WEEK 8 COURSE CONTENT : Configuring Data Sources and Connection Pooling

TOPICS:

- Introduce the concept of data sources and connection pooling in the context of database access.
- Explain how to configure data sources and connection pooling in a Spring Boot application.
- Discuss common connection pooling libraries, such as HikariCP or Apache DBCP, and their configuration options.
- Introduce Spring Data JPA and its capabilities for simplifying database access and reducing boilerplate code.
- Demonstrate how to define and implement entity classes and repositories for performing CRUD operations.
- Guide students in implementing basic CRUD operations, such as creating, reading, updating, and deleting data.

LEARNING OUTCOME:

- Understand the concept of data sources and connection pool
- Utilize connection pooling libraries such as HikariCP, Apache DBCP and the configurations.
- More concept on CRUD Operations
- Caching of information using Cacheable
- All about Repositories
- JPQL – all about Java Persistent Query Language
- Criteria API
- Data JPA and JDBC
- Transactional

DAY 1:

Introduction to Data Sources and Connection Pooling

What are Data Sources?

- Data sources are objects that manage database connections.
- They provide a way to connect to databases efficiently.

What is Connection Pooling?

- Connection pooling is a technique for reusing database connections.
- It reduces the overhead of creating and closing connections for each database operation.

Configuring Data Sources and Connection Pooling in Spring Boot

Spring Boot's DataSource

- In Spring Boot, we use the **DataSource** interface for database connections.
- It abstracts the underlying database connection details.

Common Connection Pooling Libraries

HikariCP

- HikariCP is a high-performance connection pooling library.
- It's the default choice for Spring Boot.

Apache DBCP

- Apache DBCP is another option for connection pooling.
- It's more mature but may not be as performant as HikariCP.

Configuration Options

HikariCP Configuration (application.properties)

```
# Database Connection Settings

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=myuser

spring.datasource.password=mypassword


# HikariCP Settings

spring.datasource.hikari.maximum-pool-size=10

spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=30000
```

Apache DBCP Configuration (application.properties)

```
# Database Connection Settings
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=myuser
spring.datasource.password=mypassword

# DBCP Settings
spring.datasource.dbcp2.initial-size=5
spring.datasource.dbcp2.max-total=10
spring.datasource.dbcp2.max-idle=5
```

Introduction to Spring Data JPA

What is Spring Data JPA?

- Spring Data JPA simplifies database access using JPA (Java Persistence API).
- It reduces boilerplate code when working with databases.

Entity Classes

What are Entity Classes?

- Entity classes represent database tables.
- Annotate them with **@Entity** to map them to database tables.

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double price;
    // Constructors, getters, setters...
}
```

Repositories

What are Repositories?

- Repositories are interfaces that extend **JpaRepository**.
- They provide high-level methods for database operations.

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findByName(String name);
}
```

CRUD Operations

Creating (Inserting) Data

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public Product createProduct(Product product) {
        return productRepository.save(product);
    }
}
```

```
}
```

Reading (Retrieving) Data

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    public Product getProductById(Long id) {
        return productRepository.findById(id).orElse(null);
    }
}
```

Updating Data

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public Product updateProduct(Product updatedProduct) {
        return productRepository.save(updatedProduct);
    }
}
```

Deleting Data

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }
}
```

By understanding data sources, connection pooling, Spring Data JPA, and the provided code samples, you'll be well-equipped to work with databases efficiently in your Spring Boot applications. Practice these concepts to become proficient in database operations.

DAY 2:

Querying with Spring Data JPA (JPQL, Criteria API)

Different Approaches for Querying with Spring Data JPA

Approach 1: Repository Query Methods

- Spring Data JPA provides repository query methods.
- Define custom query methods in your repository interface.
- Query methods are automatically generated based on method names.

Example:

```
public interface ProductRepository extends JpaRepository<Product, Long> {
   List<Product> findByCategory(String category);
}
```

JPQL (Java Persistence Query Language)

- JPQL is a SQL-like query language for querying JPA entities.
- Use **@Query** annotation with JPQL queries in repository methods.

Example:

```
@Query("SELECT p FROM Product p WHERE p.price > :price")
List<Product> findExpensiveProducts(@Param("price") double price);
```

Criteria API

- Criteria API is a type-safe, programmatic way to build queries.
- Use the **CriteriaBuilder** and **CriteriaQuery** classes to create queries.

