Java Database Connectivity – JDBC

JDBC (Java EE Database Connectivity)

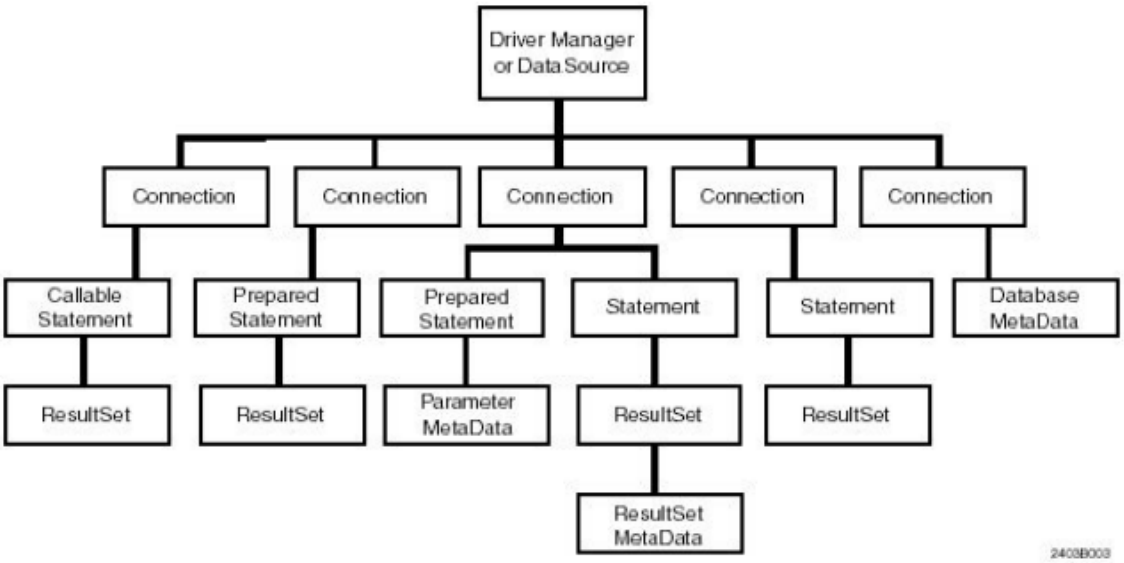
- É uma API que contém uma série de classes e interfaces para realizar a comunicação entre uma aplicação desenvolvida em Java e o banco de dados relacionais (SQL) utilizado por ela

- Foi baseado no ODBC (Open Database Connectivity) que é um padrão para acesso a sistemas gerenciadores de bancos de dados (SGBD)

- Foi criado para Java EE, mas atualmente faz parte da Java SE

- Pacote padrão: java.sql

Pacote java.sql



DriverManager

- É uma classe que atua como fábrica (factory) para criar conexões com SGDBs (sistemas gerenciadores de banco de dados)

- A conexão é realmente criada pelo driver do banco de dados, mas essa operação é acionada pelo Driver Manager, após ele encontrar dinamicamente o driver específico

- Possui outras operações para manipular drivers

String de Conexão

- Formato da String de conexão: protocol//[hosts][/database][?properties]

- O formato da string é definido pelo driver do SGDB (consulte a documentação dele)

- Ex: jdbc:mysql://mysql.db.server:3306/my_database?useSSL=false&serverTimezone=UTC

Driver JDBC

- O driver JDBC é uma biblioteca Java que implementa interfaces JDBC para realizar conexão e operações em um determinado SGDB

- Trata-se de um componente essencial para conectar a um banco de dados por meio da API JDBC

- Cada driver é específico para um SGDB (existem poucos casos nos quais um driver funciona para mais de um SGDB)

- Ex: postgresql-42.7.3.jar (PostgreSQL), mysql-connector-j-8.4.0.jar (MySQL), ojdbc11.jar (Oracle DB), db2jcc4.jar (IBM DB2) e h2-2.2.224.jar (H2 DB).

