SISTEMAS DISTRIBUÍDOS

Capítulo 5

Invocação Remota na Web

NOTA PRÉVIA

A apresentação utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2005

ORGANIZAÇÃO DO CAPÍTULO

Web services REST

Web services SOAP

INVOCAÇÃO REMOTA NA WEB

Os Web Services são uma forma de invocação remota orientada para a Web. O uso do termo Web advém de:

- Utilização do protocolo HTTP como suporte base à comunicação;
- Os próprios serviços poderem ser acessíveis ao nível da WWW (páginas e sites Web), podendo operar como componentes de aplicações web.

Em geral, tiram partido da implantação global do protocolo HTTP, da WEB e das suas tecnologias.

Tipos de Web Services

WebServices REST

A aplicação é estruturada em coleções de recursos, acedidos por HTTP através de um URL, onde a semântica das operações está mapeada para as várias operações HTTP, tais como GET, POST, DELETE, etc.

WebServices SOAP

Oferece uma forma de invocação remota cujo foco está na interoperabilidade. Por via da standardização dos seus componentes consegue ser independente da linguagem e da plataforma de sistema.

ORGANIZAÇÃO DO CAPÍTULO

Web services REST

Web services SOAP

REST: REPRESENTATIONAL STATE TRANSFER

Aproximação: uma aplicação é modelada como uma coleção de recursos

- Um recuso é identificado por um URI/URL
- Um URL devolve um documento com a representação do recurso
- Podem-se fazer referências a outros recursos usando *ligações (links)*

Estilo arquitetural, não um sistema de desenvolvimento

Aproximação proposta por Roy Fielding na sua tese de doutroamento

- Não como uma alternativa aos web services SOAP, mas como uma forma de aceder a informação

ROY FIELDING

UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

- Author of HTTP specification
- Co-founder of Apache HTTP server

REST: PRINCÍPIOS DE DESENHO

Protocolo cliente/servidor stateless: cada pedido contém toda a informação necessária para ser processado – objetivo: tornar o sistema simples.

Recursos – no sistema tudo são recursos, identificados por um URI/URL.

- Recursos tipicamente armazenados num formato estruturado que suporta hiper-ligações (e.g. JSON, XML)

Interface uniforme: todos os recursos são acedidos por um conjunto de operações bem-definidas. Em HTTP: POST, GET, PUT, DELETE (equiv. criar, ler, actualizar, remover).

Caching: para melhorar desempenho, respostas podem ser etiquetadas como permitindo ou não *caching*, via o cabeçalho HTTP addequado.

REST: CONVENÇÕES E BOAS PRÁTICAS

As operações HTTP sobre um recurso, com um dado URL/URI têm a semântica pré-definida:

- **GET** lê o recurso, reportando `404 NOT FOUND` se o recurso não existir;
- **POST** cria um novo recurso, reportando `409 CONFLICT`, se o recurso já exisitir;
- **PUT** atualiza um recurso, reportando `404 NOT FOUND` se o recurso não existir;
- **DELETE** apaga o recurso, reportando `404 NOT FOUND` se o recurso não existir.

REST: CONVENÇÕES E BOAS PRÁTICAS (2)

Para atualizar parcialmente um recurso pode-se usar a operação:

- **PATCH** atualiza parcialmente um recurso, reportando `404 NOT FOUND` se o recurso não existir.

Os dados envolvidos nas operações são bem-tipados, segundo um ***schema*** JSON ou XML .

EXEMPLO

Considere-se um serviço que mantém informação sobre servidores, disponibilizando as seguintes operações:

- Adicionar um servidor
- Remover um servidor
- Modificar um servidor
- Obter informação sobre um servidor, dado o seu id
- Pesquisar informação dum servidor

EXEMPLO: ADICIONAR SERVIDOR

- **URL**: http://myserver.com/server/35345645
- **Método**: POST
- **Body**:

```
{ id: "35345645",
name : "server 1",
props : ["a", "b", "c"]
}
```

Alternativa 1: usar o URL http://myserver.com/server/ para a criação, sendo o id passado no objeto.

