Conexión bbdd en Kotlin

Tema 6

Contenido

[1.- Base de datos. 3](#_Toc128511444)

[2.- Métodos Acceso a base de datos 3](#_Toc128511445)

[2.1.- JDBC 4](#_Toc128511446)

[2.1.1.- Descargar controlador y cargarlo en IntelliJ IDEA 4](#_Toc128511447)

[2.1.2.- Ejemplo Básico 5](#_Toc128511448)

[2.1.3.- Ejemplo con patrón DAO 6](#_Toc128511449)

[2.1.4.- Errores típicos. 9](#_Toc128511450)

[3.- Palabras de base de datos. 10](#_Toc128511451)

# 1.- Base de datos.

La bbdd es un método para guardar información de forma persistente, para poder recuperarla en el momento que sea necesario, cuando se utiliza una aplicación intermedia para recuperar la información de la bbdd y presentarla al cliente, como desarrolladores debemos de intentar realizar nuestro trabajo de recuperación de la forma más correcta para generalizar y aislar los cambios que se puedan producir en el futuro, de tal manera que los cambios sean los menos posibles en las modificaciones. Se debe generalizar al máximo, esta generalización va a producir una mayor cantidad de código a escribir al inicio, pero su ventaja será que si aislamos bien las capas de la aplicación, cuando se produce un cambio, lo tendremos bien localizado y el mantenimiento será más eficiente.

En general, la cantidad de capas necesarias para abordar una base de datos en Kotlin y asegurar un mantenimiento eficiente y un aislamiento adecuado entre la base de datos y la aplicación depende de la complejidad y los requisitos específicos del proyecto en cuestión. Sin embargo, como práctica común, se suele dividir la aplicación en tres capas principales: **la capa de presentación, la capa de lógica de negocios y la capa de acceso a datos.**

* La capa de presentación se encarga de la interfaz de usuario y la interacción con el usuario.
* La capa de lógica de negocios se encarga de procesar la lógica del negocio de la aplicación.
* La capa de acceso a datos se encarga de la comunicación con la base de datos y de realizar las operaciones de lectura y escritura en la base de datos.

**Para garantizar un aislamiento adecuado entre la base de datos y la aplicación, se puede utilizar un patrón de diseño conocido como "Data Access Object" (DAO). Este patrón se utiliza para encapsular la lógica de acceso a datos en una capa separada de la capa de lógica de negocios. De esta manera, se puede garantizar que la capa de lógica de negocios no tenga conocimiento directo de la estructura de la base de datos y que cualquier cambio en la base de datos no afecte directamente a la capa de lógica de negocios.**

## 2.- Métodos Acceso a base de datos

Algunas formas comunes de conectar Kotlin con bases de datos son:

**JDBC** (Java Database Connectivity): Para conectarse a bases de datos SQL, como MySQL, PostgreSQL o Oracle. JDBC es una API estándar de Java para acceder a bases de datos relacionales.

**ORM** (Object Relational Mapping): Es una técnica que permite mapear objetos de programación a tablas de base de datos. Algunas de las bibliotecas ORM más populares para Kotlin son Hibernate, JOOQ, Exposed o Ktorm.

**NoSQL** Kotlin puede conectarse a bases de datos NoSQL, como MongoDB o Cassandra, utilizando bibliotecas específicas para estas bases de datos.

**Reactive** Kotlin también puede conectarse a bases de datos mediante bibliotecas reactivas, como Reactive Streams, que permiten trabajar con flujos de datos asincrónicos.

## 2.1.- JDBC

JDBC (Java Database Connectivity) es una API estándar de Java que permite a los programas Java/Kotlin conectarse a bases de datos relacionales, como MySQL, PostgreSQL, Oracle, etc. JDBC proporciona una interfaz común para acceder a bases de datos, independientemente del tipo de base de datos o del driver JDBC utilizado.

Los elementos principales de JDBC y cómo funciona:

**DriverManager**: Es el punto de entrada principal para obtener una conexión a una base de datos. Proporciona métodos para registrar los controladores JDBC, buscar los controladores que se han registrado y establecer una conexión con una base de datos.

**Connection**: Representa una conexión física con una base de datos. Proporciona métodos para crear sentencias SQL y transacciones, así como para controlar el nivel de aislamiento de la transacción y la confirmación o anulación de la transacción.

**Statement**: Se utiliza para enviar comandos SQL a la base de datos y recuperar los resultados. Proporciona métodos para ejecutar sentencias SQL y recuperar resultados, como filas de datos de una tabla.

**ResultSet:** Representa un conjunto de resultados de una consulta SQL. Proporciona métodos para acceder a los valores de las columnas en una fila de resultados y para mover el cursor a la siguiente fila.

