

# Departamento de Informática

## Acceso a Bases de Datos Relacionales desde Node.JS

Juan Gualberto

Diciembre 2023

## Índice

<b>Proyecto: Gestión académica</b>	<b>2</b>
Infraestructura de contenedores . . . . .	3
Configuración del proyecto . . . . .	4
Diseño ER y creación de las tablas . . . . .	8
Configuración de la aplicación . . . . .	11
Carga y configuración de los “drivers” para MySQL . . . . .	12
Gestión de sesiones en Express . . . . .	12
Configurando el motor de plantillas . . . . .	13
Los middlewares y su uso para gestionar sesiones . . . . .	16
El enrutador para gestionar las entradas de usuario . . . . .	17
CRUD Alumnos . . . . .	21
FindAll() . . . . .	22
Save() . . . . .	23
Update() . . . . .	23
Delete() . . . . .	24
CRUD Asignaturas . . . . .	25
Maestro-detalle: matricular alumnos de asignaturas . . . . .	29
Ejercicios propuestos . . . . .	32

## Proyecto: Gestión académica

Nos piden hacer un sistema simple de gestión para un centro educativo. Solo necesitan almacenar información de profesores (nombre, apellidos y email), alumnos (nombre, apellido, teléfono y email) y asignaturas (nombre, curso y ciclo), teniendo en cuenta que una asignatura la pueden varios profesores (máximo dos) y se pueden matricular hasta hasta 32 alumnos.

Se debe implementar un sistema de login y proteger la sesión con él. El usuario que hace login puede ser administrativo, profesorado o alumnado (exclusivo). Los usuarios con perfil de administrativos (como mínimo hay que implementar este rol) pueden ver alumnos, profesores y asignaturas, así como matricular a alumnos de asignaturas y asignar profesores a las mismas.

BONUS: Los alumnos pueden ver sus asignaturas así como el profesorado que las imparte. El profesorado puede ver las asignaturas que imparte así como la lista de alumnos matriculados.

1. Entrega 1: Subir una foto del diagrama ER que resuelve el problema. Archivo diagrama.jpg
2. Entrega 2: Subir el SQL que resuelve el problema. Archivo bbdd.sql
3. Entrega 3: Subir el código de clase. Archivo clase.zip
4. Entrega 4: Subir el código con el login y el programa completo (sólo administrativos). Archivo programa.zip
5. Entrega 5: Subir el código con todos los tipos de usuarios implementados. Archivo bonus.zip

Hay que implementarlo con NodeJS, Pug, Express y MySQL. Se facilita contenedores docker y dependencias para el proyecto. No vamos a separar en diferentes archivos la lógica aún en este primer ejercicio.

Lo ideal sería estructurar el proyecto según responsabilidades, utilizando controladores para la lógica de negocio, un enrutador para manejar las rutas y un directorio para las vistas:

```
1 | -- /controllers
2 |   |-- alumnosController.js
3 |   |-- asignaturasController.js
4 |   |-- authController.js
5 | -- /middlewares
6 |   |-- sessionMiddleware.js
7 | -- /models
8 |   |-- alumnoModel.js
9 |   |-- asignaturaModel.js
10 | -- /routes
11 |   |-- alumnosRoutes.js
12 |   |-- asignaturasRoutes.js
13 |   |-- authRoutes.js
14 | -- /views
15 |   |-- alumnos
16 |     |-- index.pug
```

```
17 |         |-- add.pug
18 |         |-- edit.pug
19 |     |-- asignaturas
20 |         |-- index.pug
21 |         |-- add.pug
22 |         |-- edit.pug
23 |     |-- layout.pug
24 |-- app.js
25 |-- db.js
26 |-- package.json
27 |-- ...
```

- **controllers:** Contiene archivos que manejan la lógica de negocio para cada entidad o función específica.
- **middlewares:** Contiene archivos para los middlewares personalizados. Por ejemplo, el middleware de sesión podría ir aquí.
- **models:** Contiene archivos que definen los modelos de datos para interactuar con la base de datos.
- **routes:** Contiene archivos que definen las rutas y usan los controladores correspondientes.
- **views:** Contiene subdirectorios para cada entidad o función, y dentro de cada uno, las vistas relacionadas.
- **app.js:** Es el punto de entrada de la aplicación donde se configura Express y se definen las rutas principales.
- **db.js:** Contiene la configuración y la conexión a la base de datos.
- **package.json:** Archivo de configuración de Node.js.

Nosotros vamos, de momento, a mantener todas las rutas y controladores en el archivo **app.js**, cuando aprendamos herramientas ORM haremos esta separación como son las buenas prácticas.

**En los ejemplos verás que faltan algunos campos como teléfono, email, etc. Se ha hecho así para que seas tú quien lo complete e ir aprendiendo en vez de simplemente copiar y pegar código de estos apuntes.**

## Infraestructura de contenedores

Para montar la base de datos, en vez de instalar un servidor de MySQL en nuestro equipo, vamos a configurar un contenedor de MySQL con Adminer. Aquí tienes el fichero docker-compose.yml que

puedes alojar en la carpeta **stack** de tu proyecto para ayudar a calentar el plato más rápido cuando quieras retomarlo o un compañero de trabajo necesite tu proyecto.

Localiza las credenciales y los puertos expuestos en tu equipo anfitrión antes de comenzar para saber cómo configurar la conexión a la base de datos y cómo acceder desde **adminer** a **mysql** (fíjate que el servicio de MySQL se llama **gestion** luego en el campo servidor de *adminer* tendrás que escribir *gestion*).

```
1 version: "3"
2 services:
3   gestion:
4     image: mysql:latest
5     command: --default-authentication-plugin=mysql_native_password
6     restart: "no"
7     environment:
8       MYSQL_ROOT_PASSWORD: s83n38DGB8d72
9     ports:
10      - 33308:3306
11
12   adminer:
13     image: adminer:latest
14     restart: "no"
15     ports:
16      - 8181:8080
```

Para hacer login abrimos la URL: <http://localhost:8181>.

## Configuración del proyecto

En la carpeta del proyecto creamos el fichero **package.json**

```
1 npm install --save express express-session mysql2 pug body-parser
```

Esto añade a nuestro proyecto:

1. **express:** Este es el marco web de Node.js que te permite construir aplicaciones web y APIs de manera sencilla. Express proporciona una amplia gama de características para manejar rutas, manejo de middleware, gestionar sesiones y muchas otras utilidades para el desarrollo web.
2. **express-session:** Este paquete proporciona soporte para gestionar sesiones de usuario en Express. Permite almacenar datos de sesión en el servidor y asociar una cookie de sesión con el navegador del usuario. Puedes utilizarlo para implementar la autenticación y la persistencia de datos entre solicitudes del mismo usuario.
3. **mysql2:** Este es un controlador de MySQL para Node.js que te permite conectarte a una base de datos MySQL y realizar consultas. **mysql2** es una versión mejorada y más rápida de **mysql** y es compatible con las promesas.

