COMPLETE PROJECT REVISION GUIDE

Trading SDK - Bajaj Broking Campus Hiring Assignment

Created: January 5, 2026
Purpose: Complete reference guide for understanding, running, and explaining the Trading SDK project.

TABLE OF CONTENTS

---

1. PROJECT OVERVIEW

What is this project?

A **Wrapper SDK for Trading APIs** - a backend REST API service that simulates a stock trading platform. It provides functionality for:

- **Viewing Instruments** - List of available stocks/ETFs
- **Order Management** - Place, view, and cancel Buy/Sell orders
- **Trade Tracking** - View executed trades
- **Portfolio Management** - View holdings with profit/loss calculation

One-Line Description (For Interview)

> "A Spring Boot REST API that simulates stock trading with instruments lookup, order management, trade tracking, and portfolio with real-time P&L calculation."

What Problem Does It Solve?

In a real trading platform, users need APIs to:

1. See what stocks are available to trade

2. Place buy/sell orders
3. Track if their orders got executed
4. See their holdings and profit/loss

This SDK provides all these capabilities in a simulated environment.

---

2. ASSIGNMENT REQUIREMENTS

Original Requirements from Bajaj

The assignment asked to build a **Wrapper SDK for Trading APIs** with:

Required APIs:

Category	Endpoints Required
Instruments	GET /instruments
Orders	POST /orders, GET /orders/{id}
Trades	GET /trades
Portfolio	GET /portfolio

Deliverables Required:

1. ☑ **Source Code** - Complete Java/Spring Boot project
2. ☑ **README.md** - Setup instructions, API documentation, assumptions
3. ☑ **Sample API Usage** - cURL commands or Swagger UI

Bonus Points For:

- ☑ Swagger/OpenAPI Documentation
- ☑ Input Validation
- ☑ Error Handling
- ☑ Unit Tests

Key Constraints:

- Single user context (mock authentication)
- Can use in-memory storage (no database required)
- Simulate order execution

---

3. TECHNOLOGY STACK

Technologies Used

Technology	Version	Purpose	Why This Choice?
Java	17	Programming Language	LTS version, required for Spring Boot 3.x
Spring Boot	3.2.0	Application Framework	Industry standard, auto-configuration
Maven	3.6+	Build Tool	Dependency management, standard for Java
Spring Web	(via Boot)	REST APIs	Built-in JSON handling, embedded Tomcat
Spring Validation	(via Boot)	Input Validation	Declarative validation with annotations
Lombok	Latest	Reduce Boilerplate	Generate getters/setters/builders automatically
Springdoc OpenAPI	2.3.0	API Documentation	Modern Swagger UI for Spring Boot 3
JUnit 5	(via Boot)	Testing	Standard testing framework
ConcurrentHashMap	Built-in	Data Storage	Thread-safe in-memory storage

Why NOT Other Technologies?

Alternative	Why We Didn't Use It
Node.js/Express	Assignment implied Java preference
Django/Flask	Assignment implied Java preference
Spring MVC (without Boot)	Requires XML config, slower development
Swagger 2 (springfox)	Discontinued, doesn't support Spring Boot 3
MySQL/PostgreSQL	Assignment allows in-memory, simpler for mock SDK
MongoDB	Overkill for simple key-value storage

