

# Construindo Aplicações com Banco de Dados: Fundamentos em Java e Python

UFCG/CEEI/UASC  
Banco de Dados I  
André Luiz F. Alves

# Objetivos

## Geral

- Desenvolver aplicações Java e Python que utilizam bancos de dados para armazenar e recuperar dados.

## Específicos

- Estabelecer conexões com bancos de dados usando uma linguagem de programação;
- Executar consultas SQL para recuperar, inserir, atualizar e excluir dados em bancos de dados relacionais através de uma aplicação.
- Compreender os conceitos de ORM (Mapeamento-Objeto-Relacional)

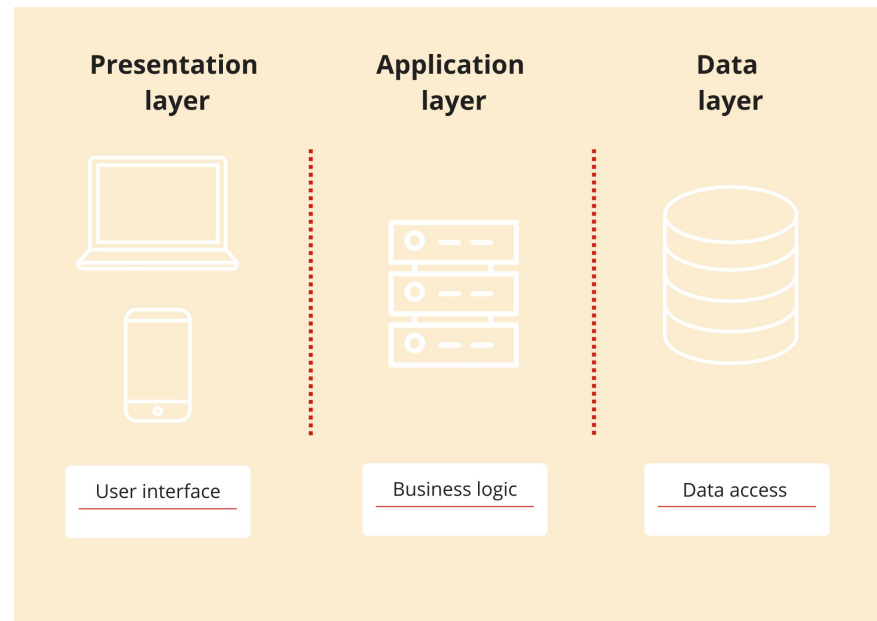
# Java: Persistência de objetos com JDBC e JPA



UFCG/CEEI/UASC  
Banco de Dados I  
André Luiz F. Alves  
Brunna Amorim  
Ana Gabrielle

# Introdução

- Aplicações são feitas de regras de negócio, interações com outros sistemas, interface... e persistência;
- A maior parte dos dados que são manipulados pelas aplicações tem que ser armazenados em banco de dados;



# Introdução

- Bancos de dados são importantes: armazenam os dados de negócio, atuam como ponto central entre as aplicações, além de oferecer integridade dos dados e controle de concorrência;
- Geralmente são utilizados banco de dados Relacionais, que armazenamos dados em tabelas de registros e colunas:
  - Simplicidade
  - Flexível
  - Robusto
  - Eficiente
  - Confiável

# Introdução

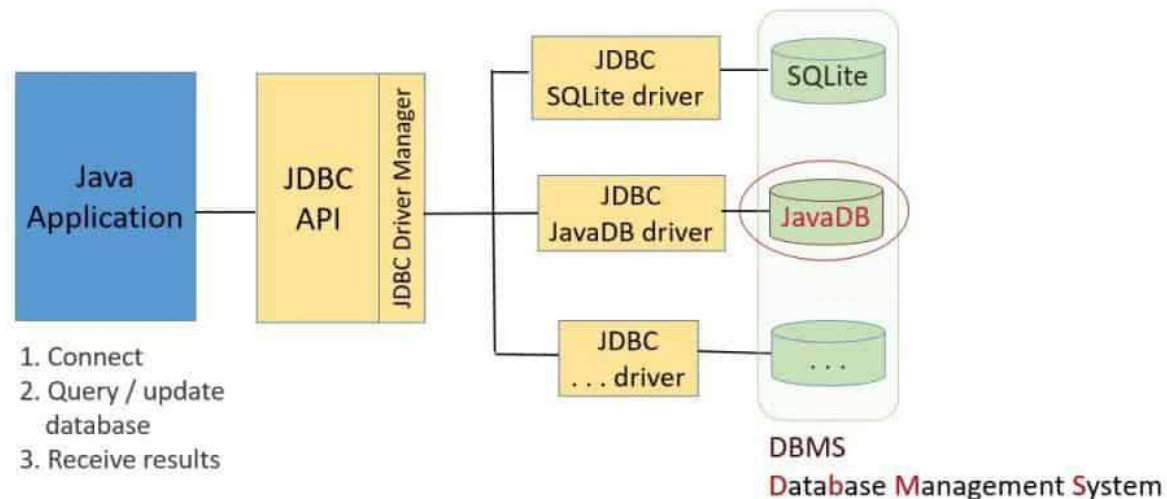
- Todo este vocabulário não existe no mundo orientado a objetos, como Java;
- Em Java, objetos é que são manipulados, objetos estes que representam instâncias de classes;
- Objetos herdam de outros, tem referência para coleções de outros objetos, e, às vezes, referenciam a si mesmo de forma recursiva;
- Temos classes concretas, abstratas, interfaces, enumerações, anotações, métodos, atributos, e assim por diante...

# Introdução

- Os objetos encapsulam estado e comportamento de uma forma elegante, mas só está acessível enquanto a JVM está rodando;
- Os objetos precisam ser persistentes. Um objeto pode armazenar seu estado para que depois possa ser recuperado é dito ser persistente;
- Há diferentes formas de persistir estado em Java:
  - Serialização (`java.io.Serializable`);
  - JDBC;
  - ORM.

