

# Лекция 7

## Разработка АС реального времени (часть 2)

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail: [zorina\\_n@mail.ru](mailto:zorina_n@mail.ru)

# Тема лекции:

## «Работа с базами данных. Принципы SOLID»

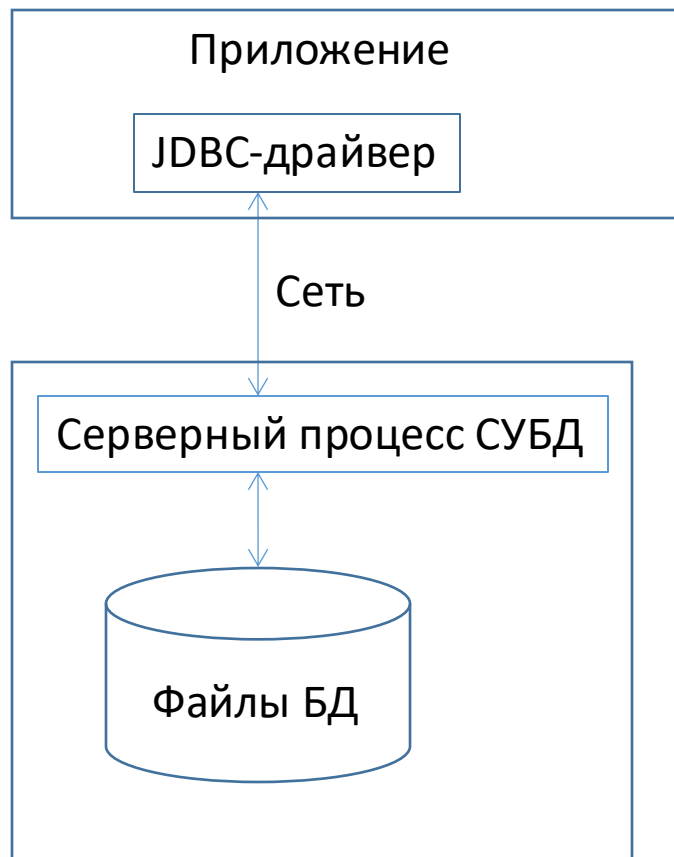
# Реляционные базы данных

```
CREATE TABLE books (  
  id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
  author VARCHAR(100) NOT NULL,  
  title VARCHAR(100) NOT NULL  
)
```

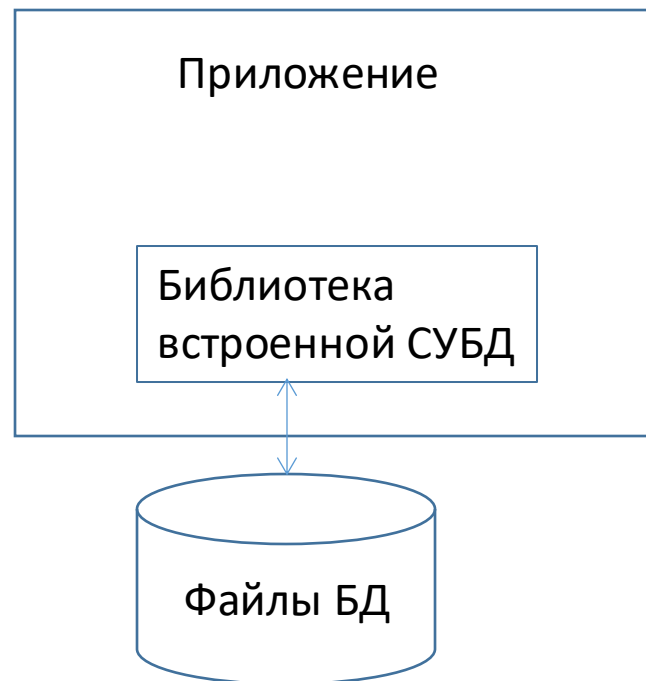
books

id	author	title
123	Л.Н. Толстой	Война и мир
124	Дж. Роулинг	Гарри Поттер
...	...	...

