**Ricardo Ghisi Tobaldini**

# *Aplicação do Estilo Arquitetural REST a um Sistema de Congressos*

Florianópolis – SC

2008/1

**Ricardo Ghisi Tobaldini**

# *Aplicação do Estilo Arquitetural REST a um Sistema de Congressos*

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientador:
Luiz Fernando Bier Melgarejo

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis – SC

2008/1

*Aplicação do Estilo Arquitetural REST a um Sistema de Congressos*

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

---

Prof. Luiz Fernando Bier Melgarejo
Orientador

Banca Examinadora

---

Prof. Dr. Rosvelter Coelho da Costa

---

Prof. Dr. José Mazzucco Jr.

# *Resumo*

O desenvolvimento deste trabalho é resultado da necessidade em integrar recursos de uma arquitetura de software híbrida para sistemas hipermídia distribuídos, conhecida como REST - Transferência de Estado Representacional - e proposta por Fielding (2000), a um sistema destinado ao gerenciamento de congressos, chamado Open Conference Systems – OCS. Para tanto, buscou-se o estado da arte sobre o estilo arquitetural REST e a Arquitetura Orientada a Recursos - uma implementação do estilo REST idealizada por Richardson e Ruby (2007). A aplicação destes elementos permitiu desenvolver uma interface de comunicação entre o sistema OCS e outros sistemas compatíveis com REST, através da disponibilização de recursos em um formato padrão e bem definido através da Arquitetura Orientada a Recursos. A implementação foi projetada para aplicação na Web, suportada pela a tecnologia Java e utilizando como base o arcabouço Restlet. Este novo sistema culminou na criação do ROCS, uma solução que reúne o estilo arquitetural REST ao sistema de congresos OCS. O uso do ROCS permite confirmar que a fusão do estilo REST com a Arquitetura Orientada a Recursos possibilita trabalhar com recursos e interações importantes para o gerenciamento de congressos.

Palavras-chaves: Arquitetura de software. REST. Arquitetura Orientada a Recursos. Open Conference Systems. OCS. Restlet.

*“Em tudo na vida a perfeição é finalmente atingida, não quando nada mais existe para acrescentar, mas quando não há mais nada para retirar.”*

***Antoine de Saint-Exupéry***

# *Agradecimentos*


Agradeço a todas as pessoas que contribuiram, de alguma forma, para a conclusão desta etapa em minha vida.

Em especial, minha mãe Ligia Ghisi, pelo apoio SEMPRE presente. Minha tia Patricia Ghisi que faz sempre o impossível e meus avós Lydio e Ângela Ghisi pelo exemplo de vida.

À equipe do Edugraf, Diego, Giovani, Pablo, Erich, Paulo e Marcelo, pelas constantes críticas, sugestões, boas idéias e contribuições para este trabalho. Ao prof. Melgarejo, coordenador do laboratório e também orientador deste trabalho, uma personalidade ímpar.

E por fim, aos meus amigos, pelas piadas, cervejas, histórias e tragédias.

# *Sumário*

**Lista de Figuras**

**Lista de Tabelas**

**1 Introdução** p. 12

1.1 Objetivo Geral . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 12

1.2 Objetivo Específico . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 13

**2 Arquitetura de Software** p. 14

2.1 Estilos Arquiteturais . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 14

2.1.1 Cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 15

2.1.2 Cliente-Servidor . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 15

2.1.3 Cliente-Servidor Sem Estado . . . . . . . . . . . . . . . . . . . . . p. 15

2.1.4 Cliente com Cache-Servidor sem Estado . . . . . . . . . . . . . . . p. 16

2.1.5 Sistema em Camadas . . . . . . . . . . . . . . . . . . . . . . . . . p. 16

2.1.6 Código sob Demanda . . . . . . . . . . . . . . . . . . . . . . . . . p. 17

2.1.7 Interface Uniforme . . . . . . . . . . . . . . . . . . . . . . . . . . p. 17

**3 REST** p. 18

3.1 Compondo o REST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 18

3.1.1 Cliente-Servidor . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 18

3.1.2 Sem Estado . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 19

3.1.3 Cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 19

3.1.4 Código sob Demanda . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 20
3.1.5 Separação em Camadas . . . . . . . . . . . . . . . . . . . . . . . . . . p. 20
3.1.6 Interface Uniforme . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 20
3.2 Elementos Arquiteturais . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 20
3.2.1 Elementos de Dados . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 20
3.2.2 Conectores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 23
3.2.3 Componentes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 24

**4 Arquitetura Orientada a Recursos** p. 26
4.1 Recursos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 26
4.2 Endereçamento . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 27
4.3 Sem-Estado . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 28
4.4 Representações . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 29
4.5 Ligações e Conectividade . . . . . . . . . . . . . . . . . . . . . . . . . . p. 30
4.6 Interface Uniforme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 30
4.6.1 GET . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 31
4.6.2 PUT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 31
4.6.3 POST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 31
4.6.4 DELETE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 32
4.6.5 HEAD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 32
4.6.6 OPTIONS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 32
4.6.7 Métodos Seguros e Idempotentes . . . . . . . . . . . . . . . . . . . . p. 32

**5 Open Conference Systems** p. 34
5.1 Papéis de Usuários . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 35
5.2 Arquitetura do Sistema . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 37
5.2.1 Páginas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 37

5.2.2 Ações . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 38
5.2.3 Modelos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 38
5.2.4 Data Access Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 38
5.2.5 Suporte . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 39
5.2.6 Formulários . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 39
5.2.7 Moldes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 40
5.3 Extensão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 40

**6 OCS Sob Perspectiva REST** p. 41

6.1 Modelo e a Ausência de Estados . . . . . . . . . . . . . . . . . . . . . . . . p. 41
6.2 Recursos e Serviços . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 42
6.3 Interface Uniforme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 42

**7 ROCS - Uma interface REST para o OCS** p. 43

7.1 Comunicação entre ROCS e OCS . . . . . . . . . . . . . . . . . . . . . . . p. 43
7.2 O Arcabouço Restlet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 44
7.3 Recursos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 45
7.4 Representações . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 46
7.4.1 Representações de Coleções . . . . . . . . . . . . . . . . . . . . . . . p. 47
7.4.2 Representações de Instâncias . . . . . . . . . . . . . . . . . . . . . . . p. 47
7.5 Ligação entre Recursos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 48

**8 Conclusão** p. 50

**Referências Bibliográficas** p. 51

# *Lista de Figuras*

1 Estilo Arquitetural Cliente-Servidor . . . . . . . . . . . . . . . . . . . . . . . . p. 18

2 Estilo Arquitetural Cliente-Servidor sem Estado . . . . . . . . . . . . . . . . . . p. 19

3 Estilo Arquitetural Cliente com Cache-Servidor sem Estado . . . . . . . . . . . p. 19

4 Fluxograma de papéis e suas funções no sistema OCS . . . . . . . . . . . . . . p. 36

5 Arquitetura do sistema OCS . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 37

6 Diagrama de Classes do Padrão de Projeto DAO . . . . . . . . . . . . . . . . . p. 39

7 Comunicação entre ROCS e OCS . . . . . . . . . . . . . . . . . . . . . . . . p. 44

8 Composição do arcabouço Restlet . . . . . . . . . . . . . . . . . . . . . . . . p. 45

9 Grafo das ligações entre os recursos do sistema . . . . . . . . . . . . . . . . . p. 49

# *Lista de Tabelas*


1 Conectores REST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 23

2 Componentes do REST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 24

3 Exemplos de uso dos métodos POST, PUT e DELETE em relação aos recursos p. 46

# *Lista de Listagens*

7.1 Exemplo de representação XML de uma coleção . . . . . . . . . . . . . . . . p. 47

7.2 Exemplo de representação XML de uma instância . . . . . . . . . . . . . . . . p. 47

7.3 Exemplo da representação de ligação entre recursos . . . . . . . . . . . . . . . p. 48

# 1 *Introdução*

Aplicações Web ocorrem cada vez mais frequentemente entre os incontáveis sítios espalhados pela Internet e podem ser implementadas utilizando-se um universo de padrões, ferramentas, tecnologias, arquiteturas e estilos arquiteturais disponíveis especificamente para tal tarefa.

Fielding (2000), em sua dissertação de doutorado, propõe um novo estilo arquitetural, chamado Representational State Transfer (REST). Embora REST possua um conjunto de restrições que o caracterizam como um estilo arquitetural, ainda permite muitas liberdades em sua aplicação à alguma arquitetura já existente. De forma agregar valor a este recurso Richardson e Ruby (2007) criaram uma arquitetura chamada Arquitetura Orientada a Recursos como forma de aplicar o estilo arquitetural REST ao desenvolvimento de aplicações Web.

Ao mesmo tempo em que surgia o interesse em aplicações para estilo REST, aparecia no Edugraf – Laboratório de Software Educacional situado no Departamento de Informática e Estatística da Universidade Federal de Santa Catarina e coordenado pelo prof. Melgarejo - a necessidade em desenvolver estudos sobre uma nova estrutura e que possibilitasse incorporar recursos em um sistema já existente destinado ao gerenciamento de congressos acadêmicos.

A partir desta necessidade surgiu a motivação para este trabalho, no qual inicia apresentando o estilo arquitetural REST em sua essência, a Arquitetura Orientada a Recursos como uma forma de utilização de REST em aplicações Web e a aplicação destes conceitos na criação de uma interface REST para o sistema de congressos Open Conference Systems.

## 1.1 Objetivo Geral

Conhecer o estilo arquitetural REST e propor uma interface REST a um sistema já existente, chamado Open Conference Systems, que não segue o estilo REST.

## 1.2 Objetivo Específico

- Conhecer a arquitetura REST;
- Conhecer a aplicação Open Conference Systems;
- Analisar esta aplicação sobre a perspectiva REST;
- Propor uma interface que segue o estilo REST para este sistema.

# 2 *Arquitetura de Software*

> A arquitetura de software de um programa ou um sistema computacional é a estrutura ou as estruturas do sistema, que constituem componentes de software, as propriedades externamente visíveis destes componentes, e a relação entre eles (BASS; CLEMENTS; KAZMAN, 2003).

A arquitetura de software é baseada no princípio da abstração. Abstrai-se detalhes de um sistema de forma a identificar melhor suas propriedades, desta forma um sistema complexo pode ter vários níveis de abstração cada um com sua própria arquitetura. Uma arquitetura se preocupa exclusivamente com o comportamento e interação entre os diversos elementos que a compõem, ignorando detalhes internos específicos de cada um destes elementos.