**PreparedStatement**: Se utiliza para ejecutar sentencias SQL parametrizadas. Permite la reutilización de una sentencia SQL con diferentes parámetros, lo que puede mejorar el rendimiento y la seguridad al evitar la inyección de SQL.

### 2.1.1.- Descargar controlador y cargarlo en IntelliJ IDEA

Descarga el driver JDBC de MySQL desde el sitio web de MySQL. Asegúrate de descargar el archivo que coincide con la versión de MySQL que estás usando. (Ya te lo doy yo).

Agrega el archivo .jar del driver JDBC de MySQL al classpath de tu proyecto en IntelliJ IDEA. Para hacerlo, sigue estos pasos:

1.- En la barra de menú superior, selecciona "File" (Archivo) > "Project Structure" (Estructura del proyecto).

2.- En el panel izquierdo de la ventana de Project Structure, haz clic en "Modules" (Módulos).

3.- Selecciona el módulo de tu proyecto en la lista de módulos.

4.- En la pestaña "Dependencies" (Dependencias), haz clic en el botón "+" en la parte inferior de la ventana.

5.- Selecciona "JARs or directories" (Archivos JAR o directorios) y busca el archivo .jar del driver JDBC de MySQL que descargaste en el paso 1. Haz clic en "OK" para guardar la dependencia.

### 2.1.2.- Ejemplo Básico

|  |
| --- |
| Ejemplo BÁSICO de conexión JDBC. |
| import java.sql.\*  fun main(args: Array<String>) {  val url = "jdbc:mysql://localhost/supermercado"  val user = "root"  val password = ""  var conn: Connection? = null  var preparedStatement: PreparedStatement?=null  try {  // Cargar el driver JDBC  Class.forName("com.mysql.cj.jdbc.Driver")  // Establecer la conexión  conn = DriverManager.getConnection(url, user, password)  // Crear la sentencia SQL  val sql = "SELECT cod\_grupo, descripcion FROM categorias"  // Crear el objeto Statement  val stmt = conn.createStatement()  // Ejecutar la consulta y obtener los resultados  val rs = stmt.executeQuery(sql)  // Procesar los resultados  while (rs.next()) {  val cod\_grupo = rs.getString("cod\_grupo")  val descripcion = rs.getString("descripcion")  println("Código de grupo: $cod\_grupo, Descripción: $descripcion")  }  //ejemplo de PreparedStatement  // Creamos una sentencia preparada (PreparedStatement) para insertar datos en la tabla  val query = "INSERT INTO categorias (cod\_grupo, descripcion) VALUES (?, ?)"  preparedStatement = conn.prepareStatement(query)  // Luego, asignamos los valores que queremos insertar a los parámetros de la sentencia preparada    preparedStatement.setInt(1, 5) //Cada vez que lo ejecutes cambia el número, para no producir error  preparedStatement.setString(2, "Alimentos")  // Ejecutamos la sentencia preparada para insertar los datos en la tabla  preparedStatement.executeUpdate()  // Finalmente, cerramos la conexión a la base de datos y la sentencia preparada  } catch (ex: SQLException) {  ex.printStackTrace()  } finally {  // Cerrar la conexión  try {  preparedStatement?.close()  conn?.close()  } catch (ex: SQLException) {  ex.printStackTrace()  }  }  } |

Este ejemplo es una primera aproximación, poco profesional, pero efectiva para realizar nuestra primera conexión a una base de datos.

### 2.1.3.- Ejemplo con patrón DAO

Definimos la interfaz del DAO para la tabla "categorias":

|  |
| --- |
| interface CategoriaDAO {  fun getCategoriaByCodigo(codigo: Int): Categoria?  fun getAllCategorias(): List<Categoria>  fun insertCategoria(categoria: Categoria): Boolean  fun updateCategoria(categoria: Categoria): Boolean  fun deleteCategoria(codigo: Int): Boolean  } |

Implementamos la clase que se conecta a la base de datos:

|  |
| --- |
| class ConexionBD {  private val url = "jdbc:odbc:supermercado"  private var conn: Connection? = null  fun conectar() {  try {  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")  conn = DriverManager.getConnection(url)  } catch (e: SQLException) {  e.printStackTrace()  } catch (e: ClassNotFoundException) {  e.printStackTrace()  }  }  fun desconectar() {  try {  conn?.close()  } catch (e: SQLException) {  e.printStackTrace()  }  }  fun getStatement(): Statement? {  return conn?.createStatement()  }  fun getPreparedStatement(sql: String): PreparedStatement? {  return conn?.prepareStatement(sql)  }  } |

Creamos la clase que representa a la entidad "categoria":

|  |
| --- |
| class Categoria(val codigo: Int, var descripcion: String) {  override fun toString(): String {  return "Categoria [codigo=$codigo, descripcion=$descripcion]"  }  } |

