

Sviluppo di Web Service Java RESTful con JAX-RS e Jersey

Indice generale

PANORAMICA SUI WEB SERVICE RESTFUL.....	3
PANORAMICA SU JAX-RS.....	4
CREAZIONE DEL PRIMO WEB SERVICE REST.....	5
PRODURRE VARI TIPI DI RAPPRESENTAZIONI DAI WEB SERVICE REST.....	6
DOWNLOAD DEL DEL DATABASE H2.....	8
CREAZIONE DEL DAO LAYER PER IL NOSTRO WEB SERVICE.....	11
IMPLEMENTAZIONE DEI METODI PER LE OPERAZIONI CRUD E LE QUERY ALLA TABELLA.....	16

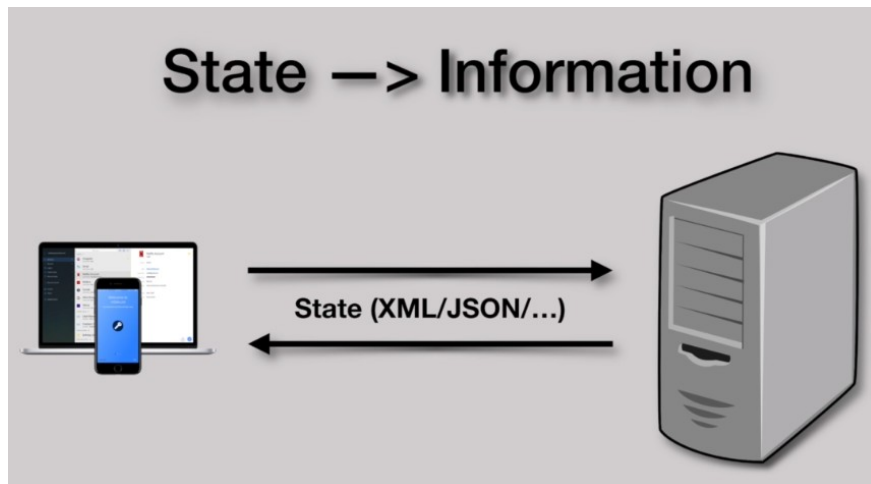
PANORAMICA SUI WEB SERVICE RESTFUL

Per prima cosa, cerchiamo di capire il significato del termine REST.

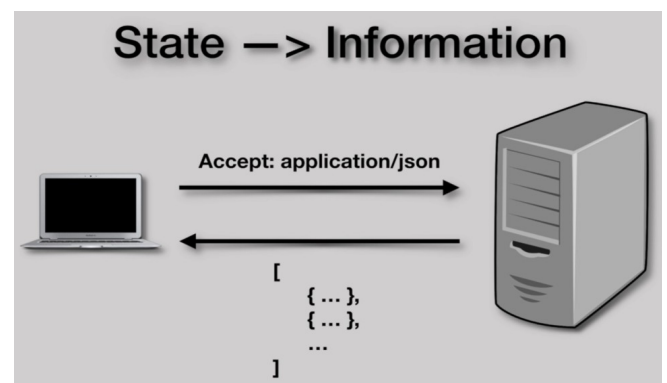
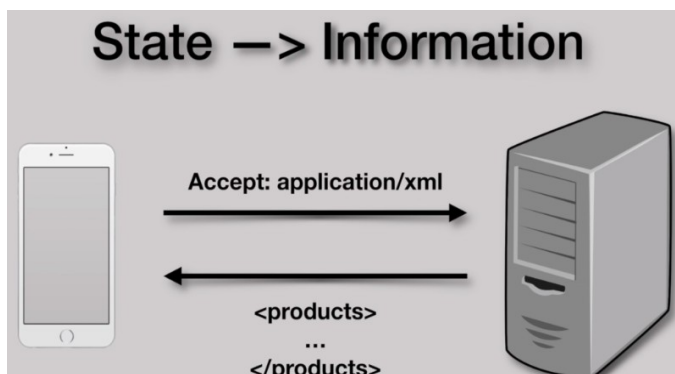
REST è l'acronimo di REpresentational State Transfer, che in italiano si traduce come "trasferimento di stato in rappresentazioni diverse".

