
  <gesmes:Sender>
    <gesmes:name>European Central Bank</gesmes:name>
  </gesmes:Sender>
  <Cube>
    <Cube time='2016-01-05'>
      <Cube currency='USD' rate='1.0746'/>
      <Cube currency='JPY' rate='127.88'/>
      <Cube currency='BGN' rate='1.9558'/>
      <Cube currency='CZK' rate='27.022'/>
      <Cube currency='DKK' rate='7.4605'/>
      <Cube currency='GBP' rate='0.73235'/>
      .
      .
    </Cube>
  </Cube>
</gesmes:Envelope>
```

Agora, voltando à pergunta inicial... e se a internet estiver indisponível no momento da invocação da conversão? Neste caso o sistema utilizará o último arquivo de cache criado, ou seja, caso dia 06 você esteja sem internet, o arquivo de cache do dia 05 será reutilizado.

Nesse momento, atente para um detalhe: se for a sua primeira chamada à API de conversão, você não terá disponível nenhum arquivo de cache. Nessa situação, se a internet estiver indisponível, uma **Exception** será lançada, conforme o código da **Listagem 25**.

Listagem 25. Exception lançada no momento da conversão com internet indisponível e sem arquivo de cache.

```
Jan 06, 2016 8:41:45 AM org.javamoney.moneta.internal.loader.LoadableResource
load
INFO: Failed to load resource input for ECBHistoricRateProvider from http://www.
ecb.europa.eu/stats/eurofxref/eurofxref-hist.xml
java.net.UnknownHostException: www.ecb.europa.eu
...
...
```

Essa **Exception** indica que o host não foi encontrado, mas analisando o código mais a fundo você notará que a API simplesmente não conseguiu acessar a internet e, como não existe arquivo de cache, a exceção é lançada.

Apesar de todas as opções que foram mostradas, novos recursos da Money API continuam sendo implementados a todo vapor e o atraso no lançamento do Java 9 provavelmente possibilitará a adição de novas funcionalidades e melhorias, permitindo que a API se torne ainda mais madura, flexível e profissional. Por fim, lembre-se que não é necessário esperar o lançamento do Java 9 para ter acesso a todos esses recursos. Para isso, basta usar o Java 8 e adicionar o Moneta como dependência do seu projeto

Autor**Alexandre Gama**alexandre.gama.lima@gmail.com

Trabalha como Engenheiro de Software no Elo7 e Instrutor na Caelum. Além de cometer em alguns projetos Open Source, é palestrante em eventos nacionais como o TDC. É apaixonado por integração entre sistemas e está se aventurando no mundo DevOps.

**Links:****Documentação da API de Streams do Java.**<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>**Página da JSR 354.**<https://www.jcp.org/en/jsr/detail?id=354>**Página do projeto Moneta, RI da especificação.**<http://javamoney.github.io/ri.html>**eBook em português de Otávio Santana, um dos responsáveis pela Money API.**<http://otaviojava.gitbooks.io/money-api/>**Site do projeto Java 9.**<http://openjdk.java.net/projects/jdk9/>**Site do provider ECB.**<http://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml>**Site do provider IMF.**http://www.imf.org/external/np/fin/data/rms_five.aspx?tsvflag=Y**Você gostou deste artigo?**

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



MVC baseado em ações no Java EE

Saiba como será o MVC 1.0, a nova JSR que visa integrar o Java EE ao velho e conhecido padrão MVC baseado em ações

Muitos dos programadores da atualidade não vivenciaram o início do desenvolvimento para a web, há quase 20 anos, quando linguagens como ASP (*Active Server Pages*) e PHP (*Hypertext Preprocessor*) dominavam a preferência dos desenvolvedores da época. Nesse tempo, quando os desenvolvedores ainda estavam engatinhando nesse novo paradigma de desenvolvimento e poucos padrões eram utilizados, era muito comum encontrar o que convencionamos chamar atualmente de código-espaguete: um código no qual misturávamos em um único arquivo a lógica de apresentação e negócio, além de diversas linguagens, como HTML, JavaScript, ASP/PHP/JSP e CSS.

Quem já teve o desprazer de dar manutenção em um código-espaguete sabe o quanto essa tarefa é e cansativa e desestimulante, uma vez que esse tipo de código é extremamente complexo e dificulta a aplicação de conceitos básicos de reaproveitamento de código. Como resposta a esse problema, alguns frameworks foram criados visando um código mais limpo, reaproveitável e de fácil manutenção. Basicamente, eles usavam padrões que orientavam o desenvolvedor a criar sua aplicação em camadas lógicas que separavam o código de negócio e persistência de dados do código de apresentação, também conhecido como Visão.

Entre os muitos padrões utilizados por esses frameworks, o MVC, sigla para Modelo-Visão-Controle, logo se tornou bastante badalado e, desde então, virou o preferido da grande maioria dos frameworks para a web. Entretanto, uma rápida pesquisa na Internet nos revela uma boa quantidade de variações, ou reinterpretações, desse padrão, sendo que atualmente as duas mais usadas no mundo Java são o MVC baseado em componentes e o MVC baseado em ações.

Assim que o Java entrou no mundo do desenvolvimento web, também era comum encontrarmos diversos

Fique por dentro

O MVC 1.0 chega ao mundo Java para trazer mais um padrão de desenvolvimento às aplicações web. Embora esse padrão já venha sendo utilizado há anos por frameworks como Spring MVC e VRaptor, até o momento não estava integrado ao arcabouço do Java EE. Entretanto, a partir do novo Java Enterprise Edition, que será lançado até o final deste ano, esse problema será resolvido, e será possível desenvolver aplicações usando o padrão MVC baseado em ações em conjunto com recursos como o CDI e Bean Validation. Portanto, entender essa JSR antes mesmo de seu lançamento, aprendendo seus recursos e como eles impactarão em seus futuros projetos, é importante para os desenvolvedores Java, pois os tornarão aptos a escolher com sólido embasamento a melhor arquitetura para seus próximos projetos.

códigos-espaguete, predominantemente feitos com a tecnologia JSP (*JavaServer Pages*). No entanto, a comunidade Java respondeu muito rapidamente a esse problema, criando frameworks como o Struts, Tapestry, JSF e muitos outros. Desde então, esse padrão é amplamente utilizado para a criação de aplicações web em Java, tanto que agora ganhará uma JSR específica para ele, chamada de MVC 1.0.

Neste artigo, analisaremos essa nova JSR através de um exemplo prático usando a implementação de referência, chamada Ozark. Trata-se de uma aplicação simples de cadastro de produtos, cujo objetivo é detalhar essa JSR e como ela poderá ser utilizada em seus futuros projetos.

O Padrão MVC e suas variações

O padrão MVC surgiu na década de 1970 e foi criado por Trygve Reenskaug para a linguagem Smalltalk. Inicialmente, ele foi criado para ser usado no desenvolvimento de aplicações para desktop, mas, posteriormente, com o advento da web, passou a ser adotado em frameworks escritos em praticamente todas as linguagens disponíveis para essa plataforma. Dada sua importância no

desenvolvimento de softwares, Martin Fowler escreveu: “o MVC foi a primeira tentativa de padronizar a construção de aplicações com interfaces gráficas”.

O objetivo desse padrão, desde a sua concepção, é separar a aplicação em camadas lógicas, deixando clara a responsabilidade de cada camada para evitar, por exemplo, que um código que é responsável pelo acesso ao banco de dados esteja misturado com um código cuja responsabilidade é exibir uma caixa de texto na tela.

Basicamente, temos três camadas nesse padrão, chamadas de Modelo, Visão e Controlador. A camada de modelo é responsável pela lógica do negócio e persistência dos dados da aplicação. No MVC, o modelo deve ser totalmente independente da visão, podendo ser reutilizado com qualquer outra tecnologia de apresentação. A visão, por sua vez, é responsável apenas por apresentar a tela da aplicação, como caixas de texto, listas de seleção, etc. Por último, temos o controlador, responsável por responder a eventos originados da interação do usuário com os componentes da tela e funciona como um intermediário entre o Modelo e a Visão, já que esses dois nunca podem interagir diretamente.

Contudo, o padrão MVC não se manteve estático durante todos esses anos. Diversas variações foram criadas, cada uma tentando resolver novos problemas que surgiam em decorrência da própria evolução das tecnologias. Entretanto, embora variem em alguns aspectos, todos eles se enquadram no padrão que Martin Fowler nomeou como *Separated Presentation*. Esse padrão é a base de todos eles e tem como principal foco a ideia de que a lógica de negócio deve sempre ser representada por uma camada totalmente independente da camada de apresentação, conforme já discutimos anteriormente.

Entre os muitos padrões que estendem o MVC, temos: MVP (Model-View-Presenter), MVA (Model-View-Adapter), AM-MVP (Application Model – Model-View-Controller), etc. Mais recentemente, duas novas opções de MVC, totalmente direcionados ao ambiente web, foram sugeridos pela comunidade: o MVC baseado em Ações e o MVC baseado em Componentes.

MVC baseado em componentes

Nesse padrão, temos uma interface gráfica totalmente construída através da composição de componentes e por um arcabouço que é responsável por esconder do desenvolvedor toda a lógica envolvida no tratamento do fluxo de requisições e respostas do protocolo HTTP. Por exemplo, o programador não precisa se preocupar em guardar o estado da aplicação entre múltiplas requisições ou, em tese, sequer precisa conhecer JavaScript para ter à sua disposição as chamadas AJAX, tão comuns no desenvolvimento web da atualidade.

Entre algumas das responsabilidades de um framework baseado nesse padrão, estão:

- Tornar transparente ao usuário a construção de URLs;
- Guardar o estado da aplicação entre múltiplas chamadas;
- Isolar o desenvolvedor das complexidades envolvidas no tratamento de requisições HTTP;

- Prover componentes gráficos que abstraiam a construção de elementos de tela.

Uma das principais vantagens dos frameworks baseados nesse paradigma está na facilidade que a composição de componentes fornece, como a reutilização de código e a facilidade de manutenção. Argumenta-se, entretanto, que a curva de aprendizado desses frameworks é maior, apesar de que, uma vez dominados, eles tornem a aplicação mais fácil de ser mantida.

Outra facilidade trazida pelos componentes está no fato de que é possível que o desenvolvedor crie uma aplicação web inteira sem sequer conhecer HTML, JavaScript e CSS, visto que eles encapsulam a lógica de exibição e tratamento de eventos de elementos gráficos, como uma caixa de texto ou lista de seleção.

Contudo, esse padrão não está imune a críticas. Um dos maiores críticos ao JSF e, consequentemente, desse padrão, é o Martin Fowler, que teceu duras críticas ao JSF em um dos boletins técnicos da ThoughtWorks, renomada empresa do ramo e da qual ele é um dos donos. Entre muitas das suas críticas, está o fato de que esse padrão tenta criar aplicações com estado persistente através de um protocolo, no caso o HTTP, que não foi criado com esse intuito, o que acaba causando uma série de problemas envolvendo o estado compartilhado no lado servidor. Para ter acesso ao boletim contendo essas críticas, e ter uma melhor compreensão do teor dessas críticas, veja a seção **Links**.

MVC baseado em ações

Muito diferente do MVC baseado em componentes, esse padrão não tem como intuito tentar esconder do programador as peculiaridades do protocolo HTTP e, muito menos, tentar afastá-lo das linguagens JavaScript, HTML e CSS, usadas para a apresentação de elementos gráficos no navegador. Portanto, nesse padrão não temos componentes de tela, como caixas de texto e grids, que nos permitem criar uma tela inteira sem sequer conhecer HTML.

Um framework baseado nesse padrão apenas redireciona as requisições HTTP para um controlador específico, que é selecionado a partir de informações contidas na própria requisição. Cabe a esse controlador especificar quais ações tomar, como a atualização de um *Model* e a resposta HTTP que será retornada. Nesse contexto, esse padrão dá ao desenvolvedor total liberdade no desenvolvimento, embora o force a conhecer sobre o protocolo HTTP para que possa desenvolver uma aplicação web.

Ao adotar esse padrão também não estamos presos a uma tecnologia específica para a exibição de telas. É possível, por exemplo, que escolhamos qualquer mecanismo de *Templating* que desejarmos, como o FreeMarker, Mustache e muitos outros. Podemos até mesmo ignorar um mecanismo de *Templating* e optar por páginas puras escritas em JSP.

MVC 1.0

A JSR 371, conhecida apenas por MVC 1.0, será lançada em conjunto com o Java EE 8, que tem previsão de lançamento para o final de 2016. Até lá, há um longo caminho, e muitas mudanças

ainda podem ocorrer. Contudo, já temos, no momento, um esboço de como será a API do MVC 1.0, o que nos permite criar uma aplicação e ver seu funcionamento na prática. Isso é possível graças ao projeto Ozark, que é a sua implementação de referência.

Quanto a uma possível competição entre o MVC 1.0 e o JSF, os autores de ambas as especificações deixaram claro que o MVC 1.0 não vem para substituir o JSF. Esse último ainda terá uma longa caminhada de atualizações e não há qualquer previsão de descontinuá-lo. De fato, a JSR 371 vem para padronizar o uso do MVC baseado em ações dentro do Java EE, de tal forma que ele possa se integrar de maneira simples e transparente com as demais JSRs, como, por exemplo, a CDI e Bean Validation.

JAX-RS

Conforme será discutido no decorrer deste artigo, a MVC 1.0 utiliza extensivamente as características da JAX-RS, JSR que permite criar serviços seguindo o modelo REST em Java. De fato, ela foi usada como base para o MVC 1.0, pois já provia grande parte das funcionalidades imaginadas para essa nova opção. Devido a esse uso da JAX-RS, o MVC 1.0 é enxuto, contendo poucas classes, interfaces e anotações para usarmos.

Consideramos esse aspecto como muito positivo, uma vez que a JAX-RS já se encontra no mercado há bastante tempo (desde 2008), é de amplo conhecimento da comunidade e está muito madura. Você notará, também, que os conceitos da JAX-RS se aplicam perfeitamente para os casos de uso do MVC 1.0, sem causar qualquer confusão ou sobreposição de ideias entre ambas as JSRs.

Aplicação de exemplo: Cadastro de produtos

Para demonstrar o uso da MVC 1.0, criaremos um projeto simples de cadastro de produtos, no qual poderemos adicionar, atualizar, excluir e listar os produtos de uma loja fictícia. Como você pode verificar, trata-se de uma aplicação de CRUD bastante simples, já que nosso maior objetivo é focar apenas nas características da JSR. Por esse motivo, não daremos muita atenção à construção das páginas de exibição dos dados, que podem ser criadas usando qualquer motor de *templating*, ou até mesmo a tecnologia JSP.

Instalação do Ozark e GlassFish

O primeiro passo na criação de nossa aplicação é a instalação da implementação de referência, chamada Ozark. Nesse quesito, temos duas opções: baixar o JAR no site do Ozark ou usar algum sistema que tenha um gerenciamento de dependências, como o Maven ou Gradle. Neste artigo, apresentaremos como realizar a instalação no Maven, no qual basta adicionar o código da *Listagem 1* na seção *dependencies* do seu arquivo *pom.xml*.

Em seguida, precisaremos de um servidor de aplicação para rodar o projeto. Para esse exemplo, utilizaremos uma versão especial do GlassFish, pois o MVC 1.0 ainda não está integrado às versões atuais do Java EE e, consequentemente, não está disponível nas versões anteriores desse servidor. A versão que utilizaremos ainda se encontra em desenvolvimento e é conhecida por *Nightly Build*, podendo ser obtida no endereço disponibilizado na seção *Links*.

Após baixar o arquivo, descompacte-o em qualquer pasta de sua preferência e o execute em seguida.

Criando a estrutura do projeto

Para este artigo, usaremos o Eclipse como nosso ambiente de desenvolvimento, entretanto, por ser baseado no Maven, este projeto pode ser usado em qualquer IDE que se integre com essa ferramenta ou em qualquer editor de sua preferência, uma vez que podemos executar o Maven através da linha de comando. Para nosso estudo de caso, adotaremos a versão **Mars.1** do Eclipse, que pode ser obtida em seu site oficial.

Listagem 1. Trecho de código em XML para adicionar o Ozark ao projeto usando o Maven.

