Introduction to JDBC and Spring JDBC/DAO

1. Plain JDBC Limitations

Plain JDBC refers to using the JDBC API directly for database interactions. While JDBC provides the fundamental capabilities to interact with a database, it comes with several limitations:

- **Boilerplate Code**: Requires a lot of repetitive and error-prone boilerplate code for opening and closing connections, managing transactions, and handling SQL exceptions.
- **Resource Management**: Manual management of database resources such as Connection, Statement, and ResultSet, which increases the risk of resource leaks.
- **Error Handling**: SQLExceptions are handled in a generic manner and can be difficult to interpret and handle effectively.
- **SQL Injection Risks**: Writing raw SQL queries can lead to SQL injection vulnerabilities if parameters are not handled properly.
- **Lack of Abstraction**: Offers no abstraction layer over JDBC, making it hard to switch between different data sources or databases.

2. Spring JDBC/DAO Advantages

Spring JDBC/DAO provides a higher level of abstraction over plain JDBC, making database access more efficient and less error-prone:

- **Simplified API**: Reduces boilerplate code for common operations like opening and closing connections and handling exceptions.
- **Exception Translation**: Translates JDBC exceptions into Spring's DataAccessException, which provides more meaningful exception handling and reduces the need to catch and handle JDBC-specific exceptions.
- **Resource Management**: Manages database connections and resources automatically, reducing the risk of resource leaks.
- **Named Parameters**: Supports named parameters in SQL queries, making queries more readable and maintainable.

- **Integration with Spring**: Seamlessly integrates with other Spring features like transaction management and dependency injection.

3. Working with Different Data Sources

Spring JDBC/DAO supports various data sources, including:

- **Embedded Databases**: Like H2 or Derby, often used for development and testing.
- **Relational Databases**: Like MySQL, PostgreSQL, Oracle, etc., commonly used in production environments.
- **Cloud Databases**: Like Amazon RDS or Google Cloud SQL.

To work with different data sources, you need to configure the appropriate DataSource bean in your Spring configuration.

Example Configuration for MySQL DataSource:

```
package com.mphasis.springjdbc;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.datasource.DriverManagerDataSource;


@Configuration

@ComponentScan(basePackages = "com.mphasis.springjdbc")

public class AppConfig {


    @Bean
```

```
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mphasis"); // replace with your DB URL
        dataSource.setUsername("root"); // replace with your DB username
        dataSource.setPassword("root"); // replace with your DB password
        return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

4. JdbcTemplate

JdbcTemplate is a central class in Spring's JDBC support. It simplifies the process of working with JDBC by handling resource management and error handling for you.

Features:

- **Simplifies JDBC Operations**: Handles the creation and release of JDBC resources, such as Connection, Statement, and ResultSet.
- **Exception Translation**: Converts SQLExceptions into Spring's DataAccessException hierarchy.

- **Query Methods**: Provides methods for querying the database and mapping the results to Java objects.

Example Usage:

Configuration:

```
package com.mphasis.springjdbc;

import javax.sql.DataSource;


import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.datasource.DriverManagerDataSource;


@Configuration

@ComponentScan(basePackages = "com.mphasis.springjdbc")

public class AppConfig {


    @Bean

    public DataSource dataSource() {

        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

        dataSource.setUrl("jdbc:mysql://localhost:3306/mphasis"); // replace with your DB URL

        dataSource.setUsername("root"); // replace with your DB username
```

```
        dataSource.setPassword("root"); // replace with your DB password

        return dataSource;

    }


    @Bean

    public JdbcTemplate jdbcTemplate(DataSource dataSource) {

        return new JdbcTemplate(dataSource);

    }

}
```

DAO Implementation:

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class UserDao {

    private final JdbcTemplate jdbcTemplate;

    public UserDao(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void createUser(String name) {
        String sql = "INSERT INTO students (name) VALUES (?)";
        jdbcTemplate.update(sql, name);
    }

    public List<String> getUsers() {
        String sql = "SELECT name FROM students";
        return jdbcTemplate.queryForList(sql, String.class);
    }
}
```

5. NamedParameterJdbcTemplate

NamedParameterJdbcTemplate is an extension of JdbcTemplate that allows you to use named parameters in SQL queries instead of traditional positional parameters.

Advantages:

- **Readability**: Named parameters make SQL queries more readable and easier to maintain.
- **Flexibility**: Helps avoid issues related to parameter ordering in SQL queries.

Example Usage:

Configuration:

```
package com.mphasis.springnamedjdbc;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import
org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

import org.springframework.jdbc.datasource.DriverManagerDataSource;


import javax.sql.DataSource;


@Configuration
```

```
@ComponentScan(basePackages = "com.mphasis.springnamedjdbc") // Update with your package name

public class AppConfig {


    @Bean

    public DataSource dataSource() {

        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

        dataSource.setUrl("jdbc:mysql://localhost:3306/mphasis"); // Replace with your DB URL

        dataSource.setUsername("root"); // Replace with your DB username

        dataSource.setPassword("root"); // Replace with your DB password

        return dataSource;

    }


    @Bean

    public NamedParameterJdbcTemplate namedParameterJdbcTemplate(DataSource dataSource) {

        return new NamedParameterJdbcTemplate(dataSource);

    }

}
```

DAO Implementation:

```
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
```

```
import org.springframework.stereotype.Repository;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Repository
public class UserDao {

    private final NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public UserDao(NamedParameterJdbcTemplate
namedParameterJdbcTemplate) {
        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
    }

    public void createUser(String name) {
        String sql = "INSERT INTO students (name) VALUES (:name)";
        Map<String, Object> params = new HashMap<>();
        params.put("name", name);
        namedParameterJdbcTemplate.update(sql, params);
    }

    public List<String> getUsers() {
        String sql = "SELECT name FROM students";
        return namedParameterJdbcTemplate.queryForList(sql, new HashMap<>(),
String.class);
    }
}
```

6. Spring JDBC/DAO with Annotations

Spring JDBC/DAO can be used with annotations to simplify configuration and eliminate boilerplate code.

Annotations:

- **@Repository**: Marks a class as a Spring-managed component that interacts with the database.
- **@Autowired**: Automatically injects the required dependencies.
- **@Transactional**: Manages transactions at the method or class level.

Summary

- **Plain JDBC Limitations**: Direct JDBC requires a lot of boilerplate code and manual resource management, and it can be error-prone.
- **Spring JDBC/DAO Advantages**: Simplifies database interactions, manages resources automatically, and provides a cleaner API for working with databases.
- **Working with Different Data Sources**: Spring