Tema 02: Conectividad con Bases de Datos Relacionales

GESTION DE RESERVAS

Con esta práctica de clase vamos a ver:

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.

Índice de contenido

1. Creación de la BBDD y CRUD básico.
2. Patrones y estrategias de conexión.
3. Acceso identificado y gestión de sesiones.
4. Web Services.
5. Procedimientos almacenados y disparadores.
6. APP híbrida para acceder al webservice.

El problema

Se trata de diseñar un sistema de gestión de reservas con usuarios e instalaciones configurables.

Con disparadores controlaremos temas como:

1. Que un usuario no pueda hacer más de una reserva en un día dado.
2. Que no se puedan hacer reservas con más de una semana de antelación.

Con filtros controlaremos que:

1. Sólo los usuarios identificados correctamente pueden hacer reservas
2. Un usuario no pueda modificar reservas de otro usuario
3. Sólo los usuarios identificados puedan hacer reservas

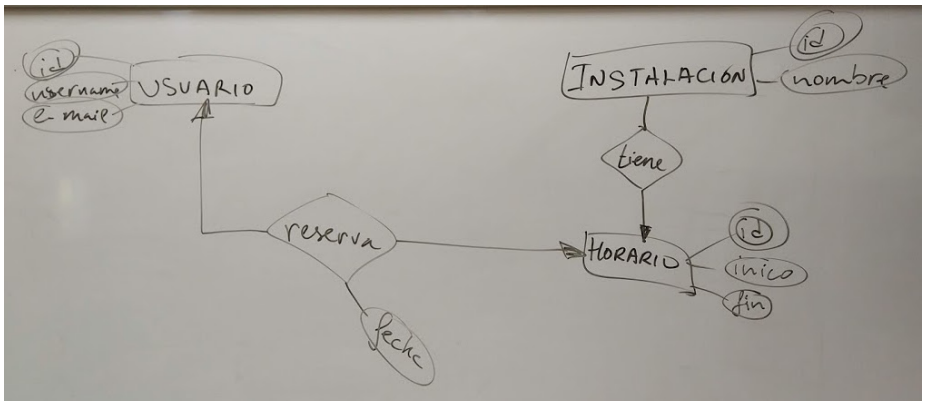

Figure 1: Diagrama ER

4. Los usuarios administradores son los únicos que pueden crear/actualizar/borrar usuarios

Preparando el entorno

Para completar esta práctica necesitamos tener instalada una JDK (al menos 1.8 o superior), Tomcat 7 o superior (lo vamos a usar como dependencia Maven), MySQL Server (usaremos un contenedor Docker), Maven, Git, y como IDE: Microsoft Visual Studio Code (plugins Java Extension Pack, Java Code Generators, Tomcat for Java y MySQL -extensión por Jun Han-).

No es necesario, pero si usamos MySQL como contenedor Docker, es más cómodo usar Linux (si no, tendremos que cambiar la IP del host o servidor de la base de datos a la máquina virtual Linux en la que creamos los contenedores).

Clonando el proyecto

```
$ git clone https://gitlab.iesvirgendelcarmen.com/juangu/tema02CRUD
```

Creando los contenedores para la base de datos

Para crear los contenedores con la base de datos y una pequeña intefaz gráfica para poder ejecutar comandos SQL y examinar tablas y datos de manera sencilla, usamos el fichero docs/stack.yml de la siguiente manera:

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

Ten cuidado al manipular el fichero YML pues es un lenguaje de marcas tabulado, es decir, los bloques se anidan con tabulaciones.

Fíjate que en la opción "restart". En producción deberás poner "always", pero ahora para desarrollo lo dejamos en "no" para forzar nosotros cuando correr o parar los contenedores con las órdenes (iniciar, parar, desactivar inicio automático con el anfitrión, activar inicio automático con el anfitrión):

```
$ docker start docker_adminer_1 docker_db_1
$ docker stop docker_adminer_1 docker_db_1
docker container update --restart=no docker_adminer_1 docker_db_1
docker container update --restart=yes docker_adminer_1 docker_db_1
```

Clase 1

Creamos la BBDD

```
-- COMO ROOT PARA CREAR LA BBDD Y EL USUARIO

-- CREAMOS LA BBDD PARA UTF-8 COLLATION EN ESPAÑOL
CREATE DATABASE gestion_reservas  \
    CHARACTER SET utf16 COLLATE utf16_spanish_ci;

-- CAMBIAMOS LA BBDD ACTIVA
USE gestion_reservas;


-- CREAMOS LAS TABLAS
CREATE TABLE `usuario` (
  `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `username` varchar(12) NOT NULL,
  `password` varchar(20) NOT NULL,
  `email` varchar(50) NOT NULL
) ENGINE='InnoDB';

CREATE TABLE `instalacion` (
  `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `nombre` varchar(50) NOT NULL
) ENGINE='InnoDB';

CREATE TABLE `horario` (
  `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `instalacion` int(11) NOT NULL,
  `inicio` time NOT NULL,
```

```
  `fin` time NOT NULL,
  FOREIGN KEY (`instalacion`) REFERENCES `instalacion` (`id`) ON DELETE CASCADE
) ENGINE='InnoDB';

CREATE TABLE `reserva` (
  `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `usuario` int(11) NOT NULL,
  `horario` int(11) NOT NULL,
  `fecha` date NOT NULL,
  FOREIGN KEY (`usuario`) REFERENCES `usuario` (`id`),
  FOREIGN KEY (`horario`) REFERENCES `horario` (`id`)
) ENGINE='InnoDB';
```

Creación del proyecto en modo interactivo (MAVEN)

Si has clonado este repositorio, no es necesario hacer este paso, sólamente cuando quieras crear un proyecto como éste.

Para crear en modo interactivo el proyecto, estructura de directorios, fichero pom.xml, etc. **desde cero**, tendríamos que usar el proyecto Maven Java Web desde el IDE o bien desde línea de comandos ejecutaríamos esta instrucción:

```
$ mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
```

Tras indicar el grupo (com.iesvdc.acceso.simplecrud) y el artefacto (simplecrud) Maven crea los ficheros necesarios.

Dependencias Maven

Antes de comenzar, veamos las dependencias (librerías) adicionales que va a necesitar nuestro proyecto.

