Web Services
RESTful Web Services

SCOMRED, November 2017

Desenvolvimento de aplicações distribuídas

Uma aplicação distribuída é constituída por objetos e classes que se encontram dispersos por vários nós de rede, consequentemente as interações entre os vários objetos envolvem necessariamente comunicações através da rede.

Uma arquitetura de aplicação distribuída gera vários desafios:

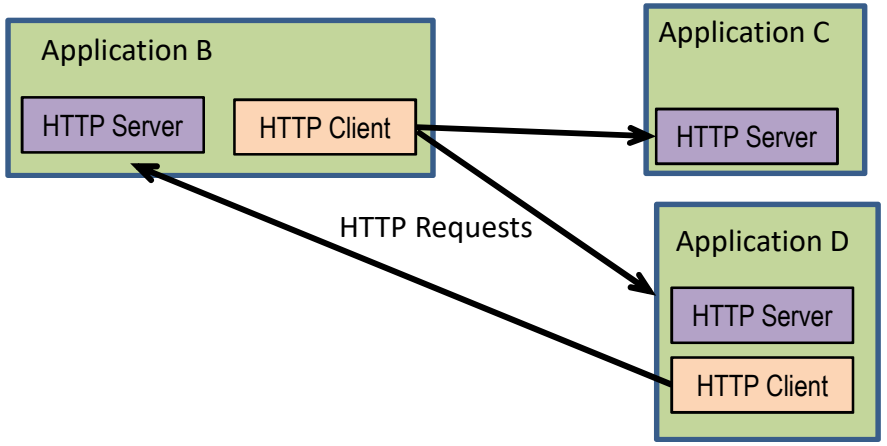
- **Protocolo de aplicação** – especifica de que forma um objeto comunica (interage) com outro objeto residente num nó diferente. Uma abordagem tradicional é o RPC (*Remote Procedure Call*), mas atualmente utiliza-se cada vez mais o HTTP para este efeito (*Web Services*).
- **Representação de dados** – os dados transferidos entre objetos residentes em diferentes nós devem ser representados de forma independente dos sistemas operativos e linguagens de programação.
- **Identificação de objetos** – se numa aplicação não distribuída os objetos são facilmente referenciados por um nome, num ambiente distribuído envolvem direta ou indiretamente a identificação do nó de rede em que se encontram.

Web Services

O HTTP é um protocolo standard com flexibilidade suficiente para ser usado na implementação de aplicações em arquitetura distribuída.

O HTTP permite a transferência de conteúdos de qualquer tipo entre clientes e servidores, para ser usado numa arquitetura de aplicação distribuída basta que os objetos envolvidos assumam papeis de clientes HTTP e servidores HTTP.

O termo ***Web Services*** refere-se a acessos disponibilizados através de servidores HTTP embebidos nas aplicações, esses acessos serão usados por aplicações que assumem o papel de cliente HTTP.



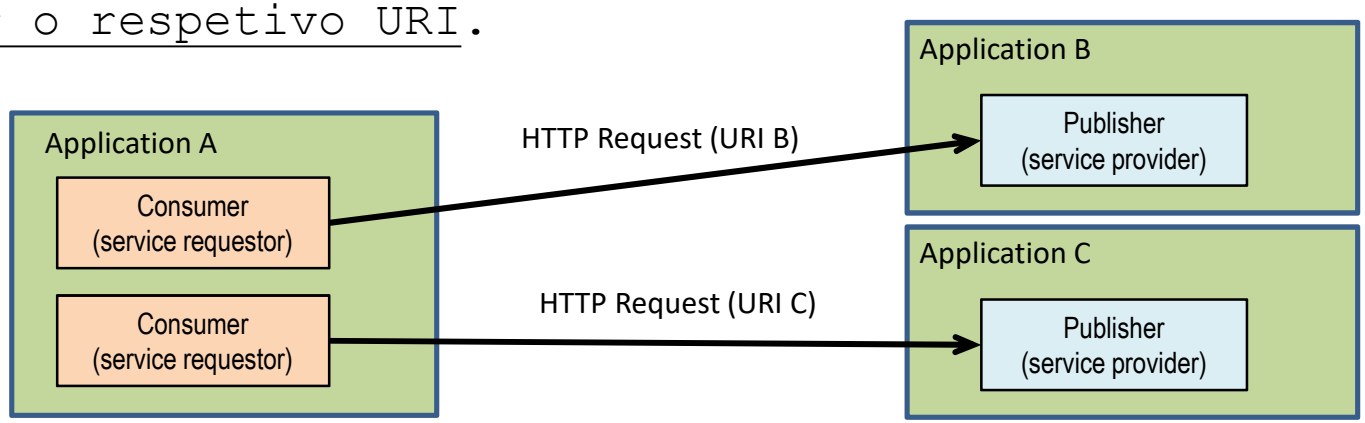
No contexto dos ***Web Services***, um servidor HTTP é um ***service provider*** ou ***publisher***, um cliente HTTP é um ***service requestor*** ou ***consumer***.

