

# **CLIMATE MONITORING**

***MANUALE TECNICO***

# INDICE

<b>1. INTRODUZIONE</b>	<b>3</b>
<b>2. CARATTERISTICHE GENERALI</b>	<b>3</b>
2.1. MAVEN	3
2.2. JAVA RMI	4
<b>3. ARCHITETTURA DEL SOFTWARE</b>	<b>4</b>
3.1. SCELTE ARCHITETTURALI	4
3.1.1. PATTERN STRUTTURALE DAO & ARCHITETTURA MULTILAYER (SERVER)	4
3.1.2. IMPLEMENTAZIONE DEL SINGLETON PATTERN	5
2.1. CLIENT	6
STRUTTURA DEL CODICE	6
<b>1. INTRODUZIONE</b>	<b>8</b>
<b>2. ANALISI DEI REQUISITI</b>	<b>8</b>
<b>3. PROGETTAZIONE DEL DATABASE</b>	<b>10</b>
3.1. SCHEMA CONCETTUALE	10
3.1.1. SCELTE PROGETTUALI	10
ENTITA'	10
VINCOLI DI INTEGRITA'	11
INTEGRITA' REFERENZIALE	11
VINCOLI AGGIUNTIVI	11
3.2. SCHEMA CONCETTUALE RISTRUTTURATO	12
3.2.1. DETTAGLI SULLA RISTRUTTURAZIONE	12
3.3. SCHEMA LOGICO	13
3.3.1. TRADUZIONE	13
<b>4. IMPLEMENTAZIONE DEL DATABASE</b>	<b>13</b>
4.1. CREAZIONE DEL DATABASE	13
4.2. ELIMINAZIONE DEL DATABASE	13
4.3. CREAZIONE DELLE RELAZIONI	13
4.4. QUERY	15
<b>5. INTEGRAZIONE DEL DATABASE NEL PROGETTO</b>	<b>17</b>
5.1. IL POM (Project Object Model)	17
5.2. CREAZIONE DEL DATABASE TRAMITE XML	17
<b>6. CREDITI</b>	<b>22</b>

# **1.INTRODUZIONE**

Con questo documento si vogliono illustrare i dettagli tecnici relativi alla progettazione e l'implementazione del progetto ClimateMonitoring.

Verrà illustrata l'architettura di sistema con le relative scelte progettuali; verrà anche presentata la struttura del codice sorgente, con annessa spiegazione per ogni componente significativa o delle scelte algoritmiche particolari; infine, verrà esposta la progettazione della componente database.

## **2. CARATTERISTICHE GENERALI**

ClimateMonitoring è un progetto software sviluppato in Java con Maven, un insieme di tools e utilities per gestire in modo facile il project building e l'integrazione di elementi esterni come librerie o database.

Si tratta di un sistema di tipo distribuito, è composto quindi da diverse componenti che devono essere in grado di comunicare da remoto attraverso la rete; in particolare, è strutturato seguendo la classica architettura Client / Server.

La comunicazione avviene per mezzo di Java RMI (Remote Method Invocation), un framework che permette l'utilizzo remoto di servizi in modalità sincrona.

Le applicazioni forniscono una semplice GUI (Graphical User Interface), costruita con la libreria Swing di Java.

Il database, parte della componente Server, è di tipo relazionale e gestito con un DBMS PostgreSQL.

### **2.1. STRUMENTI SOFTWARE USATI**

#### **2.1.1. MAVEN**

Maven è lo strumento di build automation utilizzato per gestire il progetto Java per ClimateMonitoring.

Permette, attraverso il POM (Project Object Model), di integrare facilmente alla struttura del progetto elementi esterni e di specificare dettagli utili per la fase di build.

Infatti, con i cosiddetti lifecycle, ovvero delle istruzioni di building del progetto, è possibile congiungere e integrare facilmente ogni componente, potendo anche costruire eseguibili jar in pochi passi.

Maven dispone anche di un vasto repository online di librerie (o dependencies), permettendo in qualunque momento di cercare e scaricare, attraverso il POM, qualsiasi libreria, o di caricarne di proprie.

### 2.1.2. JAVA RMI

L'architettura RMI (Remote Method Invocation) è stata scelta per implementare la comunicazione remota tra le componenti client e server.

Infatti l'applicazione server è in grado di esporre i suoi servizi (oggetti remoti) in modo del tutto trasparente attraverso delle interfacce, poichè gli oggetti remoti sono esportati e caricati su un Registro.

In particolare, Java RMI gestisce in modo automatico qualsiasi aspetto legato alla connessione e l'accesso concorrente agli oggetti, risultando quindi più facile e intuitivo rispetto la classica implementazione di un sistema distribuito attraverso i Socket.

### 2.1.3. JAVA SWING

Swing è il framework standard per la creazione di GUI per applicazioni Java, pensato come estensione del precedente AWT (Abstract Window Toolkit), ma scritto completamente in Java.

Rispetto ad altre librerie, Swing è ormai ben consolidata e viene costantemente supportata ufficialmente con gli aggiornamenti del JDK.

Risulta dunque essere intuitiva e di facile utilizzo, con una vasta disponibilità di risorse online.

