

# Laboratorio di “Sistemi Distribuiti”

## A.A. 2024-2025



REST

Emanuele Petriglia

Slide aggiornate al 22 Maggio 2025

# Indice dei contenuti

1. Introduzione
2. Interrogazione di API REST
  - JSON in 5 minuti
  - ARC
3. Implementazione di API REST
  - JAX-RS e JSON-B
  - Esercizio A
  - Esercizio B
4. Progettazione di API REST
  - Linee guida
  - Esercizio C

# 1. Introduzione

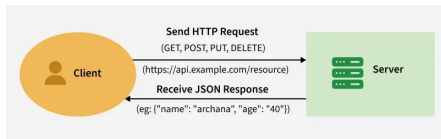
# Obiettivo

**Scorso laboratorio:** il server Web restituisce file HTML.

- I file HTML contengono sia dati sia la loro presentazione e la struttura della pagina Web.

**Questo laboratorio:** il server Web espone delle API REST che scambiano dati in formato JSON.

- Il client (browser) riceve codice HTML e JavaScript solo una volta ed ottiene i dati tramite le API. È una *single-page application* (SPA).
- Backend e frontend sono disaccoppiati.
  - Ci possono essere più client (browser e app Android/iOS).
  - Backend e frontend possono essere sviluppati in modo indipendente.

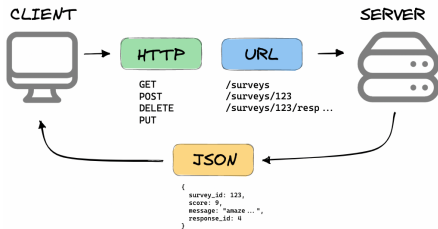


Fonte



# API REST

- **REST: Representational State Transfer** è uno stile architetturale per sistemi distribuiti.
- **Basato su HTTP**: Usa metodi HTTP (GET, POST, PUT e DELETE).
- **Senza stato**: non c'è un concetto di sessione o cronologia tra le richieste.
- **Risorse**: organizzate in collezioni.



Fonte

## 2. Interrogazione di API REST

## 2. Interrogazione di API REST

JSON in 5 minuti

# JSON in 5 minuti

- JSON → **J**ava**S**cript **O**bject **N**otation
- È un formato testuale per lo scambio di dati tra applicazioni.
  - “Human-readable”: leggibile sia dagli umani sia dalle macchine.
  - Supportato da JavaScript, Java e molti altri linguaggi.
- Tipi di dati supportati (più info su [Learn X in Y minutes](#)):
  - Stringhe: `"ciao mondo!"`,
  - Numeri: `42`, `3.14`,
  - Oggetti: `{"chiave": "value", "abc": 42}`,
  - Array: `[1, 2, 3]`,
  - Altro: `true`, `false` e `null`
- Esempio:
  - 1 `{"name": "Emanuele", "age": 26,`
  - 2  `"isStudent": true, "courses": ["PhD", "CS"],`
  - 3  `"details": { "height": 165, "weight": 60 } }`





## 2. Interrogazione di API REST

ARC

# Requisiti

Chi interroga un'API REST?

- Un client che usa il browser (e quindi HTML/JavaScript). È il browser che effettua le chiamate all'API.
- Un'applicazione nativa (es. su Android). È l'applicazione stessa che effettua le chiamate all'API.
- Strumenti specializzati per chiamare un'API.