Web Services - content

A definição dos conteúdos transferidos através de mensagens HTTP é claramente suportado pelo protocolo através das linhas de cabeçalho *entity* (*Content-type*, *Content-length*, …).

Para transacionar dados internos das aplicações, estes devem ser representados de forma independente, por exemplo, através de XML (*Extensible Mark-up Language*) ou JSON (*JavaScript Object Notation*), correspondendo aos *Content-type* HTTP **application/xml** e **application/json**.

Algo que é imposto pelo HTTP é o modelo de transações cliente-servidor, assim cabe ao ***service requestor*** ou ***consumer*** solicitar o ***service provider*** ou ***publisher***, para isso necessita de conhecer o respetivo URI.



Web Services - content

A definição geral do conceito de *Web Services* é muito abrangente, basicamente engloba qualquer tipo de comunicação entre duas aplicações através do HTTP em que os conteúdos transferidos não se destinam diretamente aos utilizadores, mas sim às próprias aplicações.

Uma definição tão genérica leva necessariamente à possibilidade de serem implementados das mais diversas formas, desde que respeite o protocolo HTTP.

Desde logo, para transferir conteúdos do servidor para o cliente, deve ser usado o método GET, para transferências do cliente para o servidor devem ser usados os métodos POST e PUT, mas o método POST pode ser usado para ao mesmo tempo transferir também conteúdos do servidor para o cliente.

Para conteúdos limitados, o próprio GET também pode ser usado para transferir dados do cliente para o servidor (*query-string*).

Um flexibilidade tão grande acaba por se tornar num problema, e torna-se desejável estabelecer determinados princípios orientadores.

RESTful Web Services

O REST (*REpresentational State Transfer*) é conjunto de princípios orientadores (*design model*) para a implementação de *Web Services*.

Web Services implementados segundo estes princípios designam-se **RESTful**. De certa forma estes princípios correspondem à imposição de restrições (*REST constraints*) relativamente a algo muito flexível que é o conceito de *Web Services*.

Recursos (*resources*) - um recurso representa um conjunto de informação que deva ser referenciada por um nome, pode ser uma entidade, um estado, um evento ou uma funcionalidade.

Cada recurso é acessível através de um URI associado ao publisher (servidor HTTP), as representações desse recurso podem ser transferidas nos dois sentidos entre o *consumer* (cliente) e o *publisher* (servidor).

As várias operações sobre um recurso devem ser o mais limitadas possível, de preferência apenas CRUD (Create, Read, Update, Delete).

RESTful Web Services methods

As várias operações possíveis sobre um recurso devem apenas CRUD (*Create, Read, Update, Delete*). Em larga medida estas operações destinam-se à gestão de coleções de recursos, assim um URI pode identificar uma coleção ou um recurso da coleção. Um recurso (item) de uma coleção é identificado pelo URI:

URI-DA-COLEÇÃO/NOME-DO-RECURSO

Para cada operação CRUD deve ser usado um método HTTP específico, o significado exato depende de o URI corresponder a uma coleção ou um item de uma coleção:

HTTP method	Resources Collection	Resource
GET	Get a collection content listing (URIs list)	Get the resource content (representation)
PUT	Replace the entire collection by a new collection.	Replace the resource content, create if it does not exist.
POST	Create a new resource (item) in the collection, the new resource URI is returned to the requestor.	
DELETE	Delete the entire collection.	Delete the resource from the collection.

RESTful Web Services – URI naming and best practices

Os nomes usados nos URI devem obedecer aos seguintes princípios:

- Nomes singulares para recursos únicos ou itens de uma coleção.
- Nomes plurais para coleções de recursos.
- Verbos para controladores ou funções.
- URI em minúsculas, eventualmente com *camel casing*.
- *Hyphens* em lugar de *underscores*.
- Evitar nomes CRUD (*Create/Read/Update/Delete*) no URI.
- Os componentes do URI devem representar a estrutura hierárquica dos dados (coleção/item).
- Usar componentes do URI para representar valores de variáveis.
- Pode ser adicionado um *query-string* ao URI.