4. PROJECT STRUCTURE

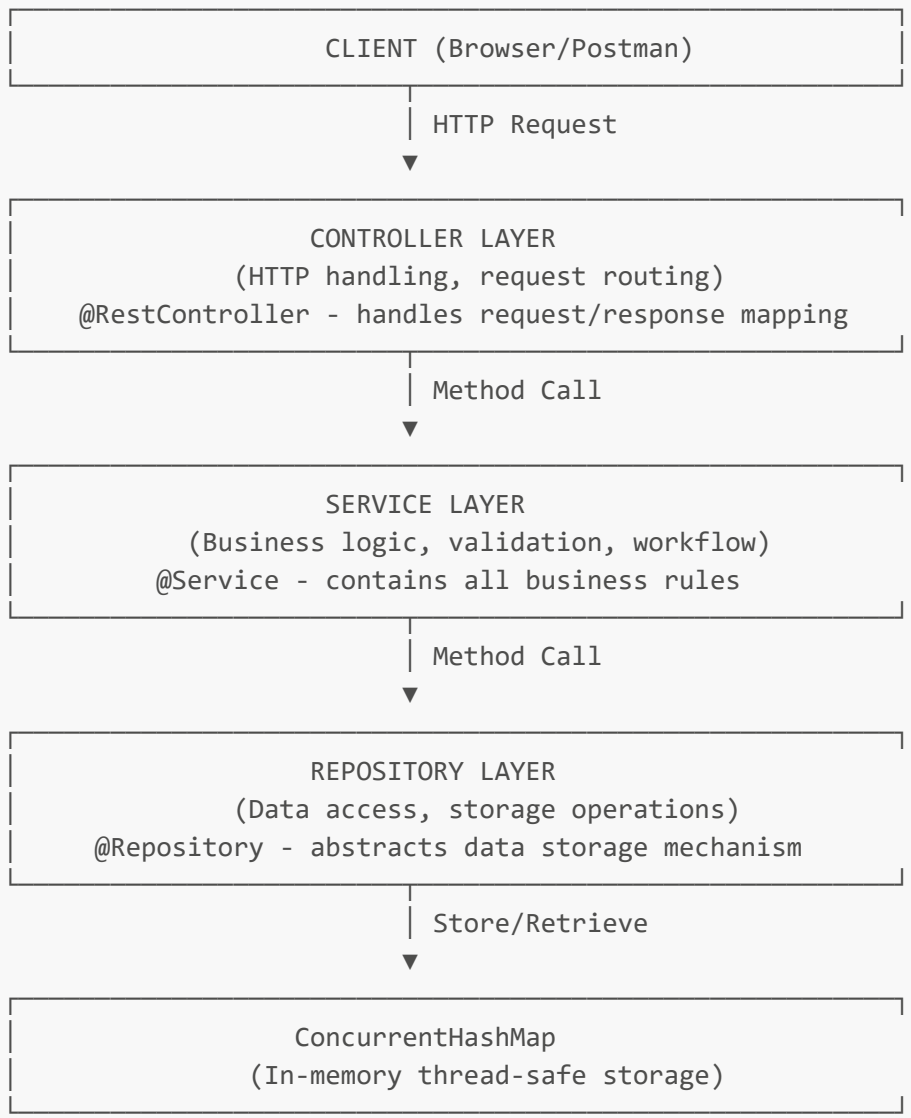
Directory Layout

```
trading-sdk/
├── pom.xml                                    # Maven dependencies & build
config
├── README.md                                  # Project documentation (for
submission)
```

```
├── INTERVIEW_CHEATSHEET.md                    # Quick reference for interviews
├── COMPLETE_PROJECT_GUIDE.md                  # This file - complete revision
guide
│
└── src/
    ├── main/
    │   ├── java/com/bajaj/tradingsdk/
    │   │   │
    │   │   ├── TradingSdkApplication.java     # Entry point - main class
    │   │   │
    │   │   ├── config/
    │   │   │   └── DataInitializer.java       # Loads sample data on startup
    │   │   │
    │   │   ├── controller/                    # REST API Endpoints (HTTP Layer)
    │   │   │   ├── InstrumentController.java  # GET /instruments
    │   │   │   ├── OrderController.java       # POST/GET/DELETE /orders
    │   │   │   ├── TradeController.java       # GET /trades
    │   │   │   └── PortfolioController.java   # GET /portfolio
    │   │   │
    │   │   ├── service/                       # Business Logic Layer
    │   │   │   ├── InstrumentService.java     # Instrument operations
    │   │   │   ├── OrderService.java          # Order placement, cancellation
    │   │   │   ├── TradeService.java          # Trade creation, retrieval
    │   │   │   └── PortfolioService.java      # Holdings, P&L calculation
    │   │   │
    │   │   ├── repository/                    # Data Access Layer
    │   │   │   ├── InstrumentRepository.java  # Store instruments
    │   │   │   ├── OrderRepository.java       # Store orders
    │   │   │   ├── TradeRepository.java       # Store trades
    │   │   │   └── PortfolioRepository.java   # Store holdings
    │   │   │
    │   │   ├── model/                         # Entity Classes
    │   │   │   ├── Instrument.java            # Stock/ETF entity
    │   │   │   ├── Order.java                 # Order entity
    │   │   │   ├── Trade.java                 # Trade entity
    │   │   │   ├── PortfolioHolding.java      # Holding entity
    │   │   │   ├── OrderType.java             # Enum: BUY, SELL
    │   │   │   ├── OrderStyle.java            # Enum: MARKET, LIMIT
    │   │   │   ├── OrderStatus.java           # Enum: NEW, PLACED, EXECUTED,
CANCELLED
    │   │   │   └── InstrumentType.java        # Enum: EQUITY, ETF
    │   │   │
    │   │   ├── dto/                           # Data Transfer Objects
    │   │   │   ├── ApiResponse.java           # Generic response wrapper
    │   │   │   ├── OrderRequest.java          # Order creation request
    │   │   │   └── OrderResponse.java         # Order response format
    │   │   │
    │   │   └── exception/                     # Exception Handling
    │   │       ├── GlobalExceptionHandler.java # Centralized error handling
    │   │       ├── ResourceNotFoundException.java
    │   │       ├── ValidationException.java
    │   │       ├── OrderException.java
    │   │       └── InsufficientHoldingsException.java
    │   │
```

```
    │   └── resources/
    │       └── application.properties          # App configuration
    │
    └── test/
        └── java/com/bajaj/tradingsdk/
            └── TradingSdkApplicationTests.java # Unit tests
```

