Phase 1 Complete Learning Guide

Domino's Restaurant Management System - Understanding Every Concept

Learning Objectives

By the end of this guide, you'll understand:

- **Why** we need so many files and modules
- **How** Java, Spring Boot, MongoDB, and Redis work together
- **What** each syntax means and when to use it
- **Where** each piece fits in the bigger picture

Part 1: The Big Picture - Why So Many Files?

Think of Building a Restaurant Chain

Question for you: If you were opening 100 Domino's restaurants, would you: A) Build everything in one giant building with all kitchens, offices, and storage mixed together? B) Create separate specialized buildings that work together?

You'd choose B, right? That's exactly why we have multiple files and modules!

Our File Structure Explained

```
restaurant-management-system/
├── pom.xml                    ← "Corporate Headquarters" (manages everything)
├── shared-models/             ← "Company Standards Manual" (rules for all)
│   ├── pom.xml                ← "Department budget"
│   └── src/main/java/
│       └── User.java          ← "Employee handbook template"
├── api-gateway/               ← "Reception Desk" (routes customers)
│   ├── pom.xml                ← "Reception department budget"
│   ├── Dockerfile             ← "Instructions to build reception"
│   └── src/main/java/
├── docker-compose.yml         ← "Building blueprint" (infrastructure)
└── .gitignore                 ← "What not to share publicly"
```

Why separate files?

1. **Maintainability**: Fix reception without affecting kitchen

2. **Scalability**: Add more receptions without changing recipes
3. **Team Collaboration**: Different teams work on different parts
4. **Deployment**: Update one service without stopping others

---

Part 2: Understanding Java Syntax and Concepts

2.1 Classes and Objects - The Building Blocks

Think of a class as a blueprint, an object as the actual building.

java

```java
// This is a CLASS - a blueprint for creating User objects
public class User {
    // FIELDS (instance variables) - what every user HAS
    private String id;          // Private = only this class can access
    private UserType type;      // UserType is an enum (we'll learn this below)

    // CONSTRUCTOR - how to build a User object
    public User() {             // Default constructor (no parameters)
        this.createdAt = LocalDateTime.now();  // Set creation time
    }

    public User(UserType type, String name) {   // Custom constructor
        this.type = type;       // 'this' refers to current object
        this.createdAt = LocalDateTime.now();
    }

    // METHODS - what a User can DO
    public String getId() {     // Getter method
        return id;              // Return the private field
    }

    public void setId(String id) {  // Setter method
        this.id = id;           // Set the private field
    }

    // BUSINESS LOGIC METHODS
    public boolean canTakeOrders() {
        // This method shows POLYMORPHISM - same method, different behavior
        return type == UserType.MANAGER || type == UserType.ASSISTANT_MANAGER;
    }
}
```

Key Java Concepts Here:

- **Encapsulation**: Private fields, public methods (controlled access)
- **Constructor Overloading**: Multiple ways to create objects
- **Method Overloading**: Same method name, different parameters
- **this keyword**: Refers to current object instance

2.2 Enums - Why We Use Them

Question: Why not just use strings like "MANAGER", "STAFF"?

```java
// BAD APPROACH - Using strings
String userType = "MANAGR"; // Typo! This will cause bugs
if (userType.equals("MANAGER")) { // This won't match due to typo
    // This code won't run!
}

// GOOD APPROACH - Using enums
public enum UserType {
    CUSTOMER,          // These are CONSTANTS
    STAFF,             // Compiler checks spelling
    DRIVER,            // IDE gives auto-completion
    MANAGER,           // Type-safe at compile time
    ASSISTANT_MANAGER  // No runtime errors from typos
}

// Usage - compiler prevents mistakes
UserType type = UserType.MANAGER; // IDE helps with auto-complete
if (type == UserType.MANAGER) {   // Type-safe comparison
    // This works perfectly!
}
```

Benefits of Enums:

1. **Type Safety**: Compiler catches mistakes
2. **Auto-completion**: IDE helps you write code
3. **Documentation**: Clear, finite set of values
4. **Performance**: Faster than string comparisons

2.3 Nested Classes - Organizing Related Data

```java
```

```java
public class User {
    // Why nested classes? Related data should stay together!

    public static class PersonalInfo { // STATIC = doesn't need User instance
        private String name;
        private String email;

        // This class groups personal data together
        // Instead of having separate fields scattered in User class
    }

    public static class EmployeeDetails { // Another logical grouping
        private String storeId;
        private String role;

        // Employee-specific data grouped together
    }
}
```

Why nested classes?

