Java RESTful

Metodi Avanzati di Programmazione
(Corso B)

Dipartimento di Informatica
Università degli Studi di Bari Aldo Moro

docente: dott. Pierpaolo Basile

Repository Github

- il codice di queste slide si trova nel package `di.uniba.map.b.lab.rest`

RESTful

- **Representational State Transfer (REST)** è un tipo di architettura software per i sistemi distribuiti
- REST è basato sulla trasmissione di dati tramite protocollo HTTP senza ulteriori livelli
- I sistemi REST non prevedono il concetto di sessione (*stateless*)
- L'architettura REST si basa su HTTP e prevede una struttura delle URI ben definita che identifica univocamente una risorsa o un insieme di risorse
- Utilizza i verbi HTTP
  - GET: recupero di informazioni
  - POST, PUT, PATCH, DELETE: per la modifica
  - OPTIONS: per altri scopi

Principi

- Lo stato dell'applicazione e le funzionalità sono divisi in **risorse** web
- Ogni risorsa è unica e indirizzabile usando la sintassi delle **URI**
- Viene definita un'**interfaccia uniforme** per la condivisione delle risorse che permette il trasferimento di stato tra client e risorse, attraverso:
  - un insieme vincolato di operazioni ben definite
  - un insieme vincolato di contenuti
  - un protocollo che è
    - client-server
    - privo di stato (stateless)
    - memorizzabile in cache
    - a livelli

Caratteristiche

Client–server: I ruoli del client e del server sono ben separati. Per esempio la gestione dei dati è delegata al server, la loro visualizzazione/manipolazione al client. I server non si devono fare carico dell'interfaccia grafica o dello stato dell'utente, in questo modo l'hardware può essere più semplice e maggiormente scalabile. Server e client possono essere sostituiti e sviluppati indipendentemente fintanto che l'interfaccia non viene modificata.

Stateless: La comunicazione client–server è vincolata in modo che nessun contesto client venga memorizzato sul server tra le richieste. Ciascuna richiesta dai vari client contiene tutte le informazioni necessarie per richiedere il servizio e lo stato della sessione è contenuto nel client.

Cacheable: I client possono mettere in cache le risposte. Queste devono in ogni modo definirsi implicitamente o esplicitamente cacheable o no, in modo da prevenire che i client possano riutilizzare stati vecchi e dati errati.

Caratteristiche

Layered system: il sistema è realizzato “a strati” (*layer*). Ciò rende possibile, per esempio, pubblicare le API in un server, memorizzare i dati in un secondo server e gestire l'autenticazione delle richieste in un terzo server.

Code on demand *(opzionale)*: i server possono temporaneamente estendere o personalizzare le funzionalità del client trasferendo del codice eseguibile. Per esempio questo può includere componenti compilati come Applet Java o linguaggi di scripting lato client come per esempio JavaScript. Il concetto di Code on demand è l'unico vincolo opzionale per la definizione di un'architettura REST.

Uniform interface: Un'interfaccia di comunicazione omogenea tra client e server permette di semplificare e disaccoppiare l'architettura, per poterla modificare separatamente a blocchi.

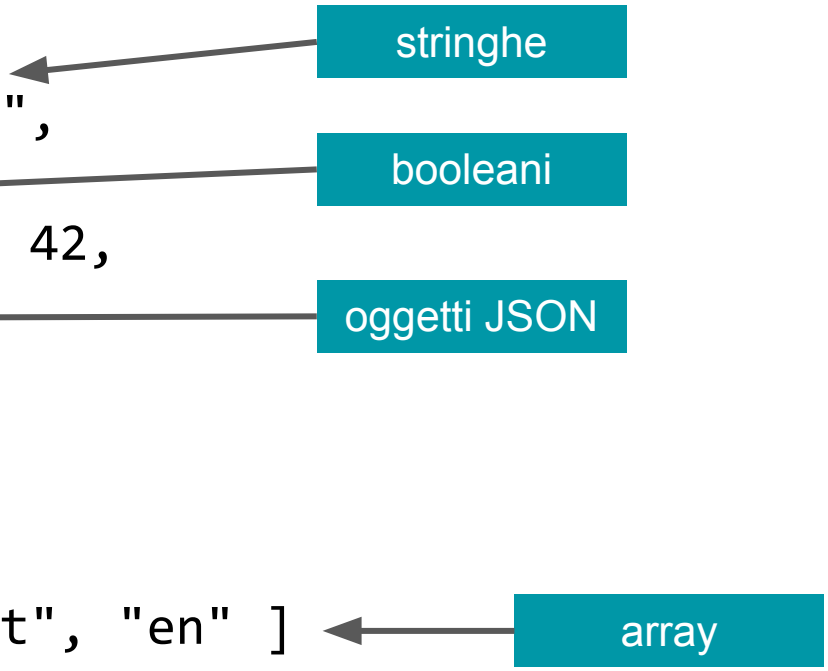
Le risorse

- Una risorsa è una fonte di informazioni alla quale si può accedere attraverso un **identificativo** univoco (URI)
- Per utilizzare tali risorse sia il client e il server comunicano attraverso un **protocollo** comune (HTTP)
- Per scambiarsi le risorse sia il client sia il server devono utilizzare un meccanismo di **rappresentazione** (JSON, XML)
- RESTful contrariamente ad altri protocolli non definisce standard per la rappresentazione o per le modalità di comunicazione

