

Programmazione Web

Lez. 13

Servlet Java

Giuseppe Psaila

Università di Bergamo

giuseppe.psaila@unibg.it

Apache Tomcat

Apache Tomcat

- È il web server scritto in Java che consente di scrivere server-side script in Java.
- Sito web ufficiale:
`https://tomcat.apache.org/`
- dal quale si può scaricare la versione per il proprio sistema operativo.

Avviare Apache Tomcat

- Su Windows:
- Dal menu delle applicazioni, cercare la cartella **Apache Tomcat xxx**
- All'interno, selezionare **Configure Tomcat**
- Autorizzare l'accesso ai dati di configurazione

Avviare Apac



Apache Tomcat 9.0 Tomcat9 Properties

General Log On Logging Java Startup Shutdown

Service Name: Tomcat9

Display name: Apache Tomcat 9.0 Tomcat9

Description: Apache Tomcat 9.0.88 Server - https://tomcat.apache

Path to executable:
"C:\Program Files\Apache Software Foundation\Tomcat 9.0\bin\Tomcat9.

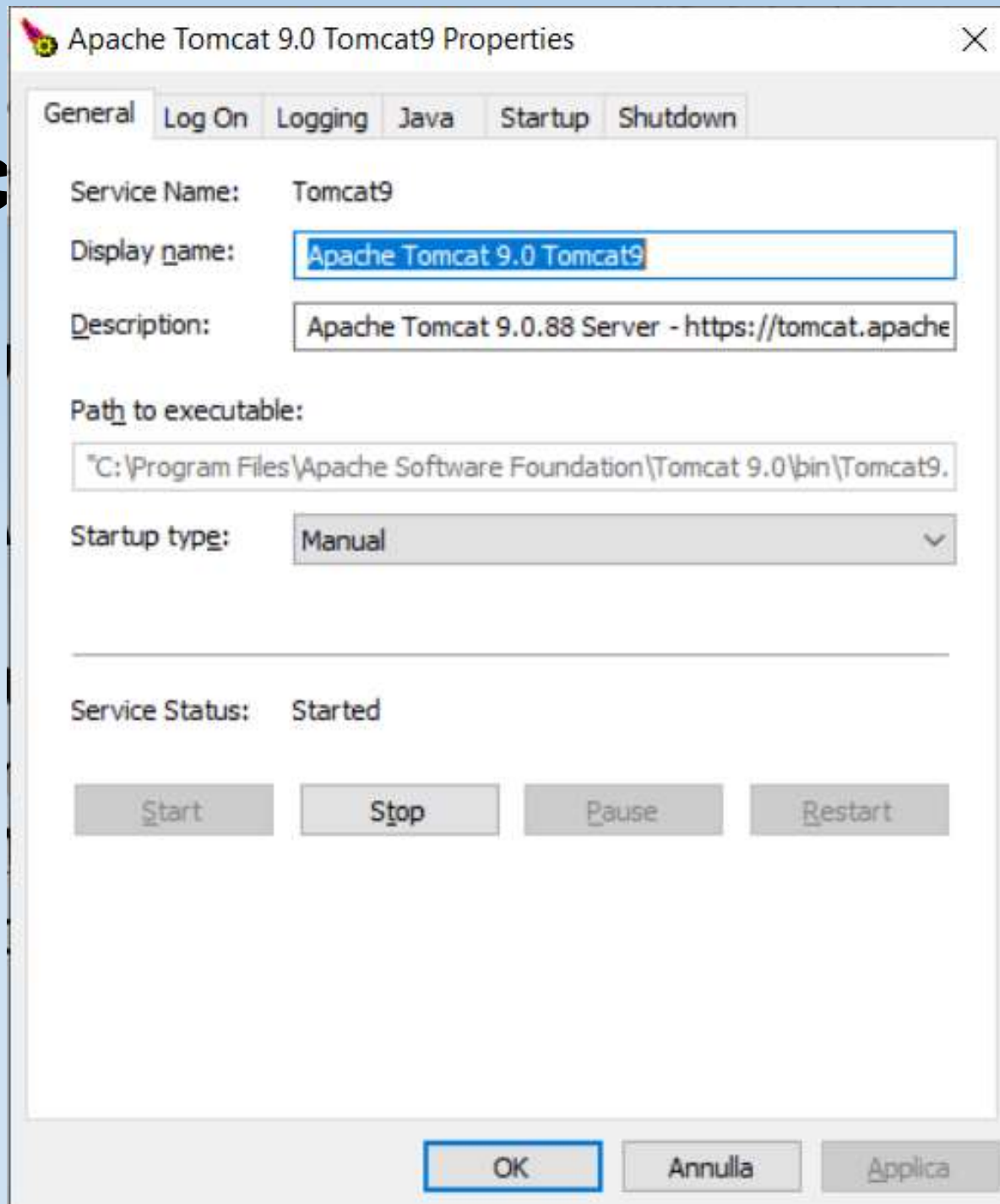
Startup type: Manual

Service Status: Stopped

Start Stop Pause Restart

OK Annulla Applica

Avviare Apac



Che Tipo di Tecnologia?

- Servlet
classi Java che vengono invocate come risorsa dalla richiesta HTTP
- JSP
Java Server Pages, cioè pagine HTML con annegato codice Java, che possono essere rese dinamiche con specifici elementi XML.

Che cosa Vediamo

- Come sono fatte le servlet.
- Come connettersi ad un DBMS relazionale (PostgreSQL).
- Come gestire i documenti JSON in Java.

Applicazioni Web Tomcat

- Tomcat organizza il server in «Applicazioni».
- Ogni applicazione ha uno spazio su disco dedicato.
- Nella cartella
C:\Program Files\Apache Software
Foundation\Tomcat 9.0\webapps
- Tra le applicazioni pre-installate, troviamo
ROOT
examples

Applicazioni Web Tomcat

- **ROOT**
è raggiungibile direttamente dal dominio