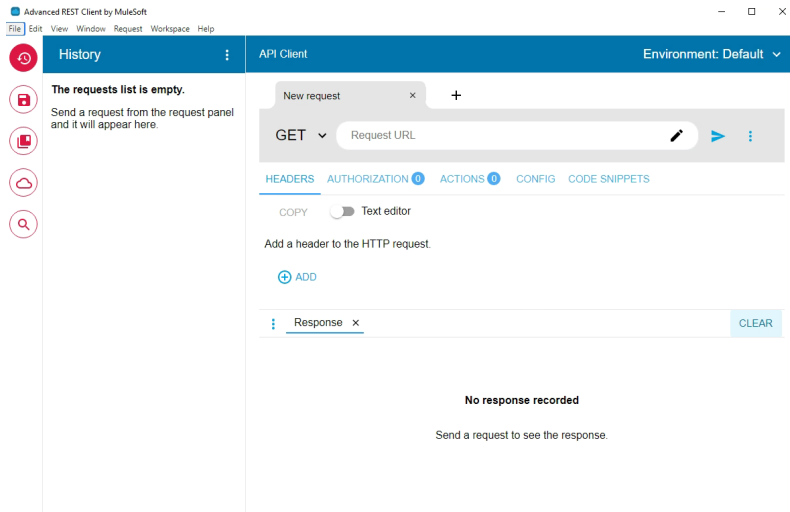
Non avendo (per adesso) un client, useremo [ARC \(Advanced REST Client\)](#), preinstallato nella VM.

Che API interroghiamo? Per adesso l'API REST esposta dal progetto `helloworld-rest`, disponibile su e-Learning, che gestisce una lista di utenti.



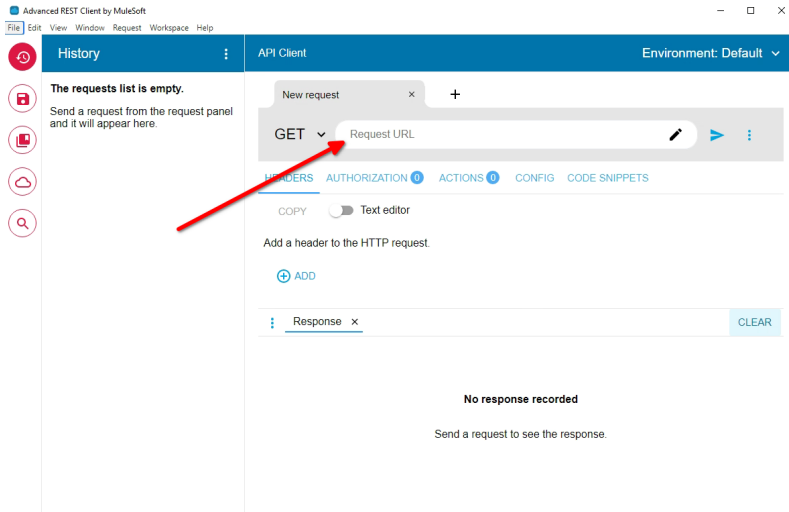
# Avvio di ARC (pt. 1)

Avviare ARC dall'icona sul Desktop:



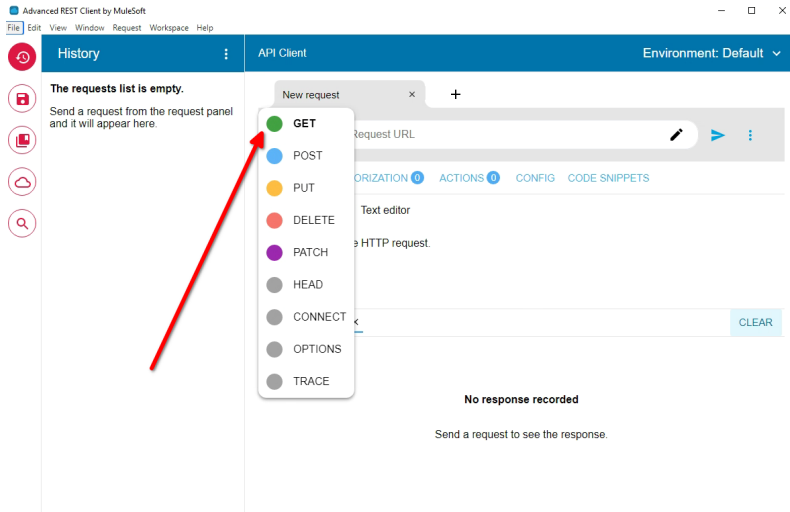
# Avvio di ARC (pt. 2)

Qui si inserisce l'URL della risorsa.



# Avvio di ARC (pt. 3)

Qui si può selezionare il metodo HTTP da usare.



