**UNIT-3**

**Advanced Data Access with Spring Data JPA: Spring Data JPA- Introduction to Spring Data JPA, Configuring JPA and connecting to databases, Creating repositories and CRUD operations. Advanced JPA- JPQL and native queries, Pagination and sorting, Relationships (One-to-One Relationships, One-to-Many Relationships, Many-to-Many Relationships).**

**Securing Spring: Enabling Spring Security, Configuring Spring Security (In-memory user store, JDBC-based user store, LDAP-backed user store, Customizing user, authentication), Securing web requests (Securing requests, Creating a custom login page, Logging out, Preventing cross-site request forgery), Knowing your user. Transaction Management- Understanding and managing transactions.**

**Textbook 1: Chapter 3:3.1 and 3.2**

**Textbook 1: Chapter 4:4.1 to 4.4**

**Advanced Data Access with Spring Data JPA**

**1. Introduction to Spring Data JPA**

**Spring Data JPA is a part of the Spring Framework that simplifies database interactions by reducing the need for boilerplate code. It provides a higher-level abstraction for working with JPA (Java Persistence API), allowing developers to focus on business logic rather than database interactions.**

**Key Features:**

* **Reduces boilerplate code**
* **Provides default implementations for CRUD operations**
* **Supports pagination and sorting**
* **Facilitates relationship mapping and querying**
* **Supports both JPQL and native SQL queries**

**What is Boilerplate Code?**

**Boilerplate code refers to repetitive, standard, or template-like code that is required in many places without significant modifications. It is often necessary for setting up basic functionality but does not contribute to business logic directly.**

**CRUD Operations?**

**CRUD stands for Create, Read, Update, and Delete—the four basic operations for interacting with a database.**

1. **Create – Adding new records to the database.**
2. **Read – Fetching or retrieving data.**
3. **Update – Modifying existing data.**
4. **Delete – Removing data from the database.**

**JPQL (Java Persistence Query Language)**

**JPQL is a query language similar to SQL but operates on JPA entity objects instead of database tables. It provides a more object-oriented approach to querying, ensuring database independence.**

**Key Features of JPQL**

* **Works with entity classes instead of table names.**
* **Uses entity field names instead of column names.**
* **Supports JOINs, filtering, sorting, and aggregation.**
* **Allows dynamic and named queries.**

**2. Configuring JPA and Connecting to Databases**

**To use Spring Data JPA, you need to configure JPA and connect it to a database.**

**Steps to Configure:**

1. **Add Dependencies:**

**<dependency>**

 **<groupId>org.springframework.boot</groupId>**

 **<artifactId>spring-boot-starter-data-jpa</artifactId>**

**</dependency>**

**<dependency>**

 **<groupId>com.h2database</groupId>**

 **<artifactId>h2</artifactId>**

 **<scope>runtime</scope>**

**</dependency>**

**What are Dependencies?**

**In software development, dependencies are external libraries or modules that a project requires to function properly. They provide additional functionality without requiring developers to write everything from scratch.**

**In a Spring Boot project, dependencies are managed using Maven (via pom.xml) or Gradle. These dependencies include essential components such as Spring Data JPA, databases (H2, MySQL, PostgreSQL, etc.), and other integrations.**

**Example of Dependencies in pom.xml:**

**2.Configure application.properties:**

**spring.datasource.url=jdbc:h2:mem:testdb**

**spring.datasource.driver-class-name=org.h2.Driver**

**spring.datasource.username=sa**

**spring.datasource.password=**

**spring.jpa.database-platform=org.hibernate.dialect.H2Dialect**

**spring.h2.console.enabled=true**

**3. Define Entity Class:**

**@Entity**

**public class Student {**

 **@Id**

 **@GeneratedValue(strategy = GenerationType.IDENTITY)**

 **private Long id;**

 **private String name;**

 **private String email;**

 **// Getters and Setters**

**}**

**4.Enable JPA Repositories:**

**@SpringBootApplication**

**@EnableJpaRepositories**

**public class Application {**

 **public static void main(String[] args) {**

 **SpringApplication.run(Application.class, args);**

 **}**

**}**

**3. Creating Repositories and CRUD Operations**

**Spring Data JPA provides repository interfaces to handle common database operations.**

**Repository Interface Example:**

**public interface StudentRepository extends JpaRepository<Student, Long> {**

**}**

**CRUD Operations:**

**@Service**

**public class StudentService {**

 **@Autowired**

 **private StudentRepository repository;**

 **public Student saveStudent(Student student) {**

 **return repository.save(student);**

 **}**

 **public List<Student> getAllStudents() {**

 **return repository.findAll();**

 **}**

 **public Optional<Student> getStudentById(Long id) {**

 **return repository.findById(id);**

 **}**

 **public void deleteStudent(Long id) {**

 **repository.deleteById(id);**

 **}**

**}**

**4. Advanced JPA**

**4.1 JPQL and Native Queries**

* **JPQL (Java Persistence Query Language): Similar to SQL but operates on entity objects instead of tables.**
* **Native Queries: Directly execute SQL queries on the database.**

