GSDC
Global Skill Development Council

JAVA FULL STACK

Project Implementation Workbook

www.gsdcouncil.org

Introduction

The purpose of this workbook is to help learners apply Java Full Stack Development concepts through hands-on projects and real-world implementation exercises. Each section presents a realistic business scenario, concrete learning objectives, and guided activities designed to mirror what developers encounter in professional enterprise environments.

Rather than focusing solely on theory, this workbook emphasizes implementation. You will design systems, write REST APIs, build Angular components, model databases, and deploy containerized applications. The skills developed here map directly to the demands of modern enterprise Java development.

Full Stack Architecture

Design and connect Angular front-ends to Spring Boot backends with clean separation of concerns.

Authentication & Security

Implement JWT-based authentication, role-based access control, and password encryption.

Deployment & DevOps

Containerize applications with Docker and build automated CI/CD pipelines for reliable delivery.

Performance & Testing

Diagnose bottlenecks, write comprehensive test suites, and optimize database queries.

Project 1: Employee Management System

PROJECT 1 OF 5

A growing company needs a centralized application to manage employees, departments, and personnel records. HR administrators require a reliable, secure interface to handle the full employee lifecycle — from onboarding through role transitions and eventual offboarding. This project serves as your foundational full-stack exercise, covering every tier of a Java enterprise application.

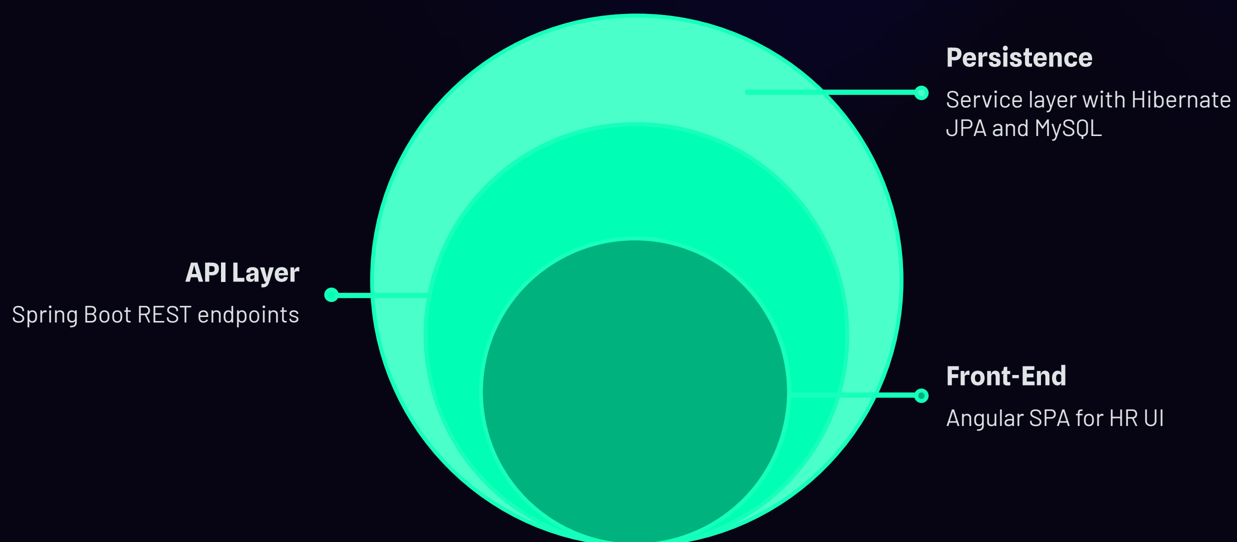
Business Goals

- Add and register new employees
- Update existing employee information
- Search and filter employee records
- Assign and manage departments
- Generate employee reports

Learning Objectives

- Design a relational database schema
- Build REST APIs using Spring Boot
- Create Angular forms with validation
- Implement full CRUD operations
- Connect front-end and back-end tiers

Project Architecture



This layered architecture enforces clean separation between the presentation, business logic, data access, and persistence tiers — a pattern you will apply in every project throughout this workbook.

Activity 1: Requirements Analysis

PROJECT 1 — EMPLOYEE MANAGEMENT SYSTEM

Before writing a single line of code, effective software engineers invest time in understanding what the system must do and the constraints it must operate within. Requirements analysis is the discipline of capturing both functional behaviors and non-functional quality attributes that define project success.

Functional Requirements

Employee Registration

Capture personal details, designation, and department assignment at hire.

Employee Update

Allow HR to modify any field in the employee record post-hire.

Employee Deletion

Soft or hard delete of records with appropriate audit trail.

Search & Department Mgmt

Filter by name, ID, or department; create and edit department records.

Non-Functional Requirements

Security

Role-based access and encrypted credentials.

Scalability

Support growing employee headcount without degradation.

Availability

Minimal downtime with clear recovery procedures.

Performance

API response times under 500ms for typical operations.

Exercise: Identify all user roles (e.g., HR Admin, Manager, Employee), the core business processes each role performs, reports the system must generate, and the primary data entities involved. Document your findings before proceeding to database design.

Activity 2: Database Design

PROJECT 1 — EMPLOYEE MANAGEMENT SYSTEM

A well-designed relational schema is the backbone of any data-driven application. For the Employee Management System, the core entities are **Employee** and **Department**, linked through a foreign key relationship. Good schema design at this stage prevents costly migrations and performance issues later in the project lifecycle.

Employee Entity

Field	Type	Constraints
Employee ID	Integer	Primary Key, Auto Increment
Name	String	Not Null, Max 100 chars
Email	String	Unique, Not Null
Designation	String	Not Null
Department ID	Integer	Foreign Key → Department

Department Entity

Field	Type	Constraints
Department ID	Integer	Primary Key, Auto Increment
Department Name	String	Unique, Not Null
Manager ID	Integer	FK → Employee (nullable)

Exercise: Draw a complete ER Diagram showing the Employee-Department relationship. Define primary keys, foreign keys, and any junction tables needed for future features such as employee skills or project assignments. Consider what indexes will support fast search queries.

