Serverlet Java

13.1 - Apache Tomcat

Apache Tomcat è un web server open source scritto in Java che implementa le specifiche Java Servlet, JavaServer Pages (JSP), Java Expression Language e Java WebSocket. In sostanza, permette di eseguire applicazioni web basate su queste tecnologie.

Tecnologie Supportate

Tomcat supporta principalmente:

- **Servlet:** Classi Java che estendono le funzionalità di un server. Vengono invocate tramite richieste HTTP per generare risposte dinamiche.
- **JSP (Java Server Pages):** Pagine HTML che possono contenere codice Java "annegato". Permettono di creare contenuti web dinamici, anche con l'uso di specifici elementi XML (tag library).

13.2 - Applicazioni Web Tomcat

Tomcat organizza il server in "Applicazioni Web". Ogni applicazione ha uno spazio su disco dedicato.

- **Posizione delle applicazioni:** Solitamente nella cartella `C:\Program Files\Apache Software Foundation\Tomcat X.Y\webapps` (dove X.Y è la versione di Tomcat).
- **Applicazioni pre-installate:**
  - `ROOT` : Applicazione di default, raggiungibile direttamente dal dominio (es. `http://localhost:8080/`).
  - `examples` : Applicazione con esempi, raggiungibile con il percorso `/examples` (es. `http://localhost:8080/examples/`).

Struttura di una Cartella di Applicazione

La cartella di un'applicazione web contiene tipicamente:

- **File HTML dell'applicazione:** (con eventuali sotto-cartelle) - Questi sono i file statici o JSP.

- **Cartella `WEB-INF`:** Contiene risorse non accessibili direttamente dal client web, ma usate dall'applicazione.
  - **`web.xml`:** Il descrittore di deployment. Configura l'applicazione, incluse servlet, mapping URL, parametri di inizializzazione, ecc.
  - **Cartella `classes`:** Contiene i file `.class` delle servlet e altre classi Java dell'applicazione.
  - **Cartella `lib`:** Contiene le librerie Java (file `.jar`) necessarie all'applicazione (es. driver JDBC, librerie JSON).

13.3 - Servlet

Creazione di una Servlet

Il processo di creazione e deploy di una servlet in Tomcat prevede:

1. **Scrittura del codice Java:** Il codice della servlet può essere scritto in qualsiasi editor o IDE.
2. **Compilazione (Passo 1):**
   - Il codice sorgente Java (`.java`) deve essere compilato in ByteCode (`.class`) usando il compilatore `javac`.
   - **Librerie necessarie:** Durante la compilazione, è necessario includere nel classpath la libreria `servlet-api.jar`. Questa si trova solitamente in `C:\Program Files\Apache Software Foundation\Tomcat X.Y\lib\`.
   - Il file `.class` generato deve essere copiato nella cartella `WEB-INF/classes` dell'applicazione web.
3. **Configurazione (Passo 2):**
   - La servlet deve essere dichiarata e mappata a un URL nel file `web.xml` (che si trova in `WEB-INF`).
   - **Dichiarazione della Servlet:**

     ```
     <servlet>
         <servlet-name>NomeLogicoServlet</servlet-name>
         <servlet-class>NomeCompletoClasseServlet</servlet-class>
     </servlet>
     ```

     Esempio:

     ```
     <servlet>
         <servlet-name>Hello</servlet-name>
         <servlet-class>Hello</servlet-class> <!-- Se la classe è nel package di default -->
     </servlet>
     ```

- **Mapping dell'URL:**

```
<servlet-mapping>
    <servlet-name>NomeLogicoServlet</servlet-name> <!-- Deve corrispondere al servlet-n
    <url-pattern>/percorsoURL</url-pattern>
</servlet-mapping>
```

Esempio:

```
<servlet-mapping>
    <servlet-name>Hello</servlet-name>
    <url-pattern>/Hello</url-pattern>
</servlet-mapping>
```

Attenzione: L'URL specificato in `<url-pattern>` è relativo al contesto dell'applicazione. Se l'applicazione è `my_examples`, l'URL completo per accedere alla servlet dell'esempio sarà `/my_examples/Hello`.

Che Cosa è una Servlet

- Una servlet è una classe Java che viene invocata dal web server (Tomcat) per gestire le richieste HTTP e generare risposte.
- **Package necessari:**
  - `javax.servlet`
  - `javax.servlet.http`
- Una servlet tipicamente estende la classe `HttpServlet`.

Struttura di base di una Servlet

- La servlet estende la classe `javax.servlet.http.HttpServlet`.
- Si fa l'override (sovrascrittura) dei metodi per gestire i tipi di richieste HTTP desiderati. I più comuni sono:
  - `doGet(HttpServletRequest request, HttpServletResponse response)`: Gestisce le richieste HTTP GET.
  - `doPost(HttpServletRequest request, HttpServletResponse response)`: Gestisce le richieste HTTP POST.
- Questi metodi vengono invocati automaticamente dal container servlet (Tomcat) in base al metodo HTTP della richiesta in arrivo.

Esempio (scheletro e import)

```
// Import necessari
import java.io.IOException;
import java.io.PrintWriter;
// import java.util.ResourceBundle; // Non usato nell'esempio specifico, ma comune

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

    // Eventualmente @Override per doPost, ecc.
}
```

La Richiesta HTTP (Oggetto `request`)

- Il parametro `request` (di tipo `HttpServletRequest`) rappresenta la richiesta HTTP in arrivo.
- Consente di accedere a:
  - Parametri della query string (per richieste GET).
  - Contenuto del corpo della richiesta (per richieste POST).
  - Header HTTP, informazioni sul client, ecc.

La Risposta HTTP (Oggetto `response`)

- Il parametro `response` (di tipo `HttpServletResponse`) permette di costruire la risposta HTTP da inviare al client.
- **Invio del contenuto:** Si ottiene un oggetto `PrintWriter` per scrivere il corpo della risposta:

```
PrintWriter out = response.getWriter();
```

- **Impostare il MIME Type (Content Type):** È fondamentale specificare il tipo di contenuto della risposta, affinché il browser possa interpretarla correttamente.

```
response.setContentType("text/html"); // Per HTML
// response.setContentType("application/json"); // Per JSON
```

Esempio (generazione di una risposta HTML semplice)

```
// All'interno di doGet o doPost
// Supponiamo che 'title' sia una variabile String definita altrove
String title = "Mia Pagina Dinamica";

response.setContentType("text/html"); // Imposta il MIME type
PrintWriter out = response.getWriter(); // Ottiene lo stream di output

out.println("<!DOCTYPE html><html>");
out.println("<head><title>" + title + "</title></head>");
out.println("<body bgcolor=\"white\">");
out.println("<h1>" + title + "</h1>");
// Altro contenuto dinamico qui...
out.println("</body>");
out.println("</html>");
```

Metodi dell'Oggetto `request` (HttpServletRequest)

Alcuni metodi utili dell'oggetto `request`:

- `String request.getQueryString()`: Restituisce la parte della query string dell'URL.
- `StringBuffer request.getRequestURL()`: Restituisce l'URL completo della richiesta.
- `String request.getServletPath()`: Estrae il path della servlet dall'URL.

