Capítulo 6

# Integração de software

Uma das etapas mais importantes quando se opta por abordar uma arquitetura voltada a componentes ou a serviços é a parte de integração de cada uma dessas partes que compõem o sistema como um todo. A maneira mais tradicional de garantir essa integração entre os sistemas e componentes é por meio de uma interface de programação de aplicação (API – application programming interface, em inglês). Dessa maneira, podemos construir portas de acesso ao nosso sistema com a vantagem de garantir quem irá acessá-lo e como se dará esse acesso (GONÇALVES, 2007).

Neste capítulo, falaremos um pouco mais sobre a biblioteca JAX-RS, que permite a construção de uma API utilizando o padrão REST através de projetos Java. Esses serviços são chamados de web services, e iremos aprender como implementar os quatro métodos principais de um sistema de cadastro utilizando a temática de serviços.

# 1 API e o desafio na integração de software

API é um acrônimo da língua inglesa que significa application programming interface (em português, interface de programação de aplicação). Isso nada mais é do que um conjunto de rotinas e padrões estabelecidos, implementados e documentados por determinada aplicação e que possam ser usados por demais aplicações. Dessa maneira, nenhum detalhe de implementação fica exposto, sendo apenas necessário o desenvolvedor saber o que deve ser enviado e o retorno dessa rotina (GONÇALVES, 2007).

**PARA SABER MAIS**

Como já vimos no capítulo 1, esse tipo de implementação permite a interoperabilidade entre os sistemas, através da comunicação entre eles. São alguns exemplos de API: Twitter Developers, Marvel Developers, Google Developers e Facebook Developers.

Para exemplificar essa integração de software, vamos construir uma aplicação com o padrão REST ou API RESTful.

## 1.1 Construindo uma aplicação com o padrão REST

Um cenário que podemos construir que é muito utilizado no dia a dia digital é o carrinho de compras de um comércio eletrônico. Este

geralmente está associado a uma lista de produtos e, além da identificação do comprador, está vinculado a determinado endereço. Vamos construir nosso serviço com base nesse cenário.

### 1.1.1 Criando o primeiro recurso

Como apresentado na introdução, toda ação disponibilizada através de uma arquitetura REST é chamada de recurso. O recurso é identificado através de uma URI (uniform resource identifier), que, em linhas gerais, pode ser vista como o caminho de acesso àquela ação.

Para implementarmos isso com o JAX-RS, precisamos criar uma classe. Essa classe deve ficar dentro do pacote resource e, como boa prática, o sufixo de seu nome também leva o nome de resource. Vamos criar agora o *CarrinhoResource*. A princípio, vamos criar um método para buscar o carrinho cadastrado no banco. Nesse momento, utilizaremos o id fixo, conforme o código:

```
import com.andreymasiero.loja.dao.CarrinhoDAO;
import com.andreymasiero.loja.model.Carrinho;

public class CarrinhoResource {

	public String busca() {
		Carrinho carrinho = new CarrinhoDAO().
busca(1l);
		return carrinho.toXML();
	}
}
```

