



Edição 55



Primeiros passos com o JavaFX 8
Conheça a API e suas novidades na prática

Hibernate Validator

Como validar objetos
e ter mais controle sobre
os dados de sua aplicação

INTRODUÇÃO A MICROSERVICES

Um novo caminho ao
desenvolvimento de aplicações

ISSN 2179625-4



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!

 **DEV**MEDIA



Edição 55 • 2015 • ISSN 2173625-4

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

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.



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

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 a 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 Easy 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

Sumário

04 - JavaFX 8: Uma introdução à arquitetura e às novidades da API

[Daniel Assunção Faria de Menezes]

Artigo no estilo Solução Completa

14 - Introdução à Arquitetura de Microservices com Spring Boot

[Marcos Alexandre Vidolin de Lima]

Conteúdo sobre Boas Práticas

24 - Hibernate Validator: como validar objetos e ter mais controle sobre os dados

[Carlos Alberto Silva]

JavaFX 8: Uma introdução à arquitetura e às novidades da API

Veja neste artigo um histórico sobre o JavaFX, os novos recursos da versão 8 e como utilizá-lo na prática

Desde a versão 1.2 do Java já existiam dois tipos de bibliotecas gráficas: AWT e Swing. A AWT foi a primeira API destinada a interfaces gráficas a ser criada e, mais tarde, foi superada pelo Swing, que trouxe diversos benefícios em relação a seu antecessor. Em sua grande maioria, as bibliotecas de criação de interfaces gráficas são simples de serem utilizadas. O obstáculo inicial fica por conta da familiarização com a sintaxe da linguagem e o grande leque de componentes disponíveis.

Com o intuito de agregar mais valor às interfaces de aplicativos desktop, o JavaFX foi anunciado em maio de 2007, no JavaOne, tendo sua primeira release lançada no dia 4 de dezembro de 2008. Nas versões iniciais, chamada de JavaFX Script, os desenvolvedores faziam uso de uma tipagem estática, verificada em tempo de compilação, junto a uma linguagem declarativa para construir as interfaces. Após a versão 2.0, a tecnologia passou a ser uma biblioteca do Java, podendo usufruir do código nativo para o desenvolvimento das aplicações, e a partir da versão 7 update 6, passou a ser distribuída nas instalações da linguagem, deixando de ser necessário baixar e instalar como uma solução externa.

A API JavaFX é usada para a criação de aplicações RIA (*Rich Internet Application*) que se comportam de maneira consistente em todas as plataformas, não sendo necessário recompilar o código fonte para ser executada em diferentes JVMs, desde que estas estejam na mesma versão na qual o código foi compilado. O termo RIA é usado para descrever Aplicações Ricas para a Internet, ou seja, aplicações que são executadas em ambiente web, mas que possuem características similares a softwares desenvolvidos para execução em ambientes desktop.

Fique por dentro

Este artigo apresentará o JavaFX, uma API do Java utilizada para a construção de interfaces ricas e de fácil usabilidade por parte do usuário. Essa promissora ferramenta conta com todos os recursos que a linguagem Java oferece, incluindo o fato de ser multiplataforma. Apresentaremos também as camadas da arquitetura do JavaFX, detalhando como cada uma contribui para a criação do layout da aplicação, além de mostrar as novidades da versão 8 e exemplos práticos de como usufruir dessa poderosa e versátil solução.

O desenvolvimento de aplicações em JavaFX pode fazer uso de qualquer biblioteca Java, acessar recursos nativos do sistema ou se conectar a aplicativos de middleware baseados em servidor. Essa possibilidade se dá pelo fato do JavaFX estar totalmente integrado ao *Java Runtime Environment* (JRE) e ao *Java Development Kit* (JDK), sendo capaz de executar aplicações que o adotem nas principais plataformas desktop (Linux, Mac OS X e Windows). Além disso, a partir da versão 8 as plataformas ARM também passaram a ser suportadas, aumentando a versatilidade da API com o uso em sistemas embarcados, como PDAs, dispositivos móveis ou máquinas com fins industriais.

Dentre os vários recursos disponíveis estão o uso de componentes nativos, gráficos 2D e 3D, efeitos, animações, músicas e vídeos, proporcionando uma vasta opção de funcionalidades para a construção do layout da aplicação. Para facilitar a criação e customização de interfaces, os desenvolvedores contam com o FXML, uma linguagem baseada em XML capaz de manipular componentes usando HTML e CSS.

Apesar de muitos desenvolvedores preferirem criar e customizar a interface através do FXML, outros acham mais fácil o uso de uma ferramenta gráfica. Pensando nisso, a Oracle disponibiliza

uma ferramenta bastante intuitiva, chamada *Scene Builder*, que auxilia a criação e a manutenção do código fonte do FXML, aumentando a produtividade e possibilitando a visualização das telas no decorrer do desenvolvimento.

O uso do *Scene Builder* é muito indicado em casos onde o desenvolvedor deseja conseguir um rápido feedback e/ou não está familiarizado com as tags FXML. A partir dele é gerado um código simples e claro, possível de alterar em qualquer editor de texto ou XML, não engessando o uso à ferramenta em outras etapas do projeto.

Neste artigo será apresentada uma visão geral da arquitetura da API JavaFX e seus principais componentes, detalhando-os e mostrando alguns exemplos na prática.

Nota

Desde o lançamento da versão 2.0 é possível integrar o Scene Builder às principais IDEs Java, como Eclipse, NetBeans e IntelliJ. Com esta ferramenta fica mais fácil utilizar os diversos componentes e controles nativos do JavaFX para composição das telas, além de possibilitar a inclusão de componentes de terceiros ou criar novos componentes.

A arquitetura

A arquitetura do JavaFX é composta por vários componentes que fornecem uma ampla variedade de funcionalidades, como gráficos, animações e recursos multimídia. Esses componentes disponibilizam uma maneira transparente e versátil de projetar interfaces ricas com diversos eventos, efeitos e diferentes modos de customização do layout.

As camadas da arquitetura são apresentadas na **Figura 1**. Nela podemos verificar os níveis que estão mais próximos ao usuário (mais acima), até as bibliotecas de renderização: Java 2D, OpenGL e D3D, mais próximas à JVM e aos recursos da máquina. Ademais, é importante notar que a API pública do Java (*JDK API Libraries & Tools*) está acessível a todos os níveis, facilitando seu uso por todos os recursos disponíveis.

Scene Graph

O *Scene Graph* é uma estrutura em árvore que representa a interface gráfica das aplicações JavaFX. Esta árvore é composta por nós, onde cada um representa um componente visual que compõe o layout. Um nó é formado por três atributos: um identificador, uma configuração de estilo e um limite de área. Além disso, cada nó pode ter apenas um pai e um ou mais filhos.

Essa arquitetura em hierarquia facilita o desenvolvimento das interfaces, já que ao aplicar um recurso ou uma propriedade de estilo em um nó, seus filhos herdarão essas configurações. Para exemplificar, podemos imaginar um painel que encapsula dois

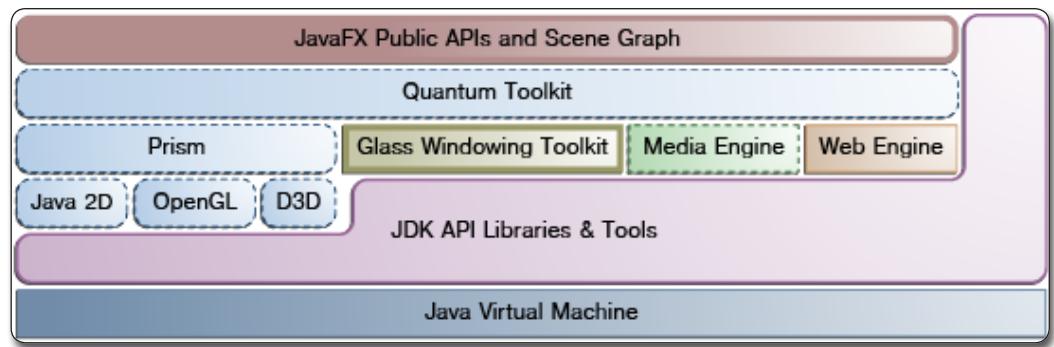


Figura 1. Arquitetura do JavaFX

botões. Diante disso, caso uma propriedade de estilo seja aplicada ao painel, os campos abaixo na hierarquia, neste caso os botões, irão receber as mesmas características atribuídas ao pai.

Para a customização da interface, os componentes podem receber listeners de eventos e efeitos. Dentre as possibilidades de efeitos, vale citar o *focus*, *blur* e *shadow*, e entre os eventos, podemos utilizar o *click* do mouse, a mudança de campo ou de valores, entre outros.

As APIs públicas do Java e suas ferramentas

Presente em todos os níveis da arquitetura, as APIs do JDK apoiam o desenvolvimento das aplicações, possibilitando liberdade e flexibilidade na construção de soluções *rich client*. Esta integração entre os melhores recursos do Java e as funcionalidades do JavaFX traz as seguintes vantagens:

- O uso de recursos como generics, anotações e *multithreading*;
- Facilidade para os desenvolvedores web utilizarem o JavaFX a partir de outras linguagens dinâmicas baseadas na JVM, como Groovy e Scala;
- Capacidade de extensão das bibliotecas nativas, possibilitando a especialização de funcionalidades, componentes e eventos;
- Uso de *listeners* e *observables*, conectando os componentes de interface ao modelo de negócio, refletindo assim o estado dos dados entre as camadas.

Sistema gráfico

Visando aprimorar a experiência do usuário com as interfaces, o sistema de renderização do JavaFX foi criado para suportar animações em duas e três dimensões. Esta renderização é realizada graças a dois pipelines, o Prism e o Quantum Toolkit, que gerenciam os recursos do sistema operacional e da máquina.

O Prism é responsável por processar as tarefas de renderização de imagens e animações utilizando os recursos do sistema operacional, como o OpenGL e o DirectX. Já o Quantum Toolkit gerencia a interação entre as tarefas do Prism junto ao Windowing Toolkit, ferramenta que gerencia os recursos de janela do sistema operacional.

Glass Window Toolkit

O Glass Window Toolkit, ou somente Glass, é responsável por mostrar as janelas (*Stage*, *Popup*, etc.) e gerenciar a fila de eventos do JavaFX. Estes eventos, no entanto, são gerenciados de forma

diferentes em cada sistema operacional. Em plataformas como o Mac OS, por exemplo, há chamadas de callback sempre que um evento é lançado, e nestes casos, o Glass recupera esses callbacks nativos, transforma-os em eventos do JavaFX e os adiciona à fila de eventos. No Windows, por sua vez, é necessário gerenciar essa fila manualmente, capturando o evento nativo do SO e adicionando o evento equivalente do JavaFX na fila.

Recursos multimídia

Os recursos multimídia do JavaFX estão disponíveis através da API `javafx.scene.media`, onde foram projetados para serem escaláveis e terem um comportamento consistente em todas as plataformas. O JavaFX é capaz de suportar mídias visuais e de som a partir de várias extensões, como mp3, AIFF, wav e flv.

Web Engine

Os recursos para deploy e apresentação de aplicações JavaFX na web são baseados no WebKit, uma engine open source de renderização de páginas web nos navegadores. Presente em vários browsers, como o Safari da Apple e o Chrome do Google, essa ferramenta oferece suporte a CSS, JavaScript, DOM, HTML5 e SVG.

Para a utilização no JavaFX, essa ferramenta embarcada que proporciona a interação entre o backend da aplicação e as páginas web, foi implementada em duas classes: `WebEngine` e `WebView`, apresentadas na Figura 2. A classe `WebEngine` disponibiliza todas as funcionalidades necessárias para o uso de páginas web, como o envio de formulários, uso de botões e navegação. A classe `WebView`, por sua vez, se responsabiliza pela integração entre o conteúdo web e o layout, além de prover os campos e métodos para a aplicação de efeitos e transformações.

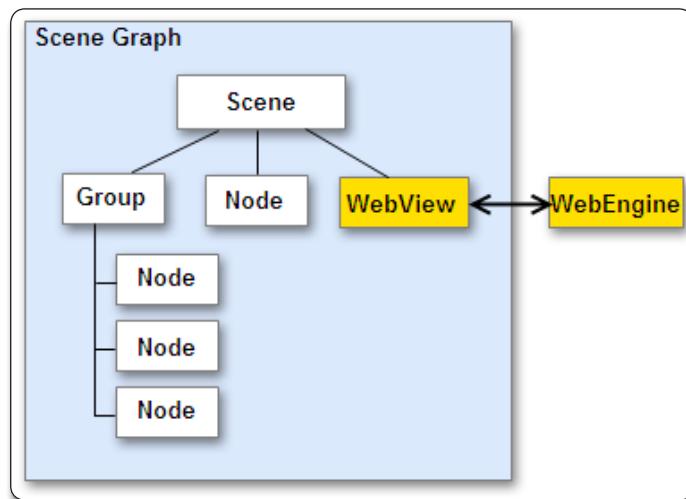


Figura 2. Comunicação entre as classes `WebView` e `WebEngine` junto aos nós do Scene Graph

JavaFX Cascading Style Sheets

O JavaFX CSS fornece a capacidade de personalização dos estilos da interface do usuário sem alterações em qualquer código Java. Para isso, o CSS pode ser aplicado a qualquer nó no Scene Graph e pode ser alterado em tempo de execução, possibilitando

trocas de estilo dinamicamente, aprimorando assim a interação do sistema com o usuário.

Esse componente é baseado na especificação 2.1 do CSS. Logo, pode ser interpretado por qualquer ferramenta de leitura de CSS, inclusive as que não suportam as extensões do JavaFX. Essa característica possibilita a união de código CSS a JavaFX e HTML em um mesmo arquivo, opção que veremos no exemplo deste tutorial.

Componentes de controle da interface

As interações com a interface no JavaFX são realizadas através de diversos tipos de componentes. Dentre eles, podemos citar: Button, Toggle Button, Hyperlink, Checkbox, Radio Button, Menu Buttons, ListView, etc., todos presentes no pacote `javafx.scene.control`.

As classes de controle são adicionadas à aplicação como nós do Scene Graph, sendo todas extensões da classe `Node`. Dessa forma podemos aplicar propriedades de estilo e efeitos em todos os componentes, por exemplo, um botão pode ter o efeito de opacidade modificada, alterando o destaque do elemento no layout. Esse efeito é demonstrado na Figura 3, onde o botão à esquerda é exibido com o valor de opacidade 1.0, o do centro com 0.8 e o da direita com o valor de opacidade 0.5.



Figura 3. Botão com diferentes valores de opacidade

Transformações em 2D e 3D

Cada componente presente no Scene Graph pode ter sua posição transformada em coordenadas x, y e z. Essa transformação possibilita movimentar os componentes na tela, seja mudando a posição, girando o componente, entre outras possibilidades. As transformações são realizadas através das seguintes classes do pacote `javafx.scene.transform`:

- **Translate** – Movimenta um nó de um ponto a outro. Por exemplo, para movimentar o componente da esquerda para a direita, basta aumentar o valor da coordenada X, para movimentar o componente na vertical, basta alterar o valor da coordenada Y;
- **Scale** – Redimensiona um nó para aparecer ser maior ou menor, dependendo do fator de escala. Essa transformação aplica o fator de escala (a quantidade que o componente deve aumentar ou diminuir) em todas as coordenadas;
- **Shear** – Gira um eixo de forma que o x e o y nunca fiquem perpendiculares, ou seja, que os eixos nunca se cruzem, deslocando os valores das coordenadas pelo multiplicador especificado no giro;
- **Rotate** – Gira um nó em relação a um determinado eixo, ou seja, define um eixo no qual o componente irá girar e aplica os fatores de deslocamento nos demais, girando o componente.

Efeitos visuais

O desenvolvimento de interfaces ricas e detalhadas em conjunto com a utilização de efeitos visuais aprimora e torna a interface mais atraente e lúdica para o usuário, facilitando o aprendizado

e simplificando seu uso no dia a dia. Pensando nisso, o JavaFX disponibiliza diferentes efeitos. Dentre eles, podemos citar:

- **Drop shadow** – Permite a renderização de um componente atrás de outro, deixando o foco do layout no componente mais visível;
- **Reflection** – Essa opção dá a ilusão do item estar refletido na tela, simulando o efeito do reflexo na água;
- **Lighting** – Simula uma fonte de luz que brilha sobre determinado conteúdo, dando ao objeto uma aparência mais realista.

Principais recursos

Além de incorporar todos os recursos da linguagem, como expressões Lambda e *default methods*, novidades do Java 8, a versão 8 do JavaFX inclui outras várias novidades. A seguir apresentamos uma descrição das principais:

- **Integração com Swing:** O uso de componentes da biblioteca Swing no JavaFX é mais uma opção para criação de interfaces viabilizada na release 2.0. A partir da versão 8 do JavaFX, o oposto também é possível, ou seja, componentes criados em JavaFX agora podem ser empregados em aplicações Swing. Isso é possível com a utilização da classe **SwingNode**;
- **Fusão do Event Dispatch Thread e do JavaFX Application Thread:** Na nova versão do JavaFX, o Event Dispatch Thread, usado pela biblioteca Swing, se fundiu ao JavaFX Application Thread. Essa junção reduziu consideravelmente a quantidade de código necessário para criar aplicações com componentes "FX" e Swing;
- **O novo tema: Modena:** Um novo tema foi introduzido. A aparência de Modena renova o visual dos componentes e possibilita a criação de interfaces com cores mais vivas e atualizadas, como pode ser verificado na **Figura 4**;

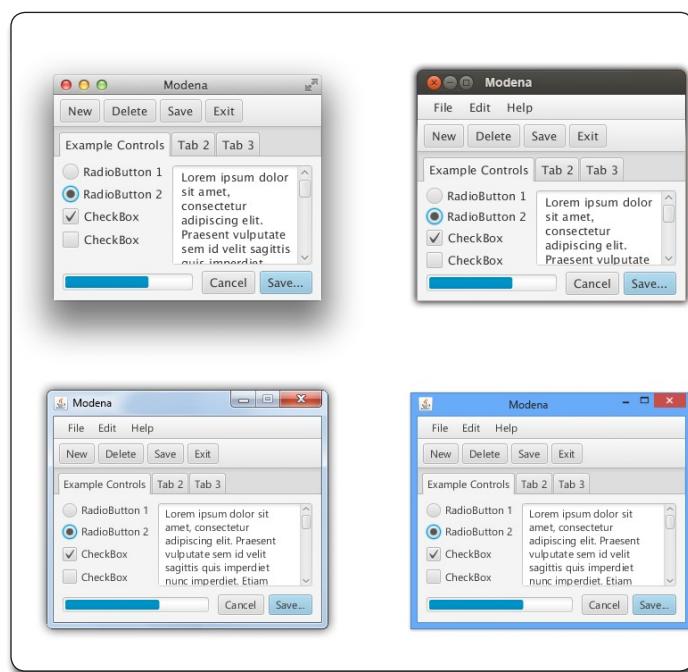


Figura 4. Novo layout do tema Modena em diferentes sistemas operacionais

- **Supporte a Rich Text:** Para aumentar as opções de personalização da interface, foi adicionado o suporte à Rich Text. Esse recurso possibilita a customização de palavras individualmente através de estilos CSS, a capacidade de escrever da direita para esquerda (*Bidirectional Text*) ou o uso de efeitos como sombreado nos textos;

- **Supporte ao OpenGL ES2:** Outro recurso adicionado foi o suporte ao OpenGL ES2, API baseada em OpenGL para tratamento de gráficos 2D e 3D em sistemas embarcados, como consoles, PDAs, smartphones e veículos. O OpenGL ES2 oferece uma interface entre o software e a aceleração gráfica por meio de profiles, que são personalizados com o intuito de pré-processar diretrizes de controle, otimizando a renderização de acordo com o ambiente onde o recurso será processado;

- **Novas funções gráficas com recursos em 3D:** Os novos componentes **shape3D** (Box, Cylinder, MeshView e Sphere), **SubScene**, **Material**, **PickResult**, **LightBase** (**AmbientLight** e **PointLight**) e **SceneAntialiasing** foram adicionados à biblioteca gráfica JavaFX 3D;

- **Supporte à arquitetura ARM:** O JavaFX agora está disponível para a plataforma ARM. O Java Development Kit para plataformas ARM inclui a base, os gráficos e os componentes de controle do JavaFX. Muitos fabricantes tendem a criar softwares embarcados na linguagem C, porém o Java traz diversas vantagens, como:

- Com a orientação a objetos as entidades podem ser representadas com mais naturalidade, como os sensores e outras representações de objetos do cliente, aumentando a manutenibilidade e produtividade;

- Menos propenso a erros que o C, a exemplo dos ponteiros;

- Sistemas portáveis. Não precisam ser recompilados para serem executados em outras plataformas ou sistemas operacionais;

- Suporte a várias bibliotecas independentes do sistema operacional, incluindo acesso ao banco de dados e componentes de interface gráfica.

