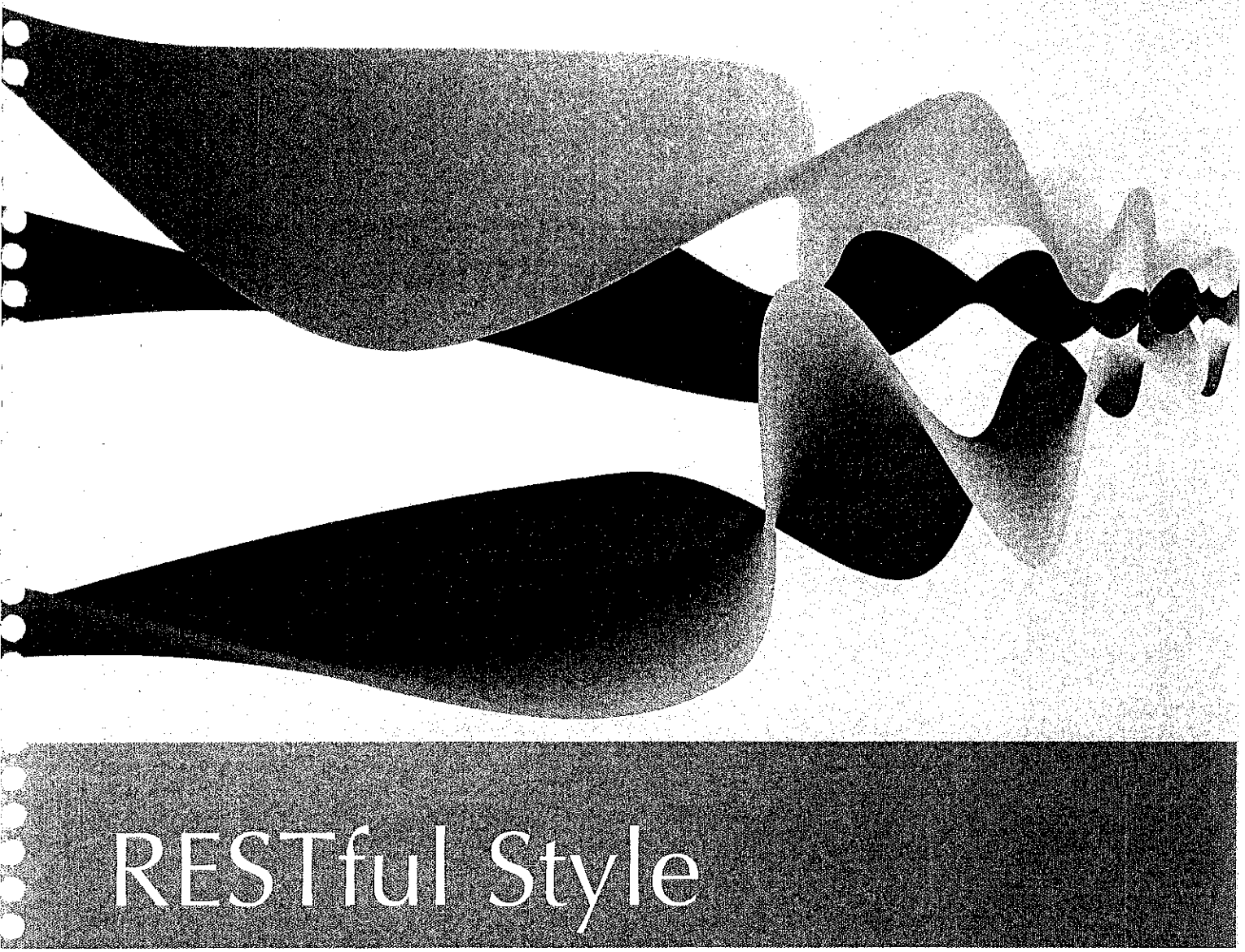


RESTful Style

The Next Generation

soa|expert

soa|expert

ÍNDICE **PÁGINA**

1 O que é REST ?

REST é um estilo de se projetar aplicativos da Web fracamente acoplados que contam com recursos nomeados, em forma de Localizador Uniforme de Recursos (URL), Identificador Uniforme de Recursos (URI) e Nome de Recurso Uniforme (URN), por exemplo, e não com mensagens. Engenhosamente, o **REST** transporta a infraestrutura já validada e bem-sucedida da Web — HTTP, ou seja, o **REST** alavanca aspectos do protocolo HTTP como pedidos **GET** e **POST**. Esses pedidos são perfeitamente mapeados para necessidades de aplicativo de negócios padrão, como: *create, read, update* e *delete* (CRUD), conforme mostrado na Tabela 1:

Tabela 1. Mapeamento CRUD/HTTP

Create	POST
Read	GET
Update	PUT
Delete	DELETE

Ao associar pedidos, que agem como verbos, com recursos, que agem como nomes, você termina com uma expressão lógica de comportamento — **GET** este documento e **DELETE** aquele registro, por exemplo.

Roy Fielding, o verdadeiro pai do REST, afirma em sua dissertação de PhD, que o REST "enfatiza a escalabilidade das interações do componente, a generalidade das interfaces, a implementação independente de componentes e componentes intermediários para reduzir a latência da interação, forçar a segurança e encapsular sistemas legados").

A construção de sistemas RESTful não é difícil, e os sistemas são altamente escaláveis, enquanto são fracamente acoplados aos dados subjacentes; eles também alavancam perfeitamente o armazenamento em **cache.**

Tudo na Web (páginas, imagens, entre outras coisas) são a essência de um recurso. O fato de o REST contar com recursos nomeados em vez de mensagens, facilita o fraco acoplamento no design do aplicativo, pois isso limita a exposição da tecnologia subjacente. Por exemplo, a URL a seguir expõe um recurso sem envolver absolutamente nada da tecnologia subjacente:

http://soaexpert.com.br/2010/03/20/analisando-metricas-sem-ambiguidade/

Esta URL representa um recurso — um artigo chamado "Analisando Métricas sem Ambiguidade." Um pedido deste recurso alavanca o comando HTTP **GET**. Observe que a URL é baseada em substantivo.

Uma versão baseada em verbo (que pode parecer com **http://soaexpert.com.br/2010/03/20/getArticle?name=analisando-metricas-sem-ambiguidade)** violaria os princípios do REST, pois ele contém uma mensagem embutida em forma de getArticle.

Você poderia imaginar a postagem de um novo recurso (quer dizer, um recurso de artigo como http://soaexpert.com.br/2010/03/22/rest-is-good-for-you/) via o comando HTTP POST. Embora você também possa imaginar APIs baseadas em verbo associadas — como createArticle?name=rest-is-good-for-you e deleteArticle?name=rest-is-good-for-you. Essas chamadas interceptam o comando HTTP GET e, em sua maior parte, ignoram a infraestrutura HTTP disponível (e bem-sucedida). Em outras palavras, elas não são RESTful.