MySQL

Necesitamos importar en la CLASS_PATH del proyecto el driver JDBC de MySQL.

```
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.1</version>
    </dependency>
```

Soporte J2EE (Servlets)

Para poder usar JSP (Java Server Pages) y Servlets (las clases necesarias), tenemos que cargar la API Web de Java:

```
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>8.0.1</version>
    <scope>provided</scope>
</dependency>
```

Soporte para JSLT (para JSP)

Para poder usar las extensiones JSLT dentro de una página JSP, necesitamos importar la API JSLT:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

Soporte JSON

Para hacer el marshalling/unmarshalling de objetos usaremos la API Gson de Google:

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
</dependency>
```

Insertando el servidor Tomcat en Maven

Para poder ejecutar Tomcat desde maven para no necesitar descargar e instalar el servidor de aplicaciones Java de la Apache foundation, añadimos estas líneas dentro de "build->plugins":

```
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
```

```
            <port>9090</port>
            <path>/</path>
            </configuration>
    </plugin>
```

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() {
        return conn;
    }
}
```

CRUD básico

LEER uno (findOne)

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

    jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;

    try {
        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");
            jsonObject="{"+ "\n"+
                "'id':'"+id+"',"+ "\n"+
                "'username':'"+username+"',"+ "\n"+
                "'password':'"+password+"',"+ "\n"+
                "'email':'"+email+"'"+ "\n"+
                "}";
        }
    }catch(Exception ex){
        // Gestión de la excepción
    }
    // devolvemos jsonObject
```

LEER todos (findAll)

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>

<sql:query var="userList" dataSource="jdbc/gestionReservas">
    select username, email, password from usuario;
</sql:query>


<%@ include file="header.jsp" %>
    <div align="center">
        <table border="1" cellpadding="5">
            <caption><h2>Lista de alumnos</h2></caption>
            <tr>
                <th>username</th>
                <th>email</th>
                <th>password</th>
            </tr>
            <c:forEach var="usuario" items="${userList.rows}">
                <tr>
                    <td><c:out value="${usuario.username}" /></td>
                    <td><c:out value="${usuario.email}" /></td>
                    <td><c:out value="${usuario.password}" /></td>
                </tr>
```

```
            </c:forEach>
        </table>
    </div>
<%@ include file="footer.jsp"%>
```

Crear

```
    Connection conexion;
    PreparedStatement pstm;
    String jdbcURL;
    jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        conexion =  DriverManager.getConnection(jdbcURL, "root", "example");
        String sql = "INSERT INTO usuario (username,password,email) VALUES(?,?,?)";
        pstm = conexion.prepareStatement(sql);
        pstm.setString(1, username);
        pstm.setString(2, password);
        pstm.setString(3, email);
        if (pstm.executeUpdate() >0) {
            // "Usuario insertado"
        } else {
            // "No se ha podido insertar"
        }
        conexion.close();
    } catch (Exception ex) {
        // "Imposible conectar a la BBDD"
    }
```

Actualizar

```
    Usuario user = new Gson().fromJson(req.getReader(), Usuario.class);
    String jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection conexion =  DriverManager.getConnection(jdbcURL, "root", "example");
        String sql = "UPDATE usuario SET username=?, password=?, email=? WHERE id=?";
        PreparedStatement pstm = conexion.prepareStatement(sql);
        pstm.setString(1, user.getUsername());
        pstm.setString(2, user.getPassword());
        pstm.setString(3, user.getEmail());
        pstm.setInt(4, user.getId());

        if (pstm.executeUpdate() >0) {
```

```
            resp.getWriter().println("Usuario insertado");
        } else {
            resp.getWriter().println("No se ha podido insertar");
        }

        conexion.close();
    } catch (Exception ex) {
        resp.getWriter().println(ex.getMessage());
        resp.getWriter().println(ex.getLocalizedMessage());
        // resp.getWriter().println("Imposible conectar a la BBDD");
    }
```

Borrar

```
    Connection conexion;
    PreparedStatement pstm;
    String jdbcURL;

    jdbcURL = JDBC_MYSQL_GESTION_RESERVAS;
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        conexion =  DriverManager.getConnection(jdbcURL, "root", "example");
        String sql = "DELETE FROM usuario WHERE id=?";
        pstm = conexion.prepareStatement(sql);
        pstm.setInt(1, Integer.parseInt(id));
        if ( pstm.executeUpdate()==0 ) {
            jsonObject="{ "+
                "'id':'"+id+"'}";
        }
    }catch(Exception ex){
        // "No se pudo eliminar"
    }
```

CRUD (patrón DAO)

Partimos de una clase base Alumno (POJO) que contiene los objetos que vamos a almacenar/recuperar de la base de datos:

```
public class AlumnoPOJO {
    private Integer id;
    private String nombre;
    private String apellido;

    public AlumnoPOJO() {
        this.id = null;
```

```
        this.nombre = null;
        this.apellido = null;
    }

    public AlumnoPOJO(String nombre, String apellido) {
        this.id = null;
        this.nombre = nombre;
        this.apellido = apellido;
    }

    public AlumnoPOJO(int id, String nombre, String apellido) {
        this.id = id;
        this.nombre = nombre;
        this.apellido = apellido;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    [.....]
```

Leer todos

```
public List<AlumnoPOJO> findAll() {
        AlumnoPOJO al;
        List<AlumnoPOJO> li_al = new ArrayList();
        try {
            // conectamos a la BBDD
            Conexion conexion = new Conexion();
            Connection conn = conexion.getConnection();
            // esta es la cadena SQL de conslulta
            String sql = "SELECT * FROM ALUMNO";
            // usamos este objeto porque es más seguro
            PreparedStatement pstmt = conn.prepareStatement(sql);
            // ejecutar la consulta contra la base de datos y
            // devuelve el resultado en el ResultSet (parecido a
            // un Array con iterador
            ResultSet rs  = pstmt.executeQuery();
            // recorro el resultset mientras tengo datos
            while (rs.next()){
```

```
                al = new AlumnoPOJO(
                    rs.getInt("id"),
                    rs.getString("nombre"),
                    rs.getString("apellido"));
                li_al.add(al);
            }
            // cerramos la conexión
            conn.close();
        } catch (SQLException ex) {
            System.out.println("ERROR"+ ex.getMessage());
            li_al = null;
        }
        return li_al;
    }
```

