irian

Software Engineering Par Excellence

Part II

Profiles

Properties

Layered Architecture

JPA

by Georgia Leustean

Spring Profiles

Environment-specific bean configuration and property injection

What are Spring Profiles?

- Segregate parts of your application configuration
- Activate beans based on environments
- Improve modularity and testing

Bean Implementations per Profile

```
public interface GreetingService {
    String greet();
}



@Service
@Profile("dev")
public class DevGreetingService implements GreetingService {
    @Override
    public String greet() {
        return "Hello from DEV!";
    }
}



@Service
@Profile("prod")
public class ProdGreetingService implements GreetingService {
    @Override
    public String greet() {
        return "Hello from PROD!";
    }
}
```

Activating Profiles

- application.properties: spring.profiles.active=dev
- VM argument: -Dspring.profiles.active=prod
- In tests: @ActiveProfiles

Ex 01

- Define one PizzaService for profile “dev” and one for profile “prod”
- Start the app with one profile at a time and see which pizzas appear for “get all” method and get fast pizza
- Start the app with no profile

Custom Properties in Spring Boot

Defining and Injecting Custom Properties into Beans

Defining Custom Properties

Add entries to application.properties:

```
myapp.feature.enabled=true
myapp.datasource.url=jdbc:mysql://localhost:3306/db
myapp.cache.ttl=600
```

Binding with @ConfigurationProperties

Define a POJO for grouped properties:

```
@Component
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {
   private boolean featureEnabled;
   private String datasourceUrl;
   private int cacheTtl;
   // getters and setters
}
```

Injecting with @Value

Use @Value for individual properties:

```
@Value("${myapp.feature.enabled}")
private boolean featureEnabled;

@Value("${myapp.datasource.url}")
private String datasourceUrl;
```

Using Properties in Beans

Example service injection:

```
@Service
public class FeatureService {
    private final MyAppProperties props;  // @Autowired via constructor

    public FeatureService(MyAppProperties props) {
        this.props = props;
    }

    public void runFeature() {
        if (props.isFeatureEnabled()) {
            // feature logic here
        }
    }
}
```