O melhor do REST é que os recursos podem ser tudo, e a forma como eles são representados também pode variar. No exemplo anterior, o recurso é um arquivo HTML; portanto, o formato da resposta seria HTML. Mas o recurso poderia ser facilmente um documento XML, um objeto serializado ou uma representação JSON. Isso não importa !

O que importa é que um recurso é nomeado e que a comunicação com ele não afeta seu estado. Não afetar o estado é algo importante, pois interações sem preservação de estado facilitam a escalabilidade.

2. Princípios Chave de REST

REST é um conjunto de princípios que definem como Web Standards como HTTP e URIs devem ser usados. A promessa é que se você aderir a princípios REST enquanto estiver desenhando sua aplicação, você terá um sistema que explora a arquitetura da Web em seu benefício. Em resumo, os cinco princípios fundamentais são:

- Dê a todas as coisas um Identificador (ID)
- Vincule as coisas
- Utilize métodos padronizados (Uniform Interface)
- Recursos com múltiplas representações (XML, JSON)
- Comunique sem estado (Statless)

2.1 Dê a todas as coisas um ID

Estamos usando o termo "coisas" ao invés do termo formalmente correto "**recurso**", porque esse é um principio simples que não deveria ser escondido por trás da terminologia.

Se você pensar sobre os sistemas que as pessoas constroem, há usualmente um conjunto de abstrações chave que merecem ser identificadas. Tudo o que deveria ser identificado deveria obviamente ter um ID - Na Web, há um conceito unificado para IDs: **A URI.** URIs compõe um namespace global e utilizando URIs para identificar seus recursos chave, significa ter um ID único e global.

O principal beneficio de um esquema de nomes consistente para as coisas é que você não tem que criar o seu próprio esquema, você pode confiar em um que já tenha sido definido, trabalha muito bem em escala global e é entendido por praticamente qualquer um.

Se a sua aplicação incluir uma abstração de Cliente, por exemplo, acreditamos que os usuários gostariam de poder enviar um link para um determinado cliente via e-mail, para um colega de trabalho, criar um favorito para ele no seu navegador, ou mesmo anotá-la em um pedaço de papel. Para esclarecer melhor, imagine que decisão de negócio terrivelmente ruim teria sido se uma loja on-line como a Amazon.com, não identificasse cada um dos seus produtos com um ID único (uma URI).

Quando confrontados com essa idéia, muitas pessoas se perguntam se isso significa que devem expor suas entradas de banco de dados (ou seus IDs) diretamente - e frequentemente ficam revoltadas pela simples idéia, uma vez que anos de prática de orientação a objetos nos disseram para esconder os aspectos de bancos de dados. Por exemplo, um recurso de Pedido poderia ser composto de itens de pedido, um endereço e muitos outros aspectos que você poderia não querer expor como um recurso identificável individualmente.

Tomando a idéia de identificar de tudo o que vale a pena a ser identificado, leva à criação de recursos que você normalmente não vê, em um típico design de aplicação: Um processo ou um passo de um processo, uma venda, uma negociação, uma solicitação de quotação - estes são todos exemplos de "coisas" que precisam ser identificadas. Isto, por sua vez, pode levar à criação de entidades mais persistentes do que em um design não RESTful.

Alguns exemplos exemplos de URIs que poderíamos ter:

```
http://soaexpert.com.br/clientes/1234
http://soaexpert.com.br/pedidos/2007/10/776654
http://soaexpert.com.br/produtos/4554
http://soaexpert.com.br/processos/aumento-salario-234
```

Como criamos URIs que possam ser lidas por seres humanos, um conceito útil, mesmo que não seja um pré-requisito para um design RESTful, deve ser bastante fácil de adivinhar seu significado: Elas obviamente identificam "itens" individuais. Mas dê só uma olhada nestes exemplos:

```
http://soaexpert.com.br/pedidos/2007/11
http://soaexpert.com.br/produtos?cor=verde
```

No início, isso pode parecer algo diferente, afinal de contas, eles não são a identificação de uma coisa, mas um conjunto de coisas (assumindo que a primeira URI identifica todos os pedidos submetidos em novembro de 2007, e a segunda, um conjunto de produtos verdes). Mas esses conjuntos são realmente coisas,**recursos**, por si só, e eles definitivamente precisam ter um identificador.

Note os benefícios de se ter um único esquema de nomes em nível global aplicável tanto para a Web em seu navegador, como para comunicação de máquina para máquina.

Para resumir o primeiro princípio: Use **URIs** para identificar tudo o que precisar ser identificado, especifique todos os recursos de "alto nível" que seu aplicativo oferece, se eles representam itens individuais, conjuntos de itens, objetos virtuais e físicos, ou resultados de computação.

2.2 Vincule as coisas

O próximo princípio que veremos tem uma descrição formal que intimida um pouco: "*Hipermídia como motor do estado do aplicativo*", as vezes abrevido como **HATEOAS**.

No seu núcleo está o conceito de hipermídia, ou em outras palavras: links. Links são algo que nós estamos familiarizados no HTML, mas eles não são de forma alguma restritos ao consumo de humanos. Considere o seguinte fragmento XML:

```
  <pedido self="http://soaexpert.com.br/clientes/1234">
     <valor>23</valor>
     <produto
ref="http://soaexpert.com.br/produtos/4554"></produto>
     <cliente
ref="http://soaexpert.com.br/clientes/1234"></cliente>
  </pedido>
```

Se você observar os links de produto e cliente nesse documento, poderá facilmente imaginar como um aplicativo pode interpretar os links para obter mais informações.

A beleza da abordagem de links com URIs é que os links podem apontar para recursos que são oferecidos por uma aplicação diferente, um outro servidor, ou até mesmo em uma empresa diferente em outro continente , porque o esquema de nomes é um padrão global, todos os recursos que compõe a Web podem ser ligados uns aos outros.

Há um aspecto ainda mais importante do princípio de hipermídia: "A parte de estado do aplicativo".

Em suma, o fato de que o servidor (ou provedor de serviços, se você preferir) oferece um conjunto de links para o cliente (o consumidor do serviço), permite ao cliente mudar o aplicativo de um estado para outro, através de um link.

Para resumir esses principios: Use liks para referênciar coisas que possam ser identificadas (recursos) sempre que for possível. Hiperlinks são o que fazem a Web ser a Web.