# Login

<b>System</b>	MySQL
<b>Server</b>	gestion
<b>Username</b>	root
<b>Password</b>	.....
<b>Database</b>	

☐ Permanent login

**Figura 1:** Ejemplo de login en Adminer con las credenciales del fichero YAML

4. **pug:** Anteriormente conocido como Jade, Pug es un motor de plantillas para Node.js que simplifica la creación de vistas HTML. Puedes utilizarlo para escribir HTML de una manera más concisa y expresiva.
5. **body-parser:** Este middleware de Express se utiliza para analizar el cuerpo de las solicitudes HTTP. Es especialmente útil cuando necesitas acceder a los datos enviados en una solicitud POST, ya que facilita la extracción de información de formularios y carga útil JSON.

Es un ejemplo de dependencias de proyecto para construir aplicaciones web con Express, gestionar sesiones de usuario, conectarse a una base de datos MySQL, utilizar un motor de plantillas para las vistas y analizar el cuerpo de las solicitudes HTTP para extraer datos.

Esto genera, además de la carpeta **node\_modules**, un archivo **package.json**. En éste último archivo, deberías tener algo similar a esto:

```
1 {  
2   "dependencies": {  
3     "body-parser": "^1.20.2",  
4     "express": "^4.18.2",  
5     "express-session": "^1.17.3",  
6     "mysql2": "^3.6.5",  
7     "pug": "^3.0.2"  
8   }  
9 }
```

Dentro del archivo **package.json** de un proyecto Node.js, hay varias secciones y campos que especifican información sobre el proyecto y sus dependencias. Algunos de los campos más relevantes son:

1. **name:** Es el nombre del paquete. Debería ser único dentro del registro npm.
2. **version:** Indica la versión actual del paquete. Sigue un esquema de versión semántica (**SemVer**). Este número se incrementa cuando se realizan cambios en el código.
  - **Major (X):** Cambios incompatibles con versiones anteriores. Si cambia este número, nuestro proyecto ya no funciona.
  - **Minor (Y):** Nuevas características de forma compatible. En teoría debe funcionar igual.
  - **Patch (Z):** Correcciones de errores compatibles con versiones anteriores. Casi con toda seguridad funcionará.

Por ejemplo, 1.2.3 significa Major: 1, Minor: 2, Patch: 3.

3. **description:** Breve descripción del paquete.
4. **main:** El archivo principal que se ejecutará cuando alguien requiera tu módulo.
5. **scripts:** Una sección que define scripts de terminal que pueden ejecutarse con **npm run**.

6. **dependencies**: Lista de dependencias necesarias para que la aplicación funcione en producción.
7. **devDependencies**: Lista de dependencias necesarias solo para desarrollo.
8. **author**: El autor del paquete.
9. **license**: La licencia bajo la cual se distribuye el paquete.
10. **keywords**: Lista de palabras clave asociadas con el paquete.
11. **repository**: La ubicación del repositorio del código fuente del paquete.
12. **engines**: Restricciones sobre las versiones de Node.js y npm compatibles con el paquete.
13. **scripts**: Una sección que permite definir comandos personalizados que se pueden ejecutar con `npm run`.
14. **eslintConfig, browserslist, etc.**: Configuraciones específicas para herramientas como ESLint, Babel, etc.

Cada vez que instalas un paquete mediante `npm install`, se agrega una entrada en **dependencies**. Cuando instalas un paquete solo para desarrollo, se agrega a **devDependencies**. Estas secciones especifican las dependencias que se deben instalar para que la aplicación funcione correctamente.

Cuando alguien más descarga tu proyecto y ejecuta `npm install`, npm instalará las dependencias listadas en **dependencies** y **devDependencies** de acuerdo con las versiones especificadas. Esto garantiza que todos tengan el mismo entorno de desarrollo y producción.

**Las versiones de las dependencias** pueden tener diferentes **prefijos** que especifican cómo deben actualizarse esas dependencias en futuras instalaciones. Aquí hay una explicación de los prefijos más comunes:

1. **Sin prefijo (ningún carácter antes de la versión):**

- Ejemplo: `"express": "4.17.1"`
- Significado: La versión exacta especificada se instalará. No se realizarán actualizaciones automáticas.

2. **^ (caret):**

- Ejemplo: `"express": "^4.17.1"`
- Significado: Permite actualizaciones automáticas de parches. Cuando ejecutas `npm update`, npm instalará automáticamente la última versión compatible con la versión especificada, pero sin cambiar la versión principal ni la menor.

3. **~ (tilde):**



- Ejemplo: `"express": "~4.17.1"`
- Significado: Permite actualizaciones automáticas de parches y menores. npm instalará automáticamente la última versión compatible con la versión especificada, pero sin cambiar la versión principal.

#### 4. `>=`, `<=`, `<`, `>`:

- Ejemplo: `"express": ">=4.17.1 <5.0.0"`
- Significado: Establece un rango de versiones permitidas. En este caso, cualquier versión mayor o igual a 4.17.1 y menor que 5.0.0 será aceptada.

Estos prefijos se utilizan para indicar cómo deben manejarse las actualizaciones de las dependencias cuando se ejecuta `npm install` o `npm update`. La elección del prefijo depende de las necesidades y restricciones del proyecto. En entornos de producción, es común especificar versiones exactas o utilizar prefijos más restrictivos para evitar actualizaciones automáticas que podrían romper la compatibilidad. En entornos de desarrollo, los prefijos `^` o `~` son comunes para permitir actualizaciones automáticas mientras se mantenga la compatibilidad.