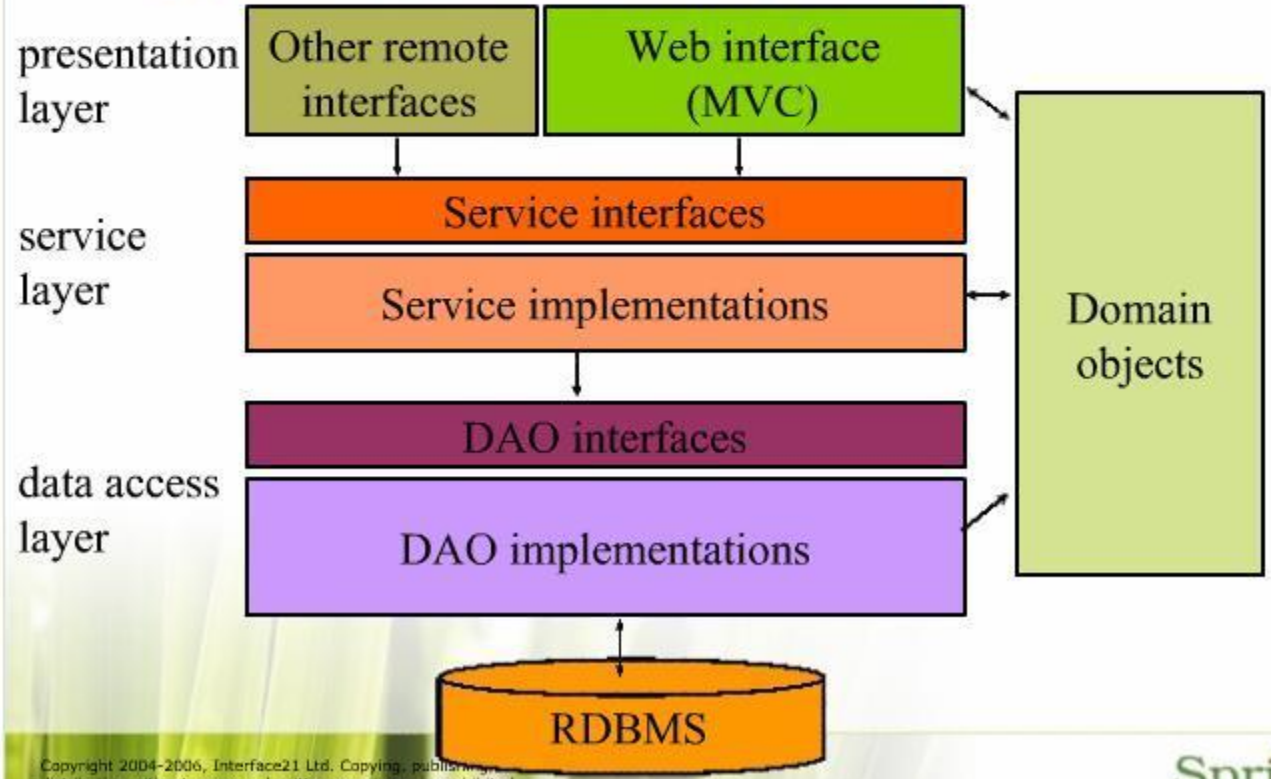
Ex 02

- Add a property pizza.names having comma separated values for names (Salami,Prosciuto, etc)
- Add a property pizza.prices with comma separated values for prices
- Read the pizza.names and pizza.prices in a class using @ConfigurationProperties and inject this class in PizzaService. Use the values from properties to create pizzas with prices
- Use a fastpizza.enabled property in the Controller to make the fast pizza endpoint return pizzas or throw an exception

Layered architecture

INTERFACE21

Typical application layering



Spring

Layered Architecture Overview

- Presentation Layer: Handles HTTP requests, UI rendering
- Service Layer: Business logic and service orchestration
- Repository/Data Access Layer: Database interactions (Spring Data)
- Domain Layer: Core domain models and entities

Example Layers in Spring

```
@RestController
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return userService.findAllUsers();
    }
}
```

Maven Basics & Dependencies

- POM: Project Object Model defines project structure
- Coordinates: groupId, artifactId, version
- Dependencies: External libraries required by the project

pom.xml Example

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>2.5.4</version>
    </dependency>
    <!-- other dependencies -->
  </dependencies>
</project>
```

Master POM & Modules

- Parent POM packaging: pom
- Defines common dependency management
- Declares modules for sub-projects

Parent POM Example

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>multi-module-parent</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <modules>
    <module>service</module>
    <module>web</module>
    <module>repository</module>
  </modules>
</project>
```

Creating New Modules

Navigate to parent project directory

Run: mvn archetype:generate -DgroupId=com.example.module -DartifactId=new-module -DarchetypeArtifactId=maven-archetype-quickstart

/OR copy-paste pom.xml of existing module in new folder and change artifactId

Add <module>new-module</module> to parent pom.xml

Implement module-specific code with its own pom

Project Structure Example

```
multi-module-parent/
 ├── pom.xml
 ├── service/
|    └── pom.xml
 ├── web/
|    └── pom.xml
└── repository/
    └── pom.xml
```

Dependency Inversion Principle

- High-level modules should not depend on low-level modules
- Both should depend on abstractions (interfaces)
- Abstractions should not depend on details
- Promotes flexible and decoupled design

Summary

- Layered architecture promotes separation of concerns
- Spring simplifies layer implementation
- Maven multi-module enables modular builds
- Parent POM centralizes configuration

Live code

- Create pizza-domain and pizza-service-api maven modules

Default Scanning Behavior

- Spring Boot scans from the package of the main application class downward.
- All sub-packages are included automatically for @Component and @Entity.
- No additional configuration needed when packages are nested under the main class’s package.

