Spring JDBC

PRAGMATIC IT Learning & Outsourcing Center

www.pragmatic.bg

Lector: Stefan Vadev
Skype: stefanvadev77
E-mail: stefan.vadev@gmail.com
Facebook:
https://www.facebook.com/stefan.vadev.7

Summary

- JDBC
- Spring JDBC

JDBC

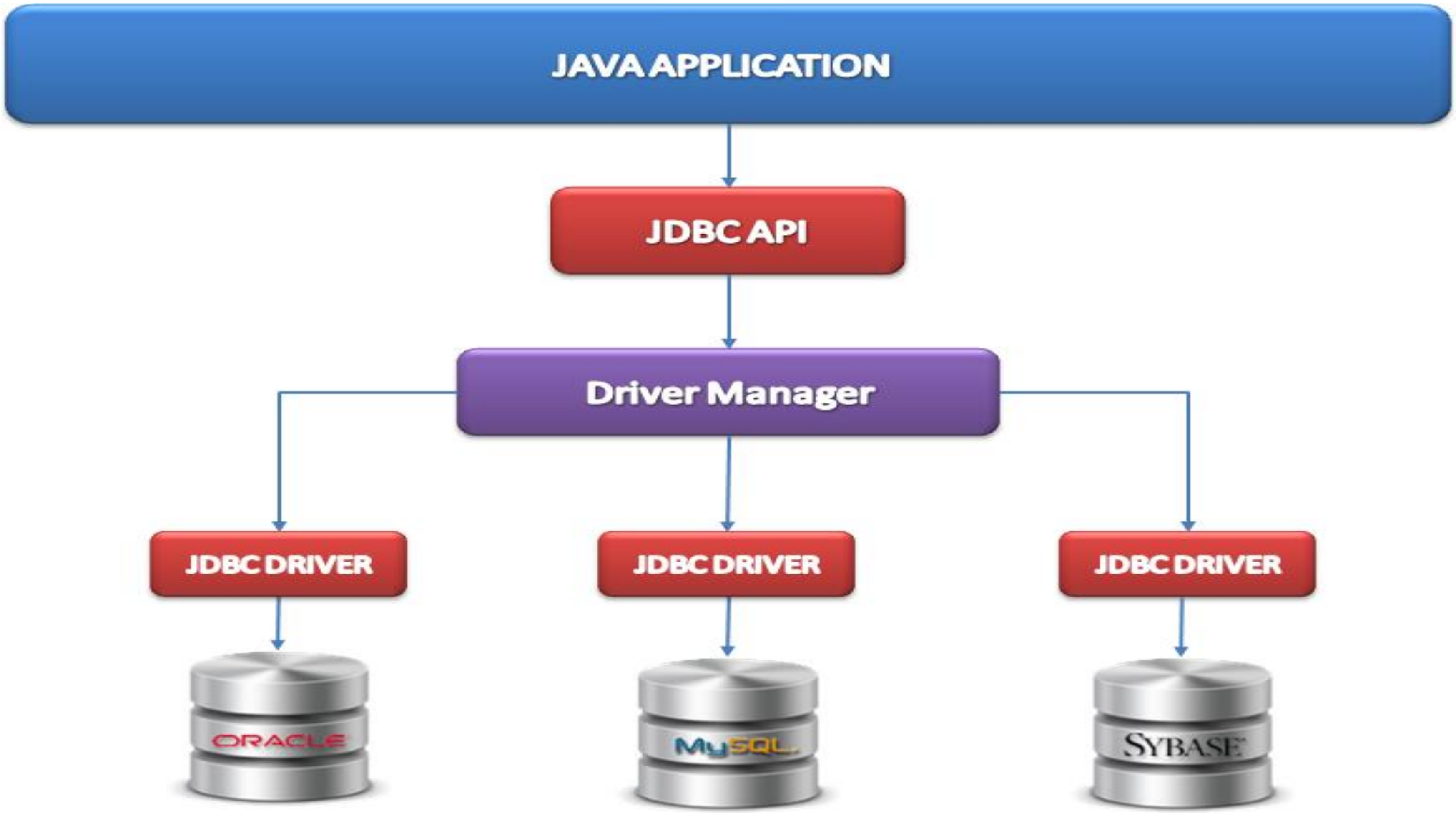
JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