2.3 Utilize os métodos padrão

Havia um pressuposto implícito na discussão dos primeiros dois princípios: a aplicação consumidora pode realmente fazer algo significativo com URIs. Se você ver uma URI escrita na lateral de um ônibus, você pode inserí-la na barra de endereços no seu navegador e presionar enter - mas como é que o seu navegador sabe o que fazer com a URI?

Ele sabe o que fazer com ela, porque todos os recursos possuem a mesma interface, e o mesmo conjunto de métodos (ou operações, se preferir). O HTTP chama esse verbos, e além dos dois que todo mundo conhece (o GET e o POST), o conjunto de métodos padrão inclui PUT, DELETE, HEAD e OPTIONS.

O significado de cada um desses métodos é definido na especificação do HTTP, juntamente com algumas garantias sobre o seus comportamentos.

Demonstrando isso em OO um cenário HTTP RESTful estende uma classe como essa (em alguma pseudo-sintaxe no estilo Java/C# concentrando-se nos métodos principais):

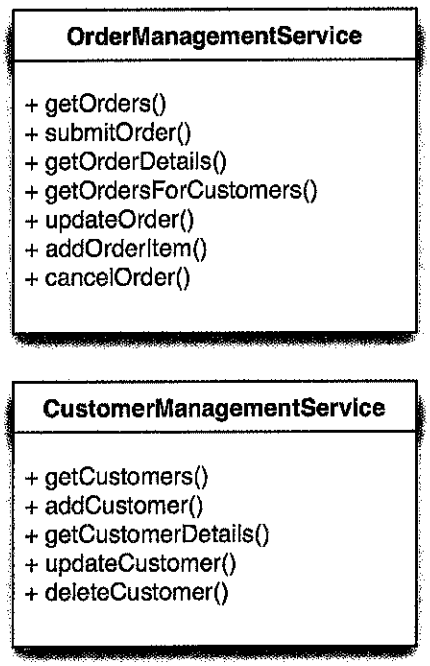
```
class Resource {
   Resource(URI u);
   Response get();
   Response post(Request r);
   Response put(Request r);
   Response delete();
}
```

Devido ao fato de a mesma interface ser usada para todos os recursos, você pode confiar que é possível obter uma representação, ou seja, uma renderização do recurso utilizando GET.

Como a semântica do GET é definida na especificação, você tem certeza das obrigações quando chamá-lo. É por isso que o método é chamado de "seguro". O GET suporta cacheamento de forma muito eficiente e sofisticada, em muitos casos, nem sequer precisa enviar uma requisição ao servidor.

Também podemos ter certeza de que o GET é idempotente, se você emitir uma requisição GET e não obter um resultado, você pode não saber se o seu pedido nunca alcançou o seu destino ou se a resposta foi perdida no caminho. A garantia da idempotência significa que você pode simplesmente emitir a requisição novamente. A idempotência também é garantida para o PUT (que basicamente significa "atualizar esse recurso com base nesses dados, ou criá-lo nessa URI se ainda não estiver lá") e para o DELETE (que você pode simplesmente tentar de novo e de novo até obter um resultado - apagar algo que não existe não é um problema). POST, que normalmente significa "criar um novo recurso", pode também ser utilizado para invocar um processamento arbitrário, portanto, não é nem seguro nem idempotente.

Considere o seguinte exemplo de um simples cenáro de compras:

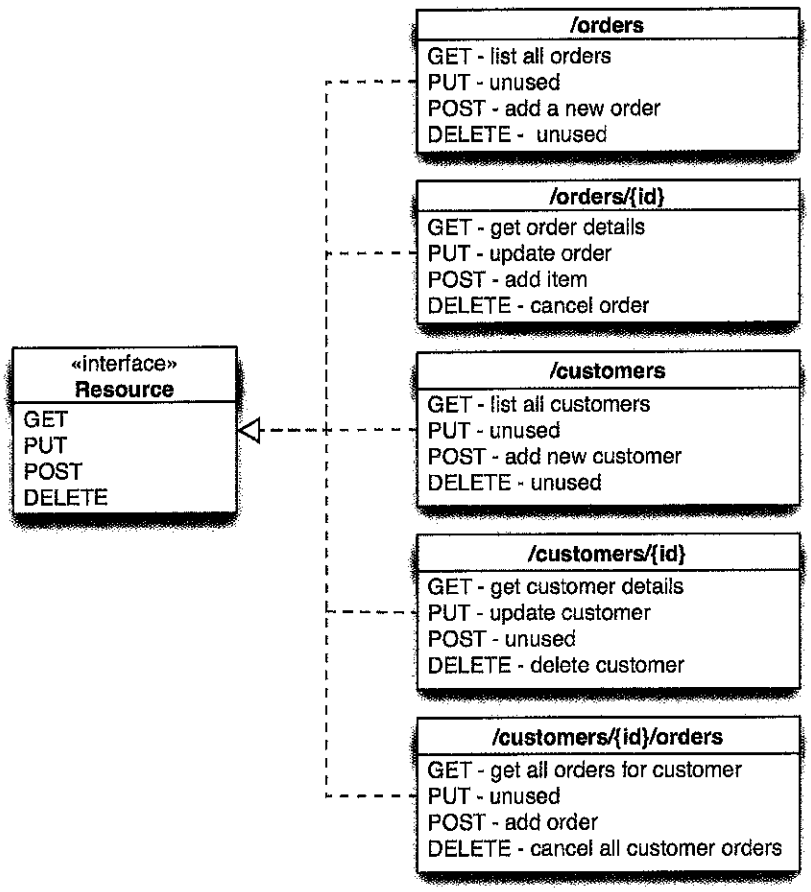


Você pode ver que há dois serviços definidos aqui (sem implicar qualquer implementação tecnológica específica).

As interfaces para esses serviços é específica para a tarefa, OrderManagement (Gerenciador de Pedidos) e CustomerManagement (Gerenciador de Clientes).

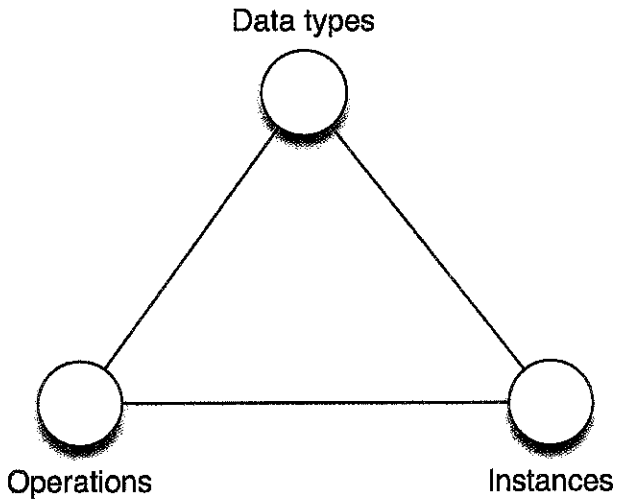
Se um cliente (aplicação) quiser consumir esse serviços, será necessário codificar sobre dessa interface específica - não há como usar um cliente que tenha sido construído antes de que essas interfaces tenham sido especificadas para interagir com elas. As interfaces definem o protocolo dos serviços do aplicativo.