Activity 3: Backend Development

PROJECT 1 — EMPLOYEE MANAGEMENT SYSTEM

Spring Boot provides a fast, opinionated framework for building production-grade REST APIs. The backend follows a layered architecture: **Controller** handles HTTP routing, **Service** contains business logic, **Repository** manages JPA queries, and **Entity** classes map to the database. Keeping these layers separate ensures testability and maintainability as the codebase grows.

01

Entity Classes

Annotate POJOs with `@Entity`, `@Table`, `@Id`, and JPA relationship annotations (`@ManyToOne`, `@OneToMany`).

02

Repository Interfaces

Extend `JpaRepository` or `CrudRepository`. Add custom query methods using Spring Data's derived query naming convention.

03

Service Layer

Annotate with `@Service`. Inject repositories via constructor injection. Implement business rules such as duplicate email checks.

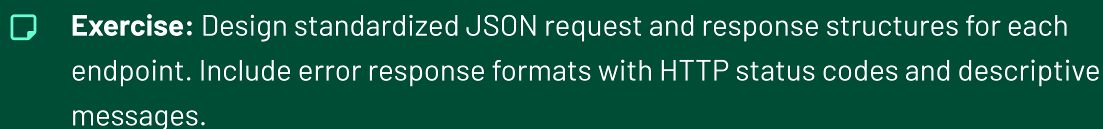
04

Controller Layer

Annotate with `@RestController` and `@RequestMapping`. Return `ResponseEntity` objects with appropriate HTTP status codes.

REST Endpoints

Endpoint	Method	Description
`/employees`	GET	Retrieve all employees
`/employees`	POST	Create a new employee record
`/employees/{id}`	PUT	Update existing employee by ID
`/employees/{id}`	DELETE	Remove employee record by ID



Exercise: Design standardized JSON request and response structures for each endpoint. Include error response formats with HTTP status codes and descriptive messages.

Activity 4: Angular Front-End

PROJECT 1 — EMPLOYEE MANAGEMENT SYSTEM

The Angular front-end provides the user-facing interface that HR administrators interact with daily. A clean, well-structured component tree makes the application easy to navigate and maintain. Each feature area should map to its own dedicated Angular component, with shared services handling data communication with the Spring Boot API via `HttpClient`.

Employee List

Tabular view of all employees with sorting, pagination, and search filter. Provides quick-access action buttons for edit and delete.

Employee Form

Reactive form for creating and editing employee records. Includes field-level validation with user-friendly error messages.

Department Management

Dedicated view for creating, editing, and deleting department entries. Allows HR admins to manage the department reference table.

Dashboard

Summary view displaying headcount by department, recent additions, and quick navigation links to key features.

Exercise: Create wireframes for each of the four pages listed above. Sketch the layout, key UI controls, and navigation flows. Consider how a user moves between the employee list, form, and dashboard in a natural workflow.

Activity 5: Testing

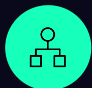
PROJECT 1 — EMPLOYEE MANAGEMENT SYSTEM

Testing is not a final step — it is woven into every phase of development. For the Employee Management System, you should write unit tests for service-layer logic, integration tests for repositories, and end-to-end API tests using tools like Postman or RestAssured. Angular components should be tested with Jasmine and Karma. A disciplined testing practice catches defects early, when they are cheapest to fix.

Core Test Cases

Scenario	Input	Expected Result
Create Employee	Valid employee data	Record saved, 201 Created returned
Invalid Email	Malformed email string	400 Bad Request with validation error
Delete Employee	Valid employee ID	Record removed, 204 No Content returned
Duplicate Email	Email already in database	409 Conflict with descriptive message
Get Non-Existent	Unknown employee ID	404 Not Found returned

Project 1 Deliverables Checklist



ER Diagram

Complete entity-relationship diagram with keys and relationships.

API Design

Documented REST endpoints with request/response schemas.

Angular UI Design

Wireframes for all four Angular components.

Test Cases

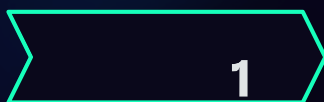
Documented test scenarios with expected results for all operations.

Project 2: E-Commerce Application

PROJECT 2 OF 5

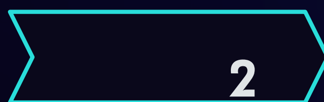
An online retailer requires a fully functional e-commerce platform that handles product catalog management, shopping cart operations, and end-to-end order processing. This project significantly increases system complexity compared to Project 1 — you will design multi-module applications, implement secure user authentication, and build transactional workflows that must remain consistent even under concurrent load.

The e-commerce domain introduces important new patterns including stateful cart management, inventory tracking, payment integration hooks, and order lifecycle state machines. These are core competencies for enterprise Java developers working in the commerce and retail sectors.



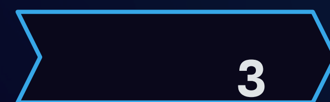
Customer Management

Registration, login, profile management, and address book.



Product Management

Product listings, search and filter, category navigation, inventory tracking.



Order Management

Shopping cart, checkout flow, payment processing, and order history.