Metodi ereditati da `ServletRequest` (interfaccia padre):

- `String request.getParameter(String name)`: Restituisce il valore del parametro della richiesta con il nome specificato. Restituisce `null` se il parametro non esiste. Se un parametro ha più valori, restituisce solo il primo.
- `java.util.Map<String, String[]> request.getParameterMap()`: Restituisce una mappa di tutti i parametri della richiesta. Le chiavi sono i nomi dei parametri (String), e i valori sono array di String (`String[]`), poiché un parametro può avere più valori (es. checkbox multiple con lo stesso nome).

Esempio: Servlet "Params" (Gestione Parametri GET)

Obiettivo: Reagire a una richiesta GET, acquisire tutti i parametri e generare una pagina HTML che li elenca con i loro valori.

```
// All'interno del metodo doGet di una servlet

response.setContentType("text/html");
PrintWriter out = response.getWriter();

java.util.Map<String, String[]> params = request.getParameterMap();
Object[] keys = params.keySet().toArray(); // Ottiene un array delle chiavi (nomi dei parametri)

out.println("<!DOCTYPE html><html><body>");

if (keys.length == 0) {
    out.println("<p>No params</p>");
} else {
    out.println("<h1>Parametri Ricevuti:</h1>");
    for (int i = 0; i < keys.length; i++) {
        String paramName = (String) keys[i]; // Converte la chiave Object in String
        String[] paramValues = params.get(paramName);
        // Prende solo il primo valore per semplicità, gestendo il caso di valore non presente
        String paramValue = (paramValues != null && paramValues.length > 0) ? paramValues[0] : "

        out.println("<p><b>" + paramName + ":</b> " + paramValue + "</p>");
    }
}
out.println("</body></html>");
```

Spiegazione del codice esempio:

1. Si ottiene la mappa di tutti i parametri (`getParameterMap()`).
2. Si estraggono le chiavi (nomi dei parametri) da questa mappa e le si convertono in un array.
3. Si itera sull'array delle chiavi.
4. Per ogni chiave (nome parametro), si recupera l'array dei suoi valori.
5. Per semplicità, si visualizza solo il primo valore associato a ciascun parametro.
6. Si genera l'HTML per visualizzare nome e valore di ogni parametro.

Metodo POST

- Il metodo `doPost` di una servlet reagisce alle richieste HTTP fatte con il metodo POST.

- **Parametri da Form HTML:** Se la servlet è invocata da un form HTML con `method="POST"`, i parametri del form sono contenuti nel corpo della richiesta. L'oggetto `request` gestisce questi parametri nello stesso modo del metodo GET (es. `request.getParameter()`, `request.getParameterMap()`).
- **Corpo della Richiesta (es. AJAX, Web Service):** Se la servlet riceve dati nel corpo della richiesta che non sono parametri di form standard (es. un documento JSON o XML inviato tramite AJAX), è necessario leggere direttamente il corpo della richiesta.
  - L'oggetto `request` fornisce i metodi:
    - `java.io.InputStream getInputStream()`: Per leggere dati binari.
    - `java.io.BufferedReader getReader()`: Per leggere dati testuali.
  - Esempio per leggere il corpo della richiesta come stringa:

```
StringBuilder buffer = new StringBuilder();
BufferedReader reader = request.getReader();
String line;
while ((line = reader.readLine()) != null) {
    buffer.append(line);
    buffer.append(System.lineSeparator()); // Mantiene i ritorni a capo, se importanti
}
String data = buffer.toString();
// Ora 'data' contiene il corpo della richiesta come stringa (es. un JSON)
```

Multi-threading nelle Servlet

- Ogni invocazione dei metodi `doGet` e `doPost` (e altri metodi di servizio) avviene tipicamente in un **thread dedicato** gestito dal container servlet (Tomcat).
- Questo permette al server di processare richieste multiple in parallelo:
  - Eseguendo la stessa servlet più volte contemporaneamente per richieste diverse.
  - Eseguendo servlet diverse in parallelo.
- **Importante:** Poiché le servlet sono multi-thread, è necessario prestare attenzione alla concorrenza se si utilizzano variabili di istanza (membri della classe servlet) che possono essere modificate. Generalmente, le servlet dovrebbero essere stateless o gestire lo stato in modo thread-safe.

13.4 - Accesso ai DB Relazionali con JDBC

JDBC (Java DataBase Connectivity) è un'API Java che definisce come un client può accedere a un database.

Protocollo JDBC

- **JDBC** sta per:
  - **J**ava
  - **D**ata**B**ase
  - **C**onnectivity
- Le interfacce del protocollo JDBC sono standardizzate nel package `java.sql`.

Driver JDBC per PostgreSQL

Per connettersi a un database specifico (es. PostgreSQL), è necessario un driver JDBC specifico per quel database.

- Il driver per PostgreSQL può essere scaricato da: https://jdbc.postgresql.org/download/
- Il file `.jar` del driver deve essere incluso nel classpath dell'applicazione (tipicamente nella cartella `WEB-INF/lib` di un'applicazione web Tomcat).

Processo Generale di Accesso al DB

Il processo generale è:

1. **Caricare il Driver JDBC.**
2. **Attivazione della connessione** con il database.
3. **Creazione ed esecuzione delle query** SQL.
4. **Elaborazione dei risultati.**
5. **Chiusura della connessione** e delle altre risorse (Statement, ResultSet).

JDBC funziona in modo simile per tutti i DBMS relazionali; l'esempio si concentrerà su PostgreSQL.

1. Caricare il Driver

Esistono due modi principali:

- **Versione "originale" (esplicita):**

```
Class.forName("org.postgresql.Driver"); // Carica esplicitamente la classe del driver
// String url = "...";
// Connection conn = DriverManager.getConnection(url);
```

Questo metodo richiede di gestire la `ClassNotFoundException`.

- **Versione "moderna" (implicita, a partire da JDBC 4.0):**
  Il `DriverManager` dovrebbe essere in grado di caricare automaticamente il driver appropriato se il JAR del driver è nel classpath, basandosi sulla stringa di connessione (URL).

```
// String url = "jdbc:postgresql://...";
// Connection conn = DriverManager.getConnection(url);
```

Se il caricamento automatico non funziona (come indicato nella slide: "che a me non ha funzionato"), si deve ricorrere al metodo esplicito con `Class.forName()`.