Em uma abordagem HTTP RESTful, você teria que começar por uma inteface genérica que compusesse o protocolo HTTP do aplicativo. Exemplo :



Analisando o exemplo acima, veremos que as operações específicas de um serviço foram mapeadas para os métodos HTTP e para diferenciá-las, criamos um universo inteiro de novos recursos.

O interessante desse exmeplo é que um GET emu ma URI que identifica cliente, é tão significativo como uma operação getCustomerDetails. Algumas pessoas usam um triângulo para visualizar isso:



Imagine as três vértices como maçanetas que você tem que virar.

Na primeira abordagem, há muitas operações e muitos tipos de dados e um número determinado de "instâncias" (essencialmente, tantas quantos os serviços tiverem). Na segunda, você tem um número determinado de operações, muitos tipos de dados e muitos objetos para invocar esse determinado método.

O ponto é ilustrar que, basicamente, se pode expressar qualquer coisa que quiser com ambas as abordagens.

Porque que isto é importante? Essencialmente, isso faz seu aplicativo ser parte da Web - sua contribuição para o que tornou a Web, a aplicação de maior sucesso da Internet é proporcional ao número de recursos que sua aplicação adiciona a ela. Em uma abordagem RESTful, um aplicativo pode adicionar alguns milhões de URIs de clientes na Web;

Se for concebido da mesma forma que os aplicativos dos tempos do CORBA, essa contribuição é frequentemente um único "endpoint" - comprando a uma porta muita pequena que ofereça entrada para

um universo de recursos apenas para aqueles que têm a chave.

A interface uniforme também permite que qualquer componente que entenda o protocolo de aplicação HTTP interaja com seu aplicativo. Exemplos de componentes que são beneficiados por isso, são clientes genéricos como o curl, o wget, proxies, caches, servidores HTTP, gateways e até mesmo o Google, o Yahoo!, o MSN e muitos outros.

Para resumir: Para que clientes possam interagir com seus recursos, eles devem implementar o protocolo de aplicação padrão (HTTP) corretamente, isto é, utilizar os métodos padrão: GET, PUT, POST e DELETE.

2.4 Recursos com múltiplas Representações

Nós ignoramos uma ligeira complicação até agora: Como é que um cliente saberá como lidar com os dados que obtém, por exemplo, como resultado de uma requisição GET ou POST? A abordagem do HTTP, permite uma separação entre as responsabilidades de manipulação de dados e invocação de operações. Em outras palavras, um cliente que sabe como lidar com um formato específico de dados pode interagir com todos os recursos que possam oferecer uma representação nesse formato. Vamos ilustrar isso novamente, com um exemplo. Utilizando a negociação de conteúdo HTTP, um cliente pode solicitar por uma representação em um formato específico:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml
```

O resultado pode ser um XML em algum formato específico de um empresa que representa os dados de um cliente. Se o cliente (HTTP) enviar uma solicitação diferente, isso é, como essa:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

O resultado poderia ser o endereço de um cliente no formato vCard.

Isso ilustra porque, idealmente, as representações de um recurso devem ser em formatos padrão - se um cliente "conhece" ambos, os protocolos de aplicação HTTP e um conjunto de formato de dados, pode interagir com qualquer aplicativo HTTP RESTful do mundo de forma muito significativa.

Infelizmente, nós não temos formatos padronizados para tudo, mas você provavelmente pode imaginar como se poderia criar um ecossistema menor em uma empresa ou em um conjunto de parceiros colaboradores com base nos formatos padrão. Certamente tudo isso não se aplica somente a dados enviados de um servidor para um cliente, mas também na direção oposta - um servidor que pode consumir dados em um formato específico não se preocupa com o tipo específico do cliente, desde que respeite o protocolo do aplicativo.

Há um outro benefício significativo de se ter múltiplos formatos de um recurso na prática: Se você fornecer ambos, um formato HTML e um XML de seus recursos, eles poderão ser consumidos não apenas pelo seu aplicativo, mas também por qualquer navegador web padrão - em outras palavras, a informação do seu aplicativo ficará disponível para qualquer um que sabe como utilizar a Web.

Existe outra forma de explorar isso: Você pode transformar a interface gráfica do seu aplicativo Web em uma API Web - além disso, o Design da API é freqüentemente movido pela ídeia de que tudo que pode ser feito pela interface gráfica deveria também poder ser feito através da API. Combinar as duas tarefas é um boa forma de obter uma melhor interface Web para ambos, seres humanos e aplicativos.

Em resumo: Ofereça diversos formatos dos recursos para diferentes necessidades.

2.4 Comunique sem Estado

Primeiramente, é importante salientar que embora REST inclua a ídeia de "não manter", isso não significa que o aplicativo que exponha suas funcionalidades não possa ter estado, de fato, isso tornaria a abordagem inútil na maioria dos cenários. REST exige que o estado seja transformado no estado do recurso ou mantido no cliente. Em outras palavras, um servidor não deveria guardar o

estado da comunicação de qualquer um dos clientes que se comunique com ele além de uma única requisição.

A razão mais óbvia para isso é escalabilidade , pois o número de clientes que podem interagir com o servidor seria consideravelmente impactado se fosse preciso manter o estado do cliente.

Mas existem outros aspectos que poderiam ser muito mais importantes: As restrições de "não manter", isolam o cliente de mudanças no servidor, dessa forma, duas solicitações consecutivas não dependem de comunicação com o mesmo servidor. Um cliente poderia receber um documento contendo links do servidor, e então faria algum processamento, e o servidor poderia ser desligado, e o disco rígido poderia ser jogado fora ou substituído, e o software poderia ser atualizado e reiniciado e se o cliente acessase um dos links recebidos do servidor, ele não teria nem percebido.

3. Modelo de Maturidade REST

Como alcançar REST? Quão maduro é a minha aplicação em relação a REST?

De acordo com o modelo, nível 3 adiciona suporte a hipermídia, aperfeiçoando um sistema através do uso de linked data – um requerimento fundamental para REST. Mas HATEOAS sozinho não implica em REST, como o próprio Roy já mencionou.

Lembremos como RMI e objetos distribuídos permitiam a navegação dentre objetos e seus estados? O exemplo a seguir ilustra tal situação:

```
orders = RemoteSystem.locate().orders();