```
<dependency>
  <groupId>com.oracle.ozark</groupId>
  <artifactId>ozark</artifactId>
  <version>1.0.0-m01</version>
  <scope>compile</scope>
</dependency>
```

Dentro dessa IDE, crie um projeto com suporte ao Maven e adicione as linhas da *Listagem 1* ao seu arquivo *pom.xml*. Lembre-se, também, de marcar a opção *Packaging* do seu projeto com a opção **war**, para que possamos realizar o *deploy* desse projeto no GlassFish. Em seguida, crie uma estrutura de pacotes conforme a *Figura 1*, na qual temos o pacote **br.com.devmedia.javamagazine.controller**, onde guardaremos nossas classes que representam controladores, e o pacote **br.com.devmedia.javamagazine.model**, no qual guardaremos nossas classes que representam modelos.

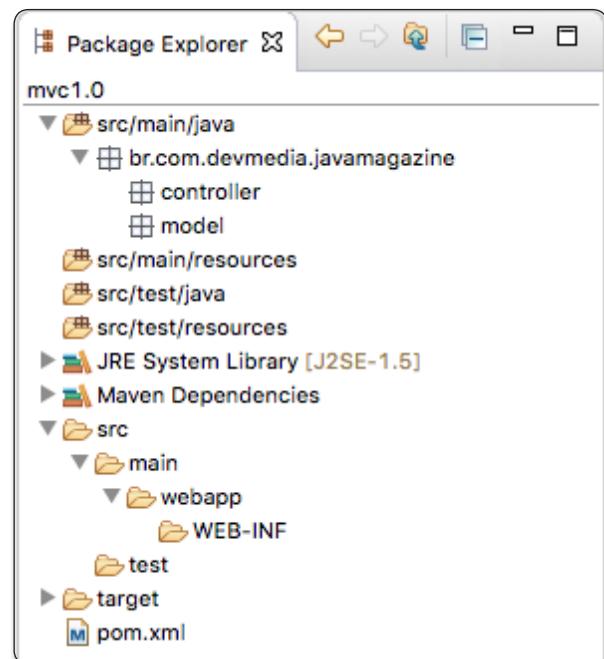


Figura 1. Estrutura de pastas do projeto

Primeiros passos: representando um produto no sistema

O primeiro passo em nosso exemplo consiste em criar uma classe que representa um produto dentro de nossa aplicação. Tentaremos simplificá-la ao máximo, evitando perder tempo em detalhes desnecessários ao intuito deste artigo. Dessa forma, crie um arquivo conforme o da **Listagem 2**, na qual temos o código da classe **Produto**, contida no pacote **br.com.devmedia.javamagazine.model**, que é formada pelos atributos **codigo**, **preco**, **nome** e **descricao**. Observe, ainda nessa listagem, que omitimos os métodos **getters** e **setters** de cada atributo; portanto, não se esqueça de criá-los em seu projeto.

Listagem 2. Trecho de código contendo a classe **Produto**.

```
package br.com.devmedia.javamagazine.model;

import java.util.ArrayList;
import java.util.List;

public class Produto {
    private String codigo;
    private String nome;
    private Double preco;
    private String descricao;
    private static List<Produto> lista = new ArrayList<Produto>();

    static {
        lista.add(new Produto("C1", "Cerveja de Trigo", 6.50, "Cerveja de Trigo"));
        lista.add(new Produto("C2", "Cerveja Pilsen", 4.50, "Cerveja Pilsen"));
        lista.add(new Produto("C3", "Cerveja Stout", 16.50, "Cerveja Stout"));
        lista.add(new Produto("C4", "Cerveja IPA", 12.50, "Cerveja IPA"));
    }

    public Produto() {
    }

    public Produto(String codigo, String nome, Double preco, String descricao) {
        this.codigo = codigo;
        this.nome = nome;
        this.preco = preco;
        this.descricao = descricao;
    }

    public static List<Produto> listar() {
        return lista;
    }
}
```

Ainda nessa listagem, observe que definimos um construtor a mais, além do construtor padrão, cujo objetivo é facilitar a criação de instâncias dessa classe, e um atributo estático, que manterá a lista de produtos que temos no nosso sistema. Para inicializar essa lista, criamos um bloco estático, que será executado assim que nossa aplicação iniciar, responsável pela instanciação e inclusão de apenas quatro produtos nela. Finalizando, teremos um método estático chamado **listar()**, que retornará essa lista de produtos que criamos de forma manual.

Ponto de entrada

O passo seguinte para a construção da aplicação consiste em criar uma classe que estende de **javax.ws.rs.core.Application**, e

cujo objetivo é servir como um ponto de entrada do sistema. Essa classe é necessária porque é ela quem define qual será a nossa URL base, a partir da qual todas as demais serão criadas.

Listagem 3. Trecho de código contendo a classe **Aplicacao**.

```
package br.com.devmedia.javamagazine;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("web")
public class Aplicacao extends Application { }
```

A nossa classe de entrada será conforme o código apresentado na **Listagem 3**, na qual criamos uma classe chamada **Aplicacao**. Observe que anotamos essa classe com **@javax.ws.rs.Application-Path**, para que possamos informar a URL base. Essa URL seguirá o padrão *http://<seudominio>/<contexto>/web/*, onde *<seudominio>* deve ser substituído pelo endereço do seu servidor, ou *localhost*, caso você esteja desenvolvendo em sua máquina, e *<contexto>* deve ser substituído pelo nome do contexto que você deu à aplicação em seu container web. Por fim, você notará, nas seções seguintes, que a partir de agora todas as URLs subsequentes terão como base essa URL.

Nosso primeiro controlador

Antes de darmos o próximo passo, precisamos entender o que é e para que servem os controladores. No MVC 1.0, um controlador é o responsável por receber requisições HTTP e tratá-las conforme as regras da aplicação em desenvolvimento. Isso significa que quando um usuário acessar uma URL do seu sistema, um objeto, ou um método, será notificado que chegou uma requisição para acessar uma determinada funcionalidade. Esse objeto (ou método) se encarregará, então, de chamar os objetos responsáveis pelas regras de negócio e, por fim, redirecionar para uma página que exibirá os dados relacionados a essa requisição.

Para tornar essa ideia mais clara, imagine uma situação onde o usuário acessa uma URL específica para editar os dados de um determinado produto. Nesse caso, o MVC 1.0 irá definir um controlador responsável por capturar essa requisição, solicitar à camada de negócios o produto em questão e, por fim, redirecionar para uma página que exibirá os dados desse produto.

Com esse conceito esclarecido, criaremos a classe **ProdutoController**, representada na **Listagem 4**, responsável por permitir que o usuário liste, edite e inclua produtos. Nessa listagem, observe que **ProdutoController** se trata de uma classe bastante simples, já que não necessita estender de outras classes ou implementar interfaces. Precisamos apenas anotá-la com **@javax.ws.rs.Path** para que possamos informar qual o padrão de URL que ela “escutará” por requisições. Neste caso em específico, ela aguardará por todas as requisições que chegarem à URL *http://<seudominio>/<contexto>/web/produtos*.

Lembre-se, conforme elucidamos na seção anterior, que essa URL é formada dessa maneira porque o MVC 1.0 está concatenando o valor **web**, que definimos na classe **Aplicacao** através da anotação **@ApplicationPath**, e o valor **produtos**, informado com a anotação **@Path** nesse controlador.

Listagem 4. Primeiro trecho de código da classe ProdutoController.

```
package br.com.devmedia.javamagazine.controller;

import javax.inject.Inject;
import javax.mvc.Controller;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

import com.oracle.ozark.core.Models;

import br.com.devmedia.javamagazine.model.Produto;

@Path("produtos")
public class ProdutoController {

    @Inject
    private Models models;

    @Controller
    @GET
    public String listar() {
        models.put("produtos", Produto.listar());
        return "/WEB-INF/jsp/listar.jsp";
    }

}
```

Continuando a análise dessa listagem, observemos o atributo **models**, que definimos como sendo do tipo **com.oracle.ozark.core.Models**, próprio do Ozark. Ao abrir a definição dessa classe, notamos que ela implementa a interface **javax.mvc.Models**, que define o comportamento padrão para objetos que são responsáveis por transitar objetos do controlador para a visão. Objetos desse tipo nos permitem exibir dados de nossa aplicação para os usuários. Outro detalhe importante é que não instanciamos esse atributo. No lugar disso, injetamos uma instância através da anotação **@javax.inject.Inject**, que pertence ao CDI.

Nota

Lembre-se que para o CDI funcionar é necessário ter o arquivo beans.xml dentro do diretório WEB-INF.

Analisemos agora o método **listar()**, o qual anotamos com **@javax.mvc.Controller** e **@javax.ws.rs.GET**. A primeira anotação é própria do MVC 1.0 e é responsável por especificar que esse método é um controlador e, portanto, está aguardando por requisições. Saiba que apenas os métodos que contêm essa anotação receberão as requisições dos usuários. Deste modo, não se esqueça de declará-la. Essa anotação também pode ser usada no escopo da classe, significando que todos os métodos públicos dela são controladores.

Já a segunda anotação informa que esse método responde apenas por requisições HTTP do tipo GET. Observe que não

usamos a anotação **@Path** para informar a URL associada. Sendo assim, qualquer requisição do tipo GET que chegue à URL `http://<seudominio>/<contexto>/web/produtos` será tratada por esse método. Por último, incluímos no atributo **models** a lista de produtos e retornamos uma **String** contendo o caminho para a página JSP que será responsável pela exibição dos dados dessa lista.

Nota

Repourei como o método `put()` da interface `com.oracle.ozark.core.Models` é idêntico ao método `put()` de `java.util.Map`? De fato, essa classe foi implementada pela equipe do Ozark apenas como uma camada acima de `java.util.Map`. Internamente, ela classe repassa essas chamadas para uma instância de `java.util.Map` que é mantida em um atributo privado.

Listagem 5. Página em JSP responsável pela listagem de produtos.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
    <head></head>
    <body>
        <h1>Listagem de Produtos</h1>
        <div>
            <table>
                <c:forEach var="produto" items="${produtos}">
                    <tr>
                        <td>${produto.codigo}</td>
                        <td><a href="produtos/editar/${produto.codigo}">
                            ${produto.nome}</a></td>
                        <td>${produto.preco}</td>
                        <td>${produto.descricao}</td>
                    </tr>
                </c:forEach>
            </table>
            <br>
            <form method="GET" action="produtos/novo/">
                <button type="submit">Incluir</button>
            </form>
        </div>
    </body>
</html>
```

Finalizando esta seção, vemos na **Listagem 5** o código em JSP responsável pela exibição da lista de produtos que está representada através da variável **produtos**, a qual incluímos no objeto **models** em nosso controlador. Para exibir essa lista, usamos a tag **c:forEach**, que irá iterar por cada produto e exibir seus dados. Por fim, temos um botão que nos redireciona para uma tela de inclusão, que será o assunto da próxima seção.

Adicionando um produto

Agora que temos uma tela de listagem funcional, podemos avançar e permitir que o usuário adicione novos produtos. Para esse objetivo, precisamos modificar algumas classes que já criamos, assim como adicionar uma nova página JSP contendo um formulário. Começaremos analisando a classe **ProdutoController**, que deverá receber mais dois métodos, apresentados na **Listagem 6**.

Listagem 6. Novos métodos do controlador ProdutoController.

```
@Path("novo")
@Controller
@GET
public String cadastro() {
    return "/WEB-INF/jsp/editar.jsp";
}

@Path("novo")
@Controller
@POST
public String incluir(@BeanParam Produto produto) {
    Produto.adicionar(produto);
    models.put("produtos", Produto.listar());
    return "/WEB-INF/jsp/listar.jsp";
}
```

O primeiro método, que chamaremos de **cadastro()**, não recebe parâmetros e será responsável apenas pelo redirecionamento para uma tela que permitirá o usuário preencher o formulário de cadastro de produtos. Observe que, diferente do método **listar()**, utilizamos a anotação **@Path** para indicar que ele responde à URL `http://<seudominio>/<contexto>/web/produtos/novo`. Também adotamos a anotação **@GET** para salientar que esse método responderá apenas às requisições cuja a forma de envio HTTP seja do tipo GET. No corpo desse método, retornamos o caminho onde se encontra a página JSP que exibirá o formulário.

Em seguida, criamos o método **incluir(Produto)**, que será responsável por realizar o cadastro do produto passado como parâmetro, incluindo-o na lista que estamos mantendo internamente. Esse método está anotado com **@POST**, pois responderá apenas a esse método HTTP.

Observe que o único parâmetro que **incluir()** recebe está anotado com **@javax.ws.rs.BeanParam**, anotação pertencente ao JAX-RS. Seu funcionamento é idêntico no MVC 1.0, pois trata-se de uma anotação que deve ser usada toda vez que precisarmos receber um objeto complexo como parâmetro em um controlador. Ao utilizá-la, estamos informando que queremos que um objeto do tipo **Produto** seja criado automaticamente a partir dos dados oriundos da requisição, que pode conter dados de um formulário, por exemplo.

Entretanto, ainda não declaramos qual dado será atribuído a qual atributo desse objeto. Por exemplo, qual dado deve ser colocado no atributo **nome**? Lembre-se que a requisição chega ao controlador com diversos dados, oriundos tanto de formulários quanto da Query String (aqueles variáveis passadas através da URL); portanto, precisamos instruir o Ozark sobre quais dados ele deve usar para criar esse objeto. Aqui entra a anotação **@FormParam**, que vamos utilizar em cada atributo da classe **Produto** para mapear o atributo para um valor do formulário, conforme a **Listagem 7**. Por exemplo, estamos mapeando o atributo **nome** para um campo de texto também chamado **nome** no formulário.

Analisemos agora o corpo do método **incluir()**, no qual adicionamos o produto à nossa lista interna invocando o método **adicionar(Produto)**, que será criado em breve, na classe **Produto**.

Logo após, mais uma vez colocamos a lista de produtos no objeto **models** e concluímos direcionando o usuário para a tela de listagem. Feito isso, para que essa funcionalidade fique completa, precisamos criar o método **adicionar(Produto)**, apresentado na **Listagem 7**.

Listagem 7. Modificações na classe **Produto** para adicionar a anotação **@FormParam**.

```
public class Produto {

    @FormParam("codigo")
    private String codigo;

    @FormParam("nome")
    private String nome;

    @FormParam("preco")
    private Double preco;

    @FormParam("descricao")
    private String descricao;

    public static void adicionar(Produto produto) {
        lista.add(produto);
    }

}
```

Neste método, apenas incluímos o produto que recebemos como parâmetro na nossa lista interna. Para finalizar essa funcionalidade, criaremos a página JSP contendo o formulário. Deste modo, crie um arquivo chamado *editar.jsp* dentro do diretório `/WEB-INF/jsp/` e defina seu conteúdo conforme a **Listagem 8**, que contém um formulário e caixas de texto para o usuário informar os dados do produto. Note que esse formulário deve ter o atributo **method** definido como POST e o atributo **action** deve ficar vazio, pois queremos fazer uma submissão para a URL atual (nesse caso em específico, `http://<seudominio>/<contexto>/web/produtos/novo`).