2. Apertura della Connessione

Indipendentemente da come il driver è stato caricato, la connessione si apre usando `DriverManager.getConnection()`:

```
String url = "jdbc:postgresql://localhost:5432/MyDB?user=MyUser&password=MyPwd";
// Sostituire localhost, 5432, MyDB, MyUser, MyPwd con i valori corretti.
Connection conn = DriverManager.getConnection(url);
```

- L'oggetto `Connection` (un'interfaccia del package `java.sql`) rappresenta la sessione con il database.

3. Creare uno Statement

L'oggetto `Connection` fornisce un metodo per creare un oggetto `Statement`, che verrà usato per eseguire le query SQL.

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

4. Eseguire la Query

Il codice SQL viene fornito come stringa al metodo `executeQuery()` (per query SELECT) o `executeUpdate()` (per INSERT, UPDATE, DELETE) dell'oggetto `Statement`.

```
String q = "SELECT \"Name\", \"Age\" FROM \"Names\"";
// Nota: In PostgreSQL, se i nomi di tabelle/campi sono case-sensitive o contengono caratteri sp
// devono essere racchiusi tra doppi apici (\"). In Java String, il doppio apice si escapa con \

ResultSet rs = stmt.executeQuery(q); // Per query SELECT
// int rowsAffected = stmt.executeUpdate(updateSql); // Per INSERT, UPDATE, DELETE
```

L'oggetto `ResultSet` contiene i risultati di una query SELECT.

5. Scansione del Result Set

Si itera sulle righe del `ResultSet` usando un ciclo `while` e il metodo `rs.next()`.
`rs.next()` sposta il cursore alla riga successiva e restituisce `true` se esiste una riga, `false` altrimenti.

```
while (rs.next()) {
    // Acquisire i dati dalla riga corrente
    String name = rs.getString("Name"); // Ottiene il valore della colonna "Name" come String
    int age = rs.getInt("Age");         // Ottiene il valore della colonna "Age" come int

    // Elaborare i dati (es. stamparli)
    // out.println("<p>Name: " + name + ", Age: " + age + "</p>");
}
```

6. Acquisizione dei Campi

I valori dei campi di una riga del `ResultSet` si accedono tramite il nome della colonna (o l'indice, sconsigliato).
Occorre usare il metodo `getXXX()` appropriato per il tipo di dato della colonna:

- `rs.getString("NomeColonna")`
- `rs.getInt("NomeColonna")`
- `rs.getDouble("NomeColonna")`
- `rs.getDate("NomeColonna")`, ecc.

7. Chiusura delle Risorse (Fondamentale!)

È cruciale chiudere `ResultSet`, `Statement` e `Connection` (in ordine inverso rispetto all'apertura) per rilasciare le risorse del database. Solitamente si fa in un blocco `finally`.

```
// In un blocco finally
if (rs != null) try { rs.close(); } catch (SQLException e) { /* log */ }
if (stmt != null) try { stmt.close(); } catch (SQLException e) { /* log */ }
if (conn != null) try { conn.close(); } catch (SQLException e) { /* log */ }
```

Eccezioni

Molti metodi JDBC possono lanciare una `SQLException`. È obbligatorio catturare e gestire queste eccezioni.

Transazioni

La gestione delle transazioni è simile a quella vista per altri driver (es. Python).

- **Auto-commit:** Di default, una nuova connessione è in modalità auto-commit (ogni singola istruzione SQL è una transazione e viene commessa immediatamente).
- **Disattivare auto-commit:**

  ```
  conn.setAutoCommit(false);
  ```

- **Commit esplicito:**

  ```
  conn.commit(); // Rende permanenti le modifiche dall'ultimo commit/rollback
  ```

- **Rollback esplicito:**

  ```
  conn.rollback(); // Annulla le modifiche dall'ultimo commit/rollback
  ```

Dopo aver disattivato l'auto-commit, è necessario chiamare `conn.commit()` per confermare un gruppo di operazioni o `conn.rollback()` per annullarle in caso di errore.

13.5 - GSON (Gestione JSON in Java)

GSON è una libreria Java open-source sviluppata da Google per serializzare oggetti Java in JSON e deserializzare JSON in oggetti Java.

- **Scopo:** Utile quando si realizzano servlet per comunicazione AJAX o come web service, dove JSON è un formato di scambio dati comune.

- **Download:** La slide indica un link (https://jar-download.com/artifacts/com.google.code.gson/gson/2.8.2/source-code), ma è tipicamente gestita tramite sistemi di build come Maven o Gradle, o scaricando il `.jar` dal repository ufficiale di Google/Maven Central. Il JAR va incluso nel classpath (es. `WEB-INF/lib`).

Come Funziona

- **Serializzazione:**
  - Dato un oggetto Java (anche complesso, con array e altri oggetti annidati), GSON lo converte in una rappresentazione stringa JSON.
  - **Limitazione:** Non devono esserci riferimenti circolari negli oggetti da serializzare, altrimenti GSON andrà in loop (a meno di configurazioni avanzate).
- **Deserializzazione:**
  - Dato un documento JSON (stringa) e una classe Java target (che può usare array e altre classi al suo interno), GSON tenta di ricostruire un'istanza della classe Java popolandone i campi con i valori del JSON.
  - La struttura dati JSON deve corrispondere (o essere mappabile) alla struttura della classe Java.

Basi dell'Utilizzo di GSON

1. **Importare il package:**

   ```
   import com.google.gson.*;
   ```

2. **Creare un oggetto `Gson`:**
   Si crea tramite `GsonBuilder` per poter configurare opzioni come il "pretty printing" (formattazione JSON leggibile).

   ```
   Gson gson = new GsonBuilder().setPrettyPrinting().create();
   // Senza pretty printing: Gson gson = new Gson();
   ```

Serializzazione (Oggetto Java -> JSON String)

Il metodo `toJson()` dell'oggetto `Gson` serializza un oggetto Java.

```
// MyObject obj = new MyObject(...);
String jsonString = gson.toJson(obj);
// Ora jsonString contiene la rappresentazione JSON dell'oggetto obj
```

Deserializzazione (JSON String -> Oggetto Java)

1. **Definire la classe Java target:** La classe deve avere campi i cui nomi corrispondono alle chiavi nel JSON (o usare annotazioni `@SerializedName` per mappare nomi diversi).

   ```
   public class Nominativo {
       String Name; // Corrisponde a "Name" nel JSON
       int Age;     // Corrisponde a "Age" nel JSON
       // Costruttori, getter, setter opzionali ma utili
   }
   ```

2. **Effettuare la deserializzazione:**
   Il metodo `fromJson()` dell'oggetto `Gson` deserializza una stringa JSON in un oggetto della classe specificata.

   ```
   String jsonData = "{\"Name\":\"Mario\", \"Age\":30}"; // Stringa JSON
   Nominativo nominativoObj = gson.fromJson(jsonData, Nominativo.class);
   // Ora nominativoObj è un'istanza di Nominativo con Name="Mario" e Age=30
   ```

   - Il primo argomento è la stringa JSON o un `Reader`.
   - Il secondo argomento è `NomeClasse.class`, che fornisce a GSON il tipo dell'oggetto da creare.

