

✓ Final Project Title

Stock Trading Simulation Platform

(Advanced Java Application with Hibernate, OOP, Collections, Multithreading, Java 8, and Email Authentication)

✓ Features & Advanced Concepts to Include

- ✓ **Object-Oriented Design** – Users, Orders, Stocks, Portfolio
 - ✓ **Hibernate ORM** – With **One-to-One**, **One-to-Many**, **Many-to-One**, **Many-to-Many**
 - ✓ **Lazy Loading & Cascade Types**
 - ✓ **Java 8 Features** – Streams, Lambdas, Functional Interfaces
 - ✓ **Collections Framework** – Maps, Lists, Sets
 - ✓ **Multithreading** – Real-time stock price updater
 - ✓ **Custom Exceptions** – For business rules
 - ✓ **Email Authentication** – Using Java Networking (JavaMail API)
 - ✓ **Advanced Database Design** – With relationships and mapping
 - ✓ **Scalable Design** – Can later become a Spring Boot application
-

✓ Step 1: High-Level Algorithm

Main Workflow

1. Initialize Application

- Configure Hibernate (SessionFactory)
- Load initial stock data (from DB)
- Start PriceUpdater thread for real-time simulation

2. User Registration & Authentication

- Register user details
- Send **OTP via email** (Java Networking)
- Verify OTP before account activation

3. User Actions

- Login
- Main Menu:
 - View all stocks (with live prices)
 - Buy Stock
 - Sell Stock
 - View Portfolio
 - View Transaction History
 - Logout

4. Buy Stock

- Validate stock availability
- Check user balance
- Deduct amount + brokerage (Functional Interface)
- Update portfolio (Hibernate **One-to-Many**)
- Record transaction

5. Sell Stock

- Validate stock in portfolio
- Update balance
- Record transaction

6. Price Updater Thread

- Runs every 5 seconds
- Randomly adjusts stock prices $\pm 1\%$
- Uses synchronized collections or **ConcurrentHashMap**

7. Exit

- Commit all changes using Hibernate
- Stop background threads gracefully

✓ Step 2: Flowchart

pgsql

START



Load Stocks from DB ← (Hibernate)



Start Price Updater Thread

```

↓
User Login / Register
↓
(Email OTP Verification)
↓
Main Menu
├── View Stocks
├── Buy Stock
├── Sell Stock
├── View Portfolio
├── View Transactions
├── Logout
↓
Exit → Stop Threads → Close Hibernate Session

```

✓ Step 3: Package Structure

```

scss
com.manish07.trading
├── app
│   └── TradingApp.java           // Main entry point
├── config
│   └── HibernateUtil.java       // SessionFactory configuration
├── model
│   ├── User.java               // @Entity
│   ├── EmailVerification.java  // OTP entity (OneToOne with User)
│   ├── Stock.java              // @Entity
│   ├── Portfolio.java          // @Entity (OneToMany with Orders)
│   ├── Order.java              // Abstract class
│   ├── BuyOrder.java
│   ├── SellOrder.java
│   └── Transaction.java        // ManyToOne(User), ManyToOne(Stock)
├── dao
│   ├── UserDAO.java
│   ├── StockDAO.java
│   ├── OrderDAO.java
│   └── impl
│       ├── UserDAOImpl.java
│       ├── StockDAOImpl.java
│       └── OrderDAOImpl.java
├── service
│   ├── AuthService.java        // Handles email OTP
│   ├── TradingService.java     // Buy/Sell logic
│   ├── PortfolioService.java
│   └── PriceUpdaterService.java
├── exception
│   ├── InsufficientFundsException.java
│   ├── StockNotFoundException.java
│   └── InvalidOrderException.java
└── util
    └── EmailSender.java        // JavaMail API for OTP

```

✓ Step 4: Class Hierarchy

Model Layer

- **User**
 - @OneToOne with **EmailVerification**
 - @OneToMany with **Portfolio**
 - **EmailVerification**
 - Stores OTP and verification status
 - **Portfolio**
 - @ManyToOne User
 - @OneToMany Orders
 - **Order (abstract)**
 - Subclasses: **BuyOrder**, **SellOrder**
 - **Stock**
 - @ManyToMany with Order
 - **Transaction**
 - Links user and stock
-