Il termine "stato" qui si riferisce allo stato di un insieme di oggetti, denominato risorsa. Per esempio i prodotti, i clienti, gli ordini e gli utenti possono essere considerati tutti come risorse.

In generale ogni risorsa ha alcune informazioni (che definiscono il suo stato) che possono essere scambiate o trasferite in formati diversi, dette rappresentazioni.



Ad esempio, un'applicazione mobile può inviare richieste al server in formato XML, mentre un'altra web application può inviare richieste in formato JSON.



È possibile effettuare una negoziazione sul formato delle informazioni scambiate tra chi ha la risorsa che effettua una richiesta e chi ha la risorsa che eroga il servizio richiesto, in modo tale da avere due rappresentazioni uguali tra loro.

Per i web service REST è necessario:

- utilizzare un URI (Uniform Resource Identifier) per accedervi;
- utilizzare i metodi HTTP (GET, POST, PUT, DELETE...), che rappresentano le azioni da eseguire su una risorsa. Ad esempio, si può effettuare una richiesta GET al server per ottenere informazioni sullo stato di uno specifico ordine, oppure una richiesta POST nel caso in cui si voglia aggiungere un nuovo stato ad una risorsa esistente.

PANORAMICA SU JAX-RS

JAX-RS è una specifica standardizzata da JCP (Java Community Process). JCP è l'istituzione che si occupa di regolare lo sviluppo della tecnologia JAVA.

JAX-RS supporta la creazione di web service REST, che è semplificata grazie all'utilizzo delle annotation presenti all'interno di questa specifica. Una nota importante è che per utilizzare JAX-RS non è richiesta alcuna configurazione.

JAX-RS ha alcune annotation che ci permettono di mappare una classe JAVA come risorsa web. Diamo un'occhiata alle annotation più comunemente utilizzate:

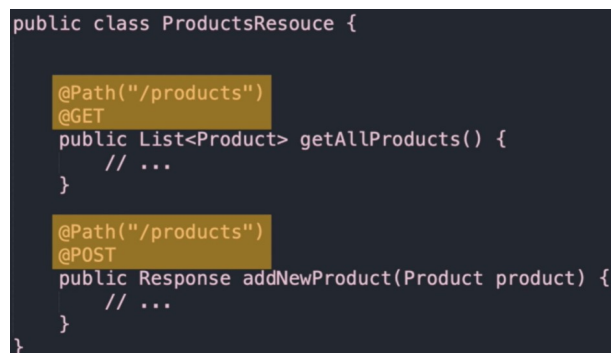
- `@Path`: corrisponde al path a cui risponde un metodo del servizio;



```
http://example.com/api/products

public class ProductsResource {
    @Path("/products")
    public List<Product> getAll() {
        // ...
    }
}
```

- `@GET`, `@POST`, `@PUT`, `@DELETE`: specificano il tipo di richiesta HTTP della risorsa;



```
public class ProductsResource {

    @Path("/products")
    @GET
    public List<Product> getAllProducts() {
        // ...
    }

    @Path("/products")
    @POST
    public Response addNewProduct(Product product) {
        // ...
    }
}
```

- `@Produces`: specifica il tipo di risposta restituita. Il tipo dell'informazione è noto come MIME type.



```
public class ProductsResource {

    @Path("/products/{id}")
    @GET
    @Produces({"application/xml", "application/json"})
    public Product getOne(@PathParam("id") int id) {
        // ...
    }
}
```

- `@Consumes`: specifica il tipo di richiesta accettata;

CREAZIONE DEL PRIMO WEB SERVICE REST

Per poter usare le API di JAX-RS, si devono scaricare e inserire all'interno del progetto in cui andremo a sviluppare i servizi REST. Se si sta utilizzando una build automation (ad esempio Maven o Gradle) si devono inserire le dipendenze, altrimenti va fatto tutto a mano.