Layer Architecture (Why This Structure?)



Why Layered Architecture?

Benefit	Explanation
Separation of Concerns	Each layer has one job - easier to understand
Testability	Can test services without controllers, mock repositories
Maintainability	Change storage without touching controllers

Benefit	Explanation
Reusability	Services can be reused across different controllers
Industry Standard	This is how production apps are built at companies like Bajaj

5. HOW EACH COMPONENT WORKS

5.1 Entry Point: TradingSdkApplication.java

```
@SpringBootApplication
@OpenAPIDefinition(info = @Info(title = "Trading SDK API", version = "1.0"))
public class TradingSdkApplication {
    public static void main(String[] args) {
        SpringApplication.run(TradingSdkApplication.class, args);
    }
}
```

What happens when you run this:

1. Spring Boot scans for all @Component, @Service, @Controller, @Repository classes
2. Creates instances (beans) and injects dependencies
3. Starts embedded Tomcat server on port 8080
4. DataInitializer.run() is called to load sample data
5. Application is ready to receive HTTP requests

5.2 Controllers: HTTP Request Handling

Example: OrderController.java

```
@RestController
@RequestMapping("/api/v1/orders")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping
    public ResponseEntity<ApiResponse<OrderResponse>> placeOrder(
            @Valid @RequestBody OrderRequest request) {
        Order order = orderService.placeOrder(request);
        OrderResponse response = OrderResponse.fromOrder(order);
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(ApiResponse.success(response, "Order placed successfully"));
    }
}
```

Key Annotations:

- `@RestController` = @Controller + @ResponseBody (returns JSON)
- `@RequestMapping("/api/v1/orders")` = base URL path
- `@PostMapping` = handles HTTP POST
- `@Valid` = triggers validation on request body
- `@RequestBody` = deserialize JSON to Java object

5.3 Services: Business Logic

Example: OrderService.placeOrder()

```
public Order placeOrder(OrderRequest request) {
    // 1. Validate the order
    validateOrder(request);

    // 2. Get market price
    Double marketPrice = instrumentService.getCurrentPrice(symbol, exchange);

    // 3. Create order object
    Order order = Order.builder()
            .orderId("ORD-" + UUID.randomUUID())
            .symbol(symbol)
            .orderType(BUY/SELL)
            .status(PLACED)
            .build();

    // 4. Save order
    orderRepository.save(order);

    // 5. If MARKET order, execute immediately
    if (orderStyle == MARKET) {
        executeOrder(order, marketPrice);
    }

    return order;
}
```

5.4 Repositories: Data Access

Example: OrderRepository.java

```
@Repository
public class OrderRepository {

    // Thread-safe in-memory storage
    private final Map<String, Order> orders = new ConcurrentHashMap<>();

    public Order save(Order order) {
```

```
        orders.put(order.getOrderId(), order);
        return order;
    }

    public Optional<Order> findById(String orderId) {
        return Optional.ofNullable(orders.get(orderId));
    }

    public List<Order> findByUserId(String userId) {
        return orders.values().stream()
                .filter(order -> order.getUserId().equals(userId))
                .collect(Collectors.toList());
    }
}
```

Why ConcurrentHashMap?