# JDBC (Java Database Connectivity)

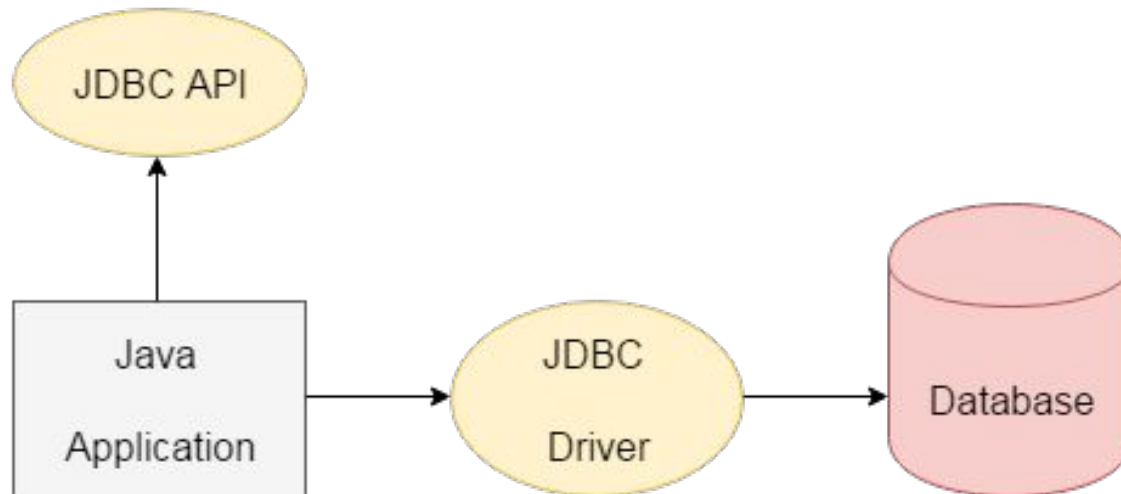
- API para acessar quaisquer dados em formato tabular que estejam armazenados em tabelas ou arquivos;
- **Interoperabilidade:** o mesmo programa Java acessa qualquer banco de dados.





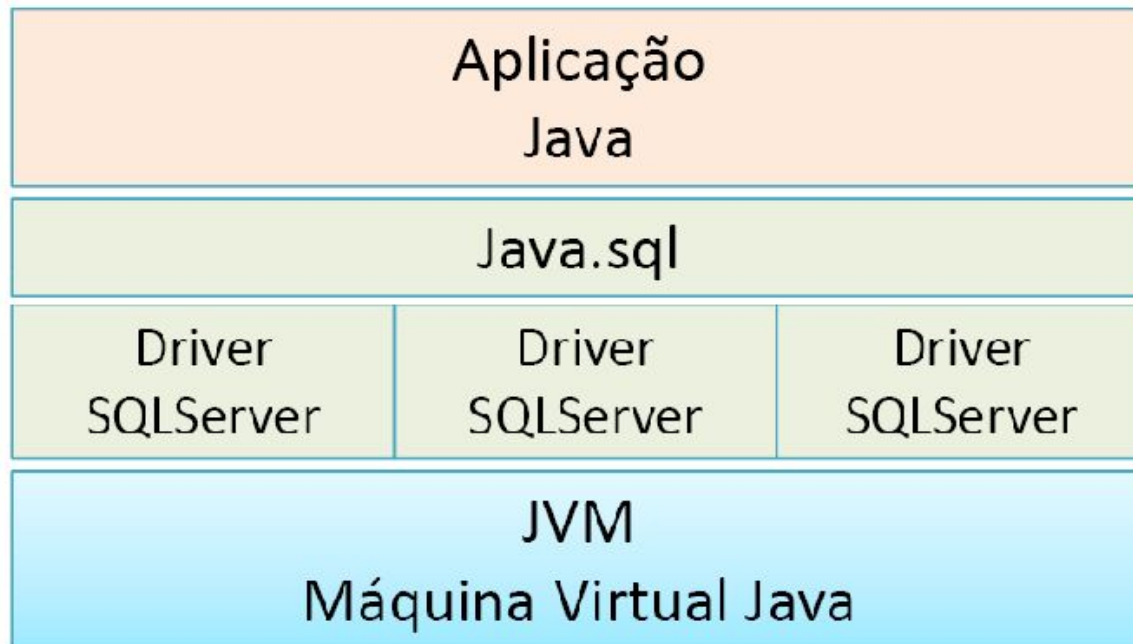
# JDBC - Tarefas

- Estabelece conexão com um BD;
- Executa transações e comando SQL (DDL e DML);
- Recebe um conjunto de resultados;



# JDBC - Aplicações

- Baseia em dois elementos:
  - O pacote `java.sql` que contém as classes que permitem o acesso ao banco de dados pela aplicação.
  - O driver para acesso ao banco de dados a ser utilizado.



# JDBC - Tarefas

## Java Database Connectivity



# JDBC -

## ⌘ Carregar Driver

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

## ⌘ Estabelecer Connection

```
Connection connection = DriverManager.getConnection(  
    "jdbc:odbc:BancodeJava" /*JDBC URL*/ , "jorge", "jhcf");
```

## ⌘ Criar Statement

```
Statement statement = connection.createStatement();
```

## ⌘ Executar Queries

```
ResultSet result = statement.executeQuery(  
    "SELECT * FROM contas WHERE numero = \'1\'");
```

## ⌘ Executar Updates

```
statement.executeUpdate(  
    "UPDATE contas SET saldo = 100 WHERE numero = \'1\'");
```

## ⌘ Fechar Statement e Connection

```
statement.close(); connection.close();
```

# JDBC - Tarefas: Carregar Driver

- Informar a aplicação Java qual o banco de dados a ser utilizado;
- Cria uma instância e registra-se junto ao DriverManager (importar a biblioteca do driver)
- **Exemplos:**
  - **Driver jdbc MySql:**

```
Class.forName("com.mysql.jdbc.Driver")
```