Cada elemento, por sua vez, pode ser composto por outras arquiteturas de forma a atender ao comportamento exigido pela arquitetura do nível superior (externa a este elemento). Esta seqüência de abstrações pode ser aplicada até que não seja mais possível decompor um elemento.

## 2.1 Estilos Arquiteturais

Um estilo arquitetural descreve uma categoria de sistema que engloba diversos aspectos: um conjunto de componentes que exercem uma função requerida para um sistema (ex: banco de dados); um conjunto de conectores que permitem a comunicação, cooperação e coordenação entre componentes; restrições e características que definem como componentes podem ser integrados; e modelos semânticos que permitem entender as propriedades do sistema como um todo através da análise das propriedades das partes que o constituem (BASS; CLEMENTS; KAZMAN, 2003). Estilos formam, assim, mecanismos de categorização de arquiteturas e definição de características comuns. Cada estilo captura a essência de um padrão de interações ignorando eventuais detalhes do resto da arquitetura.

### 2.1.1 Cache

Cache é um estilo arquitetural de replicação de informações, sua funcionalidade básica é guardar respostas à requisições feitas anteriormente e usá-las mais tarde como resposta a novas requisições equivalentes.

A replicação pode ser feita proativamente, ou seja, respostas para algumas requisições são obtidas antes mesmo que alguma requisição seja feita, antecipando os resultados. Ou ainda pode ser feita de forma tardia, onde uma resposta somente é armazenada quando solicitada pelo menos uma vez.

### 2.1.2 Cliente-Servidor

O estilo Cliente-Servidor é composto de um servidor que fornece uma série de serviços e aguarda por conexões de clientes. Do outro lado o cliente é responsável por saber quais serviços necessita e requisitá-los a um servidor. O servidor, quando solicitado, envia uma resposta - positiva ou negativa - ao cliente. Neste estilo o servidor geralmente é um processo interminável ou seja, fica aguardando requisições de clientes indefinidamente e pode servir mais de um cliente.

O princípio fundamental por trás do estilo Cliente-Servidor é a separação de responsabilidades. Quando esta separação é feita apropriadamente, simplifica o componente servidor em favorecendo a escalonabilidade e transfere a responsabilidade de interação com o usuário para o componente do cliente. A separação dos dois componentes permite que cada um se desenvolva independentemente desde que a interface de comunicação entre os dois não seja alterada.

### 2.1.3 Cliente-Servidor Sem Estado

Ao estilo Cliente-Servidor adiciona-se também uma característica de comunicação sem estado. Neste tipo de comunicação cada requisição feita pelo cliente deve conter todas as informações necessárias para que o servidor possa processar esta requisição sem que ele tenha que armazenar nenhuma informação referente ao estado do cliente. Desta forma todo o estado da sessão entre o cliente e o servidor deve ser conhecido somente pelo cliente.

Este princípio traz algumas vantagens como visibilidade, confiabilidade e escalonabilidade. A visibilidade é melhorada pois é necessário observar apenas uma requisição do cliente para saber o que está acontecendo entre cliente e servidor pelo fato de que toda a informação necessária para isto está embutida na requisição do cliente. Esta característica também facilita a tarefa de recuperação de falhas, melhorando a confiabilidade. E como o servidor precisa apenas lidar

com as requisições atuais do cliente, sem a dependência de requisições anteriores, a liberação de recursos computacionais se dá de forma mais ágil melhorando assim a escalonabilidade.

Junto com estas vantagens são agregadas algumas desvantagens também. Por não armazenar o estado de cada cliente o servidor necessitará a cada requisição várias informações adicionais para completá-la, estas informações podem se tornar repetitivas em uma série de requisições, aumentando a quantidade de dados trocados através da rede e possivelmente diminuir a performance geral da aplicação. Além disso o armazenamento de estado apenas no cliente transfere grande parte da semântica da aplicação a ele, desta forma o servidor não pode controlar se uma determinada versão do cliente está se comportando da maneira correta ou não.

### 2.1.4 Cliente com Cache-Servidor sem Estado

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 (escalonabilidade), melhorando a performance geral do sistema e diminuindo o tempo de resposta da aplicação. Apesar disto, este sistema pode prejudicar a confiabilidade da aplicação. Imagine um cenário onde um cliente A realiza uma requisição, o servidor responde e o cliente armazena esta resposta em cache. Um outro cliente B, então, realiza uma transação que modificará o conteúdo da resposta para a requisições iguais as que o cliente A realizou anteriormente. Se o cliente A repete aquela requisição mais uma vez e utiliza seu cache para aumentar a agilidade da resposta, receberá dados inconsistentes.

### 2.1.5 Sistema 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 com quem está interagindo diretamente, ou seja, o conhecimento do sistema que o componente possui é restrito a uma camada apenas.

Camadas podem ser usadas para encapsular serviços legados, proteger novos serviços de clientes legados, melhorar a escalonabilidade através da distribuição de carga entre redes e processadores ou para a aplicação de políticas de segurança, fornecendo restrições de acessos. Outra função que uma camada pode exercer é a de transformação do conteúdo das requisições e respostas.

A grande desvantagem deste sistema é um maior atraso da chegada da informação de um extremo ao outro por causa dos diversos intermediários. Para um sistema que suporta as características de cache esta desvantagem pode ser contornada através do compartilhamento de cache entre as camadas permitindo que uma requisição que já foi feita anteriormente possa ser servida através de uma camada intermediária ao invés de percorrer todo o caminho até destino final.

### 2.1.6 Código sob Demanda

Permite que as funcionalidades do cliente sejam inseridas em tempo de execução através da descarga e execução de código na forma de applets ou scripts. Esta característica pode facilitar a extensão do sistema mas também reduz a visibilidade.

### 2.1.7 Interface Uniforme

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

# 3 REST

Transferência de Estado Representacional, REST, é um estilo arquitetural híbrido para sistemas hipermídia distribuídos derivado dos estilos arquiteturais mostrados no capítulo II e combinado com características adicionais.

## 3.1 Compondo o REST

A composição de REST se dá através da agregação de dois outros estilos arquiteturais:

- Cliente com Cache-Servidor sem Estados com suporte a código sob demanda e separação em camadas;
- Interface uniforme

### 3.1.1 Cliente-Servidor

Através do uso da arquitetura Cliente-Servidor REST torna possível que os componentes, o cliente e o servidor, se desenvolvam independentemente pois existe aí uma separação de responsabilidades bem definida. A evolução independente de cada parte torna REST apto a suportar os requisitos de aplicações de múltiplos domínios organizacionais na escala da Internet.

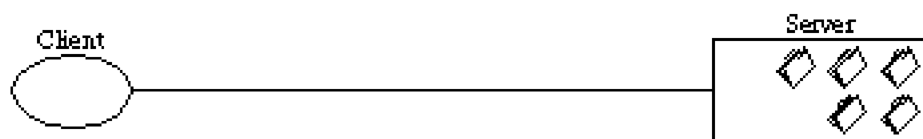


Figura 1: Estilo Arquitetural Cliente-Servidor
Fonte: Fielding (2000)

### 3.1.2 Sem Estado

Em REST as comunicações não devem depender de estados controlados pelo servidor, incorpora-se nesta característica o estilo Cliente-Servidor sem Estado. Desta forma cada requisição feita pelo cliente sempre deverá possuir todos os atributos para que ela possa ser processada pelo servidor sem que este último se aproveite de informações adicionais sobre o contexto da comunicação. Toda informação de estado deve ser conhecida somente pelo cliente.

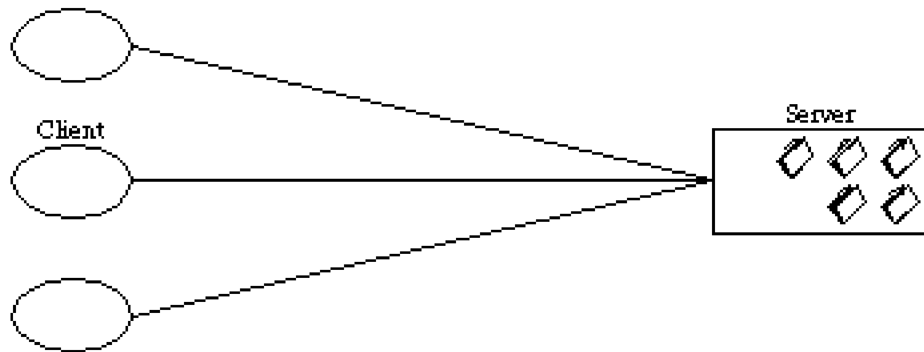


Figura 2: Estilo Arquitetural Cliente-Servidor sem Estado
Fonte: Fielding (2000)

### 3.1.3 Cache

Para melhorar a eficiência da comunicação entre cliente e servidor adiciona-se a REST características de cache formando o estilo Cliente com Cache-Servidor sem Estado.

As características de cache torna necessária a marcação explícita de uma resposta no âmbito de esta ser ou não passível de cache. Se uma resposta pode se aproveitar das vantagens oferecidas pelo cache então dá-se ao cliente o direito de reusar esta resposta futuramente.

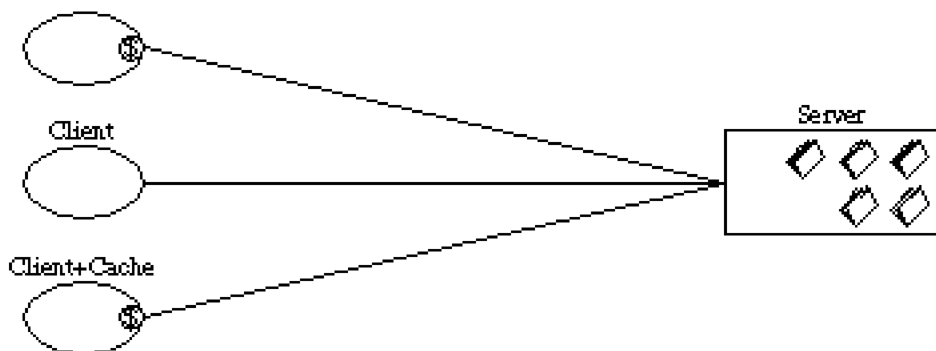


Figura 3: Estilo Arquitetural Cliente com Cache-Servidor sem Estado
Fonte: Fielding (2000)

### 3.1.4 Código sob Demanda

A utilização de código sob demanda também pode caracterizar uma aplicação REST com o objetivo de simplificar e ampliar as capacidades dos clientes. Porém, seu uso acarreta na redução de visibilidade do sistema e por isso é opcional.