EXEMPLO: ADICIONAR SERVIDOR

- **URL**: http://myserver.com/server/35345645
- **Método**: POST
- **Body**:

```
{ id: "35345645",
name : "server 1",
props : ["a", "b", "c"]
}
```

Alternativa 2: usar o URL http://myserver.com/server/, sendo o id criado no serviço e devolvido como resultado do método.

EXEMPLO: MODIFICAR SERVIDOR

- **URL**: http://myserver.com/server/35345645
- **Método**: PUT
- **Body**:

```
{ id: "35345645",
name : "server 1",
props : ["a", "b", "Z"]
}
```

A atualização tipicamente fornece uma cópia integral do recurso.

EXEMPLO: REMOVER SERVIDOR

- **URL**: http://myserver.com/server/35345645
- **Método**: DELETE
- **Body**: <empty>

EXEMPLO: OBTER INFORMAÇÃO DE SERVIDOR

- **URL**: http://myserver.com/server/35345645
- **Método**: GET
- **Body**:

- **Reply**:

```
{ id: "35345645",
name : "server 1",
props : ["a", "b", "Z"]
}
```

EXEMPLO: LISTAR SERVIDORES

- **URL**: http://myserver.com/server
- **Método**: GET
- **Body**: <empty>

- **Reply**:

```
[{ id: "35345645",
name : "server 1",
props : ["a", "b", "Z"]
}, …
]
```

EXEMPLO: PESQUISAR SERVIDORES

- **URL**: http://myserver.com/server?query=a
- **Método**: GET
- **Body**: <empty>

Pode-se usar um query parameter para filtrar os resultados que se pretendem obter

- **Reply**:

```
[{ id: "35345645",
name : "server 1",
props : ["a", "b", "Z"]
}, …
]
```

CODIFICAÇÃO DOS DADOS: XML VS. JSON

A informação transmitida nos pedidos e nas respostas é codificada tipicamente em JSON ou XML.

JSON é muito popular devido à facilidade de processamento em JavaScript e, por conseguinte, em páginas Web e aplicações móveis.

Dados binários são transferidos como HTTP octet-stream.

CODIFICAÇÃO DOS DADOS: XML VS. JSON

```
<Person firstName='John'
lastName='Smith' age='25'>
  <Address streetAddress='21 2nd
Street' city='New York' state='NY'
postalCode='10021' />
  <PhoneNumbers home='212 555-
1234' fax='646 555-4567' />
</Person>
```

(Example from wikipedia)

```
{  "Person”:  {
     "firstName": "John",
     "lastName": "Smith",
     "age": 25,
     "Address”:  {
       "streetAddress":"21 2nd Street",
       "city":"New York",
       "state":"NY",
       "postalCode":"10021"
     },
     "PhoneNumbers”: {
       "home":"212 555-1234",
       "fax":"646 555-4567"
     }
   }
}
```

REST: SUMÁRIO

Baseado em protocolos standard

- Comunicação: HTTP / HTTPS
- Identifcação de recursos: URL/URI
- Representação dos recursos: JSON, XML

O foco na simplicidade e no desempenho tornou o modelo REST muito popular.

Usado nas APIs públicas de serviços populares como: Twitter, Imgur, Facebook, Dropbox, etc.

- Alguns destes serviços forneceram anteriormente APIs em Web Services SOAP, que depois foram descontinuadas.

REST: Teoria vs. Prática

Há muitos exemplos de serviços disponibilizados num modelo REST que nem sempre aderem completamente aos princípios REST.

Dropbox (apagar um ficheiro)

Operação: POST
URL: https://api.dropboxapi.com/2/files/delete_v2
Body: { "path": "/Homework/math/Prime_Numbers.txt" }

Na API da Dropbox, um ficheiro para efeitos da sua remoção não é um recurso a apagar com HTTP DELETE.