Learning Objectives

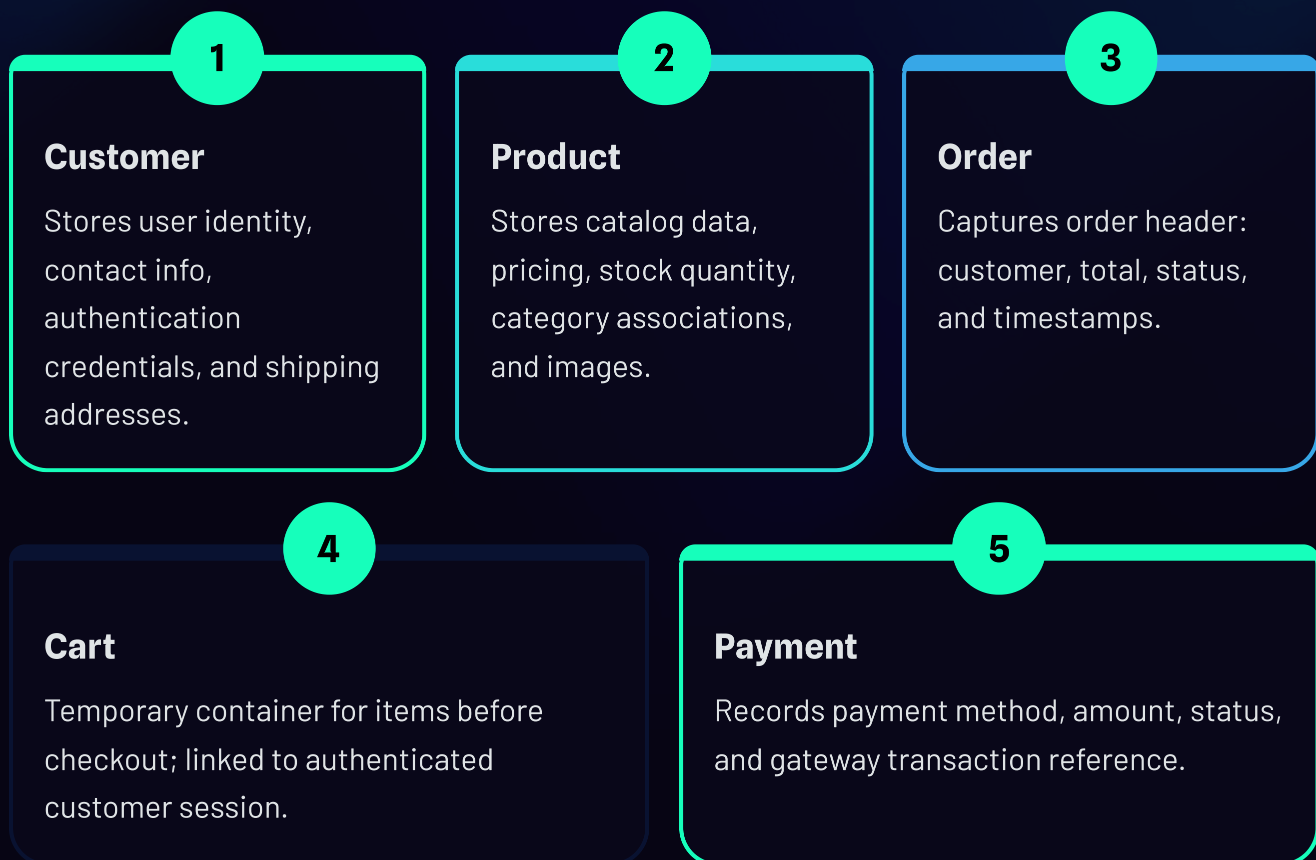
- Build multi-module Spring Boot applications with clear module boundaries
- Design normalized product catalog schemas with categories and attributes
- Implement stateful order workflows with proper status transitions
- Create secure user authentication using JWT tokens and Spring Security

Activity 1: System Design

PROJECT 2 — E-COMMERCE APPLICATION

System design for an e-commerce platform begins with identifying the core domain entities and the relationships between them. Unlike the Employee Management System, the e-commerce domain involves transactional workflows where multiple entities must be created or updated atomically. For example, placing an order must simultaneously decrement inventory, create an order record, and record a payment event — all within a single database transaction.

Before drawing architecture diagrams, map out each entity, its attributes, and how it relates to other entities. Think carefully about cardinality: a Customer can have many Orders, an Order can have many Products, and a Product can belong to many Orders. This many-to-many relationship between Orders and Products requires a junction table (`order_items`) to store line-item quantities and pricing at time of purchase.



Exercise: Draw a complete architecture diagram showing all five entities, their relationships, and the layers of the Spring Boot application. Indicate which module owns which entities.

Activity 2: API Design

PROJECT 2 — E-COMMERCE APPLICATION

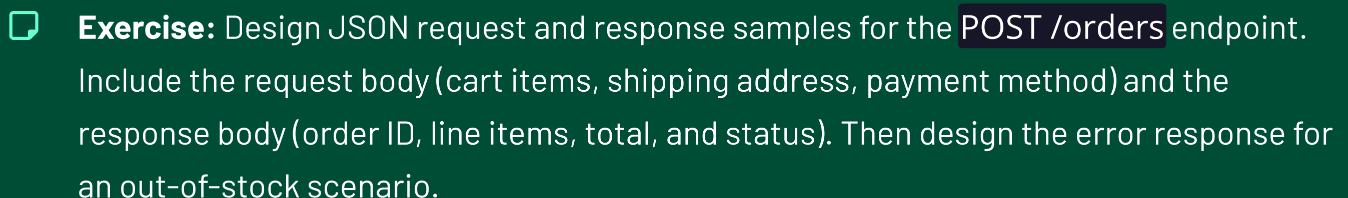
A well-designed REST API is predictable, consistent, and self-documenting. For the e-commerce platform, APIs must handle both catalog browsing (typically high read volume) and transactional operations (order placement, cart updates) which require stronger consistency guarantees. Group your endpoints logically by resource domain and follow REST conventions for HTTP method semantics.

Product APIs

Endpoint	Method
`/products`	GET — List all
`/products`	POST — Create
`/products/{id}`	GET — Get one
`/products/{id}`	PUT — Update
`/products/search`	GET — Search

Order APIs

Endpoint	Method
`/orders`	POST — Place order
`/orders/{id}`	GET — Get details
`/orders/customer/{id}`	GET — Order history
`/cart`	POST — Add to cart
`/cart/{id}`	DELETE — Remove item



Exercise: Design JSON request and response samples for the `POST /orders` endpoint. Include the request body (cart items, shipping address, payment method) and the response body (order ID, line items, total, and status). Then design the error response for an out-of-stock scenario.

Activity 3: Security Implementation

PROJECT 2 — E-COMMERCE APPLICATION

Security is non-negotiable in an e-commerce application. Customer data, payment information, and order records require strong access controls. Spring Security combined with JWT (JSON Web Tokens) provides a stateless authentication mechanism well suited to RESTful architectures. Passwords must never be stored in plaintext — use BCrypt hashing with an appropriate work factor.

Role-based access control (RBAC) ensures that customers can only view and modify their own data, while administrators have broader management capabilities. Always apply the principle of least privilege: grant each role only the permissions it strictly requires to perform its function.

