

Arquitetura REST

Lívia Couto Ruback Rodrigues

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Orientadora: Profa. Regina Maria Maciel Braga Villela



Juiz de Fora, MG
Julho de 2009

Arquitetura REST

Lívia Couto Ruback Rodrigues

Monografia submetida ao corpo docente do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora, como parte integrante dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Aprovada pela banca constituída pelos seguintes professores:

Profa. Regina Maria Maciel Braga – Orientadora
DSc. em Engenharia de Sistemas e Computação,
COPPE/UFRJ, 2000

Prof. Marco Antônio Pereira Araújo
DSc. em Engenharia de Sistemas e Computação,
COPPE/UFRJ, 2009

Profa. Jesuliana Nascimento Ulysses
Mestre em Computação
UFF

Juiz de Fora, MG
Julho de 2009

Agradecimentos

Agradeço a Deus, por me dar forças para seguir adiante.

Agradeço a meus familiares, em especial à minha mãe, pelo incentivo e apoio incondicionais em todos os momentos.

Aos amigos e colegas de sala por compartilhar comigo bons momentos e ajudar a superar os momentos difíceis.

Ao meu namorado Adriano por se fazer sempre presente, pelo carinho, paciência e dedicação.

E a todos que de alguma forma contribuíram não somente para o desenvolvimento deste trabalho, mas também para a conclusão do curso.

Sumário

Lista de Reduções	vi
Lista de Figuras	vii
Resumo.....	ix
Abstract.....	x
1 Introdução.....	1
1.1 Motivação	1
1.2 Objetivos.....	2
1.3 Organização do trabalho.....	2
2 Arquitetura de Software.....	3
2.1 Histórico	3
2.2 Conceito e Papel da Arquitetura de Software.....	4
2.4 Estilos Arquiteturais.....	9
2.5 Documentação de Arquiteturas.....	11
2.5.2 Visões Arquiteturais	11
2.5.3 Documentação de Interfaces	13
3 Arquitetura para Serviços Web	15
3.1 Introdução.....	15
3.2 Arquitetura.....	16
3.3 Modelos de Arquitetura de Serviços Web.....	19
3.3.1 Modelo Orientado a Mensagens	20
3.3.2 Modelo Orientado a Serviço	20
3.3.3 Modelo Orientado a Recursos	21
3.3.4 Modelo de Policiamento	22

3.4	Tecnologias para Serviços Web.....	22
3.5	Integração e Operações entre as Entidades	27
4	Arquitetura REST	30
4.1	Fundamentação Básica.....	30
4.1.1	Null Style.....	30
4.1.2	Cliente-servidor	31
4.1.3	Sem Estado	31
4.1.4	Cache.....	32
4.1.5	Interface Uniforme.....	32
4.1.6	Separação em Camadas.....	33
4.1.7	Código Sob Demanda	33
4.2	Elementos Arquiteturais REST.....	34
4.2.1	Elementos de Dados.....	34
4.2.1.1	Recursos e Identificadores de Recursos	35
4.2.1.2	Representação	35
4.2.2	Conectores.....	36
4.2.3	Componentes	38
4.2	Visões Arquiteturais REST	39
4.2.1	Visões de Processo.....	39
4.2.2	Visões de Conectores.....	40
4.2.3	Visões de Dados	40
4.3	Arquitetura Orientada a Recursos.....	41
4.3.1	Recursos	41
4.3.2	Endereçamento	42
4.3.3	Sem-Estado.....	42
4.3.4	Representações	43
4.3.5	Ligações e Conectividade.....	44
4.3.6	Interface Uniforme.....	44
4.4	Serviços Web segundo o modelo REST	45

5	Exemplos de Aplicação.....	48
5.1	Exemplo de abordagem tradicional de Serviços Web	48
5.2	Exemplo de Serviços Web RESTFul.....	52
5.3	Organizações que utilizam	58
5.4	Considerações.....	59
6	Conclusão.....	62

Lista de Reduções

XML	<i>Extensible Markup Language</i>
WSDL	<i>Web Services Description Language</i>
SOAP	<i>Simple Object Access Protocol</i>
OASIS	<i>Organization for the Advancement of Structured Information Standards</i>
HTTP	<i>Hypertext Transfer Protocol</i>
DSSA	<i>Domain-Specific Software Architecture</i>
ADL	<i>Architecture Description Language</i>
SEI	<i>Software Engineering Institute</i>
UML	<i>Unified Modeling Language</i>
W3C	<i>World Wide Web Consortium</i>
DCOM	<i>Distributed Component Object Model</i>
CORBA	<i>Common Object Request Broker Architecture</i>
RMI	<i>Remote Method Invocation</i>
REST	<i>Representational State Transfer</i>
WSA	<i>Web Service Architecture</i>
WSD	<i>Web Service Description</i>
UDDI	<i>Universal Description, Discovery and Integration</i>
URI	<i>Uniform Resource Identifier</i>

Lista de Figuras

Figura 2.1	Níveis de descrição de um sistema	5
Figura 2.2	Principais linguagens de descrição arquitetural	7
Figura 2.3	Exemplo de descrição ACME.....	8
Figura 2.4	Principais Estilos Arquiteturais	10
Figura 2.5	Áreas de Interesse dos Estilos Arquiteturais	10
Figura 3.1	Processo Geral de um Serviço Web	16
Figura 3.2	Relacionamento entre um WSD e seus agentes	18
Figura 3.3	Modelos da Arquitetura de Serviços Web	19
Figura 3.4	Esquema Simplificado do <i>Message Oriented Model</i>	20
Figura 3.5	Esquema simplificado do <i>Service Oriented Model</i>	21
Figura 3.6	Esquema simplificado do <i>Resource Oriented Model</i>	21
Figura 3.7	Esquema simplificado do <i>Policy Model</i>	22
Figura 3.8	Tecnologias Relacionadas aos Serviços Web	23
Figura 3.9	Estrutura da mensagem SOAP	24
Figura 3.10	Exemplo de mensagem SOAP	25
Figura 3.11	Estrutura de um documento WSDL	25
Figura 3.12	Exemplo de um documento WSDL	26
Figura 3.13	Interação entre as tecnologias Web Services	27
Figura 3.14	Operações entre as entidades relacionadas	28
Figura 4.1	<i>Null Style</i>	31
Figura 4.2	Estilo Arquitetural Cliente-Servidor	31
Figura 4.3	Estilo Arquitetural Cliente-servidor Sem Estado.....	32
Figura 4.4	Elementos de Dados REST	35
Figura 4.5	Conectores REST.....	37
Figura 4.6	Componentes REST.....	38
Figura 4.7	Métodos do protocolo HTTP utilizados pela Arquitetura Orientada a Recursos	45
Figura 4.8	Invocação e resposta de um método pela visão tradicional de Serviços Web	46
Figura 4.9	Invocação e resposta de um método pela visão REST de Serviços Web	47

Figura 5.1 Classe Filme.java	49
Figura 5.2 Classe FilmeWS.java	49
Figura 5.3 Arquivo FilmeWS.wsdl.....	50
Figura 5.4 Classe FilmeMain.java	51
Figura 5.5 Saída da execução da classe FilmeMain.java	51
Figura 5.6 Arquivo web.xml da aplicação	52
Figura 5.7 Classe Filme.java	53
Figura 5.8 Classe de Recurso FilmeResource.java.....	54
Figura 5.9 Classe de teste FilmeMain.java.....	56
Figura 5.10 Saída da execução da classe de teste FilmeMain.java.....	57
Figura 5.11 Requisição do tipo GET através de um Web Browser.....	58

Resumo

A Arquitetura de Serviços Web surgiu para permitir a interoperabilidade entre aplicações rodando em diferentes plataformas. Ela foi especificada com base em um protocolo que encapsula as mensagens – SOAP (*Simple Object Access Protocol*) e em uma linguagem que descreve as interfaces dos serviços, conhecida como WSDL (*Web Services Description Language*). A forma tradicional pela qual os serviços web são implementados hoje em dia, que segue aos padrões citados, gera uma camada de abstração que envolve o protocolo HTTP.

Este trabalho tem por finalidade apresentar os conceitos e tecnologias que envolvem uma nova abordagem no contexto de implementação de serviços – o paradigma REST. Criado a partir da tese de doutorado de Roy Fielding e baseado em outros estilos arquiteturais, tem como principal meta a simplicidade e o desempenho das aplicações. Além disso, este trabalho aborda uma aplicação prática dos conceitos arquiteturais de REST – a Arquitetura Orientada a Recursos – que permite transformar um problema em uma aplicação REST através do emprego de URI's, HTTP e XML.

Abstract

The architecture of Web services has emerged to allow interoperability between applications running on different platforms. It was specified based on a protocol that encapsulates messages - SOAP (Simple Object Access Protocol) and in a language that describes the interfaces of services, WSDL (Web Services Description Language). The traditional way in which web services are implemented today is a layer of abstraction that includes the HTTP protocol.

This paper aims to present the concepts and technologies that involve a new approach for implementation of services - the REST paradigm. Created by Roy Fielding in his thesis, it uses some architectural styles in order to get applications simplicity and better performance. Moreover, this paper addresses a practical application of REST architectural concepts - the Resource Oriented Architecture - which allows transforming a problem in an REST application through the use of URI, HTTP and XML.

1 Introdução

A complexidade dos softwares vem aumentando muito rapidamente de forma que as técnicas para lidar com os problemas existentes - conhecidas como técnicas de abstração - já se tornaram insuficientes para lidar com os desafios da nova geração de sistemas. Uma destas técnicas de abstração é a *modularização* de um sistema. Dessa forma, todo sistema de software de alto nível de abstração pode ser descrito por meio de subsistemas ou módulos inter-relacionados. O papel de analisar as propriedades de cada subsistema ou módulo é atribuído à arquitetura de software.

Com o surgimento de várias arquiteturas de software, fez-se necessária a categorização das mesmas de acordo com suas características, componentes e propriedades peculiares. Dessa forma, as classes de arquiteturas foram sendo classificadas em seus respectivos estilos arquiteturais. Concomitante ao surgimento dos estilos arquiteturais surgiu uma arquitetura com o objetivo de permitir a interoperabilidade entre aplicações e sistemas de plataformas, ambientes e arquiteturas diferentes – a Arquitetura de Serviços Web.

A abordagem tradicional do uso de Serviços Web utiliza uma interface descrita em um formato específico - WSDL (*Web Services Description Language*). Essa interface permite que sistemas interajam com um serviço Web através de mensagens em um formato específico, conhecido como SOAP (*Simple Object Access Protocol*), normalmente enviadas utilizando HTTP. Dessa forma, os serviços estão escondidos por trás dos métodos que podem ser invocados remotamente.

1.1 Motivação

O surgimento da Arquitetura de Serviços Web tornou possível a interoperabilidade entre aplicações heterogêneas. Porém, muitas vezes o desempenho do sistema é comprometido, devido ao tamanho das mensagens envelopadas no protocolo SOAP.

No entanto, surgiram algumas alternativas que podem melhorar o desempenho das

aplicações com serviços Web. Uma destas alternativas é a que tem sido mais focada atualmente é a arquitetura orientada a recursos, baseada no paradigma REST – *Representational State Transfer*.

1.2 Objetivos

O objetivo deste trabalho é estudar os conceitos relacionados ao estilo REST e a Arquitetura Orientada a Recursos, suas vantagens, desvantagens e, sobretudo suas diferenças em relação à abordagem tradicional de manipulação de Serviços Web.

1.3 Organização do trabalho

Este trabalho, além desta introdução, apresenta o conceito e papel da Arquitetura de Software no capítulo 2, onde ainda são conceituados e exemplificados estilos arquiteturais, uma visão geral de Linguagens de Descrição Arquitetural e de Documentação de Arquiteturas.

Em seguida, no capítulo 3, a Arquitetura de Serviços Web é apresentada de acordo com a sua especificação. São abordadas as tecnologias de Serviços Web existentes e os modelos de Arquiteturas definidas pela especificação.

Já o capítulo 4 conceitua o estilo REST através do processo de derivação do mesmo como um estilo arquitetural. Apresenta também a Arquitetura Orientada a Recursos, uma aplicação prática dos conceitos por trás do REST.

O capítulo 5 ilustra um exemplo de uma consulta à um acervo de filmes por ambas as abordagens – visão tradicional e REST. O capítulo 6 apresenta conclusões tiradas a partir do conhecimento adquirido com o presente trabalho.

2 Arquitetura de Software

2.1 Histórico

Atualmente, cada vez mais o computador vem representando para as pessoas e instituições uma ferramenta essencial e imprescindível para a realização de suas atividades, desde as mais básicas até as complexas. Concomitante a essa crescente dependência dos sistemas computacionais, a complexidade dos softwares vem aumentando tão rapidamente que as técnicas para lidar com a complexidade de problemas existentes - conhecidas como técnicas de abstração - já se tornaram insuficientes para lidar com os desafios da nova geração de sistemas.

O crescimento contínuo do tamanho e da complexidade dos sistemas requer cada vez mais atributos tais como confiabilidade, desempenho e manutenibilidade. A confiabilidade é fundamental para garantir que o sistema realizará as tarefas conforme o esperado e pode ser medida de forma estatística através da análise do comportamento do sistema durante um determinado período de tempo [MENDES, 2003]. O desempenho também tem se tornado um fator crítico, devido principalmente às novas necessidades dos sistemas. Já a manutenibilidade se desmembra em dois aspectos distintos: reparo em funcionalidades (correção de erros) e evolução do sistema (inserção de novos requisitos). Dessa forma, com o passar do tempo, todo e qualquer sistema precisará de manutenção.

A Engenharia de Software surgiu com o propósito de lidar com esses problemas. Porém, eles vão além do âmbito das estruturas de dados e dos algoritmos utilizados nos sistemas. Tornou-se então imprescindível, principalmente em sistemas de grande porte, projetar a estrutura global do sistema antes de iniciá-lo, ou seja, desenvolver o software orientado a uma arquitetura. Foi então a partir da necessidade de um projeto arquitetural para os sistemas que surgiu o interesse em Arquitetura de Software.

2.2 Conceito e Papel da Arquitetura de Software

De acordo com o livro *An Introduction to Software Architecture*, (David Garlan e Mary Shaw - 1994), a arquitetura de software surgiu para resolver questões que vão "além dos algoritmos e das estruturas de dados da computação. O projeto e a especificação da estrutura global de um sistema emergem como um problema importante. As questões estruturais incluem organização total e estrutura de controle global; protocolos de comunicação, sincronização e acesso aos dados; atribuição de funcionalidades a elementos de design; distribuição física; composição de elementos de design; escalonamento e desempenho; e seleção entre as alternativas de design".

Em 1995, David Garlan e Dewayne Perry definiram a arquitetura de software como: "A estrutura dos componentes de um programa/sistema, seus inter-relacionamentos, princípios e diretrizes guiando o projeto e evolução ao longo do tempo" [BUSCHMANN, 1996] .

A Arquitetura de Software pode ser entendida, de maneira geral, como o estudo da organização global dos sistemas de software e das inter-relações entre seus subsistemas e componentes. Todo sistema de software de alto nível de abstração que for projetado seguindo alguma arquitetura pode então ser descrito por meio de subsistemas ou módulos inter-relacionados. O papel da arquitetura de software é analisar as propriedades de cada subsistema ou módulo. Isso proporciona, entre outras vantagens, o desenvolvimento de linhas de produto de software – das quais podem ser criados vários sistemas com funcionalidades distintas. Essa prática é conhecida como *arquitetura de referência* ou *DSSA (Domain Specific Software Architecture)* [MENDES, 2002] e utiliza conceitos como reuso com base em componentes, classes de componentes, entre outros.