Rappresentazione dei dati

Un formato molto utilizzato per lo scambio dei dati è **JavaScript Object Notation (JSON)**. Un JSON è racchiuso tra parentesi graffe è può contenere diversi tipi di dati.

```
{
    "name": "Mario",
    "surname": "Rossi",
    "active": true,
    "favoriteNumber": 42,
    "birthday": {
        "day": 1,
        "month": 1,
        "year": 2000
    },
    "languages": [ "it", "en" ]
}
```



Definizione dei metodi

Progettare una API REST prevede la **definizione dei metodi/servizi**. Ogni metodo ha un suo path, identificativo univoco che coincide con la URL da utilizzare per richiedere il metodo tramite protocollo HTTP. Ad ogni metodo deve essere associato anche il relativo verbo HTTP da utilizzare.

Le URL possono avere dei parametri che possono essere utilizzati per scambiare informazioni con il server, ad esempio l'id della risorsa da recuperare.

A volte per rendere più semplice l'invocazione dei servizi è possibile *inserire i parametri direttamente nel path*.

Supponiamo di avere un metodo per il recupero di un libro da un store:

```
GET store.api.org/book?id={book-id}

GET store.api.org/book/{book-id}
```

Entrambe le soluzioni sono valide, anche se è preferibile evitare l'uso dei parametri nell'URL

Definizione dei metodi

Riprendendo l'esempio precedente il client potrebbe utilizzare uno dei due servizi semplicemente effettuando una richiesta HTTP/GET per ottenere un libro tramite il suo id (es. 12):

```
http://store.api.org/book?id=12
```

```
http://store.api.org/book/12
```

Una possibile risposta del server potrebbe essere:

```
{
   "id": 12,
   "title": "Alice in Wonderland",
   "price": 12.50
}
```

JAVA e JSON

Esistono diverse librerie per la gestione dei JSON all'interno di applicazioni JAVA:

- JSON-simple: https://github.com/fangyidong/json-simple
- Gson: https://github.com/google/gson

JSON-simple mette a disposizione delle API di più basso livello per la manipolazione/creazione di JSON.

Gson fornisce un'interfaccia di alto livello per la serializzazione e deserializzazione di oggetti Java direttamente in JSON.

Gson

```
public class Book {

    private String title;

    private String isbn;

    private String[] authors;

    private double price;

    public Book(String title, St
        this.title = title;
        this.isbn = isbn;
```

```
{
   "title":"Lo Zen e l'arte della
manutenzione della motocicletta",
   "isbn":"9788845902826",
   "authors":["Robert M. Pirsig"],
   "price":12.5
}
```

```
 */
public static void main(String[] args) {
    Book book = new Book("Lo Zen e l'arte della manutenzione della motocicletta",
                "9788845902826",
                new String[]{"Robert M. Pirsig"},
                12.50);
    Gson gson = new Gson();
    //Object -> JSON
    String jsonString = gson.toJson(book);
    System.out.println(jsonString);
    //JSON -> Object
    Book book2 = gson.fromJson(jsonString, Book.class);
    System.out.println(book2);
}
```

JAVA + RESTful = JAX-RS

- Jakarta RESTful Web Services, (JAX-RS) è una API di Java che fornisce supporto nella creazione di servizi Web secondo il modello architetturale REST. JAX-RS utilizza le **annotazioni**, per semplificare lo sviluppo e la distribuzione di client e endpoint di servizi Web.

esempio di annotazione, iniziano con il carattere '@'

```
@Path("book")
public class BookService {

    @GET
    @Produces("application/json")
    public Response getBookById(@DefaultValue("-1") @QueryParam("id") String bookid) {
        //CODE HERE
    }

    @GET
    @Path("/{bookid}")
    @Produces("application/json")
    public Response book(@PathParam("bookid") String bookid) {
        //CODE HERE
    }
```

Annotazioni JAX-RS

- @Path: specifica il path all'interno della URL dove sarà disponibile il servizio la cui logica è realizzata all'interno di un metodo. Il path può essere sia a livello di classe sia a livello di metodo
- @Produces/@Consumes: specifica il data type[1] che viene prodotto e/o consumato
- @GET, @POST, @DELETE…: specificano il verbo HTTP con il quale sarà richiamato il relativo servizio
- @QueryParam: identifica un parametro che verrà passato tramite parametro della URL
- @PathParam: identifica un parametro che verrà passato tramite path della URL