Security Role Matrix

Role	Browse Products	Manage Products	View Reports	Manage Users
Customer	✓	[illegible]	[illegible]	[illegible]
Manager	✓	✓	✓	[illegible]
Admin	✓	✓	✓	✓

Project 2 Deliverables

API Specifications

All endpoints documented with request/response schemas and error codes.

Security Model

JWT flow diagram, role definitions, and password hashing approach.

Database Design

Complete ER diagram including Cart, OrderItems, and Payment tables.

UI Mockups

Wireframes for product listing, cart, checkout, and order history pages.

Project 3: Student Learning Management System

PROJECT 3 OF 5

An educational institution requires a web application to manage the full lifecycle of academic courses — from creation and enrollment through assessment and grading. The Learning Management System (LMS) introduces a new layer of complexity: multiple user types with distinct roles and permissions operating on the same underlying data. A student sees their own enrollments; an instructor sees their assigned courses and student rosters; an administrator manages the full system.

This project emphasizes multi-user application design, role-based access control at the feature level, and complex relational database modeling where many-to-many relationships are common. These patterns appear throughout enterprise software in healthcare, finance, and education sectors.

Functional Modules

- **Student Management:** Registration, profile, enrollment management
- **Course Management:** Course creation, scheduling, assignment to instructors
- **Assessment Management:** Quizzes, assignments, grade recording

Learning Objectives

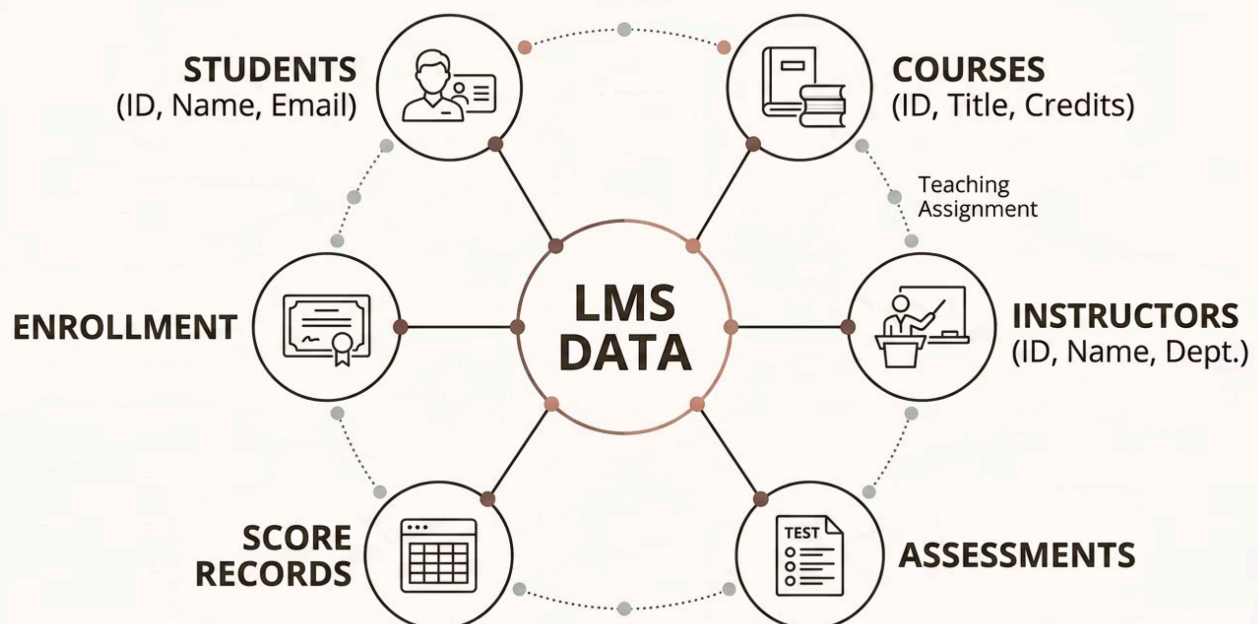
- Design and implement multi-user applications with distinct personas
- Apply role-based access control at the method and data level
- Model complex many-to-many database relationships
- Build reusable Spring Security configurations for multiple roles

Activity: LMS Data Modeling & Design

PROJECT 3 — STUDENT LEARNING MANAGEMENT SYSTEM

The LMS data model is richer than the previous two projects. Students enroll in Courses, Instructors teach Courses, and Assessments belong to Courses with individual student Score records. Each of these relationships has different cardinality and lifecycle characteristics that must be reflected in your schema design.

Consider the enrollment relationship carefully: when a student enrolls in a course, you need to capture not just the association but also metadata such as enrollment date, completion status, and final grade. This makes the enrollment itself an entity — a common pattern in enterprise data modeling called an **associative entity** or **junction entity with attributes**.



Database Schema

ER diagram with Student, Course, Instructor, Enrollment, Assessment, and Score tables with all keys defined.

Role Matrix

Table showing which actions each role (Student, Instructor, Admin) may perform on each entity.

Functional Design

UI flow diagrams for each persona's primary workflows through the LMS application.

Project 4: Inventory Management System

PROJECT 4 OF 5

A warehouse operation requires a robust inventory tracking and stock management solution. Real-time visibility into stock levels, supplier relationships, and product movements is critical to preventing stockouts and overstock situations that directly impact a business's bottom line. This project focuses on operational data management — a domain where data accuracy, transactional integrity, and reporting capabilities are paramount.

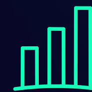
The Inventory Management System introduces dashboard-driven design, business process automation through Spring scheduled tasks or event-driven triggers, and reporting queries that aggregate large datasets. You will also practice data validation at multiple levels: Angular form validation, Spring Boot controller validation with Bean Validation annotations, and database constraints.

Product Tracking

Maintain a complete catalog of products with current stock levels, minimum thresholds, and reorder points.

Supplier Management

Record supplier contacts, pricing, lead times, and delivery schedules. Track pending purchase orders.



Inventory Reports

Generate stock valuation, low-stock alerts, movement history, and supplier performance reports.

