

UNIVERSIDADE FEDERAL DO RIO GRANDE - FURG
CAMPUS RIO GRANDE
CURSO SUPERIOR EM TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE
SISTEMAS

GABRIEL DA SILVA ALMEIDA

**Arquitetura RESTful: Estudo, Análise e
Implementação**

Trabalho de Conclusão de Curso apresentado
como requisito parcial para a obtenção do
grau de Tecnólogo em Análise e Desenvolvimento
de Sistemas

Prof. Dr. Tiago Lopes Telecken
Orientador

Rio Grande, dezembro de 2015

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Almeida, Gabriel da Silva

Arquitetura RESTful: Estudo, Análise e Implementação / Gabriel da Silva Almeida. – Rio Grande: Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas da FURG, 2015.

111 f.: il.

Trabalho de Conclusão de Curso – Universidade Federal do Rio Grande - FURG. Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas, Rio Grande, BR-RS, 2015. Orientador: Tiago Lopes Telecken .

1. Web Service. 2. Framework. 3. REST. 4. RESTful.
5. HTTP. I. , Tiago Lopes Telecken. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE - FURG

Reitora: Prof^a. Dra. Cleuza Maria Sobral Dias

Vice-Reitor: Prof. Dr. Danilo Giroldo

Pró-Reitora de Graduação: Prof^a. Dra. Denise Maria Varella Martinez

Coordenador do curso: Prof. Rafael Betito

Dedico este trabalho à minha família.

AGRADECIMENTOS

Agradeço primeiramente à Deus por estar vivo, pois estar vivo é a resposta de Deus a todos os nossos problemas; à minha família, por ter sempre me prestado o suporte e apoio para que eu estudasse, ao meu orientador Prof. Dr. Tiago Telecken por ter me orientado neste trabalho e a todos que de alguma forma contribuíram para meu crescimento e aprendizado.

RESUMO

Atualmente a arquitetura REST e seus paradigmas estão sendo adotados nas implementações de sistemas distribuídos. As implementações REST são realizadas através de Web Services, que geralmente são disponibilizadas na forma de frameworks ou Web APIs, cujo principal objetivo é a troca de dados entre aplicações, através da Web. No entanto, devido à ausência de padrões e diretrizes para o desenvolvimento, cada implementação REST segue uma linha de desenvolvimento, sendo que muitas acabam desconsiderando os princípios e paradigmas da arquitetura, o que resulta na dificuldade de construção de aplicações cliente e do desenvolvimento e publicação de novos recursos que seguem estritamente os padrões da arquitetura. Outro desafio para os desenvolvedores nas implementações é a falta de suporte ao uso de controle de hipermídia em representações de recursos, principalmente os que utilizam o formato JSON. Os controles de hipermídia podem assumir a forma de links, que guiam a navegação entre diferentes recursos. Este trabalho propõe uma abordagem de desenvolvimento de framework RESTful JavaEE, o qual a estrutura consiste de um conjunto de classes (concretas, interfaces e abstratas), explicitamente projetado para ser usado em conjunto com o Jersey 2.0, implementação de referência da JAX-RS (JSR 311), especificação do Java para serviços Web REST. Portanto o framework é uma extensão da biblioteca Jersey 2.0, que visa fornecer corretamente todos os serviços da arquitetura REST, através das classes, possibilitando ao desenvolvedor seguir as regras de utilização e construir de maneira rápida um serviço RESTful padronizado, nível três de maturidade, além de facilitar assim a criação de aplicações cliente para consumir os recursos destes serviços. O suporte fornecido pelo framework proposto possibilita que o desenvolvedor concentre esforços no desenvolvimento do domínio do problema, sem perder tempo com infraestrutura. Com a utilização da abordagem proposta, espera-se proporcionar padrões de projeto, maior produtividade e qualidade no desenvolvimento de Web Services RESTful, alinhados com os princípios arquiteturais.

Palavras-chave: Web Service. Framework. REST. RESTful. HTTP.

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
CSV	Campo Separado por Vírgula
CRUD	Create, Read, Update, Delete
DCOM	Distributed Component Object Model
ER	Entidade-Relacionamento
FURG	Universidade Federal do Rio Grande
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IDE	Integred Development Enviroment
IFRS	Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul
JavaEE	Java Platform Enterprise Edition
JavaSE	Java Platform, Standard Edition
JSON	JavaScript Object Notation
POJO	Plain Old Java Objects
REST	Representational State Transfer
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
SGDB	Sistema de Gerenciamento de Banco de Dados
SMTP	Simple Mail Transfer Protocol
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SPI	Serial Peripheral Interface
SQL	Structured Query Language

TADS Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas
UDDI Universal Description, Discovery and Integration
UML Unified Modeling Language
URI Unified Resource Identification
URL Unified Resource Locator
W3C World Wide Web Consortium
WADL Web Application Description Language
WSDL Web Services Description Language
WWW World Wide Web
XML Extensible Markup Language

LISTA DE FIGURAS

Figura 2.1:	Arquitetura Orientada a Serviços (BIH, 2006).	22
Figura 2.2:	Acesso a regras de negócio de uma aplicação com uso de serviços web. Fonte: Adaptada de Tidwell, Snell e Kulchenko (TIDWELL; SNELL; KULCHENKO, 2001).	23
Figura 2.3:	Estrutura de um envelope SOAP. Fonte: Adaptada de Tidwell, Snell e Kulchenko (TIDWELL; SNELL; KULCHENKO, 2001).	25
Figura 2.4:	A descoberta de um serviço mediante consulta a um registro UDDI. Fonte: Adaptada de Shomoyita e Ralph (SHOMOYITA; RALPH, 2011)	25
Figura 2.5:	Interface WSDL entre o Cliente e o Serviço Web. Fonte: adaptada de Gonsalves (GONSALVES, 2009)	26
Figura 2.6:	Documentos WSDL representando aplicações Web Services. Fonte: http://www.devmedia.com.br/introducao-as-tecnologias-web-services-soa-soap-wsdl-e-uddi-parte1/2873	27
Figura 2.7:	Estrutura de um Envelope SOAP representado em XML.	28
Figura 2.8:	Exemplo de requisição e resposta HTTP. Fonte: Adaptada de Gourley e Totty (GOURLEY; TOTTY, 2002).	30
Figura 2.9:	Exemplo de URI.	31
Figura 2.10:	Exemplo de URI de acesso para o recurso produto.	32
Figura 2.11:	Exemplo de retorno de dados no formato JSON.	33
Figura 2.12:	Exemplo de retorno de resposta autoexplicativa no formato de dados JSON para o método POST.	33
Figura 2.13:	Exemplo de recurso no formato JSON com links de hipermídia.	34
Figura 2.14:	Requisição RESTful utilizando o método GET do protocolo HTTP. .	35
Figura 2.15:	Requisição RESTful utilizando o método DELETE do protocolo HTTP.	35
Figura 2.16:	Endereçamento no padrão SOAP (XML-RPC).	36
Figura 2.17:	Endereçamento de recursos segundo o paradigma RESTful.	36
Figura 2.18:	Serviço Web através de Layered System. Fonte: Gabriel Almeida .	38
Figura 2.19:	Níveis de maturidade RMM. Fonte adaptada de Fowler (FOWLER, 2010)	38
Figura 2.20:	Exemplo de sintaxe básica do JSON. Fonte adaptada de Algermissen (ALGERMISSSEN, 2010).	40
Figura 2.21:	Exemplo de sintaxe básica do JSON.	41
Figura 2.22:	Exemplo de retorno JSON a partir de uma requisição GET e diagrama de classes ilustrando o modelo de entidade do recurso usando o Spring-HATEOAS.	43
Figura 2.23:	Arquitetura Apache Isis. Adaptado de BIENVENIDO (BIENVENIDO, 2013).	44

Figura 2.24: Visão geral do framework. Fonte: (JOHN; RAJASREE, 2012)	45
Figura 2.25: Exemplo de utilização de JAX-RS.	46
Figura 3.1: Diagrama de classes do framework.	51
Figura 3.2: Classe BaseDeServicoInterface parte 1.	53
Figura 3.3: Classe HATEOASEntidade.	54
Figura 3.4: Classe Link parte 1.	55
Figura 3.5: Classe InterceptadorClienteRESTful - Declaração e importação de objetos.	56
Figura 3.6: Diagrama da arquitetura e fluxo de dados do serviço web.	57
Figura 3.7: Operações para clientes no serviço web.	58
Figura 3.8: Diagrama de caso de uso referente ao estudo de caso.	58
Figura 3.9: Estrutura dos dados da tabela "Produto"para o estudo de caso proposto neste trabalho.	59
Figura 3.10: Diagrama de classes para o estudo de caso proposto.	60
Figura 3.11: Estrutura do projeto Web Service Intermediador	62
Figura 3.12: Bibliotecas utilizadas no projeto Web Service Intermediador.	63
Figura 3.13: Classe Produto Recurso parte 01.	64
Figura 3.14: Classe Produto Recurso parte 02.	65
Figura 3.15: Estrutura do projeto Web Service Final.	67
Figura 3.16: Bibliotecas utilizadas no projeto Web Service Final.	68
Figura 3.17: Classe GenericDAO parte 01.	69
Figura 3.18: Classe GenericDAO parte 02.	70
Figura 3.19: Classe ProdutoDAO parte 01.	71
Figura 3.20: Classe ProdutoDAO parte 02.	72
Figura 3.21: Classe ProdutoControlador.	73
Figura 3.22: Classe BaeDeEntidade.	74
Figura 3.23: Classe Produto (serviço web intermediador).	75
Figura 3.24: Respostas de requisições POST, PUT e DELETE, informando JSON para cadastro de produto(s) no método POST.	76
Figura 3.25: Cliente PHP - página inicial.	77
Figura 3.26: Lista de produtos JSON, XML e HTML.	78
Figura 3.27: Formulário Produto.	79
Figura 3.28: Requisição GET com PHP para resposta em formato JSON.	80
Figura 3.29: Requisição GET com PHP para resposta em formato XML.	81
Figura 3.30: Requisições GET para resposta em formato HTML e DELETE para exclusão de registro. Ambos têm respostas no formato JSON.	82
Figura 3.31: Requisição POST com PHP. Resposta autoexplicativa em formato JSON.	83
Figura 3.32: Requisição PUT para alteração de dados de registro. Resposta auto-explicativa em formato JSON.	84

LISTA DE TABELAS

Tabela 2.1:	Formatos para representação de valores e respectivos cabeçalhos HTTP	30
Tabela 2.2:	Utilização dos métodos HTTP e sua equivalência com SQL	32
Tabela 2.3:	Tabela de comparação entre as ferramentas para desenvolvimento de serviços RESTful	48

SUMÁRIO

1 INTRODUÇÃO	17
1.1 Motivação e justificativa	18
1.2 Objetivos	18
1.2.1 Objetivo geral	18
1.2.2 Objetivos específicos	18
1.3 Organização	19
2 REVISÃO BIBLIOGRÁFICA	21
2.1 SOA	21
2.2 Web Services	22
2.2.1 Ciclo de vida de um Web Service	24
2.3 O Protocolo SOAP	24
2.3.1 SOAP	24
2.3.2 Conceitos de Serviços Segundo SOAP	24
2.3.3 Protocolo para Transporte de Dados	26
2.3.4 UDDI	26
2.3.5 WSDL	26
2.3.6 XML	27
2.4 Fundamentos Sobre Web Service REST	28
2.4.1 REST	28
2.4.2 Histórico	28
2.4.3 Arquitetura ROA	29
2.4.4 Princípios adicionais ao Paradigma RESTful	35
2.4.5 Modelos de Maturidade	38
2.4.6 Descrição dos Serviços	40
2.4.7 JSON	41
2.5 Arquitetura REST em Java	41
2.5.1 JAX-RS	41
2.5.2 Arquitetura RESTful em JAX-RS (JSR 311)	42
2.6 Trabalhos Relacionados	42
2.6.1 Spring-HATEOAS	42
2.6.2 Apache Isis	43
2.6.3 Framework de John e Rajasree	44
2.6.4 Especificação JAX-RS	44
2.6.5 API Jersey	47
2.6.6 Resumo Comparativo dos Trabalhos Relacionados	47

3 PROJETO	51
3.1 Framework	51
3.1.1 Diagrama de Classes	51
3.1.2 Implementação	52
3.2 Estudo de Caso	57
3.2.1 Diagrama de Arquitetura	57
3.2.2 Diagrama de Caso de Uso	58
3.2.3 Estrutura dos dados	59
3.2.4 Diagrama de Classes	59
3.2.5 Implementação	61
4 ANÁLISE	85
5 CONCLUSÕES	87
5.1 Trabalhos Futuros	88
REFERÊNCIAS	89
APÊNDICE A ANEXOS E APÊNDICES	93

1 INTRODUÇÃO

Os Web Services surgiram como uma forma de comunicação entre aplicações distribuídas e heterogêneas. Entretanto ao desenvolver uma aplicação que necessite disponibilizar Serviços Web, tem-se duas opções de implementação para tal tarefa. Os Web Services SOAP, que são uma evolução das RPC's (Remote Procedure Calls) e os Web Services REST, que trabalham de forma semelhante a qual utilizamos a World Wide Web.

Roy Fielding (FIELDING, 2000), o criador do REST, apresenta a arquitetura como uma alternativa ao estilo RPC para a implementação de Serviços Web. Ela reúne uma série de princípios e restrições arquiteturais para o desenvolvimento de aplicações distribuídas. As informações manipuladas pela abordagem REST são organizadas em recursos Web, que constituem um conjunto coeso e mínimo de dados.

Dentre as restrições destacam-se o comportamento stateless do servidor e a manipulação de dados através de interface uniforme, com destaque especial para a utilização de controles hipermídia. Um REST framework ou Web API, são as opções mais comuns para implementar sistemas distribuídos que seguem os princípios arquiteturais REST. Quando uma implementação segue todos os princípios e restrições, ela é considerada RESTful.

Os frameworks e Web APIs utilizam o HTTP como protocolo de comunicação, o que implica em obedecer a semântica deste protocolo, ou seja, utilizar corretamente os verbos (GET, POST, PUT e DELETE) e os códigos de status. Além de utilizar o protocolo HTTP como base, as implementações de framework REST e Web APIs utilizam a Web como infraestrutura de desenvolvimento e isso implica na utilização de tecnologias padronizadas e o correto uso dos protocolos por parte das aplicações.

Os princípios da arquitetura REST diferem do conceito tradicional de serviço ao manipular as informações no formato de recursos. Recursos encapsulam as informações manipuladas pelos frameworks em um conjunto coeso e significativo de informações. Todavia, mesmo adotando o design de recursos, as implementações REST continuam sendo uma abordagem baseada em serviços.

Uma das razões de utilizar um serviço REST é a facilidade na integração de dados. Compartilhamento de bases de dados tem sua aplicação restrita a uma determinada organização. Por outro lado, com a arquitetura REST, é possível compartilhar informações em escala global através da Web. Ao disponibilizar dados na Web, perde-se o controle sobre as aplicações clientes, tornando heterogêneo o consumo dos dados e a forma de interação entre cliente e servidor. Os formatos mais utilizados são XML e JSON.

No entanto, muitas vezes apenas o suporte de tecnologias e frameworks não são suficientes para a construção rápida e padronizada de um serviço RESTful, principalmente com suporte a controles de hipermídia e utilização correta do protocolo HTTP e seus verbos.

Neste contexto, o presente trabalho propõe uma abordagem de desenvolvimento de framework RESTful JavaEE, o qual a estrutura consiste de um conjunto de classes (con-

cretas, interfaces e abstratas), explicitamente projetado para ser usado em conjunto com o Jersey 2.0, implementação de referência da JAX-RS (JSR 311), especificação do Java para serviços Web REST. Portanto o framework é uma extensão da biblioteca Jersey 2.0, que visa fornecer corretamente todos os serviços da arquitetura REST, através das classes, possibilitando ao desenvolvedor seguir as regras de utilização e construir de maneira rápida um serviço RESTful padronizado, nível três de maturidade, além de facilitar assim a criação de aplicações cliente para consumir os recursos destes serviços.

1.1 Motivação e justificativa

A motivação pelo desenvolvimento deste trabalho é apresentar os conceitos e paradigmas da arquitetura REST, bem como a oportunidade de estudo sobre o assunto e de desenvolvimento do framework citado, visando ser um componente para a construção de serviços RESTful utilizando a plataforma de desenvolvimento JavaEE, junto ao Jersey 2.0.

A justificativa deste trabalho é pelo fato da ferramenta Jersey 2.0 não oferecer implementação para uso de hipermídia nos recursos, além da interface unificada com resposta auto-explicativa e um serviço de interceptação entre o cliente e servidor. Serviços que são oferecidos pelo framework proposto.

1.2 Objetivos

A seguir serão apresentados os objetivos deste trabalho.

1.2.1 Objetivo geral

Este trabalho tem o objetivo geral de proporcionar sólidos conhecimentos sobre a arquitetura REST, bem como subsídio metodológico e ferramental para o desenvolvimento frameworks RESTful que manipulam representações de recursos. O subsídio ferramental é constituído por um framework com suporte a controles de hipermídia, para representações do formato JSON, XML, Text e HTML; interface unificada, com respostas auto-explicativas, visando interação com cliente nas respostas a requisições; utilização correta do protocolo HTTP e seus verbos e por fim, um serviço de interceptação entre cliente e servidor.

1.2.2 Objetivos específicos

Os objetivos gerais do trabalho podem ser alcançados através dos seguintes objetivos específicos:

- Realizar um estudo teórico sobre a arquitetura RESTful. Espera-se que tal estudo sirva de referência bibliográfica para interessados em compreender ou implementar a arquitetura RESTful;
- Desenvolver um framework REST seguindo estritamente os padrões e a utilização correta do protocolo HTTP e seus verbos. O framework proposto deve ter as seguintes características: proporcionar suporte ferramental baseado no padrão JSON e XML, reduzir o acoplamento entre a implementação do serviço web e aplicações cliente, diminuindo o impacto gerado pela evolução dos sistemas e auxiliar no desenvolvimento de sistemas RESTful;

- Implementar um estudo de caso. O estudo de caso será um web service com seus respectivos clientes. Para implementação deste servidor serão utilizadas as classes do framework proposto;
- Analisar as características do framework proposto com base no estudo teórico e no estudo de caso implementado.

1.3 Organização

Este documento está distribuído em cinco capítulos: Introdução (capítulo 1), Revisão Bibliográfica (capítulo 2), Projeto (capítulo 3), Análise (capítulo 4) e Conclusões (capítulo 5). A revisão bibliográfica está no capítulo 2 e apresenta inicialmente a arquitetura orientada a serviços, logo após os conceitos e características de Web Services e suas implementações através do SOAP e REST. Neste capítulo também são apresentados alguns trabalhos relacionados a frameworks e Web APIs RESTful. Em projeto, localizado no capítulo 3, é ilustrada e detalhada toda a implementação referente ao framework proposto a este trabalho junto ao estudo de caso, que trata-se da implementação dos serviços web e das aplicações cliente que irão consumir os recursos do Web Service. Em análise, apresentado no capítulo 4, é realizado uma abordagem geral sobre o trabalho. Por fim, em Conclusão, no capítulo 5, são apresentadas as conclusões sobre o trabalho e a proposta para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo será apresentado uma revisão geral dos protocolos da arquitetura SOA e de Web Services, com foco principal na arquitetura REST e desenvolvimento RESTful em Java.

2.1 SOA

O desenvolvimento de software tem contemplado vários paradigmas, tendo destaque o da orientação a objeto. Várias linguagens de programação associadas a esse paradigma foram bem sucedidas, tais como C++, Java e Visual Basic. Porém, com o começo da Internet e o desenvolvimento de aplicações Web, as necessidades empresariais e as requisições de serviços trouxeram a demanda de uma nova arquitetura de software (BIH, 2006). Nesse cenário, surge então, o conceito de SOA (Service-oriented Architecture), como uma solução para esse problema, por meio da utilização dos Web Services.

Segundo (BIH, 2006) SOA são serviços que podem se comunicar por meio de passagem de mensagens simples, coordenando alguma atividade que precise de meios de conexão de serviços entre si.

De acordo com SAMPAIO (SAMPAIO, 2006), a comunicação e a interoperabilidade entre os sistemas é o ponto chave para estabelecer a relação entre diferentes fornecedores e consumidores de serviço, embasando assim o conceito e a motivação de implementar um sistema baseado em SOA.

SOA em português significa “Arquitetura Orientada a Serviços”, sendo sua composição totalmente voltada à questão da disponibilização e implantação do conceito de serviços entre aplicações (TOMCAT, 2005).

Para um bom entendimento de SOA é preciso ter um bom conhecimento do termo serviço. Serviço é uma função bem definida e independente do contexto ou estado de outros serviços. A arquitetura SOA básica consiste de um serviço consumidor e um provedor de serviços, onde um serviço consumidor faz uma requisição para um provedor de serviços, que responde com o resultado para o serviço consumidor; o provedor de serviços pode também ser um consumidor de serviços. As conexões de requisição e resposta são definidas de maneira que seja possível que ambos os serviços possam entendê-la.

O provedor de serviços fornece as interfaces de serviços para um software independente, o qual gerencia um conjunto específico de tarefas. Ele pode representar os serviços de uma entidade de negócios ou simplesmente uma interface de serviço reutilizável de um subsistema.

O serviço consumidor é um nó na rede que descobre e invoca outros serviços para obter uma solução por meio de chamadas à operações dos serviços. Em alguns casos, o provedor pode estar no mesmo local que um consumidor de serviços e ser acessado por

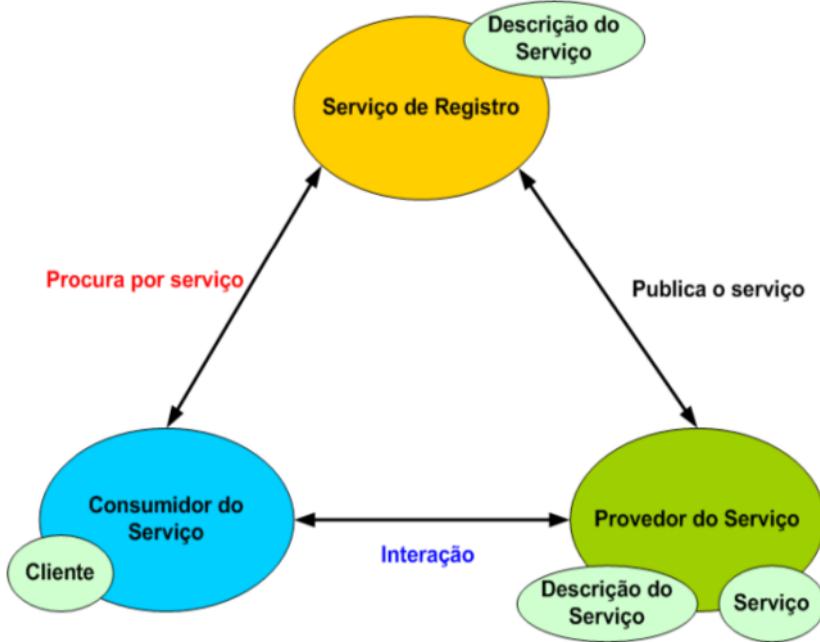


Figura 2.1: Arquitetura Orientada a Serviços (BIH, 2006).

meio de uma rede local. Em caso diferente, ele pode estar muito distante e ser acessado pela Internet. A natureza conceitual de SOA deixa a rede, os protocolos de transporte e os detalhes de segurança, para implementações específicas de SOA.

O serviço de registro atua como um repositório, onde os provedores de serviços publicam seus serviços.

Essas três entidades de SOA interagem entre si por meio de três operações básicas: publicar, buscar e utilizar. O provedor de serviços publica serviços no serviço de registro. O consumidor de serviços busca o serviço que deseja, utilizando o serviço de registro, e então faz a utilização desse serviço. Estas interações são ilustradas na figura 2.1

De maneira resumida, SOA é uma arquitetura orientada a serviços, e a implementação mais abordada para este conceito de arquitetura são os chamados Web Services, ou seja, os Web Services são implementações de SOA. No próximo capítulo será apresentado uma abordagem sobre Web Services.

2.2 Web Services

Com a evolução das redes de computadores surgiram as aplicações distribuídas. Inicialmente todo o processamento era centralizado em apenas um servidor. Com o surgimento dos middlewares, programa de computador que faz a mediação entre software e demais aplicações, o processamento começou a ser distribuído entre vários servidores. Com o avanço da internet e dos protocolos de comunicação baseados em XML, surgiram os Web Services com a missão de integrar sistemas heterogêneos (GOMES, 2010).

O termo Web Service tem sido muito abordado nos últimos anos, com várias definições apresentadas pela literatura. Uma definição aceita apresenta Web Services como um componente, ou unidade lógica de aplicação, acessível por meio de protocolos padrões da Internet, possuindo uma funcionalidade que pode ser reutilizada sem a preocupação de como é implementada.

A W3C define um Web Service como uma aplicação identificada por uma URI (Uniform Resource Identifier), cujas interfaces e ligações são definidas, descritas e descobertas utilizando-se uma linguagem padrão, XML (FERRIS; FARREL, 2003).

Os Web Services são referenciados como uma implementação de SOA (ERRADI; MAHESHWARI, 2005). As interações entre Web Services ocorrem tipicamente com chamadas SOAP, um protocolo de comunicação baseado em XML para a interação de aplicações (PAPAZOGLOU; GEORGAKOPOULOS, 2001). SOAP é apresentado como uma estrutura de sustentação para uma nova geração de aplicações de computação distribuída, independente de plataforma e de linguagens de programação. A principal função desse protocolo é encapsular as chamadas a métodos remotamente distribuídos, que serão transportados, por sua vez, por algum protocolo, tal como o HTTP, utilizado para comunicação entre um browser (navegador) e um servidor Web (COMER, 2000). Além disso, as descrições de interfaces dos Web Services são expressas usando a linguagem denominada WDSL (THOMAS; THOMAS; GHINEA, 2003). Na literatura sobre Web Services é apresentado também um protocolo para serviços de diretório, que contém as descrições dos Web Services, denominado UDDI (Universal Description Discovery and Integration). Esse protocolo funciona como um registro de serviços sendo um importante componente da arquitetura orientada a serviços (FARKAS; CHARAF, 2003). Uma arquitetura orientada a serviço é uma maneira lógica de construção de um sistema de software para prover serviços, ou para aplicações de usuários finais, ou para outros serviços distribuídos em uma rede, por meio de interfaces públicas disponíveis (nesse contexto destaca-se o WSDL) (PAPAZOGLOU, 2003). O UDDI habilita, ainda, os clientes de Web Services a localizarem serviços ou descobrirem seus detalhes, permitindo que registros operacionais sejam mantidos para diferentes propósitos em diferentes contextos (LEAVITT, 2004).

Segundo Gonsalves (GONSALVES, 2009), serviços web são definidos como “um tipo de lógica de negócio exposta por meio de uma interface de serviço para uma aplicação cliente”. Na figura 2.2 é ilustrado a definição do autor, onde apresenta o acesso a regras de negócio de uma aplicação com uso de serviços web.

