WEB DESIGN AND PROGRAMMING

SPRING BOOT DATABASES
5

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

ASST. PROF. DR. SAMET KAYA

skaya@fsm.edu.tr

kysamet@gmail.com

Front-End Technologies Overview

- Spring Boot is a powerful framework that greatly simplifies database operations in modern Java applications.
- It provides seamless integration with data persistence tools like Spring Data JPA and Hibernate.
- This allows developers to focus on business logic rather than writing boilerplate code for database connections and queries.
- **Key Goals:**
  - Simplify database configuration.
  - Implement robust CRUD (Create, Read, Update, Delete) operations.
  - Effectively model and manage relational data.

Database Design Approaches

- When developing database applications, there are three primary approaches to manage the relationship between your application code and the database schema.

- Each approach has distinct advantages and is suited for different project types.

1. **Code First**
2. **Database First**
3. **Model First**

Approach 1: Code First

- **Definition:**
  - You begin by writing your Java entity classes. Then, a tool like Hibernate automatically generates the database schema based on your code.
- **Flow:**
  - Java Entity Classes → Hibernate/JPA → Database Tables
- **Key Features:**
  - Developers focus on object-oriented code, not SQL.
  - The database schema is managed automatically.
  - Promotes database-agnostic development.
  - Ideal for Agile methodologies and rapid prototyping.
- **Best For:** New projects, startups, MVPs, and microservices.

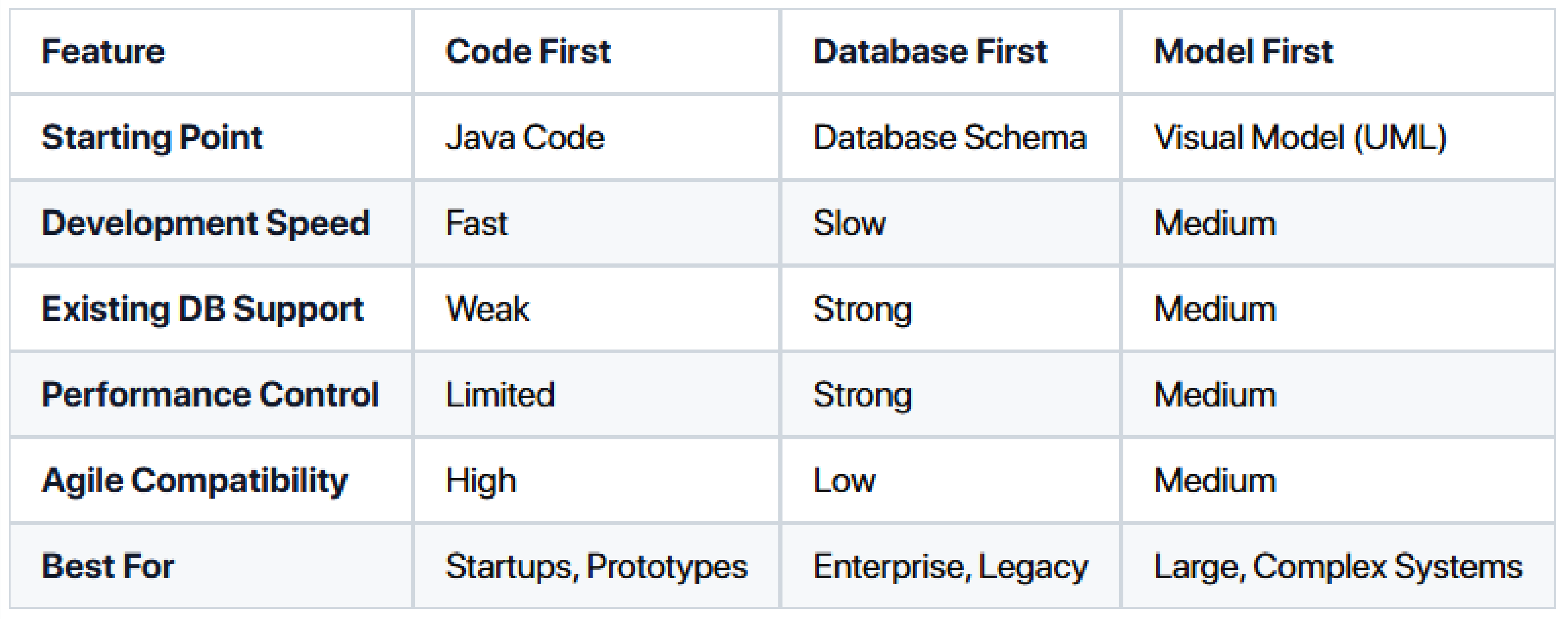
Approach 2: Database First

- **Definition:**
  - You start by designing and creating the database schema first. Then, entity classes are automatically generated from the existing database schema.
- **Flow:**
  - Database Tables → Reverse Engineering → Java Entity Classes
- **Key Features:**
  - Gives database administrators (DBAs) full control over the schema.
  - Ideal for projects with existing or complex databases.
  - Allows for fine-tuned performance optimizations at the database level.
- **Best For:** Enterprise applications, projects with legacy databases, and systems where database performance is critical.

