Annotazioni JAX-RS

L'obbiettivo di questa specifica è di consentire, attraverso questo meccanismo, il mapping tra i contenuti informativi presenti all'interno dei messaggi di richiesta/risposta HTTP con elementi di un programma java. Associare quindi le richieste a classi/metodi.

Attraverso le annotazioni realizziamo un meccanismo di injection di valori estratti dai messaggi che vengono passati ai metodi. Quando scriviamo i metodi java di questi servizi non dobbiamo prevederne l'invocazione, perchè l'invocazione sarà a carico di un'ambiente di esecuzione (ad esempio tomcat).

Annotazioni CRUD - recap

- @POST: creazione di una risorsa e ritorna la sua uri al client
- @GET: ritorna la risorsa identificata dalla uri
- @PUT: aggiorna la risorsa identificata dalla uri
- @DELETE: rimuove le risorse identificate dalla uri specificata nel messaggio di richiesta

E' importante che l'uri venga associata al metodo java che viene mandato in esecuzione quando il messaggio di richiesta viene ricevuto.

URI - @Path

L'annotazione path può essere usata sia con riferimento ad un metodo (impiegata per etichettare un metodo ed associarlo ad un percorso) oppure usata sulla classe; in questo caso stiamo definendo l'uri di base per tutti i metodi presenti nella classe.

```
@Path("/books")
class BookService {
    @GET
    @Path("/10909")
    public Book getBookDetails(...) {

    }
}
```

The method is used with URI :
http://baseURI/books/10909

Quando il server riceve un messaggio di richiesta con quella uri verifica quale metodi ha un'uri corrispondente e seleziona quel metodo per l'esecuzione; sempre ammesso che nel messaggio di richiesta venga usato anche il metodo GET.

Queste annotazioni, quindi, consentono di costruire un meccanismo di instradamento verso i metodi java.

Altre annotazioni

Queste informazioni possono essere trasferite ai metodi come parametri; possiamo estrarre informazioni di diversa natura:

- @QueryParam: estraiamo parametri presenti nella query stream contenuta nell'uri
- @FormParam: estraiamo i parametri passati attraverso un'interazione di tipo POST realizzata mediante un form.

- @PathParam: recuperiamo un parametro da un frammento del path; possiamo quindi usarlo come parametro formale.

Questo tipo di parametri vengono trasferiti attraverso il **body del messaggio di richiesto**, e non dalla request line.

Esempio di @PathParam

```
@GET
@Path("/{isbn}")
public Book getBookDetails(@PathParam("isbn") String isbn) { }
```

In questo esempio vediamo un metodo annotato @Path, ma a differenza dell'esempio precedente non troviamo un path hardcoded, ma troviamo "/{isbn}"; questa stringa è un identificatore, ovvero il **nome del parametro che useremo nella definizione del metodo**. L'isbn viene poi recuperato come parametro @PathParam("isbn"), che ci dice che vogliamo recuperare il parametro isbn dall'uri.

Esempio di @QueryParam

```
@GET
public Books getBooksByTopic(@QueryParam("topic") String topic)
  {}
```

Usiamo l'annotazione `@QueryParam("topic") String topic`. Ci aspettiamo quindi che quando il client produrrà il messaggio di richiesta, prevederà nell'uri qualcosa del tipo `?topic="topicValue"`.

JAX-RS rappresentazione delle risorse

Il fatto che si parli di "rappresentazioni" lascia pensare al fatto che sia possibile usare diverse rappresentazioni. Poichè sono possibili diversi tipi di rappresentazione, dobbiamo esplicitare qual è il formato che vogliamo rappresentare, sia per il contenuto che viene trasferito al server, sia per quanto riguarda il contenuto che viene restituito al client.

Per specificare il **content type** possiamo usare le annotazioni **@Produces e @Consumes**:

- **@Consumes**: specifichiamo quali sono i media type che il servizio è in grado di accettare
- **@Produces:** specifichiamo i tipi di media type che il server produce

HTTP status codes

Questi codici possono essere usati per segnalare al client (ad esempio) e possono essere usati per capire se un'operazione è andata a buon fine o meno.

Abbiamo 5 classi:

- **1xx: informazionale**: ci dice che la richiesta prodotta dal client è valida, ma per completare la richiesta è necessario che il client produca ulteriori scambi con il server
- **2xx: successo**, la richiesta è stata ricevuta e non ci sono errori
- 3xx: redirection, il server ha ricevuto il messaggio ma non può accettarlo; è necessario che il client faccia altro. Ad esempio ill client deve fare un'altra richiesta ad un altro server.
- **4xx: errore client**, errore nel messaggio di richiesta e quindi non può essere soddisfatta. Ad esempio abbiamo fatto una richiesta per una risorsa ma non siamo autenticati.