- **Novas funções para a classe WebView.** Esta classe foi atualizada e recebeu novas funções e melhorias, a saber:

- Renderização de conteúdo HTML;

- Obtenção do histórico da web;

- Execução de comandos JavaScript;

- Comunicação entre JavaScript e JavaFX;

- Gerenciamento de janelas e pop-ups.

- **Novos componentes de controle:** Dois novos componentes de controle para interfaces gráficas foram adicionados, o **DatePicker** e o **TreeTableView**. O **DatePicker** permite que o usuário entre com a data como um texto ou selecione uma data de um calendário que é mostrado como um pop-up. O calendário se baseia no padrão ISO-8601 ou em qualquer outra classe cronológica definida no pacote `java.time.chrono`. Já **TreeTableView** é um componente que permite visualizar um número ilimitado de linhas de dados, mostrando o resultado em uma tabela com paginação. Esse componente unifica as funcionalidades das classes **TreeView** e **TableView**.

Exemplos de aplicações com JavaFX

O desenvolvimento de aplicações JavaFX, assim como ocorre com aplicações desktop e web, pode ser simplificado com o uso de IDEs que disponibilizam recursos para a implementação do código, como é o caso do NetBeans e Eclipse. Para a implementação dos exemplos deste artigo, optamos pelo Eclipse com o plugin e(fx)clipse, que pode ser configurado seguindo o tutorial disponibilizado em sua página (veja o endereço indicado na seção **Links**). Além disso, é necessário ter instalado o JDK 8, que já contempla a versão equivalente do JavaFX.

Feito isso, para apresentar alguns recursos do JavaFX, foram projetadas duas aplicações simples: uma que utiliza recursos para aplicações web e demonstra algumas funcionalidades em um mapa online, baseada na API do Google Maps; e outra que mostra como construir objetos em 3D e faz uso de animações.

Google Maps com recursos em JavaScript

Nosso primeiro exemplo mostrará como implementar os recursos para criação e renderização de páginas web a partir do JavaFX, através das classes **WebView** e **WebEngine**. Para isso, usaremos também algumas funcionalidades da API do Google Maps, junto à visualização de um mapa, com JavaScript e HTML. O mapa possibilitará a realização de pesquisas de endereço, aproximar ou afastar o zoom e alterar a baselayer (que define como o mapa será exibido).

Para o desenvolvimento do exemplo, primeiramente precisamos criar um projeto do tipo JavaFX pelo e(fx)clipse. Isso pode ser feito acessando: *File > New > Project*. Logo após, no wizard que será aberto, selecione *JavaFX > JavaFX Project* (vide **Figura 5**). A próxima etapa é configurar o projeto, definindo seu nome e escolhendo a versão do Java/JavaFX que será usada (vide **Figura 6**).

Configurado o projeto, partiremos para a implementação, onde vamos criar quatro arquivos. Além do .java, onde ficará o código JavaFX, criaremos um arquivo JavaScript, que concentrará as funções de comunicação com a API do Google Maps, um arquivo CSS, que customizará o layout dos componentes usados, e um arquivo HTML, que irá definir o layout onde o mapa será exibido e centralizar as chamadas das funções JavaScript e propriedades de estilo CSS.

O arquivo que apresenta o código mais simples é o HTML, que no nosso exemplo foi chamado de *googleMap.html* (veja a **Listagem 1**). Em seu código, temos uma referência ao arquivo que contempla o CSS usado para customizar o layout do projeto e duas referências para dois links JavaScript. Um desses links está relacionado à API de mapas do Google, que está disponível na web e pode ser acessada pelo endereço <http://maps.google.com/maps/api/js?sensor=false>. O outro link, por sua vez, faz referência ao arquivo JavaScript que contempla a função **initialize()**, responsável por inicializar a instância do mapa. Essa função é chamada no atributo **onload** da tag **body** do HTML, ou seja, o código é executado assim que o HTML é renderizado. Ademais, como podemos verificar, no conteúdo do **body** temos somente uma tag **div**, que será utilizada para renderizar o mapa.

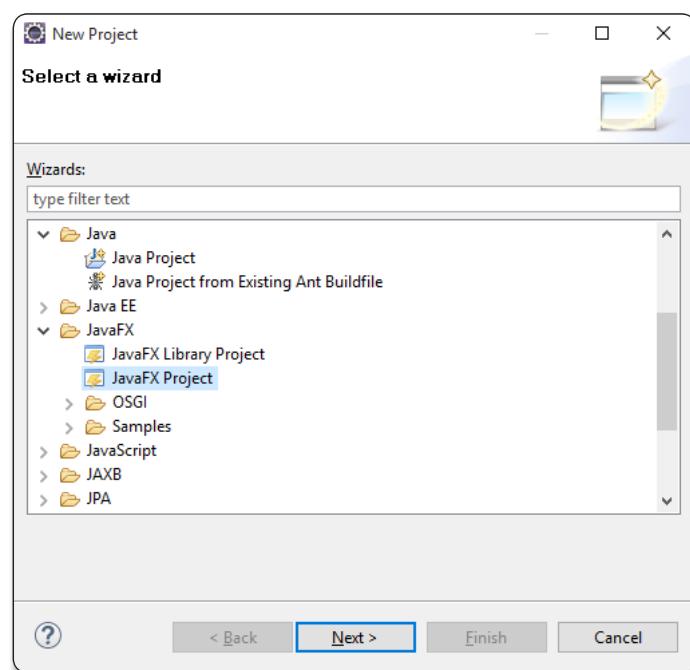


Figura 5. Criação de um projeto do tipo JavaFX pelo e(fx)clipse

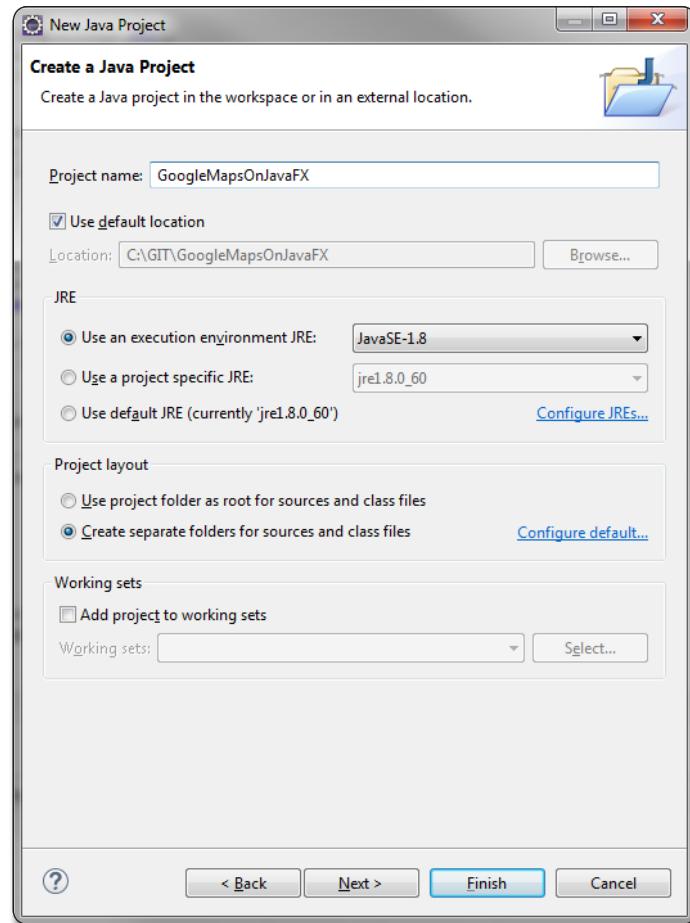


Figura 6. Configuração de um projeto JavaFX pelo e(fx)clipse

Listagem 1. Código HTML que define o layout do exemplo.

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<link rel="stylesheet" type="text/css" href="googleMap.css">
<script type="text/javascript"
src="http://maps.google.com/maps/api/js?sensor=false"/>
<script type="text/javascript" src="mapFunctions.js"/>
</head>
<body onload="initialize()">
<div id="map_canvas" style="width: 100%; height: 100%"></div>
</body>
</html>
```

A função `initialize()` foi implementada no arquivo `mapFunctions.js` e define as propriedades do mapa, configurando assim como ele será apresentado na tela (veja a **Listagem 2**). Estas definições estão relacionadas basicamente à inicialização dos componentes do mapa do Google que serão utilizados. No nosso exemplo, criamos a variável `myOptions` para especificar o ponto central onde o mapa será aberto (a coordenada com os valores de latitude e longitude que representam o centro do mapa é definida pelo método `google.maps.LatLng()`), o nível de zoom que será iniciado, a baselayer padrão e outras funcionalidades nativas que podem ser carregadas da API do Google Maps, como o street view e o controle de navegação. Note que a instância do mapa é criada usando o método `google.maps.Map()`, que recebe como parâmetro a variável `myOptions` e o elemento HTML da `div`, que indica onde o mapa será renderizado no arquivo HTML.

A próxima configuração declarada na função `initialize()` diz respeito às funcionalidades de aproximação e afastamento do mapa, definidas por `zoomIn()` e `zoomOut()`. Essas funções controlam o zoom atual, acrescentando ou diminuindo o valor de acordo com a opção que foi chamada. No caso da função `zoomIn()`, o usuário notará uma aproximação e maior detalhamento do mapa. Já com a função `zoomOut()`, acontece o contrário. Outra responsabilidade de `initialize()` é a criação das constantes que representam as opções de baselayer disponíveis (Estradas, Satélite, Híbrido e Terreno).

Por fim, `initialize()` cria a função para consulta a endereços. Essa função, de nome `goToLocation()`, pesquisa a `String` digitada pelo usuário na base de dados de endereços do Google, via serviço da própria API do Google Maps, usando a classe `google.maps.Geocoder()`, e caso encontre algum resultado, o ponto central do mapa é alterado para o novo local.

Outro arquivo que criamos para a implementação do exemplo foi o `googleMap.css`, apresentado na **Listagem 3**. Nele definimos os estilos para customização dos componentes do JavaFX e HTML. As propriedades de CSS disponíveis no JavaFX podem ser consultadas na wiki online criada pela Oracle (veja a seção [Links](#)). No nosso exemplo, foram usados atributos de customização para as tags `html`, `body` e `div`, que irá renderizar o mapa, e para as propriedades de estilo utilizadas no código do JavaFX, como a formatação do menu e do mapa.

Listagem 2. Código da função `initialize()`, responsável por inicializar o mapa e suas propriedades.

```
function initialize() {
    var latlng = new google.maps.LatLng(-19.9327448,-43.9300262);

    var myOptions = {
        zoom: 14,
        center: latlng,
        mapTypeId: google.maps.MapTypeId.ROADMAP,
        mapTypeControl: false,
        navigationControl: false,
        streetViewControl: false,
        backgroundColor: "#666970"
    };

    document.geocoder = new google.maps.Geocoder();

    document.map = new google.maps.Map(
        document.getElementById("map_canvas"),myOptions);

    document.zoomIn = function zoomIn() {
        var zoomLevel = document.map.getZoom();
        if (zoomLevel <= 20) document.map.setZoom(zoomLevel + 1);
    }

    document.zoomOut = function zoomOut() {
        var zoomLevel = document.map.getZoom();
        if (zoomLevel > 0) document.map.setZoom(zoomLevel - 1);
    }

    document.setMapTypeRoad = function setMapTypeRoad() {
        document.map.setMapTypeId(google.maps.MapTypeId.ROADMAP);
    }

    document.setMapTypeSatellite = function setMapTypeSatellite() {
        document.map.setMapTypeId(google.maps.MapTypeId.SATELLITE);
    }

    document.setMapTypeHybrid = function setMapTypeHybrid() {
        document.map.setMapTypeId(google.maps.MapTypeId.HYBRID);
    }

    document.setMapTypeTerrain = function setMapTypeTerrain() {
        document.map.setMapTypeId(google.maps.MapTypeId.TERRAIN);
    }

    document.goToLocation = function goToLocation(searchString) {
        document.geocoder.geocode( {'address': searchString},
            function(results, status) {
                if (status == google.maps.GeocoderStatus.OK) {
                    document.map.setCenter(results[0].geometry.location);
                } else {
                    alert("Erro ao buscar o endereço:" + status);
                }
            });
    }
}
```

O último arquivo que implementamos para a construção do projeto foi o `GoogleMap.java`. Nesse arquivo concentraremos todo o código JavaFX necessário para construir o exemplo, utilizando apenas dois métodos: `main()` e `start()`.

O método `main()`, apresentado na **Listagem 4**, é responsável apenas por realizar a chamada à classe `javafx.application.Application`, ponto de entrada de toda aplicação JavaFX. O início da execução se resume aos seguintes passos:

JavaFX 8: Uma introdução à arquitetura e às novidades da API

1. Construção da instância da respectiva *Application.class*;
2. Chamada ao método **init()**;
3. Chamada ao método **start(javafx.stage.Stage)**.

Listagem 3. Código do arquivo *googleMaps.css*.

```
html {  
    height: 100%;  
}  
  
body {  
    height: 100%;  
    margin: 0px;  
    padding: 0px  
}  
  
#map_canvas {  
    height: 100%;  
    background-color: #666970;  
}  
  
.map{  
    -fx-background-color: #666970;  
}  
  
.map-toolbar.button, .map-toolbar.toggle-button, .map-toolbar .label {  
    -fx-text-fill: white;  
    -fx-background-radius: 0;  
}  
  
.map-toolbar{  
    -fx-base: #505359;  
    -fx-background: #505359;  
    -fx-shadow-highlight-color: transparent;  
    -fx-spacing: 5;  
    -fx-padding: 4 4 4;  
}
```

Listagem 4. Código do método *main()*.

```
/**  
 * Método main().  
 * @param args Argumentos para a execução  
 */  
public static void main(String[] args) {  
    Application.launch(args);  
}
```

Já no método **start()**, vide **Listagem 5**, implementamos o código do JavaFX. Nele, instanciamos as classes **WebView** e **WebEngine**, responsáveis pelas operações de criação e renderização dos recursos web. Para renderizar o código HTML, chamamos o método **load()** da classe **WebEngine**, passando o arquivo *googleMap.html* como parâmetro.

O próximo passo é a criação dos botões que realizam a alteração da baselayer. Em nosso exemplo, cada botão foi criado separadamente, usando a classe **javafx.scene.control.ToggleButton**, e então são agrupados, com a classe **javafx.scene.control.ToggleGroup**. O listener para tratar o clique nos botões e realizar a troca da baselayer foi feito implementando a interface **javax.beans.value.ChangeListener**, identificando qual botão foi acionado e chamando a respectiva função criada no arquivo *mapFunctions.js*.

A funcionalidade de pesquisa de endereços, por sua vez, foi desenvolvida utilizando a classe **javafx.scene.control.TextField**

para a entrada da **String** de consulta. Nesse componente, adicionamos a mensagem “Entre com o endereço” através do método **setPromptText()**, expondo para o usuário o objetivo do campo. A pesquisa pelo endereço é realizada pelo listener adicionado ao componente. Esse listener, assim como o botão da troca de baselayer, implementa a interface **javax.beans.value.ChangeListener**, capturando o evento lançado quando o usuário está digitando no **inputText** da pesquisa, com o auxílio da classe **javafx.animation.KeyFrame**, e após um intervalo definido em milissegundos pela classe **javafx.util.Duration**, efetua a pesquisa na base de dados do Google ao executar a função **goToLocation()**.

Ainda no método **start()**, os botões relacionados ao zoomin e zoomout foram implementados utilizando a classe **javafx.scene.control.Button**, que recebe como parâmetro no construtor o label a ser apresentado na interface. Os eventos de clique foram adicionados aos botões através do método **setOnAction()**, que, por sua vez, recebe como parâmetro uma implementação da interface **javafx.event.EventHandler**, que chama a respectiva função JavaScript para aumentar ou diminuir o zoom no mapa.

Com todos os componentes de interface para interação com o mapa prontos, os agrupamos em um menu, o qual foi definido no exemplo com a classe **javafx.scene.control.ToolBar**. Em seguida, esse agrupamento foi adicionado ao **javafx.scene.layout.BorderPane**, um componente de layout do JavaFX usado para posicionar os itens na tela. O componente do menu foi adicionado no topo, e a instância da classe **WebView**, que mostrará o conteúdo da página web (*googleMap.html*), foi adicionada no centro da tela. O **BorderPane** será o nó raiz do **javafx.scene.Scene** (Scene Graph), a raiz da árvore de componentes do JavaFX.

Note ainda que o método **start()** recebe como parâmetro o **javafx.stage.Stage**, contêiner de mais alto nível do JavaFX. Na instância de **Stage** configuramos a janela que será exibida para apresentar os componentes especificados pela aplicação, que estão no objeto **scene**.

Ao executar a aplicação, o resultado será o apresentado na **Figura 7**, que mostra a janela com o mapa disponibilizado pelo serviço do Google. Observe na parte superior que temos o campo para busca por endereço, botões de incremento e diminuição do zoom e opções para a alteração da base layer (estradas, satélite, híbrido e terreno). A partir disso, ao realizar a busca por uma localização, o centro do mapa será alterado para o endereço desejado.

Cubo3D com animação

O segundo exemplo demonstra a criação de um cubo 3D e a aplicação do efeito de rotação ao mesmo, fazendo com que ele fique girando na tela. O cubo foi desenhado a partir da criação de cada lado separadamente, deixando os lados paralelos com a mesma cor, o que facilita a visualização da rotação.

A implementação desse código foi realizada em um novo projeto JavaFX, criado no Eclipse exatamente como demonstrado no primeiro exemplo. Para este caso, no entanto, definimos apenas um arquivo, a classe de nome **Cube3D**, que trará o método

Listagem 5. Método start() da classe GoogleMap.