Crear

```
public boolean create(AlumnoPOJO al){
        boolean exito=true;
        try {
            Conexion conexion = new Conexion();
            Connection conn = conexion.getConnection();
            String sql =
                "INSERT INTO ALUMNO VALUES (NULL,?,?)";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, al.getNombre());
            pstmt.setString(2, al.getApellido());
            pstmt.executeUpdate();
            conn.close();
        } catch (SQLException ex) {
            System.out.println("ERROR:  "+ex.getMessage());
            exito = false;
        }
        return exito;
    }
```

Actualizar

```
/**
     * Este método actualiza un alumno en la BBDD
     * @param old_id
     * El id antiguo del alumno
     * @param new_al
     * El objeto que contiene al alumno actualizado
```

```
     * @return
     * true si se lleva a cabo correctamente <br>
     * false si no se actualiza nada (error de conexión, no
     * estaba el alumno en la BBDD...) <br>
     */
    public boolean update(Integer old_id, AlumnoPOJO new_al) {
        boolean exito=true;
        try {
            Conexion conexion = new Conexion();
            Connection conn = conexion.getConnection();
            String sql =
                "UPDATE ALUMNO SET id=?, nombre=?, apellido=? WHERE id=?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setInt(4, old_id);
            pstmt.setInt(1, new_al.getId());
            pstmt.setString(2, new_al.getNombre());
            pstmt.setString(3, new_al.getApellido());
            if (pstmt.executeUpdate()==0) {
                exito = false;
            }
            conn.close();
        } catch (SQLException ex) {
            exito = false;
        }
        return exito;
    }
```

Borrar

```
    public boolean delete(Integer id_al){
        boolean exito=true;
        try {
            Conexion conexion = new Conexion();
            Connection conn = conexion.getConnection();
            String sql = "DELETE FROM ALUMNO WHERE id=?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setInt(1, id_al);
            if (pstmt.executeUpdate()==0) {
                exito = false;
            }
            conn.close();
        } catch (SQLException ex) {
            exito = false;
        }
        return exito;
```

```
    }
```

Clase 2

Creando los Plain Old Java Objects

Para poder hacer el marshalling/unmarshalling de objetos, necesitamos primero tener los objetos Java en memoria.

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

Ahora ya es posible con Gson, por ejemplo, desde el servlet directamente hacer el marshalling/unmarshalling de JSON a Java.

Conectando a la base de datos cargando la configuración vía JNDI

Aprovechamos también 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.

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:

```
public class Conexion {
    Connection conn;
    public Conexion(){
        if (conn==null)
            try {
                Class.forName("com.mysql.jdbc.Driver");
                this.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() {
        return conn;
    }
}
```

Sin embargo, si queremos abrir una conexión cargando la configuración desde una consulta al contexto de la aplicación, lo haríamos así:

```
import java.sql.Connection;
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;
    }
}
```

Conexión dedicada: 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.).
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.

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:

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) {

        }
    }
```

Añadiendo seguridad a la aplicación

Dada la similitud de este sistema con otros sistemas de seguridad de diferentes frameworks y tecnologías, nosotros vamos a recurrir a un filtro para asegurar partes de nuestra Web.

Un filtro nos permite inyectar precondiciones a una petición Web (HTTP GET, POST, PUT, DELETE. . . .) o bien postcondiciones, es decir, antes o después de llamar al servlet que despacha la petición del verbo HTTP, se ejecutaría también el método indicado en el filtro.

Esto nos puede ayudar, por ejemplo, para dar seguridad a nuestras aplicaciones. Si tenemos un formulario que haga POST a un servicio de login, desde el servlet que se encarga del servicio, podríamos crear la sesión o cookie que el filtro después comprobará para ver si estamos *loggeados*.

El servicio de gestión de sesión

Nuestro servicio consta de un formulario de login que hace un POST al servlet que crea la sesión y manda la cookie al cliente para mantener la sesión.

Un filtro intercepta las peticiones a las URLs protegidas y comprueba que hayamos entrado en el sistema con un usuario válido. Aún no estamos usando ACLs (listas de acceso de usuarios).

Para cerrar la sesión un servlet eliminará la cookie.

Creando la sesión

Formulario de login: Hace un POST al servlet que comprueba contra la base de datos si existe el usuario con esa contraseña.

```
<div class="container">
    <div class="jumbotron"><h2>Login</h2></div>
    <div class="form">
        <form action="login" method="POST">
            <input name="username" type="text" placeholder="Nombre de usuario" />
            <input name="pwd" type="password" placeholder="Contraseña" />
            <button type="submit">Enviar</button>
        </form>
```

```
    </div>
</div>
```

Servlet de login: Recoge los datos del formulario anterior y comprueba contra la base de datos si existe el usuario con esa contraseña.

```
/**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/login")
public class Login extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        Conexion conn = new Conexion();
        try {
            Connection conexion = conn.getConnection();

            // get request parameters for userID and password
            String user = request.getParameter("username");
            String pwd = request.getParameter("pwd");

            String sql = "SELECT * FROM usuario WHERE username=? AND password=?";
            PreparedStatement pstm = conexion.prepareStatement(sql);
            pstm.setString(1, user);
            pstm.setString(2, pwd);
            ResultSet rs = pstm.executeQuery();

            if (rs.next()) {
                HttpSession session = request.getSession();
                session.setAttribute("user", user);
                // setting session to expiry in 30 mins
                session.setMaxInactiveInterval(30 * 60);
                Cookie userName = new Cookie("ges_res.user", user);
                userName.setMaxAge(30 * 60);
                response.addCookie(userName);
                response.sendRedirect("index.jsp");
            } else {
                RequestDispatcher rd =
                    getServletContext().getRequestDispatcher("/login.jsp");
                PrintWriter out = response.getWriter();
    // out.println("<font color=red>Either user name or password is wrong."+"</font>");
                rd.include(request, response);
            }

            conexion.close();
```

```
        } catch (SQLException e) {
            // response.sendRedirect("login.jsp");
            response.getWriter().print(e.getLocalizedMessage());
        }
    }

}
```

Identificando sesión

Mapeado del Filtro: web.xml: Indicamos las URLs que están protegidas (son interceptadas) por el filtro Java.