### 3.1.5 Separação em Camadas

Ao estilo REST é adicionado também a possibilidade de separação em camadas. Seu uso permite 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 de cliente para servidor ou de servidor para cliente 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 sempre visívies a todos os intermediários.

### 3.1.6 Interface Uniforme

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

REST é definido através de quatro elementos de interface: identificação de recursos; manipulação de recursos através de representações; mensagens auto-descritivas; e hipermídia como máquina de estados da aplicação.(FIELDING, 2000)

## 3.2 Elementos Arquiteturais

REST ignora detalhes de implementação de componentes e sintaxe de protocolos para se focar nos papéis dos componentes, suas interações e interpretações de elementos de dados.(FIELDING, 2000)

### 3.2.1 Elementos de Dados

A natureza e o estado dos elementos de dados de uma arquitetura são os aspectos chaves do REST. Quando selecionamos uma ligação - na Web por exemplo - a informação necessita ser movida do local onde é armazenada para o local onde ela será processada. Esta característica

é distinta de muitos outros paradigmas de processamento distribuído, onde é possível - e geralmente mais eficiente - mover "o processamento"(em forma de uma expressão de busca ou um programa móvel) até os dados ao invés de levar os dados ao processamento.

Um arquiteto de software distribuído tem, fundamentalmente, três alternativas:(FIELDING, 2000)

- processar os dados onde eles estão localizados e enviar a resposta ao cliente em um formato específico;
- encapsular os dados e o código de processamento e enviá-los ao cliente para que este último o execute;
- enviar os dados brutos ao cliente juntamente com metainformações que descrevam os dados, permitindo ao cliente escolher e utilizar o processamento que lhe for conveniente.

Cada abordagem tem suas vantagens e desvantagens. A primeira opção - o estilo cliente-servidor tradicional - permite que toda informação sobre a natureza dos dados seja de estrito conhecimento do servidor, fazendo com que a implementação do cliente seja simplificada. Entretando toda a responsabilidade de processamento de informações fica a cargo do servidor, levando a problemas de escalonabilidade. O segundo método provê o processamento especializado ao cliente ao mesmo tempo em que o limita a processar os dados apenas do jeito que foi estipulado pelo servidor. Finalmente, utilizando-se o terceiro método, permite-se que o servidor seja simples e escalonável ao mesmo tempo em que minimiza a quantidade de dados transferidos, mas perde a vantagem da ocultação de informação e necessita que ambos cliente e servidor entendam o mesmo tipo de dados.

REST é um híbrido destes três métodos, focando-se num entendimento compartilhado dos tipos de dados através de metadados, e limitando o escopo do que é revelado através de uma interface uniforme.(FIELDING, 2000)

No REST os componentes se comunicam através da transferência de representações de um recurso em um formato selecionado dinamicamente de um conjunto de tipos de dados padrões, baseado nas capacidades e desejos do componente que irá receber as informações e da natureza do recurso. Sejam estas representações no mesmo formato que os dados brutos ou derivadas destes, isto permanece transparente.

### Recursos e Identificadores de Recursos

A abstração chave de informação no REST é um recurso.(FIELDING, 2000) 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.

Um recurso $R$ pode ser representado como uma função de mapeamento temporal $M_r(t)$, onde o tempo $t$ mapeia para um conjunto de entidades, valores ou equivalentes.(FIELDING, 2000) Um recurso pode ser mapeado para um conjunto vazio, o que permite que referências sejam feitas a um conceito antes mesmo deste conceito existir. Para identificar um recurso o REST utiliza um identificador de recurso, um exemplo de identificador de recursos é a URI[1] utilizada na Web.

### Representações

Uma representação, em REST, é uma seqüência de bytes e mais os metadados desta representação, que descrevem aqueles bytes. Exemplos de representações são: documentos em formato texto, arquivos binários, mensagens HTTP[2]. De forma mais precisa, uma representação consiste de dados, metadados para descrever os dados e, ocasionalmente, outro conjunto de metadados que descrevem os metadados (geralmente utilizado para verificação de integridade da mensagem(FIELDING, 2000)).

Os metadados são apresentados na forma de pares nome-valor, onde o nome corresponde a um padrão que define a estrutura do valor e sua semântica(FIELDING, 2000). Mensagens de respostas podem incluir metadados da representação assim como metadados do recurso (informações sobre o recurso que não são específicas à representação fornecida).

Outro elemento participante de uma representação são as informações de controle. As informações de controle são utilizadas para definir o propósito de uma mensagem entre os componentes ou parametrizar requisições e redefinir o comportamento padrão de alguns elementos. Por exemplo, o comportamento de um cache pode ser alterado através dessas informações embutidas em uma requisição ou uma resposta. Estas informações podem ser utilizadas também para indicar o estado atual de um recurso solicitado, o estado desejado para um recurso ou a representação de alguma condição de erro para uma resposta.

O formato dos dados de uma representação é conhecido como *tipo de informação*. Uma

[1] Unified Resource Identifier
[2] Hypertext Transfer Protocol

representação pode ser incluída em uma mensagem e processada de acordo com os dados de controle e a natureza do tipo de informação. Um tipo de informação pode ser intencionado para processamento automatizado, outros para serem visualizados pelo usuário e outros para ambos.(FIELDING, 2000) A composição de tipos de informação podem ser utilizadas para incluir múltiplas representações em uma única mensagem.

### 3.2.2 Conectores

REST utiliza vários tipos de conectores para encapsular a atividade de acesso a recursos e para transferir representações de recursos. Os conectores provêm uma interface abstrata para a comunicação entre componentes, melhorando a simplicidade do sistema através da separação de responsabilidades e ocultação da implementação de recursos e mecanismos de comunicação.(FIELDING, 2000)

| Conector | Exemplos da Web |
|---|---|
| cliente | libwww, libwww-perl |
| servidor | libwww, Apache API, NSAPI |
| cache | cache do navegador |
| resolução de nomes | bind (biblioteca de resolução de nomes DNS) |
| túnel | SOCKS, HTTP sobre SSL |

Tabela 1: Conectores REST

Similar à invocação procedural, a interface dos conectores permitem a passagem de parâmetros e resultados. Parâmetros de entrada da requisição consistem em informações de controle, um identificador de recurso e, opcionalmente, uma representação. Os parâmetros de saída são compostos pela informação de controle da resposta, opcionalmente metadados do recurso e uma representação - esta última também opcional.

Os dois primeiros tipos de conectores da tabela 1 - cliente e servidor respectivamente - são os conectores primários de REST. A diferença essencial entre os dois é que cabe ao cliente sempre ter iniciativa sobre o servidor, ou seja, fazer uma requisição, enquanto o servidor deve aguardar passivamente as requisições e atendê-las de forma a fornecer acesso aos seus serviços. Um componente pode incluir ambos os conectores.(FIELDING, 2000)

O conector cache é encontrado no meio, entre um cliente e um servidor, e tem a função de armazenar respostas passíveis de cache para reutilização em requisições posteriores. Um cache também pode ser utilizado no cliente de forma a evitar repetições de comunicações de rede, ou pelo servidor para diminuir a geração de respostas. Em ambos os casos o cache contribui

para a diminuição do tempo de resposta da aplicação. Tipicamente o cache é implementado no domínio do conector que o utiliza.

Um cache é capaz de determinar se uma reposta é ou não passível de reutilização pois a interface para cada recurso é genérica. Por padrão, respostas a requisições de recuperação de informações são armazenadas em cache enquanto outros tipos de requisição não o devem. Por exemplo uma requisição de alteração de dados relativos a um recurso, esta não deve ser armazenada em cache.

A resolução de nomes converte o nome, de uma URI por exemplo, em um endereço de rede. Através deste endereço de rede que é possível iniciar uma conexão com um componente. Este componente é peça chave para a existência de endereços de identificação dos recursos.

Finalmente, a última "peça" dos conectores de REST é o túnel. Sua função é criar um caminho virtual para travessia de informações de forma a transpor limites, como um firewall ou um roteador. A única razão para incluir túnel como parte da arquitetura REST, e não abstrair como parte da infraestrutura de rede, é que alguns componentes do REST podem trocar dinamicamente de um comportamento ativo para um comportamento de túnel.(FIELDING, 2000) Um exemplo deste cenário é quando utiliza-se um *proxy*[3] HTTP que, dependendo do tipo de requisição que ele intermedia deixa suas funções de *proxy* e passa a se comportar como um túnel para exercer a comunicação diretamente (sem passagem pelo proxy). Os túneis são destruídos após o término de uma comunicação entre componentes.

### 3.2.3 Componentes

Os componentes de REST são categorizados pelo papel que exercem no âmbito geral de uma aplicação. Abaixo são relacionados os tipos de componentes presentes em REST.

| Componente | Exemplos da Web |
|---|---|
| servidor de origem | Servidor HTTP (Apache, Microsoft IIS) |
| gateway | Squid, CGI, Proxy reverso |
| proxy | CERN Proxy, Gauntlet |
| agente do usuário | Navegador (Firefox, Safari, Internet Explorer) |

Tabela 2: Componentes do REST

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 comun de um agente do usuário

[3]Proxy é um tipo de serviço que atua nas requisições dos seus clientes executando os pedidos de conexão a outros servidores.

é o navegador Web, que provê acesso a informações (recursos) e processa as respostas (representações) de forma que a ser visualizada pelo usuário de acordo com a necessidade.

Um servidor de origem utiliza um conector do tipo servidor e trata das requisições de um cliente. É o servidor, também, a fonte de todas as representações dos seus recursos disponíveis aos clientes. Cada servidor deve prover seus serviços através de uma interface genérica para a sua hierarquia de recursos, detalhes de implementações destes recursos ficam escondidos por esta interface. Outra característica chave de um servidor de origem é ele ser o destino final de qualquer requisição feita por um cliente e que visa modificar

Os outros dois componentes, gateway e proxy, podem atuam, ao mesmo tempo, como cliente e servidor para a exercer a funcionalidade de encaminhamento de requisições e respostas. Mais especificamente, o proxy é um componente intermediário explicitamente escolhido pelo cliente para exercer encapsulamento de outros serviços, deslocamento de dados, aumento de performance ou algum tipo de controle de acesso de segurança. Já o gateway é transparente ao usuário e utilizado pelo servidor de origem dos recursos, funciona como um proxy reverso, provê os mesmos serviços que um proxy porém, a diferença entre os dois é que o proxy é utilizado (ou não) explicitamente pelo usuário, enquanto que o gateway é imposto pelo servidor.