## **2.2. LIBRERIE UTILIZZATE**

Di seguito sono elencate le principali librerie esterne utilizzate nel progetto software:

### ***JDBC***

Questa libreria fornisce tutti gli strumenti per stabilire la connessione con il DBMS locale attraverso l'inizializzazione di un Driver.

Predispose anche tutti gli strumenti per formattare le strutture dati del DBMS con quelle di Java.

Infine, permette di eseguire le query SQL in modo da prevenire problematiche di Injection attraverso i prepared statements.

La versione di JDBC utilizzata nel progetto è già predisposta per interfacciarsi con PostgreSQL.

### ***JBCRYPT***

Implementazione Java dell'algoritmo BCrypt per l'hashing delle password.

BCrypt è un algoritmo di slow hashing con salt, questo vuol dire che l'hash della password viene effettuato con l'aggiunta di una stringa random (salt), e il tutto avviene in tempi sufficientemente onerosi per disperdere attacchi di forza bruta o Dictionary attacks.

L'hash BCrypt è una stringa di 60 caratteri (contenente anche il salt), ed è possibile impostare l'algoritmo con un fattore di costo che aumenta esponenzialmente i tempi di esecuzione.

## **3. ARCHITETTURA DEL SOFTWARE**

Come già anticipato, ClimateMonitoring è suddiviso in una componente Client e una Server, ovvero due applicazioni Java autonome.

Le due applicazioni condividono il package ***commons***, che predispone tutte quelle classi e interfacce che ricorrono in entrambe le parti comuni; in particolare, si tratta di classi per definire la comunicazione e per rappresentare oggetti.

### **3.1. SCELTE ARCHITETTURALI**

#### **3.1.1. PATTERN STRUTTURALE DAO & ARCHITETTURA MULTILAYER (SERVER)**

Il sistema distribuito, in quanto prevede anche l'integrazione di un database e l'utilizzo di Java RMI, è stato sviluppato seguendo le linee guida definite dal pattern strutturale DAO (Data Access Object).

Questo pattern, utilizzato per organizzare l'applicazione server, permette di astrarre l'interazione con il database e di isolarla da tutto il resto all'interno del sistema: si tratta quindi di definire una struttura multilayer dove sia rispettato il principio di singola responsabilità.

Nei dettagli, per ogni relazione significativa deve essere definita un'interfaccia (con relativa classe implementativa) che fornisce i metodi per interagire con la fonte dei dati in maniera del tutto trasparente.

In questo caso, ogni classe implementativa definisce le query SQL utili per interrogare il database e implementa i metodi richiesti, occupandosi di stabilire la connessione con il database e formattare i dati letti.

Al di sopra di questo layer, sono costruite le classi che rappresentano gli oggetti remoti esposti dal server: usano quindi i metodi forniti dalle interfacce DAO per recuperare i dati col fine di soddisfare le chiamate effettuate dal client.

Come livello superficiale abbiamo i due thread principali dell'applicazione server:

- il main thread che risponde agli eventi della GUI;
- un thread ausiliario per ospitare il servizio RMI;

### 3.1.2. IMPLEMENTAZIONE DEL SINGLETON PATTERN

All'interno del progetto è stato fatto uso del pattern Singleton: si tratta di un metodo di progettazione per definire una classe che può essere istanziata una sola volta all'interno dello stesso processo.

L'implementazione del Pattern Singleton utilizzata è quella dell'Holder con Lazy Initialization (o Singleton di Bill Pugh): l'istanza singola è conservata in una classe statica interna, chiamata Holder, e il riferimento viene restituito attraverso un metodo statico della classe Singleton; di seguito un'illustrazione:

```
1  public class A {
2      private A() { }
3      private static class Holder {
4          private static final A singleton = new A();
5      }
6      public static A ottieniIstanza() {
7          return Holder.singleton;
8      }
9  }
```

Con questa soluzione è possibile anche avere una migliore gestione delle risorse, poichè l'istanza Singleton verrà istanziata solo quando è necessario; questo è ottenuto sfruttando il [principio di lazy initialization della JVM](#): la classe A, al momento dell'inizializzazione, non viene caricata completamente, perciò la classe Holder viene inizializzata solamente quando il metodo ottieniIstanza() viene invocato per la prima volta, e a seguire anche l'istanza Singleton.

Sempre per questo motivo, rispetto alla classica implementazione del Singleton, la gestione threadsafe è garantita senza alcun tipo di blocco: la JVM, nella fase di inizializzazione della classe, opera in modo sequenziale, e data la presenza di un attributo statico, ogni chiamata successiva alla definizione restituirà sempre lo stesso valore;

Data l'assenza di un metodo *synchronized*, si hanno quindi performance migliori in situazioni con numerosi accessi concorrenti.

## 2.1. CLIENT

La parte **Client** gestisce l'interazione con l'utente e la presentazione finale dei dati attraverso la GUI.

### 2.1.1. STRUTTURA DEL CODICE

```
package client
    package clientrmi
        class ClientRMI
        class ResetClient
    package frame
        class AreaOperatore
        class Cittadino
        class FrameOperatore
        class GestioneScelta
        class Login
        class Parametri
        class RegistraCentro
        class Registrazione
        class ScegliCentro
        class StampaParametri
    package registraeventi
        class Chiusura
```

### 2.1.2. ILLUSTRAZIONE DELLE CLASSI

#### **ClientRMI**

Classe che gestisce la connessione al servizio RMI per il Client.