```
<filter>
    <filter-name>UserFilter</filter-name>
    <filter-class>
        com.iesvdc.acceso.simplecrud.controller.AuthenticationFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>UserFilter</filter-name>
    <url-pattern>/user/*</url-pattern>
    <url-pattern>/installation/*</url-pattern>
    <url-pattern>/privado/*</url-pattern>
</filter-mapping>
```

Filter Java: Implementación del filtro.

```
public class AuthenticationFilter implements javax.servlet.Filter {
    public static final String AUTHENTICATION_HEADER = "Authorization";

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain filter) throws IOException, ServletException {
        if (request instanceof HttpServletRequest) {
            HttpServletRequest httpServletRequest = (HttpServletRequest) request;

            Cookie loginCookie = null;
            Cookie[] cookies = httpServletRequest.getCookies();
            if (cookies != null) {
                for (Cookie cookie : cookies) {
                    if (cookie.getName().equals("ges_res.user")) {
                        loginCookie = cookie;
                        break;
```

```
                    }
                }
            }
            if (loginCookie != null) {
                filter.doFilter(request, response);
            } else {
                // ((HttpServletResponse)response.sendRedirect("login.jsp");
                if (response instanceof HttpServletResponse) {
                    HttpServletResponse httpServletResponse =
                        (HttpServletResponse) response;
                    httpServletResponse.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                    httpServletResponse.sendRedirect("/login.jsp");
                }
            }
        }
    }

    @Override
    public void destroy() {
    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {
    }
}
```

Cerrando sesión

Servlet Logout: Elimina la cookie.

```
@WebServlet("/logout")
public class Logout extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        Cookie loginCookie = null;
        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for (Cookie cookie : cookies) {
                if (cookie.getName().equals("ges_res.user")) {
                    loginCookie = cookie;
                    break;
                }
            }
        }
```

```
        if (loginCookie != null) {
            loginCookie.setMaxAge(0);
            response.addCookie(loginCookie);
        }
        response.sendRedirect("login.jsp");
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        this.doPost(request, response);
    }

}
```

WS: Servicios REST

Hasta ahora hemos visto un enfoque tradicional (formularios con métodos GET y POST) y un enfoque híbrido (métodos GET, POST, PUT y DELETE) con algo de JavaScript.

Otro enfoque a la hora de diseñar aplicaciones para la Web es la filosofía SAP (Single Application Page), donde usamos todas las características del modelo de cajas de HTML5.

Single Application Page

Se trata de tener una aplicación HTML5+JS donde existen varias cajas ocultas que contienen las diferentes vistas de la aplicación. En cada momento vamos cambiando de una a otra según un menú principal que hace de controlador para activar/desactivar vistas.

La aplicación se comunica con un servicio REST con verbos HTTP (ej. GET/POST/PUT/DELETE).

Vamos a implementar las siguientes rutas y verbos HTTP:

Descripción	Verbo HTTP	ruta
Listar todas las instalaciones	GET	/api/instalacion
Ver el detalle de una instalación	GET	/api/instalacion/{id}
Crear una nueva instalación	POST	/api/instalacion
Borrar una instalación	DELETE	/api/instalacion/{id}
Buscar una instalación por nombre	GET	/api/instalacion/nombre/{nombre}

Veremos este apartado con más detalle en la siguiente clase.

Creación de un WebService con Tomcat y Jersey

Jersey es un Servlet (aplicación) genérica a la que le indicamos el paquete o clase que queremos exponer a nuestro servicio y **automágicamente** se encarga de hacerlo.

Añadiendo las dependencias al proyecto

Para poder usar Jersey, primero hemos de añadir las dependencias necesarias en nuestro pom.xml.

Necesitamos JAXB como marshaller/unmarshaller (artefactos jaxb*), esto nos convierte de objeto plano Java a una interpretación XML (en memoria).

Jersey dispone de tres partes:

1. Conector JAXB: Interpreta las representaciones internas JAXB (marshaller/unmarshaller)
2. Generador de JSON: Convierte las representaciones internas de/hacia JSON
3. Servlet: El sel servicio propiamente dicho (habrá que añadir una entrada en el web.xml a tal efecto).

```
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.0</version>
</dependency>

<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-core</artifactId>
  <version>2.3.0</version>
</dependency>

<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-impl</artifactId>
  <version>2.3.0</version>
</dependency>