System.out.println(orders.get(0).getProducts().get(0));

receipt = order.payment(payment_information);

System.out.println(receipt.getCode());
```

Mas e se o código acima for uma sequência de invocações remotas de EJB? Se navegando através de relacionamentos (linked data) é REST, implementar o código acima a través de HTTP também seria

(fora a falta de uniform interface).

O modelo se aproxima de REST no lado do servidor, descrevendo a importância da semantica e de media types.

O que ficou faltando?

O exemplo anterior não inspecionou a representação atual e relacionamentos antes de decidir qual seria o próximo passo e isso cria um alto acoplamento inexistente em aplicações REST. Contem ainda um conjunto fixo de instruções a serem seguidas, independentemente de qualquer resposta do servidor.

O cliente não se adapta, não responde ao resultado de suas interações, mas assume um processo fixo.

Se a API fosse http e o servidor respondesse com "Server too busy", um cliente REST de verdade tentaria novamente em 10 minutos, o que os clientes tradicionais que escrevemos não fazem.

Mesmo alcançando o máximo de maturidade no modelo de Richardson, a aplicação que envolve cliente e servidor, não se adapta ao estado do recurso, ainda não é REST: seu cliente ainda possui um alto acoplamento.

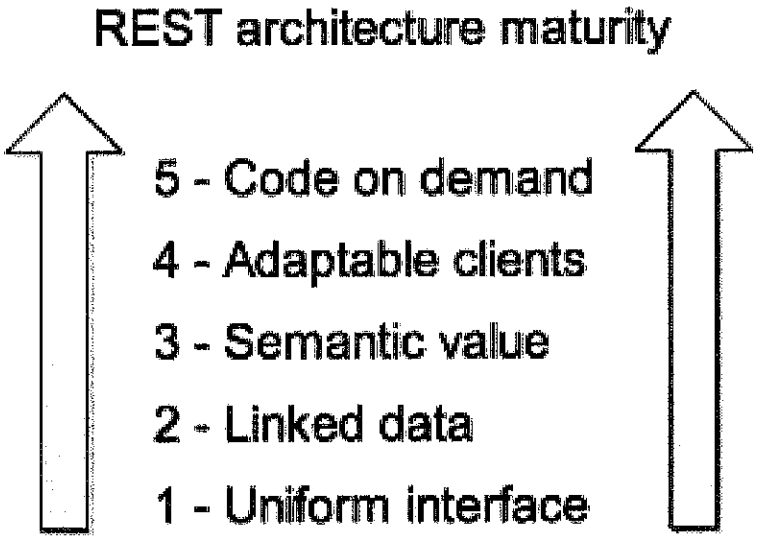
Por esse motivo **não existe tal serviço como REST web service**, mas para ter uma aplicação REST, tanto o cliente quanto o servidor devem seguir suas características arquiteturais.

Modelo de Richardson e HTTP

Relações ricas em semânticas podem ser compreendidas por clientes e são fundamentais para a criação de um sistema REST. Um ponto importante é lembrar que o modelo continua sendo muito importante para demonstrar a maturidade de uma arquitetura em cima do uso de HTTP, mas ainda não impede o alto consumo financeiro vindo do uso do "mindset de web services" de maneira mais elegante, mas com o mesmo custo de acoplamento.

Um modelo de maturidade REST

Por todos esses motivos, proponho um modelo de maturidade independente do protocolo, onde podemos medir o uso dos aspectos de REST em um sistema:



Rest Architecture Maturity

Para atingir REST, o primeiro passo é a utilização de uma **uniform interface**: um conjunto padrão de ações que podem ser executadas em todos os recursos existentes. Por exemplo, em HTTP, os verbos definem a interface uniforme, o que pode ser efetuado com cada recurso.

O segundo passo é a adoção de linked data para permitir ao cliente **navegar entre estados e recursos** de maneira uniforme. Através de HTTP, como o modelo de Richardson descreve, este é o uso de hypermídia como connectedness.

O terceiro passo é adicionar valor semântico aos links. Relações definidas como "related" podem possuir valor semântico em determinados processos, menos valor em outros. "payment" pode fazer senido para outros. A criação e adoção de media types permitem, mas não implicam, em código sendo escrito de maneira menos acoplada.

O quarto passo é criar clientes de maneira que suas decisões sejam baseadas nos relacionamentos e estado de recursos, além do conhecimento do media type. É o media type e seus relacionamentos que criam o acoplamento entre clientes e servidores, qualquer outro acoplamento vem de arquiteturas que não REST.

Todos os passos acima devem ser tomados para minimizar o acoplamento do cliente ao servidor - e vice versa.

O último passo é a evolução do cliente. Code on demand permite educar clientes como se comportar em situações específicas que não foram previstas, como por exemplo ao encontrar um novo media type desconhecido.

Apêndice A – Introdução à WebSemântica - RDF

A organização da imensa vastidão de conteúdo disponível atualmente na Internet, de uma forma simples, eficiente e focada em nossas necessidades, se tornou um problema. Dessa necessidade surgiu a Web Semântica.

Sua principal aplicação se refere à capacidade de os sistemas computacionais interpretarem o conteúdo disponível nos sites da Internet e conseguir entender de forma diferenciada uma página em que a palavra bala é um doce ou é um projétil de armas, por exemplo. Ou seja, o conteúdo é interpretado de acordo com seu contexto .

A Web Semântica representa a evolução da Web atual, enquanto a Web tradicional foi desenvolvida para ser compreendida somente pelos usuários, a Web Semântica está sendo projetada para ser entendida também pelas máquinas. Este entendimento se dá na forma de agentes computacionais, que são capazes de operar eficientemente sobre as informações, podendo entender seus significados. Portanto, auxiliando os usuários em operações na Web.

A Web Semântica tem como objetivo incorporar semântica às informações. Com isso, não somente os usuários entenderão as informações, como também as máquinas. Ela pretende fornecer estruturas e dar significado semântico ao conteúdo das páginas Web, criando assim um ambiente onde agentes de software possam trabalhar em conjunto com o usuário.

Algumas tecnologias foram desenvolvidas para a Web Semântica, tais como o XML, linguagem de marcação que permite aos usuários criarem *tags* personalizadas sobre o documento criado. Outra tecnologia utilizada pela Web Semântica é o RDF, que trabalha com um trio de informação o qual expressa o significado das informações. Cada componente do trio tem sua própria finalidade, em analogia ao sujeito, verbo e objeto de uma frase e recebe uma identificação URI. Estas tecnologias são fundamentais para o funcionamento da Web Semântica.

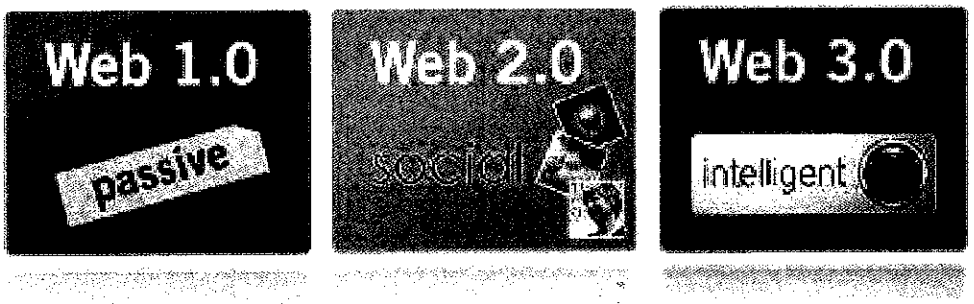
2.Web Semantica

A Web Semântica consiste na materialização da proposta de Tim Berners-Lee, o criador da Web de dotar a Web com uma representação semântica compartilhada, de uma forma que pudesse ser interpretada simultaneamente por seres humanos e máquinas, permitindo assim a inferência automática de conteúdo, futuros estados e ações.

De acordo com Berners-Lee, o modo mais simples de explicar a Web Semântica é o seguinte: "No seu computador você tem seus arquivos, os documentos que você lê, e existem arquivos de dados como agendas, programas de planejamento financeiro, planilhas de cálculo. Estes programas contêm dados que são usados em documentos fora da web. Eles não podem ser colocados na Web".

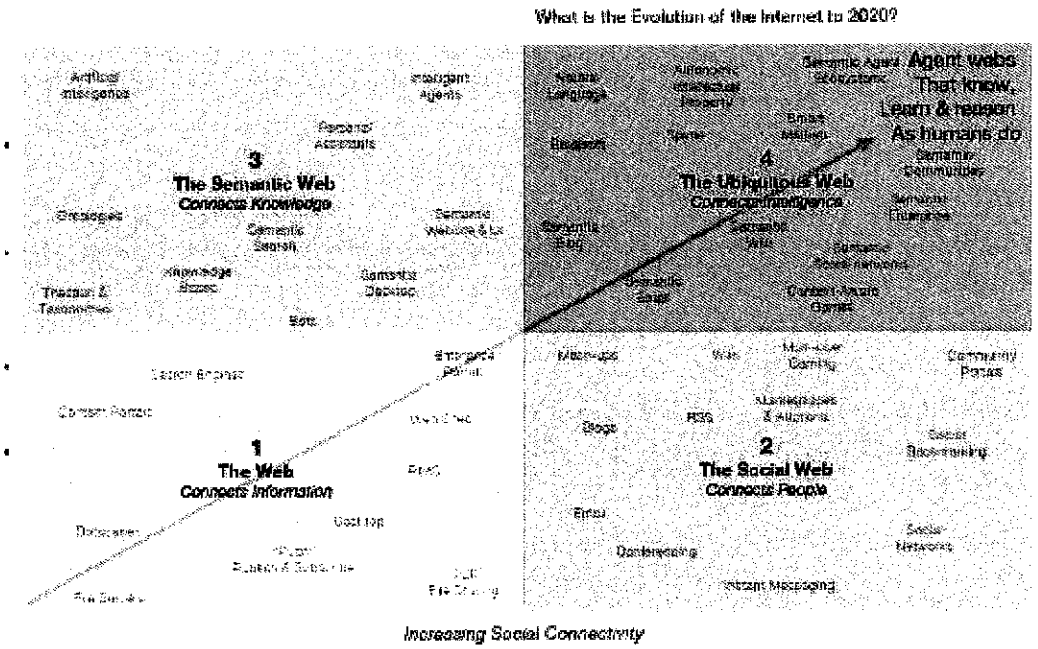