JDBC

With JDBC you can easily:

- CRUD the DB
- Using the RowMapper create Java objects from db tables and vise versa
- Do all that with minimum configuration
- Create complex DB calls. You directly write the SQL, JDBC does the job for you
- Gain amazing performance
- Handle all DB related operations with a single objects
- **JdbcTemplate** or **NamedParameterJdbcTemplate** are the only 2 objects you will need (*checkout documentation*)
- Your job is limited to finding out which exact method you'll need (they provide many for all use cases)

How JDBC works



Spring JDBC

Overview

All the classes in Spring JDBC are divided into four separate packages:

•***core*** – the core functionality of JDBC. Some of the important classes under this package include *JdbcTemplate*, *SimpleJdbcInsert, SimpleJdbcCall* and *NamedParameterJdbcTemplate*.

•***datasource*** – utility classes to access datasource. It also has various datasource implementations that could be used for testing JDBC code outside the Java EE container.

•***object*** – DB access in an object oriented manner. It allows executing queries and returning the results as a business object. It maps the query results between the columns and properties of business objects.

•***support*** – support classes for classes under *core* and *object* packages. E.g. provides the *SQLException* translation functionality.

Configuration

Let’s start with some simple configuration of the datasource (we’ll use MySQL database for this example):

```
@Configuration
@ComponentScan
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername(“root");
        dataSource.setPassword(“root");

        return dataSource;
    }
}
```

Basic queries

The JDBC template is the main API through which we'll access most of the functionality that we're interested in:

- creation and closing of connections
- executing statements and stored procedure calls
- iterating over the *ResultSet* and returning results

Let's start with a simple example to see what the *JdbcTemplate* can do:

```
int result = jdbcTemplate.queryForObject(
   "SELECT COUNT(*) FROM employee", Integer.class);
```

And here's a simple **INSERT**:

```
public int addEmplyee(int id) {
   return jdbcTemplate.update(
      "INSERT INTO employee VALUES (?, ?, ?, ?)", 5, "Bill", "Gates", "USA");
}
```

Notice the standard syntax of providing parameters – using the `?` character. Next – let's look at an alternative to this syntax.

Queries with named params

To get **support for named parameters**, we'll use the other JDBC template provided by the framework – the *NamedParameterJdbcTemplate*.
This wraps the *JbdcTemplate* and provides an alternative to the traditional syntax using "*?*" to specify parameters. Under the hood, it substitutes the named parameters to JDBC "?" placeholder and delegates to the wrapped *JDCTemplate* to execute the queries:

```
SqlParameterSource namedParameters = new
MapSqlParameterSource().addValue("id", 1);
return namedParameterJdbcTemplate.queryForObject();

   "SELECT first_name FROM employee WHERE id = :id",
namedParameters, String.class
```

Mapping Query results to Java object

Another very useful feature is the ability to map query results to java objects – by implementing ***RowMapper*** interface.
For example – for every row returned by the query, Spring uses the row mapper to populate the java bean:

```
public class EmployeeRowMapper implements RowMapper<Employee> {
    @Override
    public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
        Employee employee = new Employee();

        employee.setId(rs.getInt(“id"));
        employee.setFirstName(rs.getString(“first_name"));
        employee.setLastName(rs.getString(“last_name"));
        employee.setAddress(rs.getString(“address"));

        return employee;
    }
}
```

Mapping Query results to Java object

We can now pass the row mapper to the query API and get fully populated Java objects back:

```
String query = "SELECT * FROM EMPLOYEE WHERE id =
?";
List<Employee> employees = jdbcTemplate.queryForObject(
    query, new Object[] { id }, new EmployeeRowMapper());
```

Show me the code!

Useful resources

- Useful resources:
- https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#jdbc
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html
- http://www.baeldung.com/spring-jdbc-jdbctemplate
- https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html
- https://www.tutorialspoint.com/jdbc/index.htm
- https://www.journaldev.com/2681/jdbc-tutorial
- http://www.vogella.com/tutorials/MySQLJava/article.html

Summary

- JDBC
- Spring JDBC