When Default Scanning Suffices

- Main class at com.example.app and beans under com.example.app.services, com.example.app.controllers.
- Entities under com.example.app.domain are detected automatically.
- Simplifies project structure by convention-over-configuration.

When to Customize Scanning

- Beans or entities located outside the main package hierarchy.
- Modular projects with separate root packages.
- Shared libraries or modules not nested under application package.

Using @ComponentScan & @EntityScan

```
@SpringBootApplication
@ComponentScan(basePackages = {
    "com.example.modules.web",
    "com.example.modules.service"
})
@EntityScan("com.example.modules.data")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Configuration via application.properties

- # component scan packages
- spring.main.sources=com.example.modules.web,com.example.modules.service

- # entity scan packages
- spring.jpa.entity.scan.packages=com.example.modules.data

Summary

- Default scanning works when application class sits above all packages.
- Use @ComponentScan/@EntityScan when beans/entities lie outside default path.
- Properties can also customize scan locations.
- Custom scanning ensures all modules are detected by Spring Boot.

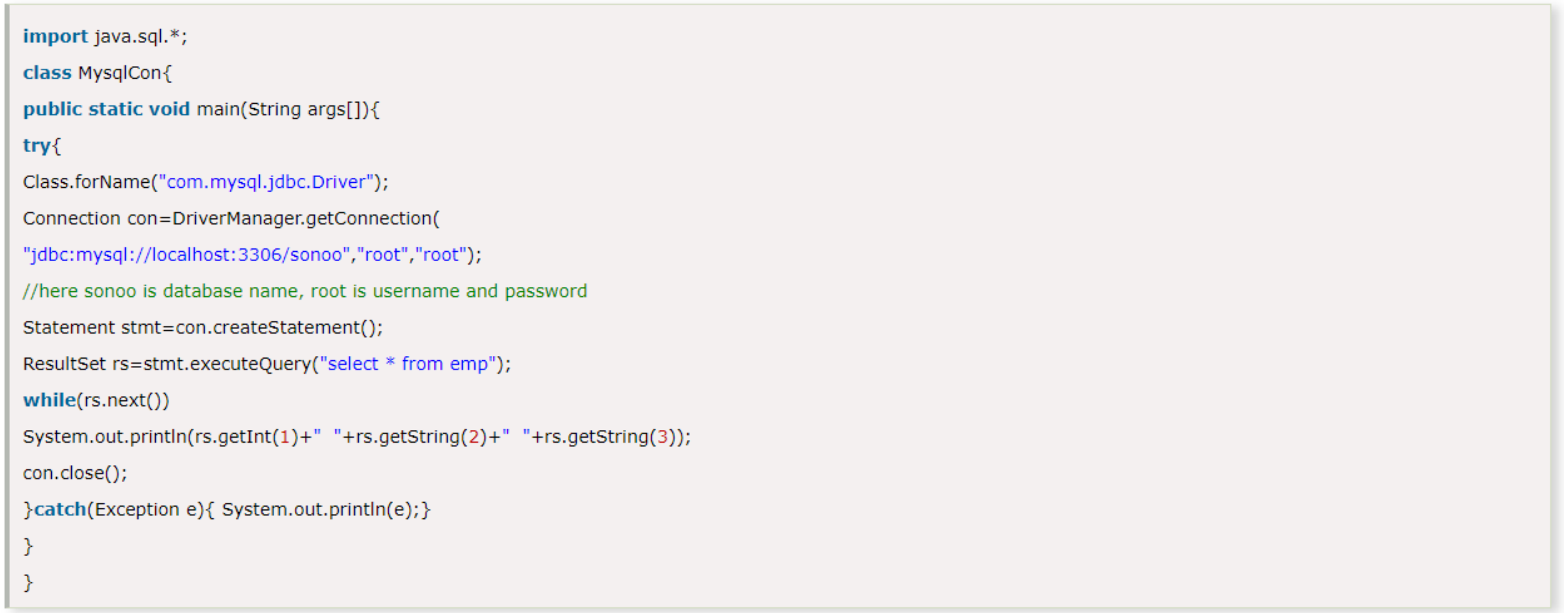
Ex 03

- Create a maven module for american-pizza-service
- Add dependency to pizza-domain
- Remove from pizza-web-lib module the Italian-pizza-service dependency and add the american-pizza-service: notice which beans are injected now at startup (which pizzas are retrieved on get all)

JPA, Hibernate & Entities

- What is ORM & JPA?
- Introduction to Hibernate
- Defining Entities
- Key Annotations: @Entity, @Table
- Annotations for Columns
- ID Generation Strategies
- Entity Relationships (Basics)
- Entity Lifecycle States
- Code Examples
- Summary & Q&A

JDBC Approach