Connection

- É a interface que define como um objeto de conexão deve se comportar

- Normalmente a conexão é implementada no driver do banco de dados

- Contempla os principais elementos para preparar as operações que precisam ser realizadas: gestão da conexão, gestão de transações e a criação de objetos de manipulação do banco de dados (consulta, execução de procedimentos de banco, etc.)

- Principais operações:

1) getConnection: fornecer conexão com SGDB, seja ela uma nova ou existente

2) close: fechar conexão com SGDB

3) commit: confirmar operações realizadas na transação e fechar a transação

4) rollback: desfazer operações realizadas na transação e fechar a transação

5) createStatement: criação de objeto para a execução de código SQL pelo driver do SGDB

6) prepareCall: criação de objeto para a chamada remota de funções pelo driver do SGDB

7) prepareStatement: criação de objeto para a execução de procedimentos de banco de dados pelo driver do SGDB

Statement

- É uma interface que define como um objeto de execução de comandos SQL deve se comportar

- Os objetos do tipo Statement são de classes implementadas em driver do banco de dados

- Pode ser utilizado para operações que retornam tuplas de dados do SGDB ou que não retornam

- Principais operações:

1) execute: executa um comando SQL que pode retornar vários dados

2) executeQuery: executa um comando SQL que retorna um único objeto de dados

3) executeUpdate: executa um comando SQL que não retorna dados. Criado para ser utilizado com comandos SQL como insert, update, delete e DDLs.

OBS: não pode ser utilizado para a execução de funções e procedimentos do banco de dados

PreparedStatement

- É uma interface que especializa a execução de comandos SQL de forma repetida

- O objeto do tipo PreparedStatement compila previamente o comando SQL para ser executado várias vezes repetidamente

* Geralmente, a preparação do comando é realizada quando o objeto é instanciado

- Como a execução é preparada previamente, a execução do comando apresenta melhor desempenho do que os comandos executados por objetos Statement

* Em objetos Statement, a preparação do comando SQL é realizada junto com o comando de execução

ResultSet

- É uma interface que define o objeto que representa o resultado de um comando de banco de dados

- O resultado é estruturado no formato de uma tabela com linhas e colunas

* Observação: a tabela pode ter apenas uma linha e uma coluna

- O ResultSet tem um ponteiro para a linha dos dados e esse ponteiro pode ser alterado por meio de métodos do objeto

- Os dados do registro (linha da tabela) podem ser acessados por índice (começando por zero) ou pelo nome da coluna

- Principais operações:

1) getInt, getBoolean, getString, etc.: retorna o dado do tipo especificado do método por índice ou pelo nome da coluna

2) first: move o ponteiro para a primeira linha da tabela de resultado

3) previous: move o ponteiro para a linha anterior da tabela de resultado

4) next: move o ponteiro para a próxima linha da tabela de resultado

5) last: move o ponteiro para a última linha da tabela de resultado

Exemplo de código:

```
String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
try (Statement stmt = con.createStatement()) {
  ResultSet rs = stmt.executeQuery(query);
  while (rs.next()) {
    String coffeeName = rs.getString("COF_NAME");
    int supplierID = rs.getInt("SUP_ID");
    float price = rs.getFloat("PRICE");
    int sales = rs.getInt("SALES");
    int total = rs.getInt("TOTAL");
    System.out.println(coffeeName + ", " + supplierID + ", " + price +
                       ", " + sales + ", " + total);
  }
} catch (SQLException e) {
  JDBCTutorialUtilities.printSQLException(e);
}
}
```