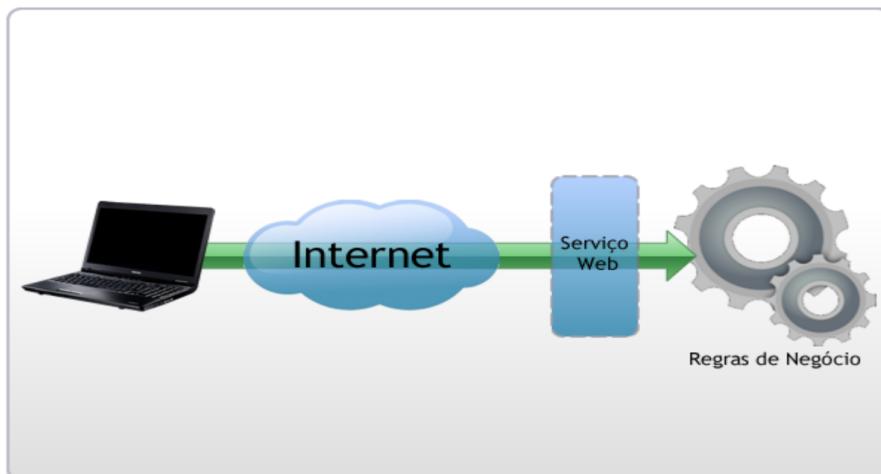


Figura 2.2: Acesso a regras de negócio de uma aplicação com uso de serviços web. Fonte: Adaptada de Tidwell, Snell e Kulchenko (TIDWELL; SNELL; KULCHENKO, 2001).

2.2.1 Ciclo de vida de um Web Service

De acordo com (KREGER et al., 2001) o desenvolvimento do ciclo de vida tem várias fases, as quais podem ser:

- **Construção:** Esta fase é composta pelo desenvolvimento e testes de um Web Service, definição da descrição da interface do serviço e a definição da descrição da implementação do serviço. Os Web Services podem surgir a partir de novas criações, transformações de aplicações existentes e composição de Web Services existentes.
- **Publicação:** Fase em que ocorre a publicação da interface do serviço e definições de implementação para o consumidor ou registro de serviços e também a publicação da parte funcional do Web service, geralmente em um provedor de serviços.
- **Execução:** Essa é a fase onde o Web service fica disponível para ser acessado. Neste estágio o Web service já está publicado, operando e acessível pela rede por meio de um provedor de serviços. Desta forma o consumidor de serviço pode finalmente encontrá-lo e utilizá-lo.
- **Gerenciamento:** Essa fase de gerenciamento cobre o gerenciamento e a administração do Web Service. É importante considerar requisitos tais como segurança, disponibilidade, desempenho, qualidade do serviço e processos de negócio.

Nos próximos capítulos serão apresentadas duas diferentes abordagens para implementações de Web Service. São elas: SOAP e REST.

2.3 O Protocolo SOAP

Nesta seção será apresentado os conceitos e serviços do protocolo SOAP.

2.3.1 SOAP

SOAP é um protocolo de aplicação padrão para serviços web e está relacionado diretamente à tecnologia XML (GONSALVES, 2009), ou seja, é uma aplicação da especificação XML (TIDWELL; SNELL; KULCHENKO, 2001). Isto significa que SOAP é basicamente uma implementação de envelopes baseado em XML para transporte de informações, bem como um conjunto de regras para traduzir tipos de aplicações e plataformas específicas para representação XML.

Conforme pode ser visto na figura 2.3, um envelope SOAP lembra muito a marcação da estrutura básica de uma página HTML, contendo um cabeçalho, sendo este opcional, e um corpo. No cabeçalho do envelope, são informados dados referentes a configurações de entrega da mensagem, autenticação ou regras de autorização ou contexto de transações. No corpo por sua vez, o conteúdo da mensagem a ser transmitida é informado.

2.3.2 Conceitos de Serviços Segundo SOAP

O princípio básico de funcionamento do SOAP é simples, consiste na descoberta do serviço registrado em um registro (UDDI) e o acesso ao mesmo. Na figura 2.4 é apresentado o fluxograma referente à descoberta de um serviço mediante consulta a um registro UDDI.

No processo geral de publicação, descoberta e consumo de um Serviço Web padrão SOAP, algumas tecnologias estão envolvidas (GONSALVES, 2009). São elas: UDDI, WSDL, SOAP, HTTP e XML.

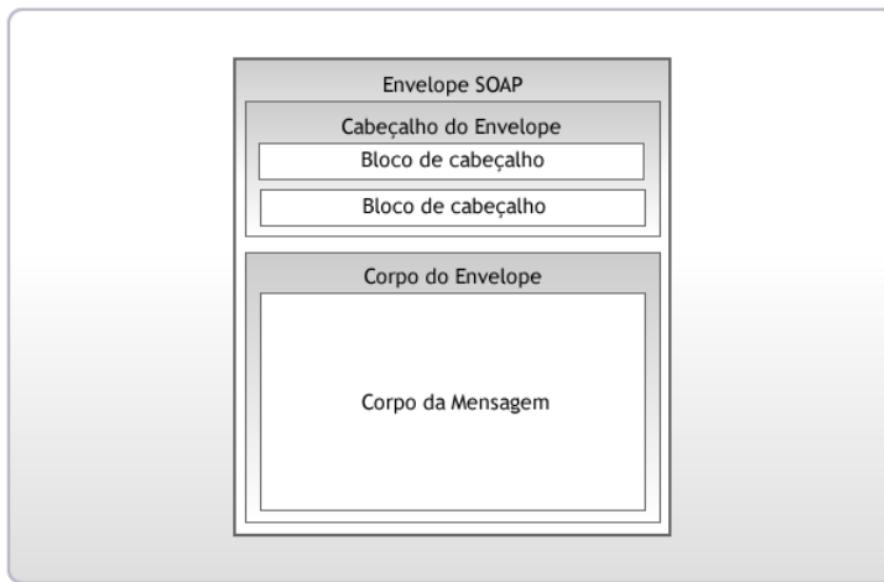


Figura 2.3: Estrutura de um envelope SOAP. Fonte: Adaptada de Tidwell, Snell e Kulchenko (TIDWELL; SNELL; KULCHENKO, 2001).

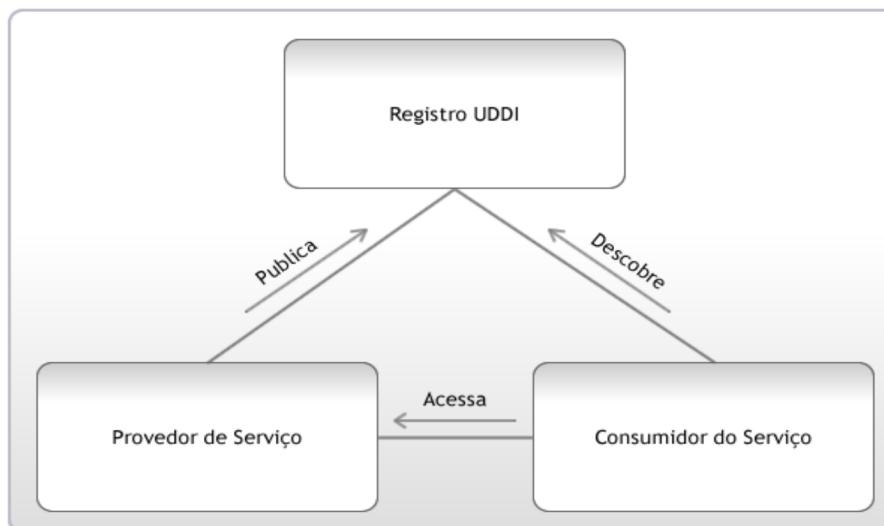


Figura 2.4: A descoberta de um serviço mediante consulta a um registro UDDI. Fonte: Adaptada de Shomoyita e Ralph (SHOMOYITA; RALPH, 2011)

2.3.3 Protocolo para Transporte de Dados

O HTTP é o protocolo padrão para o transporte de dados em Web Services, embora para configurações que exijam maior segurança no tráfego das informações o uso correto e adequado seria o protocolo HTTPS. Não tanto comum quanto o HTTP, qualquer outro protocolo para transporte como o SMTP (Simple Mail Transfer Protocol) ou FTP (File Transfer Protocol), podem ser empregados para transferência de mensagens (GONSALVES, 2009).

2.3.4 UDDI

Desenvolvido pela OASIS (Organization for the Advancement of Structured Information Standards), UDDI é um serviço de diretório que permite a publicação e descoberta de serviços web. A comunicação é realizada através do SOAP e as interfaces web service são descritas por WSDL, como pode ser visto na figura 2.5, que ilustra uma interface entre o cliente e o serviço web.

A intenção de utilização do UDDI é permitir que aplicações que precisam comunicar-seumas com as outras por meio da Web, possam encontrar informações que viabilizem esta comunicação. Mais especificamente os registros UDDI suportam o gerenciamento de meta-informação, as quais descrevem serviços em particular. Normalmente estas meta-informações são representadas em linguagem WSDL. Estes registros possibilitam ainda a descoberta de forma automática ou semiautomática da composição de serviços, podendo desta forma ser visto como um diretório para serviços web (BLAKE et al., 2007).

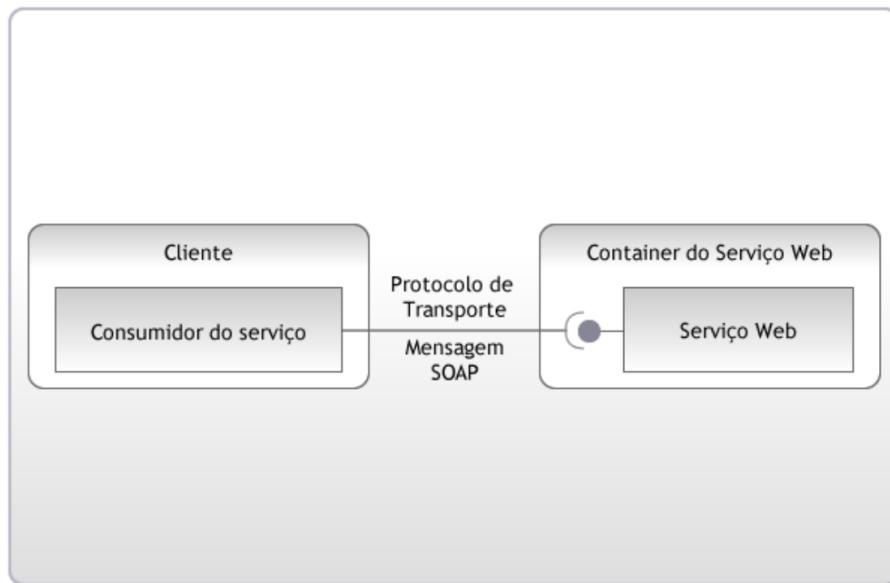


Figura 2.5: Interface WSDL entre o Cliente e o Serviço Web. Fonte: adaptada de Gonsalves (GONSALVES, 2009)

2.3.5 WSDL

O WSDL é um arquivo escrito em linguagem de definição de interface IDL (Interface Definitivo Language), e define a interface do serviço web, informando os tipos de mensagens suportadas, porta, protocolo de comunicação, operações suportadas, localização, e demais informações que o desenvolvedor do serviço web julgar necessária informar para

quem deseja consumir o serviço. Para assegurar a interoperabilidade, um serviço web padrão é preciso que o consumidor e o produtor compartilhem e entendam as mensagens. Sendo assim, este é o foco do WSDL (ORT, 2005). Na arquitetura do protocolo SOAP o registro UDDI deve apontar para um arquivo WSDL público, disponível na internet para potenciais consumidores do serviço web.

Os Web Services devem ser definidos de forma consistente para que possam ser descobertos e interfaceados com outros serviços e aplicações. A WSDL é uma especificação W3C que fornece a linguagem mais avançada para a descrição de definições de Web Services. A figura 2.6 apresenta a definição de um documentos WSDL representando aplicações Web Services.

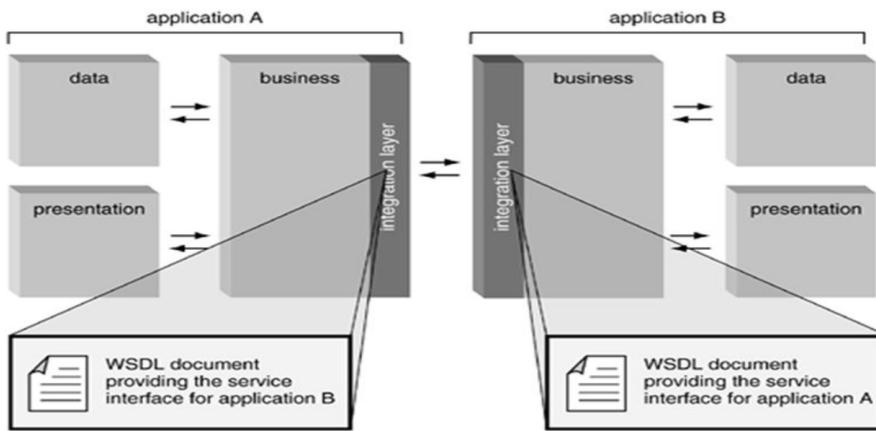


Figura 2.6: Documentos WSDL representando aplicações Web Services. Fonte: <http://www.devmedia.com.br/introducao-as-tecnologias-web-services-soa-soap-wsdl-e-uddi-parte1/2873>

2.3.6 XML

O padrão XML hoje é largamente utilizado, por facilitar a compreensão por humanos e até certo ponto por máquinas, do conteúdo que descrevem. Este padrão de marcação teve seu início em 1998 e evoluiu juntamente com o SOAP. Porém, o mesmo não está tão fortemente atrelado ao SOAP, quanto o SOAP a ele, visto que a base de funcionamento do protocolo SOAP é fundamentada na troca de mensagens serializadas e envelopadas em documentos XML. Outros padrões como o REST, que será abordado em seguida, também podem empregar o XML como meio de serialização de objetos.

Em conjunto com os esquemas permite a definição de tipos e validação de dados, uma vez que o mesmo não é somente utilizado no envio de mensagens, como também no arquivo WSDL. Devido ao fato do XML não estar vinculado a nenhuma aplicação, plataforma, Sistema Operacional, ou linguagem de programação específica a interoperabilidade do serviço web pode ser garantida. Por exemplo, uma aplicação escrita em PHP executando em Windows, pode enviar uma mensagem a uma aplicação Java executando em um Sistema Operacional Unix (TIDWELL; SNELL; KULCHENKO, 2001). A figura 2.7 apresenta um envelope SOAP em estrutura de XML.

```

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP:Header>
        <!-- conteúdo do cabeçalho do Envelope SOAP -->
    </SOAP:Header>
    <SOAP:Body>
        <!-- conteúdo do Corpo do Envelope SOAP -->
    </SOAP:Body>
</SOAP:Envelope> |

```

Figura 2.7: Estrutura de um Envelope SOAP representado em XML.

2.4 Fundamentos Sobre Web Service REST

Nesta seção será apresentado os conceitos e características sobre a arquitetura REST para desenvolvimento de Web Services. Também será abordado o padrão RESTful, o qual é o resultado de uma implementação de REST que segue e respeita rigidamente todos as regras estabelecidas pela arquitetura.

2.4.1 REST

REST (Representational State Transfer) é um estilo arquitetural, com foco em recursos, para sistemas multimídia distribuídos que enfatiza a generalização das interfaces, a escalabilidade da integração entre os componentes e a instalação independente dos mesmos (FIELDING, 2000).

Para o autor, recursos são representações de dados importantes para a aplicação. Um exemplo seria a representação de um produto ou empresa.

De forma direta REST é um paradigma, uma arquitetura que reúne um grupo de critérios a serem incorporados ao projeto de aplicações distribuídas, apesar de não existir qualquer conexão direta com algum protocolo especificamente. A definição de RESTful ainda é citada por outros autores como, Kamaleldin e Duminda (KAMALELDIN; DUMINDA, 2012) como sendo baseada em recursos, os quais são identificados por URIs únicas. Estes recursos são acessados e manipulados usando um conjunto de métodos uniformes (GET, POST, PUT e DELETE), onde cada recurso pode ter uma ou mais representação (XML, JSON, Text, etc), as quais são transferidas entre o cliente e o serviço, durante a invocação ao mesmo.

2.4.2 Histórico

A arquitetura REST foi idealizada e definida por Roy Thomas Fielding em sua tese de doutorado na Califórnia (USA), muito embora não tenha sido inicialmente definido com o propósito para o qual é utilizado hoje. De fato o propósito inicial foi “designar uma arquitetura de design e desenvolvimento para a Web moderna...” (FIELDING, 2000). Porém, partindo desta apresentação, a arquitetura REST difundiu-se como uma forma simplificada para implementação de serviços web ou "RESTful Web Services"(RICHARDSON; RUBY, 2007).

O fato de interligar REST e HTTP beneficiou a arquitetura REST de duas formas, sendo: (a) pelo fato do HTTP ser o protocolo normalizado pelo W3C como padrão para transmissão de mensagens na Web; e (b) todos os sistemas interligados na Web o implementam, ou seja, o grupo de sistemas compatíveis com REST é abrangente. Além disso, por utilizar a própria Web como infraestrutura de distribuição e acesso, torna-se inteiramente portável, não restando qualquer dependência adicional de hardware ou software

(FILHO, 2009) e (KAMALELDIN; DUMINDA, 2012).

Como resultado da interligação entre REST e HTTP, (RICHARDSON; RUBY, 2007), definiram uma arquitetura orientada a recursos chamada ROA (Resource Oriented Architecture), em que esta seguia fielmente a estrutura REST ao mesmo tempo em que empregava o HTTP.

Algumas empresas emergentes da Web 2.0 começaram então a migrar suas API's do protocolo SOAP para o emergente paradigma RESTful, algumas delas citadas a seguir:

- Google - Tendo marcado sua API SOAP (SOAP Search API) como obsoleta desde o ano de 2006, em 31 de agosto de 2009 finalmente declarou a descontinuidade definitiva (INC., 2009). A partir desta data oferece exclusivamente uma API inteiramente baseada em classes JavaScript em que os serviços web são disponibilizados por meio de RESTful. Muito embora esta API tenha representado um avanço, a mesma está marcada como obsoleta desde novembro de 2010 e pode sair do ar brevemente para dar lugar a atual “Custom Search API”, a qual elimina as classes JavaScript e disponibiliza toda a API somente por meio de serviços web RESTful;
- Twitter - Disponibiliza uma API REST para desenvolvimento de aplicações integradas fazendo uso do protocolo de autenticação. O Auth 9, o qual permite acesso aos dados do usuário sem a necessidade de informar as credenciais do mesmo (tipicamente um par contendo usuário e senha), por meio do redirecionamento de agentes de usuário (HAMMER-LAHAV, 2010).

2.4.3 Arquitetura ROA

Em engenharia de software, uma arquitetura orientada a recursos (ROA) é um estilo de arquitetura de software e paradigma de programação para a concepção e desenvolvimento de software sob a forma de recursos com interfaces "RESTful". ROA é composta por alguns conceitos (Endereçabilidade, Estado Não-Persistente, Conectividade), como não possuir um estado persistente e considerar importante a descrição dos serviços, utilizando como pontes para comunicação (HONG, 2012). Funciona basicamente como qualquer outro serviço web, que recebe uma requisição detalhando uma ação a ser executada e retorna uma resposta detalhando o resultado obtido. Observa-se que para ROA ou RESTful, tanto a forma como a ação será executada como seu escopo de execução é discriminado na requisição (FILHO, 2009). Isso se deve ao fato de que em RESTful o estado das requisições não é armazenado, e cada requisição é única.

A seguir serão detalhados os cinco componentes principais da arquitetura ROA: Recurso, Representação, Identificador Uniforme, Interface Unificada e Escopo de Execução.

2.4.3.1 Recurso

Trata-se de uma abstração ou conceito relevante no domínio tratado pelo serviço em questão, ficando a cargo do projetista a seleção de qualquer objeto do domínio, seja ele real ou fictício, concreto ou abstrato. Por exemplo, podemos citar como recursos:

- FURG - Campus Carreiros;
- A localização geográfica de uma determinada cidade;
- A intenção de compra dos clientes que acessam um e-commerce;
- Coleção de motos com motores dois tempos.

Tabela 2.1: Formatos para representação de valores e respectivos cabeçalhos HTTP

Formato	Cabeçalho
XML	application/xml
XHTML	application/xhtml+xml
JSON	application/json
RDF	application/rdf+xml

Conforme pode ser observado pelos exemplos citados, qualquer item que possa ser definido em um objeto sendo considerado e tratado por REST, é um recurso, até mesmo uma coleção de objetos do mesmo tipo (FILHO, 2009).

2.4.3.2 Representação

Na arquitetura REST, os serviços manipulam as representações dos recursos, uma vez que estes são abstratos e não podem ser constituídos fisicamente. Por exemplo, não seria possível trafegar um carro, ou uma pessoa pela rede, mas é possível trafegar sua representação. Desta forma fica claro que conceitualmente uma representação é qualquer conjunto de dados útil sobre o estado de um recurso (RICHARDSON; RUBY, 2007).

Tecnicamente pode-se dizer que uma representação consiste na serialização de um recurso, empregando-se para isso uma sintaxe específica. Dentre as sintaxes mais conhecidas e utilizadas destacam-se o XML, XHTML (Extensible Hypertext Markup Language), JSON e RDF (Resource Description Framework). Cada sintaxe de serialização é definida pelo cabeçalho da requisição, conforme ilustra a Tabela 2.1. Assim como RDF que é uma linguagem que pretende padronizar o uso de XML para definição de recursos, existem outras que empregam o XML para anotação de dados (FILHO, 2009).

Diante da diversidade de opções para serialização de recursos, um serviço web pode, eventualmente, prover acesso empregando diferentes métodos para serializar seus recursos, promovendo assim uma maior interoperabilidade entre sua interface e seus potenciais consumidores. Nestes casos, a requisição deve ser objetiva quanto a notação do que se espera como retorno. Para tanto é utilizado um cabeçalho HTTP chamado Accept. O campo Accept do cabeçalho de requisição HTTP pode ser utilizado para especificar certos tipos de mídias aceitáveis como resposta a requisição (FIELDING, et al., 1999). A Figura 2.8 ilustra a utilização deste campo do cabeçalho HTTP.

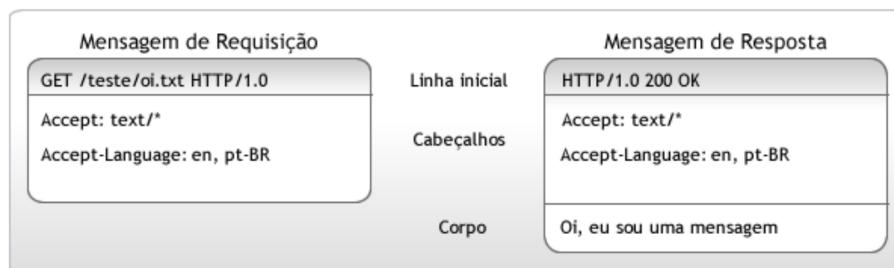


Figura 2.8: Exemplo de requisição e resposta HTTP. Fonte: Adaptada de Gourley e Totty (GOURLEY; TOTTY, 2002).

2.4.3.3 Identificador Uniforme (URI)

Na arquitetura REST, um recurso necessariamente é referenciado por pelo menos um identificador uniforme ou URI (Universal Resource Identifier), que possui as funções de identificação e localização deste recurso. Desta forma caso um objeto não seja referenciado por nenhum URI, este não pode ser considerado um recurso. Por outro lado, um recurso pode ser referenciado por um número ilimitado de URIs (O'REILLY, 2007).

O termo URI é comumente confundido com o termo URL (Universal Resource Locator), porém o URL é na verdade, um subconjunto do endereçamento URI, ou seja, o URI é um esquema de endereçamento mais abrangente e único para cada recurso disponível na Web, seja ele um documento HTML, uma imagem ou um serviço web. Um URI é composto por três partes, conforme ilustra a figura 2.9.



Figura 2.9: Exemplo de URI.

De acordo com o exemplo pode-se afirmar que: o documento “home.html” pode ser acessado através do protocolo HTTP, no domínio www.nce.ufrj.br sob o diretório “/cursos/htmlbasico/”.

Desta forma as três partes que compõe um URI são:

- Protocolo de acesso (`http://`);
- Domínio (endereço onde o recurso encontra-se hospedado);
- O caminho para o recurso propriamente dito (diretório + recurso).

2.4.3.4 Interface Unificada

O paradigma RESTful define que toda ação a ser executada é definida diretamente pelo protocolo HTTP. O protocolo HTTP define cinco métodos principais: GET, HEAD, POST, PUT e DELETE. Todos estes métodos segundo RESTful são aplicáveis aos recursos, ou objetos gerenciados pelo serviço.

O HTTP ainda oferece os métodos OPTIONS, TRACE e CONNECT, embora até o momento estes não tenham sido absorvidos pela arquitetura ROA. O uso destes métodos do protocolo HTTP agrupa simplicidade na exposição de métodos em formato de serviços web, sendo que podem ser acessados por um URI padrão.

Uma vez que o consumidor do serviço conheça os recursos oferecidos, automaticamente conhece os processos de criação, alteração, exclusão e recuperação destes recursos, bastando para isso alterar o método do protocolo na chamada do serviço. Esta característica promove uma maior interoperabilidade e conceitua-se como interface unificada (FILHO, 2009).

O paradigma RESTful como mencionado anteriormente emprega os métodos do HTTP para manipulação de recursos, desta forma, cada método do protocolo HTTP corresponde a uma ação a ser tomada sobre um recurso, tornando-se muito simples a criação e entendimento de casos CRUD (Create, Read, Update, Delete) para a manipulação de recursos.

Tabela 2.2: Utilização dos métodos HTTP e sua equivalência com SQL

Métodos HTTP	Ação	Instrução SQL	Operação
POST	Create	INSERT	Cria um novo recurso, inserindo dados.
GET	Read	SELECT	Obtém os dados de um recurso.
PUT	Update	UPDATE	Atualiza os dados de um recurso.
DELETE	Delete	DELETE	Exclui um recurso e seus dados.

A tabela 2.2 relaciona os métodos HTTP, com a ação, sua equivalência em SQL (Structured Query Language) e descreve a funcionalidade de cada item (RICHARDSON; RUBY, 2007).

Segundo Gonsalves (2009), outros métodos do protocolo HTTP são menos usados:

- HEAD, idêntico ao método GET, exceto que este não transfere o recurso. Por este motivo, é útil para verificar a viabilidade do link, ou obter o tamanho do recurso;
- OPTIONS, que retorna ao consumidor do serviço os dados referentes as opções de comunicação disponíveis para uma requisição/resposta a um recurso especificado por um URI, permitindo assim ao cliente determinar quais as opções e/ou requirementos associados a um recurso, ou as capacidades do servidor, sem que haja necessidade de obter o recurso em si;
- CONNECT, utilizado em conjunto com um proxy que pode dinamicamente ser configurado para iniciar um túnel (uma técnica pela qual o HTTP atua como um empacotador para vários protocolos de rede).