```java
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

What is ORM & JPA?

- ORM (Object-Relational Mapping) lets you map Java objects to database tables.
- JPA (Java Persistence API) is the Java standard for ORM.
- JPA defines interfaces like EntityManager to manage entities.

Introduction to Hibernate

- Hibernate is a popular implementation of JPA.
- Adds features like advanced caching and lazy loading.
- Uses SessionFactory and Session for persistence operations.

Defining Entities

- An entity is a plain Java class (POJO) mapped to a database table.
- Use @Entity above the class.
- Use @Table to specify table name (optional).

Dependency

```xml
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Key Annotations: @Entity & @Table

```
@Entity
@Table(name = "users")    // maps to 'users' table
public class User {
    @Id
    private Long id;
    // class body
}
```

Annotations for Columns

- @Column: customize column mapping
- - name: custom column name
- - nullable: allow NULL (true/false)
- - unique: enforce unique values
- - length: max length for String
- - columnDefinition: SQL fragment

Column Annotation Example

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "product_name", nullable = false, length = 100)
    private String name;

    @Column(unique = true)
    private String sku;
}
```

ID Generation Strategies

- AUTO: let provider choose the best strategy
- IDENTITY: use auto-increment column
- SEQUENCE: use database sequence object
- TABLE: use a separate table for id generation

ID Generation Example

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "order_seq")
    @SequenceGenerator(name = "order_seq", sequenceName = "order_sequence",
                       allocationSize = 1)
    private Long id;

    // other fields
}
```

Entity Relationships (Basics)

- @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
- Use mappedBy on the non-owning side
- Cascading: propagate operations to related entities
- Fetch types: EAGER vs LAZY loading

Entity Lifecycle States

- NEW: created but not persisted
- MANAGED: tracked by EntityManager
- DETACHED: no longer tracked
- REMOVED: scheduled for delete

Sample Entity Class

```
@Entity
@Table(name = "customers")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String firstName;

    @Column(nullable = false)
    private String lastName;
    // getters & setters
}
```

Field Mapping Annotations

- @Enumerated: map enums, use STRING or ORDINAL
- @Lob: map large text (CLOB) or binary (BLOB)

@Enumerated & @Lob Example

```
public enum Status { ACTIVE, INACTIVE, DELETED }

@Entity
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    private Status status;

    @Lob
    @Column(columnDefinition = "TEXT")
    private String description;
}
```

Summary

- JPA standardizes ORM in Java
- Hibernate offers powerful features
- @Entity and @Table map classes/tables
- @Column customizes columns
- ID strategies control primary key generation
- Relationships and lifecycle basics

Ex 04.1

- Add dependency to spring-boot-starter-data-jpa
- Annotate Pizza class as an entity and try starting the application

H2 Database & Spring Boot

- What is H2 Database?
- Why Use H2?
- Adding H2 Dependency
- Configuring H2 in Spring Boot
- Enabling H2 Console
- Schema Initialization
- Code Example
- Summary & Q&A

What is H2 Database?

- Lightweight, open-source Java SQL database
- Supports in-memory and disk-based modes
- JDBC API compliant and easy to embed
- Ideal for development, testing, and demos

Why Use H2?

- Zero external setup—runs in-memory
- Fast startup and teardown
- Supports console for query testing
- No need for external DB server

Adding H2 Dependency

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

application.properties Configuration