- **examples**
è raggiungibile con il percorso
/examples

Applicazioni Web Tomcat

La cartella dell'applicazione contiene:

- I file HTML dell'applicazione (con eventuali sotto-cartelle)
- La cartella **WEB-INF**, al cui interno si trova il codice dell'applicazione

Applicazioni Web Tomcat

Nella cartella **WEB-INF**, troviamo:

- Il file **web.xml** che configura l'applicazione
- La cartella **classes**, al cui interno troviamo il codice delle servlet
- La cartella **lib**, al cui interno troviamo le librerie Java (in formato jar)

Applicazioni Web Tomcat

Per creare una nuova applicazione?

- Basta copiare la cartella di un'applicazione già esistente

Esempio:

- Ho creato l'applicazione «**my_examples**» copiando la cartella «**examples**»

Servlet

Creazione di una Servlet

- Il codice della servlet può essere preparato in qualsiasi cartella, l'importante è che il ByteCode della classe venga copiato nella cartella «**classes**».
- Vediamo i passi nelle slide seguenti.

Passo 1: Compilazione

- Una volta scritto il codice della classe, compilarlo con `javac`
- Attenzione alle librerie: serve `servlet-api.jar` che si trova in
`C:\Program Files\Apache Software Foundation\Tomcat 9.0\lib\`

Passo 2: Configurazione

- Nella cartella **WEB-INF**, si trova il file **web.xml**
- Al suo interno, occorre dichiarare la nuova servlet e mapparla su un URL

- Dichiarazione:

```
<servlet>
```

```
    <servlet-name>Hello</servlet-name>
```

```
    <servlet-class>Hello</servlet-class>
```

```
</servlet>
```

Passo 2: Configurazione

- Mapping con un URL:

```
<servlet-mapping>
```

```
  <servlet-name>Hello</servlet-name>
```

```
  <url-pattern>/Hello</url-pattern>
```

```
</servlet-mapping>
```