Il costruttore inizializza la connessione con il registro ed effettua il lookup degli oggetti remoti.

Agisce come container per gli stub dei servizi remoti, le cui istanze possono essere richieste attraverso metodi statici.

#### **ResetClient**

Classe di utility che fornisce un metodo statico per implementare il cambio di i







# 1. INTRODUZIONE

Con questo documento si vuole mostrare la parte di progettazione relativa al database del progetto ClimateMonitoring.

Vengono quindi illustrate tutte le fasi di progettazione (con schemi annessi) del database, dall'analisi dei requisiti sino alla definizione delle relazioni finali.

Sono inoltre elencate e spiegate tutte le query SQL utilizzate per creare le relazioni e interrogare il database.

Infine, viene spiegato come il database viene integrato all'interno del progetto Maven per Java.

## 2. ANALISI DEI REQUISITI

I dati relativi alle aree di interesse dovranno essere salvati nel database e comprendono:

- GeonameID
- Nome del paese in UNICODE
- Nome del paese in codice ASCII
- Codice dello stato
- Nome dello stato
- Latitudine
- Longitudine

Inoltre l'applicazione ClimateMonitoring prevede la registrazione di operatori climatici, i dati di questi ultimi dovranno essere salvati nell'applicazione e riguardano:

- UserID
- Nome
- Cognome
- Codice fiscale
- Email
- Password
- Nome del centro di appartenenza

Ogni operatore registrato correttamente può registrare un centro di monitoraggio oppure sceglierlo, i dati relativi ai centri sono:

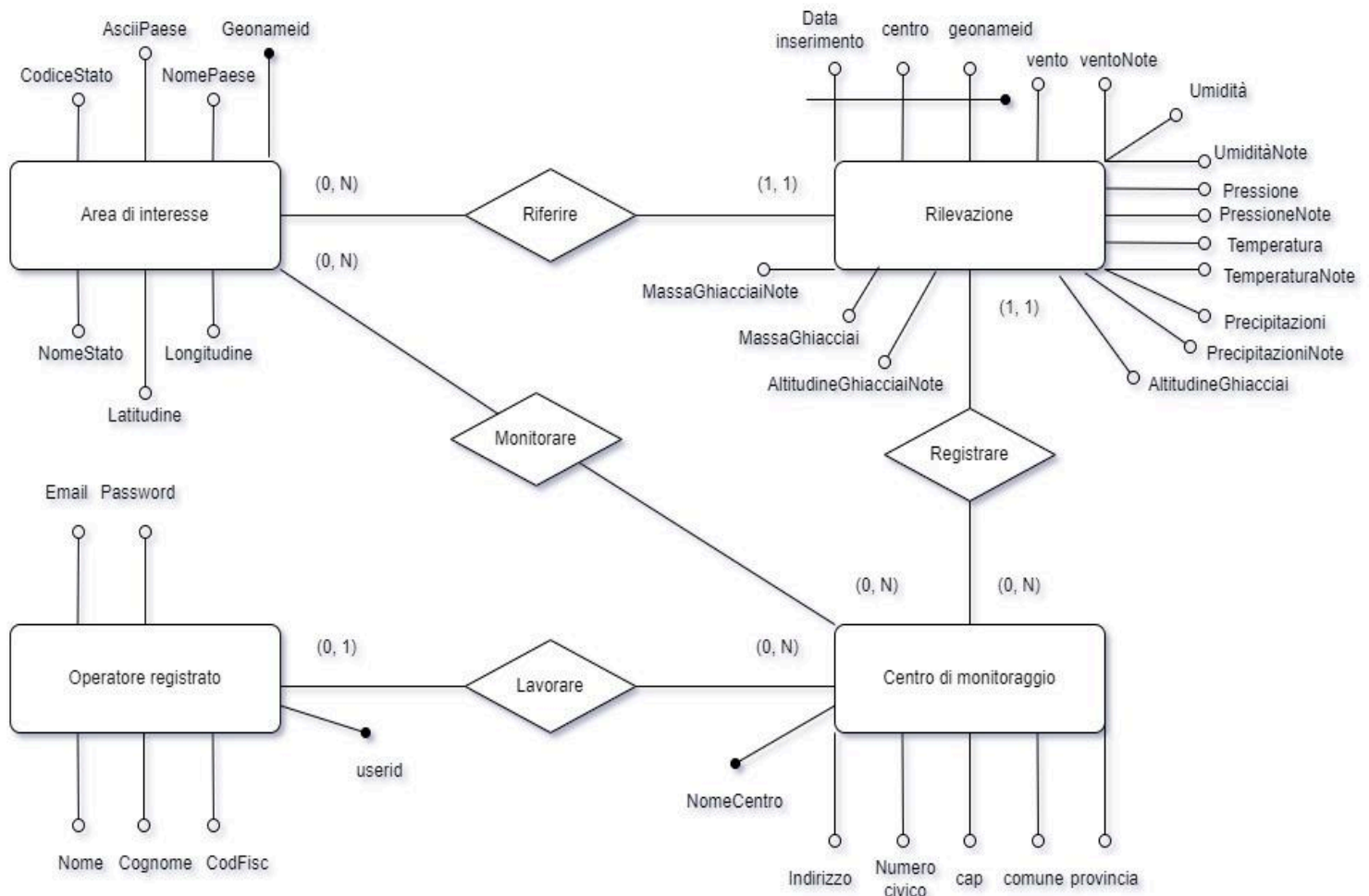
- Nome
- Indirizzo
- Numero civico
- CAP
- Comune
- Provincia

