Esercitazione di Sistemi Distribuiti e Pervasivi

Server REST

Gabriele Civitarese
gabriele.civitarese@unimi.it

EveryWare Lab
Università degli Studi di Milano
-
Docente: Claudio Bettini

Slide realizzate a partire da versioni precedenti di Letizia Bertolaja, Sergio Mascetti, Dario Freni e Claudio Bettini

Copyright

Alcune slide di questo corso sono in parte personalizzate da quelle fornite dall'autore del testo di riferimento e distribuite dalla casa editrice (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). Tutte le altre slide sono invece di proprietá del docente. Tutte le slide sono soggette a diritto d'autore e quindi non possono essere ri-distribuite senza consenso. Lo stesso vale per eventuali registrazioni o videoregistrazioni delle lezioni.

Some slides for this course are partly adapted from the ones distributed by the publisher of the reference book for this course (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). All the other slides are from the teacher of this course. All the material is subject to copyright and cannot be redistributed without consent of the copyright holder. The same holds for audio and video-recordings of the classes of this course.

REST (Representational State Transfer)

- Comunicazione machine-to-machine via HTTP
- È uno stile architetturale basato sull'utilizzo di quattro metodi HTTP: Get, Post, Put, Delete
- Introdotto nel 2000 e adottato per esempio per la realizzazione di alcuni servizi di: Yahoo, Google, Facebook, Twitter, ecc...
- Vantaggio principale rispetto a Web service SOAP e Web service WSDL-based: semplicità.

4 principi di design

1. Utilizzo esplicito di metodi HTTP
2. Essere "stateless"
3. Ogni risorsa è identificata da una URI (uniform resource identifier) e la struttura di queste risorse è analoga a quella delle directory.
4. Utilizzo di XML e/o JSON.

Utilizzo metodi espliciti di HTTP: l'errore

Una pratica comune non aderente a REST

Nella definizione di operazioni sul server:

- Uso sempre GET
- Definisco con un parametro quale operazione voglio compiere

Esempio:

```
http://myservice.it?action=adduser&name=gianmario
```

Problemi:

- Semantica: in HTTP il metodo GET è definito per ottenere informazioni, non per inserirne.
- Modifica non intenzionale dei dati del server, per esempio da parte di un crawler.

Soluzione: utilizzo esplicito di metodi HTTP

L'aderenza a REST richiede l'utlizzo esplicito di metodi HTTP

- Corrispondenza uno ad uno tra operazioni CRUD (create, read, update, and delete) e metodi HTTP

CRUD $\leftrightarrow$ HTTP:

- Create $\leftrightarrow$ POST
- Read $\leftrightarrow$ GET
- Update $\leftrightarrow$ PUT
- Delete $\leftrightarrow$ DELETE

Problema dei server stateful

Esempio server stateful

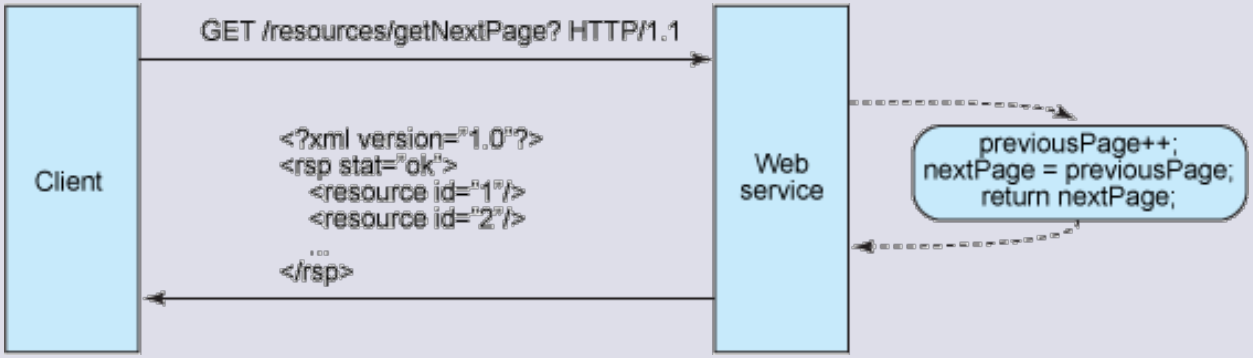

Immagine tratta da: RESTful Web services: The basics

Problemi:

- Il server che risponde alle richieste del client deve essere sempre lo stesso perchè deve memorizzare lo stato del client
- Questo pone limiti alla scalabilità, load-balancing e failover del sistema.