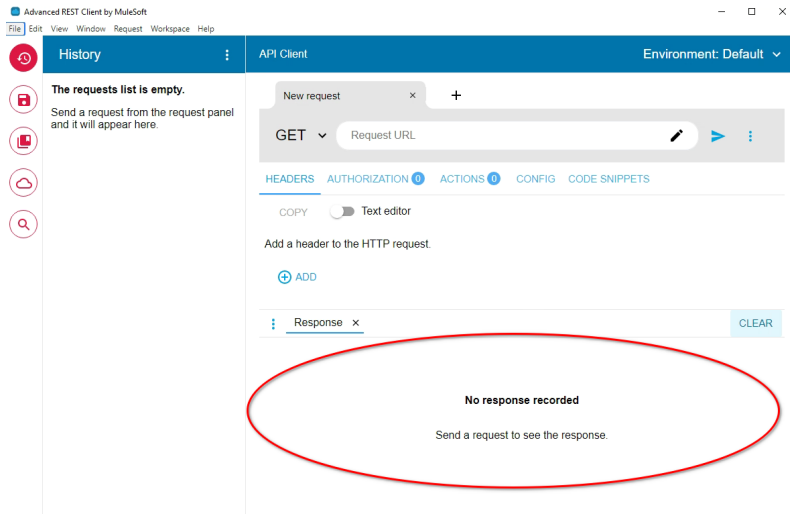
# Avvio di ARC (pt. 4)

Si possono aggiungere header.

The screenshot shows the Advanced REST Client (ARC) interface. The left sidebar contains a 'History' panel with a message: 'The requests list is empty. Send a request from the request panel and it will appear here.' The main area is titled 'API Client' and 'Environment: Default'. Below this is a 'New request' tab with a dropdown set to 'GET' and a 'Request URL' field. The 'HEADERS' tab is selected, showing a table with columns 'Name' and 'Value'. A red arrow labeled '1' points to a toggle switch on the left of the table. The first row in the table has 'Accept' in the 'Name' column and 'text/plain' in the 'Value' column. A red arrow labeled '2' points to the 'Accept' header name, and another red arrow labeled '3' points to the 'text/plain' value. Below the table is an 'ADD' button. At the bottom, there is a 'Response' tab with a 'CLEAR' button and the text 'No response recorded' and 'Send a request to see the response.'

# Avvio di ARC (pt. 5)



Il contenuto del body della risposta sarà mostrato in basso:



# Avvio del server (pt. 1)

- ❶ Scaricare da e-Learning la cartella `helloworld-rest.zip` e decomprimerla.
- ❷ Aprire la cartella `helloworld-rest` con Visual Studio Code.
- ❸ Aprire un terminale ed eseguire il seguente comando:

```
> mvn jetty:run
```

Il comando avvia un'istanza di Jetty ed espone l'API REST su <http://localhost:8080/>. Per interrompere premere  .





## Avvio del server (pt. 2)

- Il server di esempio espone i seguenti endpoint:

Metodo	URL	Descrizione
GET	/users	Restituisce l'elenco degli utenti.
POST	/users	Aggiunge un nuovo utente.
GET	/users/{id}	Restituisce uno specifico utente.
PUT	/users/{id}	Aggiorna i dati di un utente.

- Ogni utente è caratterizzato da:

Attributo	Tipo	Descrizione
id	intero	Identificativo dell'utente.
name	stringa	Nome dell'utente.
admin	booleano	Se è un amministratore.

- Le rappresentazioni sono in formato JSON (`application/json`) a eccezione dei metodi GET che supportano anche il formato testuale (`text/plain`).