- Thread-safe for concurrent requests
- Fast O(1) lookups
- No external setup required
- Perfect for mock/simulation

5.5 DTOs: Request/Response Objects

OrderRequest.java (with validation):

```
public class OrderRequest {
    @NotBlank(message = "Symbol is required")
    private String symbol;

    @NotNull(message = "Order type is required")
    private OrderType orderType;  // BUY or SELL

    @NotNull(message = "Quantity is required")
    @Min(value = 1, message = "Quantity must be > 0")
    private Integer quantity;

    private Double price;  // Required only for LIMIT orders
}
```

ApiResponse.java (generic wrapper):

```
public class ApiResponse<T> {
    private boolean success;
    private T data;
    private String message;
    private LocalDateTime timestamp;

    // Factory methods
```

```
    public static <T> ApiResponse<T> success(T data, String message) {...}
    public static <T> ApiResponse<T> error(String message) {...}
}
```

5.6 Exception Handling

GlobalExceptionHandler.java:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiResponse<Void>>
handleNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(ApiResponse.error(ex.getMessage()));
    }

    @ExceptionHandler(ValidationException.class)
    public ResponseEntity<ApiResponse<Void>> handleValidation(ValidationException
ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                .body(ApiResponse.error(ex.getMessage()));
    }
}
```

Exception → HTTP Status Mapping:

Exception	HTTP Status	When Thrown
ResourceNotFoundException	404 Not Found	Instrument/Order doesn't exist
ValidationException	400 Bad Request	Invalid input data
OrderException	400 Bad Request	Order operation failed
InsufficientHoldingsException	400 Bad Request	SELL qty > holdings

6. API DOCUMENTATION

Base URL

```
http://localhost:8080/api/v1
```

6.1 Instrument APIs

GET /instruments

Get all available trading instruments.

Request:

```
GET /api/v1/instruments HTTP/1.1
Host: localhost:8080
```

Query Parameters:

Parameter	Type	Description
exchange	String	Filter by exchange (NSE, BSE)
symbol	String	Filter by symbol
type	String	Filter by type (EQUITY, ETF)

Response:

```
{
  "success": true,
  "data": [
    {
      "symbol": "RELIANCE",
      "exchange": "NSE",
      "instrumentType": "EQUITY",
      "lastTradedPrice": 2450.50
    },
    {
      "symbol": "TCS",
      "exchange": "NSE",
      "instrumentType": "EQUITY",
      "lastTradedPrice": 3520.75
    }
  ],
  "message": "Instruments retrieved successfully",
  "timestamp": "2026-01-05T14:30:00"
}
```

6.2 Order APIs

POST /orders

Place a new order.

Request:

```
POST /api/v1/orders HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "symbol": "RELIANCE",
  "exchange": "NSE",
  "orderType": "BUY",
  "orderStyle": "MARKET",
  "quantity": 10
}
```

Order Types:

- `BUY` - Purchase shares
- `SELL` - Sell shares from holdings

Order Styles:

- `MARKET` - Execute immediately at current price
- `LIMIT` - Execute only at specified price (requires `price` field)

Response (Success):

```
{
  "success": true,
  "data": {
    "orderId": "ORD-A1B2C3D4",
    "symbol": "RELIANCE",
    "exchange": "NSE",
    "orderType": "BUY",
    "orderStyle": "MARKET",
    "quantity": 10,
    "price": 2450.50,
    "status": "EXECUTED",
    "createdAt": "2026-01-05T14:30:00",
    "executedAt": "2026-01-05T14:30:01"
  },
  "message": "Order placed successfully"
}
```

GET /orders

Get all orders for the user.

GET /orders/{orderId}

Get specific order by ID.

DELETE /orders/{orderId}

Cancel a pending order.

Note: Only orders with status `PLACED` can be cancelled.

6.3 Trade APIs

GET /trades

Get all executed trades.

Response:

```
{
  "success": true,
  "data": [
    {
      "tradeId": "TRD-X1Y2Z3",
      "orderId": "ORD-A1B2C3D4",
      "symbol": "RELIANCE",
      "exchange": "NSE",
      "quantity": 10,
      "executionPrice": 2450.50,
      "totalValue": 24505.00,
      "executedAt": "2026-01-05T14:30:01"
    }
  ],
  "message": "Trades retrieved successfully"
}
```

6.4 Portfolio APIs

GET /portfolio

Get portfolio holdings with P&L.

Response:

```
{
  "success": true,
  "data": [
    {
      "symbol": "RELIANCE",
      "exchange": "NSE",
      "quantity": 50,
      "averagePrice": 2400.00,
      "currentPrice": 2450.50,
      "currentValue": 122525.00,
      "profitLoss": 2525.00,
```

```
      "profitLossPercentage": 2.10
    }
  ],
  "message": "Portfolio retrieved successfully"
}
```

GET /portfolio/summary

Get portfolio summary with total P&L.

---

7. CODE WALKTHROUGH

Complete Order Flow (Step by Step)

Step 1: User sends HTTP request

```
POST /api/v1/orders
Content-Type: application/json

{
  "symbol": "RELIANCE",
  "exchange": "NSE",
  "orderType": "BUY",
  "orderStyle": "MARKET",
  "quantity": 10
}
```

Step 2: Controller receives request

```
// OrderController.java
@PostMapping
public ResponseEntity<ApiResponse<OrderResponse>> placeOrder(
        @Valid @RequestBody OrderRequest request) {

    // @Valid triggers validation - checks @NotNull, @NotBlank, @Min
    // If validation fails → MethodArgumentNotValidException → 400 Bad Request

    Order order = orderService.placeOrder(request);
    return ResponseEntity.status(HttpStatus.CREATED)
            .body(ApiResponse.success(response, "Order placed successfully"));
}
```

Step 3: Service processes business logic

```
// OrderService.java
public Order placeOrder(OrderRequest request) {

    // STEP 3a: Validate order
    validateOrder(request);  // Check if instrument exists, SELL holdings check

    // STEP 3b: Get market price
    Double marketPrice = instrumentService.getCurrentPrice(symbol, exchange);

    // STEP 3c: Create order object
    Order order = Order.builder()
            .orderId("ORD-" + UUID.randomUUID().toString().substring(0, 8))
            .symbol(request.getSymbol())
            .orderType(request.getOrderType())
            .orderStyle(request.getOrderStyle())
            .quantity(request.getQuantity())
            .price(marketPrice)  // or request.getPrice() for LIMIT
            .status(OrderStatus.PLACED)
            .userId(MOCK_USER_ID)
            .createdAt(LocalDateTime.now())
            .build();

    // STEP 3d: Save order
    orderRepository.save(order);

    // STEP 3e: For MARKET orders, execute immediately
    if (request.getOrderStyle() == OrderStyle.MARKET) {
        executeOrder(order, marketPrice);
    }

    return order;
}

private void executeOrder(Order order, double price) {
    // Update order status
    order.setStatus(OrderStatus.EXECUTED);
    order.setExecutedAt(LocalDateTime.now());
    orderRepository.save(order);

    // Create trade record
    tradeService.createTrade(order, price);

    // Update portfolio
    if (order.getOrderType() == OrderType.BUY) {
        portfolioService.addToPortfolio(userId, symbol, exchange, quantity,
price);
    } else {
        portfolioService.removeFromPortfolio(userId, symbol, exchange, quantity);
    }
}
```

Step 4: Repository stores data

```
// OrderRepository.java
public Order save(Order order) {
    orders.put(order.getOrderId(), order);  // ConcurrentHashMap
    return order;
}
```

Step 5: Response sent back

```
{
  "success": true,
  "data": {
    "orderId": "ORD-A1B2C3D4",
    "status": "EXECUTED"
  },
  "message": "Order placed successfully"
}
```

P&L Calculation Logic