Em outras palavras, a interface unificada é basicamente um contrato para comunicação entre cliente e servidor. São pequenas regras para deixar um componente o mais genérico possível, muito mais fácil de ser refatorado e melhorado.

Dentro desta regra, existe uma espécie de diretriz para fazer essa comunicação uniforme:

- **Identificando o recurso:** Cada recurso deve ter uma URI específica e coesa para poder ser acessado. Na figura 2.10 é ilustrado um exemplo de URI para um recurso produto.



Figura 2.10: Exemplo de URI de acesso para o recurso produto.

- **Representação do recurso:** É a forma como o recurso vai ser devolvido para o cliente. Esta representação pode ser em JSON, XML, HTML, TXT entre outras. Na figura 2.11 é apresentado um retorno do serviço Web no formato JSON, referente aos produtos cadastrados na base de dados do servidor.
- **Resposta auto-explicativa:** Além do que vimos até agora, é necessário a passagem de meta informações (metadados) na requisição e na resposta. Algumas destas informações são: código HTTP da resposta, Host, Content-Type entre outras. Na figura 2.12 é apresentado um exemplo de resposta autoexplicativa após a execução do GET, para a mesma URI recém apresentada, porém utilizando o método POST;

```
{
    produtos:
    [
        {
            "nome" : "Refrigerante Coca-Cola 2L",
            "preco" : 3.89,
            "id" : 1,
            "ativo" : true
        },
        {
            "nome" : "Refrigerante Sprite 2L",
            "preco" : 3.50,
            "id" : 2,
            "ativo" : true
        }
    ]
}
```

Figura 2.11: Exemplo de retorno de dados no formato JSON.

```
{"codigoResposta": "200", "headersRespostaCabecalhos": [{"header": "Server: Apache-Coyote/1.1"}, {"header": "Content-Type: text/plain"}, {"header": "Content-Length: 2"}, {"header": "Date: Sat, 14 Nov 2015 09:37:40 GMT"}]}
```

Figura 2.12: Exemplo de retorno de resposta autoexplicativa no formato de dados JSON para o método POST.

- **Hipermídia:** Esta parte por muitas vezes é esquecida quando falamos de REST. Consiste em retornar todas as informações necessárias na resposta para que cliente saiba navegar e ter acesso a todos os recursos da aplicação. A seguir será apresentado o conceito de HATEOAS (Hypermedia As The Engine Of Application State) para um melhor entendimento sobre a utilização de links nos recursos para informar na resposta as operações possíveis a este recurso e propôr uma possível navegação entre os demais recursos da aplicação.

2.4.3.5 HATEOAS

A Web trata-se de uma enorme rede onde todas as informações estão interligadas, através de hiperlinks. Hateoas (Hypermedia As The Engine Of Application State), embora pareça ser um conceito novo, já é utilizado na Web desde o princípio e nada mais é do que links que fazem uma ligação entre páginas ou estados de uma aplicação. Segundo Fielding (FIELDING, 2000), quando o usuário avança em uma aplicação selecionando links (transições de estado) o resultado é uma próxima página, e essa nova página representa o próximo estado da aplicação. Aplicações que seguem o conceito Restful em sua maioria não contém apenas dados, mas sim fazem uso da hipermídia contendo links para outros recursos. Na figura 2.13 é apresentado um recurso em formato JSON que representa um produto. O recurso é composto por links, sendo a maioria referentes às operações possíveis sobre o próprio recurso e os dois últimos que apontam para um recurso anterior e para um próximo recurso.

2.4.3.6 Escopo de Execução

Quanto ao escopo de execução a abordagem do paradigma RESTful defende o emprego do URI contendo não somente o endereço do recurso, mas também qualquer pa-

```
{  
    "nome" : "Refrigerante Sprite 2L",  
    "preco" : 3.50,  
    "links" : [ {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/",  
        "rel" : "POST"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/xml",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/json",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/html",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/text",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/2",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/xml/2",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/json/2",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/html/2",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/text/2",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/2",  
        "rel" : "PUT"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/2",  
        "rel" : "DELETE"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/1",  
        "rel" : "GET"  
    }, {  
        "href" : "http://localhost:8080/TCCRESTful/rest/produtos/3",  
        "rel" : "GET"  
    } ],  
    "id" : 1,  
    "dataDeCriacao" : "Nov 14, 2015 7:37:40 AM",  
    "dataDeAtualizacao" : "Nov 14, 2015 7:37:40 AM",  
    "ativo" : true  
}
```

Figura 2.13: Exemplo de recurso no formato JSON com links de hipermídia.

parâmetro necessário a identificação única e/ou outros dados para manipulação do recurso afetado.

A figura 2.14 exemplifica uma requisição para o endereço fictício galeria.com, onde supostamente uma galeria de imagens encontra-se acessível e conta com a abordagem RESTful na implementação de sua API de acesso.

```
01 GET /foto/123 HTTP/1.1
02 Host: galeria.com
```

Figura 2.14: Requisição RESTful utilizando o método GET do protocolo HTTP.

No caso ilustrado na figura 2.14, o recurso foto está sendo acessado pelo método GET e o servidor entende que deve retornar o recurso "foto" identificado pelo código 123 passado como parte do URI.

Neste outro exemplo de requisição, apresentado Figura 2.15 é efetuado o acesso ao mesmo URI, porém o método do protocolo HTTP para acesso foi alterado, de GET para DELETE, em relação ao exemplo anterior. Esta alteração basta para o servidor entender que o recurso identificado pelo código 123 deve ser excluído.

```
01 DELETE /foto/123 HTTP/1.1
02 Host: galeria.com
```

Figura 2.15: Requisição RESTful utilizando o método DELETE do protocolo HTTP.

2.4.4 Princípios adicionais ao Paradigma RESTful

Além das características comentadas anteriormente, um serviço que se diz RESTful, pode e deve apresentar mais alguns princípios. São eles, Endereçabilidade (Addressability), Estado Não-Persistente (Statelessness), Conectividade (Connectivity), Cacheable e Layered System. Princípios estes que serão abordados em maiores detalhes a seguir.

2.4.4.1 Endereçabilidade

Segundo DUMBILL (DUMBILL et al., 2001), diferentemente do paradigma RPC (Remote Procedure Call), onde um único URI serve de referência para endereçamento do serviço como um todo, e cada método é representado por um parâmetro a ser passado, não fazendo parte do endereço em si, no paradigma RESTful, cada porção de dados pode ser endereçada, isto é, receber um URI específico para sua referência. Segundo Richardson e Ruby (RICHARDSON; RUBY, 2007), serviços são classificados como endereçáveis quando seu conjunto de dados é exposto como um conjunto de recursos, cada um com seu respectivo URI. As Figuras 2.16 e 2.17 ilustram essa diferenciação no endereçamento de recursos.

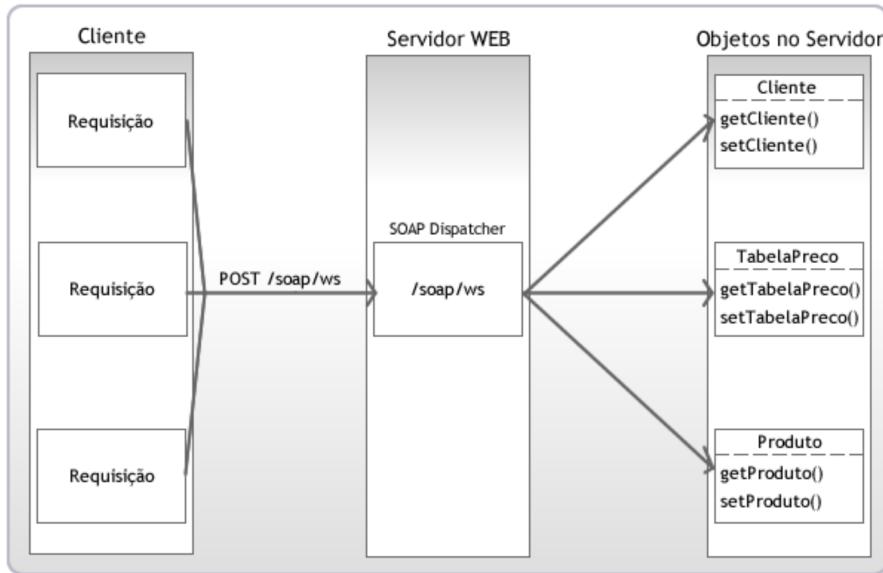


Figura 2.16: Endereçamento no padrão SOAP (XML-RPC).

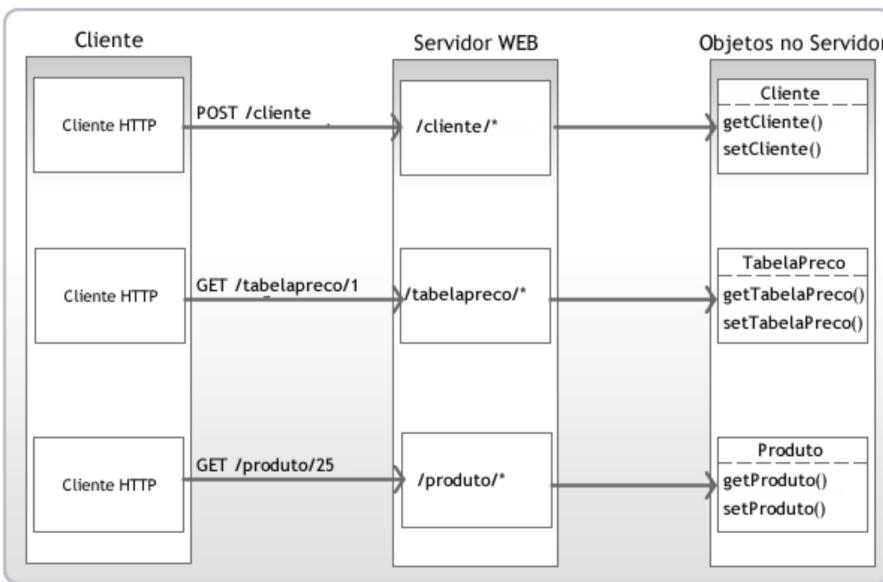


Figura 2.17: Endereçamento de recursos segundo o paradigma RESTful.

2.4.4.2 Estado Não-Persistente

Estado Não-Persistente (Stateless) está relacionado à propriedade do paradigma RESTful na qual cada requisição a um recurso é única. Segundo Richardson e Ruby (RICHARDSON; RUBY, 2007), o conceito de não persistência em uma requisição HTTP está relacionado ao fato de possibilitar o total isolamento da conexão. Sendo assim, um mesmo cliente pode mandar várias requisições para o servidor, porém, cada uma delas devem ser independentes, ou seja, toda requisição deve conter todas as informações necessárias para que o servidor consiga entendê-la e processá-la adequadamente. Neste caso, o servidor não deve guardar nenhuma informação a respeito do estado do cliente. Qualquer informação de estado deve ficar no cliente, como as sessões, por exemplo.

De forma mais prática, o paradigma RESTful dita que todo pedaço de informação importante, deve ser apresentado como um recurso, tendo um URI de acesso próprio. A técnica de Estado Não-Persistente diz por sua vez que, cada estado deve também ser endereçado por seu próprio URI. O cliente não deve ter que induzir o servidor para um determinado estado, para torná-lo receptivo a determinada requisição.

2.4.4.3 Conectividade

Segundo Richardson e Ruby (2007), “um recurso deve apontar para outros em sua representação”. Gonsalves (2009) faz uma analogia a teoria de grafos, onde enfatiza que nesta teoria, um grafo é dito conectado se todos os pares de vértices distintos deste puderem ser conectados através de algum caminho. Assim como fortemente conectado se contiver um caminho direto de u para v e um caminho direto de v para u, para cada par de vértices u, v. REST por sua vez prega que os recursos devem ser o mais conectado possível. Deste modo, formula que conectividade pode ser associada à RESTful como: “Hipermídia com um motor de estado de aplicação”.

Em outras palavras o Princípio da Conectividade em ROA está ligado a representações conectadas a outras representações, ou seja, os recursos gerenciados pelo serviço apresentam-se conectados entre si. É correto ainda fazer uma analogia do Princípio da Conectividade em ROA com a utilização de hiperlinks para realizar as ligações dos documentos de hipermídia disponibilizados na Web, em que os usuários navegam entre diferentes páginas apenas seguindo links que apontam de uma página à outra.

2.4.4.4 Cacheable

Como na World Wide Web, os clientes podem armazenar respostas em cache. As respostas devem ser implicitamente ou explicitamente definidas como cacheável, ou não, para impedir que clientes reutilizem dados obsoletos ou impróprios em resposta a novos pedidos. Quando bem gerenciado, o cache elimina parcialmente ou completamente as interações entre cliente e servidor, melhorando ainda mais a escalabilidade e o desempenho do serviço.

2.4.4.5 Layered System

Este princípio define que a aplicação deve ser composta por camadas, e estas camadas devem ser fáceis de alterar, tanto para adicionar mais camadas, quanto para removê-las. Dito isso, um dos princípios desta restrição é que o cliente nunca deve chamar diretamente o servidor da aplicação sem antes passar por um intermediador, no caso, pode ser um load balancer (balanceador de carga) ou qualquer outra máquina que faça a interface com o(s) servidor(es). Isso garante que o cliente se preocupe apenas com a comunicação

com o intermediador e o intermediador fica responsável por distribuir as requisições nos servidores, seja um ou mais, o que indifere nesse caso. O importante é ficar claro que criando um intermediador, a sua estrutura fica muito mais flexível à mudanças. A imagem 2.18 ilustra a arquitetura de um serviço web através de Layered System ou Load Balancer.

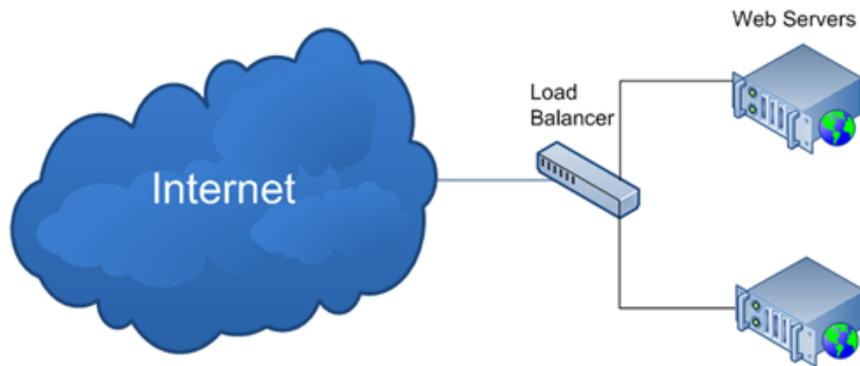


Figura 2.18: Serviço Web através de Layered System. Fonte: Gabriel Almeida

2.4.5 Modelos de Maturidade

Modelos de maturidade são importantes pois oferecem mecanismos para avaliar e comparar a qualidade de Web APIs. Richardson Maturity Model e CoHA são dois modelos de maturidade que podem ser utilizados para classificar e avaliar implementações de Web APIs. A seguir estes dois modelos serão apresentados.

2.4.5.1 Richardson Maturity Model

Richardson Maturity Model (RMM) classifica o design de Web APIs em quatro níveis de maturidade (WEBBER; PARASTATIDIS; ROBINSON, 2010). A figura 2.19 apresenta os níveis do modelo, partindo do mais baixo ao topo da pirâmide, ou seja, a glória do REST.

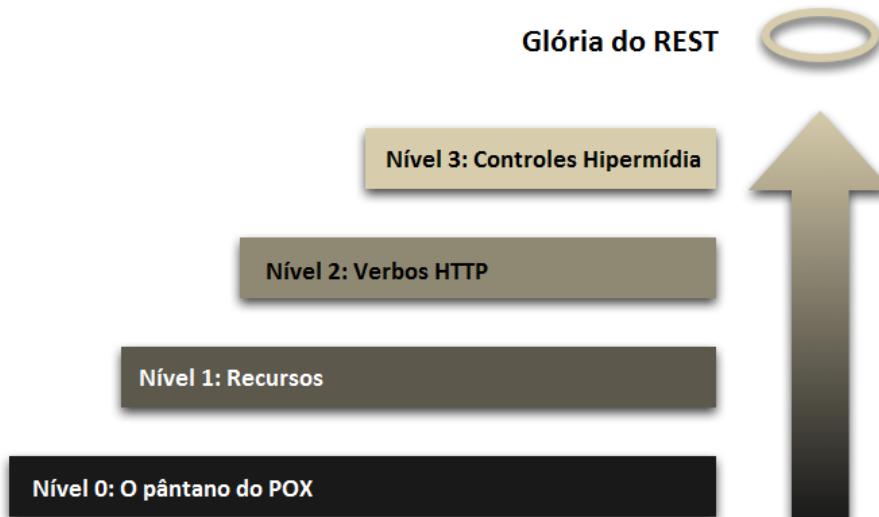


Figura 2.19: Níveis de maturidade RMM. Fonte adaptada de Fowler (FOWLER, 2010)

Este modelo quebra os principais elementos de uma funcionalidade REST em três passos. Estes introduzem recursos, verbos HTTP e controles de hipermídia.

Quanto aos níveis de maturidade, são descritos nos seguintes itens:

- **Nível 0:** Resumidamente é a ausência de qualquer regra, é apenas a utilização do HTTP como transporte das operações no servidor. Normalmente se usa apenas um endpoint (URI) e um verbo HTTP. O formato mais utilizados é XML, utilizado como uma espécie de envelope, denominado Plain Old XML (POX). A comunicação é realizada através do estilo RPC (Remote Procedure Call), que concentra todos os serviços em um único ponto de acesso;
- **Nível 1:** Exige que as informações sejam estruturadas na forma de recursos. A API é dividida em diferentes endpoints que apontam para um ou mais recursos. Cada recurso é identificado e recebe um endereço, permitindo sua manipulação individual. Os recursos também disponibilizam um conjunto de operações que são aplicadas sobre os próprios recursos;
- **Nível 2:** Implementação de verbos HTTP para diferentes tipos de operações que deseja executar. Uma mesma URI pode aceitar mais de um verbo HTTP, por exemplo: "GET /user" pode retornar todos os usuários e "POST /user" passando os atributos do usuário pode criar um novo. As quatro operações básicas disponibilizadas pelo protocolo HTTP são GET, POST, PUT e DELETE. Neste nível de maturidade exige-se o uso correto dessas operações sobre os recursos. Isto implica que recursos devem ser criados através da operação POST, alterados por PUT, recuperados por GET e removidos por DELETE. Esse nível exige também o uso correto de status code, para descrever corretamente as respostas das requisições realizadas;
- **Nível 3:** Este nível de maturidade exige a aplicação de HATEOAS, que adiciona controles hipermídia às representações. O uso de controles hipermídia permite aos clientes da Web API manipular os recursos de forma exploratória e desacoplada dos detalhes de implementação.

2.4.5.2 CoHA Maturity Model

Classification of HTTP-based APIs (CoHA), proposto por Algermissen (ALGERMISSEN, 2010), define cinco níveis de maturidade. CoHA é uma modelo de maturidade mais amplo, pois considera implementações de sistemas baseados em HTTP, sem ficar restrito à implementações REST. Os níveis deste modelo são apresentadas através da figura 2.20.

O nível mais baixo de maturidade é denominado WS-* e se aplica a implementações baseadas na tecnologia SOAP que tratam o protocolo HTTP apenas como mecanismo de transporte de dados. Nesse nível, URIs identificam apenas serviços, todas as informações necessárias para a sua execução são encapsuladas no envelope SOAP. É considerada a pior abordagem do ponto de vista de custo/benefício, pois apresenta alto custo de adoção, possui dependência com fornecedores de tecnologia além de resultar em alto custo para evolução dos sistemas.

O segundo nível é denominado RPC URI-Tunneling, o qual exige que as informações sejam estruturadas como recursos, embora as operações não respeitem necessariamente a semântica do protocolo utilizado pela Web API. Embora os recursos sejam identificáveis, esse nível não garante a manipulação de recursos através de várias representações. Esse

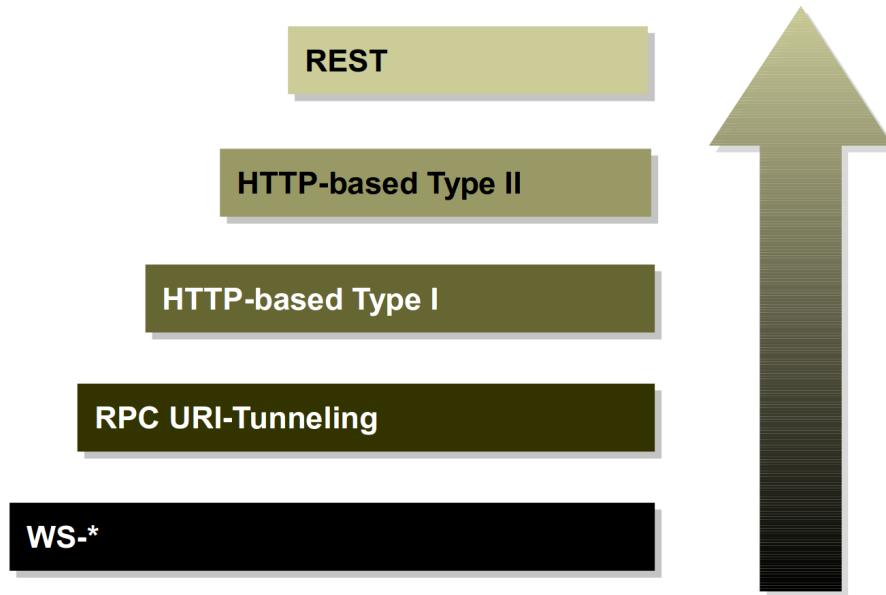


Figura 2.20: Exemplo de sintaxe básica do JSON. Fonte adaptada de Algermissen (ALGERMISSSEN, 2010).

nível resulta em alto acoplamento entre cliente e servidor e dificulta a evolução independente entre as partes.

O terceiro nível é denominado HTTP-based Type I e exige que recursos sejam manipulados de forma a respeitar a semântica do protocolo utilizado pela Web API. Os recursos são manipulados através de representações, permitindo utilizar diferentes formatos de dados. Esse nível exige o comportamento stateless do servidor, ampliando a capacidade de escalabilidade da Web API. Possui baixo custo de adoção, entretanto, possui alto custo de evolução devido ao alto acoplamento entre clientes e servidor.

O quarto nível é denominado HTTP-based Type II e exige que as mensagens sejam auto-descritivas. Define que os recursos, operações e formatos de dados sejam descritos no momento do design. Web APIs implementadas com esse nível de maturidade são consideradas simples na perspectiva da interação cliente-servidor, pois os recursos são manipulados através de uma interface uniforme.

O quinto e último nível é denominado REST pois obedece a todas as restrições descritas pelos princípios da arquitetura. Os recursos são identificáveis e endereçáveis, manipulados através de representações, possuem controles de hipermídia e as mensagens são auto-descritivas. Esse nível de maturidade exige alta curva de aprendizado, dificultando a adoção. Entretanto, apresenta o menor custo de manutenção e evolução, pois as mudanças na Web API não causam impactos nas aplicações clientes.

2.4.6 Descrição dos Serviços

Em REST ainda não existe uma linguagem ou padrão bem formado como meio de descrição de serviços, diferente da arquitetura SOAP, que tem a linguagem WSDL bem definida para este fim. No entanto, existe a linguagem WADL que vem demonstrando ser promissora para ocupar o lugar de instrumento padrão de descrição de serviços RESTful. No próximo item deste capítulo será apresentado uma breve abordagem sobre o WADL.

2.4.6.1 WADL

Ao formular o padrão WADL, Hadley (HADLEY, 2009), membro integrante do grupo de pesquisas da Sum Microsystems, elencou alguns pontos positivos de sua utilização, entre eles:

- Suporte ao desenvolvimento de ferramentas para modelagem de recursos;
- Geração automática de código para a manipulação de recursos;
- Configuração de cliente e servidor, a partir de um único formato portável.

Memso com a expectativa do WADL ocupar o lugar de instrumento padrão de descrição de serviços RESTful, por se tratar de um ambiente fechado e controlado, não é desejável nem aceitável o consumo dos serviços por terceiros. Portanto, pesquisas sobre este assunto não foram aprofundadas.

2.4.7 JSON

JSON ou JavaScript Object Notation é um formato de documento leve para transferência de dados com sintaxe definida no padrão ECMAScript/Javascript para definição de estruturas de dados em JavaScript, esse formato pode ser facilmente lido ou escrito tanto por humanos quanto por máquinas. O JSON é um subconjunto da notação de objetos JavaScript, mas seu uso não requer JavaScript exclusivamente. A figura 2.21 ilustra a sintaxe básica para uma simples representação de alunos, sendo que cada aluno possui os atributos nome e sobrenome.

```
{
  "alunos" :
  [
    {"nome": "Fulano", "sobrenome": "de Tal"},
    {"nome": "Beltrano", "sobrenome": "da Silva"},
    {"nome": "Sicrano", "sobrenome": "da Silva Sauro"}
  ]
}
```

Figura 2.21: Exemplo de sintaxe básica do JSON.

2.5 Arquitetura REST em Java

Neste capítulo será apresentado uma abordagem geral sobre a arquitetura RESTful em Java, com destaque para o JAX-RS (Java API for RESTful Web Services) e o Jersey, uma API em Java para o desenvolvimento RESTful, o qual é uma espécie de extensão do JAX-RS, onde implementa as suas especificações além de outras funcionalidades, tornando mais fácil e rápido o desenvolvimento de Web Services RESTful em Java e desenvolvimento cliente.

2.5.1 JAX-RS

Para dar suporte a REST em Java, a comunidade através de um JCP(Java Community Process) desenvolveu a especificação JSR-311 (Java Specification Request 311), mais conhecida como JAX-RS (Java API for RESTful Web Services). Segundo Silveira and others (SILVEIRA; OTHERS., 2013).

O JAX-RS, lançado pela Sun em 2007, é a especificação do Java para serviços REST, e pode ser utilizada juntamente com outras tecnologias do JavaEE, como por exemplo fazer integração com EJBs (Enterprise JavaBeans).

Os objetivos desta especificação são:

- Desenvolvimento baseado em Pojos (Plain Old Java Object) onde as classes que representam esses objetos são disponibilizadas como recursos;
- Foco no HTTP;
- Inclusão no JavaEE;
- Independência de contêiner, podendo ser portátil e qualquer contêiner JEE;
- Independência de formato.

O JAX-RS implementa um recurso web como uma classe recurso e requisições são processadas por métodos recursos.

2.5.2 Arquitetura RESTful em JAX-RS (JSR 311)

A especificação JAX-RS é baseada em anotações ou @annotations para a implementação de Web Service RESTful, com base em Java e HTTP.