Attenzione: l'URL viene appeso a quello dell'applicazione, quindi
`/my_examples/Hello`

Che Cosa è una Servlet

- È una classe Java che viene invocata dal web server.
- I package necessari sono:
`javax.servlet`
`javax.servlet.http`
- Una servlet estende la classe `HttpServlet`

Struttura

- La servlet estende la classe `HttpServlet`
- Essenzialmente, deve fare l'override di due metodi:
`doGet`
`doPost`
- che vengono invocati in base al metodo HTTP della chiamata

Esempio

```
import java.io.IOException;  
import java.io.PrintWriter;  
import java.util.ResourceBundle;  
  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;
```

Esempio

```
public class Hello extends HttpServlet {  
    @Override  
    public void doGet(  
        HttpServletRequest request,  
        HttpServletResponse response)  
        throws IOException, ServletException  
    {  
...  
    }
```

La Richiesta HTTP

- Parametro `request`
oggetto della classe `HttpServletRequest` .
- Consente di accedere alla richiesta HTTP
per ottenere la query string
o per ottenere il contenuto della richiesta (metodo
POST).
- Fra poco vediamo come.

La Risposta HTTP

- Parametro **response**
oggetto della classe **HttpServletResponse** .
- Consente di preparare la risposta
inviando il contenuto da produrre .
- Come? Attraverso un oggetto **PrintWriter**
PrintWriter out =
response.getWriter() ;

La Risposta HTTP: MIME Type

- Metodo `response.setContentType(String)`
imposta il MIME type della risposta

Esempio

```
{ PrintWriter out = response.getWriter() ;  
  response.setContentType("text/html") ;  
  out.println("<!DOCTYPE html><html>") ;  
  out.println("<body bgcolor=\"white\">") ;  
  out.println("<h1>" + title + "</h1>") ;  
  out.println("</body>") ;  
  out.println("</html>") ;  
}
```

Richiesta (**HttpServletRequest**)

Vediamo alcuni metodi dell'oggetto request

- **String request.getQueryString()**
fornisce la stringa con la query string estratta dall'URL
- **String request.getRequestURL()**
fornisce l'URL
- **String request.getServletPath()**
estrae il path dall'URL

Richiesta (ServletRequest)

Metodi ereditati da ServletRequest

- `String request.getParameter(name)`
fornisce il valore del parametro con il nome indicato (null se non esiste)
- `Map<String, String[]>`
 `request.getParameterMap()`
Fornisce una mappa con tutti i parametri (i parametri possono avere molti valori, quindi serve un array di String)

Esempio: Servlet Params

Obiettivo:

- Reagire al metodo GET
- Acquisire tutti i parametri
- Generare una pagina HTML con il loro valore

Esempio: Servlet Params (1/3)

```
{  
    response.setContentType("text/html");  
    PrintWriter out =  
        response.getWriter();  
    java.util.Map<String, String[]> params =  
        request.getParameterMap();  
    Object [] keys =  
        params.keySet().toArray();  
}
```

Spiegazione

- Estraiamo la mappa delle chiavi, con il nome **params**.
- Poi, estraiamo l'insieme delle chiavi (di tipo **Set**) e lo trasformiamo in array.
- **keyset** è un array di **Object**, ognuno dei quali lo convertiremo in **String**.

Esempio: Servlet Params (1/3)

```
out.println("<!DOCTYPE html>");  
out.println("<html>");  
out.println("<body>");  
  
if (keys.length==0)  
    out.println("<p>No params</p>");  
else
```


Spiegazione

- Se l'insieme delle chiavi è vuoto, si stampa il relativo messaggio.