RESTful Web Services – Stateless server

Os servidores (*publishers*) não guardam informação de contexto entre pedidos sucessivos de um mesmo *consumer*, tratam todos os pedidos de forma igual. Se o *consumer* necessitar de manter um contexto pode proceder de uma de duas formas:

- Criar um contexto no servidor, esse contexto será um recurso identificado por um URI que o *consumer* terá de referenciar em pedidos subsequentes.
- Manter localmente um contexto e juntamente com cada pedido enviar ao servidor esse contexto. Cada pedido deve conter todos os dados necessários ao seu processamento.

O método GET deve ser ***safe***, isso significa que não altera o estado do recurso.

Os métodos PUT, GET, e DELETE devem ser ***idempotent***, isso significa que pedidos repetidos com exatamente com os mesmos dados não produzem alterações adicionais ao estado do recurso para além do efeito do primeiro pedido.

O método POST não é ***safe*** nem ***idempotent***.

RESTful Web Services – API

Implementar aplicações distribuídas através de ***RESTful Web Services***, é muito mais simples de forem usadas bibliotecas que implementam as funcionalidades base como por exemplo efetuar pedidos via HTTP (*consumers*), lidar com esses pedidos (*publishers*) e em particular manusear os respetivos conteúdos em XML.

A API JAX-RS permite usar anotações em Java que simplificam de forma significativa a implementação de *RESTful Web Services*. Através das anotações JAX-RS é possível dar a qualquer classe e métodos uma interface de *RESTful Web Service*. Jersey é o nome de código pelo qual é conhecida uma implementação JAX-RX disponibilizada pela Oracle em Java.

O exemplo ao lado, com anotações JAX-RX, define um método que produz um conteúdo de texto quando solicitado através do pedido **GET /hello:**

```
@Path(“hello”)
public class ola {
        @GET
        @Produces("text/plain; charset=UTF-8")
        public String show() {
                return(“Hello world”);
        }
}
```

XML – eXtensible Mark-up Language

O XML foi desenvolvido como forma de representar sob a forma de texto dados tendo em vista o seu armazenamento e também a sua transferência entre aplicações em ambiente de rede (**Content-type: application/xml**).

O formato XML é um compromisso com o objetivo de ser manipulável quer por humanos quer por aplicações.

Tal como o HTML, o XML é estruturado em TAGS definidos entre os símbolos < e >, no entanto enquanto o HTML define um conjunto de TAGS com significados pré estabelecidos, o XML não.

No XML os TAGS são estabelecidos pelas aplicações de acordo com as suas necessidades.

O HTML é usado para especificar a forma de apresentar os dados, o XML apenas se preocupa com os dados propriamente ditos.

XML – regras

Um conteúdo XML pode, opcionalmente, começar por uma linha designada ***XML prolog***:

<?xml version="1.0" encoding="UTF-8"?>

No XML os nomes dos TAGS são ***case sensitive***. O conteúdo pode não ter o *XML prolog*, mas é obrigatório ter um **TAG raiz** que abrange todo o conteúdo.

Em XML todos os TAGS têm de ser fechados, se o TAG não contém dados pode ser ele próprio fechado, nesse caso termina com **/>**, exemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<lista>
  <user />
  <user></user>
</lista>
```

Neste exemplo o TAG raiz é <lista>, os dois TAG <user> estão ambos vazios, embora declarados de forma diferente. Se o conteúdo de um TAG incluir o símbolo **<**, terá de ser representado através de **<**, igualmente o símbolo **&** terá de ser representado através de **&**.

XML – atributos

Os TAGS podem conter atributos, um atributo é um par **nome="valor"** incluído no TAG. Em XML, tal como o TAGS, também os atributos são ***case sensitive***.

Os valores dos atributos devem encontrar-se obrigatoriamente entre aspas ou plicas. O objetivo de um atributo é associar um valor ou propriedade a um TAG, mas o número de atributos em cada TAG deve ser limitado, existindo muitos atributos devem ser convertidos em sub-TAGS.

É boa prática usar um atributo para identificar o elemento e usar sub-TAGS para representar as restantes propriedades do elemento. Exemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user id="200">
    <nome>UtilizadorA</nome>
    <e-mail>userA@domain.com</email>
    <phone>9999909</phone>
    <password>xxxxxx</password>
  </user>
  <user id="210" />
</users>
```

Teste de Web Services RESTful - Postman

O ***postman*** é uma ferramenta fundamental para testar *Web Services* durante o seu desenvolvimento. Usando apenas um browser só é possível realizar testes com pedidos através do método GET.