A utilização de uma arquitetura no gerenciamento do processo de software pode também prover suporte para a estimativa de custos e na gerência do processo e principalmente pode atuar como a estrutura principal para atender aos requisitos do sistema [MENDES, 2002]. Para entender melhor o seu papel, a figura 2.1 ilustra o afinilamento nas descrições arquiteturais de um sistema através das fases do processo de desenvolvimento do software, que começa pela concepção do sistema, passa pela análise e aceitação dos requisitos e vai até sua efetiva implementação.

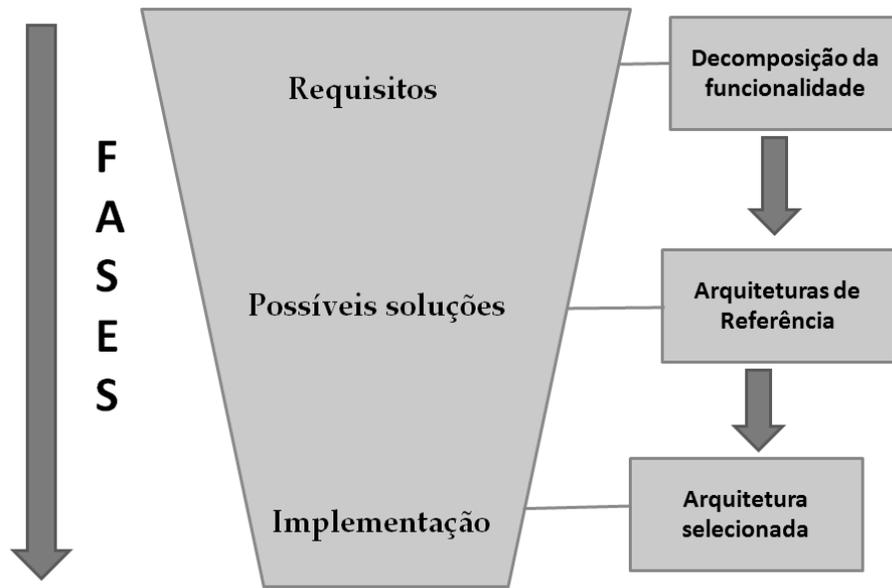


Figura 2.1 Níveis de descrição de um sistema [Mendes, adaptado, 2003]

Conforme apresentado na figura 2.1, a fase inicial não requer nenhuma tomada de decisão no âmbito arquitetural e é responsável pela decomposição das funcionalidades do sistema em módulos. A compreensão das funcionalidades pode ser feita através da análise dos serviços oferecidos por sistemas similares. Durante a fase final, a análise do sistema em módulos permite encontrar a arquitetura ideal que satisfaça às reais necessidades do sistema dentre a gama de arquiteturas de referência disponíveis na fase intermediária.

Analisando as fases como um todo, a arquitetura de um sistema engloba o particionamento das funções entre subsistemas ou módulos e a interação entre eles. Portanto, a arquitetura é composta de subsistemas ou componentes, conectores (interligando os componentes), e de padrões de conexão (tipos de interação e compartilhamento de informações entre esses componentes). Durante o desenvolvimento do software, várias questões são levantadas no processo de modularização do sistema e as suas respostas são fundamentais para a construção de uma boa solução. A função de avaliar essas questões e decidir qual alternativa é a melhor para cada situação-problema é atribuída a um profissional especializado, o *arquiteto de software*.

Ao arquiteto de software cabe tomar as decisões técnicas, liderar, coordenar as atividades e os artefatos técnicos no decorrer do projeto. Porém, a visão do arquiteto deve ser ampla - e não detalhada - pois ele deve ter um conhecimento geral do sistema em um nível alto de abstração.

Uma arquitetura de software pode ser especificada utilizando duas abordagens principais:

Linguagens de Descrição Arquiteturais e/ou Estilos Arquiteturais. Serão detalhadas a seguir estas duas abordagens.

2.3 Linguagem de Descrição Arquitetural (*ADL - Architectural Description Language*)

A partir da necessidade natural de representar de forma linguística as arquiteturas de software principalmente visando facilitar a comunicação entre os envolvidos nos projetos, surgiram as ADLs (em português, LDAs – Linguagens de Descrição Arquitetural). As linguagens de descrição arquitetural possibilitam a descrição de um sistema em um alto nível de abstração, oferecem uma base formal para a descrição e análise do comportamento do sistema e têm a pretensão de serem entendidas não só por humanos, mas também por máquinas [CHAVEZ, 2009].

Outra vantagem proporcionada pelo uso de ADLs é a reusabilidade, pois elas podem prover, em situações específicas, a geração automática de sistemas. Cada ADL define a natureza e propriedades do componente, a semântica das conexões e o comportamento do sistema como um todo.

Segundo o Software Engineering Institute (SEI) [SEI, 2009], as principais linguagens de descrição arquitetural são (figura 2.2):

ADL	Organização
AADL	Society for Automotive Engineers (SAE)
ACME	Carnegie Mellon University
AESOP	Carnegie Mellon University
CODE	University of Texas at Austin
ControlH & MetaH	Honeywell Technology Center
Demeter	Northeastern University
FR	Ohio State University
Gestalt	Siemens Corporate Research, Inc.
Modechart	University of Texas at Austin
Rapide	Stanford University
RESOLVE	Ohio State University
UML	Rational Software Corporation
UniCon	Carnegie Mellon University
Wright	Carnegie Mellon University

Figura 2.2 Principais linguagens de descrição arquitetural [SEI, adaptado, 2009]

A seguir será abordada mais detalhadamente uma das ADL's citadas – a linguagem Acme, que surgiu a partir da necessidade de um padrão frente à proliferação de ADL's que surgiram, cada uma com suas capacidades específicas.

Acme

Acme é uma linguagem simples e genérica para a descrição de arquiteturas que pode ser usada como ferramenta para converter formatos de design e/ou como base para desenvolver novas arquiteturas e ferramentas de análise.

A linguagem Acme surgiu em 1995 a partir de contribuições e *feedbacks* de alguns membros da comunidade de projeto de arquiteturas. Os princípios da linguagem e os trabalhos para desenvolvimento de ferramentas foram financiados por um grupo da Carnegie Mellon University aliado a outros colaboradores. Surgiu com o objetivo de fornecer uma linguagem que pudesse ser utilizada como suporte para troca de arquiteturas entre diferentes ferramentas de design de arquiteturas [ACME, 2009].

A linguagem possui três características principais:

- Descrição de arquiteturas: a linguagem fornece um conjunto de construtores para

descrever a estrutura, tipo e estilo das arquiteturas e as propriedades dos elementos que compõem as mesmas.

- Troca de arquitetura: através de um formato de trocas genérico para design de arquiteturas, a Acme permite a integração de ferramentas desenvolvidas com outras ferramentas complementares.
- Extensibilidade para projeto de novas arquiteturas e ferramentas de análise: a solução fornecida pela Acme fornece uma base sólida e extensível que evita a reconstrução de infraestruturas padrão. Dessa forma torna-se possível desenvolver ferramentas usando Acme como linguagem de representação nativa, sendo compatível com diversas ADL's e ferramentas sem esforços adicionais de integração.

O site do projeto (<http://www.cs.cmu.edu/~acme/>) oferece uma introdução e uma visão geral do projeto Acme, tais como a especificação da linguagem, artigos, exemplos e downloads das bibliotecas e ferramentas.

A figura 2.3 ilustra um exemplo de uma descrição ACME. O componente cliente é declarado para se ter uma porta única para a requisição e outra somente para a resposta. O conector desempenha dois papéis: chamador e chamado (*caller* e *callee*). A topologia deste sistema é declarada através da lista de anexos [ACME, 2009].

```
System simple_cs = {
  Component client = {Port send-request}
  Component server = {Port receive-request}
  Connector rpc = {Roles {caller, callee}}
  Attachments : {
    client.send-request to rpc.caller;
    server.receive-request to rpc.callee
  }
}
```

Figura 2.3 Exemplo de descrição ACME [ACME, 2009]

Porém, a maioria dos trabalhos relacionados a ADL's atualmente ainda tem objetivos acadêmicos e são relativamente difíceis de analisar, pois ainda não há um entendimento universal no que se refere ao que uma ADL deve representar (no que diz respeito ao comportamento do

sistema).

Além disso, as representações de ADL's usadas hoje em dia não são suportadas por ferramentas comerciais e a maioria delas é muito focada em um determinado tipo de análise.

2.4 Estilos Arquiteturais

A partir da necessidade natural de organizar o conhecimento de projeto de software de maneira útil para uma possível reutilização, as arquiteturas de software foram sendo caracterizadas e, de acordo com suas características, componentes e propriedades peculiares, foram catalogadas. Dessa forma, classes de arquiteturas (ou famílias) foram sendo identificadas de acordo com os aspectos característicos a elas, culminando no surgimento dos **estilos arquiteturais**.

Para determinar um estilo arquitetural que retrata a arquitetura de software do sistema, um profissional de software deve determinar a classe a qual pertence a organização de um sistema. Para isso, são definidas características dos componentes (e os mecanismos de interação entre eles) e conectores do sistema, a topologia da arquitetura, entre outros.

2.4.1 Conceito e Propriedades

Shaw e Garlan [GARLAN; SHAW, 1994] definiram estilos arquiteturais como "Padrões ou idiomas de projeto que guiam a organização de módulos e subsistemas em sistemas completos". Já Medvidovic [MEDVIDOVIC, 2000] definiu como "idiomas de projeto chave que permitem a exploração de padrões estruturais e evolutivos adequados e que facilitam a reutilização de componentes, conectores e processo".

O *Application Architecture Guide 2.0* (Microsoft patterns and practices) [MICROSOFT, 2009], lista alguns dos estilos arquiteturais mais relevantes, descritos na figura 2.4.

Estilo de Arquitetura	Descrição
<i>Client-Server</i>	Segrega o sistema em duas aplicações, onde o cliente faz uma requisição de serviço ao servidor.
<i>Arquitetura baseada em Componentes</i>	Decompõe o design da aplicação em componentes lógicos e funcionais que são independentes de local e expõe interfaces de comunicação bem definidas.
<i>Arquitetura em Camadas</i>	Separa as responsabilidades da aplicação em grupos bem definidos (camadas).
<i>Message-Bus</i>	Um sistema de software que pode receber e enviar mensagens baseado em um conjunto de formatos conhecidos, de forma que sistemas possam se comunicar uns com os outros sem necessidade de conhecer o destinatário atual.
<i>N-tier / 3-tier</i>	Separa funcionalidade em segmentos de forma muito similar ao estilo de camadas, mas com cada segmento sendo uma camada localizado fisicamente em um computador separado.
<i>Orientado a Objetos</i>	Um estilo arquitetural baseado na divisão de tarefas para uma aplicação ou sistema em reutilização individual e objetos auto-suficientes, cada um contendo os dados e comportamentos relevantes ao objeto.
<i>Apresentação Separada</i>	Separa a lógica para gerenciar a interação do usuário da visualização da interface de usuário (UI) e dos dados com os quais o usuário trabalha.
<i>Service-Oriented Architecture (SOA)</i>	Refere-se a aplicações que expõe e consome funcionalidades como um serviço usando contratos e mensagens.

Figura 2.4 Principais Estilos Arquiteturais [MICROSOFT, adaptado, 2008]

Cada um desses estilos arquiteturais pode ser classificado de acordo com as áreas específicas de interesse de cada um. A figura 2.5 agrupa os estilos em áreas de interesse:

Categoria	Estilos de Arquitetura
<i>Comunicação</i>	Service-Oriented Architecture (SOA), Message Bus, Pipes e Filtros
<i>Deployment</i>	Client/server, 3-Tier, N-Tier
<i>Domain</i>	Domain Model, Gateway
<i>Interação</i>	Apresentação Separada
<i>Estrutura</i>	Baseado em Componentes, Orientado a Objetos, Arquitetura em Camadas

Figura 2.5 Áreas de Interesse dos Estilos Arquiteturais [MICROSOFT, adaptado, 2008]

Existem também outros estilos arquiteturais que não foram classificados originalmente, tais como: *Peer-to-Peer* [VERMA, 2004], *Shared Nothing Architecture* [STONEBRAKER, 1986] e REST (*Representational State Transfer*) [FIELDING, 2000].

2.5 Documentação de Arquiteturas

Além de técnicas para criar e avaliar arquiteturas de software, o SEI (Software Engineering Institute) desenvolve técnicas para documentá-las.

O foco da comunidade de engenharia de software se voltou para documentação de arquiteturas a partir de uma necessidade percebida pelo SEI de uma representação formal das mesmas, pois muitas vezes as arquiteturas criadas não são documentadas (e conseqüentemente comunicadas) de forma efetiva. Isso faz com que os envolvidos com o sistema não tenham acesso a uma representação adequada da arquitetura.

Baseado nessa necessidade, o principal motivo pelo qual as arquiteturas de software devem ser documentadas é que a documentação representa o artefato principal em todas as fases do desenvolvimento de sistemas [MERSON, 2006].

2.5.2 Visões Arquiteturais

Neste contexto, o SEI destaca a documentação de arquiteturas através de múltiplas *views*. Uma *view* representa uma perspectiva da arquitetura e contém elementos e relações entre os elementos. A definição de quais tipos de elementos e as relações de cada *view* servem de base para criar uma organização padrão para a documentação da arquitetura [MERSON, 2006].

A questão mais importante a ser considerada no contexto das *views* é como identificar quais delas merecem ser documentadas. Para auxiliar nessa tarefa, as *views* foram classificadas em cinco categorias: *Module Views*, *Runtime Views*, *Deployment Views*, *Implementation Views* e *Data Model*.

Module Views

As *module views* mostram a estrutura do sistema em termos de unidades de código, ou

seja, representam a planta (*blue prints*) para a construção do software. A notação mais utilizada para representar as *module views* são os diagramas UML – mais especificamente os diagramas de classes. Através deles, o sistema é decomposto em módulos e as dependências e hierarquias entre eles são representadas nos diagramas.

Runtime Views

As *Runtime Views* representam um “raio-x” do sistema em execução e são úteis para entender o funcionamento do sistema e analisar algumas propriedades que se manifestam somente em tempo de execução, como por exemplo desempenho. As notações usadas para representar *runtime views* geralmente são informais com a utilização de caixas e linhas, embora a UML tenha definido alguns tipos de elementos e relações que facilitam a criação de *runtime views*.

Implementation Views

As *Implementation Views* são responsáveis por exibir a estrutura do software (como é ou como deverá ser) em termos de arquivos, pacotes e diretórios, tanto para o ambiente de desenvolvimento quanto para de produção.

Notações informais são também muito utilizadas na representação de *implementation views*, embora a notação UML seja uma alternativa atraente.

Deployment Views

Uma *Deployment View* mostra a estrutura de hardware na qual o sistema é executado, geralmente uma rede. Também possui as notações formal e informal, sendo a informal mais utilizada.

Data Model

Os *Data Models* normalmente são usados para representar bases de dados cuja estrutura

precisa ser modelada. O *data model* inicia como um modelo conceitual/lógico que vai sendo refinado até conter todas as informações necessárias para a criação da base de dados física.

A notação mais utilizada para esse tipo de *view* é a de diagramas UML de entidade-relacionamento, nos quais as classes contenham apenas atributos e nenhum método.

Durante o processo de documentação de arquiteturas, o número de *views* e de seus elementos pode aumentar tanto a ponto de tornar impraticável manter todas as informações em uma só *view*. A técnica usada para particionar o design em módulos que possam ser visualizados, editados e entendidos separadamente é conhecida como técnica do *refinamento* [CLEMENT, 2001].

O refinamento pode ser feito de duas maneiras diferentes: a primeira ocorre quando uma visão é composta de muitos elementos e então a *view* é particionada em duas ou mais *views*, cada uma contendo um subconjunto de elementos. Neste caso as novas *views* criadas são “irmãs”. A segunda forma de refinamento é mais comum: quando uma determinada *view* possui um elemento muito complexo, é criada outra *view* para mostrar a estrutura interna desse elemento. Neste caso a nova *view* criada é “filha” da original.