<dependency>
  <groupId>javax.activation</groupId>
```

```
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
  </dependency>

  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-jaxb</artifactId>
    <version>2.25.1</version>
  </dependency>
  <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>
      <version>2.25.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet-core</artifactId>
    <version>2.25.1</version>
  </dependency>
```

Creando el POJO

Para que podamos trabajar con JAXB, necesitamos trabajar con objetos Java sencillos. Ejemplo:

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

public class Instalacion {

    private int id;
    private String name;

    public Instalacion(){}

    public Instalacion(String name) {
        this.name = name;
    }

    public Instalacion(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
```

```
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Patrón DAO

Usaremos el patrón DAO (Data Access Object) para crear una clase que se encargue de salvar el *desfase objeto-relacional*. Ejemplo:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.iesvdc.acceso.simplecrud.model;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

/**
 *
 * @author juangu
 */
public class InstalacionDAO {
    // CRUD, findAll, finById, count
```

```
Connection conn;

public InstalacionDAO(){
    conn = new Conexion().getConnection();
}

public InstalacionDAO(Connection conexion){
    this.conn=conexion;
}

public boolean create(Instalacion instala){
    boolean exito=true;
    try {
        // Conexion conexion = new Conexion();
        // Connection conn = conexion.getConnection();
        String sql =
            "INSERT INTO instalacion VALUES (NULL,?,?)";
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, instala.getId());
        pstmt.setString(2, instala.getName());
        pstmt.executeUpdate();
        // conn.close();
    } catch (SQLException ex) {
        System.out.println("ERROR:  "+ex.getMessage());
        exito = false;
    }
    return exito;
}

public Instalacion findById(Integer id){
    Instalacion instala;
    try {
        String sql =
            "SELECT * FROM instalacion WHERE id=?";
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, id);
        System.err.println("\nID:: "+id+"\n");
        ResultSet rs  = pstmt.executeQuery();
        rs.next();
       instala = new Instalacion(
            rs.getInt("id"),
            rs.getString("nombre"));
        // conn.close();
    } catch (SQLException ex) {
       instala = null;
    }
```

```
    return instala;
}

public List<Instalacion> findAll() {
    Instalacion instala;
    List<Instalacion> li_ins = new ArrayList();
    try {

        String sql = "SELECT * FROM instalacion";

        PreparedStatement pstmt = conn.prepareStatement(sql);

        ResultSet rs  = pstmt.executeQuery();
        // recorro el resultset mientras tengo datos
        while (rs.next()){
           instala = new Instalacion(
                rs.getInt("id"),
                rs.getString("nombre"));
            li_ins.add(instala);
        }
        // cerramos la conexión
        // conn.close();
    } catch (SQLException ex) {
        System.out.println("ERROR"+ ex.getMessage());
        li_ins = null;
    }
    return li_ins;
}


/**
 * Este método busca Instalacions en la BBDD por nombre:
 * @param nombre
 * El nombre a buscar
 * @return
 * Devuelve:
 * null: si hayinstalagún error (no se puede conectar a la BBDD...). <br>
 * ArrayList vacío (length == 0): si no hay nadie con ese nombre. <br>
 * ArrayList con Instalacions: si hayinstalaumnos con ese nombre.<br>
 */
public List<Instalacion> findByNombre(String nombre){
    Instalacion instala;
    List<Instalacion> li_ins = new ArrayList();
    try {
        // conectamos a la BBDD
        Conexion conexion = new Conexion();
```

```
        Connection conn = conexion.getConnection();
        // esta es la cadena SQL de conslulta
        String sql = "SELECT * FROM instalacion WHERE nombre=?";
        // usamos este objeto porque es más seguro
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, nombre);
        // ejecutar la consulta contra la base de datos y
        // devuelve el resultado en el ResultSet (parecido a
        // un Array con iterador
        ResultSet rs  = pstmt.executeQuery();
        // recorro el resultset mientras tengo datos
        while (rs.next()){
           instala = new Instalacion(
                rs.getInt("id"),
                rs.getString("nombre"));
            li_ins.add(instala);
        }
        // cerramos la conexión
        conn.close();
    } catch (SQLException ex) {
        System.out.println("ERROR"+ ex.getMessage());
        li_ins = null;
    }
    return li_ins;
}
```

```
/**
 * Este método actualiza uninstalaumno en la BBDD
 * @param old_al
 * El objeto que contiene los datos antiguos delinstalaumno
 * @param new_al
 * El objeto que contiene los datos nuevos delinstalaumno
 * @return
 * true si se lleva a cabo correctamente <br>
 * false si no se actualiza nada (error de conexión, no
 * estaba elinstalaumno en la BBDD...) <br>
 */
public boolean update(Instalacion old_al, Instalacion new_al) {

    return update(old_al.getId(),new_al);
}

/**
```

```
     * Este método actualiza una instalación en la BBDD
     * @param old_id
     * El id antiguo delinstalaumno
     * @param new_al
     * El objeto que contieneinstalainstalaumno actualizado
     * @return
     * true si se lleva a cabo correctamente <br>
     * false si no se actualiza nada (error de conexión, no
     * estaba elinstalaumno en la BBDD...) <br>
     */
    public boolean update(Integer old_id, Instalacion new_al) {
        boolean exito=true;
        try {
            Conexion conexion = new Conexion();
            Connection conn = conexion.getConnection();
            String sql =
                "UPDATE instalacion SET id=?, nombre=? WHERE id=?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setInt(3, old_id);
            pstmt.setInt(1, new_al.getId());
            pstmt.setString(2, new_al.getName());
            if (pstmt.executeUpdate()==0) {
                exito = false;
            }
            conn.close();
        } catch (SQLException ex) {
            exito = false;
        }
        return exito;
    }

    /**
     * Este método borra de la BBDD el Instalacion cuyos datos
     * coinciden con los de el objeto que se le pasa como
     * parámetro
     * @paraminstalainstalaumno a borrar
     * @return
     * true si borra uninstalaumno <br>
     * false si elinstalaumno no existe o no se puede conectar a la BBDD <br>
     */
    public boolean delete(Instalacion instala){
        return delete(instala.getId());
    }

    public boolean delete(Integer id_al){
        boolean exito=true;
```

```
        try {
            Conexion conexion = new Conexion();
            Connection conn = conexion.getConnection();
            String sql = "DELETE FROM instalacion WHERE id=?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setInt(1, id_al);
            if (pstmt.executeUpdate()==0) {
                exito = false;
            }
            conn.close();
        } catch (SQLException ex) {
            exito = false;
        }
        return exito;
    }

}
```

El servicio web (WS)