O ***postman*** permite realizar todos os tipos de pedidos do HTTP e definir conteúdos para os pedidos POST e PUT, permite também definir as linhas de cabeçalho HTTP e muitos outros detalhes do pedido.

Também sob o ponto de vista da análise da resposta obtida, o ***postman*** permite ver detalhes que não estão normalmente acessíveis num browser.

Existem muitas versões gratuitas de aplicações do tipo ***postman***, incluindo versões sob a forma de extensão ou *plugin* para os browsers mais populares.

RESTful Web Services – Project

Utilizando os princípios RESTful, pretende-se o desenvolvimento de várias aplicações em ambiente distribuído:

- USERS - Web Service RESTful de gestão de utilizadores (CRUD) e função de validação de password (autenticação).
- MBOARD - Web Service RESTful de gestão de quadro de mensagens (CRUD).
- Aplicações cliente de consola.

O objetivo é que a aplicação USERS seja suficientemente flexível para poder ser utilizada por outras aplicações de diversos tipos, no caso será usada pela aplicação MBOARD para validar os utilizadores que pretendem colocar mensagens.

RESTful Web Services – Project - USERS

A primeira fase do projeto consiste no desenvolvimento da aplicação USERS (Web Service) e uma aplicação de consola para a correspondente gestão de utilizadores:

- pelo administrador – gestão de todos os utilizadores.

- por utilizadores – gestão do próprio utilizador.

A aplicação USERS usa o número de porto 8085 e disponibiliza o URI /users.

Cada utilizador é identificado por um nome de login único (USER), o URI na forma /users/USER refere-se ao utilizador com nome de login USER. Além do nome de login, cada utilizador deve ter também um nome, uma password e um endereço de e-mail.

Método GET: deve ser suportado, fornecendo de acordo com o URI usado os dados públicos de todos os utilizadores (GET /users) ou de apenas um utilizador (GET /users/USER). Os dados públicos não incluem a password do utilizador.

Método PUT: apenas suportado para o URI /users/USER.

- Permite ao administrador criar um novo utilizador ou atualizar dados de um utilizador já existente, o nome de login não pode ser alterado.

- Permite a um utilizador atualizar os seus dados.

- Permite validar as credenciais de um utilizador

O conteúdo dos pedidos PUT, deve ser em XML e ter a seguinte forma geral:

```
<request>
<auth><login>utilizador</login><pass>password</pass></auth>
(…)
</request>
```

O TAG <auth> serve para autenticar o utilizador que está a realizar o pedido. Se o TAG <login> for omitido assume-se que se trata do utilizador correspondente ao URI (USER).

Se o pedido não tiver mais nenhum conteúdo, serve apenas para validar o acesso e a resposta será vazia.

Método PUT: URI /users/USER.

Se tem um conteúdo além do TAG <auth> esse conteúdo deve ser a descrição de um utilizador através do TAG <user>:

```
<user>
<login>login name</login
<name>Nome do utilizador</name>
<pass>password</pass>
<email>mail address</email>
</user>
```

Um pedido PUT com o TAG <user> destina-se a criar um novo utilizador ou alterar dados de um utilizador existente, os campos do utilizador que forem omitidos não serão alterados, ou não serão definidos no caso de um novo utilizador.

A resposta ao PUT neste caso deve ter a seguinte forma:

A password nunca deve ser incluída nas respostas.

```
<response>
<user>
<login>login name</login
<name>Nome do utilizador</name>
<email>mail address</email>
</user>
</response>
```

Respostas aos pedidos GET

Têm a forma geral:

```
<response>
<user>
<login>login name</login>
<name>Nome do utilizador</name>
<email>mail address</email>
</user>
</response>
```

Neste caso o TAG <user> pode ocorrer várias vezes, caso seja uma resposta a um pedido GET /users. Novamente as passwords nunca devem ser incluídas.

Método DELETE: apenas válido para o URI /users/USER. Tem a forma:

```
<request>
<auth><login>utilizador</login><pass>password</pass></auth>
</request>
```

Apenas o administrador pode eliminar utilizadores.

Códigos de resposta

Devem ser implementados pelo menos os seguintes códigos de resposta (status):

200 Ok	Sucesso na operação (GET/PUT/DELETE)
501 Not Implemented	Método não implementado pela aplicação.
403 Forbidden	Acesso não autorizado ou operação não permitida.
405 Method Not Allowed	O método usado não é aceite para o URI a que foi aplicado.
404 Not Found	URI inexistente.