SQLException

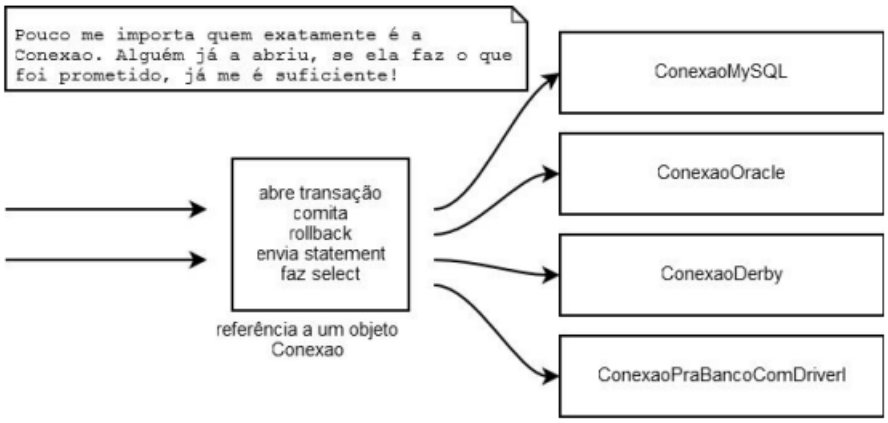
- É a exceção base da API JDBC para fornecer informações sobre erros na manipulação de banco de dados

- Herda da classe Exception

- Principais subclasses: BatchUpdateException, RowSetWarning, SerialException, SQLClientInfoException, SQLNonTransientException, SQLRecoverableException, SQLTransientException, SQLWarning, SyncFactoryException, SyncProviderException.

Abstração em Interfaces

- A API JDBC é um bom exemplo da utilização de interfaces para a definição do contrato de operações e de objetos e estes são implementados pelo driver do SGDB



Injeção de SQL

- O ataque de Injeção de SQL (SQL Injection) consiste na inserção (“injeção”) de comandos SQL na aplicação por meio da sua entrada de dados

- Esse tipo de ataque pode proporcionar desde vazamento de dados até a destruição do banco de dados

- O ataque pode ocorrer quando objetos de entrada são usados diretamente para gerar comandos SQL sem validação ou bloqueio da entrada de comandos SQL

- Ex: campo “nome” preenchido com SELECT * FROM usuários ao invés do nome solicitado na interface do usuário

Protegendo de Injeção de SQL

- Validar tipos na entrada de dados. Por exemplo, converter campo cuja informação deveria ser preenchida com número inteiro para “int” ou “Integer” antes de passar seu conteúdo para o comando SQL

* Essa estratégia não permite proteger a entrada de campos texto (String)

- Não exibir para o usuário final, códigos e mensagens de erros com informações que comprometam a segurança do banco de dados. Exemplo:

```
[ERROR] [MY-011263] [Server] Could not use
/var/log/mysql/mysql-slow.log for logging (error 28 - No
space left on device). Turning logging off for the server
process. To turn it on again: fix the cause, then either
restart the query logging by using "SET GLOBAL
SLOW QUERY LOG=ON" or restart the MySQL server.
```

- Utilizar parâmetros para a passagem de valores para os comandos SQL:

* Colocar o símbolo de interrogação para marcar os locais de inserção de valores;

SELECT * FROM tb_carro WHERE marca = ?

* Criar objeto do tipo PreparedStatement com o comando marcado

* Inserir os valores usando parâmetros posicionais (começando por um)

pstmt.setString(3, “BYD”);

JPA (Java Persistence API)

- É uma API de Java EE para gerenciar dados relacionais em aplicações Java

- Proporciona uma maneira padrão de mapeamento objeto-relacional (ORM) para Java

- Permite que desenvolvedores trabalhem com bancos de dados usando objetos Java sem a necessidade de escrever SQL manualmente

- Faz parte da especificação Java EE, mas pode ser usado em Java SE

Entidades

- São classes POJO (Plain Old Java Object) que representam tabelas em um banco de dados