Esempio: Servlet Params (1/3)

```
{
    for(int i=0; i<keys.length; i++ )
    {
        String item = (String)keys[i];
        String value = params.get(item) [0];
        out.println("<p>" + item +
                    "=" + value + "</p>");
    }
}
out.println("</body>"); out.println("</html>");
} // doGet
```

Spiegazione

- Se l'insieme delle chiavi non è vuoto,
- si scandisce l'array e per ogni chiave si estrare il valore.
- Per semplicità, prendiamo solo il primo valore dell'eventuale insieme di valori del singolo parametro.
- Notate il casting della chiave verso il tipo **String**

Metodo POST

- Il metodo `doPost` reagisce alla chiamata fatta con il metodo POST.
- Se la servlet è invocata da una form di HTML, i parametri sono contenuti nel corpo della richiesta.
- L'oggetto `request` gestisce i parametri nello stesso modo in cui li gestisce per il metodo GET.

Metodo POST: Corpo

- Se la servlet viene invocata da una chiamata AJAX o come web service, si deve poter acquisire il corpo della richiesta,
- che potrebbe essere un documento JSON
- o un documento XML.
- L'oggetto request fornisce i metodi
`getInputStream()`
`getReader()`

Metodo POST: Corpo

```
StringBuilder buffer = new StringBuilder();  
BufferedReader reader = request.getReader();  
String line;  
while ((line = reader.readLine()) != null) {  
    buffer.append(line);  
    buffer.append(System.lineSeparator());  
}  
String data = buffer.toString();
```

Metodo POST: Corpo

- In questo modo si acquisisce il contenuto del corpo della richiesta
- che viene messo in una stringa
- attraverso uno **StringBufer**.

Multi-threading

- Ogni invocazione dei metodi `doGet` e `doPost` avviene in un thread dedicato.
- In questo modo, il server è in grado di processare richieste multiple in parallelo,
- eseguendo sia la stessa servlet tante volte in modo parallelo o eseguendo tante servlet diverse in parallelo.

Accesso ai DB Relazionali

Protocollo JDBC

JDBC

- Java
- DataBase
- Connectivity
- Le interfacce del protocollo JDBC sono standardizzate nel package `java.sql`

Download Driver per PostgreSQL

- Il driver può essere scaricato da
[`https://jdbc.postgresql.org/download/`](https://jdbc.postgresql.org/download/)

Processo

Abbiamo capito che il processo è:

- Attivazione della connessione con il database
- Esecuzione delle query
- Chiusura della connessione

Il JDBC funziona allo stesso modo per tutti i DBMS relazionali. Noi lo vediamo con PostgreSQL.

Caricare il Driver

Per prima cosa, occorre caricare il driver. Non si deve importare il package, occorre caricarlo in modo specifico.

Ci sono due modi, che possiamo chiamare

- Versione originale
- Versione moderna

Caricare il Driver

Versione originale

```
Class.forName("org.postgresql.Driver");  
String url =  
"jdbc:postgresql://localhost:5432/MyDB?us  
er=MyUser&password=MyPwd";  
Connection conn =  
    DriverManager.getConnection(url);
```

Caricare il Driver

**Versione Moderna
(che a me non ha funzionato)**

```
String url =  
"jdbc:postgresql://localhost:5432/MyDB?us  
er=MyUser&password=MyPwd";  
Connection conn =  
    DriverManager.getConnection(url);
```

Caricare il Driver

Secondo quanto dichiarato, il **DriverManager** dovrebbe essere in grado di caricare il driver giusto in modo automatico, partendo dalla stringa di connessione.

Se questo non succede, occorre caricare in modo esplicito il driver, con

```
Class.forName("org.postgresql.Driver");
```

(catturare l'eccezione **ClassNotFoundException**).

Apertura della Connessione

In entrambi i casi, il `DriverManager` apre la connessione con il database

```
String url =  
"jdbc:postgresql://localhost:5432/MyDB?us  
er=MyUser&password=MyPwd";  
Connection conn =  
    DriverManager.getConnection(url);
```

Apertura della Connessione

Il tipo `Connection` è un'interfaccia del package `java.sql`.