## Diseño ER y creación de las tablas

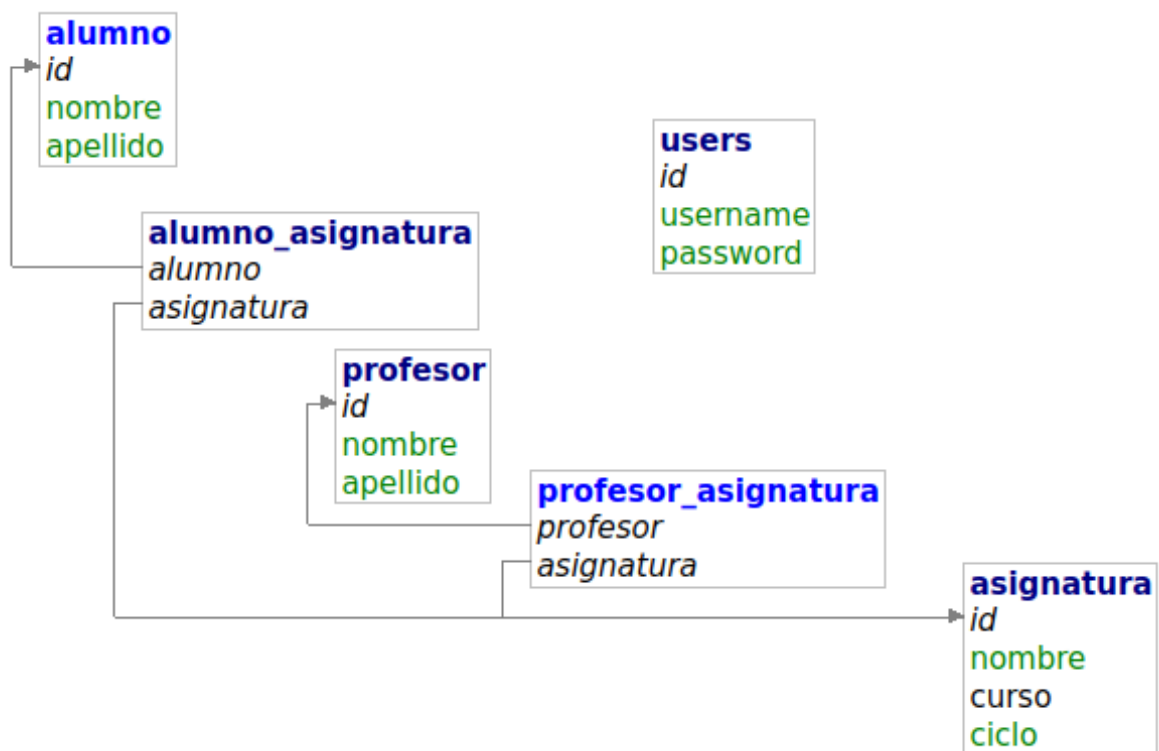
Las tablas son:

- **alumno**(id, nombre, apellido, email, teléfono)
- **profesor**(id, nombre, apellido, email)
- **asignatura**(id, nombre, ciclo, curso)
- **matricula**(alumno, asignatura)
- **impartir**(profesor, asignatura)
- **user**(id, username, passwd)

Es necesaria una tabla aparte para gestionar credenciales y mantenerlas a salvo de errores de programación (por ejemplo una API REST mal hecha de alumno o profesor puede exponer el repositorio en su totalidad y se vería comprometida no sólo la información del alumnado y profesorado, sino también sus credenciales).

Un ejemplo de SQL para generar las tablas, podría ser (faltan columnas como los correos electrónicos y los teléfonos):

```
1 CREATE DATABASE IF NOT EXISTS `gestion`;  
2  
3 USE `gestion`;  
4  
5 DROP TABLE IF EXISTS alumno_asignatura;  
6 DROP TABLE IF EXISTS profesor_asignatura;
```



**Figura 2:** Diagrama Entidad Relación de las tablas anteriores

```
7 DROP TABLE IF EXISTS alumno;
8 DROP TABLE IF EXISTS profesor;
9 DROP TABLE IF EXISTS asignatura;
10
11 -- para los logins
12 CREATE TABLE IF NOT EXISTS users (
13   `id` INT AUTO_INCREMENT PRIMARY KEY,
14   `username` VARCHAR(255) NOT NULL,
15   `password` VARCHAR(255) NOT NULL
16 );
17
18 INSERT INTO `users` (`username`, `password`)
19   VALUES ('pepe', 'Secreto_123');
20
21 CREATE TABLE profesor (
22   id INT AUTO_INCREMENT PRIMARY KEY,
23   nombre VARCHAR(25) NOT NULL ,
24   apellido VARCHAR(50) NOT NULL );
25
26 CREATE TABLE alumno (
27   id INT AUTO_INCREMENT PRIMARY KEY,
28   nombre VARCHAR(25) NOT NULL ,
29   apellido VARCHAR(50) NOT NULL );
30
31 CREATE TABLE asignatura (
32   id INT AUTO_INCREMENT PRIMARY KEY,
33   nombre VARCHAR(60) NOT NULL ,
34   curso SMALLINT,
35   ciclo VARCHAR(50));
36
37 CREATE TABLE alumno_asignatura (
38   alumno INT NOT NULL,
39   asignatura INT NOT NULL,
40   PRIMARY KEY (alumno, asignatura),
41   FOREIGN KEY (alumno) REFERENCES alumno(id) ,
42   FOREIGN KEY (asignatura) REFERENCES asignatura(id)
43 );
44
45 CREATE TABLE profesor_asignatura (
46   profesor INT NOT NULL,
47   asignatura INT NOT NULL,
48   PRIMARY KEY (profesor, asignatura),
49   FOREIGN KEY (profesor) REFERENCES profesor(id),
50   FOREIGN KEY (asignatura) REFERENCES asignatura(id)
51 );
```

Para limitar la cantidad de alumnos por asignatura a un máximo de 32, podemos hacer un disparador (trigger) en MySQL como este:

```
1 DELIMITER //
2
```

```
3 CREATE TRIGGER before_insert_alumno_asignatura
4 BEFORE INSERT ON alumno_asignatura
5 FOR EACH ROW
6 BEGIN
7     DECLARE alumno_count INT;
8     SET alumno_count = (SELECT COUNT(*) FROM alumno_asignatura WHERE
9                           asignatura = NEW.asignatura);
10
11     IF alumno_count >= 32 THEN
12         SIGNAL SQLSTATE '45000'
13         SET MESSAGE_TEXT = 'No se puede asignar más de 32 alumnos a la
14                             asignatura';
15     END IF;
16 END;
17 //
18
19 CREATE TRIGGER before_update_alumno_asignatura
20 BEFORE UPDATE ON alumno_asignatura
21 FOR EACH ROW
22 BEGIN
23     DECLARE alumno_count INT;
24     SET alumno_count = (SELECT COUNT(*) FROM alumno_asignatura WHERE
25                           asignatura = NEW.asignatura);
26
27     IF alumno_count >= 32 THEN
28         SIGNAL SQLSTATE '45000'
29         SET MESSAGE_TEXT = 'No se puede asignar más de 32 alumnos a la
30                             asignatura';
31     END IF;
32 END;
33 //
34
35 DELIMITER ;
```

## Configuración de la aplicación

Para comenzar, creamos el archivo app.js y en él escribimos:

```
1 const express = require('express');
2 const session = require('express-session');
3 const mysql = require('mysql2');
4 const bodyParser = require('body-parser');
5 const path = require('path');
```

Esto sirve para cargar las dependencias necesarias (como los imports de Java) que contienen los objetos, funciones y métodos que usaremos con posterioridad.

Creamos el objeto para la aplicación:

```
1 const app = express();
2 const port = 8000;
```

## Carga y configuración de los “drivers” para MySQL

Cargamos el “driver” MySQL y conectamos (recuerda que en el package.json tenemos la dependencia, echa un ojo al archivo). No vamos a usar herramientas ORM aún, eso lo veremos más adelante, con eso ya podemos conectar a la base de datos y ejecutar sentencias DDL y DML contra ella.