Primero creamos el recurso que usará Jersey:

```
package com.iesvdc.acceso.simplecrud.controller.service;

import com.iesvdc.acceso.simplecrud.model.*;

import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.DELETE;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

/**
 *
 * @author Juangu <jgutierrez at iesvirgendelcarmen.coms>
 */
```

```
@Path("/")
public class InstalacionResource {

    @GET
    @Path("instalacion")
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public List<Instalacion> getInstalacions() {
        InstalacionDAO al_dao = new InstalacionDAO();
        List<Instalacion> list_al;
        try {
            list_al = al_dao.findAll();
        } catch (Exception ex) {
            list_al = new ArrayList<>();
            Logger.getLogger(ex.getLocalizedMessage());
        }
        return list_al;
    }

    @GET
    @Path("instalacion/{id}")
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public Instalacion getInstalacionById(@PathParam("id") String id) {
        InstalacionDAO al_dao = new InstalacionDAO();
        Instalacion al;
        try {
            al = al_dao.findById(Integer.parseInt(id));
        } catch (Exception ex) {
            al = new Instalacion(-1, "Error");
            Logger.getLogger(ex.getLocalizedMessage());
        }
        return al;
    }


    @GET
    @Path("instalacion/nombre/{nombre}")
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public List<Instalacion> getInstalacionByNombre(@PathParam("nombre") String nombre) {
        InstalacionDAO al_dao = new InstalacionDAO();
        List<Instalacion> list_al;
        try {
            list_al = al_dao.findByNombre(nombre);
        } catch (Exception ex) {
            list_al = new ArrayList<>();
            Logger.getLogger(ex.getLocalizedMessage());
        }
```

```
        return list_al;
    }


    @PUT
    @Path("instalacion/{id}")
    @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public void updateInstalacion(@PathParam("id") Integer id, Instalacion al) {
        InstalacionDAO al_dao = new InstalacionDAO();
        try {
            al_dao.update(id, al);
        } catch (Exception ex) {
            Logger.getLogger(ex.getLocalizedMessage());
        }
    }


    @POST
    @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    @Path("alumno")
    public Response createInstalacion(Instalacion al) {
        InstalacionDAO al_dao = new InstalacionDAO();
        try {
            al_dao.create(al);
        } catch (Exception ex) {
            Logger.getLogger(ex.getLocalizedMessage());
            return Response.status(400).entity(al).build();
        }
        return Response.status(200).entity(al).build();
    }

    @DELETE
    @Path("instalacion/{id}")
    public void deleteInstalacion(@PathParam("id") Integer id) {
        InstalacionDAO al_dao = new InstalacionDAO();
        try {
            al_dao.delete(id);
        } catch (Exception ex) {
            Logger.getLogger(ex.getLocalizedMessage());
        }
    }
}
```

Ahora hay que modificar el archivo **web.xml** para dar de alta el servlet Jersey en la ruta *"/rest"*:

```
<servlet>
    <servlet-name>jersey-servlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>com.iesvdc.acceso.simplecrud.controller.service</param-value>
    </init-param>
    <init-param>
        <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>jersey-servlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Probando el servicio

Hasta que tengamos el frontend, podemos hacer nuestros tests con la extensión de Mozilla Firefox. "REST Client".

Repaso disparadores

La sintaxis para crear un disparador es la siguiente:

```
CREATE TRIGGER nombre_disparador
  [BEFORE|AFTER] [INSERT|DELETE|UPDATE|...]
  ON nombre_tabla FOR EACH ROW
  BEGIN
    [cuerpo del disparador]
  END;
```

Ejemplo 1: No más de una reserva al día por persona

Sea el dominio del proyecto (gestión de reservas), imaginemos que no queremos delegar sólo en el código de la aplicación que se pueda reservar más de una vez por usuario (si esta condición la incluimos en la base de datos, será más segura, será mucho más difícil que un usuario malintencionado pueda romperla).

La alternativa es crear un disparador que primero compruebe si ya tenemos reservas en el día que vamos a insertar la nueva reserva:

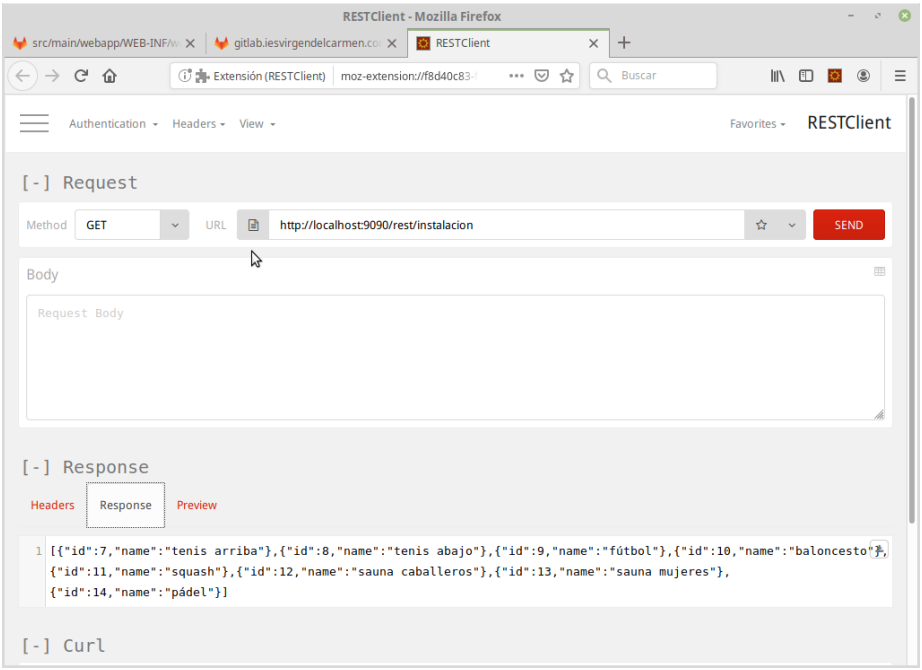

Figure 2: Extensión RESTClient de firefox

```
SELECT COUNT(*) FROM `reserva` WHERE `reserva`.`fecha` = NEW.`fecha`
```

Fíjate en el prefijo "NEW.", en los disparadores, recuerda que esto sirve para acceder a la nueva tupla que queremos insertar, seguido de un punto y luego el nombre de cada columna para acceder al valor.

Antes de insertar una nueva reserva **BEFORE INSERT**, para cada tupla que vamos a insertar **FOR EACH ROW** comprobamos con un **IF** si ya había más de una reserva ese día para un usuario dado. Si ya la había, no procedemos a la inserción y devolvemos un mensaje de error *'sólo se permite una reserva al día'*, con código de error *45001* (este código nos lo devuelve MySQL cuando hagamos la consulta).