"A Web Semântica é uma extensão da Web atual, onde a informação possui um significado claro e bem definido, possibilitando uma melhor interação entre computadores e pessoas". Deste modo, é evidente que o objetivo final da Web Semântica é atender as pessoas e não os computadores, mas para isso torna-se necessário construir instrumentos que forneçam sentido lógico e semântico para as máquinas. Assim, pode-se verificar que a Web Semântica é uma tentativa inversa de solução, comparando-se com as tradicionalmente desenvolvidas, onde a idéia é pensar nas máquinas para que estas possam servir aos humanos de maneira mais eficiente.

A Web Semântica representa a evolução da Web atual. A Web tradicional foi desenvolvida para ser entendida apenas pelos usuários, já a Web Semântica foi idealizada para ser compreendida também pelas máquinas. Para isso utiliza diversas tecnologias, que são capazes de operar de maneira eficiente sobre as informações, podendo entender seus significados, assim, auxiliando os usuários em operações na Web . Ela tem como objetivo fornecer estruturas e dar significado semântico ao conteúdo das páginas Web, criando um ambiente cooperativo, onde agentes de software e usuários possam atuar juntos.

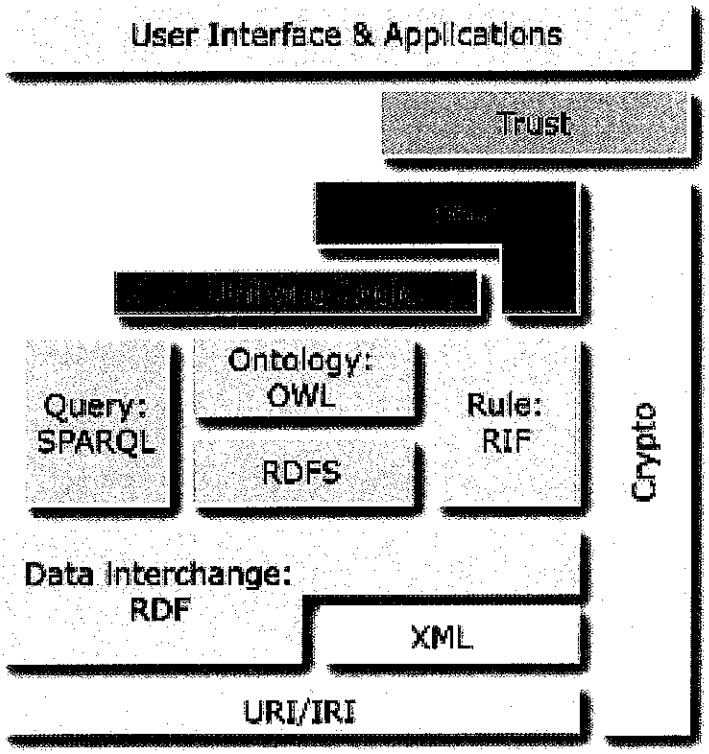


A figura abaixo, mostra a evolução da Web, com uma projeção até o ano de 2020. O eixo horizontal mostra o aumento da conectividade social, enquanto o eixo vertical mostra o aumento da conectividade de conhecimento e raciocínio.

Começa no canto inferior esquerdo, mostrando a Web, que conecta informações. No canto inferior direito está e Web Social, que conecta pessoas. No canto superior esquerdo se encontra a Web Semântica, que conecta conhecimento. E por fim, no canto superior direito está a Web Ubíqua, que conecta inteligência. Dentro de cada quadro se encontram termos descrevendo características de casa tipo de Web.



A Web Semântica é construída de acordo com princípios que são implementados em camadas de tecnologias e de padrões Web. Na figura abaixo, podemos observar as camadas da Web Semântica.



A camada Unicode/URI (*Uniform Resource Identifier*) fornece a interoperabilidade em relação à codificação de caracteres e ao endereçamento e nomeação de recursos da Web Semântica.

O Unicode é um padrão de codificação para fornecer uma representação numérica universal e sem ambigüidade para cada caractere de maneira independente da plataforma de software e do idioma. O URI (*Uniform Resource Identifier*) é um padrão para identificar um recurso físico ou abstrato de maneira única e global.

Um identificador URL (*Uniform Resource Locator*) é um caso específico de URI, formado pela concatenação de seqüências de caracteres para identificar o protocolo de acesso ao recurso, o endereço da máquina na qual o recurso pode ser encontrado e o próprio recurso em questão. É o elemento básico das estruturas a partir dos quais os demais componentes são construídos.

A camada XML, com as definições de *namespace* e *schema*, torna possível a integração das definições da Web Semântica com outros padrões baseados em XML. XML também permite a criação de novas *tags* para atender aplicações específicas. Fornece a interoperabilidade em relação à sintaxe de descrição de recursos da

Web Semântica. A *Extensible Markup Language* (XML) é uma linguagem para representação sintática de recursos de maneira independente de plataforma. Os documentos que tem sua estrutura e conteúdo representados na linguagem XML são denominados de documentos XML. A XML *Schema* é uma linguagem de definição para descrever uma gramática (ou esquema) para uma classe de documentos XML. A linguagem XML *Schema* fornece elementos para descrever a estrutura e restringir o conteúdo de documentos XML.

Os espaços de nomes (*namespaces*) fornecem um método para qualificar os nomes de elementos e atributos, utilizados nos documentos XML, através da associação destes nomes com os espaços de nomes identificados por referências de URI. Os espaços de nomes são úteis para distinguir entre dois elementos definidos com um mesmo nome, mas que pertencem a esquemas diferentes. Alem disso, um documento pode associar elementos previamente definidos à sua estrutura, desde que utilize referencias aos esquemas que definem esses elementos.

As camadas RDF e RDF *Schema* tornam possível a definição de vocabulários que podem ser atribuídos por URIs e também possibilitam estabelecer declarações sobre objetos com URIs. A camada RDF permite a representação de metadados e é acessível por máquinas. RDF Schema é um modelo de tipos de dados simples que permite a criação de classes e propriedades. Permite obter estruturas de informação sem ambigüidades. Fornece um *framework* para representar informação (metadados) sobre recursos. As principais especificações do *Resource Description Framework* (RDF) abrangem um modelo de dados (para expressar declarações sobre os recursos), uma sintaxe baseada em XML (para o intercâmbio das declarações) e uma linguagem de definição de esquemas para vocabulários.

A RDF fornece um modelo de dados fundamentado na idéia de expressar declarações simples sobre recursos; cada declaração consiste de uma tripla (sujeito, predicado, objeto).

Por exemplo na declaração: "a data de criação da página http://soaexpert.com.br/Aulas/IIS é

13/03/2010"

Sujeito: "http://soaexpert.com.br/Aulas/IIS"

Predicado : "data de criação"

Objeto : "13/03/2010"

Um conjunto de triplas (ou declarações) é chamado de grafo RFD, que pode ser ilustrado como um diagrama de nós e arcos orientados, no qual cada tripla é representada como uma ligação nó-arco-nó. O RDF fornece uma sintaxe baseada na linguagem XML, denominada de ***RDF/XML***, para realizar o intercâmbio desses grafos.

Alem do modelo e da sintaxe, a RDF também fornece uma linguagem, denominada ***RDF/Schema***, para a definição de esquemas para os vocabulários (termos) utilizados nas declarações. A *RDF-Schema* estende a especificação básica do RDF para permitir a definição de vocabulários. Assim, o *RDF-Schema* é uma linguagem mínima para a representação de Ontologias simples. Basicamente, essa linguagem fornece o suporte necessário para descrever classes e propriedades, e também para indicar quais propriedades são utilizadas para a descrição de uma classe.