✓ Step 5: Database Design & ER Diagram

Entities and Relationships

- **User (user_id PK)**
 - OneToOne → EmailVerification
 - OneToMany → Portfolio
 - **EmailVerification (id PK)**
 - user_id (FK)
 - **Stock (stock_id PK)**
 - ManyToMany with Order
 - **Portfolio (portfolio_id PK)**
 - ManyToOne → User
 - OneToMany → Orders
 - **Order (order_id PK)**
 - ManyToOne → Portfolio
 - ManyToMany → Stock
-

ER Diagram (Simplified)

```
sql

User  — 1 —> EmailVerification
|
1
|
Portfolio — 1 —> Order — N —> Stock
```

✓ Where Hibernate Fits

- All DAOs will use **Hibernate Session API**
- **HibernateUtil** will manage **SessionFactory**
- Use **CascadeType.ALL** and **FetchType.LAZY**
- Enable **Hibernate mappings** for relationships:
 - `@OneToOne(mappedBy="user")`
 - `@OneToMany(mappedBy="portfolio")`
 - `@ManyToMany(mappedBy="orders")`

✓ Algorithm for PriceUpdater Multithreading

1. Create **PriceUpdaterService** implements **Runnable**
2. Use **ScheduledExecutorService** or a **while(true)** loop with **Thread.sleep(5000)**
3. Lock the stock list using **synchronized** or use **ConcurrentHashMap**
4. For each stock:
 - Randomly adjust price by $\pm 1\%$
5. Notify any observers (optional)

✓ Detailed Class Responsibilities

- ✓ **User** – Holds personal details, portfolio
- ✓ **Portfolio** – Tracks user's stocks
- ✓ **Stock** – Represents market stock with live price
- ✓ **Order** – Buy/Sell operations
- ✓ **TradingService** – Core logic for buying/selling
- ✓ **AuthService** – OTP verification for user
- ✓ **PriceUpdaterService** – Multithreading for stock price changes
- ✓ **EmailSender** – Sends OTP via SMTP
- ✓ **DAO Layer** – Handles Hibernate persistence

✓ Development Roadmap

Phase 1: Setup

- Create Maven project, add Hibernate + JavaMail dependencies
- Configure `hibernate.cfg.xml` and `HibernateUtil.java`

Phase 2: Model + Mapping

- Create entities and apply annotations for relationships

Phase 3: DAO Layer

- Implement CRUD using Hibernate

Phase 4: Services

- Implement `TradingService`, `AuthService`, `PriceUpdaterService`

Phase 5: Multithreading

- Start price updater in a background thread

Phase 6: Email OTP

- Use JavaMail API to send OTP

Phase 7: Testing

- Test user registration, stock price update, buy/sell flow

✅ **Next Step:** Do you want me to **prepare a full ER diagram (in text format with table structure) + Hibernate mapping annotations for all relationships** OR should I **give you a detailed method-wise responsibility for each class in the package structure** first?

```
python
```

```
from reportlab.lib.pagesizes import A4
from reportlab.pdfgen import canvas
from reportlab.lib.units import inch

# File path
pdf_file_path = "/mnt/data/Stock_Trading_Simulation_Project_Details.pdf"

# Create canvas
c = canvas.Canvas(pdf_file_path, pagesize=A4)
width, height = A4

# Title
c.setFont("Helvetica-Bold", 16)
c.drawCentredString(width / 2.0, height - 40, "Stock Trading Simulation Project - Detailed Design")
```

```

c.setFont("Helvetica", 10)
text = c.beginText(40, height - 70)
text.setFont("Helvetica", 10)
text.setLeading(14)

# Content for PDF
content = ""
Project Title:
Stock Trading Simulation Platform

Key Features:
✓ OOP with Java
✓ Hibernate ORM with advanced mappings (OneToOne, OneToMany, ManyToOne, ManyToMany)
✓ Lazy Loading & Cascade Types
✓ Java 8 Features (Streams, Lambdas, Functional Interfaces)
✓ Collections Framework
✓ Multithreading for stock price updates
✓ Email Authentication (Java Networking)
✓ Custom Exceptions
✓ Database Design with ER diagram

High-Level Algorithm:
1. Initialize Hibernate, load stock data
2. Start PriceUpdater Thread
3. Register/Login User (with Email OTP verification)
4. Main Menu: View Stocks, Buy, Sell, Portfolio, Transactions
5. Buy/Sell: Validate, update portfolio, record transaction
6. Persist data using Hibernate
7. Exit safely

Flowchart:
START → Load Stocks → Start PriceUpdater → User Login/Register (OTP) → Main Menu → Exit

Package Structure:
com.manish07.trading
├── app
├── config
├── model
├── dao
│   └── impl
├── service
├── exception
└── util

Class Hierarchy:
- User → EmailVerification (OneToOne)
- User → Portfolio (OneToMany)
- Portfolio → Orders (OneToMany)
- Orders → Stocks (ManyToMany)

ER Diagram (Simplified):
User — 1 —> EmailVerification
User — 1 —> Portfolio —> Orders —> Stocks

Where Hibernate Fits:
- SessionFactory via HibernateUtil
- DAOs for CRUD
- CascadeType.ALL, FetchType.LAZY

Algorithm for PriceUpdater:
- Runnable service updates stock prices every 5s using ScheduledExecutorService

Development Roadmap:
Phase 1: Setup Maven, Hibernate
Phase 2: Create Models & Mapping
Phase 3: DAO Layer
Phase 4: Services (Trading, Auth, Email)
Phase 5: Multithreading for price updates
Phase 6: Email OTP using JavaMail
Phase 7: Testing
"""

text.textLines(content)
c.drawText(text)

