



**Università
degli Studi
di Ferrara**

SMART FARM

PROGETTO PER INGEGNERIA DEL SOFTWARE AVANZATA

Dott. Furini Francesco
Dott. Zerbinati Alessandro

INDICE

- Descrizione del caso d'uso
- Specifiche da soddisfare
- Progettazione UML
- Implementazione
- Tecnologie Utilizzate
- Test
- Deployment

GESTIONE DI UNA FATTORIA INTELLIGENTE

Descrizione del caso d'uso

- L'applicativo desktop **Smart Farm** è stato progettato per permettere ad un'azienda agricola di medie-grandi dimensioni di gestire le varie aree produttive di cui dispone.
- L'applicativo è personalizzabile per gestire diversi **allevamenti** e **piantagioni**.
- Oltre a fungere da **gestionale**, l'obiettivo è quello di facilitare il monitoraggio del terreno tramite **sensori IoT** che rilevano diversi parametri, monitorare il sistema di irrigazione e la produzione generale.

GESTIONE DI UNA FATTORIA INTELLIGENTE

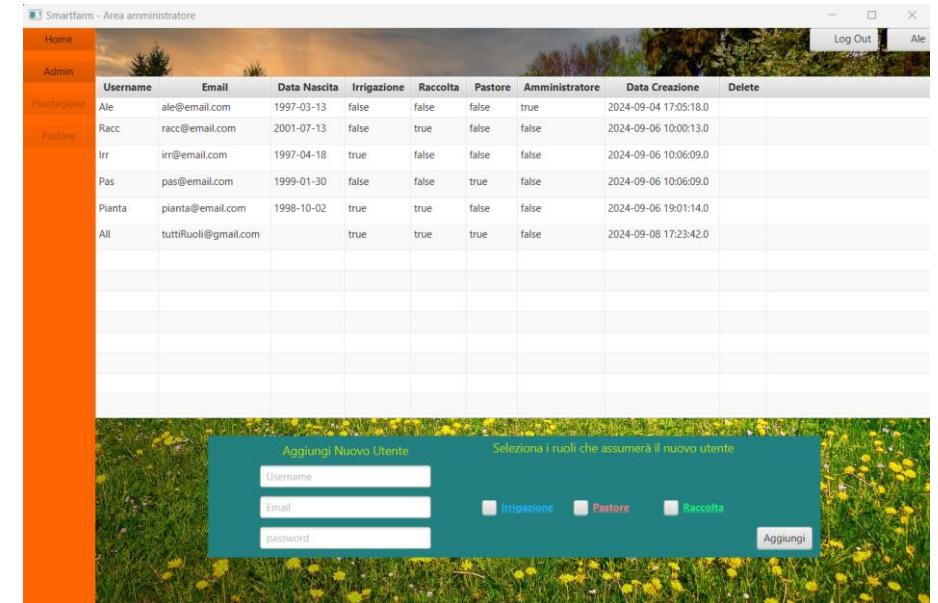
Funzionalità principali di alto livello

- **Login e area personale** – per l'accesso alle informazioni dell'account.
- **Admin page** – ruolo per il proprietario dell'azienda, in grado di gestire i propri dipendenti e assegnare ruoli.
- **Piantagione** page – permette di manipolare le piantagioni. Dalla pagina si accede ad altre due aree: gestione irrigazione e gestione zone.
- **Allevamento** page – permette di suddividere gli animali della fattoria in varie stalle e tenere traccia del rendimento dell'intera fattoria.

Specifiche - Admin

L'utente che accede alla pagina Admin deve poter:

- Visualizzare tutti gli utenti nel database;
- Aggiungere, modificare o eliminare un utente;
- Definire Username, Email e Password per un nuovo utente e attivarne l'area di competenza. I nuovi utenti riceveranno la password e lo username dall'admin e potranno modificarli al primo login.