Infine ogni operatore con un centro di appartenenza può fare delle rilevazioni su delle aree di interesse monitorate dal centro, queste rilevazioni andranno salvate e dovranno contenere i seguenti dati:

- ❖ Nome del centro
- ❖ Area monitorata
- ❖ Data di inserimento
- ❖ Tutti i parametri climatici e le relative note:
  - Vento
  - Umidità
  - Pressione
  - Temperatura
  - Precipitazioni
  - Altitudine dei ghiacciai
  - Massa dei ghiacciai

## 3. PROGETTAZIONE DEL DATABASE

### 3.1. SCHEMA CONCETTUALE



#### 3.1.1. SCELTE PROGETTUALI

##### ENTITA'

- OPERATORE REGISTRATO

Include i relativi attributi che un operatore climatico deve poter avere. Come chiave primaria e quindi attributo univoco si utilizzano l'intero userID e come chiave esterna il nome del centro di appartenenza dell'operatore rappresentato dall'attributo NomeCentro.

- CENTRO DI MONITORAGGIO

Ha come attributi i dati di un centro di monitoraggio e come chiave primaria il nome univoco del centro identificato dall'attributo NomeCentro.

- RILEVAZIONE

Contiene una chiave primaria composta, infatti gli attributi GeonameID, NomeCentro e DataInserimento formano la PRIMARY KEY. Come chiavi esterne si utilizzano GeonameID e NomeCentro.

- **COORDINATE MONITORAGGIO**

Possiede come chiave primaria GeonameID, ovvero un codice univoco che rappresenta un'area di interesse.

### ***VINCOLI DI INTEGRITA'***

- **NOT NULL**

Fare in modo che non vengano lasciati vuoti attributi importanti.

- **ON UPDATE CASCADE**

Viene usato per garantire che quando il valore di una colonna chiave primaria in una tabella viene aggiornato, anche i valori corrispondenti nelle colonne di chiave esterna nelle tabelle collegate vengano aggiornati automaticamente.

- **ON DELETE CASCADE**

Viene usato per garantire che quando una riga viene eliminata nella tabella con la chiave primaria, tutte le righe correlate nelle tabelle collegate vengano eliminate automaticamente.

### ***INTEGRITA' REFERENZIALE***

Sono state definite le seguenti relazioni tra entità, che implicano quindi il rispetto dell'integrità referenziale con attribuzione di una chiave esterna:

- Operatore Registrato - Centro di Monitoraggio
- Centro di Monitoraggio - Rilevazione
- Area di Interesse - Rilevazione
- Area di Interesse - Centro di Monitoraggio.

### ***VINCOLI AGGIUNTIVI***

- **VALORI DELLE COORDINATE GEOGRAFICHE**

*CHECK(Latitudine >= -90 AND Latitudine <= 90)*

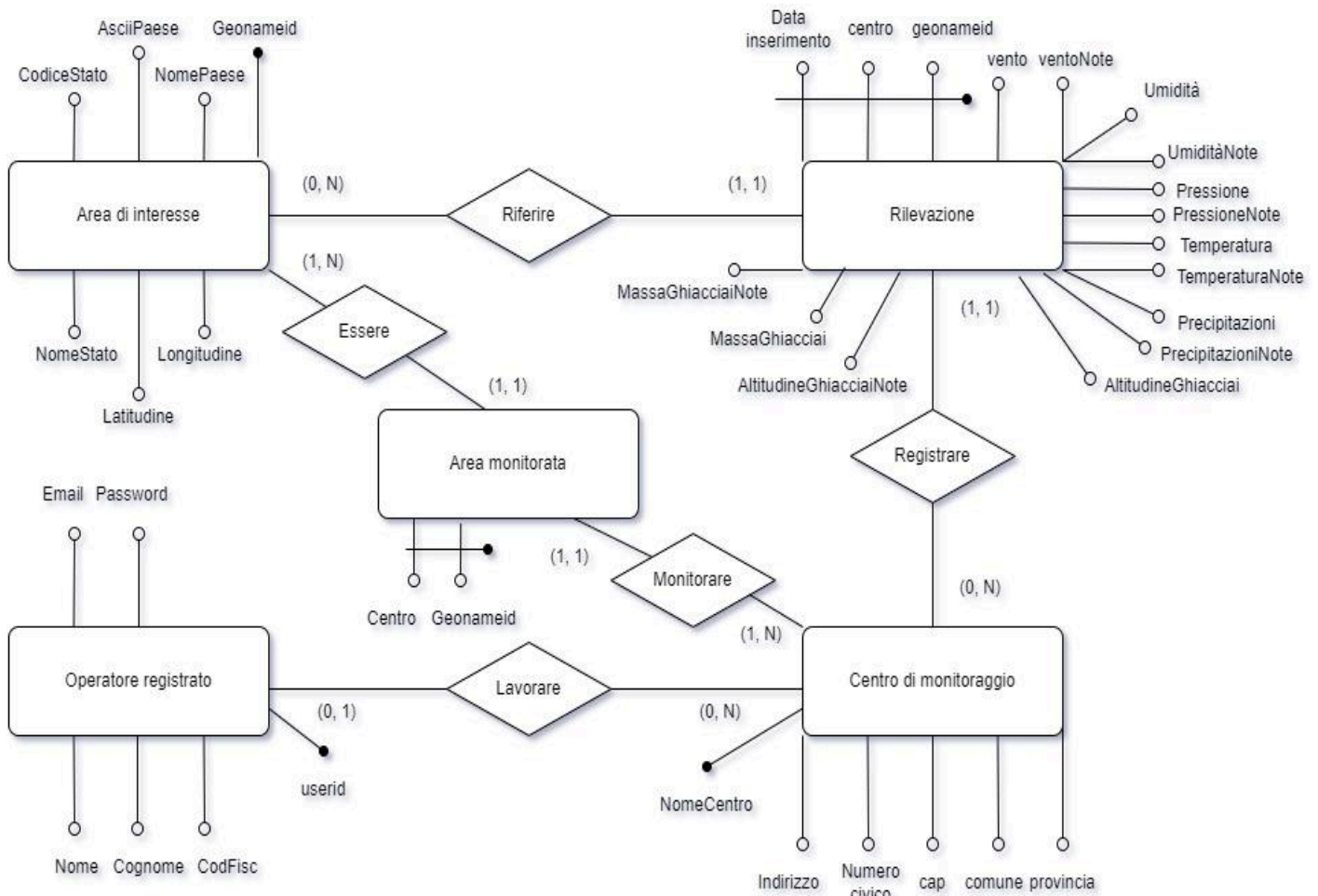
*CHECK(Longitudine >= -180 AND Longitudine <= 180)*