```
@Override
public void start(Stage stage) {
    // Instancia as classes WebView e WebEngine.
    final WebView webView = new WebView();
    final WebEngine webEngine = webView.getEngine();
    // Carrega o HTML onde foi criado o mapa do Google
    webEngine.load(this.getClass().getResource("googleMap.html").toString());
    // Cria os botões para alternar as base layers
    final ToggleGroup baselayerGroup = new ToggleGroup();
    final ToggleButton road = new ToggleButton("Road");
    road.setSelected(true);
    road.setToggleGroup(baselayerGroup);
    final ToggleButton satellite = new ToggleButton("Satellite");
    satellite.setToggleGroup(baselayerGroup);
    final ToggleButton hybrid = new ToggleButton("Hybrid");
    hybrid.setToggleGroup(baselayerGroup);
    final ToggleButton terrain = new ToggleButton("Terrain");
    terrain.setToggleGroup(baselayerGroup);
    baselayerGroup.selectedToggleProperty().addListener(
        new ChangeListener<Toggle>() {
            public void changed(ObservableValue<? extends Toggle>
                observableValue, Toggle toggle, Toggle toggle1) {
                if (road.isSelected()) {
                    webEngine.executeScript("document.setMapTypeRoad()");
                } else if (satellite.isSelected()) {
                    webEngine.executeScript("document.setMapTypeSatellite()");
                } else if (hybrid.isSelected()) {
                    webEngine.executeScript("document.setMapTypeHybrid()");
                } else if (terrain.isSelected()) {
                    webEngine.executeScript("document.setMapTypeTerrain()");
                }
            }
        });
    }

    // Barra de botões para que o usuário possa escolher a base layer
    HBox buttonBar = new HBox();
    buttonBar.getChildren().addAll(road, satellite, hybrid, terrain);
    // Campo para inserção da localização
    final TextField searchField = new TextField();
    searchField.setPromptText("Entre com o endereço");
    searchField.textProperty().addListener(new ChangeListener<String>() {
        public void changed(ObservableValue<? extends String>
            observableValue, String s, String s1) {
            if (inputTextLocation != null) {
                inputTextLocation.stop();
            }
            inputTextLocation = new Timeline();
            inputTextLocation.getKeyFrames().add(
                new KeyFrame(new Duration(550), new EventHandler<ActionEvent>() {
                    public void handle(ActionEvent actionEvent) {
                        webEngine.executeScript("document.goToLocation('" + searchField.getText() + "')");
                    }
                }));
            inputTextLocation.play();
        }
    });

    // Botão de ZoomIn. Aproximando o mapa
    Button zoomIn = new Button("Zoom +");
    zoomIn.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent actionEvent) {
            webEngine.executeScript("document.zoomIn()");
        }
    });

    // Botão de ZoomOut. Afastando o mapa
    Button zoomOut = new Button("Zoom -");
    zoomOut.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent actionEvent) {
            webEngine.executeScript("document.zoomOut()");
        }
    });

    // Espaço horizontal para ajuste no layout
    Region spacer = new Region();
    HBox.setHgrow(spacer, Priority.ALWAYS);
    // Cria a barra de ferramentas com os controles da view
    ToolBar toolBar = new ToolBar();
    toolBar.getStyleClass().add("map-toolbar");
    toolBar.getItems().addAll(new Label("Localização:"), searchField, zoomIn, zoomOut, spacer, buttonBar);
    // Cria um layout do tipo BorderPane
    BorderPane root = new BorderPane();
    root.getStyleClass().add("map");
    root.setCenter(webView);
    root.setTop(toolBar);
    // Atribui o título da aplicação
    stage.setTitle("Google Maps");
    // Cria a scene, definindo o tamanho da janela e a cor de background
    Scene scene = new Scene(root, 1050, 590, Color.web("#666970"));
    stage.setScene(scene);
    // Atribui o arquivo de propriedades CSS.
    scene.getStylesheets().add(getClass().getResource("googleMaps.css").toExternalForm());
    // Apresenta o stage, não raiz.
    stage.show();
}
```

start() que representa o ponto inicial para execução de aplicações JavaFX. O código desse método pode ser visto na **Listagem 6**.

A primeira responsabilidade desse método é a instânciação dos pontos geométricos **xAxis** e **yAxis**, que irão representar as coordenadas X e Y, usadas como base para a criação dos lados do cubo. Esses pontos geométricos foram construídos a partir da classe **javafx.geometry.Point3D**.

Para compor o cubo, os seis lados foram criados separadamente, e como especificamos que os lados paralelos devem ter a mesma cor, foram definidas as cores: azul, vermelho e verde. Em seguida, cada lado foi posicionado de forma a criar o volume do cubo, o que é feito através da rotação

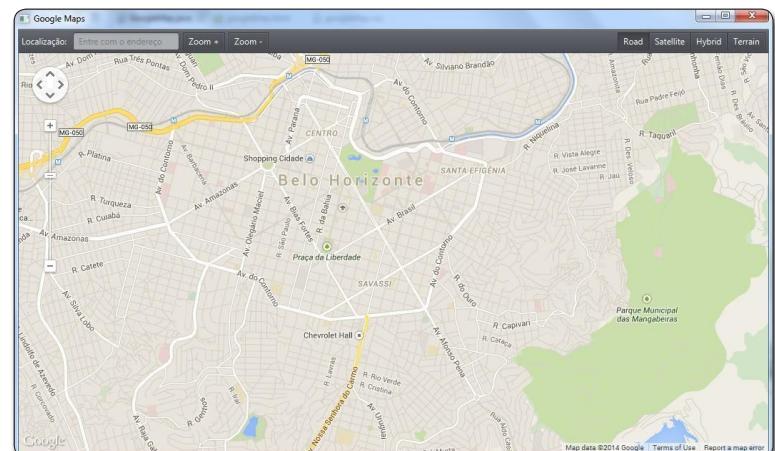


Figura 7. Aplicação JavaFX usando browser embarcado para acessar recursos do Google Maps

JavaFX 8: Uma introdução à arquitetura e às novidades da API

Listagem 6. Código do método start() para o exemplo do Cubo3D.

```
@Override
public void start(Stage primaryStage) {
    final Point3D xAxis = new Point3D(1, 0, 0);
    final Point3D yAxis = new Point3D(0, 1, 0);

    Rectangle bottomFace = createRectangle(Color.BLUE);

    Rectangle topFace = createRectangle(Color.BLUE);
    topFace.getTransforms().addAll(new Translate(0, 0, CUBE_SIZE));

    Rectangle frontFace = createRectangle(Color.GREEN);
    frontFace.getTransforms().addAll(
        new Rotate(-90, xAxis), new Translate(0, -CUBE_SIZE, 0)
    );

    Rectangle backFace = createRectangle(Color.GREEN);
    backFace.getTransforms().addAll(
        new Rotate(90, xAxis), new Translate(0, 0, -CUBE_SIZE)
    );

    Rectangle rightFace = createRectangle(Color.RED);
    rightFace.getTransforms().addAll(
        new Rotate(90, yAxis), new Translate(-CUBE_SIZE, 0, 0)
    );
}

Rectangle leftFace = createRectangle(Color.RED);
leftFace.getTransforms().addAll(
    new Rotate(-90, yAxis),
    new Translate(0, 0, -CUBE_SIZE)
);

Group root = new Group();
root.getChildren().addAll(topFace, bottomFace, frontFace,
    backFace, rightFace, leftFace);
root.relocate(CUBE_SIZE / 2, CUBE_SIZE / 2);

RotateTransition rotation = new RotateTransition(new Duration(1500), root);
rotation.setAxis(new Point3D(1, 1, 1));
rotation.setFromAngle(0);
rotation.setToAngle(360);
rotation.setCycleCount(Transition.INDEFINITE);

Scene scene = new Scene(root, 2 * CUBE_SIZE, 2 * CUBE_SIZE);
PerspectiveCamera camera = new PerspectiveCamera(false);
scene.setCamera(camera);
primaryStage.setScene(scene);
primaryStage.show();
rotation.play();
}
```

em relação aos eixos X e Y com a classe `javafx.scene.transform.Rotate`. Com essa transformação (rotação), os lados de mesma cor foram posicionados paralelamente, e juntos, foram dispostos de forma a criar a geometria do cubo. O método que concentra a criação de cada lado foi chamado de `createRectangle()`, mostrado na [Listagem 7](#). Esse método cria um retângulo a partir da cor desejada, usando um tamanho fixo de altura e largura.

Listagem 7. Código do método createRectangle().

```
private Rectangle createRectangle(Color color) {
    Rectangle rectangle = new Rectangle();
    rectangle.setWidth(CUBE_SIZE);
    rectangle.setHeight(CUBE_SIZE);
    rectangle.setFill(color);
    return rectangle;
}
```

Com todos os lados do cubo prontos, é necessário agrupá-los em um único objeto. Com esse objetivo, instanciamos a classe `javafx.scene.Group` para representar o cubo como um nó, ou seja, um componente do Scene Graph, que é a árvore de componentes da aplicação. Criado o componente do cubo, de nome `root`, cada lado é adicionado como um filho do mesmo, e logo após, o cubo é posicionado na tela através do método `relocate()`.

O próximo passo é a implementação do efeito de rotação, o que foi feito através da classe `javafx.animation.RotateTransition`. Ao instanciar essa classe, passamos no construtor o tempo da rotação em milissegundos (1500), usando uma instância da classe `javafx.util.Duration`, e o componente do cubo. Com a instância da animação de rotação em mãos, representada pela variável `rotation`, note que atribuímos ainda o eixo no qual o efeito será

aplicado. Para isso, criamos mais uma instância da classe `javafx.geometry.Point3D`, definindo que o movimento de rotação do cubo ocorrerá no sentido dos três eixos: X, Y e Z.

Por fim, criamos uma instância da classe `javafx.scene.Scene`, que representa o Scene Graph. Nesse objeto, definimos o tamanho da janela a ser criada para apresentar os componentes a partir do nó raiz, que no nosso exemplo é a instância da classe `javafx.scene.Group`. O resultado pode ser visto na [Figura 8](#), que mostra a imagem do cubo em rotação.

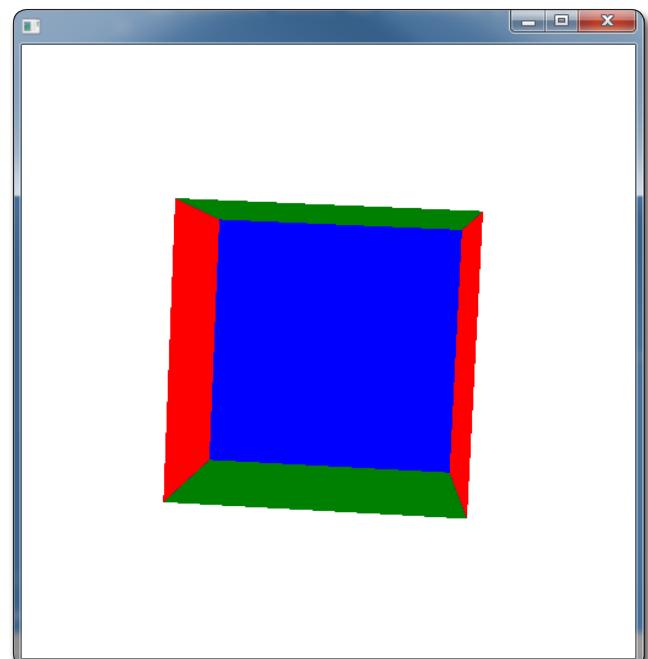


Figura 8. Cubo 3D criado com JavaFX

O JavaFX é a aposta para a nova geração de aplicativos rich client da Oracle. Projetado para oferecer uma aparência leve e com otimização de hardware, o JavaFX permite a implementação de sistemas com diversas opções de animação e efeitos, o que garante uma interface gráfica diferenciada e rica em usabilidade, e possibilita ainda a preservação do código nativo de aplicações já desenvolvidas, o reuso de bibliotecas Java, a conexão com outras aplicações de middleware baseadas em servidor, assim como acessar recursos nativos do sistema. Portanto, vale a pena conferir todos os recursos que essa API disponibiliza e utilizá-la no seu dia a dia.

Autor



Daniel Assunção Faria de Menezes

daniel.afmenezes@gmail.com

Bacharel em Ciéncia da Computação pela PUC-Minas, Pós-graduado em Engenharia de Software pela PUC-Minas. Atua há mais de nove anos com desenvolvimento de software com diversas linguagens. Entusiasta e apaixonado por tecnologia



Links:

Como configurar o e(fx)clipse.

<http://www.eclipse.org/efxclipse/install.html>

Visão geral do JavaFX.

<http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784>

Exemplos online de aplicações desenvolvidas com JavaFX.

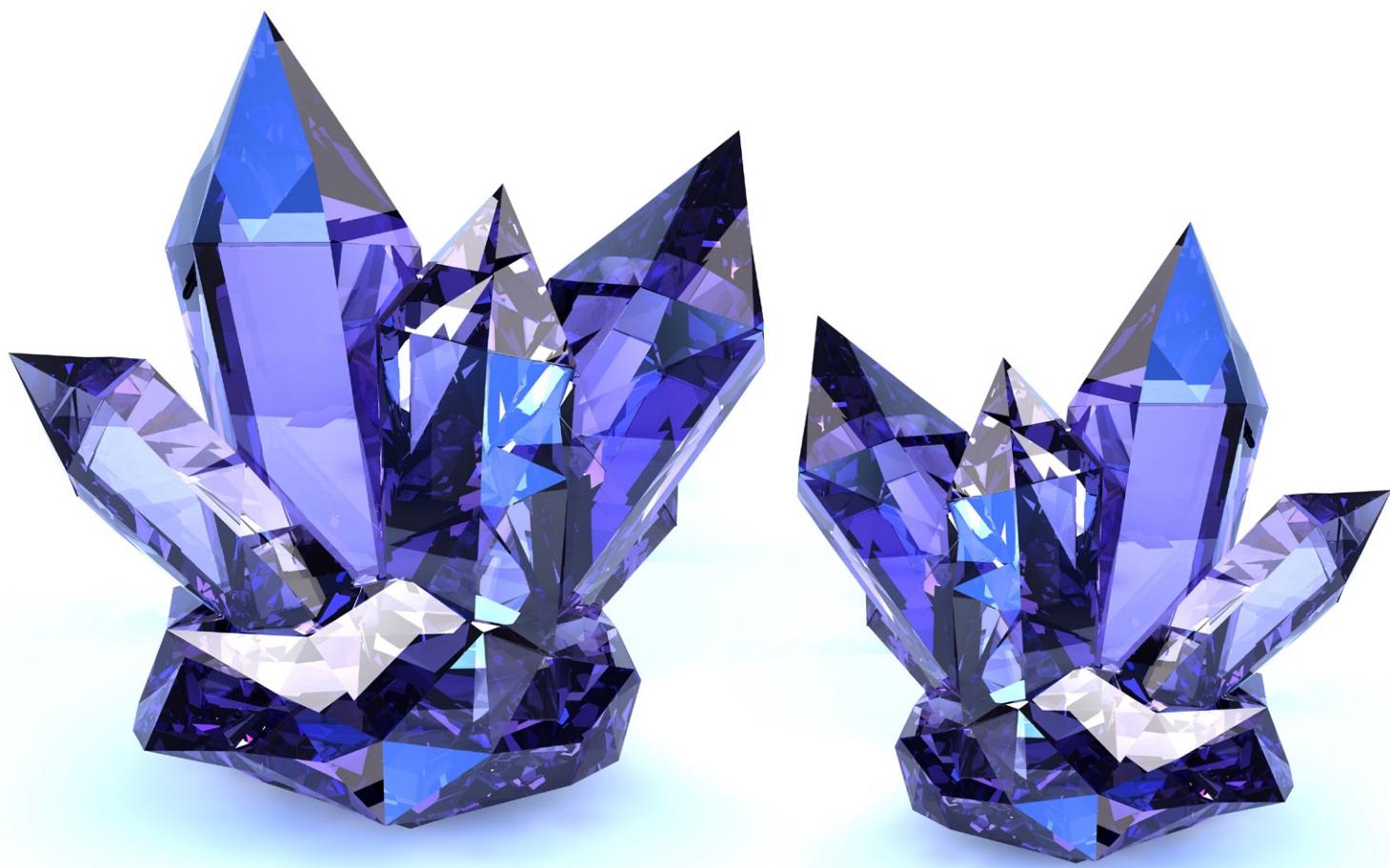
<http://download.oracle.com/otndocs/products/javafx/2/samples/Ensemble/index.html>

Adicionando elementos HTML em aplicações JavaFX.

<http://docs.oracle.com/javafx/2/webview/jfxpub-webview.htm>

JavaFX CSS Reference Guide.

<http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>



Arquitetura de Microservices com Spring Boot na prática

Aprenda neste artigo os principais conceitos e desenvolva um microsserviço para distribuição de pacotes de viagens a diferentes clientes de um portal

Uma das funções mais importantes e desafiadoras de um arquiteto de software é a definição da arquitetura para o sistema a ser desenvolvido. Iniciada entre as primeiras fases do projeto, esta etapa é fundamental para o bom andamento do mesmo. Para se ter uma noção mais precisa, uma decisão incorreta pode ocasionar atrasos na entrega do sistema, comprometer a qualidade, a segurança ou até mesmo inviabilizar o desenvolvimento.

Dada a importância dessa fase para a fundação do sistema, inúmeros padrões de desenvolvimento, frameworks e modelos arquiteturais são elaborados para resolver problemas comuns utilizando as melhores práticas. Neste cenário, um dos padrões arquiteturais que mais tem se destacado é a arquitetura de microservices, justamente pelos benefícios adquiridos ao adotar esta solução. Antes de discutirmos mais detalhes sobre essa arquitetura, no entanto, precisamos entender outro padrão arquitetural muito utilizado, talvez o mais comum deles: a arquitetura monolítica.

Portanto, no decorrer do artigo iremos debater sobre os conceitos introdutórios dos padrões de arquitetura monolítica e de microservice e, de forma prática, iremos aplicar esses conceitos para solucionar um problema elaborado para o caso de uso do artigo, que fará uso do Spring Boot como base.

Aplicações Monolíticas

Sistemas construídos sobre o padrão arquitetural monolítico são compostos basicamente por módulos ou funcionalidades agrupadas que, juntas, compõem o software. Como exemplos de aplicações que fazem uso do modelo monolítico, podemos citar os sistemas ERP (*Enterprise Resource Planning*), como o SAP ou Protheus. Estes possuem, normalmente, módulos para controle do setor financeiro, contábil, compras, entre outros, agrupados numa única solução que acessa o banco de dados.

Fique por dentro

Este artigo apresenta de forma prática um dos assuntos que vem ganhando mais destaque entre arquitetos e desenvolvedores de software: o padrão de arquitetura microsserviços (ou microservices). Para isso, exploraremos aqui os principais conceitos, vantagens e desvantagens, assim como um caso de uso. A partir do conteúdo exposto o leitor será capaz de dar os primeiros passos na construção de sistemas baseados nessa arquitetura na plataforma Java utilizando o Spring Boot.

Na **Figura 1** podemos verificar um exemplo de ERP construído com base numa arquitetura monolítica.

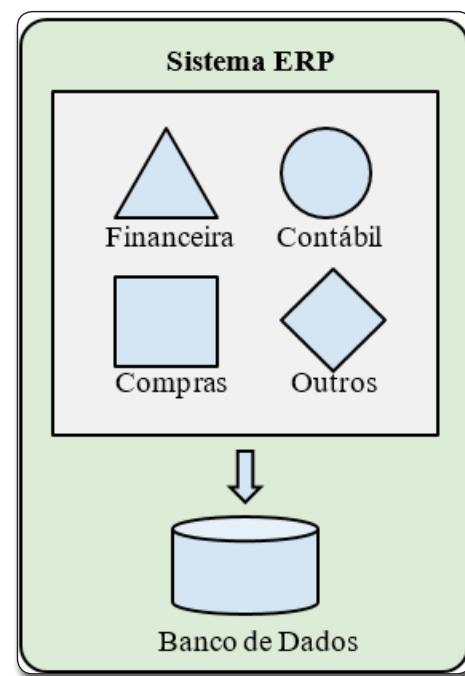


Figura 1. Exemplo de Arquitetura Monolítica

Introduzida a mais tempo que o padrão de microservices, a arquitetura monolítica também é comumente utilizada no desenvolvimento de aplicações web. No entanto, assim como qualquer solução, ele provê vantagens e desvantagens.

Dentre as vantagens, podemos citar:

- Implantar a aplicação é relativamente simples, uma vez que basta fazer o deploy de um único WAR ou EAR no servidor de aplicação;
- Simples de monitorar e manter;
- Ambiente de desenvolvimento mais simples, onde uma tecnologia core é utilizada em todo o projeto.

Dentre as desvantagens, podemos citar:

- Uma mudança feita em uma pequena parte da aplicação exige que todo o sistema seja reconstruído e implantado;
- Escalar a aplicação exige escalar todo o aplicativo, em vez de apenas partes que precisam de maior demanda;
- Alta curva de aprendizado para novos desenvolvedores que entram no time de desenvolvimento, uma vez o desenvolvedor deve entender o contexto de todos os módulos do projeto;
- Necessidade de reimplantar tudo para alterar um único componente. Sendo assim, o risco de falhas aumenta e consequentemente um ciclo maior de testes deve ser aplicado.

Microservices

Em suma, o estilo arquitetural de microservices consiste no desenvolvimento de uma aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo. Os serviços podem ser construídos de forma modular, com base na capacidade de negócio da organização em questão.

Pelo fato dos serviços serem construídos de forma independente, a implantação e a escalabilidade podem ser tratadas de acordo com a demanda. Essa independência também permite que os serviços sejam escritos em diferentes linguagens de programação e diferentes tecnologias, visando atender a necessidades específicas da melhor maneira. Outro ponto importante é que cada serviço pode ser gerenciado por diferentes times de desenvolvimento, possibilitando a formação de equipes especializadas.

Para termos uma melhor noção de onde podemos aplicar este conceito, listamos a seguir algumas vantagens e desvantagens.

Dentre as vantagens, podemos citar:

- Fácil entendimento e desenvolvimento do projeto;
- Fácil e rápida implantação (build e deploy);
- Redução do tempo de startup, pois os microservices são menores que aplicações monolíticas em termos de código;
- Possibilidade de aplicar a melhor ferramenta para um determinado trabalho.

Dentre as desvantagens, podemos citar:

- Dificuldade em implantar e operar sistemas distribuídos;
- Como cada microservice geralmente tem sua própria base de dados, o gerenciamento de transação se torna mais difícil (múltiplas bases de dados);

- Implantar uma alteração em um serviço utilizado por muitos sistemas demanda coordenação e cautela.

Organizando os microservices com a API Gateway

Depois de conhecermos os conceitos por traz dos microservices, uma das primeiras perguntas que nos vem à cabeça é: meus clientes ou consumidores (sistemas terceiros) precisam conhecer todos os microservices para se comunicar? Neste caso sim (ou teríamos que desenvolver um conjunto de microservices apenas para possibilitar a comunicação), e isso não é bom!

Como exemplo para melhorar o entendimento, vamos utilizar o caso do ERP citado anteriormente e decompor aquela arquitetura monolítica em serviços. Isso nos daria, por exemplo, um serviço específico para o módulo compras, outro para o financeiro, outro para o contábil e assim por diante.

Uma vez que os microservices estejam prontos para uso, imagine que nosso cliente seja uma aplicação web responsável apenas pela camada de apresentação do ERP, e que esta deva se comunicar com todos os módulos (serviços). Sendo assim, para evitar que essa aplicação conheça o endereço de cada um dos serviços, devemos incluir uma camada de abstração, pois seria muito complexo para o cliente se encarregar de manter estes endereços atualizados, por exemplo.

Para resolver este problema, um padrão foi criado: a API Gateway. Com o gateway os clientes terão um único ponto de entrada, isto é, precisam conhecer apenas um endereço. A partir disso, a API Gateway se encarrega de encaminhar (rotear) as requisições para os respectivos serviços e devolver a resposta para o cliente. A **Figura 2** apresenta o sistema ERP utilizando a nova arquitetura.

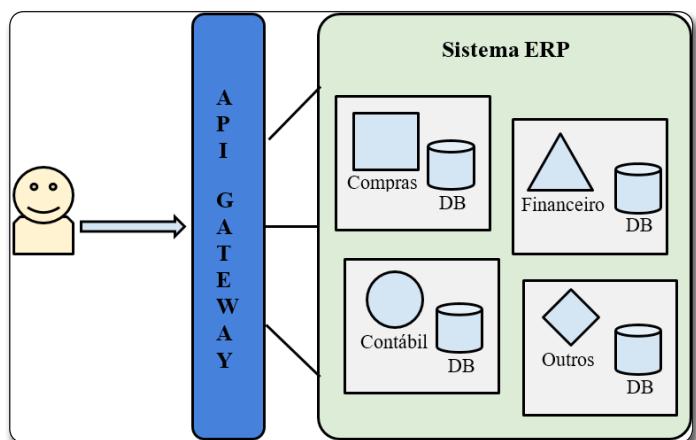


Figura 2. Exemplo de arquitetura de microservices com a API Gateway

Caso de Uso

Para aplicar os conceitos que apresentamos até aqui, vamos implementar um caso de uso utilizando como exemplo uma empresa fictícia de viagens. Esta empresa possui um portal que exibe em seu site as viagens e promoções cadastradas pelos parceiros, que podem ser, companhias aéreas ou agências de viagens.

Arquitetura de Microservices com Spring Boot na prática

Para entender melhor como o portal da empresa funciona, vamos analisar a **Figura 3**, que ilustra o desenho arquitetural do sistema.

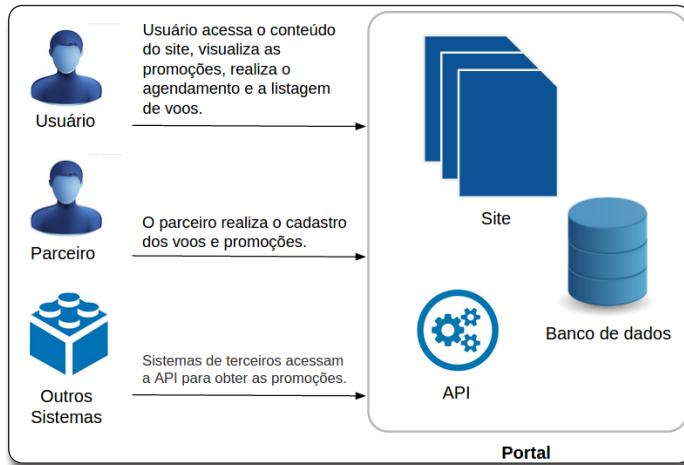


Figura 3. Arquitetura do portal

Analisando esta imagem, podemos destacar algumas das funções que o portal possui:

- Exibir as viagens e promoções para qualquer usuário através do site;
- Fornecer uma opção administrativa para os clientes parceiros poderem cadastrar as viagens e suas promoções de destaque;
- Disponibilizar via API (JSON) as viagens em promoção. Essas informações são obtidas por outros sistemas que consomem a API do portal e as publicam em seus sites ou aplicativos em forma de propaganda.

Repare que o portal é responsável por exibir todo o conteúdo de viagens e promoções para seus usuários, disponibilizar uma parte administrativa para clientes/parceiros cadastrarem pacotes de viagens, e também por expor a API para outros sistemas.

Os problemas apresentados

O problema com essa abordagem é que para qualquer alteração no portal, ou seja, para cada nova versão, seja para corrigir algum bug ou adicionar alguma funcionalidade, todos os recursos ficarão indisponíveis.

Outro problema são as frequentes “quedas” que o portal vem sofrendo, devido à grande quantidade de clientes que consomem a API. Essas quedas, causadas pelo número de requisições, acabam indisponibilizando toda a aplicação.

Solução proposta

Para resolver os problemas listados anteriormente e, principalmente, atender à recente demanda de consultas a promoções, o portal irá passar por uma reformulação arquitetural. Vamos transformar sua arquitetura monolítica em uma arquitetura de microservices com o intuito de ganhar escalabilidade. Para isso, desacoparemos a API de promoções de viagens do portal e cons-

truiremos uma aplicação (microservice) independente. Na **Figura 4** apresentamos a nova arquitetura.

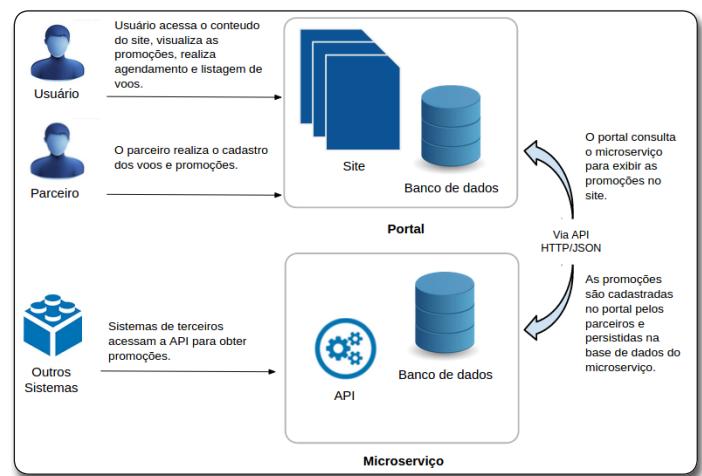


Figura 4. Arquitetura do portal utilizando microservice

Com a proposta de uma nova arquitetura definida, vamos colocar em prática os conceitos apresentados. Partindo da premissa que nosso portal já existe, o nosso dever será desenvolver apenas o serviço. Desta forma, tornaremos nosso exemplo mais simples e manteremos o foco no objeto de estudo. Por fim, para testar nosso microservice, iremos simular seu acesso pelo portal e por sistemas clientes através de um cliente REST.

Gradle

O Gradle é uma ferramenta open source de automação de build que também vem sendo muito utilizada para a construção de projetos. Assim como o Ant e o Maven, fornece mecanismos para facilitar o trabalho de configuração e manutenção do projeto, simplificando, por exemplo, as tarefas de gerenciamento de dependências e empacotamento da aplicação. Um dos grandes diferenciais do Gradle é a possibilidade de configuração de forma declarativa ou programaticamente, através da linguagem Groovy em vez de XML.

Instalando o Gradle

A instalação do Gradle é bastante simples. Basta fazer o download da distribuição pelo endereço indicado na seção [Links](#), descompactar o pacote no diretório de sua preferência e configurar as variáveis de ambiente de acordo com o sistema operacional.

Para verificar se a instalação foi concluída com sucesso, execute o comando `gradle -version` no terminal. O console deve exibir a versão do Gradle, do Groovy e outros detalhes. Se a instalação e configuração foram realizadas com sucesso, algo semelhante à Figura 5 deve ser apresentado na saída do console.

Iniciando o desenvolvimento do microservice

Agora que o Gradle está configurado e pronto para uso, iremos iniciar a construção do projeto de microservices utilizando alguns

comandos do mesmo. Para isso, crie o diretório que irá conter todos os arquivos do microservice (em nosso exemplo, *promocoes*) e depois, através do terminal, execute o seguinte comando dentro desse diretório:

```
gradle init --type java-library
```

Este comando informa ao Gradle para criar um projeto Java neste local. Concluída a execução, a estrutura básica para iniciar a implementação do projeto estará pronta (vide **Figura 6**).

Note que juntamente com a estrutura de diretórios foi criado um arquivo chamado *build.gradle* na raiz do projeto. Este arquivo é responsável por conter os detalhes de configuração de build do projeto. É neste arquivo também que devemos incluir plugins para adicionar funcionalidades ao Gradle, e configurar as dependências.

Com o comando que executamos anteriormente, o Gradle já se encarregou de incluir algumas configurações básicas para o

```
Gradle 2.2.1
-----
Build time: 2014-11-24 09:45:35 UTC
Build number: none
Revision: 6fcf59c06f43a4e6b1bcb401f7686a8601a1fb4a

Groovy: 2.3.6
Ant: Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM: 1.8.0_40 (Oracle Corporation 25.40-b25)
OS: Linux 3.2.0-70-generic-pae i386
```

Figura 5. Testando a instalação/configuração do Gradle

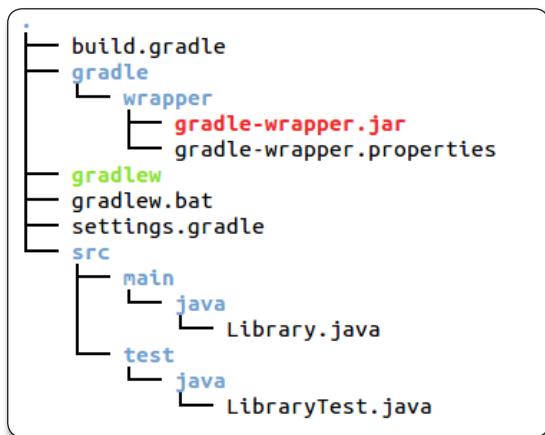


Figura 6. Estrutura de diretórios do projeto

projeto Java, como o plugin do Java e algumas dependências. Para verificar os detalhes dessas configurações, abra o arquivo *build.gradle* em um editor de texto. O conteúdo deste deve ser semelhante ao da **Listagem 1**. Vale ressaltar que os comentários gerados automaticamente foram removidos para melhor visualização.

Listagem 1. Código do arquivo *build.gradle*.

```
01. apply plugin:'java'
02.
03. repositories {
04.     jcenter()
05. }
06.
07. dependencies {
08.     compile 'org.slf4j:slf4j-api:1.7.12'
09.     testCompile 'junit:junit:4.12'
10. }
```

Na linha 1 observamos a declaração do plugin Java. Com esse plugin o Gradle consegue executar tarefas específicas para projetos que adotam essa linguagem: como executar o mesmo, compilar o código, gerar artefatos, realizar testes, entre outros.

Em seguida, nas linhas 3 a 5, temos a declaração do repositório (JCenter), ou seja, informamos onde o Gradle irá buscar as dependências declaradas. Logo abaixo, nas linhas 7 a 10 encontramos as dependências do projeto. Neste caso, temos a biblioteca de Log SLF4J e a biblioteca JUnit para testes unitários.

Repare que as dependências foram declaradas utilizando as palavras-chave **compile** e **testCompile**. Essas declarações são disponibilizadas pelo pugin do Java e indicam ao Gradle em que escopo as dependências devem ser empregadas, formando um agrupamento. Ou seja, as dependências declaradas no escopo **testCompile**, por exemplo, são adotadas para compilar o código de teste do projeto. Outros tipos de agrupamentos que podem ser utilizados são exibidos na **Tabela 1**.

Os desenvolvedores acostumados com o Maven podem estranhar um pouco a forma com que as dependências são especificadas no Gradle. O caractere dois pontos (“:”) é utilizado para separar os atributos grupo, nome e versão da declaração.

A **Listagem 2** demonstra como a mesma dependência (JUnit) pode ser declarada em ambos os gerenciadores.

Agora que já sabemos o essencial sobre o arquivo *build.gradle*, iremos editá-lo e incluir o plugin do Eclipse. Esse plugin irá fornecer ao projeto algumas funcionalidades.

Nome do Agrupamento	Contexto
Compile	Reúne as dependências necessárias para compilar o código fonte de produção do projeto.
Runtime	Reúne as dependências necessárias para as classes de produção em tempo de execução. Por padrão, também inclui as dependências de tempo de compilação.
testCompile	Reúne as dependências necessárias para compilar o código de teste do projeto. Por padrão, também inclui as classes de produção compiladas e as dependências de tempo de compilação.
testRuntime	Reúne as dependências necessárias para executar os testes. Por padrão, também inclui as dependências de compilação, tempo de execução e teste de compilação.

Tabela 1. Tipos de agrupamentos para as dependências

Listagem 2. Declaração de dependência com Maven e Gradle.

//Exemplo utilizando o Maven:

```
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
<scope>compile</scope>
</dependency>
</dependencies>
```

//Exemplo utilizando o Gradle:

```
dependencies {
    compile 'org.slf4j:slf4j-api:1.7.12'
}
```

Entre elas, a capacidade de preparar o projeto para ser importado no Eclipse. Para tal, basta adicionar o seguinte código no arquivo:

```
apply plugin:'eclipse'
```

Feito isso, salve e feche o editor e, no terminal, execute o comando:

```
gradle tasks
```

Deste modo, o Gradle irá acessar o arquivo em questão para checar os plugins existentes, e na sequência irá exibir todas as tarefas (ou tasks) disponíveis para execução. Observe na listagem exibida na saída do console que temos algumas tarefas referentes ao Eclipse, conforme expõe a **Figura 7**.

Com os comandos (tasks) para execução em mãos, iremos utilizar o seguinte para que o projeto possa ser importado para o Eclipse:

```
gradle eclipse
```

```
Documentation tasks
-----
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
-----
components - Displays the components produced by root project 'microservice'. [incubating]
dependencies - Displays all dependencies declared in root project 'microservice'.
dependencyInsight - Displays the insight into a specific dependency in root project 'microservice'.
help - Displays a help message.
model - Displays the configuration model of root project 'microservice'. [incubating]
projects - Displays the sub-projects of root project 'microservice'.
properties - Displays the properties of root project 'microservice'.
tasks - Displays the tasks runnable from root project 'microservice'.

IDE tasks
-----
cleanEclipse - Cleans all Eclipse files.
eclipse - Generates all Eclipse files.

Verification tasks
-----
check - Runs all checks.
test - Runs the unit tests.
```

Figura 7. Tarefas disponíveis

Ao final da execução o projeto estará pronto para ser importado, o que pode ser feito acessando o menu *File > Import > Existing Projects into Workspace*.

Uma vez que preparamos a estrutura básica do projeto, vamos iniciar as configurações mais específicas de nossa aplicação, adicionando o suporte ao Spring Boot, framework base para criação do microservice.

Spring Boot

Sabemos como é complexo e trabalhoso configurar aplicações nos padrões Java EE. Mesmo utilizando o Spring para realizar a inversão de controle, injeção de dependências e outras facilidades, ainda é preciso realizar o trabalho que muitos desenvolvedores tentam evitar, a escrita de XML e muitas linhas de configurações.

Percebendo isso, o Spring iniciou o projeto chamado Spring Boot com o intuito de eliminar a necessidade de XML para configuração, adotando o modelo de convenção sobre configuração (do inglês *Convention over Configuration – CoC*). Esse é um modelo de desenvolvimento de software que busca diminuir o número de decisões que os desenvolvedores precisam tomar, visando ganhar simplicidade sem perder flexibilidade.

Seguindo este princípio, basicamente com algumas anotações em nosso projeto conseguimos utilizar diversos frameworks de mercado sem muito esforço. Essa facilidade é alcançada através dos módulos do Spring Boot. Alguns deles, autoconfiguráveis, são destacados a seguir:

- Spring Data JPA;
- Spring Data MongoDB;
- Spring Web MVC;
- Spring Boot Actuator.

É importante destacar que apesar das configurações serem realizadas de forma automática, o Spring Boot nos possibilita ter total controle sobre o projeto. Deste modo, qualquer convenção pode ser substituída por uma configuração mais específica como, por exemplo, definir a conexão com a base de dados.

O Spring Boot não gera código ou realiza edições em arquivos do projeto para aplicar a convenção. Em vez disso, quando a aplicação é inicializada, ele injeta dinamicamente beans e configurações no contexto da aplicação. Por exemplo, se a dependência do HSQLDB foi adicionada ao projeto, e nenhuma especificação de base de dados foi declarada, então um banco de dados em memória será configurado.

Configurando o Spring Boot

Com os conceitos e principais características do Spring Boot apresentados, chegou a hora de colocá-lo em prática e utilizá-lo na implementação de nosso microservice. Para isso, vamos voltar ao *build.gradle* e incluir algumas novas linhas, de acordo com a **Listagem 3**.

Listagem 3. Código do arquivo *build.gradle*.