**JPQL Example:**

**@Query("SELECT s FROM Student s WHERE s.email = ?1")**

**Student findByEmail(String email);**

**Native Query Example:**

**@Query(value = "SELECT \* FROM student WHERE email = ?1", nativeQuery = true)**

**Student findByEmailNative(String email);**

**2 Pagination and Sorting**

**Spring Data JPA provides built-in pagination and sorting using Pageable and Sort objects.**

**Example:**

**Page<Student> findAll(Pageable pageable);**

**List<Student> findAll(Sort sort);**

**Usage:**

**PageRequest pageable = PageRequest.of(0, 5, Sort.by("name").ascending());**

**Page<Student> students = studentRepository.findAll(pageable);**

**4.3 Relationships in JPA**

**JPA supports different types of relationships:**

**One-to-One Relationship: Each employee has one company ID card**

**@Entity**

**public class Passport {**

 **@Id**

 **@GeneratedValue(strategy = GenerationType.IDENTITY)**

 **private Long id;**

 **private String number;**

 **@OneToOne(mappedBy = "passport")**

 **private Student student;**

**}**

**One-to-Many Relationship: E-commerce Platform: A product can have multiple reviews.**

**@Entity**

**public class Course {**

 **@Id**

 **@GeneratedValue(strategy = GenerationType.IDENTITY)**

 **private Long id;**

 **private String title;**

 **@OneToMany(mappedBy = "course")**

 **private List<Student> students;**

**}**

**Many-to-Many Relationship:**

**@Entity**

**public class Student {**

 **@ManyToMany**

 **@JoinTable(name = "student\_course",**

 **joinColumns = @JoinColumn(name = "student\_id"),**

 **inverseJoinColumns = @JoinColumn(name = "course\_id"))**

 **private List<Course> courses;**

**}**

**University System: A student can enroll in multiple courses, and a course can have multiple students.**

**E-learning platforms (Coursera, Udemy, etc.): A student can enroll in many courses, and a course can have many students.**

**1. Securing Spring**

**Spring Security is a robust framework that provides authentication, authorization, and protection against common attacks. It is highly customizable and integrates deeply with the Spring ecosystem.**

**1.1 Enabling Spring Security**

**Theory:**

**By default, Spring Boot includes auto-configuration for security when the spring-boot-starter-security dependency is present. This enables basic security features such as form-based login, CSRF protection, and password encoding.**

**Key Concepts:**

* **Secure endpoints automatically**
* **Prevent unauthorized access**
* **Provide default login form**
* **Enforce user roles and permissions**

**How to Enable:**

**Simply include the dependency:**

**<dependency>**

 **<groupId>org.springframework.boot</groupId>**

 **<artifactId>spring-boot-starter-security</artifactId>**

**</dependency>**

**1.2 Configuring Spring Security**

**a) In-Memory User Store**

**Theory:**

**In-memory authentication is the simplest form of user store, ideal for development or testing. Users and their roles are defined directly in the application code.**

**Use Case: Small-scale apps, proof-of-concepts, or learning environments.**

**@Bean**

**public UserDetailsService userDetailsService() {**

 **UserDetails user = User.withDefaultPasswordEncoder()**

 **.username("user")**

 **.password("pass")**

 **.roles("USER")**

 **.build();**

 **return new InMemoryUserDetailsManager(user);**

**}**

**b) JDBC-based User Store**

**Theory:**

**For production systems, user information is often stored in relational databases. Spring Security can be configured to fetch user details from the database using JDBC.**

**Advantages:**

* **Centralized user management**
* **Persistent user store**
* **Integration with existing DB schema**

**Spring expects two tables: users and authorities. Custom queries can be used if your schema is different.**

**@Bean**

**public UserDetailsService jdbcUserDetailsService(DataSource dataSource) {**

 **return new JdbcUserDetailsManager(dataSource);**

**}**

**c) LDAP-backed User Store**

**Theory:**

**LDAP (Lightweight Directory Access Protocol) is commonly used in enterprise environments for centralized user authentication and management.**