In caso di Fallimento della Deserializzazione

- GSON potrebbe fallire la deserializzazione se la struttura JSON non corrisponde alla classe target o se ci sono ambiguità.
- Per casi complessi, `GsonBuilder` può essere personalizzato per implementare una "custom JSON deserialization" (Type Adapters), ma questo è un argomento avanzato non trattato in dettaglio.

13.6 - Esempio: Server per AJAX con Servlet, JDBC e GSON

Questo esempio mostra una servlet che riceve una richiesta AJAX (POST) con dati JSON, interroga un database (PostgreSQL) in base a questi dati, e restituisce un risultato in formato JSON.

Scenario:

1. Una pagina web invia una richiesta AJAX (POST) alla servlet.
2. Il corpo della richiesta AJAX è un documento JSON contenente criteri di ricerca (es. nome, età).
3. La servlet:
    - Deserializza il JSON della richiesta.
    - Costruisce ed esegue una query SQL sul database.
    - Raccoglie i risultati.
    - Serializza i risultati in un nuovo documento JSON.
    - Invia il JSON di risposta al client.

1. Classi Java per i Documenti JSON

- **`Nominativo.java` (per il JSON in entrata e per gli elementi nella lista di risultati):**

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

    public Nominativo() { } // Costruttore di default

    public Nominativo(String n, int a) { // Costruttore per creare istanze
        this.Name = n;
        this.Age = a;
    }
}
```

- **`Result.java` (per il JSON in uscita):**

```
import java.util.ArrayList;

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

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

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

2. Metodo Utile per Costruire la Query SQL (`addToCond`)

Un metodo helper per aggiungere condizioni alla clausola WHERE.

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

3. Implementazione della Servlet (Metodo `doPost`)

```
// ... (import necessari: IOException, PrintWriter, ServletException, HttpServlet, ecc.)
// ... (import per JDBC: Connection, DriverManager, ResultSet, SQLException, Statement)
// ... (import per GSON: Gson, GsonBuilder)
// ... (import per le classi Nominativo e Result)
// ... (import per BufferedReader)

public class AjaxServletExample extends HttpServlet { // Nome di esempio

    // Metodo helper addToCond definito sopra o nella stessa classe

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // a. Impostare content type della risposta e PrintWriter
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        PrintWriter out = response.getWriter();

        // b. Estrarre il documento JSON dalla richiesta
        StringBuilder buffer = new StringBuilder();
        BufferedReader reader = request.getReader();
        String line;
        while ((line = reader.readLine()) != null) {
            buffer.append(line);
        }
        String jsonDataFromRequest = buffer.toString();

        // c. Deserializzare il JSON in entrata
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        Nominativo qryParams = gson.fromJson(jsonDataFromRequest, Nominativo.class);

        String WName = (qryParams != null && qryParams.Name != null) ? qryParams.Name : null;
        // Se Age non è fornito nel JSON, GSON lo imposta a 0 per int.
        // Potremmo voler trattare 0 come "nessun filtro" o come valore specifico.
        // Qui assumiamo che se qryParams.Age è 0, non si filtra per età a meno che non sia espl
        int WAge = (qryParams != null) ? qryParams.Age : -1; // Usiamo -1 per "non specificato"

        Result queryResult = new Result(); // Oggetto per i risultati

        // d. Connessione al DB e preparazione query
```

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
String dbUrl = "jdbc:postgresql://localhost:5432/MyDB?user=MyUser&password=MyPwd"; // Ac

try {
    Class.forName("org.postgresql.Driver");
    conn = DriverManager.getConnection(dbUrl);
    stmt = conn.createStatement();

    // e. Preparare la query SQL dinamicamente
    String sqlConditions = "";
    if (WName != null && !WName.isEmpty()) {
        // ATTENZIONE: vulnerabile a SQL Injection. Usare PreparedStatement in produzior
        sqlConditions = addToCond(sqlConditions, "\"Name\" = '" + WName.replace("'", "'
    }
    if (WAge >= 0) { // Filtra per età se >= 0 (0 potrebbe essere un'età valida)
        sqlConditions = addToCond(sqlConditions, "\"Age\" = " + WAge);
    }

    String sqlQuery = "SELECT \"Name\", \"Age\" FROM \"Names\"" + sqlConditions;
    if (sqlConditions.isEmpty()) {
        sqlQuery = "SELECT \"Name\", \"Age\" FROM \"Names\""; // Nessun filtro, selezior
    }
     sqlQuery += ";"; // Aggiungi il terminatore


    // f. Eseguire la query e popolare l'oggetto Result
    rs = stmt.executeQuery(sqlQuery);
    while (rs.next()) {
        queryResult.append(rs.getString("Name"), rs.getInt("Age"));
    }

} catch (SQLException e) {
    // Gestione errore SQL (es. inviare JSON di errore)
    // queryResult.elementi = -1; // o un campo 'error' in Result
    out.print("{\"error\":\"SQL Error: " + e.getMessage().replace("\"", "\\\"") + "\"}"
    // Loggare l'eccezione e.printStackTrace();
} catch (ClassNotFoundException e) {
    // Gestione errore driver (es. inviare JSON di errore)
    out.print("{\"error\":\"Database Driver Error: " + e.getMessage().replace("\"", "\\'
    // Loggare l'eccezione e.printStackTrace();
} finally {
```

```
            // g. Chiudere le connessioni DB
            try { if (rs != null) rs.close(); } catch (SQLException e) { /* log */ }
            try { if (stmt != null) stmt.close(); } catch (SQLException e) { /* log */ }
            try { if (conn != null) conn.close(); } catch (SQLException e) { /* log */ }
        }

        // h. Serializzare l'oggetto Result in JSON e inviarlo
        // Solo se non è già stato scritto un errore
        if (!response.isCommitted()) {
             out.print(gson.toJson(queryResult));
        }
        out.flush();
    }
}
```

Esempio di JSON

- **Documento JSON Ricevuto (esempio):**

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

- **Documento JSON Inviato (esempio di risposta):**

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

14.1 - Parametri di Configurazione (Contesto)

Parametrizzare le Servlet

- È una cattiva pratica di programmazione inserire valori di configurazione (come stringhe di connessione al database, percorsi di file, ecc.) direttamente nel codice sorgente.
- Negli esempi precedenti (Lezione 13), la stringa di connessione al DB era "hardcoded".
- Un buon design prevede che questi valori siano esterni al codice, per facilitare la configurazione e la manutenzione senza dover ricompilare l'applicazione.
- È necessario, quindi, parametrizzare l'applicazione.

Contesto dell'Applicazione

- Nel file `web.xml` (il descrittore di deployment dell'applicazione web), si può definire il "Contesto dell'Applicazione".
- Il contesto è un insieme di parametri di configurazione globali per l'intera applicazione web.
- Le servlet all'interno dell'applicazione possono accedere a questi parametri di contesto.

Definire un Parametro di Contesto

Per definire un parametro di contesto nel file `web.xml`, si usa l'elemento `<context-param>`:

```
<context-param>
    <param-name>NomeDelParametro</param-name>
    <param-value>ValoreDelParametro</param-value>
</context-param>
```

Esempio: Definire la stringa di connessione al DB come parametro di contesto:

```
<context-param>
    <param-name>DBConnString</param-name>
    <param-value>jdbc:postgresql://localhost:5432/MyDB?user=MyUser&amp;password=MyPwd</param-va
</context-param>
```

Attenzione al carattere `&` in XML:

- Il carattere `&` (e commerciale) è un carattere riservato in XML e viene usato per definire le "entità" (es. `&lt;` per `<`, `&gt;` per `>`, `&amp;` per `&`).

- Nelle stringhe di connessione JDBC, il `&` è usato per separare i parametri (es. `user=MyUser&password=MyPwd`).
- Quando si scrive una stringa di connessione contenente `&` nel `param-value` di `web.xml`, il `&` deve essere sostituito con la sua entità XML: `&`.
  Esempio: `?user=MyUser&password=MyPwd`

Acquisizione del Contesto e dei Parametri nella Servlet

1. **Acquisire il Contesto:**
   All'interno di una servlet, si può ottenere l'oggetto `ServletContext` (che rappresenta il contesto dell'applicazione) usando il metodo `getServletContext()` (ereditato da `GenericServlet`).

   ```
   ServletContext context = getServletContext();
   ```

2. **Acquisire un Parametro di Inizializzazione dal Contesto:**
   Una volta ottenuto l'oggetto `ServletContext`, si può recuperare il valore di un parametro di contesto usando il metodo `getInitParameter(String name)`.

   ```
   String dbConnString = context.getInitParameter("DBConnString");
   // Ora dbConnString contiene il valore definito nel web.xml
   ```

14.2 - Gestire le Sessioni

HTTP è Stateless

- Il protocollo HTTP è intrinsecamente "stateless" (senza stato). Ogni richiesta HTTP da un client a un server è trattata come una transazione indipendente, senza alcuna conoscenza delle richieste precedenti.
- Questo significa che, di default, il server non ricorda nulla dell'utente tra una richiesta e l'altra.

Esempi di Inadeguatezza dello Stateless

L'approccio stateless non è adeguato per molte funzionalità web comuni:

- **Login degli utenti:** Mantenere l'utente autenticato attraverso più pagine.
- **Procedure multi-passo:** Come un wizard di registrazione o un processo di checkout.
- **Carrello della spesa (shopping cart):** Ricordare gli articoli che un utente ha aggiunto al carrello mentre naviga nel sito.

Qual è il Problema dello Stateless?

- Manca la certezza che le chiamate HTTP successive provengano dallo stesso utente o che riguardino la stessa interazione o processo logico.

Soluzione: le Sessioni

- Una **Sessione** è un meccanismo per mantenere lo "stato" dell'interazione di un utente con un'applicazione web attraverso più richieste HTTP.
- Può essere vista come un insieme di variabili (o attributi) associate a un utente specifico.
- Queste variabili "sopravvivono" tra una chiamata HTTP e l'altra, permettendo agli script server-side (come le servlet) di recuperarle e gestire il processo in modo stateful.

Idea di base: i Cookie

- I **cookie** sono piccole porzioni di dati che un server può inviare a un browser.
- Il browser memorizza questi cookie e li ritrasmette automaticamente al server con ogni successiva richiesta HTTP verso lo stesso dominio.
- I cookie sono uno dei meccanismi principali usati per implementare le sessioni.

Il SessionId come Cookie

- La gestione delle sessioni avviene tipicamente sul server.
- Invece di inviare l'intero stato della sessione (tutte le variabili) al client, il server:
  i. Crea una sessione unica per l'utente sul server.
  ii. Genera un identificatore univoco per questa sessione (il **SessionId**).
  iii. Invia solo il SessionId al client, solitamente tramite un cookie (es. `JSESSIONID` in Java).
- Il browser del client memorizza questo cookie SessionId e lo invia con ogni richiesta successiva.
- Il server usa il SessionId ricevuto per recuperare i dati della sessione specifica dell'utente memorizzati sul server.
- Questo approccio permette al server di controllare la durata della sessione e di memorizzare dati sensibili sul server anziché inviarli al client.

Sessioni nelle Servlet

- Le sessioni in Java Servlet sono gestite tramite l'oggetto `javax.servlet.http.HttpSession` .
- **Ottenere/Creare una Sessione:**
  L'oggetto sessione è associato alla richiesta HTTP. Si ottiene tramite il metodo `getSession()` dell'oggetto `HttpServletRequest` :

```
HttpSession session = request.getSession(true);
```

- Il parametro booleano indica cosa fare se non esiste già una sessione associata alla richiesta:
  - `true` : Se non esiste una sessione, ne crea una nuova. Se esiste, restituisce quella esistente. Questo è l'uso più comune.
  - `false` : Se non esiste una sessione, restituisce `null` . Se esiste, restituisce quella esistente. Utile per verificare se una sessione è già attiva senza crearne una nuova.

```
HttpSession session = request.getSession(false);
if (session == null) {
    // Nessuna sessione esistente, l'utente non è "loggato" o la sessione è scaduta
} else {
    // Sessione esistente
}
```

- **Scoprire se la Sessione è Nuova:**
  È possibile verificare se una sessione è stata appena creata (cioè, se il client non ha inviato un SessionId valido o se `request.getSession(true)` l'ha creata in questa richiesta) usando il metodo `isNew()` dell'oggetto `HttpSession` .