Questi due vincoli servono a controllare che i valori inseriti per la latitudine e la longitudine rientrino nel range di valori accettabili, rispettivamente [-90, 90] e [-180, 180].

- **PUNTEGGI SUI PARAMETRI CLIMATICI**

$CHECK(Vento \geq 0 \text{ AND } Vento \leq 5)$

Per ogni categoria di parametri climatici, è stato definito un vincolo del tutto simile a quella riportato sopra che verifica se il punteggio attribuito alla valutazione rientri nel range definito per la scala di intensità dei fenomeni (con l'aggiunta del punteggio 0 per le valutazioni nulle), cioè [0, 5].



## 3.2. SCHEMA CONCETTUALE RISTRUTTURATO

### 3.2.1. DETTAGLI SULLA RISTRUTTURAZIONE

Si è scelto di sviluppare la relazione N:N tra le entità Area di interesse e l'entità Centro di monitoraggio per semplificare la gestione dei dati aggiungendo la nuova entità Area monitorata che contiene una chiave primaria ed esterna formata dalla coppia GeonameID e Centro che sono attributi chiave primaria di Area di interesse e Centro di monitoraggio.

### 3.3. SCHEMA LOGICO

#### 3.3.1 TRADUZIONE

Si traducono le relazioni al plurale come richiesto nelle linee guida del progetto.

La traduzione avviene nel seguente modo:

- Area di interesse → CoordinateMonitoraggio
- Operatore registrato → OperatoriRegistrati
- Centro di monitoraggio → CentriMonitoraggio
- Rilevazione → ParametriClimatici
- Area monitorata → AreeMonitorateDaCentri

**CoordinateMonitoraggio** (GeonameID, NomePaese, AsciiPaese, CodiceStato, NomeStato, Latitudine, Longitudine)

**OperatoriRegistrati** (UserID, Nome, Cognome, CodFisc, Email, Password, NomeCentroCentriMonitoraggio)

**CentriMonitoraggio** (NomeCentro, Indirizzo, NumeroCivico, cap, Comune, Provincia)

**ParametriClimatici** (GeonameID<sub>CoordinateMonitoraggio</sub>, NomeCentro<sub>CentriMonitoraggio</sub>, DataInserimento, Vento, nventoNote, Umidita, UmiditaNote, Pressione, PressioneNote, Temperatura, TemperaturaNote, AltitudineGhiacciai, AltitudineGhiacciaiNote, MassaGhiacciai, MassaGhiacciaiNote)

**AreeMonitorateDaCentri** (NomeCentro<sub>CentriMonitoraggio</sub>, GeonameID<sub>CoordinateMonitoraggio</sub>)

## 4. IMPLEMENTAZIONE DEL DATABASE

In questa sezione vengono illustrate le query SQL utilizzate nel programma per la definizione del database relazione di ClimateMonitoring

### 4.1. CREAZIONE DEL DATABASE

```
CREATE DATABASE climatemonitoring;
```

### 4.2. ELIMINAZIONE DEL DATABASE

```
DROP DATABASE IF NOT EXISTS climatemonitoring;
```

### 4.3. CREAZIONE DELLE RELAZIONI

#### AREA DI INTERESSE

```
CREATE TABLE CoordinateMonitoraggio (  
    GeonameID char(10) PRIMARY KEY,
```

```

NomePaese varchar(70) NOT NULL,
AsciiPaese varchar(70) NOT NULL,
CodiceStato char(2) NOT NULL,
NomeStato varchar(70) NOT NULL,
Latitudine float NOT NULL CHECK (Latitudine >= -90 AND Latitudine <= 90),
Longitudine float NOT NULL CHECK (Longitudine >= -180 AND Longitudine <=
180)
);