Ou

```
Class.forName("com.mysql.jdbc.Driver").newInstance()
```

# JDBC - Tarefas: Estabelecer Conexão

- java.sql.Connection
- java.sql.DriverManager

```
public void conexaoBD() {  
    try {  
        Connection conn = DriverManager.getConnection("urlBD", "usuario", "senha");  
        setConnection(conn);  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

## MySql:

```
String url = "jdbc:mysql://localhost/meuBanco";  
String user = "root";  
String pass = "123456"  
Connection con= DriverManager.getConnection(url,user,pass);
```

# JDBC - Tarefas: Executar Comandos

- ⌘ Executa comandos estáticos SQL
- ⌘ A cada momento, apenas um ResultSet por Statement
- ⌘ Principais Métodos
  - ⏏ ResultSet executeQuery(String sql)
    - ⏏ SQL statement SELECT
  - ⏏ int executeUpdate(String sql)
    - ⏏ SQL statement INSERT, UPDATE ou DELETE
  - ⏏ close(); cancel();
  - ⏏ get(set)MaxRows(); get(set)MaxFieldSize();

# JDBC - Tarefas: Executar Comandos

```
Statement stmt = con.createStatement();
```

```
stmt.executeUpdate("CREATE TABLE  
Empregado(matricula int, nome varchar(20),  
endereco varchar(32), salario float)");
```

```
stmt.executeUpdate( "INSERT INTO Empregado  
values(1000, 'Biliu','Rua das Cruzetas, h,  
Brasil', 30000.00)" );
```

Os comandos SQL são passados ao objeto Statement como string;



# JDBC - Tarefas - Executar Comandos

- Executa transações e comando SQL (DDL e DML)
  - Statement (`java.sql.Statement`)
    - Uso genérico
    - Comandos SQL estáticos
    - Não aceita parâmetros
  - PreparedStatement (`java.sql.PreparedStatement`)
    - Comandos SQL dinâmicos
    - Aceita parâmetros em tempo de execução

# JDBC - Tarefas - Executar Comandos

## – Statement e execute()

```
public static void main(String args[]){
    conexaoBD();
    boolean success = executeSQLDDLCommand("CREATE TABLE empregado "
        + "(matricula int, nome varchar(20), endereco varchar(32), salario float);");
    if (success) System.out.println("Tabela Empregado criada com sucesso");
}

public static boolean executeSQLDDLCommand(String sql){
    if (getConnection() != null){
        Statement statement;
        try {
            statement = connection.createStatement();
            return statement.execute(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    return false;
}
```

# JDBC - Tarefas - Executar Comandos

- Executa comando SQL (DDL e DML)
  - Statement e executeUpdate()

```
private static void insereEmpregadosStatement() {
    String sqlEmpregado1 = "INSERT INTO empregado(matricula, nome, endereco, salario) "
        + "VALUES (1, 'Daniel', 'Rua Aprígio Veloso', 1500.0);";
    String sqlEmpregado2 = "INSERT INTO empregado(matricula, nome, endereco, salario) "
        + "VALUES (2, 'Priscila', 'Rua Elpidio de Almeida', 4500.0);";
    String sqlEmpregado3 = "INSERT INTO empregado(matricula, nome, endereco, salario) "
        + "VALUES (3, 'Debora', 'Rua Manoel Tavares', 3000.0);";

    executeUpdateStatement(sqlEmpregado1);
    executeUpdateStatement(sqlEmpregado2);
    executeUpdateStatement(sqlEmpregado3);
}

public static void executeUpdateStatement(String sql){
    if (getConnection() != null){
        Statement statement;
        try {
            statement = connection.createStatement();
            statement.executeUpdate(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

# JDBC - Tarefas - Executar Comandos

- PreparedStatement e executeUpdate()
  - Usado principalmente quando queremos passar parâmetros pelo SQL.
  - Os parâmetros aparecem no comando como '?' e são substituídos conforme sua posição no comando.
  - Resolve problemas de [SQL INJECTION](#).
  - Muitas vezes o comando já vai para o SGBD compilado o que reduz seu tempo de processamento;

# JDBC - Tarefas - Executar Comandos

- PreparedStatement e executeUpdate()

```
PreparedStatement updateSalario =  
con.prepareStatement("UPDATE Empregado set  
salario = ? where matricula = ?");
```

Precisamos então suprir os valores que estão com ?. Isto é feito usando métodos setXXX definidos na interface PreparedStatement

```
updateSalario.setFloat(1, 10000);  
updateSalario.setInt(2, 1000);  
updateSalario.executeUpdate();
```

Então o preparedStatement seria equivalente a:

```
stmt.executeUpdate("UPDATE Empregado set  
salario = 10000 where matricula = 1000");
```

# JDBC - Tarefas - Executar Comandos

- PreparedStatement e executeUpdate()

```
PreparedStatement updateSalarios;  
  
String updateString = ("UPDATE Empregado set salario = ?  
where matricula = ?");  
  
updateSalarios = con.prepareStatement(updateString);  
  
int [] novosSalarios = {2000, 1500, 3000, 500};  
int [ ] matriculas = {1000, 1050, 2000, 1078};  
for( i = 0 ; i < matriculas.length; i++) {  
    updateSalarios.setFloat(1, novosSalarios[i]) ;  
    updateSalarios.setInt(2, matriculas[i]);  
    updateSalarios.executeUpdate();  
}
```

# JDBC - Tarefas - Executar Comandos

- PreparedStatement e executeUpdate()

```
private static void insereEmpregadosPreparedStatement() {
    executeUpdatePreparedStatement(4, "Tiago", "Rua Joaquim Caroca", 7700.0);
    executeUpdatePreparedStatement(5, "Mariana", "Rua Rodrigues Alves", 1700.0);
    executeUpdatePreparedStatement(6, "Júlio", "Rua José de Alenca", 2700.0);
}

public static void executeUpdatePreparedStatement(int matricula, String nome, String endereco, double salario){
    if (getConnection() != null){
        PreparedStatement preparedStatement;
        try {
            preparedStatement = connection.prepareStatement("INSERT INTO "
                + "empregado(matricula, nome, endereco, salario) VALUES (?, ?, ?, ?);");
            preparedStatement.setInt(1, matricula);
            preparedStatement.setString(2, nome);
            preparedStatement.setString(3, endereco);
            preparedStatement.setFloat(4, (float) salario);
            preparedStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

# JDBC - Tarefas - Executar Comandos

## – executeQuery()

JDBC retorna resultados em um objeto da classe `ResultSet`.