# Реляционные базы данных



**Клиент-серверная СУБД**



**Встроенная СУБД**

# Реляционные базы данных

Клиент-серверные СУБД (системы управления базами данных):

- PostgreSQL
- Oracle
- MySQL
- Microsoft SQL Server

# Реляционные базы данных

Встроенные СУБД:

- H2
- Apache Derby

# Реляционные базы данных

Язык SQL, под-язык Data Manipulation Language (DML):

- Create:  
**INSERT INTO books (author, title)**  
**VALUES**  
**('Л.Н. Толстой', 'Война и мир')**
- Retrieve:  
**SELECT id, author, title**  
**FROM books**
- Update:  
**UPDATE books**  
**SET author = 'Л.Н. Толстой', title = 'Война и мир'**  
**WHERE id = 1**
- Delete:  
**DELETE FROM books WHERE id = 1**

# Реляционные базы данных

Пакет java.sql – классы JDBC (Java Database Connectivity):

- Connection
- PreparedStatement
- ResultSet
- SQLException
- DriverManager
- ...



# Реляционные базы данных

```
import java.sql.*;
```

```
...
```

```
try (Connection connection = DriverManager.getConnection("jdbc:...")) {  
    try (PreparedStatement ps = connection.prepareStatement(  
        "create table if not exists books (" +  
        " id int generated by default as identity primary key," +  
        " author varchar(100) not null," +  
        " title varchar(100) not null" +  
        ")"  
    )) {  
        ps.execute();  
    }  
}
```

# Реляционные базы данных

JDBC-драйвер – библиотека для выполнения SQL-запросов для конкретной БД

`java.sql.Connection:`

`oracle.jdbc.OracleConnection (ojdbc18.jar)`

`org.postgresql.jdbc.PgConnection (postgresql.jar)`

`org.h2.jdbc.JdbcConnection (h2.jar)`

Если мы используем переносимый SQL, то программа сможет работать с любой БД, поддерживающей стандарты SQL

# JDBC URL

PostgreSQL: **`jdbc:postgresql://localhost:5432/postgres`**

Oracle: **`jdbc:oracle:thin:@//localhost:1521/orcl`**

H2: **`jdbc:h2:~/example_db`**

H2 – встроенная СУБД:

```
dependencies {  
    runtimeOnly("com.h2database:h2:1.4.200")  
}
```

Connection c = DriverManager.getConnection(**`"jdbc:h2:~/example_db"`**)

# Реляционные базы данных

JDBC-драйвер подключается через конфигурацию `runtimeOnly`. Это значит, что его классы недоступны на этапе компиляции. Мы используем только интерфейсы и классы из пакета `java.sql`:

```
Connection c = DriverManager.getConnection("jdbc:h2:~/example_db");  
// Создается объект org.h2.jdbc.JdbcConnection  
  
PreparedStatement ps = c.prepareStatement("...");  
// Возвращается объект org.h2.jdbc.JdbcPreparedStatement
```

# Реляционные базы данных

Как DriverManager понимает, какой класс нужно использовать при обращении к методу getConnection?

Для этого используется интерфейс java.sql.Driver:

```
public interface Driver {  
    boolean acceptsURL(String url);  
    Connection connect(String url, Properties info);  
}
```

# Реляционные базы данных

DriverManager перебирает все зарегистрированные драйверы и вызывает у них метод `acceptsURL`. Если он возвращает `true`, то вызывается метод `connect`.

`java.sql.Driver:`

`org.postgresql.Driver`

`oracle.jdbc.OracleDriver`

`org.h2.Driver`

# Реляционные базы данных

Мы не регистрируем драйверы вручную, они ищутся автоматически в ClassPath при помощи встроенного в Java механизма сервисов. Этот механизм немного похож на сканирование классов в Spring, но отличается от него:

- это встроенный в Java механизм
- вместо аннотаций используется описание в папке jar-архива META-INF
- вместо dependency injection используется вызов `java.util.ServiceLoader.load`

# Реляционные базы данных

```
ServiceLoader<Driver> drivers = ServiceLoader.load(Driver.class);  
String url = "jdbc:h2:~/example_db";  
for (Driver driver : drivers) {  
    System.out.println(driver.getClass());  
    System.out.println(driver.acceptsURL(url));  
}
```