Listagem 8. Página JSP para criação e edição de produtos.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<header> </header>
<body>
    <h1>Cadastrar Produto</h1>

    <div>
        <form method="POST" action="">
            Código: <input type="text" name="codigo" id="codigo" value="${produto.codigo}" /><br>
            Nome: <input type="text" name="nome" id="nome" value="${produto.nome}" /><br>
            Preço: <input type="text" name="preco" id="preco" value="${produto.preco}" /><br>
            Descrição: <input type="text" name="descricao" id="descricao" value="${produto.descricao}" /><br>
            <button type="submit">Enviar</button>
        </form>
    </div>
</body>
</html>
```

Note, ainda, que cada campo do tipo `<input>` referencia um valor do produto através do atributo `value`. Como estamos implementando a funcionalidade de adição, a variável `produto` ainda não está disponível através do objeto `models` (de forma semelhante ao que fizemos na tela de listagem), mas estará assim que implementarmos a funcionalidade de edição na próxima seção. Nessa tela, o mais importante é o nome dado a cada campo do formulário. Lembra-se da conexão que fizemos, nos parágrafos anteriores, com os atributos da classe `Produto` usando a anotação `@FormParam`? Deste modo, não se esqueça de utilizar exatamente os mesmos nomes declarados nessas anotações para os campos do formulário.

Editando produtos

A próxima funcionalidade a ser desenvolvida permitirá ao usuário modificar um produto. Para tal, precisaremos, mais uma vez, editar algumas classes e arquivos de nosso projeto. Começamos adicionando mais dois métodos ao controlador de produtos, que estão descritos na [Listagem 9](#).

Listagem 9. Novos métodos para a classe ProdutoController.

```
@Path("{codigo}")
@Controller
@GET
public String preEditar(@PathParam("codigo") String cod) {
    models.put("produto", Produto.obter(cod));
    return "/WEB-INF/jsp/editar.jsp";
}

@Path("{codigo}")
@Controller
@POST
public String editar(@PathParam("codigo") String codigo,
    @BeanParam Produto produto) {
    Produto.editar(produto);
    models.put("produtos", Produto.listar());
    return "/WEB-INF/jsp/listar.jsp";
}
```

Nessa listagem temos algumas novidades, principalmente com relação ao uso da anotação `@Path` e à obtenção de variáveis passadas para nosso controlador a partir da URL. Para que precisamos disso? Porque para editarmos um produto, precisamos saber qual o identificador dele, de tal forma que possamos recuperá-lo de nosso mecanismo de persistência e apresentá-lo em um formulário para que o usuário o edite.

Existem diversas formas de passar essa informação para o nosso controlador, mas, neste exemplo, faremos isso através do endereço da URL. Assim, na nossa aplicação, caso o usuário queira editar o produto cujo código é COD1, basta acessar o endereço `http://<seudominio>/<contexto>/web/produtos/COD1`.

A dúvida que nos resta, então, é saber como obter o valor COD1 que está contido nessa URL. Quem já usou o JAX-RS sabe que essa não é uma tarefa difícil, já que se trata exatamente do mesmo mecanismo. O JAX-RS nos permite informar a existência de valores variáveis na URL através da anotação

`@Path`, bastando definir uma variável colocando-a entre chaves, da mesma forma como fizemos na [Listagem 9](#). Nesse código, a anotação `@Path` foi definida contendo apenas o texto `{codigo}`, o que informa ao MVC 1.0 que temos um valor variável na URL. A partir disso, o método `preEditar()` será chamado utilizando URLs como:

- `http://localhost/<contexto>/web/produtos/C1`;
- `http://localhost/<contexto>/web/produtos/ABC1`;
- `http://localhost/<contexto>/web/produtos/123456`.

Contudo, apenas isso não é suficiente, pois ainda queremos que esse valor nos seja passado como um parâmetro do tipo `String` para o método `preEditar()`. Para isso, adicionamos um parâmetro chamado `cod` a esse método, anotando-o com `@PathParam("codigo")`. Note que fizemos uma conexão entre o valor passado para essa anotação e o valor que colocamos entre chaves na anotação `@Path`. Por fim, observe que no corpo do método `preEditar()` obtemos o produto com esse código, o incluímos no `model` e redirecionamos o usuário para a tela de edição.

Concluindo essa funcionalidade, vamos adicionar o método `editar()`, que recebe dois parâmetros: o código do produto e o próprio produto que foi editado pelo usuário. Nesse método, utilizamos exatamente os mesmos mecanismos que adotamos em `incluir()`, implementado na [Listagem 6](#), e `preEditar()`, na [Listagem 9](#), mudando apenas o corpo do método, que atualiza o produto na nossa lista estática usando o método `editar()` da classe `Produto`, apresentado na [Listagem 10](#).

Nota

Os desenvolvedores mais atentos devem ter observado o uso da anotação `@POST` em um método no controlador cujo objetivo é editar um produto. Os mais puritanos dirão, certamente, que o mais correto é usar a anotação `@PUT`, e eles têm toda razão, contudo, os navegadores aceitam apenas os métodos POST e GET, ignorando o método PUT no envio de formulários. Na verdade, você pode até colocar o atributo `method` da tag `form` como sendo PUT, mas isso não adiantará muita coisa, pois os dados do formulário serão enviados usando o método GET.

Listagem 10. Novo método para atualizar produtos.

```
public static void editar(Produto produto) {
    Produto e = obter(produto.codigo);
    e.codigo = produto.codigo;
    e.descricao = produto.descricao;
    e.nome = produto.nome;
    e.preco = produto.preco;
}
```

Validando dados

O que devemos fazer caso o usuário peça para cadastrar um produto com o nome contendo o valor nulo? E se ele informar um preço com o valor -10? Devemos aceitar o cadastro desse produto? Isso depende da regra de negócio de cada sistema e, em nosso caso em especial, colocaremos algumas restrições para que cadastros desse tipo não sejam permitidos. Porém, como fazer isso?

Através da integração do MVC 1.0 com a Bean Validation, uma JSR que também faz parte do Java EE, e cujo objetivo é validar os atributos de um determinado objeto.

Quem já usou a Bean Validation sabe o quanto é fácil utilizá-la e, para quem a está vendo pela primeira vez, faremos algumas modificações em nossas classes para lhes mostrar o quanto o uso dessa JSR é simples e intuitivo.

Começaremos alterando a classe **Produto**, pois precisamos anotar alguns atributos com anotações específicas da Bean Validation. Essas alterações encontram-se na **Listagem 11**, onde podemos ver o uso das anotações `@javax.validation.constraints.NotNull` e `@javax.validation.constraints.Min`. A primeira anotação deve ser usada nos atributos que não devem aceitar valores nulos, enquanto a segunda deve ser usada quando queremos definir um valor numérico mínimo.

Listagem 11. Anotando a classe Produto com anotações da Bean Validation.

```
public class Produto {  
  
    @FormParam("codigo")  
    @NotNull  
    private String codigo;  
  
    @FormParam("nome")  
    @NotNull  
    private String nome;  
  
    @FormParam("preco")  
    @NotNull  
    @Min(0)  
    private Double preco;  
  
    @FormParam("descricao")  
    private String descricao;  
}
```

Em seguida, precisamos de algumas modificações na classe **ProdutoController**, conforme a **Listagem 12**. Iniciamos com a injecão de um objeto do tipo `javax.mvc.validation.ValidationResult` na variável **validationResult**, cujo objetivo é manter o resultado da validação. Conforme veremos a seguir, é através desse objeto que saberemos se houve alguma falha de validação e quais foram essas falhas.

Nosso próximo passo consiste em adicionar a anotação `@javax.validation.executable.ValidateOnExecution` ao método responsável pela inclusão de um produto, pois é ela quem informa que a validação dos objetos deve ocorrer no ciclo de execução da requisição. E quem também precisa de uma anotação é o parâmetro **produto**, recebido por esse método. Nesse caso utilizamos `@javax.validation.Valid`, que avisa ao MVC 1.0 que esse parâmetro deve ser validado.

Agora estamos aptos a usar o objeto **validationResult**, através do método `isFailed()`, para checar se a requisição falhou devido a problemas de validação. Observe, na **Listagem 12**, que quando detectamos uma falha, colocamos os erros de validação no objeto

model, de tal forma que possamos obtê-las na nossa visão para exibi-las ao usuário, de acordo com a **Listagem 13**.

Listagem 12. Alterações na classe ProdutoController para usar a Bean Validation.

```
@Path("produtos")  
public class ProdutoController {  
  
    @Inject  
    private ValidationResult validationResult;  
  
    @Path("novo")  
    @Controller  
    @POST  
    @ValidateOnExecution(type = ExecutableType.NONE)  
    public String incluir(@BeanParam @Valid Produto produto) {  
        String result = "/WEB-INF/jsp/listar.jsp";  
  
        if (validationResult.isFailed()) {  
            models.put("erros", validationResult.getAllViolations());  
            models.put("produto", produto);  
            result = "/WEB-INF/jsp/editar.jsp";  
        } else {  
            Produto.adicionar(produto);  
            models.put("produtos", Produto.listar());  
        }  
  
        return result;  
    }  
}
```

Listagem 13. Exibindo os erros de validação na tela editar.jsp.

```
<table>  
    <c:forEach var="erro" items="${erros}">  
        <tr>  
            <td style="color: red">${erro.message}</td>  
        </tr>  
    </c:forEach>  
</table>
```

Tratando exceções

Suponha que ocorreu um erro totalmente inesperado em nossa aplicação ou, então, o usuário executou uma ação não planejada, como digitar uma URL para editar um produto que não existe no sistema. O que fazemos nessas situações? Simplesmente deixamos aparecer aquela tela genérica, na qual é exibida a pilha de erros do Java? Obviamente, a resposta é não. Contudo, como tratamos uma exceção de forma centralizada na nossa aplicação, de forma que ela seja capturada independentemente de onde tenha sido lançada?

Para resolver esse problema, podemos nos beneficiar do uso dos Exception Mappers do JAX-RS, que são nada mais do que objetos que ficam “escutando” por exceções que não foram tratadas, através de blocos `try-catch`, no fluxo normal da aplicação. Para exemplificar seu uso, vamos alterar a classe **Produto**, conforme a **Listagem 14**, alterando a lógica do método `obter()` para lançar a exceção `ObjectNotFoundException` (contida na **Listagem 15**) quando não for encontrado um objeto com o código informado.

Note que não realizaremos qualquer modificação, no nosso controlador, para tratar essa exceção, de forma que ela continuará

MVC baseado em ações no Java EE

seu fluxo na pilha até ser exibida ao usuário. Caso você esteja curioso, execute a aplicação agora e digite uma URL para editar um produto inexistente, como `http://<seudominio>/<contexto>/web/produtos/codigonaosexiste`. Provavelmente será exibida uma tela informando que uma exceção foi lançada e não tratada.

Listagem 14. Modificando o método `obter()`, da classe `Produto`, para lançar uma exceção.

```
public static Produto obter(String codigo) {
    Produto result = null;

    for (Produto produto : lista) {
        if (produto.codigo.equals(codigo)) {
            result = produto;
            break;
        }
    }

    if (result == null) {
        throw new ObjectNotFoundException();
    }

    return result;
}
```

Agora, vamos tratar essa exceção de forma que essa tela não seja exibida. No lugar dela, iremos exibir uma página em JSP, informando que houve um problema. Começamos, então, criando a classe `ObjectNotFoundExceptionMapper`, que ficará responsável por interceptar todas as exceções do tipo `ObjectNotFoundException` (**Listagem 15**) que forem lançadas e não tratadas no fluxo.

Esta nova classe encontra-se na **Listagem 16**, onde implementamos a interface `javax.ws.rs.ext.ExceptionMapper` e informamos, através de Generics, que a exceção que esse Exception Mapper tratará será a `ObjectNotFoundException`. Para que essa classe seja reconhecida pelo JAX-RS, precisamos anotá-la com `@javax.ws.rs.ext.Provider`.

Listagem 15. Código da classe `ObjectNotFoundException`.

```
package br.com.devmedia.javamagazine.exception;

public class ObjectNotFoundException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Em seguida, implementamos o método `toResponse()`, oriundo da interface `ExceptionMapper`, e retornamos uma resposta com o status `Status.BAD_REQUEST`, que corresponde ao erro 400 do HTTP. Finalizamos esse método redirecionando o usuário para a tela `erro.jsp`, que se trata de uma tela simples, contendo apenas um texto informativo. De agora em diante, sempre que uma exceção

`ObjectNotFoundException` ocorrer e não for tratada por um bloco `try/catch`, esse Exception Mapper será invocado e o usuário será redirecionado para uma tela de erro.

Listagem 16. Código da classe `ObjectNotFoundExceptionMapper`.

```
@Provider
public class ObjectNotFoundExceptionMapper implements ExceptionMapper<ObjectNotFoundException> {

    @Override
    public Response toResponse(ObjectNotFoundException exception) {
        return Response
            .status(Status.BAD_REQUEST)
            .entity("/WEB-INF/jsp/erro.jsp")
            .build();
    }
}
```

A anotação `@View`

Certamente, você deve ter observado que todo método de nosso controlador precisa informar a página a ser exibida ao usuário. Deve ter notado, também, que fizemos isso devolvendo uma `String` que determina o caminho até essa página. Essa tática é interessante quando a página a ser exibida é definida de forma dinâmica, ou seja, não é pré-definida, dependendo de alguma ação do usuário. Isso ocorre, por exemplo, nos métodos `incluir()` e `editar()` do nosso controlador.

Entretanto, observe que também existem métodos que usam a mesma página (ou visão), independentemente das ações que o usuário realize. Nesses casos, podemos nos beneficiar do uso da anotação `@javax.mvc.View`, passando como parâmetro para ela o caminho para a página, conforme a **Listagem 17**. Observe que, ao usar essa anotação, não necessitamos mais de uma `String` no nosso método, o que justifica termos mudado o retorno dele para `void`.

Listagem 17. Utilização da anotação `@View` na classe `ProdutoController`.

```
@Controller
@GET
@View("/WEB-INF/jsp/listar.jsp")
public void listar() {
    models.put("produtos", Produto.listar());
}
```

Ainda existem mais duas formas de avisar ao MVC qual a página a ser exibida: devolvendo do método do controlador um objeto do tipo `javax.ws.rs.core.Response`, da mesma forma como fizemos na nossa classe `ObjectNotFoundExceptionMapper`, ou retornando um objeto do tipo `javax.mvc.Viewable`. A **Listagem 18** ilustra o uso da classe `Response`, onde alteramos o método `editar()` para retornar um objeto desse tipo. Por último, na **Listagem 19**, alteramos o método `incluir()`, para devolver um `Viewable`.

Listagem 18. Utilização do Response na classe ProdutoController.

```
@Path("{codigo}")
@Controller
@POST
@ValidateOnExecution(type = ExecutableType.NONE)
public Response editar(@PathParam("codigo") String codigo,
@BeanParam @Valid Produto produto) {
    String result = "/WEB-INF/jsp/listar.jsp";

    if (validationResult.isFailed()) {
        models.put("erros", validationResult.getAllViolations());
        models.put("produto", produto);
        result = "/WEB-INF/jsp/editar.jsp";
    } else {
        Produto.editar(produto);
        models.put("produtos", Produto.listar());
    }

    return Response.status(Status.OK).entity(result).build();
}
```

Listagem 19. Utilizando a classe Viewable para endereçar páginas.

```
@Path("novo")
@Controller
@POST
@ValidateOnExecution(type = ExecutableType.NONE)
public Viewable incluir(@BeanParam @Valid Produto produto) {
    String result = "/WEB-INF/jsp/listar.jsp";

    if (validationResult.isFailed()) {
        models.put("erros", validationResult.getAllViolations());
        models.put("produto", produto);
        result = "/WEB-INF/jsp/editar.jsp";
    } else {
        Produto.adicionar(produto);
        models.put("produtos", Produto.listar());
    }

    return new Viewable(result);
}
```

Processadores de template

Caso você seja um desenvolvedor Java web de longa data, provavelmente já trabalhou com algum processador de templates, conhecido simplesmente por Motor de Template (do inglês *Template Engine*). Para quem nunca trabalhou com essas ferramentas, esses motores são responsáveis por processar arquivos de template, que são formados por marcações HTML em conjunto com marcações específicas que só esse motor reconhece. É trabalho de um processador, por exemplo, substituir toda marcação \${produto.nome} que ele encontrar pelo valor do atributo **nome** contido em um objeto chamado **produto**, que foi disponibilizado pelo controlador para ele.