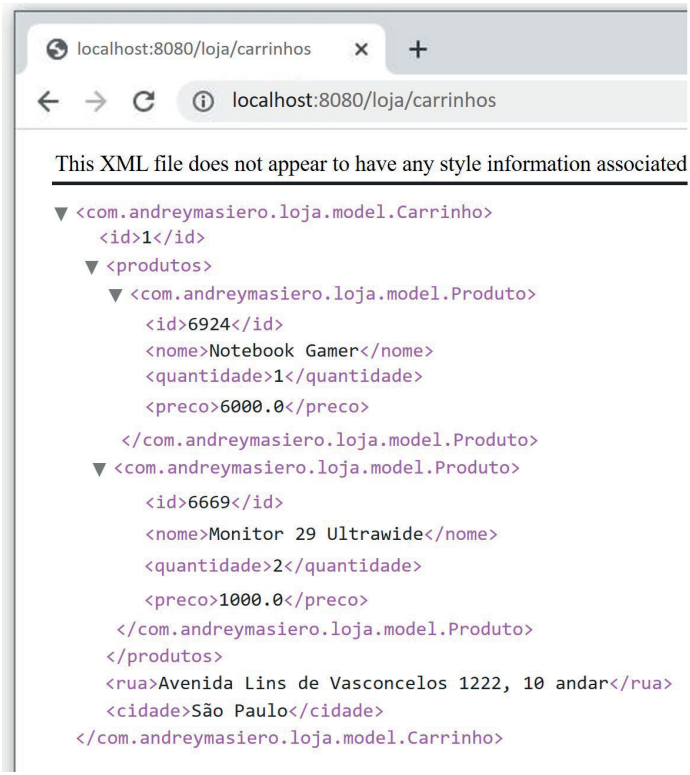
Com a classe criada, precisamos começar a trabalhar com as anotações do JAX-RS para configurar nosso serviço. A primeira anotação é *Path*, que será utilizada nesse momento na classe para determinar a URI do serviço geral.

Na sequência, dizemos, com a anotação *GET*, que o método *busca* será acessado através do método *GET* do protocolo HTTP. O método deverá produzir uma saída utilizando uma das representações já apresentada. Nesse primeiro momento, vamos trabalhar com o XML. Dessa maneira, vamos informar, através da anotação *Produces,* que será retornado um XML. O código é atualizado para:

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

import com.andreymasiero.loja.dao.CarrinhoDAO;
import com.andreymasiero.loja.model.Carrinho;

@Path("/carrinhos")
public class CarrinhoResource {

	@GET
	@Produces(MediaType.APPLICATION_XML)
	public String busca() {
		Carrinho carrinho = new CarrinhoDAO().
busca(1l);
		return carrinho.toXML();
	}
}
```

Nesse momento, devemos subir nossa aplicação em um servidor de containers, como Tomcat, GlassFish ou WildFly para realizarmos nossos primeiros testes. Se tudo der certo, basta digitarmos o path de nosso URI, que deve ficar parecido com http://localhost:8080/loja/carrinhos. Podemos conferir o resultado da requisição ao endereço na figura 1.

**Figura 1 – Resultado da requisição ao recurso de busca de carrinhos**



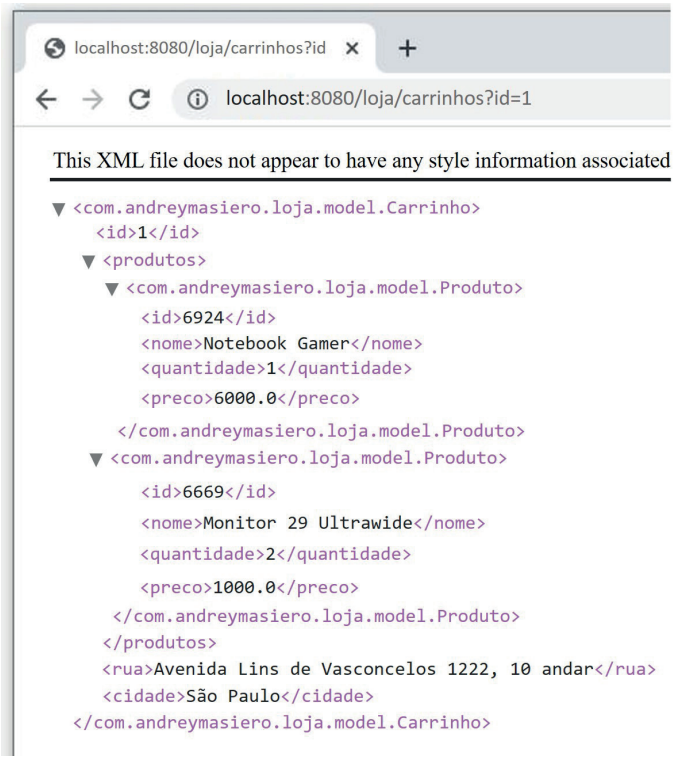

## 1.1.2 Recebendo parâmetros

Como em uma aplicação web qualquer, podemos receber parâmetros em uma requisição do tipo *GET* por meio de uma query string (ou string de consulta), por exemplo: http://localhost:8080/loja/carrinhos?id=1. Contudo, como é possível acessar esse parâmetro em nosso serviço, utilizaremos uma anotação chamada de *QueryParam* e colocaremos ela junto ao parâmetro do método. Confira a refatoração de nosso método *busca* para atender a essa requisição:

```
        @GET
        @Produces(MediaType.APPLICATION_XML)
        public String busca(@QueryParam("id") long id) {
                Carrinho carrinho = new CarrinhoDAO().
busca(id);
                return carrinho.toXML();
        }
```

Assim, ao reiniciar o servidor, podemos invocar o endereço apresentado anteriormente, com o resultado demonstrado na figura 2.

**Figura 2 – Resultado utilizando a passagem de parâmetro por query string**



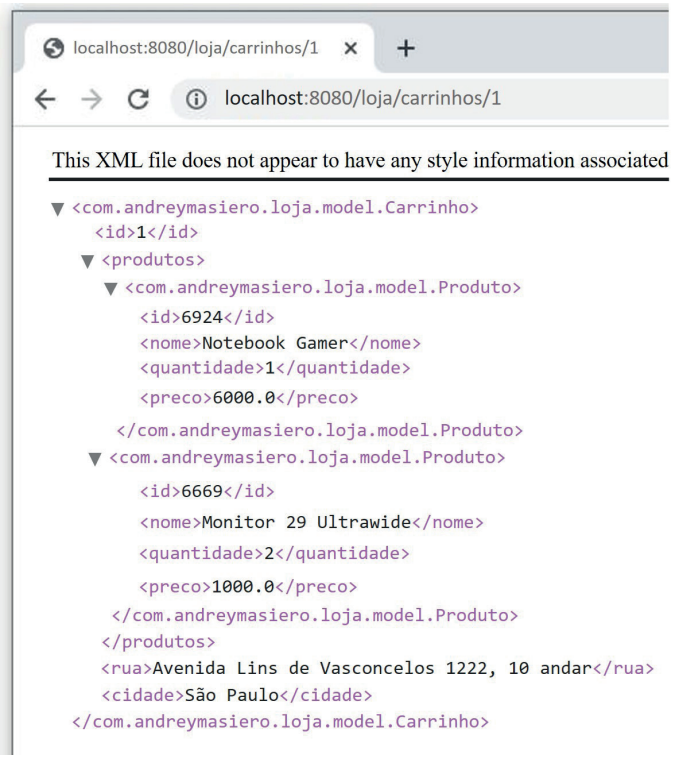
Contudo, apesar do sucesso na passagem de parâmetro, isso não define um recurso. O recurso ainda está em carrinhos, e instanciamos a informação desejada de acordo com o parâmetro passado através de uma query string. O ideal, quando trabalhamos com recursos, é que a URI seja definida da seguinte maneira: http://localhost:8080/loja/carrinhos/1.

Para isso, precisamos mudar o path do nosso método, e os parâmetros serão recebidos via path, não mais via query string. Confira como ficou o código referente ao método *busca* para atender aos princípios do REST.