```
01. buildscript {  
02.     repositories { mavenCentral() }  
03.     dependencies {  
04.         classpath("org.springframework.boot:spring-boot-gradle-plugin:  
05.             1.2.3.RELEASE")  
06.     }  
07.  
08.     apply plugin:'java'  
09.     apply plugin:'eclipse'  
10.     apply plugin:'spring-boot'  
11.  
12.     repositories {  
13.         jcenter()  
14.     }  
15.  
16.     dependencies {  
17.         compile'org.slf4j:slf4j-api:1.7.5'  
18.         testCompile'junit:junit:4.11'  
19.     }
```

Entre as linhas 1 a 6 informamos ao Gradle um repositório para que ele possa localizar e baixar a dependência do plugin Spring Boot Gradle. Já na linha 10, declaramos o plugin propriamente dito. Essa configuração provê o suporte do Spring Boot ao projeto.

Este plugin disponibiliza tarefas para que seja possível empacotar nossa aplicação em JARs ou WARs executáveis, e também permite que as versões das dependências gerenciadas pelo Spring Boot sejam omitidas, o que possibilita obter sempre a versão estável mais recente.

As dependências do projeto

Para agregar funcionalidades ao microservice como, por exemplo, acesso à base de dados, iremos utilizar alguns recursos. Como esperado, estes serão declarados como dependências do projeto.

Dito isso, as três dependências que iremos utilizar são:

1. spring-boot-starter-web: essa dependência provê o básico necessário para iniciarmos um projeto web com Tomcat e Spring MVC;

2. spring-boot-starter-data-jpa: dependência que provê suporte à *Java Persistence API* (JPA) e Hibernate;

3. hsqldb: com essa dependência podemos utilizar uma base de dados em memória. O HyperSQL DataBase ou HSQLDB é um servidor de banco de dados simples escrito em Java. Iremos adotá-lo pela facilidade de não ser necessário instalar um banco de dados como o MySQL, Oracle, etc.

Dessa forma, a seção de dependências de nosso *build.gradle* deve ficar semelhante ao código apresentado na **Listagem 4**.

Listagem 4. Declaração de dependências final do arquivo *build.gradle*.

```
dependencies {  
    compile 'org.slf4j:slf4j-api:1.7.12'  
    compile 'org.springframework.boot:spring-boot-starter-web'  
    compile 'org.springframework.boot:spring-boot-starter-data-jpa'  
    compile 'org.hsqldb:hsqldb:2.3.3'  
  
    testCompile 'junit:junit:4.12'  
}
```

Classe de configuração

Agora que declaramos todas as dependências do projeto, iremos iniciar a codificação das classes, começando pela classe de configuração. Para isso, crie o pacote **br.com.javamagazine.promocoes.config** e, dentro dele, a classe **AppConfig**, que será responsável por conter algumas configurações básicas da aplicação como, por exemplo, detalhes de conexão com o banco de dados, configurações específicas de cache, entre outras. Assim, tudo o que geralmente configuraremos via XML, faremos via código Java. Tais ajustes são feitos, basicamente, por meio de anotações, como podemos verificar na **Listagem 5**.

Listagem 5. Código da classe *AppConfig*.

```
01. package br.com.javamagazine.promocoes.config;  
02.  
03. import org.springframework.boot.SpringApplication;  
04. import org.springframework.boot.autoconfigure.SpringBootApplication;  
05. import org.springframework.boot.orm.jpa.EntityScan;  
06. import org.springframework.context.annotation.ComponentScan;  
07. import org.springframework.data.jpa.repository.config.EnableJpaRepositories;  
08.  
09. @SpringBootApplication  
10. @EntityScan("br.com.javamagazine.promocoes.domain")  
11. @EnableJpaRepositories("br.com.javamagazine.promocoes.repository")  
12. @ComponentScan({"br.com.javamagazine.promocoes.service",  
13.                  "br.com.javamagazine.promocoes.controller"})  
13. public class AppConfig {  
14.  
15.     public static void main(String[] args) {  
16.         SpringApplication.run(AppConfig.class, args);  
17.     }  
18.  
19. }
```

A anotação **@SpringBootApplication**, presente na linha 9, reúne as funcionalidades das seguintes anotações:

- @Configuration:** Indica que a classe contém um código com definições de beans para o contexto de aplicação;
- @EnableAutoConfiguration:** Informa ao Spring Boot para ativar as configurações automáticas, utilizando como base as dependências declaradas e propriedades de configurações;
- @ComponentScan:** Indica ao Spring para buscar por componentes, configurações e serviços nos diretórios informados no parâmetro da anotação. Com isso, o Spring consegue encontrar os beans declarados.

Nota

O Spring Boot trouxe a anotação `@SpringBootApplication` como um facilitador ou uma alternativa às anotações `@Configuration`, `@EnableAutoConfiguration` e `@ComponentScan`, que são frequentemente utilizadas em conjunto.

Na linha 10 incluímos a anotação `@EntityScan` e passamos um pacote como parâmetro. Essa anotação faz com que o Spring Boot escaneie esse pacote em busca de entidades (classes anotadas com `@Entity`), e assim, as classes encontradas nesse local passam a ser gerenciadas pelo Spring.

Na sequência (vide linha 11), temos a anotação `@EnableJpaRepositories`. Esta recebe como parâmetro o pacote das entidades do tipo Repository a serem escaneadas. As classes desse tipo são as responsáveis por encapsular o mecanismo de persistência.

Por último, temos a anotação `@ComponentScan`, que em nosso caso recebe duas Strings, representando os dois pacotes a serem escaneados. Note que mencionamos anteriormente que a anotação `@SpringBootApplication` já inclui as funcionalidades da anotação `@ComponentScan`. Porém, para especificar os pacotes que devem ser escaneados, devemos fazer uso da mesma.

Nota

Normalmente, em aplicações que utilizam o Spring MVC, anotamos uma classe de configuração com `@EnableWebMvc`. Caso isso não seja feito e o projeto faça uso deste framework, o Spring Boot adiciona essa anotação automaticamente quando detecta o `spring-webmvc` no classpath.

Classe de modelo

Nesta etapa iremos definir a entidade que irá representar um pacote em nosso sistema. Sendo assim, após criar o pacote `br.com.javamagazine.promocoes.domain`, crie uma classe de nome `Pacote` com o código apresentado na **Listagem 6**. Essa classe define o objeto que será exibido no portal e disponibilizado via API para outros sistemas plugáveis que desejarem divulgar as promoções do portal de viagens.

Por se tratar de uma entidade, observe que temos a anotação `@Entity` declarada sobre a classe (linha 12). Esta informa ao Spring que objetos desse tipo devem ser tratados como objetos persistentes. Já nas linhas 15 e 16, temos a declaração das anotações `@Id` e `@GeneratedValue` para o atributo `codigo`, definindo assim o ID da tabela e a forma ou estratégia em que esse ID será gerado (automaticamente, neste caso).

O repositório

O repositório tem o papel de encapsular e abstrair a camada de persistência. É no repositório que encontramos o código responsável por pesquisar, inserir, atualizar e remover registros de nossa aplicação no banco de dados, arquivo, ou outro mecanismo de armazenamento.

Fazendo uso dessa abstração, o nosso microservice deve permitir que as operações de CRUD (*Create, Read, Update e Delete*) possam ser realizadas. Para viabilizar esse trabalho, iremos utilizar mais um framework disponibilizado pelo Spring, o Spring Data.

Listagem 6.

Código da entidade Pacote.

```
01. package br.com.javamagazine.promocoes.domain;
02.
03. import java.io.Serializable;
04. import java.util.Date;
05.
06. import javax.persistence.Entity;
07. import javax.persistence.GeneratedValue;
08. import javax.persistence.GenerationType;
09. import javax.persistence.Id;
10. import javax.persistence.Table;
11.
12. @Entity
13. public class Pacote implements Serializable {
14.
15.     @Id
16.     @GeneratedValue(strategy = GenerationType.IDENTITY)
17.     private Long codigo;
18.     private String origem;
19.     private String destino;
20.     private String descricao;
21.     private Date ida;
22.     private Date volta;
23.     private Double valor;
24.
25.     // métodos getters e setters omitidos
26.
27. }
```

O projeto Spring Data é um framework criado para realizar o acesso a dados de uma maneira bem simples, mas sem perder a robustez. Com este framework, conseguimos executar todas as operações do CRUD sem que haja a necessidade de implementarmos as operações. Para tal, basta criar uma interface, para definir nosso repositório, que estenda a interface `CrudRepository`.

A interface `CrudRepository` tem como diferenciais as seguintes características:

- Estender essa interface permite que o Spring encontre os repositórios e crie os objetos automaticamente;
- Fornece métodos que permitem a realização de algumas operações comuns, sem a necessidade de declará-los, como: `findOne()`, `findAll()`, `save()` e `delete()`.

Além disso, quando estendemos essa interface, devemos informar dois argumentos em sua declaração, sendo:

1. O primeiro, o tipo da entidade que deve ser gerenciada pelo repositório;
2. E o segundo, o tipo do identificador único da entidade.

A **Listagem 7** apresenta o código de nosso repositório. Observe na linha 8 que utilizamos a anotação `@Repository` para indicar que este será um componente da camada de persistência gerenciado pelo Spring.

Já na linha 9 estendemos a interface `CrudRepository` passando a entidade `Pacote` e o tipo de objeto que representa o identificador de nossa classe (neste caso, o tipo `Long`, do atributo `codigo`, que representa o nosso ID).

O serviço

A camada de serviço é responsável por conter as regras de negócio da aplicação. Em outras palavras, é nessa camada que

implementamos a lógica para que a aplicação execute operações com base no negócio.

A interface de serviço para a entidade **Pacote** de nosso microservice será a interface **PacoteService**, apresentada na **Listagem 8**. Como podemos verificar, ela não possui declaração de anotações ou outro detalhe relevante, pois é apenas uma interface POJO.

Listagem 7. Código da classe PacoteRepository.

```
01. package br.com.javamagazine.promocoes.repository;
02.
03. import org.springframework.data.repository.CrudRepository;
04. import org.springframework.stereotype.Repository;
05.
06. import br.com.javamagazine.promocoes.domain.Pacote;
07.
08. @Repository
09. public interface PacoteRepository extends CrudRepository<Pacote, Long> {
10.
11. }
```

Listagem 8. Código da interface PacoteService.

```
01. package br.com.javamagazine.promocoes.service;
02.
03. import java.util.Collection;
04.
05. import br.com.javamagazine.promocoes.domain.Pacote;
06.
07. public interface PacoteService {
08.
09.     Pacote findByCodigo(Long codigo);
10.
11.     Collection<Pacote> findAll();
12.
13.     Pacote create(Pacote pacote);
14.
15.     Pacote update(Pacote pacote);
16.
17.     void delete(Pacote pacote);
18.
19. }
```

Em seguida, na **Listagem 9**, apresentamos a classe de serviço **PacoteServiceImpl**, que implementa a interface criada anteriormente. Além disso, essa classe possui a anotação **@Service**, o que indica ao Spring que um objeto do tipo **PacoteService** deve ser criado no contexto da aplicação. Já nas linhas 15 e 16, declararmos o atributo que representa um objeto do tipo **PacoteRepository**. Este recebe a anotação **@Autowired** para que o Spring possa realizar a injeção de dependência, e assim, teremos um repositório pronto para ser utilizado.

Logo após, temos os métodos responsáveis por filtrar um **Pacote** por código, listar todos os pacotes, criar, atualizar e remover um determinado **Pacote**.

O controller REST

Agora vamos implementar a API REST de nossa solução, para que o microservice possa se comunicar com outras aplicações. Essa implementação será feita no controller, onde iremos declarar os endpoints, ou seja, as operações que nosso serviço irá realizar a partir do mapeamento de métodos HTTP e URIs.

Para construir nosso controller, crie uma classe chamada **PacoteController** no pacote **br.com.javamagazine.promocoes.controller**. Veja o código na **Listagem 10**. Ao anotar essa classe

Listagem 9. Código da classe PacoteServiceImpl.

```
01. package br.com.javamagazine.promocoes.service.impl;
02.
03. import java.util.Collection;
04.
05. import org.springframework.beans.factory.annotation.Autowired;
06. import org.springframework.stereotype.Service;
07.
08. import br.com.javamagazine.promocoes.domain.Pacote;
09. import br.com.javamagazine.promocoes.repository.PacoteRepository;
10. import br.com.javamagazine.promocoes.service.PacoteService;
11.
12. @Service
13. public class PacoteServiceImpl implements PacoteService {
14.
15.     @Autowired
16.     private PacoteRepository repository;
17.
18.     @Override
19.     public Pacote findByCodigo(Long codigo) {
20.         return repository.findOne(codigo);
21.     }
22.
23.     @Override
24.     public Collection<Pacote> findAll() {
25.         return (Collection<Pacote>) repository.findAll();
26.     }
27.
28.     @Override
29.     public Pacote create(Pacote pacote) {
30.         return repository.save(pacote);
31.     }
32.
33.     @Override
34.     public Pacote update(Pacote pacote) {
35.         return repository.save(pacote);
36.     }
37.
38.     @Override
39.     public void delete(Pacote pacote) {
40.         repository.delete(pacote);
41.     }
42.
43. }
```

com **@RestController** (vide linha 16), anotação introduzida na versão 4.0 do Spring, informamos que **PacoteController** será um controller REST.

Já a anotação **@RequestMapping**, declarada na linha 17, provê ao Spring informações de mapeamento dos endpoints para os métodos de nossa classe. Ao declarar esta anotação no nível de classe passando o valor **/api/v1/pacotes** para o atributo **value**, informamos ao Spring MVC que esse será o prefixo padrão utilizado para compor os endereços do controller. Assim, se tivermos um método contendo essa mesma anotação com o valor **/qualquer/coisa**, a URL completa para acessar esse método via HTTP deve ser **/api/v1/pacotes/qualquer/coisa**. Além disso, essa anotação disponibiliza o atributo **produces**, no qual especificamos o tipo de retorno ou Media Type padrão que todos os métodos devem assumir. Em nosso caso, declararmos **MediaType.APPLICATION_JSON_VALUE**, constante que representa o texto “**application/json**”.

Note que também utilizamos essa anotação para os métodos do controller, como pode ser visto nas linhas 23, 28, 33, 38 e 43.

Arquitetura de Microservices com Spring Boot na prática

Listagem 10. Código da classe PacoteController.

```
01. package br.com.javamagazine.promocoes.controller;
02.
03. import java.util.Collection;
04.
05. import org.springframework.beans.factory.annotation.Autowired;
06. import org.springframework.http.MediaType;
07. import org.springframework.web.bind.annotation.PathVariable;
08. import org.springframework.web.bind.annotation.RequestBody;
09. import org.springframework.web.bind.annotation.RequestMapping;
10. import org.springframework.web.bind.annotation.RequestMethod;
11. import org.springframework.web.bind.annotation.RestController;
12.
13. import br.com.javamagazine.promocoes.domain.Pacote;
14. import br.com.javamagazine.promocoes.service.PacoteService;
15.
16. @RestController
17. @RequestMapping(value = "/api/v1/pacotes",
produces = MediaType.APPLICATION_JSON_VALUE)
18. public class PacoteController {
19.
20.     @Autowired
21.     private PacoteService service;
22.
23.     @RequestMapping(value = "/{codigo}", method = RequestMethod.GET)
24.     public Pacote get(@PathVariable Long codigo) {
25.         return service.findByCodigo(codigo);
26.     }
27.
28.     @RequestMapping(method = RequestMethod.GET)
29.     public Collection<Pacote> getAll() {
30.         return service.findAll();
31.     }
32.
33.     @RequestMapping(method = RequestMethod.POST,
consumes = MediaType.APPLICATION_JSON_VALUE)
34.     public Pacote create(@RequestBody Pacote pacote) {
35.         return service.create(pacote);
36.     }
37.
38.     @RequestMapping(method = RequestMethod.PUT,
consumes = MediaType.APPLICATION_JSON_VALUE)
39.     public Pacote update(@RequestBody Pacote pacote) {
40.         return service.update(pacote);
41.     }
42.
43.     @RequestMapping(method = RequestMethod.DELETE,
consumes = MediaType.APPLICATION_JSON_VALUE)
44.     public void delete(Long codigo) {
45.         Pacote pacote = service.findByCodigo(codigo);
46.         service.delete(pacote);
47.     }
48.
49. }
```

Nessas declarações podemos encontrar os atributos **method** e **consumes**. O atributo **method** define o método HTTP que dever ser utilizado. Já o atributo **consumes** define o Media Type adotado para receber as requisições.

Desta forma, nosso controller define os endpoints da API conforme detalhado na **Tabela 2**. Com isso, chegamos ao fim da implementação do microservice. Agora, precisamos apenas rodar e testar a aplicação.

Testando o microservice

Com o microservice implementado, podemos enfim inicializar a aplicação. Para isso, através do Gradle, via terminal, execute o seguinte comando na raiz do projeto:

```
gradle run
```

Método HTTP	URL	Descrição
POST	/api/v1/pacotes	Cria um pacote.
GET	/api/v1/pacotes	Obtém uma lista com todos os pacotes cadastrados.
GET	/api/v1/pacotes/:id	Obtém o pacote específico que possui o ID (código) informado.
PUT	/api/v1/pacotes/:id	Atualiza o pacote especificado de acordo com o ID (código) informado.
DELETE	/api/v1/pacotes/:id	Remove o pacote especificado de acordo com o ID (código) informado.

Tabela 2. Operações suportadas pelo controller



Figura 8. Rodando a aplicação

Ao fazer isso, o console deve apresentar algo semelhante ao conteúdo exibido na **Figura 8**, onde são listados os passos do processo de startup da aplicação.

Para que possamos testar nosso microservice sem a necessidade de utilizar o portal ou um cliente para consumir nossa API, iremos utilizar um plugin do Google Chrome – de nome *Advanced REST Client* – que permite realizar todas as operações necessárias.

Para instalar esse plugin, busque pelo mesmo na página do Chrome Web Store e, em seguida, clique em *Usar no Chrome*. Como podemos verificar na **Figura 9**, sua interface é bastante simples. Outro diferencial é que ele fornece algumas opções para facilitar a criação e testes de chamadas HTTP personalizadas.

Para isso, na tela do plugin temos o campo URL, onde informamos o endereço que desejamos acessar, e logo abaixo é possível definir o método HTTP utilizado para a chamada. Com essas opções configuradas, basta clicar no botão *Send* (localizado no final do formulário), para realizar a chamada REST.

Agora que conhecemos o básico da ferramenta, vamos testar o endpoint de listagem de pacotes fornecido pelo microservice. Para tanto, defina no campo URL o endereço <http://localhost:8080/api/v1/pacotes> e selecione o método GET.

Caso tudo ocorra conforme o esperado, devemos receber o status 200. No entanto, como não temos nenhuma promoção na base de dados, nada será retornado no corpo da resposta do cliente REST.

Populando a base de dados

Para que possamos explorar ao máximo nosso microservice, iremos popular a base de dados da aplicação. Isso pode ser feito com a própria API, com a inserção de alguns registros de promoção.

De acordo com o nosso exemplo, esse cadastro ocorre via formulário no portal e é persistido na base de dados do microservice de promoções. Para que não precisemos carregar o portal e ensinar como utilizá-lo, ainda utilizando o plugin, faremos as inclusões executando um POST passando os dados que desejamos incluir.

Deste modo, mantenha o endereço <http://localhost:8080/api/v1/pacotes/> no campo URL, selecione o método HTTP POST (note que apesar de utilizar o mesmo endereço, alteramos o método HTTP), inclua manualmente o JSON da **Listagem 11** no payload da requisição, escolha o valor “application/json” no combobox *Content-Type* localizado abaixo do Payload e, por fim, clique no botão *Send*. Na **Figura 10** podemos ver como configurar o plugin para realizar o POST.

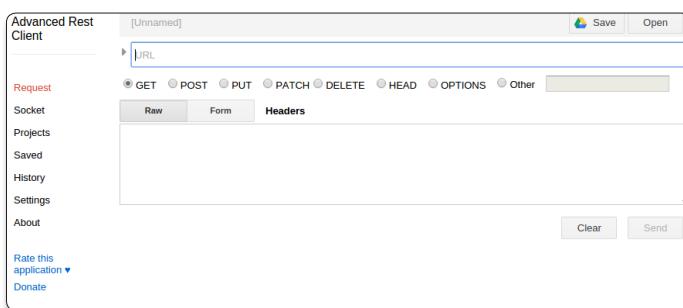


Figura 9. Tela do plugin Advanced REST Client

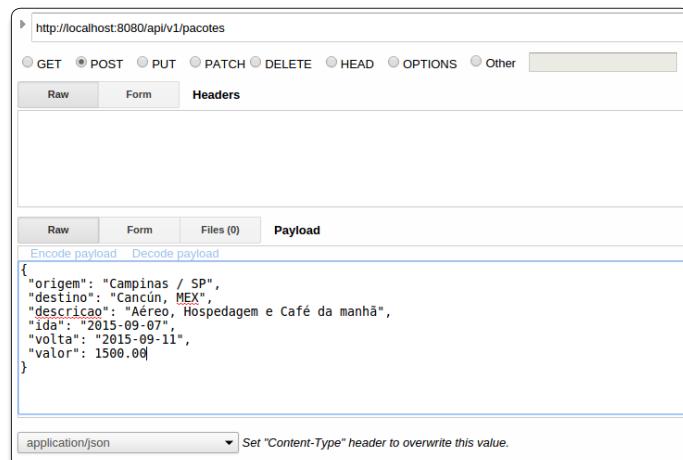


Figura 10. Exemplo utilizando o método POST

Nesse momento, o método responsável por criar uma promoção será executado e uma resposta contendo o objeto recém-criado será retornada.

Agora, repita esse procedimento algumas vezes para cadastrar outros pacotes de promoção. Depois, volte a configurar a ferramenta para listar todos os pacotes e veja a resposta obtida.

Por fim, com base na listagem exibida na **Tabela 2**, teste outros métodos para obter um pacote, atualizar ou remover determinado pacote.

Note que em nosso exemplo implementamos apenas operações do CRUD para os pacotes de promoção. O comum, no entanto, é que outras funcionalidades sejam agregadas, como o ranqueamento de pacotes por visualização ou venda, estatísticas, entre outros, formando assim um microservice mais completo.

O padrão arquitetural de microservices vem conquistando adeptos nos últimos meses, e como verificamos, com todo o sentido. Diante disso, é possível encontrar o tema em destaque em muitos sites renomados e com entusiasmo em eventos de tecnologia. Como um reforço para essa afirmação, hoje, grandes empresas como Netflix, Amazon e eBay já adotam microservices.

No entanto, como demonstrado, existem alguns pontos que devem ser considerados antes da adoção deste novo modelo de desenvolvimento, pesando sempre os prós e contras e o contexto ao qual a solução proposta será direcionada.

Autor



Marcos Alexandre Vidolin de Lima

marcosvidolin@gmail.com / [@marcosvidolin](https://www.linkedin.com/in/marcosvidolin)

É Bacharel em Ciência da Computação, possui certificações em Java e na plataforma Cloud do Google, é entusiasta e amante de novas tecnologias. Atua profissionalmente como desenvolvedor Java EE na Cl&T. Escreve artigos em revistas especializadas como “Java Magazine” e “Easy Java Magazine” e nas horas vagas mantém seu blog pessoal (www.marcosvidolin.com).



Links:

Site do projeto Gradle.

<https://gradle.org/>

Endereço para download do Gradle.

<http://gradle.org/gradle-download/>

Documentação oficial do Spring Boot.

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

Artigo de Martin Fowler sobre Microservices.

<http://martinfowler.com/articles/microservices.html>

Plugin Advanced REST Client do Google Chrome.

<https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfjeloo>

Código do Artigo.

<https://github.com/marcosvidolin/jm-ArquiteturaDeMicroservicesComSpringBoot>

Hibernate Validator: como validar objetos e ter mais controle sobre os dados

Aprenda neste artigo como melhorar a qualidade dos dados que sua aplicação necessita para um correto funcionamento

Avalidação de dados é uma tarefa bastante comum em sistemas onde a entrada de dados é feita principalmente pelo usuário. Em qualquer aplicação, receber dados sempre foi um grande problema, pois não há como prever todas as maneiras que o usuário utilizará para informar seus dados. Por exemplo, caso o campo que receberá uma data não especifique o formato esperado pela aplicação, não há como garantir que o usuário entrará com o dado no formato desejado, o que acaba se tornando um desafio aos programadores.

Diante disso, é necessário realizar o tratamento adequado de todas as informações que são enviadas para evitar que problemas inesperados venham a ocorrer durante o processamento, como exceções que interromperão a execução do sistema, problemas de integridade ou corrupção dos dados e também questões relacionadas à falha de segurança.

Cada informação passada para a aplicação possui um propósito. Assim, é natural que algumas regras sejam introduzidas com o intuito de controlar a interação entre o usuário e o sistema, garantindo deste modo a correta execução das regras de negócio. E com o intuito de garantir o cumprimento dessas regras que o conceito de validação foi implementado. Validar é determinar se os dados capturados pelo sistema seguem as regras lógicas definidas para manter a sua consistência.

Neste contexto, conseguir consolidar e implementar a validação usando uma solução de qualidade como o Hibernate Validator pode melhorar significativamente a confiabilidade do software, especialmente ao longo do tempo, e torná-lo mais amigável ao usuário.

Fique por dentro

Este artigo é útil por apresentar como expressar e validar regras existentes no domínio de negócios de uma aplicação. Para isso, aprenderemos de forma básica e simples os principais conceitos relativos ao Hibernate Validator, fazendo uma análise geral sobre seu funcionamento, como configurar um ambiente de desenvolvimento com Maven, Eclipse, MySQL e Tomcat para uso desta tecnologia e, em seguida, como desenvolver uma aplicação com JSF e o padrão arquitetural MVC que faz uso de seus recursos.

Considerando que as políticas de validação não estruturadas e não controladas vão levar ao aumento dos custos de suporte e manutenção, uma estratégia de validação consolidada pode minimizar significativamente o efeito cascata de mudanças para sua base de código. Além disso, a camada de validação também passa a ser uma ferramenta muito útil durante o processo de depuração, ajudando a localizar defeitos que comprometem o bom funcionamento da aplicação.

Engajado nesta causa, o Hibernate Validator é um projeto *open source* conduzido pela empresa Red Hat que permite a validação dos dados, presentes nas classes que modelam o domínio da aplicação, em qualquer arquitetura (Web, Desktop, etc.) e em tempo de execução. A motivação para a utilização dessa biblioteca é poder validar os dados diretamente no domínio da aplicação, em vez de realizar esse processo por camadas. Dessa forma, é possível validar campos numéricos, definir se datas informadas serão obrigatoriamente maiores ou menores que a data atual, verificar se o campo pode ser vazio ou não diretamente nas classes de domínio de maneira centralizada e flexível, mantendo o código claro e enxuto.

A partir disso, ao longo desse artigo abordaremos os principais conceitos e características do Hibernate Validator, e veremos como desenvolver um sistema de cadastro de projetos considerando como um dos seus principais requisitos não funcionais a confiabilidade do sistema. Para tanto, serão empregados, além do Validator, a linguagem de programação Java, o ambiente de desenvolvimento Eclipse integrado ao Maven, o sistema de gerenciamento de banco de dados MySQL, o framework JSF e o container web Tomcat. Ademais, o Hibernate será usado como solução integrante da camada de persistência, viabilizando a interface entre a aplicação e o MySQL.

Hibernate Validator

O Hibernate é um framework de mapeamento objeto relacional muito popular entre os desenvolvedores Java. Distribuído sob a licença LGPL, foi criado por Gavin King em 2001, sendo atualmente o framework de persistência de dados mais utilizado. Segundo a documentação oficial: “o Hibernate pretende retirar do desenvolvedor cerca de 95% das tarefas mais comuns de persistência de dados”.

Sua principal característica é a transformação de classes Java em representações de tabelas da base de dados (e dos tipos de dados Java para os da SQL). O Hibernate gera os comandos SQL e libera o desenvolvedor do trabalho manual de transformação, mantendo o programa portável para quaisquer bancos de dados SQL.

O Hibernate Validator, por sua vez, é a implementação de referência da *JSR 303 – Bean Validation API*. Disponibilizada em dezembro de 2009, a partir da especificação do Java EE 6, na qual foi introduzida a especificação Bean Validation 1.0, o objetivo principal dessa API é permitir a validação dos dados de forma fácil e rápida, através do uso de anotações e, de forma alternativa, utilizando arquivos XML na configuração.

Com o lançamento mais recente da plataforma Java EE, agora na versão 7, a JSR 349 foi divulgada, introduzindo a versão 1.1 da API de validação e trazendo novidades como:

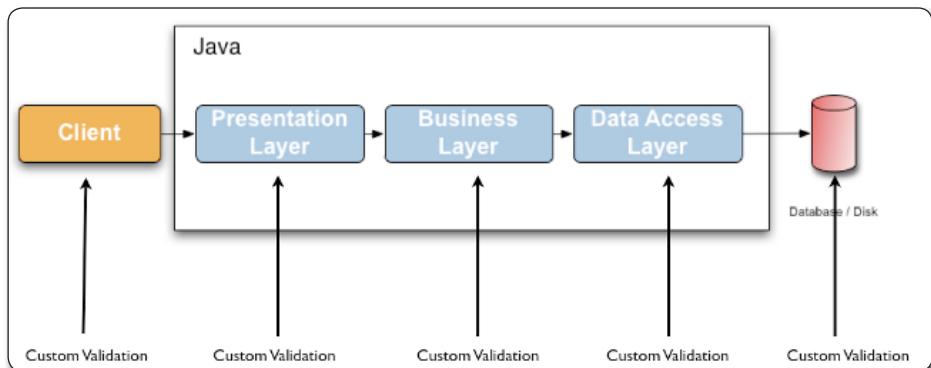


Figura 1. Validação realizada em várias camadas. Fonte: Hibernate Validator Reference Guide

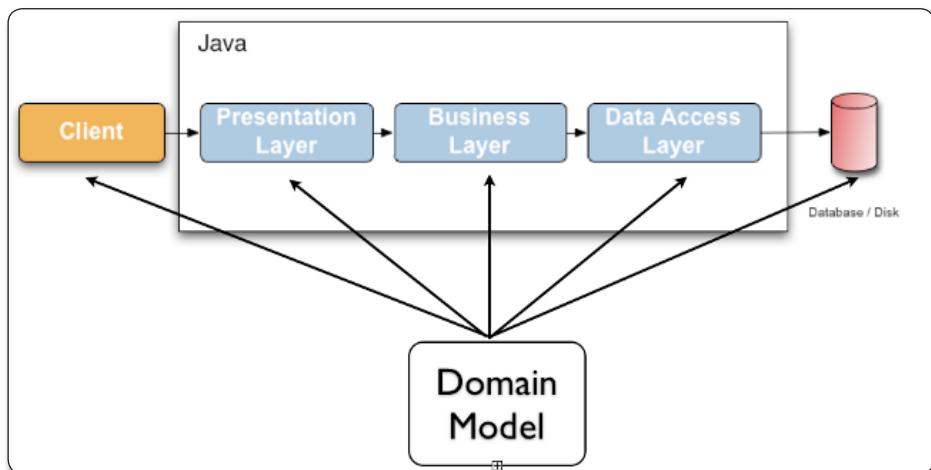


Figura 2. Validação centralizada no modelo de domínio. Fonte: Hibernate Validator Reference Guide

- Uso de injeção de dependências e integração com CDI;
- Validação de parâmetros e retornos de métodos;
- Uso de grupos de conversão;
- Suporte à concatenação de mensagens de violação através de *expression language*;
- Integração com outras especificações, como JAX-RS.

Antes do surgimento dessa API, cada framework implementava um mecanismo proprietário para validar as informações, o que criava problemas de incompatibilidade e dificultava a integração com outros frameworks.

Com o surgimento dessa especificação, possibilitou-se uma API padrão para validação que é flexível e suficiente para ser utilizada pelos mais diversos tipos de frameworks. Além disso, a Bean Validation API viabiliza a validação de dados nas classes do domínio da aplicação, processo

esse que é mais simples do que quando é feito por camada. Na validação por camada, é necessário verificar o mesmo dado diversas vezes (nas camadas de apresentação, negócio, persistência, etc.), a fim de garantir a consistência da informação, conforme mostra a **Figura 1**. Ao validar os dados diretamente nas classes de domínio, o processo todo fica centralizado, uma vez que os objetos destas classes normalmente trafegam entre as camadas da aplicação.

O Hibernate Validator, como principal implementação da Bean Validation API, segue a premissa estabelecida pelo DRY (*Don't Repeat Yourself*), que especifica uma forma de adicionar regras e respectivas verificações para validação automática dos dados, de maneira que estas validações sejam implementadas uma e somente uma vez em toda a aplicação e gerenciadas de maneira centralizada, eliminando a duplicação entre as diversas camadas. A **Figura 2** mostra esta configuração, em que todas as

camadas podem invocar a verificação concentrada em um único lugar. Com isso, evita-se a reescrita de código, uma vez que o mesmo não é mais inserido em diferentes partes da aplicação, e facilita-se a manutenção, proporcionando, simultaneamente, economia no tempo de desenvolvimento.

As validações do framework são definidas através de restrições realizadas diretamente nos Java Beans, especificados no modelo de domínio. Essas restrições são utilizadas para definir regras a respeito dos dados que podem ser atribuídos a um objeto. Assim, quando um processo de validação de dados é executado, é feita uma verificação para checar se os mesmos estão de acordo com as regras estabelecidas.

As restrições são demarcadas através de Java *annotations*, sendo possível adicioná-las tanto nas propriedades do bean quanto nas chamadas de métodos, garantindo que os retornos dos métodos ou os valores dos parâmetros sejam validados. Outra maneira de adicionar regras é diretamente na classe. Neste caso, não apenas uma propriedade é submetida à validação, mas todo o objeto.

Restrições no nível de classe são úteis caso a validação dependa de uma correlação entre várias propriedades de um objeto. Por exemplo, uma classe **Carro** que tenha dois atributos (**quantidadeDeAssentos** e **passageiros**). Deve ser assegurado que a lista de passageiros não tenha mais entradas do que os assentos disponíveis. Dessa forma, o validador da restrição tem acesso ao objeto **Carro**, possibilitando comparar o número de assentos com o número de passageiros.

O Hibernate Validator oferece várias validações, mas não limita o desenvolvedor a elas, ou seja, também possibilita a criação de novas regras, uma vez que nem sempre esses validadores serão suficientes. Por exemplo, a biblioteca dispõe de anotações que verificam a numeração de cartões de crédito, e-mail, URL e, até mesmo, CNPJ, CPF e Título Eleitoral, mas também podem ser criadas novas validações como, por exemplo, para certificar se o valor inserido em um campo é um valor válido para a placa de um carro.

As restrições padrão são:

- **@NotNull**: Verifica se um dado não é nulo;
- **@Null**: Verifica se um dado é nulo;
- **@AssertFalse** e **@AssertTrue**: Checa se o dado é verdadeiro ou falso. Estas validações podem ser aplicadas ao tipo primitivo **boolean** ou à classe **Boolean**;
- **@Min** e **@Max**: Validam o valor mínimo ou o valor máximo para os tipos **BigDecimal**, **BigInteger**, **String**, **byte**, **short** e suas classes wrappers correspondentes;
- **@Size**: Valida se o tamanho do dado está entre os valores especificados nos atributos **min** e **max**. Aplica-se a **Strings**, **Collections** e **Arrays**;
- **@Pattern**: Valida o dado de acordo com uma expressão regular especificada pelo atributo **regexp** da anotação. Funciona somente para dados do tipo **String**;
- **@Past**: Checa se o valor do campo ou propriedade deve ser uma data passada em relação à atual;
- **@Future**: Verifica se o valor do campo ou propriedade deve ser uma data futura em relação à atual;

- **@Valid**: Opção utilizada para validar atributos que referenciam outras classes, impondo uma validação recursiva aos objetos associados.

Cada *annotation* é associada a uma implementação de validação, que verifica se a instância da entidade obedece à regra relacionada. O Hibernate faz isto automaticamente antes que uma inserção ou atualização seja realizada no banco de dados, mas o desenvolvedor também pode chamar a validação a qualquer momento em sua aplicação.

Nota

Todos os elementos do Hibernate Validator (classes, interfaces, annotations, etc.) pertencem ao pacote `javax.validation`. Portanto, sempre que algum elemento da API for referenciado, será necessário realizar o import desse pacote ou mesmo de seus subpacotes.

Configuração do exemplo

Vamos partir para a parte prática e desenvolver uma aplicação que possibilite ao leitor visualizar como o *framework* funciona e sua utilidade.

No entanto, antes de começar a codificar, é importante instalar os softwares que nos auxiliarão nesse trabalho e também preparar o ambiente de desenvolvimento para uso do Validator.

Preparando o ambiente de desenvolvimento

Como IDE (Ambiente de Desenvolvimento Integrado), optamos pelo Eclipse, por se tratar de uma solução extremamente poderosa e flexível, assim como por ser uma das mais utilizadas pelos desenvolvedores.

O processo de instalação do Eclipse é bastante simples, sendo necessário apenas ter um JDK instalado no sistema. Na seção **Links** está disponível o endereço para download do JDK e da IDE, cuja versão adotada no exemplo foi o Eclipse Luna SR2 (4.4.2).

Concluído o download, descompacte-o no seu sistema de arquivos, em um local de sua preferência. Neste artigo optamos pela pasta `C:\Eclipse_Luna`. Em seguida, é preciso apenas executar o arquivo `eclipse.exe`.

Integrando o Maven ao Eclipse

O Maven é uma das ferramentas mais conhecidas e utilizadas por profissionais que adotam o Java em seus projetos. Com o objetivo de simplificar o gerenciamento de dependências e o ciclo de vida do desenvolvimento (compilação, testes unitários, empacotamento e distribuição), esta solução da Apache garante as seguintes metas:

- Prover um padrão para o desenvolvimento de aplicações;
- Fornecer mecanismos para uma clara definição da estrutura do projeto;
- Controlar versão e artefatos;
- Viabilizar/facilitar o compartilhamento de bibliotecas entre projetos.

Como maior atrativo, a principal facilidade viabilizada pelo Maven é o gerenciamento de dependências e este será o nosso motivador para empregá-lo em parceria com o Eclipse.

O endereço para download desta solução encontra-se na seção **Links**. Ao acessá-lo, opte pela versão 3.3.1. Para a instalação, crie uma pasta no sistema de arquivos (`C:\Maven`) e copie o arquivo baixado para dentro dela, descompactando-o em seguida.

Com o objetivo de integrar o Maven ao Eclipse, recomenda-se instalar o plugin M2E. No entanto, como a versão da IDE que estamos utilizando já vem com esse plugin e uma instalação interna do Maven, não precisaremos adicionar o M2E.

Para conhecer a versão do Maven que está configurada, com o Eclipse aberto, acesse o menu `Window > Preferences` e escolha a opção `Maven > Installations`, como sugere a **Figura 3**. Ao fazer isso você poderá notar a presença da versão “embocada”. Apesar disso, vamos optar pela versão do Maven que baixamos anteriormente, por ser mais recente.

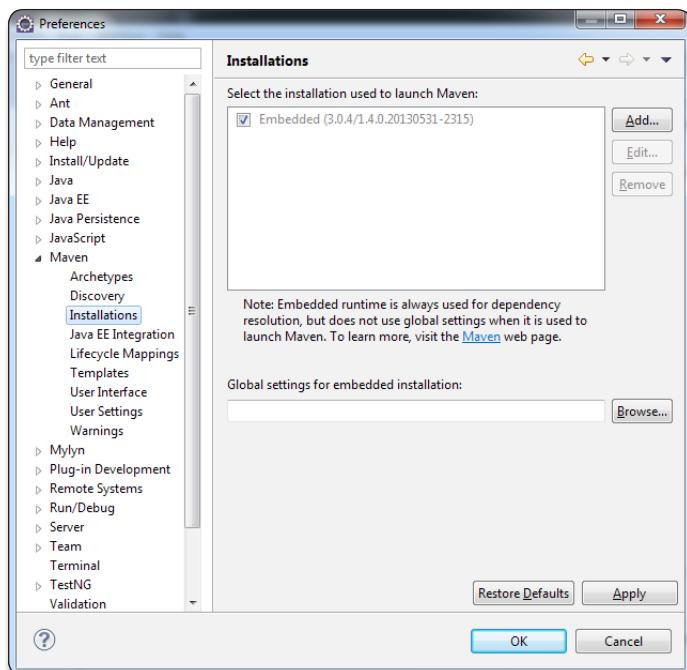


Figura 3. Instalação padrão do Maven no Eclipse

Deste modo, para adicionar o Maven ao Eclipse, ainda na **Figura 3**, clique em `Add` e selecione a pasta `C:\Maven\apache-maven`. Feito isso, automaticamente o arquivo de configuração do Maven é localizado e a versão desejada é adicionada à IDE. Para concluir, clique em `Ok`.

Adicionando o Tomcat ao Eclipse

Para adicionarmos o Tomcat ao Eclipse, é necessário acessar a aba `Servers`, clicar em `Add Server` e selecionar o caminho da instalação deste servidor web, como mostra a **Figura 4**. Além do caminho de instalação do Tomcat, é necessário informar o diretório de instalação do Java. O Tomcat 8.0 exige que a versão utilizada do Java seja igual ou superior a 7.

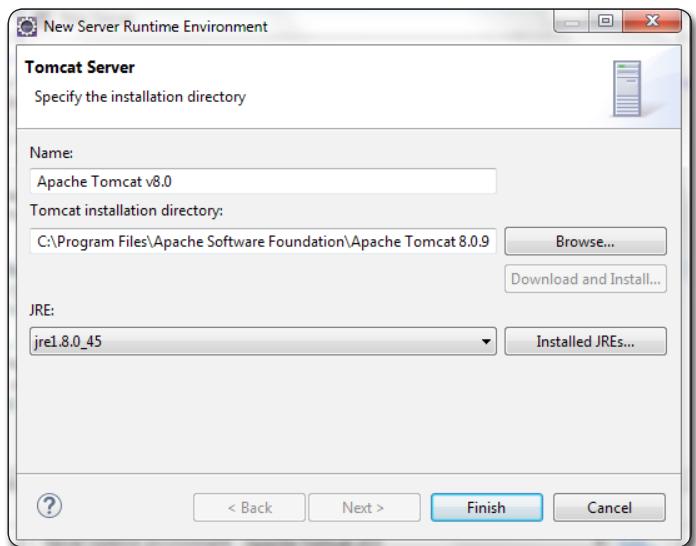


Figura 4. Adicionando o Tomcat ao Eclipse

O sistema gerenciador de banco de dados

O sistema gerenciador de banco de dados que utilizaremos para viabilizar a persistência em nosso exemplo será o MySQL (veja o endereço na seção **Links**). Ao se deparar com as diferentes versões, opte pela adotada neste material, a *Community Server 5.6.22*.

Após baixar o respectivo arquivo, execute os seguintes passos para instalação:

1. Inicie o instalador e aguarde enquanto o sistema operacional o configura;
2. Na tela seguinte, serão exibidos os termos do contrato de licença do produto. Aceite os termos e clique em `Next`;
3. Chegou o momento de especificar o tipo de instalação e depois clicar em `Next`. A escolha vai depender de cada profissional e para que ele vai usar o MySQL. Aqui, optamos pela opção *Custom*;
4. No próximo passo o usuário deve escolher os produtos que deseja instalar. Escolha o *MySQL Server 5.6.22* e clique em `Next`. Feito isso, serão exibidos os produtos selecionados na etapa anterior. Clique então em `Execute` para continuar;
5. No final do processo de instalação, a coluna *Status* passará a exibir a informação “*Complete*”. Assim, pressione `Next` mais duas vezes;
6. Agora o usuário deve escolher o tipo de configuração. Para os nossos objetivos, apenas mantenha os valores padrão e clique em `Next`;
7. Na próxima tela, informe a senha do administrador e continue (`Next`);
8. Chega o momento de configurar o MySQL como um serviço do Windows. Para tanto, apenas mantenha os valores *default* e clique em `Next`;
9. A janela apresentada mostra todas as configurações que serão aplicadas. Confirme as escolhas feitas previamente, clique em `Execute` e, na tela seguinte, em `Next`;
10. Por fim, uma janela confirma a configuração do MySQL. Então, pela última vez, clique em `Next`, e logo depois, em `Finish`.

Desenvolvendo o cadastro de projetos

Agora que temos o banco de dados instalado, assim como o Eclipse e o Maven integrados e o Tomcat incorporado ao Eclipse, vamos partir para a construção de uma solução web lançando mão do JSF 2.2 e do container Tomcat 8.0. O objetivo desta aplicação será gerenciar o cadastro de projetos de uma corporação.

O sistema conterá, para o fácil entendimento desse estudo de caso, apenas uma entidade, de nome **Projeto**, e a partir dela serão construídas as principais operações necessárias para se ter um cadastro: salvar, atualizar, listar e excluir.

Além disso, cada projeto conterá as seguintes propriedades: código, nome, nome do responsável pelo projeto, CPF do responsável, e-mail, data de início, data de fim e descrição. A **Figura 5** mostra a representação gráfica da tabela.

Criando o banco de dados

Com o MySQL instalado, crie o banco *projetobd* executando o script SQL indicado na **Listagem 1**. Neste comando, as duas primeiras linhas especificam a geração do banco de dados e como entrar em seu contexto. A tabela e seus respectivos atributos são especificados na sequência do código.

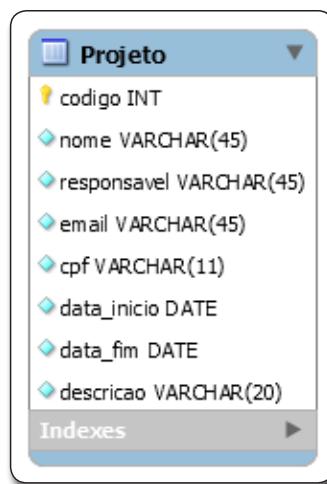


Figura 5. Representação gráfica da tabela projeto

Listagem 1. Script para criação do banco de dados.