Nel nostro caso utilizziamo Maven, quindi possiamo importare la seguente dipendenza all'interno del file pom.xml:

```
<dependencies>
  <dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0</version>
  </dependency>

  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.17</version>
  </dependency>

  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>1.17</version>
  </dependency>

  <dependency>
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-json-provider</artifactId>
    <version>2.9.5</version>
  </dependency>
</dependencies>
```

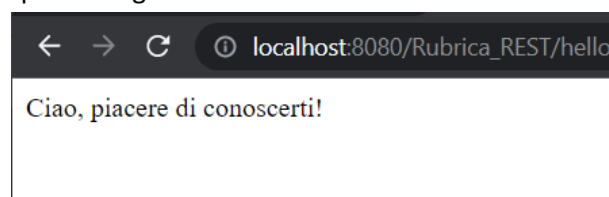
Ora possiamo iniziare a creare il nostro primo servizio REST:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/hello")
public class RisorsaHello {

    @GET
    public String saluto(){ //http://localhost:8080/Rubrica_REST/hello
        return "Ciao, piacere di conoscerti!";
    }
}
```

Il risultato di questo esempio è il seguente:



PRODURRE VARI TIPI DI RAPPRESENTAZIONI DAI WEB SERVICE REST

Per definire qual è il tipo di rappresentazione dell'informazione rilasciata da una risorsa, si deve utilizzare l'annotation `@Produces` sopra i vari metodi del servizio, come nell'esempio seguente:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/hello")
public class RisorsaHello {

    @GET
    @Produces({"text/plain"})
    public String saluto(){ //http://localhost:8080/Rubrica_REST/hello
        return "Ciao, piacere di conoscerti, da Paolo";
    }

    @GET
    @Produces({"application/xml"})
    public String salutoXML(){
        return "<?xml version='1.0' ?>\n" +
            "\n" +
            "<saluto>\n" +
            "    <messaggio>Ciao, piacere di conoscerti</messaggio>\n" +
            "    <da>Paolo</da>\n" +
            "</saluto>";
    }

    @GET
    @Produces({"application/json"})
    public String salutoJSON(){
        return "{\n" +
            "  \"messaggio\": \"Ciao. piacere di conoscerti\"\n" +
            "  \"da\": \"Paolo\"\n" +
            "}";
    }
}
```

I risultati di questo esempio sono i seguenti:

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

- Method:** GET
- URL:** http://localhost:8080/Rubrica_REST/hello
- Headers Tab:** Selected, showing a table with 7 headers. The 'Accept' header is highlighted with a red box, with a checked checkbox and the value 'text/plain'.
- Body Tab:** Shows the response body as 'Ciao, piacere di conoscerti, da Paolo'.

KEY	VALUE
<input checked="" type="checkbox"/> Accept	text/plain
Key	Value

Body: Ciao, piacere di conoscerti, da Paolo

GET ▼ http://localhost:8080/Rubrica_REST/hello

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Headers 6 hidden

	KEY	VALUE
<input checked="" type="checkbox"/>	Accept	application/xml
	Key	Value

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize XML ▼

```
1 <?xml version="1.0" ?>
2 <saluto>
3   <messaggio>Ciao, piacere di conoscerti</messaggio>
4   <da>Paolo</da>
5 </saluto>
```