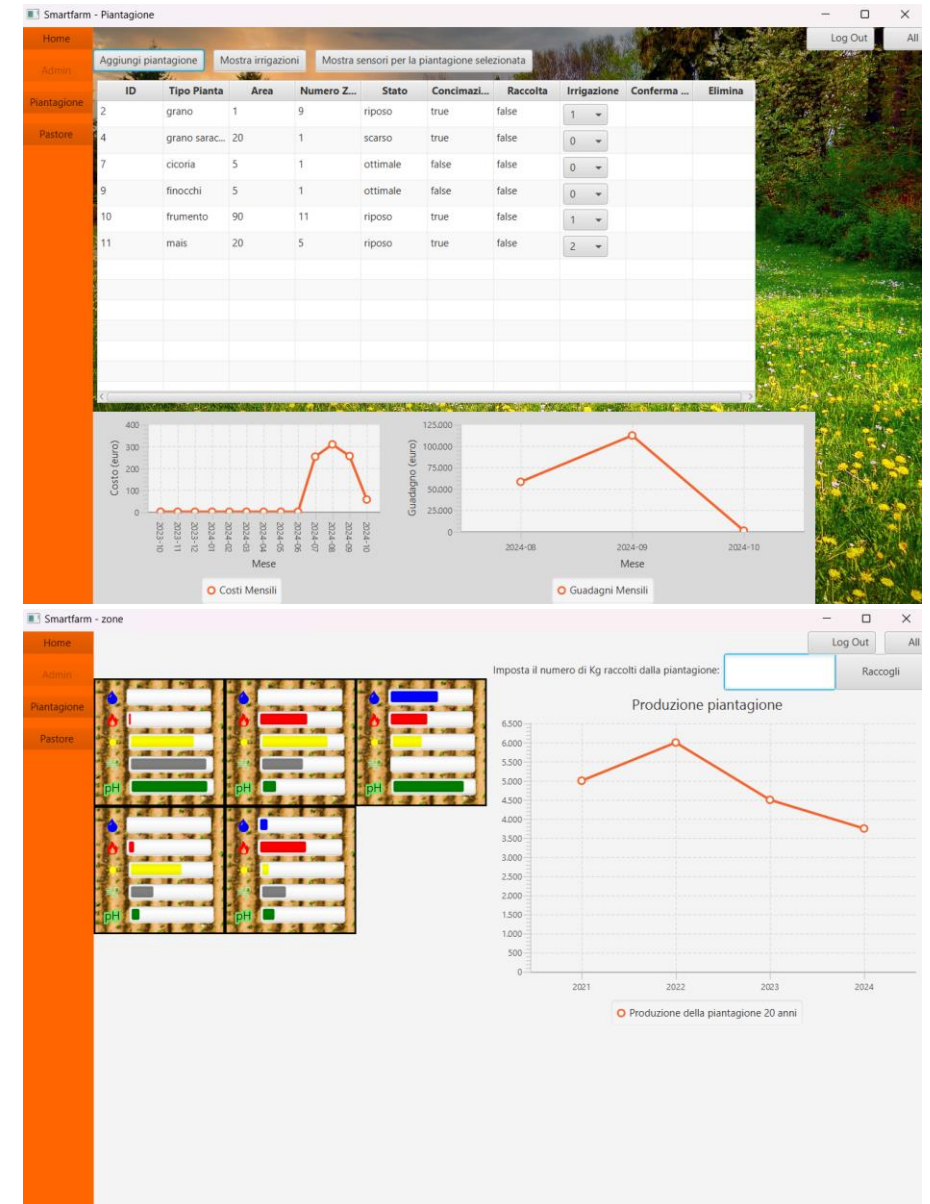


Questo meccanismo è pensato per la gestione di un'azienda con molti dipendenti, per mantenere il principio Least Privilege e separare le funzioni degli utenti.

Specifiche - Piantagione

L'utente che accede alla pagina Piantagione deve poter:

- Vedere la lista di tutte le piantagioni.
- Confrontare spese e guadagni tramite grafici.
- Visualizzare dettagli della singola piantagione: sensori presenti nelle zone.
- Configurare l'irrigazione automatica per una piantagione.



Specifiche - Allevamento

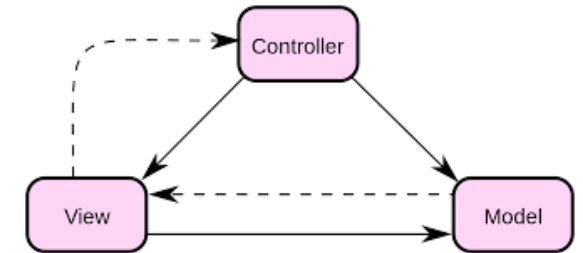
L'utente che accede alla pagina Allevamento deve poter:

- Visualizzare e gestire tutte le Stalle.
- Visualizzare costi e guadagni dell'ultimo anno prodotti dagli allevamenti.
- Automatizzare il consumo di mangime giornaliero.
- Visualizzare gli animali di una stalla e gestire visite veterinarie.
- Gestire le scorte di mangime nel magazzino.
- Gestire la produzione degli allevamenti.