```

### OPERATORE REGISTRATO

```

CREATE TABLE OperatoriRegistrati (
    UserID int PRIMARY KEY,
    Nome varchar(30) NOT NULL,
    Cognome varchar(30) NOT NULL,
    CodFisc char(16) NOT NULL,
    Email varchar(30) NOT NULL,
    Password varchar(60) NOT NULL,
    NomeCentro varchar(20) REFERENCES CentriMonitoraggio(NomeCentro) ON UPDATE
    CASCADE ON DELETE SET NULL
);

```

### CENTRO DI MONITORAGGIO

```

CREATE TABLE CentriMonitoraggio (
    NomeCentro varchar(20) PRIMARY KEY,
    Indirizzo varchar(50) NOT NULL,
    numeroCivico int NOT NULL,
    cap char(5) NOT NULL,
    Comune varchar(50) NOT NULL,
    Provincia char(2) NOT NULL
);

```

### RILEVAZIONE

```

CREATE TABLE ParametriClimatici (
    GeonameID char(7) REFERENCES CoordinateMonitoraggio(GeonameID) ON UPDATE
    CASCADE ON DELETE CASCADE,
    NomeCentro varchar(20) REFERENCES CentriMonitoraggio(NomeCentro) ON UPDATE
    CASCADE ON DELETE CASCADE,
    DataInserimento timestamp NOT NULL,
    Vento int NOT NULL CHECK (Vento >= 0 AND Vento <= 5),
    VentoNote varchar(256),
    Umidita int NOT NULL CHECK (Umidita >= 0 AND Umidita <= 5),
    UmiditaNote varchar(256),
    Pressione int NOT NULL CHECK (Pressione >= 0 AND Pressione <= 5),
    PressioneNote varchar(256),
    Temperatura int NOT NULL CHECK (Temperatura >= 0 AND Temperatura <= 5),
    TemperaturaNote varchar(256),
    Precipitazioni int NOT NULL CHECK (Precipitazioni >= 0 AND Precipitazioni <=
    5),
    PrecipitazioniNote varchar(256),

```



```

        AltitudineGhiacciai int NOT NULL CHECK (AltitudineGhiacciai >= 0 AND
        AltitudineGhiacciai <= 5),
        AltitudineGhiacciaiNote varchar(256),
        MassaGhiacciai int NOT NULL CHECK (MassaGhiacciai >= 0 AND MassaGhiacciai <=
        5),
        MassaGhiacciaiNote varchar(256),

        PRIMARY KEY (GeonameID, NomeCentro, DataInserimento)
    );

```

## AREA MONITORATA

```

CREATE TABLE AreeMonitorateDaCentri (
    NomeCentro varchar(20) REFERENCES CentriMonitoraggio(NomeCentro) ON UPDATE
    CASCADE ON DELETE CASCADE,
    GeonameID char(7) REFERENCES CoordinateMonitoraggio(GeonameID) ON UPDATE
    CASCADE ON DELETE CASCADE,

    PRIMARY KEY (NomeCentro, GeonameID)
);

```

## 4.4. QUERY

### Aggiungere un'area di interesse

```

INSERT INTO coordinateMonitoraggio (GeonameID, NomePaese, AsciiPaese, CodiceStato,
NomeStato, Latitudine, Longitudine);

```

### Aggiungere operatore

```

INSERT INTO OperatoriRegistrati (userid, nome, cognome, codfisc, email, password,
nomecentro)
    VALUES (?, ?, ?, ?, ?, ?, ?);

```

### Aggiungere centro di monitoraggio

```

INSERT INTO CentriMonitoraggio (nomecentro, indirizzo, numerocivico, cap, comune,
provincia)
    VALUES (?, ?, ?, ?, ?, ?);

```

### Aggiungere una rilevazione

```

INSERT INTO ParametriClimatici (geonameid,nomecentro, datainserimento, vento,
ventonote, umidita, umiditanote, pressione, pressionenote, temperatura,
temperaturanote, precipitazioni, precipitazioninote, altitudineghiacciai,
altitudineghiacciainote, massaghiacciai, massaghiacciainote, )
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);

```

### Aggiungere area monitorata da un centro

```

INSERT INTO AreeMonitorateDaCentri (geonameid, nomecentro)
    VALUES (?, ?);

```

### Ottenere tutti paesi

```
SELECT * FROM CoordinateMonitoraggio
```

### Ottenere tutti paesi dato il nome

```
SELECT * FROM CoordinateMonitoraggio WHERE asciipaese ILIKE ?
```

### Ottenere tutti paesi dato il codice dello stato

```
SELECT * FROM CoordinateMonitoraggio  
WHERE CodiceStato = ?
```

### Ottenere tutti paesi dato il nome e il codice dello stato

```
SELECT *  
FROM CoordinateMonitoraggio  
WHERE UPPER(asciipaese) = ? AND CodiceStato = ?
```