2.5.3 Documentação de Interfaces

Embora não seja muito comum a documentação de interfaces, ela é necessária, pois a descrição de uma interface é considerada também uma informação arquitetural, pois expõe detalhes essenciais à integração daquele componente ao restante do sistema ou até mesmo a sistemas externos [CLEMENT, 2001].

Uma questão importante é quais itens são relevantes para serem documentados em uma interface. Merson [MERSON, 2006] sugere sete dos dez itens listados por Clement [CLEMENT, 2001]:

- Nome e descrição da interface;
- Métodos/Funções/Operações disponíveis (sintaxe, descrição, restrição de uso e exceções);
- Tipos de dados e constantes;
- Guia de Variabilidade

- Atributos de Qualidade
- Decisões de Projeto
- Exemplos de uso

Como apresentado no presente capítulo, os estilos arquiteturais surgiram para guiar a organização de módulos e subsistemas em sistemas completos. Os estilos foram classificados de acordo com as respectivas áreas de interesse de arquiteturas de software já existentes.

Além da escolha de um estilo arquitetural no desenvolvimento de um sistema, é também importante a utilização de uma linguagem de descrição arquitetural, que tem como principais vantagens a comunicação efetiva entre os envolvidos no projeto e a reutilização.

Porém, independente do foco do estilo arquitetural, é indispensável conhecer como a arquitetura de software e os estilos surgiram e a importância da documentação da arquitetura selecionada em todas as fases do projeto de software, através da utilização diferentes perspectivas da arquitetura – *views*.

3 *Arquitetura para Serviços Web*

3.1 **Introdução**

De acordo com a definição do World Wide Web Consortium (W3C) [W3C, 2009], um serviço Web é "um sistema de software responsável por proporcionar a interação entre duas máquinas diferentes através de uma rede. Para possibilitar essa interação, utiliza uma interface descrita em um formato específico - WSDL (*Web Services Description Language*). Essa interface permite que sistemas interajam com um serviço Web através de mensagens SOAP (*Simple Object Access Protocol*) ou de outros protocolos, normalmente enviadas utilizando HTTP com uma serialização em XML em conjunto com outros padrões utilizados na Web" [W3C, 2009]. Esses padrões serão detalhados mais adiante.

O W3C é um consórcio de empresas de tecnologia que cria padrões comuns para o conteúdo da Web e é apoiado por grandes empresas como Microsoft, IBM, HP, entre outras. No final do ano 2000 foi criado um grupo no W3C com propósito de desenvolver uma arquitetura que permitisse a interoperabilidade entre aplicações e sistemas de plataformas, ambientes e arquiteturas diferentes. Alguns padrões foram propostos, como o DCOM (*Distributed Component Object Model*) [DCOM, 2009], CORBA (*Common Object Request Broker Architecture*) [CORBA, 2009], e RMI (*Remote Method Invocation*) [RMI, 2009]. Porém as abordagens já existentes foram bem sucedidas somente na integração de redes locais e tiveram muitas limitações quando foram transpostas para sistemas de grande porte rodando sobre a Web, principalmente devido à independência de plataforma que deixou a desejar.

Visando atender a essas necessidades, foi proposta uma arquitetura computacional voltada para serviços Web, que tornou possível a interação entre diferentes tipos de aplicações mesmo rodando em plataformas e sistemas operacionais diferentes.

O objetivo deste trabalho é apresentar, de forma comparativa, um novo estilo de

implementação de serviços web – REST.

3.2 Arquitetura

A arquitetura de Serviços Web descrita neste trabalho tomou como base a referência arquitetural do W3C, o WSA (*Web Service Architecture*) [WSA, 2004] e irá fornecer uma definição comum de serviços Web, através de um modelo conceitual e um contexto no qual os componentes da arquitetura serão inter-relacionados. É importante ressaltar que este trabalho não tem a intenção de mostrar como os serviços Web devem ser implementados e sim de descrever as características arquiteturais mínimas comuns a todos eles.

A figura 3.1 ilustra o processo geral de um Serviço Web.

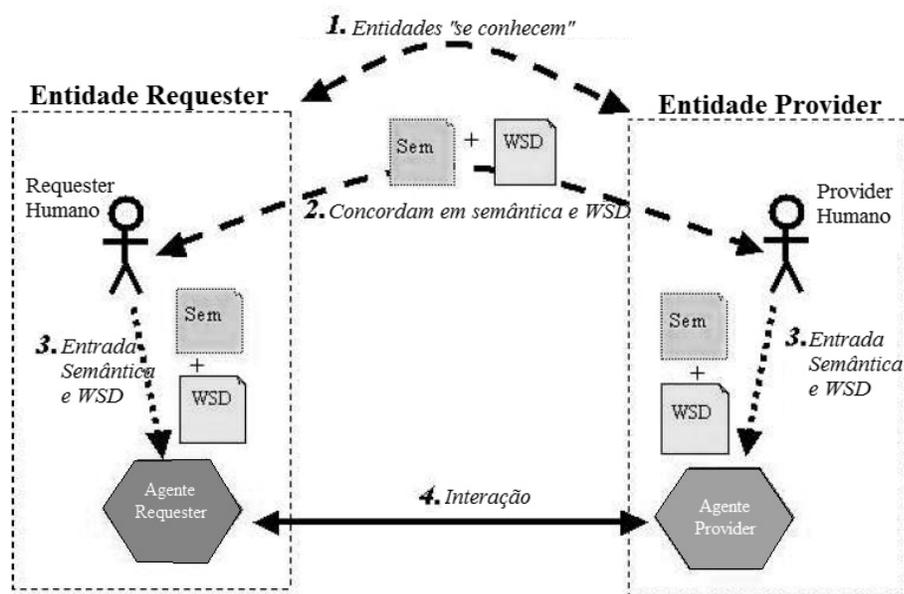


Figura 3.1 Processo Geral de um Serviço Web [WSA, 2004]

Há muitas maneiras pelas quais uma entidade pode acessar e usar um Serviço Web. Em geral, os passos necessários ilustrados na figura 3.1 são:

1. **As entidades *Requester* e *Provider* tornam-se conhecidas** (ou pelo menos uma se torna conhecida da outra).

2. **As entidades *Requester* e *Provider* chegam a um acordo** em relação ao WSD e a semântica que irá coordenar a interação entre os agentes.
3. **O WSD e a semântica são consumados** pelos agentes.
4. **As entidades *Requester* e *Provider* trocam mensagens**, através de seus respectivos agentes.

Apresentamos a seguir uma breve descrição dos principais componentes presentes na interação com um serviço Web.

a) Agentes e Serviços

Um serviço Web é uma noção abstrata que deve ser implementada por um agente concreto. O agente é um módulo de software ou hardware responsável por receber ou enviar mensagens, enquanto o serviço é caracterizado pelo conjunto de funcionalidades que serão fornecidas. A diferença entre um agente e um serviço pode ser visualizada mais claramente se um serviço Web for implementado usando um agente com diferentes linguagens de programação, porém com a mesma funcionalidade. Dessa forma, o Serviço Web permaneceria o mesmo [MEDYK, 2006].

Basicamente, um agente pode ser entendido como a concretização da noção abstrata (que nesse contexto representa um serviço). Em termos de padrões de projeto, um serviço representa uma interface a ser implementada por uma classe concreta, representada por um agente.

b) Requisitores (*Requesters*) e Provedores (*Providers*)

A proposta de um serviço Web é fornecer uma determinada funcionalidade em nome de alguma pessoa ou organização. A entidade *Provider* é a pessoa ou organização que fornece o agente adequado para implementar um determinado serviço, enquanto a entidade *Requester* é a pessoa ou organização que faz uso da entidade *Provider*.

Um *Requester Agent* é usado para a troca de mensagens com o *Provider Agent*. Ambos podem iniciar a troca de mensagens. Porém, na maioria dos casos, quem inicia a troca é o *Requester Agent*. Para garantir a troca de mensagens, as entidades precisam definir e acordar o

que o padrão de serviços Web chama de *semântica* [MEDYK, 2006].

c) Descrição do Serviço

Os mecanismos de troca de mensagens são documentados em um *Service Description* (descrição do serviço). O WSD (*Web Service Description*) é uma especificação para a interface do serviço Web, implementadas através de uma WSDL (*Web Service Description Language*).

A WSDL define os padrões dos formatos das mensagens, tipos de dados, protocolos de transportes, e formatos da serialização do transporte usados para a comunicação entre os agentes *Requester* e *Provider*. Mais adiante, um tópico abordará especificamente WSDL. De forma geral, o WSD representa um acordo de interação com um determinado serviço. A figura 3.2 ilustra o relacionamento entre um documento WSD e os agentes. [WSA, W3C]

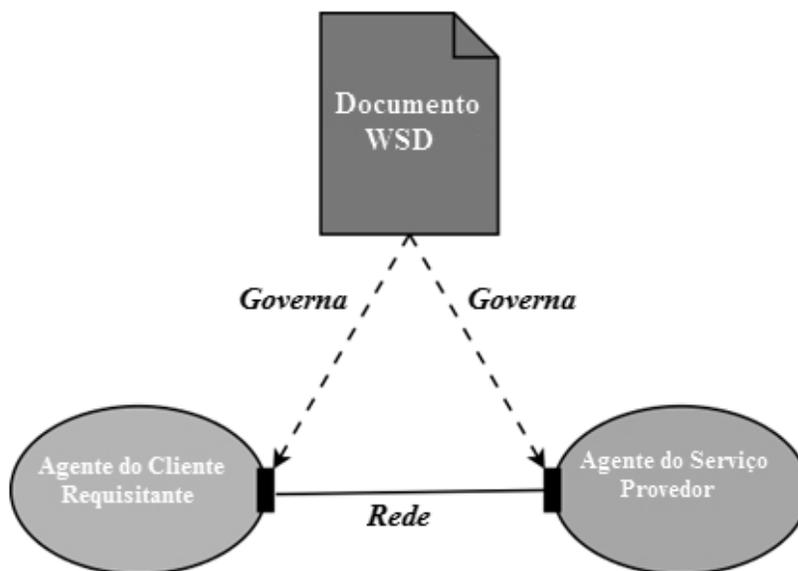


Figura 3.2 Relacionamento entre um WSD e seus agentes [WSA, 2004]

d) Semântica

A semântica de um Serviço Web é a expectativa compartilhada a respeito do comportamento do serviço, em particular a resposta para as mensagens que são enviadas aos

3.3.1 Modelo Orientado a Mensagens

O *Message Oriented Model* (Modelo Orientado a Mensagens) foca em aspectos da arquitetura que estão relacionados com o processamento de mensagens, como, por exemplo, sua estrutura, o modo de transporte, entre outros aspectos relevantes [MEDYK, 2006]. A figura 3.4 ilustra um esquema simplificado do modelo.

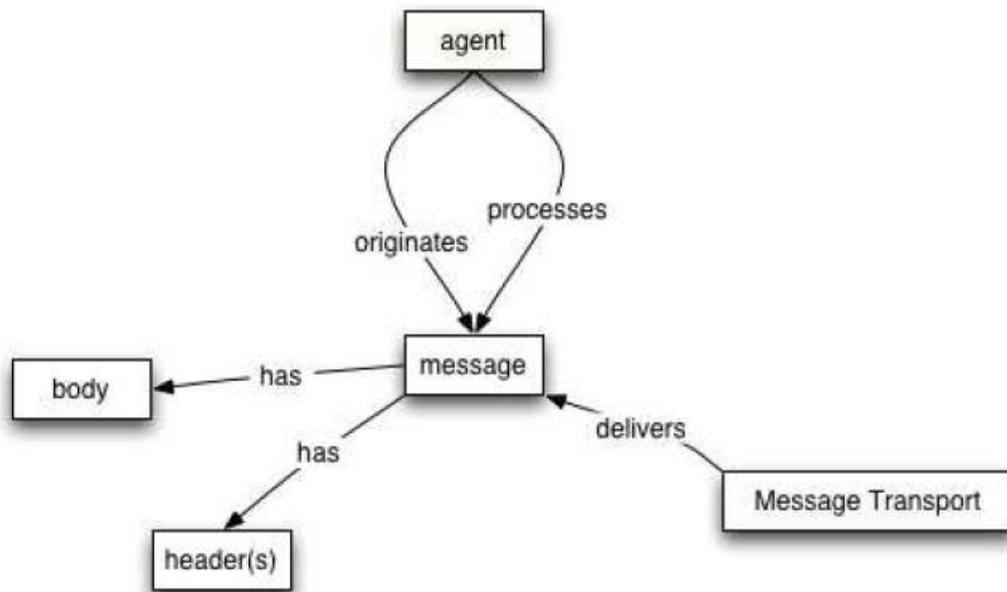


Figura 3.4 Esquema Simplificado do *Message Oriented Model* [WSA, 2004]

3.3.2 Modelo Orientado a Serviço

O *Service Oriented Model* (Modelo Orientado a Serviços) foca nos aspectos *serviço* e *ação*. Em sistemas distribuídos, os serviços podem não ser perfeitamente adequados se não houver uma modelagem da estrutura das mensagens utilizadas. Porém, o oposto é possível, já que as mensagens não precisam necessariamente estar atreladas a serviços. A figura 3.5 ilustra um esquema simplificado do modelo [MEDYK, 2006].

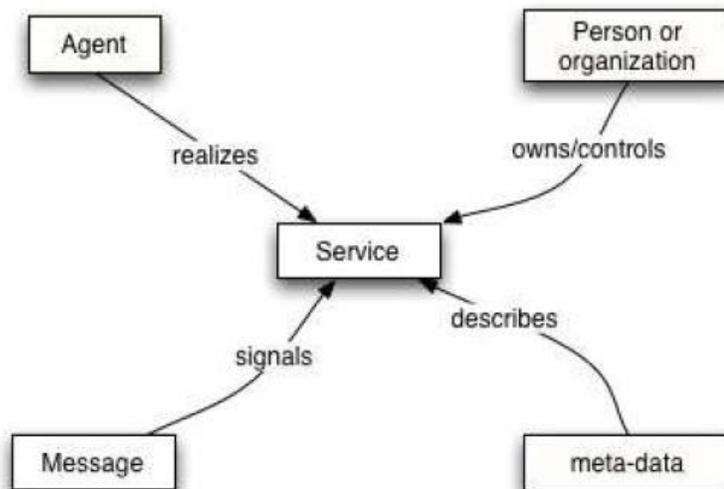


Figura 3.5 Esquema simplificado do *Service Oriented Model* [WSA, 2004]

3.3.3 Modelo Orientado a Recursos

O *Resource Oriented Model* (Modelo Orientado a Recursos) foca nos recursos e é reutilizado da arquitetura web, o qual é expandida para o conceito de serviços Web para incorporar o relacionamento entre os recursos e as entidades que os fornecem. A figura 3.6 ilustra um esquema simplificado do modelo [MEDYK, 2006].

Foi a partir do conceito comum de recursos do modelo orientado a recursos que surgiu o estilo REST e a arquitetura orientada a recursos.

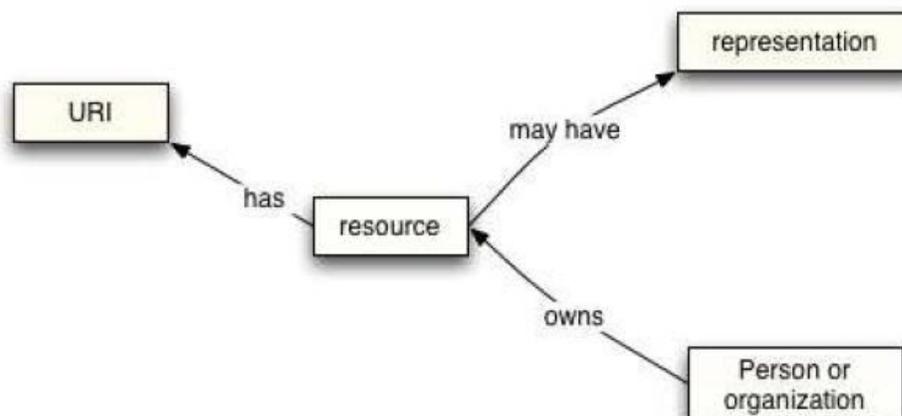


Figura 3.6 Esquema simplificado do *Resource Oriented Model* [WSA, 2004]

3.3.4 Modelo de Policiamento

O *Policy Model* (Modelo de Policiamento) foca no comportamento dos agentes e serviços. Os recursos do modelo são generalizados uma vez que o policiamento – vigilância do comportamento – pode ser aplicado igualmente a documentos, descrições do serviço, e a recursos computacionais. A figura 3.7 ilustra um esquema simplificado do modelo [MEDYK, 2006].