Approach 3: Model First

- **Definition:**
  - You start with a visual model, like a UML diagram. From this model, both the database schema and the Java entity classes are generated.
- **Flow:**
  - UML/Visual Model → (Database Schema + Java Entity Classes)
- Key Features:
  - Provides a high-level, abstract view of the system.
  - Keeps code and database synchronized with the design.
  - Excellent for documentation and team collaboration.
- **Best For:** Large, complex, long-term projects where architectural design and clear documentation are paramount.

Comparison of Approaches



Feature	Code First	Database First	Model First
Starting Point	Java Code	Database Schema	Visual Model (UML)
Development Speed	Fast	Slow	Medium
Existing DB Support	Weak	Strong	Medium
Performance Control	Limited	Strong	Medium
Agile Compatibility	High	Low	Medium
Best For	Startups, Prototypes	Enterprise, Legacy	Large, Complex Systems

Which Approach to Choose?

- **Choose Code First:**
  - Starting a new project
  - Fast development is important
  - Database independence required
- **Choose Database First:**
  - Working with existing database
  - Database performance is critical
  - Complex database schemas exist
- **Choose Model First:**
  - Large and complex projects
  - Team collaboration is important
  - Architectural design is critical
  - Long-term projects

Core Concepts: ORM, JPA, Hibernate

- **ORM (Object-Relational Mapping):**
  - A technique that acts as a "bridge" between your object-oriented Java code and a relational database.
  - It maps database tables to Java classes.
- **JPA (Java Persistence API):**
  - A specification (a standard set of rules and interfaces) for ORM in Java.
  - It defines the "how-to" for persistence with annotations like @Entity, @Id, @Column.
- **Hibernate:**
  - The most popular implementation of the JPA specification.
  - It's the "engine" that reads your JPA annotations and performs the actual database work (generating SQL, managing transactions, etc.).

Core Concepts - ORM

- ORM (Object-Relational Mapping)
- Technique that maps database tables to Java classes
- Columns mapped to properties
- Database operations performed through Java objects

Basic Concept:

```
Database Table (users)      →     Java Class (User)
├── id                      →     ├── Long id
├── name                    →     ├── String name
├── email                   →     ├── String email
└── age                     →     └── Integer age
```

Core Concepts - ORM

- **Advantages**:
  - Reduces need to write SQL
  - Increases code portability
  - Type safety
  - Easy maintenance
- **Disadvantages**:
  - Performance issues with complex queries
  - Steep learning curve
  - Native SQL may be more effective in some cases

Core Concepts - JPA

- **JPA (Java Persistence API)**
- **Definition:**
  - Standard API for data persistence in Java
  - Hibernate is the most popular JPA implementation
  - Specification defining how database operations should be performed
- **JPA's Role:**
  - Only a specification, not an implementation
  - Defines which classes to use and which annotations to write

- **Key Components:**
  - **EntityManager:** Manages database operations (CRUD)
  - **Persistence Context:** Tracks lifecycle of managed objects
  - **Query Language (JPQL):** Query language similar to SQL but works on objects

JPA (Java Persistence API) Example

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

    @Column(name = "user_name", length = 100)
    private String name;
}
```

Core Concepts - Spring Data JPA

- **Spring Data JPA**
- **Definition:**
  - Abstraction layer provided by Spring that further simplifies JPA
  - Makes database operations minimalist using Repository Pattern
- **Repository Pattern:**
  - Interface that abstracts database operations
  - Developers only define the interface
  - Spring Data JPA automatically creates implementation

Spring Data JPA Example

```
public interface UserRepository extends
JpaRepository<User, Long> {
    // Spring Data JPA automatically implements
these methods:
    // save(), findById(), findAll(), delete()