REST: SUPORTE JAVA

Definido em JAX-RS (JSR 311)

Suporte linguístico baseado na utilização de anotações

- Permite definir que um dado URL leva à execução dum dado método
- Permite definir o modo de codificação da resposta
  - JSON – mecanismo leve de codificação de tipos (RFC 4627)
  - XML – usando mecanismo standard de codificação de cobjectos java em XML fornecido pelo JAXB

JAX-RS: SERVIDOR

Do lado do servidor, o suporte Java para REST (JAX-RS / JSR 311) permite ao programador concentrar-se na lógica das operações e evitar muitos aspectos de baixo nivel, ao nível do protocolo HTTP, conversão de dados, etc.

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

JAX-RS: SERVIDOR (EXEMPLO)

@Singleton permite indicar que se quer apenas uma instância do recurso.

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

JAX-RS: SERVIDOR (EXEMPLO)

@Path permite definir a path, que será concatenada ao URL base do servidor

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PA

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

@POST / @GET / @PUT / @DELETE
Operação HTTP que leva à execução do método

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final Strin

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

@Consumes usado para especificar o formato dos dados enviado no corpo do pedido (para o parâmetro server)

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

@Producesusado para especificar o formato dos dados retornado no corpo da resposta

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLI
    public Response addServer

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

@Path permite definir a path a concatenar à define à path definida no recurso. {id} permite aceitar qualquer valor e atribui-lo a um parâmetro usando o @PathParam("id")

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) {   ......   }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathPa

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

@QueryParam permite obter um parâmetro opcional passado como um query parameter no URL

JAX-RS: SERVIDOR (EXEMPLO)

```
    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public void updateServer(@PathParam("id") String id,
                                    Server srv) {   ......   }

    @DELETE
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server delServer(@PathParam("id") String id) {...}
}
```

JAX-RS: SERVIDOR (EXEMPLO)

Servidor por pedido

```
URI baseUri =
UriBuilder.fromUri("http://0.0.0.0/").port(8080).build();
ResourceConfig config = new ResourceConfig();
config.register(ServerResource.class);
JdkHttpServerFactory.createHttpServer(baseUri, config);
```

Servidor único

```
URI baseUri =
UriBuilder.fromUri("http://0.0.0.0/").port(8080).build();
ResourceConfig config = new ResourceConfig();
config.register(new ServerResource());
JdkHttpServerFactory.createHttpServer(baseUri, config);
```

JAX-RS: SERVIDOR : OBSERVAÇÕES

O programador define o mapeamento entre as operações REST sobre os recursos e os métodos Java correspondentes; especifica a codificação dos dados a utilizar JSON ou XML, mas não precisa de fornecer código para codificar/descodificar os dados.

A capacidade de introspeção da linguagem Java permite gerar o resto do código do servidor automaticamente. O programador não se preocupa com sockets, canais de entrada ou saída; HTTP 1.0/1.1 ou 2.0, etc.

JAX-RS: CLIENTE

O suporte linguistico é muito limitado. A invocação é concisa, mas não se assemelha a uma invocação de um procedimento local.

O programador tem que construir um pedido explicitamente, em vez de invocar uma única função com parâmetros e um resultado.

NOTA: Existem várias bibliotecas que automatizam o processo de criar clientes para servidores REST.

JAX-RS: CLIENTE (EXEMPLO)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
WebTarget target = client.target( serverUrl)
      .path( RestMediaResources.PATH );
Response r = target.request()
      .accept(MediaType.APPLICATION_JSON)
      .post(Entity.entity(srv,
            MediaType.APPLICATION_JSON));
if( r.getStatus() == Status.OK.getStatusCode()
            && r.hasEntity() )
      System.out.println("Response: "
            +r.readEntity(String.class ) );
else
      System.out.println("Status: " + r.getStatus() );
```

JAX-RS: CLIENTE (EXEMPLO)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
WebTarget target = client.target( serverUrl)
      .path( RestMediaResources.PATH );
Response r = target.request()
      .accept(MediaType.APPLICATION_JSON)
      .post(Entity.entity(srv,
            MediaType.APPLICATION_JSON));
if( r.getStatus() == Status.OK.getStatusCode()
            && r.hasEntity() )
```