Progettazione e UML

L'applicazione è stata progettata secondo il pattern **MVC** in modo top-down.



Partendo dallo schema **ER** entità-relazione si è ottenuta la definizione del database.

Le classi sono state generate nel seguente ordine di macrocategorie MVC:

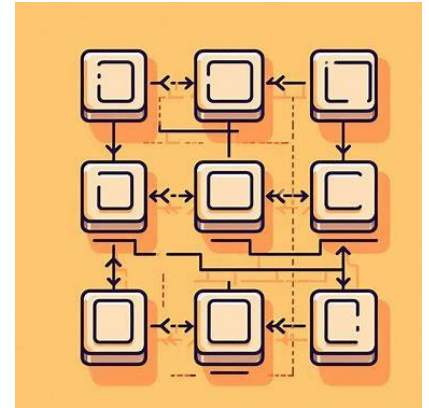
- **Model** – Lo schema del database viene mappato con le classi Model Object e l'accesso CRUD gestito all'interno delle classi DAO e DAO implementation.
- **View** – Per ogni pagina visualizzabile dell'applicazione si definisce un file fxm, contenente gli elementi grafici da gestire unicamente tramite il controller associato
- **Controller** – ogni view possiede un unico controller java che gestisce ogni evento e visualizzazione, accedendo ai metodi del DAO per recuperare o manipolare i dati dal database.

Questo pattern consente una separazione netta tra database, interfaccia utente e logica di business.

Progettazione e UML

Flusso dei dati per la gestione di una Stalla (esempio):

- Stalla.java – mappa i campi della tabella Stalla e fornisce i getter e setter
- DatabaseConnection.java – centralizza la connessione col database, utilizzata da DAOFactory in modo statico
- DAOFactory.java – centralizza le chiamate alle implementazioni dei DAO
- GenericDAO.java – interfaccia generica per tutti i DAO che definisce i metodi CRUD
- StallaDAO.java – interfaccia che estende GenericDAO definendo metodi specifici
- StallaDAOImpl.java – implementa l'interfaccia con metodi che accedono al database
- PastoreController.java – collega oggetti fxml con dati recuperati dai DAO.
- pastorePage.fxml – mostra gli elementi grafici a schermo
- MainApp.java – entry point dell'applicazione che porta alla Home.



Implementazione (Model Object)

Stalla.java

Mappatura 1:1 con la tabella MySQL omonima. Classe POJO contenente i metodi getter e setter per definire gli oggetti usati in StallaDAOImpl.




Table Name: Schema: **fattoria**

Charset/Collation: Engine:

Comments:






Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
 etichetta_stalla	VARCHAR(13)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'struttura_A'
 capienza	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 razza	VARCHAR(15)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 ora_pranzo	TIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 ora_cena	TIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Tabella MySQL di stalla

Variabili private

```
src > main > java > com > unife > project > model > mo > J Stalla.java > Stalla
1  package com.unife.project.model.mo;
2
3  import java.time.LocalDateTime;
4
5  public class Stalla {
6      private String etichettaStalla;
7      private int capienza;
8      private String razza;
9      private LocalDateTime oraPranzo;
10     private LocalDateTime oraCena;
11
12     public Stalla(){
13     }
14
```

```
public String getEtichettaStalla() {
    return etichettaStalla;
}

public void setEtichettaStalla(String etichettaStalla) {
    this.etichettaStalla = etichettaStalla;
}
```

Metodi getter e setter

Implementazione (Data Access Object)

StallaDAOImpl.java

```
//costruttore, utilizzato da DAOFactory
public StallaDAOImpl(Connection connection) {
    this.connection = connection;
}

@Override
public void save(Stalla stalla) {
    String sql = "INSERT INTO stalla (etichetta_stalla, capienza, razza, ora_pranzo, ora_cena) " +
        "VALUES (?, ?, ?, ?, ?)";

    try (PreparedStatement ps = connection.prepareStatement(sql)) {
        ps.setString(parameterIndex:1, stalla.getEtichettaStalla());
        ps.setInt(parameterIndex:2, stalla.getCapienza());
        ps.setString(parameterIndex:3, stalla.getRazza());
        ps.setObject(parameterIndex:4, stalla.getOraPranzo());
        ps.setObject(parameterIndex:5, stalla.getOraCena());
        int rowsInserted = ps.executeUpdate();
        if (rowsInserted > 0) {
            System.out.println(x:"Una nuova stalla è stata registrata correttamente!");
        }
    } catch (SQLException e){
        e.printStackTrace();
        System.out.println(x:"Errore nella registrazione di una stalla");
    }
}
```

In questo snippet viene mostrato il metodo costruttore della classe, sfruttato da DAOFactory per aprire la connessione al database.