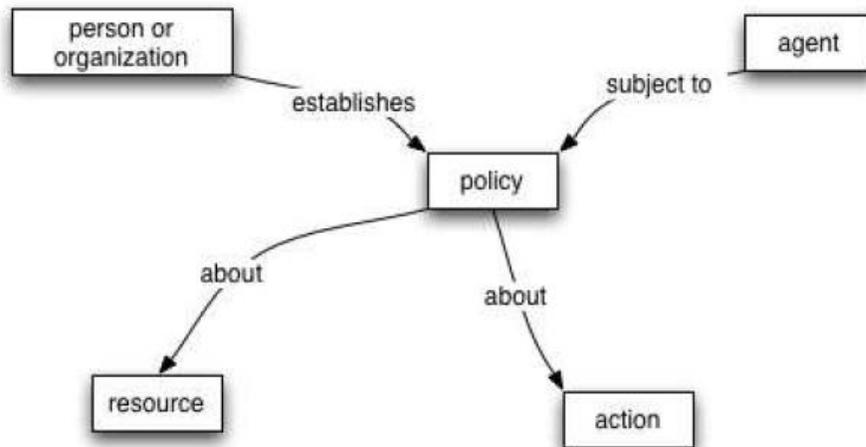


Figura 3.7 Esquema simplificado do *Policy Model* [WSA, 2004]

O policiamento se aplica sobre os recursos. Este modelo é aplicado sobre agentes que necessitam acessar diretamente estes recursos e é estabelecido pelo administrador dos mesmos. O modelo se relaciona com outros aspectos, tais como políticas de segurança, gerenciamento e aplicações.

3.4 Tecnologias para Serviços Web

A arquitetura para serviços Web envolve muitas camadas e tecnologias inter-relacionadas. Há diversas formas de visualizá-las, da mesma forma como existem muitas formas de implementação de Serviços Web. A figura 3.8 ilustra de forma geral as tecnologias relacionadas aos Serviços Web [W3C, 2009].

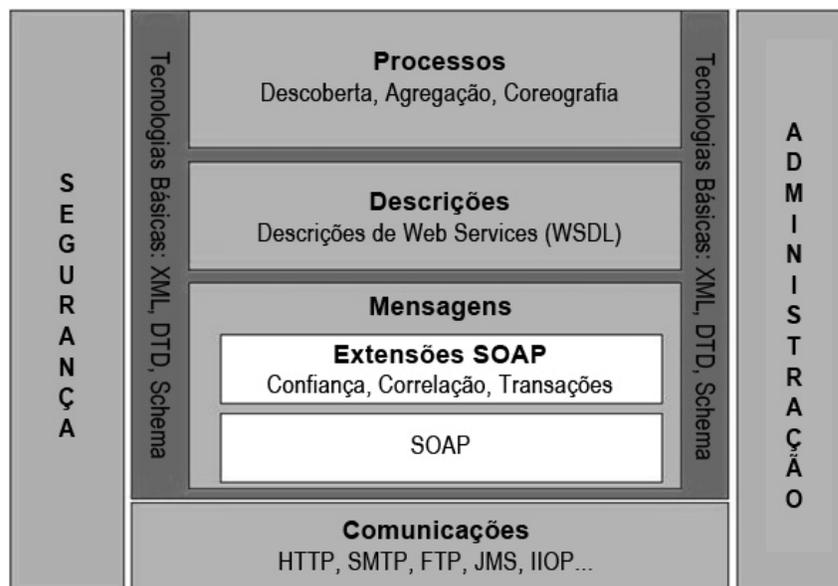


Figura 3.8 Tecnologias Relacionadas aos Serviços Web [W3C, 2009]

A seguir as principais tecnologias presentes na figura acima serão detalhadas.

XML (eXtensible Markup Language)

XML é uma linguagem de marcação criada pelo W3C a partir do SGML (*Standard Generalized Markup Language*). Por oferecer padronização, flexibilidade e a possibilidade de descrever classes com diversos tipos de dados, o uso de XML reduziu significativamente os custos de desenvolvimento. Estas classes de dados são documentos XML. [XML, W3C]

O objetivo principal da linguagem é facilitar o compartilhamento de informações pela Web, podendo ser usado também por serviços Web.

SOAP (Simple Object Access Protocol)

SOAP é um protocolo baseado em XML para a troca de informações estruturadas em ambientes distribuídos [MEDYK, 2006]. O protocolo provê uma forma de possibilitar a passagem de comandos e parâmetros entre as entidades *Requester* e *Provider*, independente da plataforma de implementação ou de linguagem de programação utilizada.

A descrição das mensagens utilizando SOAP e o protocolo de comunicação utilizado

(geralmente HTTP - *Hypertext Transfer Protocol*) formam a camada fundamental de comunicação dos serviços Web [MEDYK, 2006].

Uma mensagem SOAP, como ilustrada da figura 3.9, consiste basicamente nos seguintes elementos:

1. **Envelope:** É o elemento raiz do documento XML. Toda mensagem SOAP deve contê-lo.
2. **Header:** É um cabeçalho opcional. Quando utilizado, o header deve ser o primeiro elemento do envelope.
3. **Body:** É um elemento obrigatório e contém o *payload* (informação a ser transportada para o seu destino final). Pode conter um elemento opcional *fault*, usado para carregar mensagens de status e erros retornados pelos nós ao processarem a mensagem.

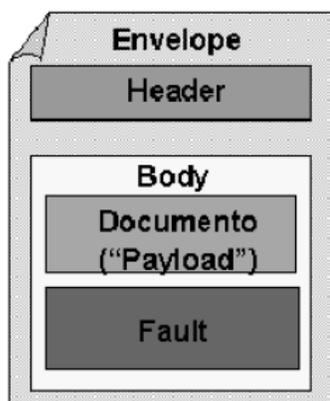


Figura 3.9 Estrutura da mensagem SOAP [MEDYK, 2006]

A figura 3.10 ilustra um exemplo de mensagem de requisição SOAP. O corpo da mensagem contém os dados necessários à busca do preço do livro informado.

```

<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
  ...
  </env:Header>
  <env:Body>
    <m:GetPrecoLivro xmlns:m="http://traca.com.br/livraria/definicoes">
      <m:PrecoLivroRequisicao>
        <m:isbn>1234567890</m:isbn>
      </m:PrecoLivroRequisicao>
    </m:GetPrecoLivro>
  </env:Body>
</env:Envelope>

```

Figura 3.10 Exemplo de mensagem SOAP [SOUZA, 2009]

WSDL (*Web Service Description Language*)

Um serviço Web deve definir todas as suas interfaces, operações, esquemas de codificação, entre outros, em um documento para que a entidade *Requester* saiba o formato dos métodos a serem chamados e quais parâmetros devem ser passados. WSDL é a linguagem de metadados utilizada nesses documentos para estes fins [MEDYK, 2006].

Um documento WSDL define um *XML Schema* (linguagem baseada em XML para definição de regras de validação em documentos XML) para descrever um serviço Web. Possui uma parte abstrata, que descreve a interface de um serviço e as operações suportadas e uma parte concreta, que especifica os formatos de dados e protocolos específicos que serão utilizados. A figura 3.11 mostra a estrutura de um documento WSDL.



Figura 3.11 Estrutura de um documento WSDL [ROSSINI, 2008]

A figura 3.12 ilustra um exemplo de um documento WSDL que contém a definição de um

método – `getQuantidadeEmEstoque`. O documento define também o parâmetro da requisição – código - e os parâmetros da resposta - o código e a quantidade em estoque.

```
<element name="getQuantidadeRequest">
  <complexType>
    <all><element name="codigo" type="xsd:string"/></a
  </complexType>
</element>

<element name="getQuantidadeResponse">
  <complexType>
    <all>
      <element name="codigo" type="xsd:string"/>
      <element name="quantidade" type="xsd:integer"/>
    </all>
  </complexType>
</element>

<interface name="OperacoesRequestResponse">
  <operation name="getQuantidadeEmEstoque">
    <input message="getQuantidadeRequest" />
    <output message="getQuantidadeResponse" />
  </operation>
</interface>
```

Figura 3.12 Exemplo de um documento WSDL [SOUZA, 2009]

UDDI (Universal Description Discovery and Integration)

O *Service Registry* (Registro de Serviços) é a entidade que representa a localização central onde o provedor de serviços pode relacionar os seus serviços web, possibilitando assim a pesquisa e a descoberta desses serviços. Este papel é desempenhado pela UDDI, que consiste em um serviço estruturado na forma de repositórios para nomeação e localização dos serviços web [BELLWOOD, 2002].

O padrão UDDI foi criado em agosto de 2000 pela organização OASIS [OASIS, 2006] e especifica um protocolo para obter e atualizar um diretório comum de informação dos serviços Web. O diretório inclui informações sobre *Service Providers*, os serviços que eles fornecem e os protocolos que são implementados por aquele serviço [MEDYK, 2006].

O padrão foi desenvolvido para ser acessado através de mensagens SOAP e fornecer acesso aos documentos WSDL descrevendo os protocolos e os formatos das mensagens necessários para a interação com determinado serviço listado no diretório.

O UDDI pode ser visto como as “páginas amarelas” dos serviços Web. De forma análoga às páginas amarelas tradicionais, um *UDDI Registry* oferece informações categorizadas sobre os serviços e as funcionalidades que eles oferecem.

A figura 3.13 mostra a interação entre as tecnologias citadas.

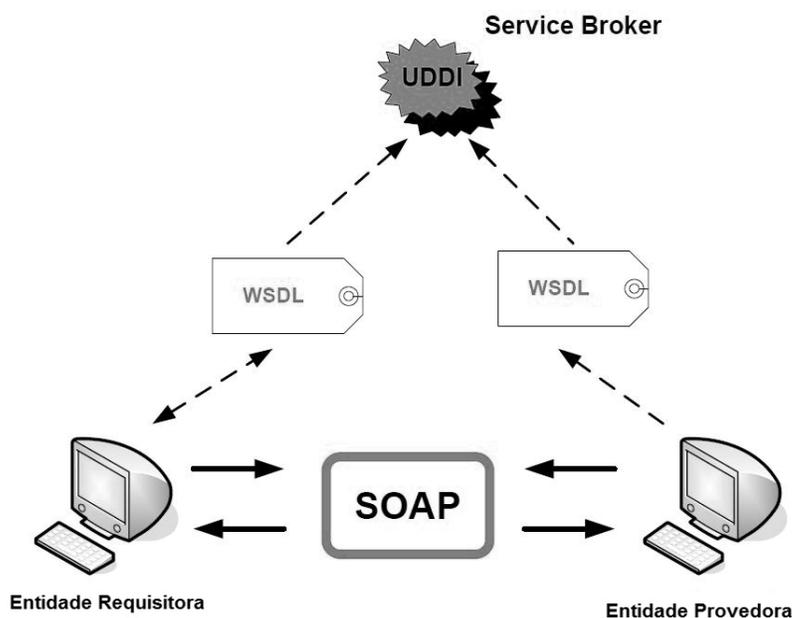


Figura 3.13 Interação entre as tecnologias Web Services [MEDYK, 2006]

3.5 Integração e Operações entre as Entidades

A arquitetura de serviços Web, de forma geral, é baseada na integração entre três elementos ou entidades: *Service Provider* (Provedor de Serviços), *Service Requestor* (Cliente do Serviço) e *Service Registry* (Registro de Serviço). A figura 3.14 ilustra as operações entre as entidades citadas.

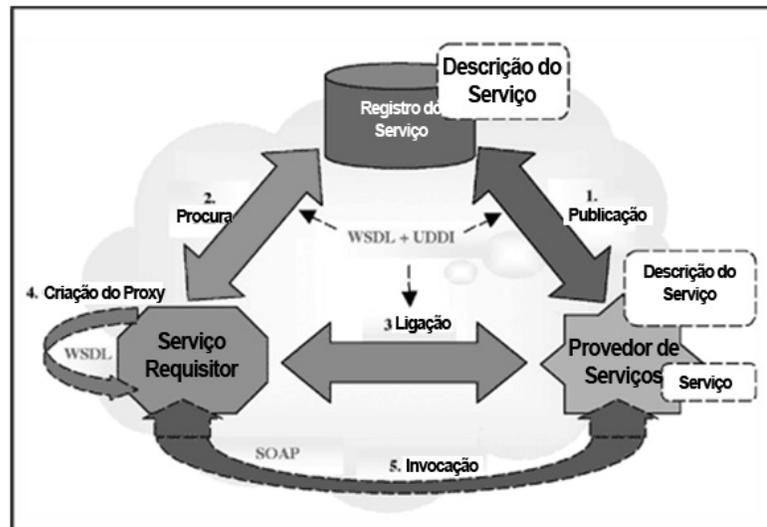


Figura 3.14 Operações entre as entidades relacionadas [W3C, 2009]

As principais operações ilustradas na figura acima são: *Publish* (Publicação), *Find* (Busca) e *Bind* (Vinculação). De uma maneira geral o provedor do serviço (*Service Provider*) hospeda o Serviço Web, tornando-o disponível para os clientes (*Service Requestor*) acessarem. Os clientes são quaisquer aplicações que queiram iniciar uma interação com os serviços Web (pode ser também um acesso direto através de um browser). Já os *Services Registries* são os servidores de registro e busca dos serviços através de arquivos de descrição dos mesmos (especificados de acordo com a linguagem WSDL e registrados nos *UDDI Registries*) que foram publicados pelos *Services Providers*. Dessa forma os clientes buscam por serviços nos servidores de registro e recuperam as informações necessárias para a implementação do mesmo.

Dessa forma, a Arquitetura de Serviços Web surgiu para permitir a interoperabilidade entre diferentes tipos de aplicação mesmo rodando em plataformas e sistemas operacionais diferentes, visto que as abordagens já existentes – DCOM, CORBA e RMI deixaram a desejar no que se refere à independência de plataforma.

Como apresentado, o processo geral de um serviço web é dado a partir da interação entre duas entidades – *Requester* (consumidora do serviço) e *Provider* (fornecedora do serviço). A troca de mensagens se dá somente após a consumação de acordos da descrição e da semântica do serviço.

Foram apresentados os quatro modelos definidos pela especificação da Arquitetura de

Serviços Web: Modelo Orientado a Mensagens, Modelo Orientado a Serviço, Modelo Orientado a Recursos e Modelo de Policiamento. Cada modelo foca em um aspecto da arquitetura e é gerado a partir de um ponto de vista diferente. Um desses pontos de vista – o do Modelo Orientado a Recursos – provém de um conceito comum com o paradigma REST – o recurso.

As tecnologias relacionadas aos serviços web – XML, SOAP, WSDL e UDDI – descritas representam a forma tradicional como são implementados os serviços web e será comparada com o paradigma REST mais adiante.

4 Arquitetura REST

REST (Representation State Transfer) é um estilo arquitetural híbrido para sistemas distribuídos derivado da combinação de alguns estilos arquiteturais (descritos abaixo) com algumas características adicionais. Foi criado a partir de um capítulo da tese de doutorado de Roy Fielding no ano de 2000 [FIELDING, 2000].