e

> "***Clients are heavy-weight objects*** *that manage the* ***client-side communication*** *infrastructure. Initialization as well as disposal of a Client instance may be a rather expensive operation. It is therefore* ***advised to construct only a small number of Client instances*** *in the application."*
>
> Por exemplo, no trabalho não devem estar a criar clientes para o serviço Users de cada vez que precisam de verificar uma password… devem reusar o cliente existente.

ORGANIZAÇÃO DO CAPÍTULO

Web services REST

Web services SOAP

WEB SERVICES SOAP: MOTIVAÇÃO

A Web service is a software system designed to support ***interoperable machine-to-machine interaction*** *over a network.*

It has an ***interface described in a machine-processable format*** *(specifically WSDL).*

Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically ***conveyed using HTTP*** *with an* ***XML serialization*** *in conjunction with other Web-related standards*

WEB SERVICES SOAP: CARACTERÍSTICAS PRINCIPAIS...

Modelo para acesso a servidores: **invocação remota**

Desenhado para garantir **inter-operabilidade**

Protocolo: **HTTP** e **HTTPS** (eventualmente SMTP)

Referências: URL/URI

Representação dos dados: **XML**

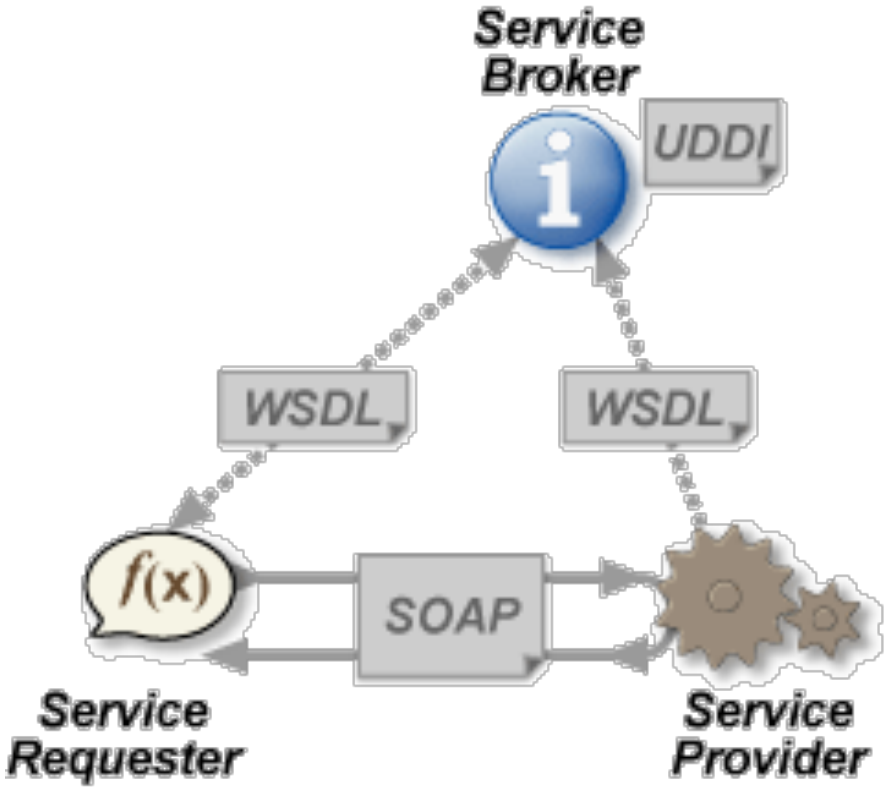
COMPONENTES BÁSICOS

SOAP: protocolo de invocação remota

- Oneway
- Pedido-resposta
- Notificação
- Notificação-resposta

WSDL: linguagem de especificação de serviços

UDDI: serviço de registo



SOAP (Simple Object Access Protocol)

Protocolo de comunicação visando a troca de informação estruturada na implementação de WebServices.

