IES Virgen del Carmen

JUNTA DE ANDALUCIA
CONSEJERÍA DE EDUCACIÓN

Departamento de Informática

Programación de bases de datos relacionales

Juan Gualberto

Noviembre 2023

UNIÓN EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Índice

Gestión de Inventario

Con este proyecto vamos a aprender a crear aplicaciones que gestionen información almacenada en bases de datos relacionales. Vamos a construir una aplicación que lee la información de JSON/XML (importa archivos JSON) y la pasa a la base de datos. Dejaremos propuesto el proceso contrario: exportar la base de datos o parte de ella a JSON.

En este proyecto tenemos que crear unas clases modelo etiquetadas con JAXB para que podamos hacer marshalling y unmarshalling de JSON y los objetos que se cargan o guardan desde/hacia JSON a su vez de se guardan/leen (respectivamente) desde una base de datos MySQL.

En verdad estamos haciendo un programa para cargar y exportar ficheros JSON en una base de datos.

Con esta unidad integrada que vamos a trabajar en clase pretendemos cubrir los siguientes objetivos:

1. El desfase objeto-relacional.
2. Gestores de bases de datos embebidos e independientes.
3. Protocolos de acceso a bases de datos. Conectores.
4. Establecimiento de conexiones.
5. Definición de objetos destinados al almacenamiento del resultado de operaciones con bases de datos. Eliminación de objetos finalizada su función.
6. Ejecución de sentencias de descripción de datos.
7. Ejecución de sentencias de modificación de datos.
8. Ejecución de consultas.
9. Utilización del resultado de una consulta.
10. Ejecución de procedimientos almacenados en la base de datos.
11. Gestión de transacciones.

El desfase objeto-relacional

El desfase objeto-relacional en el acceso a datos se refiere a la discrepancia o incompatibilidad que puede existir entre los sistemas de gestión de bases de datos relacionales (RDBMS) y los modelos de objetos utilizados en el desarrollo de aplicaciones orientadas a objetos.

En el contexto de la programación orientada a objetos, los objetos se definen mediante clases y tienen atributos y métodos asociados. Sin embargo, las bases de datos relacionales almacenan los datos en tablas con filas y columnas, y utilizan el lenguaje SQL para manipular y consultar los datos. Esta diferencia fundamental entre los modelos de objetos y los modelos relacionales puede dar lugar a un desfase o falta de correspondencia cuando se trata de acceder a los datos de una base de datos relacional desde una aplicación orientada a objetos.

El desfase objeto-relacional puede manifestarse de diferentes maneras. Algunas de las dificultades comunes incluyen:

- Mapeo de objetos a tablas: El mapeo de objetos a tablas de bases de datos puede ser complicado debido a las diferencias en la estructura y semántica de los dos modelos. Por ejemplo, cómo se representan las relaciones entre objetos y cómo se traducen a relaciones de tablas.
- Herencia y polimorfismo: Los sistemas de gestión de bases de datos relacionales generalmente no admiten directamente conceptos de herencia y polimorfismo, que son características fundamentales de la programación orientada a objetos. Esto puede dificultar la representación de jerarquías de clases y la manipulación de objetos polimórficos en una base de datos relacional.
- Consultas y consultas complejas: La expresión de consultas complejas que involucran múltiples objetos y relaciones puede ser más difícil en un entorno objeto-relacional. La sintaxis y las operaciones disponibles en SQL pueden no ser tan expresivas o flexibles como en los lenguajes de consulta de objetos.

Para abordar este desfase, han surgido varios enfoques y tecnologías, como los mapeadores objeto-relacional (ORM) que proporcionan una capa de abstracción entre la base de datos relacional y la aplicación orientada a objetos. Estas herramientas facilitan el mapeo de objetos a tablas y proporcionan una interfaz más orientada a objetos para trabajar con los datos almacenados en la base de datos relacional. También existen bases de datos orientadas a objetos y bases de datos NoSQL que intentan superar las limitaciones de los sistemas de gestión de bases de datos relacionales en términos de soporte para la programación orientada a objetos.

Supongamos que tenemos una aplicación Java para gestionar una tienda en línea. Tenemos una clase Producto en Java que representa un producto en la tienda, con atributos como id, nombre y precio. Queremos almacenar y recuperar estos productos en una base de datos relacional utilizando SQL.

Primero, veamos cómo se definiría la clase Producto en Java:

```java
public class Producto {
    private int id;
    private String nombre;
    private double precio;

    // Constructor, getters y setters

    // Otros métodos de la clase
}
```

Ahora, para almacenar y recuperar objetos Producto en una base de datos relacional, necesitamos crear una tabla correspondiente en la base de datos y escribir consultas SQL.

Supongamos que utilizamos una base de datos MySQL y tenemos una tabla llamada productos con las columnas id, nombre y precio.

El código SQL para crear la tabla sería:

```sql
CREATE TABLE productos (
    id INT PRIMARY KEY,
    nombre VARCHAR(100),
    precio DECIMAL(10, 2)
);
```

Para almacenar un objeto Producto en la base de datos, tendríamos que traducir sus atributos a una consulta SQL de inserción:

```java
public void insertarProducto(Producto producto) {
    String sql = "INSERT INTO productos (id, nombre, precio) VALUES (?, ?, ?)";

    try (Connection conn = obtenerConexion();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setInt(1, producto.getId());
        stmt.setString(2, producto.getNombre());
        stmt.setDouble(3, producto.getPrecio());

        stmt.executeUpdate();
    } catch (SQLException e) {
        // Manejo de excepciones
    }
}
```

Aquí estamos utilizando una consulta parametrizada para evitar la concatenación directa de valores en la consulta, lo cual podría conducir a vulnerabilidades de seguridad como la inyección de SQL.

Para recuperar productos de la base de datos, podríamos ejecutar una consulta SQL y luego mapear los resultados a objetos Producto en Java:

```java
public List<Producto> obtenerProductos() {
    List<Producto> productos = new ArrayList<>();
    String sql = "SELECT id, nombre, precio FROM productos";

    try (Connection conn = obtenerConexion();
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        while (rs.next()) {
            int id = rs.getInt("id");
            String nombre = rs.getString("nombre");
            double precio = rs.getDouble("precio");

            Producto producto = new Producto(id, nombre, precio);
            productos.add(producto);
        }
```

```
17      } catch (SQLException e) {
18          // Manejo de excepciones
19      }
20
21      return productos;
22  }
```

En este ejemplo, ejecutamos una consulta SELECT para obtener todos los productos de la tabla productos. Luego, iteramos sobre los resultados del conjunto de resultados (ResultSet) y creamos objetos Producto a partir de ellos.

Este ejemplo muestra cómo se realiza el mapeo entre los objetos Java y las tablas SQL en un entorno objeto-relacional. El desfase objeto-relacional se manifiesta en la necesidad de escribir código adicional para traducir entre los objetos y las consultas SQL, así como en las diferencias de sintaxis y estructura entre los modelos de objetos y los modelos relacionales.

Para aprender a desarrollar una aplicación con un lenguaje orientado a objetos que almacene información en un sistema relacional, lo vamos a combinar con lo aprendido en el tema anterior (Manejo de archivos y Marshalling de objetos) y así el objetivo de esta pieza de software será importar/exportar datos de un archivo JSON/XML a una base de datos.

Gestores de bases de datos embebidos e independientes

Gestores de bases de datos embebidos son aquellos que se incorporan directamente en el código de la aplicación, no necesitamos un servicio, demonio o servidor para gestionar la base de datos.

Ejemplo de gestores embebidos:

- sqlite

Gestores de bases de datos independientes son aquellos que necesitan un demonio, proceso o servidor que gestione los archivos de la base de datos y al que nos conectamos desde la aplicación:

- MySQL
- Oracle
- PosgreSQL
- MariaDB
- SQL Server Express (Microsoft)
- etc.

Tenemos una lista muy interesante de sistemas de bases de datos en esta Web: https://db-engines.com/en/ranking.

Protocolos de acceso a bases de datos. Conectores.

El conector es una librería o ayuda en el lenguaje de programación que nos facilita la conexión a la base de datos.

En Java usamos la Java Database Connectivity (JDBC) es la especificación de una interfaz de programación de aplicaciones (API) que permite que los programas Java accedan a sistemas de gestión de bases de datos. La API JDBC consiste en un conjunto de interfaces y clases escribas en el lenguaje de programación Java.

La cadena de conexión obedece al patrón:

```
jdbc:driver:host:puerto/basededatos
```

Para MySQL por ejemplo sería:

```
jdbc:mysql://localhost:3306/inventario
```

En este ejemplo nos conectamos a un servidor local que está en el puerto 3306 escuchando y a la base de datos inventario.

Para Oracle:

```
jdbc:oracle:thin:@localhost:1521/oracleservice
```

En este ejemplo nos conectamos a un servidor local Oracle que está en el puerto 1521 y concretamente al servicio “oracleservice”.

Establecimiento de conexiones

Como hemos dicho antes, en Java se pueden establecer conexiones a bases de datos utilizando la API JDBC (Java Database Connectivity). JDBC proporciona una interfaz estándar para interactuar con diferentes sistemas de gestión de bases de datos.

Para establecer una conexión a una base de datos MySQL en Java tenemos que seguir los siguientes pasos:

1. Importamos las clases necesarias de java.sql para trabajar con JDBC y java.sql.SQLException para manejar las excepciones relacionadas con la base de datos.
2. Definimos, al menos, la URL de la base de datos, el nombre de usuario y la contraseña.
3. Registramos el controlador JDBC correspondiente para el sistema de gestión de base de datos que estamos utilizando. En este caso, estamos utilizando el controlador JDBC para MySQL (com.mysql.cj.jdbc.Driver), pero este paso puede variar según la base de datos que estés utilizando.

4. Luego, utilizamos el método DriverManager.getConnection() para establecer la conexión proporcionando la URL, el nombre de usuario y la contraseña.
5. Una vez establecida la conexión, puedes realizar operaciones en la base de datos, como consultas, actualizaciones, inserciones, etc.
6. Finalmente, cerramos la conexión utilizando el método close().

Recuerda que debes proporcionar todos los detalles específicos de tu base de datos, no sólo la URL, el nombre de usuario y la contraseña, en el código para que la conexión se establezca correctamente.

Ejemplo de conexión:

```
1  public class Conexion {
2
3      Connection conn;
4      Properties prop;
5
6      /**
7       * Constructor, lee archivo de propiedades y abre
8       * la conexión.
9       */
10     public Conexion() {
11         // Vía JDBC
12         if (conn == null) {
13             try (FileInputStream fis = new FileInputStream("db.
                   properties")) {
14
15                 // Registrar el controlador JDBC
16                 Class.forName("com.mysql.cj.jdbc.Driver");
17                 prop = new Properties();
18                 prop.load(fis);
19                 this.conn = DriverManager.getConnection(
20                         "jdbc:mysql://localhost:33306/inventario",
21                         prop);
22             } catch (SQLException | ClassCastException | IOException e)
                    {
23                 Logger.getLogger(Conexion.class.getName()).severe(e.
                       getLocalizedMessage());
24             }
25         }
26     }
27
28     public Connection getConnection() {
29         return conn;
30     }
31
32     public void destroy() {
33         if (this.conn != null) {
34             try {
35                 this.conn.close();
36             } catch (SQLException e) {
```

```
37
38                  }
39              }
40          }
41  }
```

Siendo el fichero de propiedades el siguiente:

```
1  user=root
2  password=zx76wbz7FG89k
3  useUnicode=yes
4  useJDBCCompliantTimezoneShift=true
```

Definición de objetos destinados al almacenamiento del resultado de operaciones con bases de datos.

La definición y manipulación de objetos destinados al almacenamiento del resultado de operaciones con bases de datos es una parte fundamental de la mayoría de los lenguajes de programación. Estos objetos, generalmente conocidos como "objetos de dominio" o **"entidades"**, representan los datos recuperados de la base de datos y se utilizan para interactuar con esos datos en la aplicación.

