

Programación de bases de datos relacionales

Juan Gualberto

Junio 2023

Índice

Gestión de Inventario	3
El desfase objeto-relacional	3
Gestores de bases de datos embebidos e independientes	6
Protocolos de acceso a bases de datos. Conectores.	7
Establecimiento de conexiones	7
Definición de objetos destinados al almacenamiento del resultado de operaciones con bases de datos. Eliminación de objetos finalizada su función.	9
Ejecución de sentencias de descripción de datos.	9
Ejecución de sentencias de modificación de datos. CRUD básico.	9
LEER uno (findOne)	9
LEER todos (findAll)	10
Crear	10
Actualizar	11
Borrar	12
Ejecución de consultas.	13
Utilización del resultado de una consulta.	13
Ejecución de procedimientos almacenados en la base de datos.	13
Gestión de transacciones.	13
Creación de la BBDD	13
Preparando los contenedores Docker para la base de datos	13
Creación de la Base de Datos y las tablas	15
Conexión a la base de datos	16
Creando los Plain Old Java Objects (POJO)	16
Conectando a la base de datos cargando la configuración vía JNDI	22
Cuando inicializar y cerrar la conexión desde un servlet	24
Conexión dedicada	24
Conectando a la Base de Datos	25
CRUD (patrón DAO)	25
InstalacionDao.java	26
Leer todos/ uno (InstalacionDaoImpl.java)	26
Crear (InstalacionDaoImpl.java)	27
Actualizar (InstalacionDaoImpl.java)	27
Borrar (InstalacionDaoImpl.java)	29

Introducción a Docker	29
Comandos útiles de docker	30
Instalación de Docker	30
Cómo crear un contenedor desde la imagen de Ubuntu	31
Copias de los contenedores	31
Apache2 como contenedor	33

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:

```
1 public class Producto {  
2     private int id;  
3     private String nombre;  
4     private double precio;  
5  
6     // Constructor, getters y setters  
7  
8     // Otros métodos de la clase  
9 }
```

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:

```
1 CREATE TABLE productos (  
2     id INT PRIMARY KEY,  
3     nombre VARCHAR(100),  
4     precio DECIMAL(10, 2)  
5 );
```

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

```
1 public void insertarProducto(Producto producto) {  
2     String sql = "INSERT INTO productos (id, nombre, precio) VALUES (?,  
3         ?, ?)";  
4     try (Connection conn = obtenerConexion();  
5         PreparedStatement stmt = conn.prepareStatement(sql)) {  
6  
7         stmt.setInt(1, producto.getId());  
8         stmt.setString(2, producto.getNombre());  
9         stmt.setDouble(3, producto.getPrecio());  
10  
11         stmt.executeUpdate();  
12     } catch (SQLException e) {  
13         // Manejo de excepciones  
14     }  
15 }
```

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:

```
1 public List<Producto> obtenerProductos() {  
2     List<Producto> productos = new ArrayList<>();  
3     String sql = "SELECT id, nombre, precio FROM productos";  
4  
5     try (Connection conn = obtenerConexion();  
6         Statement stmt = conn.createStatement();  
7         ResultSet rs = stmt.executeQuery(sql)) {  
8  
9         while (rs.next()) {  
10             int id = rs.getInt("id");  
11             String nombre = rs.getString("nombre");  
12             double precio = rs.getDouble("precio");  
13  
14             Producto producto = new Producto(id, nombre, precio);  
15             productos.add(producto);  
16         }  
17     }
```

```
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
- PostgreSQL
- 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 escritas 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      *
8      */
9     public Conexion() {
10         // Vía JDBC
11         if (conn == null) {
12             try (FileInputStream fis = new FileInputStream("db.
13                 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                 {
24                     Logger.getLogger(Conexion.class.getName()).severe(e.
25                         getLocalizedMessage());
26                 }
27             }
28         }
29     }
30
31     public Connection getConnection() {
32         return conn;
33     }
34
35     public void destroy() {
36         if (this.conn != null) {
37             try {
38                 this.conn.close();
39             } catch (SQLException e) {
40
41             }
42         }
43     }
44 }
```

```
40 }
```

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. Eliminación de objetos finalizada su función.

Ejecución de sentencias de descripción de datos.

Ejecución de sentencias de modificación de datos. CRUD básico.