**Nota:** realmente es un objeto, no es como hacíamos en Java que tenemos la interfaz JDBC y el Driver MySQL que es una implementación para dicha interfaz y así conectábamos a bases de datos relacionales.

```
1 // Configuración de MySQL
2 const db = mysql.createConnection({
3   host: 'localhost',
4   port: 33308,
5   user: 'root',
6   password: 's83n38DGB8d72',
7   database: 'gestion',
8 });
9
10 // Conexión a MySQL
11 db.connect(err => {
12   if (err) {
13     console.error('Error al conectar a MySQL:', err);
14     return;
15   }
16   console.log('Conexión exitosa a MySQL');
17 });
```

## Gestión de sesiones en Express

A continuación añadimos el código para gestión de sesiones. Recuerda que habíamos añadido en el *package.json* la dependencia *express-session*:

```
1 // Configuración de sesiones
2 app.use(
3   session({
4     secret: 'un supersecreto inconfesable',
5     // Cambiar a una clave segura en producción
6     resave: false,
7     saveUninitialized: true,
8   })
9 );
```

```
9    );
```

## Configurando el motor de plantillas

Llega el momento de configurar Pug: Activamos **pug** como motor de plantillas HTML. La ruta por defecto es `views`, no hace falta indicarlo pero lo ponemos para saber cómo cambiarlo si fuera necesario.

```
1 app.set('view engine', 'pug');
2 app.set('views', path.join(__dirname, 'views'));
```

Configuramos el middleware para analizar el cuerpo de las solicitudes:

```
1 app.use(bodyParser.urlencoded({ extended: true }));
```

Vamos a crear la página por defecto de inicio. Para ello definimos la ruta `/` que nos lleva al **index**. Será necesario crear un fichero `index.pug` en las vistas. Añadimos esto al `app.js`:

```
1 // ruta por defecto
2 app.get('/', (req, res) => {
3     res.render('index', { user: req.session.user });
4 });
```

Si quieres probar lo que llevamos para ver cómo funciona, añade esta línea al final del archivo **app.js**:

```
1 // Iniciar el servidor
2 app.listen(port, () => {
3     console.log(`Servidor iniciado en http://localhost:${port}`);
4 });
```

**IMPORTANTE:** estas deben ser *siempre* las últimas líneas del servicio, cuando digamos que añadidas algo más, es encima de ellas.

Comenzamos con las vistas, para las vistas usamos Pug, anteriormente conocido como “Jade”, posiblemente el motor de plantillas más usado para Node.js y para la plataforma web en general. Permite a los desarrolladores definir la estructura de un documento HTML utilizando una sintaxis simplificada y expresiva mediante tabulaciones. En lugar de utilizar etiquetas HTML tradicionales, Pug utiliza sangrías y espacios para representar la jerarquía de los elementos en el documento.

Por refrescar un poco, un ejemplo simple en Pug podría ser:

```
1 html
2   head
3     title My Page
4   body
5     h1 Welcome to My Page
```

```
6   p This is a paragraph.
7   ul
8     li Item 1
9     li Item 2
```

Este código Pug se compilaría en HTML equivalente:

```
1  <html>
2    <head>
3      <title>My Page</title>
4    </head>
5    <body>
6      <h1>Welcome to My Page</h1>
7      <p>This is a paragraph.</p>
8      <ul>
9        <li>Item 1</li>
10       <li>Item 2</li>
11     </ul>
12   </body>
13 </html>
```

Pero volvamos a nuestro proyecto. Creamos el fichero **index.pug** en la carpeta **views** con este contenido:

```
1  html
2    head
3      title= pageTitle
4
5    body
6      nav
7        ul
8          li
9            a(href="/") Inicio
10         if user
11           li
12             a(href="/alumnos") Alumnos
13           li
14             a(href="/asignaturas") Asignaturas
15           li
16             a(href="/logout") Cerrar sesión
17         else
18           li
19             a(href="/login") Iniciar Sesión
20
21     h1
22       Bienvenido a Gestión Académica
23
24     p
25       seleccione una opción de la lista para continuar
26
27     footer
28       p IES Virgen del Carmen (2023)
```

Si te fijas hay muchas partes de la página que se podrían repetir, como la barra de navegación, el pie de página y, aunque no esté, la carga del JavaScript y CSS para que se vea correctamente la Web también (por ejemplo así será si usamos Bootstrap).

Para facilitar la reutilización de partes de la Web, Pug ofrece los **layouts**, que no son más que plantillas para no tener que repetir el mismo código en todas las vistas de la Web.

Así entonces creamos un archivo nuevo llamado **layout.pug** con el siguiente contenido:

```
1 // views/layout.pug
2 html
3   head
4     title= pageTitle
5
6   body
7     nav
8       ul
9         li
10          a(href="/") Inicio
11          if user
12            li
13              a(href="/alumnos") Alumnos
14            li
15              a(href="/asignaturas") Asignaturas
16            li
17              a(href="/logout") Cerrar sesión
18          else
19            li
20              a(href="/login") Iniciar Sesión
21
22   block content
23
24   footer
25     p IES Virgen del Carmen (2023)
```

Y modificamos el **index.pug** para adaptarlo al uso del layout así:

```
1 extends layout
2
3 block content
4   h1 Bienvenido, #{user || 'Invitado'}
5   if user
6     p
7       a(href='/logout') Cerrar Sesión
8   else
9     p
10      a(href='/login') Iniciar Sesión
```



## Los middlewares y su uso para gestionar sesiones

En Express, un middleware es una función que tiene acceso al objeto de solicitud (`req`), al objeto de respuesta (`res`), y a la siguiente función en la pila de middleware (`next`). Los middlewares en Express se utilizan para realizar tareas específicas durante el procesamiento de una solicitud, como modificar objetos de solicitud o respuesta, ejecutar código antes o después de la manipulación de rutas, y controlar el flujo de ejecución.

Los middlewares pueden realizar una variedad de tareas, como:

1. **Modificación de la solicitud y respuesta:** Pueden agregar, modificar o eliminar propiedades de los objetos `req` y `res`. Por ejemplo, parsear datos del cuerpo de la solicitud, agregar encabezados personalizados, etc.
2. **Ejecución de código antes de las rutas:** Pueden realizar acciones antes de que se ejecuten las rutas principales. Esto es útil para la autenticación, la autorización y otras tareas de preprocesamiento.
3. **Control del flujo de ejecución:** Pueden decidir si permitir que la solicitud continúe al siguiente middleware o ruta en la cadena, o si detener el flujo y enviar una respuesta al cliente.

Los middlewares se pueden utilizar de diversas maneras en Express:

- **Middleware de aplicación:** Se ejecutan en cada solicitud. Pueden ser configurados con `app.use()`.

```
1 const express = require('express');
2 const app = express();
3
4 // Middleware de aplicación
5 app.use((req, res, next) => {
6   // Código del middleware
7   next(); // Llama al siguiente middleware o ruta
8 });
```

- **Middleware de ruta:** Se aplican solo a rutas específicas.

```
1 app.get('/ruta', (req, res, next) => {
2   // Middleware de ruta
3   next();
4 });
```

- **Middleware de error:** Se utilizan para manejar errores en la aplicación. Se definen con cuatro parámetros (`err`, `req`, `res`, `next`).

```
1 app.use((err, req, res, next) => {
2   // Middleware de error
```

```
3   res.status(500).send('Algo salió mal!');
4 }));
```

Los middlewares se ejecutan en orden y la ejecución continúa hasta que se completa la cadena de middlewares o se llama a `next(err)` con un error (en caso de middleware de error).