Aquí hay algunas buenas prácticas y conceptos clave relacionados con la definición de objetos destinados al almacenamiento del resultado de operaciones con bases de datos:

1. **Modelado de Datos**: Antes de definir los objetos de dominio, es fundamental comprender la estructura de los datos en la base de datos. Esto incluye tablas, columnas, relaciones y tipos de datos.
2. **Clases de Entidades**: Cada tabla en la base de datos debe tener una clase de entidad correspondiente en Java. Esta clase contendrá propiedades que representan las columnas de la tabla.
3. **Mapeo Objeto-Relacional (ORM)**: El uso de herramientas ORM, como Hibernate o JPA, simplifica la definición de objetos de dominio y la interacción con la base de datos. Estas herramientas mapean automáticamente las tablas de la base de datos a las clases de entidad en Java.
4. **Encapsulación de Datos**: Las propiedades de una clase de entidad deben ser privadas y se acceden a través de métodos getter y setter para garantizar la encapsulación de datos. Esto permite el control sobre cómo se acceden y modifican los datos.
5. **Constructores Personalizados**: Además de los constructores predeterminados, es útil proporcionar constructores personalizados para inicializar objetos de entidad de manera coherente y segura.
6. **Validación de Datos**: Agregar lógica de validación en los métodos setter o en métodos específicos de validación para garantizar que los datos cumplan con las restricciones y reglas de negocio.

7. **Sobrescritura de Métodos**: Es útil sobrescribir los métodos `equals()`, `hashCode()`, y `toString()` para facilitar la comparación de objetos, trabajar con colecciones y depurar.
8. **Relaciones entre Entidades**: Si existen relaciones entre tablas en la base de datos, estas relaciones deben reflejarse en las clases de entidad. Por ejemplo, una relación uno a uno, uno a muchos o muchos a muchos.
9. **Serializable**: Si es necesario, implementar la interfaz `Serializable` para permitir la serialización de objetos, como en el caso de almacenamiento en caché o transferencia a través de la red.
10. **Auditoría y Registro**: Puede ser beneficioso incluir campos de auditoría, como fecha de creación, fecha de modificación y el usuario responsable, para el seguimiento y la auditoría de los datos.
11. **Pruebas Unitarias**: Asegúrate de que las clases de entidad se puedan probar de manera aislada utilizando pruebas unitarias para garantizar su correcto funcionamiento.

La definición adecuada de objetos de dominio es esencial para el diseño y el rendimiento de aplicaciones que interactúan con bases de datos. También es importante tener en cuenta las prácticas de diseño de bases de datos para lograr una integración eficiente y coherente.

Eliminación de objetos finalizada su función.

Una vez hemos realizado las operaciones necesarias contra la base de datos, es necesaria la liberación de recursos y la eliminación de objetos una vez que han cumplido su propósito y ya no son necesarios en la aplicación. Aunque este concepto es más amplio y no se limita únicamente a conexiones de bases de datos, las conexiones de bases de datos son un ejemplo común en el contexto de la eliminación de objetos.

De hecho, un fallo muy común es no liberar correctamente objetos y/o recursos y esto es explotado por usuarios maliciosos para acceder a información que no deberían tener. Puedes ver más intormación en la Web de MITRE: https://cwe.mitre.org/top25/index.html.

En el caso de conexiones de bases de datos, es esencial administrarlas adecuadamente para evitar problemas de rendimiento, fugas de recursos y bloqueos. Aquí hay algunos puntos clave relacionados con la eliminación de conexiones de bases de datos:

1. **Cierre de Conexiones**: Cada vez que se abre una conexión a una base de datos, es fundamental cerrarla una vez que ya no se necesite. Esto se logra llamando al método `close()` en la conexión. No cerrar conexiones puede agotar los recursos de la base de datos y afectar negativamente el rendimiento.

2. **Uso de Bloques `try-with-resources`**: En Java, se recomienda utilizar la estructura `try-with-resources` al trabajar con conexiones de bases de datos. Esto garantiza que las conexiones se cierren automáticamente al salir del bloque `try`, incluso en caso de excepciones.

```
try (Connection connection = DriverManager.getConnection(url,
    username, password)) {
    // Trabajar con la conexión
} catch (SQLException e) {
    // Manejo de excepciones
}
// La conexión se cierra automáticamente al salir del bloque try
```

3. **Pooling de Conexiones**: En aplicaciones empresariales, es común utilizar pooling de conexiones. En lugar de abrir y cerrar conexiones de manera individual, se mantienen en un pool y se reutilizan. Esto mejora la eficiencia y el rendimiento, ya que se evita el costo de abrir y cerrar conexiones repetidamente.

4. **Administración de Transacciones**: Es importante administrar las transacciones de manera adecuada. Una transacción debe comprometerse (`commit`) o deshacerse (`rollback`) según sea necesario antes de cerrar la conexión.

5. **Monitorización de Recursos**: Es útil monitorear el uso de recursos, como conexiones de bases de datos, para identificar posibles problemas de fugas de recursos o bloqueos.

Si bien la eliminación de conexiones de bases de datos es un aspecto crítico, la eliminación de objetos finalizados su función se aplica a muchos otros recursos en una aplicación, como archivos, sockets, recursos de memoria y más. La gestión adecuada de estos recursos es esencial para mantener la eficiencia, la estabilidad y la seguridad de la aplicación.

Ejecución de sentencias de descripción de datos.

La Ejecución de Sentencias de Descripción de Datos (DDL, por sus siglas en inglés, Data Definition Language) se refiere a un conjunto de comandos utilizados en sistemas de gestión de bases de datos (DBMS) para definir, modificar y eliminar la estructura de la base de datos. Estas sentencias no se utilizan para manipular los datos en sí, sino para definir la estructura de la base de datos y sus objetos. Algunas de las tareas comunes realizadas mediante DDL incluyen la creación de tablas, la definición de restricciones de integridad, la modificación de esquemas y la eliminación de objetos de la base de datos.

Aquí hay algunas de las sentencias DDL más comunes y su función:

1. **CREATE**: La sentencia `CREATE` se utiliza para crear objetos de la base de datos, como tablas, índices, vistas, procedimientos almacenados, funciones y más. Por ejemplo, `CREATE TABLE`

se utiliza para crear una tabla en la base de datos.

2. **ALTER**: La sentencia `ALTER` se utiliza para modificar la estructura de la base de datos existente. Puede usarse para agregar, modificar o eliminar columnas en una tabla, cambiar el nombre de objetos, agregar restricciones y más. Por ejemplo, `ALTER TABLE` se utiliza para modificar una tabla.
3. **DROP**: La sentencia `DROP` se utiliza para eliminar objetos de la base de datos, como tablas, vistas, índices u otros objetos. Por ejemplo, `DROP TABLE` se utiliza para eliminar una tabla.
4. **TRUNCATE**: La sentencia `TRUNCATE` se utiliza para eliminar todos los datos de una tabla, pero no la estructura de la tabla en sí. Es más eficiente que la eliminación de filas una por una.
5. **COMMENT**: La sentencia `COMMENT` se utiliza para agregar comentarios o descripciones a objetos de la base de datos, como tablas, columnas o vistas.
6. **RENAME**: Algunos sistemas de gestión de bases de datos permiten la sentencia `RENAME` para cambiar el nombre de un objeto existente, como una tabla.
7. **GRANT y REVOKE**: Aunque técnicamente son sentencias DCL (Data Control Language), `GRANT` y `REVOKE` se utilizan para otorgar o revocar permisos y privilegios a usuarios y roles en la base de datos. Esto afecta la seguridad y el acceso a los objetos de la base de datos.

Las sentencias DDL son esenciales para diseñar y administrar bases de datos. Los administradores de bases de datos y los desarrolladores utilizan estas sentencias para crear, modificar y mantener la estructura de la base de datos, lo que incluye definir las tablas y sus relaciones, aplicar restricciones de integridad, definir índices y otros elementos esenciales. La ejecución de DDL tiene un impacto directo en la estructura de la base de datos y, por lo tanto, debe realizarse con precaución y consideración.

Ejecución de sentencias de modificación de datos. CRUD básico.

La Ejecución de Sentencias de Modificación de Datos (DML, por sus siglas en inglés, Data Manipulation Language) se refiere a un conjunto de comandos utilizados en sistemas de gestión de bases de datos (DBMS) para realizar operaciones que modifican los datos almacenados en la base de datos. Estas operaciones se conocen comúnmente como operaciones CRUD, que representan las siguientes acciones:

1. **Create (Crear)**: Insertar nuevos registros en una tabla.
2. **Read (Leer)**: Recuperar datos de una tabla.
3. **Update (Actualizar)**: Modificar registros existentes en una tabla.
4. **Delete (Eliminar)**: Eliminar registros de una tabla.

A continuación, te proporcionaré un ejemplo de un CRUD básico en Java utilizando el lenguaje SQL para realizar operaciones de modificación de datos en una tabla de una base de datos ficticia. Ten en cuenta que necesitarás una base de datos real o una base de datos en memoria (como H2) para ejecutar este ejemplo.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class CrudExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/tu_base_de_datos";
        String usuario = "tu_usuario";
        String contraseña = "tu_contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contraseña)) {
            // Create (Insert)
            insertarRegistro(connection, "EjemploNombre", "EjemploApellido");

            // Read (Select)
            consultarRegistros(connection);

            // Update (Update)
            actualizarRegistro(connection, 1, "NuevoNombre", "NuevoApellido");

            // Read (Select)
            consultarRegistros(connection);

            // Delete (Delete)
            eliminarRegistro(connection, 1);

            // Read (Select)
            consultarRegistros(connection);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void insertarRegistro(Connection connection, String nombre, String apellido) throws SQLException {
        String insertQuery = "INSERT INTO tabla_ejemplo (nombre, apellido) VALUES (?, ?)";
        try (PreparedStatement preparedStatement = connection.prepareStatement(insertQuery)) {
            preparedStatement.setString(1, nombre);
```

```
40              preparedStatement.setString(2, apellido);
41              int filasAfectadas = preparedStatement.executeUpdate();
42              if (filasAfectadas > 0) {
43                  System.out.println("Registro insertado con éxito.");
44              }
45          }
46      }
47
48      private static void consultarRegistros(Connection connection)
            throws SQLException {
49          String selectQuery = "SELECT * FROM tabla_ejemplo";
50          try (PreparedStatement preparedStatement = connection.
                prepareStatement(selectQuery);
51               ResultSet resultSet = preparedStatement.executeQuery()) {
52              while (resultSet.next()) {
53                  int id = resultSet.getInt("id");
54                  String nombre = resultSet.getString("nombre");
55                  String apellido = resultSet.getString("apellido");
56                  System.out.println("ID: " + id + ", Nombre: " + nombre
                        + ", Apellido: " + apellido);
57              }
58          }
59      }
60
61      private static void actualizarRegistro(Connection connection, int
            id, String nuevoNombre, String nuevoApellido) throws
            SQLException {
62          String updateQuery = "UPDATE tabla_ejemplo SET nombre = ?,
                apellido = ? WHERE id = ?";
63          try (PreparedStatement preparedStatement = connection.
                prepareStatement(updateQuery)) {
64              preparedStatement.setString(1, nuevoNombre);
65              preparedStatement.setString(2, nuevoApellido);
66              preparedStatement.setInt(3, id);
67              int filasAfectadas = preparedStatement.executeUpdate();
68              if (filasAfectadas > 0) {
69                  System.out.println("Registro actualizado con éxito.");
70              }
71          }
72      }
73
74      private static void eliminarRegistro(Connection connection, int id)
            throws SQLException {
75          String deleteQuery = "DELETE FROM tabla_ejemplo WHERE id = ?";
76          try (PreparedStatement preparedStatement = connection.
                prepareStatement(deleteQuery)) {
77              preparedStatement.setInt(1, id);
78              int filasAfectadas = preparedStatement.executeUpdate();
79              if (filasAfectadas > 0) {
80                  System.out.println("Registro eliminado con éxito.");
81              }
```

```
82          }
83      }
84  }
```

Este ejemplo de CRUD en Java utiliza JDBC (Java Database Connectivity) para interactuar con una base de datos MySQL. Puedes adaptar el código para que se ajuste a tu base de datos y requerimientos específicos. El CRUD incluye la inserción de registros, consulta de registros, actualización de registros y eliminación de registros en una tabla ficticia llamada “tabla_ejemplo”.

LEER uno (findOne)

Veamos otro ejemplo más concreto de la operación **findOne** también referida en muchas herramientas como **findById** (buscar por la clave):