[1] https://en.wikipedia.org/wiki/Media_type

JAVA + RESTful = JAX-RS

```
@Path("book")
public class BookService {

    @GET
    @Produces("application/json")
    public Response getBookById(@DefaultValue("-1") @QueryParam("id") String bookid) {
        //CODE HERE
    }

    @GET
    @Path("/{bookid}")
    @Produces("application/json")
    public Response book(@PathParam("bookid") String bookid) {
        //CODE HERE
    }
```

1. definisce il “root” path associato a tutti i metodi presenti nella classe
2. definisce un metodo GET e il tipo di dati prodotto, inoltre viene definito un parametro di tipo query con valore di default =-1
3. viene definito un path aggiuntivo per questo metodo e un parametro (*bookid*) definito all’interno del path

Annotazioni JAX-RS

Esempio di annotazione per un metodo PUT:

```
@PUT
@Path("/add")
@Consumes("application/json")
@Produces("application/json")
public Response add(String json) {
    //CODE HERE
}
```

In questo caso il parametro `json` conterrà il contenuto della richiesta HTTP passato tramite PUT.

Logica dei metodi

Implementiamo la logica dei metodi per recuperare e salvare un book.

```
@Path("book")
public class BookService {

    @GET
    @Produces("application/json")
    public Response getBookById(@DefaultValue("-1") @QueryParam("id") String bookid) {
        System.out.println("Book id: " + bookid);
        Book book = new Book("Lo Zen e l'arte della manutenzione della motocicletta",
                "9788845902826",
                new String[]{"Robert M. Pirsig"},
                12.50);
        Gson gson = new Gson();
        //Object -> JSON
        String jsonString = gson.toJson(book);
        return Response.ok(jsonString, MediaType.APPLICATION_JSON).build();
    }
```

Questo è il servizio GET che ci permette di recuperare un Book tramite id come parametro di query della URL. In questo caso viene restituito sempre lo stesso book, nella realtà il book potrebbe essere recuperato da un DB o altra sorgente.

Logica dei metodi

```
@GET
@Path("/{bookid}")
@Produces("application/json")
public Response book(@PathParam("bookid") String bookid) {
    System.out.println("Book id: " + bookid);
    Book book = new Book("Lo Zen e l'arte della manutenzione della motocicletta",
            "9788845902826",
            new String[]{"Robert M. Pirsig"},
            12.50);
    Gson gson = new Gson();
    //Object -> JSON
    String jsonString = gson.toJson(book);
    return Response.ok(jsonString, MediaType.APPLICATION_JSON).build();
}
```

Questo è il servizio GET che ci permette di recuperare un Book tramite id come path della URL.

Da notare l'utilizzo della classe Response per costruire la risposta HTTP da restituire a seguito della chiamata.

Logica dei metodi

```
@PUT
@Path("/add")
@Consumes("application/json")
@Produces("application/json")
public Response add(String json) {
    Gson gson = new Gson();
    Book book = gson.fromJson(json, Book.class);
    System.out.println("Add book: " + book);
    return Response.ok().build();
}
```

Questo è il servizio PUT che ci permette di aggiungere un Book.

La PUT è un verbo HTTP il cui contenuto, in questo caso il JSON del book da aggiungere, è inserito direttamente nella richiesta HTTP. In questo caso il parametro json memorizza il contenuto della richiesta HTTP che viene convertito in oggetto utilizzando Gson.

Server HTTP

Dopo aver creato le classi che forniscono i servizi con le relative annotazioni è necessario avere a disposizione un **server HTTP** che si occupi di gestire le richieste dai vari client. E' possibile utilizzare dei container come *Apache Tomcat* o *Glassfish* che conterranno la relativa web application.

Per semplicità utilizzeremo **grizzly** che è un server HTTP già incluso in Jersey che un framework REST per Java che permette di realizzare sia client sia server REST.

- https://eclipse-ee4j.github.io/jersey.github.io/

Server HTTP grizzly

```
public class RESTServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        URI baseUri = UriBuilder.fromUri("http://localhost/").port(4321).build();
        ResourceConfig config = new ResourceConfig(BookService.class);
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        try {
            server.start();
            System.out.println(String.format("Jersey app started with WADL available at "
                    + "%sapplication.wadl\nHit enter to stop it...", "http://localhost:4321/"));
            System.in.read();
            server.shutdown();
        } catch (IOException ex) {
            Logger.getLogger(RESTServer.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

URI che sarà la root di tutti i servizi messi a disposizione da questo server. In questo caso: http://localhost:4321/

Server HTTP grizzly