A camada denominada Ontologia fornece suporte para a evolução de vocabulários e para processar e integrar a informação existente sem problemas de indefinição ou conflito de terminologia e também pode definir relações entre conceitos diferentes. A linguagem *RDF-Schema* permite a construção de ontologias com expressividade e inferência limitadas, pois fornece um conjunto básico de elementos para a modelagem, e poucos desses elementos podem ser utilizados para inferência. A *Web Ontology Language* (OWL) estende o vocabulário da RDF *Schema* para a inclusão de elementos com maior poder com relação a expressividade e inferência. Alem disso, a linguagem OWL fornece três módulos (ou dialetos), *OWL Lite, OWL DL* e *OWL Full*, para permitir o uso da linguagem por aplicações com diferentes requisitos de expressividade e inferência. O desenvolvedor pode escolher o módulo OWL adequado, de acordo com os requisitos da sua aplicação. Esta camada é o núcleo da Web Semântica, que não pode ser construída sem ela.

Apêndice B - Microformats

O que são Microformats ?

Microformats é uma série de especificações que tem como foco principal relacionar a informação ou os dados com os humanos em primeiro lugar e em segundo com as máquinas. É uma nova maneira de se pensar sobre dados. Esta série de especificações constitui um "dicionário" de conteúdo semântico para (X)HTML, que tem como base os web standards e é escrito para descrever a informação de forma mais simples possível.

A função destas especificações é enriquecer a informação inserida em páginas web com *meta* informação. "Meta" é uma palavra de origem grega que significa "além de" (beyond) e é usada geralmente como prefixo em palavras que indicam conceitos que explicam ou falam de outros conceitos. Esta é a função das meta tags, fornecer informações que "estão além" daquilo que é visto em um primeiro momento, ou seja, o próprio conteúdo. Mas microformats se refere a descrever trechos de conteúdos específicos de um site, como datas de eventos, informações de contato, descrição de links etc, coisas que estão além do escopo das conhecidas meta tags.

Ou seja, tratando "pequenos formatos" (micro + format) de dados (informação) válidos no código do seu XHTML é possível enriquecer a maneira com a qual lidamos com a informação e a maneira pela qual as máquinas armazenam, indexam, organizam e relacionam toda essa meta?informação. A função dessas especificações é fornecer o máximo de meta-informação sobre o conteúdo que você insere, ou seja, descrevendo os seus próprios dados.

Por que Microformats?

Pense na quantidade de informação que existe hoje circulando na web. Pense nos seus e-mails, nos comentários em blogs, nos artigos, nas fotos, versões de arquivos, textos, documentos, arquivos de áudio e vídeo, feeds etc. Para organizar toda essa informação com precisão é preciso padrões.

Os mecanismos de busca e os spiders possuem um algoritmo cuja função principal é verificar o que é mais relevante, de forma a criar relações entre as informações obtidas sobre cada documento da web, que por sua vez geram ratings e rankings baseados nestas inter-relações.

Parte desse critério de avaliação é relacionar a meta-informação com a própria informação. Quando você digita uma palavra no Google, ele vai retornar uma lista do que é mais relevante relacionado com a palavra que você digitou. As tags do HTML/XHTML são relativamente muito limitadas para descrever todos os diferentes tipos de informação que nós inserimos na web. Microformats tenta suprir esta lacuna estendendo para um outro nível as possibilidades de descrever esta informação.

Qual o ambiente propício para Microformats? Como posso aplicar isto?

Para aplicar Microformats é preciso aprender antes de mais nada a utilizar um código semanticamente correto. A primeira descrição de dados real começa pelo uso das tags certas com o objetivo pelo qual elas foram criadas. Veja em outro artigos sobre como definir isso, qual a melhor maneira de exibir dados tabulares, listagens, headings etc.

Modularidade

Com um código bem formatado e tags usadas corretamente, está lançado o ambiente perfeito para você aplicar microformats em trechos específicos de códigos para resolver questões específicas.

Praticamente em tudo que você for ler sobre Microformats, uma coisa que deve estar bem clara na sua mente é o conceito de modularidade. Pense que uma página escrita em (X)HTML é feita de diversas partes e de diversos trechos distintos de código. Temos links, parágrafos, listagens, tabelas (sim, por que não?), abreviações, endereços, títulos e subtítulos e por aí vai. Compare isso com várias peças de Lego encaixadas uma nas outras. Imagine que você pode arrancar ou colocar outras peças sem descaracterizar o restante ou ter que fazer tudo do zero. Isso é modularidade. Se um trecho está errado, jogue o trecho fora e coloque outro no lugar. Simples assim!

Microformats te fornece várias peças diferentes, para problemas específicos, que podem ser colocadas onde for necessário sem que você precise refazer alguma coisa do início.

A W3C dedicou toda uma documentação chamada de Modularization of XHTML (Modularização do XHTML), que serve de base para poder compreender que modularidade está no sangue do XHTML. Nesta documentação você vai encontrar vários tópicos terminados com a palavra "module", que traduzido do inglês significa "módulo" ou "unidade". Ou seja, são dezenas de unidades (módulos, peças) individuais que podem fazer parte de um documento XHTML, seja ele complexo ou simples. As especificações Microformats se aproveitam desse conceito e oferece soluções padronizadas e semânticas prontas para serem encaixadas onde for necessário. Se você tem um site e se o código dele é semanticamente correto, basta você implementar e tratar estes trechos específicos sem precisar de alterações estruturais.

O manifesto Microformat é o seguinte:

- Criado para resolver problemas específicos.
- Ser o mais simples possível
- Criado primeiro para humanos e máquinas em segundo lugar
- Reutilizar blocos de código o quanto for necessário, utilizados dentro dos padrões web.
- Aplicação em blocos e trechos específicos (modularidade)

Microformats na prática

Se você entendeu todo este conceito até aqui, é mais fácil entender como usar. Como uma série de especificações, esta também possui padrões a serem seguidos. Microformats já possui algumas especificações estabelecidas e alguns drafts que ainda não foram completamente estabelecidos. A série de especificações é:

- hCalendar
- hCard
- rel-license

- rel-nofollow
- rel-tag
- VoteLinks
- XFN (*http://gmpg.org/xfn/*) (see also: xfn?implementations)
- XMDP (*http://gmpg.org/xmdp/*)
- XOXO

Estas especificações oferecem aos autores padrões suficientes para descreverem suas informações de forma mais consistente possível. Elas são aplicadas combinando tags e nome de classes para descrever a informação. Veja o exemplo da aplicação do hCalendar abaixo