```
Statement stmt = con.createStatement();
```

```
ResultSet rs =  
stmt.executeQuery("Select * from  
empregado");
```



# JDBC - Tarefas - Executar Comandos

## ResultSet

- ⌘ Tabela de dados gerada pela execução de um comando `executeQuery()`
- ⌘ Linhas são acessadas seqüencialmente - `next()`;
- ⌘ Colunas são acessadas em qualquer ordem
  - ☐ métodos `getXXX(int columnIndex|String collName)`
    - ☐ \* `getBoolean(), _Byte(), _Int(), _Float(), _Double(), Long(), Short(), String()`
    - ☐ `byte[] getBytes()`
    - ☐ `java.lang.BigDecimal getBigDecimal();`
    - ☐ `InputStream getBinaryStream()`
    - ☐ `getMetaData();`

# JDBC - Tarefas - Executar Comandos


- ResultSet representa um cursor para os resultados obtidos da consulta.
  - usamos o método `next()` da classe `ResultSet` para avançarmos o cursor para a próxima tupla.
    - Inicialmente o cursor é posicionado para uma linha acima da primeira tupla do resultado.
    - Então para varremos todo o resultado precisamos de um laço do tipo:

```
while (rs.next()) {  
    ...  
}
```

# JDBC - Tarefas - Executar Comandos

- Para cada tipo de dado em SQL usamos o acessor equivalente em JDBC.
  - Por exemplo: varchar é lido com getString(), int com getInt(), double com getDouble(), etc..
  - Então para varremos todo o resultado precisamos de um laço do tipo:

```
while (rs.next()) {  
    int mat = rs.getInt("Matricula");  
    String nome = rs.getString("Nome");  
    String ender = rs.getString("Endereco");  
    float salario = rs.getFloat("Salario");  
}
```



nome da coluna ou um inteiro  
que representa a sequência dos  
campos no comando select

# JDBC - Tarefas

- Executa comando SQL (DDL e DML)
  - Statement e executeQuery()

```
executeQueryStatement("SELECT * FROM empregado;");
```

```
public static void executeQueryStatement(String sql){  
    if (getConnection() != null){  
        Statement statement;  
        try {  
            statement = connection.createStatement();  
            ResultSet rs = statement.executeQuery(sql);  
            while(rs.next()){  
                System.out.println("Empregado " + rs.getString("nome") + " = R$" + rs.getFloat("salario"));  
            }  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# JDBC - Tarefas

