Clase 17: Material Complementario

Sitio:	Centro de E-Learning - UTN.BA	Imprimido por:	Nilo Crespi
Curso:	Curso de Backend Developer - Turno Noche	Día:	Friday, 23 de January de 2026, 10:32
Libro:	Clase 17: Material Complementario		

Tabla de contenidos

1. Integración de Node.js con Base de datos (parte 1)

Temario:

- Instalación de las dependencias.
- Conexión a la BD.
- Ejecución de consultas.

1.1. Integración de Node.js con Base de datos (parte 1)

Instalación de las dependencias

La manera más simple de instalar un módulo de Node.js de forma local es usar el comando **npm install** en la carpeta en la cual vamos a trabajar.

Esto combina dos pasos:

- Marca la última versión del módulo como una dependencia en tu archivo **package.json**.
- Descarga el módulo en tu directorio **node_modules**. Esto te permite usar el módulo cuando desarrollas de forma local.

Carga módulos Node.js

Utilizamos la función require() de Node.js para cargar cualquier módulo de Node.js que instalemos. También podemos usar la función require() para importar archivos locales que implementemos junto con nuestra función.

Dependencia mysql

Una de las maneras más fáciles de conectarse a MySQL es mediante el uso del módulo mysql . Esta dependencia maneja la conexión entre la aplicación Node.js y el servidor MySQL. Se instala así:

```
npm i mysql
```

Dependencia útil

La dependencia útil nos va a permitir, entre otras cosas, convertir ciertas funciones que utilizan callbacks al modelo de promesas asincrónico.

```
npm i util
```

Conexión a la BD

Archivo .env y variables de entorno

Comúnmente usamos archivos sin nombre y con la extensión .env para almacenar datos sensibles como contraseñas, claves de API o datos de conexión de nuestra aplicación sin tener que escribirlos en el código y de esa forma hacerlos públicos.

Estos archivos no suelen incluirse en sistemas de control de versiones como git.

Para utilizar los valores almacenados en este archivo instalaremos una librería llamada **dotenv**, cuya función es incorporar los contenidos de este archivo a las variables de entorno y de esta forma hacerlas disponibles para nuestra aplicación accediendo a las mismas mediante process.env..nombre_de_la_variable.

En este caso incluiremos los siguientes valores en nuestro archivo .env.

```
MYSQL_HOST=locahost
MYSQL_DB_NAME=nombre_de_las_base_de_datos
MYSQL_USER=root
MYSQL_PASSWORD=
```

Estos valores representan:

- **MYSQL_HOST:** el nombre de host de la base de datos a la que se está conectando.
- **MYSQL_DB_NAME:** nombre de la base de datos que se utilizará para esta conexión.
- **MYSQL_USER:** el usuario de MySQL con el que autenticarse.
- **MYSQL_PASSWORD**: es la contraseña de ese usuario de MySQL.

Conectándonos a la base de datos

Crearemos un archivo llamado bd.js en el cual utilizamos la función require() para cargar los módulos de mysql y útil.

Todas las consultas en la conexión de MySQL se realizan una tras otra. Esto significa que si desea hacer 10 consultas y cada consulta tarda 2 segundos, se tardará 20 segundos en completar toda la ejecución. La solución es crear 10 conexiones y ejecutar cada consulta en una conexión diferente. Esto se puede hacer automáticamente utilizando el conjunto de conexiones y se ejecutarán todas las 10 consultas en

paralelo.

Para esto, la librería de node.js mysql nos permite crear un pool o grupo de conexiones usando los datos de conexión que cargamos previamente.

Cuando usamos pool ya no necesitamos la conexión. Podemos consultar directamente al grupo de conexiones y el módulo MySQL buscará la siguiente conexión libre para ejecutar nuestra consulta.

```
var mysql = require('mysql');
var util = require('util');


var pool = mysql.createPool({
    connectionLimit: 10,
    host: process.env.MYSQL_HOST,
    user: process.env.MYSQL_USER,
    password: process.env.MYSQL_PASSWORD,
    database: process.env.MYSQL_DB_NAME
})

pool.query = util.promisify(pool.query);


module.exports = pool;
```

En este ejemplo creamos una variable llamada pool, donde guardamos la referencia al grupo de conexiones, utilizando los datos que la librería dotenv extrajo de nuestro archivo .env. Por último, tomamos el método query de pool y lo convertimos al modelo de promesas asincrónicas usando la librería util.

Después de esto solo queda exportar la variable pool para incluirla donde necesitemos hacer uso de la base de datos en nuestra aplicación.

