Project Description

I tried to create a lightweight java framework to better understand core concepts of spring boot and similar technologies like Jakarta EE. For now this framework provide features such as dependency injection, object-relation mapping (ORM) and database connection pooling.

Dependency injection is a design patter than ensures that objects receive the dependencies they require without needing to construct them itself. In our framework, we used constructor injection. Similarly object-relation mapping simplifies database interactions by providing an ORM layer that maps java objects to relation database tables. Through annotations like @Entity, @Column and @PrimaryKey our framework automates the mapping process. Database connection pooling is another important feature, by reusing connections instead of creating new ones for each request, connection pooling minimizes resource usage and improve performance.

Annotations and Reflection:

Java annotations are used to provide meta data for your Java code, Being meta data. Java annotations do not directly affect the execution of your code. Annotations are defined in their own file just like a Java class or interface. Normally java annotations are not present in your Java code after compilation. it is possible, however, to define your own annotations that are available at runtime.

Here is custom Java annotation example:

```
@interface MyAnnotation {
    String   value();
}
```

These annotations can be accessed via Java Reflection. Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

In other words java annotations are just markers, with some properties, but with no behavior of their own. So whenever we see an annotation there is a piece of Java code that looks for that annotation and contains the real intelligence to do something useful with it.
Unfortunately is pretty difficult to identify exactly which piece of code is processing the annotation if it is inside a library.

Example: Insert Method for ORM

To illustrate the usage of our ORM, let's consider an example of inserting records into a database table. We will use the Account record (Records in Java are immutable data classes, providing getters, setters, toString, and hashCode methods out of the box), which represents a simple entity with annotated fields.

```
@Entity(table = "Account")
public record Account(@PrimaryKey long accountId, @Column String username, @Column double balance,
@Column boolean isActive) {
}
```

Our objective here is to obtain the names of variables and constructors at runtime. This information is important for executing SQL inserts via JDBC (Java Database Connectivity). We aim to map the fields of a class to SQL string, which we will then execute against the database. The first step involves creating custom annotations. These annotations server to mark the fields of our Java objects, indicating their role in the database schema.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Column {}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface PrimaryKey {}
```

The next step, after defining the record Account and custom annotations would be implementing the Entity Manager class, which is responsible for managing entities and executing database operations.

```
public class EntityManager<T> {
    private Connection connection;
```

```
    private AtomicLong id = new AtomicLong(0L);

    public static <T> EntityManager<T> getConnection() throws SQLException {
        return new EntityManager<>();
    }

    private EntityManager() throws SQLException {
        this.connection = DriverManager.getConnection("jdbc:mysql://localhost:3307/user_security",
"zeljko", "zeljkoo123");
    }

    public void write(T t) throws IllegalArgumentException, IllegalAccessException, SQLException {
        Class<? extends Object> clazz = t.getClass();
        Field[] declaredFields = clazz.getDeclaredFields();
        Field pkey = null;

        List<Field> columns = new ArrayList<>();
        StringJoiner joiner = new StringJoiner(",");

        for (Field field : declaredFields) {
            if (field.isAnnotationPresent(PrimaryKey.class)) {
                pkey = field;
            } else if (field.isAnnotationPresent(Column.class)) {
                columns.add(field);
            }
        }

        int columnsSize = columns.size() + 1;
        String qMarks = IntStream.range(0, columnsSize)
                .mapToObj(e -> "?")
                .collect(Collectors.joining(",", "(", ")"));

        columns.forEach(field -> joiner.add(field.getName()));

        String sql = String.format("INSERT INTO %s (%s,%s) VALUES %s",
                clazz.getSimpleName(),
                pkey.getName(),
                joiner,
                qMarks);

        System.out.println(sql);


        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            if (pkey.getType() == long.class) {
                pstmt.setLong(1, id.incrementAndGet());
            }
            int index = 2;
            for (Field field : columns) {
                field.setAccessible(true);
                if (field.getType() == int.class) {
                    pstmt.setInt(index++, (int) field.get(t));
                } else if (field.getType() == String.class) {
                    pstmt.setString(index++, (String) field.get(t));
                } else if (field.getType() == double.class) {
                    pstmt.setDouble(index++, (double) field.get(t));
                } else if (field.getType() == boolean.class) {
                    pstmt.setBoolean(index++, (boolean) field.get(t));
                }
            }
            pstmt.executeUpdate();
        }

    }
}
```

In the EntityManager class which takes `<T>` any class, the write method is responsible for inserting records into the database. It dynamically constructs the SQL insert statement based on the provided object's fields and annotations, and the executes the statement using JDBC.

In this snippet, after retrieving the Account object using the TransactionService, we obtain an instance of the EntityManager using the static getConnection method. and we call the **write** method of the EntityManager to insert the Account object into the database.

```
public class Application {
    public static void main(String[] args) throws Exception {

        // Initialize application context
        ApplicationContext applicationContext = new ApplicationContext(ApplicationConfig.class);

        TransactionService transactionService = applicationContext.getInstance(TransactionService.class);
        Account account = transactionService.findAccountById(123);
        System.out.println(account);

        // Get EntityManager instance and insert the account into the database
        EntityManager<Account> entityManager = EntityManager.getConnection();
        entityManager.write(account);
    }
}
```

Suppose we want to insert records into a database without using ORM. In this scenario, we would need to manually construct the SQL insert statements and execute them using JDBC.

```
public class AccountRepository {

    public static void insertAccount(Account account) {
        String sql = "INSERT INTO Account (accountId, username, balance, isActive) VALUES (?, ?, ?, ?)";

        try (Connection connection = DatabaseManager.getConnection();
             PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setLong(1, account.getAccountId());
            pstmt.setString(2, account.getUsername());
            pstmt.setDouble(3, account.getBalance());
            pstmt.setBoolean(4, account.isActive());

            pstmt.executeUpdate();

            System.out.println("Account inserted successfully.");
        } catch (SQLException e) {
            System.err.println("Error inserting account: " + e.getMessage());
        }
    }
}
```

In this approach we manually construct the SQL insert statement and handle database operations. While this method provides more control over the SQL queries, it requires writing boilerplate code for each entity and operations.