1. **Logical Grouping**: Related fields stay together
2. **Namespace Organization**: Avoid naming conflicts
3. **Encapsulation**: Hide implementation details
4. **Code Readability**: Clearer structure

Part 3: Spring Boot - Understanding the Magic

3.1 What is Spring Boot Really?

Traditional Java Application:

```java
```

```
// OLD WAY - You had to do everything manually
public class TraditionalApp {
    public static void main(String[] args) {
        // Manually create web server
        Server server = new Server(8080);

        // Manually configure database connection
        DataSource dataSource = new DataSource("mongodb://...");

        // Manually wire dependencies
        UserService userService = new UserService(dataSource);
        UserController controller = new UserController(userService);

        // Manually start everything
        server.start();
    }
}
```

Spring Boot Way:

java

```
// NEW WAY - Spring Boot does the heavy lifting
@SpringBootApplication  // This annotation does A LOT!
public class ApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
        // That's it! Spring Boot auto-configures everything
    }
}
```

3.2 Understanding Spring Boot Annotations

java

```java
@SpringBootApplication // This combines 3 annotations:
// 1. @Configuration - "This class has configuration settings"
// 2. @EnableAutoConfiguration - "Automatically configure based on dependencies"
// 3. @ComponentScan - "Look for other Spring components in this package"

public class ApiGatewayApplication {

    @Bean // "Spring, please manage this object for me"
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        // Spring automatically provides 'builder' parameter (Dependency Injection)
        return builder.routes()
            .route("health-check", r -> r.path("/api/health")
                .uri("http://localhost:8080/health"))
            .build();
    }
}
```

What @Bean does:

1. **Object Management**: Spring creates and manages this object
2. **Dependency Injection**: Spring provides parameters automatically
3. **Lifecycle Management**: Spring handles creation/destruction
4. **Singleton Pattern**: By default, Spring creates one instance

3.3 Spring's Dependency Injection Magic

java

```
// Traditional way - manual dependency management
public class UserController {
    private UserService userService;

    public UserController() {
        this.userService = new UserService();  // Tight coupling!
        // If UserService constructor changes, this breaks
    }
}

// Spring way - dependency injection
@RestController  // Spring manages this
public class UserController {
    private final UserService userService;

    // Constructor injection - Spring provides UserService automatically
    public UserController(UserService userService) {
        this.userService = userService;  // Loose coupling!
    }

    // Or use field injection
    @Autowired
    private UserService userService; // Spring injects this automatically
}
```

Benefits of Dependency Injection:

1. **Loose Coupling**: Classes don't create their dependencies
2. **Easy Testing**: Can inject mock objects for testing
3. **Configuration**: Change implementations without code changes
4. **Lifecycle Management**: Spring handles object creation/destruction

---

Part 4: MongoDB - Document Database Deep Dive

4.1 Why MongoDB Instead of MySQL?

Relational Database (MySQL) Approach:

```
sql
```

```sql
-- Multiple tables with relationships
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    type ENUM('CUSTOMER', 'STAFF')
);

CREATE TABLE addresses (
    id INT PRIMARY KEY,
    user_id INT,
    street VARCHAR(200),
    city VARCHAR(100),
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE preferences (
    id INT PRIMARY KEY,
    user_id INT,
    preference_type VARCHAR(50),
    preference_value TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

-- To get complete user data, you need JOINs
SELECT u.*, a.*, p.*
FROM users u
LEFT JOIN addresses a ON u.id = a.user_id
LEFT JOIN preferences p ON u.id = p.user_id
WHERE u.id = 123;
```

MongoDB (Document) Approach:

javascript

```
// Single document with all related data
{
  "_id": "user123",
  "type": "CUSTOMER",
  "personalInfo": {
    "name": "John Doe",
    "email": "john@example.com",
    "address": {          // Embedded document
      "street": "123 Main St",
      "city": "Mumbai",
      "pincode": "400001"
    }
  },
  "preferences": {        // Another embedded document
    "favoriteOrders": ["pizza1", "pizza2"],
    "paymentMethods": ["RAZORPAY", "CASH"],
    "dietaryRestrictions": {
      "vegetarian": true,
      "allergies": ["nuts"]
    }
  }
}

// To get complete user data - single query!
db.users.findOne({"_id": "user123"})
```