    // Custom queries defined by method name
    List<User> findByName(String name);
    User findByEmail(String email);
    List<User> findByAgeGreaterThan(Integer age);
}
```

- **Advantages:**
  - Reduces boilerplate code
  - Automatic SQL query generation from method names
  - Automatic pagination & sorting
  - Custom queries with @Query annotation

Core Concepts - Hibernate

- **Hibernate**
- **Definition**:
  - Most widely used ORM framework implementing JPA
  - Implements all JPA features and provides additional features
- **Hibernate's Responsibilities:**
  - **Automatic SQL Generation:** Automatically generates SQL queries from class definitions
  - **Object-Relational Mapping:** Converts Java objects to database rows
  - **Lazy Loading**: Loads related data when needed
  - **Caching**: Keeps data in memory for performance

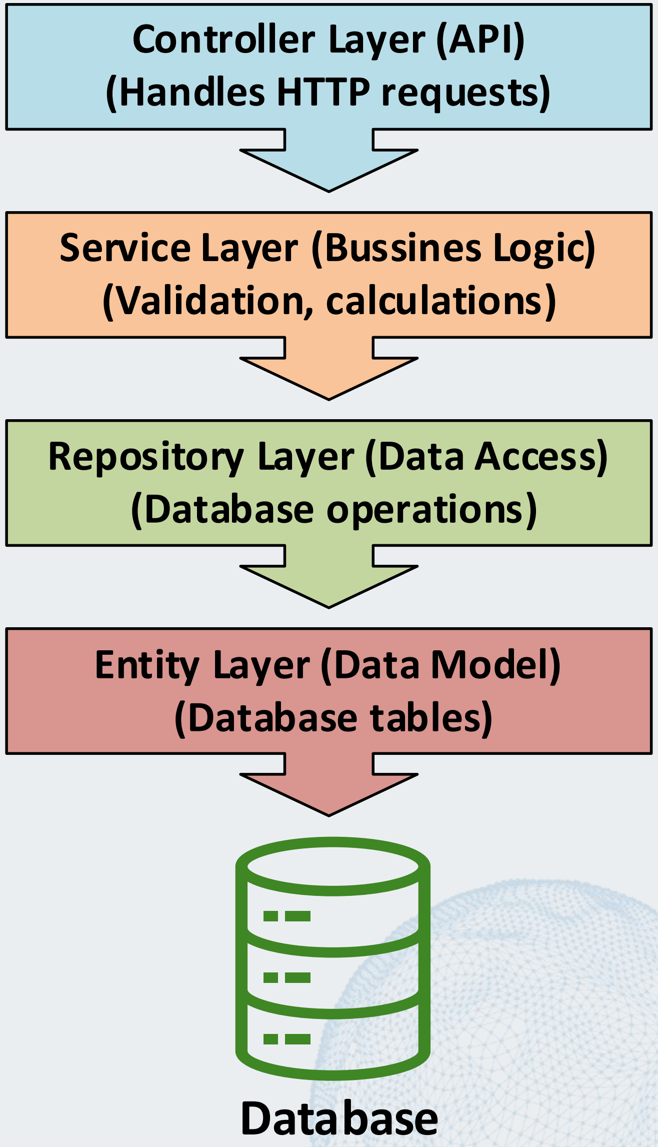
Hibernate Example

```
// Hibernate automatically converts this code to SQL
User user = session.get(User.class, 1L);
user.setName("New Name");
session.save(user);

// SQL generated by Hibernate:
// SELECT * FROM users WHERE id = 1
// UPDATE users SET user_name = 'New Name' WHERE id = 1
```

Spring Boot Layered Architecture

- Layered architecture is a design pattern that divides an application into layers with different responsibilities.
- Each layer is dependent on the layer below it and performs only its own task.
- **Benefits:**
  - Separated responsibilities
  - Testability
  - Easy maintenance
  - Reusability
  - Scalability
  - Database independence



Controller Layer (API)
(Handles HTTP requests)

Service Layer (Bussines Logic)
(Validation, calculations)

Repository Layer (Data Access)
(Database operations)

Entity Layer (Data Model)
(Database tables)

Database

Entity Layer

- Entity Layer - Data Model
- **Definition:**
  - Layer that converts database tables to Java classes
  - Each Entity corresponds to a database table
- **Responsibilities:**
  - Define database table structure
  - Map columns to Java properties
  - Define relationships

Entity Layer Example

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

    @Column(name = "username", nullable = false, unique =
true)
    private String username;

    @Column(name = "email", nullable = false, unique = true)
    private String email;

    @Column(name = "created_at")
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdAt;
}
```

- **Key Annotations:**
  - **@Entity** - Marks class as database table
  - **@Table** - Defines table name and properties
  - **@Id** - Specifies primary key
  - **@GeneratedValue** - Enables automatic ID generation
  - **@Column** - Defines column properties