Embora o MVC 1.0 use por padrão o JSP, não estamos presos a essa tecnologia, já que podemos utilizar diversos motores de

template que existem no mercado, como o Velocity, Mustache, FreeMarker, Handlebars e Thymeleaf. No caso do Mustache, por exemplo, basta adicionarmos a dependência ao Ozark Mustache em nosso *pom.xml*, conforme a **Listagem 20**, e informarmos, em nossos controladores, o caminho para um template do Mustache, no lugar de uma página JSP.

Listagem 20. Adicionando suporte ao Mustache no pom.xml.

```
<dependency>
    <groupId>com.oracle.ozark.ext</groupId>
    <artifactId>ozark-mustache</artifactId>
    <version>1.0.0-m01</version>
    <scope>compile</scope>
</dependency>
```

Capturando eventos

Embora seja de uso pouco trivial em soluções comerciais, o MVC 1.0 disponibiliza um evento, integrado ao CDI, que pode ser capturado por sua aplicação através da anotação **@javax.enterprise.event.Observes**. Esse evento pode ser interceptado para, por exemplo, realizar ações de auditoria ou, simplesmente, para fazer o *log* do que aconteceu em seu sistema com o objetivo de identificar os motivos de alguns erros terem ocorrido. Outra possível utilização diz respeito à possibilidade de avaliar quais funcionalidades do sistema são mais utilizadas, integrando sua aplicação a ferramentas como o Google Analytics.

Listagem 21. Capturando eventos do MVC.

```
public class Auditoria {

    public void auditar(@Observes ControllerMatched event) {
        System.out.println("Match: " + event.getResourceInfo().getResourceMethod());
    }
}
```

A **Listagem 21** apresenta como podemos capturar esse evento através da classe **Auditoria**. Nessa classe, criamos o método **auditar()**, que recebe como parâmetro um objeto do tipo **ControllerMatched** e que está anotado com **@Observes**. Esse método será invocado sempre que o MVC 1.0 encontrar um controlador responsável pelo tratamento de uma requisição e nele podemos, por exemplo, realizar o *log* das ações do usuário.

O MVC 1.0 nos brinda com mais uma forma de criar aplicações para a web com o Java EE. Antes, tínhamos apenas o JSF como opção, mas, em breve, poderemos utilizar também o MVC baseado em ações, em contrapartida ao MVC baseado em componentes. Isso é muito importante para o ecossistema do Java, pois atualmente o desenvolvedor encontra-se de mãos atadas, tendo o JSF como sua única opção para seu próximo projeto web.

MVC baseado em ações no Java EE

A integração do MVC ao Java EE é ainda mais relevante quando verificamos um crescimento no interesse por frameworks como o Spring MVC e o VRaptor, que adotam o MVC baseado em ações, o que nos indica que esse padrão ainda é, ou sempre foi, muito utilizado por profissionais que atuam com o Java. O lançamento dessa JSR, portanto, fomentará o surgimento de mais frameworks, aumentando o leque de opções para o desenvolvedor, que poderá optar por aquele que melhor se adapta às suas necessidades.

Por fim, embora essa API ainda não esteja oficialmente lançada, você já pode ter uma ideia de como ela será através do projeto Ozark. Conhecer essa nova JSR, mesmo que mudanças em sua API ainda estejam por vir, é importante para você ter uma ideia do que virá pela frente, tornando-o apto a compreender como ela afetará seus futuros projetos.

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Autor



Marlon Silva Carvalho

marlon.carvalho@gmail.com

Programador poliglota há mais de 20 anos, é fã de programação para dispositivos móveis e web. Tem na tecnologia sua paixão, na família sua motivação e na fabricação de cervejas artesanais seu hobby predileto. Também é Bacharel pela Universidade Católica do Salvador, Pós-graduado em Sistemas Distribuídos pela Universidade Federal da Bahia e Organizer do Google Developers Group de Salvador. Siga seu twitter em @marlonscarvalho ou acesse sua página pessoal em <http://marlon.co/>.



Links:

MVC 1.0.

<https://www.jcp.org/en/jsr/detail?id=371>

Ozark.

<http://ozark.java.net>

Texto do Martin Fowler sobre a importância do MVC.

<http://martinfowler.com/eaaDev/uiArchs.html>

Padrão Separated Presentation.

<http://martinfowler.com/eaaDev/SeparatedPresentation.html>

Boletim Técnico da ThoughtWorks sobre o JSF.

<http://thoughtworks.fileburst.com/assets/technology-radar-jan-2014-pt.pdf>

Spring MVC.

<https://spring.io/>

GlassFish Nightly Build.

http://dlc.sun.com.edgesuite.net/glassfish/4.1/nightly/glassfish-4.1-b13-03_16_2015.zip

Autorizações dinâmicas com Spring Security

Aprenda como criar uma infraestrutura com Spring Security que permite a definição dinâmica de permissões de acesso

ESTE ARTIGO É DO TIPO MENTORING

SAIBA MAIS: WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI

Segurança é um requisito fundamental em grande parte das aplicações web, pois elas estão mais sujeitas a ataques, por conta de sua maior exposição. Para se ter uma ideia, mais de 1 milhão de incidentes foram reportados em 2014 ao Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil (CERT.br). Uma forma de minimizar a ocorrência desses incidentes, preservando a integridade do sistema e de suas informações, consiste em restringir o acesso a determinadas partes da aplicação a usuários autenticados, que, por sua vez, podem acessar somente funcionalidades relacionadas às suas competências.

Nesse contexto, o Spring Security é um framework destinado a proteger aplicações web, garantindo a identidade de seus usuários, por meio da funcionalidade de autenticação, e provendo controle de acesso a recursos, por meio da funcionalidade de autorização. Essas funcionalidades são embasadas por duas estruturas de dados básicas: usuário e autorização. Usuário contém os dados utilizados para validação da identidade dos usuários, como nome e senha, enquanto autorização contém os perfis de acesso, que agrupam diversas funcionalidades da aplicação. Um exemplo típico de perfil de acesso é o de “administrador”, que geralmente possui acesso a todas as funcionalidades. Nesse momento, saiba que a relação entre usuário e autorização é de “muitos para muitos”, onde cada usuário pode possuir várias autorizações e cada autorização pode pertencer a vários usuários.

Cenário

Este artigo é útil em situações onde alterações nas políticas de acesso aos recursos da aplicação ocorrem frequentemente. Ele apresenta uma infraestrutura, baseada no framework Spring Security, que permite alterar as permissões de acesso dos usuários sem necessidade de codificação e, consequentemente, geração de novos releases. Ademais, apresenta também alguns dos novos recursos da versão 4 do Spring Security. Após sua leitura, o leitor será capaz de realizar o gerenciamento de acessos de seus sistemas com maior flexibilidade e agilidade.

As associações entre os perfis de acesso e as funcionalidades geralmente são definidas de forma estática, conforme exemplos contidos no material de referência e guias disponíveis no site do Spring Security. Essa construção cobre grande parte das situações existentes, mas torna necessária a participação de um desenvolvedor e a geração de uma nova versão da aplicação web sempre que for preciso alterar quais funcionalidades um perfil pode acessar. No entanto, infelizmente nem sempre é possível esperar uma próxima versão para o estabelecimento de uma nova política de acesso.

Diante desse cenário, o objetivo deste artigo é apresentar uma modelagem de dados alternativa mais flexível, que permita a criação e exclusão de perfis de acesso, assim como a associação e dissociação dinâmica de funcionalidades. Com isso, seremos capazes de criar aplicações, com segurança baseada em Spring Security, que suportem alterações constantes de políticas de acesso.

Spring Security 4

Utilizaremos neste artigo a versão 4 do Spring Security, lançada em março de 2015, para gerenciar a segurança de uma aplicação exemplo, que nos auxiliará na fixação dos conceitos apresentados. Essa versão apresenta como principais características a inclusão

Autorizações dinâmicas com Spring Security

do suporte ao protocolo WebSocket, integração com o Spring Data e disponibilização de anotações para utilização em testes.

Suporte a WebSocket

A primeira nova característica do Spring Security 4 é a inclusão do suporte ao gerenciamento de autorizações aplicadas às mensagens do protocolo WebSocket (vide **BOX 1**), por meio da abstração Spring Messaging. Essa nova opção preenche uma lacuna de quase dois anos desde o lançamento da versão 4 do Spring Framework, em 2013, que incorporou esse protocolo.

Com isso, podemos, agora, aplicar políticas de acesso às mensagens transitas via WebSocket, tanto por Java Configuration quanto por XML, utilizando as autorizações associadas ao usuário autenticado. A **Listagem 1** apresenta um exemplo de configuração por Java Configuration, onde cada mensagem com destino iniciado em `/user/` requer que o usuário se encontre autenticado. Já a **Listagem 2** apresenta uma configuração por XML, onde todas as mensagens do tipo CONNECT são aceitas, todas as mensagens do tipo SUBSCRIBE são negadas e cada mensagem com destino iniciado em `/user/` requer que o usuário se encontre autenticado e possua a autorização **ROLE_USER**.

BOX 1. WebSocket

WebSocket é um protocolo, padronizado pela W3C e parte da especificação do HTML5, que proporciona comunicação bidirecional full-duplex entre clientes web e servidores sobre uma única conexão TCP. Essa tecnologia foi criada como uma alternativa ao método tradicional, que consiste na abertura de múltiplas conexões HTTP, o que acaba por gerar sobrecarga e aumento de latência.

Listagem 1. Configuração por Java Configuration.

```
01 @Configuration
02 public class WebSocketSecurityConfig extends AbstractSecurityWeb
  SocketMessageBrokerConfigurer {
03
04   protected void configureInbound(MessageSecurityMetadataSource
    Registry messages) {
05     messages.simpDestMatchers("/user/*").authenticated();
06   }
07 }
```

Listagem 2. Configuração por XML.

```
01 <websocket-message-broker>
02   <intercept-message type="CONNECT" access="permitAll"/>
03   <intercept-message type="SUBSCRIBE" access="denyAll"/>
04   <intercept-message pattern="/user/**" access="hasRole('USER')"/>
05 </websocket-message-broker>
```

Integração com Spring Data

Outra importante novidade do Spring Security 4 é a possibilidade de utilizar as informações do usuário autenticado diretamente em consultas criadas com a anotação `@Query` do Spring Data. Em versões anteriores seria necessário recolher essas informações na camada web e passá-las como parâmetro, mas a versão 4 permite referenciar os atributos do **principal** (objeto que armazena os dados do usuário autenticado) diretamente.

Na **Listagem 3**, por exemplo, o método `findInbox()` recupera todas as mensagens endereçadas ao usuário autenticado acessando seu id diretamente, sem necessidade de recebê-lo como parâmetro.

Listagem 3. Consulta utilizando atributo de usuário autenticado.

```
01 @Repository
02 public interface MessageRepository extends PagingAndSortingRepository
  <Message,Long> {
03   @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
04   Page<Message> findInbox(Pageable pageable);
05 }
```

Suporte a testes

O Spring Security permite, por meio de anotações, restringir o acesso a métodos, exigindo que o usuário autenticado possua determinadas autorizações. Em suas versões anteriores, entretanto, não era possível verificar essas restrições durante a execução dos testes automatizados. Para sanar essa deficiência, a versão 4 passou a disponibilizar novas anotações voltadas aos testes que precisem acessar métodos que requeiram autorização, a saber:

- `@WithMockUser` – Possibilita a execução de um teste utilizando um usuário autenticado fictício. É possível especificar o nome do usuário e suas autorizações;
- `@WithUserDetails` – Possibilita a execução de um teste utilizando os dados de um usuário existente, recuperado por meio de uma chamada a um bean que implemente `UserDetailsService`; e
- `@WithSecurityContext` – Possibilita a criação de anotações que definam `SecurityContext` personalizados para os testes.

O Spring Security 4 também inclui integração com versões superiores à 4.1.3 do Spring MVC Test, possibilitando a realização de testes que envolvam: proteção contra CSRF (vide **BOX 2**); operações realizadas por usuários (fictícios ou existentes) autenticados; e autenticação de usuários.

Modelagem tradicional

Como citado anteriormente, o Spring Security se baseia em duas estruturas de dados: **Usuário** e **Autorização**, representadas pelas interfaces `UserDetails` e `GrantedAuthority`, respectivamente.

BOX 2. Cross-Site Reference Forgery (CSRF)

CSRF é um tipo de ataque onde um usuário autenticado é levado a executar ações indesejadas em uma aplicação web. Nesse tipo de ataque, o atacante utiliza engenharia social para fazer com que a vítima acesse sites maliciosos enquanto se encontra autenticada na aplicação web alvo. Esses sites então enviam requisições falsas que utilizam as autorizações da vítima para realizar operações que beneficiam o atacante como, por exemplo, transferências de dinheiro em sites bancários.

Ataques CSRF geralmente ocorrem por meio de requisições do tipo GET ou POST, pois servidores geralmente não aceitam requisições do tipo PUT e DELETE provenientes de outros domínios, a não ser em caso de má configuração ou bugs.

Como forma de prevenção, recomenda-se o envio, em cada requisição, de um token CSRF gerado aleatoriamente pelo servidor por usuário e por sessão de usuário. O servidor deve passar então a verificar todas as requisições, rejeitando todas que apresentem um token CSRF incorreto.

Nas assinaturas dessas interfaces, apresentadas na **Figura 1**, verificamos que um usuário deve possuir um conjunto de autorizações, um nome, uma senha e alguns indicadores (se está bloqueado, expirado, etc.), enquanto uma autorização é composta basicamente por um texto, com prefixo **ROLE_**, que identifica um direito de acesso delegado aos usuários.

O material de referência e os guias disponibilizados no site do Spring Security sugerem a associação de cada autorização a vários recursos de uma aplicação web. Também sugerem a associação direta entre autorizações e usuários. Essas informações e as interfaces providas nos levam a uma modelagem tradicional, conforme a apresentada na **Figura 2**. Neste exemplo, optou-se, por questão de simplificação, pela não persistência dos indicadores. Além disso, foram incluídos identificadores (ids) numéricos para cada entidade, pois, seguindo as boas práticas de banco de dados, não é recomendável definir como identificadores atributos passíveis de modificação (caso dos atributos nome e senha, da classe **Usuario**, e nome, da classe **Autorizacao**).

Seguindo a mesma linha de raciocínio, em uma configuração típica de acessos no Spring Security, como a apresentada na **Listagem 4**, cada autorização possui acesso a um determinado conjunto estático de funcionalidades. Para melhor comprehendê-la, basta lembrar que a tag **intercept-url** define um padrão de URL a interceptar no parâmetro **pattern** e quais autorizações podem acessá-la no parâmetro **access**. Neste exemplo, a autorização **ROLE_ADMIN** possui acesso a todas as URLs, **ROLE_GERENTE** possui acesso somente às URLs que seguem o padrão `/seguro/produto/**` e `/seguro/venda/**` e **ROLE_VENDEDOR** possui acesso às URLs que seguem o padrão `/seguro/venda/**` e `/seguro/consulta/**`.