Un ejemplo sencillo de middleware que registra la hora de cada solicitud puede verse así:

```
1 app.use((req, res, next) => {
2   console.log('Hora de la solicitud:', new Date());
3   next();
4 });
```

Este middleware se ejecutará en cada solicitud, registrando la hora antes de pasar al siguiente middleware o ruta.

**Volvamos a nuestro proyecto.** Vamos a crear un sencillo MiddleWare que compruebe si hemos hecho login o no. Recuerda que las peticiones HTTP son sin estado. Para ver si se han hecho unas operaciones u otras previamente necesitamos crear sesiones y almacenar en ellas información.

Para ver si estamos *loggeados* basta con comprobar si hemos grabado previamente, por tanto, una variable de sesión. Cuando hacemos *login*, vamos a crear una variable de sesión *user* que contiene el nombre de usuario. Si esa variable de sesión existe, se hizo login desde ese navegador. Si no existe, aún no se hizo. Esto hay que hacerlo antes del enrutador.

```
1 // Middleware para gestionar la sesión de usuario
2 app.use((req, res, next) => {
3   res.locals.user = req.session.user || null;
4   if (!req.session.user && !req.path.match("/login"))
5     res.redirect("/login")
6   else
7     next();
8 });
9
10 // RUTAS
11 // ruta por defecto
12 app.get('/', (req, res) => {
13   res.render('index');
14 });
```

## El enrutador para gestionar las entradas de usuario

Ya vimos anteriormente la ruta `/` por defecto. Ahora vamos a ver cómo añadir más rutas para la gestión de entrada y salida de usuarios en nuestra Web.

Encima del **app.listen** seguimos introduciendo rutas (per debajo de la ruta por defecto `/`):

```
1 // ruta por defecto
2 app.get('/', (req, res) => {
3   res.render('index');
4 });
5
6 // ruta para el login
7 app.get('/login', (req, res) => {
8   res.render('login');
9 });
10
11
12 app.post('/login', (req, res) => {
13   const { username, password } = req.body;
14
15   // Verificación de credenciales en MySQL
16   const query = 'SELECT * FROM users WHERE username = ? AND password = ?';
17   db.query(query, [username, password], (err, results) => {
18     if (err) {
19       console.error('Error al verificar las credenciales:', err);
20       res.render("error", {mensaje: "Credenciales no válidas."});
21     } else {
22       if (results.length > 0) {
23         req.session.user = username;
24         res.redirect('/');
25       } else {
26         res.redirect('/login');
27       }
28     }
29   });
30 });
31
32
33 app.get('/logout', (req, res) => {
34   req.session.destroy(err => {
35     if (err) res.render("error", {mensaje: err});
36     else res.redirect('/login');
37   });
38 });
```

### Vamos a explicar paso a paso este código:

- Definimos una ruta HTTP GET para “/login”. Cuando alguien accede a esta ruta, se renderiza la plantilla ‘login’ utilizando `res.render()`. Esto asume que se está utilizando un motor de plantillas como Pug (antes conocido como Jade), ya que `res.render()` generalmente se utiliza con este tipo de motores para renderizar vistas.

```
1 // Ruta para el login
2 app.get('/login', (req, res) => {
```

```
3   res.render('login');  
4 });
```

- Definimos una ruta HTTP POST para “/login”. Esta ruta maneja el envío de datos del formulario de inicio de sesión. Extrae el nombre de usuario y la contraseña de `req.body`, luego realiza una consulta a una base de datos MySQL para verificar las credenciales.
  - Si hay un error durante la consulta, se renderiza la plantilla de error con un mensaje.
  - Si las credenciales son válidas (es decir, si la consulta devuelve resultados), se establece una variable de sesión `req.session.user` con el nombre de usuario y se redirige al usuario a la ruta principal (“/”).
  - Si las credenciales no son válidas, se redirige al usuario nuevamente a la página de inicio de sesión (“/login”).

```
1 app.post('/login', (req, res) => {  
2   const { username, password } = req.body;  
3  
4   // Verificación de credenciales en MySQL  
5   const query = 'SELECT * FROM users WHERE username = ? AND password =  
6     ?';  
7   db.query(query, [username, password], (err, results) => {  
8     if (err) {  
9       console.error('Error al verificar las credenciales:', err);  
10      res.render("error", {mensaje: "Credenciales no válidas."});  
11    } else {  
12      if (results.length > 0) {  
13        req.session.user = username;  
14        res.redirect('/');  
15      } else {  
16        res.redirect('/login');  
17      }  
18    }  
19  });  
20 });
```

- Para terminar definimos una ruta HTTP GET para “/logout”. Cuando alguien accede a esta ruta, se destruye la sesión del usuario utilizando `req.session.destroy()`. Esto elimina la información de la sesión y, por lo tanto, “cierra sesión” en la aplicación.
  - Si hay algún error durante la destrucción de la sesión, se renderiza la plantilla de error con un mensaje.
  - Si la destrucción de la sesión tiene éxito, el usuario es redirigido a la página de inicio de sesión (“/login”).

```
1 app.get('/logout', (req, res) => {  
2   req.session.destroy(err => {
```

```
3     if (err) res.render("error", {mensaje: err});
4     else res.redirect('/login');
5   });
6 });
```

### Vamos con las vistas a las que se hace referencia.

Primero tenemos la vista de login. Es el archivo **login.pug**, créalo en la carpeta **views**:

```
1 extends layout
2
3 block content
4   h1 Iniciar Sesión
5   form(action='/login', method='post')
6     label(for='username') Usuario:
7     input(type='text', id='username', name='username', required)
8     br
9     label(for='password') Contraseña:
10    input(type='password', id='password', name='password',
11          required)
12    br
13    button(type='submit') Iniciar Sesión
```

Este archivo Pug extiende un diseño principal (`layout.pug`) y línea por línea lo que hacemos es:

- **extends layout**: Indica que esta plantilla Pug extiende el diseño principal llamado “layout”. Esto significa que heredará la estructura y el contenido definidos en el archivo “layout.pug”.
- **block content**: Este bloque define el contenido específico de esta vista y se insertará en el bloque correspondiente del diseño principal.
- **h1 Iniciar Sesión**: Encabezado que indica el propósito de la página: iniciar sesión.
- **form(action='/login', method='post')**: Un formulario HTML con la acción de enviar datos a la ruta “/login” mediante el método POST.
- **label (for='username') Usuario**:: Etiqueta para el campo de nombre de usuario.
- **input(type='text', id='username', name='username', required)**: Campo de entrada de texto para el nombre de usuario, marcado como obligatorio.
- **br**: Salto de línea.
- **label (for='password') Contraseña**:: Etiqueta para el campo de contraseña.
- **input(type='password', id='password', name='password', required)**: Campo de entrada de contraseña, marcado como obligatorio.
- **br**: Otro salto de línea.
- **button(type='submit') Iniciar Sesión**: Botón de envío del formulario.