4.1 Fundamentação Básica

Esta seção irá fornecer uma visão geral de REST, através do processo de derivação do mesmo como um estilo arquitetural.

4.1.1 Null Style

Uma das perspectivas do processo de projeto de arquiteturas parte do princípio de que o mesmo começa tomando as necessidades do sistema como um todo, sem características e então, de forma incremental elas vão sendo identificadas e aplicadas aos elementos do sistema a fim de diferenciar o espaço de projeto e permitir que o comportamento do sistema flua naturalmente em harmonia com o todo.

Esta perspectiva é conhecida como *Null Style* (Figura 4.1). Basicamente, representa um conjunto simples e vazio de características. A partir de uma visão arquitetural, descreve um sistema o qual não distingue limites entre componentes. Este é o ponto inicial da descrição de REST por Fielding em sua tese de doutorado [FIELDING, 2000].



Figura 4.1 *Null Style* [FIELDING, 2000]

4.1.2 Cliente-servidor

A primeira característica adicionada ao *Null Style* deriva do estilo arquitetural cliente-servidor (Figura 4.2).

O princípio por trás da arquitetura cliente-servidor é o de *separação de responsabilidades*. Ao separar as responsabilidades de interface das de armazenamento de dados, a utilização da arquitetura cliente-servidor permite que os componentes, o cliente e o servidor, se desenvolvam de forma independente. A evolução independente de cada parte torna a arquitetura apta a suportar os requisitos de aplicação de múltiplos domínios organizacionais na escala da Internet.

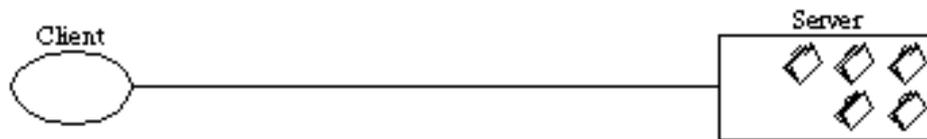


Figura 4.2 Estilo Arquitetural Cliente-Servidor [FIELDING, 2000]

4.1.3 Sem Estado

A próxima característica a ser adicionada à interação cliente-servidor é a baseada no princípio de que as comunicações devem ser *stateless*, ou em português, sem estado (Figura 4.3).

Em REST, as comunicações não devem depender de estados controlados pelo servidor. Toda informação de estado deve ser conhecida somente pelo cliente. Dessa forma, todas as requisições feitas pelo cliente deverão sempre possuir todos os atributos necessários para que ela possa ser processada sem que o mesmo se aproveite de informações adicionais no contexto de comunicação.

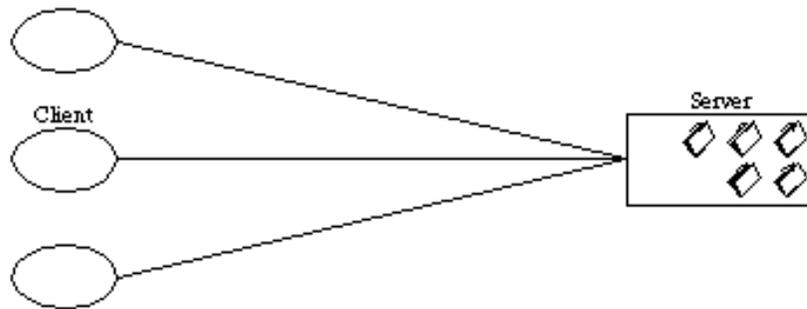


Figura 4.3 Estilo Arquitetural Cliente-servidor Sem Estado [FIELDING, 2000]

4.1.4 Cache

Combinando-se os estilos Cliente-Servidor, Sem Estado e Cache chega-se ao *Cliente com Cache-Servidor Sem Estado*.

A grande vantagem deste estilo sobre o anterior é de poder, parcialmente ou até totalmente, eliminar algumas interações entre cliente e servidor, sobrecarregando menos o servidor. Com o uso do cacheamento, melhora-se o desempenho geral do sistema, pois diminui o tempo de resposta da aplicação [TOBALDINI, 2008].

Para melhorar a eficiência da comunicação entre o cliente e o servidor, adiciona-se a REST a característica de cache, o que torna necessária a marcação explícita de uma resposta no âmbito de esta ser ou não passível de cache. Em caso afirmativo, o cliente tem o direito de reusar esta resposta futuramente, podendo se aproveitar das vantagens oferecidas pelo cacheamento.

4.1.5 Interface Uniforme

A característica principal que diferencia REST de outras arquiteturas baseadas em rede é a sua ênfase em uma interface uniforme entre componentes [FIELDING, 2000].

Aplicando o princípio de generalização de engenharia de software à interface dos componentes, a arquitetura é simplificada e a visibilidade melhorada. O problema dessa uniformização é que toda a troca de informação é feita de forma genérica ao invés de ser específica para cada necessidade [TOBALDINI, 2008].

Para obter uma interface uniforme, várias características arquiteturais são necessárias para guiar o comportamento dos componentes.

REST é definido através de quatro elementos de interface: Identificação de Recursos; Manipulação de Recursos através de Representações; Mensagens Auto-descritivas e Hiperlinks como Máquina de Estados da Aplicação. Estes elementos serão detalhados na seção 4.2.

4.1.6 Separação em Camadas

Um sistema em camadas permite a uma arquitetura ser composta por diversos níveis hierárquicos, restringindo os componentes participantes de tal forma que nenhum deles possa enxergar através da camada adjacente, ou seja, o conhecimento do sistema que o componente possui é restrito à sua camada apenas [FIELDING, 2000].

A grande desvantagem do uso de sistemas em camadas é o maior atraso na chegada da informação de um extremo ao outro, por conta das camadas intermediárias. Para um sistema que suporta as características de cache essa desvantagem pode ser contornada através do compartilhamento de cache entre as camadas permitindo que uma requisição que já foi feita anteriormente possa ser utilizada por uma camada intermediária ao invés de percorrer todo o caminho até o destino final [TOBALDINI, 2008].

Ao estilo REST é adicionada também a possibilidade de separação em camadas. Seu uso provê uma maior flexibilidade e simplicidade na comunicação entre sistemas diversos.

Embora as interações entre cliente e servidor ocorram sempre nos dois sentidos, cada troca de dados pode ser tratada como um fluxo independente. Isto é possível devido ao fato de que as mensagens REST são sempre auto-descritivas e seus significados são sempre visíveis a todos os seus intermediários.

4.1.7 Código Sob Demanda

A utilização de código sob demanda permite que as funcionalidades do cliente sejam inseridas em tempo de execução através da descarga e execução de código em forma de *scripts*. Esta característica pode facilitar a extensão do sistema, porém reduz a visibilidade.

O recurso de código sob demanda em REST tem o objetivo de simplificar e ampliar as capacidades dos clientes. Porém o seu uso é opcional, pois acarreta em redução de visibilidade do sistema. [FIELDING, 2000]

Dessa forma, REST foi gerado a partir de um *null style* em composição com características de outros estilos arquiteturais. Embora cada uma dessas características possa ser considerada de forma isolada, descrevê-las em termos de suas derivações de estilos arquiteturais torna mais fácil entender a razão por trás destas escolhas.

4.2 Elementos Arquiteturais REST

REST é uma abstração de elementos arquiteturais em sistemas distribuídos. O estilo ignora detalhes de implementação de componentes e sintaxe de protocolos para focar nos papéis dos componentes, suas interações e interpretação de elementos de dados [TOBALDINI, 2008].

A seguir, os elementos arquiteturas de REST serão detalhados.

4.2.1 Elementos de Dados

A natureza e o estado dos elementos de dados de uma arquitetura são os aspectos-chave de REST. Quando um link é selecionado – na *Web*, por exemplo – a informação precisa ser movida do local onde ela será processada. Esta característica difere de outros paradigmas de processamento distribuído, onde é possível – e geralmente mais eficiente – levar o “processamento” até os dados ao invés de levar os dados até o processamento.

No REST, os componentes se comunicam através da transferência de representações de um recurso em um formato dentre um conjunto de tipos de dados padrão selecionado dinamicamente baseado nas capacidades ou desejos do componente que irá receber as informações e na natureza do recurso. Sejam estas representações no mesmo formato que os dados ou derivadas deste, a comunicação se dá de forma transparente. Portanto, REST se beneficia da separação de responsabilidades do estilo cliente servidor sem o problema da escalabilidade [FIELDING, 2000].

A figura 4.4 ilustra os elementos de dados do estilo REST e seus respectivos exemplos.

Elementos de Dados	Exemplos Modernos na Web
Recurso	Conceito alvo pretendido para uma referência de hipertexto
Identificador de Recurso	URL, URN
Representação	Documento HTML, Imagem JPEG
Representação de Metadados	Tipos de mídia, data da última modificação
Recursos de Metadados	Links, permutações, variações
Dados de Controle	<i>If-modified-since</i> , controle de cache

Figura 4.4 Elementos de Dados REST [FIELDING, 2000]

4.2.1.1 Recursos e Identificadores de Recursos

A abstração chave de informação no estilo REST é um *recurso*. Qualquer informação que possa receber um nome pode ser um recurso: um documento, uma imagem, um serviço, uma coleção de outros recursos, e assim por diante. Em outras palavras qualquer conceito que possa ser alvo de uma referência deve se encaixar na definição de recurso [TOBALDINI, 2008].

REST usa um *identificador de recurso* para identificar um recurso específico envolvido em uma interação entre os componentes. Um exemplo de um identificador de recursos é o URI (*Unified Resource Identifier*).

4.2.1.2 Representação

Componentes REST executam ações sob um recurso utilizando uma *representação* para capturar o estado atual e o estado desejado de um recurso solicitado e transferir a representação entre os componentes.

Uma representação, em REST, é uma seqüência de bytes acrescida dos meta-dados dessa representação que descrevem estes bytes. Uma representação pode ser, por exemplo, um documento, um arquivo, uma mensagem HTTP, entre outros. Mais especificamente uma

representação consiste em dados, meta-dados para descrever os dados e, ocasionalmente, outro conjunto de meta-dados que descrevem meta-dados (utilizados para verificação da integridade da mensagem).

Os meta-dados são apresentados na forma de pares nome-valor, onde o nome corresponde a um padrão que define a estrutura e a semântica do valor. As mensagens de resposta em REST incluem tanto os meta-dados da representação quanto os meta-dados do recurso (informações sobre o recurso que não são específicas à representação específica) [FIELDING, 2000].

Outro elemento participante de uma representação é a informação de controle (*control data*). Informações de controle definem o propósito de uma mensagem entre os componentes, assim como a natureza da ação solicitada ou a finalidade de uma resposta. Elas são usadas para parametrizar requisições e redefinir o comportamento padrão de alguns elementos. Por exemplo, o comportamento de um cache pode ser modificado através de informações de controle embutidas em uma requisição ou resposta.

O formato dos dados de uma representação é conhecido como *media-type* (ou em português, tipo de informação). Uma representação pode ser incluída em uma mensagem e processada de acordo com as informações de controle da mensagem e a natureza do tipo de informação.

4.2.2 Conectores

REST utiliza vários tipos de conectores (resumidos na figura 4.5) para encapsular a atividade de acesso a recursos e para transferir representações dos mesmos. Os conectores provêm uma interface abstrata para a comunicação entre componentes, melhorando a simplicidade do sistema através da separação das responsabilidades e ocultando a implementação de recursos e mecanismos de comunicação [FIELDING, 2000].

Conector	Exemplos Modernos na Web
Cliente	libwww, libwww-perl
Servidor	libwww, Apache API, NSAPI
Cache	Cache do Browser, Akamai cache network
Resolvedor	Bind (DNS lookup library)
Túnel	SOCKS, SSL after HTTP CONNECT

Figura 4.5 Conectores REST [FIELDING, 2000]

Todas as interações REST são, por natureza, sem estado. Isto é, cada requisição contém toda a informação necessária para o conector entender a requisição, independente de qualquer requisição que tiver precedido a mesma.

A interface do conector é semelhante à invocação procedural, porém com diferenças em relação à passagem de parâmetros e resultados. Os parâmetros de entrada consistem em informações de controle da requisição acrescidas de um identificador de recurso e, opcionalmente, uma representação. Os parâmetros de saída são compostos pela informação de controle da resposta e opcionalmente meta-dados do recurso ou uma representação [TOBALDINI, 2008].

Os dois primeiros tipos de conectores da figura – *cliente* e *servidor* – são os conectores primários. A diferença básica entre eles é que cabe ao cliente inicializar a comunicação a partir de uma requisição, enquanto o servidor deve aguardar passivamente e atender às requisições de forma a fornecer acesso aos seus serviços. Um componente pode incluir ambos os conectores [FIELDING, 2000].

O conector *cache* é encontrado entre o cliente e o servidor e tem a função de armazenar as respostas passíveis de cache para que elas possam ser reutilizadas em requisições futuras. Um cache também pode ser utilizado no cliente para evitar repetições de comunicações de rede, ou pelo servidor para evitar a repetição no processo de geração de respostas. Em ambos os casos, o cache contribui para diminuir o tempo de resposta – latência – da aplicação [TOBALDINI, 2008].

A resolução de nomes (*resolver*) traduz, parcial ou completamente, identificadores de recurso (um URI, por exemplo) em endereços de rede. Essa conversão torna possível estabelecer uma conexão entre componentes.

Finalmente, a última “peça” dos conectores de REST é o *túnel*. Sua função é de criar um caminho virtual para o tráfego de dados de forma a transpor limites, como um *firewall* ou um roteador.

A única razão para incluir o túnel como parte da arquitetura REST e não abstrair como parte da infra-estrutura de rede é que alguns componentes de REST podem trocar dinamicamente de um comportamento ativo para um comportamento de túnel. O exemplo desse cenário é o *Proxy HTTP* que, dependendo do tipo de requisição que ele intermedia, deixa suas funções de *proxy* e passa a se comportar como um túnel para permitir a comunicação direta (sem passagem pelo *proxy*). Os túneis são destruídos após o término de uma comunicação entre componentes [TOBALDINI, 2008].

4.2.3 Componentes

Os componentes REST são categorizados de acordo com o papel que eles exercem no âmbito geral de uma aplicação, de acordo com a figura 4.6.

Componente	Exemplos Modernos na Web
Servidor de Origem	Apache httpd, Microsoft IIS
<i>Gateway</i>	Squid, CGI, Proxy Reverso
<i>Proxy</i>	CERN Proxy, Netscape Proxy, Gauntlet
Agente do Usuário	Netscape Navigator, Lynx, MOMspider

Figura 4.6 Componentes REST [FIELDING, 2000]

O *agente do usuário* utiliza um conector do tipo cliente para iniciar uma requisição e, em seguida, se torna o destino final de uma resposta.

O tipo mais comum de um agente do usuário é o navegador web que provê acesso à informação (recursos) e processa as respostas (representações) de forma a ser visualizada pelo usuário de acordo com a necessidade [TOBALDINI, 2008].

Um *servidor de origem* utiliza um conector do tipo servidor e trata das requisições de um cliente. Cada servidor deve prover seus serviços através de uma interface genérica para a sua

hierarquia de recursos. Os detalhes de implementação destes recursos ficam escondidos por esta interface [FIELDING, 2000].

Os outros dois componentes – *gateway* e *proxy* – atuam simultaneamente como cliente e servidor para permitir o encaminhamento de requisições e respostas. O *proxy* é um componente intermediário escolhido pelo cliente que torna possível o encapsulamento de outros serviços, tradução de dados, melhoria de *performance* ou algum tipo de controle de acesso de segurança. Já o *gateway* funciona como um *proxy* reverso. Ele fornece os mesmos serviços que um *proxy*, porém o *gateway* é transparente ao usuário e utilizado pelo servidor de origem, enquanto o *proxy* é utilizado explicitamente pelo usuário.

