**1. Project Overview**

**Project Name:** Heart Disease Prediction System  
**Architecture:** 3-Tier (Presentation Layer, Business Logic Layer, Data Access Layer)  
**Technology Stack:** Java, Spring Boot, MySQL, AWS  
**Machine Learning Model:** Logistic Regression, Random Forest (Trained in Python and saved as a serialized model)

2. Project Structure

heart-disease-prediction/

│

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ ├── com/

│ │ │ │ ├── heartdisease/

│ │ │ │ │ ├── controller/

│ │ │ │ │ ├── service/

│ │ │ │ │ ├── model/

│ │ │ │ │ ├── repository/

│ │ │ │ │ ├── config/

│ │ │ │ │ └── HeartDiseasePredictionApplication.java

│ │ ├── resources/

│ │ │ └── application.properties

├── data/

│ ├── heart-disease.csv

├── model/

│ └── heart\_model.pkl

├── pom.xml

└── README.md

**3. Implementation**

**Step 1: Data Preparation and Model Training**

First, we train a machine learning model in Python and save it as a serialized model (heart\_model.pkl) that will be used in our Java application.

**Step 2: Presentation Layer (Frontend)**

This layer will be a simple web interface using Spring Boot and Thymeleaf.

**Step 3: Business Logic Layer (Backend)**

This layer will contain the logic to load the model, accept user input, and return predictions.

**Step 4: Data Access Layer (DAL)**

In this case, we assume that no direct database interaction is required since the model is used for predictions directly. However, if needed, you can add a database layer to store user inputs, logs, or results.

**Step 5: Configuration**

**Step 1: Create the Spring Boot Project**

You can create the project using the Spring Initializr or manually with Maven/Gradle.

1. Go to [Spring Initializr](https://start.spring.io/).
2. Fill in the project details:
   * **Group:** com.heartdisease
   * **Artifact:** heart-disease-prediction
   * **Name:** heart-disease-prediction
   * **Dependencies:** Spring Web, Spring Data JPA, MySQL Driver, Lombok, Thymeleaf, Spring Boot DevTools
3. Generate the project and unzip it.

**Step 2: Define the Model**

Create a Patient entity to represent the data model.

**Step 3: Define the Repository**

Create a Patient Repository interface to handle database operations.

**Step 4: Define the Service Layer**

Create a Heart Disease Service class for business logic.

**Step 5: Define the Controller Layer**

Create a HeartDiseaseController class to handle HTTP requests.

**Step 6: Create the View**

Create a basic HTML form using Thymeleaf for user input in the templates/index.html.

**Step 7: Configure Database in application.properties**

Add your database connection details in the application.properties file.

**Step 8: Run the Application**

Run the application by executing the HeartDiseasePredictionApplication.java file.

Step 9: Integration with Machine Learning Model

The code above assumes that you have an existing trained machine learning model (likely a .pkl file from your Python code). To integrate this model with your Java application, you can use a service like AWS Lambda or host it as a REST API on AWS or Azure and call it from your Java service.

Alternatively, you can use a Java-based library like Deep Java Library (DJL) or TensorFlow for Java if you want to directly use the model in your application.

Step 10: Deploying to Cloud

You can deploy the Spring Boot application to AWS or Azure using their respective services like Elastic Beanstalk (AWS) or Azure App Service.

AWS Elastic Beanstalk: Upload your .jar file directly.

Azure App Service: You can deploy using the Maven plugin for Azure or using the Azure portal.

**1. Using Deep Java Library (DJL)**

**Deep Java Library (DJL)** is a deep learning library for Java that allows you to load and run pre-trained models directly in Java applications. It supports multiple deep learning frameworks, including TensorFlow, PyTorch, and MXNet.

**Step 1: Add DJL Dependencies**

First, add the necessary dependencies to your pom.xml file.

**Step 2: Load the Pre-trained Model**

You can load a pre-trained model (e.g., TensorFlow SavedModel) into your Java application using DJL.

In this example:

* The loadModel() method loads the model from a specified path.
* The predictHeartDisease() method takes the input features as a float array and runs the prediction.

You would replace "path/to/your/model" with the actual path where your TensorFlow SavedModel or other model types are located.

**Step 3: Use the Service in Controller**

In your controller, you can now call the HeartDiseaseService to get predictions.

**2. Using TensorFlow for Java**

TensorFlow for Java allows you to use TensorFlow models directly within your Java application.

**Step 1: Add TensorFlow Dependencies**

Add TensorFlow dependencies to your pom.xml:

**Step 2: Load the TensorFlow Model**

Load a TensorFlow SavedModel in Java.

**Step 3: Use the Service in Controller**

Use the service in your controller to perform predictions just like in the DJL example.

**Deployment**

Once your application is running locally, you can deploy it to the cloud (AWS, Azure) using services like AWS Elastic Beanstalk, AWS EC2, or Azure App Services. Simply package your application as a .jar file and deploy it using the cloud platform's deployment tools.

**Conclusion**

DJL and TensorFlow for Java allow you to bring machine learning capabilities into your Java applications by directly loading and using models in the Java environment. This approach is highly scalable and integrates well with enterprise applications, offering a seamless way to use AI in your Java projects.