Mas, e se, por um acaso, as políticas de acesso forem alteradas? E se, por exemplo, descobrirem que usuários com a autorização **ROLE_GERENTE** também precisam acessar URLs que sigam o padrão `/seguro/venda/**`? Nesse caso específico, bastaria acrescentar **ROLE_GERENTE**

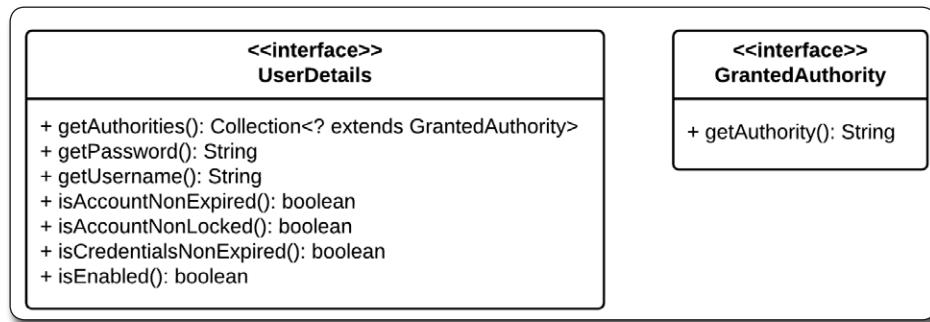


Figura 1. Interfaces UserDetails e GrantedAuthority

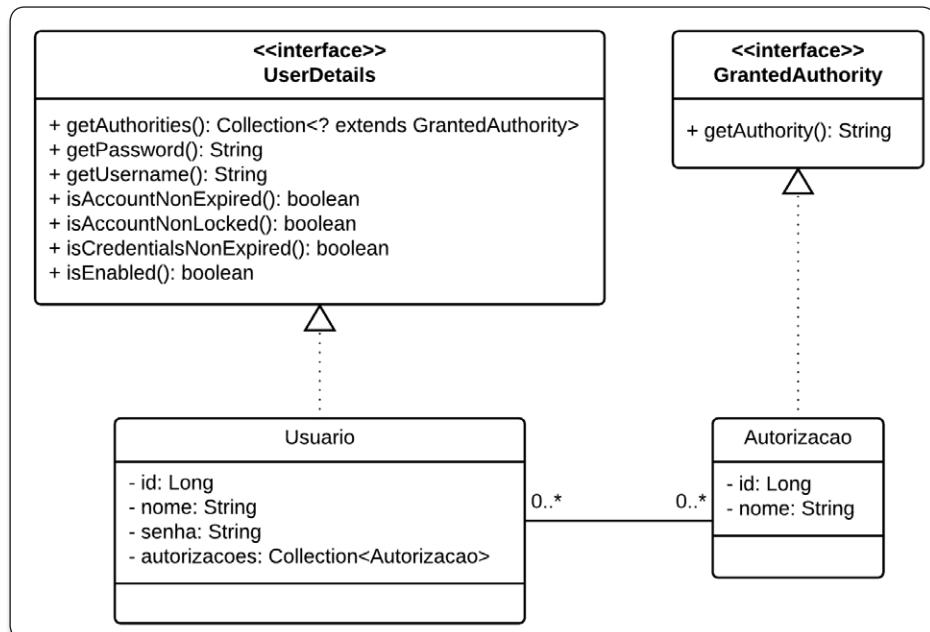


Figura 2. Diagrama de classes contemplando usuários e suas autorizações

Listagem 4. Exemplo de configuração do Spring Security.

```

01 <sec:intercept-url pattern="/seguro/perfil/**" access="ROLE_ADMIN"/>
02 <sec:intercept-url pattern="/seguro/produto/**" access="ROLE_GERENTE,ROLE_ADMIN"/>
03 <sec:intercept-url pattern="/seguro/usuario/**" access="ROLE_ADMIN"/>
04 <sec:intercept-url pattern="/seguro/venda/**" access="ROLE_VENDEDOR,ROLE_GERENTE,ROLE_ADMIN"/>
05 <sec:intercept-url pattern="/seguro/consulta/**" access="ROLE_VENDEDOR,ROLE_ADMIN"/>

```

no parâmetro **access** da linha 5 e enviar ao cliente no próximo release. Mas e se o cliente não puder esperar pelo próximo release? Deparamo-nos assim com a primeira limitação dessa modelagem: a incapacidade de lidar com alterações nas políticas de acesso de forma dinâmica, sem necessidade de codificação.

Além do arquivo de configuração do Spring Security, existe ao menos mais um elemento em aplicações web que geralmente é afetado pelas políticas de acesso:

os menus. Para evitar que os usuários tenham conhecimento das seções da aplicação às quais eles não possuem acesso, escondemos esses itens nos menus usando as mesmas regras utilizadas na configuração de acessos. O menu JSF apresentado na **Listagem 5**, por exemplo, segue as mesmas regras apresentadas na **Listagem 4** para definir quais links são apresentados. Para melhor comprehendê-lo, basta lembrar que a tag JSF **commandLink** gera um link (tag HTML **a**) para o destino definido no

Autorizações dinâmicas com Spring Security

parâmetro **action**, renderizado somente se a condição presente no parâmetro **rendered** for verdadeira. Essa condição, nos casos apresentados, utiliza o método **isUserInRole()** – disponibilizado na requisição (**request**) – que retorna verdadeiro se há um usuário autenticado e ele possui a autorização passada como parâmetro.

Listagem 5. Menu dinâmico que apresenta links de acordo com as autorizações do usuário.

```
01 <h:commandLink id="produto" value="Produto" immediate="true"
02 action="/seguro/produto/produto?faces-redirect=true"
03 rendered="#{request.isUserInRole('ROLE_GERENTE')}
04 or request.isUserInRole('ROLE_ADMIN')"/>
05 <h:commandLink id="consulta" value="Consulta" immediate="true"
06 action="/seguro/consulta/consulta?faces-redirect=true"
07 rendered="#{request.isUserInRole('ROLE_VENDEDOR')}
08 or request.isUserInRole('ROLE_ADMIN')"/>
09 <h:commandLink id="venda" value="Venda" immediate="true"
10 action="/seguro/venda/venda?faces-redirect=true"
11 rendered="#{request.isUserInRole('ROLE_VENDEDOR')}
12 or request.isUserInRole('ROLE_GERENTE')
13 or request.isUserInRole('ROLE_ADMIN')"/>
14 <h:commandLink id="usuario" value="Usuário" immediate="true"
15 action="/seguro/usuario/usuario?faces-redirect=true"
16 rendered="#{request.isUserInRole('ROLE_ADMIN')"/>
17 <h:commandLink id="perfil" value="Perfil" immediate="true"
18 action="/seguro/perfil/perfil?faces-redirect=true"
19 rendered="#{request.isUserInRole('ROLE_ADMIN')}/>
```

No caso discutido anteriormente, onde usuários com a autorização **ROLE_GERENTE** também precisem acessar URLs que sigam o padrão **"/seguro/venda/**"**, precisaríamos agregar manualmente a autorização **ROLE_GERENTE** nas condições de renderização do componente **commandLink** que inicia na linha 5 do menu. No entanto, o que aconteceria se, por esquecimento, isso não ocorresse? Simples, os usuários com a autorização poderiam acessar a página diretamente, mas não visualizariam o link no menu da aplicação. Encontramos aqui a segunda limitação dessa modelagem: a revisão de políticas de acesso pode causar inconsistências na aplicação, por conta de redundâncias não controladas.

Uma modelagem mais flexível

Apresentadas as limitações da modelagem tradicional, exploraremos agora algumas modificações capazes de aumentar sua flexibilidade utilizando apenas as customizações disponibilizadas pelo Spring Security.

Iniciaremos as alterações com uma nova definição: toda autorização será associada a uma única funcionalidade. Com essa alteração, cada autorização deixa de representar um perfil de acesso, com várias funcionalidades associadas, e passa a representar uma única funcionalidade. Essa relação “um para um” entre autorização e funcionalidade faz com que somente seja preciso modificar a configuração ou um menu se uma funcionalidade for incluída ou excluída da aplicação, momento em que será preciso incluir ou excluir a autorização e link correspondentes. A configuração atualizada pode ser verificada na **Listagem 6**, com as novas autorizações **ROLE_PERFIL**, **ROLE_PRODUTO**, **ROLE_USUARIO**, **ROLE_VENDA** e **ROLE_CONSULTA** associadas

às suas respectivas funcionalidades. De forma análoga, o menu revisado, apresentado na **Listagem 7**, também apresenta seus links de acordo com as novas autorizações definidas.

Listagem 6. Exemplo revisado de configuração do Spring Security.

```
01 <sec:intercept-url pattern="/seguro/perfil/**" access="ROLE_PERFIL"/>
02 <sec:intercept-url pattern="/seguro/produto/**" access="ROLE_PRODUTO"/>
03 <sec:intercept-url pattern="/seguro/usuario/**" access="ROLE_USUARIO"/>
04 <sec:intercept-url pattern="/seguro/venda/**" access="ROLE_VENDA"/>
05 <sec:intercept-url pattern="/seguro/consulta/**" access="ROLE_CONSULTA"/>
```

Listagem 7. Menu dinâmico revisado.

```
01 <h:commandLink id="produto" value="Produto" immediate="true"
02 action="/seguro/produto/produto?faces-redirect=true"
03 rendered="#{request.isUserInRole('ROLE_PRODUTO')"/>
04 <h:commandLink id="consulta" value="Consulta" immediate="true"
05 action="/seguro/consulta/consulta?faces-redirect=true"
06 rendered="#{request.isUserInRole('ROLE_CONSULTA')"/>
07 <h:commandLink id="venda" value="Venda" immediate="true"
08 action="/seguro/venda/venda?faces-redirect=true"
09 rendered="#{request.isUserInRole('ROLE_VENDA')"/>
10 <h:commandLink id="usuario" value="Usuário" immediate="true"
11 action="/seguro/usuario/usuario?faces-redirect=true"
12 rendered="#{request.isUserInRole('ROLE_USUARIO')"/>
13 <h:commandLink id="perfil" value="Perfil" immediate="true"
14 action="/seguro/perfil/perfil?faces-redirect=true"
15 rendered="#{request.isUserInRole('ROLE_PERFIL')}/>
```

À primeira vista essa pequena modificação parece eliminar as limitações anteriormente apresentadas, permitindo alterar livremente as políticas de segurança sem a necessidade de codificação, pois é possível associar ou dissociar funcionalidades individuais da aplicação a qualquer usuário. Infelizmente, essa definição apenas substitui as limitações antigas por uma nova: agora todo usuário precisa receber acesso individual a cada uma das funcionalidades que acessa e isso pode se tornar extremamente trabalhoso de manter em aplicações grandes, com dezenas de funcionalidades. Além disso, seria preciso manter um controle paralelo para garantir que usuários com as mesmas funções possuam os mesmos acessos.

Os problemas causados mostram que poder agrupar diversas funcionalidades sob uma única autorização possui certas vantagens, pois, em geral, usuários pertencem a grupos com necessidades de acesso idênticas. Para restaurar essa importante característica perdida, realizaremos uma segunda modificação, na qual agregaremos uma nova estrutura de dados, denominada **Perfil**, ao modelo, como mostra a **Figura 3**.

Essa nova estrutura representa um perfil de acesso e agrupa um conjunto de autorizações relacionadas. Com isso, eliminamos a última limitação e o usuário volta a se associar a grupos de autorizações. Ao contrário da modelagem tradicional, entretanto, nosso modelo é mais flexível e permite que os grupos de autorizações (perfis), por se tratarem de entidades, possam ser criados ou excluídos dinamicamente. Ou seja, mantivemos as características originais, eliminamos as limitações e ainda agregamos uma nova característica.

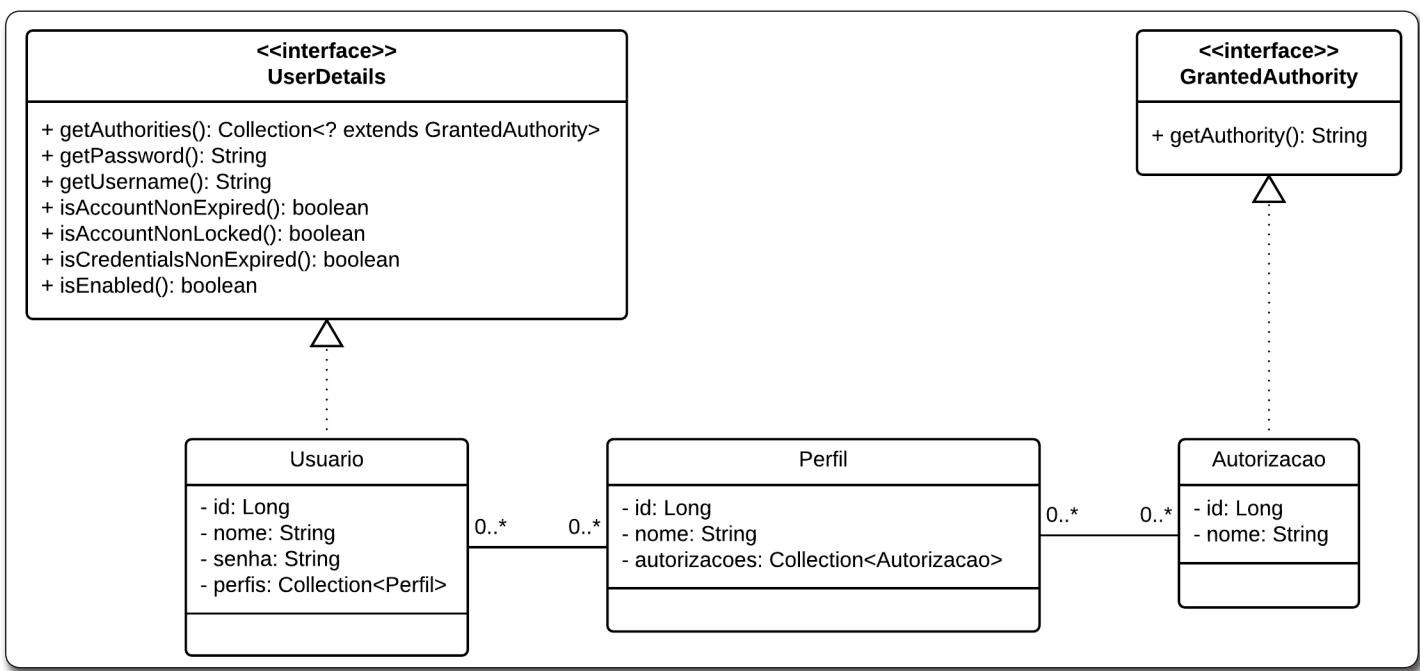


Figura 3. Diagrama de classes revisado

Uma aplicação com definição dinâmica de perfis de acesso

Para demonstrar o que foi citado até aqui na prática, acompanharemos, a partir de agora, o desenvolvimento de uma pequena aplicação web que possui como intuito exemplificar a nova modelagem apresentada. Para melhor compreensão dividiremos nossa aplicação nas seguintes partes: banco de dados, persistência, serviços e apresentação. A Figura 4 apresenta a forma como elas se relacionam.

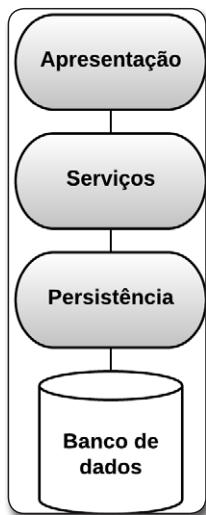


Figura 4. Estrutura da aplicação web

Nas seções a seguir detalharemos os principais elementos de cada uma das partes da aplicação. Para melhor compreensão, recomenda-se baixar o código-fonte completo, que consiste em um projeto Maven/Eclipse e se encontra disponível na área de downloads desta edição.

Banco de dados

Vamos começar a exploração de nossa aplicação pelo banco de dados. Ele é composto pelas tabelas **USR_USUARIO**, **PER_PERFIL**, **AUT_AUTORIZACAO**, **USP_USUARIO_PERFIL** e **PEA_PERFIL_AUTORIZACAO**, conforme o modelo lógico apresentado na Figura 5, e espelha o diagrama de classes apresentado na Figura 3.