```
-- Sólo deja una reserva para cada usuario
DROP TRIGGER IF EXISTS reserva_diaria;

DELIMITER $$
CREATE TRIGGER `reserva_diaria`
  BEFORE INSERT ON `reserva` FOR EACH ROW
  BEGIN
    IF (SELECT COUNT(*) FROM `reserva`
    WHERE `reserva`.`fecha` = NEW.`fecha`
      AND `reserva`.`usuario`= NEW.`usuario`) > 0
```

```
    THEN
      SIGNAL sqlstate '45001'
        SET message_text = 'Sólo se permite una reserva al día.';
    END IF;
  END;
$$
```

Ejemplo 2: No podemos reservar con más de dos semanas de antelación

Volvamos al problema inicial, el sistema de gestión de reservas. Si no controlamos un poco las fechas de las reservas, podemos encontrarnos que usuarios malintencionados pueden comprometer la integridad del sistema o situaciones indeseadas. Por tanto deberíamos controlar que:

1. No se pueda borrar o actualizar reservas pasadas:
    1. No se puede borrar reservas ya pasadas. Si se permitiera borrar reservas, podríamos borrar una ya pasada, luego podríamos librarnos de pagar las instalaciones que hemos disfrutado (disparador "ON DELETE").
    2. Al día siguiente, o al haber pasado la hora de una reserva, si ponen esa misma reserva a otro nombre, puede ocurrir que a final de mes en vez de cobrar al usuario que ha disfrutado de la instalación, se le va a cobrar a otro (disparador "ON UPDATE").
2. No se puede reservar en fechas anteriores al día actual. No tiene sentido.
3. No se permite reservar con más de dos semanas de antelación. En caso contrario, un usuario podría reservar todos los días del año a las 18:00 una instalación, por ejemplo.

Ejemplo de disparador (caso 1.1): Observa cómo no podemos usar ahora "NEW.*" para acceder a un campo, pues no existiría al borrar, sólo podemos usar "OLD.*" para ello:

```
DROP TRIGGER IF EXISTS reserva_pasado;

DELIMITER $$
CREATE TRIGGER `reserva_pasado`
  BEFORE DELETE ON `reserva` FOR EACH ROW
  BEGIN
    IF ( OLD.`fecha` < CURDATE())
    THEN
      SIGNAL sqlstate '45004'
        SET message_text = 'No se permite eliminar una fecha pasada.';
    END IF;
  END;
$$
```

En este ejemplo surge la pregunta... ¿y si borro el mismo día de la reserva?. Esto ya sería una consulta más complicada, pues habría que mirar que la hora actual es menor que la hora de inicio del horario de la reserva (habría que hacer una consulta más compleja con un JOIN). Puedes intentarlo para ampliar.

Ejemplo de disparador (caso 1.2): No se puede cambiar de nombre (a otra persona) una reserva el día de la misma o cuando ha pasado:

```
DROP TRIGGER IF EXISTS reserva_actualizar_pasado;

DELIMITER $$
CREATE TRIGGER `reserva_actualizar_pasado`
  BEFORE UPDATE ON `reserva` FOR EACH ROW
  BEGIN
    IF ( OLD.`fecha` <= CURDATE())
    THEN
      SIGNAL sqlstate '45004'
        SET message_text = 'No se permite actualizar una reserva ya o casi pasada.';
    END IF;
  END;
$$
```

Ejemplo de disparador (casos 2 y 3):

```
-- Si machacamos reservas antiguas bajo otro nombre, nos libramos de pagar a final de mes..
DROP TRIGGER IF EXISTS reserva_semanal;

DELIMITER $$
CREATE TRIGGER `reserva_semanal`
  BEFORE INSERT ON `reserva` FOR EACH ROW
  BEGIN
    IF ( NEW.`fecha` < CURDATE())
    THEN
      SIGNAL sqlstate '45002'
        SET message_text = 'No se permite reservar en una fecha anterior a la actual.';
    ELSEIF ( NEW.`fecha` > DATE_ADD(CURDATE(), INTERVAL 14 DAY) )
    THEN
        SIGNAL sqlstate '45003'
          SET message_text = 'No se permite reservar con más de dos semanas de antelación.';
    END IF;
  END;
$$
```

Aplicación híbrida

Recordamos que se trata de tener una aplicación HTML5+JS donde existen varias cajas ocultas que contienen las diferentes vistas de la aplicación. En cada momento vamos cambiando de una a otra según un menú principal que hace de controlador para activar/desactivar vistas.

La aplicación se comunica con el servicio REST con verbos HTTP (ej. GET/POST/PUT/DELETE).

Ya tenemos implementados los siguientes *end-points* en el **back-end**:

Descripción	Verbo HTTP	ruta
Listar todas las instalaciones	GET	/api/instalacion
Ver el detalle de una instalación	GET	/api/instalacion/{id}
Crear una nueva instalación	POST	/api/instalacion
Borrar una instalación	DELETE	/api/instalacion/{id}
Buscar una instalación por nombre	GET	/api/instalacion/nombre/{nombre}

Ahora vamos a preparar una APP híbrida como **front-end**, que con AJAX haga esas consultas y muestre los resultados.

Modificando el backend para que soporte peticiones de otro origen

Para resolver el problema del CORS deberemos crear un filtro que intercepte peticiones de origen cruzado y las permita sólo para los servicios visibles a nuestra aplicación.

Ejemplo de filtro CORS Java:

```
import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletResponse;

    public class CORSFilter implements Filter {

        public void doFilter(
          ServletRequest req,
```

```
            ServletResponse res,
            FilterChain chain) throws IOException, ServletException {

           HttpServletResponse response = (HttpServletResponse) res;
           response.setHeader("Access-Control-Allow-Origin", "*");
           response.setHeader("Access-Control-Allow-Methods", "POST, GET, PUT, DELETE");
           response.setHeader("Access-Control-Max-Age", "3600");
           response.setHeader("Access-Control-Allow-Headers",
            "Origin, x-requested-with, Content-Type, Accept");
           chain.doFilter(req, res);
          }

          public void init(FilterConfig filterConfig) {}

          public void destroy() {}

}
```

Recuerda modificar el fichero **web.xml** para que la ruta del Web Service tenga permitido el acceso desde orígenes desconocidos:

```
  <filter>
        <filter-name>CorsFilter</filter-name>
        <filter-class>com.iesvdc.acceso.simplecrud.controller.CORSFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>CorsFilter</filter-name>
        <url-pattern>/rest/*</url-pattern>
    </filter-mapping>
```

Preparación del entorno

Creamos una carpeta frontend en nuestro proyecto Web. Para crear el proyecto Cordova, ejecutamos los siguientes comandos en la terminal dentro del directorio recién creado:

```
$ npm install -g cordova
$ cordova create reservas com.iesvdc.dam.acceso ReservAPP
$ cd reservas
$ cordova platform add browser
$ npm install jquery
$ npm install popper.js
$ npm install bootstrap
```

Ahora, de los directorios node_modules que se han creado para cada una de las instalaciones. Me creo un script añadir dependencias:

```
#!/bin/bash

mkdir -p www/vendor/js www/vendor/css
echo "Añadiendo jQuery"
cp node_modules/jquery/dist/jquery.min.js \
  node_modules/jquery/dist/jquery.min.map www/vendor/js
echo "Añadiendo PopperJS"
cp node_modules/popper.js/dist/popper.min.js \
  node_modules/popper.js/dist/popper.min.js.map www/vendor/js
echo "Añadiendo Bootstrap"
cp node_modules/bootstrap/dist/css/bootstrap.min.css www/vendor/css
cp node_modules/bootstrap/dist/js/bootstrap.min.js \
  node_modules/bootstrap/dist/js/bootstrap.min.js.map www/vendor/js
```

Para lanzar un navegador Web Cordova:

```
cordova run browser
```

Ejemplo de aplicación HTML5/JS

En la carpeta **www** recién creada por *cordova*, vamos a editar el fichero index.html. La idea es hacer una SAP (Single Application Page), luego vamos a crear una serie de vistas, que llamaremos "paneles", que serán cajas ocultas (etiqueta div). Con un menú vamos mostrando una u otra vista (panel) según donde vamos pulsando.

Ejemplo de paneles (fichero index.html):

```
<div class="row">
  <div class="panel" id="panel_inicio">
    <h2>Modelo servicio REST</h2>
  </div>
  <div class="panel" id="panel_inst_read">
    <h2>Listado de instalaciones</h2>
    <div id="panel_inst_list" class="input-group mb-3">
        Aquí va el listado.
    </div>
  </div>
  <div class="panel" id="panel_inst_update">
    <h2>Actualización de una instalación</h2>
    <div class="input-group mb-3">
      <select class="custom-select" id="select_inst_update">
        <option>instalacion</option>
      </select>
      <button id="btn_inst_update" type="button" class="btn btn-primary">
        actualizar</button>
    </div>
```

```
    </div>
    <div class="panel" id="panel_inst_delete">
      <h2>Borrado de instalaciones</h2>
      <div class="input-group mb-3">
        <select class="custom-select" id="select_inst_delete">
          <option>instalacion</option>
        </select>
        <button id="btn_inst_delete" type="button" class="btn btn-danger">borrar</button>
      </div>
    </div>
    <div class="panel" id="panel_inst_create">
      <h2>Alta de instalacion</h2>
      <div class="input-group mb-3">
        <input type="text" class="form-control" id="nombre_inst_create" />
        <button id="btn_inst_create" type="button" class="btn btn-success">crear</button>
      </div>
    </div>
  </div>
```

Ahora con JavaScript (y jQuery) es muy fácil ocultar todos los DIV de la clase PANEL y mostrar únicamente la vista o panel que nos interese en cada momento. Bastaría con hacer un:

```
$(".panel").hide();
$("#id_panel_a_mostrar").show();
```

Fíjate bien en los ID que hemos dado a cada panel y ahora veamos el menú de la WebApp, observa también los ID de cada menú del CRUD:

```
  <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
    <a id="menu_inst_read" class="dropdown-item" href="#">Listado</a>
    <a id="menu_inst_create" class="dropdown-item" href="#">Alta</a>
    <a id="menu_inst_update" class="dropdown-item" href="#">Actualización</a>
    <a id="menu_inst_delete" class="dropdown-item" href="#">Baja</a>
  </div>
```

¿Has visto cómo cada menu_inst_* tiene su panel panel_inst_* ? Esto lo hemos hecho así para ahora con JavaScript conectar el evento de pulsar en un menú con su panel correspondiente (fichero www/js/controller.js):

```
$.controller.active_panel = "#panel_inicio";

$.controller.activate = function (panel_name) {
    $($.controller.active_panel).hide();
    $(panel_name).show();
    $.controller.active_panel = panel_name;
};
```

```
$.controller.init = function (panel_inicial) {
    $('[id^="menu_"]').each(function () {
        var $this = $(this);
        var menu_id = $this.attr('id');
        var panel_id = menu_id.replace('menu_', 'panel_');

        $("#" + menu_id).click(function () {
            $.controller.activate("#" + panel_id);
        });
    });
```

Ahora falta darle funcionalidad a cada botón del controlador para terminar el CRUD del servicio. Además de completar el código de cada evento "on click" (fichero www/js/controller.js), posiblemente tengamos que añadir algún panel más al fichero index.html:

```
    $.controller.init = function (panel_inicial) {

    ....

    $("#btn_inst_create").click(()=>{
        console.log("btn_inst_create-onClick::TODO:: falta llamar al doPost");
    });

    $("#btn_inst_update").click(()=>{
        console.log("btn_inst_update-onClick::TODO:: falta llamar al doGet "+
            "para poblar el formulario y luego hacer el doPut");
    });

    $("#btn_inst_delete").click(()=>{
        console.log("btn_inst_delete-onClick::TODO:: falta llamar al doDelete");
    });

};
```

Ya está hecho el **READ** del **CRUD de instalación** para ayudarte en la tarea, fíjate cómo se hace el "binding" de evento "pulsar sobre el menú read" y generar una tabla a partir del JSON que pedimos al Webservice.

```
    $("#menu_inst_read").click(()=>{
        console.log("listando instalaciones... ");
        $.controller.doGet($.controller.url+"instalacion", function(datos){
            console.log("listando instalaciones: "+datos);
            $("#panel_inst_read").empty();
            let tabla=$("<table/>");
            datos.forEach(instalacion => {
                let fila=$("<tr/>");
                fila.append("<td>"+"Instalación "+instalacion.id+"</td>");
```

```
                fila.append("<td>"+instalacion.name+"</td>");
                fila.append("<td>"+
                  "<span class=\"btn btn-success\">ACTUALIZAR</span> "+
                  "</td>");
                fila.append("<td><span class=\"btn btn-danger\">BORRAR</span> </td>");
                tabla.append(fila);
            });
            $("#panel_inst_read").append(tabla);
        });
    });
```

Localtunel

Para probar el servicio en Internet:

npm install -g localtunnel

lt –port 9090