LEER uno (findOne)

```
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)

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

```
1 Connection conexion;
2 PreparedStatement pstmt;
3 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     pstmt = conexion.prepareStatement(sql);
10    pstmt.setString(1, username);
11    pstmt.setString(2, password);
12    pstmt.setString(3, email);
13    if (pstmt.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 pstmt = conexion.prepareStatement(sql);
11      pstmt.setString(1, user.getUsername());
12      pstmt.setString(2, user.getPassword());
13      pstmt.setString(3, user.getEmail());
14      pstmt.setInt(4, user.getId());
15
16      if (pstmt.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

```
1      Connection conexion;
2      PreparedStatement pstmt;
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         pstmt = conexion.prepareStatement(sql);
11         pstmt.setInt(1, Integer.parseInt(id));
12         if ( pstmt.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.

Utilización del resultado de una consulta.

Ejecución de procedimientos almacenados en la base de datos.

Gestión de transacciones.

Creación de la BBDD

Preparando los contenedores Docker para la base de datos

Instalamos Docker si no estuviese:

```
1 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:

```
1 version: '3.1'
2
3 services:
4
5   db-inventario:
6     image: mysql
7     # 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
8     # (this is just an example, not intended to be a production
      configuration)
9     command: --default-authentication-plugin=mysql_native_password
10    restart: "no"
11    environment:
12      MYSQL_ROOT_PASSWORD: zx76wbz7FG89k
```

```
13     ports:
14         - 33306:3306
15
16     adminer:
17         image: adminer
18         restart: "no"
19         ports:
20             - 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 a 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 contenedor 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:

```
1 docker stack_adminer_1 stack_db_1
2 docker rm stack_adminer_1 stack_db_1
```

```
3 docker volume prune
4 docker rmi adminer mysql
```

Creación de la Base de Datos y las tablas

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

```
1 DROP DATABASE `inventario`;
2
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 incrementan 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.

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):

```
1 package com.iesvdc.acceso.simplecrud.model;
2
3 public class Usuario {
4
5     String username;
6     String email;
7     String password;
8     Integer id;
9
10    Usuario() {
11
12    }
13
14    Usuario(String username, String email, String password){
15        this.username = username;
16        this.password = password;
17        this.email = email;
18    }
19
20    Usuario(Integer id, String username, String email, String password)
21    {
22        this.id = id;
23        this.username = username;
24        this.password = password;
25        this.email = email;
26    }
27
28    public String getUsername() {
29        return this.username;
30    }
31
32    public void setUsername(String username) {
33        this.username = username;
34    }
35
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*):

```
1 package com.iesvdc.acceso.simplecrud.model;
2
3 public class Instalacion {
4
5     private int id;
6     private String name;
7
8     public Instalacion() {}
```

```
9
10 // GETTERS AND SETTERS
11
12 @Override
13 public boolean equals(Object o) {
14     if (o == this)
15         return true;
16     if (!(o instanceof Instalacion)) {
17         return false;
18     }
19     Instalacion instalacion = (Instalacion) o;
20
21     return instalacion.id == this.id &&
22         instalacion.name.compareTo(this.name)==0;
23 }
24
25 @Override
26 public String toString() {
27     return "{" +
28         " id='" + getId() + "'" +
29         ", name='" + getName() + "'" +
30         "}";
31 }
32
33
34 }
```

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

```
1 package com.iesvdc.acceso.simplecrud.model;
2
3 import java.time.LocalDateTime;
4
5 public class Horario {
6     Instalacion instalacion;
7     LocalDateTime inicio;
8     LocalDateTime fin;
9     Long id;
10
11     public Horario() {
12     }
```

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

```
1 package com.iesvdc.acceso.simplecrud.model;
2
3 import java.sql.Date;
4
5 public class Reserva {
6     Long id;
7     Usuario usuario;
```

```
8     Horario horario;  
9     Date fecha;  
10  
11  
12     public Reserva() {  
13     }  
14  
15     public Reserva(Usuario usuario, Horario horario, Date fecha) {  
16         this.usuario = usuario;  
17         this.horario = horario;  
18         this.fecha = fecha;  
19     }  
20  
21     public Reserva(Long id, Usuario usuario, Horario horario, Date  
22         fecha) {  
23         this.id = id;  
24         this.usuario = usuario;  
25         this.horario = horario;  
26         this.fecha = fecha;  
27     }  
28     public Long getId() {  
29         return this.id;  
30     }  
31  
32     public void setId(Long id) {  
33         this.id = id;  
34     }  
35  
36     public Usuario getUsuario() {  
37         return this.usuario;  
38     }  
39  
40     public void setUsuario(Usuario usuario) {  
41         this.usuario = usuario;  
42     }  
43  
44     public Horario getHorario() {  
45         return this.horario;  
46     }  
47  
48     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 marshallin-