As classes e os métodos são anotados com @annotations que permitem expô-los como recursos, foco principal do REST. Utiliza URI's nas @annotations para nomear os recursos. A implementação de referência da JSR-311 é o Jersey, mas existem outras implementações, entre elas o Restlet, RESTEasy, Apache CXF entre outros.

2.6 Trabalhos Relacionados

A seguir serão apresentados alguns trabalhos relacionados ao desenvolvimento de serviços RESTful e ao final do capítulo um resumo comparativo dos trabalhos relacionados com o framework proposto.

2.6.1 Spring-HATEOAS

Spring-HATEOAS é um framework destinado ao desenvolvimento de Web Services baseados em hipermídia na linguagem de programação Java. As representações de recursos são disponibilizadas no formato HAL (Hypermedia Application Language). Spring-HATEOAS apresenta o conceito de Resource Representation Class (Representação de Classe de Recurso), que descreve apenas as propriedades da representação. As operações ficam separadas em classes Resource Controller (Controle de Recursos), que manipulam as classes de representação e adicionam os links necessários, como pode ser visualizado através da figura 2.22. HAL é um formato para expressar controles hipermídia em documentos JSON e XML. Embora o formato JSON HAL permita representar controles hipermídia diretamente no documento JSON, os detalhes de execução necessitam de interpretação humana (HATEOAS, 2013).

Este projeto fornece algumas APIs para facilitar a criação de representações REST que seguem o princípio HATEOAS quando trabalha-se com a Spring e, especialmente, Spring MVC. O principal problema que tenta resolver é a criação de links em representações de recursos.

O Spring-HATEOAS conta com as seguintes funcionalidades:

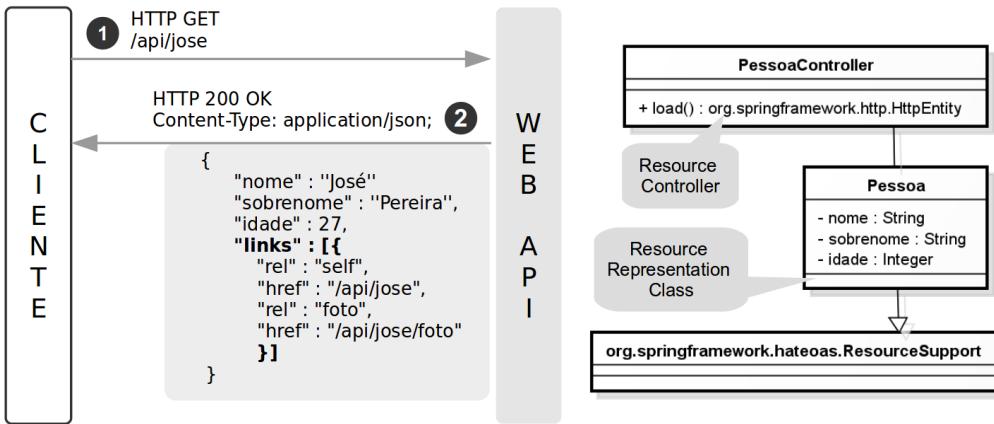


Figura 2.22: Exemplo de retorno JSON a partir de uma requisição GET e diagrama de classes ilustrando o modelo de entidade do recurso usando o Spring-HATEOAS.

- Classes de modelo para ligação, modelos de representação de recurso;
- Classe Link para criar links apontando para métodos do controlador Spring MVC;
- Suporte para formatos hipermídia como HAL.

2.6.2 Apache Isis

Apache Isis (BIENVENIDO, 2013) é uma implementação Java da especificação RESTful Objects, a qual é voltada para o desenvolvimento de aplicações RESTful que permite que clientes tenham acesso aos objetos de domínio através de HTTP, tendo como retorno representações de recursos no formato JSON (HAYWOOD, 2012). Ele é um framework extensível e personalizável para desenvolvimento rápido de aplicações orientadas a domínio e que faz uso de anotações para orientar a exposição dos objetos de domínio. Utiliza um modelo arquitetural hexagonal, no qual os objetos de domínio assumem a posição central, enquanto o framework se responsabiliza pela persistência, segurança e apresentação. Pela sua natureza extensível e customizável, o framework suporta diversas tecnologias, sendo que RESTful Objects é apenas uma delas. No RESTful Objects, as classes de domínio podem expor propriedades, operações e coleções que referenciam outras entidades. RESTful Objects permite representar dados e especificar tipos, além de vincular links com informações completas de execução. Sendo assim, ele permite o controlo hipermídia que alteram o estado dos recursos, inclusive a nível individual de propriedades. De maneira resumida, o framework funciona usando o conceito de "convenção sobre configuração", no qual desenvolvedores escrevem objetos POJOs de domínio, seguindo um conjunto de convenções e anotações. Essas informações são então interpretadas pelo framework. O Apache Isis pode gerar em tempo de execução uma representação do modelo de domínio na forma de um aplicativo web ou como API RESTful, seguindo a especificação Restful Objects.

Na figura 2.23 é apresentada a arquitetura do Apache Isis.

Apache Isis é uma solução ampla que abrange diferentes aspectos do sistema. A interface RESTful é apenas um pequeno módulo do framework. Outros módulos tratam questões relacionadas à persistência, segurança, testes unitários, comunicação remota, dentre outras. A abrangência do Apache Isis é um fator fundamental na decisão de sua

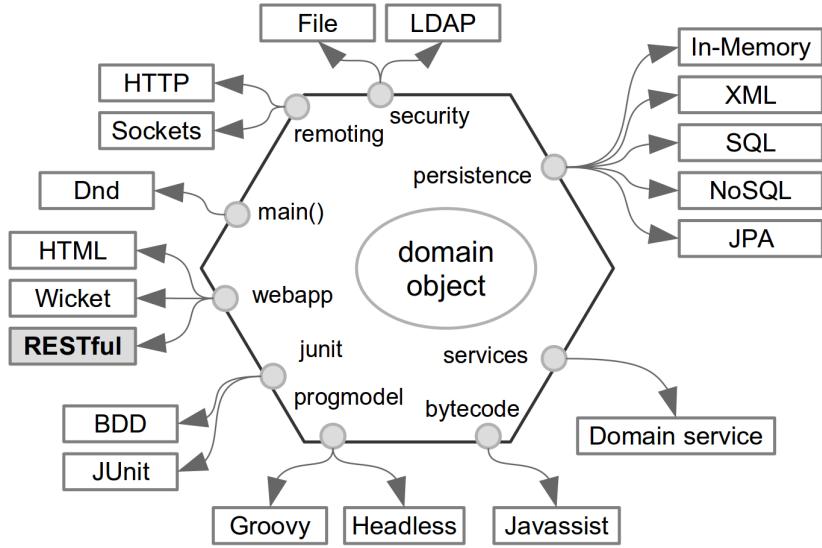


Figura 2.23: Arquitetura Apache Isis. Adaptado de BIENVENIDO (BIENVENIDO, 2013).

adoção, pois para usufruir das características da integração REST, é imperativo adotar todo modelo proposto pelo framework.

2.6.3 Framework de John e Rajasree

John e Rajasree (JOHN; RAJASREE, 2012) propõem um framework para descrição, descoberta e composição de serviços RESTful através de anotações semânticas na documentação da Web API. O framework realiza anotações em documentos HTML utilizando a combinação entre marcações de Microformats e RDFS.

A Figura 2.24 mostra como o framework realiza as anotações semânticas. O elemento `hresource` é a anotação base para a descrição do recurso, onde todas as demais anotações estão encapsuladas. A anotação `name` é utilizada para dar um nome ao recurso. Através da anotação `url` é possível associar o endereço onde o recurso é disponibilizado. Todas as propriedades do recurso devem ser anotadas através da tag `<attribute>`. Embora enriquecer semanticamente a documentação de Web APIs proporcione maior integração com agentes autônomos, o cliente fica restrito à documentação HTML como único guia para interação com a Web API.

2.6.4 Especificação JAX-RS

A especificação JAX-RS (Java API for RESTful Web Services) define um conjunto de APIs para o desenvolvimento de serviços Web, empregando a linguagem de programação Java, que obedecem aos princípios arquiteturais REST (ORACLE, 2012). JAX-RS proporciona um conjunto de anotações capazes de expor classes POJOs (Plain Old Java Objects) como recursos. A especificação JAX-RS adota o HTTP como protocolo para o transporte das representações, e oferece o suporte para a utilização correta da semântica do protocolo, ou seja, permite utilizar os verbos (GET, POST, PUT, DELETE), formatos (JSON, XML, HTML, Text entre outros) e códigos de mensagens de respostas. Na figura 2.25 é apresentado um exemplo de utilização de JAX-RS.

No exemplo da figura 2.25 são ilustradas algumas anotações do JAX-RS. Através da

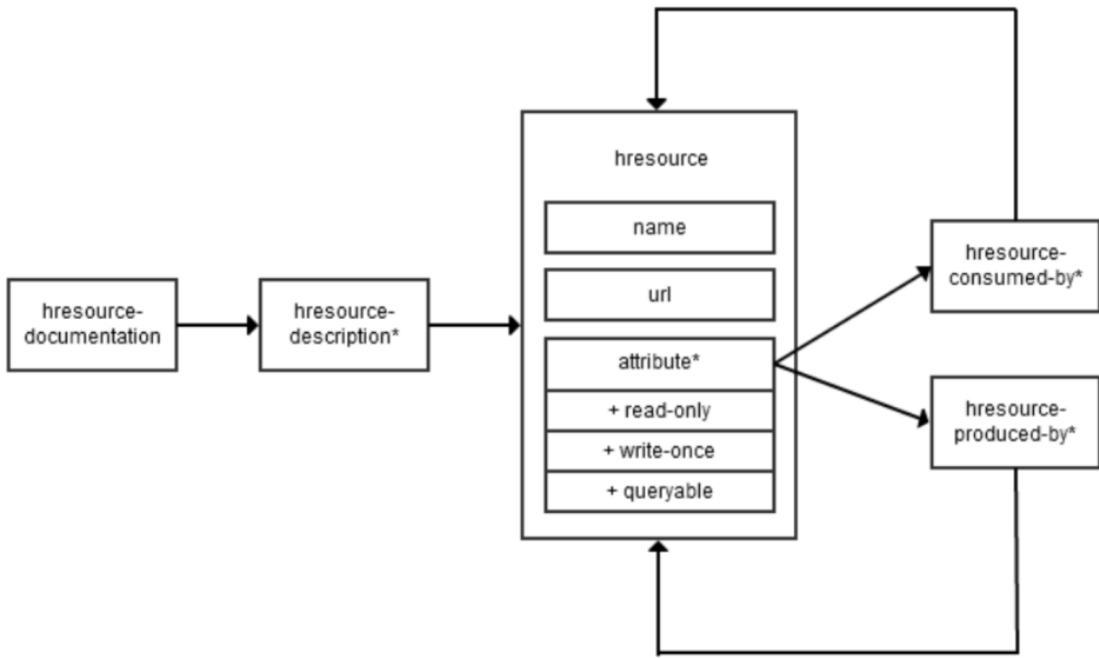


Figura 2.24: Visão geral do framework. Fonte: (JOHN; RAJASREE, 2012).

anotação @Path aplicada sobre uma classe, cria-se um recurso Web, dados de um produto por exemplo, que pode ser acessado através da URI /produto/id. Para tornar o endereçamento dos recursos mais flexível, uma URI pode conter variáveis que são mapeadas para propriedades. A variável é mapeada para uma propriedade da classe através da anotação @PathParam, que especifica o nome da variável. Quatro métodos são implementados para permitir a manipulação dos recursos, que são anotados com @GET, @POST, @PUT e @DELETE e criam a correspondência entre os tipos de requisições HTTP e a execução dos métodos anotados. As anotações @Consumes e @Produces são responsáveis por mapear os tipos de dados de entrada e saída, respectivamente, aceitos por cada método. Os argumentos que constituem a assinatura dos métodos são automaticamente mapeados do formato aceito descrito pela anotação @Consumes para o argumento referenciado. Cada método retorna um objeto do tipo Response, que representa a resposta a ser enviada pelo protocolo HTTP. O objeto Response retornado pelo método contém o código de status HTTP para a requisição e, opcionalmente, uma representação de recurso vinculado, denominada entity (entidade). Essa representação, quando presente, é automaticamente mapeada para o formato indicado pela anotação @Produces. Caso mais de um formato seja suportado, é escolhido o formato de preferência do cliente, indicado no cabeçalho da requisição HTTP.

```

package test;

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

@Path("produto/{id}")
public class ProdutoJaxRS {

    @PathParam("id")
    private Long id;
    private String nome;
    private String preco;

    @GET
    @Produces(MediaType.APPLICATION_JSON)

    public Response carregar() {
        // codigo para carregar uma pessoa
        return null;
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response criar(Produto p) {
        // codigo para criar pessoa p
        return null;
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)

    public Response atualizar(Produto p) {
        // codigo para alterar Produto p
        return null;
    }

    @DELETE
    public Response remover() {
        // codigo para remover uma pessoa
        return null;
    }
}

```

Figura 2.25: Exemplo de utilização de JAX-RS.

2.6.5 API Jersey

O Jersey é a implementação de referência da JSR-311, onde sem problemas suporta expor seus dados em uma variedade de tipos de mídia de representação e abstrair os detalhes de baixo nível da comunicação cliente-servidor (JERSEY - RESTFUL WEB SERVICES IN JAVA, 2015). Sua principal característica é simplificar o desenvolvimento de serviços Web RESTful e seus clientes em Java, ou seja, ele fornece a sua própria API que estende o kit de ferramentas JAX-RS com características e utilitários adicionais para simplificar ainda mais o serviço RESTful e desenvolvimento cliente. Jersey também expõe numerosas SPIs (protocolos que permitem a comunicação do microcontrolador com diversos outros componentes, formando uma rede) de extensão para que os desenvolvedores possam estender o Jersey para melhor atender às suas necessidades.

Os principais objetivos do projeto podem ser resumidos nos seguintes itens:

- Acompanha a API JAX-RS e fornece lançamentos regulares de implementações de referência de qualidade da produção que vem com o GlassFish - servidor de aplicação open source liderado pela Sun Microsystems para a plataforma J2EE;
- Fornecer APIs para estender o Jersey e construir uma comunidade de usuários e desenvolvedores;
- Tornar mais fácil para construir serviços Web RESTful, utilizando Java e Java Virtual Machine.

2.6.6 Resumo Comparativo dos Trabalhos Relacionados

A seguir, a tabela 2.3 apresenta uma comparação entre os trabalhos relacionados e o framework proposto para este trabalho. Dentre as características comparadas estão: o controle de hipermídia, respostas auto-explicativas, classe cliente para implementação de Layered System, Cache e o ponto de aplicação da proposta.

Tabela 2.3: Tabela de comparação entre as ferramentas para desenvolvimento de serviços RESTful

	Spring-HATEOAS	Apache Isis	Framework de John e Rajasree
Controle de hipermídia	Parcial	Sim	Não
Respostas auto-explicativas	Não informado	Não informado	Não
Classe cliente para implementação de Layered System	Não	Não	Não
Cache	Não informado	Não informado	Não
Aplicado sobre	Código	Código	Documentação
	Especificação JAX-RS	API Jersey	Framework proposto
Controle de hipermídia	Não	Parcial	Sim
Respostas auto-explicativas	Parcial	Parcial	Sim.
Classe cliente para implementação de Layered System	Não	Não	Sim
Cache	Não informado	Não informado	Não
Aplicado sobre	Código	Código	Código

Após analisar as ferramentas, foi possível observar que referente ao suporte a controles de hipermídia, o Spring-HATEOAS apresenta parcialmente a funcionalidade, pois utiliza o formato de dados JSON HAL que exige a intervenção humana para ser possível interpretar os detalhes das requisições. O Apache Isis e o Jersey 2.0 também possuem suporte ao controle de hipermídia, porém em relação ao Apache Isis, nada foi encontrado a respeito da sua implementação nas referências bibliográficas pesquisadas. Já o Jersey conta com uma implementação bastante complexa na classe *Link* disponibilizada pela API. O framework proposto apresenta uma classe *Link* simples o suficiente para a representação de links, sem complexidade. É possível armazenar links de imagens, por exemplo, sem grandes esforços.

Para as respostas auto-explicativas, nada foi encontrado a respeito para o framework Spring-HATEOAS e o Apache Isis. Na especificação JAX-RS e no Jersey 2.0 as respostas autoexplicativas são parciais, visto que as respostas são baseadas em Response do pacote `HttpResponde` do Java ou então no String para as saídas XML e JSON. No framework proposto as respostas autoexplicativas são observadas em todos os métodos, sendo que quando trabalha-se com formato json, o retorno autoexplicativo é JSON. O mesmo acontece quando trabalha-se com um formato xml, por exemplo. Todavia, se for trabalhar com algum formato diferente de JSON e XML, o retorno padrão é JSON, visto que este é o formato padrão de documento estruturado utilizado pelo framework.

Em relação a existência de uma classe cliente para implementação de Layerd System, em nenhum framework foi encontrado esta funcionalidade. A especificação JAX-RS e o Jersey 2.0 contam com classes e métodos para realizar requisições POST, GET, PUT e DELETE, porém não existe uma classe apenas que faça, com o intuito de centralizar as operações e funcionalidades. No framework proposto foi desenvolvido a classe *InterceptorClienteRESTful* para implementação de Layered System e uso em aplicações cliente de mesma tecnologia, no caso o Java.

Quanto ao cache, o Jersey conta com tal implementação, quanto aos demais, nada foi encontrado a respeito nas referências bibliográficas pesquisadas. O framework proposto também não dispõe desta funcionalidade, o que torna um trabalho futuro a ser implementado. Entretanto, desenvolvedores podem utilizar-se de objetos nativos de classes do Java para implementação de cache no cliente.

Resumindo, o framework proposto se propõe a implementar alguns itens comparativos da tabela, além de disponibilizar novas funcionalidades e guiar o desenvolvedor a seguir as regras para a construção de Web Services RESTful nível três de maturidade.

No próximo capítulo será apresentado o projeto que foi desenvolvido neste trabalho.

3 PROJETO

Este capítulo divide-se em dois subcapítulos.

O subcapítulo 3.1 apresenta a modelagem e implementação do framework RESTful proposto para este trabalho.

Já no subcapítulo 3.2 é apresentado a modelagem e implementação referente ao estudo de caso, que trata-se do desenvolvimento e implementação do Web Service JavaEE utilizando as classes do framework proposto, além de duas aplicações cliente que consomem os recursos disponíveis deste serviço web. Para as aplicações cliente, uma foi escrita utilizando a linguagem de programação Java, na plataforma de desenvolvimento JavaSE (Java Standard Edition), voltada a aplicações desktop, e a outra utilizando PHP, sendo esta última uma aplicação Web.

3.1 Framework

A seguir será apresentado o diagrama de classes referente ao framework RESTful proposto a este trabalho, além do código-fonte das classes, ilustrando o desenvolvimento e as regras de negócio aplicadas.

3.1.1 Diagrama de Classes

O diagrama de classes mostrado na figura 3.1 ilustra a representação da estrutura e relação das classes do framework.

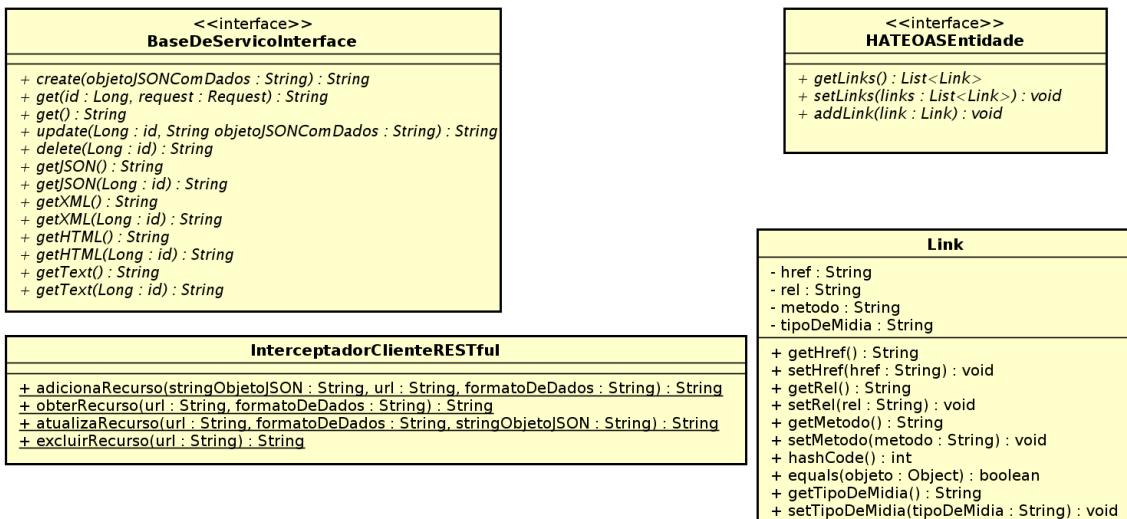


Figura 3.1: Diagrama de classes do framework.

Neste diagrama, a classe interface *BaseDeServicoInterface* implementa a assinatura de métodos para todas as operações CRUD de um recurso. Respectivamente, os métodos *create*, *get*, *update* e *delete* são compostos pelas anotações @POST, @GET, @PUT e @DELETE, referentes aos métodos do protocolo HTTP, sendo estas anotações da API Jersey 2.0, herdadas da especificação JAX-RS. Além disso, a classe respeita a correta utilização dos verbos HTTP, sendo que o padrão de representação e de consumo de recursos é o JSON, porém para as representações GET existem assinaturas de métodos com retornos em diferentes formatos, tais como XML, HTML e Text.

A classe *InterceptadorClienteRESTful*, além de implementar a interceptação entre o cliente e servidor, retorna respostas autoexplicativas em relação a execução de métodos do protocolo HTTP, tais como POST, GET, PUT e DELETE. Ela também é uma classe cliente, mas neste caso só pode ser utilizada se a aplicação cliente for em java. Esta classe será abordada em todos os sentidos e apresentada no subcapítulo 3.2, quando apresentada a sua utilização.

Na classe interface *HATEOASEntidade* as assinaturas dos métodos são equivalentes a manipulação de links que uma classe de entidade pode ter. Este exemplo também é apresentado no estudo de caso com mais detalhes, quando de fato esta interface for usada na implementação de representação de um recurso com hipermídia. A classe *HATEOASEntidade* é composta por uma lista de objetos da classe *Link*.

A classe *Link* contém basicamente a estrutura de uma tag ou marcação *<link>* do html e conta com os seguintes atributos:

- **href:** Localização do documento ligado;
- **rel:** Relação entre o documento corrente e o documento ligado;
- **metodo:** Relaciona o método HTTP a ser executado em conjunto com href.

A seguir será apresentado o código-fonte em Java referente a implementação das classes do framework RESTful.

3.1.2 Implementação

Este capítulo descreve os detalhes de implementação das classes do framework proposto para este trabalho.

Através da figura 3.2, a classe de interface *BaseDeServicoInterface* pode ser analisada. Para os métodos *create*, *get*, *update* e *getJSON*, o formato de dados para consumo e produção é o JSON. Porém, os métodos *getXML* retornam dados no formato XML, enquanto que nos métodos *getHTML* as anotações são para retornos em HTML. Por fim, os métodos *getText* retornam dados em formato de texto. Vale observar que existe uma limitação da API Jersey 2.0 para a criação de dois ou mais métodos que utilizem a anotação @POST em uma mesma classe de recurso. Neste caso, tendo a necessidade de consumo de dados em formato JSON e XML para o método POST, por exemplo, seria necessário criar duas classes de recurso, sendo uma para cada fim. Já as anotações @GET, @PUT e @DELETE podem existir em mais de um método nas classes de recurso. Neste projeto, como pode ser visto na figura 3.2, o método *create(String objetoJSONComDados)* que carrega a assinatura @POST, utiliza somente o formato de dados JSON como consumo.

Na figura 3.3 é apresentada a classe de interface *HATEOASEntidade*, o qual é disponibilizada para ser implementada pelas classes de entidade, referente ao modelo de domínio de um determinado objeto. A classe trabalha com uma lista de objetos da classe *Link*,

```

1 package servicos;
2 import javax.ws.rs.Consumes;
3 import javax.ws.rs.DELETE;
4 import javax.ws.rs.GET;
5 import javax.ws.rs.POST;
6 import javax.ws.rs.PUT;
7 import javax.ws.rs.Path;
8 import javax.ws.rs.PathParam;
9 import javax.ws.rs.Produces;
10 import javax.ws.rs.core.MediaType;
11 @Consumes(MediaType.APPLICATION_JSON)
12 @Produces(MediaType.APPLICATION_JSON)
13 public interface BaseDeServicoInterface {
14     @POST @Consumes(MediaType.APPLICATION_JSON)
15     public String create(String objetoJSONComDados);
16
17     @GET @Path("/{id}") @Produces(MediaType.APPLICATION_JSON)
18     public String get(@PathParam("id") Long id);
19
20     @GET @Produces(MediaType.APPLICATION_JSON)
21     public String get();
22
23     @PUT @Path("/{id}") @Consumes(MediaType.APPLICATION_JSON)
24     public String update(@PathParam("id") Long id, String objetoJSONComDados);
25
26     @DELETE @Path("/{id}")
27     public String delete(@PathParam("id") Long id);
28
29     @GET @Path("/json") @Produces(MediaType.APPLICATION_JSON)
30     public String getJSON();
31
32     @GET @Path("/json/{id}") @Produces(MediaType.APPLICATION_JSON)
33     public String getJSON(@PathParam("id") Long id);
34
35     @GET @Path("/xml") @Produces(MediaType.APPLICATION_XML)
36     public String getXML();
37
38     @GET @Path("/xml/{id}") @Produces(MediaType.APPLICATION_XML)
39     public String getXML(@PathParam("id") Long id);
40
41     @GET @Path("/html") @Produces(MediaType.TEXT_HTML)
42     public String getHTML();
43
44     @GET @Path("/html/{id}") @Produces(MediaType.TEXT_HTML)
45     public String getHTML(@PathParam("id") Long id);
46
47     @GET @Path("/text") @Produces("text/plain")
48     public String getText();
49
50     @GET @Path("/text/{id}") @Produces("text/plain")
51     public String getText(@PathParam("id") Long id);
52 }

```

Figura 3.2: Classe BaseDeServicoInterface parte 1.

classe que será apresentada a seguir, justamente para implementar a funcionalidade de conectividade entre os recursos, ou seja, o controle de hipermídia em recursos.

```

1 package servicios;
2 import java.util.List;
3
4 public interface HATEOASEntidade {
5     public List<Link> getLinks();
6     public void setLinks(List<Link> links);
7     public void addLink(Link link);
8 }
```

Figura 3.3: Classe HATEOASEntidade.

A figura 3.4 ilustra a implementação da classe *Link*. Conta com os atributos href, que encarrega-se de armazenar o endereço de localização do documento ligado ao recurso; rel, para definir a relação entre o documento corrente e o documento ligado; e método, responsável por relacionar o método HTTP a ser executado em conjunto com href.