4.2 MongoDB Java Integration

java

```java
@Document(collection = "users") // Maps to MongoDB collection
public class User {
    @Id // MongoDB's _id field
    private String id; // MongoDB generates this automatically

    @Field("type") // Custom field mapping (optional)
    @Indexed       // Creates database index for fast queries
    private UserType type;

    @Field("personalInfo")
    private PersonalInfo personalInfo; // Embedded document

    // MongoDB stores this as nested JSON automatically
}
```

MongoDB Annotations Explained:

- **@Document**: Maps Java class to MongoDB collection
- **@Id**: Marks the MongoDB _id field
- **@Field**: Custom field name mapping
- **@Indexed**: Creates database index for performance
- **@DBRef**: Reference to another document (like foreign key)

4.3 MongoDB Queries in Java

java

```java
// Spring Data MongoDB Repository
public interface UserRepository extends MongoRepository<User, String> {
    // Spring generates implementation automatically!

    // Query methods - Spring converts method names to MongoDB queries
    List<User> findByType(UserType type);
    // Becomes: db.users.find({"type": "MANAGER"})

    Optional<User> findByPersonalInfoEmail(String email);
    // Becomes: db.users.findOne({"personalInfo.email": "john@example.com"})

    List<User> findByPersonalInfoAddressCity(String city);
    // Becomes: db.users.find({"personalInfo.address.city": "Mumbai"})

    // Custom queries using @Query annotation
    @Query("{'type': ?0, 'personalInfo.address.city': ?1}")
    List<User> findByTypeAndCity(UserType type, String city);
}
```

Spring Data Magic:

1. **Method Name Queries**: Converts method names to database queries
2. **Automatic Implementation**: No need to write query code
3. **Type Safety**: Compile-time checking of field names
4. **Custom Queries**: @Query annotation for complex queries

Part 5: Redis - Caching Layer Explained

5.1 Why Redis? Performance Comparison

java

```java
// Without Redis - Every request hits MongoDB
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User getUserById(String id) {
        return userRepository.findById(id); // Database query every time
        // ~50ms per query
    }
}

// With Redis - Cache frequently accessed data
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    @Cacheable("users") // Spring automatically caches return value
    public User getUserById(String id) {
        // First call: queries database and caches result
        // Subsequent calls: returns from cache in ~1ms
        return userRepository.findById(id);
    }

    @CacheEvict("users") // Removes from cache when user is updated
    public User updateUser(User user) {
        return userRepository.save(user);
    }
}
```

5.2 Redis Data Types and Use Cases

java

```java
// Redis supports various data types
@Component
public class CacheService {
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    // Simple key-value caching
    public void cacheUser(String userId, User user) {
        redisTemplate.opsForValue().set("user:" + userId, user, 30, TimeUnit.MINUTES);
    }

    // List operations (for real-time order queue)
    public void addToOrderQueue(String orderId) {
        redisTemplate.opsForList().leftPush("kitchen:queue", orderId);
    }

    // Set operations (for online drivers)
    public void markDriverOnline(String driverId) {
        redisTemplate.opsForSet().add("drivers:online", driverId);
    }

    // Hash operations (for session data)
    public void storeSession(String sessionId, Map<String, String> sessionData) {
        redisTemplate.opsForHash().putAll("session:" + sessionId, sessionData);
    }
}
```

Part 6: Docker - Containerization Explained

6.1 Why Docker? The "It Works on My Machine" Problem

Without Docker:

```
Developer's Machine:
- Java 21
- MongoDB 7.0
- Redis 7.2
- Windows 11

Production Server:
- Java 17 (different version!)
- MongoDB 6.0 (older version!)
- Redis 6.0 (older version!)
- Ubuntu Linux (different OS!)

Result: "It works on my machine but not in production!"
```

With Docker:

dockerfile

```dockerfile
# Dockerfile - Exact same environment everywhere
FROM openjdk:21-jdk-slim  # Exact Java version

WORKDIR /app
COPY target/api-gateway-1.0.0.jar app.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

6.2 Understanding Dockerfile Syntax

dockerfile

```
# Every instruction creates a new layer

FROM openjdk:21-jdk-slim    # Base image (like inheriting from a class)
# "Start with an image that has Java 21 pre-installed"

WORKDIR /app                # Set working directory inside container
# "cd /app" inside the container

COPY target/api-gateway-1.0.0.jar app.jar
# Copy file from host machine to container
# Source: target/api-gateway-1.0.0.jar (on your computer)
# Destination: app.jar (inside container)

EXPOSE 8080                 # Document which port the app uses
# "This container listens on port 8080"