Вывод:

```
class org.h2.Driver  
true
```



# Реляционные базы данных

При загрузке jar-файла Java ищет в нем папку META-INF/services. Если она есть, то она должна содержать файлы с именем, совпадающим с полным именем класса сервиса (в нашем случае `java.sql.Driver`). Файл должен содержать имя класса, реализующего этот сервис.

h2-1.4.200.jar:

- META-INF/services
  - `java.sql.Driver`, содержимое: `"org.h2.Driver"`

# Реляционные базы данных

В случае, когда зарегистрировано несколько реализаций `java.sql.Driver`, подходящий драйвер выбирается по JDBC URL.

# JDBC – INSERT

```
try (PreparedStatement ps = connection.prepareStatement(
    "insert into books (id, author, title) values (?, ?, ?)"
)) {
    ps.setInt(1, 123);
    ps.setString(2, "Л.Н. Толстой");
    ps.setString(3, "Война и мир");
    int inserted = ps.executeUpdate();
}
```

Параметризованные запросы: параметры задаются символом "?", а их значения – через вызовы `PreparedStatement.setXXX`. Нумерация параметров идет от 1.

# JDBC – PreparedStatement

**Нужно всегда использовать задание параметров запроса через параметры!**

**Нельзя "вклеивать" значения параметров в сам SQL-запрос:**

```
try (PreparedStatement ps = connection.prepareStatement(
    "insert into books (id, author) values (" + id + ", '" + author + "')"
)) {
    int inserted = ps.executeUpdate();
}
```

# JDBC – PreparedStatement

Иначе ваше приложение может быть подвержено уязвимости SQL Injection:

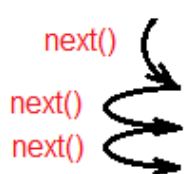


# JDBC – SELECT

```
try (PreparedStatement ps = connection.prepareStatement(
    "select id, author from books where title = ?"
)) {
    ps.setString(1, "Война и мир");
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            int id = rs.getInt(1);
            String author = rs.getString(2);
            System.out.println(id + ": " + author);
        }
    }
}
```

# JDBC – ResultSet

```
Select Emp_Id, Emp_No, Emp_Name from Employee
```



	1	2	3
	EMP_ID	EMP_NO	EMP_NAME
▶	1	7839	E7839 ... KING ...
	2	7566	E7566 ... JONES ...
	3	7902	E7902 ... FORD ...
	4	7369	E7369 ... SMITH ...
	5	7698	E7698 ... BLAKE ...
	6	7499	E7499 ... ALLEN ...
	7	7521	E7521 ... WARD ...
	8	7654	E7654 ... MARTIN ...
	9	7782	E7782 ... CLARK ...
	10	7788	E7788 ... SCOTT ...
	11	7844	E7844 ... TURNER ...
	12	7876	E7876 ... ADAMS ...
	13	7900	E7900 ... ADAMS ...
	14	7934	E7934 ... MILLER ...

```
ResultSet rs = statement.executeQuery( );
```

```
while (rs.next()) {  
    int empId = rs.getInt(1);  
    String empNo = rs.getString(2);  
    String empName = rs.getString("Emp_Name");  
}
```

# JDBC – INSERT

Получение сгенерированных значений столбцов:

```
try (PreparedStatement ps = connection.prepareStatement(
    "insert into books (author, title) values (?, ?)", new String[] {"id"}
)) {
    ps.setString(1, "Л.Н. Толстой");
    ps.setString(2, "Война и мир");
    int inserted = ps.executeUpdate();
    try (ResultSet rs = ps.getGeneratedKeys()) {
        rs.next();
        int id = rs.getInt(1);
    }
}
```



# Пулы соединений

Интерфейс Connection описывает активное соединение с БД (в случае клиент-серверной БД – это сетевое соединение).

```
public class BookRepository {  
    public List<Book> getAllBooks() {  
        try (Connection connection = ???) {  
        }  
    }  
}
```

# Пулы соединений

Плохие варианты работы с соединением:

1. Открывать соединение каждый раз в методах BookRepository:

```
Connection connection = DriverManager.getConnection("...")
```

Соединение открывается медленно (порядка 0.1 секунд). Наше приложение не будет справляться с большой нагрузкой

# Пулы соединений

Плохие варианты работы с соединением:

2. Держать Connection в поле BookRepository. Как правило, только один поток может одновременно работать с одним соединением. Хотя соединения обычно потокобезопасны, но это достигается с помощью синхронизации. Поэтому при большом количестве параллельных запросов к BookRepository только один из них будет работать, остальные будут ждать.