Por fim, é apresentado a classe *InterceptadorClienteRESTful* através das figuras 3.5, A.1, A.2, A.3 e A.4, sendo que as figuras A.1, A.2, A.3 e A.4 estão localizadas no apêndice deste documento.

A classe foi implementada para ser utilizada na intermediação entre o cliente e servidor, porém ela também é utilizada como cliente e por basear-se em parâmetros de entrada nos métodos, é possível trabalhar com todos os formatos suportados pela especificação JAX-RS, desde que implementados no serviço web. Como já mencionado, a classe também retorna respostas autoexplicativas aos métodos POST, GET, PUT e DELETE.

```

1 package servicos;
2
3@import javax.xml.bind.annotation.XmlAttribute;
4 import javax.xml.bind.annotation.XmlType;
5
6 @XmlType(namespace="http://www.w3.org/1999/xlink")
7 public class Link {
8     private String href;
9     private String rel;
10    private String metodo;
11    private String tipoDeMidia;
12@    public Link(String href, String rel, String metodo) {
13        this.href = href;
14        this.rel = rel;
15        this.metodo = metodo;
16    }
17    public Link() {}
18@XmlAttribute
19    public String getHref() { return href; }
20    public void setHref(String href) { this.href = href; }
21@XmlAttribute
22    public String getRel() { return rel; }
23    public void setRel(String rel) { this.rel = rel; }
24    public String getMetodo() { return metodo; }
25    public void setMetodo(String metodo) { this.metodo = metodo; }
26@Override
27    public int hashCode() {
28        final int prime = 31;
29        int result = 1;
30        result = prime * result + ((href == null) ? 0 : href.hashCode());
31        result = prime * result + ((rel == null) ? 0 : rel.hashCode());
32        return result;
33    }
34@Override
35    public boolean equals(Object obj) {
36        if (this == obj) return true;
37        if (obj == null) return false;
38        if (getClass() != obj.getClass()) return false;
39        Link other = (Link) obj;
40        if (href == null) {
41            if (other.href != null) return false;
42        } else if (!href.equals(other.href)) return false;
43        if (rel == null) {
44            if (other.rel != null) return false;
45        } else if (!rel.equals(other.rel)) return false;
46        return true;
47    }
48    public String getTipoDeMidia() { return tipoDeMidia; }
49    public void setTipoDeMidia(String tipoDeMidia) { this.tipoDeMidia = tipoDeMidia; }
50}

```

Figura 3.4: Classe Link parte 1.

```
1 package servicios;
2
3 import java.io.ByteArrayInputStream;
4 import java.io.IOException;
5
6 import org.apache.commons.httpclient.Header;
7 import org.apache.commons.httpclient.HttpClient;
8 import org.apache.commons.httpclient.HttpException;
9 import org.apache.commons.httpclient.methods.DeleteMethod;
10 import org.apache.commons.httpclient.methods.InputStreamRequestEntity;
11 import org.apache.commons.httpclient.methods.PostMethod;
12 import org.apache.commons.httpclient.methods.PutMethod;
13 import org.apache.commons.httpclient.methods.RequestEntity;
14
15 import com.sun.jersey.api.client.Client;
16 import com.sun.jersey.api.client.ClientResponse;
17 import com.sun.jersey.api.client.WebResource;
18
19
20 public class InterceptadorClienteRESTful {
```

Figura 3.5: Classe InterceptadorClienteRESTful - Declaração e importação de objetos.

3.2 Estudo de Caso

Neste capítulo será apresentado a arquitetura, modelagem e implementação do estudo de caso desenvolvido para ser utilizado em conjunto com o framework proposto neste trabalho. Primeiramente foi implementado dois Web Services REST na plataforma de desenvolvimento JavaEE, sendo que o primeiro é apenas um intermediador entre o cliente e o serviço web final. Já o segundo é o serviço web final, ou seja, onde realmente ocorre toda a persistência e as consultas na base de dados. O cliente realiza requisições ao serviço web intermediador, que por sua vez requisita e obtém as respostas no serviço web final. Após o serviço web intermediador receber as respostas do serviço web final, ele retorna as mesmas ao cliente. Além das implementações dos dois Web Services citados, também foi implementado uma aplicação cliente Web utilizando a linguagem de programação PHP e outra desktop desenvolvida na plataforma de desenvolvimento JavaSE.

3.2.1 Diagrama de Arquitetura

A figura 3.6 ilustra a arquitetura e o fluxo de dados do serviço web. Os clientes enviam requisições HTTP ao serviço web intermediador. Este serviço web intermediador fica responsável por transferir estas requisições ao serviço web final, onde ocorre as operações na base de dados de acordo com a instrução recebida e envia a resposta ao serviço web intermediador. Uma vez que o serviço web intermediador obtém as respostas, responde aos clientes com as mesmas.

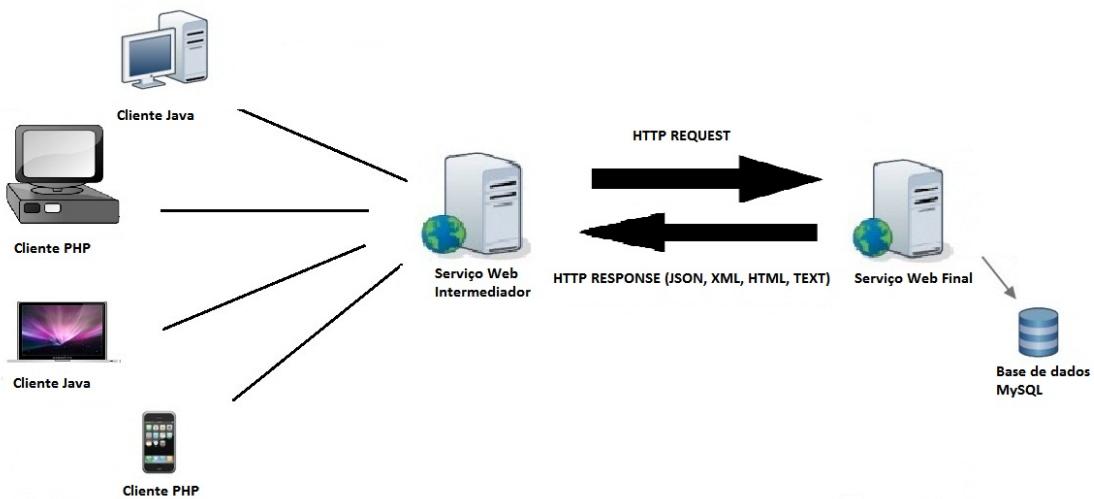


Figura 3.6: Diagrama da arquitetura e fluxo de dados do serviço web.

A figura 3.7 ilustra as operações possíveis aos clientes no serviço web. Através da imagem é possível observar que os clientes podem criar recursos, bem como obter, atualizar ou remover determinado recurso.

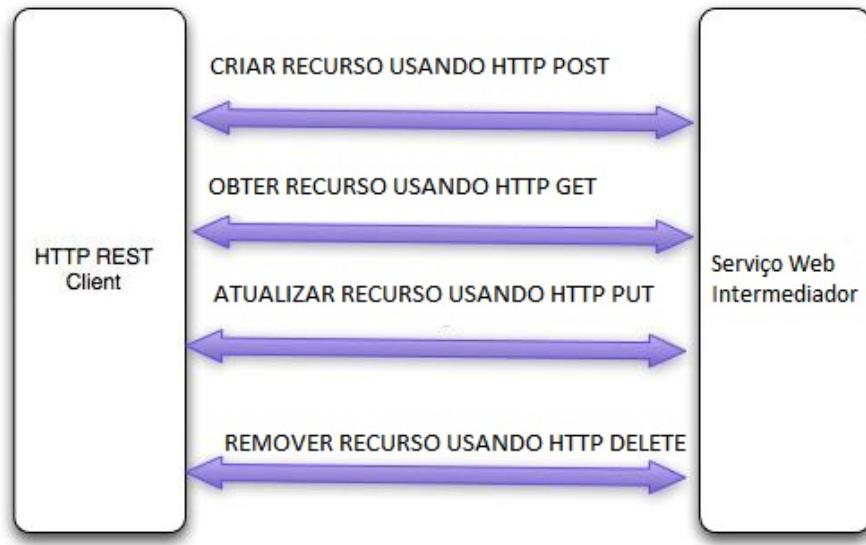


Figura 3.7: Operações para clientes no serviço web.

3.2.2 Diagrama de Caso de Uso

No diagrama de caso de uso da figura 3.8 é apresentado as operações que os usuários das aplicações cliente podem realizar no serviço web.

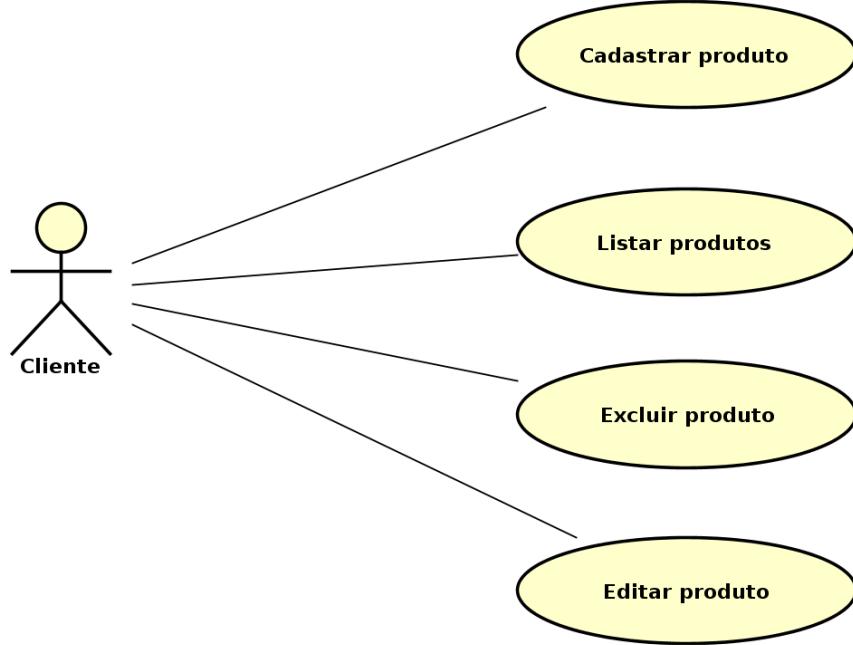


Figura 3.8: Diagrama de caso de uso referente ao estudo de caso.

3.2.3 Estrutura dos dados

A estrutura dos dados é ilustrado através da figura 3.9, onde mostra a estrutura da tabela produto, utilizada neste estudo de caso.

Produto	
!	id BIGINT(20)
◆	ativo BIT(1)
◆	dataDeAtualizacao DATETIME
◆	dataDeCriacao DATETIME
◆	nome VARCHAR(255)
◆	preco DOUBLE
Indexes	
PRIMARY	

Figura 3.9: Estrutura dos dados da tabela "Produto" para o estudo de caso proposto neste trabalho.

3.2.4 Diagrama de Classes

O diagrama de classes para o estudo de caso é apresentado através da figura 3.10. Através do diagrama é possível visualizar os relacionamentos entre as classes do Web Service e do framework proposto para este trabalho.

A classe *Produto*, referente à classe de entidade, modelo de domínio, implementa a classe de interface *HATEOASEntidade* do framework, que por sua vez utiliza uma lista de objetos da classe *Link* para então trabalhar com os controles de hipermídia. Neste contexto, a classe *Produto* implementa a interface *HATEOASEntidade* apenas para obrigar o desenvolvedor a trabalhar com a classe *Link* e implementar na própria classe *Produto* os métodos *getLinks*, para obter a lista de objetos da classe *Link*, *setLinks*, para definir uma coleção de objetos da classe *Link* e também o *addLink*, para adicionar um *Link*. Neste caso toda a lógica que deve ser aplicada a estes métodos fica a cargo do desenvolvedor.

Além de implementar a interface *HATEOASEntidade*, a classe *Produto* herda os atributos e métodos da classe abstrata *BaseDeEntidade*. Esta classe foi desenvolvida apenas para que todas as classes de entidade que a herdem tenham os atributos *id*, *dataDeCriacao*, *dataDeAtualizacao* e um booleano *ativo*, junto com seus respectivos métodos getters and setters.

Já a classe *ProdutoRecurso* utiliza um objeto da classe *Produto* quando implementa a *BaseDeServicoInterface*, que visa fornecedor a assinatura correta para implementação de uso do protocolo HTTP e seus verbos. A classe *ProdutoRecurso* também utiliza objeto da classe *InterceptadorClienteRESTful*, onde esta fica responsável pela intermediação entre

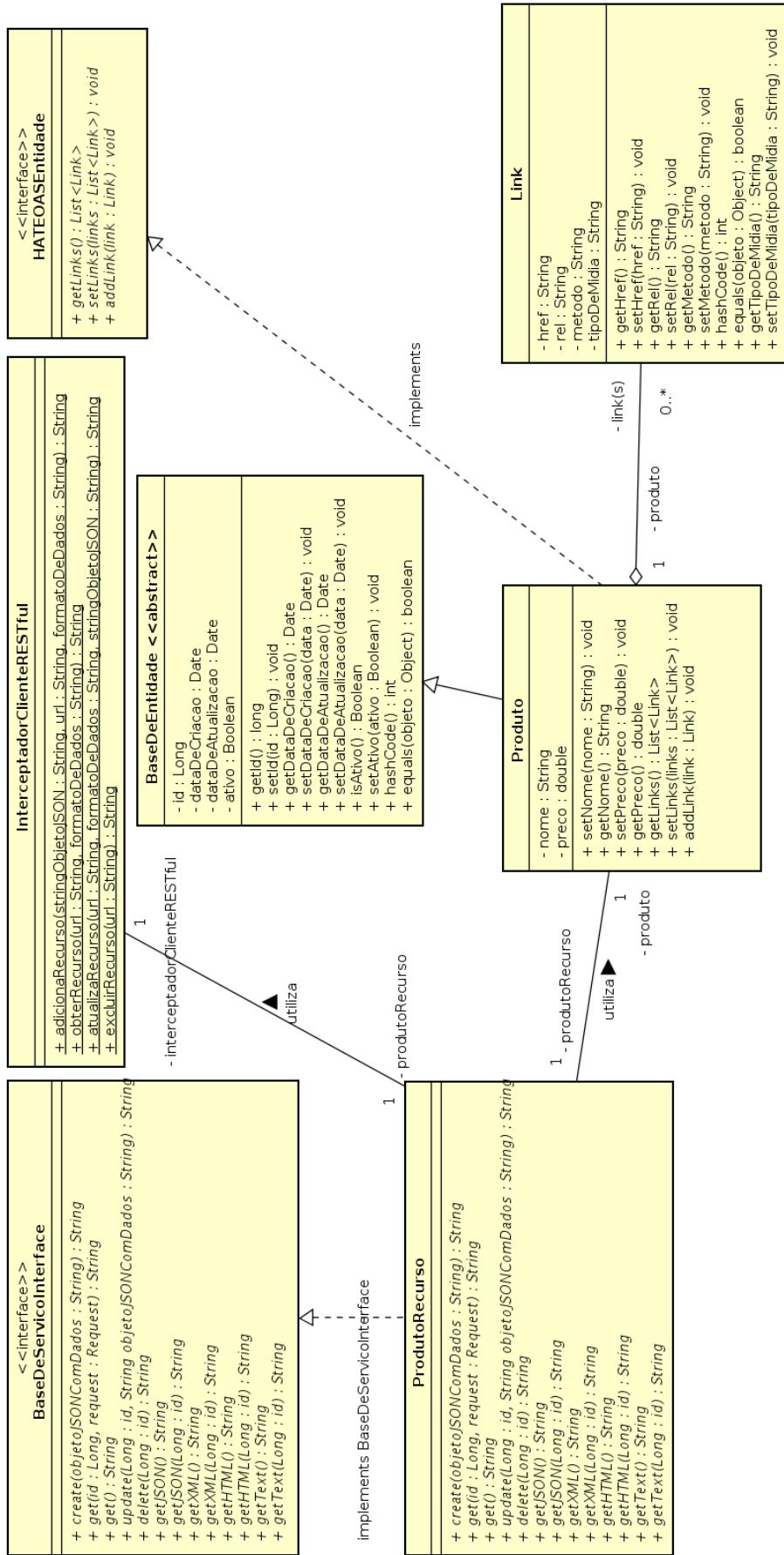


Figura 3.10: Diagrama de classes para o estudo de caso proposto.

o cliente e servidor, buscando os dados em um outro serviço web e retornando os mesmos ao cliente com respostas autoexplicativas, no mesmo formato de dados em que trabalha-se. As respostas autoexplicativas são retornadas em todos os métodos. Apenas no método get, quando encontrado resultados para a requisição, o retorno são os próprios dados, caso contrário também é retornado uma resposta autoexplicativa.

3.2.5 Implementação

Esta seção apresenta as estruturas e códigos-fonte referentes às implementações dos serviços web intermediador e final, ambos utilizando o framework proposto a este trabalho, além das aplicações cliente que consomem os recursos destes serviços. Primeiramente serão apresentados os serviços web implementados para o projeto, logo após, as aplicações cliente Java e PHP.

3.2.5.1 Serviço Web Intermediador

A figura 3.11 ilustra a estrutura do projeto java web dinâmico, criado na plataforma de desenvolvimento JavaEE, referente ao web serviço intermediador. Através da figura 3.12 são apresentadas as bibliotecas que foram utilizadas para a construção do serviço web intermediador. Entre as bibliotecas, estão basicamente arquivos das APIs Jersey e JAX-RS. Já a figura A.5, localizada no apêndice deste documento, apresenta a configuração do Web Service através do arquivo web.xml. Nesta configuração são definidos, por exemplo, a classe da API Jersey responsável por gerenciar as requisições, o pacote com as classes de recursos, além do mapeamento de requisições. Através das figuras 3.13 e 3.14 é apresentado a classe *ProdutoRecurso* do serviço web intermediador, o qual carrega a anotação @Path("/produtos"), o que a torna acessível por este caminho com base no endereço do servidor. Destaca-se nesta classe, a implementação da interface *BaseDeServiçoInterface* e a utilização de objetos da classe *InterceptadorClienteRESTful*, ambas do framework proposto para este trabalho. Esta classe *InterceptadorClienteRESTful*, como já mencionado, encarrega-se de buscar os dados requisitados pelo cliente em um outro serviço web e retorna estas respostas ao cliente, caracterizando um serviço de intermediação entre cliente e servidor (Layered System).

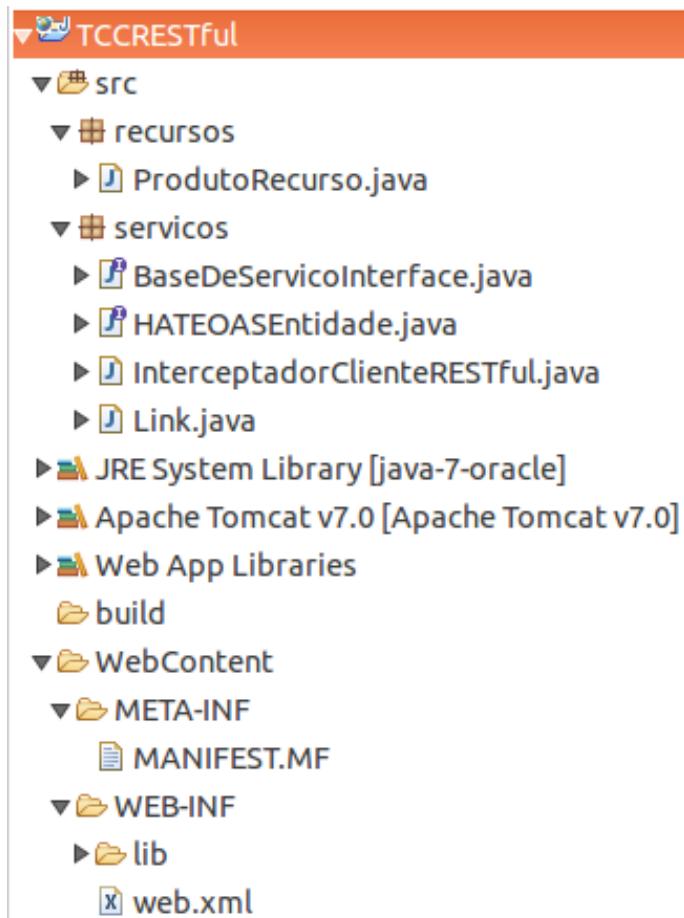


Figura 3.11: Estrutura do projeto Web Service Intermediador

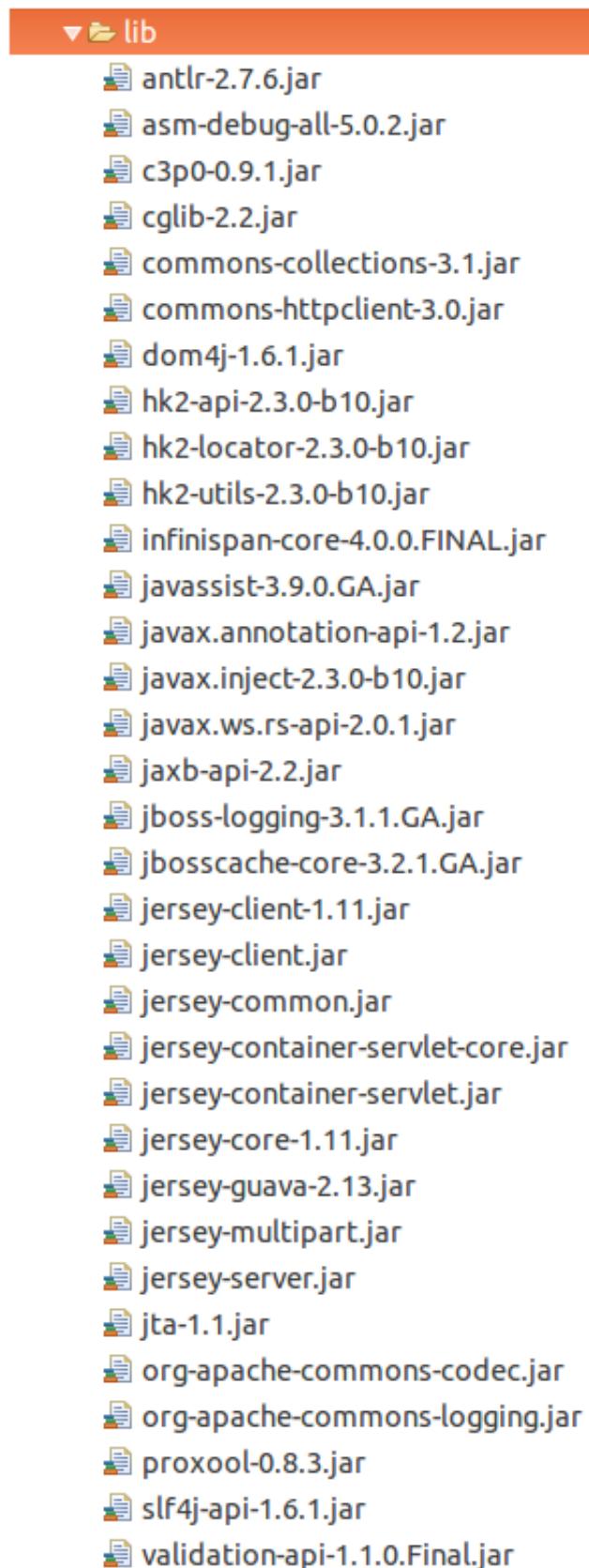


Figura 3.12: Bibliotecas utilizadas no projeto Web Service Intermediador.

```

1 package recursos;
2
3 import javax.ws.rs.Path;
4 import servicios.BaseDeServicioInterface;
5 import servicios.InterceptorClienteRESTful;
6
7 @Path("/produtos")
8 public class ProdutoRecurso implements BaseDeServicioInterface {
9
10    private InterceptorClienteRESTful interceptorClienteRESTful;
11
12    public ProdutoRecurso() {
13        super();
14    }
15
16    @Override
17    public String create(String objetoJSONComDados) {
18        try {
19            InterceptorClienteRESTful = new InterceptorClienteRESTful();
20            return interceptorClienteRESTful.adicionaRecurso(objetoJSONComDados,
21                    "http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/",
22                    "json");
23        } catch (Exception e) {
24            e.printStackTrace();
25            return "null";
26        }
27    }
28
29    @Override
30    public String get(Long id) {
31        InterceptorClienteRESTful = new InterceptorClienteRESTful();
32        return interceptorClienteRESTful.obterRecurso("http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/" + id, "json");
33    }
34
35    @Override
36    public String get() {
37        InterceptorClienteRESTful = new InterceptorClienteRESTful();
38        return interceptorClienteRESTful.obterRecurso("http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/", "json");
39    }
40
41    @Override
42    public String update(Long id, String objetoJSONComDados) {
43        try {
44            InterceptorClienteRESTful = new InterceptorClienteRESTful();
45            return interceptorClienteRESTful.atualizarRecurso(
46                    "http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/" + id,
47                    "json", objetoJSONComDados);
48        } catch (Exception e) {
49            e.printStackTrace();
50            return "null";
51        }
52    }

```

Figura 3.13: Classe Produto Recurso parte 01.

```

53
54@override
55 public String delete(Long id) {
56     try {
57         interceptadorClienteRESTful = new InterceptadorClienteRESTful();
58         return interceptadorClienteRESTful.excluirRecurso("http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/"+id+ "/");
59     } catch (Exception e) {
60         e.printStackTrace(); return "null";
61     }
62 }
63
64@override
65 public String getJSON() {
66     return get();
67 }
68@override
69 public String getJSON(Long id) {
70     return get(id);
71 }
72
73@override
74 public String getXML() {
75     interceptadorClienteRESTful = new InterceptadorClienteRESTful();
76     return interceptadorClienteRESTful.obterRecurso("http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/xml", "xml");
77 }
78@override
79 public String getXML(Long id) {
80     interceptadorClienteRESTful = new InterceptadorClienteRESTful();
81     return interceptadorClienteRESTful.obterRecurso("http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/xml/" + id, "xml");
82 }
83
84@override
85 public String getHTML() {
86     interceptadorClienteRESTful = new InterceptadorClienteRESTful();
87     return interceptadorClienteRESTful.obterRecurso("http://localhost:8080/TCCRESTfulImplementacao/rest/produtosI/html", "html");
88 }
89@override
90 public String getHTML(Long id) {
91     return "";
92 }
93
94@override
95 public String getText() {
96     return "";
97 }
98@override
99 public String getText(Long id) {
100    return "";
101 }
102
103 }

```

Figura 3.14: Classe Produto Recurso parte 02.

3.2.5.2 Serviço Web Final

A seguir será apresentado a implementação do web service final, serviço web responsável por todo negócio e persistência a dados.