# 4 *Arquitetura Orientada a Recursos*

O estilo arquitetural REST define apenas um conjunto de restrições, que caracterizam uma aplicação dita *RESTful*, de forma muito abstrata e geral. Neste capítulo é mostrada a Arquitetura Orientada a Recursos, uma aplicação prática dos conceitos REST.

Através desta arquitetura é possível transformar um problema em uma aplicação REST através do emprego de URIs, HTTP e XML que funciona como o restante da Web (RICHARDSON; RUBY, 2007). Ela é composta por recursos, seus nomes, suas representações, a ligação entre estes recursos e propriedades como endereçamento, falta de estados, conectividade e interface uniforme.

## 4.1 Recursos

Um recurso é alguma coisa que pode ser armazenada em um computador e representada como uma seqüência de bits, como por exemplo um documento, uma imagem, etc. Ele pode ser algum objeto físico, como uma maçã, ou um conceito abstrato, como "coragem"(RICHARDSON; RUBY, 2007).

São exemplos de recursos:

- Um número primo qualquer
- O número primo após 5
- Os números primos entre 5 e 200
- Informações sobre tainha
- Os números de venda de automóveis no mês de março

Para ser útil, um recurso precisa ser acessível, isto é, deve ter um nome e um lugar onde seja possível encontrá-lo. Para isto é utilizado um Identificador de Recurso, desta forma, um recurso precisa ter pelo menos um identificador para que se torne acessível.

REST, enquanto um estilo arquitetural, prevê a utilização de um identificador de recurso, no caso da arquitetura orientada a recursos são utilizadas URIs para identificação de cada recurso. As URIs devem ser utilizadas de forma estruturada, intuitiva e uniforme. Por exemplo, se para obter informações sobre Tainha utiliza-se a URI `/informaçõesSobre/tainha` e para informações sobre Robalo a URI `/aRespeitoDe/robalo`. O objetivo dos destes dois recursos é o mesmo: representar informações sobre uma espécie de peixe, porém, suas URIs são compostas de forma não uniforme, uma prática não aconselhável.

Um recurso pode ser referenciado por uma ou mais URIs. Se um recurso possui diversas URIs é mais fácil para os clientes utilizar este recurso, o problema é que cada URI adicional dilui o valor de todas as outras, pois alguns clientes usariam uma URI e outros outra e não há uma forma automática de verificar que todas estas URIs se referem ao mesmo recurso. Este problema pode ser contornado através do uso de uma URI principal e a sua indicação como tal na resposta para as URIs secundárias.

## 4.2 Endereçamento

Uma aplicação é endereçável se ela expõe aspectos interessantes sobre seu conjunto de dados como recursos (RICHARDSON; RUBY, 2007). Como são atribuídas URIs a recursos, uma aplicação é endereçável quando ela expõe pelo menos uma URI para cada informação que ela deseja servir. Ainda, o endereçamento significa que uma URI, sozinha, contém informações suficientes para executar alguns tipos de interação (WORLD WIDE WEB CONSORTIUM, ).

O protocolo HTTP é perfeitamente endereçável. O sítio `http://www.google.com.br` pode ser tomado como um exemplo de aplicação HTTP deste tipo. Quando preenchemos o campo de busca com alguma expressão, esta expressão é traduzida em uma URI que possui todas as informações para que a busca ocorra como solicitada pelo usuário. A capacidade de endereçamento permite que seja possível utilizar esta mesma URI para se obter exatamente aquela pesquisa em outra ocasião, ou ainda, é possível repassar esta URI a outra pessoa e ela obterá esta mesma pesquisa. Caso esta aplicação não fosse endereçável, seria necessário sempre entrar no sítio, preencher o campo de busca e clicar no botão `Pesquisar` para que o resultado desejado fosse obtido.

## 4.3 Sem-Estado

A ausência de estados significa que cada requisição feita pelo cliente ocorre de maneira isolada no servidor, isto é, não há lembrança de estados entre uma requisição e outra. Quando uma requisição é feita, ela deverá conter todos os dados para que seja completada pelo servidor e este nunca irá se beneficiar de informações de requisições anteriores.

Pode-se considerar a ausência de estados em termos de endereçamento. Este último diz que cada informação interessante deve ser esposta na forma de um recurso e ter sua URI própria. No caso da ausência de estados, cada estado possível deverá ser expressado na forma de um recurso e ter sua própria URI (RICHARDSON; RUBY, 2007).

Um contra-exemplo de aplicação com estados é o FTP. Quando conecta-se a um servidor FTP o usuário é direcionado a um diretório padrão que depende da configuração do servidor. Qualquer comando que o usuário executar será relativo a aquele diretório. Ele poderá apagar arquivos, renomear, baixar e enviar arquivos, sempre relativos ao diretório em que o usuário se encontra perante ao servidor, fazendo com que seu uso seja simples. Pode-se considerar este diretório como sendo o estado atual do usuário, o qual o servidor deverá manter sempre em sincronia com a aplicação cliente. Este trabalho de sincronia de estado é uma tarefa complexa em redes confiáveis, o que não é uma característica da Internet, o que o torna um protocolo mais complexo que o HTTP.

O servidor sem estados nunca perde o ponto da aplicação que o cliente se encontra, pois a cada requisição são enviados ao servidor todas as informações necessárias. Em analogia ao FTP, o cliente nunca executaria uma ação no diretório errado caso o servidor perdesse a sincronia de estado com o cliente.

A simplicidade do protocolo permite características adicionais interessantes como o fácil balanceamento de carga entre servidores, já que não é necessário sincronia de estados de clientes entre servidores, e também a possibilidade de *cache*.

Uma forma de quebrar a ausência de estados no servidor é utilizando o conceito de sessões. Neste caso, quando um usuário realiza a primeira requisição com o servidor, lhe é atribuído um identificador de sessão, um número ou uma cadeia de caracteres. Este identificador poderá ser atrelado a um cliente através do uso de *cookies*[2] ou propagados como um componente da URI.

[2] *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. É utilizado em alguns mecanismos de autenticação e controle de estado de sessão com o servidor.

## 4.4 Representações

Uma aplicação é dividida em diversos recursos, cada recurso é organizado de maneira a transmitir uma idéia ao usuário, como por exemplo “informações sobre Tainha”. Porém um servidor web não pode enviar uma idéia ao usuário, ao invés disso ele precisa enviar uma seqüência de bytes em um formato específico, que tenha alguma utilidade ao cliente. Isto é uma representação de um recurso, uma informação útil sobre o estado de um recurso (RICHARDSON; RUBY, 2007).

Alguns recursos são, por natureza, dados. Uma lista de cores de um modelo de carro tem como representação seus dados, os itens da lista. Esta lista pode ser em um formato XML, uma página HTML, puramente texto ou ainda uma imagem com a reprodução de cada cor, assim temos ainda várias representações para um mesmo recurso.

Porém há alguns tipos de recursos, um objeto físico por exemplo, que não são naturalmente representáveis através de dados. No caso de existir um recurso para uma geladeira, é impossível obter uma geladeira através do acesso a este recurso pois objetos físicos não são dados. Entretanto, a geladeira possui dados a seu respeito, os metadados. Cada prateleira da geladeira poderia ser descrita como um recurso que fornece informações sobre os alimentos que ali estão, assim como a temperatura interna da geladeira, o número de vezes que o compressor foi acionado para manter a temperatura e a posição do termostato que a regula.

As representações também podem circular no caminho inverso, do cliente para o servidor. Pode ser enviada uma representação para um novo recurso e o servidor trata de criar este recurso e associar a representação enviada a ele, ou ainda uma nova representação para um recurso já existente.

Caso um recurso possua mais de uma representação, é necessário escolher qual representação deve ser obtida. Este problema pode ser resolvido de diversas maneiras dentro das restrições impostas pelo estilo REST. A Arquitetura Orientada a Recursos recomenda apenas um jeito de isto ser feito, utilizando URIs distintas para cada tipo de representação. Esta abordagem permite que cada URI possua todas as informações para resolver o recurso, incluindo o tipo de representação desejada, por outro lado, isto causa a diluição de um mesmo recurso em várias URIs.

## 4.5 Ligações e Conectividade

Uma representação de um recurso contém dados sobre ele e, além disso, pode incluir ligações para outros recursos. Tomando como exemplo, novamente, a pesquisa sobre Tainha no Google, a representação desta busca é uma página HTML contendo diversas ligações para recursos que contém algum tipo de informação relacionada a Tainhas. Desta forma o servidor guia o usuário às informações, servindo ligações

A Web, da forma como as pessoas usam, possui um alto grau de conectividade. Usuários experientes podem entrar diretamente uma URI no navegador e navegar através de sítios mudando esta URI, mas também é possível acessar a estas URIs de um ponto de partida único, como um sítio de busca, e navegar através de ligações. Páginas possuem ligações entre si e, possivelmente, até entre sítios (RICHARDSON; RUBY, 2007).

Este mesmo conceito pode ser aplicado a outras aplicações web de forma que os recursos exponham, além das suas informações de dados, metadados contendo ligações para outros recursos relacionados.

## 4.6 Interface Uniforme

REST especifica uma interface uniforme, mas sua generalidade não diz muito a respeito como ela é. Na arquitetura aqui explorada esta interface é provenientente do procolo HTTP.

O protocolo HTTP define oito métodos dos quais seis são utilizados na arquitetura orientada a recursos, são eles: GET, PUT, POST, DELETE, HEAD e OPTIONS.

A cada método está associada uma função, com exceção do método PUT que pode ser utilizado de duas maneiras.

- GET: obter uma representação de um recurso;
- PUT: criar um novo recurso ou modificar um recurso já existente;
- POST: criar um novo recurso;
- DELETE: remover um recurso existente;
- HEAD: obter apenas a representação pertinente ao metadados de um recurso;
- OPTIONS: obter uma relação dos métodos suportados por um determinado recurso.

### 4.6.1 GET

O método GET é utilizado quando deseja-se obter a representação de um recurso. Quando é feita uma requisição a um recurso, uma URI, utilizando-se este método, o servidor irá responder ao cliente o conteúdo da representação deste recurso. Sempre que é inserida uma URI na barra de localização do navegador e pressionada a tecla *Enter* o navegador utiliza o método GET para se comunicar com o servidor e dizer a ele que está interessado em obter a representação para aquela URI.

### 4.6.2 PUT

Ao contrário do GET, o método PUT é utilizado para enviar representações ao servidor de forma que ele disponibilize-a através de um recurso.