```
// PortfolioService.java
public List<PortfolioHolding> getPortfolio(String userId) {
    List<PortfolioHolding> holdings = portfolioRepository.findByUserId(userId);

    for (PortfolioHolding holding : holdings) {
        // Get current market price
        Double currentPrice = instrumentService.getCurrentPrice(
            holding.getSymbol(), holding.getExchange());

        // Calculate values
        double currentValue = holding.getQuantity() * currentPrice;
        double investment = holding.getQuantity() * holding.getAveragePrice();
        double profitLoss = currentValue - investment;
        double profitLossPercentage = (profitLoss / investment) * 100;

        // Set calculated values
        holding.setCurrentPrice(currentPrice);
        holding.setCurrentValue(currentValue);
        holding.setProfitLoss(profitLoss);
        holding.setProfitLossPercentage(profitLossPercentage);
    }

    return holdings;
}
```

Formula:

```
Investment = Quantity × Average Buy Price
Current Value = Quantity × Current Market Price
P&L = Current Value - Investment
P&L % = (P&L / Investment) × 100
```

---

8. DESIGN DECISIONS & WHY

Decision 1: In-Memory Storage (ConcurrentHashMap)

What we chose:

```
private final Map<String, Order> orders = new ConcurrentHashMap<>();
```

Why ConcurrentHashMap?

Feature	HashMap	ConcurrentHashMap
Thread-safe	❌ No	☑ Yes
Concurrent reads	❌ Can corrupt	☑ Safe
Concurrent writes	❌ Can corrupt	☑ Safe
Performance	Faster (single-threaded)	Slightly slower but safe

Why NOT a database?

Approach	Pros	Cons	Decision
ConcurrentHashMap	Fast, no setup, thread-safe	Data lost on restart	☑ Used (assignment allows)
H2 (embedded)	SQL, persists in session	Extra config	Not needed for mock
PostgreSQL	Production-ready	Needs server setup	Overkill for assignment

How to switch to database later?

Only change Repository layer:

```
// Current (in-memory)
@Repository
public class OrderRepository {
    private final Map<String, Order> orders = new ConcurrentHashMap<>();
}

// Future (JPA)
```

```
@Repository
public interface OrderRepository extends JpaRepository<Order, String> {
    List<Order> findByUserId(String userId);
}
```

Controllers and Services remain **unchanged** - that's the beauty of layered architecture!

Decision 2: Springdoc OpenAPI (not Swagger 2)

Why?

- Swagger 2 (springfox) is **discontinued** since 2020
- Doesn't support Spring Boot 3.x
- Springdoc is the modern replacement
- Same Swagger UI experience

Decision 3: Separate DTOs from Entities

Why not expose entities directly?

```
// ❌ BAD: Exposing entity directly
@PostMapping
public Order placeOrder(@RequestBody Order order) {
    return orderRepository.save(order);
}
// Problems:
// - User can set internal fields like status, userId
// - Entity changes break API contract
// - No validation control
```

```
// ☑ GOOD: Using DTOs
@PostMapping
public ApiResponse<OrderResponse> placeOrder(@Valid @RequestBody OrderRequest
request) {
    Order order = orderService.placeOrder(request);
    return ApiResponse.success(OrderResponse.fromOrder(order), "Success");
}
// Benefits:
// - Control exactly what fields are accepted
// - Validation on request DTO
// - Response format independent of entity
// - Security - hide internal fields
```

Decision 4: Centralized Exception Handling

Why @RestControllerAdvice?

```
// ❌ WITHOUT centralized handling - Code duplication!
@PostMapping
public ResponseEntity<?> placeOrder(@RequestBody OrderRequest request) {
    try {
        Order order = orderService.placeOrder(request);
        return ResponseEntity.ok(order);
    } catch (ResourceNotFoundException e) {
        return ResponseEntity.status(404).body(e.getMessage());
    } catch (ValidationException e) {
        return ResponseEntity.status(400).body(e.getMessage());
    } catch (Exception e) {
        return ResponseEntity.status(500).body("Error");
    }
}
// This try-catch is in EVERY method - DRY violation!
```