- **5xx: errore server**, il messaggio di richiesta potrebbe essere valido ma il server non è in grado di produrre il messaggio di risposta; ad esempio il tentativo di accesso ad una risorsa produce a tempo di esecuzione un'eccezione.

Sono codici di 3 cifre, ed abbiamo 5 classi.

Codici più diffusi

- **101 - Switching Protocols**: abbiamo analizzato in che modo si possa passare da HTTP/1.1 ad HTTP/2.0, il server risponde con un codice di stato che indica che ha accettato lo switching del protocollo.
- **200 OK**: Ha un significato diverso a seconda del tipo di richiesta si effettua (POST o GET). Con GET significa che la risorsa è presente, mentre con POST significa che nel body troviamo il risultato dell'operazione.
- **202 Creato**: se il msg di risposta contiene il codice 201 vuol dire che la risorsa è stata creata sul server, e l'uri della risorsa creata è recuperabile dal campo location del messaggio di risposta.
- **204 No Content**: indichiamo che non è presente un contenuto nel messaggio di risposta; può essere usato quando cancelliamo una risposta.
- **301 Moved Permanently**: indica al client che la risorsa richiesta è stata spostata su un altro server, il cui uri è recuperabile dal campo location della risposta.
- **304 Not Modified**: ci indica che la risorsa che abbiamo richiesto non è stata modificata. E' una ridirezione alla cache del browser.
- **400 Bad Request**: indica che abbiamo dei campi mancanti della richiesta
- **401 Non autorizzato**: indica che la risorsa richiesta non può essere autenticata senza un'autenticazione. Il messaggio di risposta contiene anche un campo **WWW-Authenticate** che viene usato per far autenticare il client.
- **403 Proibito**: la risorsa richiesta non può essere a ceduta, anche con autenticazione.
- **404 Not Found**: la risorsa richiesta non è presente.
- **500 Internal Server Error**:
- **501 Non implementato**: la funzionalità non è stata ancora implementata.

Esercizi JAX-RS

L'implementazione che useremo è **Jersey**

Esercizio 10.1

Implement, deploy and test a simple RESTful service to perform the multiplication of two real numbers.

Le applicazioni di tipo Spring Boot prevedono di essere lanciate così come lanciamo solitamente un programma java, e non troviamo quindi più il server all'interno del quale dispieghiamo l'applicazione. Il server viene avviato nel momento in cui decidiamo di far partire l'applicazione java.

Dobbiamo scrivere una classe Products il cui codice da vita al servizio per effettuare il prodotto tra due numeri. Dobbiamo prevedere, all'interno della classe, delle annotazioni che ci permettono di attivare l'esecuzione di un metodo quando sarà ricevuto un messaggio di richiesta.

