**PHASE 1: Setup & Plan (Day 1)**

**1. Create a GitHub Repo**

* **Action:** Set up a GitHub repository to version-control your project. This will help track progress, collaborate, and manage the code.
* **Task:** Go to GitHub, create a new repository, and name it something like "Stock Price Movement Predictor."

**2. Decide on Project Folder Structure**

* **Action:** Structure your project with separate folders for Java and Python. A simple structure could look like this:

/Stock-Price-Predictor

/java-backend

/python-ml-api

/docs

* **Task:** Create the folder structure locally and commit it to the GitHub repository.

**3. Write README**

* **Action:** Write the README for your repository to explain the project.
* **Contents to Include:**
  + **Project Goal:** "The project aims to predict whether a stock will go up or down the next day based on historical data using machine learning."
  + **Tech Stack:** List technologies used for both backend (Java/Spring Boot) and the ML model (Python/FastAPI).
  + **Planned Features:** Include features like stock prediction, stock price graph, storing user searches, etc.

**4. Install Postman**

* **Action:** Postman is a tool to test your API endpoints, both Java and Python.
* **Task:** Download and install Postman on your system to test APIs later in the project.

**PHASE 2: Java Backend (Day 1)**

**1. Set Up Spring Boot Project**

* **Action:** Use Spring Initializr to create a new Spring Boot project with dependencies like Spring Web and Spring Security (if you plan to add authentication later).
* **Task:** Initialize a Spring Boot project, download it, and import it into your IDE (e.g., IntelliJ IDEA, Eclipse).

**2. Create REST Endpoint /predict**

* **Action:** In your Spring Boot app, create a controller with an endpoint /predict that will be responsible for receiving the stock ticker and forwarding the request to your Python FastAPI service.
* **Task:** Set up a basic controller in Java with a POST method that accepts JSON input (e.g., ticker symbol) and forwards the request to the FastAPI service.

**3. Use RestTemplate or WebClient**

* **Action:** Use RestTemplate or WebClient to make HTTP calls from Java to the Python FastAPI service. This will allow you to send the stock ticker to Python and receive a prediction.
* **Task:** Set up a REST client in Java to make requests to the FastAPI /predict endpoint.

**4. Send Ticker to Python API**

* **Action:** When the /predict endpoint is called from Java, extract the ticker (e.g., AAPL) from the request and forward it to the Python API.
* **Task:** Ensure that you correctly pass the ticker data to FastAPI and handle the response (prediction result) in Java.

**5. Display Prediction in Java**

* **Action:** Return the prediction from the FastAPI service back to the client (either in JSON or HTML).
* **Task:** Make sure the frontend or API client can see the result: whether the stock price is predicted to go up or down.

**PHASE 3: ML API with FastAPI (Day 2)**

**1. Set Up Python Project**

* **Action:** Create a new directory for the FastAPI Python backend and set up a virtual environment using venv.
* **Task:** Initialize a Python project and install necessary libraries like fastapi, uvicorn, and yfinance.

**2. Install FastAPI + Uvicorn**

* **Action:** Install FastAPI to build the API and Uvicorn to serve it.
* **Task:** Run pip install fastapi uvicorn to get started.

**3. Install yfinance to Fetch Stock Data**

* **Action:** Use the yfinance library to fetch historical stock price data for the requested ticker.
* **Task:** Install yfinance by running pip install yfinance and set up the code to pull stock data for the provided ticker.

**4. Create Logistic Regression Model**

* **Action:** Use a simple logistic regression model with scikit-learn to predict whether the stock price will go up or down.
* **Task:** Write code to train the model on historical stock data (you can find historical stock data using yfinance), and use the trained model to predict the next day's movement.

**5. Expose /predict Endpoint**

* **Action:** Create a FastAPI POST endpoint /predict that takes the stock ticker as input and returns the model’s prediction.
* **Task:** Set up the FastAPI endpoint, accept ticker input, make the prediction using your ML model, and return the result (prediction and confidence).

**PHASE 4: Java ↔ Python Integration (Day 2)**

**1. Ensure FastAPI is Running Locally**

* **Action:** Start your FastAPI server locally on port 8000 (e.g., uvicorn main:app --reload).
* **Task:** Make sure the FastAPI app is running and accessible at http://localhost:8000.

**2. Test Java to Python Integration**

* **Action:** From your Spring Boot backend, make sure you're able to send a request to the FastAPI /predict endpoint with a stock ticker and get a response back.
* **Task:** Test the integration to ensure that Java can successfully interact with FastAPI, get predictions, and display them in the response.

**PHASE 5: Bonus Features (Optional)**

**1. Show Stock Price Graph**

* **Action:** Use Chart.js (or Java-based solutions) to show a graphical representation of the stock price over time.
* **Task:** Integrate the stock price data visualization into your app, either through a frontend or in the Java backend.

**2. Store User Searches**

* **Action:** Use MongoDB or PostgreSQL to store user queries (e.g., stock ticker searches).
* **Task:** Set up a database to log the stock tickers users search for and integrate it with the Java backend.

**3. Deploy ML API Online**

* **Action:** Host the Python FastAPI service on platforms like Render, Railway, or Heroku to make it accessible online.
* **Task:** Deploy your FastAPI service on a platform and update the Java backend to call the deployed service instead of the local one.

**4. Add Authentication (Optional)**

* **Action:** If needed, implement authentication using Spring Security to protect API endpoints.
* **Task:** Configure Spring Security to restrict access to the /predict endpoint or other parts of your app.

**PHASE 6: Final Touches & Presentation (Day 3)**

**1. Prepare README with Screenshots**

* **Action:** Update the README file with clear documentation, including features, installation instructions, and a list of tools used. Add screenshots of the app in action.
* **Task:** Take screenshots of the application and include them in your README.

**2. Make a Demo Video**

* **Action:** Create a demo video to showcase how the app works.
* **Task:** Use Loom or OBS Studio to record a video showing how to interact with the app and demonstrate its functionality.

**3. Write About the Architecture**

* **Action:** Add a section in the README explaining the architecture of the project, including the flow of data between the Java backend and Python FastAPI.
* **Task:** Explain how the prediction works at a high level and provide details about the tech stack.