Repository Layer

- **Repository Layer - Data Access**
- **Definition:**
  - Interfaces that perform database operations
  - Contains CRUD operations and custom queries
- **Responsibilities:**
  - Abstract database operations
  - Centralize data access logic
  - Facilitate switching between different databases

Repository Layer Example

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Automatic methods: save(), findById(), findAll(), delete()

    // Custom queries - Method naming convention
    User findByUsername(String username);
    List<User> findByIsActiveTrue();

    // JPQL custom query
    @Query("SELECT u FROM User u WHERE u.email = :email")
    Optional<User> findByEmail(@Param("email") String email);

    // Native SQL query
    @Query(value = "SELECT * FROM users WHERE created_at > DATE_SUB(NOW(), INTERVAL 7
DAY)",
           nativeQuery = true)
    List<User> findUsersCreatedInLastWeek();
}
```

Service Layer

- **Service Layer - Business Logic**
- **Definition:**
  - Layer containing application's business logic
  - Retrieves data from Repository, processes it, and returns to Controller
- **Responsibilities:**
  - Apply business rules
  - Perform data validation
  - Coordinate multiple repository operations
  - Error management
  - Manage transactions

Service Layer Example

```
@Service
@Transactional
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public UserDTO createUser(CreateUserRequest request) {
        // Validation
        if (userRepository.findByUsername(request.getUsername()) != null) {
            throw new IllegalArgumentException("Username already exists");
        }

        // Create entity
        User user = new User();
        user.setUsername(request.getUsername());
        user.setEmail(request.getEmail());

        // Save to database
        User savedUser = userRepository.save(user);

        // Convert to DTO
        return convertToDTO(savedUser);
    }
}
```

Controller Layer

- **Controller Layer - HTTP Requests**
- **Definition:**
  - Layer that handles HTTP requests and returns responses
  - Defines REST API endpoints
- **Responsibilities:**
  - Receive HTTP requests
  - Send requests to Service
  - Return responses in HTTP format
  - Error management

Controller Layer Example

```java
@RestController
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private UserService userService;

    @PostMapping
    public ResponseEntity<UserDTO> createUser(@RequestBody CreateUserRequest request) {
        UserDTO userDTO = userService.createUser(request);
        return ResponseEntity.status(HttpStatus.CREATED).body(userDTO);
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserDTO> getUserById(@PathVariable Long id) {
        UserDTO userDTO = userService.getUserById(id);
        return ResponseEntity.ok(userDTO);
    }

}
```

Database Configuration | Maven Pom.xml

- Database Connection Setup
- Dependencies (pom.xml):

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

<!-- MySQL Driver -->
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Database Configuration | Application.properties

- Application.properties

```
# Database connection
spring.datasource.url=jdbc:mysql://localhost:3306/myapp
spring.datasource.username=root
spring.datasource.password=password

# Hibernate configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

- **DDL-Auto Options:**
  - **update** - Creates or updates tables automatically
  - **create** - Drops and recreates tables on each startup
  - **validate** - Only validates, makes no changes
  - **create-drop** - Drops tables when application shuts down

CRUD Operations

- Operations:
  - Create
  - Read
  - Update
  - Delete
- These operations are fundamentals of all database applications.

CRUD Operations - Create

```java
// Service method
public Product createProduct(Product product) {
    // Validation
    if (product.getName() == null || product.getName().isEmpty()) {
        throw new IllegalArgumentException("Product name cannot be empty");
    }

    if (product.getPrice() == null || product.getPrice() < 0) {
        throw new IllegalArgumentException("Product price must be positive");
    }

    // Save to database
    return productRepository.save(product);
}

// Controller method
@PostMapping
public ResponseEntity<Product> createProduct(@RequestBody Product product) {
    Product createdProduct = productService.createProduct(product);
    return ResponseEntity.status(HttpStatus.CREATED).body(createdProduct);
}
```

- Create Operation - Adding New Records
- **Definition:**
  - New record added to database
  - Performed using save() method

CRUD Operations - Read

```
// Get all products
public List<Product> getAllProducts() {
    return productRepository.findAll();
}

// Get product by ID
public Optional<Product> getProductById(Long id) {
    return productRepository.findById(id);
}

// Search products by keyword
public List<Product> searchByName(String keyword) {
    return productRepository.findByNameContaining(keyword);
}

// Get products in price range
public List<Product> findByPriceRange(Double minPrice, Double maxPrice)
{
    return productRepository.findByPriceRange(minPrice, maxPrice);
}

// Get active products
public List<Product> getActiveProducts() {
    return productRepository.findByIsActiveTrue();
}
```