Quando um método PUT é executado em recurso já existente, a representação deste recurso é substituida pelos dados enviados pelo cliente, junto com a requisição. Deste jeito o PUT atua como forma de atualizar a representação de um recurso. Caso este método seja submetido a um URI que ainda não existe, ele poderá ter a função de criar o recurso e ao mesmo tempo atribuir uma representação a ele.

O formato da representação enviada pelo cliente, em ambos os casos, depende da implementação do serviço, assim como a obrigatoriedade ou não de ela ser enviada. Também vai depender da implementação do serviço a utilização de um ou outro ou ambos os comportamentos.

### 4.6.3 POST

O método POST é utilizado para a criação de recursos subordinados a outros recursos, isto é, recursos que existirão em relação a um recurso pai.

Embora semelhante em função com o método PUT, o que o diferencia deste último é a incapacidade do cliente de inferir a URI do recurso que está sendo criado, ou seja, será o servidor que irá decidir qual será a URI resultante desta operação e que dará acesso ao recurso desejado. Este é o caso, por exemplo, quando é inserida uma nova linha em um banco de dados onde o identificador desta linha é um número gerado automaticamente pelo servidor, isto não irá permitir ao cliente inferir qual é o identificador das informações recém incluídas por ele.

Como resposta, o servidor irá enviar ao cliente um código de informação que irá dizer a

respeito do sucesso ou falha da operação. Juntamente, é possível enviar um outro dado, como uma mensagem de erro no caso de falha ou um dado de localização que informará onde está situado o novo recurso recém criado.

### 4.6.4 DELETE

Quando é necessária a destruição de um recurso existente, o método DELETE deverá ser utilizado. Opcionalmente poderá ser retornado ao cliente um código de confirmação da remoção ou erro, embora o não-retorno, em caso de sucesso, seja perfeitamente aceitável.

### 4.6.5 HEAD

Este método funciona de maneira idêntica ao método GET com a diferença de retornar ao cliente apenas os metadados do recurso, descartando totalmente todo o resto da representação. Isto é útil quando deseja-se saber que tipo de recurso é aquele, ou ainda, saber se o recurso existe sem a necessidade de obter o conteúdo completo de sua representação.

### 4.6.6 OPTIONS

Por fim, o método OPTIONS permite ao cliente descobrir quais métodos um determinado recurso está apto a responder. Caso a resposta de um recurso para este método fosse GET e HEAD, seria possível acessá-lo apenas para leitura, sem a possibilidade de executar um PUT, POST ou DELETE. O conjunto dos métodos permitidos pode ser diferente caso o cliente seja autenticado pelo servidor, possibilitando por exemplo que visitantes apenas tenham acesso a leitura de informações, enquanto que administradores, após autenticados, tenham o poder de atualizar, incluir ou remover recursos.

### 4.6.7 Métodos Seguros e Idempotentes

Se a interface uniforme do HTTP for utilizada da forma correta, ganha-se duas propriedades sobre os métodos (RICHARDSON; RUBY, 2007). Os métodos GET e HEAD são ditos seguros e GET HEAD, PUT e DELETE idempotentes.

Os métodos GET e HEAD executam somente a leitura de dados no servidor e nunca são utilizados para mudar o estado de nenhum recurso, por isso são ditos seguros. Mas isso não significa que eles não possam ter efeitos efeitos colaterais, como é o caso de um contador de

acessos que é incrementado cada vez que é feito um GET em um recurso. Como o cliente não pediu pelo efeito colateral, ele não é o responsável pelo seu acontecimento, um cliente nunca deve realizar uma requisição GET ou HEAD apenas pelo seus efeitos colaterais e esses efeitos nunca devem ser tão grandes a ponto de o cliente desejar não ter realizado tais operações.

Uma operação idempotente, na matemática, é uma operação que sempre resulta em um mesmo valor independentemente do número de vezes que é executada. Por exemplo, quando uma função $f(x)$ é aplicada para um mesmo valor, o resultado obtido será sempre o mesmo.

A arquitetura orientada a recursos estipula que as operações GET, HEAD, PUT e DELETE são idempotentes, ou seja, independente de quantas vezes forem acionadas para um recurso o resultado será o mesmo que acioná-las apenas uma vez.

É fácil observar este conceito para os métodos GET e HEAD, que nunca realizam alterações em um recurso. No caso de a execução de um DELETE, na primeira vez, o recurso é removido e não existirá mais, caso seja aplicado uma segunda vez o recurso continuará não existindo e o mesmo ocorrerá para $n$ execuções deste método neste recurso, ou seja, apagá-lo uma vez ou $n$ vezes causa o mesmo resultado.

Quando um recurso é criado ou atualizado através de PUT a idempotência também deverá ocorrer, se da primeira vez o recurso não existia ele será criado, nas requisições seguintes ele será recriado com as mesmas informações, ou seja, permanecerá do mesmo jeito. Porém há uma restrição que deve ser levada em consideração neste caso, se um recurso possui algum valor relativo ao número de requisições PUT executadas sobre ele, a idempotência não ocorre pois teremos um recurso com sua representação mudando a cada requisição. Este último comportamento deverá ser evitado sempre, favorecendo a idempotência.

A idempotência e a segurança dos métodos permitem ao cliente realizar requisições de forma confiável. Caso uma requisição seja feita e, aparentemente, nada aconteceu para o cliente, basta repetir a operação até que uma resposta seja enviada, confirmando que os desejos do cliente foram atendidos.

O método POST não se encaixa em nenhuma das duas classificações, a execução de duas requisições idênticas do método POST irá criar dois recursos, em lugares diferente, mas com o mesmo conteúdo, portanto POST não é idempotente e nem seguro.

# 5 *Open Conference Systems*

O Open Conference Systems, ou simplesmente OCS, é um sistema de código aberto e livre, desenvolvido através do projeto Public Knowledge Project[1], e oferece uma solução Web para o gerenciamento e publicação de congressos acadêmicos. O OCS permite aos organizadores de um congresso, segundo Public Knowledge Project (2007a):

- publicar as regras do congresso rapidamente de forma fácil e barata.
- administrar eficientemente tarefas trabalhosas da organização do congresso como o registro dos participantes e o envio de artigos.
- criar uma comunidade interativa na web para os participantes do congresso
- criar um acervo *online* de artigos dos congressos

O OCS foi desenvolvido com o objetivo reduzir a energia e o tempo necessários às tarefas de gerenciamento de um congresso acadêmico, ao mesmo tempo em que melhora a eficiência dos processos editoriais e a rastreabilidade de eventos como por exemplo o envio de um artigo por um autor ou a avaliação de um artigo por um revisor.

A instalação do OCS envolve apenas a configuração de servidores, utilizando-se de serviços de um banco de dados, e um servidor HTTP com suporte à linguagem de script PHP. Após instalado, funciona como um sítio na Web, permitindo aos seus usuários a utilização dos seus serviços através de um navegador Web. Entende-se aqui por "usuário"todas as os papéis envolvidos na organização, manutenção e colaboração do congresso, Ex: autores, avaliadores, etc.

[1]O projeto Public Knowledge Project é uma iniciativa de pesquisa e desenvolvimento que tem como objetivo melhorar a qualidade acadêmica de pesquisa através do desenvolvimento de ambientes inovadores de publicação *online* e compartilhamento de conhecimento. Localizado na University of British Columbia, Simon Fraser University e a Stanford University, o PKP produz, além do OCS, o Open Journal Systems (sistema semelhante ao OCS mas direcionado para a gerência de revistas acadêmicas), o Open Archives Harvester e o Lemon8-XML.

Além do gerenciamento de um único congresso, o OCS foi projetado para servir a múltiplos congressos e múltiplas edições de cada congresso, de forma que as informações fiquem centralizadas e sempre disponíveis através de um ponto único de acesso.

## 5.1 Papéis de Usuários

É utilizado o conceito de papel para cada usuário, de forma a restringir a cada um deles um conjunto de operações possíveis e o reflexo dessas operações no sistema. Cada usuário pode ter um ou mais papéis, cenário que pode ser diferente a cada edição de um mesmo congresso.

O OCS possui os seguintes papéis:

- Administrador do Sítio: Cuida de toda a instalação, configuração do sítio e criação de novos congresos no sistema.
- Gerente do Congresso: Administra o sítio raíz de um congresso onde, abaixo dele, estarão as diversas edições do congresso. Entre as responsabilidades deste gerente estão a administração das contas de usuários e também as edições do congresso.
- Diretor do Congresso: Gerencia o processo de submissão de artigos e publicação para uma edição de um congresso, além de definir as datas e prazos para eventos pertinentes ao congresso como a janela para submissão de artigos, avaliação, etc.
- Gerente de Inscrições: Responsável pelas inscrições das pessoas que participarão do congresso.
- Diretor de Áreas Temáticas: Responsável pela gerência dos trabalhos que participarão de uma área temática específica do congresso. Este papel pode acompanhar o processo de revisão dos artigos, decidindo se um determinado artigo será aceito ou não no congresso.
- Revisor: Avalia os artigos distribuídos a ele, recomendando ou não a inclusão de um artigo ao congresso.
- Autor: Submete seus artigos ao congresso.
- Leitor: Usuários que tem acesso às regras do congresso para leitura, dependendo de ajustes feitos pelo gerente do congresso este usuário precisará ou não ser registrado no sistema.

A figura 4 demostra como os papeis se relacionam para formar o sistema.