## Пулы соединений

Поэтому используются пулы соединений (по аналогии с пулами потоков).

Например, пул с максимальным количеством соединений = 20 позволяет параллельно работать с БД 20 потокам.

При этом соединения не закрываются, а по возможности переиспользуются.

# Пулы соединений

Интерфейс `javax.sql.DataSource` задает абстрацию пула соединений:

```
public interface DataSource {  
    Connection getConnection();  
}
```

Для возврата соединения в пул можно его закрыть – при этом реально соединение с БД не закрывается (т.к. `DataSource` возвращает не `org.h2.jdbc.JdbcConnection`, а "обертку", которая реализует `java.sql.Connection` и перенаправляет вызовы методов `org.h2.jdbc.JdbcConnection`, кроме метода `close`).

# Пулы соединений

```
public class MyPooledConnection implements Connection {  
    private final Connection realConnection;  
    private final MyPool pool;  
    public PreparedStatement prepareStatement(String sql) {  
        return realConnection.prepareStatement(sql);  
    }  
    public void close() {  
        pool.free(realConnection);  
    }  
}
```

# Пулы соединений

Есть несколько реализаций пулов соединений:

- HikariCP
- Apache Commons DBCP
- C3PO

# Пулы соединений

## Пример конфигурации HikariCP:

```
HikariConfig config = new HikariConfig();  
config.setAutoCommit(false);  
config.setJdbcUrl("jdbc:h2:~/example_db");  
DataSource dataSource = new HikariDataSource(config);
```



# Работа с БД в приложении

```
public class BookRepository {  
    private final DataSource ds;  
    public BookRepository(DataSource ds) { this.ds = ds; }  
  
    public List<Book> getAllBooks() {  
        try (Connection connection = ds.getConnection()) {  
            try (PreparedStatement ps = connection.prepareStatement("select id, author, title from books")) {  
                try (ResultSet rs = ps.executeQuery()) {  
                    List<Book> books = new ArrayList<>();  
                    while (rs.next()) {  
                        int id = rs.getInt(1);  
                        String author = rs.getString(2);  
                        String title = rs.getString(3);  
                        books.add(new Book(id, author, title));  
                    }  
                    return books;  
                }  
            }  
        }  
    }  
    catch (SQLException ex) {  
        throw new RuntimeException(ex);  
    }  
}
```

# Работа с БД в приложении

```
public Optional<Book> updateBook(int id, BookDetails details) {  
    checkDetails(details);  
    try (Connection connection = ds.getConnection()) {  
        try (PreparedStatement ps = connection.prepareStatement(  
            "update books set author = ?, title = ? where id = ?"  
        )) {  
            ps.setString(1, details.getAuthor());  
            ps.setString(2, details.getTitle());  
            ps.setInt(1, id);  
            if (ps.executeUpdate() > 0) {  
                return Optional.of(new Book(id, details.getAuthor(), details.getTitle()));  
            } else {  
                return Optional.empty();  
            }  
        }  
    }  
    catch (SQLException ex) {  
        throw new RuntimeException(ex);  
    }  
}
```

# Работа с БД в приложении

@Configuration

```
public class BookConfig {
```

```
    private final DataSource ds;
```

```
    public BookConfig(DataSource ds) {  
        this.ds = ds;  
    }
```

@Bean

```
public BookRepository bookRepository() {  
    BookRepository bookRepo = new BookRepository(ds);  
    bookRepo.init(); // Создаем структуру БД при необходимости  
    return bookRepo;  
}
```

@Bean

```
public BookService bookService() {  
    return new BookService(bookRepository());  
}  
}
```

# Работа с БД в приложении

```
dependencies {  
    implementation("org.springframework.boot:spring-boot-starter-web:2.4.5")  
    implementation("org.springframework.boot:spring-boot-starter-jdbc:2.4.5")  
    runtimeOnly("com.h2database:h2:1.4.200")  
}
```

Исходный код: <https://github.com/osobolev/restdemo2>

## Работа с БД в приложении

Класс BookService просто перенаправляет вызовы методов в BookRepository. Единственная логика, которая в нем остается – валидация параметров addBook и updateBook.