```

```
c.showPage()
c.save()

pdf_file_path

Result
'/mnt/data/Stock_Trading_Simulation_Project_Details.pdf'
```

Your detailed project document has been generated. You can download it using the link below:

 [Download the PDF](#)

Do you want me to **expand this PDF further with ER diagram sketches, method responsibilities for each class, and a Gantt-style roadmap** to make it fully comprehensive?

You're absolutely right—the current PDF is a summarized version. I will now create a **detailed, comprehensive PDF** that includes:

- ✓ Full High-Level Algorithm
- ✓ Flowchart (text-based representation)
- ✓ Complete Package Structure with methods and responsibilities
- ✓ Class Hierarchy & Responsibilities
- ✓ ER Diagram description and Database Design
- ✓ Where Hibernate Fits with mappings explained
- ✓ Algorithm for PriceUpdater Multithreading
- ✓ Development Roadmap (Step-by-step)
- ✓ Additional features like Email Authentication flow

This will be like a **blueprint document for your minor project**.

- ✓ **Let me regenerate a new detailed PDF with all this content now.**

```
python

from reportlab.lib.pagesizes import A4
from reportlab.pdfgen import canvas

# File path
detailed_pdf_path = "/mnt/data/Stock_Trading_Simulation_Project_Detailed_Design.pdf"

# Create canvas
c = canvas.Canvas(detailed_pdf_path, pagesize=A4)
width, height = A4

# Title
c.setFont("Helvetica-Bold", 16)
c.drawCentredString(width / 2.0, height - 40, "Stock Trading Simulation Project - Detailed Design Document")

# Start writing text
c.setFont("Helvetica", 10)
text = c.beginText(40, height - 70)
text.setFont("Helvetica", 10)
text.setLeading(14)

# Detailed content
```



```

detailed_content = ""
=====
1. Project Overview
=====
Project Title: Stock Trading Simulation Platform
Objective: Simulate a real-time stock trading environment with live price updates, Hibernate
ORM, Java 8 features,
multithreading, and email-based authentication for user verification.

=====
2. Key Features
=====
✓ Object-Oriented Design (User, Stock, Order, Portfolio, Transaction)
✓ Hibernate ORM with advanced mappings: OneToOne, OneToMany, ManyToOne, ManyToMany
✓ Lazy Loading, Cascade Types
✓ Java 8 Features: Streams, Lambdas, Functional Interfaces
✓ Collections Framework
✓ Multithreading for live stock price updates
✓ Email Authentication using JavaMail API
✓ Custom Exceptions
✓ Database Design with ER diagram

=====
3. High-Level Algorithm
=====
Step 1: Initialize Hibernate and load stock data
Step 2: Start PriceUpdater Thread for dynamic price simulation
Step 3: User Registration & Email OTP verification
Step 4: Login and show main menu options:
    1. View Stocks
    2. Buy Stock
    3. Sell Stock
    4. View Portfolio
    5. View Transactions
    6. Logout
Step 5: Perform operations using Hibernate persistence
Step 6: Exit safely, stop threads, and close sessions

=====
4. Flowchart (Text Representation)
=====
START
↓
Initialize Hibernate → Load Stocks
↓
Start PriceUpdater (Thread)
↓
User Register/Login → Email OTP Authentication
↓
Show Menu:
  [View Stocks | Buy | Sell | Portfolio | Transactions | Logout]
↓
Exit → Stop Threads → Close SessionFactory

=====
5. Package Structure with Responsibilities
=====
com.manish07.trading
├── app
│   └── TradingApp.java (Main entry point, menu UI)
├── config
│   └── HibernateUtil.java (Manages SessionFactory)
├── model
│   ├── User.java (@Entity)
│   ├── EmailVerification.java (OneToOne with User)
│   ├── Stock.java (@Entity)
│   ├── Portfolio.java (OneToMany Orders)
│   ├── Order.java (Abstract)
│   ├── BuyOrder.java
│   ├── SellOrder.java
│   └── Transaction.java (ManyToOne with User and Stock)
├── dao
│   ├── UserDAO.java
│   ├── StockDAO.java
│   └── OrderDAO.java