ENTRYPOINT ["java", "-jar", "app.jar"]
# Command to run when container starts
# Like double-clicking app.jar
```

6.3 Docker Compose - Orchestrating Multiple Services

yaml

```yaml
# docker-compose.yml - Define multiple services
services:
  mongodb:                    # Service name
    image: mongo:7.0          # Use pre-built MongoDB image
    container_name: dominos-mongodb  # Custom container name
    ports:
      - "27017:27017"         # Port mapping: host:container
    environment:              # Environment variables
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password123
    volumes:                  # Persistent storage
      - mongodb_data:/data/db                    # Named volume
      - ./init.js:/docker-entrypoint-initdb.d/init.js  # File binding
    networks:
      - dominos-network       # Custom network

  redis:
    image: redis:7.2-alpine   # Lightweight Redis image
    command: redis-server --appendonly yes --requirepass redispassword123
    # Override default startup command

  api-gateway:                # Our application service
    build: ./api-gateway      # Build from Dockerfile
    ports:
      - "8080:8080"
    depends_on:               # Start order dependency
      - mongodb
      - redis
    networks:
      - dominos-network

volumes:                      # Persistent data storage
  mongodb_data:               # MongoDB data survives container restarts
  redis_data:

networks:                     # Custom network for service communication
  dominos-network:
    driver: bridge            # Default Docker network type
```

Docker Compose Benefits:

1. **Multi-service Management**: Start/stop all services together

2. **Service Discovery**: Services can find each other by name
3. **Environment Consistency**: Same setup for all developers
4. **Easy Scaling**: `docker-compose up --scale api-gateway=3`

Part 7: Maven - Project Management Deep Dive

7.1 Why So Many pom.xml Files?

```
restaurant-management-system/
├── pom.xml                    ← Parent POM (Corporate Headquarters)
├── shared-models/
│   └── pom.xml                ← Child POM (Department budget)
├── api-gateway/
│   └── pom.xml                ← Child POM (Department budget)
└── user-service/
    └── pom.xml                ← Child POM (Department budget)
```

Think of it like a company structure:

- **Parent POM**: CEO decisions that affect everyone
- **Child POMs**: Department-specific decisions

7.2 Parent POM Explained

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>

  <!-- This project's identity -->
  <groupId>com.dominos</groupId>          <!-- Company identifier -->
  <artifactId>restaurant-management-system</artifactId>  <!-- Project name -->
  <version>1.0.0</version>               <!-- Version number -->
  <packaging>pom</packaging>             <!-- This is a parent project -->

  <!-- Global settings for all children -->
  <properties>
    <java.version>21</java.version>      <!-- All modules use Java 21 -->
    <spring-boot.version>3.2.0</spring-boot.version>  <!-- Spring Boot version -->
  </properties>

  <!-- Child modules -->
  <modules>
    <module>shared-models</module>       <!-- Build order matters! -->
    <module>api-gateway</module>
    <module>user-service</module>
  </modules>

  <!-- Version management for all children -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>${spring-boot.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

7.3 Child POM Explained

xml

```xml
<!-- api-gateway/pom.xml -->
<project>
  <!-- Inherit from parent -->
  <parent>
    <groupId>com.dominos</groupId>
    <artifactId>restaurant-management-system</artifactId>
    <version>1.0.0</version>
  </parent>

  <!-- This module's identity -->
  <artifactId>api-gateway</artifactId>    <!-- Inherits groupId and version -->

  <!-- Module-specific dependencies -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webflux</artifactId>
      <!-- Version comes from parent dependencyManagement -->
    </dependency>

    <!-- Depend on our shared models -->
    <dependency>
      <groupId>com.dominos</groupId>
      <artifactId>shared-models</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
</project>
```

Benefits of Multi-Module Structure:

1. **Shared Dependencies**: Common versions managed in parent
2. **Build Order**: Maven builds shared-models before api-gateway
3. **Code Reuse**: shared-models used by all services
4. **Independent Development**: Teams can work on different modules

---

Part 8: Understanding the Complete Flow

8.1 Application Startup Flow

java

```
// 1. JVM starts and loads ApiGatewayApplication class
public static void main(String[] args) {
    // 2. SpringApplication.run() starts Spring Boot
    SpringApplication.run(ApiGatewayApplication.class, args);
}

// 3. Spring Boot auto-configuration kicks in
@SpringBootApplication  // Triggers component scanning
public class ApiGatewayApplication {