Uno dei metodi implementati è 'save'. L'Override viene richiesto dall'interfaccia GenericDAO, estesa con StallaDAO.

Questo metodo esegue un inserimento dentro il database tramite la query INSERT INTO (*) VALUES (*).

Il PreparedStatement inserisce i valori presi da dentro l'oggetto stalla passato come argomento nella chiamata di funzione.

Viene eseguita la query SQL o lanciata un'eccezione.

Implementazione (Controller)

PastoreController.java

Definisce il comportamento di ogni elemento FXML.

Nello snippet un'estratto del controller che ridefinisce il metodo 'initialize()', chiamato automaticamente al caricamento della pagina.

Qui si preparano le colonne di una tabella per mostrare i dati di Stalla.

Sempre all'interno di initialize() è impostata la chiamata al metodo 'loadStalleData()' che viene mostrato nella terza immagine.

Questo metodo sfrutta DAOFactory per accedere al database e recuperare tutti i record di Stalla, indirettamente.

Il risultato viene salvato in una List<Stalla> 'stalleData' privata che viene poi usata come argomento per il metodo 'setItems()' una volta che loadStalleData() è stata eseguita.

```
@FXML
private void initialize() {
    // Inizializza le colonne della tabella
    nomeColumn.setCellValueFactory(new PropertyValueFactory<>(property:"etichettaStalla"));
    capienzaColumn.setCellValueFactory(new PropertyValueFactory<>(property:"capienza"));
    razzaColumn.setCellValueFactory(new PropertyValueFactory<>(property:"razza"));
    pranzoColumn.setCellValueFactory(new PropertyValueFactory<>(property:"oraPranzo"));
    cenaColumn.setCellValueFactory(new PropertyValueFactory<>(property:"oraCena"));
```

```
// Carica i dati delle stalle dal database
loadStalleData();
stalleTable.setItems(stalleData);
```

```
// Carica i dati dei costi e dei guadagni
loadCostiData();
loadGuadagniData();
```

```
// Aggiungi la colonna del pulsante di conferma
addConfirmButtonToTable();
```

```
// Aggiungi la colonna del pulsante "goTo"
addGoToButtonToTable();
```

```
private void loadStalleData() {
    // Carica i dati delle piantagioni dal database e impostali nella tabella
    stalleData.clear();
    List<Stalla> stalle = DAOFactory.getStallaDAO().findAll();

    stalleData.addAll(stalle);
}
```

Implementazione (Entry Point)

MainApp.java

```
public class MainApp extends Application {  
  
    private SensorSimulator sensorSimulator;  
  
    @Override  
    public void start(Stage primaryStage) {  
        try {  
  
            // Inizializza e attiva il SensorSimulator  
            sensorSimulator = new SensorSimulator();  
            sensorSimulator.start();  
  
            // Carica il file FXML della schermata di home  
            FXMLLoader loader = new FXMLLoader(getClass().getResource(name="/com/unife/project/view/home.fxml"));  
            Parent root = loader.load();  
  
            // Imposta la scena  
            Scene scene = new Scene(root);  
            WindowUtil.setWindow(primaryStage, scene, title:"Smartfarm - Home");  
            primaryStage.show();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- start () : prima di avviare l'app lancia uno script timerizzato che simula ogni 5 minuti una rilevazione di ogni sensore attivo nel database, come se avesse dati in tempo reale. Poi carica il file 'home.fxml' e il suo controller, mostrando quindi la pagina di benvenuto dove è presente il login. Start viene chiamato in automatico all'entrata dell'app.
- Stop() : viene chiamato in automatico alla chiusura dell'app. si occupa di chiudere le connessioni con il database e ferma anche lo script che simula i sensori.

Tecnologie utilizzate



Docker per la containerizzazione dell'app; **DockerHub** per ospitare le immagini.



JUnit e **Mockito** come framework per il testing.



Java V11 come linguaggio di programmazione principale.



Maven per il building e la gestione delle dipendenze.



JavaFX per la definizione di un'interfaccia grafica; **VcxSrv** per abilitare JavaFx dal container.



GitHub come Version Control System e repository remoto; **GitHub Actions** come gestore della pipeline CI/CD.

Test

Ecco un esempio dei test che sono stati effettuati, in questo caso si sta controllando che il metodo save della classe StallaDAO consenta l'inserimento nella tabella stalla di un nuovo record, prendendo le informazioni dall'oggetto omonimo appena creato.

Il metodo setUp con l'annotazione di JUnit Jupiter @BeforeEach permette di simulare la connessione al database prima di ogni test.