```
1       String jsonObject="{}";
2       Connection conexion;
3       PreparedStatement pstm;
4       String jdbcURL;
5
6       jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;
7
8       try {
9           Class.forName("com.mysql.cj.jdbc.Driver");
10          conexion =  DriverManager.getConnection(jdbcURL, "root", "
                example");
11          String sql = "SELECT * FROM usuario WHERE id=?";
12          pstm = conexion.prepareStatement(sql);
13          pstm.setInt(1, Integer.parseInt(id));
14          ResultSet rs = pstm.executeQuery();
15          if ( rs.next() ) {
16              String username = rs.getString("username");
17              String password = rs.getString("password");
18              String email = rs.getString("email");
19              // String id = rs.getString("id");
20              jsonObject="{"+ "\n"+
21                  "'id':'"+id+"',"+ "\n"+
22                  "'username':'"+username+"',"+ "\n"+
23                  "'password':'"+password+"',"+ "\n"+
24                  "'email':'"+email+"'"+ "\n"+
25                  "}";
26          }
27      }catch(Exception ex){
28          // Gestión de la excepción
29      }
30      // devolvemos jsonObject
```

LEER todos (findAll)

Veamos ahora un ejemplo detallado de buscar todos los registros de la base de datos:

```
    String jsonObject="{}";
    Connection conexion;
    PreparedStatement pstm;
    String jdbcURL;

    jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;

    try {
        Persona p;
        List<Persona> pl = new ArrayList<Persona>();
        Class.forName("com.mysql.cj.jdbc.Driver");
        conexion =  DriverManager.getConnection(jdbcURL, "root", "
           example");
        String sql = "SELECT * FROM usuario WHERE id=?";
        pstm = conexion.prepareStatement(sql);
        pstm.setInt(1, Integer.parseInt(id));
        ResultSet rs = pstm.executeQuery();
        if ( rs.next() ) {
            String username = rs.getString("username");
            String password = rs.getString("password");
            String email = rs.getString("email");
            // String id = rs.getString("id");
            p = new Persona(username, password, email);
            pl.add(p)
        }
    }catch(Exception ex){
        // Gestión de la excepción
    }
    // devolvemos pl
```

Crear

Sea un objeto **usuario** del que queremos guardar los datos:

- *id*: Identificador único del usuario.
- *username*: Nombre de usuario.
- *password*: Contraseña.
- *email*: Correo electrónico.

La siguiente porción de código Java explica cómo podríamos insertar en la tabla *ususario* sus datos:

```
    Connection conexion;
    PreparedStatement pstm;
    String jdbcURL = "jdbc:mysql://localhost:33306/inventario";
```

```
4       jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;
5       try {
6           Class.forName("com.mysql.jdbc.Driver");
7           conexion =  DriverManager.getConnection(jdbcURL, "root", "
                example");
8           String sql = "INSERT INTO usuario (username,password,email)
                VALUES(?,?,?)";
9           pstm = conexion.prepareStatement(sql);
10          pstm.setString(1, username);
11          pstm.setString(2, password);
12          pstm.setString(3, email);
13          if (pstm.executeUpdate() >0) {
14              // "Usuario insertado"
15          } else {
16              // "No se ha podido insertar"
17          }
18          conexion.close();
19      } catch (Exception ex) {
20          // "Imposible conectar a la BBDD"
21      }
```

En operaciones que impliquen pedir datos al usuario, como por ejemplo aquí, siempre que sea posible será preferible usar **PreparedStatement** frente a la ejecución directa de la consulta para prevenir frente a ataques de inyección de código. Como se ve en el ejemplo, cada interrogante posteriormente se sustituye por la variable correspondiente, no permitiendo la introducción de otros caracteres maliciosos que puedan dar lugar a la inyección.

Actualizar

En el siguiente ejemplo vamos a ver una sentencia de actualización. Suponemos que lo único que no cambia es el ID de usuario luego el resto de campos son susceptibles de ser actualizados:

```
1
2       User user = new User(1, "obijuan", "Secreto123, "obijuan@star.wars"
            );
3
4       String jdbcURL = "jdbc:mysql://localhost:33306/inventario";
5
6       try {
7           Class.forName("com.mysql.cj.jdbc.Driver");
8           Connection conexion =  DriverManager.getConnection(jdbcURL, "
                root", "example");
9           String sql = "UPDATE usuario SET username=?, password=?, email
                =? WHERE id=?";
10          PreparedStatement pstm = conexion.prepareStatement(sql);
11          pstm.setString(1, user.getUsername());
12          pstm.setString(2, user.getPassword());
13          pstm.setString(3, user.getEmail());
```

```
14          pstm.setInt(4, user.getId());
15
16          if (pstm.executeUpdate() >0) {
17              // println("Usuario insertado");
18          } else {
19              // println("No se ha podido insertar");
20          }
21
22          conexion.close();
23      } catch (Exception ex) {
24          // gestión de excepciones
25      }
```

Borrar

A continuación vemos un ejemplo de cómo borrar un registro de la base de datos:

```
1       Connection conexion;
2       PreparedStatement pstm;
3       String jdbcURL;
4
5       jdbcURL = "jdbc:mysql://localhost:33306/inventario";
6       try {
7           Class.forName("com.mysql.cj.jdbc.Driver");
8           conexion =  DriverManager.getConnection(jdbcURL, "root", "
                example");
9           String sql = "DELETE FROM usuario WHERE id=?";
10          pstm = conexion.prepareStatement(sql);
11          pstm.setInt(1, Integer.parseInt(id));
12          if ( pstm.executeUpdate()==0 ) {
13              // no se ha encontrado el ID en la base de datos
14          } else {
15              // usuario eliminado correctamente
16          }
17      }catch(Exception ex){
18          // "No se pudo eliminar"
19      }
```

Ejecución de consultas.

Resumiendo lo visto hasta ahora, la ejecución de consultas de una base de datos en Java implica el uso de sentencias SQL para recuperar datos de una base de datos. Java proporciona una API estándar llamada JDBC (Java Database Connectivity) que facilita la conexión a bases de datos y la ejecución de consultas SQL. Aquí hay un resumen de cómo ejecutar consultas de bases de datos en Java:

1. **Establecer una Conexión a la Base de Datos**:

- Utiliza la clase `java.sql.Connection` para establecer una conexión a tu base de datos. Esto generalmente se hace proporcionando la URL de conexión, el nombre de usuario y la contraseña.

```
Connection connection = DriverManager.getConnection(url, usuario,
    contraseña);
```

2. **Preparar la Consulta**:

   - Utiliza objetos de tipo `PreparedStatement` o `Statement` para preparar tus consultas SQL. Los `PreparedStatement` son preferibles debido a su seguridad y mejor rendimiento.

```
String sql = "SELECT * FROM tabla WHERE condicion = ?";
PreparedStatement preparedStatement = connection.prepareStatement(
    sql);
preparedStatement.setString(1, valorCondicion);
```

3. **Ejecutar la Consulta**:

   - Utiliza el método `executeQuery()` para consultas `SELECT` que devuelven un conjunto de resultados, y `executeUpdate()` para consultas `INSERT`, `UPDATE` o `DELETE` que modifican la base de datos.

```
ResultSet resultSet = preparedStatement.executeQuery();
```

4. **Recuperar y Procesar Resultados**:

   - Si la consulta devuelve resultados, utiliza un objeto `ResultSet` para acceder a los datos recuperados. Puedes iterar a través del `ResultSet` para obtener los valores de las columnas.

```
while (resultSet.next()) {
    String columna1 = resultSet.getString("nombre_columna1");
    int columna2 = resultSet.getInt("nombre_columna2");
    // Realiza operaciones con los datos
}
```

5. **Cerrar Recursos**:

   - Es importante cerrar todos los recursos, como la conexión, el `PreparedStatement` y el `ResultSet`, cuando hayas terminado de usarlos. Esto se puede hacer en un bloque `try-with-resources` para garantizar que se cierren adecuadamente.

```
try (Connection connection = DriverManager.getConnection(url,
    usuario, contraseña);
```

```
2       PreparedStatement preparedStatement = connection.
            prepareStatement(sql);
3       ResultSet resultSet = preparedStatement.executeQuery()) {
4      // Operaciones de consulta y procesamiento de resultados
5  } catch (SQLException e) {
6      e.printStackTrace();
7  }
```

6. **Manejo de Excepciones**:
    - Las operaciones de base de datos pueden generar excepciones de tipo `SQLException`. Es importante manejar estas excepciones adecuadamente, ya sea registrándolas, notificando al usuario o tomando medidas específicas en caso de error.
7. **Seguridad**:
    - Evita concatenar valores directamente en las consultas SQL para prevenir ataques de inyección SQL. En su lugar, utiliza parámetros o `PreparedStatement` para asignar valores de manera segura en las consultas.
8. **Optimización de Consultas**:
    - Para consultas que se ejecutan con frecuencia, considera la indexación de columnas relevantes para mejorar el rendimiento. También puedes utilizar técnicas de optimización de consultas, como la limitación de resultados y la paginación.
9. **Pruebas Unitarias**:
    - Realiza pruebas unitarias para garantizar que tus consultas funcionen correctamente y produzcan los resultados esperados.

Java ofrece una gran flexibilidad y potencia para interactuar con bases de datos a través de JDBC. Los desarrolladores pueden trabajar con una variedad de bases de datos relacionales y no relacionales, y ejecutar consultas complejas para satisfacer las necesidades de sus aplicaciones.

Utilización del resultado de una consulta.

El resultado de una consulta SQL se recupera generalmente en forma de un conjunto de resultados, que es representado en Java por un objeto de tipo `ResultSet`. Puedes utilizar el `ResultSet` para acceder a los datos recuperados de la base de datos y realizar diversas operaciones con ellos. Aquí hay algunas formas comunes de utilizar el resultado de una consulta:

1. **Recuperar Datos**: Utiliza métodos como `getString()`, `getInt()`, `getDouble()`, etc., del objeto `ResultSet` para recuperar los valores de las columnas del conjunto de resultados. Puedes acceder a los datos por nombre de columna o por posición.

```java
ResultSet resultSet = preparedStatement.executeQuery();
while (resultSet.next()) {
    String nombre = resultSet.getString("nombre");
    int edad = resultSet.getInt("edad");
    // Realiza operaciones con los datos recuperados
}
```

2. **Iterar a través de Resultados**: Utiliza un bucle `while` o `for` para iterar a través de los resultados del `ResultSet`. Esto te permite procesar todos los registros recuperados por la consulta.

```java
while (resultSet.next()) {
    // Procesar cada registro
}
```

3. **Almacenar en Estructuras de Datos**: Puedes almacenar los resultados en estructuras de datos de Java, como listas, mapas o arreglos, para su posterior procesamiento o presentación.

```java
List<Registro> registros = new ArrayList<>();
while (resultSet.next()) {
    String nombre = resultSet.getString("nombre");
    int edad = resultSet.getInt("edad");
    registros.add(new Registro(nombre, edad));
}
```

4. **Realizar Cálculos o Transformaciones**: Puedes realizar cálculos, transformaciones o filtrar los datos recuperados antes de usarlos. Esto es útil cuando necesitas manipular los datos antes de mostrarlos o almacenarlos.