```
01. CREATE database projetobd;
02. USE empresabd;
03.
04. CREATE TABLE IF NOT EXISTS projeto (
05.     codigo INT NOT NULL,
06.     nome VARCHAR(45) NOT NULL,
07.     responsavel VARCHAR(45) NOT NULL,
08.     email VARCHAR(45),
09.     cpf VARCHAR(14) NOT NULL,
10.     data_inicio DATE,
11.     data_fim DATE,
12.     descricao VARCHAR(20),
13.     PRIMARY KEY (codigo)
14. ) ENGINE = InnoDB;
```

Criando o projeto Maven no Eclipse

Com todos os recursos instalados, vamos partir para a elaboração do projeto web criando um novo com o auxílio do Maven, no Eclipse. Como utilizaremos anotações JPA, é importante que o projeto adote o Java 1.5+.

Dito isso, com o Eclipse aberto, clique em *New > Other*. Na tela que surgir, selecione *Maven > Maven Project*, marque a opção *Create a simple project (skip archetype selection)* e pressione *Next*.

Na próxima tela, devemos identificar o projeto, o que é feito ao especificar estas opções e clicar em *Finish*:

- Group ID: **br.com.devmedia**;
- Artifact ID: **hibernate-validator**;
- Packaging: **war**;
- Version: **0.0.1-SNAPSHOT**.

Para completar a configuração inicial do projeto, vamos determinar alguns parâmetros adicionais para que a aplicação seja executada conforme desejamos. Assim, clique com o botão direito na raiz do projeto e depois na opção *Properties*.

Na nova tela, selecione a opção *Project Facets*. A partir disso, vamos ajustar as configurações relacionadas às versões do Java, JSF e do módulo web. Portanto, selecione a versão 3.1 para *Dynamic Web Module*; na opção *Java*, escolha a versão 1.8; e para o JavaServer Faces opte pela versão 2.2.

Apesar desses ajustes, observe que a estrutura do projeto ainda não está definida com as configurações de um projeto Maven. Para isso, é necessário atualizá-lo. Portanto, clique com o botão direito sobre o projeto e accese o menu *Maven > Update Project*. Na tela que aparecer, selecione a opção *hibernate-validator*, que acabamos de criar, e clique em *Ok* para atualizá-lo de acordo com as definições do Maven. Assim, o projeto passará a ter a estrutura apresentada na **Figura 6**.

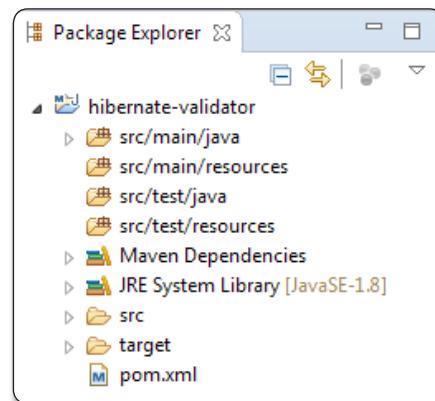


Figura 6. Visão Package Explorer da estrutura de diretórios do projeto

Adicionando as dependências

Após criar o projeto devemos inserir as dependências, isto é, as bibliotecas que serão utilizadas pela aplicação. Para isso, devemos fazer ajustes que refletirão no conteúdo do arquivo *pom.xml*, que é o principal arquivo do Maven e que contém todas as configurações relacionadas ao projeto.

Para inserir as dependências, clique com o botão direito sobre o POM e acesse *Maven > Add Dependency*. Na nova janela, informe o nome das bibliotecas que deseja utilizar (Hibernate Core, Hibernate Validator, JSF e o driver do MySQL), como exemplificado na **Figura 7**, e clique em *Ok*.

Concluída essa etapa, o *pom.xml* terá o conteúdo mostrado na **Listagem 2**.

Selecionadas as dependências, após clicar com o botão direito do mouse sobre o projeto, acesse *Maven > Update Project* e automaticamente as bibliotecas serão incorporadas ao *classpath* do projeto.

Criando o modelo e inserindo as anotações de validação

Nosso próximo passo é criar e mapear a entidade *projeto*, assim como definir as restrições que desejamos. Deste modo, crie a classe **Projeto**, classe de negócios que será mapeada pelo Hibernate como uma tabela no banco de dados. Ademais, como será possível notar em seu código, **Projeto** é um tipo de classe simples, definida como um POJO, que contém todos os atributos encapsulados através de métodos de acesso e disponibiliza o construtor padrão.

Antes de implementar essa classe, no entanto, crie o pacote **br.com.jm.projeto.model**. Para isso, com o botão direito do mouse sobre o projeto *hibernate-validator*, acesse *New > Package* e informe seu nome. Em seguida, novamente com o botão direito do mouse, acesse *New > Class*. Na tela que surgir, informe “Projeto” no campo nome. A **Listagem 3** mostra o código dessa classe.

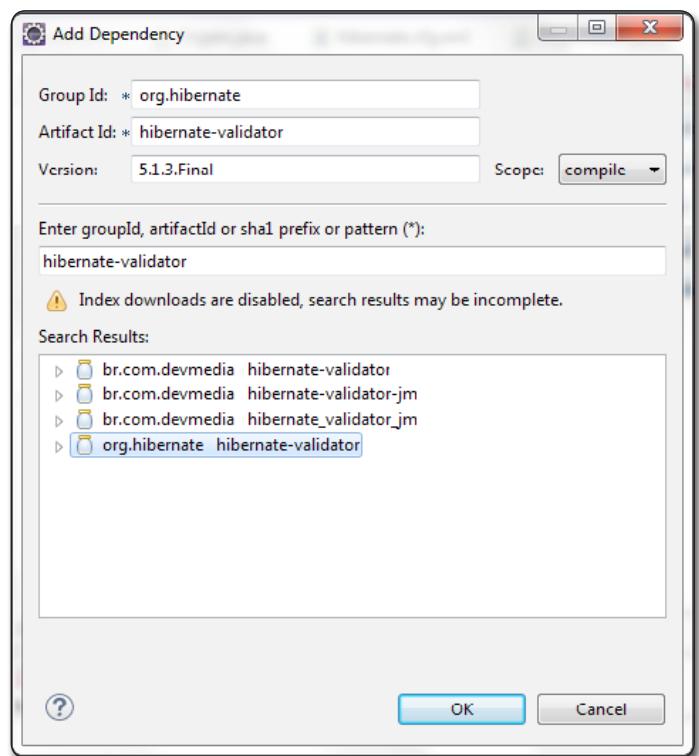


Figura 7. Adicionando dependências no arquivo pom.xml

Listagem 2. Configuração do arquivo pom.xml.

```

01. <project xmlns="http://maven.apache.org/POM/4.0.0"
02.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04.     http://maven.apache.org/xsd/maven-4.0.0.xsd">
05.   <modelVersion>4.0.0</modelVersion>
06.   <groupId>br.com.devmedia</groupId>
07.   <artifactId>hibernate-validator</artifactId>
08.   <version>0.0.1-SNAPSHOT</version>
09.   <packaging>war</packaging>
10.  <dependencies>
11.    <dependency>
12.      <groupId>javax</groupId>
13.      <artifactId>javaee-api</artifactId>
14.      <version>7.0</version>
15.    </dependency>
16.    <dependency>
17.      <groupId>javax.servlet</groupId>
18.      <artifactId>javax.servlet-api</artifactId>
19.      <version>3.1.0</version>
20.      <scope>provided</scope>
21.    </dependency>
22.    <dependency>
23.      <groupId>org.hibernate</groupId>
24.      <artifactId>hibernate-core</artifactId>
25.      <version>4.3.10.Final</version>
26.    </dependency>
27.    <dependency>
28.      <groupId>com.sun.faces</groupId>
29.      <artifactId>jsf-api</artifactId>
30.      <version>2.2.4</version>
31.    </dependency>
32.    <dependency>
33.      <groupId>com.sun.faces</groupId>
34.      <artifactId>jsf-impl</artifactId>
35.      <version>2.2.9</version>
36.    </dependency>
37.    <dependency>
38.      <groupId>mysql</groupId>
39.      <artifactId>mysql-connector-java</artifactId>
40.      <version>5.1.34</version>
41.    </dependency>
42.    <dependency>
43.      <groupId>org.hibernate</groupId>
44.      <artifactId>hibernate-validator</artifactId>
45.      <version>5.1.3.Final</version>
46.    </dependency>
47.  </dependencies>
48.  <build>
49.    <plugins>
50.      <plugin>
51.        <groupId>org.apache.maven.plugins</groupId>
52.        <artifactId>maven-compiler-plugin</artifactId>
53.        <version>3.1</version>
54.        <configuration>
55.          <source>1.8</source>
56.          <target>1.8</target>
57.        </configuration>
58.      </plugin>
59.    </plugins>
60.  </build>
61. </project>
```

Hibernate Validator: como validar objetos e ter mais controle sobre os dados

Listagem 3. Código da classe Projeto.

```
01. package br.com.jm.projeto.model;
02.
03. //imports omitidos
04.
05. @Entity
06. @Table
07. public class Projeto implements Serializable {
08.
09.     private static final long serialVersionUID = 1L;
10.
11.     public Projeto() {}
12.
13.     @Id
14.     @Column
15.     @GeneratedValue(strategy = GenerationType.AUTO)
16.     private int codigo;
17.
18.     @NotEmpty(message="Informe o nome do Projeto")
19.     @Column
20.     private String nome;
21.
22.     @NotEmpty(message="Informe o nome do Responsável pelo Projeto")
23.     @Column
24.     private String responsavel;
25.
26.     @CPF(message="CPF inválido")
27.     @NotEmpty(message="Informe o número do CPF")
28.     @Column
29.     private String cpf;
30.
31.     @Size(max = 10, message="Descrição deve ter no máximo {max} caracteres. Você digitou: " + ${validatedValue}")
32.     @Column
33.     private String descricao;
34.
35.     @Pattern(regexp = "[^\\@]+(\\.[^\\@]+)*@[\\w\\.-]+\\.(\\.[\\w\\.-]+)*",
36.             message = "Email inválido")
37.     @Column
38.     private String email;
39.
40.     @Future(message="A data deve ser maior do que a atual.
41. Você digitou: " + ${validatedValue}")
42.     @Temporal(TemporalType.DATE)
43.     @Column
44.     private Date dataInicio;
45.
46.     @Temporal(TemporalType.DATE)
47.     @Column
48.     private Date dataFim;
49.
50.     public Date getDataFim() {
51.         return dataFim;
52.     }
53.     public void setDataFim(Date dataFim) {
54.         this.dataFim = dataFim;
55.     }
56.
57.     public boolean isFimdeSemana(@NotEmpty(message =
58.         "Data de término não pode ser nulo")Date dataTermino) {
59.         return true;
60.     }
61.     //gets e sets omitidos
```

O leitor mais atento identificará em **Projeto** anotações referentes ao Hibernate e ao Hibernate Validator. Logo no início, na linha 5, declaramos **@Entity**. Esta anotação sinaliza que haverá uma tabela relacionada a essa classe no banco de dados e que os objetos desta classe serão persistidos. Já as anotações **@Column** e **@Table** são usadas para indicar que os campos representam colunas e tabelas, respectivamente, no banco de dados.

A anotação **@Id** (linha 13), por sua vez, informa qual atributo será mapeado como chave primária da tabela. Na linha 15 verificamos também a anotação **@GeneratedValue**, que geralmente acompanha **@Id** e que serve para indicar que o valor do atributo que define a chave primária deve ser criado pelo banco ao persistir o registro. E a anotação **@Temporal(TemporalType.DATE)**, presente nas linhas 40 e 44, especifica que os campos **dataInicio** e **dataFim** irão trabalhar com valores no formato de uma data.

As demais anotações são referentes ao processo de validação e serão explicadas a seguir:

- **Linhas 18, 22 e 27:** A anotação **@NotEmpty** é direcionada para atributos do String, Collection, Map ou Array e verifica se qualquer um desses não é nulo e nem vazio;
- **Linha 26:** A anotação **@Cpf**, inserida na versão 5.0.0 Alpha1 do Hibernate Validator, verifica se o valor informado corresponde a um CPF válido, de acordo com as regras nacionais;
- **Linha 31:** A anotação **@Size** validará a **String** para verificar se seu tamanho tem no máximo 10 caracteres. Observe neste caso

que o atributo **message**, que personaliza a mensagem de erro a ser exibida, possui a expressão **max** entre chaves. Este é um recurso da especificação que permite inserir na mensagem de erro os valores dos próprios atributos das anotações. Além disso, concatenamos a mensagem de erro com a variável **validatedValue**, recuperada via *Expression Language*, para exibir o valor digitado pelo usuário. A concatenação com *EL* é um novo recurso implementado na versão 1.1 de Bean Validation;

- **Linha 35:** A anotação **@Pattern** é utilizada quando se deseja checar um campo a partir de uma expressão regular. Nesse caso foi criada uma expressão regular para verificar se a informação repassada corresponde a um e-mail válido. Além da expressão regular, outra forma de verificar se o endereço de e-mail informado é válido é através da anotação **@Email**;
- **Linhas 39 e 48:** A anotação **@Future** verifica se a data informada é posterior à data atual.

Vale ressaltar que a validação também pode ser realizada em métodos de acesso. A linha 48 ilustra essa situação, onde a validação é feita diretamente no método **getDataFim()**.

Outro recurso disponível é a validação de parâmetros passados para os métodos. Para demonstrar esse recurso, implementamos o método de negócio **isFimdeSemana()** na classe **Projeto** para verificar se a data cadastrada como término do projeto refere-se a um dia de final de semana. Na assinatura desse método, observe

que antes do parâmetro **dataTermino**, colocamos uma validação especificando que o valor não pode ser vazio e uma mensagem de erro.

Criando o Controller da aplicação

A camada de controle do JSF é composta pelos Managed Beans, elementos responsáveis por controlar o fluxo de processamento e estabelecer a ligação entre nosso modelo e a camada de visão. Sabendo disso, para criar o nosso bean gerenciado, clique com o botão direito do mouse sobre o projeto, selecione *New > Class* e nomeie a classe como **ProjetoBean**. O seu código é apresentado na **Listagem 4**, e como verificado, deve ficar no pacote **br.com.jm.projeto.managedbean**.

Listagem 4. Código da classe ProjetoBean.