# Interrogazione GET (pt. 1)

Per cominciare interrogare l'endpoint `/users` con il metodo GET e farsi restituire la rappresentazione in formato testuale.

http://localhost:8080/users × +

GET `http://localhost:8080/users` ✎ ▶ ⋮

HEADERS AUTHORIZATION 0 ACTIONS 0 CONFIG CODE SNIPPETS

COPY ☐ Text editor

Name	Value
Accept	text/plain

+ ADD

⋮ Response × Headers CLEAR

200 OK Time: 11 ms Size: 99 Bytes ⋮

```
1 [{"size": 2, "elements": [{"id": 1, "name": "Luca", "admin": false}, {"id": 2, "name": "Laura", "admin": true}]]
```



# Interrogazione GET (pt. 2)

Riprovare la stessa chiamata di prima ma con la rappresentazione in formato JSON.

The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:8080/users`
- Method:** GET
- Headers:** A table with one header row and one data row.

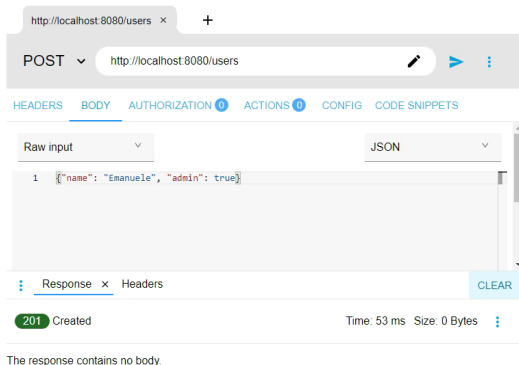
Name	Value
Accept	application/json
- Response:** 200 OK. Time: 156 ms, Size: 75 Bytes.
- Response Body (JSON):**

```
1 [
2   {
3     "admin": false,
4     "id": 1,
5     "name": "Luca"
6   },
7   {
```



# Interrogazione POST

Per fare la chiamata POST `/users`, è necessario scrivere nel body il formato JSON dell'utente da aggiungere e l'header `Content-Type: application/json`.



Nella scheda "Headers" della risposta è presente l'header `Location` che contiene l'URL dell'utente appena creato.



## Altre interrogazioni

Provare a fare le seguenti interrogazioni, prestando attenzione al corpo ed header sia della richiesta sia della risposta e al codice di stato restituito:

- GET /users/1 (sia in formato testuale che JSON),
- GET /users/999 (utente non esistente),
- POST /users (con un JSON non valido),
- PUT /users/1 (con un nuovo nome in JSON),
- PUT /users/1 (con un JSON non valido).

### Attenzione

Per PUT, è sempre necessario scrivere l'intero formato dell'utente in JSON. Il campo `id` viene sempre ignorato perché gestito lato server.



### 3. Implementazione di API REST

### 3. Implementazione di API REST

JAX-RS e JSON-B

# Come sono implementate le API REST?

Come per le **Jakarta Servlet**, si fa uso ulteriori due specifiche di **Jakarta EE**:

- **Jakarta RESTful Web Services** (JAX-RS) 3.1.0: specifiche e API per creare webapp che espongono API REST ([Sito Web](#), [Javadoc](#));
- **Jakarta JSON Binding** (JSON-B) 3.0.1: specifiche e API per convertire oggetti Java in JSON e viceversa ([Sito Web](#), [Javadoc](#)).



Problema: nel codice Java si usano le specifiche JAX-RS e JSON-B, ma sono senza implementazione.

Soluzione: si usa il framework [Jersey](#), implementa tutte e tre le specifiche. In più si usa Jetty per avviare il server Web.





# Info sugli esercizi

- Per i prossimi esercizi (e per il progetto) non è necessario modificare il file `pom.xml` di Maven.
  - Per avviare il server bisogna sempre eseguire:
    - > `mvn jetty:run` # *Metodo preferito.*
    - > `mvn package` # *Oppure il file WAR per il deploy.*
- Il primo comando avvia un'istanza di Jetty ed espone le API REST su <http://localhost:8080/>. Per interrompere premere  .



# La classe User

```
1  // Deve essere una Plain Old Java Object (POJO).
2  public class User {
3      private int id;
4
5      private String name;
6
7      private boolean admin;
8
9      // Getter e setter...
10 }
```

Attenzione!

Da qui in poi gli `import` per brevità non sono mostrati.