```java
while (resultSet.next()) {
    double precio = resultSet.getDouble("precio");
    double descuento = precio * 0.1; // Aplicar un descuento del
        10%
    // Realizar otras operaciones
}
```

5. **Presentar los Datos**: Los datos recuperados de la base de datos se pueden utilizar para mostrar información al usuario. Esto puede incluir la presentación de datos en una interfaz de usuario, informes, gráficos u otros medios.

6. **Actualizar o Modificar Datos**: En algunas situaciones, es posible que desees actualizar o modificar los datos recuperados y guardar los cambios en la base de datos. Para ello, necesitas construir y ejecutar sentencias SQL de modificación de datos (DML).

7. **Realizar Operaciones de Negocio**: Los datos recuperados se pueden utilizar para realizar operaciones de negocio, como cálculos, generación de informes, toma de decisiones, etc.

8. **Manejar Errores y Excepciones**: Es importante manejar adecuadamente los errores y excep-

ciones que puedan surgir al trabajar con los resultados de una consulta. Esto puede incluir la comprobación de valores nulos, la gestión de excepciones de SQL y la notificación adecuada al usuario en caso de problemas.

En resumen, el resultado de una consulta se utiliza para acceder y procesar los datos de la base de datos, lo que te permite realizar una amplia variedad de operaciones según los requisitos de tu aplicación. La manipulación de datos recuperados es esencial en la mayoría de las aplicaciones, ya que permite interactuar con la información almacenada en la base de datos y utilizarla de manera significativa.

Ejecución de procedimientos almacenados en la base de datos.

La ejecución de procedimientos almacenados en una base de datos es una técnica que permite a los desarrolladores encapsular lógica de negocio en la base de datos misma. Un procedimiento almacenado es una colección de instrucciones SQL que se almacenan en la base de datos y se pueden invocar mediante un nombre o identificador. Esto ofrece varias ventajas:

1. **Reutilización de Código**: Los procedimientos almacenados permiten reutilizar lógica de negocio en múltiples partes de una aplicación. Esto evita la duplicación de código y garantiza la consistencia en la aplicación.
2. **Mejora del Rendimiento**: Los procedimientos almacenados se pueden compilar y optimizar en la base de datos, lo que puede resultar en un mejor rendimiento en comparación con la ejecución de consultas SQL individuales desde la aplicación.
3. **Seguridad**: Los procedimientos almacenados pueden controlar el acceso a los datos al definir quién tiene permiso para ejecutarlos. Esto ayuda a aplicar políticas de seguridad de manera centralizada.
4. **Mantenibilidad**: Si se necesita cambiar la lógica de negocio, se puede modificar el procedimiento almacenado sin necesidad de actualizar la aplicación. Esto facilita el mantenimiento a largo plazo.
5. **Transacciones**: Los procedimientos almacenados pueden agrupar un conjunto de operaciones en una transacción, lo que garantiza la atomicidad de las operaciones y evita problemas de consistencia de datos.
6. **Abstracción de Datos**: Los procedimientos almacenados permiten abstraer detalles de implementación en la base de datos. Los cambios en la estructura de las tablas no afectan a las aplicaciones que utilizan procedimientos almacenados.

Para ejecutar un procedimiento almacenado desde una aplicación Java, se pueden seguir estos pasos:

1. **Preparar la Llamada al Procedimiento**: Debes preparar una llamada al procedimiento almacenado utilizando una instancia de `CallableStatement`. Puedes pasar parámetros al procedimiento almacenado y especificar cualquier valor de retorno.

```
CallableStatement callableStatement = connection.prepareCall("{
    call NombreDelProcedimiento(?, ?, ?)}");
callableStatement.setInt(1, parametro1);
callableStatement.setString(2, parametro2);
callableStatement.registerOutParameter(3, Types.INTEGER); //
    Ejemplo de parámetro de salida
```

2. **Ejecutar el Procedimiento Almacenado**: Utiliza el método `execute()` o `executeQuery()` del `CallableStatement` para ejecutar el procedimiento almacenado.

```
callableStatement.execute();
```

3. **Recuperar Resultados**: Si el procedimiento almacenado devuelve valores, puedes utilizar el `CallableStatement` para recuperarlos.

```
int valorRetorno = callableStatement.getInt(3); // Ejemplo de
    recuperación de valor de retorno
```

4. **Cerrar Recursos**: Al igual que con otras operaciones de base de datos, es importante cerrar adecuadamente los recursos cuando hayas terminado de utilizarlos.

```
callableStatement.close();
```

Es importante tener en cuenta que la sintaxis y la forma de invocar procedimientos almacenados pueden variar según la base de datos que estés utilizando. Además, debes asegurarte de que la base de datos y el controlador JDBC sean compatibles con la ejecución de procedimientos almacenados.

En resumen, la ejecución de procedimientos almacenados es una práctica común en el desarrollo de aplicaciones empresariales, ya que proporciona ventajas significativas en términos de reutilización, rendimiento y seguridad. Permite separar la lógica de negocio de la capa de acceso a datos y ofrece una forma eficaz de interactuar con una base de datos desde una aplicación Java.

Gestión de transacciones.

La gestión de transacciones en bases de datos es fundamental para garantizar la integridad y la consistencia de los datos en aplicaciones. En Java, puedes gestionar transacciones utilizando la API JDBC (Java Database Connectivity). Aquí tienes una descripción general de la gestión de transacciones en Java y un ejemplo con MySQL:

Conceptos Clave de la Gestión de Transacciones:

1. **Inicio de Transacción**: Debes iniciar explícitamente una transacción antes de ejecutar operaciones que involucren cambios en la base de datos. Esto se hace generalmente llamando al método `setAutoCommit(false)` en la conexión, lo que deshabilita el modo de autocommit.
2. **Commit**: Cuando todas las operaciones dentro de una transacción se han ejecutado con éxito y deseas que los cambios se confirmen de manera permanente en la base de datos, debes llamar al método `commit()` de la conexión.
3. **Rollback**: Si ocurre un error o una excepción durante una transacción y deseas deshacer todos los cambios realizados dentro de esa transacción, debes llamar al método `rollback()` de la conexión.
4. **AutoCommit**: Si `setAutoCommit(true)` está habilitado (modo de autocommit), cada sentencia SQL se confirma automáticamente como una transacción independiente. Si está deshabilitado (`setAutoCommit(false)`), debes llamar explícitamente a `commit()` o `rollback()`.
5. **Puntos de Guardado (Savepoints)**: Puedes establecer puntos de guardado dentro de una transacción para poder volver a un estado anterior si ocurre un error. Esto se logra mediante `Savepoint` y los métodos `setSavepoint()` y `rollbackToSavepoint()`.

Ejemplo de Gestión de Transacciones en Java con MySQL:

A continuación, te proporciono un ejemplo de gestión de transacciones en Java utilizando MySQL y JDBC:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TransactionExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/tu_base_de_datos";
        String usuario = "tu_usuario";
        String contraseña = "tu_contraseña";

        try (Connection connection = DriverManager.getConnection(url,
            usuario, contraseña)) {
            // Deshabilitar el modo de autocommit
            connection.setAutoCommit(false);

            // Operaciones dentro de la transacción
            Statement statement = connection.createStatement();
            statement.executeUpdate("INSERT INTO tabla_ejemplo (nombre,
                edad) VALUES ('Juan', 30)");
            statement.executeUpdate("INSERT INTO tabla_ejemplo (nombre,
                edad) VALUES ('María', 28)");

```

```
21              // Confirmar la transacción
22              connection.commit();
23          } catch (SQLException e) {
24              // En caso de error, realizar un rollback para deshacer los
                    cambios
25              if (connection != null) {
26                  try {
27                      connection.rollback();
28                  } catch (SQLException ex) {
29                      ex.printStackTrace();
30                  }
31              }
32              e.printStackTrace();
33          }
34      }
35  }
```

En este ejemplo, se deshabilita el modo de autocommit (`setAutoCommit(false)`) antes de realizar las operaciones. Si ocurre una excepción durante las operaciones, se llama a `rollback()` para deshacer los cambios. Si las operaciones tienen éxito, se confirman con `commit()`. La gestión de transacciones es esencial para garantizar que los cambios en la base de datos sean consistentes y seguros.

Creación de la BBDD

Preparando los contenedores Docker para la base de datos

Instalamos Docker si no estuviese:

```
sudo apt install docker.io docker-registry docker-compose
```

Para no tener que usar “sudo” debemos añadir el usuario al grupo de Docker:

En el fichero /etc/group buscamos una línea similar a:

> docker:x:136

Y le añadimos nuestro username:

> docker:x:136:usuario

Ahora o bien lanzamos los servicios con systemctl o bien reiniciamos la máquina, lo que más rápido sea.

Creamos un archivo docker-compose.yml para la base de datos MySQL:

```
version: '3.1'

services:

  db-inventario:
    image: mysql
    # NOTE: use of "mysql_native_password" is not recommended: https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-caching-sha2-password
    # (this is just an example, not intended to be a production configuration)
    command: --default-authentication-plugin=mysql_native_password
    restart: "no"
    environment:
      MYSQL_ROOT_PASSWORD: zx76wbz7FG89k
    ports:
      - 33306:3306

  adminer:
    image: adminer
    restart: "no"
    ports:
      - 8181:8080
```

Este grupo de dos servicios (contenedores) levanta:

- **db-inventario**: Es un servicio basado en la imagen por defecto de MySQL, configuramos el usuario root con la contraseña *example* y (gracias la documentación de la imagen oficial de dockerhub) es fácil ver cómo incluir un archivo SQL de *entrypoint* para inicializar la base de datos. Si el fichero contiene errores el contenedor no arranca. Sólo cuando esté terminado y probado debemos incluir estas líneas en el fichero YAML. El servicio MySQL del contenedor expone su puerto 3306 en el puerto 33306 de la máquina física. Observa que no usamos el mismo puerto por si en la máquina física ya estaba corriendo este servicio.
- **adminer**: Es un servicio basado en la imagen del mismo nombre y que, por defecto, se conecta al servidor *db* para que podamos interactuar sencillamente y sin instalar nada adicional con MySQL.
-
- un servidor Adminer (sobre Tomcat) que no es más que una interfaz Web para poder interactuar con MySQL de manera gráfica. Para abrir el contendor de adminer en un navegador hemos de abrir la dirección http://127.0.0.1:8181. De nuevo observa cómo no usamos el puerto por defecto 8080 por ser esta una dirección muy usada en otros servicios y para evitar conflictos con los mismos cuando usamos la máquina para muchos proyectos.

Para crear y levantar el servicio la primera vez:

```
docker-compose -f docker-compose.yml up -d
```

Ya puedes entrar en el servidor http://127.0.0.1:8181 y conectarte al servidor **db-inventario** con el usuario **root** y la contraseña **zx76wbz7FG89k** que hemos puesto en el fichero YAML.

Recuerda que es importante incluso en desarrollo no usar jamás contraseñas por defecto.

Si en algún momento quieres borrar los contenedores puedes hacerlo con este comando, donde *stack* debe ser sustituido por el nombre de la carpeta donde estaba el fichero *docker-compose.yml* y que dará nombre a los contenedores creados:

```
docker stack_adminer_1 stack_db_1
docker rm stack_adminer_1 stack_db_1
docker volume prune
docker rmi adminer mysql
```

Creación de la Base de Datos y las tablas

Primero creamos la base de datos, con el siguiente comando SQL:

```
DROP DATABASE `inventario`;

```

```
3 CREATE DATABASE `inventario`;
```

¿Por qué piensas que borramos la base de datos entera? Porque al crear las tablas usaremos AUTO_INCREMENT en los identificadores, que son contadores que nunca se resetean salvo que borremos la base de datos completamente, es más incluso incrementean cuando intentamos introducir algún dato erróneo (por ejemplo que viole una restricción *unique* o una *foreign key*), estos *auto_increment* incrementan a pesar de que no se pudo introducir o crear el dato.

```
1  -- usuario
2
3  CREATE TABLE `usuario` (
4      `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
5      `username` varchar(12) UNIQUE NOT NULL,
6      `password` varchar(20) NOT NULL,
7      `tipo` ENUM('admin', 'usuario', 'operario'),
8      `email` varchar(50) NOT NULL ) ENGINE='InnoDB';
9
10 -- estancia
11
12 CREATE TABLE `estancia` (
13     `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
14     `nombre` varchar(25) UNIQUE NOT NULL,
15     `descripcion` varchar(100) NOT NULL
16 ) ENGINE='InnoDB';
17
18 -- inventario
19
20 CREATE TABLE `inventario` (
21     `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
22     `nombre` varchar(25) UNIQUE NOT NULL,
23     `descripcion` varchar(100) NOT NULL,
24     `estancia` INT,
25     Foreign Key (`estancia`) REFERENCES `estancia`(`id`)
26 ) ENGINE='InnoDB';
27
28 -- incidencias
29
30 CREATE TABLE `incidencia` (
31     `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
32     `asunto` varchar(25) UNIQUE NOT NULL,
33     `descripcion` varchar(100) NOT NULL,
34     `inventario` int,
35     `operario` int,
36     `usuario` int,
37     `estado` ENUM('abierta', 'en proceso', 'cerrada', 'pendiente
           externo'),
38     Foreign Key (inventario) REFERENCES inventario(id),
39     Foreign Key (operario) REFERENCES usuario(id),
40     Foreign Key (usuario) REFERENCES usuario(id)
41 ) ENGINE='InnoDB';
```

A continuación te recordamos algunas de las palabras reservadas de SQL para MySQL Server que hemos usado:

- **int**: Tipo básico entero de 4 bytes con signo.
- **VARCHAR**: Cadena de caracteres de longitud máxima 65535. Entre paréntesis la longitud máxima permitida para esa columna.
- **ENUM**: Es muy parecido al *Enum* de Java. Objeto de tipo cadena de caracteres donde sólo se pueden dar unos valores concretos que se indican.
- **AUTO_INCREMENT**: Como ya hemos hablado antes, se trata de un contador.
- **NOT NULL**: Aplicado a una columna implica que no puede ser nulo (NULL)
- **PRIMARY KEY**: Al final de la declaración de una columna significa que es la llave principal.
- **Foreign Key (usuario) REFERENCES usuario(id)**: Significa que la columna usuario (en la tabla anterior se trata de una columna de tipo *int*) toma su valor de los valores posibles de **id** en la tabla **usuario**.
- **ENGINE='InnoDB'**: Tipo de motor de almacenamiento. En MySQL podemos elegir entre
  - **MyISAM**: Recomendable para aplicaciones en las que dominan las sentencias SELECT ante los INSERT/UPDATE. Mayor velocidad en general a la hora de recuperar datos.
  - **InnoDB**: Permite tener las características ACID (Atomicity, Consistency, Isolation and Durability), garantizando la integridad de nuestras tablas. Tiene soporte de transacciones y bloqueo de registros.