# Работа с БД в приложении

Параметры соединения с БД (а также другие параметры приложения Spring Boot) задаются в файле `application.properties`. Этот файл может лежать в:

1. Корень ClassPath (в проекте файл при этом находится в `main/resources`)
2. Папка `/config` в ClassPath (в проекте файл при этом находится в `main/resources/config`)
3. Текущая папка при запуске приложения
4. Подпапка `/config` текущей папки

## Работа с БД в приложении

Путь к application.properties также можно указать при запуске приложения:

```
java -Dspring.config.location=my.properties -jar myproject.jar
```

В нашем примере application.properties находится в main/resources и будет загружаться из ClassPath.

# Работа с БД в приложении

В случае БД H2 файл `application.properties` содержит

`spring.datasource.url=jdbc:h2:~/books_db`

В случае клиент-серверных БД также обычно требуется указать свойства:

- `spring.datasource.username`
- `spring.datasource.password`



## Работа с БД в приложении

Также в файле `application.properties` можно задать свойство `server.port` – номер порта веб-сервера.

Spring Boot также поддерживает формат YAML для настроек приложения: файл `application.yml`:

```
spring:
```

```
  datasource:
```

```
    url: jdbc:h2:~/books_db
```

```
server:
```

```
  port: 9000
```

## Работа с БД в приложении

Можно добавлять в файл `apploader.properties` собственные свойства и использовать их в приложении:

`application.properties`:

`my.property=Some value`

В коде:

```
@Value("${my.property}")
```

```
private String myProperty;
```

# Работа с БД в приложении

Spring JDBC позволяет немного сократить код JDBC.  
Для этого используется класс JdbcTemplate:

```
public class BookRepoSpringJdbc {  
  
    private final JdbcTemplate jdbc;  
  
    public BookRepoSpringJdbc(DataSource ds) {  
        this.jdbc = new JdbcTemplate(ds);  
    }  
  
    private static RowMapper<Book> getBookRowMapper() {  
        return (rs, rowNum) -> new Book(rs.getInt(1), rs.getString(2), rs.getString(3));  
    }  
  
    public List<Book> getAllBooks() {  
        return jdbc.query("select id, author, title from books", getBookRowMapper());  
    }  
}
```

[Исходный код](#)

# Работа с БД в приложении

```
public Optional<Book> getBook(int id) {  
    List<Book> found = jdbc.query("select id, author, title from books where id = ?", getBookRowMapper(), id);  
    Book book = DataAccessUtils.singleResult(found);  
    return Optional.ofNullable(book);  
}
```

```
public Book addBook(BookDetails details) {  
    KeyHolder keyHolder = new GeneratedKeyHolder();  
    jdbc.update(  
        con -> con.prepareStatement("insert into books (author, title) values (?, ?)", new String[] {"id"}),  
        keyHolder  
    );  
    Integer id = keyHolder.getKeyAs(Integer.class);  
    assert id != null;  
    return new Book(id.intValue(), details.getAuthor(), details.getTitle());  
}
```

# Работа с БД в приложении

Альтернативный подход – JPA:

```
@Entity @Table(name = "books")
```

```
public class BookEntity {
```

```
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private int id;
```

```
    private String author;
```

```
    private String title;
```

```
    // getters/setters
```

```
}
```

# Работа с БД в приложении

```
public interface BookRepository extends CrudRepository<Book, Integer> {  
}
```

Автоматически определяет методы для поиска всех/по ID, сохранения сущности, удаления всех/по ID.