```
@InjectMocks
private StallaDAOImpl stallaDAO;

//private Stalla stalla;

@BeforeEach
public void setUp() throws SQLException {
    MockitoAnnotations.openMocks(this);
    when(connection.prepareStatement(anyString())).thenReturn(preparedStatement);
}

@Test
public void testSave() throws SQLException {
    Stalla stalla = new Stalla();
    stalla.setEtichettaStalla(etichettaStalla:"ovini-1");
    stalla.setCapienza(capienza:40);
    stalla.setRazza(razza:"ovino-merino");
    stalla.setOraPranzo(LocalTime.of(hour:11,minute:45));
    stalla.setOraCena(LocalTime.of(hour:21,minute:10));

    when(preparedStatement.executeUpdate()).thenReturn(value:1);

    stallaDAO.save(stalla);

    verify(connection, times(wantedNumberOfInvocations:1)).prepareStatement(
        sql:"INSERT INTO stalla (etichetta_stalla, capienza, razza, ora_pranzo, ora_cena) VALUES (?, ?, ?, ?, ?)"
    );

    verify(preparedStatement, times(wantedNumberOfInvocations:1)).setString(parameterIndex:1, stalla.getEtichettaStalla());
    verify(preparedStatement, times(wantedNumberOfInvocations:1)).setInt(parameterIndex:2, stalla.getCapienza());
    verify(preparedStatement, times(wantedNumberOfInvocations:1)).setString(parameterIndex:3, stalla.getRazza());
    verify(preparedStatement, times(wantedNumberOfInvocations:1)).setObject(parameterIndex:4, stalla.getOraPranzo());
    verify(preparedStatement, times(wantedNumberOfInvocations:1)).setObject(parameterIndex:5, stalla.getOraCena());
    verify(preparedStatement, times(wantedNumberOfInvocations:1)).executeUpdate();
}
```

Test

```
39
40     //private Stalla stalla;
41
42     @BeforeEach
43 > public void setUp() throws SQLException { ...
47
48     @Test
49 > public void testSave() throws SQLException { ...
73
74     @Test
75 > public void testUpdate() throws SQLException { ...
96
97     @Test
98 > public void testDelete() throws SQLException { ...
119
120
121     @Test
122 > public void testFindByEtichetta() throws SQLException { ...
157 }
```

Complessivamente sono stati realizzati ed eseguiti più di 60 test sui metodi CRUD implementati nelle classi DAOImpl, come quello appena visto. Per ottenerli è stato utilizzato il framework Mockito, con i suoi metodi che hanno permesso di simulare la connessione al database e di verificare il comportamento dei vari metodi utilizzati nei Data Access Object(verify), inoltre è stato utilizzato il framework Junit già citato nella pagina precedente.

Deployment

- Per il deployment dell'applicazione sono stati utilizzati DockerHub e Github Actions, che hanno permesso di costruire una pipeline di Continuous Integration e Continuous Deployment.
- La pipeline permette di testare automaticamente l'applicazione ad ogni push sul branch main e di rilasciare una versione aggiornata delle immagini docker dell'applicazione.



Deployment

- La pipeline è stata ottenuta partendo dal template Github Actions "Java CI with Maven" per ottenere il file di configurazione maven.yml ed apportando alcune modifiche. Per adattarla alla nostra applicazione, composta da 2 immagini (app e db).