Este archivo Pug crea una vista específica para la página de inicio de sesión, aprovechando el diseño principal definido en “layout.pug”. La estructura general de la página, como el encabezado, pie de página, y cualquier otro contenido común, será proporcionada por el diseño principal, mientras que

este archivo específico define el contenido único para la página de inicio de sesión.

Recuerda que incluimos `body-parser` como dependencia de nuestro proyecto y añadimos al principio configuración `urlencoded({ extended: true })` que se utiliza para analizar datos de formularios HTML. En este caso, al hacer el POST, `req.body` contendrá un objeto con propiedades correspondientes a los campos del formulario (`username` y `password` en este caso). **‘body-parser’ es crucial para manejar datos de formularios y otros datos en el cuerpo de las solicitudes POST. Sin él, los datos del formulario no se analizarán automáticamente y `req.body` sería `undefined` o estaría vacío.**

A continuación creamos el archivo **`error.pug`**. Se trata de la vista para mostrar errores por defecto. Fíjate cómo le hemos pasado el texto del mensaje en la variable **`mensaje`** desde la ruta a la vista.

```
1 extends layout
2
3 block content
4   h3 #{mensaje}
```

En el enrutador debemos crear la siguiente ruta:

```
1 app.get('/error', (req, res) => {
2   res.render('error');
3 });
```

## CRUD Alumnos

Por fin llegó la hora de nuestro primer CRUD (acrónimo de las operaciones básicas Create Read Update Delete). Comenzamos con los alumnos.

RUTA	VERBO	ACCIÓN	EXPLICACIÓN
/alumnos	GET	<code>findAll()</code>	Tabla con la lista de alumnos
/alumnos-add	GET	<code>save()</code>	Formulario para añadir alumnos
/alumnos-add	POST	<code>save()</code>	Formulario manda los datos y redirección
/alumnos-edit/:id	GET	<code>update()</code>	Formulario para modificar alumno con <code>ID=:id</code>
/alumnos-edit/:id	POST	<code>update()</code>	Formulario manda y actualiza alumno con <code>ID=:id</code>

RUTA	VERBO	ACCIÓN	EXPLICACIÓN
/alumnos-delete/:id	GET	delete()	Pregunta si borrar alumno con ID=:id
/alumnos-delete/:id	POST	update()	Borra alumno con ID=:id

## FindAll()

Primero vamos a definir la ruta para mostrar la lista de alumnos:

```

1 // Rutas
2 app.get('/alumnos', (req, res) => {
3   // Obtener todos los alumnos de la base de datos
4   db.query('SELECT * FROM alumno', (err, result) => {
5     if (err) res.render("error", {mensaje: err});
6     else res.render('alumnos', { alumnos: result });
7   });
8 });

```

La vista asociada es el archivo **alumnos.pug**:

```

1 extends layout
2
3 block content
4   h1 Lista de Alumnos
5   table
6     thead
7       tr
8         th ID
9         th Nombre
10        th Apellido
11        th Acciones
12      tbody
13        each alumno in alumnos
14          tr
15            td= alumno.id
16            td= alumno.nombre
17            td= alumno.apellido
18            td
19              a(href=`/alumnos-edit/${alumno.id}`) Editar
20              span |
21              a(href=`/alumnos-delete/${alumno.id}`) Eliminar
22      p
23        a(href='/alumnos-add') Agregar Nuevo Alumno

```

## Save()

Ahora vamos a ver las rutas para mostrar el formulario y el POST que recoge los datos del mismo:

```
1 app.get('/alumnos-add', (req, res) => {
2   res.render('alumnos-add');
3 });
4
5 app.post('/alumnos-add', (req, res) => {
6   // Insertar un nuevo alumno en la base de datos
7
8   const { nombre, apellido } = req.body;
9   db.query('INSERT INTO alumno (nombre, apellido) VALUES (?, ?)', [
10     nombre, apellido], (err, result) => {
11     if (err) res.render("error", {mensaje: err});
12     else res.redirect('/alumnos');
13   });
14 });
```

Como puedes ver, después de actualizar, si no hay error, nos lleva al listado de alumnos donde poder ver si se añadió correctamente.

El formulario es la vista **alumnos-add.pug**:

```
1 extends layout
2
3 block content
4   h1 Agregar Alumno
5   form(action='/alumnos-add', method='post')
6     label(for='nombre') Nombre:
7     input(type='text', id='nombre', name='nombre', required=true)
8     br
9     label(for='apellido') Apellido:
10    input(type='text', id='apellido', name='apellido', required=true)
11    br
12    button(type='submit') Agregar Alumno
13    br
14    a(href='/') Volver a la Lista
```

## Update()

Para actualizar, primero cargamos un formulario con los datos del alumno que queremos editar, y luego se actualizan en el POST, empezamos por las rutas:

```
1 app.get('/alumnos-edit/:id', (req, res) => {
2
3   const alumnoId = req.params.id;
```



```
4 // Obtener un alumno por su ID
5 db.query('SELECT * FROM alumno WHERE id = ?', [alumnoId], (err,
  result) => {
6   if (err) res.render("error", {mensaje: err});
7   else res.render('alumnos-edit', { alumno: result[0] });
8 });
9 });
10
11 app.post('/alumnos-edit/:id', (req, res) => {
12
13   const alumnoId = req.params.id;
14   // Actualizar un alumno por su ID
15   const { nombre, apellido } = req.body;
16   db.query('UPDATE alumno SET nombre = ?, apellido = ? WHERE id = ?',
    [nombre, apellido, alumnoId], (err, result) => {
17     if (err)
18       res.render("error", {mensaje: err});
19     else
20       res.redirect('/alumnos');
21   });
22
23 });
```

Ahora veamos el formulario **alumnos-edit.pug**:

```
1 extends layout
2
3 block content
4   h1 Editar Alumno
5   form(action=`/alumnos-edit/${alumno.id}`, method='post')
6     label(for='nombre') Nombre:
7     input.form-control(type='text', id='nombre', name='nombre', value=
      alumno.nombre, required=true)
8     br
9     label(for='apellido') Apellido:
10    input.form-control(type='text', id='apellido', name='apellido',
      value=alumno.apellido, required=true)
11    br
12    button(type='submit') Guardar Cambios
13    br
14    a(href='/alumnos') Volver a la Lista
```