Example:

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Product> query = builder.createQuery(Product.class);
Root<Product> root = query.from(Product.class);
query.select(root).where(builder.gt(root.get("price"), price));
List<Product> expensiveProducts =
entityManager.createQuery(query).getResultList();
```

Database Transactions and Isolation Levels

Importance of Database Transactions

Ensuring Data Consistency and Integrity

- Database transactions group multiple database operations into a single unit of work.
- Ensures that the database remains in a consistent state.
- Supports the principles of ACID (Atomicity, Consistency, Isolation, Durability).

Managing Database Transactions in Spring Boot

Spring's Transaction Management

- Spring provides declarative transaction management.
- Use **@Transactional** annotation to mark methods or classes as transactional.

Example:

```
@Service
@Transactional
public class ProductService {
   @Autowired
   private ProductRepository productRepository;

   public Product createProduct(Product product) {
      return productRepository.save(product);
   }

   // Other transactional methods
}
```

Isolation Levels and Their Impact

Isolation Levels

- Different isolation levels define the visibility of changes made by one transaction to other concurrent transactions.
- Common isolation levels: **READ_UNCOMMITTED**, **READ_COMMITTED**, **REPEATABLE_READ**, **SERIALIZABLE**.
- Lower isolation levels provide better performance but may lead to data anomalies.

Example with **@Transactional** and Isolation Level

```
@Service

@Transactional(isolation = Isolation.READ_COMMITTED)

public class ProductService {

   // Transactional methods

}
```

Understanding these querying approaches, transaction management, and isolation levels will help you build efficient and reliable Spring Boot applications that interact with databases. Practice these concepts to become proficient in database operations and transaction management.

Day 3:

Caching Data with Spring Boot

Introduction to Caching

What is Caching?

- Caching is a mechanism to store frequently accessed data in memory.
- It improves application performance by reducing the need to fetch data from the database repeatedly.
- Caching also reduces the load on the database server.

Benefits of Caching

- Faster response times

- Reduced database load
- Improved scalability

Configuring and Utilizing Caching in Spring Boot

Enabling Caching in Spring Boot

- Add **@EnableCaching** annotation to your main application class.
- This enables Spring's caching functionality.

```
@SpringBootApplication

@EnableCaching

public class MyApplication {

   public static void main(String[] args) {

      SpringApplication.run(MyApplication.class, args);

   }

}
```

Caching Annotations

@Cacheable

- Annotate methods with **@Cacheable** to cache the method's result.
- Specify the cache name to use.

```
@Cacheable("productsCache")

public List<Product> getAllProducts() {

   // Fetch products from the database

   return productService.fetchAllProducts();

}
```

@CacheEvict

- Use **@CacheEvict** to remove items from the cache.
- Specify the cache name and the condition for eviction.

```
@CacheEvict(value = "productsCache", allEntries = true)
public void refreshCache() {
    // Refresh the cache
}
```

Spring Data JDBC

Introduction to Spring Data JDBC

What is Spring Data JDBC?

- Spring Data JDBC is a module of the Spring Data project.
- It provides a simple and efficient way to interact with databases using plain old Java objects (POJOs).
- Spring Data JDBC focuses on data access without the need for complex mapping and ORM (Object-Relational Mapping).

Comparing Spring Data JPA and Spring Data JDBC

Spring Data JPA

- Uses JPA (Java Persistence API).
- Requires entity classes with annotations.
- Provides high-level abstractions and complex query capabilities.
- Suitable for complex domain models and relationships.

Spring Data JDBC

- Works with plain POJOs, no need for annotations.
- Offers basic CRUD operations with simple queries.
- Suitable for simpler data models with less complexity.
- Emphasizes performance and ease of use.

Configuring Spring Data JDBC

Dependency in pom.xml

- Add the Spring Data JDBC dependency to your **pom.xml** file.

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

```
</dependency>
```

Repository Interface

- Create a repository interface that extends **CrudRepository** or **JdbcRepository**.

```
public interface ProductRepository extends CrudRepository<Product, Long> {

}
```

Entity Class

- Define a POJO class for your entity.

```
public class Product {

   private Long id;

   private String name;

   private double price;

   // Getters and setters

}
```

Performing Basic Database Operations

Creating Data

- Use the **save** method to insert data.

```
Product product = new Product("Laptop", 999.99);

productRepository.save(product);
```

Reading Data

- Use the **findById** method to retrieve data.

```
Product product = productRepository.findById(1L).orElse(null);
```

Updating Data

- Modify the object and save it again.

```
product.setPrice(899.99);
productRepository.save(product);
```

Deleting Data

- Use the **delete** method to remove data.

```
productRepository.delete(product);
```

By understanding caching mechanisms and Spring Data JDBC, you'll be able to optimize your Spring Boot applications for better performance and efficient database access. Practice these concepts to become proficient in caching and data access.

Day 4: 7. Review and Assessment:

- Conduct a comprehensive review of the topics covered throughout the week.
- Review key concepts, code examples, and best practices related to database access with Spring Boot.
- Engage students in a discussion to reinforce their understanding and clarify any doubts or questions.
- Administer an assessment, such as a coding assignment or quiz, to evaluate students' knowledge and skills in working with database access in Spring Boot.

Throughout the week:

- Provide hands-on coding exercises and examples to reinforce understanding.
- Encourage students to work with a sample database schema and perform various operations using Spring Boot.

- Discuss real-world scenarios and challenges related to database access and how Spring Boot can address them.
- Conduct interactive discussions and Q&A sessions to address students' questions and concerns.