```
// ☑ WITH centralized handling - Clean!
@PostMapping
public ResponseEntity<?> placeOrder(@Valid @RequestBody OrderRequest request) {
    Order order = orderService.placeOrder(request);
    return ResponseEntity.ok(ApiResponse.success(order, "Success"));
}
// No try-catch needed - GlobalExceptionHandler catches all!
```

Decision 5: Mock User Authentication

Current implementation:

```
private static final String MOCK_USER_ID = "USER001";
```

Why mock?

- Assignment says "single user context"
- Real auth would require Spring Security + JWT
- Can be added later without changing architecture

How to add real auth later:

```
// Replace hardcoded user with:
String userId = SecurityContextHolder.getContext()
                    .getAuthentication().getName();
```

9. HOW TO RUN & TEST

Prerequisites

Tool	Version	Check Command
Java	17+	`java -version`
Maven	3.6+	`mvn -version`

Running the Application

Method 1: Using Maven (Recommended)

```
cd d:\Placement\Bajaj\trading-sdk
mvn org.springframework.boot:spring-boot-maven-plugin:run
```

Method 2: Using JAR

```
cd d:\Placement\Bajaj\trading-sdk
mvn clean package
java -jar target/trading-sdk-1.0.0.jar
```

Accessing the Application

Service	URL
API Base	http://localhost:8080/api/v1
Swagger UI	http://localhost:8080/swagger-ui.html
OpenAPI JSON	http://localhost:8080/api-docs

Testing with cURL

1. Get all instruments

```
curl http://localhost:8080/api/v1/instruments
```

2. Place a BUY MARKET order

```

```

```
curl -X POST http://localhost:8080/api/v1/orders `
  -H "Content-Type: application/json" `
  -d
'{"symbol":"RELIANCE","exchange":"NSE","orderType":"BUY","orderStyle":"MARKET","qu
antity":10}'
```

3. Get all orders

```
curl http://localhost:8080/api/v1/orders
```

4. Get all trades

```
curl http://localhost:8080/api/v1/trades
```

5. Get portfolio

```
curl http://localhost:8080/api/v1/portfolio
```

6. Cancel an order

```
curl -X DELETE http://localhost:8080/api/v1/orders/{orderId}
```

Running Unit Tests