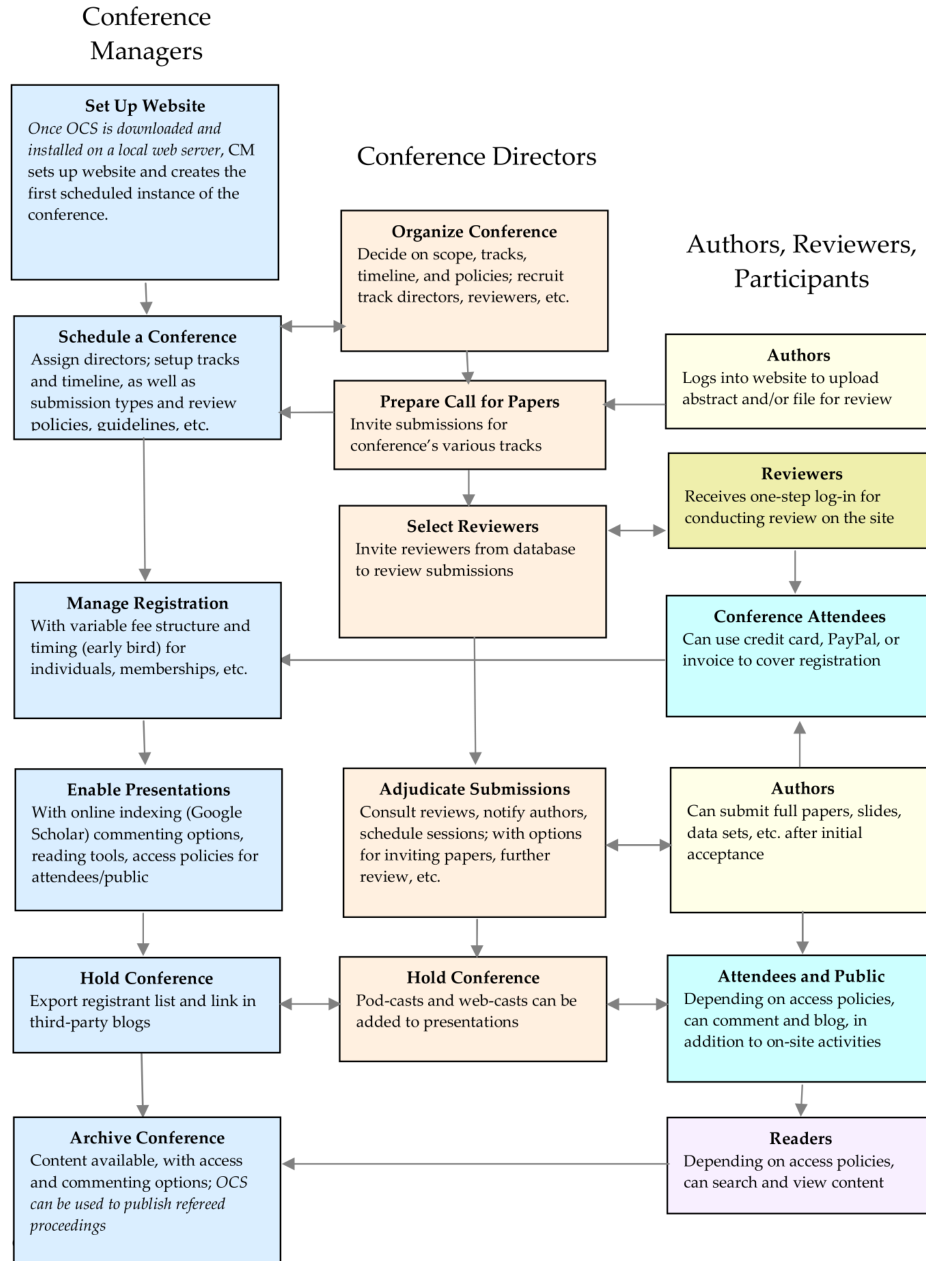


Figura 4: Fluxograma de papéis e suas funções no sistema OCS
Fonte: Public Knowledge Project (2007b)

## 5.2 Arquitetura do Sistema

O OCS emprega a arquitetura MVC, onde o sistema é separado em três partes chamadas modelo, visão e controle. Neste sistema cada uma dessas três partes é subdividida em pequenos módulos que desempenham funções pespecíficas. Na figura 5 é possível visualizar como esta arquitetura é implementada no OCS.

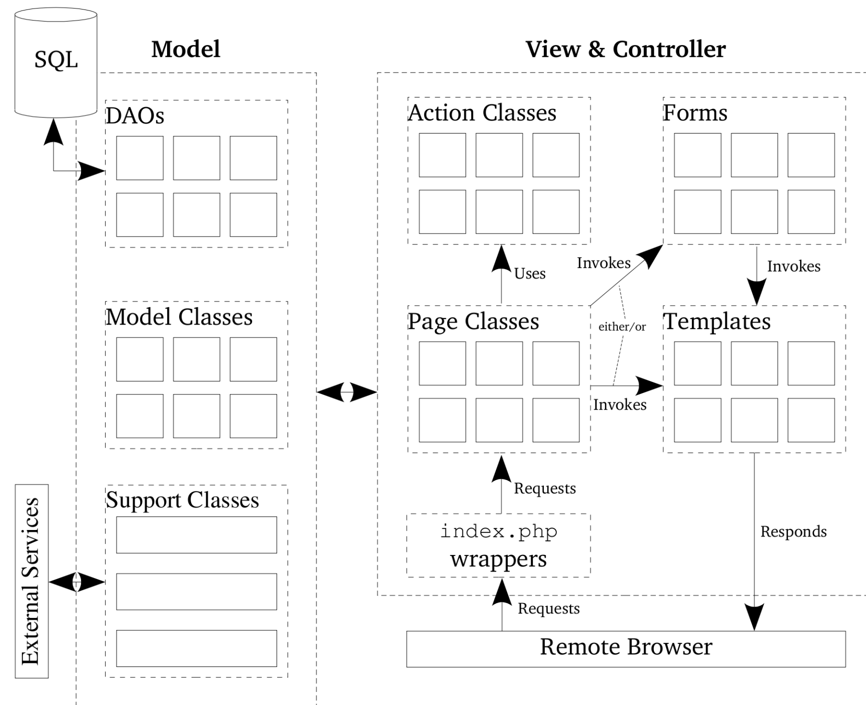


Figura 5: Arquitetura do sistema OCS
Fonte: Public Knowledge Project (2007c)

### 5.2.1 Páginas

As Páginas, chamadas pelo sistema de *Page Classes*, são responsáveis pelo tratamento das requisições feitas pelo usuário através do navegador Web, garantindo que estas requisições são válidas e os requerimentos de autenticação são satisfeitos, algumas vezes o tratamento de uma requisição é realizado de acordo com o papel do usuário que as faz. Quando as informações da requisição são provenientes de formulários, as tarefas de validação dessas informações são repassadas às classes mais especializadas chamadas Formulários, vistos mais adiante.

Após o processo de validação ter finalizado sem problemas, o processamento das informações é delegado a outras classes do sistema. Se depois do processamento é necessário o envio de uma resposta ao usuário, então as Páginas reasumem o controle e dispararam um Molde

gerador destas respostas.

### 5.2.2 Ações

As Ações são as classes as quais são delegados serviços pelas Páginas. São elas que desenvolvem o processamento propriamente dito das requisições do usuário, sem preocupações de validação, autenticação e apresentação (estes resolvidos pelas Páginas).

Os tipos mais comuns de tarefas realizadas pelas Ações são o envio de mensagens eletrônicas, manipulação de banco de dados e tratamento de envio de arquivos. Estas tarefas são dependentes do papel do usuário no sistema, deste modo, para cada papel de usuário existe uma classe de Ações responsável pela realização de suas tarefas.

### 5.2.3 Modelos

Os Modelos são classes que contém uma representação das informações do banco de dados na memória para fácil manipulação através da aplicação, são exclusivamente encapsuladoras de conteúdo. Os métodos presente nessas classes são, em sua maioria, do tipo obter/fixar. É importante observar que estas classes, apesar de possuírem informações provenientes do, ou com destino ao, banco de dados, não tratam diretamente com ele, isto é feio através das classes DAO explicadas a seguir.

### 5.2.4 Data Access Object

Data Access Object é um padrão de projeto que implementa o mecanismo de acesso necessário para trabalhar com uma fonte de dados. A fonte de dados pode ser um SGBD, um repositório baseado em LDAP ou *sockets* de baixo nível(ALUR; CRUPI; MALKS, 2004). O objetivo do DAO é retirar do modelo da aplicação as chamadas específicas à fonte de dados, como conseqüência oferece à aplicação certa independência ao tipo de fonte de dados a ser utilizado.

Uma aplicação que utiliza DAO apenas conhece esta interface e interaje diretamente com ela quando é necessário tratar consultas e modificações à camada de persistência. Esta interação é realizada através de mensagens que recebem como parâmetro, ou retornam, objetos que encapsulam informações chamados *ValueObjects*.

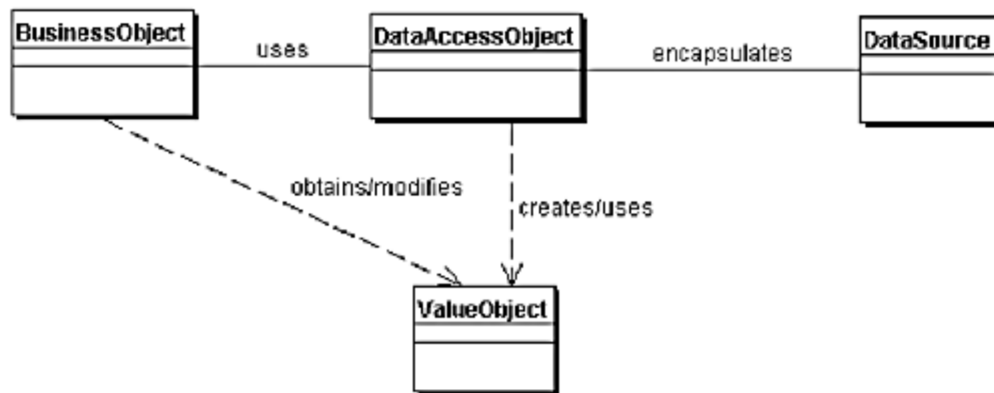


Figura 6: Diagrama de Classes do Padrão de Projeto DAO
Fonte: Alur, Crupi e Malks (2004)

No OCS os *ValueObjects* são chamados de Modelos (vistos no item anterior) e o módulo DAO utiliza ainda um arcabouço chamado ADOdb[1] para acesso ao SGBD.

### 5.2.5 Suporte

As classes de Suporte são compostas por módulos que realizam diversas tarefas indispensáveis ao sistema mas que não fazem parte diretamente do modelo de negócios da aplicação. Algumas tarefas realizadas por essas classes são:

- Envio de Mensagens Eletrônicas
- Internacionalização
- Dispositivos de Segurança
- Gerenciamento de Arquivos
- Gerenciamento de Sessões

### 5.2.6 Formulários

Os Formulários são classes que desempenham as tarefas relacionadas a validação e tratamento de erros de dados enviados pelo usuário através do preenchimento de formulários. Exemplos de aplicação são a verificação da entrada de uma data, notificação do usuário caso

[1] ADOdb é um arcabouço para as linguagens PHP e Python que oferece uma abstração para acesso a SGBD, suportando os sistemas MySQL, PostgreSQL, Interbase, Firebird, Informix, Oracle, MS SQL, Foxpro, Access, ADO, Sybase, FrontBase, DB2, SAP DB, SQLite, Netezza, LDAP, ODBC e ODBTP (LIM, 2008).

um campo de preenchimento obrigatório não tenha sido completado, se uma senha fornecida pelo usuário possuía no mínimo oito caracteres, etc.