```
01. package br.com.jm.projeto.managedbean;
02.
03. //imports omitidos
04.
05. @ManagedBean
06. @RequestScoped
07. public class ProjetoBean implements Serializable{
08.
09.     private static final long serialVersionUID = 1L;
10.     private Projeto projeto = new Projeto();
11.
12.     public String salvar(){
13.         ProjetoDao dao = new ProjetoDaolmp();
14.         dao.save(projeto);
15.         FacesContext.getCurrentInstance().addMessage(
16.             null,
17.             new FacesMessage(FacesMessage.SEVERITY_INFO,
18.                 "Manutenção de projeto:",
19.                 "projeto incluido com sucesso!"));
20.         return "sucesso";
21.     }
22.     //métodos get e set omitidos
23. }
```

Na linha 6, com a anotação **@RequestScoped**, definimos que esse *Managed Bean* terá escopo de requisição. Na linha 12, declaramos uma variável de instância do tipo **Projeto** e na linha 14 implementamos o método **salvar()**, que realiza a validação do objeto **Projeto** e direciona o fluxo da aplicação para alguma página – neste caso, a página *sucesso.xhtml*, que criaremos em breve.

Criando a camada View da aplicação

O próximo passo será criar a página web por onde serão especificadas as informações do projeto. Essa página se comunica diretamente com os métodos e atributos da classe **ProjetoBean**.

Portanto, clique com o botão direito do mouse sobre o projeto, selecione *New > HTML File*, nomeie o arquivo como *project_form.xhtml* e clique em *Next*. Na lista que aparece, escolha o tipo de *template HTML (xhtml 1.0 strict)* e clique em *Finish*. A página deve ficar com o código semelhante ao apresentado na **Listagem 5**.

Neste arquivo podemos ver um formulário simples contendo campos de texto e um botão que chama o método **salvar()** do *Managed Bean* (linha 28). Na linha 7 observe a presença da

tag **<f: validateBean>**. Esta tem como objetivo integrar o *Bean Validation* ao JSF. Assim, sinalizamos que todos os campos que estiverem dentro dela serão checados. Outra tag importante é a **<h:messages>**, vista na linha 5, que é responsável por mostrar na tela os possíveis erros que ocorrerão na validação.

Para concluir a implementação da *view*, vamos criar a página que será exibida caso a operação de persistência seja realizada com sucesso, ou seja, o preenchimento do formulário não viole as restrições aplicadas à classe **Projeto**. O código da página *sucesso.xhtml* pode ser visto na **Listagem 6**.

Listagem 5. Formulário de entrada de dados.

```
01. <h:body>
02.   <h2>Preencha o formulário abaixo</h2>
03.   <f:view>
04.     <h:form id="frmProjeto" method="post">
05.       <h:messages style="color:red;margin:8px;" />
06.       <h:panelGrid columns="2" style="horizontal-align:center">
07.         <f:validateBean>
08.           <h:outputText value="Nome do Projeto:" />
09.           <h:inputText value="#{projetoBean.projeto.nome}" />
10.           <h:outputText value="Responsável:" />
11.           <h:inputText value="#{projetoBean.projeto.responsavel}" />
12.           <h:outputText value="CPF do Responsável:" />
13.           <h:inputText value="#{projetoBean.projeto.cpf}" />
14.           <h:outputText value="Email do Responsável:" />
15.           <h:inputText value="#{projetoBean.projeto.email}" />
16.           <h:outputText value="Data de Inicio do Projeto:" />
17.           <h:inputText value="#{projetoBean.projeto.dataInicio}">
18.             <f:convertDateTime pattern="dd/MM/yyyy" type="date"/>
19.           </h:inputText>
20.           <h:outputText value="Data de Término do Projeto:" />
21.           <h:inputText value="#{projetoBean.projeto.dataFim}">
22.             <f:convertDateTime pattern="dd/MM/yyyy" type="date"/>
23.           </h:inputText>
24.           <h:outputText value="Descrição:" />
25.           <h:inputTextarea value="#{projetoBean.projeto.descricao}" />
26.         </f:validateBean>
27.       </h:panelGrid>
28.       <h:commandButton action="#{projetoBean.salvar}" value="Enviar" />
29.       <input type="reset" value="Limpar" />
30.     </h:form>
```

Listagem 6. Código da página *sucesso.xhtml*.

```
01. <h:body>
02.   <f:view>
03.     Cadastro realizado com sucesso
04.   </f:view>
05. </h:body>
```

Configurando o Hibernate e a aplicação

Nesse momento vamos voltar nossa atenção à configuração do *Hibernate*, onde devemos informar os detalhes para acesso ao banco de dados, assim como realizar o mapeamento da nossa classe de domínio, o que deve ser feito em um arquivo XML.

Sendo assim, crie o arquivo *hibernate.cfg.xml* clicando com o botão direito do mouse sobre a pasta *src/main/resources* e selecionando *Novo > Documento XML*. O arquivo criado deve ser parecido com o que mostra a **Listagem 7**.

Hibernate Validator: como validar objetos e ter mais controle sobre os dados

Listagem 7. Conteúdo do arquivo hibernate.cfg.xml.

```
01. <hibernate-configuration>
02.   <session-factory>
03.     <property name="hibernate.dialect">
04.       org.hibernate.dialect.MySQLDialect</property>
05.     <property name="hibernate.connection.driver_class">
06.       com.mysql.jdbc.Driver</property>
07.     <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
08.       empresabd?zeroDateTimeBehavior=convertToNull</property>
09.     <property name="hibernate.connection.username">root</property>
10.    <property name="hibernate.connection.password">1234</property>
11.    <property name="hibernate.show_sql">true</property>
12.    <mapping class="com.jm.entidade.Projeto"/>
13.  </session-factory>
14. </hibernate-configuration>
```

As propriedades que configuramos são explicadas a seguir:

- **dialect (linha 5):** define o dialeto/linguagem com o qual o Hibernate se comunicará com a base de dados;
- **connection.driver_class (linha 6):** configura a classe do driver JDBC;
- **connection.url (linha 7):** determina a URL de conexão com o banco de dados;
- **connection.username (linha 8):** local onde deve ser informado o nome do usuário para conexão com o banco;
- **connection.password (linha 9):** local onde deve ser informada a senha;
- **show_sql (linha 10):** opção que possibilita visualizarmos o script SQL gerado pelo Hibernate;
- **mapping (linha 11):** local onde devemos indicar a(s) classe(s) que está(ão) mapeada(s) para viabilizar as operações de persistência/consulta.

Listagem 8. Conteúdo do arquivo web.xml.

```
01. <welcome-file-list>
02.   <welcome-file>project_form.xhtml</welcome-file>
03. </welcome-file-list>
04. <context-param>
05.   <param-name>javax.faces.VALIDATE_EMPTY_FIELDS</param-name>
06.   <param-value>true</param-value>
07. </context-param>
08. <servlet>
09.   <servlet-name>Faces Servlet</servlet-name>
10.  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
11.  <load-on-startup>1</load-on-startup>
12. </servlet>
13. <servlet-mapping>
14.   <servlet-name>Faces Servlet</servlet-name>
15.   <url-pattern>*.xhtml</url-pattern>
16. </servlet-mapping>
```

Outro arquivo necessário é o *web.xml*, que contém as demais configurações do nosso projeto. Para criá-lo, clique com o botão direito do mouse sobre a pasta *WEB-INF*, selecione *Novo > Documento XML* e defina seu nome. A **Listagem 8** mostra o conteúdo desse arquivo.

Note que dentro da tag *<welcome-file>*, na linha 2, é definida a página que será tida como inicial pela aplicação. Já entre as

linhas 8 e 12 é configurado o *servlet* do JSF, e entre as linhas 13 e 16 é especificado o padrão de URL através do qual o Servlet, dado em *<servlet-name>*, pode ser acessado.

Já o parâmetro *javax.faces.VALIDATE_EMPTY_FIELDS*, vide linha 5, quando configurado como *true*, força o JSF a acionar a validação de campos vazios, pois por padrão o JSF trata os campos vazios como nulos.

Criando a conexão com o banco de dados

Agora devemos criar dentro do pacote *br.com.jm.projeto.util*, a classe auxiliar *HibernateUtil*, responsável por carregar as configurações do *hibernate.cfg.xml* e implementar os métodos de controle das conexões e transações com o banco. Seu código fonte pode ser visto na **Listagem 9**.

Observe que nessa classe temos apenas o atributo *sessionFactory* e o método estático *getSessionFactory()*, que cria uma *SessionFactory* para o Hibernate de acordo com o arquivo de configurações. A partir disso, é possível instanciar objetos do tipo *org.hibernate.Session* que, por sua vez, serão utilizados para realizar as tarefas de persistência do framework, como apresentado a seguir.

Listagem 9. Código da classe HibernateUtil.

```
01. package util;
02.
03. //imports omitidos
04.
05. public class HibernateUtil {
06.
07.   private static SessionFactory sessionFactory;
08.
09.   public static SessionFactory getSessionFactory() {
10.     if (sessionFactory == null) {
11.       Configuration configuration = new Configuration().configure();
12.       ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
13.         .applySettings(configuration.getProperties()).build();
14.       sessionFactory = configuration.buildSessionFactory(serviceRegistry);
15.       SchemaUpdate se = new SchemaUpdate(configuration);
16.       se.execute(true, true);
17.     }
18.
19.     return sessionFactory;
20.   }
21.
22. }
```

Persistindo a entidade Projeto

Com a classe *Projeto* mapeada, os arquivos de configuração definidos e a classe auxiliar implementada, vamos codificar as operações de persistência. Para isso, crie a interface *ProjetoDao* em um pacote de nome *br.com.jm.projeto.dao*.

Para criar o pacote, clique com o botão direito do mouse sobre o projeto *hibernate-validator*, escolha *New > Package* e informe seu nome. Em seguida, clique sobre ele, novamente com o botão direito, acesse *New > Interface* e dê o nome de *ProjetoDao*. O código fonte deve ficar semelhante ao apresentado na **Listagem 10**. Nele, podemos verificar todas as operações que serão realizadas sobre o banco de dados.

Feito isso, precisamos criar a classe para implementar essa interface. Assim, com o botão direto sobre o pacote **br.com.jm.projeto.dao**, acesse *New > Class* e nomeie como **ProjetoDaoImp** (vide **Listagem 11**).

Listagem 10. Código da interface ProjetoDao.

```
01. package br.com.jm.projeto.dao;
02.
03. //imports omitidos
04.
05. public interface ProjetoDao {
06.     public void save(Projeto projeto);
07.     public Projeto getProjeto(long id);
08.     public List<Projeto> list();
09.     public void remove(Projeto projeto);
10.    public void update(Projeto projeto);
11. }
```

Listagem 11. Código da classe ProjetoDaoImp.

```
01. package br.com.jm.projeto.dao;
02.
03. //imports omitidos
04.
05. public class ProjetoDaoImp implements ProjetoDao {
06.
07.     public void save(Projeto projeto) {
08.         Session session = HibernateUtil.getSessionFactory().openSession();
09.         Transaction t = session.beginTransaction();
10.         session.save(projeto);
11.         t.commit();
12.     }
13.
14.     public Projeto getProjeto(long id) {
15.         Session session = HibernateUtil.getSessionFactory().openSession();
16.         return (Projeto) session.load(Projeto.class, id);
17.     }
18.
19.     public List<Projeto> list() {
20.         Session session = HibernateUtil.getSessionFactory().openSession();
21.         Transaction t = session.beginTransaction();
22.         List lista = session.createQuery("from Projeto").list();
23.         t.commit();
24.         return lista;
25.     }
26.
27.     public void remove(Projeto projeto) {
28.         Session session = HibernateUtil.getSessionFactory().openSession();
29.         Transaction t = session.beginTransaction();
30.         session.delete(projeto);
31.         t.commit();
32.     }
33.
34.     public void update(Projeto projeto) {
35.         Session session = HibernateUtil.getSessionFactory().openSession();
36.         Transaction t = session.beginTransaction();
37.         session.update(projeto);
38.         t.commit();
39.     }
40. }
```

- Linha 9: A operação **beginTransaction()** inicia uma transação com o banco de dados;
- Linha 10: O método **save()** realiza a persistência do objeto;
- Linha 11: O método **commit()** finaliza a transação e a sessão é encerrada.

Os demais métodos dessa classe (**list()**, **remove()** e **update()**) são semelhantes ao **save()** e por isso analisaremos aqui apenas o que há de diferente no código de cada um. Vejamos:

- Linha 22: O método **createCriteria()** especifica uma query para recuperar todos os objetos do tipo **Projeto** presentes no banco de dados;
- Linha 30: O método **delete()** remove o objeto passado como parâmetro;
- Linha 37: O método **update()** realiza a atualização do objeto passado como parâmetro.

Testando a aplicação

Enfim, chegamos ao momento de testar a aplicação. Para isso, clique com o botão direto sobre o projeto, acesse *Run As > Run on Server*, escolha o servidor Tomcat e clique em *Finish*.

Ao iniciar a aplicação, a página inicial será exibida. Para testar a implementação, clique no botão *Enviar* sem preencher qualquer campo. Desta forma, podemos verificar se a validação de preenchimento obrigatório está funcionando.

A **Figura 8** mostra o resultado desse teste. Note que como nenhuma informação foi inserida nos campos do formulário, o sistema alerta o usuário com algumas mensagens de erro.

Figura 8. Formulário de cadastro de projeto em branco

Vejamos uma análise das linhas mais importantes dessa classe:

- Linha 7: O método **save()** recebe como parâmetro os dados do projeto e realiza a inserção deste no banco de dados;
- Linha 8: A variável do tipo **Session** recebe o resultado da chamada ao método **openSession()**, que é chamado a partir do retorno do método **getSessionFactory()**;

Já em situações em que todos os campos são preenchidos conforme o esperado e, portanto, nenhum erro de validação é registrado (vide **Figura 9**), o projeto é internalizado no banco de dados e o usuário é redirecionado para a página de sucesso (**Figura 10**).

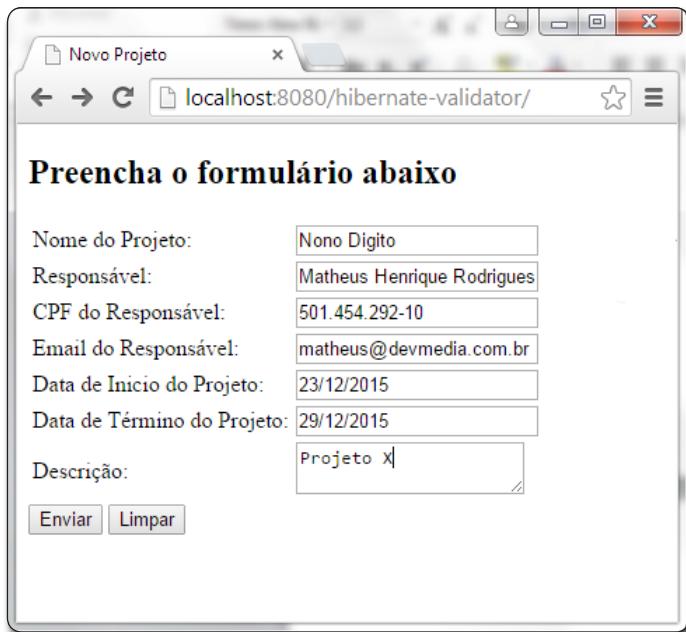


Figura 9. Formulário de cadastro de projeto preenchido corretamente

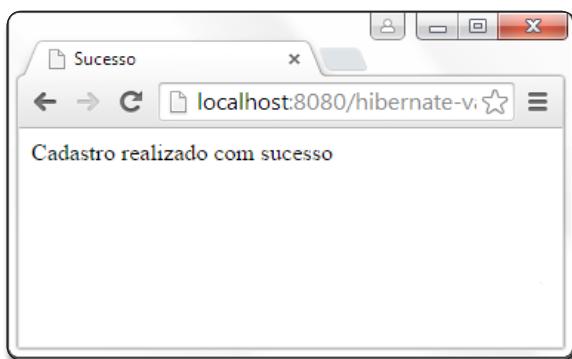


Figura 10. Página indicando o sucesso do cadastro

Uma das grandes vantagens da Bean Validation API, e consequentemente do Hibernate Validator, é o fato da validação ocorrer no modelo de domínio da aplicação, de forma centralizada, evitando assim a reescrita de código e tornando-o mais legível, o que facilita o trabalho do desenvolvedor durante tarefas de implementação de novos recursos e manutenção.

Por fim, vale ressaltar que diversos frameworks para desenvolvimento web já fazem integração com a Bean Validation, como é o caso do próprio Hibernate, utilizado em nosso exemplo, do JPA, Spring e JSF. Isso mostra a importância, o alcance e o nível de flexibilidade da API.

Autor



Carlos Alberto Silva
casilvamg@hotmail.com
É formado em Ciéncia da Computação pela Universidade Federal de Uberlândia (UFU), com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI) e em Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial pelo Instituto Federal do Triângulo Mineiro (IFTM). Trabalha atualmente na empresa Algar Telecom como Analista de TI. Possui as seguintes certificações: OCJP, OCWCD e ITIL.



Links:

Site do Hibernate.
<http://hibernate.org>

Site do Hibernate Validator.
<http://hibernate.org/validator/>

Site do MySQL.
<http://www.mysql.com/>

Endereço para download do driver do MySQL.
<http://dev.mysql.com/downloads/connector/j/>

Endereço para download do Eclipse.
<https://eclipse.org/downloads/>

Endereço para download do JDK.
<http://www.oracle.com/technetwork/java/javase/downloads>

Endereço para download do Maven.
<http://maven.apache.org/download.html>

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.

Suporte 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