Estabelece as regras e as mensagens trocadas, usando o formato XML.

Procura ser neutro e independente da plataforma de sistema para garantir a inter-operabilidade.

Implementado sobre protocolos applicacionais tais como HTTP ou SMTP (Email).

WSDL (Web Services Description Language)

Usado para definir um documento XML que descreve a interface de um Web Service, a vários níveis:

- Define a interface do serviço, indicando quais as operações disponíveis;
- Define as mensagens trocadas na interacção (e.g. na invocação duma operação, quais as mensagens trocadas);
- Permite definir a forma de representação dos dados;
- Descreve como e onde aceder ao serviço
  - Inclui o URL de uma instância do serviço.

WSDL (Web Services Description Language)

Especificação WSDL bastante verbosa – normalmente criada a partir de interface ou código do servidor

- Em Java e .NET existem ferramentas para criar especificação a partir de interfaces Java
- Sistemas de desenvolvimento possuem *wizards* que simplificam tarefa

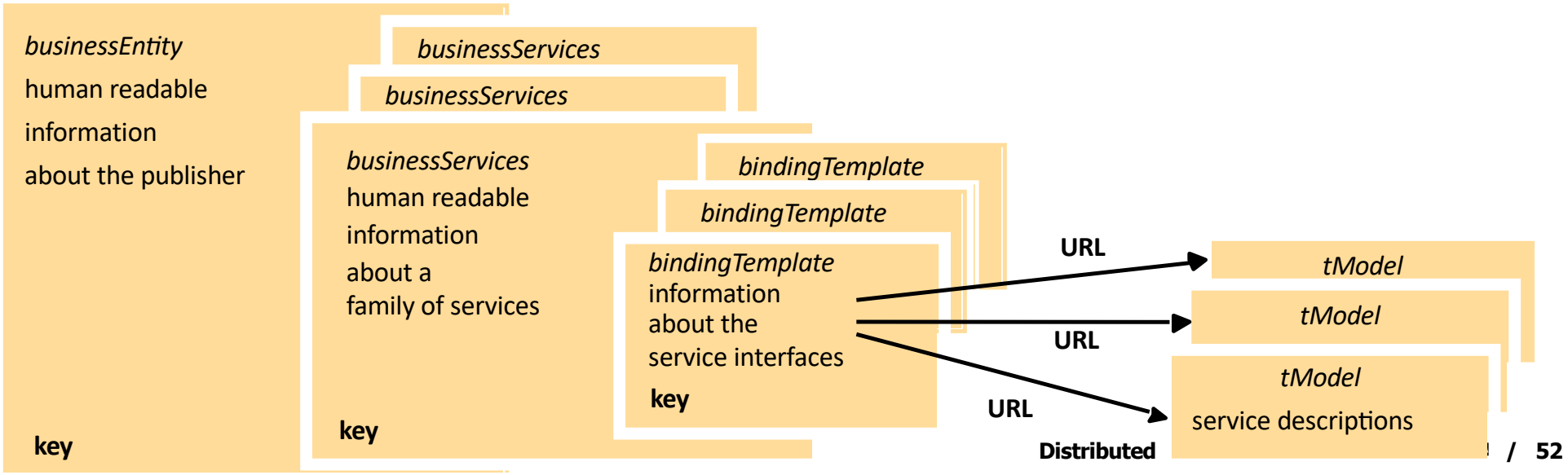
O documento WSDL pode tipicamente ser obtido diretamente a partir de uma instância do serviço em execução.

http://mywebservice.org/my-web-service?wsdl

UDDI (Universal Description, Discovery and Integration)

Diretório de WebServices com a possibilidade de pesquisa com base em atributos.