- Executa comando SQL (DDL e DML)
  - PreparedStatement e executeQuery()

```
executeQueryPorMatriculaPreparedStatement(1);
```

```
public static void executeQueryPorMatriculaPreparedStatement(int matricula){  
    if (getConnection() != null){  
        PreparedStatement preparedStatement;  
        try {  
            preparedStatement = connection.prepareStatement("SELECT * FROM empregado WHERE matricula = ?");  
            preparedStatement.setInt(1, matricula);  
            ResultSet rs = preparedStatement.executeQuery();  
            while(rs.next()){  
                System.out.println("Empregado " + rs.getString("nome") + " = R$" + rs.getFloat("salario"));  
            }  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# JDBC –

## Transações

- Iniciara transação antes do primeiro comando SQL

`connection.setAutoCommit(false)`

- Caso ocorra erro, não comitar as alterações

`connection.rollback()`

- Em caso de sucesso, comitar as alterações

`connection.commit()`

# JDBC – Liberando

## Recursos

- Após o uso precisamos fechar a conexão e o statement/preparedStatement, usando método close();
  - Statement.close();
  - preparedStatement.close();
  - connection.close();

# **Exemplo Completo: JDBC**



# JDBC –

## Desvantagens

- “Hard-Code”;
- Manutenção prejudicada;
- Específico para um Banco de Dados.

# JPA - Java Persistence API



# Persistindo objetos com Java

- Deve-se poder armazenar o estado de um objeto, mesmo após o objeto ser destruído;
- Deve ser possível criar um novo objeto com o mesmo estado do anterior;
- Operações não devem ser limitadas a um único objeto
  - Associações devem ser salvas

# Persistindo objetos com Java

- A aplicação deve trabalhar diretamente com objetos, ao invés de linhas e colunas da base de dados;
- Conceitos da Orientação a Objetos não devem ser restringidos pela solução adotada;
- Lógica de negócio deve ser implementada na aplicação, utilizando-se Java, e não diretamente na base de dados;

# Diferenças dos paradigmas: Custo

- Necessário escrever muito código para (tentar) contornar o problema;
- Código se torna repetitivo e de difícil manutenção;
- Modelagem dos objetos fica prejudicada
- Outras camadas ficam fortemente acopladas à Camada de Persistência
- Produtividade pode ser fortemente afetada

# Estratégias de Persistência

- Stored Procedures:
  - Lógica de negócio sai da aplicação e vai para a base de dados;
  - Perde-se a portabilidade;
- JDBC e SQL
  - Necessário escrever bastante código de baixo nível;

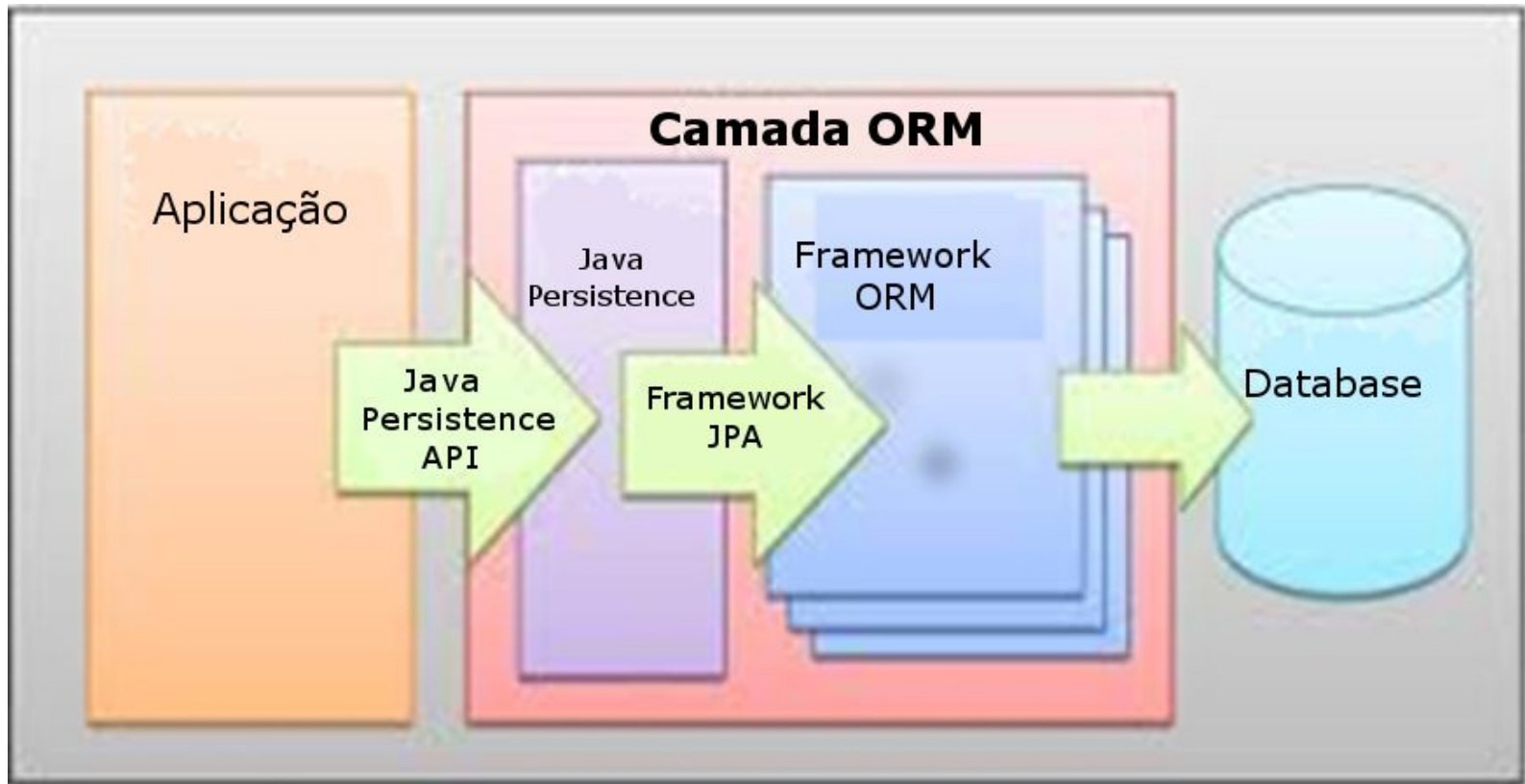
# Estratégias de Persistência

- JPA
  - **especificação** oficial que descreve como deve ser o comportamento dos frameworks de persistência
- Frameworks de terceiros
  - TopLink
    - Desenvolvido pela Oracle;
      - Gratuito para avaliação e na fase de desenvolvimento;
  - Apache OpenJPA
  - **Hibernate**



# Estratégias de Persistência

- JPA





# Hibernate

- Framework de mapeamento Objeto Relacional;
- Framework de persistência Java mais utilizado e documentado;
- Preenche a lacuna entre a base de dados relacional e a aplicação orientada a objetos;
- Suporta classes desenvolvidas com agregação, herança, polimorfismo, composição e coleções.

# Mapeamento Objeto Relacional (ORM)

- Permite a persistência de objetos em tabelas de uma base de dados relacional;
- Automático e Transparente;
- Utiliza metadados para descrever o relacionamento entre os objetos e a base de dados
  - XML
  - Annotations

# Mapeamento Objeto Relacional (ORM)

<b>Banco de dados</b>	<b>Linguagem Orientada a Objetos</b>
Tabela	Classe
Coluna	Atributo
Registro	Objeto

# Vantagens

- Produtividade
  - Elimina a necessidade de escrita de grande parte do código relativo à persistência;
  - Abstração do dialeto SQL.
- Manutenibilidade
  - Menos código e Maior entendimento da aplicação;
  - Camada de abstração.
- Portabilidade
  - Suporte a diversos tipos de banco de dados.

# Mapeamento - Entidade

- Para ser entidade, a classe deve obedecer as seguintes regras:
  - A classe deve ser anotada com `@javax.persistence.Entity` (ou definida no descritor XML como entidade);
  - A anotação `@javax.persistence.Id` deve ser usada para denotar uma chave primária;
  - A classe deve ter, pelo menos, um construtor vazio, que deve ter visibilidade pública ou *protected*
  - A classe deve ser de primeiro nível. Uma enumeração ou interface não pode ser designada como uma entidade;
  - A classe não deve ser final e nenhum método ou variável de instância da classe deve ser final.

# Mapeamento - @Entity

- Criando uma entidade

```
@Entity
@Table(name = "TB_USER")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(name = "ID")
    private Integer id;

    @Column(name = "FIRST_NAME")
    private String firstName;

    @Column(name = "LAST_NAME")
    private String lastName;

    @Column(name = "EMAIL")
    private String email;

    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTHDAY")
    private Date birthday;
```

# @Entity

## Annotations da classe

```
@Entity  
@Table(name = "TB_USER")  
public class User {
```

- Entity
  - Informa que a classe é uma entidade e será persistida pelo Hibernate;
  - Este nome será usado para referenciar a entidade nas consultas.

# @Table

## Annotations da classe