Exercise: Design the full inventory workflow from product receipt (goods-in) through storage, pick-and-pack, and dispatch. Identify which workflow steps require database writes and which events could trigger automated low-stock notifications.

Activity: Inventory Module Design

PROJECT 4 — INVENTORY MANAGEMENT SYSTEM

Breaking the Inventory Management System into clear modules keeps the codebase navigable and allows teams to work in parallel. The two primary modules are **Product** — managing the catalog and stock levels — and **Supplier** — managing vendor relationships and inbound deliveries. A shared **Reporting** module queries across both domains.

Product Module

- Add new product with SKU, description, category, and unit price
- Update stock levels on receipt or adjustment
- Search inventory by name, SKU, or category
- Set minimum stock thresholds for auto-alerts
- View product movement history

Supplier Module

- Add and manage supplier contact information
- Record and track inbound deliveries against purchase orders
- Evaluate supplier lead times and fulfillment rates
- Link products to preferred and backup suppliers
- Generate supplier performance summaries

Project 4 Deliverables

1

Inventory Process Diagram

Visual workflow covering goods-in, storage, adjustment, and dispatch stages.

2

API Specifications

Full REST endpoint list for Product and Supplier modules with HTTP methods and payloads.

3

Dashboard Design

Wireframe of the inventory overview dashboard showing key metrics and low-stock alerts.

Project 5: Customer Relationship Management (CRM)

PROJECT 5 OF 5

A sales organization needs a centralized platform to manage leads, track customer interactions, and forecast revenue. The CRM Application is the most business-process-intensive project in this workbook. It requires modeling the customer journey from initial lead acquisition through active customer management and ongoing opportunity tracking — with data flowing through multiple pipeline stages and involving multiple team members across a sales organization.

Building a CRM reinforces key patterns: event-driven state machines for pipeline stages, complex reporting queries for forecasting, and collaborative workflows where multiple users act on shared records. These are sophisticated enterprise requirements that demand careful design before implementation begins.

CRM Core Modules

- **Lead Management:** Create leads, assign to representatives, track status through qualification
- **Customer Management:** Customer profiles, full contact interaction history, account notes
- **Opportunity Tracking:** Sales pipeline visualization, stage transitions, revenue forecasting by rep and period

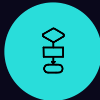
CRM Deliverables & Microservices Introduction

PROJECT 5 — CRM + MICROSERVICES EXERCISE

Before submitting your CRM deliverables, complete the customer journey mapping exercise below. Understanding how a prospect moves from first contact to active customer — and the touchpoints along the way — directly informs your data model and API design. Each stage transition in the pipeline represents a state change that must be persisted, audited, and potentially trigger downstream actions such as notifications or task assignments.

CRM Architecture

System architecture diagram showing all modules, their interactions, and integration points.



Workflow Diagrams

State machine diagrams for lead-to-opportunity and opportunity-to-customer transitions.

API Documentation

Comprehensive endpoint documentation with authentication requirements and sample payloads.

Microservices Implementation Exercise

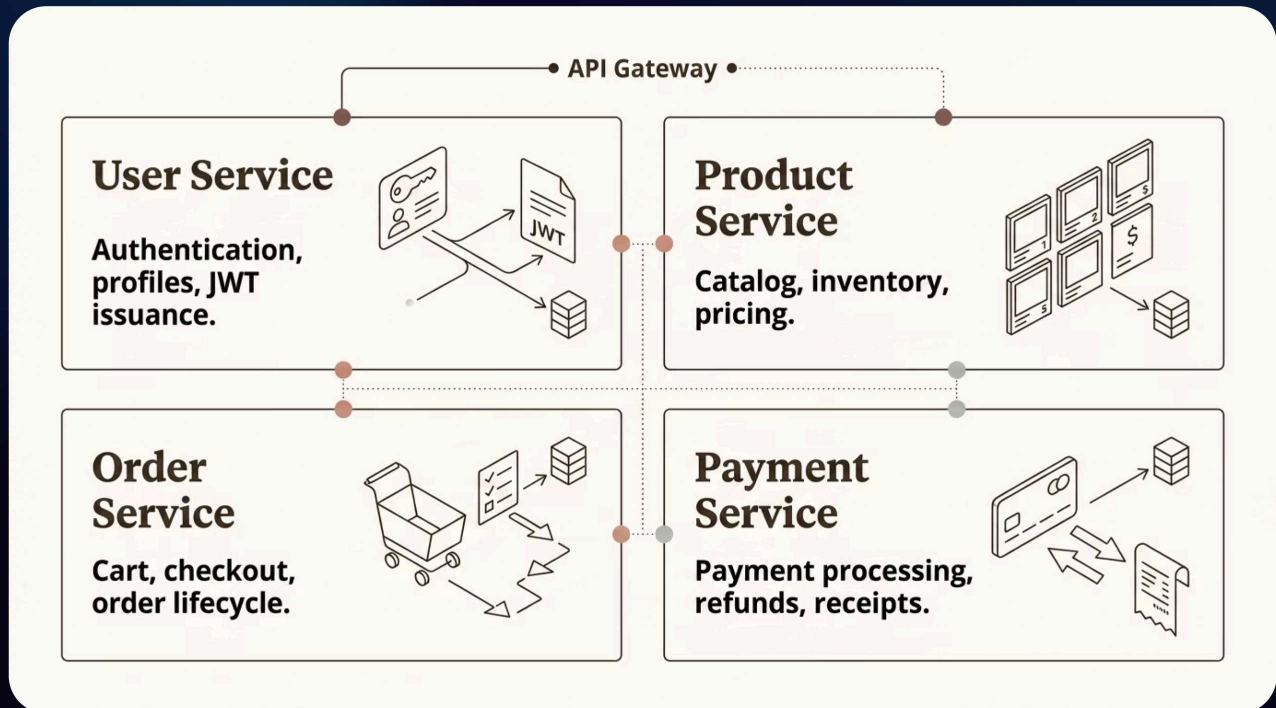
A large enterprise wants to convert a monolithic CRM application into a microservices architecture. This exercise introduces the architectural thinking required to decompose a tightly coupled system into independently deployable services. The goal is not simply to split the code — it is to align service boundaries with business capabilities and data ownership, so each service can be developed, deployed, and scaled independently without coordinating with other teams.