g/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/
9                 gestion_reservas?" +
10                 "useUnicode=true&useJDBCCompliantTimezoneShift=true&
11                 serverTimezone=UTC"+
12                 "&user=root&password=example");
13             } catch (SQLException | ClassNotFoundException ex) {
14                 Logger.getLogger(Conexion.class.getName()).log(Level.
15                     SEVERE, null, ex);
16             }
17     }
18     public Connection getConnection() {
19         return conn;
20     }
21 }
```

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;
2 import java.sql.SQLException;
3 import javax.naming.Context;
4 import javax.naming.InitialContext;
5 import javax.sql.DataSource;
6
7 /**
8  *
9  * @author juangu
10  */
11 public class Conexion {
```

```
12
13     Connection conn;
14     Context ctx;
15     DataSource ds;
16
17     public Conexion() {
18         // Vía DataSource con Contexto inyectado
19         try {
20             if (ctx == null)
21                 ctx = new InitialContext();
22             if (ds == null)
23                 ds = (DataSource) ((Context) ctx.lookup(
24                     "java:comp/env")).lookup("jdbc/gestionReservas");
25             conn = ds.getConnection();
26         } catch (Exception ex) {
27             System.out.println("## Conexion ERROR ## " + ex.
28                 getLocalizedMessage());
29             ctx = null;
30             ds = null;
31             conn = null;
32         }
33     }
34
35     public Connection getConnection() {
36         return conn;
37     }
38 }
```

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:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Context path="/">
3     <Resource name="jdbc/gestionReservas" global="jdbc/gestionReservas"
4         url="jdbc:mysql://localhost:33306/gestion_reservas"
5         auth="Container" type="javax.sql.DataSource"
6         maxTotal="100" maxIdle="30" maxWaitMillis="10000"
7         username="root" password="example" driverClassName="com.
8             mysql.cj.jdbc.Driver"
9     />
10 </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:

```
1 gestionReservas: {
2     url:"jdbc:mysql://localhost:33306/gestion_reservas",
3     auth:"Container",
4     type:"javax.sql.DataSource",
5     maxTotal:"100",
```



```
6     maxIdle:"30",
7     maxWaitMillis:"10000",
8     username:"root",
9     password:"example",
10    driverClassName:"com.mysql.cj.jdbc.Driver"
11 }
```

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 saltamos 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()`

```
1 public class Alumno extends HttpServlet {
2
3     Connection conn;
4 }
```

```

5      @Override
6      public void init() throws ServletException {
7          Conexion conexion = new Conexion();
8          this.conn = conexion.getConnection();
9      }

```

Cerrar la conexión desde el método destroy()

```

1  @Override
2      public void destroy() {
3          try {
4              this.conn.close();
5          } catch (SQLException ex) {
6
7          }
8      }

```

Conectando a la Base de Datos

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                  conn =
8                      DriverManager.getConnection("jdbc:mysql://localhost/
9                          gestion_reservas?" +
10                             "useUnicode=true&useJDBCCompliantTimezoneShift=true&
11                             serverTimezone=UTC"+
12                             "&user=root&password=example");
13              } catch (SQLException | ClassNotFoundException ex) {
14                  Logger.getLogger(Conexion.class.getName()).log(Level.
15                      SEVERE, null, ex);
16              }
17      }
18      public Connection getConnection() {
19          return conn;
20      }
21  }

```

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(
37                 "nombre"));
38             li_ins.add(instala);
39         }
40         // cerramos la conexión
41         conn.close();
42     } catch (SQLException ex) {
43         System.out.println("ERROR" + ex.getMessage());
44         li_ins = null;
45     }
46     return li_ins;
}
```

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
5   *               delinstalaumno
6   * @param new_al El objeto que contiene los datos nuevos
7   *               delinstalaumno
8   * @return true si se lleva a cabo correctamente <br>
9   *         false si no se actualiza nada (error de conexión, no
10   *         estaba
11   *         elinstalaumno en la BBDD...) <br>
12   */
13  @Override
14  public boolean update(Instalacion old_al, Instalacion new_al) {
15
16      return update(old_al.getId(), new_al);
17  }
18
19  /**
20   * Este método actualiza una instalación en la BBDD
21   *
22   * @param old_id El id antiguo delinstalaumno
23   * @param new_al El objeto que contieneinstalainstalaumno
24   *               actualizado
25   * @return true si se lleva a cabo correctamente <br>
26   *         false si no se actualiza nada (error de conexión, no
27   *         estaba
28   *         elinstalaumno en la BBDD...) <br>
29   */
30  @Override
31  public boolean update(Integer old_id, Instalacion new_al) {
32      boolean exito = true;
33      try {
34          Conexion conexion = new Conexion();
35          Connection conn = conexion.getConnection();
36          String sql = "UPDATE instalacion SET id=?, nombre=? WHERE
37                      id=?";
38          PreparedStatement pstmt = conn.prepareStatement(sql);
39          pstmt.setInt(3, old_id);
40          pstmt.setInt(1, new_al.getId());
41          pstmt.setString(2, new_al.getName());
42          if (pstmt.executeUpdate() == 0) {
43              exito = false;
44          }
45          conn.close();
46      } catch (SQLException ex) {
47          exito = false;
48      }
49      return exito;
50  }
```

Borrar (InstalacionDaoImpl.java)