As figuras 3.15 e 3.16 apresentam respectivamente a estrutura e bibliotecas utilizadas na implementação do serviço web final. Aqui destacam-se os serviços do Gson, biblioteca do Google escrita em Java para conversão objetos em representações json e vice-versa, API Jersey 2.0 e JAX-RS, Jettison, outra biblioteca para a manipulação de dados em JSON, utilizada no projeto para trabalhar com vetores de objetos, log4j para a geração e gravação de logs na aplicação, conector JDBC do MySQL, base de dados utilizada no projeto, além do Hibernate, framework para o mapeamento objeto-relacional escrito na linguagem Java que facilita o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação, mediante o uso de arquivos (XML) ou anotações Java. Na figura A.5 é apresentado a configuração do projeto web definida no arquivo web.xml, mesma configuração utilizada no projeto web service intermediador, e já a figura A.6 apresenta o arquivo persistence.xml, arquivo de configuração utilizado pelo Hibernate para conexão na base de dados. Ambas as figuras estão localizadas no apêndice deste documento.

Por utilizar o Hibernate para acesso a base de dados, foi criada uma classe abstrata no projeto, que contém um modelo padrão para consultas utilizando a tecnologia. A classe foi denominada *GenericDAO* e pode ser visualizada através das figuras 3.17 e 3.18. A classe *ProdutoDAO* herda os atributos e métodos da classe *GenericDAO* e pode ser visualizada através das figuras 3.19 e 3.20. Nesta classe foram implementados os métodos *getIdAnteriorRegistro* (Long id), *getIdProximoRegistro* (Long id), *getIdPrimeiroRegistro()* e *getIdUltimoRegistro()*, justamente para poder implementar a conectividade nos recursos que foram obtidos individualmente. A figura 3.21 apresenta a classe *ProdutoControlador*, responsável pela intermediação entre a camada de apresentação e persistência a dados. O modelo ou domínio da aplicação pode ser observado através da figura 3.22 e 3.23, respectivamente ilustrando as classes *BaseDeEntidade* e *Produto*. Por fim, a classe *produtoRecurso*, do projeto web service final, pode ser observada através das figuras A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.14, A.15, A.16, A.17 e A.18, todas localizadas no apêndice deste documento.

A seguir serão apresentados os detalhes de implementação da aplicação cliente java, que foi desenvolvida para consumir recursos dos serviços web.

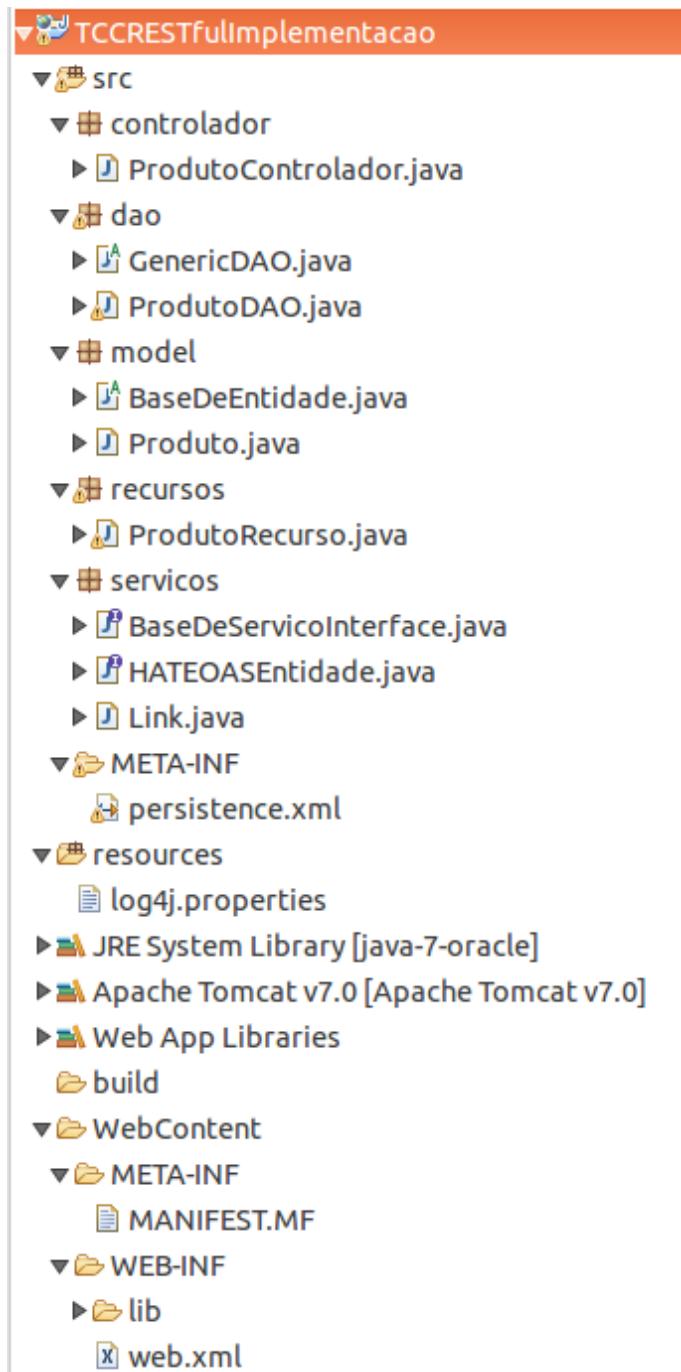


Figura 3.15: Estrutura do projeto Web Service Final.

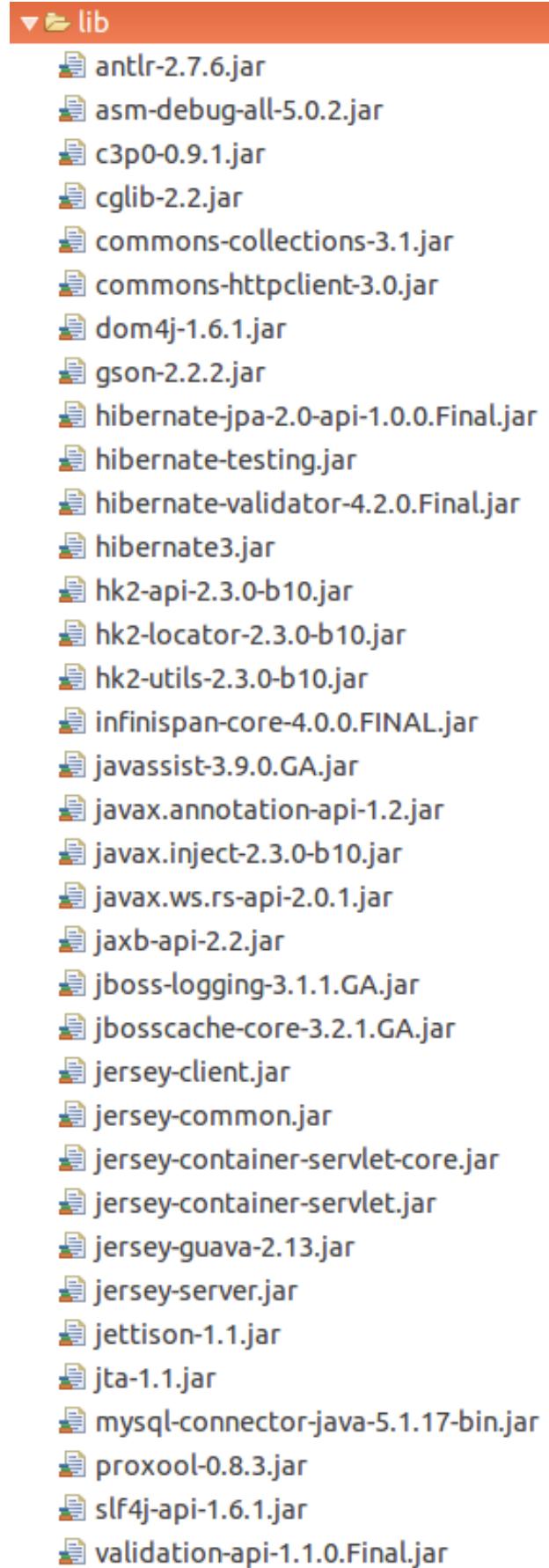


Figura 3.16: Bibliotecas utilizadas no projeto Web Service Final.

```

1@ package dao; import java.io.Serializable; import java.util.List; import java.util.Map; import java.util.Map.Entry;
2 import javax.persistence.EntityManager; import javax.persistence.EntityManagerFactory;
3 import javax.persistence.NoResultException; import javax.persistence.Persistence; import javax.persistence.Query;
4 import javax.persistence.criteria.CriteriaQuery;
5 public abstract class GenericDAO<T> implements Serializable {
6
7     private static final long serialVersionUID = 1L;
8     private static final EntityManagerFactory emf = Persistence.createEntityManagerFactory("RESTfulTCCPersistenceUnit");
9     private EntityManager em;
10    private Class<T> classeEntidade;
11@   public void beginTransaction() {
12        setEm(emf.createEntityManager());
13        getEm().getTransaction().begin();
14    }
15    public void commit() { getEm().getTransaction().commit(); }
16    public void rollback() { getEm().getTransaction().rollback(); }
17
18    public void closeTransaction() { getEm().close(); }
19
20    public void commitAndCloseTransaction() { commit(); closeTransaction(); }
21
22    public void flush() { getEm().flush(); }
23
24@   public void joinTransaction() {
25        setEm(emf.createEntityManager());
26        getEm().joinTransaction();
27    }
28
29    public GenericDAO() { }
30    public GenericDAO(Class<T> classeEntidade) { this.setClasseEntidade(classeEntidade); }
31
32    public void save(T classeEntidade) { getEm().persist(classeEntidade); }
33
34@   public void delete(Class<T> classeEntidade, Long id) {
35        T entidadeParaRemover = getEm().find(classeEntidade, id);
36        getEm().remove(entidadeParaRemover);
37    }
38
39    public T update(T classeEntidade) { return getEm().merge(classeEntidade); }
40
41    public T find(int idEntidade) { return getEm().find(getClasseEntidade(), idEntidade); }
42
43    public T findReferenceOnly(int idEntidade) { return getEm().getReference(getClasseEntidade(), idEntidade); }
44
45    // Using the unchecked because JPA does not have a
46    // em.getCriteriaBuilder().createQuery()<T> method
47@ SuppressWarnings({ "unchecked", "rawtypes" })
48    public List<T> findAll() {
49        CriteriaQuery cq = getEm().getCriteriaBuilder().createQuery();
50        cq.select(cq.from(getClasseEntidade()));
51        return getEm().createQuery(cq).getResultList();
52    }

```

Figura 3.17: Classe GenericDAO parte 01.

```

53
54
55
56
57 // Using the unchecked because JPA does not have a
58 // query.getSingleResult()<T> method
59@ SuppressWarnings("unchecked")
60 protected T findOneResult(String namedQuery, Map<String, Object> parametros) {
61     T resultado = null;
62     try {
63         Query query = getEm().createNamedQuery(namedQuery);
64         // Method that will populate parameters if they are passed not null and empty
65         if (parametros != null && !parametros.isEmpty()) {
66             populateQueryParameters(query, parametros);
67         }
68         resultado = (T) query.getSingleResult();
69
70     } catch (NoResultException e) {
71         System.out.println("No result found for named query: " + namedQuery);
72     } catch (Exception e) {
73         System.out.println("Error while running query: " + e.getMessage());
74         e.printStackTrace();
75     }
76
77     return resultado;
78 }
79
80 private void populateQueryParameters(Query query, Map<String, Object> parameters) {
81     for (Entry<String, Object> entry : parameters.entrySet()) {
82         query.setParameter(entry.getKey(), entry.getValue());
83     }
84 }
85
86 public Class<T> getClasseEntidade() { return classeEntidade; }
87 //não verifica a conversão. o cast. Por isso usa-se @SuppressWarnings("unchecked")
88@ SuppressWarnings("unchecked")
89 public void setClasseEntidade(Object classeEntidade) {
90     this.classeEntidade = (Class<T>) classeEntidade;
91 }
92
93 public void setClasseEntidade(Class<T> classeEntidade) {
94     this.classeEntidade = classeEntidade;
95 }
96
97 public EntityManager getEm() {
98     return em;
99 }
100
101 public void setEm(EntityManager em) {
102     this.em = em;
103 }
104 }

```

Figura 3.18: Classe GenericDAO parte 02.

```

1 package dao;
2 import java.util.List; import javax.persistence.EntityManager; import javax.persistence.EntityManagerFactory;
3 import javax.persistence.Persistence; NoResultException; import javax.persistence.Persistence; import javax.persistence.Query;
4 import javax.persistence.criteria.CriteriaQuery;
5 import model.Produto;
6 public class ProdutoDAO extends GenericDAO<Produto> {
7     public ProdutoDAO() { super(); }
8     private static final long serialVersionUID = 1L;
9     public Produto find(Long id) {
10         EntityManagerFactory factory = Persistence.createEntityManagerFactory("RESTfulTCCPersistenceUnit");
11         EntityManager entityManager = factory.createEntityManager();
12         Produto produto1 = entityManager.find(Produto.class, id);
13         factory.close();
14         entityManager.close();
15         return produto1;
16     }
17     public Long getAnteriorRegistro(Long id) {
18         EntityManagerFactory factory = Persistence.createEntityManagerFactory("RESTfulTCCPersistenceUnit");
19         EntityManager entityManager = factory.createEntityManager();
20         Query query = entityManager.createQuery("SELECT p FROM Produto p WHERE p.id < :id ORDER BY id DESC ").setMaxResults(1);
21         query.setParameter("id", id);
22         try {
23             Produto p = (Produto) query.getSingleResult();
24             factory.close();
25             entityManager.close();
26             return p.getId();
27         } catch (NoResultException e) {
28             factory.close();
29             entityManager.close();
30             e.printStackTrace();
31         }
32         return null;
33     }
34     public Long getProximoRegistro(Long id) {
35         EntityManagerFactory factory = Persistence.createEntityManagerFactory("RESTfulTCCPersistenceUnit");
36         EntityManager entityManager = factory.createEntityManager();
37         Query query = entityManager.createQuery("SELECT p FROM Produto p WHERE p.id > :id ORDER BY id ASC").setMaxResults(1);
38         query.setParameter("id", id);
39         try {
40             Produto p = (Produto) query.getSingleResult();
41             factory.close();
42             entityManager.close();
43             return p.getId();
44         } catch (NoResultException e) {
45             factory.close();
46             entityManager.close();
47             e.printStackTrace();
48         }
49         return null;
50     }
51

```

Figura 3.19: Classe ProdutoDAO parte 01.

```

52 @
53     EntityManagerFactory factory = Persistence.createEntityManagerFactory("RESTfulTCCPersistenceUnit");
54     EntityManager entityManager = factory.createEntityManager();
55     Query query = entityManager.createQuery("SELECT p FROM Produto p ORDER BY id ASC").setMaxResults(1);
56
57     try {
58         Produto p = (Produto) query.getSingleResult();
59         entityManager.close();
60         factory.close();
61         return p.getId();
62     } catch (NoResultException e) {
63         factory.close();
64         entityManager.close();
65         e.printStackTrace();
66     }
67     return null;
68 }
69     public Long getIdPrimoRegistro() {
70         EntityManagerFactory factory = Persistence.createEntityManagerFactory("RESTfulTCCPersistenceUnit");
71         EntityManager entityManager = factory.createEntityManager();
72         Query query = entityManager.createQuery("SELECT p FROM Produto p ORDER BY id DESC").setMaxResults(1);
73
74         try {
75             Produto p = (Produto) query.getSingleResult();
76             entityManager.close();
77             return p.getId();
78         } catch (NoResultException e) {
79             factory.close();
80             entityManager.close();
81             e.printStackTrace();
82         }
83     }
84
85     public List<Produto> findAll() {
86         beginTransaction();
87         CriteriaQuery cq = getEm().getCriteriaBuilder().createQuery();
88         cq.select(cq.from(Produto.class));
89         List<Produto> produtosRetorno = getEm().createQuery(cq).getResultList();
90         commit();
91         closeTransaction();
92         return produtosRetorno;
93     }
94
95 }

```

Figura 3.20: Classe ProdutoDAO parte 02.

```

1 package controlador;
2
3 import java.util.List;
4
5 import dao.ProdutoDAO;
6 import model.Produto;
7
8 public class ProdutoControlador {
9
10    ProdutoDAO produtoDAO = null;
11
12    public ProdutoControlador() { this.produtoDAO = new ProdutoDAO(); }
13
14    public void save(Produto classeEntidade) { produtoDAO.save(classeEntidade); }
15
16    public Produto find(Long id) { return produtoDAO.find(id); }
17
18    public List<Produto> findAll() { return produtoDAO.findAll(); }
19
20    public void delete(Class<Produto> classeEntidade, Long id) { produtoDAO.delete(classeEntidade, id); }
21
22    public Produto update(Produto classeEntidade) { return produtoDAO.update(classeEntidade); }
23
24    public void beginTransaction() { produtoDAO.beginTransaction(); }
25
26    public void commit() { produtoDAO.commit(); }
27
28    public void rollback() { produtoDAO.rollback(); }
29
30    public void closeTransaction() { produtoDAO.closeTransaction(); }
31
32    public void commitAndCloseTransaction() { produtoDAO.commitAndCloseTransaction(); }
33
34    public void flush() { produtoDAO.flush(); }
35
36    public void joinTransaction() { produtoDAO.joinTransaction(); }
37
38    public Produto findReferenceOnly(int idEntidade) { return findReferenceOnly(idEntidade); }
39
40    public Long getIdAnteriorRegistro(Long id) { return produtoDAO.getIdAnteriorRegistro(id); }
41
42    public Long getIdProximoRegistro(Long id) { return produtoDAO.getIdProximoRegistro(id); }
43
44    public Long getIdPrimeiroRegistro() { return produtoDAO.getIdPrimeiroRegistro(); }
45
46    public Long getIdUltimoRegistro() { return produtoDAO.getIdUltimoRegistro(); }
47
48}
49
50

```

Figura 3.21: Classe ProdutoControlador.

```

1 package model;
2 import java.util.Date;
3 import javax.persistence.Column;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7 import javax.persistence.Inheritance;
8 import javax.persistence.InheritanceType;
9 import javax.persistence.MappedSuperclass;
10 import javax.persistence.Temporal;
11 import javax.persistence.TemporalType;
12 @MappedSuperclass
13 @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
14 public abstract class BaseDeEntidade {
15     @Id
16     @GeneratedValue(strategy=GenerationType.IDENTITY)
17     private Long id;
18     @Temporal(TemporalType.TIMESTAMP)
19     @Column(updatable=false)
20     private Date dataDeCriacao = new Date();
21     @Temporal(TemporalType.TIMESTAMP)
22     private Date dataDeAtualizacao = new Date();
23     private Boolean ativo = Boolean.TRUE;
24     public Long getId() { return id; }
25     public void setId(Long id) { this.id = id; }
26     public Date getDataDeCriacao() { return dataDeCriacao; }
27     public void setDataDeCriacao(Date dataDeCriacao) { this.dataDeCriacao = dataDeCriacao; }
28     public Date getDataDeAtualizacao() { return dataDeAtualizacao; }
29     public void setDataDeAtualizacao(Date data) { this.dataDeAtualizacao = data; }
30     public Boolean isAtivo() { return ativo; }
31     public void setAtivo(Boolean ativo) { this.ativo = ativo; }
32     @Override
33     public int hashCode() {
34         final int prime = 31;
35         int result = 1;
36         result = prime * result + ((id == null) ? super.hashCode() : id.hashCode());
37         return result;
38     }
39     @Override
40     public boolean equals(Object obj) {
41         if (this == obj) return true;
42         if (obj == null) return false;
43         if (getClass() != obj.getClass()) return false;
44         BaseDeEntidade other = (BaseDeEntidade) obj;
45         if (id == null) {
46             if (other.id != null) return false;
47             else return super.equals(obj);
48         } else if (!id.equals(other.id)) return false;
49         return true;
50     }
51 }
52 }
```

Figura 3.22: Classe BaseDeEntidade.

```

1 package model;
2 import java.util.ArrayList;
3 import java.util.List;
4 import javax.persistence.Column;
5 import javax.persistence.Entity;
6 import javax.persistence.Transient;
7 import servicos.HATEOASEntidade;
8 import servicos.Link;
9
10 @Entity
11 public class Produto extends BaseDeEntidade implements HATEOASEntidade {
12
13     public Produto(Long id, String nome, double preco) {
14         super.setId(id);
15         this.nome = nome;
16         this.preco = preco;
17     }
18     @Column
19     private String nome;
20
21     @Column
22     private double preco;
23
24     @Transient
25     private List<Link> links = new ArrayList<>();
26
27     public Produto() {}
28
29     public String getNome() { return nome; }
30
31     public void setNome(String nome) { this.nome = nome; }
32
33     public double getPreco() { return preco; }
34
35     public void setPreco(double preco) { this.preco = preco; }
36
37     @Override
38     public List<Link> getLinks() { return links; }
39
40     @Override
41     public void setLinks(List<Link> links) { for(int a = 0; a < links.size(); a++) addLink(links.get(a)); }
42
43     @Override
44     public void addLink(Link link) {
45         links.add(link);
46     }
47 }
48 }
```

Figura 3.23: Classe Produto (serviço web intermediador).

3.2.5.3 Aplcação Cliente Java

Para a implementação da aplicação cliente java, foram replicadas do web service final, as classes de modelo denominadas Produto e BaseDeEntidade. Este projeto utiliza objetos da classe *InterceptadorClienteRESTful* do framework proposto para consumir os recursos do serviço web, já que trata-se de um cliente desenvolvido na mesma tecnologia . Portanto, nesta seção, será mostrado apenas uma imagem referente às respostas de requisições para os métodos POST, GET, PUT e DELETE, visto que as requisições ao serviço web são realizadas através dos métodos apresentados na classe *InterceptadorClienteRESTful*. A figura 3.24 apresenta as respostas para requisições POST, PUT e DELETE, respectivamente, no formato JSON.

```

Retorno do método POST:
{"codigoResposta": '200',"headersRespostaCabecalhos": [{"header": "Server: Apache-Coyote/1.1
"}, {"header": "Content-Type: text/plain
"}, {"header": "Content-Length: 220
"}, {"header": "Date: Thu, 26 Nov 2015 22:36:52 GMT
"}]}

Retorno do método PUT (ID = 21):
{"codigoResposta": '200',"headersRespostaCabecalhos": [{"header": "Server: Apache-Coyote/1.1
"}, {"header": "Cache-Control: private
"}, {"header": "Expires: Wed, 31 Dec 1969 21:00:00 BRT
"}, {"header": "Content-Type: text/plain
"}, {"header": "Content-Length: 312
"}, {"header": "Date: Thu, 26 Nov 2015 22:52:59 GMT
"}]}

Retorno do método DELETE(ID = 229):
{"codigoResposta": '200',"headersRespostaCabecalhos": [{"header": "Server: Apache-Coyote/1.1
"}, {"header": "Cache-Control: private
"}, {"header": "Expires: Wed, 31 Dec 1969 21:00:00 BRT
"}, {"header": "Content-Type: text/plain
"}, {"header": "Content-Length: 312
"}, {"header": "Date: Thu, 26 Nov 2015 22:47:39 GMT
"}]}

```

Figura 3.24: Respostas de requisições POST, PUT e DELETE, informando JSON para cadastro de produto(s) no método POST.

Na próxima seção serão apresentadas algumas telas de interface da aplicação cliente PHP e também alguns trechos de código-fonte da implementação.

3.2.5.4 Aplicaçao Cliente PHP

Esta seção apresenta figuras que ilustram algumas telas da aplicação e alguns trechos de código-fonte referentes a implementação na linguagem de programação de PHP. A figura 3.25 ilustra a página inicial da aplicação. Já a figura 3.26 apresenta as listas de dados da aplicação, sendo que estes dados foram consumidos em formatos JSON, XML e HTML, e depois tratados pra exibiçõ em HTML, já que trata-se de uma aplicação Web e a base é o navegador. Através da figura 3.27 são apresentados os formulários para cadastro e edição de produtos, sendo que no formulário de cadastro é possível informar uma lista de objetos produto no formato json para cadastro de vários itens. Já no formulário de edição é possível navegar entre os recursos com a implementação de conectividade entre os mesmos. É possível avançar a um próximo ou retroceder a um registro anterior.

Jersey

RESTful Web Services in Java.

Home Produto ▾

Novo
Lista JSON
Lista XML
Lista HTML

The diagram illustrates a RESTful Web Service architecture. It shows a client (represented by a computer monitor icon) sending an "HTTP request" (labeled "XML/JSON") over the "Internet" to a "Web Server". The Web Server is connected to a "database" (represented by a cylinder icon). The Web Server returns a response to the client, which is shown as a "Smartphone".

Rest WebService

Gabriel Almeida
Tecnologia em Análise e Desenvolvimento de Sistemas
Universidade Federal do Rio Grande

Figura 3.25: Cliente PHP - página inicial.

Lista de Produtos JSON						
ID	Nome	Preço	Editar	Excluir	Imagem	
21	GM VECTRA GT 2008/2009	29800				
223	Hyundai Azera 2008 PRETO	38900				
232	Toyota Corolla 2012 Altis Branco	58900				
237	CBR 600 RR 2008	33000				
238	CBR 1000 RR Fireblade 2007 Preta	33900				

Lista de Produtos XML						
ID	Nome	Preço	Editar	Excluir	Imagem	
21	GM VECTRA GT 2008/2009	29800.0				
223	Hyundai Azera 2008 PRETO	38900.0				
232	Toyota Corolla 2012 Altis Branco	58900.0				
237	CBR 600 RR 2008	33000.0				
238	CBR 1000 RR Fireblade 2007 Preta	33900.0				

Lista de Produtos HTML						
ID	Nome	Preço	Imagen 1	Imagen 2		
21	GM VECTRA GT 2008/2009	29800.0				
223	Hyundai Azera 2008 PRETO	38900.0				
232	Toyota Corolla 2012 Altis Branco	58900.0				
237	CBR 600 RR 2008	33000.0				
238	CBR 1000 RR Fireblade 2007 Preta	33900.0				

Figura 3.26: Lista de produtos JSON, XML e HTML.

Você está em: Página Inicial > Produto >Novo

Novo Produto

Nome:

Preço:

JSON Produtos:

```
[{"nome": "Produto 01", "preco": "599"}, {"nome": "Produto02", "preco": "200"}]
```

Você está em: Página Inicial > Produto >Editar

Editar Produto

Nome:

Preço:

Figura 3.27: Formulário Produto.