⚠ **Key Consideration:** Microservices introduce distributed systems complexity — network latency, partial failures, and eventual consistency. Decompose only when the organizational and scalability benefits outweigh this added complexity.

Microservices Design Exercise

ADVANCED EXERCISE — MICROSERVICES ARCHITECTURE

Designing a microservices architecture requires more than splitting code into separate Spring Boot applications. You must define clear **service boundaries** aligned with bounded contexts, establish **data ownership** so no two services share the same database, and choose appropriate **communication methods** — synchronous REST/gRPC for request-response interactions and asynchronous messaging (e.g., Kafka, RabbitMQ) for event-driven workflows.



When defining service boundaries, apply the **Single Responsibility Principle** at the service level: each service should be responsible for one business capability. The User Service owns identity and authentication data. The Product Service owns catalog and inventory. The Order Service orchestrates the checkout workflow. The Payment Service handles financial transactions and integrates with external payment gateways.

Project Deliverables

Architecture Diagram

Visual showing all four services, their databases, and communication paths.

Service Contracts

OpenAPI specifications or interface definitions for inter-service APIs.

API Gateway Design

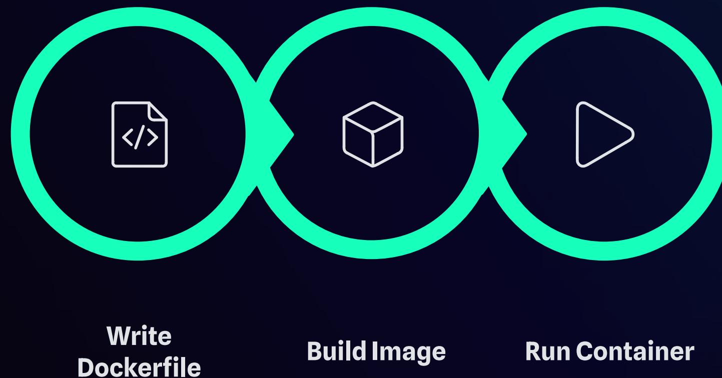
Routing rules, authentication enforcement, and rate limiting configuration.

Docker Implementation Exercise

DEVOPS MODULE

Containerization with Docker ensures that your application runs identically across development laptops, CI/CD pipelines, and production servers. The classic "it works on my machine" problem disappears when every environment uses the same container image. For Java full-stack applications, you will typically build separate containers for the Spring Boot backend, the Angular front-end (served via Nginx), and the MySQL database — orchestrated together using Docker Compose for local development.

Understanding Docker is no longer optional for enterprise Java developers. Containerization is the foundation upon which Kubernetes orchestration, cloud deployments, and modern CI/CD pipelines are built. Mastering the Dockerfile authoring process, image build optimization, and container lifecycle management is an essential professional skill.



The Dockerfile defines your application's runtime environment declaratively. For a Spring Boot application, you begin with an OpenJDK base image, copy the compiled JAR, expose the application port, and define the entrypoint command. Multi-stage Dockerfiles can further reduce image size by separating the build environment from the runtime environment.

Exercise

Document the full container lifecycle for your Employee Management System: write the Dockerfile, build the image, run the container locally, and map environment variables for database connection. Record the commands used at each step and note any troubleshooting observations.

Deliverable: A Docker Deployment Guide including your Dockerfile, docker-compose.yml, build and run commands, and notes on environment variable management for different deployment targets.

CI/CD Pipeline Exercise

DEVOPS MODULE

Continuous Integration and Continuous Deployment (CI/CD) transforms software delivery from a stressful, manual event into a reliable, automated process. When a developer pushes code to the repository, a well-designed pipeline automatically builds, tests, packages, and deploys the application — providing rapid feedback on whether the change broke anything and reducing the time from code completion to production availability.

For Java full-stack applications, the pipeline typically uses tools like **GitHub Actions**, **Jenkins**, or **GitLab CI** to orchestrate the stages. Maven or Gradle handle compilation and test execution. Docker builds and pushes the image to a registry. Deployment targets may include a cloud VM, Kubernetes cluster, or a managed container service like AWS ECS or Azure Container Apps.

Source Code Commit

Developer pushes to the main branch or opens a pull request. Pipeline is triggered automatically by the webhook.

Build

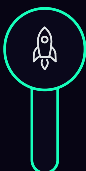
Maven compiles the Java code, resolves dependencies, and produces the application JAR artifact.

Test

JUnit unit tests and Spring Boot integration tests execute. Pipeline fails fast if any test fails.

Package

Docker image is built from the Dockerfile and pushed to the container registry with a versioned tag.



Deploy

The new container image is pulled and the running service is updated with zero-downtime rolling deployment.

Exercise: Design a complete CI/CD workflow for the Employee Management System. Identify the tools you would use at each pipeline stage, the conditions under which the pipeline should fail, and how you would handle rollback if a bad deployment is detected.

Security Implementation Workbook

SECURITY MODULE

Security is a cross-cutting concern that must be addressed at every layer of a full-stack application. In the enterprise Java ecosystem, **Spring Security** provides the foundational framework for authentication and authorization, while complementary standards like **JWT** (JSON Web Tokens) enable stateless, scalable authentication across distributed microservices.

This workbook section consolidates security implementation exercises across all five projects. Rather than treating security as an afterthought, treat each exercise here as a design discipline — plan your security model before writing application code, because retrofitting security onto an insecure system is far more costly than designing it in from the start.