```
1  /**
2   * Este método borra de la BBDD el Alumno cuyos datos coinciden con
3   * los de el
4   * objeto que se le pasa como parámetro
5   *
6   * @param instala a borrar
7   * @return true si borra un instala <br>
8   *         false si el instala no existe o no se puede conectar
9   *         a la BBDD
10  */
11  @Override
12  public boolean delete(Instalacion instala) {
13      return delete(instala.getId());
14  }
15
16  @Override
17  public boolean delete(Integer id_al) {
18      boolean exito = true;
19      try {
20          Conexion conexion = new Conexion();
21          Connection conn = conexion.getConnection();
22          String sql = "DELETE FROM instalacion WHERE id=?";
23          PreparedStatement pstmt = conn.prepareStatement(sql);
24          pstmt.setInt(1, id_al);
25          if (pstmt.executeUpdate() == 0) {
26              exito = false;
27          }
28          conn.close();
29      } catch (SQLException ex) {
30          exito = false;
31      }
32      return exito;
33  }
```

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:

```
1 $ docker ps
```

Parar un contenedor:

```
1 $ docker stop [contenedor]
```

Borrar un contenedor:

```
1 $ docker rm [contenedor]
```

Listado de imágenes:

```
1 $ docker image ls
```

Borrar una imagen:

```
1 $ docker rmi [imagen]
```

Listado de volúmenes:

```
1 $ docker volume ls
```

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

```
1 $ docker volume prune
```

Borrar un volumen concreto:

```
1 $ docker volume rm [volumen]
```

Borrar todo lo relacionado con un docker-compose.yml

```
1 $ 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:

```
1 # apt update
2 # apt install net-tools
3 # apt install apache2
4 # apt install mysql-server
5 # apt install php
6 # 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
```

```
1 version: '2'
2 services:
3   mariadb:
4     image: 'bitnami/mariadb:latest'
5     environment:
6       - MARIADB_USER=bn_joomla
7       - MARIADB_DATABASE=bitnami_joomla
8       - ALLOW_EMPTY_PASSWORD=yes
9     volumes:
10      - 'mariadb_data:/bitnami'
11   joomla:
12     image: 'bitnami/joomla:latest'
13     environment:
14       - MARIADB_HOST=mariadb
15       - MARIADB_PORT_NUMBER=3306
16       - JOOMLA_DATABASE_USER=bn_joomla
17       - JOOMLA_DATABASE_NAME=bitnami_joomla
18       - ALLOW_EMPTY_PASSWORD=yes
19     labels:
20       kompose.service.type: nodeport
21     ports:
22       - '80:80'
23       - '443:443'
24     volumes:
25       - 'joomla_data:/bitnami'
26     depends_on:
27       - mariadb
28   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.

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

```
$ mkdir public-html
```

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

```
1 <!DOCTYPE html>
2 <html lang="es">
3   <head>
4     <meta charset="utf-8" />
5     <title>Apache2</title>
6   </head>
7   <body>
8     <h1>Hola Mundo</h1>
9   </body>
10 </html>
```

Ahora creamos el fichero “Dockerfile” con este contenido:

cat > Dockerfile

```
1 FROM httpd:2.4
2 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:

```
1 FROM httpd:2.4
2 COPY ./public-html/ /usr/local/apache2/htdocs/
3 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:

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

Añadimos la autenticación a Apache

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

Creo un script para automatizar la actualización de los contenedores

(fichero reload.sh)

```
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 "Generando la nueva imagen: "
8 docker build -t mi-imagen-apache2 .
9 echo "Creando el contenedor y poniéndolo en marcha"
10 docker run -dit --name mi-container-apache -p 8080:80 mi-imagen-apache2
11 echo "TERMINADO!!!"
12 ```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 Dockerfile:
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: Modificamos 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!