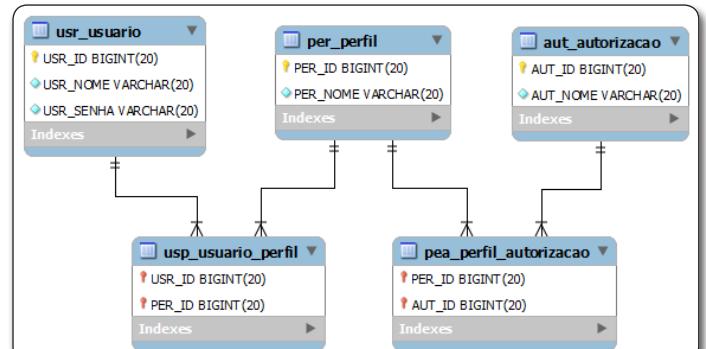


Figura 5. Modelo de dados da aplicação

Nesse modelo, a tabela **USR_USUARIO** corresponde à classe **Usuario**, **PER_PERFIL** corresponde à classe **Perfil** e **AUT_AUTORIZACAO** corresponde à classe **Autorizacao**. A tabela **USP_USUARIO_PERFIL**, por outro lado, corresponde ao relacionamento do tipo “muitos para muitos” entre as classes **Usuario** e **Perfil**, onde cada usuário pode possuir vários perfis de acesso e cada perfil de acesso pode estar associado a vários usuários. De forma análoga, **PEA_PERFIL_AUTORIZACAO** representa o relacionamento do tipo “muitos para muitos” entre as classes **Perfil** e **Autorizacao**.

Autorizações dinâmicas com Spring Security

Nota

Encontra-se disponível na pasta “script” do projeto um arquivo com comandos SQL, compatíveis com o SGBD MySQL, para criação das tabelas e inclusão de um conjunto mínimo de registros, contendo autorizações, perfis de acesso e usuários.

Persistência

Passemos agora à aplicação propriamente dita, iniciando pela camada de persistência, que contém as entidades (classes persistidas), responsáveis por mapear as tabelas do banco de dados, e os repositórios, responsáveis por realizar a persistência de dados e prover consultas diversas.

Comecemos pelas classes persistidas, verificando a forma como elas mapeiam suas tabelas correspondentes e interagem com as interfaces do Spring Security.

A primeira classe persistida, **Usuario**, é apresentada na **Listagem 8**. Ela mapeia a tabela **USR_USUARIO**, por meio de anotações JPA, e implementa a interface **UserDetails**.

Para melhor compreendê-la, analisemos suas anotações:

- A anotação **@Entity** (linha 1) indica que a classe **Usuario** é persistente;
- A anotação **@Table** (linha 2) indica que o nome da tabela correspondente não é igual ao nome da classe (padrão), mas sim **USR_USUARIO**;
- As anotações **@Id** (linha 10) e **@GeneratedValue** (linha 11) estão associadas ao atributo **id** e indicam que ele é o identificador da tabela e que seus valores devem ser gerados automaticamente (parâmetro **strategy** com valor **GenerationType.IDENTITY**);
- As anotações **@Column** (linhas 15 e 18) mapeiam, de forma análoga a **@Table**, os atributos **nome** e **senha** às colunas **USR_NOME** e **USR_SENHA**, além de permitir a definição de algumas restrições, como tamanho (**length**) e se aceita valores nulos (**nullable**) ou repetidos (**unique**);
- Por último, temos as anotações **@ManyToMany** (linha 21), **@JoinTable** (linha 22) e **@LazyCollection** (linha 25), que mapeiam o relacionamento entre **Usuario** e **Perfil** no atributo **perfis**:
 - A anotação **@ManyToMany** indica que o relacionamento é do tipo “muitos para muitos”;
 - A anotação **@JoinTable** indica qual tabela possibilita o relacionamento (**USP_USUARIO_PERFIL**) e quais colunas dela estão associadas a cada classe;
 - A anotação **@LazyCollection** serve para indicar se os dados de **Perfil** devem ser recuperados junto com **Usuario** (como configurado com a opção **LazyCollectionOption.FALSE**) ou somente quando forem acessados.

Analisemos agora as implementações dos métodos da interface **UserDetails** do Spring Security:

- O método **getAuthorities()** – linha 29 – deve retornar uma coleção de objetos que implementem a interface **GrantedAuthority** (em nosso exemplo, a classe **Autorizacao**), representando as autorizações que o usuário possui. Contudo, como não existe associação direta entre **Usuario** e **Autorizacao**, precisamos

Listagem 8. Código da classe Usuario.

```
01 @Entity
02 @Table(name = "USR_USUARIO")
03 public class Usuario implements UserDetails{
04
05     /**
06      * Serial Version UID
07     */
08     private static final long serialVersionUID = -7275092339436075624L;
09
10    @Id
11    @GeneratedValue(strategy=GenerationType.IDENTITY)
12    @Column(name = "USR_ID")
13    private Long id;
14
15    @Column(name = "USR_NOME", unique=true, length = 20, nullable = false)
16    private String nome;
17
18    @Column(name = "USR_SENHA", length = 50, nullable = false)
19    private String senha;
20
21    @ManyToMany
22    @JoinTable(name = "USP_USUARIO_PERFIL",
23        joinColumns = { @JoinColumn(name = "USR_ID") },
24        inverseJoinColumns = { @JoinColumn(name = "PER_ID") })
25    @LazyCollection(LazyCollectionOption.FALSE)
26    private List<Perfil> perfis = new ArrayList<Perfil>();
27
28    @Override
29    public Collection<? extends GrantedAuthority> getAuthorities() {
30        Set<Autorizacao> autorizacoes = new HashSet<Autorizacao>();
31        if(perfis != null) {
32            for(Perfil perfil: perfis) {
33                autorizacoes.addAll(perfil.getAutorizacoes());
34            }
35        }
36        return autorizacoes;
37    }
38
39    @Override
40    public String getPassword() {
41        return this.senha;
42    }
43
44    @Override
45    public String getUsername() {
46        return this.nome;
47    }
48
49    @Override
50    public boolean isAccountNonExpired() {
51        return true;
52    }
53
54    @Override
55    public boolean isAccountNonLocked() {
56        return true;
57    }
58
59    @Override
60    public boolean isCredentialsNonExpired() {
61        return true;
62    }
63
64    @Override
65    public boolean isEnabled() {
66        return true;
67    }
```

varrer todos os perfis de acesso e retornar o acumulado de todas as autorizações associadas. Dessa forma, apesar de utilizarmos um modelo de dados diferente, o Spring Security continua a receber os dados na estrutura original;

- O método `getPassword()` – linha 40 – deve retornar a senha do usuário e, por esse motivo, retorna o conteúdo do atributo `senha`;

- O método `getUsername()` – linha 45 – deve retornar um nome único, utilizado pelo usuário para se autenticar na aplicação, e, por esse motivo, retorna o conteúdo do atributo `nome`, que é único para cada registro, segundo o parâmetro `unique` (linha 15);

- Os demais métodos (`isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentialsNonExpired()` e `isEnabled()`) representam indicadores adicionais sobre o estado do usuário. Como informado anteriormente, eles não serão utilizados e, portanto, retornam sempre valor verdadeiro (`true`), sinalizando que a conta não se encontra expirada nem bloqueada, que as credenciais não expiram e que o usuário está habilitado a utilizar a aplicação.

Continuando nossa análise das classes, passemos à nova classe **Perfil**, apresentada na **Listagem 9**. Ela é basicamente um POJO (*Plain Old Java Object*), pois possui apenas atributos e os métodos get e set correspondentes. Suas anotações atuam de forma análoga às da classe **Usuario**, sendo que o relacionamento do tipo “muitos para muitos” com a classe **Autorizacao** é mapeado no atributo `autorizacoes` e corresponde à tabela **PEA_PERFIL_AUTORIZACAO** (linha 19).

Listagem 9. Código da classe Perfil.

```
01 @Entity
02 @Table(name = "PER_PERFIL")
03 public class Perfil implements Serializable {
04
05     /**
06      * Serial Version UID
07     */
08     private static final long serialVersionUID = -4187201805156095411L;
09
10    @Id
11    @GeneratedValue(strategy=GenerationType.IDENTITY)
12    @Column(name = "PER_ID")
13    private Long id;
14
15    @Column(name = "PER_NOME", unique=true, length = 20, nullable = false)
16    private String nome;
17
18    @ManyToMany
19    @JoinTable(name = "PEA_PERFIL_AUTORIZACAO",
20        joinColumns = { @JoinColumn(name = "PER_ID") },
21        inverseJoinColumns = { @JoinColumn(name = "AUT_ID") })
22    @LazyCollection(LazyCollectionOption.FALSE)
23    private List<Autorizacao> autorizacoes = new ArrayList<Autorizacao>();
```

Finalmente, temos a classe **Autorizacao**, apresentada na **Listagem 10**, que, assim como a classe **Usuario**, implementa uma interface (**GrantedAuthority**) do Spring Security. Suas anotações atuam de forma análoga às das classes apresentadas anteriormente. Além disso, ela implementa, na linha 19, o método

`getAuthority()`, proveniente da interface, que deve retornar um texto identificador da autorização. Para esse fim, esse método retorna o conteúdo do atributo `nome`, que apresenta um valor único para cada registro, segundo o parâmetro `unique` (linha 15).

Listagem 10. Código da classe Autorizacao.

```
01 @Entity
02 @Table(name = "AUT_AUTORIZACAO")
03 public class Autorizacao implements GrantedAuthority{
04
05     /**
06      * Serial Version UID
07     */
08     private static final long serialVersionUID = 2084366362705923518L;
09
10    @Id
11    @GeneratedValue(strategy=GenerationType.IDENTITY)
12    @Column(name = "AUT_ID")
13    private Long id;
14
15    @Column(name = "AUT_NOME", unique=true, length = 20, nullable = false)
16    private String nome;
17
18    @Override
19    public String getAuthority() {
20        return this.nome;
21    }
```

Repositórios

Passemos agora aos repositórios, criados com o auxílio do framework Spring Data JPA, que elimina a necessidade de escrever qualquer código para realização da persistência e provê facilitadores, como query methods (veja o **BOX 3**) e a anotação `@Query`, para a criação de consultas.

Em nossa aplicação, cada classe persistida (**Usuario**, **Perfil** e **Autorizacao**) possui um repositório próprio. Para a criação desses, basta criar uma nova interface que estenda uma das interfaces fornecidas pelo Spring Data JPA. Como necessitaremos apenas das operações básicas de CRUD (*Create, Read, Update, and Delete*), utilizaremos a interface **org.springframework.data.repository.CrudRepository**.

Na **Listagem 11** podemos visualizar o repositório correspondente à classe **Usuario**. Como indicado anteriormente, ele estende a interface **CrudRepository**, que recebe dois parâmetros: o primeiro corresponde à classe associada (**Usuario**) e o segundo ao tipo do atributo identificador da classe (**Long**). Note que nosso repositório possui também dois query methods: o primeiro (linha 8) busca o usuário que possua o nome idêntico ao parâmetro recebido; e o segundo (linha 15) busca todos os usuários que possuam, em qualquer parte de seus nomes, o texto passado como parâmetro, sem diferenciar letras maiúsculas e minúsculas. A principal vantagem do uso do Spring Data reside no fato de não ser preciso criar classes concretas que implementem essas interfaces. Basta utilizá-las diretamente.

Os repositórios correspondentes às classes **Perfil** e **Autorizacao** seguem o mesmo princípio e se chamam **PerfilRepository** e **AutorizacaoRepository**, respectivamente.

Autorizações dinâmicas com Spring Security

BOX 3. Query methods

O Spring Data possui uma infraestrutura para repositórios capaz de analisar os nomes dos métodos e gerar consultas (queries) automaticamente.

Para viabilizar esse processo, existe, obviamente, uma estrutura que os nomes dos métodos devem seguir. Primeiramente, eles devem utilizar os prefixos **find...By, read...By, query...By, count...By** ou **get...By**. A primeira palavra reservada (**find, read, query, count** ou **get**) define o tipo de operação a realizar com os registros que satisfazem os critérios da consulta, sendo que **count** retorna um valor inteiro único, correspondente à quantidade de registros, enquanto as demais retornam os registros propriamente ditos. Entre essa primeira palavra reservada e a segunda (**By**) podem ser adicionados modificadores (como **Distinct, Top e Last**) que afetam a quantidade ou formato dos resultados retornados. Finalmente, após a segunda palavra reservada devem aparecer as condições da consulta (como, por exemplo, **NotNullNome**, que restringe o resultado aos registros que possuem o atributo denominado **nome** não nulo) e, opcionalmente, a ordenação dos resultados. Essas condições podem ser concatenadas com o uso das palavras reservadas **And** (operação lógica **E**) ou **Or** (operação lógica **Ou**).

Agora que já fomos apresentados à estrutura do nome de um query method, vamos falar de seus parâmetros. Cada um deles é associado a um dos nomes de atributos mencionados na condição, seguindo a ordem de apresentação. Ou seja, o primeiro parâmetro é associado ao primeiro atributo mencionado na condição e assim por diante. Ademais, a forma como cada parâmetro é comparado com o atributo varia de acordo com o comparador colocado logo após o nome do atributo. Em caso de nenhum comparador ser mencionado, subentende-se que a comparação a realizar é a de igualdade.

Enfim, query methods são excelentes ferramentas para se criar consultas simples, contudo, consultas complexas inevitavelmente resultarão em métodos com nomes gigantescos e ilegíveis. Nesses casos, recomenda-se a utilização da anotação **@Query**, que aceita comandos em JPQL ou SQL nativo.

Listagem 11. Repositório da classe de entidade Usuario.

```
01 public interface UsuarioRepository extends CrudRepository<Usuario, Long>{  
02  
03     /**  
04      * Busca usuario com nome especificado  
05      * @param nome Nome  
06      * @return Usuario  
07     */  
08     public Usuario findByNome(String nome);  
09  
10    /**  
11      * Busca usuarios com nome similar ao especificado  
12      * @param nome Nome  
13      * @return List<Usuario>  
14     */  
15     public List<Usuario> findByNomeContainingIgnoreCase(String nome);  
16  
17 }
```

Serviços

Com o banco de dados criado e a camada de persistência pronta, abordaremos agora a camada de serviços. Nela são implementadas as regras de negócio utilizadas na camada de apresentação de nossa aplicação.

O primeiro serviço que analisaremos se encontra na classe **SegurancaServiceImpl**, apresentada na **Listagem 12**. Esse serviço é chamado pelo Spring Security durante o processo de autenticação de um usuário e, para que isso seja possível, ele precisa implementar a interface **org.springframework.security.core.userdetails.UserDetailsService**. Essa interface define apenas o

método **loadUserByUsername()** – linha 12 – que busca um usuário com nome idêntico ao parâmetro recebido. Para implementá-lo, note que utilizamos o query method **findOne()** de **UsuarioRepository**.

A forma como devemos configurar o Spring Security para que ele utilize esse serviço será apresentada ao final, quando discutirmos as configurações referentes à camada de apresentação.

Listagem 12. Gerenciador de autenticação – código da classe SegurancaServiceImpl.

```
01 @Service("segurancaService")  
02 public class SegurancaServiceImpl implements UserDetailsService {  
03  
04     @Autowired  
05     private UsuarioRepository usuarioRepository;  
06  
07     public void setUsuarioRepository(UsuarioRepository usuarioDao) {  
08         this.usuarioRepository = usuarioDao;  
09     }  
10  
11     @Override  
12     public UserDetails loadUserByUsername(String userName)  
throws UsernameNotFoundException {  
13         Usuario usuario = usuarioRepository.findOne(userName);  
14         if(usuario == null){  
15             throw new UsernameNotFoundException("Usuário não encontrado!");  
16         }  
17         return usuario;  
18     }  
19  
20 }
```

Nesse código, a anotação **@Service** (linha 1) indica que essa classe é um bean gerenciado pelo Spring que se encontra na camada de serviços. O parâmetro **segurancaService** representa o nome a ser utilizado pelo bean. Na linha 4 temos a anotação **@Autowired**, que realiza a injecção de dependência no atributo **usuarioRepository**, ou seja, ela indica ao Spring que um bean que implemente a interface **UsuarioRepository** deve ser carregado no atributo **usuarioRepository** por meio de seu setter – que se encontra na linha 7.