Authentication Exercise

Implement a complete login and logout flow using Spring Security and JWT. Key tasks:

- Configure Spring Security filter chain
- Implement `UserDetailsService`
- Generate and validate JWT tokens
- Hash passwords with BCrypt
- Implement token refresh logic

Authorization Exercise

Create role-based access control governing what authenticated users may do. Key tasks:

- Define roles: ADMIN, MANAGER, USER
- Annotate controllers with `@PreAuthorize`
- Restrict data access at the repository level
- Test authorization with security-aware unit tests

Security Review Checklist

Security Control	Implementation Method	Status
Password Encryption	BCrypt with strength factor ≥ 10	�
JWT Authentication	Spring Security + JJWT library	�
Input Validation	Bean Validation / `@Valid`	�
HTTPS Enforcement	SSL certificate + HTTP redirect	�
CORS Configuration	Spring Security CORS policy	�
SQL Injection Prevention	Parameterized queries / JPA	�

API Development Workbook

API MODULE

REST API design is both a technical discipline and a craft. Well-designed APIs are intuitive to consume, easy to version, and resilient to change. Poorly designed APIs become maintenance burdens — with inconsistent naming, confusing status codes, and opaque error messages that waste developer time. This workbook section focuses on applying REST best practices across the domains covered in the five projects: User Management, Product Management, Orders, and Payments.

As you work through the exercises here, approach each API design decision from two perspectives: the **API producer** (the Spring Boot developer building the endpoint) and the **API consumer** (the Angular developer or third-party system calling it). Designing from the consumer's perspective leads to APIs that are genuinely pleasant to work with and reduces integration friction.

User Management APIs

Registration, login, profile retrieval, and password reset. Focus on clear error responses for auth failures and proper use of 401 vs 403 status codes.

Product Management APIs

CRUD operations for product catalog with pagination, filtering, and sorting. Practice returning consistent paginated response envelopes.

Orders & Payments APIs

Transactional endpoints requiring idempotency keys for payment operations. Design for retry safety and partial failure scenarios.

REST API Design Exercise

API MODULE — DESIGN PRACTICE

Apply the REST API best practices covered in this workbook to design complete API specifications for the four domains below. For each domain, define the full set of endpoints, select appropriate HTTP methods, design request and response JSON schemas, and document error conditions. Use **OpenAPI 3.0 (Swagger)** as your documentation format — it is the industry standard and integrates directly with Spring Boot via the `springdoc-openapi` library.

Pay particular attention to URL design. Resources should be nouns, not verbs: prefer `/orders/{id}/cancel` over `/cancelOrder`. Use plural nouns for collections. Nest sub-resources logically, but avoid nesting deeper than two levels to keep URLs manageable. HTTP methods carry semantic meaning — use them correctly: GET for reads, POST for creates, PUT/PATCH for updates, DELETE for removals.

User Management

- POST /users/register — Create new user account
- POST /auth/login — Authenticate and receive JWT
- GET /users/{id} — Retrieve user profile
- PUT /users/{id} — Update profile information

Product Management

- GET /products — List with pagination and filters
- POST /products — Create new product (Admin only)
- PUT /products/{id} — Update product details
- DELETE /products/{id} — Remove from catalog

Orders

- POST /orders — Place new order from cart
- GET /orders/{id} — Retrieve order details
- POST /orders/{id}/cancel — Cancel pending order
- GET /users/{id}/orders — Customer order history

Payments

- POST /payments — Initiate payment for an order
- GET /payments/{id} — Retrieve payment status
- POST /payments/{id}/refund — Initiate refund

- **✓ Proper HTTP Methods**

  Match HTTP verb semantics: GET (read), POST (create), PUT (replace), PATCH (partial update), DELETE (remove).

- **✓ Meaningful URLs**

  Resource-oriented, noun-based, hierarchical, lowercase with hyphens for readability.

- **✓ Input Validation**

  Validate at controller entry using `@Valid` with Bean Validation annotations. Return 400 with field-level error details.

- **✓ Standardized Responses**

  Consistent response envelope: `data`, `message`, `status`, and optional `pagination` metadata.

- **✓ Error Handling**

  Global exception handler using `@ControllerAdvice`. Map exceptions to appropriate HTTP status codes with descriptive messages.

Database Design Workbook

DATABASE MODULE

A solid database design is the foundation upon which a reliable, performant application is built. Poor schema design — missing indexes, denormalized data, absent foreign key constraints — causes cascading problems that are expensive to fix once an application is in production. This workbook section provides structured design exercises across three of the five project domains, with a focus on applying normalization principles, designing for query performance, and establishing referential integrity.

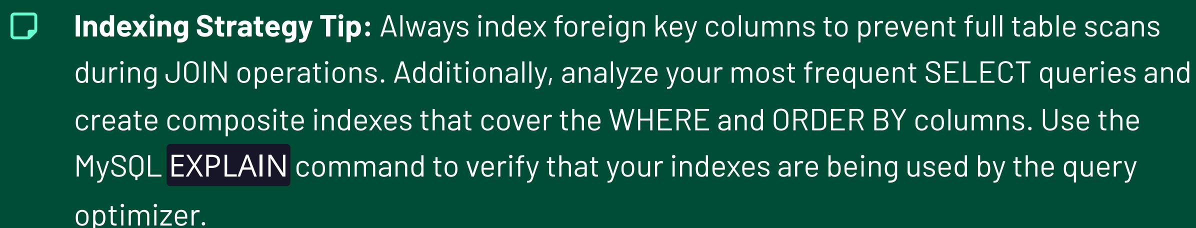
When designing schemas, work through normalization methodically. First Normal Form (1NF) eliminates repeating groups. Second Normal Form (2NF) removes partial dependencies on composite keys. Third Normal Form (3NF) eliminates transitive dependencies. Most enterprise schemas should target 3NF, with selective denormalization applied only where performance measurements justify it.

Design Exercises

- **HR System:** Employee, Department, Position, and Salary History tables
- **E-Commerce Platform:** Customer, Product, Category, Order, OrderItem, and Payment tables
- **CRM Application:** Lead, Contact, Account, Opportunity, and Activity tables

Schema Review Checklist

- All tables normalized to 3NF
- Primary keys defined on every table
- Foreign keys with appropriate cascade rules
- Indexes on all foreign key columns
- Composite indexes for common query patterns
- Appropriate column types and lengths chosen
- NOT NULL constraints applied where appropriate
- Unique constraints on natural key fields



Indexing Strategy Tip: Always index foreign key columns to prevent full table scans during JOIN operations. Additionally, analyze your most frequent SELECT queries and create composite indexes that cover the WHERE and ORDER BY columns. Use the MySQL `EXPLAIN` command to verify that your indexes are being used by the query optimizer.