Configurando VS Code para acceder a MySQL

Antes de comenzar debemos crear la base de datos desde Adminer:

Creamos la base de datos "inventario":

Buscamos en las extensiones las siguientes y las instalamos:

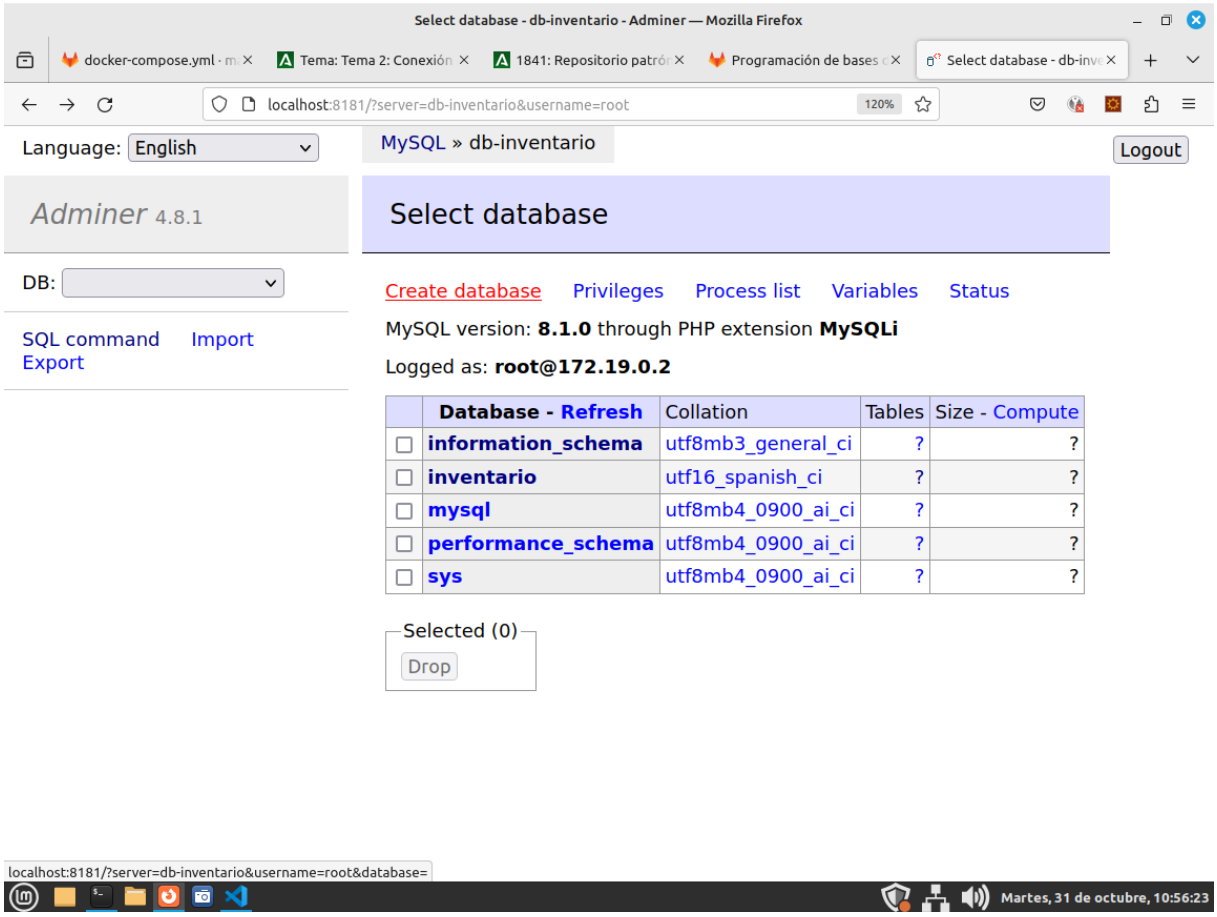


Figura 1: Boton de create database en la página de inicio de Adminer

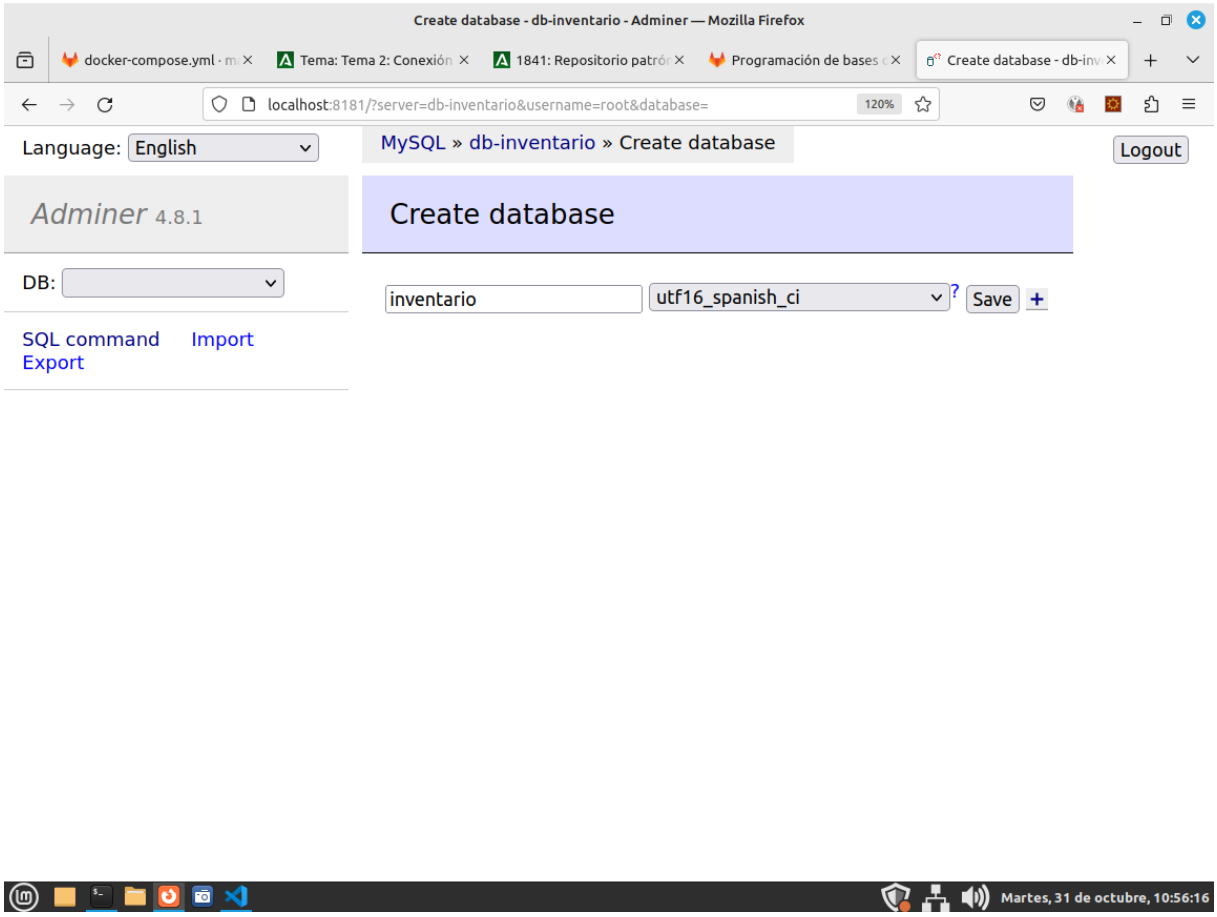
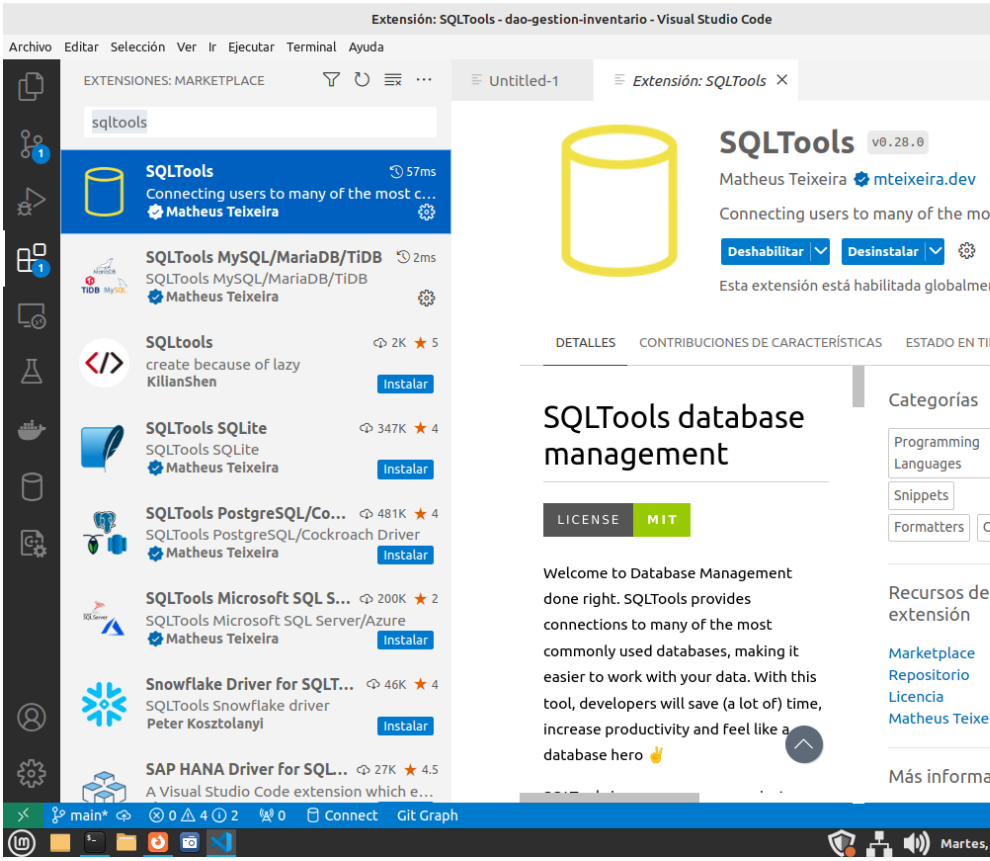
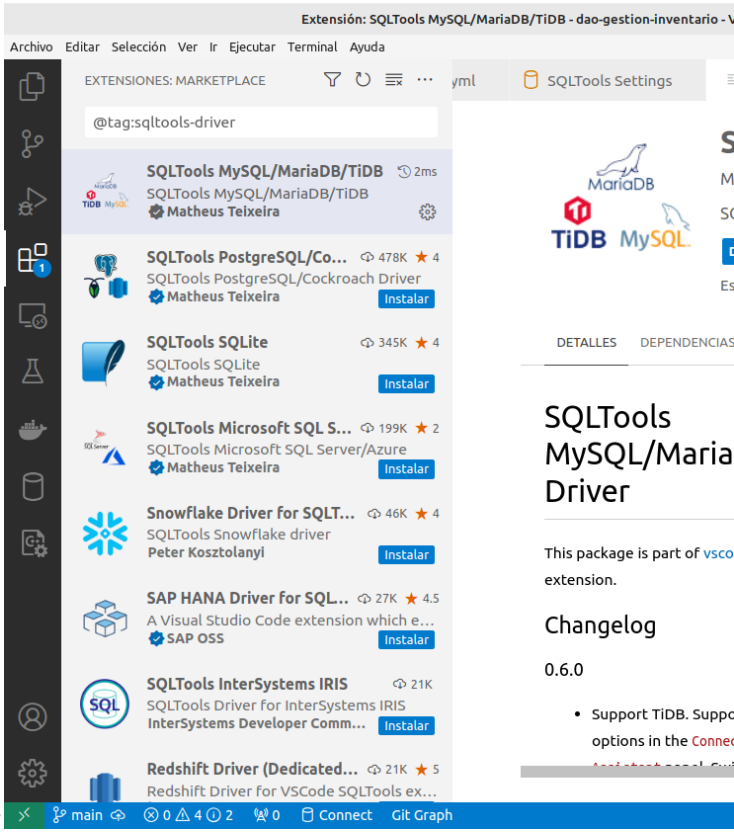