# La classe UserResource

```
1  // Risponde a "http://localhost/users".
2  @Path("users")
3  public class UserResource {
4      static private List<User> users = new ArrayList<User>();
5
6      static {
7          var user = new User();
8          user.setId(1);
9          user.setName("Luca");
10         users.add(user);
11
12         // ...
13     }
14     // ...
15 }
```

## Attenzione!

Jersey a ogni richiesta crea una nuova istanza di `UserResource`, ecco perché gli attributi sono statici. Si può evitare con `@Singleton`.



# GET /users in testo semplice

```
1  @Path("users")
2  public class UserResource {
3      @GET // GET "/users"
4      @Produces(MediaType.TEXT_PLAIN) // "Content-Type: text/plain"
5      public String getUsers() {
6          var buf = new StringBuffer();
7          buf.append(String.format("[size = %d, elements = [",
8              ↪ users.size()));
9
10         List<String> userFormatted = new ArrayList<>(users.size());
11         synchronized (users) {
12             for (var user : users) {
13                 userFormatted.add(user.toString());
14             }
15         }
16
17         buf.append(String.join(", ", userFormatted));
18         buf.append("]]");
19
20         return buf.toString();
21     }
22 }
```



# GET /users in JSON

```
1  @Path("users")
2  public class UserResource {
3      @GET // GET "/users"
4          // "Content-Type: application/json"
5      @Produces(MediaType.APPLICATION_JSON)
6      public List<User> getUsersJson() {
7          return users;
8      }
9      // ...
10 }
```

Dove avviene la conversione da `List<User>` in JSON?

Jersey, vedendo il `MediaType`, utilizza in automatico il modulo di conversione e converte l'oggetto in una stringa JSON!

# GET /users/{id} in JSON

```
1  @Path("users")
2  public class UserResource {
3      @Path("/{id}")
4      @GET // GET "/users/{id}"
5      @Produces(MediaType.APPLICATION_JSON)
6      public Response getUserJson(@PathParam("id") int id) {
7          synchronized (users) {
8              for (var user : users)
9                  if (user.getId() == id)
10                     return Response.ok(user).build();
11          }
12
13          return Response.status(Response.Status.NOT_FOUND).build();
14      }
15      // ...
16  }
```

Response è una classe con tanti metodi utili per specificare il body, header e codici di stato da restituire ([Javadoc](#)). Bisogna sempre chiamare build() alla fine!



# POST /users

```
1  @Path("users")
2  public class UserResource {
3      @POST // POST "/users"
4      @Consumes(MediaType.APPLICATION_JSON)
5      public Response addUser(User user) {
6          if (user.getName() == null) {
7              return Response.status(Status.BAD_REQUEST).build();
8          }
9
10         user.setId(users.size() + 1);
11         synchronized (users) { users.add(user); }
12
13         try {
14             var uri = new URI("/users/" + user.getId());
15
16             // Imposta in automatico l'header "Location".
17             return Response.created(uri).build();
18         } catch (URISyntaxException e) {
19             return Response.serverError().build();
20         }
21     }
22     // ...
23 }
```



## PUT /users/{id}

```
1  @Path("users")
2  public class UserResource {
3      @Path("/{id}")
4      @PUT // PUT "/users{id}"
5      @Consumes(MediaType.APPLICATION_JSON)
6      @Produces(MediaType.APPLICATION_JSON)
7      public Response setUser(@PathParam("id") int id, String rawUser) {
8          User oldUser = null;
9          synchronized (users) {
10              for (var user : users)
11                  if (user.getId() == id) { oldUser = user; break; }
12          }
13
14          if (oldUser == null) {
15              return Response.status(Status.NOT_FOUND).build();
16          }
17
18          var jsonb = JsonbBuilder.create();
19          try {
20              var user = jsonb.fromJson(rawUser, User.class);
21
22              synchronized (oldUser) {
23                  oldUser.setName(user.getName());
24                  oldUser.setAdmin(user.isAdmin());
25              }
26              return Response.ok(oldUser).build();
27          } catch (JsonbException e) {
28              return Response.status(Status.BAD_REQUEST).build();
29          }
30      }
31      // ...
32  }
```





# Chi espone le API?

La classe `jakarta.ws.rs.core.Application` implementata da Jersey carica in automatico tutte le risorse e metodi decorati con `@Path` e `@Provider`.