GET ▼ http://localhost:8080/Rubrica_REST/hello

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Headers 6 hidden

	KEY	VALUE
<input checked="" type="checkbox"/>	Accept	application/json
	Key	Value

Body Cookies Headers (6) Test Results

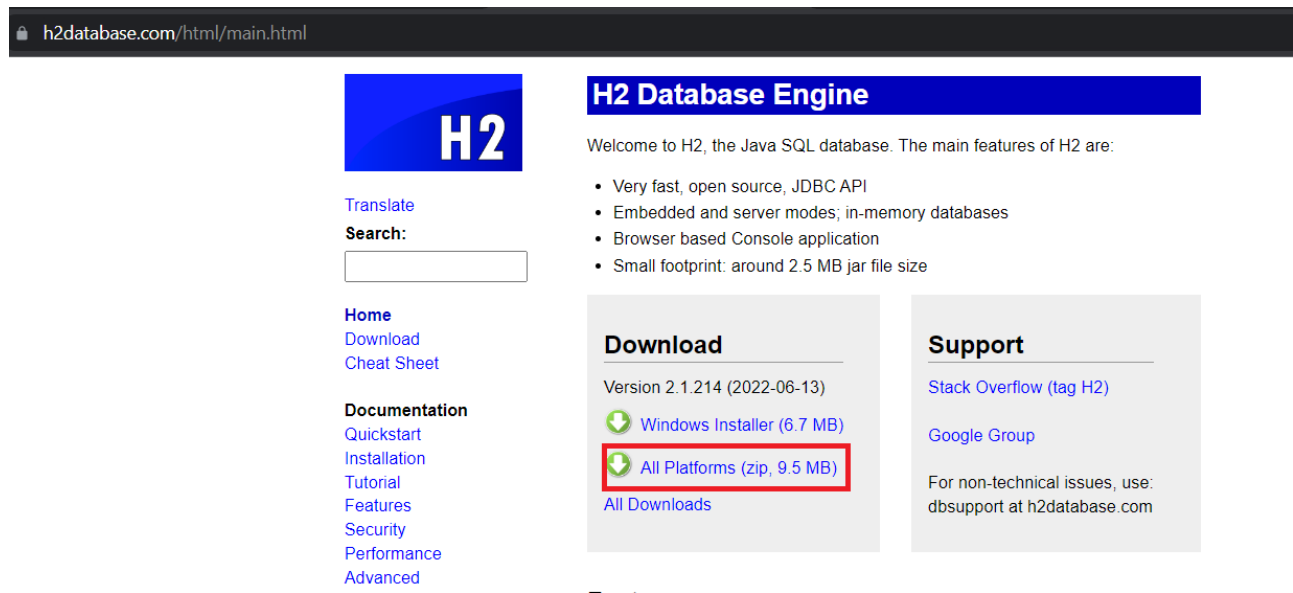
Pretty Raw Preview Visualize JSON ▼

```
1 {
2   "saluto": {
3     "messaggio": "Ciao. piacere di conoscerti"
4     "da": "Paolo"
5   }
6 }
```

DOWNLOAD DEL DEL DATABASE H2

Per questo corso utilizzeremo un database chiamato H2. Questo database non ha bisogno di essere configurato, ma verrà utilizzato quasi subito dopo il download.

Vediamo tutti i passaggi per il download di H2 e la creazione di un database:



The screenshot shows the H2 Database Engine website. The header includes the URL `h2database.com/html/main.html`. The main content area features the H2 logo, a search bar, and a list of links under 'Home' and 'Documentation'. The 'Download' section highlights the 'All Platforms (zip, 9.5 MB)' option, which is circled in red. The 'Support' section provides links to Stack Overflow and Google Group.