4.2 Visões Arquiteturais REST

Como mencionado no capítulo 2, as visões de uma arquitetura são utilizadas para descrever como os elementos arquiteturais trabalham em conjunto para formar uma arquitetura. Serão abordados três tipos de visões dentro do conceito arquitetural REST – Visões de Processo, Visões de Conectores e Visão de Dados.

4.2.1 Visões de Processo

Uma visão de processo de uma arquitetura é particularmente útil para extrair as relações e interações entre os componentes, revelando o caminho dos dados como um fluxo através do sistema [FIELDING, 2000]. Porém, a interação de componentes de um sistema real geralmente envolve muitos componentes, resultando em uma visão geral complexa.

A separação de responsabilidades entre cliente e servidor da arquitetura REST torna mais simples a implementação de componentes, reduz a complexidade da semântica dos conectores, melhora o desempenho e aumenta a escalabilidade de componentes. As características de sistemas em camadas permitem que intermediários – *proxies*, *gateways* e *firewalls* – possam ser inseridos em vários pontos da comunicação sem alterar a interface entre os componentes, o que permite que eles participem da comunicação e melhora do desempenho através do uso de cache.

Serviços podem ser implementados utilizando uma hierarquia de intermediários e

servidores de origem distribuídos. A característica natural de REST – Sem-Estado – permite de cada interação seja independente das outras e permite que componentes desempenhem um papel determinado dinamicamente de acordo com o foco de cada requisição.

Os conectores devem ter conhecimento da existência de outros conectores somente durante o escopo da comunicação, embora eles possam usar o cache para armazenar informações de outros componentes por razões de desempenho.

4.2.2 Visões de Conectores

A visão dos conectores de uma arquitetura foca nos mecanismos de comunicação entre os componentes. Para a arquitetura REST, interessam somente as características que definem uma interface de recursos genérica.

Conectores do tipo cliente consideram a identificação de recurso para escolher o mecanismo de comunicação mais conveniente para cada requisição. Por exemplo, um cliente pode ser configurado para se conectar à um determinado componente de *proxy* somente quando o identificador indicar que ele é um recurso local. Da mesma forma, um cliente também pode ser configurado para rejeitar requisições de um determinado subconjunto de identificadores [FIELDING, 2000].

4.2.3 Visões de Dados

A visão de dados de uma arquitetura revela o estado de uma aplicação como um fluxo de informação através dos componentes. Devido ao foco da arquitetura REST ser em sistemas distribuídos, sua visão de dados é uma estrutura coesa de informação e alternativas de controle pelas quais um usuário pode realizar as tarefas desejadas.

Os estados da aplicação são controlados e armazenados por um agente usuário e podem ser decompostos em representações de vários servidores. O usuário pode manipular diretamente o estado (por exemplo, o histórico de um *Web Browser*), antecipar mudanças à um estado e pular de uma aplicação para outra (por exemplo, *bookmarks*).

4.3 Arquitetura Orientada a Recursos

O estilo arquitetural REST define somente um conjunto de características que caracterizam uma aplicação dita *RESTful*, de forma muito abstrata e geral. Esta seção introduz a Arquitetura Orientada a Recursos, uma aplicação prática dos conceitos arquiteturais REST.

A Arquitetura Orientada a Recursos torna possível transformar um problema em uma aplicação REST através do emprego de URI's, HTTP e XML. Ela é composta por recursos, seus nomes, suas representações, a ligação entre eles e algumas propriedades: Endereçamento, Sem-Estado, Conectividade e Interface Uniforme [RICHARDSON; RUBY, 2007].

A seguir os conceitos acerca da arquitetura serão contextualizados e exemplificados.

4.3.1 Recursos

Um recurso é algo que possa ser armazenado em um computador e representado como uma seqüência de bits, como por exemplo, um documento ou uma imagem. Um recurso pode ser um objeto físico, como por exemplo, uma maçã, ou até mesmo um conceito abstrato, como “coragem”.

São exemplos de recursos:

- A última versão de uma distribuição de um software;
- Um número primo qualquer;
- Uma lista de *bugs* de um banco de dados;
- A relação entre dois conhecidos, Alice e Bob.

Para ser acessado, o recurso deve ser associado a um nome e um lugar onde ele possa ser encontrado. Para isto é utilizado um identificador de recurso, dessa forma, um recurso precisa ter ao menos um identificador para que se torne acessível [TOBALDINI, 2008].

REST, enquanto um estilo arquitetural, utiliza um identificador de recurso. No caso da arquitetura orientada a recursos, os identificadores de recurso utilizados são as URI's (*Unified Resource Identifier*).

As URI's devem ser utilizadas de forma estruturada, intuitiva e uniforme

[RICHARDSON; RUBY, 2007]. Um exemplo de prática não aconselhável desse cenário é a utilização das URI's `/informacoessobre/rest` e `/arespeitode/rest` para se obter informações sobre REST. O objetivo destes dois recursos é o mesmo: representar informações sobre a arquitetura REST, porém suas URI's são compostas de forma não-uniforme.

O que torna essa prática não aconselhável é o fato de não haver uma forma automática de verificar que várias URI's se referem ao mesmo recurso, embora esse problema possa ser contornado através do uso de um URI principal e sua indicação como tal na resposta para URI's secundárias.

4.3.2 Endereçamento

Uma aplicação é endereçável se ela expõe aspectos interessantes sobre seu conjunto de dados como recursos. Como URI's são atribuídas a recursos, uma aplicação é endereçável quando ela expõe pelo menos um URI para cada informação que ela deseja servir [RICHARDSON; RUBY, 2007].

Um exemplo que ilustra a propriedade de endereçamento é o protocolo HTTP. O sítio `http://www.google.com.br` pode ser tomado como exemplo de aplicação HTTP endereçável. Quando a busca de uma expressão é realizada, a mesma é traduzida em um URI que possui todas as informações para que a mesma possa ser repetida e obter a mesma resposta em outra ocasião. Além disso, este URI poderá ser repassado para outra pessoa que necessitar obter a mesma pesquisa [TOBALDINI, 2008].

Se esta aplicação não fosse endereçável o recurso mencionado não seria possível. Sempre que uma pesquisa fosse necessária, seria preciso entrar no sítio e preencher o campo de busca e solicitar a pesquisa.

4.3.3 Sem-Estado

A ausência de estados (*statelessness*) significa que cada requisição feita pelo cliente ocorre de maneira isolada no servidor, isto significa que não há lembrança de estados entre uma requisição e outra. Quando uma requisição é feita, ela deverá conter todas as informações necessárias para que seja completada pelo servidor, que nunca irá se utilizar de informações de requisições anteriores [RICHARDSON; RUBY, 2007].

Pode-se considerar a ausência de estados em termos de endereçamento. O endereçamento prega que cada informação interessante deve ser exposta na forma de um recurso e ter seu próprio URI. No caso da ausência de estados, de forma análoga ao endereçamento, cada estado possível deve ser representado na forma de um recurso e ter seu próprio URI.

Uma forma de quebrar a ausência de estados no servidor é utilizando o conceito de *sessões*. Quando um cliente faz sua primeira requisição ao servidor, lhe é atribuído um identificador de sessão – que pode ser um número ou uma cadeia de caracteres. Este identificador pode ser propagado como um componente do URI ou ser associado a um cliente através do uso de *cookies*. Cookies são informações enviadas pelo servidor e que permanecem sem modificações no cliente. A cada vez que uma requisição é feita pelo cliente, os cookies são enviados ao servidor [TOBALDINI, 2008].

4.3.4 Representações

Uma aplicação é composta por diversos recursos. Cada recurso criado deve transmitir alguma “idéia” ao usuário, como por exemplo, “informações sobre REST”. Porém, um servidor não pode enviar uma idéia ao usuário, mas sim uma seqüência de bytes em um formato específico para que possa ter utilidade ao cliente. Isto é uma representação de um recurso [RICHARDSON; RUBY, 2007].

Um recurso é a origem de uma representação e as representações são basicamente alguns dados relevantes sobre o estado atual de um recurso. Alguns recursos são, por natureza, dados. Um exemplo desse tipo de recurso é uma lista de *bugs*, que tem como representação seus próprios dados – os itens da lista. Esta lista pode ser representada por um documento XML, uma página HTML ou até mesmo um documento de texto puro [RICHARDSON; RUBY, 2007].

Porém, alguns tipos de recursos representam objetos físicos ou outro elemento que não possa ser naturalmente representado através de dados, como por exemplo, uma geladeira. Considerando o cenário de uma geladeira, seria impossível obter uma geladeira através do acesso a este recurso. Porém, o objeto geladeira possui informações úteis a seu respeito – seus metadados – como, por exemplo, sua temperatura interna, a posição do termostato que a regula, entre outros.

Como já mencionado, um recurso pode ter mais de uma representação. Torna-se necessário então escolher qual representação deve ser obtida. Este problema pode ser resolvido

de várias maneiras dentro das restrições impostas pela arquitetura REST. Porém, a Arquitetura Orientada a Recursos recomenda apenas uma maneira de solucionar este problema: utilizando URI's distintas para cada tipo de representação. Esta abordagem torna possível que cada URI possua todas as informações acerca do recurso, incluindo o tipo de representação desejada. Por outro lado, isto causa “diluição” de um mesmo recurso em várias URI's [TOBALDINI, 2008].

4.3.5 Ligações e Conectividade

Uma representação de um recurso pode conter, além dos dados sobre ele, ligações para outros recursos. Tomando como exemplo, novamente, uma pesquisa de informações sobre REST em um sítio de pesquisa, como o Google, a representação desta busca seria uma página HTML contendo várias ligações para outros recursos que contenham algum tipo de informações relacionadas à REST. Dessa forma, o servidor guia o usuário às informações através das ligações [TOBALDINI, 2008].

Da forma tradicional como a *Web* é usada, ela possui um alto grau de conectividade. Qualquer usuário experiente pode inserir diretamente um URI em um *Browser* e navegar através de sítios mudando este URI. Também é possível acessar estas URI's através de um único ponto de partida – como um sítio de busca – e navegar através das ligações [RICHARDSON; RUBY, 2007].

4.3.6 Interface Uniforme

Como já mencionado, a arquitetura REST especifica uma interface uniforme, porém de forma genérica. A arquitetura orientada a recursos utiliza os conceitos de uma interface uniforme proveniente do protocolo HTTP [TOBALDINI, 2008].

O protocolo HTTP define oito métodos, dos quais seis são utilizados na arquitetura em questão, sumarizados na figura 4.7.

MÉTODO	FUNÇÃO
GET	Obter uma representação de um recurso
PUT	Criar um novo recurso / Modificar um já existente
POST	Criar um novo recurso
DELETE	Remover um recurso existente
HEAD	Obter apenas a representação pertinente ao meta-dados de um recurso
OPTIONS	Obter uma relação dos métodos suportados por um determinado recurso.

Figura 4.7 Métodos do protocolo HTTP utilizados pela Arquitetura Orientada a Recursos [TOBALDINI, 2008].

4.4 Serviços Web segundo o modelo REST

De acordo com a visão tradicional para os Serviços Web, a fase de interação é implementada através de troca de mensagens SOAP. A figura 4.8 ilustra um exemplo de invocação de procedimento e sua respectiva resposta ao pedido, baseados na visão tradicional dos Serviços Web (baseada em troca de mensagens SOAP) [NUNES; DAVID, 2009]. O exemplo representa uma visão simplificada de um serviço que disponibiliza informações de alunos.



Figura 4.8 Invocação e resposta de um método pela visão tradicional de Serviços Web [NUNES; DAVID, 2009]

Segundo a abordagem convencional do desenvolvimento de Serviços Web, os serviços estão escondidos por trás de métodos que podem ser invocados.

A arquitetura orientada a recursos se tornou uma alternativa para a construção de Serviços Web tendo por base os pressupostos estabelecidos com o modelo REST. Em um Serviço Web compatível com a arquitetura REST, os recursos devem ser expostos e ter uma identificação global. Ainda no cenário utilizado como exemplo, na visão REST, os recursos correspondem a documentos XML com a informação sobre os alunos. Cada URI identifica de forma única a informação sobre um estudante.

Depois de expostos os recursos, a interação é realizada utilizando as operações genéricas do protocolo HTTP. Dessa forma, para obter a representação da informação relativa a um estudante, utiliza-se o método GET sobre o URI do recurso. A figura 4.9 ilustra o pedido e a resposta de uma requisição baseada nos princípios REST [NUNES; DAVID, 2009].

```
GET http://univ.example.com/studentInfo/123456789
```



```
<?xml version="1.0" ?>  
<u:StudentInfo xmlns:u="http://univ.example.com">  
  <u:code>123456789</u:code>  
  <u:name>João Silva</u:name>  
  <u:age>19</u:age>  
</u:StudentInfo>
```

Figura 4.9 Invocação e resposta de um método pela visão REST de Serviços Web [NUNES; DAVID, 2009]

5 Exemplos de Aplicação

Para contextualizar a apresentação dos conceitos das duas abordagens de Serviços Web apresentadas nos capítulos anteriores – visão tradicional e visão REST, um exemplo de serviço em Java será ilustrado por ambas as implementações.

A IDE utilizada nos exemplos foi o Eclipse versão *Galileo* (<http://www.eclipse.org/galileo/>) e o contêiner web utilizado foi o Apache Tomcat, (<http://tomcat.apache.org/>) em sua versão 6.0.

O serviço oferecido consiste em um acervo simplificado de filmes e os métodos oferecidos são os de cadastro e busca dos mesmos. Por questões de simplicidade, os dados dos exemplos não são persistidos e as camadas de acesso aos dados (padrão DAO) e de serviço (padrão Service) foram omitidas. Além disso, os métodos *get* e *set* de atributos foram omitidos, como também declarações do tipo `import`.

5.1 Exemplo de abordagem tradicional de Serviços Web

Atualmente, a plataforma Java possui uma gama de componentes que permitem implementar serviços pela abordagem tradicional sem precisar manipular diretamente SOAP e WSDL. As principais opções são os servidores de aplicação Java EE completos ou frameworks independentes, como o Apache Axis (<http://ws.apache.org/axis2/>). Pela sua maturidade; por abstrair o uso do protocolo SOAP e ser compatível com WSDL, o Apache Axis 2 foi escolhido para o desenvolvimento do exemplo.

A classe `Filme.java`, ilustrada na figura 5.1 representa a classe de modelo do exemplo. Os dados do filme, neste contexto, representa o serviço que será oferecido por uma entidade *Provider* – por exemplo, um servidor que armazena o acervo de filmes – e consumido por um determinado usuário – que representa a entidade *Requester*.

```

/**
 *
 * @author Livia
 *
 */
public class Filme {

    private int codFilme;
    private String titulo;
    private int ano;
    private String diretor;
    private String genero;

    public Filme(int codFilme, String titulo, int ano, String diretor, String genero) {
        super();
        this.codFilme = codFilme;
        this.titulo = titulo;
        this.ano = ano;
        this.diretor = diretor;
        this.genero = genero;
    }
}

```

Figura 5.1 Classe Filme.java

A classe de serviço que contém os métodos a serem invocados remotamente e que gerou o WSDL é ilustrada na figura 5.2.

```

/**
 *
 * @author Livia
 *
 */
public class FilmeWS {
    FilmeService filmeService = new FilmeService();

    // métodos para acesso aos filmes

    public Filme cadastrarFilme(@WebParam Filme filme){
        return filmeService.cadastrar(filme);
    }

    public Filme buscarFilme(@WebParam Filme filme){
        return filmeService.cadastrar(filme);
    }
}

```

Figura 5.2 Classe FilmeWS.java

O arquivo WSDL ilustrado na figura 5.3 foi criado de forma automática pelo Eclipse, através da opção Create Web Service. O WSDL representa a descrição do serviço e disponibiliza todos os métodos oferecidos pelo mesmo e os parâmetros de entrada e saída.