```
# H2 Database Settings
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA Settings
spring.jpa.hibernate.ddl-auto=create
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Test data scripts logging Configuration
logging.level.org.springframework.jdbc.datasource.init.ScriptUtils=DEBUG
logging.level.org.springframework.boot.autoconfigure.jdbc.DataSourceInitializationConfiguration=DEBUG


# Fine-grained logging for Hibernate SQL & DDL
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.tool.schema.internal=DEBUG
```

Enabling H2 Console

```
# Enable Web Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Schema Initialization

- Place schema.sql and data.sql in src/main/resources
- Spring Boot runs them on startup
- Define tables and initial data

Summary & Q&A

- H2 is ideal for dev and testing
- Add H2 dependency at runtime scope
- Configure datasource and JPA settings
- Enable console for database inspection
- Use schema.sql/data.sql for init

Ex 04.2

- Configure h2 database
- Add data.sql file in resources with an sql script that inserts data in pizza table
- Start the app and check h2 console if pizzas have been inserted

What is Spring Data?

- Spring Data simplifies data access layers.
- Provides repository abstractions for various datastores (JPA, MongoDB, etc.).
- Reduces boilerplate CRUD code through interface definitions.
- Automatically implements common repository interfaces at runtime.

Defining a Repository Interface

```
public interface UserRepository
    extends JpaRepository<User, Long> {
    // Custom query method derived from method name
    List<User> findByLastName(String lastName);
}
```

Custom Query Methods with Keywords

Derived query methods: Spring Data parses method names

Keywords: And, Or, Between, LessThan, GreaterThan, Like

Sorting: OrderBy{Property}{Asc|Desc}, Top{N}, First{N}

```java
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastNameAndAgeGreaterThan(String lastName, int age);
    List<User> findByCreatedDateBetween(LocalDate start, LocalDate end);
    List<User> findTop5ByOrderByScoreDesc();
    boolean existsByEmail(String email);
}
```

Behind the Scenes: Implementation

- Spring Data creates a proxy for your interface at startup.
- The default implementation class is SimpleJpaRepository.
- This class uses an injected EntityManager to perform operations.
- Methods like save(), findAll(), delete() are executed via JPA.

The SimpleJpaRepository Class

```
public class SimpleJpaRepository<T, ID>
  implements JpaRepository<T, ID> {
    private final EntityManager em;

    public SimpleJpaRepository(JpaEntityInformation<T, ?> info,
                               EntityManager em) {
        this.em = em;
    }

    public <S extends T> S save(S entity) {
        if (entityInformation.isNew(entity)) {
            em.persist(entity);
            return entity;
        } else {
            return em.merge(entity);
        }
    }
    // Other CRUD methods use em.find(), em.remove(), etc.
}
```

Using Repositories in Services

```
@Service
public class UserService {
    private final UserRepository repo;

    @Autowired
    public UserService(UserRepository repo) {
        this.repo = repo;
    }

    public List<User> getAllUsers() {
        return repo.findAll();
    }

    public User createUser(User user) {
        return repo.save(user);
    }
}
```

Summary

- Define repositories as interfaces extending JpaRepository.
- Spring Data provides runtime implementations via SimpleJpaRepository.
- SimpleJpaRepository uses EntityManager for all data operations.
- Inject repositories into services or controllers with @Autowired.

Ex 05

- Create PizzaRepository
- Inject into PizzaService implementation
- Refactor service methods to return pizzas from database
- Refactor controller to only use one PizzaService implementation