```
        @Path("/{id}")
        @GET
        @Produces(MediaType.APPLICATION_XML)
        public String busca(@PathParam("id") long id) {
                Carrinho carrinho = new CarrinhoDAO().
busca(id);
                return carrinho.toXML();
        }
```

Pronto, agora podemos acompanhar o teste na figura 3.

**Figura 3 – Resultado da requisição através de um URI**




Nesse momento, aprendemos as maneiras mais comuns de enviar parâmetros em uma requisição através do método *GET* para um serviço do tipo REST.

### 1.1.3 Cadastrando um carrinho

Nesse momento, temos um recurso disponível para consultar as informações cadastradas em nosso servidor. Contudo, precisamos muitas vezes permitir que o nosso cliente cadastre informações em nosso servidor, principalmente se estamos saindo de uma arquitetura monolítica para uma arquitetura orientada a serviços ou arquitetura de microsserviços.

Quando trabalhamos com manipulação de dados em nosso servidor, principalmente com a inclusão de novos dados, é necessário utilizar o método coerente do protocolo HTTP para tal funcionalidade. O método em questão é o *POST*, que permite determinada segurança quanto à sequência de requisições realizadas. Adiante, falaremos mais sobre essas questões.

Vamos então trabalhar no método *adiciona*, que receberá a string do XML que irá representar nosso objeto *carrinho*. Ao mesmo tempo, retornará uma string do XML com o status da operação. Assim, novas duas anotações serão apresentadas, *POST* e *Consumes*. Confira como ficará o nosso método:

```
        @POST
        @Consumes(MediaType.APPLICATION_XML)
        @Produces(MediaType.APPLICATION_XML)
        public String adiciona(String conteudo) {
                Carrinho carrinho = (Carrinho) new
XStream().fromXML(conteudo);
                new CarrinhoDAO().adiciona(carrinho);
                return "<status>ok</status>";
        }
```

O método *POST*, diferentemente do *GET*, necessita de uma chamada via aplicação para ser testado. Apesar de termos nosso método de adição de carrinho funcionando, retornar um XML como uma tag status não é a melhor prática. O protocolo HTTP já possui diversos códigos padronizados para retornarmos após consumir quaisquer recursos disponíveis nos sistemas. São padronizações de códigos:

- 1XX – Informativo.
- 2XX – Sucesso.
- 3XX – Redirecionamento.
- 4XX – Erros do cliente.

- 5XX – Erros do servidor.

Quando implementamos um web service (ou serviço web), os erros que geralmente ocorrem e devem ser tratados pelos sistemas são:

- 200 – Ok.
- 201 – Created (para quando criamos algo com o recurso).
- 204 – No content (sem conteúdo).
- 404 – Recurso não encontrado.
- 405 – Método não permitido.
- 500 – Erro interno do servidor.

Dessa maneira, a melhor forma de retornar um status para o cliente ao processar uma requisição no servidor é por meio de códigos HTTP. Sendo assim, vamos alterar um pouco o código do método de adicionar um carrinho ao sistema. A classe *Response* do JAX-RS fará essa comunicação entre o cliente e o servidor. No caso do *POST*, que irá salvar informações dentro do servidor, o status que iremos trabalhar é o 201. Ele indica que o estado do servidor foi alterado a partir de uma criação de registro novo.

```
	@POST
	@Consumes(MediaType.APPLICATION_XML)
	@Produces(MediaType.APPLICATION_XML)
	public Response adiciona(String conteudo) {
		Carrinho carrinho = (Carrinho) new
XStream().fromXML(conteudo);
		new CarrinhoDAO().adiciona(carrinho);
		URI uri = URI.create("/loja/carrinhos/" +
carrinho.getId());
		return Response.created(uri).build();
	}
```

O método *created* da classe *Response* recebe como parâmetro um URI com a identificação do recurso criado. A identificação é retornada como um parâmetro de location, para que possa ser consumida na sequência, se necessário. Utilizando o programa cURL através da linha de comando, é possível verificar o cabeçalho de requisição e resposta do recurso.