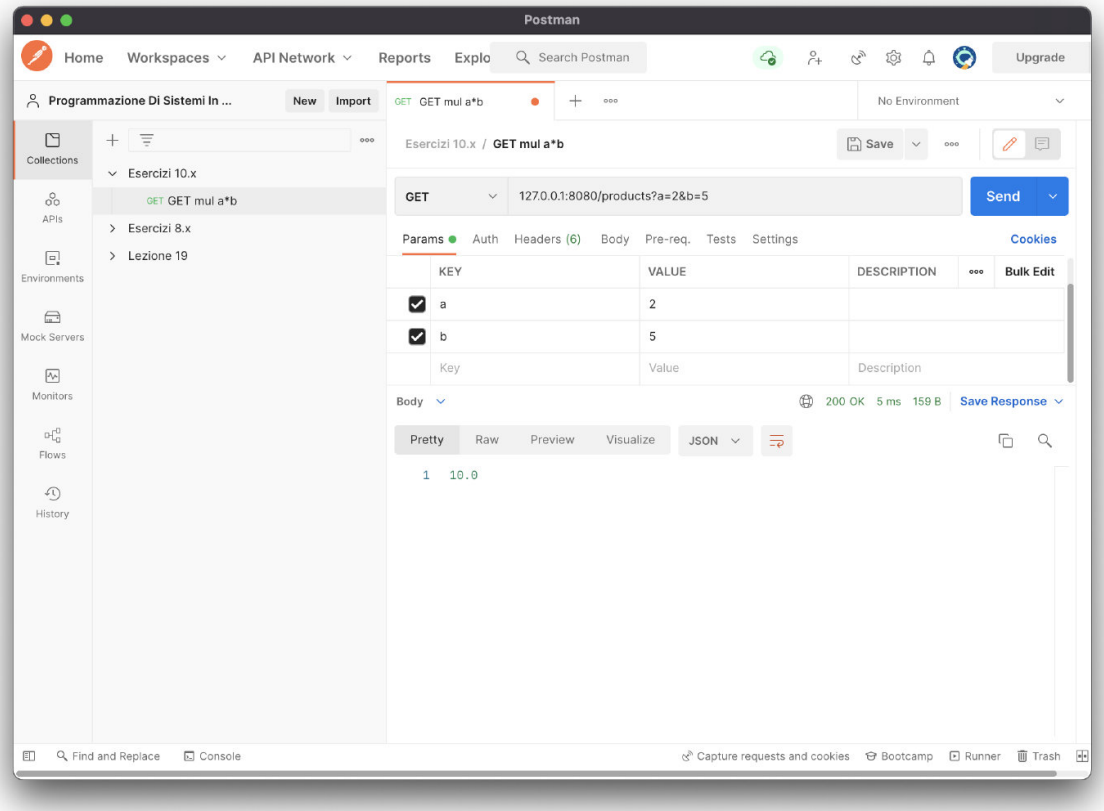
Non ci dimentichiamo di modificare il codice di Application:

```java
import org.glassfish.jersey.server.ResourceConfig;

@SpringBootApplication
public class Application extends ResourceConfig{
    public Application() {
        register(Products.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

> non ci dimentichiamo di includere jersey

Con Products.class java mette a disposizione delle classi particolari che sono usate per catturare gli eventi. Con `register(Products.class)` stiamo dicendo al framework la classe che vogliamo usare per instanziare il servizio.

A questo punto ci basta mandare in esecuzione l'applicazione ed effettuare una query con Postman:



Come possiamo vedere, l'uri di base da indicare non è più quella di prima, ovvero dove dobbiamo specificare l'uri che identifica la locazione del server (ad esempio 127.0.0.1:8080/**Esercizio9.x/path**), ma ci basterà indicare il path specificato all'interno della classe (/products).

Per impostare un'uri di base ci basta specificare all'interno di **application.properties** partirà dalla "root" /Esercizio10.1 (o quello che specificheremo).