Figura 2: Creación de inventario database

- **sqltools** de Matheus Teixeira





- **SQLTools MySQL** o bien por **@tag:sqltools-driver**

Creamos una nueva conexión con esta información:

Clave	valor
Connection name*	Inventario
Connection group	(vacío)
Connect using*	(vacío)
Server and Port	
Server Address*	localhost
Port*	33306
Database*	inventario
Username*	root
Password mode	Save as plaintext in settings
Password*	zx76wbz7FG89k

Debería verse así:

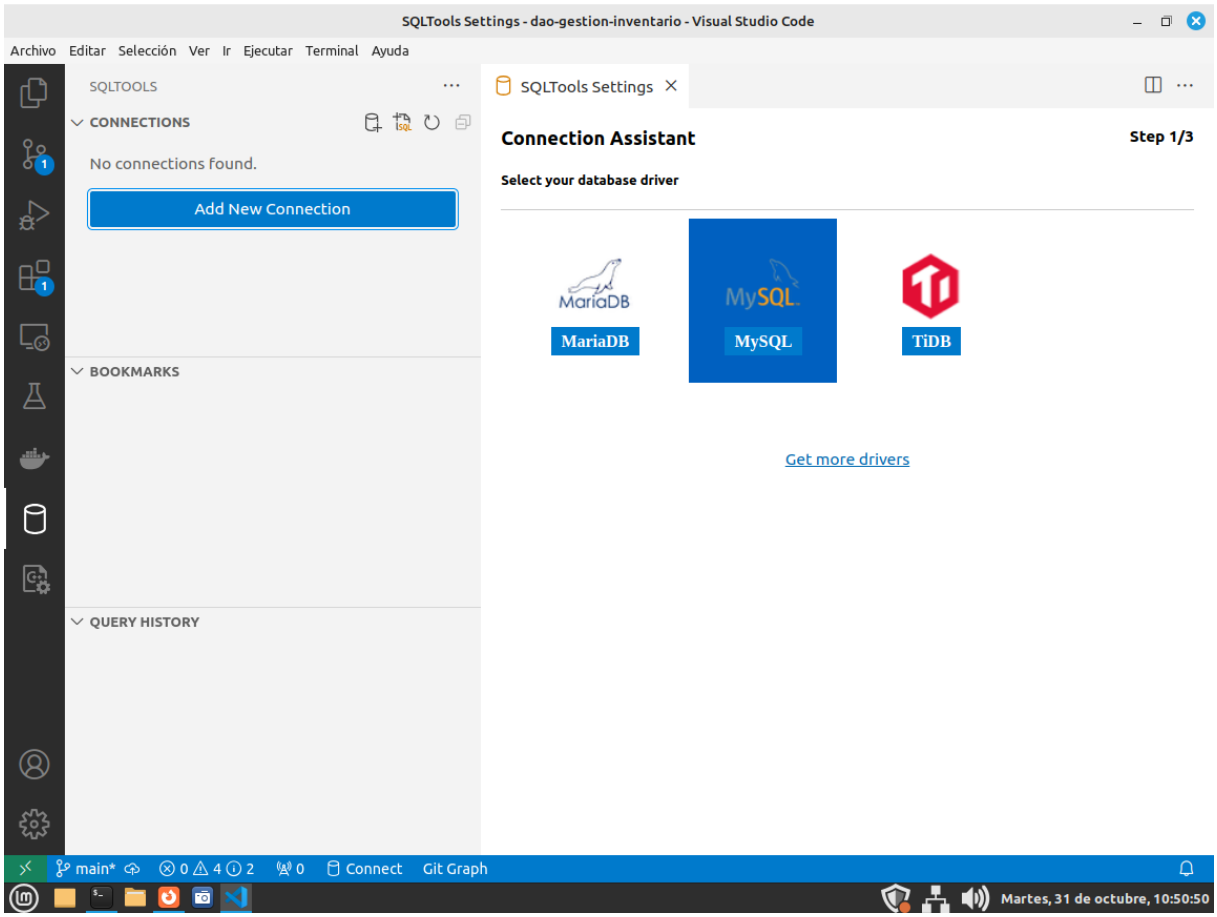


Figura 3: Paso 1

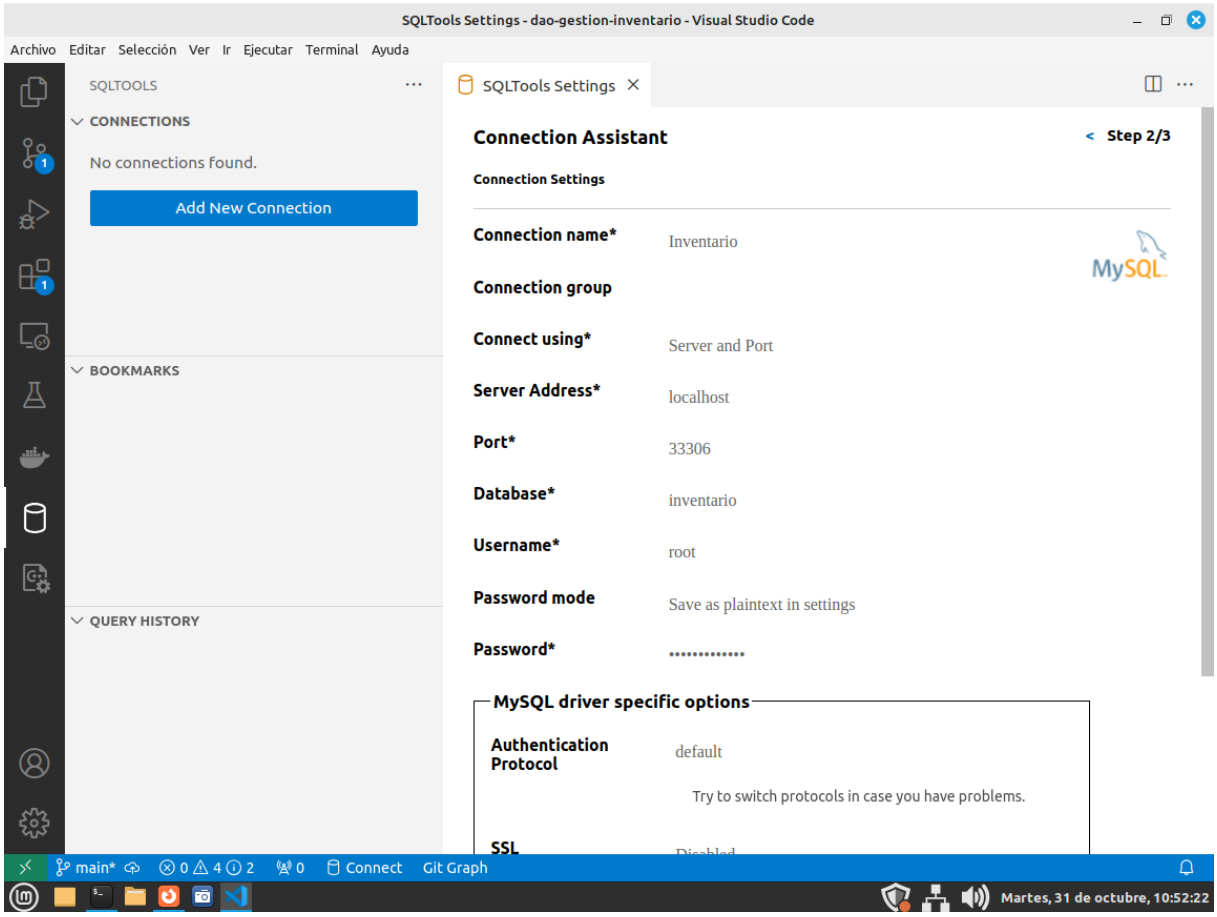


Figura 4: Paso 2

Conexión a la base de datos

Creando los Plain Old Java Objects (POJO)

Para poder hacer el marshalling/unmarshalling de objetos, te recomendamos primero tener los objetos en Java, es lo que llamamos el modelo de datos, en algunos sitios llamados clases entidad y en Spring Java POJOs (Plain Old Java Object) .

A tal efecto creamos clases sencillas como Usuario, Instalacion, Reserva, etc. con sus correspondientes constructores, getters y setters.

Usuario.java (ten cuidado con los métodos *equals* para comparar dos usuarios):

```java
package com.iesvdc.acceso.simplecrud.model;

public class Usuario {

    String username;
    String email;
    String password;
    Integer id;

    Usuario() {

    }

    Usuario(String username, String email, String password){
        this.username = username;
        this.password = password;
        this.email = email;
    }

    Usuario(Integer id, String username, String email, String password)
        {
        this.id = id;
        this.username = username;
        this.password = password;
        this.email = email;
    }


    public String getUsername() {
        return this.username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

```

```
36      public String getEmail() {
37          return this.email;
38      }
39
40      public void setEmail(String email) {
41          this.email = email;
42      }
43
44      public String getPassword() {
45          return this.password;
46      }
47
48      public void setPassword(String password) {
49          this.password = password;
50      }
51
52      public Integer getId() {
53          return this.id;
54      }
55
56      public void setId(Integer id) {
57          this.id = id;
58      }
59
60      @Override
61      public String toString() {
62          return "{" +
63              " username='" + getUsername() + "'" +
64              ", email='" + getEmail() + "'" +
65              ", password='" + getPassword() + "'" +
66              ", id='" + getId() + "'" +
67              "}";
68      }
69
70      @Override
71      public boolean equals(Object o) {
72          if (o == this)
73              return true;
74          if (!(o instanceof Usuario)) {
75              return false;
76          }
77
78          Usuario u = (Usuario) o;
79
80          return u.getId() == this.id &&
81              u.getUsername().compareTo(this.username)==0 &&
82              u.getPassword().compareTo(this.password)==0 &&
83              u.getEmail().compareTo(this.email)==0;
84      }
85  }
```

Instalación (ten cuidado con los métodos *equals*):

```java
package com.iesvdc.acceso.simplecrud.model;

public class Instalacion {

    private int id;
    private String name;

    public Instalacion(){}

    // GETTERS AND SETTERS

    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Instalacion)) {
            return false;
        }
        Instalacion instalacion = (Instalacion) o;

        return instalacion.id == this.id &&
            instalacion.name.compareTo(this.name)==0;
    }

    @Override
    public String toString() {
        return "{" +
            " id='" + getId() + "'" +
            ", name='" + getName() + "'" +
            "}";
    }


}
```

Horario (ten cuidado con los métodos *equals* para comparaciones):

```java
package com.iesvdc.acceso.simplecrud.model;

import java.time.LocalTime;

public class Horario {
    Instalacion instalacion;
    LocalTime    inicio;
    LocalTime    fin;
    Long id;

    public Horario() {
    }
```

Reserva (ten cuidado con los métodos *equals*):

```java
package com.iesvdc.acceso.simplecrud.model;

import java.sql.Date;

public class Reserva {
    Long id;
    Usuario usuario;
    Horario horario;
    Date fecha;


    public Reserva() {
    }

    public Reserva(Usuario usuario, Horario horario, Date fecha) {
        this.usuario = usuario;
        this.horario = horario;
        this.fecha = fecha;
    }

    public Reserva(Long id, Usuario usuario, Horario horario, Date
        fecha) {
        this.id = id;
        this.usuario = usuario;
        this.horario = horario;
        this.fecha = fecha;
    }

    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Usuario getUsuario() {
        return this.usuario;
    }

    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
    }

    public Horario getHorario() {
        return this.horario;
    }

    public void setHorario(Horario horario) {
```

```
49          this.horario = horario;
50      }
51
52      public Date getFecha() {
53          return this.fecha;
54      }
55
56      public void setFecha(Date fecha) {
57          this.fecha = fecha;
58      }
59
60      public Reserva id(Long id) {
61          this.id = id;
62          return this;
63      }
64
65      public Reserva usuario(Usuario usuario) {
66          this.usuario = usuario;
67          return this;
68      }
69
70      public Reserva horario(Horario horario) {
71          this.horario = horario;
72          return this;
73      }
74
75      public Reserva fecha(Date fecha) {
76          this.fecha = fecha;
77          return this;
78      }
79
80      @Override
81      public boolean equals(Object o) {
82          if (o == this)
83              return true;
84          if (!(o instanceof Reserva)) {
85              return false;
86          }
87          Reserva reserva = (Reserva) o;
88          return usuario.equals(reserva.usuario) &&
89              horario.equals(reserva.horario) &&
90              fecha.compareTo(reserva.fecha)!=0;
91      }
92
93
94
95      @Override
96      public String toString() {
97          return "{" +
98              " id='" + getId() + "'" +
99              " usuario='" + this.getUsuario().toString() + "'" +
```

```
100                 ", horario='" + this.getHorario().toString() + "'" +
101                 ", fecha='" + getFecha().toString() + "'" +
102                 "}";
103         }
104
105
106 }
```

Ahora ya sería posible con Gson o JAXB por ejemplo, desde el servlet directamente hacer el marshalling/unmarshalling de JSON a Java. No obstante no recomendamos esta opción porque no responde a ningún estándar como sí lo hace por ejemplo Jersey (JAX-RS 2.1, que es la especificación de la JSR 370: JavaTM API for RESTful Web Services).