```
*   Trying ::1:8080...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> POST /loja/carrinhos HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.65.2
> Accept: */*
> Content-Length: 351
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 351 out of 351 bytes
* Mark bundle as not supporting multiuse
< HTTP/1.1 201
< Location: http://localhost:8080/loja/carrinhos/3
< Content-Length: 0
< Date: Sun, 22 Mar 2020 17:34:14 GMT
<
* Connection #0 to host localhost left intact
```

Podemos conferir o código do status do HTTP retornado (201) e o parâmetro da localização do recurso criado na requisição.

### 1.1.4 Excluindo recursos

Com um método de consulta e outro de cadastro feitos, podemos trabalhar com a exclusão de informação do nosso servidor. Porém, nesse caso, não iremos excluir o carrinho, pois já aprendemos a trabalhar com o recurso único como o carrinho de compras. A ideia, ao utilizar o método *DELETE* do HTTP, é excluir um produto que se encontra no

carrinho. Para isso, vamos trabalhar com um path definido com mais de um parâmetro, refletindo isso no nosso método *removeProduto*.

Quando um recurso é definido com mais de um parâmetro, dizemos que iremos acessar um subrecurso a partir de um principal, conforme o método:

```
        @Path("/{id}/produto/{produtoId}")
        @DELETE
        public Response removeProduto(@PathParam("id") long
id, @PathParam("produtoId") long produtoId) {
                Carrinho carrinho = new CarrinhoDAO().
busca(id);
                carrinho.remove(produtoId);
                return Response.ok().build();
        }
```

### 1.1.5 Atualizando a quantidade de produtos

O próximo passo de implementação em nosso serviço REST é a possibilidade de alterar a quantidade de um produto em nosso carrinho. Poderíamos simplesmente excluir o objeto desatualizado e incluir o novo objeto, mas isso seria uma adaptação técnica muito grande e não ficaria registrada como uma operação de atualização.

A operação mais adequada para essa situação é a fornecida pelo método *PUT* em serviços REST. O *PUT* significa, para nosso serviço, a troca de um recurso por outro. Se fizermos uma requisição para o recurso "/carrinhos/1/produto/6669", podemos enviar um novo objeto através de uma representação, com uma quantidade menor de monitores para comprar, por exemplo, em nosso e-commerce.

O nosso método de atualização de recurso deve ser implementado pelo código:

```
        @Path("/{id}/produto/{produtoId}")
        @PUT
        @Consumes(MediaType.APPLICATION_XML)
        public Response alteraProduto(@PathParam("id")
long id, @PathParam("produtoId") long produtoId, String
conteudo) {
                Carrinho carrinho = new CarrinhoDAO().
busca(id);
         Produto produto = (Produto) new XStream().
fromXML(conteudo);
         carrinho.troca(produto);
                return Response.ok().build();
        }
```

Entretanto, ao realizar a troca de um produto inteiro, permitimos ao cliente fazer a troca de informações que não devem ser alteradas por ele, como o valor dos produtos dentro do carrinho. Isso gera uma inconsistência de dados que não deve ocorrer em nenhum sistema.

Para corrigir esse problema, devemos deixar a URI mais específica, com a informação da propriedade do recurso que estamos alterando, e deixar o método de atualização também mais específico. Confira o novo código:

```
        @Path("/{id}/produto/{produtoId}/quantidade")
        @PUT
        @Consumes(MediaType.APPLICATION_XML)
        public Response alteraProduto(@PathParam("id")
long id, @PathParam("produtoId") long produtoId, String
conteudo) {
                Carrinho carrinho = new CarrinhoDAO().
busca(id);
         Produto produto = (Produto) new XStream().
fromXML(conteudo);
         carrinho.trocaQuantidade(produto);
                return Response.ok().build();
        }
```

## Considerações finais

Neste capítulo, apresentamos como criar uma API de cadastro básico com o padrão arquitetural REST, entendendo, em linhas gerais, o processo de integração de software mais utilizado nas soluções, o padrão arquitetural para serviços mais popular no mercado de trabalho. Agora que você sabe como trabalhar com os métodos principais do HTTP para serviços, não deixe de construir as novas operações com todos os serviços para compartilhar o seu sistema.

## Referências

GONÇALVES, Edson. **Desenvolvendo aplicações web com JSP, Servlets, JavaServer Faces, Hibernate, EJB 3 Persistence e Ajax**. Rio de Janeiro: Ciência Moderna, 2007.