```
if (session.isNew()) {
    // La sessione è stata appena creata
} else {
    // La sessione esisteva già
}
```

  Restituisce un `boolean` .
- **Attributi di Sessione:**
  - Un attributo di sessione può essere un qualsiasi oggetto Java (String, Integer, liste, oggetti custom, ecc.).
  - **Recuperare un Attributo:** Per ottenere il valore di un attributo dalla sessione, si usa il metodo `getAttribute(String name)` :

```
Object attributeValue = session.getAttribute("nomeAttributo");
// È necessario fare un cast al tipo corretto
// Esempio: String username = (String) session.getAttribute("username");
```

    Se l'attributo non esiste, `getAttribute()` restituisce `null` .
  - **Esempio: Shopping Cart:**

```
// Supponiamo esista una classe ShoppingCart
ShoppingCart items = (ShoppingCart) session.getAttribute("items");

if (items != null) {
    // L'utente ha già un carrello, usalo
    // doSomethingWith(items);
} else {
    // L'utente non ha un carrello (es. prima visita o nuova sessione)
    items = new ShoppingCart(/*...parametri costruttore...*/);
    // doSomethingElseWith(items); o inizializza il carrello
    // Poi va salvato nella sessione (vedi sotto)
}
```

Spiegazione:

a. Si tenta di recuperare un oggetto `ShoppingCart` dalla sessione con la chiave "items".
b. Se l'oggetto esiste (non è `null`), significa che l'utente ha già un carrello.
c. Se non esiste, se ne crea uno nuovo.
d. L'oggetto (nuovo o esistente e modificato) deve poi essere (ri)messo nella sessione.

- **Impostare (o Sostituire) un Attributo:** Per memorizzare o aggiornare un attributo nella sessione, si usa il metodo `setAttribute(String name, Object value)`:

```
session.setAttribute("items", items); // Salva/aggiorna l'oggetto 'items' nella session
// session.setAttribute("username", "MarioRossi");
```

- **La Sessione nella Risposta:**
  Non è necessario inviare manualmente la sessione. Il container servlet (Tomcat) gestisce automaticamente l'invio del cookie SessionId (es. `JSESSIONID`) al client quando una sessione viene creata o utilizzata. Il browser poi lo reinvierà.
- **Chiudere la Sessione (Invalidarla):**
  Per terminare una sessione (es. al logout dell'utente), occorre "invalidarla". Questo rimuove tutti gli attributi dalla sessione e la rende non più utilizzabile.

```
session.invalidate();
```

Da quel momento, le successive invocazioni della servlet (anche se il client invia il vecchio SessionId) non troveranno un oggetto sessione valido associato (o `request.getSession(false)` restituirà `null`, e `request.getSession(true)` ne creerà una nuova).

14.3 - Pattern MVC (Model-View-Controller)

Model-View-Controller

- È un pattern architetturale molto diffuso, inventato prima del World Wide Web, ma che ha trovato un'applicazione naturale nello sviluppo di applicazioni web.
- Separa le responsabilità dell'applicazione in tre componenti interconnessi:
  - **Model (Modello):**
    - Rappresenta i dati dell'applicazione e la logica di business.
    - È l'insieme dei dati su cui si deve operare (es. la base dati, oggetti Java che rappresentano entità).
    - Gestisce l'accesso ai dati e la loro manipolazione.
  - **View (Vista):**
    - È responsabile della presentazione dei dati all'utente.
    - È ciò che l'utente vede (l'interfaccia utente, es. pagine HTML).
    - Dovrebbe essere "stupida" e limitarsi a visualizzare i dati forniti dal Controller/Model.
  - **Controller (Controllore):**
    - Riceve l'input dell'utente (es. richieste HTTP).
    - Interagisce con il Model per recuperare o modificare i dati.
    - Seleziona la View appropriata per presentare i risultati all'utente.
    - È il gestore delle azioni e delle trasformazioni da operare sui dati.

Obiettivo Fondamentale dell'MVC

- **Disaccoppiare l'elaborazione dalla visualizzazione.**
- Rendere la View (visualizzazione) il più indipendente possibile dall'attività di elaborazione (logica di business e accesso ai dati).
- Questo porta a codice più manutenibile, testabile e flessibile (es. si può cambiare la View senza toccare il Model, o viceversa).
- Esistono molte implementazioni e variazioni del pattern MVC.
- Nelle applicazioni web, un obiettivo chiave è disaccoppiare il codice procedurale (Java nelle servlet) dalla generazione del codice HTML.

Limiti dei Server-Side Script stile PHP/JSP Semplice

- Nelle pagine JSP semplici (o PHP classiche), il codice procedurale (Java o PHP) è spesso "annegato" direttamente nella pagina HTML.
- Questo porta a un **stretto accoppiamento** tra la logica di elaborazione e la presentazione.

- È meglio rispetto a servlet che generano HTML direttamente tramite `out.println()`, ma è ancora lontano da un vero disaccoppiamento MVC.

ThymeLeaf come Soluzione MVC per Servlet

- **ThymeLeaf** è un motore di template Java che offre un modo interessante ed efficace per introdurre il pattern MVC nelle applicazioni basate su servlet.
- **Come funziona (in ottica MVC):**
  - La **Servlet** agisce come **Controller**:
    - Riceve la richiesta.
    - Interagisce con il **Model** (es. accede al database, prepara oggetti Java con i dati).
    - Prepara un "contesto" ThymeLeaf (una mappa di dati) che contiene le informazioni da visualizzare.
    - Inoltra questo contesto e il nome di un template ThymeLeaf al motore di ThymeLeaf.
  - Il **Template ThymeLeaf** agisce come **View**:
    - È un file HTML (o XML) che contiene markup standard e speciali attributi ThymeLeaf (`th:*`).
    - Il motore ThymeLeaf processa il template, sostituendo gli attributi `th:*` con i dati presi dal contesto fornito dalla servlet.
    - Il risultato è una pagina HTML finale inviata al client.
- Consente di tenere la logica di presentazione (HTML e direttive ThymeLeaf) separata dalla logica di business e di controllo (codice Java nella servlet).
- Il controllo della generazione della pagina finale rimane in carico alla servlet (Controller), che prepara un model (contesto ThymeLeaf) dedicato per la View.

14.4 - ThymeLeaf

ThymeLeaf è un moderno motore di template Java server-side per ambienti web e standalone.

Sito Ufficiale

- **Home Page:** https://www.thymeleaf.org/
- **Distribuzione su GitHub (releases):** https://github.com/thymeleaf/thymeleaf/releases/

Librerie

Per utilizzare ThymeLeaf in un'applicazione web (con Tomcat), è necessario aggiungere i file `.jar` di ThymeLeaf alla cartella `WEB-INF/lib` dell'applicazione.
La slide indica di copiare:

- Tutti i file `.jar` contenuti nella cartella `lib` del file ZIP di distribuzione di ThymeLeaf.
- Alcuni file specifici dalla cartella `dist` (i nomi dei file suggeriscono integrazioni con Spring, ma per l'uso base con servlet il core di ThymeLeaf è il principale):
  - `thymeleaf-X.Y.Z.RELEASE.jar` (il core)
  - Le altre librerie menzionate (`thymeleaf-extras-springsecurity*` , `thymeleaf-spring*`) sono per l'integrazione con il framework Spring e potrebbero non essere strettamente necessarie per un uso base con servlet pure, a meno che non si utilizzino tali funzionalità.

TemplateEngine

- Il cuore di ThymeLeaf è il `TemplateEngine` .
- Esso fornisce un metodo `process` che:
  i. Prende in input un **template** (un file HTML/XML).
  ii. Prende in input delle **variabili di contesto** (dati da visualizzare).
  iii. Elabora il template, sostituendo le espressioni ThymeLeaf con i valori dal contesto.
  iv. Genera l'output finale (es. una stringa HTML).

Variabili di Contesto (ThymeLeaf `Context`)

- Un "contesto" ThymeLeaf (`org.thymeleaf.context.Context` o `org.thymeleaf.context.WebContext` per applicazioni web) è un insieme di variabili (oggetti Java) che servono come input per l'elaborazione del template.
- Si crea un nuovo oggetto `Context` e vi si aggiungono le variabili prima di passarlo al metodo `process` del `TemplateEngine` .

Creazione del Contesto:

```
import org.thymeleaf.context.Context;

// ...
Context context = new Context(); // Inizialmente vuoto
context.setVariable("name", "John"); // Aggiunge una variabile "name" con valore "John"
// context.setVariable("user", myUserObject); // Si possono aggiungere oggetti complessi
```

Che Cosa è un Template ThymeLeaf

- Un template ThymeLeaf è tipicamente un file HTML (ma può essere anche XML, testo, ecc.).
- Contiene markup standard e specifici **attributi ThymeLeaf** che iniziano con il prefisso `th:` (es. `th:text`, `th:if`, `th:each`).
- Questi attributi sono definiti in uno specifico namespace XML (`xmlns:th="http://www.thymeleaf.org"`), che di solito si dichiara nell'elemento `<html>` o nell'elemento specifico dove si usano gli attributi `th:`.

Esempio di Template (hello.html):

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"> <!-- Namespace ThymeLeaf -->
<head>
    <meta charset="UTF-8">
    <title>ThymeLeaf Examples</title>
</head>
<body>
    <!-- Il tag <p> stesso verrà mantenuto, ma il suo contenuto sarà sostituito -->
    <p th:text="${'Hello ' + name}">Contenuto placeholder che verrà sostituito</p>
</body>
</html>
```

- `th:text="${'Hello ' + name}"`: L'attributo `th:text` rimpiazzerà il corpo dell'elemento `<p>` con il risultato dell'espressione.
- `${'Hello ' + name}`: È un'espressione ThymeLeaf Standard. Accede alla variabile `name` dal contesto.

Esempio di Output generato (supponendo `name` ="John" nel contesto):

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>ThymeLeaf Examples</title>
</head>
<body>
    <p>Hello John</p>
</body>
</html>
```

Package da Importare (per la Servlet)

Per usare ThymeLeaf in una servlet, sono necessari diversi import:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context; // o WebContext
import org.thymeleaf.templateresolver.ClassLoaderTemplateResolver; // o ServletContextTemplateRe
import org.thymeleaf.templatemode.TemplateMode;
// import org.thymeleaf.TemplateSpec; // Non sempre necessario per uso base
// import org.thymeleaf.*; // Può essere usato come wildcard se si usano molte classi
// import org.thymeleaf.context.*; // Wildcard
// import org.thymeleaf.templateresolver.*; // Wildcard
```

Creare il TemplateEngine (nella Servlet)

Il processo di creazione del `TemplateEngine` richiede alcuni passaggi:

1. **Creare un'istanza di `TemplateEngine` :**

   ```
   TemplateEngine templateEngine = new TemplateEngine();
   ```

2. **Creare un `TemplateResolver` :**

   Il `TemplateResolver` ha il compito di trovare e caricare i file dei template.

   - `ClassLoaderTemplateResolver` : Carica template dal classpath (es. se i template sono in `WEB-INF/classes/templates/`).
   - `ServletContextTemplateResolver` : Più comune per applicazioni web, carica template relativi alla root dell'applicazione web (es. `WEB-INF/templates/`). (Non mostrato nella slide, ma spesso preferibile per `web.xml` configurate).

   La slide mostra `ClassLoaderTemplateResolver` :

   ```
   ClassLoaderTemplateResolver templateResolver =
       new ClassLoaderTemplateResolver(Thread.currentThread().getContextClassLoader());
   ```

3. **Configurare il `TemplateResolver` :**

   - `setTemplateMode()` : Specifica il tipo di template (es. `TemplateMode.HTML` , `TemplateMode.XML`).

     ```
     templateResolver.setTemplateMode(TemplateMode.HTML);
     ```

   - `setPrefix()` : Specifica il percorso della cartella dove si trovano i template, relativo alla radice di caricamento del resolver (es. se i template sono in `WEB-INF/classes/ThTemplates/` , e si usa `ClassLoaderTemplateResolver` che parte da `classes`).

```
templateResolver.setPrefix("/ThTemplates/"); // I template saranno cercati in /ThTempla
```

- `setSuffix()` : (Opzionale, ma comune) Specifica l'estensione dei file template (es. `.html`).

```
// templateResolver.setSuffix(".html"); // Se si imposta, nel process si usa solo "Hell
```

4. **Associare il `TemplateResolver` al `TemplateEngine` :**

```
templateEngine.setTemplateResolver(templateResolver);
```

Spiegazione:

- Il `TemplateResolver` recupera il file del template e ne gestisce il formato.
- Il `setPrefix` indica una cartella all'interno del classpath (se si usa `ClassLoaderTemplateResolver`). I file template dovrebbero essere copiati in `WEB-INF/classes/ThTemplates/` .

Elaborazione del Template (nella Servlet)

Una volta configurato il `TemplateEngine` e preparato il `Context` , si può elaborare il template:

```
// Supponiamo:
// TemplateEngine templateEngine; (già inizializzato)
// Context context; (già popolato con variabili)
// PrintWriter out = response.getWriter(); (per inviare l'output al client)

templateEngine.process("Hello.html", context, out);
// "Hello.html" è il nome del file template (relativo al prefix configurato)
// context contiene i dati
// out è dove viene scritto l'HTML generato
```

14.5 - Esempio Complesso: Visualizzare Nominativi con ThymeLeaf

Riprende l'esempio dei nominativi (dal database) della Lezione 13, ma invece di serializzarli in JSON, li visualizza in una pagina HTML usando ThymeLeaf.

- Si usa la stessa struttura dati (classi `Nominativo` e `Result`) usata per l'esempio JSON.
- Invece di serializzare l'oggetto `Result` in JSON, lo si mette nel contesto di ThymeLeaf.
- Si usa un template ThymeLeaf per elaborare e visualizzare questi dati.

Struttura Dati (Riepilogo)

- **`Nominativo.java`:**

```
public class Nominativo {
    public String Name; // Devono essere public per l'accesso diretto da ThymeLeaf,
    public int Age;     // o avere getter public (es. getName(), getAge())
                        // ThymeLeaf usa la reflection o JavaBean conventions.

    public Nominativo() { }
    public Nominativo(String n, int a) { this.Name = n; this.Age = a; /*...*/ }
}
```

- **`Result.java`:**

```
import java.util.ArrayList;
public class Result {
    public int elementi = 0;
    public ArrayList<Nominativo> list; // Anche qui, public o con getter getList()

    public Result() { list = new ArrayList<>(); /*...*/ }
    public void append(String Name, int Age) { /*...*/ }
}
```

Nota: Per l'accesso da ThymeLeaf tramite espressioni come `${res.elementi}` o `${item.Name}`, i campi devono essere `public` o devono esistere metodi getter conformi alla convenzione JavaBeans (es. `getElementi()`, `getName()`). La slide mostra i campi `public`.