Da questo momento in poi, useremo costrutti del JDBC.

Creare uno Statement

L'oggetto `Connection` fornisce un metodo per creare uno `Statement`, che poi eseguirà la query.

```
Statement stmt =  
    conn.createStatement();
```

Eseguire la Query

Il codice SQL viene fornito come parametro del metodo `executeQuery` dello `Statement`

```
String q =  
    "SELECT \"Name\", \"Age\" +  
    \" FROM \"Names\";\"  
ResultSet rs =  
    stmt.executeQuery( q );
```

Eseguire la Query

Ricordatevi che, in PostgreSQL, i nomi dei campi e delle tabelle devono essere racchiusi tra virgolette; per questo, usiamo il carattere di \"

```
String q =
```

```
    "SELECT \"Name\", \"Age\" +  
    \" FROM \"Names\" ;
```

Scansione del Result Set

Un semplice ciclo `while` scandisce il result set

```
while ( rs.next() ) {  
    String name =  
        rs.getString("Name") ;  
    int age     = rs.getInt("Age") ;  
    out.println("<p>Name: " + name +  
        " Age: " + age + "</p>") ; }  
}
```

Scansione del Result Set

Il metodo `next()` di un result set restituisce `false` se non vi sono più righe da acquisire

```
while ( rs.next() ) {
```

Acquisizione dei Campi

Ai valori dei campi si accede tramite il loro nome.
Ma occorre usare un metodo specifico per ogni tipo di dato

```
String  name =  
    rs.getString("Name") ;  
int age   = rs.getInt("Age") ;
```


Eccezioni

È obbligatorio catturare le eccezioni, quando si usano i metodi del JDBC.

L'eccezione è **SQLException**

Transazioni

La gestione delle transazioni non è diversa da quanto visto con il driver per Python.

- Di default, una nuova connessione è in auto-commit mode, cioè ogni query è una transazione.
- `conn.setAutoCommit(false) ;`
per disattivare l'auto-commit mode.
- `conn.commit()` effettua il commit
- `conn.rollback()` effettua il rollback

GSON

GSON

- È una libreria rilasciata da Google
- per serializzare/deserializzare oggetti Java in JSON.
- Se stiamo realizzando una servlet per comunicazione AJAX o come web service, diventa fondamentale saperla usare.

Download

- lo ho trovato la libreria a questo indirizzo:
`https://jar-download.com/artifacts/com.google.code.gson/gson/2.8.2/source-code`
- Ma se usate i repository lo trovate facilmente

Come Funziona

Serializzazione

- Dato un oggetto Java, anche complesso (con array e altri oggetti al suo interno),
- lo serializza in JSON (non devono esserci riferimenti circolari).

Come Funziona

Deserializzazione

- Data una classe Java (che può usare al suo interno array e altre classi),
- il documento JSON viene deserializzato nella struttura dati radicata nella classe scelta.
- In pratica, la struttura dati viene ricostruita, se è possibile riconoscere le classi da utilizzare.

Basi

- Package da importare
`import com.google.gson.*;`
- Occorre creare un oggetto sulla classe `Gson`,
- Tramite un oggetto `GsonBuilder`

```
Gson gs=  
    new GsonBuilder()  
        .setPrettyPrinting().create();
```


Serializzazione

- L'oggetto Gson effettua sia la serializzazione che la deserializzazione.
- Serializzazione di un oggetto n (in una stringa)

`gs.toJson(n)`

Deserializzazione

- Per prima cosa, occorre definire la classe target (per un documento JSON simile)

```
public class Nominativo
{
    String Name;
    int Age;
}
```

