Java Backend Development Live-85

lecture-12

Agenda

- Spring JDBC
  - JdbcTemplate
  - Datasource
  - RowMapper
  - NamedParameterJdbcTemplate
- Connection Pool
- JPA (Hibernate)
  - JPA Annotations
  - Entity Mapping
  - JPA Transactions
  - Request Flow

Spring JDBC

It provides you methods to write the queries directly, so it saves a lot of work and time.

JdbcTemplate class: It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

Example:

```
JdbcTemplate jdbcTemplateObject = new JdbcTemplate(dataSource);

String selectQuery = "select * from person";

List <Person> persons = jdbcTemplateObject.query(selectQuery, new PersonMapper());
```

Datasource

Datasource bean is required by spring jdbctemplate to communicate with the database.

```
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName( "com.mysql.jdbc.Driver" );
        dataSource.setUrl( "jdbc:mysql://localhost:3306/dbname" );
        dataSource.setUsername( "username");
        dataSource.setPassword( "password");
        return dataSource;
    }
```

RowMapper

Implementations of this interface perform the actual work of mapping each row to a result object.

```
public class PersonMapper implements RowMapper<Person> {

   public Person mapRow(ResultSet rs, int rowNum) throws SQLException {

       Person person = new Person();

       person.setId(rs.getInt("id"));

       person.setName(rs.getString("name"));

       return person;

   }

}
```

NamedParameterJdbcTemplate

It allows the use of named parameters rather than traditional '?' placeholders.

```
MapSqlParameterSource in = new MapSqlParameterSource();

in.addValue("id", id);

in.addValue("name",  new String(“Rahul”));

String SQL = "update person set name = :name where id = :id";

NamedParameterJdbcTemplate jdbcTemplateObject = new NamedParameterJdbcTemplate(dataSource);

jdbcTemplateObject.update(SQL, in);
```

DAO Layer with Spring JDBC

Separate Dao/Repository class for each entity.

Use NamedParameterJdbcTemplate.

Create RowMapper for each entity.

Define queries in property file or DAO class.

```
person.query.find.by.id=SELECT * FROM person WHERE ID = :id
```

```
@Value("${person.query.find.by.id}")
private String findById;
```

Connection Pool

- A connection pool is a cache of database connections maintained to reuse for future requests.
- Instead of creating and closing a database connection for every query, the application borrows connections from the pool.
- This reduces the overhead of establishing connections repeatedly, Limits the number of connections to the database, And allows multiple threads to share a small number of database connections efficiently.
- If you want connection pooling in a Spring JDBC application, you need to use a connection pooling library like **HikariCP**, Apache DBCP, or C3P0, and configure it as your **DataSource**.

Spring Data JPA

Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to **ORM** tools.

Spring Data takes this simplification one step further and makes it possible to remove the DAO implementations entirely.

Spring Data provides multiple repository interfaces that are used for different purposes.

Hibernate

It is an open source, lightweight, ORM (Object Relational Mapping) tool.

Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

Hibernate uses Hibernate Query Language which makes it database independent.

It supports auto DDL operations.

Hibernate has Auto Primary Key Generation support.

It supports Cache memory.

Important Properties

```
# To Create Datasource
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
# DDL mode.
spring.jpa.hibernate.ddl-auto=update
# Whether to enable logging of SQL statements.
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.generate_statistics=true
```

JPA Annotations

@Entity: This annotation let Spring know that the Employee class is actually representing a table in the database.

@Table(name = "persons"): With this we can declare the name of the table corresponding to this data model.

@Id: This annotation defines the id field as the primary key.

@GeneratedValue(strategy = GenerationType.AUTO/GenerationType.IDENTITY): This annotation is used to delegate the database to pick an appropriate strategy for the primary key generation.

@Column(name = "email", nullable = false, unique = true): With this annotation we can associate a field with specific constraints.

JPA Mapping annotation

@OneToOne: Branch <--> Location

```
@OneToOne(  cascade = CascadeType.ALL)
  @JoinColumn(name = "locationId")
  private Location location;
```

@JoinColumn & **@OneToOne** should be mappedBy attribute when foreign key is held by one of the entities.

@ManyToOne: Employee <--> Branch

```
@ManyToOne
  @JoinColumn(name = "branchId")
  private Branch branch;
```

Cascading: When we perform some action on the target entity, the same action will be applied to the associated entity.

@OneToMany: Branch <--> Employee

```
@OneToMany(mappedBy = "branch", fetch = FetchType.EAGER)
 private Set<Employee> employees;
```

FetchType: EAGER, LAZY

LAZY = fetch when needed in a Transaction

EAGER = fetch immediately

JPA Transactions

Spring 3.1 introduces the **@EnableTransactionManagement** annotation that we can use in a **@Configuration** class to enable transactional support

@Transactional either at the class or method level

By default, rollback happens for runtime, unchecked exceptions only. The checked exception does not trigger a rollback of the transaction. We can configure this behavior with the **rollbackFor** and annotation parameters.

At a high level, Spring creates **proxies** for all the classes annotated with @Transactional, either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.

In the programmatic approach, we rollback the transactions using **TransactionAspectSupport**,

Request Flow