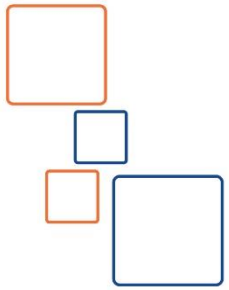
Spring Framework

THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND



Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Course Outline

- Lesson 1: Using JDBC into Spring Framework
- Lesson 2: Spring JDBC Module
- Lesson 3: Spring ORM
- Lesson 4: Spring ORM using Hibernate Framework
- Lesson 5: Spring ORM Hibernate Integration
- Lesson 6: Spring ORM using JPA Framework
- Lesson 7: Spring ORM JPA Integration
- *** **References & Recommended Reading**

Lesson 1

Using JDBC into Spring Framework

Using JDBC into Spring Framework

- Spring in this model only
  - Declare the datasource
  - Manage the lifecycle of this datasource.
- Using JDBC Connection native connection from predefined datasource
- You deal with datasource as Java normal classes
- You are managing every thing connection, transaction, exception handling, business logic

Using JDBC into Spring Framework (Ex.)

- We can also use JDBC API internal inside Spring application by declaring DataSource as it will be like driver manager

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/customerdb" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>
```

Using JDBC into Spring Framework (Ex.)

- You will inject your datasource into your DAO classes or connection factory classes.
- Your DAO Class as Follows:

```
public class JdbcCustomerDAO implements CustomerDAO {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using JDBC into Spring Framework (Ex.)

- You have to manage manually:
  - Define connection parameters.
  - Open the connection.
  - Specify the SQL statement.
  - Declare parameters and provide parameter values
  - Prepare and execute the statement.
  - Set up the loop to iterate through the results (if any).
  - Do the work for each iteration.
  - Process any exception.
  - Handle transactions.
  - Close the connection, the statement, and the resultset.

Using JDBC into Spring Framework (Ex.)

- You can obtain new connection by calling:
  - Connection connection = dataSource.getConnection();

Using JDBC into Spring Framework (Ex.)

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) VALUES (?, ?, ?)";
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        PreparedStatement preparedStatement = conn.prepareStatement(sql);
        preparedStatement.setInt(1, customer.getId());
        preparedStatement.setString(2, customer.getName());
        preparedStatement.setInt(3, customer.getAge());
        preparedStatement.executeUpdate();
        preparedStatement.close();
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
```

Using JDBC into Spring Framework (Ex.)

Java™ Education and Technology Services

```
        finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    System.err.println(e.getMessage());
                }
            }
        }
}
```

Using JDBC into Spring Framework (Ex.)

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Connection conn = null;
    Customer customer = null;
    try {
        conn = dataSource.getConnection();
        PreparedStatement preparedStatement = conn.prepareStatement(sql);
        preparedStatement.setInt(1, customerId);
        ResultSet resultSet = preparedStatement.executeQuery();
        if (resultSet.next()) {
            customer = new Customer();
            customer.setId(resultSet.getInt("id"));
            customer.setName(resultSet.getString("name"));
            customer.setAge(resultSet.getInt("age"));
        }
        resultSet.close();
        preparedStatement.close();
    }
```

Using JDBC into Spring Framework (Ex.)

```
        catch (SQLException e) {
            System.err.println(e.getMessage());
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    System.err.println(e.getMessage());
                }
            }
        }
        return customer;
    }
```

Lesson 2

Spring JDBC Module

Spring JDBC Framework

- Spring Framework JDBC provide abstraction of JDBC Layer.
- The Spring Framework's JDBC framework consists of four different packages:
    1. org.springframework.jdbc.core
    - The org.springframework.jdbc.core package contains
        - The JdbcTemplate class and its various callback interfaces, plus a variety of related classes.
    - A subpackage named org.springframework.jdbc.core.simple contains
        - The SimpleJdbcInsert and SimpleJdbcCall classes.
    - Another subpackage named org.springframework.jdbc.core.namedparam contains
        - the NamedParameterJdbcTemplate class and the related support classes.

Spring JDBC Framework

- The Spring Framework's JDBC framework consists of four different packages:

  2. org.springframework.jdbc.datasource

  - The org.springframework.jdbc.datasource package contains
    - A utility class for easy DataSource access and various simple DataSource implementations that you can use for testing and running unmodified JDBC code outside of a Java EE container.
  - A subpackage named org.springfamework.jdbc.datasource.embedded
    - Provides support for creating embedded databases by using Java database engines, such as HSQL, H2, and Derby.