```
@Entity  
@Table(name = "TB_USER")  
public class User {
```

- Table
  - Informa o nome da tabela em que os objetos da classe devem ser armazenados;
  - Caso seja omitida, o nome da tabela é igual ao nome da entidade;
  - Name, catalog, schema, uniqueConstraints



# Mapeamento – @Entity - @Id

## Propriedades

```
@Id  
@GeneratedValue(strategy = GenerationType.SEQUENCE)  
@Column(name = "ID")  
private Integer id;
```

- Representa o identificador do objeto;
- Preenchido automaticamente quando o objeto é salvo;
- Mapeado para a chave primária da tabela
- Se duas instâncias tiverem o mesmo identificador, elas representam a mesma linha da tabela

# @Id -

## Estratégias

- Hibernate suporta diversas estratégias para geração automática de chaves
  - IDENTITY
  - *SEQUENCE*
  - INCREMENT
  - AUTO
- A estratégia deve ser especificada através da Annotation *@GeneratedValue*

# @Column

```
@Column(name = "FIRST_NAME")  
private String firstName;
```

- Atributos
  - name
    - Nome da coluna a qual a propriedade é mapeada
  - unique
    - Se o valor deve ser único ou não
  - nullable
    - Se a propriedade pode ser nula ou não

# Mapeando Propriedades: Tipos

Java type	Standard SQL built-in type
<code>int</code> or <code>java.lang.Integer</code>	INTEGER
<code>long</code> or <code>java.lang.Long</code>	BIGINT
<code>short</code> or <code>java.lang.Short</code>	SMALLINT
<code>float</code> or <code>java.lang.Float</code>	FLOAT
<code>double</code> or <code>java.lang.Double</code>	DOUBLE
<code>java.math.BigDecimal</code>	NUMERIC
<code>java.lang.String</code>	CHAR(1)
<code>java.lang.String</code>	VARCHAR
<code>byte</code> or <code>java.lang.Byte</code>	TINYINT
<code>boolean</code> or <code>java.lang.Boolean</code>	BIT

# Mapeando Propriedades: Tipos

Java type	Standard SQL built-in type
<code>java.util.Date</code> or <code>java.sql.Date</code>	DATE
<code>java.util.Date</code> or <code>java.sql.Time</code>	TIME
<code>java.util.Date</code> or <code>java.sql.Timestamp</code>	TIMESTAMP
<code>java.util.Calendar</code>	TIMESTAMP
<code>java.util.Calendar</code>	DATE

# Mapeando propriedades

- Ao mapear uma classe através da Annotation *@Entity*, todas as propriedades serão consideradas persistentes
- Propriedades não persistentes devem receber a Annotation *@Transient* ou o modificador *transient*
- *Por default, o nome da coluna será o mesmo da propriedade*

# Mapeando coleções

- Objetos podem possuir coleções
- As coleções podem ter duas naturezas
  - Coleções de valores
    - Tipos primitivos
    - Componentes
  - Coleções de entidades

# Mapeando coleções

- As associações podem ter duas características
  - **Direção:** Se uma classe pode navegar para a outra e vice-versa (Bidirecional) ou não (Unidirecional).
  - **Cardinalidade:** Quantos objetos estão envolvidos na associação: um-para-um, um-para-muitos, muitos-para-um, muitos-para-muitos.
    - As anotações utilizadas para representar estas associações são: **@OneToOne**, **@OneToMany**, **@ManyToOne** e **@ManyToMany**



# @JoinTable

- No banco de dados, associações de muito-para-muitos são representadas através de tabelas auxiliares. A anotação **@JoinTable** pode ser utilizada para especificar detalhes da tabela que representa essa associação.
- Atributos
  - **name**: nome da tabela que representa a associação;
  - **@joinColumns**: as colunas que devem ser usadas para fazer join com a tabela primária da entidade.

# Coleção de tipos primitivos: Set

- Um Item possui um conjunto de imagens
- Somente o nome da imagem é persistido
- Nomes não devem ser duplicados para um determinado Item

ITEM	
ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE	
ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

# Classe Item

```
@Entity
public class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name;

    @CollectionOfElements
    @JoinTable(name = "ITEM_IMAGE", joinColumns = @JoinColumn(name =
"ITEM_ID"))
    @Column(name = "FILENAME", nullable = false)
    private Set<String> images = new HashSet<String>();
}
```

# Implementando Associações

- Coleções devem ser acessadas através de suas Interfaces
  - Set, ao invés de HashSet
  - List, ao invés de ArrayList

# Entity Manager

- Serviço central para todas ações de persistência;
- Provê API para criar consultas, buscar objetos, sincronizar objetos e inserir objetos no banco de dados;
- Objetos Session e Transaction

# Consultas

- Objeto Query responsável por criar e executar as consultas;
- Uma das partes mais interessantes do acesso a dados
- Consultas são escritas utilizando conceitos de orientação a objetos
  - Objetos no lugar de tabelas
  - Propriedades no lugar de colunas
- Experiência em SQL não é desprezada

# Consultas

- Podem ser feitas de três maneiras
  - Hibernate Query Language (HQL)
  - Criteria API e Query by Example
  - Utilizando SQL

# Consultas: Exemplos HQL

```
public List<User> getAll() {  
    EntityManager em = getEntityManager();  
    List<User> resultado = null;  
    try {  
        TypedQuery<User> query = em.createQuery("SELECT u FROM User u", User.class);  
        resultado = query.getResultList();  
    } catch (PersistenceException pe) {  
        pe.printStackTrace();  
    }  
    return resultado;  
}
```



# Consultas

- Envolve alguns passos
  - Criar a consulta com as restrições necessárias
  - Adicionar parâmetros à consulta
  - Executar a consulta e recuperar o resultado
    - A forma de execução da consulta e obtenção dos dados pode ser configurada

# Adicionando parâmetros à consulta

- Parâmetros não devem ser adicionados na própria String da consulta
  - "from User u where u.firstName like '" + fn + "'"
- Evita-se ataque do tipo SQL Injection
  - foo' and callSomeStoredProcedure() and 'bar' = 'bar
- Parâmetros podem ser adicionados através de sua posição ou de seu nome

# Adicionando parâmetros pelo nome

- Nome do parâmetro é precedido de “:”
- Os valores são adicionados através de métodos sets