```
cd d:\Placement\Bajaj\trading-sdk
mvn test
```

---

10. INTERVIEW PREPARATION

How to Introduce the Project (2-3 minutes)

> "I built a Trading SDK - a RESTful backend service that simulates a stock trading platform using Java 17 and Spring Boot 3.2.
>
> It provides four main API categories:
>
> 1. **Instruments** - View available stocks and ETFs

> 2. **Orders** - Place, view, and cancel buy/sell orders
> 3. **Trades** - Track executed trades
> 4. **Portfolio** - View holdings with real-time P&L calculation
>
> Key technical decisions:
>
> - Layered architecture (Controller → Service → Repository)
> - In-memory storage with ConcurrentHashMap for thread-safety
> - Swagger UI for API documentation
> - Centralized exception handling with @RestControllerAdvice
>
> MARKET orders execute immediately while LIMIT orders stay pending. SELL orders validate against available holdings."

Code Walkthrough Strategy

When asked "Walk me through your code":

1. **Start with pom.xml** - Show dependencies
2. **Show TradingSdkApplication.java** - Entry point
3. **Pick one flow** - Order placement is best
4. **Trace Controller → Service → Repository**
5. **Highlight design patterns** - DTO, exception handling

Key Points to Emphasize

- ☑ "Thread-safe with ConcurrentHashMap"
- ☑ "Layered architecture - easy to swap database later"
- ☑ "Validation at request level with annotations"
- ☑ "Centralized error handling - DRY principle"
- ☑ "MARKET vs LIMIT order logic"
- ☑ "P&L calculation in real-time"

---

11. COMMON QUESTIONS & ANSWERS

Architecture Questions

Q: "Why didn't you use a database?"

> "The assignment allowed in-memory storage for a mock SDK. ConcurrentHashMap is thread-safe and requires zero setup. For production, I'd use PostgreSQL with Spring Data JPA - only the Repository layer changes, Services and Controllers remain unchanged."

Q: "Is your code thread-safe?"

> "Yes. I use ConcurrentHashMap which handles concurrent reads and writes safely. Multiple users can place orders simultaneously without data corruption."

Q: "Why layered architecture?"

> "Separation of concerns. Controllers handle HTTP, Services contain business logic, Repositories manage data. This makes testing easier, code reusable, and future changes isolated to one layer."

Technology Questions

Q: "Why Spring Boot over other frameworks?"

> "Industry standard in Indian fintech. Auto-configuration reduces boilerplate. Embedded Tomcat means no separate server setup. Huge community and documentation."

Q: "Why Springdoc and not Swagger 2?"

> "Swagger 2 (springfox) is discontinued and doesn't support Spring Boot 3. Springdoc is the modern replacement with the same Swagger UI experience."

Q: "Why Lombok?"

> "Reduces boilerplate. Without Lombok, each entity needs 50+ lines for getters, setters, constructors, builders. With @Data and @Builder, it's 10 lines."

Design Questions

Q: "Why separate DTOs from entities?"

> "Security and flexibility. DTOs control exactly what goes in/out of the API. Entities might have internal fields we shouldn't expose. Also allows request validation without affecting entity."

Q: "Why @RestControllerAdvice?"

> "Centralized exception handling. Without it, every controller method needs try-catch blocks - DRY violation. With it, exceptions are caught globally and converted to proper HTTP responses."

Scenario Questions

Q: "What if user tries to SELL more than they own?"

> "The validateOrder() method checks holdings before placing SELL orders. If quantity > holdings, it throws InsufficientHoldingsException which becomes a 400 Bad Request."

Q: "What happens when server restarts?"

> "Data is lost since it's in-memory. For production, I'd use a database. The architecture supports easy migration - just change Repository layer."

Q: "How would you add authentication?"

> "Add Spring Security with JWT. Create JwtTokenProvider for token generation. Add JwtAuthenticationFilter. Replace MOCK_USER_ID with user from SecurityContext."

Q: "How would you add a new exchange like MCX?"

> "Just add instruments with exchange='MCX'. My design already supports multiple exchanges - there's an exchange field in every entity. For production, I'd add exchange trading hours validation."

Code Modification Questions

Q: "Can you add pagination to /orders?"

> "Yes. Add Pageable parameter to controller, return Page instead of List. Spring Data handles it automatically."

```
@GetMapping
public Page<Order> getOrders(Pageable pageable) {
    return orderService.getOrders(pageable);
}
// GET /orders?page=0&size=10&sort=createdAt,desc
```

Q: "How to add order history?"

> "Add endpoint GET /orders/history with date filters. Add method in service to query orders by date range and status."

Q: "How to make LIMIT orders execute?"

> "Add a matching engine - a scheduled job that compares LIMIT order prices with current market prices and executes matching orders."

---

12. FUTURE IMPROVEMENTS

If Given More Time

Improvement	How I'd Implement It
Database	Add spring-data-jpa, create entities with @Entity, extend JpaRepository
Authentication	Spring Security + JWT tokens
Caching	Redis for frequently accessed data (instruments, portfolio)
Rate Limiting	Spring Cloud Gateway or bucket4j
Async Processing	@Async for order processing, message queue for notifications
Pagination	Pageable for all list endpoints

Improvement	How I'd Implement It
Audit Trail	@CreatedDate, @LastModifiedDate, who created/modified
API Versioning	Already using /v1, can add /v2 for breaking changes

Production Considerations

Aspect	Current	Production
Storage	ConcurrentHashMap	PostgreSQL + Redis cache
Auth	Mock user	Spring Security + JWT
Deployment	Local	Docker + Kubernetes
Monitoring	Logs only	Prometheus + Grafana
Security	None	HTTPS, CORS, rate limiting

📌 QUICK REFERENCE

Key Commands

```
# Start app
mvn org.springframework.boot:spring-boot-maven-plugin:run

# Run tests
mvn test

# Build JAR
mvn clean package
```

Key URLs

- Swagger: http://localhost:8080/swagger-ui.html
- API: http://localhost:8080/api/v1

Key Files

- Entry point: `TradingSdkApplication.java`
- Order flow: `OrderService.java`
- Error handling: `GlobalExceptionHandler.java`
- Validation: `OrderRequest.java`

📖 Use this document to revise before your interview. Read through the Q&A section multiple times.

🍀 Good luck!