As implementações de códigos-fonte estão ilustradas através das figuras 3.28, 3.29, 3.30, 3.31 e 3.32. Os trechos de código-fonte referem-se a requisições HTTP correspondentes aos métodos POST, GET PUT e DELETE executadas em PHP. A figura 3.28 mostra uma requisição GET para retorno de recurso no formato JSON e logo após os dados sendo tratados para serem inseridos na página HTML que será exibida no navegador. Na figura 3.29 é apresentado uma requisição GET para retorno de recurso no formato de dados XML e após os dados também sendo tratados para serem exibidos junto ao HTML da página. Já a figura 3.30 apresenta uma requisição GET para retorno em HTML e também ilustra uma requisição DELETE, sendo a última a fim de remover algum produto na base de dados do servidor. A figura 3.31 apresenta duas requisições POST, sendo que na primeira é enviado apenas um objeto produto no formato JSON e na segunda, uma coleção ou lista de objetos para cadastro em remessa. Por fim a figura 3.32 apresenta a requisição PUT, utilizada para alterar dado(s) de produto(s).

```
<?php
$url = file_get_contents("http://localhost:8080/TCCRESTful/rest/produtos/json");
echo '<div class="row">';
echo '<div class="col-xs-1">ID</div>';
echo '<div class="col-xs-5">Nome</div>';
echo '<div class="col-xs-3">Preço</div>';
echo '<div class="col-xs-1">Editar</div>';
echo '<div class="col-xs-1">Excluir</div>';
echo '<div class="col-xs-1">Imagem</div>';
echo '</div>';

$arrayJSON = json_decode($url, true);
foreach ($arrayJSON as $i => $produto) {
    echo '<div class="row">';
    echo '<div class="col-xs-1">' . $produto["id"] . '</div>';
    echo '<div class="col-xs-5">' . $produto["nome"] . '</div>';
    echo '<div class="col-xs-3">' . $produto["preco"] . '</div>';
    $arrayLinks = $produto["links"];
}

$achouPUT = 0;
foreach ($arrayLinks as $linkProduto) {
    if (!empty($linkProduto["metodo"])) {
        if ($linkProduto["metodo"] == "PUT") {
            echo '<div class="col-xs-1">'.
                '<a href="EditarProduto.php?id=' . $produto["id"] . '&lista=json">' .
                '</a></div>';
            $achouPUT++;
        }
    }
}
if (!$achouPUT) echo '<div class="col-xs-1"></div>';

$achouDELETE = 0;
foreach ($arrayLinks as $linkProduto) {
    if (!empty($linkProduto["metodo"])) {
        if ($linkProduto["metodo"] == "DELETE") {
            echo '<div class="col-xs-1"><a href="ExcluirProduto.php?id=' . $produto["id"] . '>' .
                '</a></div>';
            $achouDELETE++;
        }
    }
}
if (!$achouDELETE) echo '<div class="col-xs-1"></div>';

$achouImage = 0;
foreach ($arrayLinks as $linkProduto) {
    if (!empty($linkProduto["tipodeMidia"])) {
        if ($linkProduto["tipodeMidia"] == "image") {
            if ($achouImage) echo '<div class="col-xs-1">' .
                '<a target=_blank href=' . $linkProduto["href"] . '>' .
                '&nbsp;</a>' ;
            $achouImage++;
        }
    }
}
if ($achouImage) echo '</div>';
else if (!$achouImage) echo '<div class="col-xs-1"></div>';
echo '</div>';

?>
```

Figura 3.28: Requisição GET com PHP para resposta em formato JSON.

```

<?php
$urlXML = file_get_contents("http://localhost:8080/TCCRESTful/rest/produtos/xml");
$xml = simplexml_load_string($urlXML);
echo '<div class="row">';
echo '  <div class="col-xs-1">ID</div>';
echo '  <div class="col-xs-5">Nome</div>';
echo '  <div class="col-xs-3">Preco</div>';
echo '  <div class="col-xs-1">Editor</div>';
echo '  <div class="col-xs-1">Excluir</div>';
echo '  <div class="col-xs-1">Imagen</div>';
echo '</div>';
$qtdProdutos = count($xml->produto);
for($a = 0; $a < $qtdProdutos; $a++) {
    echo '<div class="row">';
    echo '  <div class="col-xs-1">' . $xml->produto[$a]->id . '</div>';
    echo '  <div class="col-xs-5">' . $xml->produto[$a]->nome . '</div>';
    echo '  <div class="col-xs-3">' . $xml->produto[$a]->preco . '</div>';
    $achouPUT = 0;
    $qtdLinks = count($xml->produto[$a]->linksHATEOAS->Link);
    for($b = 0; $b < $qtdLinks; $b++) {
        if(empty($xml->produto[$b]->link[$b]->metodo))
            if($xml->produto[$a]->link[$b]->metodo == "PUT") {
                echo '<div class="col-xs-1"><a href=' . EditarProduto.php?id=' . $xml->produto[$a]->id . '"></a></div>';
                $achouPUT++;
            }
        if(!$achouPUT) echo '<div class="col-xs-1"></div>';
    }
    if(!$achouDELETE) echo '<div class="col-xs-1"></div>';
    $achouDELETE = 0;
    for($b = 0; $b < $qtdLinks; $b++) {
        if(empty($xml->produto[$a]->link[$b]->metodo))
            if($xml->produto[$a]->link[$b]->metodo == "DELETE") {
                echo '<div class="col-xs-1"><a href=' . ExcluirProduto.php?id=' . $xml->produto[$a]->id . '"></a></div>';
                $achouDELETE++;
            }
        if(!$achouImage) echo '<div class="col-xs-1"></div>';
    }
    if(!$achouImage) echo '<div class="col-xs-1"></div>';
    echo '</div>';
}
?>

```

Figura 3.29: Requisição GET com PHP para resposta em formato XML.

```

<?php

$url = file_get_contents("http://localhost:8080/TCCRESTful/rest/produtos/html");
echo $url;

?>
<?php
if(isset($_GET["id"])) {
    $id = $_GET["id"];
    $result = file_get_contents(
        'http://localhost:8080/TCCRESTful/rest/produtos/'.$id,
        false,
        stream_context_create(array(
            'http' => array(
                'method' => 'DELETE'
            )
        )));
}

$resultAux = $result;
$resultAux = str_replace('\'', '\"', $result);
$resultAux2 = str_replace('\"', '\'', $resultAux);
$resultAux3 = str_replace('\'' , '\"', $resultAux2);

$arrayJSONResposta = json_decode($resultAux3, true);
$codigoDeResposta = $arrayJSONResposta["codigoResposta"];
$arrayHeadersRespostaCabecalhos = $arrayJSONResposta["headersRespostaCabecalhos"];
$stringRetornoAlert = '{"codigoResposta": "'.$codigoDeResposta.'", "headersRespostaCabecalhos":[';
for($a = 0; (!empty($arrayHeadersRespostaCabecalhos[$a])); $a++) {
    $stringRetornoAlert .= ($a == 0) ?
        '{"header": "'.$arrayHeadersRespostaCabecalhos[$a]["header"].'"'
        :
        ', {"header": "'.$arrayHeadersRespostaCabecalhos[$a]["header"].'"';
}
$stringRetornoAlert .= ']}';
echo $stringRetornoAlert;
echo "<br/><br/><br/><a href='ProdutoListaJSON.php'>Lista de produtos JSON</a>";
die();
}
?>

```

Figura 3.30: Requisições GET para resposta em formato HTML e DELETE para exclusão de registro. Ambos têm respostas no formato JSON.

```

<?php
if(isset($_POST["nomeProduto"])) and (isset($_POST["precoProduto"])) {
    $nome = $_POST["nomeProduto"];
    $preco = $_POST["precoProduto"];
    $data = array("name" => $nome, "preco" => $preco);
    $data_string = json_encode($data);
    $data_string = "[" . $data_string . "]";
}

$result = file_get_contents('http://localhost:8080/TCCRESTful/rest/produtos/' , null,
    stream_context_create(array(
        'http' => array(
            'method' => 'POST',
            'header' => array('Content-Type: application/json' . "\r\n",
                'Authorization: username:key' . "\r\n",
                'Content-Length: ' . strlen($data_string) . "\r\n"
            ),
            'content' => $data_string
        )
    )));
}

$jsonResposta = json_encode($result);
$codigoDeResposta = explode("\n", stripslashes($jsonResposta));
$codigoDeResposta = explode(":", $codigoDeResposta[0]);
$codigoDeResposta = str_replace("", "", $codigoDeResposta[1]);
if($codigoDeResposta == 200) echo "<script>alert('Requisição ao servidor enviada com sucesso.');" . </script>";

}

if(isset($_POST["jsonProdutosTextArea"])){
    MODELO DE PREENCHIMENTO NO TEXT-AREA
    [
        {"name": "Hyundai i30 2.0 2011", "preco": "45000"}, {"name": "XR 250 Tornado 2008 Laranja", "preco": "7890"} ]
    $content = $_POST["jsonProdutosTextArea"];
    $result = "";
    $result = file_get_contents('http://localhost:8080/TCCRESTful/rest/produtos/' , null,
        stream_context_create(array(
            'http' => array(
                'method' => 'POST',
                'header' => array('Content-Type: application/json' . "\r\n",
                    'Authorization: username:key' . "\r\n",
                    'Content-Length: ' . strlen($content) . "\r\n"
                ),
                'content' => $content
            )
        )));
}
// ?>
}

```

Figura 3.31: Requisição POST com PHP. Resposta autoexplicativa em formato JSON.

```

<?php

if( (isset($_POST["nomeProduto"])) and (isset($_POST["precoProduto"])) ) {
    $nomeProduto = $_POST["nomeProduto"];
    $precoProduto = $_POST["precoProduto"];
    $idProduto = $_POST["idProduto"];

    $data = array("name" => "".$nomeProduto.", "preco" => "".$precoProduto."");
    $data_string = json_encode($data);
    $data_string = $data_string;

    $result = file_get_contents('http://localhost:8080/TCCRESTful/rest/produtos/'.$idProduto, null,
        stream_context_create(array(
            'http' => array(
                'method' => 'PUT',
                'header' => array('Content-Type: application/json'."\r\n"
                    . 'Authorization: username:key'."\r\n"
                    . 'Content-Length: ' . strlen($data_string) . "\r\n"
                ),
                'content' => $data_string
            )
        )));
}

echo $result;

$formato = !empty(isset($_GET["lista"])) ? $_GET["lista"] : "json";
if(($formato == "XML") || ($formato == "xml")) $formato = "xml";
else $formato = "json";
if($formato == "json") echo "<br/><br/><br/><a href='ProdutoListaJSON.php'>Lista de produtos JSON</a>";
else echo "<br/><br/><br/><a href='ProdutoListaXML.php'>Lista de produtos XML</a>";
die();
}

?>

```

Figura 3.32: Requisição PUT para alteração de dados de registro. Resposta autoexplicativa em formato JSON.

4 ANÁLISE

Atualmente a arquitetura REST e seus paradigmas estão sendo bastante adotados nas implementações de sistemas distribuídos. As implementações REST são realizadas através de Web Services, que geralmente são disponibilizadas na forma de frameworks ou Web APIs, cujo principal objetivo é a troca de dados entre aplicações, através da Web. No entanto, devido à ausência de padrões e diretrizes para o desenvolvimento, cada implementação REST segue uma linha de desenvolvimento, sendo que muitas acabam desconsiderando os princípios e paradigmas da arquitetura, o que resulta na dificuldade de construção de aplicações cliente e do desenvolvimento e publicação de novos recursos que seguem estritamente os padrões da arquitetura. Outro desafio para os desenvolvedores nas implementações é a falta de suporte ao uso de controle de hipermídia em representações de recursos, principalmente os que utilizam o formato JSON. Os controles de hipermídia podem assumir a forma de links, que guiam a navegação entre diferentes recursos. Diante deste cenário, este trabalho destinou-se a modelagem e implementação de um framework RESTful para a construção de serviços web RESTful nível três de maturidade, na plataforma de desenvolvimento Java.

A estrutura do framework consiste de um conjunto de classes (concretas, interfaces e abstratas), explicitamente projetado para ser usado em conjunto com o Jersey 2.0, implementação de referência da JAX-RS (JSR 311), especificação do Java para serviços Web REST. Portanto o framework é uma extensão da biblioteca Jersey 2.0, que visa fornecer corretamente todos os serviços da arquitetura REST. Em relação a maturidade na arquitetura do framework, foi implementado o nível três, já que é o nível mais alto do paradigma REST e também porque as outras ferramentas (Web APIs e frameworks) abordadas no capítulo 2 não disponibilizam uma arquitetura para a construção de serviços RESTful neste nível de maturidade. O modelo de maturidade utilizado para avaliar e classificar a implementação do framework proposto para este trabalho foi o "Richardson Maturity Model - Modelo de Maturidade Richardson", já que no modelo "CoHA Maturity Model" o nível mais alto é o próprio REST não necessariamente implementado no nível três de maturidade da escala de Richardson e Ruby (RICHARDSON; RUBY, 2007). Outro motivo que explica a escolha pelo modelo de Richardson e Ruby (RICHARDSON; RUBY, 2007), é o fato de que nesta abordagem o foco é a arquitetura REST, diferente do modelo CoHA que considera um cenário mais amplo, pois avalia implementações de sistemas baseados em HTTP, sem ficar restrito à implementações REST.

Em relação ao modelo de Richardson e Ruby (RICHARDSON; RUBY, 2007), o nível 0 é a ausência de qualquer regra, ou seja, apenas a utilização do HTTP como transporte das operações no servidor. Normalmente se usa apenas um endpoint (URI) e um verbo HTTP. Já no nível 1, existe a aplicação de resources - recursos. A API é dividida em diferentes endpoints que apontam para um ou mais resources. No nível 2, ocorre a imple-

mentação de verbos HTTP para diferentes tipos de operações que deseja executar. Uma mesma URI pode aceitar mais de um verbo HTTP, por exemplo: GET/produtos pode retornar todos os produtos e POST/produtos informando os atributos de um produto pode criar um novo. No nível de maturidade, exige a aplicação de HATEOAS, que adiciona controles hipermídia às representações. O uso de controles hipermídia permite aos clientes da Web API manipular os recursos de forma exploratória e desacoplada dos detalhes de implementação.

Este trabalho se propôs a implementação de um framework que segue os conceitos e técnicas abordados nos níveis 1, 2 e 3 de maturidade, no modelo Richardson e Ruby (RICHARDSON; RUBY, 2007). Através do framework é possível desenvolver serviços RESTful utilizando aplicação de recursos com a implementação de verbos HTTP para diferentes tipos de operações que deseja executar. Este serviço fica disponível através da classe de interface *BaseDeServiçoInterface*, que aplica os princípios de endereçabilidade, representação de recursos em diferentes formatos, identificador uniforme e a interface unificada, sendo que este último princípio é obtido junto a utilização da classe *InterceptorClienteRESTful*, que retorna respostas autoexplicativas às requisições dos clientes e pode ser aplicada também para o uso de Layered System na arquitetura do serviço web, dividindo esta arquitetura em camadas, visando maior facilidade na manutenção e menor impacto aos clientes quando ocorrer manutenções nos servidores, além da descentralização de responsabilidades em apenas um serviço Web. Também é possível implementar os princípios de HATEOAS e conectividade, ou seja, o controle de hipermídia nos recursos através das classes *HATEOASEntidade* e *Link*, o qual a classe de modelo carrega uma lista de links que podem conter ligações para um próximo registro ou anterior, bem como conter um endereço de imagem, música, vídeo ou documento.

No próximo capítulo serão apresentadas as conclusões referentes a este trabalho e também propostas para trabalhos futuros.

5 CONCLUSÕES

Neste trabalho foi realizada uma revisão bibliográfica sobre SOA, Web Services e suas implementações em SOAP e REST, fundamentos sobre web service REST e a arquitetura REST em Java. Por fim, foram descritos alguns frameworks e Web APIs existentes para desenvolvimento de serviços Web REST, citando as características de cada ferramenta.

Espera-se que esta revisão bibliográfica sirva de alguma maneira a todos que desejarem aprender sobre o desenvolvimento de Web Service REST, assim como os auxilie na criação de serviços web RESTful nível três de maturidade no modelo de Richardson e Ruby (RICHARDSON; RUBY, 2007).

O framework implementado para este trabalho, apresentado no capítulo 3, teve como objetivos principais e específicos, proporcionar suporte ferramental baseado no padrão JSON e XML, reduzir o acoplamento entre a implementação do serviço web e aplicações cliente, visando diminuir o impacto gerado pela evolução dos sistemas, além de auxiliar e guiar o desenvolvedor a criação de serviços web RESTful nível três de maturidade, funcionalidade que outras ferramentas e framwrks de mercado apresentam, porém de maneira parcial e muitas vezes deixando o desenvolvedor acoplado a algumas classes complexas. Além destas características, o framework utiliza totalmente software livre, deixando o projeto livre a futuras implementações para novas funcionalidades e modelos de recursos.

Este projeto de graduação, além desta monografia, conta com a implementação do framework proposto, visando facilidade para desenvolvedores na construção de serviços web RESTful nível três de maturidade, já que fornece todos os serviços necessários para tal. No estudo de caso e para testar o framework, foi implementado os Web Services Intermediador e Final, visto que o serviço web intermediador foi implementado apenas para demonstrar o projeto sendo dividido em camada de sistema ou Layered System, além de mostrar a utilização da classe cliente *InterceptadorClienteRESTful* do framework. Já no serviço web final é onde ocorre toda as consultas e persistência na base de dados. Ainda foram implementadas duas aplicações clientes junto ao estudo de caso, sendo uma aplicação desktop Java e outra web em PHP. Estas aplicações comunicam-se com o serviço web intermediador e consomem os recursos disponibilizados pelo mesmo.

Conclui-se após a realização deste estudo que o framework proposto fornece de forma correta seguindo os princípios e paradigmas da arquitetura, todos os serviços RESTful nível três de maturidade no modelo de Richardson e Ruby (RICHARDSON; RUBY, 2007), tais como a representação de dados através de recursos, identificador uniforme, interface unificada com respostas auto-explicativas, controle de hípermídia e Layered System. Ele fornece todo o suporte ferramental para que desenvolvedores possam utilizá-los para a construção rápida e padroniza de serviços RESTful nível três de maturidade. Vale observar que foi notável a facilidade para integração entre sistemas através da arquitetura

REST e de representações nos formatos JSON, XML, HTML e Text, ainda que este último não tenha sido implementado. Exceto texto, os formatos citados são estruturados, e partindo desta característica, a manipulação destes em diferentes linguagens de programação torna-se fácil. Mesmo que o formato texto não seja estrutura, pode ser aplicado alguma estrutura sobre o mesmo, como por exemplo, o formato CSV.

Finalizando, chega-se a conclusão de que quatro motivos para o uso do REST seriam a agilidade e simplicidade, uso correto do protocolo HTTP, produtividade e clareza, além da manutenção. A escolha pelo java neste projeto de framework, justifica-se pelo motivo de ser uma tecnologia bastante utilizada no mercado de trabalho e que está em constante desenvolvimento e por já existir a JSR ou JSR311, especificação oficial do Java para implementações de serviços web REST.

A seguir serão apresentadas algumas propostas de trabalhos futuros.

5.1 Trabalhos Futuros

Em continuidade ao estudo realizado neste trabalho, pretende-se realizar a implementação de uma classe que fique responsável por controlar o cache nas requisições HTTP GET. Esta classe ficaria junto ao framework. Além disso, também é planejado a implementação de uma tabela de links, para então armazenar os links de cada registro de domínio em uma tabela da base de dados. Porém neste caso é necessário a implementação de upload de imagens, música e vídeo para o servidor através do Web Service, o que torna um outro trabalho futuro. Por fim, clientes mais completos.

REFERÊNCIAS

- ALGERMISSEN, J. **Classification of HTTP-based APIs.** [S.l.]: <http://nordsc.com/ext/classification-of-http-based-apis.html>., 2010.
- BIENVENIDO, D. **Apache Isis - Java Framework for Domain-Driven Design.** [S.l.]: <http://www.infoq.com/news/2013/01/apache-isis-java-domain-driven>, 2013.
- BIH, J. **Service oriented architecture (SOA) a new paradigm to implement dynamic e-business solutions.** [S.l.]: Ubiquity, New York, p.1-1, 08 ago. 2006, 2006.
- BLAKE, M. B. et al. **Binding Now or Binding Later:** the performance of uddi registries. [S.l.]: System Sciences 40th Annual Hawaii International Conference on, Janeiro 2007., 2007.
- COMER, D. **Internetworking with TCP/IP:** principles, protocols, and architectures. [S.l.]: 4.ed. Prentice Hall, 2000. 750 p., 2000.
- DUMBILL, E. et al. **Programming Web Services with XML-RPC.** [S.l.]: Sebastopol - O'Reilly, 2001., 2001.
- ERRADI, A.; MAHESHWARI, O. **A broker-based approach for improving web services reliability.** [S.l.]: IEEE International Conference on Web Services (ICWS 05), 2005., 2005.
- FARKAS, P.; CHARAF, H. **Web Services Planning Concepts.** [S.l.]: Journal Of .net Technologies, p.9-12, 2003., 2003.
- FERRIS, C.; FARREL, J. **What are Web Services?** [S.l.]: Communications Of The Acm, New York, página 31, jun.2003, 2003.
- FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures.** [S.l.]: University of California. Irvane. 2000., 2000.
- FILHO, O. F. F. **Serviços Semânticos:** uma abordagem restful. [S.l.]: Escola Politécnica da Universidade de São Paulo. São Paulo, p. 103. 2009., 2009.
- FOWLER, M. **Richardson Maturity Model. Steps toward the glory of rest.** [S.l.]: <http://martinfowler.com/articles/richardsonMaturityModel.html>. Acesso em: 21/11/2015, 2010.
- GOMES, D. A. **Web Services SOAP em Java.** [S.l.]: Novatec 2010., 2010.

GONSALVES, A. **Beginning Java EE 6 Platform with GlassFish 3 - From Novice to Professional.** [S.l.]: New York - Apress, 2009., 2009.

GOURLEY, D.; TOTTY, B. **HTTP: the definitive guide.** [S.l.]: Sebastopol, CA, USA: O'Reilly, 2002. 10 p., 2002.

HADLEY, M. **Web Application Description Language.** [S.l.]: W3C Member Submission, 2009. Disponivel em www.w3.org/Submission/wadl/ Acesso em 23 fev. 2012., 2009.

HAMMER-LAHAV, E. **RFC 5849 - The OAuth 1.0 Protocol.** [S.l.]: Internet Engineering Task Force (IETF), 2010. Disponivel em tools.ietf.org/html/rfc5849. Acesso em 21 mar. 2012., 2010.

HATEOAS, S. **SPRING. Spring HATEOAS - Reference Documentation.** [S.l.]: <http://projects.spring.io/spring-hateoas>, 2013.

HAYWOOD, D. **Restful Objects Specification(v1.0.0).** [S.l.]: <http://restfulobjects.org>, 2012.

HONG, Y. **A Resource-Oriented Middleware Framework for Heterogeneous Internet of Things.** [S.l.]: Cloud and Service Computing (CSC), 2012 International Conference on, p. 12- 16, 22-24, Novembro 2012., 2012.

INC., G. **A well earned retirement for the SOAP Search API - The Official Google Code.** [S.l.]: Disponivel em googlecode.blogspot.com.br/2009/08/well-earned-retirement-for-soap-search.html. Acesso em 21 mar. 2012., 2009.

JERSEY - RESTful Web Services in Java. jersey.java.net/.

JOHN, D.; RAJASREE, M. S. **A framework for the description, discovery and composition of restful semantic web services.** [S.l.]: In Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology. New York, NY, USA <http://doi.acm.org/10.1145/2393216.2393232>. p. 88–93, 2012.

KAMALELDIN, M.; DUMINDA, W. **Performance Analysis of Web Services on Mobile Devices.** [S.l.]: Procedia Computer Science, v. 10, p. 744-751, 2012., 2012.

KREGER, H. et al. **Web Services Conceptual Architecture (WSCA 1.0).** [S.l.]: Disponível em www.cs.uoi.gr/zarras/mdw-ws/WebServicesConceptualArchitectu2.pdf. Acesso em 24 fev. 2008. maior 2001., 2001.

LEAVITT, N. **Are Web Services Finally Ready to Deliver.** [S.l.]: Computer, p.14-18, 2004., 2004.

O'REILLY, T. **Design Patterns and Business Models for the Next Generation of Software.** [S.l.]: O'REILLY Spreading the knowledge of innovators, 2005. Disponivel em <<http://oreilly.com/web2/archive/what-is-web-20.html>>. Acesso em 15 abr. 2012., 2007.

ORT, E. **Service Oriented Architecture and Web Services - Concepts, Tecnologies, and tools.** [S.l.]: The Source - Sum microsystems, 2005.

PAPAZOGLOU, M. P. **Service -Oriented Computing - Concepts, Characteristics and Directions.** [S.l.]: Proceedings Of The Fourth International Conference On Web Information Systems Engineering (wise 03), 2003., 2003.

- PAPAZOGLOU, M. P.; GEORGAKOPOULOS, D. **Service-Oriented Computing.** [S.l.]: Communications Of The Acm, p.25-28, out. 2001., 2001.
- RICHARDSON, L.; RUBY, S. **RESTful Web Services.** [S.l.]: O'Reilly, 2007., 2007.
- SAMPAIO, C. **SOA e Web Services em Java.** [S.l.]: Editora Brasport. 2006, 2006.
- SHOMOYITA, J.; RALPH, D. **Using a Cloud-Hosted Proxy to support Mobile Consumers of RESTful Services.** [S.l.]: Procedia Computer Science, v. Volume 5, p. 625-632, 2011., 2011.
- SILVEIRA, P.; OTHERS. **Introdução a Arquitetura e Design de Software.** [S.l.]: Uma visão geral sobre a plataforma Java, 1a edição São Paulo. Casa do código 2013., 2013.
- THOMAS, J. P.; THOMAS, M.; GHINEA, G. **Modeling of Web Services Flow.** [S.l.]: Proceedings Of The Ieee International Conference On E-commerce (cec'03), 2003., 2003.
- TIDWELL, D.; SNELL, J.; KULCHENKO, P. **Programing Web Services with SOAP.** [S.l.]: 1. ed. O'Reilly, 2001. 216 p., 2001.
- TOMCAT, A. **The apache software foundation.** [S.l.]: <http://tomcat.apache.org>, 2005.
- WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. **REST in Practice:** hypermedia and systems architecture. [S.l.]: O'Reilly, 2010. ISBN 978-0-596-80582-1, 2010.