### Ottenere un operatore dato l'userID

```
SELECT * FROM OperatoriRegistrati  
WHERE userid = ?
```

### Ottenere i punti di interesse associati ad un centro

```
SELECT CoordinateMonitoraggio.* FROM CoordinateMonitoraggio NATURAL JOIN  
AreeMonitorateDaCentri NATURAL JOIN CentriMonitoraggio  
WHERE NomeCentro = ?
```

### Aggiornare il centro di monitoraggio associato ad un operatore

```
UPDATE OperatoriRegistrati  
SET NomeCentro = ?  
WHERE UserID = ?
```

### Ottenere un paese associato a un centro di monitoraggio

```
SELECT CoordinateMonitoraggio.*  
FROM AreeMonitorateDaCentri NATURAL JOIN CoordinateMonitoraggio  
WHERE AreeMonitorateDaCentri.NomeCentro = ? AND  
UPPER(CoordinateMonitoraggio.AsciiPaese) = ? AND CoordinateMonitoraggio.CodiceStato  
= ?
```

### Ottenere i centri di monitoraggio

```
SELECT * FROM CentriMonitoraggio
```

### Contare i centri di monitoraggio presenti con lo stesso nome (massimo 1)

```
SELECT COUNT(*) FROM CentriMonitoraggio  
WHERE NomeCentro = ?
```

### Ottenere tutte le rilevazioni su un'area di interesse

```
SELECT * FROM ParametriClimatici NATURAL JOIN AreeMonitorateDaCentri  
WHERE GeonameID = ?
```

## 5. INTEGRAZIONE DEL DATABASE NEL PROGETTO

In questa sezione viene illustrato come il database viene integrato nel progetto, sviluppato con Maven.

### 5.1. IL POM (Project Object Model)

Ogni progetto Maven contiene il file *pom.xml*.

Questo file è essenziale per la gestione dei progetti Maven e serve a diversi scopi chiave:

1. Definizione

contiene tutte le informazioni riguardanti il progetto, come il suo nome, la versione, l'autore e la descrizione.

2. Gestione delle Dipendenze

Specifica le librerie e i framework necessari per il progetto, consentendo a Maven di scaricare e includere automaticamente le dipendenze corrette durante il processo di build.

3. Configurazione dei Plugin

Permette di configurare plugin Maven che estendono le funzionalità del build, come la compilazione del codice, la creazione di pacchetti e l'esecuzione dei test.

4. Gestione dei Cicli di Vita

Definisce e personalizza il ciclo di vita del progetto, che comprende fasi come la compilazione, il test e il packaging.

5. Ereditarietà e Aggregazione

Supporta la creazione di progetti multi-modulo e la gestione di progetti complessi tramite ereditarietà e aggregazione, facilitando la gestione di più moduli all'interno di un singolo progetto.

Questo file è dunque essenziale per configurare il nostro progetto e inizializzare il database creando le tabelle e inserendo i dati fondamentali per utilizzare l'applicazione.

### 5.2. CREAZIONE DEL DATABASE TRAMITE XML

Il codice XML utilizzato è il seguente:

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <!-- La versione del modello POM -->
  <modelVersion>4.0.0</modelVersion>
  <!-- Identificatore unico del progetto -->
  <groupId>com.sample</groupId> <!-- Gruppo o organizzazione a cui appartiene il progetto -->
  <artifactId>sample</artifactId> <!-- Nome dell'artefatto (prodotto del build) -->
  <version>1.0-SNAPSHOT</version> <!-- Versione del progetto -->
  <packaging>jar</packaging>

  <dependencies>
    <!-- Dipendenza per il driver PostgreSQL -->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.7.3</version>
    </dependency>

    <!-- Dipendenza per l'AbsoluteLayout (una libreria esterna di NetBeans) -->
    <dependency>
      <groupId>org.netbeans.external</groupId>
      <artifactId>AbsoluteLayout</artifactId>
      <version>RELEASE210</version>
    </dependency>

    <!-- Dipendenza per la libreria bcrypt (per la crittografia delle password) -->
    <dependency>
      <groupId>org.mindrot</groupId>
      <artifactId>jbcrypt</artifactId>
      <version>0.4</version>
    </dependency>

    <!-- Dipendenza per JUnit (framework di testing) solo per la fase di test -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>

    <!-- Dipendenza per le annotazioni di JetBrains -->
    <dependency>
      <groupId>org.jetbrains</groupId>
      <artifactId>annotations</artifactId>
      <version>13.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Plugin per eseguire comandi SQL durante la build e inizializzare il database -->
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>sql-maven-plugin</artifactId>
        <version>1.5</version>

        <dependencies>

          <!-- Dipendenza per il driver PostgreSQL per il plugin SQL -->
          <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>

```

```

    <version>42.7.3</version>
  </dependency>
</dependencies>

<configuration>
  <driver>org.postgresql.Driver</driver> <!-- Driver JDBC per PostgreSQL -->
  <url>jdbc:postgresql://localhost:5432</url> <!-- URL di connessione al database -->
  <settingsKey>sensibleKey</settingsKey> <!-- Chiave per le impostazioni -->
</configuration>