# Работа с БД в приложении

Почему JPA это зло:

- Работает только с изменяемыми классами
- При сложных связях между сущностями становится трудно контролировать порядок загрузки (N+1 problem: <https://habr.com/ru/company/otus/blog/529692/>)
- Не всегда генерируется эффективный SQL-запрос
- Как правило, сущности нельзя отдавать в слой представления напрямую – нужно преобразовывать их в DTO
- Дает небольшое упрощение в самой простой части приложения

# Принципы SOLID

- S – Single Responsibility Principle
- O – Open/Closed Principle
- L – Liskov Substitution Principle
- I – Interface Segregation Principle
- D – Dependency Inversion Principle



# Принципы SOLID

Single Responsibility Principle

Принцип единственной ответственности

# Принципы SOLID

## Open/Closed Principle

### Принцип открытости/закрытости

Принцип открытости/закрытости означает, что классы/интерфейсы должны быть:

- открыты для расширения: означает, что поведение класса может быть расширено путём наследования.
- закрыты для изменения: в результате расширения класса не должны вноситься изменения в код, который использует базовый класс.

# Принципы SOLID

## Liskov Substitution Principle

### Принцип подстановки Барбары Лисков

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

```
void checkArea(Square s) {  
    assert(s.getArea() == s.getSide() * s.getSide());  
}
```

# Принципы SOLID

Interface Segregation Principle

Принцип разделения интерфейса

Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические

# Принципы SOLID

Dependency Inversion Principle

Принцип инверсии зависимостей

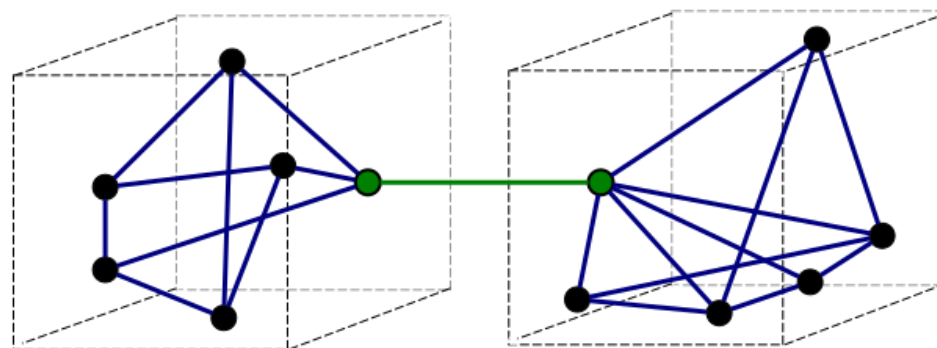
Dependency Injection + использование интерфейсов  
для связей между уровнями

# Cohesion & coupling

Качества хорошего кода: Loose Coupling, High Cohesion

Coupling (Зацепление)

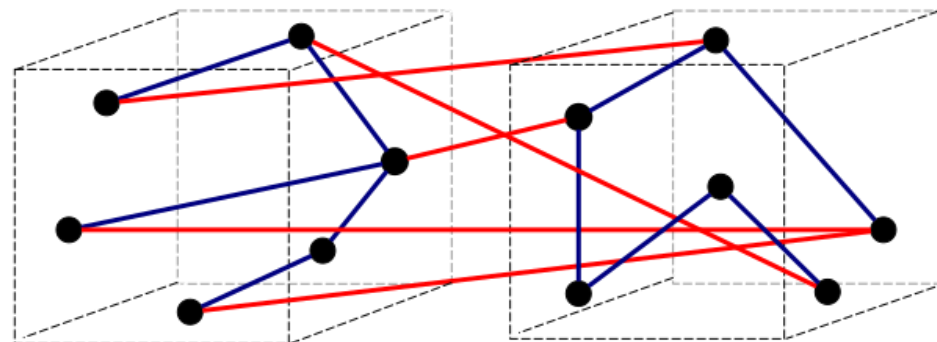
степень взаимозависимости  
между программными  
модулями



a) Good (loose coupling, high cohesion)

Cohesion (Связность):

мера силы взаимосвязанности  
элементов внутри модуля



b) Bad (high coupling, low cohesion)

- <https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html>