### 5.2.7 Moldes

Os Moldes tem a função de transformar os dados fornecidos pela aplicação em informações visualizáveis pelo usuário do sistema. O OCS utiliza o arcabouço Smarty para realizar essa função.

O Smarty é um arcabouço de moldes e apresentação. Ele provê ao programador e ao projetista de moldes um conjunto de ferramentas para automatizar tarefas da camada de apresentação de uma aplicação (NEW DIGITAL GROUP INC., 2008).

## 5.3 Extensão

O OCS possui uma infraestrutura que provê alguns mecanismos para extender e modificar o comportamento do sistema sem que sejam necessárias modificações do código base da aplicação. Os conceitos envolvidos nesta infraestrutura são categorias, extensões e *hooks* (PUBLIC KNOWLEDGE PROJECT, 2007c).

Uma categoria define um comportamento de uma extensão e cada extensão deve pertencer a apenas uma categoria. Um exemplo de categoria neste sistema é `importexport`, que implementa certos métodos utilizados para a importação e exportação de dados.

Extensões são coleções de código e recursos que implementam uma funcionalidade adicional ou modificam o comportamento de funcionalidades já existentes. Uma extensão é carregada quando a categoria da qual faz parte é solicitada pelo sistema.

Além das categorias, o OCS utiliza *hooks* como uma forma de criar pontos de extensão. Cada hook possui um nome e é disparado quando algum evento ocorre. Um exemplo de hook é o `Request::redirect`, que é chamado antes de qualquer ação de redirecionamento. Este hook pode ser utilizado para reescrever uma URL de redirecionamento ou até mesmo interceptar o redirecionamento e impedir que ele ocorra.

# 6 OCS Sob Perspectiva REST

A arquitetura do sistema OCS é organizada conforme o padrão de projeto Modelo-Visão-Controle, onde a principal característica e boa prática de projeto de software, é a separação entre o modelo da aplicação e a apresentação(FOWLER, 2003).

Em REST é possível observar alguns componentes do MVC. O modelo da aplicação poderia ser visto como recursos, a visão composta das representações dos recursos e o controle disponível através dos métodos que implementam a interface uniforme. Entretando, uma aplicação que segue o padrão MVC não necessariamente é uma aplicação REST, como é o caso do Open Conference Systems.

## 6.1 Modelo e a Ausência de Estados

Uma das características do componente Modelo no OCS é a utilização de estados para que a semântica da comunicação entre as duas outras partes, Visão e Controle, sejam válidas. Esta abordagem vai de encontro a uma das principais características de REST, a ausência de estados.

No OCS, quando um usuário deseja exercer uma função atribuída ao seu papel no sistema, este precisa, antes de mais nada, ter uma sessão iniciada na aplicação. Este início de sessão implica em criar um estado que amarra a aplicação cliente ao servidor, disponibilizando as funções específicas a que o papel daquele usuário tem acesso.

A forma REST de resolver este problema seria enviar, junto com o acesso ao recurso que realiza determinada operação, as credenciais do usuário. Caso as credenciais fossem aceitas a operação seria completada e seu resultado enviado ao cliente na forma de uma representação, caso contrário seria disparado algum erro.

## 6.2 Recursos e Serviços

Embora o OCS seja organizado através de URIs elas, muitas vezes, expõem funções do sistema (serviços) ao invés de exporem aspectos sobre conjuntos de dados na forma de recursos. Um exemplo desta característica é a maneira de adicionar novos usuários ao sistema.

O jeito REST de fazer isso seria através de um método POST ou PUT em um recurso que represente a coleção de usuários, por exemplo `/usuários`. No OCS isto é feito através da execução de um método POST na URI `/user/createAccount`, neste caso é possível observar que a operação de criação de um novo usuário está ligada à função da URI, sendo o método POST apenas uma forma de envio dos dados à aplicação.

## 6.3 Interface Uniforme

A utilização de URIs para expor serviços ao invés de recursos também fere o princípio da Interface Uniforme, já que cada serviço possui sua maneira própria de ser utilizado. Deste modo a semântica dos métodos que compõem a Interface Uniforme (GET, PUT, POST e DELETE) são distorcidas.

No caso do OCS, o método POST pode ser utilizado para realizar uma pesquisa no sistema através do envio das informações a serem pesquisadas para a URI `/search/advancedResults`. Desta forma o método POST passa a ser seguro, pois pesquisas não afetam o conjunto de dados, e idempotente (execuções consecutivas tem sempre o mesmo reflexo no sistema).

Outro exemplo é a forma como são removidos usuários de um congresso. O OCS utiliza o método GET aplicado a uma URI (ex:`/manager/removeUser/3`) para desempenhar esta tarefa, o que fere gravemente a segurança deste método. Neste caso acontece uma operação com efeitos colaterais sérios através de um método que deveria ser inofensivo ao conjunto de dados do sistema.

# 7 *ROCS - Uma interface REST para o OCS*

A necessidade específica, dentro deste projeto, era de criar uma aplicação do tipo *Rich Client* que possibilitasse um administrador de um congresso exercer suas funções com maior liberdade, para isso, usufruindo de uma interface gráfica mais amigável e poderosa em comparação com a interface Web disponível no OCS. Para isto seria necessária a criação de um esquema de comunicação entre essas duas aplicações.

O sistema ROCS surgiu como uma resposta a esta necessidade, como o nome sugere, a contração de REST + OCS. A idéia é propor uma interface REST ao sistema OCS de modo a externalizar seus recursos, permitindo então sua utilização por outros sistemas compatíveis com REST.

## 7.1 Comunicação entre ROCS e OCS

Os sistemas ROCS e OCS são implementados utilizando tecnologias distintas (Java e PHP, respectivamente) e que não possuem mecanismos próprios que permitam o uso de objetos da implementação de uma aplicação diretamente na outra, eliminando a possibilidade da troca direta de objetos entre essas duas aplicações sem que fosse necessárias adaptações no sistema OCS.

A opção mais simples e que atenderia as necessidades do ROCS seria, então, a utilização da base da dados utilizada pelo OCS, onde se encontra uma boa parte da implementação do modelo desta aplicação. A figura 7 ilustra a posição das duas aplicações e um sistema de gerenciamento de banco de dados como forma de troca de dados entre elas.

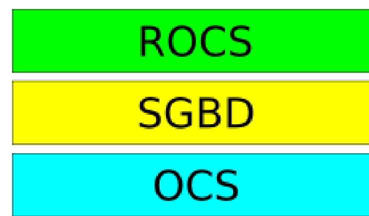


Figura 7: Comunicação entre ROCS e OCS

## 7.2 O Arcabouço Restlet

A implementação do projeto ROCS foi feita na linguagem de programação Java com a utilização do arcabouço Restlet. Este arcabouço foi escolhido por possuir uma implementação muito próxima dos conceitos de REST apresentados por Roy Fielding, facilitando o desenvolvimento de aplicações deste tipo.

Algumas características interessantes oferecidas pelo Restlet:

- Conceitos chave de REST, como Interface Uniforme, Recurso, Representação, Conectores, etc, possuem classes Java equivalentes;
- Pode ser utilizado tanto na parte do cliente quanto do servidor;
- Tratamento flexível de URIs
- Atua como um servidor Web completo;
- Suporte nativo para representações XML, compatível com DOM, SAX e XPath;
- Suporte a autenticação HTTP básica e HTTP sobre SSL (HTTPS);
- Utiliza múltiplas linhas de execução de forma a melhorar a escalonabilidade.

O arcabouço Restlet é composto de duas partes principais, a API Restlet, que suporta os conceitos REST, facilitando o tratamento de chamadas para aplicações cliente e servidor, e uma implementação para esta API (NOELIOS CONSULTING, 2008). Esta separação permite que diferentes implementações desta mesma API sejam utilizadas, sejam elas de código aberto ou comerciais. Para o projeto ROCS foi utilizada a implementação de referência Noelios Restlet Engine provida pela Noelios Consulting sob a licensa GPL[1].

A figura 8 demostra graficamente a posição da aplicação a ser desenvolvida perante os dois componentes do arcabouço Restlet.

[1] General Public License

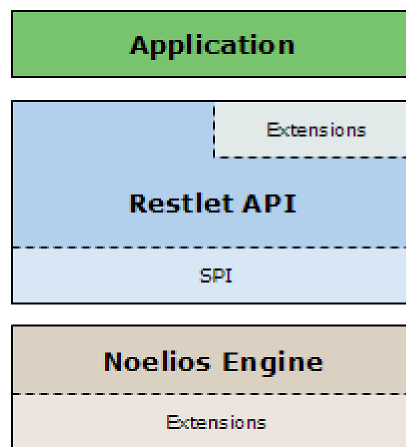


Figura 8: Composição do arcabouço Restlet
Fonte: Noelios Consulting (2008)

## 7.3 Recursos

Segundo Richardson e Ruby (2007), pode-se separar a etapa de projeto de recursos, capazes de tratar requisições de leitura e modificação, em 9 partes.

1. Entender o conjunto de dados
2. Dividir o conjunto de dados em recursos, e para cada tipo de recurso:
3. Nomear os recursos através de URIs
4. Expor um subconjunto da interface uniforme
5. Projetar as representações aceitas pelo servidor
6. Projetar as representações a serem servidas ao cliente
7. Integrar este recurso a outros recursos existentes através de ligações
8. Considerar o curso típico de eventos
9. Considerar condições de erro

Com base nesta receita, os recursos necessários foram levantados e implementados no decorrer do projeto de forma a atender as funcionalidades desejadas. Durante este processo também foram definidos padrões de representação, de emprego dos métodos que compôem a interface uniforme (GET, POST, PUT e DELETE) e de composição das URIs dos recursos.

Os métodos foram empregados seguindo a arquitetura orientada a recursos definida por Richardson e Ruby (2007), com a opção de utilizar o método PUT apenas como forma de modificar a representação de um recurso já existente, pois ela permite também utilizar este método para a criação de novos recursos. Os métodos restantes - GET, POST e DELETE - foram utilizados da forma convencional, como esta arquitetura propõe.