Conectando a la base de datos cargando la configuración vía JNDI

En un entorno genérico, si por ejemplo queremos usar el patrón singleton y usar un único objeto conexión en nuestro código, podríamos hacer lo siguiente:

Abrir la conexión:

```
1  public class Conexion {
2      Connection conn;
3      public Conexion(){
4          if (conn==null)
5              try {
6                  Class.forName("com.mysql.jdbc.Driver");
7                  this.conn =
8                  DriverManager.getConnection("jdbc:mysql://localhost/
                       gestion_reservas?"+
9                  "useUnicode=true&useJDBCCompliantTimezoneShift=true&
                       serverTimezone=UTC"+
10                 "&user=root&password=example");
11             } catch (SQLException | ClassNotFoundException ex) {
12                 Logger.getLogger(Conexion.class.getName()).log(Level.
                       SEVERE, null, ex);
13             }
14     }
15     public Connection getConnection() {
16         return conn;
17     }
18 }
```

Para cargar la conexión por JNDI (Java Naming and Directory Interface), es decir pedimos a Java que localice, en el contexto actual, un objeto que será la información de la conexión a la base de datos, lo haríamos así:

```
1  import java.sql.Connection;
```

```java
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

/**
 *
 * @author juangu
 */
public class Conexion {

    Connection conn;
    Context ctx;
    DataSource ds;

    public Conexion() {
        // Vía DataSource con Contexto inyectado
        try {
            if (ctx == null)
                ctx = new InitialContext();
            if (ds == null)
                ds = (DataSource) ((Context) ctx.lookup(
                    "java:comp/env")).lookup("jdbc/gestionReservas");
            conn = ds.getConnection();
        } catch (Exception ex) {
            System.out.println("## Conexion ERROR ## " + ex.
               getLocalizedMessage());
            ctx = null;
            ds = null;
            conn = null;
        }
    }

    public Connection getConnection() {
        return conn;
    }
}
```

Esto implica que en el directorio *src/main/webapp/META-INF* de nuestro proyecto Maven, deberíamos tener un fichero llamado **context.xml** con el siguiente contenido:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/">
    <Resource name="jdbc/gestionReservas" global="jdbc/gestionReservas"
              url="jdbc:mysql://localhost:33306/gestion_reservas"
              auth="Container" type="javax.sql.DataSource"
              maxTotal="100" maxIdle="30" maxWaitMillis="10000"
              username="root" password="example" driverClassName="com.
                 mysql.cj.jdbc.Driver"
    />
</Context>
```

En este archivo podemos ver que el recurso será accesible vía JDNI bajo el nombre "jdbc/gestionReservas", que es como tener un objeto en memoria con las propiedades siguientes:

```
gestionReservas: {
   url:"jdbc:mysql://localhost:33306/gestion_reservas",
   auth:"Container",
   type:"javax.sql.DataSource",
   maxTotal:"100",
   maxIdle:"30",
   maxWaitMillis:"10000",
   username:"root",
   password:"example",
   driverClassName:"com.mysql.cj.jdbc.Driver"
}
```

Cuando inicializar y cerrar la conexión desde un servlet

Cuando hemos de tomar la decisión de cuando conectar a la base de datos desde un servlet, nos encontramos frente cuatro opciones:

1. **Conexión por transacción**: dentro de cada método doGet, doPost, doPut o doDelete abrimos y cerramos la conexión. En el método init() del servlet cargamos el driver JDBC correspondiente (Oracle, MySQL, PostgreSQL, etc.). Éste es el modelo que seguiremos en los servicios REST, no es el más óptimo pero sí es seguro. Además cuando saltemos a JPA con Hibernate en Spring, nos resultará más fácil el cambio.
2. **Conexión dedicada**: al crear el servlet abrimos la conexión (método init() del mismo) y se cierra al descargar el servlet (método destroy()). Por tanto el driver y la conexión se cargan en el método init().
3. **Conexión por sesión**: cargamos el driver JDBC necesario en el método init(). No abrimos la conexión hasta el primer do(Get|Put|Delete|Post). En la sesión de usuario vamos pasando la conexión abierta de unos métodos a otros.
4. **Conexión cacheada**: Con un "pool" de conexiones. El servidor de aplicaciones (Tomcat, Jetty, Glashfish…) es el encargado de cargar el driver y abrir la conexión la primera vez que se necesita y ofrecerla a cada servlet que la necesita. Hay que crear el "pool" de conexiones en el servidor, lo que implica que tenemos acceso a su administración. Si compartimos el servidor o bien conectamos a diferentes bases de datos, puede ser peligroso tocar las configuraciones porque podemos dejar al resto de aplicaciones del servidor sin conexión a su base de datos.

Aunque lo normal es delegar en el servidor de aplicaciones la gestión de conexiones al servidor de base de datos, una receta muy común es abrir y cerrar la conexión desde los métodos init() y destroy() de

los mismos. Esto es lo que se llama una **conexión dedicada**. Veámoslo en los siguientes ejemplos.

Conexión dedicada

Abrir la conexión desde el método init()

```
public class Alumno extends HttpServlet {

    Connection conn;

    @Override
    public void init() throws ServletException {
        Conexion conexion = new Conexion();
        this.conn = conexion.getConnection();
    }
```

Cerrar la conexión desde el método destroy()

```
@Override
    public void destroy() {
        try {
            this.conn.close();
        } catch (SQLException ex) {

        }
    }
```

Conectando a la Base de Datos

Abrir la conexión:

```
public class Conexion {
    Connection conn;
    public Conexion(){
        if (conn==null)
            try {
                Class.forName("com.mysql.jdbc.Driver");
                conn =
                DriverManager.getConnection("jdbc:mysql://localhost/
                    gestion_reservas?" +
                "useUnicode=true&useJDBCCompliantTimezoneShift=true&
                    serverTimezone=UTC"+
                "&user=root&password=example");
            } catch (SQLException | ClassNotFoundException ex) {
                Logger.getLogger(Conexion.class.getName()).log(Level.
                    SEVERE, null, ex);
            }
    }
    public Connection getConnection() {
```

```
16          return conn;
17      }
18  }
```

CRUD (patrón DAO)

Ya tenemos las clases base que contiene los objetos que vamos a almacenar/recuperar de la base de datos, ahora hay que hacer las interfaces para los DAO de cada uno y luego su implementación, vemos sólo el ejemplo de Instalación:

InstalacionDao.java

```
1  package com.iesvdc.acceso.simplecrud.dao;
2
3  import java.util.List;
4
5  import com.iesvdc.acceso.simplecrud.model.Instalacion;
6
7  public interface InstalacionDao {
8      public boolean create(Instalacion instala);
9      public Instalacion findById(Integer id);
10     public List<Instalacion> findAll();
11     public List<Instalacion> findByNombre(String nombre);
12     public boolean update(Instalacion old_al, Instalacion new_al);
13     public boolean update(Integer old_id, Instalacion new_al);
14     public boolean delete(Instalacion instala);
15     public boolean delete(Integer id_al);
16 }
```

Leer todos/ uno (InstalacionDaoImpl.java)

```
1      @Override
2      public Instalacion findById(Integer id) {
3          Instalacion instala;
4          try {
5              Conexion conexion = new Conexion();
6              Connection conn = conexion.getConnection();
7              String sql = "SELECT * FROM instalacion WHERE id=?";
8              PreparedStatement pstmt = conn.prepareStatement(sql);
9              pstmt.setInt(1, id);
10             System.err.println("\nID:: " + id + "\n");
11             ResultSet rs = pstmt.executeQuery();
12             rs.next();
```

```
13              instala = new Instalacion(rs.getInt("id"), rs.getString("
                    nombre"));
14              conn.close();
15          } catch (SQLException ex) {
16              instala = null;
17          }
18          return instala;
19      }
20
21      @Override
22      public List<Instalacion> findAll() {
23          Instalacion instala;
24          List<Instalacion> li_ins = new ArrayList<Instalacion>();
25          try {
26              Conexion conexion = new Conexion();
27              Connection conn = conexion.getConnection();
28
29              String sql = "SELECT * FROM instalacion";
30
31              PreparedStatement pstmt = conn.prepareStatement(sql);
32
33              ResultSet rs = pstmt.executeQuery();
34              // recorro el resultset mientras tengo datos
35              while (rs.next()) {
36                  instala = new Instalacion(rs.getInt("id"), rs.getString
                        ("nombre"));
37                  li_ins.add(instala);
38              }
39              // cerramos la conexión
40              conn.close();
41          } catch (SQLException ex) {
42              System.out.println("ERROR" + ex.getMessage());
43              li_ins = null;
44          }
45          return li_ins;
46      }
```

Crear (InstalacionDaoImpl.java)

```
1       @Override
2       public boolean create(Instalacion instala) {
3           boolean exito = true;
4           try {
5               Conexion conexion = new Conexion();
6               Connection conn = conexion.getConnection();
7               String sql = "INSERT INTO instalacion VALUES (NULL,?,?)";
8               PreparedStatement pstmt = conn.prepareStatement(sql);
9               pstmt.setInt(1, instala.getId());
10              pstmt.setString(2, instala.getName());
```

```
11              pstmt.executeUpdate();
12              conn.close();
13          } catch (SQLException ex) {
14              System.out.println("ERROR:  " + ex.getMessage());
15              exito = false;
16          }
17          return exito;
18      }
```

Actualizar (InstalacionDaoImpl.java)

```
1       /**
2        * Este método actualiza uninstalaumno en la BBDD
3        *
4        * @param old_al El objeto que contiene los datos antiguos
             delinstalaumno
5        * @param new_al El objeto que contiene los datos nuevos
             delinstalaumno
6        * @return true si se lleva a cabo correctamente <br>
7        *         false si no se actualiza nada (error de conexión, no
             estaba
8        *         elinstalaumno en la BBDD...) <br>
9        */
10      @Override
11      public boolean update(Instalacion old_al, Instalacion new_al) {
12
13          return update(old_al.getId(), new_al);
14      }
15
16      /**
17       * Este método actualiza una instalación en la BBDD
18       *
19       * @param old_id El id antiguo delinstalaumno
20       * @param new_al El objeto que contieneinstalainstalaumno
             actualizado
21       * @return true si se lleva a cabo correctamente <br>
22       *         false si no se actualiza nada (error de conexión, no
             estaba
23       *         elinstalaumno en la BBDD...) <br>
24       */
25      @Override
26      public boolean update(Integer old_id, Instalacion new_al) {
27          boolean exito = true;
28          try {
29              Conexion conexion = new Conexion();
30              Connection conn = conexion.getConnection();
31              String sql = "UPDATE instalacion SET id=?, nombre=? WHERE
                    id=?";
32              PreparedStatement pstmt = conn.prepareStatement(sql);
```

```
33              pstmt.setInt(3, old_id);
34              pstmt.setInt(1, new_al.getId());
35              pstmt.setString(2, new_al.getName());
36              if (pstmt.executeUpdate() == 0) {
37                  exito = false;
38              }
39              conn.close();
40          } catch (SQLException ex) {
41              exito = false;
42          }
43          return exito;
44      }
```

Borrar (InstalacionDaoImpl.java)