**Benefits:**

* **Centralized user control**
* **Scalability for large orgs**
* **Integration with corporate Active Directory**

**Spring Boot supports LDAP configuration through simple properties, enabling seamless connection to LDAP servers.**

**d) Customizing User Authentication**

**Theory:**

**For more flexibility, Spring allows custom authentication logic. You can define your own user store, credentials check, or authorization rules.**

**Use a custom UserDetailsService:**

**@Service**

**public class CustomUserDetailsService implements UserDetailsService {**

 **@Override**

 **public UserDetails loadUserByUsername(String username) {**

 **// Load user from DB and return a UserDetails object**

 **}**

**}**

**1.3 Securing Web Requests**

**a) Securing Requests**

**Theory:**

**Web application security involves controlling access to different parts of the application based on user roles. Spring Security allows URL-level access control.**

**Common Use Cases:**

* **Protect admin routes**
* **Limit access based on role**
* **Secure REST APIs**

**Example:**

**http.authorizeRequests()**

 **.antMatchers("/admin/\*\*").hasRole("ADMIN")**

 **.antMatchers("/user/\*\*").authenticated()**

 **.anyRequest().permitAll();**

**b) Creating a Custom Login Page**

**Theory:**

**Spring Security provides a default login page, but for better user experience and branding, you can design your own.**

**You can use:**

* **HTML forms**
* **Thymeleaf integration**
* **Error messages for failed login**

**http.formLogin()**

 **.loginPage("/custom-login")**

 **.permitAll();**

**c) Logging Out**

**Theory:**

**Logging out invalidates the current session and clears security context.**

**http.logout()**

 **.logoutUrl("/logout")**

 **.logoutSuccessUrl("/login?logout");**

**Spring also provides:**

* **CSRF token clearing**
* **Session invalidation**
* **Cookie deletion**

**d) Preventing Cross-Site Request Forgery (CSRF)**

**Theory:**

**CSRF is a type of attack where unauthorized commands are transmitted from a user that the application trusts. Spring Security protects against this by requiring a special token in every modifying request (POST, PUT, DELETE).**

**Best Practice: Never disable CSRF in production applications unless you're building a stateless REST API.**

**In forms:**

**<input type="hidden" name="\_csrf" value="${\_csrf.token}"/>**

**1.4 Knowing Your User**

**Theory:**

**Understanding who the user is allows you to personalize the application and apply correct authorization rules.**

**Use Spring Security context:**

**Authentication auth = SecurityContextHolder.getContext().getAuthentication();**

**String username = auth.getName();**

**You can also inject the Principal object in controllers:**

**@GetMapping("/profile")**

**public String profile(Principal principal) {**

 **return "Welcome " + principal.getName();**

**}**

**2. Transaction Management in Spring**

**2.1 What is a Transaction?**

**Theory:**

**A transaction is a set of operations that must all succeed or fail as a single unit. It ensures the ACID properties:**

* **Atomicity – All or nothing**
* **Consistency – Transitions from one valid state to another**
* **Isolation – Operations are isolated from others**
* **Durability – Changes are permanent after commit**

**2.2 Managing Transactions in Spring**

**Theory:**

**Spring offers both declarative and programmatic transaction management. The declarative model using annotations is preferred for simplicity and clarity.**

**2.3 Declarative Transaction Management**

**Use @Transactional to mark methods where all operations should be executed in a transaction.**

**@Transactional**

**public void updateAccountBalance() {**

 **// debit and credit operations**

**}**

**Rollback Rules:**

* **By default, rolls back on RuntimeException**
* **Use rollbackFor for other exceptions**

**@Transactional(rollbackFor = IOException.class)**

**2.4 Programmatic Transaction Management**

**Theory:**

**Sometimes, you need fine-grained control, such as deciding to commit or roll back manually.**

**TransactionStatus status = transactionManager.getTransaction(def);**

**try {**

 **// do operations**

 **transactionManager.commit(status);**

**} catch (Exception e) {**

 **transactionManager.rollback(status);**

**}**

**2.5 Transaction Propagation and Isolation**

**Propagation determines how transactions behave when called within other transactions:**

****

****

**@Transactional(propagation = Propagation.REQUIRES\_NEW,**

 **isolation = Isolation.SERIALIZABLE)**

**Best Practices in Transaction Management**

* **Apply @Transactional at the service layer, not controllers.**
* **Keep transaction scopes small and efficient.**
* **Avoid nested transactions unless necessary.**
* **Use proper isolation levels for consistency and concurrency.**