- Protocolo de ligação ou binding entre o cliente e o servidor
- Os fornecedores dos serviços publicam a respectiva interface
- O protocolo de inspeção permite verificar se uma dado serviço existe baseado na sua identificação
- O UDDI permite encontrar o serviço baseado na sua definição – capability lookup



WebServices: Clientes

O coódigo de ligação ao web service do cliente é tipicamente gerado de forma automática.

Com base neste código, o cliente invoca uma função/método, passando os parâmetros e recebendo eventualmente o resultado do retorno.

A geração deste código tem como base o documento WSDL associado ao web service a invocar.

A geração pode ser estática, em tempo de compilação; Dinâmica, sendo o código gerado em tempo de execução.

Tal é possível porque o WSDL está pensado para ser processado por máquinas e é exaustivo na descrição do serviço.

WEB SERVICES: EXTENSÕES WS-*

Para além do mecanismo de invocação remota de base, existem diversas extensões que oferecem *features* e semânticas mais sofisticadas:

WS-Reliability: especificação que permite introduzir fiabilidade nas invocações remotas

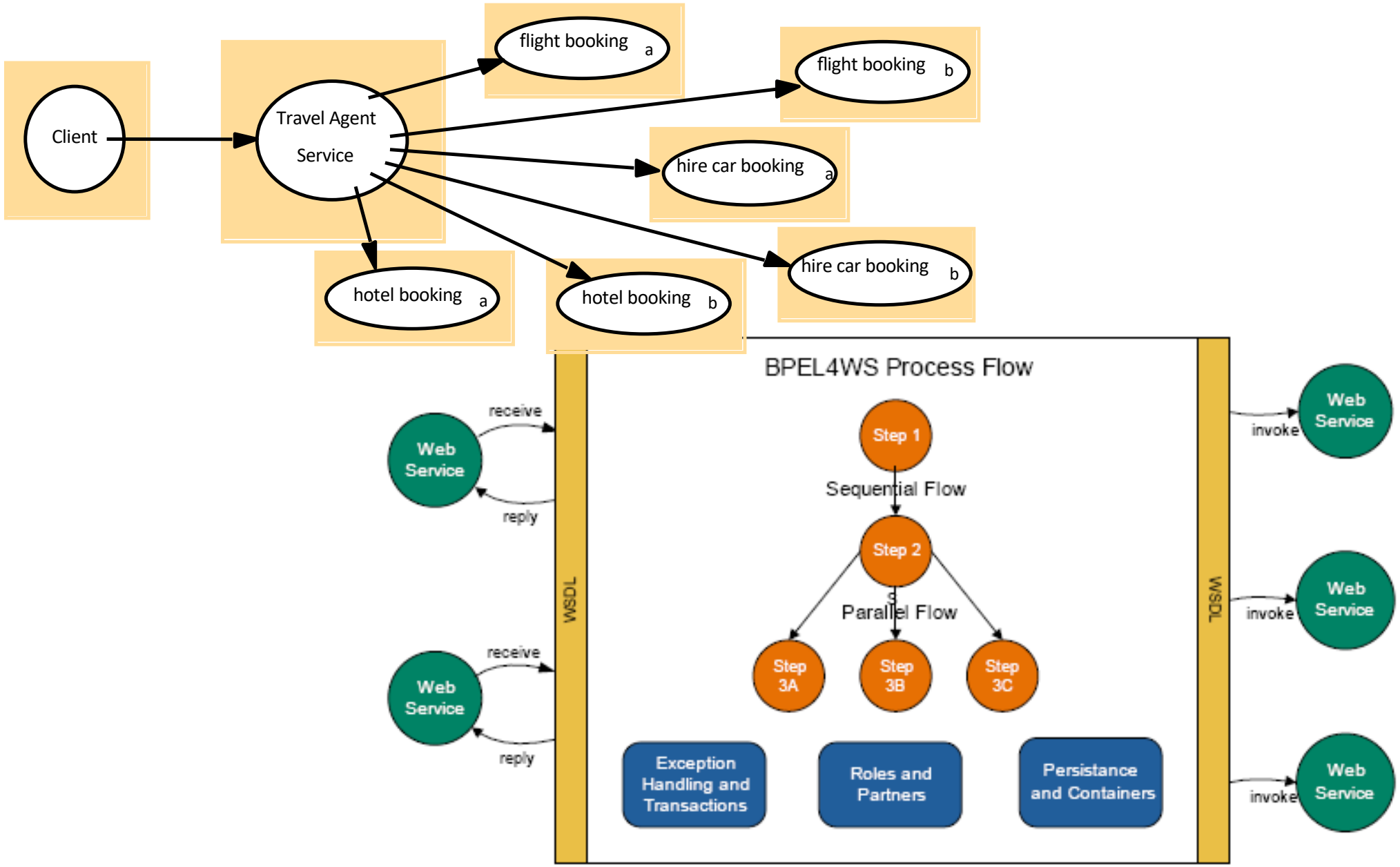
WS-Security: define como efectuar invocações seguras