```
.github > workflows > maven.yml
20 jobs:
21   ci:
26     - name: Checkout code
27       uses: actions/checkout@v4
28
29     - name: Set up JDK 11
30       uses: actions/setup-java@v4
31       with:
32         java-version: '11'
33         distribution: 'temurin'
34         cache: maven
35
36     - name: Build with Maven
37       run: mvn -B package --file pom.xml
38
39     - name: Verify with Maven
40       run: mvn verify
41
42     #CD: Deploy the application on docker
43
44     - name: Docker login
45       if: github.ref == 'refs/heads/main'
46       env:
47         DOCKER_HUB_USERNAME: ${ secrets.DOCKER_HUB_USERNAME_ALE
48         DOCKER_HUB_PASSWORD: ${ secrets.DOCKER_HUB_PASSWORD_ALE
49       run: echo $DOCKER_HUB_PASSWORD | docker login --username $D
50
51     - name: Build docker image
52       if: github.ref == 'refs/heads/main'
53       run: |
54         docker build --pull --rm -f "Dockerfile.app" -t blue5
55         docker build --pull --rm -f "Dockerfile.db" -t blue56
56
57     - name: Checkout code
58       uses: actions/checkout@v4
59
60     - name: Push docker image
61       if: github.ref == 'refs/heads/main'
62       run: |
63         docker image push blue563/smartfarm2:app
64         docker image push blue563/smartfarm2:db
65
```

Deployment

Per costruire le 2 immagini docker del server mysql con il database (db) e dell'applicazione (app) sono stati utilizzati due dockerfile:

Dockerfile.app, dove tramite maven:

- vengono impostati la versione di java con cui compilare l'applicazione,
- vengono copiate le dipendenze dell'applicazione nel container,
- si ottiene l'eseguibile jar dell'applicazione per poi copiarlo nel container.
- Inoltre vengono installate alcune librerie necessarie per collegare l'applicazione al server X11, necessario per l'utilizzo della GUI di javaFX tramite container, ma anche il client mysql per permettere la connessione con il container del database.

```
Dockerfile.app > ...
1 FROM maven:3.8-openjdk-11-slim AS build
2
3 WORKDIR /app
4
5
6 COPY pom.xml .
7
8 RUN mvn dependency:resolve
9 RUN mvn dependency:copy-dependencies
10
11 COPY src ./src
12 RUN mvn package
13
14 FROM openjdk:11-jdk-slim
15 WORKDIR /app
16
17 RUN apt-get update && apt-get install -y \
18     libgl1-mesa-glx \
19     libgl1-mesa-dri \
20     libx11-6 \
21     libxext6 \
22     libxrender1 \
23     libgtk-3-0 \
24     libxtst6 \
25     libxi6 \
26     libfreetype6 \
27     default-mysql-client \
28     && rm -rf /var/lib/apt/lists/*
29
30 COPY target/smart-farm-management-1.0-SNAPSHOT.jar /app.
31
32
33 COPY --from=build /app/target/dependency/* ./libs/
```

Deployment

- Dockerfile.db, in cui vengono impostate la versione di mysql utilizzata e le credenziali per accedervi e in cui viene caricato un dump del database locale.

```
✓ Dockerfile.db > ...  
1 FROM mysql:8.1.0  
2 ENV MYSQL_ROOT_PASSWORD=Pannocchie98!?  
3 ENV MYSQL_DATABASE=fattoria  
4 COPY initdb/Dump4Docker.sql /docker-entrypoint-initdb.d/|
```

Deployment

- Per testare manualmente l'utilizzo dell'applicazione tramite docker è stato inoltre utilizzato Docker-compose, che permette di effettuare il build e l'avvio di più container in modo semplice e compatto.
- In particolare sono stati utilizzati i comandi docker-compose build per ottenere le immagini e docker-compose up e docker-compose down -v per avviare e fermare i container.

```
docker-compose.yml
1  services:
2    app:
3      image: 'blue563/smart_farm_2_dock_file/your-image:app'
4      build:
5        context: .
6        dockerfile: Dockerfile.app
7      depends_on:
8        - db
9      environment:
10        DISPLAY: host.docker.internal:0
11        MYSQL_HOST: db
12        MYSQL_PORT: 3306
13        MYSQL_DATABASE: fattoria
14        MYSQL_USER: root
15        MYSQL_PASSWORD: Pannocchie98!?
16        MYSQL_ROOT_PASSWORD: Pannocchie98!?
17      command: sh -c "sleep 20s; echo avvio ; java --module-path ./lib
18      volumes:
19        - /tmp/.X11-unix:/tmp/.X11-unix
20      networks:
21        - app-network
22
23    db:
24      image: 'blue563/smart_farm_2_dock_file/your-image:db'
25      build:
26        context: .
27        dockerfile: Dockerfile.db
28      environment:
29        MYSQL_ROOT_PASSWORD: ${ secrets.PASSWORD }
30        MYSQL_DATABASE: fattoria
31      ports:
32        - "3306:3306"
33      volumes:
34        - .\mysql:/var/lib/mysql
35      networks:
36        - app-network
37
38  networks:
39    app-network:
40      driver: bridge
```



grazie per l'attenzione