A aplicação conta ainda com duas outras classes de serviços, **UsuarioServiceImpl** e **PerfilServiceImpl**, que disponibilizam alguns dos métodos dos repositórios de usuários e perfis para utilização na camada de apresentação. No entanto, não as analisaremos pelo fato do código utilizar recursos semelhantes.

Configurações

Antes de seguirmos para a próxima camada, vamos analisar todas as configurações necessárias à nossa aplicação. Verificaremos inicialmente as configurações do Spring Framework, que cobrem todo o back-end de nossa aplicação, e depois partiremos para as demais configurações, cujo entendimento é necessário para que possamos melhor compreender os componentes da apresentação que as referenciam.

As configurações do Spring Framework, contidas no arquivo **applicationContext.xml**, apresentado em sua totalidade na **Listagem 13**, são analisadas a seguir:

- A tag **component-scan** (linha 17) indica, por meio de seu parâmetro **base-package**, qual pacote contém as classes que possuem anotações reconhecidas pelo Spring Framework;
- A tag **repositories** (linha 19) indica, por meio de seu parâmetro **base-package**, qual pacote contém os repositórios gerenciados pelo Spring Data JPA;
- A tag **bean** (linha 22) instancia um **DataSource**, que contém as configurações referentes à conexão com o SGBD. Suas propriedades devem ser alteradas de acordo com a instância de SGBD local;

Listagem 13. Conteúdo do arquivo *applicationContext.xml*.

```

01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03   xmlns:p="http://www.springframework.org/schema/p"
04   xmlns:tx="http://www.springframework.org/schema/tx"
05   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
06   xmlns:context="http://www.springframework.org/schema/context"
07   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
08   xsi:schemaLocation="http://www.springframework.org/schema/beans
09     http://www.springframework.org/schema/beans/spring-beans.xsd
10    http://www.springframework.org/schema/tx
11    http://www.springframework.org/schema/tx/spring-tx.xsd
12    http://www.springframework.org/schema/data/jpa
13    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
14    http://www.springframework.org/schema/context
15    http://www.springframework.org/schema/context/spring-context.xsd">
16
17 <context:component-scan base-package="br.com.javamagazine"/>
18
19 <jpa:repositories base-package="br.com.javamagazine.repository"/>
20
21 <!-- DataSource -->
22 <bean id="dataSource"
23   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
24   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
25   <property name="url" value="jdbc:mysql://localhost:3306/spring"/>
26   <property name="username" value="root"/>
27   <property name="password" value="adm1n"/>
28 </bean>
29
30 <bean id="entityManagerFactory"
31   class="org.springframework.orm.jpa.LocalContainerEntityManager
32   FactoryBean"
33   p:packagesToScan="br.com.javamagazine.model"
34   p:dataSource-ref="dataSource"
35   >
36   <property name="jpaVendorAdapter">
37     <bean class="org.springframework.orm.jpa.vendor.HibernateJpa
38       VendorAdapter">
39       <property name="generateDdl" value="false"/>
40       <property name="showSql" value="true"/>
41       <property name="databasePlatform" value="org.hibernate.dialect.
42         MySQLDialect"/>
43     </bean>
44   </property>
45 </bean>
46
47 <bean id="transactionManager"
48   class="org.springframework.orm.jpa.JpaTransactionManager">
49   <property name="entityManagerFactory" ref="entityManagerFactory"/>
50 </bean>
51
52 </beans>

```

- A segunda tag **bean** (linha 30) define um **EntityManagerFactory**, utilizado pelo JPA para persistência de dados. Seu parâmetro **packagesToScan** informa o pacote que contém as classes de persistência (**Usuario**, **Perfil** e **Autorizacao** em nossa aplicação);
- A última tag **bean** (linha 44) define um **TransactionManager**, responsável por gerenciar as transações de banco de dados;

Passaremos agora aos demais arquivos de configuração, referentes à camada de apresentação, ou front-end, de nossa aplicação. O arquivo de configuração principal dessa camada é o *web.xml*, apresentado na **Listagem 14**. Vejamos a sua composição:

- Na linha 1 temos a tag raiz **web-app**, cujos elementos definem a aplicação web;
- Na linha 9, a tag opcional **display-name** especifica um nome para a aplicação, visualizado na GUI de algumas ferramentas de desenvolvimento;
- Entre as linhas 12 e 21 configuramos o suporte ao Spring Framework. Inicialmente configuramos **ContextLoaderListener** como um listener (linha 12) de nossa aplicação. Ele é responsável por criar o **ApplicationContext** com base no parâmetro **contextConfigLocation**, definido logo em seguida (linha 18). Esse parâmetro carrega todos os arquivos de configuração que sigam o padrão especificado. Em nosso caso, serão carregados os arquivos *applicationContext.xml*, que contém as configurações gerais do Spring Framework, e *applicationContext-security.xml*, que contém as configurações exclusivas do Spring Security;
- Entre as linhas 24 e 35 definimos o Spring Security como um filtro de nossa aplicação. Na linha 32, especificamente, indicamos que ele será acionado para todas as requisições direcionadas à nossa aplicação;
- Entre as linhas 38 e 40 sinalizamos o arquivo *index.xhtml* como a página inicial de nossa aplicação, por meio da tag **welcome-file**;
- Por fim, entre as linhas 44 e 63, informamos o servlet do JSF, responsável por processar as páginas suportadas por esse framework, cujas URLs seguem os padrões “/faces/”, “*.xhtml” ou “*.jsf”.

O segundo arquivo de configuração da camada de apresentação a ser analisado é o *faces-config.xml*, visualizado na **Listagem 15** e responsável pelas configurações referentes ao framework JSF. Além de sua tag principal **faces-config**, esse arquivo apenas define a classe **org.springframework.web.jsf.el.SpringBeanFacesELResolver** como a responsável por resolver Expression Languages (EL) do Spring no JSF. Ou seja, isso torna possível acessar todos os beans definidos no Spring em páginas e beans gerenciados pelo JSF.

Por fim, analisaremos o arquivo *applicationContext-security.xml*, apresentado na **Listagem 16**, que contém as configurações específicas do Spring Security e habilita nossa infraestrutura para definição dinâmica de permissões de acesso:

- A tag **beans** (linha 2) é a raiz desse arquivo de configuração e importa os schemas necessários para a análise de seu conteúdo;

Autorizações dinâmicas com Spring Security

Listagem 14. Conteúdo do arquivo web.xml.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xmlns="http://java.sun.com/xml/ns/javaee"
04   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
07   id="WebApp_ID" version="3.0">
08
09 <display-name>Autorizações Dinâmicas</display-name>
10
11 <!-- Suporte ao Spring -->
12 <listener>
13   <listener-class>
14     org.springframework.web.context.ContextLoaderListener
15   </listener-class>
16 </listener>
17
18 <context-param>
19   <param-name>contextConfigLocation</param-name>
20   <param-value>/WEB-INF/applicationContext*.xml</param-value>
21 </context-param>
22
23 <!-- Spring Security -->
24 <filter>
25   <filter-name>springSecurityFilterChain</filter-name>
26   <filter-class>
27     org.springframework.web.filter.DelegatingFilterProxy
28   </filter-class>
29 </filter>
30 <filter-mapping>
31   <filter-name>springSecurityFilterChain</filter-name>
32   <url-pattern>*</url-pattern>
33   <dispatcher>FORWARD</dispatcher>
34   <dispatcher>REQUEST</dispatcher>
35 </filter-mapping>
36
37 <!-- Página inicial -->
38 <welcome-file-list>
39   <welcome-file>index.xhtml</welcome-file>
40 </welcome-file-list>
41
42
43 <!-- JSF -->
44 <servlet>
45   <servlet-name>Faces Servlet</servlet-name>
46   <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
47   <load-on-startup>1</load-on-startup>
48 </servlet>
49
50 <servlet-mapping>
51   <servlet-name>Faces Servlet</servlet-name>
52   <url-pattern>/faces/*</url-pattern>
53 </servlet-mapping>
54
55 <servlet-mapping>
56   <servlet-name>Faces Servlet</servlet-name>
57   <url-pattern>*.xhtml</url-pattern>
58 </servlet-mapping>
59
60 <servlet-mapping>
61   <servlet-name>Faces Servlet</servlet-name>
62   <url-pattern>*.jsf</url-pattern>
63 </servlet-mapping>
64
65 </web-app>
```

Listagem 15. Conteúdo do arquivo faces-config.xml.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
05     http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
06   version="2.2">
07
08 <application>
09   <el-resolver>
10     org.springframework.web.jsf.el.SpringBeanFacesELResolver
11   </el-resolver>
12 </application>
13
14 </faces-config>
```

- Na linha 12, temos a tag **global-method-security**, que habilita as anotações do Spring Security e assim permite proteger métodos por meio da verificação de autorizações;
- A tag **http** (linha 14) contém todas as configurações referentes à segurança da aplicação. Seu parâmetro **auto-config** com valor verdadeiro habilita um conjunto de configurações básicas pré-definidas, enquanto seu parâmetro **use-expressions** com valor falso indica que desejamos utilizar texto plano, ao invés de EL, nas configurações;
- As tags **intercept-url** (linhas 17 a 20) foram explicadas anteriormente quando apresentamos o modelo tradicional e aqui associam cada padrão com uma autorização única, seguindo a nova modelagem proposta;
- Já a tag **form-login** permite substituir a página de autenticação de usuários (login) padrão por uma personalizada, que pode apresentar um design mais integrado à aplicação. O endereço dessa página é definido no parâmetro obrigatório **login-page**. Já o parâmetro opcional **login-processing-url** possibilita alterar a URL utilizada pelo Spring Security para realizar o processo de login (por padrão, "/login"), enquanto os parâmetros **username-parameter** e **password-parameter** viabilizam alterar os nomes dos parâmetros que contêm o nome de usuário e a senha (por padrão, "username" e "password", respectivamente). Finalmente, o parâmetro **authentication-failure-url** permite definir qual URL será acessada quando da falha no processo de autenticação (por padrão, "/login?error"). Em nosso caso, redirecionamos para a própria página de login, enviando um parâmetro **erro** com valor verdadeiro.

Nota

A partir da versão 4 do Spring Security, os nomes padrões para os elementos da página de login foram alterados. Portanto, ao realizar uma migração, é necessário realizar alterações na página de login customizada ou configurar os parâmetros **login-processing-url**, **username-parameter** e **password-parameter** com os nomes utilizados pela versão de origem.

- De forma análoga à tag **login**, a tag **logout** (linha 30) possibilita redefinir as operações realizadas durante o logout da aplicação. Em nosso caso, configuramos que a sessão deve ser invalidada

(parâmetro **invalidate-session** com valor verdadeiro), os cookies excluídos (parâmetro **delete-cookies** com valor verdadeiro) e o fluxo redirecionado para a página inicial (parâmetro **logout-success-url** com valor “/index.xhtml”);

- A última tag de configuração de segurança, **csrf** (linha 34), permite habilitar ou desabilitar a proteção contra CSRF do Spring Security. Pelo fato de utilizarmos o framework JSF, que já implementa essa proteção, para a camada de apresentação, optamos por desabilitá-la (parâmetro **disabled** com valor verdadeiro).

Nota

A partir da versão 4 do Spring Security, a proteção contra CSRF é ativada por padrão. Se a proteção não for manualmente desativada e as páginas não definirem um token CSRF, ocorrerão erros durante a submissão de formulários.

- Por fim, na tag **authentication-manager** (linha 38) configuramos os detalhes do processo de gerenciamento de autenticações. Em sua tag interna **authentication-provider** apontamos nosso serviço de autenticação, **segurancaService**, como provedor de autenticação, por meio do parâmetro **user-service-ref**. Também definimos a criptografia a utilizar nas senhas como MD5, por meio do parâmetro **hash** da tag **password-encoder**. A utilização desse tipo de criptografia impede que seja possível descobrir a senha de um usuário ao invadir o banco de dados ou interceptar as informações de autenticação.

Apresentação

A última camada de nossa aplicação contém as páginas web responsáveis pela interação com os usuários. Para construí-las, utilizaremos o framework JavaServer Faces (JSF) 2.2.9.

De forma a manter uma experiência uniforme de navegação, definiremos um layout geral para a aplicação, que define um estilo único, uma identidade visual, a ser seguido por todas as páginas. Ele se encontra disponível no arquivo *layout.xhtml* e pode ser visualizado na **Figura 6**, onde A corresponde ao menu lateral dinâmico, que apresenta os links de acordo com as

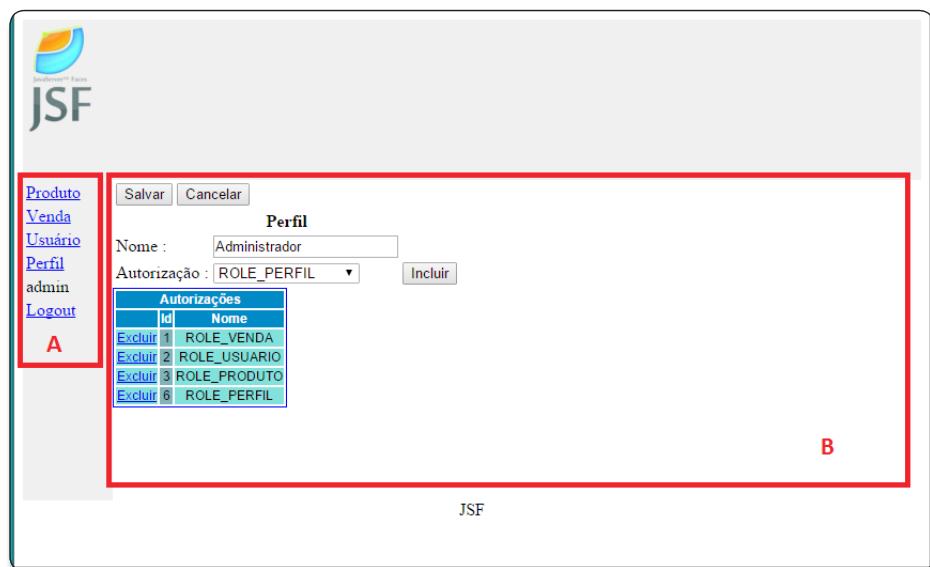


Figura 6. Layout geral da aplicação

Listagem 16. Conteúdo do arquivo *applicationContext-security.xml*.

```

01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans:beans
03   xmlns:sec="http://www.springframework.org/schema/security"
04   xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
05   xmlns:beans="http://www.springframework.org/schema/beans"
06   xsi:schemaLocation="http://www.springframework.org/schema/beans
07   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
08   http://www.springframework.org/schema/security
09   http://www.springframework.org/schema/security/spring-security.xsd">
10
11 <!-- Anotações -->
12 <sec:global-method-security pre-post-annotations="enabled"/>
13
14 <sec:http auto-config="true" use-expressions="false">
15
16 <!-- Restringe acesso a pasta seguro /seguro/** -->
17 <sec:intercept-url pattern="/seguro/perfil/**" access="ROLE_PERFIL"/>
18 <sec:intercept-url pattern="/seguro/produto/**" access="ROLE_PRODUTO"/>
19 <sec:intercept-url pattern="/seguro/usuario/**" access="ROLE_USUARIO"/>
20 <sec:intercept-url pattern="/seguro/venda/**" access="ROLE_VENDA"/>
21
22 <!-- Configurações da página de login -->
23 <sec:form-login login-page="/login.jsf"
24   login-processing-url="/j_spring_security_check"
25   username-parameter="j_username"
26   password-parameter="j_password"
27   authentication-failure-url="/login.jsf?erro=true"/>
28
29 <!-- Propriedades de logout -->
30 <sec:logout invalidate-session="true"
31   delete-cookies="true"
32   logout-success-url="/index.xhtml"/>
33
34 <sec:captcha disabled="true"/>
35 </sec:http>
36
37 <!-- Autenticador custom -->
38 <sec:authentication-manager alias="authenticationManager">
39   <sec:authentication-provider user-service-ref="segurancaService">
40     <sec:password-encoder hash="md5"/><sec:password-encoder>
41   </sec:authentication-provider>
42 </sec:authentication-manager>
43
44 </beans:beans>
```

Autorizações dinâmicas com Spring Security

autorizações do usuário autenticado, e **B** corresponde à área onde o conteúdo é apresentado.