WS-Coordination: fornece enquadramento para coordenar as ações de aplicações distribuídas (coreografia de web services)

WS-AtomicTransaction: forncece coordenação transaccional (distribuída) entre múltiplos web services

...

WS-COORDINATION



WEB SERVICES: RESUMO

Standard na indústria porque apresenta uma solução para integrar diferentes aplicações

Permite:

- Utilização de standards
  - HTTP, XML
- Reutilização de serviços (arquiteturas orientadas para os serviços)
  - WS-Coordination
- Modificação parcial/evolução independente dos serviços
  - WSDL

WEB SERVICES: PROBLEMAS

Desempenho:

- Complexidade do XML
- Difícil fazer *caching*: noção de operação + estado não permite explorar o *caching* da infraestrutura HTTP

Complexidade

- A normalização gerou especificações complexas que tornam necessário a utilização de ferramentas de suporte.

WEB SERVICES: SUPORTE EM JAVA

Os Web Services são suportados através da extensão Java API for XML Web Services (JAX-WS). Fez parte da distribuição standard até à versão 8 (1.8) da linguagem.

A partir da versão Java 9, passou a ser uma dependência externa.

JAX-WS: SERVIDOR (EXEMPLO)

```
@WebService(serviceName=SoapServers.NAME,
targetNamespace=SoapServers.NAMESPACE,
endpointInterface=SoapServers.INTERFACE)
public class SoapServersWebService implements SoapServer
{
    static final String NAME = "servers";
    static final String NAMESPACE = "http://sd2019";
    static final String INTERFACE =
"server.soap.SoapServers";

    @WebFault
    public class ServersException extends Exception
    {... }
```

JAX-WS: SERVIDOR (EXEMPLO)

@WebService permite indicar que se trata dum servidor de Web Services.

```
@WebService(serviceName=SoapServers.NAME,
targetNamespace=SoapServers.NAMESPACE,
endpointInterface=SoapServers.INTERFACE)
public class SoapServersWebService implements SoapServer
{
    static final String NAME = "servers";
    static final String NAMESPACE = "http://sd2019";
    static final String INTERFACE =
"server.soap.SoapServers";

    @WebFault
    public class ServersException extends Exception
    {... }
```

JAX-WS: SERVIDOR (EXEMPLO)

```
@WebService(serviceName=SoapServers.NAME,
targetNamespace=SoapServers.NAMESPACE,
endpointInterface=SoapServers.INTERFACE)
public class SoapServersWebService implements SoapServer
{
    static final String NAME = "servers";
    static final String NAMESPACE = "http://sd2019";
    static final String INTERFACE =
"server.soap.SoapServers";

    @WebFault
    public class ServersException extends Exception
    {... }
```

@WebFault permite definir uma exceção a ser lançada pelos métodos do servidor

JAX-WS: SERVIDOR (EXEMPLO)

```
    @WebMethod
    void createServer( Server srv) throws ServersException
    {... }

    @WebMethod
    Server getServer (String id) throws ServersException
    {...}
    ...

    public static void main(String[] args) {
        ...
        Endpoint.publish(serverURI,
           new SoapServersWebService());
        ...
    }
}
```