Preparare il Contesto nella Servlet

1. **Connessione al DB e Query (simile alla Lezione 13):**
   Si esegue la query al database e si popola un oggetto `Result` (chiamato `r` nell'esempio).

```
// All'interno di doGet o doPost
Result r = new Result();
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
// ... (try-catch per Class.forName, DriverManager.getConnection, createStatement)
try {
    Class.forName("org.postgresql.Driver");
    String url = "jdbc:postgresql://localhost:5432/MyDB?user=MyUser&password=MyPwd"; // Usa
    conn = DriverManager.getConnection(url);
    stmt = conn.createStatement();

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

    while (rs.next()) {
        r.append(rs.getString("Name"), rs.getInt("Age"));
    }
} catch (SQLException | ClassNotFoundException e) {
    // Gestione eccezioni
    e.printStackTrace(response.getWriter()); // Semplificato, invia stack trace
} finally {
    // Chiusura risorse DB
}
```

2. **Inizializzazione del `TemplateEngine` (come visto prima):**

```
TemplateEngine templateEngine = new TemplateEngine();
ClassLoaderTemplateResolver templateResolver =
    new ClassLoaderTemplateResolver(Thread.currentThread().getContextClassLoader());
templateResolver.setTemplateMode(TemplateMode.HTML);
templateResolver.setPrefix("/ThTemplates/"); // Assicurati che esista WEB-INF/classes/ThTem
// templateResolver.setSuffix(".html"); // Opzionale
templateEngine.setTemplateResolver(templateResolver);
```

3. **Creare il Contesto ThymeLeaf e Aggiungere i Dati:**

```
Context context = new Context();
context.setVariable("res", r); // "res" è il nome con cui l'oggetto r sarà accessibile nel
```

4. **Processare il Template:**

```
// response.setContentType("text/html;charset=UTF-8"); // Importante
// PrintWriter out = response.getWriter();
templateEngine.process("Nominativi.html", context, out); // "Nominativi.html" è il nome del
```

Il Template ThymeLeaf (`Nominativi.html`)

Il template deve:

- Creare una tabella HTML se ci sono nominativi (`res.elementi > 0`).
- Scrivere un messaggio se non ci sono nominativi (`res.elementi == 0`).

Approccio di ThymeLeaf:

- Usa attributi XML aggiuntivi definiti nel namespace `th:` .
- Non definisce nuovi elementi HTML, ma arricchisce quelli esistenti con attributi.
- **`th:if`** : Condiziona la visualizzazione dell'elemento a cui è applicato. Se l'espressione è `false` , l'elemento e i suoi figli non vengono renderizzati.
- **`th:each`** : Itera su una collezione. L'elemento a cui è applicato viene ripetuto per ogni elemento della collezione.

Cuore del Template (`WEB-INF/classes/ThTemplates/Nominativi.html`):

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Elenco Nominativi</title>
    <meta charset="UTF-8">
</head>
<body>
    <h1>Nominativi</h1>

    <!-- Sezione da mostrare se non ci sono elementi -->
    <div th:if="${res.elementi == 0}">
        <p>Nessun nominativo trovato</p>
    </div>

    <!-- Tabella da mostrare se ci sono elementi -->
    <table border="1" th:if="${res.elementi > 0}">
        <thead>
            <tr>
                <th>Nome</th>
                <th>Età</th>
            </tr>
        </thead>
        <tbody>
            <!-- Itera sulla lista 'list' dell'oggetto 'res' -->
            <!-- Per ogni elemento della lista, crea una variabile 'item' -->
            <tr th:each="item : ${res.list}">
                <td th:text="${item.Name}">Nome Placeholder</td>
                <td th:text="${item.Age}">Età Placeholder</td>
            </tr>
        </tbody>
    </table>

</body>
</html>
```

Espressioni di ThymeLeaf:

- Il valore di un attributo `th:` è un'espressione che il `TemplateEngine` valuta.
- Le espressioni sono tipicamente racchiuse in `${...}` (Variable Expressions).
- Per `th:if` , l'espressione deve valutare a un booleano.
    - `th:if="${res.elementi == 0}"` : Accede al campo `elementi` dell'oggetto `res` (che è nel contesto) e lo confronta con 0.

Variabile dal Contesto:

- `res` nell'espressione `${res.elementi}` si riferisce alla variabile "res" che è stata messa nel `Context` dalla servlet (`context.setVariable("res", r)`).

Iteratori (`th:each`):

- `th:each="item : ${res.list}"` :
  - Si applica all'elemento `<tr>` . Questo `<tr>` (e i suoi `<td>` figli) verrà ripetuto.
  - `${res.list}` : Accede al campo `list` (che è un `ArrayList<Nominativo>`) dell'oggetto `res` .
  - `item` : È la variabile di iterazione. Ad ogni iterazione, `item` conterrà un oggetto `Nominativo` dalla lista.
  - Vengono generate tante copie di `<tr>` quanti sono gli elementi nella lista.

Contenuto di un Elemento (`th:text`):

- `th:text="${item.Name}"` :
  - Sostituisce il contenuto dell'elemento `<td>` con il valore del campo `Name` dell'oggetto `item` corrente.
  - Similmente per `${item.Age}` .

Output HTML Generato (Esempio):

Se `res.list` contiene due nominativi (Pippo, 30 e Pluto, 20):

```html
<!DOCTYPE html>
<html>
<head>
    <title>Elenco Nominativi</title>
    <meta charset="UTF-8">
</head>
<body>
    <h1>Nominativi</h1>

    <table border="1">
        <thead>
            <tr>
                <th>Nome</th>
                <th>Età</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Pippo</td>
                <td>30</td>
            </tr>
            <tr>
                <td>Pluto</td>
                <td>20</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

La slide mostra anche un output renderizzato nel browser.

Riassumendo su ThymeLeaf

- Rispetto ad altri approcci MVC (come JSP con taglib complesse o framework più pesanti), ThymeLeaf è una soluzione interessante perché:
  - È **estremamente snella** e facile da integrare.
  - **Disaccoppia** bene la View (template HTML) dal Controller (servlet).
  - Lo stesso template può essere utilizzato da diverse servlet (se i dati nel contesto hanno nomi consistenti).

- Nel template si vede chiaramente la **struttura dell'HTML** da generare, poiché i template ThymeLeaf sono essi stessi HTML validi (possono essere aperti e visualizzati nei browser, anche se le parti dinamiche non saranno processate). Questo è chiamato "Natural Templating".