```
1       /**
2        * Este método borra de la BBDD el Alumno cuyos datos coinciden con
             los de el
3        * objeto que se le pasa como parámetro
4        *
5        * @paraminstalainstalaumno a borrar
6        * @return true si borra uninstalaumno <br>
7        *         false si elinstalaumno no existe o no se puede conectar
             a la BBDD
8        *         <br>
9        */
10      @Override
11      public boolean delete(Instalacion instala) {
12          return delete(instala.getId());
13      }
14
15      @Override
16      public boolean delete(Integer id_al) {
17          boolean exito = true;
18          try {
19              Conexion conexion = new Conexion();
20              Connection conn = conexion.getConnection();
21              String sql = "DELETE FROM instalacion WHERE id=?";
22              PreparedStatement pstmt = conn.prepareStatement(sql);
23              pstmt.setInt(1, id_al);
24              if (pstmt.executeUpdate() == 0) {
25                  exito = false;
26              }
27              conn.close();
28          } catch (SQLException ex) {
29              exito = false;
30          }
31          return exito;
32      }
```

Introducción a Docker

Docker es un sistema de gestión de contenedores, para automatizar el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

En vez de usar máquinas virtuales (más pesadas), docker usa el espacio de Kernel de Linux y comparte partes con él, aislando y simplificando las aplicaciones.

Si usamos docker en una máquina con MAC OS o Microsoft Windows, será necesario tener una máquina virtual Linux que será la que nos ofrezca el soporte Linux. Así las últimas versiones de Windows incorporan el WSL (Windows Subsystem for Linux) por ejemplo, con las que se hace mucho más sencillo utilizar Docker.

Comandos útiles de docker

Listado de contenedores:

```
$ docker ps
```

Parar un contenedor:

```
$ docker stop [contenedor]
```

Borrar un contenedor:

```
$ docker rm [contenedor]
```

Listado de imágenes:

```
$ docker image ls
```

Borrar una imagen:

```
$ docker rmi [imagen]
```

Listado de volúmenes:

```
$ docker volume ls
```

Borrar todos los volúmenes que no estén usados por ningún contenedor:

```
$ docker volume prune
```

Borrar un volumen concreto:

```
$ docker volume rm [volumen]
```

Borrar todo lo relacionado con un docker-compose.yml

```
$ docker-compose -f file down
```

Instalación de Docker

Para instalar en sistemas Linux basados en Debian:

$ sudo apt update $ sudo apt install docker.io docker-compose

Para que el usuario local, por ejemplo, matinal o tarde puedan usar docker, crear contenedores, manipularlos, etc. deberemos añadir nuestros usuarios al grupo docker con el comando:

> $ sudo nano /etc/group

ó

> $ sudo vim /etc/group

modificamos la última línea:

> docker:x:131:matinal

Ahora cerramos sesión y volvemos a entrar para que coja los cambios.

Para asegurar que está todo andando, reiniciamos el servicio:

> sudo systemctl restart docker

Cómo crear un contenedor desde la imagen de Ubuntu

> $ git pull ubuntu $ docker run –name lamp-server-plantilla -it ubuntu:latest bash

Ahora nos abre una terminal dentro del nuevo contenedor:

```
# apt update
# apt install net-tools
# apt install apache2
# apt install mysql-server
# apt install php
# apt-get install libapache2-mod-php php-mysql
```

Copias de los contenedores

En este capítulo vamos a aprender cómo llevarnos los datos de nuestros contenedores sin tener que estar guardando imágenes completas.

La idea es hacer copias de seguridad solamente de los volúmenes de datos. https://docs.docker.com/storage/volumes/

Como es un poco lioso se puede usar el contenedor (loomchild/volume-backup) que contiene un script que facilita la creación/restauración de volúmenes.

https://github.com/loomchild/volume-backup

https://hub.docker.com/r/loomchild/volume-backup

Por ejemplo, joomla https://hub.docker.com/r/bitnami/joomla/

Se crean dos contenedores con dos volúmenes (uno para webserver y otro mariadb):

> $ mkdir joomla-docker $ cd joomla-docker $ curl -sSL https://raw.githubusercontent.com/bitnami/bitnami-docker-joomla/master/docker-compose.yml > docker-compose.yml $ docker-compose up -d $ cat docker-compose.yml

```yaml
version: '2'
services:
  mariadb:
    image: 'bitnami/mariadb:latest'
    environment:
      - MARIADB_USER=bn_joomla
      - MARIADB_DATABASE=bitnami_joomla
      - ALLOW_EMPTY_PASSWORD=yes
    volumes:
      - 'mariadb_data:/bitnami'
  joomla:
    image: 'bitnami/joomla:latest'
    environment:
      - MARIADB_HOST=mariadb
      - MARIADB_PORT_NUMBER=3306
      - JOOMLA_DATABASE_USER=bn_joomla
      - JOOMLA_DATABASE_NAME=bitnami_joomla
      - ALLOW_EMPTY_PASSWORD=yes
    labels:
      kompose.service.type: nodeport
    ports:
      - '80:80'
      - '443:443'
    volumes:
      - 'joomla_data:/bitnami'
    depends_on:
      - mariadb
volumes:
```

```
29    mariadb_data:
30      driver: local
31    joomla_data:
32      driver: local
```

Se han creado los dos volúmenes (se han concatenado a los nombres de volumen el directorio desde donde se ha ejecutado):

> $ docker volume ls DRIVER VOLUME NAME local joomla-docker_joomla_data local joomla-docker_mariadb_data

IMPORTANTE: Antes de hacer backup hay que parar los contenedores

> $ docker-compose stop

Backup volúmenes

> $ docker run -v joomla-docker_joomla_data:/volume –rm loomchild/volume-backup backup - > joomla-docker_joomla_data.tar.bz2 $ docker run -v joomla-docker_mariadb_data:/volume –rm loomchild/volume-backup backup - > joomla-docker_mariadb_data.bz2 $ ls joomla-docker_joomla_data.tar.bz2 joomla-docker_mariadb_data.bz2

Para hacer la prueba, he eliminado los volúmenes para luego restaurarlos desde los ficheros.

> $ docker volume rm joomla-docker_joomla_data joomla-docker_mariadb_data

Restaurar volúmenes:

> $ cat joomla-docker_joomla_data.tar.bz2 | docker run -i -v joomla-docker_joomla_data:/volume –rm loomchild/volume-backup restore - $ cat joomla-docker_mariadb_data.bz2 | docker run -i -v joomla-docker_mariadb_data:/volume –rm loomchild/volume-backup restore -

Se vuelven a iniciar los contenedores:

$ docker-compose up -d

Apache2 como contenedor

Instalación del servidor

A continuación vamos a ver los pasos a seguir para crear el contenedor (Docker) a partir de una imagen personalizada.

Pimero preparamos los archivos de la Web que contendrá el servidor:

$ mkdir public-html

$ cat > ./public-html/index.html

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8" />
    <title>Apache2</title>
  </head>
  <body>
    <h1>Hola Mundo</h1>
  </body>
</html>
```

Ahora creamos el fichero “Dockerfile” con este contenido:

cat > Dockerfile

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/ (CTRL+C)
```

El siguiente paso es generar la imagen que nos servirá de plantilla para nuestros contenedores:

$ docker build -t mi-imagen-apache2 .

Ahora podemos crear un contenedor “vivo”, lo creamos y lanzamos con:

$ docker run -dit –name mi-container-apache -p 8080:80 mi-imagen-apache2

Configuración

El servidor escuchará el puerto 8008, NO eliminar el puerto 80.

Rehacer la imagen (cada vez que hagamos cambios):

$ docker build -t mi-imagen-apache2 .

Parar el contenedor (para poder borrarlo):

$ docker stop mi-container-apache

Borrar el contenedor:

$ docker rm mi-container-apache

Crear de nuevo el contenedor y lanzarlo:

$ docker run -dit –name mi-container-apache -p 8080:80 mi-imagen-apache2

Copiar el archivo de configuración del contenedor a la máquina física (se hace en la misma carpeta donde está el Dockerfile):

$ docker cp mi-container-apache:/usr/local/apache2/conf/httpd.conf .

Modificamos el Dockerfile:

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
COPY ./httpd.conf /usr/local/apache2/conf/
```

Creación del fichero htpasswd

En el directorio de trabajo (donde está el Dockerfile) ejecuto:

$ htpasswd -c .htpasswd usuario

Si no lo tengo instalado:

$ sudo apt install apache2-utils

Añadir el .htpassword a la imagen

Modifico el Dockerfile para que quede así:

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
COPY ./httpd.conf /usr/local/apache2/conf/
COPY .htpasswd /usr/local/apache2/conf/
```

Añadimos la autenticación a Apache

```
<Location "/dos">
    AuthType basic
    AuthName "DOS: Autenticación Básica"
    AuthUserFile /usr/local/apache2/conf/.htpasswd
    Require valid-user
</Location>
```

Creo un script para automatizar la actualización de los contenedores

(fichero reload.sh)

```
#!/bin/bash
echo "INICIO"
echo "Parando el contenedor: "
docker stop mi-container-apache
echo "Eliminando el contenedor: "
docker rm mi-container-apache
echo "Generando la nueva imagen: "
docker build -t mi-imagen-apache2 .
echo "Creando el contenedor y poniéndolo en marcha"
docker run -dit --name mi-container-apache -p 8080:80 mi-imagen-apache2
echo "TERMINADO!!!"
```bash
```
```

```
13
14 Autenticación DIGEST
15
16 Primero: creamos el archivo de contraseñas:
17
18 htdigest -c .htdigest restringido pepe
19
20 Segundo: modificamos el Dokcerfile:
21
22 ```Dockerfile
23 FROM httpd:2.4
24 COPY ./public-html/ /usr/local/apache2/htdocs/
25 COPY ./httpd.conf /usr/local/apache2/conf/
26 COPY .htpasswd /usr/local/apache2/conf/
27 COPY .htdigest /usr/local/apache2/conf/
```

Tercero: Modifcamos el httpd.conf añadiendo al final:

```
1 <Location "/tres">
2     AuthType digest
3     AuthName "restringido"
4     AuthDigestProvider file
5     AuthUserFile /usr/local/apache2/conf/.htdigest
6     Require valid-user
7 </Location>
```

Añadiendo seguridad SSL al contenedor:

Modificamos el script de generación de imágenes como sigue:

```
1  #!/bin/bash
2  echo "INICIO"
3  echo "Parando el contenedor: "
4  docker stop mi-container-apache
5  echo "Eliminando el contenedor: "
6  docker rm mi-container-apache
7  echo "Eliminando certificados antiguos"
8  rm -fr ssl
9  echo "Creando certificados nuevos:"
10 mkdir ssl
11 openssl genrsa -out ssl/server.key 1024
12 openssl req -new -key ssl/server.key -out ssl/server.csr
13 openssl x509 -req -days 365 -in ssl/server.csr -signkey ssl/server.key
       -out ssl/server.crt
14 echo "Generando la nueva imagen: "
15 docker build -t mi-imagen-apache2 .
16 echo "Creando el contenedor y poniéndolo en marcha"
17 docker run -dit --name mi-container-apache -p 8080:80 mi-imagen-apache2
18 docker run -dit --name mi-container-apache mi-imagen-apache2
19 sleep 4
20 docker ps
```

```
21 echo "TERMINADO!!!"
```

Modificamos el Dockerfile:

```
1 FROM httpd:2.4
2 COPY ./public-html/ /usr/local/apache2/htdocs/
3 COPY ./httpd.conf /usr/local/apache2/conf/
4 COPY ./httpd-ssl.conf /usr/local/apache2/conf/extra/
5 COPY .htpasswd /usr/local/apache2/conf/
6 COPY .htdigest /usr/local/apache2/conf/
7 COPY ./ssl/ /usr/local/apache2/conf/
8 COPY ./extra/ /usr/local/apache2/conf/extra/
```

Modificamos el httpd.conf:

```
1  LoadModule ssl_module modules/mod_ssl.so
2
3  Listen 443
4  <VirtualHost *:443>
5      ServerName localhost
6      SSLEngine on
7      SSLCertificateFile "/usr/local/apache2/conf/server.crt"
8      SSLCertificateKeyFile "/usr/local/apache2/conf/server.key"
9  </VirtualHost>
10
11 #<IfModule ssl_module>
12 #SSLRandomSeed startup builtin
13 #SSLRandomSeed connect builtin
14 #</IfModule>
```

Ejecutamos el script “reload” y ¡listo!

Ejercicios propuestos

Gestión de reservas