Soluzione: approccio stateless

Principi dei server Stateless

- Ogni richiesta del client è completa e indipendente
  - L'header e il body HTTP contengono tutti i parametri necessari al server per compiere l'operazione richiesta.
- Il server non deve tenere uno stato del client.

Esempio

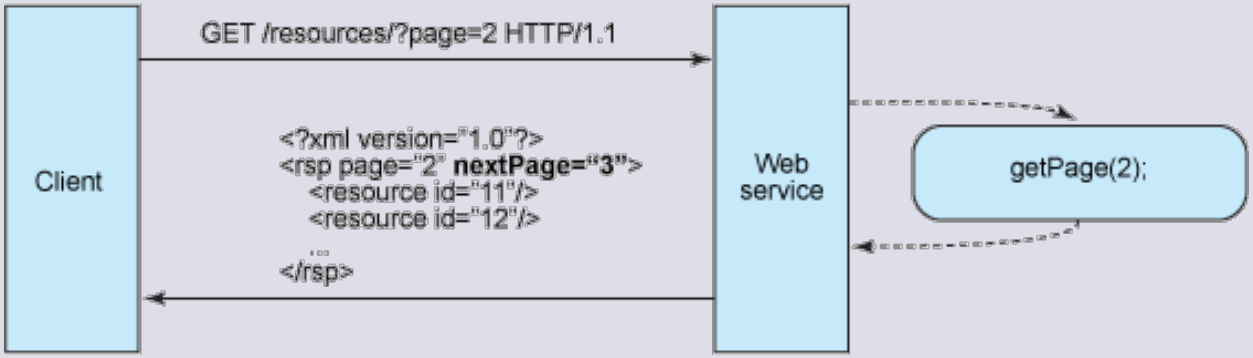

Immagine tratta da: RESTful Web services: The basics

Le risorse nei server REST

Le risorse sono l'elemento centrale dei server REST

- Le operazioni CRUD si riferiscono alle risorse
- In ogni operazione il client trasmette o riceve una rappresentazione di una risorsa

L'importanza dell'organizzazione

- La semplicità di interazione con un server REST dipende fortemente da come sono organizzate le risorse
- È opportuno seguire alcune convenzioni di rappresentazione che aiutano a:
  - Definire il client che interagisce con il server REST
  - Limitare gli errori.

Convenzioni nell'organizzazione delle risorse

- Struttura ad albero "predictable". Esempio:

  `http://www.repubblica.it/tecnologia/2013/05/01/foto/festa_dei_lavoratori_l_omaggio_di_google-57826702/1/`

- Evitare di riportare la tecnologia server-side (es: .php) così si può cambiare
- Tenere tutto lower-case
- Sostituire spazi con "-" o "_"

Quali tecnologie useremo?

- Jersey, come libreria per definire quali metodi chiamare in base a:
  - Verbo HTTP
  - URI
- JAXB,per fare il marshalling e l'unmarshalling automatico in JSON o XML

Jersey

- Liberia Open Source per lo sviluppo di Web Service REST
- Sfrutta API JAX-RS
- Si integra con diverse servlet (Tomcat, GlassFish, . . .)
- Permette di mappare le richieste HTTP su metodi Java

Hello World

```
@Path("/hello")
public class Hello {
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        public String sayPlainTextHello(){
          return "Hello, world";
    }
}
```