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://filme" xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://filme"
xmlns:intf="http://filme" xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/" xmlns:wSDLsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
<wsdl:types>
<schema elementFormDefault="qualified" targetNamespace="http://filme" xmlns="http://www.w3.org/2001/XMLSchema">
<element name="cadastrarFilme">
<complexType><sequence><element name="filme" type="impl:Filme"/></sequence></complexType>
</element>
<complexType name="Filme">
<sequence>
<element name="ano" type="xsd:int"/>
<element name="codFilme" type="xsd:int"/>
<element name="diretor" nillable="true" type="xsd:string"/>
<element name="genero" nillable="true" type="xsd:string"/>
<element name="titulo" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<element name="cadastrarFilmeResponse">
<complexType><sequence><element name="cadastrarFilmeReturn" type="impl:Filme"/></sequence></complexType>
</element>
<element name="buscarFilme">
<complexType><sequence><element name="filme" type="impl:Filme"/></sequence></complexType>
</element>
<element name="buscarFilmeResponse">
<complexType><sequence><element name="buscarFilmeReturn" type="impl:Filme"/></sequence></complexType>
</element>
</schema>
</wsdl:types>

<wsdl:message name="buscarFilmeRequest"><wsdl:part element="impl:buscarFilme" name="parameters"/></wsdl:message>
<wsdl:message name="cadastrarFilmeRequest"><wsdl:part element="impl:cadastrarFilme" name="parameters"/></wsdl:message>
<wsdl:message name="buscarFilmeResponse"><wsdl:part element="impl:buscarFilmeResponse" name="parameters"/></wsdl:message>
<wsdl:message name="cadastrarFilmeResponse"><wsdl:part element="impl:cadastrarFilmeResponse" name="parameters"/></wsdl:message>
<wsdl:portType name="FilmeWS">
<wsdl:operation name="cadastrarFilme">
<wsdl:input message="impl:cadastrarFilmeRequest" name="cadastrarFilmeRequest"/>
<wsdl:output message="impl:cadastrarFilmeResponse" name="cadastrarFilmeResponse"/>
</wsdl:operation>
<wsdl:operation name="buscarFilme">
<wsdl:input message="impl:buscarFilmeRequest" name="buscarFilmeRequest"/>
<wsdl:output message="impl:buscarFilmeResponse" name="buscarFilmeResponse"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="FilmeWSSoapBinding" type="impl:FilmeWS">
<wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="cadastrarFilme"><wsdlsoap:operation soapAction=""/>
<wsdl:input name="cadastrarFilmeRequest"><wsdlsoap:body use="literal"/></wsdl:input>
<wsdl:output name="cadastrarFilmeResponse"><wsdlsoap:body use="literal"/></wsdl:output>
</wsdl:operation>

<wsdl:operation name="buscarFilme"><wsdlsoap:operation soapAction=""/>
<wsdl:input name="buscarFilmeRequest"><wsdlsoap:body use="literal"/></wsdl:input>
<wsdl:output name="buscarFilmeResponse"><wsdlsoap:body use="literal"/></wsdl:output>
</wsdl:operation>
</wsdl:binding>

```

Figura 5.3 Arquivo FilmeWS.wsdl

O plugin do Eclipse permite também que se gere o cliente de um serviço a partir de um documento WSDL. A ferramenta gera automaticamente as classes cliente, entre elas a Proxy, cuja instância permite a invocação dos métodos do serviço, como descrito na classe de teste FilmeMain.java, ilustrada na figura 5.4.

```

/**
 *
 * @author Livia
 *
 */
public class FilmeMain {
    public static void main(String[] args) {
        try {
            // Criação de um Filme de maneira tradicional
            Filme filme = new Filme();
            filme.setCodFilme(4);
            filme.setTitulo("Sete Vidas");
            filme.setAno(2009);
            filme.setDiretor("Gabriele Muccino");
            filme.setGenero("Drama");

            // Instância de um objeto da classe Proxy
            FilmeWSProxy servico = new FilmeWSProxy();

            // Invocação do método de cadastro
            servico.cadastrarFilme(filme);
            System.out.println("Filme " + filme.getTitulo() + " cadastrado com sucesso!");

            // Invocação do método de busca
            Filme filmeFound = new Filme();
            filmeFound = servico.buscarFilme(filme);
            System.out.println("\nBuscando dados do filme " + filmeFound.getTitulo()+ " \n");
            System.out.println("- Diretor: " + filmeFound.getDiretor());
            System.out.println("- Ano: " + filmeFound.getAno());
            System.out.println("- Gênero: " + filmeFound.getGenero());

        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}

```

Figura 5.4 Classe FilmeMain.java

A execução da classe FilmeMain.java gera a saída ilustrada na figura 5.5.

```

<terminated> FilmeMain (2) [Java Application] C:\Arquivos de programas\Java\jdk1.6.0_13\bin\javaw.exe (27/06/2009 18:09:09)
Filme Sete Vidas cadastrado com sucesso!

Buscando dados do filme Sete Vidas

- Diretor: Gabriele Muccino
- Ano: 2009
- Gênero: Drama

```

Figura 5.5 Saída da execução da classe FilmeMain.java

5.2 Exemplo de Serviços Web RESTful

Para oferecer suporte a serviços REST em Java, foi criada a JAX-RS, uma API que busca simplificar o desenvolvimento da linha de serviços REST e oferecer uma forma padrão de implementação da mesma [PEREIRA, 2008]. Sua implementação de referência é conhecida como *Jersey* (<https://jersey.dev.java.net/>).

Com a utilização de Jersey, um serviço é implementado como um recurso web através de uma classe *Recurso* e as requisições são tratadas por métodos das mesmas. Uma classe *Recurso* contém anotações da JAX-RS para indicar os mapeamentos e as operações existentes [PEREIRA, 2008].

Para utilizar o Jersey em uma aplicação RESTful, os prefixos de URI's do serviço devem ser mapeados para um Servlet do Jersey. A figura 5.6 mostra o arquivo `web.xml` da aplicação configurado com este mapeamento. Como todas as URI's são do serviço REST, toda a aplicação é mapeada para o Servlet do Jersey.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee">
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>
      com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <init-param>
      <param-name>
        com.sun.ws.rest.config.feature.Redirect
      </param-name>
      <param-value>true</param-value>
    </init-param>

    <init-param>
      <param-name>
        com.sun.ws.rest.config.feature.ImplicitViewables
      </param-name>
      <param-value>true</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>

  </servlet>

  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Figura 5.6 Arquivo `web.xml` da aplicação

No nosso exemplo, à classe de modelo (`Filme.java`) foi incluída uma anotação do tipo `@XmlRootElement`, que indica que o valor de uma instância desta classe será representado como um elemento XML (figura 5.7).

Para contextualizar o exemplo nos conceitos apresentados no capítulo anterior, pode-se considerar a classe de modelo como uma representação de um recurso a ser disponibilizado. Enquanto a visão tradicional disponibiliza um serviço, a visão REST torna um recurso acessível para se obter a sua representação (os dados do filme solicitado).

```
/**
 *
 * @author Livia
 */
@XmlRootElement
public class Filme {

    private int codFilme;
    private String titulo;
    private int ano;
    private String diretor;
    private String genero;

    public Filme(int codFilme, String titulo, int ano, String diretor, String genero) {
        super();
        this.codFilme = codFilme;
        this.titulo = titulo;
        this.ano = ano;
        this.diretor = diretor;
        this.genero = genero;
    }
}
```

Figura 5.7 Classe `Filme.java`

A classe de recurso do exemplo é a `FilmeResource.java` (figura 5.8). Nela estão todos os serviços com o prefixo `/filme`.

A classe deve ser anotada com `@Path` contendo o prefixo. Esta anotação é necessária para que a classe seja associada com o prefixo nas requisições. Além disso, a classe deve conter as anotações `@Consumes` e `@Produces`, que declaram que os serviços da mesma são capazes de consumir e gerar conteúdo nos formatos `text/XML` e `application/json` [PEREIRA, 2008]. O formato JSON (*JavaScript Object Notation*) é um subconjunto da notação de objeto de JavaScript que pode ser interpretado por qualquer linguagem.

```

@Path("filme")
@Consumes( { "text/xml", "application/json" })
@Produces( { "text/xml", "application/json" })
public class FilmeResource {

    private FilmeService filmeService;

    public FilmeResource() {
        this.filmeService = new FilmeService();
    }

    @GET
    @Path("{codFilme}")
    public Response buscarFilme(@PathParam("codFilme") String codFilme) {
        Filme filme = filmeService.buscar(new Integer(codFilme).intValue());
        if (filme == null) {
            return Response.status(HttpServletResponse.SC_NOT_FOUND).build();
        }
        Response resposta = Response.ok(filme).build();
        return resposta;
    }

    @POST
    public Response cadastrarFilme(Filme filme) {
        filme = filmeService.cadastrar(filme);
        try {
            return Response.created(new URI("" + filme.getCodFilme())).build();
        } catch (URISyntaxException e) {
            throw new RuntimeException(e);
        }
    }
}

```

Figura 5.8 Classe de Recurso FilmeResource.java

O primeiro método oferecido pelo recurso é o de busca de filmes. O método `buscarFilme` foi anotado com `@Path("{codFilme}")`. A junção da anotação do método `@Path("{codFilme}")` com a da classe (`@Path("filme")`) especifica que o método é capaz de responder a requisições para a URI `/filme/codFilme`, como por exemplo `/filme/1`.

Como também foi inserida a anotação `@GET` no método de busca, todas as requisições do tipo HTTP GET serão tratadas por este método. A anotação `@PathParam` é utilizada para injetar no parâmetro `codFilme` o valor que veio da URI.

O outro método oferecido é o de cadastro de filmes. A utilização da anotação `@POST` garante que todas as requisições do tipo HTTP POST serão encaminhadas para o método

anotado. O parâmetro do método `cadastrarFilme` é um objeto `Filme`, que será enviado no formato XML em uma requisição.

Para utilização do recurso por uma aplicação cliente RESTful, as solicitações HTTP devem ser feitas através da classe `HttpClient` (da API `commons-http-client`). Esta API permite que se montem requisições HTTP e sejam recebidas suas respostas, da mesma forma que ocorreria com um browser simples [PEREIRA, 2008].

Para consumir o recurso disponibilizado pelo lado servidor, a classe `FilmeMain.java` (figura 5.9) efetua um cadastro de um filme e realiza uma busca do mesmo.

A conversão do filme criado em um arquivo XML é feita com o auxílio da classe `XStream`, que pertence a uma biblioteca criada para serializar objetos.

```

/**
 * @author Livia
 */
public class FilmeMain {
    public static void main(String[] args) {
        try {

            // E agora a criação de um Filme de maneira RESTful
            Filme filme = new Filme();
            filme.setCodFilme(4);
            filme.setTitulo("Sete Vidas");
            filme.setAno(2009);
            filme.setDiretor("Gabriele Muccino");
            filme.setGenero("Drama");

            // Montando requisição com o commons-http-client
            HttpClient client = new HttpClient();
            PostMethod method = new PostMethod("http://localhost:8080/REST/filme/");

            // Geração de XMLs com o XStream
            XStream xstream = new XStream();
            xstream.alias("filme", Filme.class);
            String filmeXml = xstream.toXML(filme);
            System.out.println(filmeXml);

            // Definindo corpo da requisição
            StringRequestEntity requestEntity;
            requestEntity = new StringRequestEntity(filmeXml, "text/xml", "UTF-8");
            method.setRequestEntity(requestEntity);
            client.executeMethod(method);

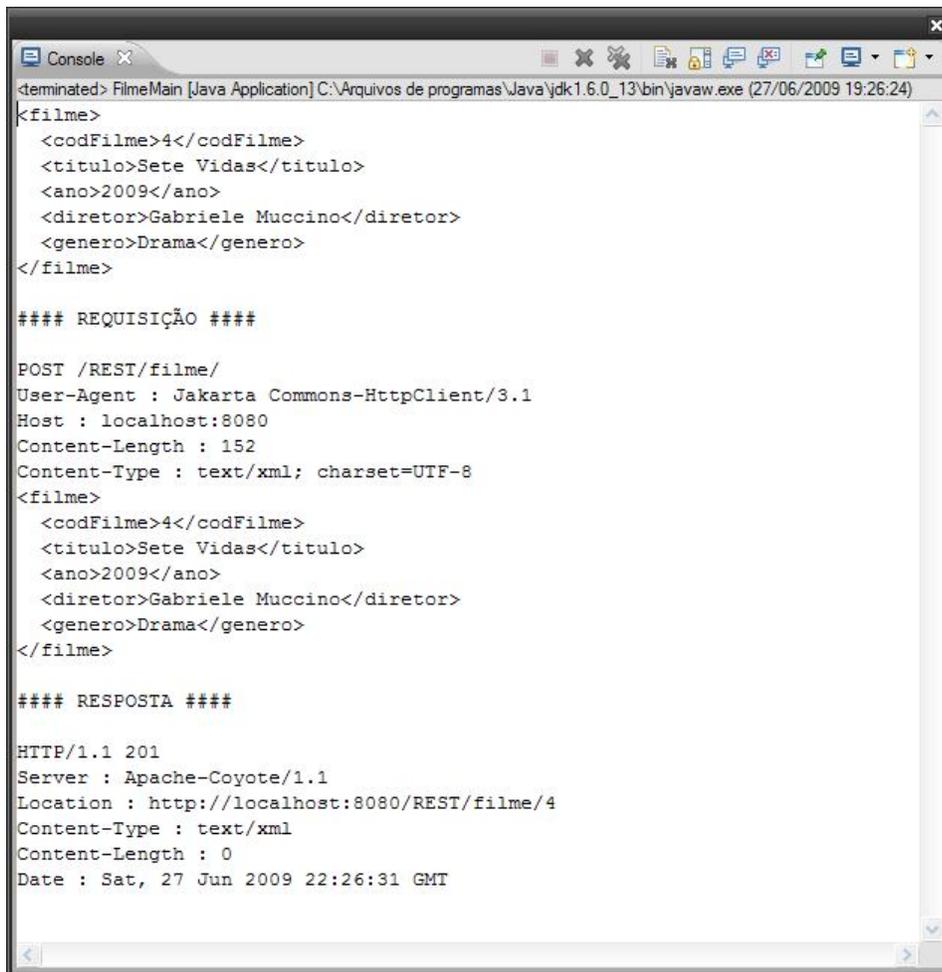
            // Imprime o cabeçalho e o corpo da requisição
            System.out.println("\n#### REQUISIÇÃO ####\n");
            System.out.println(method.getName() + " " + method.getPath());
            Header[] headersRequest = method.getRequestHeaders();
            for (Header header : headersRequest) {
                System.out.println(header.getName() + " : " + header.getValue());
            }
            method.getRequestEntity().writeRequest(System.out);

            // Imprime o cabeçalho e o corpo da resposta
            System.out.println("\n\n#### RESPOSTA ####\n");
            System.out.println(method.getStatusLine().getHttpVersion() + " " + method.getStatusLine().getStatusCode());
            Header[] headersResponse = method.getResponseHeaders();
            for (Header header : headersResponse) {
                System.out.println(header.getName() + " : " + header.getValue());
            }
            System.out.println(method.getResponseBodyAsString());
        } catch (Exception e) {
            System.err.print(e.getMessage());
        }
    }
}

```

Figura 5.9 Classe de teste FilmeMain.java

Na execução da classe, ainda são exibidos o cabeçalho e o corpo tanto da requisição quanto da resposta, para fins de demonstração. A figura 5.10 mostra a saída da execução da classe FilmeMain.java.