Spring JDBC Framework

- The Spring Framework's JDBC framework consists of four different packages:

3. org.springframework.jdbc.object

- The org.springframework.jdbc.object package contains
  - Classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects.
- This approach is modeled by JDO (Java Data Objects), although objects returned by queries are naturally disconnected from the database.
- This higher-level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package

Spring JDBC Framework

- The Spring Framework's JDBC framework consists of four different packages:

4. org.springframework.jdbc.support

  - The org.springframework.jdbc.support package
    - Provides SQLException translation functionality and some utility classes.
  - Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package.
  - This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling.
  - All translated exceptions are unchecked, which gives you the option of catching the exceptions from which you can recover while letting other exceptions be propagated to the caller.

Spring JDBC Framework (Ex.)

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, the statement, and the resultset.	X	

Using JdbcTemplate

- JdbcTemplate is the central class in the JDBC core package.
- It handles the creation and release of resources, which helps you avoid common errors, such as forgetting to close the connection.
- It performs the basic tasks of the core JDBC workflow (such as statement creation and execution), leaving application code to provide SQL and extract results.
- The JdbcTemplate class:
  - Runs SQL queries
  - Updates statements and stored procedure calls
  - Performs iteration over ResultSet instances and extraction of returned parameter values.
  - Catches JDBC exceptions and translates them to the generic.

Using JdbcTemplate (Ex.)

- Instances of the JdbcTemplate class are thread safe.
- Single instance of a JdbcTemplate is sufficient.
- Safely inject this shared reference into multiple DAOs

Using JdbcTemplate (Ex.)

- Your DAO Class as Follows:

```
public class JdbcCustomerDAO implements CustomerDAO {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve the number of rows in a customer table:

```
@Override
public int count() {
    String SQL = "select count(*) from Customer";
    int rowCount = jdbcTemplate.queryForObject(SQL, Integer.class);
    return rowCount;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve the number of rows in a customer where age is greater than or equal the input parameter:

```
@Override
public int countByAgeGreaterThan(int age) {
    String SQL = "select count(*) from Customer where age >= ?";
    int rowCount = jdbcTemplate.queryForObject(SQL,
            new Object[]{age}, Integer.class);
    return rowCount;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve Customer Object (Single Object) by customer id:

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Object[] args = new Object[]{customerId};
    SqlRowSet rowset = jdbcTemplate.queryForRowSet(sql, args);
    Customer customer = null;
    if (rowset.next()) {
        customer = new Customer();
        customer.setId(rowset.getInt("id"));
        customer.setName(rowset.getString("name"));
        customer.setAge(rowset.getInt("age"));
    }
    return customer;
}
```

Java™ Education and Technology Services

Using JdbcTemplate (Ex.)

Querying (SELECT)

- We find some of challenges to bind the rowset into the entities object, So spring provide some classes to auto-bind between the rowset and the entities.
- Spring Provide auto-bind by org.springframework.jdbc.core.RowMapper interface.
- You can do this by
  - Either by making your class that implement RowMapper interface.
  - Or by using built-in classes that implement RowMapper interface.

Using JdbcTemplate (Ex.)

Querying (SELECT)

- Using BeanPropertyRowMapper that use bean property to identify the relation between the bean property and the column in table.
- The following function retrieve Customer Object (Single Object) by customer id:

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Object[] args = new Object[]{customerId};
    Customer customer = jdbcTemplate.queryForObject(
            sql, args, new BeanPropertyRowMapper<>(Customer.class));
    return customer;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- Using Custom Row Mapper that you are define the relation between the bean property and the column in table.

```
public class CustomerRowMapper implements RowMapper {

    @Override
    public Object mapRow(ResultSet rs, int rowNum)
            throws SQLException {
        Customer customer = new Customer();
        customer.setId(rs.getInt("id"));
        customer.setName(rs.getString("name"));
        customer.setAge(rs.getInt("age"));
        return customer;
    }
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve Customer Object (Single Object) by customer id:

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Customer customer = (Customer) jdbcTemplate.queryForObject(
            sql, new Object[]{customerId}, new CustomerRowMapper());
    return customer;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve All Customer Objects (Multi-Object) :

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customers = new ArrayList<>();
    List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
    for (Map row : rows) {
        Customer customer = new Customer();
        customer.setId((row.get("id")));
        customer.setName((String) row.get("name"));
        customer.setAge((int) row.get("age"));
        customers.add(customer);
    }
    return customers;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- Using Custom Result Set Extractor that you are define the relation between the bean property and the column in table

```
public class CustomerResultSetExtractor
        implements ResultSetExtractor<List<Customer>> {