As URIs dos recursos seguiram um padrão de forma a facilitar sua leitura por pessoas, para isto foi utilizado o conceito de plural e singular para separar coleções e instâncias de uma entidade. Por exemplo, quando deseja-se fazer referência à coleção de usuários cadastrados no sistema deve-se recorrer à URI `/usuarios`, caso a necessidade seja um usuário específico sua URI será composta no singular, como em `/usuario/{identificadorDoUsuário}`. Também foi explorada a composição de recursos de forma hierárquica como, por exemplo, quando deseja-se obter a coleção de edições de um determinado congresso, dada através da URI `/congresso/{identificaçãoDoCongresso}/edicoes`.

A separação de coleções e instâncias de recursos através do uso de plural e singular também trouxe uma particularidade quanto ao uso dos métodos POST, PUT e DELETE. O método GET, neste caso, é indiferente, fornecendo a representação do recurso seja ele uma coleção ou uma instância de uma entidade. O método POST possui a semântica "adicionar à coleção de" e por isso é aplicado somente às URIs que representam coleções. Os outros dois métodos, PUT e DELETE, são aceitos em URIs de recursos que representam uma instância de uma entidade com a semântica, respectivamente, "modificar o recurso" e "remover o recurso". A tabela 3 mostra exemplos relacionados a esses métodos.

| Método | URI | Semântica |
|---|---|---|
| POST | /usuarios | "adicionar **um usuário** à coleção de **usuários**" |
| POST | /congresso/8/edicoes | "adicionar **uma edição** à coleção de **edições** do **congresso** identificado por **8**" |
| PUT | /usuario/38 | "modificar o **usuário** identificado por **38**" |
| DELETE | /congresso/8 | "remover o **congresso** identificado por **8**" |

Tabela 3: Exemplos de uso dos métodos POST, PUT e DELETE em relação aos recursos

## 7.4 Representações

As representações, quando aplicáveis, estão na forma de documentos XML. Segundo Harold e Means (2004), XML é uma linguagem de meta-marcação para documentos de texto. Dados são incluidos em um XML como pedaços de textos e envolvidos por marcações que os

descrevem. A unidade básica de dados e marcação do XML é chamada *elemento*. A especificação do XML define a sintaxe exata que as marcações devem seguir: como elementos são delimitados por etiquetas, com o que uma etiqueta se parece, quais nomes são aceitos como elementos, onde os atributos são colocados, e assim sucessivamente.

Neste sistema são utilizados dois modelos de documentos XML, um utilizado nos recursos de coleções e outro nos recursos de instância.

### 7.4.1 Representações de Coleções

Os documentos XML que representam coleções do sistema são compostos de uma marcação raíz, que o identifica como sendo uma coleção, chamada "lista". Esta marcação raíz possui como filhos marcações que representam um item da lista, sendo uma marcação para cada item da lista.

A marcação raíz possui dois atributos chamados "de" e "url". O atributo "de" especifica o tipo dos elementos listados e o atributo "url" é uma URI raíz para recursos daquele tipo. Assim como a marcação raíz, uma marcação que indica uma instância de um elemento da lista possui um atributo chamado "url" que é uma ligação para um determinado recurso presente na lista. Desta forma, a URI completa para um recurso presente nesta lista pode ser montada através da concatenação entre os atributos "url" da marcação raíz e de um elemento da lista.

Listagem 7.1: Exemplo de representação XML de uma coleção

```
<lista de="usuarios" url="http://localhost:8080/usuario/">
        <usuario url="1"/>
        <usuario url="2"/>
        <usuario url="3"/>
        <usuario url="4"/>
</lista>
```

### 7.4.2 Representações de Instâncias

A representação XML de uma instância é formada por um elemento raíz, que tem o nome em referência ao tipo do recurso, composto por vários outros elementos filhos que encapsulam as diversas informações pertinentes a um recurso. O elemento raíz possui ainda um atributo chamado "url" que indica qual a URI exata para acesso ao recurso em questão.

Listagem 7.2: Exemplo de representação XML de uma instância

```
<usuario url="http://localhost:8080/usuario/1">
        <codigo>1</codigo>
        <usuario>admin</usuario>
        <senha>b25f3ab3fd05e97661df7c0bf679fbe4c241d7fb</senha>
        <primeiroNome>admin</primeiroNome>
        <nomeDoMeio>xpto</nomeDoMeio>
        <correioEletronico>xpto@edugraf.ufsc.br</correioEletronico>
        <dataDeRegistro>2007-12-20 10:07:56.0</dataDeRegistro>
        <dataDoUltimoLogin>2008-04-16 15:17:10.0</dataDoUltimoLogin>
</usuario>
```

## 7.5 Ligação entre Recursos

As ligações entre recursos são informadas ao cliente através de elementos específicos embutidos nas representações de cada recurso, quando aplicável. O exemplo da listagem 7.3 demonstra a ligação entre um "congresso" e a coleção de edições deste congresso através da marcação "edicoes". Ainda, a figura 9 ilustra o grafo das ligações entre os recursos disponíveis no sistema.

Listagem 7.3: Exemplo da representação de ligação entre recursos

```
<congresso url="http://localhost:8080/congresso/2">
        <codigo>2</codigo>
        <titulo>Um Congresso</titulo>
        <descricao/>
        <edicoes url="http://localhost:8080/congresso/2/edicoes/"/>
</congresso>
```

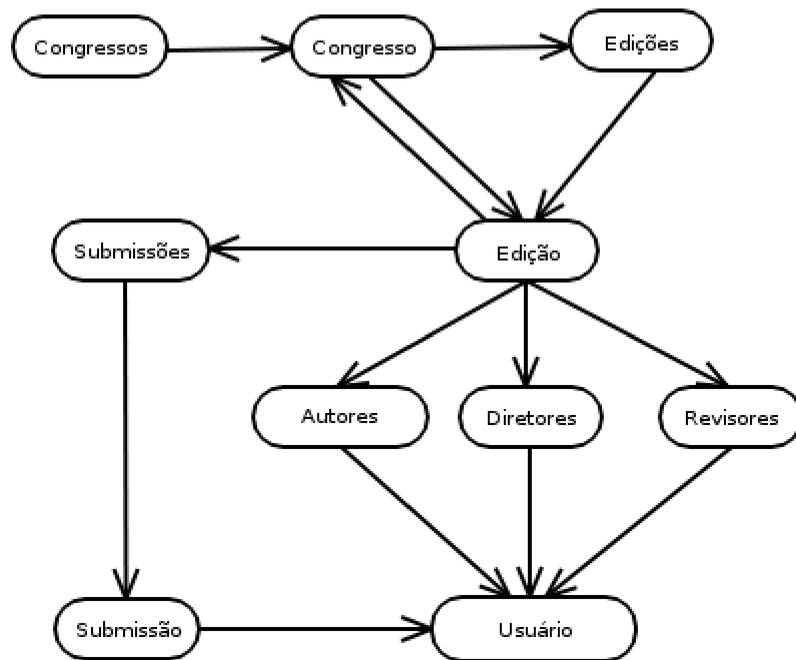


Figura 9: Grafo das ligações entre os recursos do sistema

# 8 Conclusão

O desenvolvimento de uma interface REST para o Open Conference Systems permitiu atender os objetivos estabelecidos nestes trabalho, são eles: conhecer a arquitetura REST e a aplicação Open Conference Systems, analisar esta aplicação sobre a perspectiva REST e propor uma interface que segue o estilo REST para este sistema.

Para o desenvolvimento e implementação da interface foi necessário procurar o estado da arte em REST, nos quais os pressupostos de Fielding (2000) ao apresentar o estilo REST e Richardson e Ruby (2007) pela Arquitetura Orientada a Recursos formaram o eixo que norteou esta aplicação.

Com o intuito de analisar a possibilidade de aplicar o estilo REST no OCS foi necessário conhecer a arquitetura e os parâmetros, os papeis e as funções do sistema já existente.

O próximo passo foi integrar o estilo REST com as funções exercidas pelo OCS, compatibilizando fundamentos opostos - como a existência/ausência de estados armazenados no servidor - e transformar os serviços e funções do OCS em recursos REST, onde são utilizadas a Interface Uniforme, as Representações e as Ligações entre Recursos.

A utilização do ROCS permitirá que aplicações compatíveis com o estilo REST e a Arquitetura Orientada a Recursos utilizem os recursos providos pelo sistema OCS, possibilitando uma gama de interações que só tem a acrescentar na maneira como as informações serão manipuladas.

Acredita-se que este trabalho terá relevância para a academia por se tratar de um assunto atual e de interesse à comunidade científica e aos profissionais de Ciências da Computação por apresentar informações técnicas e científicas sobre a arquitetura de software baseada no estilo REST.

# *Referências Bibliográficas*

ALUR, D.; CRUPI, J.; MALKS, D. *Core J2EE Patterns: Best Practices and Design Strategies*. 2. ed. Englewood Cliffs: Prentice Hall, 2004.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Boston: Addison-Wesley, 2003.

FIELDING, R. *Architectural Styles and the Design of Network-based Software Architectures*. 100 p. Tese (Doutorado) — University of California, 2000.

FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2003.

HAROLD, E. R.; MEANS, W. S. *XML in a Nutshell*. 3. ed. [S.l.]: O'Reilly Media, Inc, 2004.

LIM, J. *ADOdb Database Abstraction Library for PHP (and Python)*. 2008. Disponível em: <http://www.slac.stanford.edu/BFROOT/www/Physics/Tools/Pid/Electrons/index.html>. Acesso em: 06 mai. 2008.

NEW DIGITAL GROUP INC. *Is Smarty right for me?* 2008. Disponível em: <http://www.smarty.net/rightforme.php>. Acesso em: 06 mai. 2008.

NOELIOS CONSULTING. *Restlet 1.0 - Tutorial*. 2008. Disponível em: <http://www.restlet.org/documentation/1.0/tutorial>.

PUBLIC KNOWLEDGE PROJECT. *OCS FAQ*. 2007. Disponível em: <http://pkp.sfu.ca/ocs_faq>. Acesso em: 28 abr. 2008.

PUBLIC KNOWLEDGE PROJECT. *OCS In An Hour*. 2. ed. [S.l.], 2007.

PUBLIC KNOWLEDGE PROJECT. *Open Journal Systems Technical Reference*. 3. ed. [S.l.], 2007.

RICHARDSON, L.; RUBY, S. *RESTful Web Services*. 1. ed. Sebastopol: O'Reilly Media, Inc, 2007.

WORLD WIDE WEB CONSORTIUM. *URIs, Addressability, and the use of HTTP GET and POST*. Disponível em: <http://www.w3.org/2001/tag/doc/whenToUseGet.html>. Acesso em: 13 mai. 2008.