```
deminated> FilmeMain [Java Application] C:\Arquivos de programas\Java\jdk1.6.0_13\bin\javaw.exe (27/06/2009 19:26:24)
<filme>
  <codFilme>4</codFilme>
  <titulo>Sete Vidas</titulo>
  <ano>2009</ano>
  <diretor>Gabriele Muccino</diretor>
  <genero>Drama</genero>
</filme>

#### REQUISIÇÃO ####

POST /REST/filme/
User-Agent : Jakarta Commons-HttpClient/3.1
Host : localhost:8080
Content-Length : 152
Content-Type : text/xml; charset=UTF-8
<filme>
  <codFilme>4</codFilme>
  <titulo>Sete Vidas</titulo>
  <ano>2009</ano>
  <diretor>Gabriele Muccino</diretor>
  <genero>Drama</genero>
</filme>

#### RESPOSTA ####

HTTP/1.1 201
Server : Apache-Coyote/1.1
Location : http://localhost:8080/REST/filme/4
Content-Type : text/xml
Content-Length : 0
Date : Sat, 27 Jun 2009 22:26:31 GMT
```

Figura 5.10 Saída da execução da classe de teste `FilmeMain.java`

As requisições do tipo GET podem ainda ser feitas através de um browser, como ilustra a figura 5.11. Através das URI's `http://localhost:8080/REST/filme/1` e `http://localhost:8080/REST/filme/2`, os dados dos filmes de códigos 1 e 2, respectivamente, foram retornados no formato XML.

Neste contexto, estas URI's representam os identificadores de recurso definidos por Fielding. Ainda de forma contextual, o browser utilizado na figura 5.11 – o Web Browser do Eclipse – representa o agente do usuário, também definido por Fielding.



Figura 5.11 Requisição do tipo GET através de um Web Browser

5.3 Utilização do REST

Buscando, sobretudo simplicidade e maior desempenho das aplicações, o paradigma REST tem sido utilizado em larga escala por algumas organizações, listadas a seguir.

Google

No ano de 2008, mais de um ano após o Google ter descontinuado o Google SOAP Search API – conjunto de bibliotecas para acessar alguns serviços do Google através de serviços web em sua forma tradicional, utilizando SOAP – a organização lançou o AJAX Search API (<http://code.google.com/intl/pt-BR/apis/ajaxsearch/>), que oferece uma interface simples REST. A API suporta o método HTTP GET e retorna mensagens do tipo JSON [GOOGLE, 2009].

A utilização da API é simples. Basta fazer uma solicitação GET e processar uma resposta JSON. Além disso, não é necessária a utilização de uma chave de licença – obrigatória para a Google SOAP Search API. Outra vantagem é a facilidade de uso e o número ilimitado de

requisições.

Porém, a API ainda apresenta algumas restrições, como o número máximo de oito resultados por requisição e a não-permissão para alterar a ordem dos resultados da pesquisa [GOOGLE, 2009].

Twitter

O *twitter* representa um dos mais recentes e bem sucedidos exemplos de redes sociais da Web. Oferece uma API para desenvolvedores web possibilitando que os usuários acessem de forma simples as diversas funcionalidades que a ferramenta disponibiliza – a *Twitter REST API* (<http://apiwiki.twitter.com/Twitter-API-Documentation>).

A API permite automatizar todas as funcionalidades da ferramenta que são acessadas manualmente. A mesma torna possível, através de requisições HTTP, obter *tweets* – mensagens enviadas – de um determinado usuário, responder um usuário ou até mesmo filtrar *tweets* com base em critérios pré-definidos [IBM, 2009].

Para ilustrar este cenário, pode-se considerar a URI http://twitter.com/statuses/user_timeline.atom?id=liviaruback. A mesma retorna o *timeline* – última mensagem enviada – em formato XML, do usuário *liviaruback*.

Assim como A API REST do Google, a *Twitter REST API* ainda tem algumas limitações como o número máximo de 100 requisições por hora. Porém, Caso sejam necessárias mais de 100 requisições, pode-se requisitar ao *twitter* pertencer à uma “lista branca” que permite aos seus usuários mais de 100 acessos por hora [IBM, 2009].

5.4 Considerações

Como demonstrado nas seções anteriores, a implementação de Serviços Web pode ser feita de duas maneiras: 1) da forma como a Arquitetura de Serviços Web foi especificada (seção 4.5.1) ou 2) através de princípios da Arquitetura Orientada a Recursos (seção 4.5.2).

A Arquitetura de Serviços Web baseada na visão tradicional de Serviços Web (utilizando

SOAP, WSDL e UDDI) é muito bem definida; segue regras para a troca de dados, segurança, entre outros. Já a Arquitetura Orientada a Recursos, por ser uma aplicação prática dos conceitos REST definidos por Fielding, segue as características de vários estilos arquiteturais.

A principal diferença entre as duas abordagens citadas é teórica e não tecnológica: A Arquitetura de Serviços Web é baseada em serviços, enquanto na Arquitetura Orientada a Recursos, os serviços são encarados como recursos e são identificados por uma URI.

A visão tradicional para implementação de Serviços Web é baseada em trocas de mensagens no formato do protocolo SOAP e que devem seguir um contrato WSDL. Nesse contexto, o protocolo HTTP é utilizado somente para transporte. Ambos os lados – cliente e servidor – precisam conhecer e entender SOAP e WSDL para desempacotar e utilizar os dados.

Dessa forma, cria-se uma abstração da comunicação onde ferramentas devem encapsular a informação para que o seu receptor a entenda e a processe. Além disso, as trocas de mensagens precisam ser envelopadas dentro de um pacote SOAP, e o que é realmente utilizado é uma parte muito pequena do conteúdo contido no mesmo. Estas informações desnecessárias podem comprometer o desempenho de um sistema.

O fato é que, para se construir um Serviço Web é necessário apenas: um cliente, um serviço, informação, um meio de encapsular esta informação – geralmente XML – e de um meio para acessar esta informação – o protocolo HTTP.

De acordo com a visão arquitetural de REST, o protocolo HTTP já é rico o suficiente para que seja criada uma abstração para a construção de serviços.

A Arquitetura Orientada a Recursos faz uso unicamente do protocolo HTTP para a comunicação, através do acesso direto aos métodos nativos – GET, POST, PUT, DELETE, HEAD e OPTIONS. Como a Web é baseada em HTTP, em alguns casos a utilização de REST é mais adequada, pois aumenta a velocidade de comunicação entre os serviços e conseqüentemente o desempenho geral da aplicação.

É importante destacar que o paradigma REST não surgiu para substituir a visão já existente e sim pra complementar a mesma. Quando realmente existe a necessidade de serem manipulados vários formatos - em vez de só XML – ou quando for uma interação do tipo aplicação-aplicação não faz sentido usar REST, mas sim a visão tradicional.

Porém, em determinadas aplicações, REST é claramente mais adequado do que a visão tradicional, por exemplo, quando se pode ter uma interação usuário-aplicação. Um exemplo que

ilustra este cenário é um serviço que poderia ser acessado via Ajax através de um browser diretamente por um usuário.

Em casos como esse, a utilização de REST é muito mais simples. Basta ter um conhecimento sobre o protocolo HTTP e seus métodos invocando-os a partir de métodos nativos do protocolo. No exemplo descrito nas sessões anteriores, por exemplo, para se buscar informações sobre um determinado filme, a abordagem REST visivelmente é a mais simples, visto que a solicitação GET pode ser realizada até mesmo através de um browser.

Porém, para integrações em aplicações corporativas, nas quais os serviços oferecidos devem possuir um contrato formal, serem desacoplados e reutilizáveis, o paradigma REST não é a melhor opção, pois não possui um padrão oficial para a descrição dos seus serviços.

Já no caso de serviços para celulares, PDA's e outros dispositivos móveis, onde cada requisição tem um custo relativamente alto, utilizar serviços REST sem dúvida é a melhor opção, devido, sobretudo à simplicidade e ao desempenho. A maior prova disso tem sido a utilização em larga escala do paradigma por organizações descritas anteriormente, como o Google e *Twitter*.

6 *Conclusão*

Nos últimos tempos, o aumento da complexidade dos sistemas de software, aliado à crescente importância de aplicações distribuídas eficientes acarretou mudanças nos projetos dos sistemas. Fez-se necessário, então, o surgimento de modelos em diferentes níveis de abstração para a solução de problemas relacionados ao projeto de sistemas. A Arquitetura de Software surgiu como o objetivo de identificar propriedades e relacionamentos relevantes em um projeto de um sistema.

Para que o conhecimento de projeto de software fosse catalogado de maneira organizada e útil para uma futura reutilização, surgiram os estilos arquiteturais. Os estilos arquiteturais surgiram com a finalidade de classificar as classes de arquiteturas de software de acordo com suas características peculiares.

Paralelo ao surgimento dos estilos arquiteturais e visando atender necessidades de interação entre diferentes tipos de aplicações mesmo rodando em plataformas e sistemas operacionais diferentes, a Arquitetura de Serviços Web surgiu como uma arquitetura computacional voltada para serviços Web - componentes que permitem às aplicações enviar e receber dados em formato XML.

A Arquitetura de Serviços Web foi especificada de acordo com o protocolo SOAP – responsável por empacotar as mensagens de requisição e resposta – e com base em uma linguagem de descrição responsável por expor as interfaces dos Serviços Web – WSDL. Atualmente, a maioria dos serviços web faz uso dos padrões definidos na especificação, ou seja, as mensagens são trocadas encapsuladas pelo protocolo SOAP e seguem um contrato WSDL.

Esta maneira de manipular serviços muitas vezes pode comprometer o desempenho do sistema, visto que se gera uma camada de abstração acima do protocolo HTTP. Além disso, o que é realmente utilizado é uma parte muito pequena do conteúdo contido nas mensagens.

Este trabalho apresentou um estilo arquitetural para lidar com serviços web – o paradigma REST. O mesmo surgiu a partir de uma tese de doutorado de Roy Fielding em 2000 e tem emergido como uma alternativa capaz de melhorar o desempenho das aplicações com serviços

Web. A aplicação prática dos conceitos definidos por Fielding é conhecida como Arquitetura Orientada a Recursos.

A partir de um mesmo cenário – de consultas a um acervo de filmes – foram ilustradas, além das características de ambas as abordagens, suas principais diferenças.

Embora as ferramentas e IDE's para a manipulação de serviços Web ainda sejam recentes, REST tem ganhado cada dia mais credibilidade, devido à sua simplicidade e à melhoria de desempenho dos sistemas que o utilizam.

Referências Bibliográficas

[ACME. **The Acme Project**. 2009. Disponível em: <http://www.cs.cmu.edu/~acme/index.html>. Acesso em maio de 2009]

[AMORIM, S. S. **A Tecnologia Web Services e sua Aplicação num Sistema de Gerência de Telecomunicações**. Dissertação (Mestrado Profissional de Engenharia de Computação) - Universidade Estadual de Campinas, Instituto de Computação, Março 2004. Disponível em: <http://libdigi.unicamp.br/document/?code=vtls000332721>. Acesso em maio de 2009]

[BELLWOOD, T. **UDDI Version 2.04 API Specification**. [S.l.], 2002. Disponível em: <http://www.uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm>. Acesso em maio de 2009]

[BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P. e STAL, M. **Pattern Oriented Software Architecture: A System Of Patterns**, John Wiley e Sons, 1996]

[CLEMENTS, P., et al. **Documenting Software Architectures: Views and Beyond**. 2ª Edição. Addison Wesley, 2001]

[CHAVEZ, CHRISTINA VON FLACH G. **Linguagens de Descrição Arquitetural**, Disponível em [https://disciplinas.dcc.ufba.br/pastas/MAT161/2006.2%20\(antes%20do%20CurriculoNovo\)/Projeto/apss2-6.ModelagemSA-ADL.ppt](https://disciplinas.dcc.ufba.br/pastas/MAT161/2006.2%20(antes%20do%20CurriculoNovo)/Projeto/apss2-6.ModelagemSA-ADL.ppt) Acesso em junho de 2009]

[CORBA. **The official CORBA standard from the OMG group**. 2009. Disponível em <http://www.omg.org/docs/formal/04-03-12.pdf>. Acesso em junho de 2009]

[DCOM. **[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol Specification**. 2009. Disponível em [http://msdn.microsoft.com/pt-br/library/cc201989\(en-us\).aspx](http://msdn.microsoft.com/pt-br/library/cc201989(en-us).aspx). Acesso em junho de 2009]

[FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Dissertação de Doutorado - University of California, Irvine, 2000. Disponível em http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Acesso em junho de 2009]

[GARLAN, David; SHAW, Mary. **An Introduction to Software Architecture**. Carnegie Mellon University, 1994. Disponível em http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf. Acesso em maio de 2009]

[GOOGLE, SEARCH API REST. Disponível em

<http://googlesystem.blogspot.com/2008/04/google-search-rest-api.html>. Acesso em julho de 2009]

[IBM, **Using the Twitter REST API**. Disponível em <http://www.ibm.com/developerworks/web/library/x-twitterREST/index.html?ca=drs-> . Acesso em julho de 2009]

[MICROSOFT. **Application Architecture Guide 2.0**. Microsoft patterns and practices. 2008. Disponível em <http://apparchguide.codeplex.com/>. Acesso em maio de 2009]

[MEDYK, Sérgio. **Web Services em Gerência de Redes**. Material de Apoio. Paraná, 2006 Disponível em

<http://www.ppgia.pucpr.br/~jamhour/Download/pub/Mestrado%202006/Web%20Service.pdf>. Acesso em maio de 2009]

[MEDVIDOVIC, N.; TAYLOR, R. **A Classification and Comparison Framework for Software Architecture Description Languages**, IEEE Transactions on Software Engineering, 26(1), Janeiro 2000.]

[MENDES, Antonio. **Arquitetura de Software**, 2002. Desenvolvimento orientado para arquitetura. Rio de Janeiro]

[MERSON, Paulo. **Mini-Curso: Como documentar arquitetura de software**. https://disciplinas.dcc.ufba.br/pastas/MATA63/leitura/Merson05_minicurso_SBES1.pdf. Acesso em maio de 2009]

[NUNES, Sérgio; DAVID, Gabriel. **Uma Arquitectura Web para Serviços Web**. Universidade do Porto, Portugal. Disponível em

<http://repositorio.up.pt/aberto/bitstream/10216/281/2/Uma%20Arquitectura%20Web%20para%20Servi%C3%A7os%20Web.pdf> Acesso em junho de 2009]

[OMG, I. **Unified Modeling Language**. [S.l.], 2008. Disponível em: <http://www.uml.org/>. Acesso em maio de 2009]

[PEREIRA, Bruno. **Web Services REST**. 2008. Artigo da edição 56 da Java Magazine.]

[RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services**. Web Services for the real world. O'Reilly Media, Sebastopol: 2007]

[RMI. **Remote Method Invocation Home**. 2009. Disponível em <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>. Acesso em junho de 2009]

[ROSSINI, Sérgio. **Serviços Web Semânticos**. Juiz de Fora: 2008. 80p. Monografia em Ciência da Computação - UFJF]

[SOFTWARE ENGINEERING INSTITUTE, CarnegieMellon. <http://www.sei.cmu.edu/architecture/index.html>. Acesso em maio de 2009]

[SOUZA, JAIRO DE **Mini-Curso: Tecnologias para desenvolvimento de uma arquitetura orientada a serviços**, 2008. Disponível em <http://www.semanadacomputacao.ufjf.br/arquivos.php> . Acesso em junho de 2009]

[STONEBRAKER, Michael. **The Case for Shared Nothing Architecture**. Publicado em *Database Engineering*, Volume 9, Número 1 (1986)]

[TOBALDINI, Ricardo Ghisi. **Aplicação do Estilo Arquitetural REST a um Sistema de Congressos**. Florianópolis: 2008. Monografia em Ciências da Computação – UFSC]

[VERMA, Dinesh C. **Legitimate Applications of Peer-to-Peer Networks**. John Wiley & Sons, Inc, 2004. Disponível em: http://media.wiley.com/product_data/excerpt/98/04714636/0471463698.pdf. Acesso em maio de 2009]

[W3C. **World Wide Web Consortium**, 2009. Disponível em: <http://www.w3.org/>. Acesso em maio de 2009]

[WSA. **Web Services Architecture**, 2004. Disponível em: <http://www.w3.org/TR/ws-arch/wsa.pdf>. Acesso em junho de 2009]con