Ci sono tanti modi per fare il deploy, quello predefinito con le Servlet è un file `webapp/WEB-INF/web.xml` che contiene:

```
1  <web-app version="5.0">
2      <servlet>
3          <servlet-name>jakarta.ws.rs.core.Application</servlet-name>
4      </servlet>
5
6      <servlet-mapping>
7          <servlet-name>jakarta.ws.rs.core.Application</servlet-name>
8          <url-pattern>/*</url-pattern>
9      </servlet-mapping>
10 </web-app>
```



### 3. Implementazione di API REST

#### Esercizio A

## Consegna (pt. 1)

- Implementare un'API REST con i seguenti endpoint:

Metodo	URL	Descrizione
GET	/keyboards/	Restituisce una <b>mappa</b> delle tastiere (id, tastiera).
POST	/keyboards/	Aggiunge una nuova tastiera.
GET	/keyboards/{id}	Restituisce una specifica tastiera.
DELETE	/keyboards/{id}	Rimuove una specifica tastiera.

- Le informazioni sulla tastiera sono: id, name, manufacturer, year, ergonomic, backlight. id è **generato dal server**.
- La rappresentazione è **solo** in formato JSON.
- Partire dallo scheletro iniziale** presente su e-Learning.
- Attenzione alla gestione della concorrenza!



## Consegna (pt. 2)

Gli endpoint devono restituire i seguenti codici HTTP (P = POST, G = GET, D = DELETE):

Codice	Metodo	Descrizione
200 OK	G	Successo, risorsa restituita.
201 Created	P	Successo, risorsa creata.
204 No content	D	Successo, risorsa eliminata.
400 Bad request	P, G, D	Errore client, input malformato.
404 Not found	G	Errore client, risorsa non esistente.

### Attenzione

Il metodo POST in caso di successo bisogna restituire una risposta con body vuoto e l'header `Location` che contiene l'URL alla risorsa appena creata!



### 3. Implementazione di API REST

#### Esercizio B

## Consegna (pt. 1)

A partire dall'esercizio precedente aggiungere i seguenti endpoint:

Metodo	URL
GET	/keyboards/{id}/comments/{comment_id}
DELETE	/keyboards/{id}/comments/{comment_id}
POST	/keyboards/{id}/comments/
PUT	/keyboards/{id}/{property}
GET	/keyboards/{id}/{property}

I primi tre gestiscono i commenti su una tastiera. Gli ultimi due permettono di accedere o sovrascrivere una singola proprietà della tastiera.

Attenzione a gestire la concorrenza!



## Consegna (pt. 2)

- Ogni commento ha un ID specifico per la tastiera a cui fa riferimento e una stringa come contenuto.
- GET /keyboards/{id}/{comments} restituisce l'intera lista (o mappa) dei commenti.
- PUT /keyboards/{id}/{property} su id e comments restituisce il codice HTTP 405 Method Not Allowed.
- PUT /keyboards/{id}/{property} restituisce il codice HTTP 400 Bad Request se il dato è malformato, 404 Not Found se la proprietà non esiste, 204 No content per successo.
- GET /keyboards/{id}/{property} restituisce 404 Not Found se la proprietà non esiste.



## Suggerimenti (pt. 1)

- **Domanda:** come gestire i commenti per una tastiera?

**Soluzione:** creare una classe `Comment` e inserire una mappa o una lista di commenti nella classe `Keyboard`. Jersey in automatico converte da/a JSON anche questo attributo.

- **Domanda:** in POST `/keyboards/{id}/comments/` come gestire il body?