Esercizio 10.2

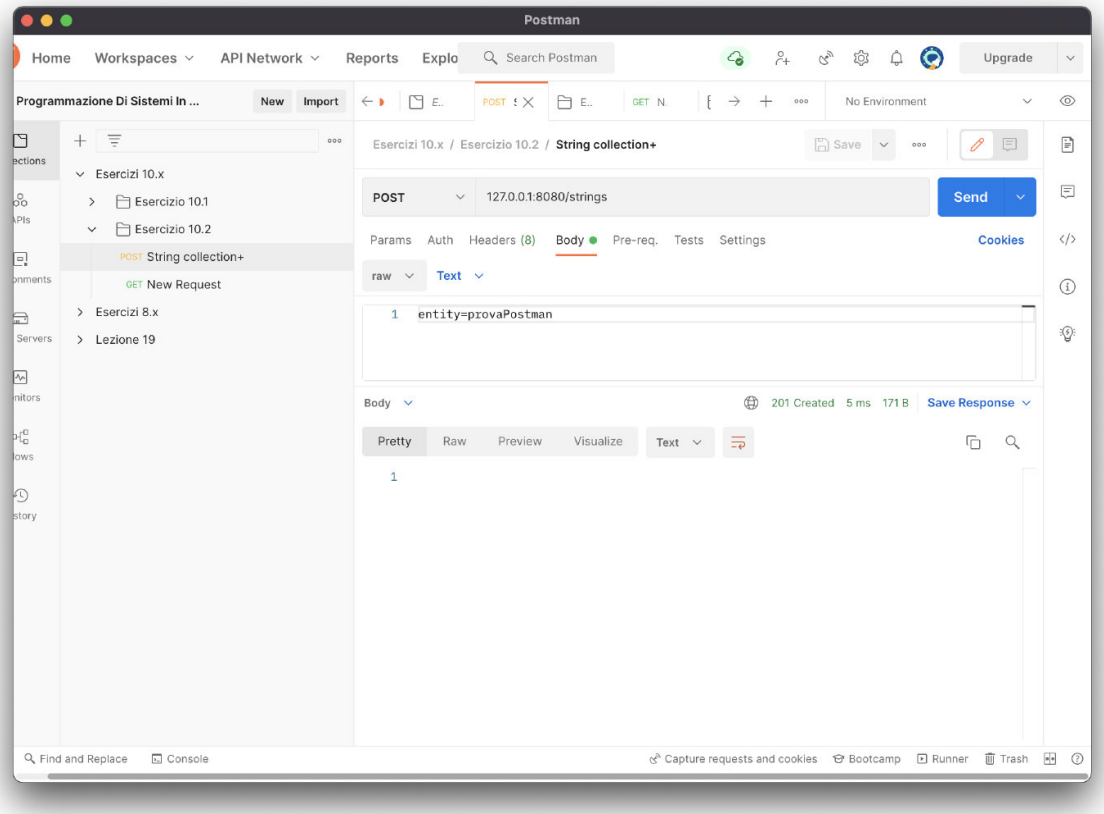
Implement, deploy and test a simple RESTful service to handle a collection of strings. Consider to implement an operation to create a new string in the collection and an operation to get a string specified by an id.

Visto che vogliamo che la lista all'interno della classe sia mantenuta, invece di passare **la classe**, passiamo **l'oggetto**, in modo da non perdere lo stato dell'applicazione ogni volta.

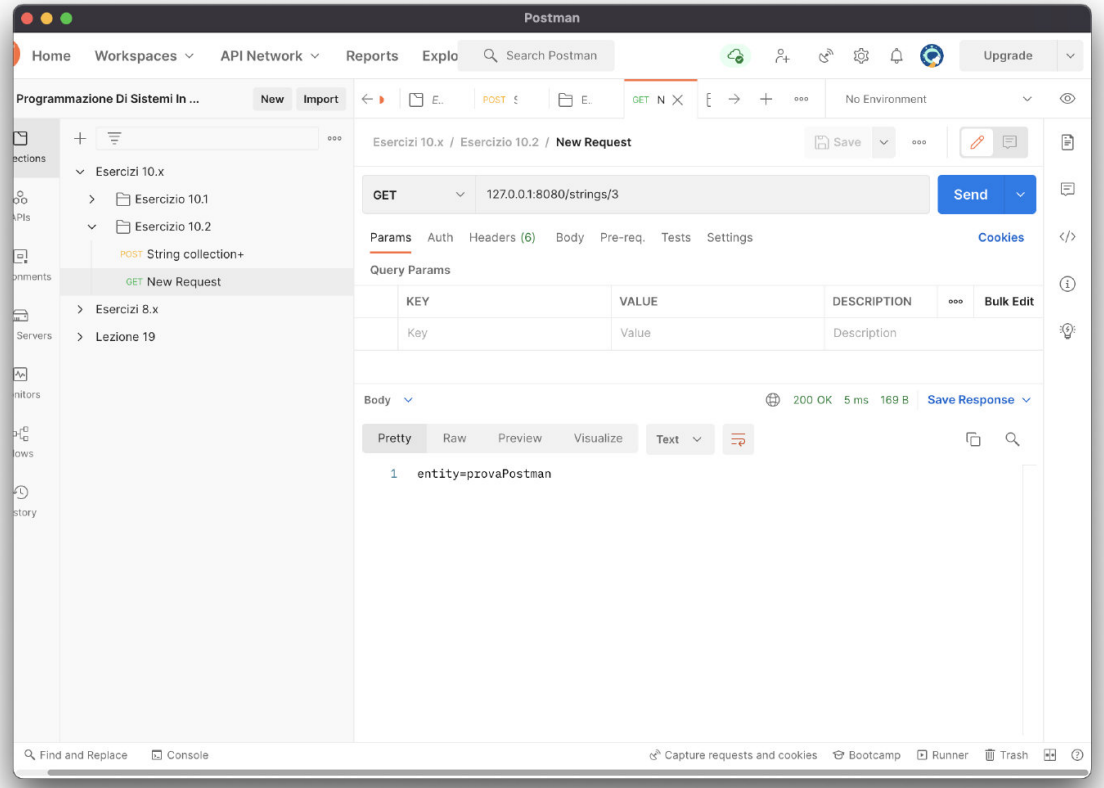
Ci basta usare il client per inviare una richiesta prima POST e poi GET:

```
<terminated> StringsClient [Java Application] /Applications/Eclipse.app/Contents/Eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_15.0.2.v20210201-0955/jre/bin/java  (8 feb 2022, 11:33:15 – 11:33:17)
Stringa di richiesta: Entity{entity=second, variant=Variant[mediaType=text/plain, language=null, encoding=null], annotations=[]}
Hai creato una stringa: second
http://127.0.0.1:8080/strings/1

Stringa richiesta: second
```

Oppure usare postman prima per inviare la POST:



E poi la GET:



fine lezione 34