Deserializzazione

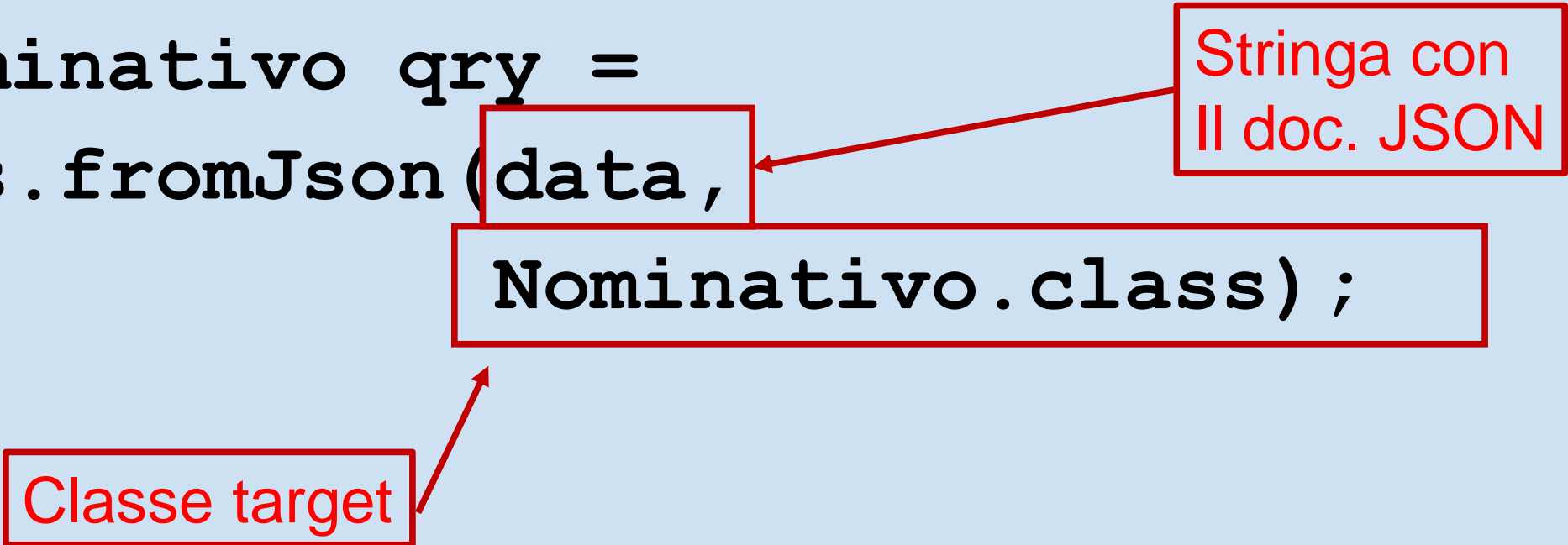
- Avendo una stringa di nome `data` con il documento JSON, effettuiamo la deserializzazione

```
Nominativo qry =
```

```
gs.fromJson(data,
```

```
Nominativo.class);
```

Stringa con
Il doc. JSON



Classe target

In caso di Fallimento?

- GSON potrebbe fallire nella deserializzazione se
- due classi target (annidate) diverse possono essere scelte a partire dallo stesso documento JSON.
- L'oggetto `GsonBuilder` può essere personalizzato, per realizzare una «**custom JSON deserialization**».
- Noi non lo vediamo, se vi dovesse servire, sapete che cosa cercare.

Esempio: Server per AJAX

- La pagina web invia una richiesta AJAX, per selezionare i nominativi nella tabella del DB in base a nome ed età, come documento JSON.
- La servlet deve deserializzare il documento JSON, fare la query, produrre un nuovo documento JSON con il risultato.