    @Override
    public List<Customer> extractData(ResultSet resultSet)
            throws SQLException, DataAccessException {
        List<Customer> customers = new ArrayList<>();
        while (resultSet.next()) {
            Customer customer = new Customer();
            customer.setId(resultSet.getInt("id"));
            customer.setName(resultSet.getString("name"));
            customer.setAge(resultSet.getInt("age"));
            customers.add(customer);
        }
        return customers;
    }
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve All Customer Objects (Multi-Object) :

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customers = jdbcTemplate
            .query(sql, new CustomerResultSetExtractor());
    return customers;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve All Customer Objects (Multi-Object) using BeanPropertyRowMapper:

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customers = jdbcTemplate.query(
            sql, new BeanPropertyRowMapper<>(Customer.class));
    return customers;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve All Customer Objects (Multi-Object) using Custom Row Mapper:

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customer = jdbcTemplate.query(
            sql, new CustomerRowMapper());
    return customer;
}
```

Using JdbcTemplate (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) VALUES (?, ?, ?)";
    Object[] args = new Object[]{customer.getId(),
        customer.getName(), customer.getAge()};
    jdbcTemplate.update(sql, args);
}
```

Using JdbcTemplate (Ex.)

Updating (update)

- The following function update name and age of customer where customer id is passed from parameter:

```
@Override
public void update(Customer customer) {
    String sql = "update CUSTOMER set name = ?,age=? where id = ?";
    Object[] args = new Object[]{customer.getName(),
        customer.getAge(), customer.getId()};
    jdbcTemplate.update(sql, args);
}
```

Using JdbcTemplate (Ex.)

Updating (delete)

- The following function delete customer by customer id which is passed in parameter:

```
@Override
public void delete(int customerId) {
    String sql = "delete from CUSTOMER where id = ?";
    Object[] args = new Object[]{customerId};
    jdbcTemplate.update(sql, args);
}
```

Using JdbcTemplate (Ex.)

- You can use the execute method to run any arbitrary SQL.
- Consequently, the method is often used for DDL statements.
- It is heavily overloaded with variants that take callback interfaces, binding variable arrays, and so on.
- The following example creates a table:

```
@Override
public void createXTable() {
    String sql = "create table xxx (id integer, name varchar(100))";
    jdbcTemplate.execute(sql);
}
```

Using NamedParameterJdbcTemplate

- Spring Also provide another class instead of JdbcTemplate called NamedParameterJdbcTemplate.
- Which adds only the usage of named parameter instead of indexing parameter.
- You can declare it by same way as JdbcTemplate.

```
public class JdbcCustomerDAO1 implements CustomerDAO {

    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        namedParameterJdbcTemplate
                = new NamedParameterJdbcTemplate(dataSource);
    }
```

Using NamedParameterJdbcTemplate (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) "
            + "VALUES (:id, :name, :age)";
    Map<String, Object> args = new HashMap<>();
    args.put("id", customer.getId());
    args.put("name", customer.getName());
    args.put("age", customer.getAge());
    namedParameterJdbcTemplate.update(sql, args);
}
```

Using NamedParameterJdbcTemplate (Ex.)

Querying (SELECT)

- If your query didn't take any arguments just pass an empty map
- The following function retrieve the number of rows in a customer table:

```
@Override
public int count() {
    String SQL = "select count(*) from Customer";
    Map<String, Object> args = new HashMap<>();
    int rowCount = namedParameterJdbcTemplate
            .queryForObject(SQL, args, Integer.class);
    return rowCount;
}
```

- Also You could use all query classes as you used before like (RowMapper, BeanPropertyRowMapper, ResultSetExtractor, etc)

Using SimpleJdbcTemplate

- For earlier versions in Spring till version 4
- Spring Provide a class called SimpleJdbcTemplate instead of JdbcTemplate
- This class inherit from JdbcTemplate and provide the dynamic parameter.

```
int update(String sql,Object... parameters)
```

```
String query="update employee set name=? where id=?";
return template.update(query,e.getName(),e.getId());
```

Using JdbcDaoSupport

- Spring Also provide another class Called JdbcDaoSupport.
- You will define your class that inherit from JdbcDaoSupport and use the function getJdbcTemplate() from super class to get the instance of JdbcTemplate.
- Your DAO Class as Follows:

```
public class JdbcCustomerDAO2 extends JdbcDaoSupport
        implements CustomerDAO {
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using JdbcDaoSupport (Ex.)