**Soluzione:** Jersey quando vede un parametro Stringa mette dentro il body senza deserializzare da JSON. La conversione va fatta a mano:

```
1  @Path("/{id}/comments")
2  @POST
3  @Consumes(MediaType.APPLICATION_JSON)
4  public Response addComment(@PathParam("id") int id, String rawContent) {
5      try {
6          // rawContent contiene l'intero body in formato JSON,
7          // ma non sappiamo se è malformato o no.
8          var jsonb = JsonbBuilder.create(); // Parser JSON.
9          var content = jsonb.fromJson(rawContent, String.class);
10     } catch (JsonbException e) {
11         return Response.status(Status.BAD_REQUEST).build();
12     }
13 }
```





## Suggerimenti (pt. 2)

- **Domanda:** Per PUT /keyboards/{id}/{property} come trattare il body?

**Soluzione:** usare uno `switch` e fare il parsing manuale del JSON dal body in base al parametro:

```
1  var jsonb = JsonbBuilder.create();
2  switch (property) {
3      case "name":
4          var name = jsonb.fromJson(body, String.class);
5          keyboard.setName(name);
6          return Response.noContent().build();
7      case "year":
8          var year = jsonb.fromJson(body, Integer.class);
9          keyboard.setYear(year);
10         return Response.noContent().build();
11         // ...
12         default: // Proprietà non trovata.
13             return Response.status(Status.NOT_FOUND).build();
14     }
```



## Suggerimenti (pt. 3)

- **Domanda:** Per GET /keyboards/{id}/{property} perché se restituisco name o manufacturer esce un JSON non valido?

**Risposta:** perché se si fa

```
Response.ok(keyboard.getName()).build();
```

si bypassa la conversione in JSON di Jersey.

**Soluzione:** convertire le stringe a JSON a mano:

```
1 var jsonb = JsonbBuilder.create();  
2 var nameJson = jsonb.toJson(keyboard.getName());  
3 return Response.ok(nameJson).build();
```



## 4. Progettazione di API REST

## 4. Progettazione di API REST

Linee guida

# Linee guida

- ➊ Trovare i sostantivi e costruire gli endpoint (es. un insieme di utenti diventa `/users`),
- ➋ Definire il formato di rappresentazione (JSON),
- ➌ Selezionare le operazioni possibili (GET, POST...),
- ➍ Specificare che codici di stato restituire.

Dove descrivere le API? Esistono formati appositi (es. [OpenAPI](#)). Per questo laboratorio (e progetto) usare un file testuale semplice (Markdown).

## Attenzione

- Distinguere le **collezioni** dalle **singole risorse**.
- Attenzione all'uso e differenza di **PUT** e **POST**!

Rivedere le slide della lezione su REST (Argomento 5).



## 4. Progettazione di API REST

### Esercizio C

# Consegna

Progettare e implementare un'API REST, testarla quindi con ARC.

## Problema

L'API permette la gestione di una rubrica telefonica di un singolo utente. L'utente deve poter aggiungere, rimuovere o aggiornare un contatto. In un contatto sono salvate le informazioni sul nome, cognome, uno o più numeri di telefono, email e data di nascita. L'email e la data di nascita è opzionale. Per ogni numero di telefono, oltre al numero, si salva anche un'etichetta (es. "Casa", "Lavoro"). Un numero di telefono può essere associato a più contatti.

L'utente deve poter aggiungere o rimuovere i singoli numeri di telefono associati a un contatto. L'utente deve poter cercare tra i contatti in base a un numero di telefono fornito.

**Partire dallo scheletro iniziale** presente su e-Learning.



# Suggerimenti

- Per la ricerca si possono adottare più strade, due esempi:
  - ① Usare i **parametri query** (es. `?searchBy=XX`). In tal caso si può usare l'annotazione di JAX-RS `@QueryParam`.
  - ② Definire un percorso (es. `/search/{XX}`) e usare i parametri di percorso con `@PathParam`.
- La documentazione sulle API va scritta in un file testuale in modo simile a quello presente nello scheletro.
- Il parsing di una mappa in JSON da una stringa si fa nel modo seguente:

```
1 Map<?, ?> rawPhoneNumbers = jsonb.fromJson(body, Map.class);
2 Map<String, String> phoneNumbers = new HashMap<String,
  ↪ String>(rawPhoneNumbers.size());
```