Classi per i Documenti JSON

```
public class Nominativo {  
    String Name;  
    int Age;  
  
    public Nominativo() { }  
    public Nominativo(String n, int a) {  
        Name = n;    Age = a;  
    }  
}
```

Classi per i Documenti JSON

La classe `Nominativo` serve

- Per deserializzare la richiesta
- Come elemento del documento JSON di risposta

Classi per i Documenti JSON

```
public class Result
{
    public int elementi=0;
    ArrayList<Nominativo> list;

    public Result()
    {
        list = new ArrayList<Nominativo>();
    }
}
```


Classi per i Documenti JSON

```
public void append(String Name, int Age)
{
    list.add(new Nominativo(Name, Age) );
    elementi++;
}
}
```

Classi per i Documenti JSON

Il documento in output avrà:

- Un campo **elementi**, che indica gli elementi nel risultato
- Un campo **list**, che è una lista di documenti di tipo **Nominativo**

Metodo Utile

```
private String addToCond(String prevcond,  
                          String newpiece) {  
    String r;  
    if (prevcond.length() == 0)  
        r = " WHERE " + newpiece;  
    else  
        r = prevcond + " AND " + newpiece;  
    return r;  
}
```

Metodo Utile

Il metodo `addToCond` crea la clausola **WHERE** della query

- Se la condizione iniziale è vuota, aggiunge la parola **WHERE**
- Se la condizione iniziale non è vuota, aggiunge la parola **AND**

Metodo doPost

```
public void doPost(  
    HttpServletRequest request,  
    HttpServletResponse response)  
    throws IOException, ServletException  
{  
    response.setContentType("application/json");  
    PrintWriter out = response.getWriter();
```

Estrazione Doc. JSON

```
StringBuilder buffer = new StringBuilder();  
BufferedReader reader = request.getReader();  
String line;  
while ((line = reader.readLine()) != null) {  
    buffer.append(line);  
    buffer.append(System.lineSeparator());  
}  
String data = buffer.toString();
```

Deserializzazione Doc. JSON

```
Gson gs=  
new GsonBuilder().setPrettyPrinting().create();  
Nominativo qry =  
    gs.fromJson(data, Nominativo.class);  
String WName = qry.Name;  
String WAge = String.valueOf(qry.Age);  
  
Result r = new Result();
```

Deserializzazione Doc. JSON

- Le stringhe **WName** e **WAge** contengono i valori che sono stati ricevuti tramite il documento JSON deserializzato come oggetti **Nominativo**.
- Un oggetto **Result** è creato e assegnato alla variabile **r**.

Prepararsi per la Query

```
try
{
    Class.forName("org.postgresql.Driver");
    String url =
"jdbc:postgresql://localhost:5432/MyDB?user=MyU
ser&password=MyPwd";
    Connection conn =
DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();
```

Preparare la Query

```
String Cond="";  
if(WName != null && WName.length()>0)  
    Cond = addToCond(Cond,  
                      "\"Name\" = '" + WName + "'");  
if(WAge != null && WAge.length()>0)  
    Cond = addToCond(Cond, "\"Age\" = " + WAge);  
String q =  
"SELECT \"Name\", \"Age\" FROM \"Names\""  
    + Cond + ";;";
```

Eseguire la Query

```
ResultSet rs = stmt.executeQuery( q );
```

```
while ( rs.next() )
```

```
{
```

```
    String name = rs.getString("Name");
```

```
    int age = rs.getInt("Age");
```

```
    r.append(name, age);
```

```
}
```

Chiudere le Connessioni

```
rs.close();  
stmt.close();  
conn.close();  
}
```

Catturare le Eccezioni

```
catch (SQLException e) {  
    out.println("SQL error: " +  
        e.getMessage());  
}  
  
catch (ClassNotFoundException e) {  
    out.println("COnnection error: " +  
        e.getMessage());  
}
```

Serializzare e Inviare il Documento JSON

```
out.println(gs.toJson(r));  
}
```

Documento Ricevuto

```
{ "Name" : "Pippo",  
  "Age" : 30  
}
```

Documento Inviato

```
{ "elementi": 1,  
  "list": [ { "Name": "Pippo", "Age": 30 } ]  
}
```