<executions>

  <!-- Esecuzione per eliminare il database esistente -->
  <execution>
    <id>drop database</id>
    <phase>compile</phase> <!-- Fase della build in cui eseguire il comando -->
    <goals>
      <goal>execute</goal> <!-- Obiettivo del plugin -->
    </goals>
    <configuration>
      <url>jdbc:postgresql://localhost:5432</url>
      <autocommit>true</autocommit> <!-- Abilita l'auto-commit per le operazioni -->
      <driver>org.postgresql.Driver</driver>
      <username>postgres</username> <!-- Nome utente per il database -->
      <password>root</password> <!-- Password per il database -->
      <sqlCommand>drop database if exists climatemonitoring</sqlCommand> <!-- Comando
SQL da eseguire -->
    </configuration>
  </execution>

  <!-- Esecuzione per creare il database -->
  <execution>
    <id>create-database</id>
    <phase>compile</phase>
    <goals>
      <goal>execute</goal>
    </goals>
    <configuration>
      <url>jdbc:postgresql://localhost:5432</url>
      <autocommit>true</autocommit>
      <driver>org.postgresql.Driver</driver>
      <username>postgres</username>
      <password>root</password>
      <sqlCommand>create database climatemonitoring</sqlCommand>
    </configuration>
  </execution>

  <!-- Esecuzione per creare le tabelle nel database -->
  <execution>
    <id>create-tables</id>
    <phase>compile</phase>
    <goals>
      <goal>execute</goal>
    </goals>
    <configuration>
      <url>jdbc:postgresql://localhost:5432/climatemonitoring</url>
      <autocommit>true</autocommit>
      <driver>org.postgresql.Driver</driver>
      <username>postgres</username>
      <password>root</password>
      <srcFiles>
        <srcFile>data/sql_commands.sql</srcFile> <!-- File SQL con i comandi per
creare le tabelle -->
      </srcFiles>
    </configuration>
  </execution>

```

```

        </executions>
    </plugin>

    <!-- Plugin per compilare il codice sorgente -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
            <source>1.8</source>
            <target>1.8</target>
            <encoding>UTF-8</encoding>
            <compilerArgs>
                <arg>-Xlint:-options</arg>
            </compilerArgs>
        </configuration>
    </plugin>

    <!-- Plugin per gestire le risorse del progetto -->
    <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
            <encoding>UTF-8</encoding>
        </configuration>
    </plugin>

    <!-- Plugin per creare pacchetti JAR con dipendenze incluse -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>3.3.0</version>
        <executions>

            <!-- Creazione di un pacchetto JAR per il client -->
            <execution>
                <id>make-client-assembly</id>
                <phase>package</phase>
                <goals>
                    <goal>single</goal>
                </goals>
                <configuration>
                    <finalName>clientCM</finalName> <!-- Nome finale del JAR del client -->
                    <appendAssemblyId>false</appendAssemblyId>
                    <archive>
                        <manifest>
                            <mainClass>client.frame.GestioneScelta</mainClass> <!-- Classe principale
dell'applicazione client -->
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <outputDirectory>${project.basedir}/Eseguibili</outputDirectory> <!-- Directory
di destinazione del JAR del client -->
                </configuration>
            </execution>

            <!-- Creazione di un pacchetto JAR per il client -->
            <execution>
                <id>make-server-assembly</id>
                <phase>package</phase>
                <goals>
                    <goal>single</goal>
                </goals>
                <configuration>

```

```

        <finalName>serverCM</finalName> <!-- Nome finale del JAR del server -->
        <appendAssemblyId>false</appendAssemblyId>
        <archive>
            <manifest>
                <mainClass>server.servermi.ServerFrame</mainClass> <!-- Classe principale
dell'applicazione server -->
            </manifest>
        </archive>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <outputDirectory>${project.basedir}/Eseguibili</outputDirectory> <!-- Directory
di destinazione del JAR del server -->
        </configuration>
    </execution>
</executions>
</plugin>

    <!-- Plugin per generare la documentazione Javadoc -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.8.0</version>
        <executions>
            <execution>
                <goals>
                    <goal>javadoc</goal>
                </goals>
                <phase>prepare-package</phase>
            </execution>
        </executions>
    </plugin>

    <!-- Copia i file di Javadoc dalla directory di build alla directory specificata ed
elimina la cartella target -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.8</version>
        <executions>
            <execution>
                <phase>verify</phase>
                <goals>
                    <goal>run</goal>
                </goals>
                <configuration>
                    <target>
                        <copy todir="${project.basedir}/doc/javadoc">
                            <fileset dir="${project.build.directory}/site/apidocs"/>
                        </copy>
                        <delete dir="${project.build.directory}"/>
                    </target>
                </configuration>
            </execution>
        </executions>
    </plugin>

</plugins>
</build>
</project>

```

Il database viene prima eliminato con **DROP DATABASE** e poi viene ricreato con **CREATE DATABASE**, questo viene fatto per poter riconfigurare il database al suo stato iniziale con la possibilità di ricreare le tabelle e reinserire tutti i dati necessari

solamente con il seguente comando: `mvn clean install` oppure `./mvnw clean install`.

## 6. CREDITI

L'applicazione Climate Monitoring è stata sviluppata nell'ambito di un progetto software universitario.

*Università degli studi dell'Insubria  
Corso di Laurea (triennale) in Informatica  
Laboratorio Interdisciplinare B  
A.A. 2023 / 2024*

Autori:

- 753291 - Massini Riccardo
- 753216 - Abignano Luca
- 754696 - Artale Lorenzo