Implementamos la clase que implementa la interfaz DAO para la tabla "categorias":

|  |
| --- |
| class CategoriaDAOImpl : CategoriaDAO {  private val conexion = ConexionBD()  override fun getCategoriaByCodigo(codigo: Int): Categoria? {  conexion.conectar()  val query = "SELECT \* FROM categorias WHERE cod\_grupo = ?"  val ps = conexion.getPreparedStatement(query)  ps?.setInt(1, codigo)  val rs = ps?.executeQuery()  var categoria: Categoria? = null  if (rs?.next() == true) {  categoria = Categoria(rs.getInt("cod\_grupo"), rs.getString("descripcion"))  }  ps?.close()  conexion.desconectar()  return categoria  }  override fun getAllCategorias(): List<Categoria> {  conexion.conectar()  val query = "SELECT \* FROM categorias"  val st = conexion.getStatement()  val rs = st?.executeQuery(query)  val categorias = mutableListOf<Categoria>()  while (rs?.next() == true) {  val categoria = Categoria(rs.getInt("cod\_grupo"), rs.getString("descripcion"))  categorias.add(categoria)  }  st?.close()  conexion.desconectar()  return categorias  }  override fun insertCategoria(categoria: Categoria): Boolean {  conexion.conectar()  val query = "INSERT INTO categorias (cod\_grupo, descripcion) VALUES (?, ?)"  val ps = conexion.getPreparedStatement(query)  ps?.setInt(1, categoria.codigo)  ps?.setString(2, categoria.descripcion)  val result = ps?.executeUpdate()  ps?.close()  conexion.desconectar()  return result == 1  }  override fun updateCategoria(categoria: Categoria): Boolean {  conexion.conectar()  val query = "UPDATE categorias SET descripcion = ? WHERE cod\_grupo = ?"  val ps = conexion.getPreparedStatement(query)  ps?.setString(1, categoria.descripcion)  ps?.setInt(2, categoria.codigo)  val result = ps?.executeUpdate()  ps?.close()  conexion.desconectar()  return result == 1  }  override fun deleteCategoria(codigo: Int): Boolean {  conexion.conectar()  val query = "DELETE FROM categorias WHERE cod\_grupo = ?"  val ps = conexion.getPreparedStatement(query)  ps?.setInt(1, codigo)  val result = ps?.executeUpdate()  ps?.close()  conexion.desconectar()  return result == 1  }  } |

Capa de negocio para presentar los resultados por la salida estándar:

|  |
| --- |
| fun main() {  val categoriaDAO = CategoriaDAOImpl()  // Obtener todas las categorias  val categorias = categoriaDAO.getAllCategorias()  println("Lista de categorias:")  categorias.forEach { println(it) }  // Obtener una categoria por codigo  val codigo = 1  val categoria = categoriaDAO.getCategoriaByCodigo(codigo)  if (categoria != null) {  println("Categoria con codigo $codigo:")  println(categoria)  } else {  println("No se encontró la categoria con codigo $codigo")  }  // Insertar una nueva categoria  val nuevaCategoria = Categoria(4, "Frutas")  if (categoriaDAO.insertCategoria(nuevaCategoria)) {  println("Se insertó correctamente la categoria $nuevaCategoria")  } else {  println("No se pudo insertar la categoria $nuevaCategoria")  }  // Actualizar una categoria existente  val categoriaExistente = Categoria(1, "Hogar y electrodomésticos")  if (categoriaDAO.updateCategoria(categoriaExistente)) {  println("Se actualizó correctamente la categoria $categoriaExistente")  } else {  println("No se pudo actualizar la categoria $categoriaExistente")  }  // Eliminar una categoria existente  val codigoAEliminar = 3  if (categoriaDAO.deleteCategoria(codigoAEliminar)) {  println("Se eliminó correctamente la categoria con codigo $codigoAEliminar")  } else {  println("No se pudo eliminar la categoria con codigo $codigoAEliminar")  }  } |

### 2.1.4.- Errores típicos.

No conecta con la base de datos:

* Mirar si has cargado el controlador.
* Mirar si la dirección a la base de datos es la correcta.
* Mirar si has lanzado el gestor de base de datos.
* Mirar el usuario y la contraseña

Conectamos, pero no realiza las operaciones:

Mirar los permisos de usuario en la base de datos.

* Mirar si la sentencia SQL es correcta.
* No inserta, mirar si el código de identificación no está duplicado.
* No borra, mirar si es clave externa y debemos realizar borrado en cascada.

# 3.- Palabras de base de datos.

ORM.-

JDBC.-

JPA.-

HIBERNATE.-

SPRING DATA JPA.-

JPQL.-