Essa prática, de definir uma identidade visual, representa o motivo para a criação de nossa página de login personalizada, pois, apesar de o Spring Security fornecer uma página de login própria, ela não possui formatação alguma, destoando do restante da aplicação e prejudicando a experiência do usuário. Nossa página de

login, por outro lado, se integra ao layout, conforme pode ser visualizado na **Figura 7**. Para que ela substitua a página padrão, utilizamos a configuração descrita na seção anterior (vide análise do arquivo *applicationContext-security.xml*).

- Para melhor compreendermos o código da página de login, apresentado na **Listagem 17** e disponível no arquivo *login.xhtml*, revisemos cada uma de suas tags:

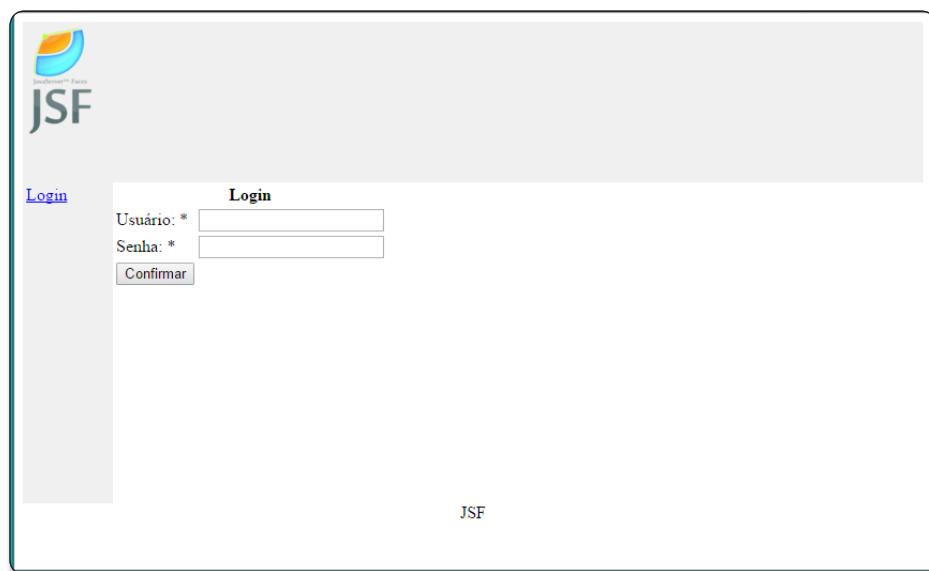


Figura 7. Página de login personalizada

Listagem 17. Código da página de login personalizada.

```
01 <?xml version="1.0" encoding="ISO-8859-1"?>
02 <ui:composition template="/layout/layout.xhtml"
03   xmlns:ui="http://java.sun.com/jsf/facelets"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:f="http://java.sun.com/jsf/core">
06
07   <ui:define name="principal">
08     <form action="j_spring_security_check" method="post">
09       <h:panelGroup id="login">
10         <h:outputText value="Usuário ou senha incorretos!">
11           rendered="#{param.error}" style="color: red;">
12         <h:panelGrid id="camposLogin" columns="2">
13           <f:facet name="header">
14             <h:outputText value="Login"/>
15           </f:facet>
16           <h:outputLabel for="j_username" value="Usuário: * />
17           <h:inputText id="j_username" required="true" label="Usuário"/>
18           <h:outputLabel for="j_password" value="Senha: * />
19           <h:inputSecret id="j_password" label="Senha" required="true"/>
20           <h:commandButton value="Confirmar"/>
21         </h:panelGrid>
22       </h:panelGroup>
23     </form>
24   </ui:define>
25 </ui:composition>
```

- A tag JSF principal desse arquivo, **composition** (linha 2), define, por meio de seu parâmetro **template**, que a página utilizará o layout contido no arquivo *layout.xhtml*. Ela também realiza a importação de bibliotecas, associando-as a prefixos (**ui**, **h** e **f**, nesse caso), para utilização de seus componentes na página;
- A próxima tag JSF, **define** (linha 7), especifica, por meio de seu parâmetro **name**, em qual seção do layout o conteúdo que ela contém deve ser incluído (nesse caso, o conteúdo será incluído na seção com nome “principal”);
- A tag HTML **form** (linha 8) engloba os campos de entrada que contêm as informações de autenticação (nome de usuário e senha). Em seu parâmetro **action**, que define a ação a realizar quando da submissão do formulário, configuramos a URL *j_spring_security_check*, disponibilizada pelo Spring Security para autenticação. Já em seu parâmetro **method**, que indica o método de envio dos dados, configuramos POST, por exigência do Spring Security. Essa tag HTML foi utilizada pela necessidade de se definir uma ação, o que não é possível na tag correspondente do JSF;
- A tag JSF **panelGroup** (linha 9) corresponde à tag HTML **span** e serve para organização de conteúdo;
- A tag JSF **outputText** (linha 10) emite uma mensagem de erro na cor vermelha. Ela apenas é renderizada se a página de login receber um parâmetro de nome **erro** com valor verdadeiro;
- Já a tag JSF **panelGrid** (linha 12) serve para organização de componentes e corresponde a uma tabela com duas colunas. Assim, cada par de componentes em seu interior é apresentado em uma linha distinta;
- A tag JSF **facet** (linha 13), por sua vez, permite alterar um elemento da tag que a contém. Nesse caso, ela altera o cabeçalho (header) do **panelGrid**, utilizando uma tag JSF **outputText** (linha 14) para exibir o texto “Login”;
- A tag JSF **outputLabel** (linha 16) é responsável por apresentar a etiqueta “Usuário: **”, que se encontra ao lado do campo de entrada que recebe o nome de usuário;

- Já a tag JSF **inputText** (linha 17) corresponde ao campo de entrada responsável por receber o nome de usuário destinado à autenticação. Ela possui um identificador (**id**) **j_username**, que, conforme vimos na configuração do Spring Security, está associado ao nome de usuário. Seu parâmetro **required** com valor verdadeiro indica que o formulário não será submetido enquanto o campo não possuir algum valor;
- De forma análoga à tag JSF **outputLabel** anterior, a localizada na linha 18 apresenta a etiqueta “Senha: **” associada ao campo de entrada que recebe a senha do usuário;
- Na linha 19, a tag JSF **inputSecret** provê um campo para entrada de texto próprio para senhas, cujo conteúdo não pode ser visualizado (todo caractere digitado é substituído por •). Ela possui o identificador (**id**) **j_password**, que, conforme vimos na configuração do Spring Security, está associado à senha do usuário. Seu parâmetro **required**, assim como no campo de entrada para nome, impede a submissão do formulário se nenhuma senha for fornecida;
- Por último, a tag JSF **commandButton** (linha 20) cria um botão com o texto “Confirmar” que submete o formulário, chamando a ação definida na tag HTML **form**.

Nesse ponto, já possuímos toda a infraestrutura necessária para a segurança de nossa aplicação, pois já contamos com meios para restringir o acesso às funcionalidades da aplicação, por meio das configurações constantes no arquivo *applicationContext-security.xml*, e autenticar um usuário, por meio da página de login. Entretanto, para que a aplicação se torne funcional, precisamos tratar de um elemento fundamental, presente na área A do layout geral: o menu da aplicação. Conforme explicação anterior, esse elemento está presente em todas as páginas da aplicação e disponibiliza um conteúdo dinâmico, baseado nas autorizações do usuário autenticado. Seu código é similar aos exemplos de menus apresentados durante este artigo, se encontra no arquivo *menu.xhtml* e pode ser visualizado na **Listagem 18**. Ele possui as seguintes tags JSF dignas de nota:

- A tag JSF **composition** (linha 2), como explicado anteriormente, define a utilização de um template e importa bibliotecas. No caso do menu, que faz parte do template da aplicação, ela apenas importa duas bibliotecas, associadas aos prefixos **ui** e **h**, que contêm os componentes utilizados em sua construção;
- A tag JSF **form** (linha 5) corresponde à tag HTML homônima e define um formulário que permite o envio de informações ao servidor;
- Já a tag JSF **panelGrid** (linha 6), também explicada anteriormente, cria uma tabela para organizar os componentes. Nesse caso, a tabela gerada possui uma coluna, fazendo com que somente um componente seja exibido por linha;
- As tags JSF **commandLink** foram explicadas anteriormente, quando discutimos a influência das políticas de segurança nos menus de uma aplicação. Em nosso menu elas servem para o mesmo propósito e utilizam os mesmos parâmetros já apresentados;
- A tag JSF **outputText** (linha 22) apresenta o nome do usuário

que se encontra autenticado na aplicação. Para tanto, ela utiliza as informações disponibilizadas no atributo **userPrincipal** da requisição (**request**). Essa tag somente é renderizada se existe um usuário autenticado, de acordo com a condição presente no parâmetro **rendered**;

- A tag JSF **outputLink** (linha 24) cria um link que aciona a URL de logout do Spring Security, cuja forma de operação foi discutida na seção anterior. Conforme a condição presente no parâmetro **rendered**, esse link somente é apresentado se existe um usuário autenticado.

Listagem 18. Código do menu da aplicação.

```

01 <?xml version="1.0" encoding="ISO-8859-1"?>
02 <ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
03   xmlns:h="http://java.sun.com/jsf/html">
04
05   <h:form id="mainMenu">
06     <h:panelGrid id="menu" columns="1">
07       <h:commandLink id="produto" value="Produto" immediate="true"
08         action="/seguro/produto/produto?faces-redirect=true"
09         rendered="#{request.isUserInRole('ROLE_PRODUTO')}"/>
10       <h:commandLink id="venda" value="Venda" immediate="true"
11         action="/seguro/venda/venda?faces-redirect=true"
12         rendered="#{request.isUserInRole('ROLE_VENDA')}"/>
13       <h:commandLink id="usuario" value="Usuário" immediate="true"
14         action="/seguro/usuario/usuario?faces-redirect=true"
15         rendered="#{request.isUserInRole('ROLE_USUARIO')}"/>
16       <h:commandLink id="perfil" value="Perfil" immediate="true"
17         action="/seguro/perfil/perfil?faces-redirect=true"
18         rendered="#{request.isUserInRole('ROLE_PERFIL')}"/>
19       <h:commandLink id="login" value="Login" immediate="true"
20         action="/login?faces-redirect=true"
21         rendered="#{empty request.userPrincipal}"/>
22       <h:outputText id="usuarioLogado" value="#{request.userPrincipal.name}"
23         rendered="#{not empty request.userPrincipal}"/>
24       <h:outputLink value="#{facesContext.getExternalContext.requestContextPath}
25         /logout"
26         rendered="#{not empty request.userPrincipal}"/>
27       <h:outputText value="Logout"/>
28     </h:panelGrid>
29   </h:form>
30
31 </ui:composition>

```

Finalizamos assim o nosso tour pelos principais elementos de cada uma das camadas da aplicação web. A aplicação completa conta com telas CRUD para o gerenciamento de usuários e perfis de acesso (apresentada na **Figura 6**). Essas funcionalidades foram incluídas para que possamos verificar os efeitos na aplicação causados por alterações nas políticas de segurança. Assim, ao alterar os perfis e realizar uma nova autenticação com algum dos usuários afetados, por exemplo, teremos a oportunidade de comprovar que nossa solução é capaz de alterar permissões de acesso dinamicamente.

A flexibilidade alcançada com a definição dinâmica de permissões traz consigo novas preocupações, do ponto de vista de segurança: é preciso se preocupar com quais usuários devem possuir esse poder e como monitorar as alterações realizadas nessas permissões.

Autorizações dinâmicas com Spring Security

O conteúdo deste artigo é apenas um primeiro passo em direção à solução completa, pois, idealmente, todas as tabelas envolvidas deveriam ser passíveis de auditoria, de forma a possibilitar o rastreamento das modificações realizadas e coibir operações com fins maliciosos. Essas funcionalidades adicionais poderiam ser alcançadas, por exemplo, com a utilização do recurso de auditoria do Spring Data JPA, para registrar data e usuário responsável por qualquer alteração no banco de dados, ou de uma solução mais completa, como o Hibernate Envers, que permite manter um histórico de todos os dados.

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Autor



Emanuel Mineda Carneiro

emanuel.mineda@fatec.sp.gov.br

Professor na FATEC São José dos Campos. Trabalha com desenvolvimento de software desde 2005. Bacharel em Ciência da Computação pela Universidade Federal de Itajubá (UNIFEI). Mestre em Ciências pelo Programa de Pós-Graduação em Engenharia Eletrônica e Computação do Instituto Tecnológico da Aeronáutica (ITA).



Links:

CERT.br

<http://www.cert.br/>

Spring Security.

<http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/>

Spring Data JPA.

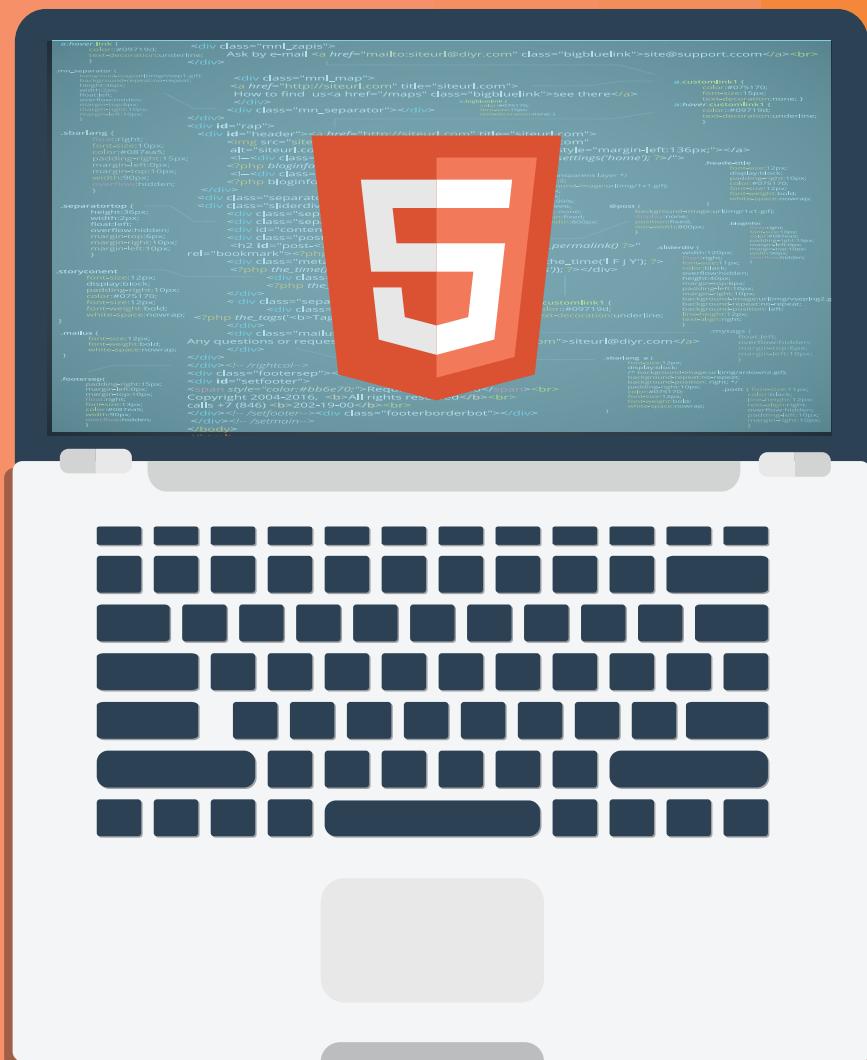
<http://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet.

https://www.owasp.org/index.php/CSRF_Prevention_Cheat_Sheet

Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!



DEVMEDIA

<http://www.devmedia.com.br/guias/guia-html/3>

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.

Supporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486