```
String queryString = "from User u where  
    u.firstName like :fn";
```

```
Query q = em.createQuery(queryString).setString(  
    "fn", user.getFirstName());
```

# Adicionando parâmetros pela posição

- A consulta contém “?” para indicar a existência de alguma parâmetro
- Os valores também são adicionado através de métodos sets

```
String queryString =  
"from User u where u.firstName like ? and u.email = ?";
```

```
Query q = em.createQuery(queryString)  
.setString(1,user.getFirstName())  
.setString(2,user.getEmail());
```

# Executando a consulta

- Se mais de um objeto pode ser retornado, chama-se o método *getResultList()*
  - *List list = query.getResultList();*
- Se somente um objeto pode ser retornado, chama-se o método *getSingleResult()*
  - *User user = (User) query.getSingleResult();*
  - O método retorna *null* se nenhum objeto for encontrado
  - Se a consulta retornar mais de um objetos, a

# Junção com HQL

- Coluna de junção não precisar ser informada na consulta
  - Informação é extraída do mapeamento
- É necessário ser informado somente o nome da associação
  - Nome do atributo que referencia a classe ou coleção de classes

# **Exemplo Completo: JPA+Hibernate**

# JDBC x Hibernate

JDBC	Hibernate
Necessidade de mapear código para representação de dados	Mapeamento de dados automático e transparente
Suporte apenas de SQL	Suporte de HQL (independente de BD) e SQL
Conhecimento SQL obrigatório	Conhecimento SQL desejável
Melhor Performance	Mais lento, porém há cache na aplicação



# Python: Conexão com Banco de Dados



UFCG/CEEI/UASC  
Banco de Dados I  
André Luiz F. Alves

# Introdução

## Abordagens para Conexão com Banco de Dados em Python:

1. Usando Drivers Específicos do Banco de Dados
2. Usando Frameworks de Mapeamento Objeto-Relacional (ORM)

# Introdução

## 1- Usando Drivers Específicos do Banco de Dados:

- Instalação do driver oficial para o banco de dados específico ao qual você deseja se conectar:
  - `mysql-connector-python` para MySQL, `psycopg2` para PostgreSQL ou `cx_Oracle` para Oracle...
    - `pip install....`
- API Python personalizada para interagir com as funcionalidades correspondentes do banco de dados.

# Introdução

## 1- Usando Drivers Específicos do Banco de Dados:

- **Vantagens:**

- **Integração Nativa:** comunicação direta e otimizada com o banco de dados, garantindo recuperação e manipulação de dados eficientes.
- **Recursos Específicos do Banco de Dados:** acesso a recursos específicos do banco de dados, como *procedures*, *function*, *triggers* ou tipos de dados avançados.

- **Desvantagens:**

- **Dependência do Driver:** Cada banco de dados requer seu próprio driver, aumentando o número de bibliotecas a serem gerenciadas.
- **Portabilidade Limitada:** O código que usa drivers específicos do banco de dados pode não ser facilmente transferível para outros bancos de dados.

# Introdução

## 2- Usando Frameworks de Mapeamento Objeto-Relacional (ORM):

- abstração de nível superior para interações com bancos de dados, focando em conceitos orientados a objetos em vez de consultas SQL de baixo nível.
- Frameworks ORM populares para Python:
  - **SQLAlchemy**, Django ORM e Peewee.



# Introdução

## 2- Usando Frameworks de Mapeamento Objeto-Relacional (ORM):

- **Vantagens:**

- **Abordagem Orientada a Objetos:** mapeamento das tabelas de banco de dados para objetos Python
- **Portabilidade:** O código ORM geralmente é mais portátil em diferentes bancos de dados, pois o framework lida com as interações subjacentes do banco de dados.
- **Redução de Código Boilerplate:** redução da quantidade de código SQL repetitivo necessário para operações comuns de banco de dados.

- **Desvantagens:**

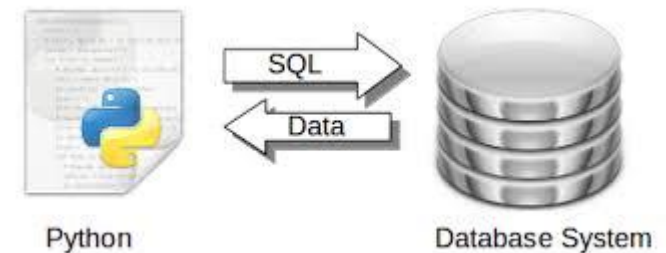
- **Sobrecarga de Desempenho:** introdução de sobrecarga de desempenho em comparação com o uso direto do driver devido à camada de abstração.
- **Curva de Aprendizagem:** aprendizado adicional para compreender e usar efetivamente um framework ORM.

# Introdução

## Qual Abordagem escolher?

- Para projetos simples ou que exigem controle direto do banco de dados, os drivers específicos do banco de dados oferecem uma abordagem direta e eficiente.
- Para projetos maiores com modelos de dados complexos ou que enfatizam portabilidade e desenvolvimento orientado a objetos, os ORMs fornecem uma solução mais estruturada e fácil de manter.
- Considere fatores como requisitos de desempenho, experiência do desenvolvedor e objetivos do projeto ao tomar sua decisão.

# Usando Drivers Específicos do Banco de Dados





# Usando Drivers Específicos do Banco de Dados

## 1. Instalar e importar o Driver do Banco de Dados

- Para se conectar a um banco de dados específico, você precisará instalar o driver apropriado para Python:
  - `pip install mysql-connector-python`
  - `pip install psycopg2`
  - `pip install cx_Oracle`
  - ..
- Importar a biblioteca do Driver
  - `import mysql.connector`
  - `import psycopg2`
  - `import cx_Oracle`
  - ...

# Usando Drivers Específicos do Banco de Dados

## 2. Estabelecer a Conexão

- Crie um objeto de conexão usando a função `connect()`, fornecendo suas credenciais e configuração do banco de dados
  - - `db = DRIVER.connect( host="localhost",  
user="seu_nome_de_usuario",  
password="sua_senha",  
database="nome_do_seu_banco_de_dados" )`
- onde `DRIVER = { mysql.connector, psycopg2, cx_Oracle, ... }`

# Usando Drivers Específicos do Banco de Dados

## 3. Criar um Objeto de Cursor

- Um objeto de cursor é usado para executar instruções SQL no banco de dados:

```
cursor = db.cursor()
```

# Usando Drivers Específicos do Banco de Dados

## 4. Executar Instruções SQL:

- Use o objeto de cursor para executar consultas SQL, como SELECT, INSERT, UPDATE ou DELETE:

```
# Exemplo: Selecionar todos os registros da tabela 'usuarios'
cursor.execute("SELECT * FROM usuarios")
# Obter os resultados
resultados = cursor.fetchall()
# Imprimir os resultados
for linha in resultados:
    print(linha)
```

# Usando Drivers Específicos do Banco de Dados

## 5. Confirmar Alterações

- Se você fez alguma alteração no banco de dados (INSERT, UPDATE, DELETE), confirme as alterações

```
db.commit()
```

## 6. Fechar a Conexão

- Sempre feche a conexão com o banco de dados quando terminar para liberar recursos

```
db.close()
```

# Usando Drivers Específicos do Banco de Dados

## Dicas Adicionais:

- Use o tratamento de exceções para lidar com gracejo os erros que podem ocorrer durante as operações do banco de dados.
- Parametrize suas consultas SQL para evitar vulnerabilidades de injeção de SQL.
- Use instruções preparadas para melhorar o desempenho e a segurança.
- Considere usar pool de conexões para aplicativos de alto tráfego.

# Usando Frameworks de Mapeamento Objeto-Relacional (ORM)

SQLAlchemy

# Usando Framework ORM

## ● SQLAlchemy



- **Compatibilidade com Diversos Bancos de Dados:** biblioteca poderosa e versátil para interagir com SGBDs (PostgreSQL, MySQL, Oracle, Microsoft SQL Server, ...);
- **ORM:** camada de abstração que simplifica o processo de conexão, consulta e manipulação de dados, bem como definição e manipulação de relacionamentos entre tabelas de banco de dados ;
- **Expressividade e Flexibilidade:** Oferece uma API expressiva e flexível que permite escrever consultas complexas de forma intuitiva,
- **Suporte a Transações e Controle de Transações:** Garantir a integridade dos dados em operações complexas.
- **Ferramentas de Mapeamento e Engenharia Reversa:** Possui ferramentas de engenharia reversa que permitem gerar código Python a partir de esquemas de banco de dados existentes;
- **Suporte a Escalabilidade e Desempenho:** Projetado para oferecer desempenho e escalabilidade eficientes, mesmo em ambientes de alto tráfego e grandes volumes de dados.



# Usando Framework ORM: SQLAlchemy



## 1. Instalação do SQLAlchemy

```
pip install sqlalchemy
```

## 2. Conectando ao Banco de Dados

```
from sqlalchemy import create_engine
# Criando a string de conexão Sqlite
engine = create_engine('sqlite:///meu_banco.db')
# meu_banco.db será criado se não existir e armazenará os dados da sua aplicação.
```

```
#mysql
connection_string =
'mysql+mysqlconnector://usuario:senha@host/nome_banco'
engine = create_engine(connection_string, echo=True)
# 'echo=True' é opcional e imprime as consultas SQL geradas
```

# Usando Framework ORM: SQLAlchemy

## 3. Mapeamento de Objetos e Tabelas

```
from sqlalchemy import Column, Integer, String, Text
# Definindo a classe Usuario
class Usuario(Base):
    __tablename__ = 'usuarios'
    id = Column(Integer, primary_key=True)
    nome = Column(String(255), nullable=False)
    email = Column(Text, unique=True)
```

# Usando Framework ORM: SQLAlchemy

## 4. Manipulando Objetos



```
from db import Usuario, engine, Session
# Criando uma sessão
session = Session(bind=engine)
# Criando um novo usuário
novo_usuario = Usuario(nome='João Silva',
email='joaosilva@example.com')
# Adicionando o usuário à sessão e salvando no BD
session.add(novo_usuario)
session.commit()
```

# Usando Framework ORM: SQLAlchemy

## 5. Consultando Dados



```
# Buscando todos os usuários
usuarios = session.query(Usuario).all()
# Imprimindo os dados dos usuários
for usuario in usuarios:
    print(f"ID: {usuario.id}, Nome: {usuario.nome},
          Email: {usuario.email}")
```

# Usando Framework ORM: SQLAlchemy

## 5. Consultando Dados com Filtros



```
# Buscando um usuário pelo ID 1 usuario =
session.query(Usuario).get(1)
# Verificando se o usuário foi encontrado
if usuario:
    print(f"Nome: {usuario.nome}, Email:
{usuario.email}")
else:
    print("Usuário não encontrado")
```

# Usando Framework ORM: SQLAlchemy

## 6. Atualizando Registros:



```
# Buscando o usuário com ID 2
usuario = session.query(Usuario).get(2)
# Atualizando o email do usuário
if usuario:
    usuario.email = "novoemail@example.com"
    session.commit()
    print("Email atualizado com sucesso!")
else:
    print("Usuário não encontrado")
```

# Usando Framework ORM: SQLAlchemy

## 6. Excluindo Registros



```
# Buscando o usuário com ID 3
usuario = session.query(Usuario).get(3)
# Excluindo o usuário do banco de dados
if usuario:
    session.delete(usuario)
    session.commit()
    print("Usuário excluído com sucesso!")
else:
    print("Usuário não encontrado")
```

**Exemplo Completo:**

**SQLAlchemy**



# Referências

- 1 <https://docs.oracle.com/javase/tutorial/jdbc/>
- 2 <https://www.oracle.com/java/technologies/persistence-jsp.html>
- 3 <http://hibernate.org/>
- 4 <https://docs.sqlalchemy.org/orm/>