- **Read Operation - Retrieving Data**
- **Definition**:
  - Data read from database
  - Performed using methods like findById(), findAll()

CRUD Operations - Update

```
// Service method
public Product updateProduct(Long id, Product productUpdate) {
    // Find existing product
    Optional<Product> existingProduct = productRepository.findById(id);

    if (existingProduct.isPresent()) {
        Product product = existingProduct.get();

        // Update fields
        if (productUpdate.getName() != null) {
            product.setName(productUpdate.getName());
        }
        if (productUpdate.getPrice() != null) {
            product.setPrice(productUpdate.getPrice());
        }
        if (productUpdate.getStock() != null) {
            product.setStock(productUpdate.getStock());
        }

        // Save updated product
        return productRepository.save(product);
    }
    return null;
}
```

- **Update Operation - Modifying Records**
- **Definition:**
  - Existing record information updated
  - Performed using save() method

CRUD Operations - Delete

```java
// Hard delete - Permanently remove from database
public void deleteProduct(Long id) {
    if (productRepository.existsById(id)) {
        productRepository.deleteById(id);
    } else {
        throw new IllegalArgumentException("Product not found");
    }
}

// Soft delete - Mark as inactive
public void softDeleteProduct(Long id) {
    Optional<Product> product = productRepository.findById(id);

    if (product.isPresent()) {
        Product p = product.get();
        p.setIsActive(false);
        productRepository.save(p);
    }
}

// Controller method
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
    productService.deleteProduct(id);
    return ResponseEntity.noContent().build();
}
```

- **Delete Operation - Removing Records**
- **Definition:**
  - Record deleted from database
  - Performed using deleteById() method

Relationships - One-to-Many

```java
// Category Entity
@Entity
@Table(name = "categories")
public class Category {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name", nullable = false,
unique = true)
    private String name;

    @OneToMany(mappedBy = "category",
cascade = CascadeType.ALL)
    private List<Product> products = new
ArrayList<>();
}
```

```java
// Product Entity
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Long id;

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

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "category_id",
nullable = false)
    private Category category;
}
```

Relationships - Many-to-Many

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

    @Column(name = "name", nullable = false,
unique = true)
    private String name;

    @ManyToMany(mappedBy = "tags")
    private List<Product> products = new
ArrayList<>();
}
```

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

    @ManyToMany
    @JoinTable(
        name = "product_tags",
        joinColumns = @JoinColumn(name =
"product_id"),
        inverseJoinColumns =
@JoinColumn(name = "tag_id")
    )
    private List<Tag> tags = new ArrayList<>();
}
```

Custom Queries | @QUERY

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

    // JPQL - Uses entity names
    @Query("SELECT p FROM Product p WHERE p.price BETWEEN :minPrice AND :maxPrice")
    List<Product> findByPriceRange(
        @Param("minPrice") Double minPrice,
        @Param("maxPrice") Double maxPrice
    );

    // JPQL with JOIN
    @Query("SELECT p FROM Product p JOIN p.tags t WHERE t.id = :tagId")
    List<Product> findByTagId(@Param("tagId") Long tagId);

    // Count query
    @Query("SELECT COUNT(p) FROM Product p WHERE p.category.id = :categoryId")
    Long countByCategory(@Param("categoryId") Long categoryId);
}
```

Method Naming Conventions

```
public interface UserRepository extends JpaRepository<User, Long> {
    // Find by single field
    User findByUsername(String username);
    User findByEmail(String email);

    // Find with conditions
    List<User> findByAgeGreaterThan(Integer age);
    List<User> findByAgeLessThan(Integer age);
    List<User> findByAgeBetween(Integer minAge, Integer maxAge);

    // Find with LIKE
    List<User> findByNameContaining(String keyword);
    List<User> findByNameStartingWith(String prefix);
    List<User> findByNameEndingWith(String suffix);

    // Find with boolean
    List<User> findByIsActiveTrue();
    List<User> findByIsActiveFalse();

    // Find with multiple conditions
    List<User> findByNameAndEmail(String name, String email);
    List<User> findByNameOrEmail(String name, String email);

    // Sorting
    List<User> findByAgeOrderByNameAsc(Integer age);
    List<User> findByAgeOrderByNameDesc(Integer age);
}
```

- **Spring Data JPA Method Naming**
- **Automatic Query Generation**:
  - Spring Data JPA automatically generates queries from method names

ASST. PROF. DR. SAMET KAYA

FATIH SULTAN MEHMET VAKIF UNIVERSITY

skaya@fsm.edu.tr

kysamet@gmail.com

Some images and icons in this presentation are sourced from Freepik.