Performance Optimization Exercise

PERFORMANCE MODULE

Users have reported that application response times have degraded significantly. What was once a snappy experience now feels sluggish and unresponsive. Performance issues in full-stack Java applications rarely have a single cause — they typically result from multiple contributing factors across the front-end, back-end, and database layers that compound each other. Effective performance engineering requires systematic investigation at each tier before jumping to conclusions or applying premature optimizations.

Begin with **measurement**: you cannot optimize what you have not measured. Use APM tools (Application Performance Monitoring) like New Relic, Datadog, or Spring Boot Actuator with Micrometer to establish baseline metrics. Identify which specific operations are slow before touching any code. A disciplined approach — measure, hypothesize, change one thing, measure again — is far more effective than guessing.

Front-End Issues

- Large Angular components loading too much data on initialization
- Unnecessary re-rendering due to poorly configured change detection
- Unoptimized images and missing lazy loading for modules
- Excessive HTTP calls — consolidate with backend-for-frontend patterns

Backend Issues

- Slow synchronous API operations blocking request threads
- Missing caching for frequently read, rarely changed data
- N+1 query problems from improper Hibernate fetch strategies
- No pagination on large collection endpoints

Database Issues

- Missing indexes on filtered and joined columns
- Inefficient queries returning more data than needed
- No query result caching for stable reference data
- Connection pool misconfiguration causing contention

Exercise: Given the three investigation areas above, prioritize which layer you would investigate first and justify your choice. Then list three specific code changes or configuration adjustments you would make at each layer to address the identified issues. Use SQL `EXPLAIN` output and Spring Boot Actuator metrics to support your hypotheses.

Troubleshooting Lab

DEBUGGING MODULE

Troubleshooting is a systematic problem-solving skill that separates senior engineers from beginners. Rather than randomly trying fixes, effective troubleshooting follows a disciplined process: observe symptoms, gather data, form hypotheses, test hypotheses with targeted changes, and verify the fix. The two case studies below present realistic production scenarios that Java full-stack developers encounter regularly.

Case Study 1: API Response Degradation

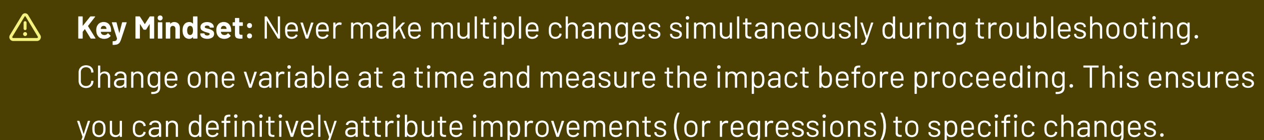
API response time increases from 200ms to 3 seconds over a 48-hour period with no code deployments.

- **What metrics should be reviewed?** Database query execution time, JVM heap usage, thread pool utilization, and connection pool wait time.
- **What logs should be examined?** Spring Boot application logs for slow query warnings, Hibernate SQL logs with timing, and database slow query log.
- **How would you identify root cause?** Correlate the timing of degradation with database growth, check for missing indexes on recently added query patterns, and review whether a scheduled job is causing lock contention.

Case Study 2: Application Crashes Under Traffic

The application becomes unresponsive and crashes when concurrent users exceed 200 simultaneous sessions.

- **What scaling options exist?** Horizontal scaling (multiple application instances behind a load balancer), vertical scaling (larger JVM heap allocation), and read replicas for database load.
- **What monitoring metrics help?** JVM garbage collection frequency, thread pool queue depth, connection pool exhaustion rate, and CPU/memory utilization per pod.
- **What architecture improvements apply?** Introduce connection pooling with HikariCP, add a Redis cache tier, implement rate limiting at the API gateway, and consider async processing for non-critical operations.



⚠ **Key Mindset:** Never make multiple changes simultaneously during troubleshooting. Change one variable at a time and measure the impact before proceeding. This ensures you can definitively attribute improvements (or regressions) to specific changes.

Project Review Template

FINAL REVIEW

Use this comprehensive review template to evaluate the completeness and quality of each project submission in this workbook. A thorough review covers four dimensions: architecture and design quality, testing coverage, deployment readiness, and security posture. Projects that score well across all four dimensions are ready for a professional code review and production deployment consideration.

This template is also useful for peer reviews and instructor evaluations. Go through each checklist item systematically. Items marked incomplete are not simply boxes left unchecked — each represents a gap that would create risk in a production environment. Use incomplete items as learning targets for revision.

Architecture Review

Area	Completed
Front-End Design	☐
API Design	☐
Database Design	☐
Security Design	☐

Testing Review

Area	Completed
Unit Tests	☐
Integration Tests	☐
API Testing	☐
UI Testing	☐

Deployment Review

Area	Completed
Dockerization	☐
CI/CD Pipeline	☐
Monitoring Setup	☐
Backup Strategy	☐

Completion Standard: All checklist items should be completed before a project is considered production-ready. Partial completion is acceptable during learning phases, but document which gaps remain and create a specific action plan for addressing each one.

5

Full-Stack Projects

End-to-end applications covering HR, E-Commerce, LMS, Inventory, and CRM domains.

4

Review Dimensions

Architecture, Testing, Deployment, and Security reviewed on every project.

3

Deployment Skills

Docker, CI/CD pipelines, and cloud-ready microservices architecture.

1

Complete Workbook

A comprehensive portfolio demonstrating enterprise Java full-stack readiness.

CERTIFIED JAVA FULL STACK DEVELOPER (CFSD)

ABOUT GSDC CERTIFICATION

EBOOK

Extensive and exclusive Ebook created by world's experts to help you with understanding core concepts.

LEARNING MATERIALS

Get access to learning materials such as videos, ebooks, templates, and practice exams, which will help you clear the certification exam.

CREATED BY EXPERTS

GSDC certifications are created and authored by world's leading experts in the field.

LEARNING OBJECTIVE

- Gain insights into autonomous decision-making processes
- Apply knowledge using ready-to-implement templates
- Demonstrate ability to work with Agentic AI models
- Validate your skills wit

Enroll now with the code **LEARN20** To avail **20%** discount

Enroll Now

www.gsdcouncil.org