H2 Database Engine

Welcome to H2, the Java SQL database. The main features of H2 are:

- Very fast, open source, JDBC API
- Embedded and server modes; in-memory databases
- Browser based Console application
- Small footprint: around 2.5 MB jar file size

Download

Version 2.1.214 (2022-06-13)

- Windows Installer (6.7 MB)
- All Platforms (zip, 9.5 MB)**

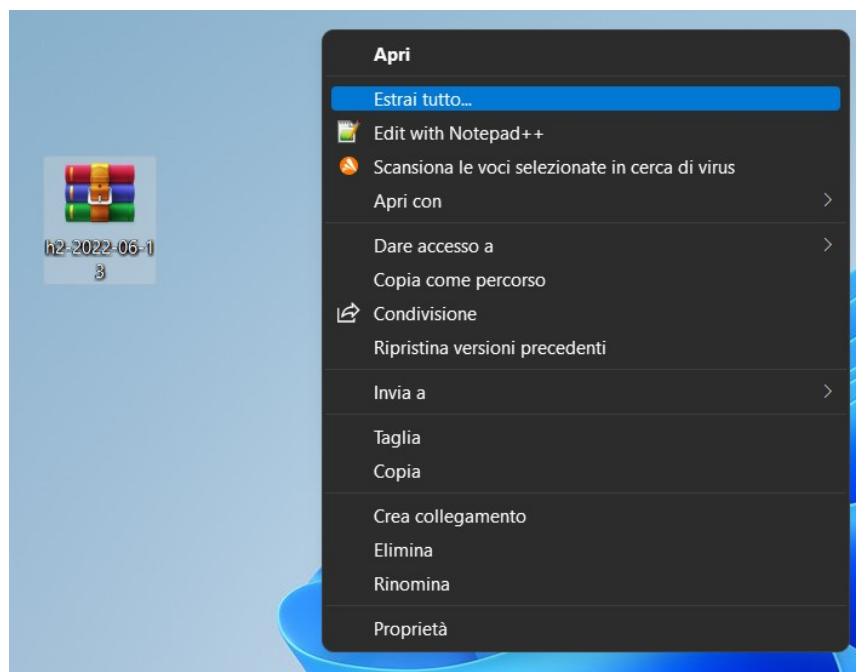
[All Downloads](#)

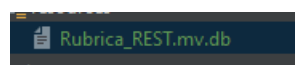
Support

[Stack Overflow \(tag H2\)](#)

[Google Group](#)

For non-technical issues, use: [dbsupport at h2database.com](mailto:dbsupport@h2database.com)





Creare un file con estensione .mv.db e inserirlo all'interno del progetto

← ↻ ⚠ Non sicuro | 192.168.1.9:8082/test.do?jsessionid=264bc8a6a112

English ▼ Preferenze Accessori Aiuto

Accesso

Configurazioni salvate: 1 Generic H2 (Embedded) ▼

Nome configurazione: 2 Generic H2 (Embedded) Salva Rimuovi

Classe driver: 3 org.h2.Driver

JDBC URL: 4 jdbc:h2:./devdb

Nome utente: 5 sa

Password: **opzionale**

7 Connetti Prova la connessione 6

Connessione riuscita

Non sicuro | 192.168.1.9:8082/login.do?jsessionid=264bc8a6a112805a4388abcf2aea78c

Auto inserimento Massimo numero righe: 1000 Auto completamento [Disattivo] Auto select On

jdbc:h2://devdb
 INFORMATION_SCHEMA
 Utenti
 H2 2.1.214 (2022-06-13)

Esegui	Run Selected	Auto completamento	Annulla	Comando SQL:
--------	--------------	--------------------	---------	--------------

Qui è possibile inserire le query SQL. Nel nostro caso creo una tabella denominata 'CONTACTS'

Comandi importanti

		Mostra questa pagina di aiuto
		Mostra l'elenco dei comandi eseguiti
	Ctrl+Invio	Esegue il corrente comando SQL
	Shift+Invio	Executes the SQL statement defined by the text selection
	Ctrl+Space	Auto completamento
		Disconnette dal database

Programma SQL di esempio

Cancella la tabella se esiste	DROP TABLE IF EXISTS TEST;
Crea una nuova tabella tramite ID e NOME colonne	CREATE TABLE TEST((ID INT PRIMARY KEY, NAME VARCHAR(255));
Aggiunge una nuova riga	INSERT INTO TEST VALUES(1, 'Hello');
Aggiunge un'altra riga	INSERT INTO TEST VALUES(2, 'World');
Seleziona tutto dalla tabella	SELECT * FROM TEST ORDER BY ID;
Cambia dei dati in una riga	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Rimuove una riga	DELETE FROM TEST WHERE ID=2;
Aiuto	HELP ...