```
<span class="vevent">
<a class="url" xhref="http://www.web2con.com/"
mce_href="http://www.web2con.com/"  >
<span class="summary" >Web 2.0 Conference</span>:

<abbr class="dtstart" title="2005?10?05">October
5</abbr>?
<abbr class="dtend" title="2005?10?08" >7</abbr>,
at the <span class="location" >Argent Hotel, San
Francisco, CA</span>

</a>
</span>
```

O exemplo acima pode ser renderizado da seguinte maneira:

Web 2.0 Conference: October 5?7, at the Argent Hotel, San Francisco, CA

Se for a primeira vez que você está vendo isso, talvez sua reação seja a mesma que eu tive quanto vi pela primeira vez: espanto. Através de nome de classes padronizados, é possível ampliar a descrição daquilo que está sendo inserido combinando tags semanticamente corretas com classes. A aplicação de microformats gira basicamente em torno disso: criar relações descritivas entre conteúdo e meta informação inserida de forma que comibine tags semanticamente corretas, classes e atributos de HTML.

A propriedade de poder ser uma "função genérica" que os nomes das classes podem agregar já era conhecida dos desenvolvedores mais experientes desde o CSS1 – note que além de nomes de classes serem utilizadas para chamar um seletor em CSS , elas também são utilizadas "para finalidades gerais processadas por user agents" ? utilizando nomes de classes padronizadas para ampliar a descrição da informação. É preciso lembrar também que as classes em um primeiro momento são "sem significado" definido (ao contrário das tags de XHTML) tanto quanto tags em XML.

Veja um exemplo mais simples para descrever uma licensa utilizando a especificação rel?license apenas aplicando o atributo "rel" no link:

```
<a xhref="http://creativecommons.org/licenses/by/2.0/"
mce_href="http://creativecommons.org/licenses/by/2.0/"
rel="license" >
cc by 2.0
  </a>
```

O atributo "rel" deve ser usado para indicar a relação que o site que insere o link tem com o site referenciado. Para isso foi criada uma série de profiles distintos que representam diferentes tipos de relação que o seu site pode ter com os sites que você linka.