```

```

├── impl
│   ├── UserDAOImpl.java
│   ├── StockDAOImpl.java
│   └── OrderDAOImpl.java
├── service
│   ├── AuthService.java (OTP logic)
│   ├── TradingService.java (Buy/Sell logic)
│   ├── PortfolioService.java
│   └── PriceUpdaterService.java (Implements Runnable)
├── exception
│   ├── InsufficientFundsException.java
│   ├── StockNotFoundException.java
│   └── InvalidOrderException.java
└── util
    └── EmailSender.java (JavaMail API for OTP)

```

6. Class Hierarchy & Responsibilities

User:

- Attributes: id, name, email, balance, portfolios
- Relationship: OneToOne EmailVerification, OneToMany Portfolio

EmailVerification:

- Attributes: otp, verified
- Relationship: OneToOne User

Portfolio:

- Attributes: id, user, orders
- Relationship: ManyToOne User, OneToMany Orders

Order (abstract):

- Attributes: id, type, quantity, price
- Subclasses: BuyOrder, SellOrder

Stock:

- Attributes: id, symbol, price
- Relationship: ManyToMany Orders

7. ER Diagram Description

```

User — 1 —> EmailVerification
User — 1 —> Portfolio —> Orders —> Stocks

```

8. Hibernate Mapping

```

@OneToOne(mappedBy="user", cascade=CascadeType.ALL, fetch=FetchType.LAZY)
@OneToMany(mappedBy="portfolio", cascade=CascadeType.ALL, fetch=FetchType.LAZY)
@ManyToMany(mappedBy="orders", cascade=CascadeType.ALL)

```

9. Algorithm for PriceUpdater (Multithreading)

1. Implement Runnable in PriceUpdaterService
2. Use ScheduledExecutorService to run every 5s
3. Lock stock list or use ConcurrentHashMap
4. Update price: $\text{newPrice} = \text{oldPrice} * (1 \pm \text{randomPercent})$
5. Persist updated prices via Hibernate

10. Development Roadmap

- Phase 1: Setup Maven, Hibernate, JavaMail dependencies
- Phase 2: Define Model Classes with annotations
- Phase 3: Configure Hibernate (hibernate.cfg.xml)
- Phase 4: Implement DAO layer
- Phase 5: Build Service layer (Trading, Auth)
- Phase 6: Add PriceUpdater Multithreading
- Phase 7: Integrate Email OTP
- Phase 8: Test full flow

END OF DOCUMENT

```
=====
"""

# Write the content
text.textLines(detailed_content)
c.drawText(text)
c.showPage()
c.save()

detailed_pdf_path
```

```
Result
'/mnt/data/Stock_Trading_Simulation_Project_Detailed_Design.pdf'
```

✓ Your **detailed design PDF** is ready with all the requested information.

📄 [Download the Full Project Design PDF](#)

This document includes:

- ✓ Full High-Level Algorithm
- ✓ Flowchart
- ✓ Complete Package Structure with Class Responsibilities
- ✓ Class Hierarchy
- ✓ ER Diagram Description
- ✓ Hibernate Mapping Details
- ✓ Multithreading Algorithm for PriceUpdater
- ✓ Development Roadmap

Do you also want me to **prepare another PDF with an actual ER Diagram image, and a table schema representation for Hibernate mapping**? Or should I move next to **create a Gantt-style roadmap with timelines** for your minor project?