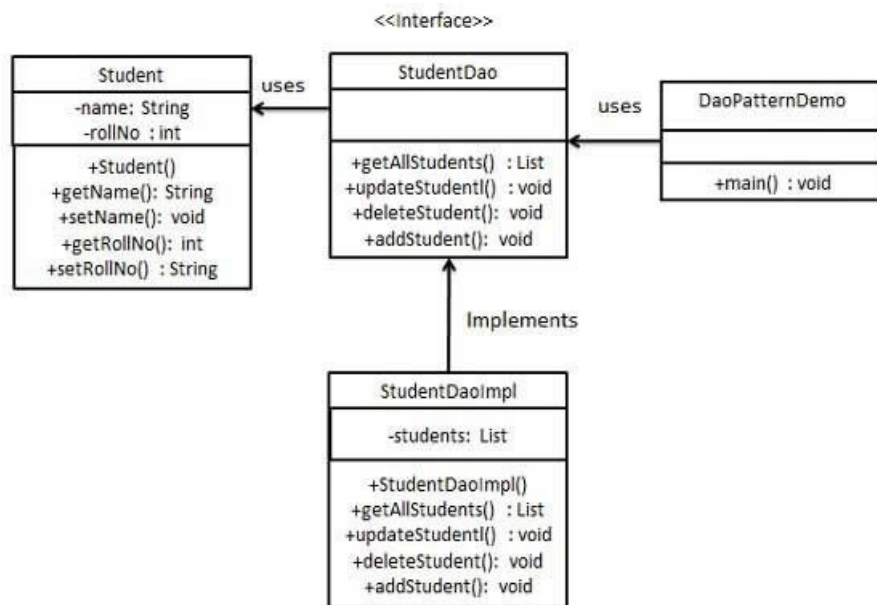
Aggiunta di altri driver per l'accesso al database

I drivers per il database possono essere inseriti aggiungendo la posizione del file Jar del driver stesso alle variabili di ambiente H2DRIVERS o CLASSPATH. E:
H2DRIVERS in C:/Programs/hsqldb/lib/hsqldb.jar.

CREAZIONE DEL DAO LAYER PER IL NOSTRO WEB SERVICE

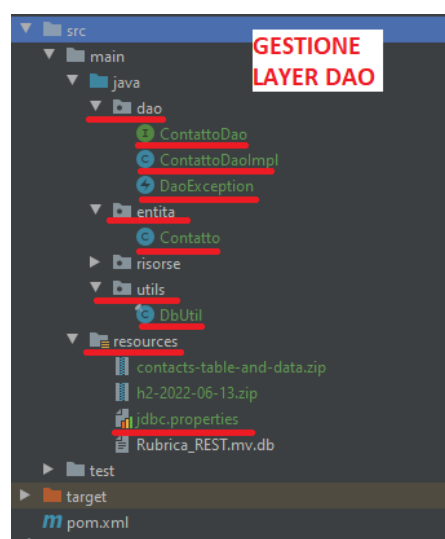
Il DAO (Data Access Object) è un pattern architetturale per la gestione della persistenza in un database. Consente di stratificare l'accesso al data layer da parte del business layer.

È strutturato in questa maniera:



- `Student` è la classe che rappresenta l'entità del database, con le variabili che corrispondono ai campi della tabella. Tutte le variabili avranno i loro metodi getter e setter;
- `StudentDao` è l'interfaccia che contiene le operazioni che si eseguono per fare le query alla tabella;
- `StudentDaoImpl` è l'implementazione dell'interfaccia `StudentDao`, in cui si definisce cosa fanno questi metodi.