## Delete()

A la hora de borrar vamos a pedir confirmación, haremos un GET donde se pregunte por ella y luego en un POST es donde realmente se lleva a cabo:

```
1 app.get('/alumnos-delete/:id', (req, res) => {
2
```

```
3     const alumnoId = req.params.id;
4     // Obtener y mostrar el alumno a eliminar
5     db.query('SELECT * FROM alumno WHERE id = ?', [alumnoId], (err,
6         result) => {
7         if (err)
8             res.render("error", {mensaje: err});
9         else
10            res.render('alumnos-delete', { alumno: result[0] });
11    });
12 });
13
14 app.post('/alumnos-delete/:id', (req, res) => {
15     const alumnoId = req.params.id;
16     // Eliminar un alumno por su ID
17     db.query('DELETE FROM alumno WHERE id = ?', [alumnoId], (err,
18         result) => {
19         if (err)
20             res.render("error", {mensaje: err});
21         else
22             res.redirect('/alumnos');
23     });
24 });
```

Ahora creamos el formulario de confirmación **alumnos-delete.pug**:

```
1 extends layout
2
3 block content
4     h1 Eliminar Alumno
5     p ¿Estás seguro de que deseas eliminar al alumno con el siguiente
6         detalle?
7     ul
8         li Nombre: #{alumno.nombre}
9         li Apellido: #{alumno.apellido}
10    form(action=`/alumnos-delete/${alumno.id}`, method='post')
11        button(type='submit') Eliminar Alumno
12    br
13    a(href='/alumnos') Cancelar
```

## CRUD Asignaturas

Ahora que ya sabemos cómo funcionan las rutas, intenta identificar cada uno de los pasos de cada CRUD en las rutas de asignaturas:

```
1 app.get('/asignaturas', (req, res)=> {
2     db.query('SELECT * FROM asignatura', (err, result) => {
3         if (err)
```

```
4     res.render("error", {mensaje: err});
5     else {
6         res.render("asignaturas", {asignaturas: result});
7     }
8 });
9 });
10
11 app.get('/asignaturas-add', (req, res)=> {
12     res.render("asignaturas-add");
13 });
14
15 app.post('/asignaturas-add', (req, res)=> {
16     // Insertar un nuevo alumno en la base de datos
17     const { nombre, ciclo, curso } = req.body;
18     db.query('INSERT INTO asignatura (nombre, ciclo, curso) VALUES (?, ?, ?)', [nombre, ciclo, curso], (err, result) => {
19         if (err) throw err;
20         res.redirect('/asignaturas');
21     });
22 });
23
24 app.get('/asignaturas-edit/:id', (req, res)=> {
25     const asignaturaId = req.params.id;
26     db.query('SELECT * FROM asignatura WHERE id = ?', [asignaturaId], (err, result) =>{
27         if (err)
28             res.render("error", {mensaje: err});
29         else
30             res.render("asignaturas-edit", {asignatura: result[0]});
31     });
32 });
33
34
35
36 app.post('/asignaturas-edit/:id', (req, res) => {
37
38     const asignaturaId = req.params.id;
39     // Actualizar un asignatura por su ID
40     const { nombre, ciclo, curso } = req.body;
41     db.query('UPDATE asignatura SET nombre = ?, ciclo = ?, curso = ? WHERE id = ?', [nombre, ciclo, curso, asignaturaId], (err, result) => {
42         if (err)
43             res.render("error", {mensaje: err});
44         else
45             res.redirect('/asignaturas');
46     });
47
48 });
49
50 app.get('/asignaturas-delete/:id', (req, res) => {
```

```
51
52   const asignaturaId = req.params.id;
53   // Obtener y mostrar el asignatura a eliminar
54   db.query('SELECT * FROM asignatura WHERE id = ?', [asignaturaId], (
55     err, result) => {
56     if (err)
57       res.render("error", {mensaje: err});
58     else
59       res.render('asignaturas-delete', { asignatura: result[0] });
60   });
61 });
62
63 app.post('/asignaturas-delete/:id', (req, res) => {
64   const asignaturaId = req.params.id;
65   // Eliminar un asignatura por su ID
66   db.query('DELETE FROM asignatura WHERE id = ?', [asignaturaId], (err,
67     result) => {
68     if (err)
69       res.render("error", {mensaje: err});
70     else
71       res.redirect('/asignaturas');
72   });
73 });
```

Vista **asignaturas.pug**:

```
1  extends layout
2
3  block content
4    h1 Lista de Asignaturas
5    table
6      thead
7        tr
8          th ID
9          th Nombre
10         th Ciclo
11         th Curso
12      tbody
13        each asignatura in asignaturas
14          tr
15            td= asignatura.id
16            td= asignatura.nombre
17            td= asignatura.ciclo
18            td= asignatura.curso
19            td
20              a(href=`/asignaturas-edit/${asignatura.id}`) Editar
21              span |
22              a(href=`/asignaturas-delete/${asignatura.id}`) Eliminar
23    p
```

```
24      a(href='/asignaturas-add') Agregar Nueva Asignatura
```

Vista **asignaturas-add.pug**:

```
1  extends layout
2
3  block content
4    h1 Agregar Asignatura
5    form(action='/asignaturas-add', method='post')
6      label(for='nombre') Nombre:
7      input(type='text', id='nombre', name='nombre', required=true)
8      label(for='curso') Curso:
9      input(type='text', id='curso', name='curso', required=true)
10     label(for='ciclo') Ciclo:
11     input(type='text', id='ciclo', name='ciclo', required=true)
12     br
13     button(type='submit') Agregar Asignatura
14   br
15   a(href='/asignaturas') Volver a la Lista
```

Vista **asignaturas-edit.pug**:

```
1  extends layout
2
3  block content
4    h1 Editar Asignatura
5    form(action='/asignaturas-edit/${asignatura.id}', method='post')
6      br
7      label(for='nombre') Nombre:
8      input.form-control(type='text', id='nombre', name='nombre', value=
9        asignatura.nombre, required=true)
10     br
11     label(for='curso') Curso:
12     input.form-control(type='text', id='curso', name='curso', value=
13       asignatura.curso, required=true)
14     br
15     label(for='ciclo') Ciclo:
16     input.form-control(type='text', id='ciclo', name='ciclo', value=
17       asignatura.ciclo, required=true)
18     br
19     button.btn.btn-primary(type='submit') Guardar Cambios
20     br
21     a(href='/asignaturas') Volver a la Lista
```

Vista **asignaturas-delete.pug**:

```
1  extends layout
2
3  block content
4    h1 Eliminar Asignatura
5    p ¿Estás seguro de que deseas eliminar la asignatura con el
```

```
    siguiente detalle?  
6    ul  
7      li Nombre: #{asignatura.nombre}  
8      li Ciclo: #{asignatura.ciclo}  
9      li Curso: #{asignatura.curso}  
10   form(action=`/asignaturas-delete/${asignatura.id}`, method='post')  
11     button(type='submit') Eliminar Asignatura  
12   br  
13   a(href='/asignaturas') Cancelar
```

## Maestro-detalle: matricular alumnos de asignaturas

Un maestro-detalle, en el contexto de bases de datos y desarrollo de aplicaciones, se refiere a una interfaz de usuario que permite interactuar con información relacionada en dos niveles: el nivel maestro y el nivel detalle.