JAX-WS: SERVIDOR (EXEMPLO)

@WebMethod especifica que é um método do servidor de Web Services

```
    @WebMethod
    void createServer( Server srv) throws ServersException
    {... }

    @WebMethod
    Server getServer (String id) throws ServersException
    {...}
    ...

    public static void main(String[] args) {
        ...
        Endpoint.publish(serverURI,
           new SoapServersWebService());
        ...
    }
}
```

JAX-WS: SERVIDOR (EXEMPLO)

```
    @WebMethod
    void createServer( Server srv) throws ServersException
    {... }

    @WebMethod
    Server getServer (String id) throws ServersException
    {...}
    ...

    public static void main(
        ...
        Endpoint.publish(serverURI,
           new SoapServersWebService());
        ...
    }
}
```

Para publicar um servidor de Web Services usando o suporte nativo do JAX-WS.

JAX-WS: CLIENTE (EXEMPLO)

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,
        SoapServersWebService.NAME );

Service service = Service.create(
           "http://<ip>:<port>/path?wsdl", QNAME);
SoapServers servers = service.getPort(
           servers.soap.SoapServers.class );

Server srv = ...

servers.createServer( srv );

srv = servers.getServer( "someid");
```

JAX-WS: CLIENTE (EXEMPLO)

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,
        SoapServersWebService.NAME );

Service service = Service.c
          "http://<ip>:<pc
SoapServers servers = service.getPort(
          servers.soap.SoapServers.class );

Server srv = ...

servers.createServer( srv );

srv = servers.getServer( "someid");
```

Para criar um cliente de Web Services usando o suporte nativo do JAX-WS.

JAX-WS: CLIENTE (EXEMPLO)

Obtém uma referência para o servidor por configuração direta.

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,
        SoapServersWebService.NAME );

Service service = Service.create(
           "http://<ip>:<port>/path?wsdl", QNAME);
SoapServers servers = service.getPort(
           servers.soap.SoapServers.class );
```

```
Server srv = ...

servers.createServer( srv );

srv = servers.getServer( "someid");
```

JAX-WS: CLIENTE (EXEMPLO)

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,
        SoapServersWebService.NAME );

Service service = Service.create(
            "http://<ip>:<port>/path?wsdl", QNAME);
SoapServers servers = service.getPort(
            servers.soap.SoapServers.class );

Server srv = ...

servers.createServer( srv );

srv = servers.getServer( "someid");
```

Com a referência para o servidor, invocar um método remoto é idêntico a invocar um método local.

WEBSERVICES: SOAP VS REST

SOAP:

getServer(id)
addServer(id, srv)
removeServer(id)
updateServer (id, srv)
listServers()
findServer(query)

REST:

http://myserver.com/server/{id}
(GET, POST, DELETE, PUT)

http://myserver.com/server
http://myserver.com/server?query=a+b

REST vs. RPCs/Web Services

Nos sistemas de RPCs/Web Services a ênfase é nas operações que podem ser invocadas.

Nos sistemas REST, a ênfase é nos recusos, na sua representação e em como estes são afectados por um conjunto de métodos standard.

WEBSERVICES: SOAP VS REST

REST:

- Mais simples
- Mais eficiente
- Complicado implementar serviços complexos (numa abordagem purista)
- Uso generalizado para disponibilização de serviços na Internet

Web services

- Mais complexo
- Grande suporte da indústria para serviços "internos"

E.g.:

- http://www.oreillynet.com/pub/wlg/3005

PARA SABER MAIS

G. Coulouris, J. Dollimore and T. Kindberg, Gordon Blair, Distributed Systems - Concepts and Design, Addison-Wesley, 5th Edition

- Web services – capítulo 9.1-9.4

REST: http://en.wikipedia.org/wiki/Representational_State_Transfer