Utilizziamo ora il pattern DAO layer per gestire il data layer della nostra applicazione Rubrica, come nell'esempio seguente:



Contatto.java

```
package entita;

/**
 * Classe che rappresenta l'entità del database, con le variabili
 * che corrispondono ai campi della tabella.
 */

public class Contatto {

    private Integer id;
    private String nome;
    private String genere;
    private String email;
    private String telefono;
    private String citta;
    private String nazione;

    public Contatto() {
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getGenere() {
        return genere;
    }

    public void setGenere(String genere) {
        this.genere = genere;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
```

```

        this.telefono = telefono;
    }

    public String getCitta() {
        return citta;
    }

    public void setCitta(String citta) {
        this.citta = citta;
    }

    public String getNazione() {
        return nazione;
    }

    public void setNazione(String nazione) {
        this.nazione = nazione;
    }
}

```

ContattoDao.java

```

package dao;

import entita.Contatto;

import java.util.List;

/**
 * Interfaccia che contiene le operazioni che si eseguono per fare le query alla tabella;
 */

public interface ContattoDao {

    //OPERAZIONI CRUD
    public Contatto aggiuntaContatto(Contatto contatto) throws DaoException;
    public Contatto ricercaContattoPerId(Integer id) throws DaoException;
    public Contatto modificaContatto(Contatto contatto) throws DaoException;
    public void cancellazioneContatto(Integer id) throws DaoException;

    //QUERY
    public List<Contatto> ricercaTuttiIContatti() throws DaoException;
    public List<Contatto> ricercaContattoPerCitta(String citta) throws DaoException;
    public List<Contatto> ricercaContattoPerPaese(String paese) throws DaoException;
}

```

DaoException.java

```

package dao;

public class DaoException extends Exception {

    private static final long serialVersionUID = -8093755388375054066L;

    public DaoException() {
    }
}

```

```

    public DaoException(String message) {
        super(message);
    }

    public DaoException(Throwable cause) {
        super(cause);
    }
}

```

ContattoDaoImpl.java

```

package dao;

import entita.Contatto;

import java.util.List;

/**
 * Classe che implementa l'interfaccia ContattoDao
 */
public class ContattoDaoImpl implements ContattoDao {

    public Contatto aggiuntaContatto(Contatto contatto) throws DaoException {
        return null;
    }

    public Contatto ricercaContattoPerId(Integer id) throws DaoException {
        return null;
    }

    public Contatto modificaContatto(Contatto contatto) throws DaoException {
        return null;
    }

    public void cancellazioneContatto(Integer id) throws DaoException {
    }

    public List<Contatto> ricercaTuttiIContatti() throws DaoException {
        return null;
    }

    public List<Contatto> ricercaContattoPerCitta(String citta) throws DaoException {
        return null;
    }

    public List<Contatto> ricercaContattoPerPaese(String paese) throws DaoException {
        return null;
    }
}

```

DbUtil.java

```
package utils;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ResourceBundle;

/**
 * Classe per aprire una connessione con il database H2
 */
public final class DbUtil {
    private static String driver, url, user, password;

    private DbUtil() {
    }

    static {
        ResourceBundle rb = ResourceBundle.getBundle("jdbc");
        driver = rb.getString("jdbc.driver");
        url = rb.getString("jdbc.url");
        user = rb.getString("jdbc.user");
        password = rb.getString("jdbc.password");
    }

    public static Connection getConnection() throws ClassNotFoundException, SQLException {
        Class.forName(driver);
        return DriverManager.getConnection(url, user, password);
    }
}
```

jdbc.properties

```
#Dati per connessione al database H2
#####
jdbc.url=jdbc:h2:./devdb
jdbc.driver=org.h2.Driver
jdbc.user=sa
jdbc.password=
#####
```

IMPLEMENTAZIONE DEI METODI PER LE OPERAZIONI CRUD E LE QUERY ALLA TABELLA