APÊNDICE A ANEXOS E APÊNDICES

```

21  public String adicionaRecurso(String stringObjetoJSON, String url, String formatoDeDados) throws IOException, HttpException{
22@    if( (formatoDeDados != "json") && (formatoDeDados != "xml") ) formatoDeDados = "json";
23
24    PostMethod postMethod = new PostMethod(url);
25    RequestEntity entity = new InputStreamRequestEntity(new ByteArrayInputStream(stringObjetoJSON.getBytes()), "application/"+formatoDeDados);
26
27    postMethod.setRequestEntity(entity);
28    postMethod.setRequestBody(entity);
29    HttpClient client = new HttpClient();
30
31    String retorno = "0";
32    try {
33      int result = client.executeMethod(postMethod);
34      if(formatoDeDados == "json") {
35        retorno = "{"+
36          "\"codigoResposta\": "+ result+", "
37          "\"headersRespostaCabecalhos\": [";
38        Header[] headers = postMethod.getResponseHeaders();
39        for(int i = 0; i < headers.length; i++) retorno += (i != (headers.length-1)) ?
40          "{\"header\":\\\""+headers[i].clone().toString()+"\\\","
41          : "
42            "\"header\":\\\""+headers[i].clone().toString()+"\\\"}";
43        retorno += "
44      }";
45    } else {
46      retorno = "<?xml version=\"1.0\"?> ";
47      retorno += "<codigoResposta>"+result+"</codigoResposta>";
48      retorno += "<headersRespostaCabecalhos>";
49      Header[] headers = postMethod.getResponseHeaders();
50      for(int i = 0; i < headers.length; i++) retorno += (i != (headers.length-1)) ?
51        "<header>"+headers[i].clone().toString()+"</header> "
52        : "
53          "<header>"+headers[i].clone().toString()+"</header> ";
54      retorno += "</headersRespostaCabecalhos>";
55    }
56  }
57  } finally {
58    postMethod.releaseConnection();
59  }
60  System.out.println("InterceptadorClienteRESTful=>adicionaRecurso: \n"+retorno+"\n");
61  return retorno;
62}
63

```

Figura A.1: Classe InterceptadorClienteRESTful - Método adicionarRecurso.

```

64 public String obterRecurso(String url, String formatoDeDados) {
65     if( (formatoDeDados != "json") && (formatoDeDados != "xml") ) formatoDeDados = "json";
66     String retorno = "null";
67     try {
68         Client client = Client.create();
69         WebResource webResource = client.resource(url);
70         ClientResponse response = webResource.accept("application/"+formatoDeDados).get(ClientResponse.class);
71         int result = response.getStatus();
72
73         if( (formatoDeDados == "json") && (response.getStatus() != 200) ) return "{\"codigoResposta\":\""+String.valueOf(result)+"\"}";
74         else if( (response.getStatus() != 200) ) {
75             retorno = "<?xml version='1.0'?> ";
76             retorno += "<codigoResposta>" + result + "</codigoResposta>";
77             retorno += "
```

Figura A.2: Classe InterceptadorClienteRESTful - Método obterRecurso.

```

90
91  public String atualizarRecurso(String url, String formatoDeDados, String stringObjetoJSON) throws IOException, HttpException{
92      if( (formatoDeDados != "json") && (formatoDeDados != "xml") ) formatoDeDados = "json";
93
94      PutMethod putMethod = new PutMethod(url);
95      RequestEntity entity = new InputArrayInputStream(stringObjetoJSON.getBytes(), "application/"+formatoDeDados);
96
97      putMethod.setRequestEntity(entity);
98      putMethod.getRequestEntity();
99
100     HttpClient client = new HttpClient();
101     String retorno = "0";
102     try {
103         int result = client.executeMethod(putMethod);
104         if(formatoDeDados == "json") {
105             retorno = "{"+
106                 "\\"codigoResposta\\": \""+ result+"\"";
107             retorno += "\\"headersRespostaCabeca\\hos\\":[" ;
108             Header[] headers = putMethod.getResponseBodyHeaders();
109             for(int i = 0; i < headers.length; i++) retorno += (i != (headers.length-1)) ?
110                 "{\"header\": \""+headers[i].toString()+"\","
111                 :"{"\\"header\\": \\""+headers[i].toString()+"\""
112                 :"{"\\"header\\": \\""+headers[i].toString()+"\"}";
113             retorno += "}";
114         } else {
115             retorno = "<?xml version='1.0\\'> ";
116             retorno += "<codigoResposta>" + result + "</codigoResposta>";
117             retorno += "<headersRespostaCabeca\\hos\\":[" ;
118             Header[] headers = putMethod.getResponseBodyHeaders();
119             for(int i = 0; i < headers.length; i++) retorno += (i != (headers.length-1)) ?
120                 "<header>" + headers.clone()[i].toString() + "</header>" +
121                 "<header>" + headers.clone()[i].toString() + "</header>" ;
122             retorno += "</headersRespostaCabeca\\hos\\>";
123         }
124     }
125     System.out.println("InterceptadorClienteRESTful=>atualizarRecurso: \n"+retorno+"\n");
126     return retorno;
127     } finally{
128         putMethod.releaseConnection();
129     }
130 }
131

```

Figura A.3: Classe InterceptadorClienteRESTful - Método atualizarRecurso.

```

132
133@  public String excluirRecurso(String url) throws IOException, IOException{
134      DeleteMethod deleteMетод = new DeleteMethod(url);
135      HttpClient client = new HttpClient();
136      String retorno = "0";
137      try {
138          int result = client.executeMethod(deleteMетод);
139          retorno = "{"+
140                  "\"codigoResposta\": " + result+"," ;
141                  "\"headersRespostaCabecalhos\": [";
142          Header[] headers = deleteMетод.getResponseHeaders();
143          for(int i = 0; i < headers.length; i++) retorno += (i != (headers.length-1)) ?
144                  "{\"header\": \"" + headers[i].toString() + "\"," +
145                  "\"header\": \\" + headers[i].toString() + "\",
146                  \""]"+
147                  "}";
148
149          System.out.println("InterceptadorClienteRESTful=>excluirRecurso: \n" + retorno + "\n");
150          return retorno;
151      } finally {
152          deleteMетод.releaseConnection();
153      }
154
155
156
157
158

```

Figura A.4: Classe InterceptadorClienteRESTful - Método excluirRecurso.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 @<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 xmlns="http://java.sun.com/xml/ns/javaee"
4 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
5 id="WebApp_ID" version="3.0">
6
7 <display-name>TCCRRESTfulWebService</display-name>
8
9 <servlet>
10 <servlet-name>Web Service RESTful</servlet-name>
11 <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
12 <init-param>
13 <param-name>jersey.config.server.provider.packages</param-name>
14 <param-value>recursos</param-value>
15 </init-param>
16 <load-on-startup>1</load-on-startup>
17 </servlet>
18 <servlet-mapping>
19 <servlet-name>Web Service RESTful</servlet-name>
20 <url-pattern>/rest/*</url-pattern>
21 </servlet-mapping>
22 <security-constraint>
23 <web-resource-collection>
24 <web-resource-name>Permitted Actions</web-resource-name>
25 <url-pattern>/rest/*</url-pattern>
26 <http-method>GET</http-method>
27 <http-method>POST</http-method>
28 <http-method>PUT</http-method>
29 <http-method>DELETE</http-method>
30 </web-resource-collection>
31 </security-constraint>
32 </web-app>
33

```

Figura A.5: Configuração do Web Service final e intermediador definida no arquivo web.xml.



The screenshot shows a code editor window with the title "persistence.xml". The XML code inside the editor is as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="1.0"
3   xmlns="http://java.sun.com/xml/ns/persistence">
4   <persistence-unit name="RESTfullTCCPersistenceUnit">
5     <provider>org.hibernate.ejb.HibernatePersistence</provider>
6     <class>model.Produto</class>
7     <properties>
8       <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
9       <property name="hibernate.hbm2ddl.auto" value="update" />
10      <property name="hibernate.show_sql" value="true" />
11      <property name="hibernate.format_sql" value="true" />
12      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
13      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/RESTfullTCC" />
14      <property name="javax.persistence.jdbc.user" value="root" />
15      <property name="javax.persistence.jdbc.password" value="MySQLrootadmin123" />
16    </properties>
17  </persistence-unit>
18 </persistence>
```

Figura A.6: Arquivo persistence.xml, referente a configuração do Hibernate para acesso a base de dados MySQL.

```
1 package recursos;
2
3 import java.util.ArrayList;
4 import java.util.Date;
5 import java.util.List;
6 import javax.ws.rs.Path;
7 import org.codehaus.jettison.json.JSONArray;
8 import org.codehaus.jettison.json.JSONException;
9 import org.codehaus.jettison.json.JSONObject;
10 import com.google.gson.Gson;
11 import model.Produto;
12 import controlador.ProdutoControlador;
13 import servicos.BaseDeServiçoInterface;
14 import servicos.Link;
15
16 @Path("/produtosI")
17 public class ProdutoRecurso implements BaseDeServiçoInterface {
18     public ProdutoRecurso() { super(); }
19 }
```

Figura A.7: Classe ProdutoRecurso parte 01.

```

20@override
21 public String create(String objetoJSONComDados) {
22     String objetoJSONOriginal = objetoJSONComDados;
23     objetoJSONComDados = "{produtos:" + objetoJSONOriginal + "}";
24     JSONObject produtoJSONObject;
25     try {
26         produtoJSONObject = new JSONObject(objetoJSONComDados);
27         JSONArray arrayProdutosJSON = null;
28         if(produtoJSONObject.getJSONObject("produtos").length() > 1) {
29             arrayProdutosJSON = produtoJSONObject.getJSONObject("produtos");
30             Gson gson = new Gson();
31             List<Produto> listaDeProdutosConsumidos = new ArrayList<Produto>();
32             for(int a = 0; a < arrayProdutosJSON.length(); a++) {
33                 String produtoEmJSON = arrayProdutosJSON.getString(a);
34                 listaDeProdutosConsumidos.add(a, gson.fromJson(produtoEmJSON, Produto.class));
35             }
36             ProdutoControlador produtoControlador = new ProdutoControlador();
37             produtoControlador.beginTransaction();
38             for(int a = 0; a < listaDeProdutosConsumidos.size(); a++) {
39                 Produto produto = listaDeProdutosConsumidos.get(a);
40                 produto.setDataCriacao(new Date());
41                 produto.setaDataAtualizacao(new Date());
42                 produtoControlador.save(produto);
43             }
44             produtoControlador.commit();
45             return "200";
46         } else if(produtoJSONObject.getJSONObject("produtos").length() == 1) {
47             objetoJSONComDadosOriginal = objetoJSONOriginal.substring(1, (objetoJSONComDadosOriginal.length() -1) );
48             ProdutoControlador produtoControlador = new ProdutoControlador();
49             produtoControlador.beginTransaction();
50             Gson gson = new Gson();
51             Produto produto = gson.fromJson(objetoJSONComDadosOriginal, Produto.class);
52             produtoControlador.save(produto);
53             produtoControlador.commit();
54             return "200";
55         }
56     } catch (JSONException e) {
57         e.printStackTrace();
58         return "500";
59     }
60 }
61 return "500";
62 }
63
64

```

Figura A.8: Classe ProdutoRecurso parte 02.

```

65@override
66    public String get(Long id) {
67        ProdutoControlador produtoControlador = new ProdutoControlador();
68        Produto produto = null;
69        produto = produtoControlador.find(id);
70
71        Long idAnterior = null;
72        idAnterior = produtoControlador.getIdAnteriorRegistro(id);
73
74        Long idProximo = null;
75        idProximo = produtoControlador.getIdProximoRegistro(id);
76
77        if(produto != null) {
78            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "POST"));
79            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "GET"));
80            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml", "self", "GET"));
81            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json", "self", "GET"));
82            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html", "self", "GET"));
83            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text", "self", "GET"));
84            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produto.getId(), "self", "GET"));
85            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + produto.getId(), "self", "GET"));
86            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json/" + produto.getId(), "self", "GET"));
87            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html/" + produto.getId(), "self", "GET"));
88            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text/" + produto.getId(), "self", "GET"));
89            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produto.getId(), "self", "PUT"));
90            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produto.getId(), "self", "DELETE"));
91
92        if( (idAnterior != null) && (produto.getId() != idAnterior) )
93            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + idAnterior, "anterior", "GET"));
94        else {
95            idAnterior = produtoControlador.getIdUltimoRegistro();
96            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + idAnterior, "anterior", "GET"));
97        }
98
99        if( (idProximo != null) && (produto.getId() != idProximo) )
100            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + idProximo, "proximo", "GET"));
101        else {
102            idProximo = produtoControlador.getIdPrimeiroRegistro();
103            produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + idProximo, "proximo", "GET"));
104        }
105
106        Gson gson = new Gson();
107        return gson.toJson(produto);
108    }
109    return "204";
110}
111

```

Figura A.9: Classe ProdutoRecurso parte 03.

```

112
113 @Override
114 public String get() {
115     ProdutoControlador produtoControlador = new ProdutoControlador();
116     List<Produto> listaDeProdutos = produtoControlador.findAll();
117
118     int contador = 0;
119     for (Produto produtoLinha : listaDeProdutos) {
120         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "POST"));
121         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "GET"));
122         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml", "self", "GET"));
123         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json", "self", "GET"));
124         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html", "self", "GET"));
125         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text", "self", "GET"));
126         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "GET"));
127         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+produtolinha.getId()", "self", "GET"));
128         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json/+produtolinha.getId()", "self", "GET"));
129         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html/+produtolinha.getId()", "self", "GET"));
130         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text/+produtolinha.getId()", "self", "GET"));
131
132         if (contador == 0) {
133             Link link = new Link();
134             link.setHref("http://dc544.4shared.com/img/_UEh72Awba/s/1512ff005b0/CRF250L_Almeida");
135             link.setRel("photo");
136             link.setRel("image");
137             link.setRel("link");
138             produtoLinha.addLink(link);
139             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "DELETE"));
140
141             Link link2 = new Link();
142             link2.setHref("http://dc229.4shared.com/img/wpt-akOnba/s/71512ff0c130/Honda-CRF250L-1");
143             link2.setRel("photo");
144             link2.setRel("image");
145             produtoLinha.addLink(link2);
146         } else if (contador == (listaDeProdutos.size() - 1)) {
147             Link link = new Link();
148             link.setHref("http://dc254.4shared.com/img/2yakLk7M/s/7141e56762e0/header_newsletter");
149             link.setRel("photo");
150             link.setRel("image");
151             produtoLinha.addLink(link);
152             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "PUT"));
153         } else {
154             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "PUT"));
155             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "DELETE"));
156
157

```

Figura A.10: Classe ProdutoRecurso parte 04.

```

158 //Implementação da conectividade entre recursos.
159 if(contador == 0) {
160     produtoLinha.addLink(
161         new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get((listaDeProdutos.size() - 1)).getId(),
162             "anterior", "GET");
163     if((listaDeProdutos.get(contador + 1)) != null)
164         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get((contador + 1)).getId(),
165             "proxima", "GET"));
166 } else if(contador == (listaDeProdutos.size() - 1)) {
167     produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(0).getId(), "proximo", "GET"));
168     produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get((listaDeProdutos.size() - 2)).getId(),
169             "anterior", "GET"));
170 } else {
171     if(listaDeProdutos.get(contador - 1) != null) produtoLinha.addLink(
172         new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(contador - 1).getId(),
173             "anterior", "GET"));
174     if(listaDeProdutos.get(contador + 1) != null) produtoLinha.addLink(
175         new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(contador + 1).getId(),
176             "proxima", "GET"));
177     contador++;
178 }
179 }
180 if(listaDeProdutos.size() > 0) {
181     Gson gson = new Gson();
182     return gson.toJson(listaDeProdutos);
183 }
184 return "204";
185 }
186
187

```

Figura A.11: Classe ProdutoRecurso parte 05.

```
188
189@Override
190 public String update(Long id, String objetoJSONComDados) {
191     ProdutoControlador produtoControlador = new ProdutoControlador();
192     produtoControlador.beginTransaction();
193     Gson gson = new Gson();
194     Produto produto = produtoControlador.find(id);
195     if(produto != null) {
196         Produto produtoAux = gson.fromJson(objetoJSONComDados, Produto.class);
197         produto.setDataDeAtualizacao(new Date());
198         produto.setLinks(produtoAux.getLinks());
199         produto.setNome(produtoAux.getNome());
200         produto.setPreco(produtoAux.getPreco());
201         produtoControlador.update(produto);
202         produtoControlador.commit();
203         return "200";
204     }
205     return "204";
206 }
207
208
209@Override
210 public String delete(Long id) {
211     ProdutoControlador produtoControlador = new ProdutoControlador();
212     Produto produto = produtoControlador.find(id);
213     if(produto != null) {
214         produtoControlador.beginTransaction();
215         produtoControlador.delete(Produto.class, produto.getId());
216         produtoControlador.commit();
217         return "200";
218     }
219     return "500";
220 }
221
222
223@Override
224 public String getJSON() {
225     return get();
226 }
227
228@Override
229 public String getJSON(Long id) {
230     return get(id);
231 }
232
```

Figura A.12: Classe ProdutoRecurso parte 06.

```

233 @Override
234     public String getXML() {
235         ProdutoControlador produtoControlador = new ProdutoControlador();
236         List<Produto> listaDeProdutos = produtoControlador.findAll();
237         String resultado = "";
238         resultado += "<?xml version='1.0\'?"> +
239             "<produtos>";
240         int contador = 0;
241         for (Produto produtoLinha : listaDeProdutos) {
242             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "POST"));
243             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "GET"));
244             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml", "self", "GET"));
245             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json", "self", "GET"));
246             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html", "self", "GET"));
247             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text", "self", "GET"));
248             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produtoLinha.getId(), "self", "GET"));
249             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + produtoLinha.getId(), "self", "GET"));
250             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json/" + produtoLinha.getId(), "self", "GET"));
251             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html/" + produtoLinha.getId(), "self", "GET"));
252             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text/" + produtoLinha.getId(), "self", "GET"));
253
254         if (contador == 0) {
255             Link link = new Link();
256             link.setHref("http://dc544.4shared.com/img/_UEh72Awba/s7/1512ff005b0/CRF250L_Almeida");
257             link.setRel("photo");
258             link.setTipoDeMedia("image");
259             produtoLinha.addLink(link);
260             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produtoLinha.getId(), "self", "DELETE"));
261
262             Link link2 = new Link();
263             link2.setHref("http://dc229.4shared.com/img/wpt-akONba/s7/1512ff0c139/Honda-CRF250L-1");
264             link2.setRel("photo");
265             link2.setTipoDeMedia("image");
266             produtoLinha.addLink(link2);
267         } else if (contador == (listaDeProdutos.size() - 1)) {
268             Link link = new Link();
269             link.setHref("http://dc254.4shared.com/img/zyaKlk7W/s7/141e56762e0/header_newsletter");
270             link.setRel("photo");
271             link.setTipoDeMedia("image");
272             produtoLinha.addLink(link);
273             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produtoLinha.getId(), "self", "PUT"));
274         } else {
275             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produtoLinha.getId(), "self", "PUT"));
276             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/" + produtoLinha.getId(), "self", "DELETE"));
277
278

```

Figura A.13: Classe ProdutoRecurso parte 07.

```

278
279 //Implementação da conectividade entre recursos.
280
281 if(contador == 0) {
282     produtoLinha.addLink(
283         new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get((listaDeProdutos.size() - 1)).getId(),
284             "anterior", "GET"));
285
286     if((listaDeProdutos.get(contador + 1)) != null)
287         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get((contador + 1)).getId(),
288             "proxima", "GET"));
289
290 } else if(contador == (listaDeProdutos.size() - 1)) {
291     produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(0).getId(), "proxima", "GET"));
292
293     if(listaDeProdutos.get(contador + 1) != null)
294         produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(contador + 1).getId(),
295             "anterior", "GET"));
296
297 } else {
298     if(listaDeProdutos.get(contador - 1) != null) produtoLinha.addLink(
299         new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(contador - 1).getId(),
300             "anterior", "GET"));
301
302     if(listaDeProdutos.get(contador + 1) != null) produtoLinha.addLink(
303         new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/" + listaDeProdutos.get(contador + 1).getId(),
304             "proxima", "GET"));
305
306     contador++;
307
308     resultado += "<produto>" +
309         "<id>" + produtoLinha.getId() + "</id>" +
310         "<name>" + produtoLinha.getName() + "</name>" +
311         "<preco>" + produtoLinha.getPrice() + "</preco>" +
312         "<linkSHATEAS>";
313
314     for(int a = 0; a < produtoLinha.getLinks().size(); a++) {
315         resultado += (produtoLinha.getLinks().get(a).getTipoDeMidia() != null) ?
316             "<link>" +
317             "<href>" + produtoLinha.getLinks().get(a).getHref() + "</href>" +
318             "<rel>" + produtoLinha.getLinks().get(a).getRel() + "</rel>" +
319             "<metodo>" + produtoLinha.getLinks().get(a).getMethod() + "</metodo>" +
320             "</link>" +
321             "<link>" +
322             "<href>" + produtoLinha.getLinks().get(a).getHref() + "</href>" +
323             "<rel>" + produtoLinha.getLinks().get(a).getRel() + "</rel>" +
324             "<metodo>" + produtoLinha.getLinks().get(a).getMethod() + "</metodo>" +
325             "</link>" +
326             "</linksHATE0AS>" +
327             "</produto>" +
328         resultado += "</produtos>" +
329         if(listaDeProdutos.size() > 0) return resultado;
330     }
331
332     return "204";
333
334 }

```

Figura A.14: Classe ProdutoRecurso parte 08.

```

328
329 @Override
330 public String getXML(Long id) {
331     ProdutoControlador produtoControlador = new ProdutoControlador();
332     Produto produto = produtoControlador.findById(id);
333
334     Long idAnterior = null;
335     idAnterior = produtoControlador.getIdAnteriorRegistro(id);
336
337     Long idProximo = null;
338     idProximo = produtoControlador.getIdProximoRegistro(id);
339
340     if(produto != null) {
341         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "POST"));
342         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "GET"));
343         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml", "self", "GET"));
344         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json", "self", "GET"));
345         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html", "self", "GET"));
346         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text", "self", "GET"));
347         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produto.getId()", "self", "GET"));
348         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+produto.getId()", "self", "GET"));
349         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json/+produto.getId()", "self", "GET"));
350         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html/+produto.getId()", "self", "GET"));
351         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text/+produto.getId()", "self", "GET"));
352         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produto.getId()", "self", "PUT"));
353         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/-produto.getId()", "self", "DELETE"));
354
355     Link link = new Link();
356     link.setHref("http://dc254.4shared.com/img/2yaklk7M/s7/141e56762e0/header_newsletter");
357     link.setRel("photo");
358     link.setTipoDeMedia("image");
359     produto.addLink(link);
360
361     if( (idAnterior != null) && (produto.getId() != idAnterior) ) produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+idAnterior",
362                                         "anterior", "GET"));
363
364     else {
365         idAnterior = produtoControlador.getIdUltimoRegistro();
366         produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+idAnterior", "anterior", "GET"));
367
368         if( (idProximo != null) && (produto.getId() != idProximo) ) produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+idProximo",
369                                         "proximo", "GET"));
370
371         else {
372             idProximo = produtoControlador.getIdPrimeiroRegistro();
373             produto.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+idProximo", "proximo", "GET"));
374
375             String resultado = "";
376             resultado += "<?xml version='1.0\'?> " +
377             "<produtos>";
378

```

Figura A.15: Classe ProdutoRecurso parte 09.

```

379 resultado += "<produto>" +
380     "<id>" + produto.getId() + "</id>" +
381     "<nome>" + produto.getNome() + "</nome>" +
382     "<preco>" + produto.getPreco() + "</preco>" +
383     "<linksHATE0AS>" ;
384 for(int a = 0; a < produto.getLinks().size(); a++) {
385     resultado += (produto.getLinks().get(a).getTipoDeMidia() != null) ?
386         "<link>" +
387         "<href>" + produto.getLinks().get(a).getHref() + "</href>" +
388         "<rel>" + produto.getLinks().get(a).getRel() + "</rel>" +
389         "<tipoDeMidia>" + produto.getLinks().get(a).getTipoDeMidia() + "</tipoDeMidia>" +
390         "</link>" +
391         "<link>" +
392         "<href>" + produto.getLinks().get(a).getHref() + "</href>" +
393         "<rel>" + produto.getLinks().get(a).getRel() + "</rel>" +
394         "<metodo>" + produto.getLinks().get(a).getMetodo() + "</metodo>" +
395         "</link>" ;
396     resultado +=
397         "</linksHATE0AS>" +
398         "</produto>" ;
399 resultado += "</produtos>" ;
400 return resultado;
401 }
402 return "204";
403 }
404 }
405 }
406 
```

Figura A.16: Classe ProdutoRecurso parte 10.

```

@Overriede
408 public String getHTML() {
409     ProdutoControlador produtoControlador = new ProdutoControlador();
410     List<Produto> listaDeProdutos = produtoControlador.findAll();
411     String resultado = "<!DOCTYPE html> +
412         <html> +
413             <head> +
414                 <style> +
415                     "table, th, td {+
416                         border: 1px solid black; +
417                         border-collapse: collapse; +
418                     } +
419                     th, td {+
420                         padding: 15px; +
421                     } +
422                     </style> +
423             </head> +
424         <body> +
425             <table style='width:100%>" +
426                 <tr> +
427                     <th>Id</th> +
428                     <th>Nome</th> +
429                     <th>Preço</th> +
430                     <th>Imagem 1</th> +
431                     <th>Imagem 2</th> +
432                 </tr> ;
433             for (Produto produtoLinha : listaDeProdutos) {
434                 Link link = new Link();
435                 link.setHref("http://dc544.4shared.com/img/_UEh72Awba/s7/1512ff005b0/CRF250L_Almeida");
436                 link.setRel("photo");
437                 link.setIpodeMedia("image");
438                 produtoLinha.addLink(link);
439             }
440             Link link2 = new Link();
441             link2.setHref("http://dc229.4shared.com/img/wpt-akONba/s7/1512ff0130/Honda-CRF250L-1");
442             link2.setRel("photo");
443             link2.setIpodeMedia("image");
444             produtoLinha.addLink(link2);
445             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "POST"));
446             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/", "self", "GET"));
447             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml", "self", "GET"));
448             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json", "self", "GET"));
449             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html", "self", "GET"));
450             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text", "self", "GET"));
451             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "GET"));
452             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/xml/+produtolinha.getId()", "self", "GET"));
453             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/json/+produtolinha.getId()", "self", "GET"));
454             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/html/+produtolinha.getId()", "self", "GET"));
455             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/text/+produtolinha.getId()", "self", "GET"));
456             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "PUT"));
457             produtoLinha.addLink(new Link("http://localhost:8080/TCCRESTful/rest/produtos/+produtolinha.getId()", "self", "DELETE"));

```

Figura A.17: Classe ProdutoRecurso parte 11.

```
459     resultado += "<tr>" +  
460         "<td>" + produtoLinha.getId() + "</td>" +  
461         "<td>" + produtoLinha.getNome() + "</td>" +  
462         "<td>" + produtoLinha.getPreco() + "</td>" +  
463         "<td><a href='blank' target='_blank'>" + produtoLinha.getLinks() .get(0) .getHref() + ">IMAGE 01</a></td>" +  
464         "<td><a href='blank' target='_blank'>" + produtoLinha.getLinks() .get(1) .getHref() + ">IMAGE 02</a></td>" +  
465     "</tr>";  
466  
467     resultado += "468         "</body>" +  
469         "</html>";  
470  
471     if(listadeProdutos.size() > 0) {  
472         return resultado;  
473     }  
474     return "204";  
475 }  
476  
477 @Override  
478 public String getHTML(Long id) {  
479     return "";  
480 }  
481  
482 @Override  
483 public String getText() {  
484     return "";  
485 }  
486  
487 @Override  
488 public String getText(Long id) {  
489     return "";  
490 }  
491  
492 }
```

Figura A.18: Classe ProdutoRecurso parte 12.