    // 4. Spring creates beans defined by @Bean annotations
    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        // 5. Spring injects RouteLocatorBuilder dependency
        return builder.routes()
            .route("health-check", r -> r.path("/api/health")
                .uri("http://localhost:8080/health"))
            .build();
    }
}

// 6. Spring starts embedded Netty web server on port 8080
// 7. Application is ready to receive requests
```

8.2 Request Processing Flow

java

```
// 1. HTTP request comes to API Gateway (port 8080)
// GET /api/health

// 2. Spring Cloud Gateway RouteLocator matches request
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("health-check", r -> r.path("/api/health")  // Matches "/api/health"
            .uri("http://localhost:8080/health"))         // Routes to this URI
        .build();
}

// 3. Gateway forwards request to /health endpoint

// 4. @RestController handles the request
@RestController
public class ApiGatewayApplication {
    @GetMapping("/health")  // Matches GET /health
    public String health() {
        return "API Gateway is running successfully!";  // Returns response
    }
}

// 5. Response flows back: Controller → Gateway → Client
```

8.3 Future Data Flow (When we add User Service)

java

```
// 1. Client request: POST /api/users/register
// 2. API Gateway routes to User Service
// 3. User Service processes request:

@RestController
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private UserService userService;  // Spring injects this

    @PostMapping("/register")
    public ResponseEntity<User> register(@RequestBody UserRegistrationRequest request) {
        // 4. Controller calls service layer
        User user = userService.createUser(request);
        return ResponseEntity.ok(user);
    }
}

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;  // Spring injects this

    public User createUser(UserRegistrationRequest request) {
        // 5. Service calls repository layer
        User user = new User(request.getType(), request.getName());
        return userRepository.save(user);  // 6. Saves to MongoDB
    }
}

// 7. Response flows back: Repository → Service → Controller → Gateway → Client
```

Part 9: Why This Architecture?

9.1 Separation of Concerns

java

```
// BAD - Everything in one class
public class MonolithicUserHandler {
    public String handleUserRequest(String request) {
        // Parse HTTP request
        // Validate data
        // Business logic
        // Database operations
        // Format response
        // Handle errors
        // Log everything
        // All mixed together!
    }
}

// GOOD - Separated responsibilities
@RestController // Only handles HTTP requests/responses
public class UserController {
    public ResponseEntity<User> getUser(@PathVariable String id) {
        return ResponseEntity.ok(userService.findById(id));
    }
}

@Service // Only handles business logic
public class UserService {
    public User findById(String id) {
        return userRepository.findById(id).orElseThrow();
    }
}

@Repository // Only handles data access
public interface UserRepository extends MongoRepository<User, String> {
    // Spring Data handles implementation
}
```

9.2 Scalability Benefits

yaml

```yaml
# Can scale each service independently
version: '3.8'
services:
  api-gateway:
    replicas: 2      # 2 instances of gateway

  user-service:
    replicas: 5      # 5 instances if users service is heavily used

  order-service:
    replicas: 10     # 10 instances during peak hours

  mongodb:
    replicas: 3      # 3 database instances for high availability
```

9.3 Technology Benefits Summary

Technology	Purpose	Why This Choice
Java 21	Programming Language	Latest LTS, performance improvements, modern syntax
Spring Boot	Application Framework	Auto-configuration, dependency injection, production-ready
MongoDB	Database	Document model matches objects, flexible schema, scales horizontally
Redis	Caching	In-memory speed, persistence options, data structures
Docker	Containerization	Environment consistency, easy deployment, isolation
Maven	Build Tool	Dependency management, multi-module support, IDE integration

---

Part 10: Next Steps - What You've Learned

Concepts You Now Understand:

✅ **Java OOP**: Classes, objects, inheritance, polymorphism, encapsulation
✅ **Spring Boot**: Dependency injection, auto-configuration, annotations
✅ **MongoDB**: Document databases, embedded documents, indexing
✅ **Redis**: Caching strategies, data types, performance benefits
✅ **Docker**: Containerization, multi-service orchestration
✅ **Maven**: Multi-module projects, dependency management
✅ **Microservices**: Service boundaries, scalability, separation of concerns

Skills You Can Now Apply:

✅ Create multi-module Maven projects
✅ Design MongoDB document schemas
✅ Use Spring Boot annotations effectively
✅ Understand Docker containerization
✅ Implement caching strategies with Redis
✅ Build scalable microservices architectures

Ready for Phase 2: Now that you understand the foundation, you're prepared to implement the User Service with authentication, working hours tracking, and role-based access control!