- Uma classe é considerada uma entidade se for anotada com @Entity

- Cada instância de uma entidade corresponde a uma linha da tabela

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    // getters e setters
}
```

Mapeamento de Atributos

- Os campos de uma entidade são mapeados para colunas da tabela do banco de dados

- @Column: define mapeamentos adicionais para a coluna de banco de dados associada

- @Temporal: especifica o tipo de dado temporal (DATE, TIME, TIMESTAMP) para atributos de data

- @Lob: define que o atributo é um Large Object (LOB)

Chaves Primárias

- A chave primária de uma entidade é especificada com @Id

- @GeneratedValue: define a estratégia de geração da chave primária

* AUTO: o provedor JPA escolhe a estratégia apropriada

* IDENTITY: a chave é gerada pela coluna de identidade do banco de dados

* SEQUENCE: usa uma sequência de banco de dados

* TABLE: usa uma tabela especial para gerar valores de chave primária

Relacionamentos

- Relacionamentos entre entidades são especificados por meio de anotações

* @OneToOne: mapeia um relacionamento um-para-um

* @OneToMany: mapeia um relacionamento um-para-muitos

* @ManyToOne: mapeia um relacionamento muitos-para-um

* @ManyToMany: mapeia um relacionamento muitos-para-muitos

* @JoinColumn: especifica a coluna de junção para um relacionamento

* @JoinTable: define a tabela de junção para um relacionamento muitos-para-muitos

- Exemplo:

```java
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
    // getters e setters
}
```

Consultas JPQL

- JPQL (Java Persistence Query Language) é uma linguagem de consulta similar ao SQL mas opera sobre entidades em vez de tabelas

- Usada para criar consultas para buscar, inserir, atualizar ou deletar entidades

- Exemplo:

```
TypedQuery<User> query = entityManager.createQuery("SELECT u FROM User u WHERE u.email = :email", User.class);
query.setParameter("email", "example@example.com");
User user = query.getSingleResult();
```

EntityManager

- A interface EntityManager é usada para interagir com o contexto de persistência

- Principais operações:

1) persist: salva uma nova entidade

2) merge: atualiza uma entidade existente

3) remove: remove uma entidade

4) find: busca uma entidade pelo seu identificador

5) createQuery: cria uma consulta JPQL

Transações

- A gestão de transações é crucial para garantir a consistência dos dados

- Em um ambiente Java EE, transações são geralmente gerenciadas pelo contêiner EJB

- Em Java SE, podem ser gerenciadas manualmente usando EntityTransaction

- Exemplo:

```
EntityTransaction transaction = entityManager.getTransaction();
try {
    transaction.begin();
    entityManager.persist(newUser);
    transaction.commit();
} catch (Exception e) {
    transaction.rollback();
}
```

Cache

- JPA suporta dois níveis de cache:

* Primeiro nível (EntityManager cache): o escopo é a transação

* Segundo nível (cache compartilhado): o escopo é a aplicação, configurável via provider JPA

Injeção de Dependência

- A EntityManager pode ser injetada em um ambiente Java EE usando @PersistenceContext

- Exemplo:

```
@PersistenceContext
private EntityManager entityManager;
```

Persistência em Herança

- JPA suporta herança entre entidades.

- Estratégias de herança:

* Single Table (Tabela Única): todas as classes de uma hierarquia são mapeadas para uma única tabela

* Joined: cada classe de entidade é mapeada para sua própria tabela

* Table per Class (Tabela por Classe): cada classe concreta é mapeada para sua própria tabela

- Exemplo:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    // getters e setters
}

@Entity
public class Employee extends Person {
    private String department;
    // getters e setters
}
```

Diferença entre JDBC e JPA

- A diferença entre JDBC e JPA é fundamental, pois ambas são tecnologias usadas para acessar e manipular bancos de dados em Java, mas operam em níveis diferentes de abstrações e têm finalidades distintas.

Nível de Abstração:

JDBC:
- JDBC é uma API de nível mais baixo para a comunicação direta com bancos de dados relacionais
- Requer que os desenvolvedores escrevam SQL explícito para realizar operações de banco de dados
- Não fornece mecanismos para o mapeamento objeto-relacional, portanto, a transformação de objetos Java em tabelas de banco de dados deve ser feita manualmente

JPA:
- JPA é uma especificação que define uma API para o mapeamento objeto-relacional (ORM) em Java. Ele opera em um nível mais alto de abstração
- Permite aos desenvolvedores trabalhar com objetos Java para realizar operações de banco de dados, sem precisar escrever SQL explícito
- Fornece um mapeamento entre as classes Java e as tabelas do banco de dados, permitindo que as entidades Java sejam diretamente persistidas no banco de dados.