Caratteristiche

- Le annotazioni vengono usate per definire l'interazione tra Jersey e il nostro codice.
- Una o più classi gestiscono le **azioni** (es: GET)
- Ciascuna classe gestisce tutte le azioni per uno specifico livello dell'albero delle URI
- Le risorse vere e proprie sono oggetti in memoria centrale (in opportune strutture dati) o dati su DB

Composizione delle URI

- Ogni URI è composta da due parti:
  - Un prefisso definito a livello di configurazione (e.g. Tomcat)
    - Ad esempio: `http://localhost:8080/Rubrica/rest`
  - Un suffisso che cambia in funzione delle risorse alle quali vogliamo applicare le azioni
    - Ad esempio: `/utente/Giandavide`

JAXB

- Ulteriore specifica per il marshalling e l'unmarshalling di oggetti Java (esclusivamente JavaBean) in XML/JSON
- Integrata in Jersey
- Uso di annotazioni per:
  - Definire la radice XML (@XMLRootElement)
  - Specificare i tipi delle variabili (se diversi da quelli usati in Java)
  - I nomi da assegnare agli elementi (se diversi da quelli specificati nel codice)

JAXB (2)

Esempio

```
@XmlRootElement
public class Word {
  private String name, definition;
  public Word() { }

  public String getName() {return name;}
  public void setName(String name) {this.name = name;}
  public String getDefinition() {return definition;}
  public void setDefinition(String definition) {this.definition = definition;}
}
```

Per un tutorial veloce...

http://www.vogella.com/tutorials/JAXB/article.html

Verbi HTTP

- Per abbinare l'esecuzione di un medoto ad un verbo HTTP è necessario usare delle annotazioni:
  - @GET
  - @POST
  - @PUT
  - @DELETE

Esempio

```
@GET
public String sayHello() {
   return "Hello, world";
}
```

Annotazione @Path

- Definibile per una classe e/o per i metodi
- Può contenere valori sia variabili che costanti
- Le parti variabili sono dette *path templates*

Esempio

```
@Path("/dictionary") //path costante
public class DictonaryResource {

@GET
@Path("{word}") //template
@Produces({"application/xml", "application/json"})
```

Altre annotazioni

@PathParam

Permette di associare i template a variabili.

```
...
@GET
@Path("{word}") //template
@Produces({"application/xml", "application/json"})
public Response getWord(@PathParam("word") String name){
...
```

Altre annotazioni (2)

@Produces

Permette di specificare il formato ritornato da un metodo (header Accept in HTTP)

```
...
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello() {
    return "Hello, world";
}
@GET
@Produces(MediaType.TEXT_XML)
public String sayXMLHello() {
   return "<?xml version=\"1.0\"?>" + "<hello> Hello, World! </Hello>";
}
@GET
@Produces(MediaType.TEXT_HTML)
                ...
```

Altre annotazioni (3)

@Consumes

Specifica in quale formato sono attesi i dati dal client

```
...
@POST
@Path("/user")
@Consumes({"application/xml", "application/json"})
public Response put(User userElement) {
    users.addUser(u);
    return Response.ok().build();
    }
...
```

Response

La classe Response viene utilizzata per generare la risposta HTTP usando codici standard (200 per OK, 404 not found, . . .)

```
//risposta 200
return Response.ok().build();
//risposta 200 con annesso oggetto serializzato (via JAXB, in base a Produces)
return Response.ok(oggetto).build();
//risposta 404
return Response.status(Response.Status.NOT_FOUND).build();
```

Esercizio

Dizionario REST

- Realizzare un servizio REST che permetta di gestire un dizionario di parole. Dovrà essere possibile:
  - Inserire una parola e la sua definizione
  - Modificare la definizione di una parola
  - Data una parola, visualizzare la sua definizione
  - Cancellare una parola
- Gestire i casi di errore con opportune risposte HTTP (parola gi inserita, parola non presente, . . .)
- Fate attenzione ai problemi di sincronizzazione!
- Un modo per testare al volo un server REST è l'utilizzo di appositi strumenti tipo *Advanced REST Client* (plugin per Chrome)

Riferimenti

- RESTful Web services: The basics
- REST- Fielding dissertation, Chap. 5