- **Nivel Maestro:** Este nivel representa una entidad principal o principal en la base de datos. Por ejemplo, podría ser un formulario que muestra información sobre un cliente en un sistema de gestión de clientes.
- **Nivel Detalle:** Este nivel representa información detallada relacionada con la entidad maestra. En el ejemplo del cliente, el nivel detalle podría incluir información sobre las órdenes realizadas por ese cliente.

Un formulario maestro-detalle permite al usuario ver y editar información en ambos niveles al mismo tiempo. Por lo general, el formulario maestro mostrará información resumida o clave de la entidad maestra, mientras que el formulario detalle mostrará información más detallada y específica relacionada con la entidad maestra seleccionada.

En el contexto de una base de datos relacional, la relación maestro-detalle se establece a través de claves primarias y foráneas. Por ejemplo, en una base de datos de clientes y órdenes, la tabla de clientes podría ser la entidad maestra, y la tabla de órdenes sería la entidad detalle con una clave foránea que hace referencia a la clave primaria de la tabla de clientes.

Este tipo de formulario es común en sistemas de administración, como sistemas de gestión de bases de datos, sistemas de gestión de clientes (CRM), sistemas de gestión de inventario, entre otros, donde es crucial ver y manipular datos relacionados de manera eficiente.

Veamos un pequeño ejemplo con matrículas de alumnos (ver las asignaturas de las que está matriculado un alumno). Para poder comprobar cómo funciona vamos a hacer un formulario para matricular directamente y debajo el que lleva al maestro-detalle. Creamos las rutas:

```
1 // Rutas  
2 app.get('/matricular', (req, res) => {
```

```
3 // Obtener lista de alumnos y asignaturas
4 const queryAlumnos = 'SELECT * FROM alumno';
5 const queryAsignaturas = 'SELECT * FROM asignatura';
6
7 db.query(queryAlumnos, (errAlumnos, resultAlumnos) => {
8   if (errAlumnos) throw errAlumnos;
9
10  db.query(queryAsignaturas, (errAsignaturas, resultAsignaturas) => {
11    if (errAsignaturas) throw errAsignaturas;
12
13    res.render('matriculas', {
14      alumnos: resultAlumnos,
15      asignaturas: resultAsignaturas,
16    });
17  });
18 });
19 });
20
21 app.post('/matricular', (req, res) => {
22   const { alumno, asignatura } = req.body;
23
24   // Verificar si la matrícula ya existe
25   const queryExistencia = 'SELECT * FROM alumno_asignatura WHERE alumno
    = ? AND asignatura = ?';
26   db.query(queryExistencia, [alumno, asignatura], (errExistencia,
    resultExistencia) => {
27     if (errExistencia) throw errExistencia;
28
29     if (resultExistencia.length === 0) {
30       // Matricular al alumno en la asignatura
31       const queryMatricular = 'INSERT INTO alumno_asignatura (alumno,
    asignatura) VALUES (?, ?)';
32       db.query(queryMatricular, [alumno, asignatura], (errMatricular)
    => {
33         if (errMatricular) throw errMatricular;
34
35         res.redirect('/matricular');
36       });
37     } else {
38       // Matrícula ya existe
39       res.render('error', {mensaje: 'La matrícula ya existe'});
40     }
41   });
42 });
43
44 app.get('/asignaturas/:alumnoId', (req, res) => {
45   const alumnoId = req.params.alumnoId;
46
47   // Obtener asignaturas matriculadas para el alumno seleccionado
48   const queryAsignaturasMatriculadas = `
49     SELECT asignatura.nombre as asignatura, alumno.*
```

```

50     FROM asignatura, alumno, alumno_asignatura
51     WHERE alumno_asignatura.alumno = ?
52         AND asignatura.id = alumno_asignatura.asignatura
53         AND alumno.id = alumno_asignatura.alumno;`;
54
55     db.query(queryAsignaturasMatriculadas, [alumnoId], (err, result) => {
56         if (err) res.render('error', {mensaje: err});
57         else {
58             const asignaturas = result;
59             db.query('select * from alumno where alumno.id=?', [alumnoId], (
60                 err, result) => {
61                 if (err) res.render('error', {mensaje: err});
62                 else
63                     res.render('asignaturas-alumno', {alumno: result[0],
64                         asignaturasMatriculadas: asignaturas});
65             });
66         }
67     });
68 }
69 });
70 });

```

Archivo **matriculas.pug**:

```

1  extends layout
2
3  block content
4
5      h1 Gestión de asignaturas
6
7      h2 Matricular Alumnos en Asignaturas
8
9      form(action="/matricular", method="post")
10         label(for="alumno") Selecciona un alumno:
11         select(name="alumno" id="alumno")
12             each alumno in alumnos
13                 option(value=alumno.id)= `${alumno.nombre} ${alumno
14                     .apellido}`
15
16         label(for="asignatura") Selecciona una asignatura:
17         select(name="asignatura" id="asignatura")
18             each asignatura in asignaturas
19                 option(value=asignatura.id)= asignatura.nombre
20
21         button(type="submit") Matricular
22
23     hr
24
25     h2 Maestro detalle: Alumnos matriculados de una asignatura
26
27     label(for="alumnoAsignaturas") Selecciona un alumno para ver sus
28     asignaturas matriculadas:
29     select(name="alumnoAsignaturas" id="alumnoAsignaturas" onchange="

```



```
location = '/asignaturas/' + this.value;")
28     option
29     each alumno in alumnos
30         option(value=alumno.id)= ` ${alumno.nombre} ${alumno.
            apellido}`
```

Archivo **asignaturas-alumno.pug**:

```
1  extends layout
2
3  block content
4      h1 Asignaturas Matriculadas
5      h2 Alumno: #{alumno.nombre} #{alumno.apellido}
6      if asignaturasMatriculadas.length > 0
7          ul
8              each asignatura in asignaturasMatriculadas
9                  li= asignatura.asignatura
10     else
11         p El alumno no está matriculado en ninguna asignatura.
12
13     a(href="/matricular") Volver atrás
```

## Ejercicios propuestos

1. En los ejemplos verás que faltan algunos campos como teléfono, email, etc. Se ha hecho así para que seas tú quien lo complete e ir aprendiendo en vez de simplemente copiar y pegar código de estos apuntes. El primer ejercicio es completar los campos que faltan. (2p)
2. Hacer el CRUD de profesores. (2p)
3. Hacer la lógica para asignar profesores a asignaturas.(2p)
4. Crear un disparador para limitar a dos los profesores por asignatura.(1p)
5. **BONUS:** Implementar control de acceso por perfiles. El usuario que hace login puede ser administrativo, profesorado o alumnado (exclusivo). Los alumnos pueden ver sus asignaturas así como el profesorado que las imparte. El profesorado puede ver las asignaturas que imparte así como la lista de alumnos matriculados.(4p)

La puntuación es sobre 10 aunque los ejercicios sumen 11 puntos en total.