Simplificação do Código:

JDBC:
- JDBC requer mais código boilerplate para gerenciar conexões, criar e executar instruções SQL, manipular ResultSets, e lidar com exceções
- A complexidade aumenta conforme o número de operações de banco de dados cresce, pois cada operação deve ser codificada manualmente

JPA:
- JPA simplifica o código ao permitir que os desenvolvedores trabalhem com entidades Java, abstraindo muitos dos detalhes de implementação do SQL
- Inclui recursos como cache de primeiro e segundo nível, mapeamento de herança, e gerenciamento automático de transações

Gerenciamento de Transações:

JDBC:
- JDBC requer que os desenvolvedores gerenciem transações explicitamente usando métodos como ‘setAutoCommit(false)’, ‘commit()’, e ‘rollback()’
- A responsabilidade pelo controle de transações recai inteiramente sobre o desenvolvedor

JPA:
- JPA pode gerenciar transações automaticamente em ambientes Java EE, usando contêineres EJB ou CDI
- Fornece a API EntityTransaction para gerenciamento manual de transações em ambientes Java SE.

Consulta e Manipulação de Dados:

JDBC:
- As consultas e manipulações de dados são feitas diretamente com SQL
- O desenvolvedor deve criar manualmente instruções SQL para cada operação e lidar com os resultados usando a interface ResultSet

JPA:
- Usa JPQL (Java Persistence Query Language), uma linguagem de consulta orientada a objetos, para buscar e manipular dados
- Permite a criação de consultas tipadas e a execução de operações complexas sem a necessidade de SQL explícito

Desempenho:

JDBC:
- Potencialmente mais rápido e eficiente, pois permite o controle direto sobre a execução de SQL
- A ausência de camadas de abstração adicionais pode resultar em melhor desempenho em operações críticas e de baixa latência

JPA:
- Pode ser menos eficiente em certas situações devido à sobrecarga do mapeamento objeto-relacional e da abstração.
- Utiliza caching e outras otimizações para melhorar o desempenho em muitos casos, mas pode não ser tão eficiente quanto o SQL nativo em operações complexas

Flexibilidade:

JDBC:
- Menos portátil, pois as instruções SQL podem variar entre diferentes bancos de dados.
- Depende diretamente dos drivers JDBC específicos do banco de dados em uso

JPA:
- Mais portátil entre diferentes bancos de dados e fornecedores de JPA (como Hibernate, EclipseLink, OpenJPA), desde que não se usem extensões específicas de fornecedores
- A configuração de persistência e o mapeamento são independentes do banco de dados subjacente

Exemplo comparativo:

JDBC:

```
// JDBC Example: Fetching a user by ID
Connection conn = DriverManager.getConnection(dbUrl, dbUser, dbPassword);
String query = "SELECT * FROM users WHERE id = ?";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setInt(1, userId);
ResultSet rs = pstmt.executeQuery();
User user = null;
if (rs.next()) {
    user = new User();
    user.setId(rs.getInt("id"));
    user.setName(rs.getString("name"));
    // other fields
}
rs.close();
pstmt.close();
conn.close();
```

JPA:

```
// JPA Example: Fetching a user by ID
EntityManager em = emf.createEntityManager();
User user = em.find(User.class, userId);
em.close();
```

Conclusão:

- **JDBC** é mais adequado para situações onde o desempenho e o controle fino sobre o SQL são críticos, ou onde uma abordagem de baixo nível é necessária

- **JPA** é ideal para desenvolvedores que procuram uma abordagem de alto nível para gerenciar dados de banco de dados com menos código boilerplate, abstração de SQL e facilidade de manutenção