Querying (SELECT)

- All Configuration are identical as JdbcTemplate, the only difference is obtaining an object from JdbcTemplate from the super class.
- The following function retrieve the number of rows in a customer table:

```
@Override
public int count() {
    String SQL = "select count(*) from Customer";
    int rowCount = getJdbcTemplate().queryForObject(SQL, Integer.class);
    return rowCount;
}
```

- Also You could use all query classes as you used before like (RowMapper, BeanPropertyRowMapper, ResultSetExtractor, etc)

Using JdbcDaoSupport (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) "
            + "VALUES (?, ?, ?)";
    Object[] args = new Object[]{customer.getId(),
        customer.getName(), customer.getAge()};
    getJdbcTemplate().update(sql, args);
}
```

Using JdbcDaoSupport (Ex.)

Why ???

- There is another benefit to deal with JdbcDaoSupport instead of JdbcTemplate because:
  - You can get the current connection with JDBC Connection by using getConnection();
  - Also You can release connection by calling releaseConnection(); to force release for the current connection.

Using SimpleJdbcDaoSupport

- For earlier versions in Spring till version 4
- Spring Provide a class called SimpleJdbcDaoSupport instead of JdbcDaoSupport
- This class inherit from JdbcDaoSupport and provide an instance from SimpleJdbcTemplate.
- Your DAO Class as Follows:

```
public class JdbcCustomerDAO extends SimpleJdbcDaoSupport
        implements CustomerDAO {
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using SimpleJdbcDaoSupport (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insertNamedParameter(Customer customer) {
    String sql = "INSERT INTO CUSTOMER "
            + "(id, name, age) VALUES (:id, :name, :age)";
    Map<String, Object> parameters = new HashMap<>();
    parameters.put("id", customer.getCustId());
    parameters.put("name", customer.getName());
    parameters.put("age", customer.getAge());
    getSimpleJdbcTemplate().update(sql, parameters);
}
```

Simplifying JDBC Operations

- The SimpleJdbcInsert and SimpleJdbcCall classes provide a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver.
- This means that you have less to configure up front, although you can override or turn off the metadata processing if you prefer to provide all the details in your code.
- Inserting Data by Using SimpleJdbcInsert.
- Calling a Stored Procedure with SimpleJdbcCall.

Inserting Data by Using SimpleJdbcInsert

- We start by looking at the SimpleJdbcInsert class with the minimal amount of configuration options.
- You should instantiate the SimpleJdbcInsert in the data access layer's initialization method.
- You do not need to subclass the SimpleJdbcInsert class. Instead, you can create a new instance and set the table name by using the withTableName method.
- Configuration methods for this class follow the fluid style that returns the instance of the SimpleJdbcInsert, which lets you chain all configuration methods.

Inserting Data by Using SimpleJdbcInsert (Ex.)

- Your DAO Class as Follows:

```
public class JdbcCustomerDAO3 implements CustomerDAO {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertCustomer;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
        insertCustomer = new SimpleJdbcInsert(dataSource)
                .withTableName("customer")
                .usingColumns("id", "name", "age");
    }
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    Map<String, Object> parameters = new HashMap<>(3);
    parameters.put("id", customer.getId());
    parameters.put("name", customer.getName());
    parameters.put("age", customer.getAge());
    insertCustomer.execute(parameters);
}
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

- If you have auto-generated keys in database so you could choose to use usingGeneratedKeyColumns() method.
- The following example uses the same insert as the preceding example, but, instead of passing in the id, it retrieves the auto-generated key and sets it on the new Customer object.
- When it creates the SimpleJdbcInsert, in addition to specifying the table name, it specifies the name of the generated key column with the usingGeneratedKeyColumns method.

Inserting Data by Using SimpleJdbcInsert (Ex.)

- Your DAO Class as Follows:

```
public class JdbcCustomerDAO3 implements CustomerDAO {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertCustomer;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
        insertCustomer = new SimpleJdbcInsert(dataSource)
                .withTableName("customer")
                .usingColumns( "name", "age")
                .usingGeneratedKeyColumns("id");
    }
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    Map<String, Object> parameters = new HashMap<>(2);
    parameters.put("name", customer.getName());
    parameters.put("age", customer.getAge());
    insertCustomer.execute(parameters);
}
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

- Using SqlParameterSource to Provide Parameter Values:
- Using a Map to provide parameter values works fine, but it is not the most convenient class to use.
- Spring provides a couple of implementations of the SqlParameterSource interface that you can use instead.
  - The first one is BeanPropertySqlParameterSource, which is a very convenient class if you have a JavaBean-compliant class that contains your values.
  - It uses the corresponding getter method to extract the parameter values.

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    SqlParameterSource parameters
            = new BeanPropertySqlParameterSource(customer);
    insertCustomer.execute(parameters);
}
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The second one is the MapSqlParameterSource that resembles a Map but provides a more convenient addValue method that can be chained..
- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("name", customer.getName())
            .addValue("age", customer.getAge());
    insertCustomer.execute(parameters);
}
```