```
public class RESTServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        URI baseUri = UriBuilder.fromUri("http://localhost/").port(4321).build();
        ResourceConfig config = new ResourceConfig(BookService.class);
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        try {
            server.start();
            System.out.println(String.format("Jersey app started with WADL available at "
                    + "%sapplication.wadl\nHit enter to stop it...", "http://localhost:4321/"));
            System.in.read();
            server.shutdown();
        } catch (IOException ex) {
            Logger.getLogger(RESTServer.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

Configura le risorse disponibili che verranno **automaticamente** create utilizzando le **annotazioni** presenti nella classe **BookService**

Server HTTP grizzly

```
public class RESTServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        URI baseUri = UriBuilder.fromUri("http://localhost/").port(4321).build();
        ResourceConfig config = new ResourceConfig(BookService.class);
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        try {
            server.start();
            System.out.println(String.format("Jersey app started with WADL available at "
                    + "%sapplication.wadl\nHit enter to stop it...", "http://localhost:4321/"));
            System.in.read();
            server.shutdown();
        } catch (IOException ex) {
            Logger.getLogger(RESTServer.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

Crea il server utilizzando la URI e i servizi presenti in config e avvia il server.

Server HTTP grizzly

```
public class RESTServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        URI baseUri = UriBuilder.fromUri("http://localhost/").port(4321).build();
        ResourceConfig config = new ResourceConfig(BookService.class);
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        try {
            server.start();
            System.out.println(String.format("Jersey app started with WADL available at "
                    + "%sapplication.wadl\nHit enter to stop it...", "http://localhost:4321/"));
            System.in.read();
            server.shutdown();
        } catch (IOException ex) {
            Logger.getLogger(RESTServer.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

Il server è attivo è può accettare le richieste HTTP relative ai servizi messi a disposizione.
Il main rimane in esecuzione fino alla pressione del tasto INVIO.

Client REST

Client

- Dopo aver realizzato i nostri servizi REST e reso disponibile il server HTTP siamo pronti ad accettare le richieste ricevute attraverso il protocollo HTTP
- Un client non deve far altro che preparare la richiesta HTTP in base alle specifiche del servizio e inoltrarla utilizzando il protocollo HTTP e il corrispettivo verbo previsto dal servizio
- Java mette a disposizione nel suo framework delle classi per gestire le richieste HTTP
- Le classi standard di Java lavorano ad un livello troppo basso che ne rendono complesso l'utilizzo. Il framework Jersey oltre a permettere la realizzazione del server mette a disposizione anche delle classi per realizzare in modo semplice la componente client

Client

Utilizzando il framework Jersey è semplice inviare delle richieste attraverso l'utilizzo delle classi `Client` e `WebTarger`.
Ad esempio, per utilizzare i servizi creati in precedenza:

```
public static void main(String[] args) {
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target("http://localhost:4321");
    Response resp = target.path("book").queryParam("id", "1").request(MediaType.APPLICATION_JSON).get();
    System.out.println(resp);
    System.out.println(resp.readEntity(String.class));

    resp = target.path("book/1").request(MediaType.APPLICATION_JSON).get();
    System.out.println(resp);
    System.out.println(resp.readEntity(String.class));
```

Il `client` permette di effettuare le connessioni HTTP in modo semplice. Il `WebTarget` memorizza la URL dove sono disponibili i servizi e attraverso i metodi `path`, `queryParam`, `request` permette di costruire la richiesta HTTP.

L'oggetto di tipo `Response` contiene il risultato della risposta HTTP.

Client

Esempio di richiesta PUT con invio di un JSON:

```
Gson gson = new Gson();
Book book = new Book("Lo Zen e l'arte della manutenzione della motocicletta",
        "9788845902826",
        new String[]{"Robert M. Pirsig"},
        12.50);
resp=target.path("book/add").request(MediaType.APPLICATION_JSON)
        .put(Entity.entity(gson.toJson(book), MediaType.APPLICATION_JSON));
System.out.println(resp);
```

In questo caso il metodo `put` prende in input una `entity` che è il contenuto da inviare al server tramite la richiesta PUT.


NOT SURE IF REST
STANDS FOR RELAXATION
imgflip.com

Esercizi

1. Utilizzando la classe `Articolo` dell'esercizio sulle collection[1] realizzare un servizio REST che permetta l'aggiunta, il recupero e la cancellazione di un `Articolo` all'interno di una **base di dati**
2. Realizzare un client che utilizzi i servizi REST realizzati al punto 1
3. Utilizzando la documentazione presente a questo indirizzo https://openweathermap.org/current realizzare due metodi che recuperano le condizioni meteo in base al nome della città e in base alle coordinate geografiche (latitudine, longitudine). Per utilizzare il servizio è necessario ottenere una api-key registrandosi gratuitamente qui https://openweathermap.org/price

[1] `di.uniba.map.b.lab.collection.esercizi`