Ejecución de consultas

SELECT

```
var pool = require('./bd');

pool.query("select * from alumnos").then(function(resultados){
    console.log(resultados);
});
```

INSERT

```
var pool = require('./bd');

var obj = {
    nombre: 'Juan',
    apellido: 'Lopez'
}

pool.query("insert into alumnos set ?", [obj]).then(function(resultados) {
    console.log(resultados);
});
```

UPDATE

```
var pool = require('./bd');

var id = 1;
var obj = {
    nombre: 'Pablo',
    apellido: 'Gomez'
}

pool.query("update alumnos set ? where id=?", [obj, id]).then(function(resultados) {

    console.log(resultados);
});
```

DELETE

```
var pool = require('./bd');

var id = 1;

pool.query("delete from alumnos where id = ?", [id]).then(function(resultados) {
    console.log(resultados);
});
```

Bibliografía:

- Google Cloud. Disponible desde la URL:

https://cloud.google.com/functions/docs/writing/specifying-dependencies-nodejs?hl=es-419

- npmjs. Disponible desde la URL: https://www.npmjs.com/

2. Integración de Node.js con Base de datos (parte 2)

Temario:

- Login.
- Logout.
- Páginas privadas.

2.1. Integración de Node.js con Base de datos (parte 2)

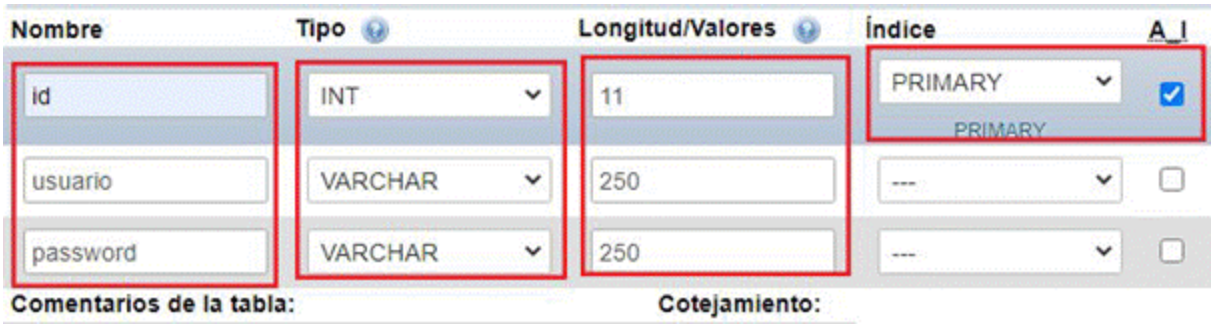
Login

Creamos la base de datos: cervecería. Para el charset o juego de caracteres seleccionamos utf8_unicode_ci que nos permitirá guardar acentos y caracteres especiales sin dificultad.

Con nuestra base de datos ya definida vamos a crear nuestra primera tabla, la cual va a almacenar a los usuarios que vamos a permitir ingresar a nuestro panel de administración.

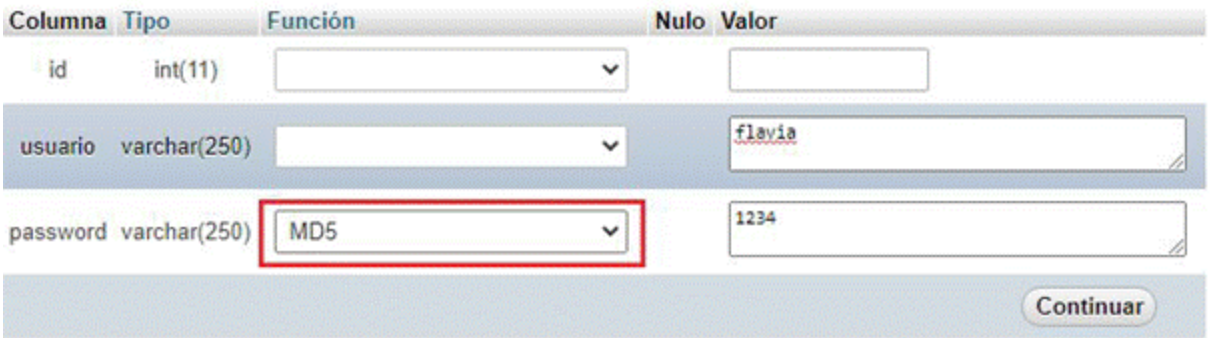
La tabla se llamará usuarios y contará con 3 campos:

- id: un entero de 11 dígitos que será nuestra clave primaria.
- usuario: un varchar de máximo 250 caracteres.
- password: otro varchar de máximo 250 caracteres



Cuando aceptemos estos valores tendremos acceso a la opción de menú insertar,

que nos permitirá agregar registros a nuestra tabla.



Es importante que en el campo password seleccionamos la función MD5 . Esta función es un algoritmo de encriptación de una sola vía, lo que significa que una vez encriptado el valor no puede ser revertido a un valor legible. Esto lo hacemos por cuestiones de seguridad, ya que si alguna persona llegara a tener acceso a nuestra base de datos, si no encriptan las contraseñas, tendría acceso a las claves de todos los usuarios de nuestra aplicación.

Una vez finalizada la creación de nuestra base de datos instalaremos las dependencias necesarias en nuestro proyecto para poder comenzar con la programación del login.

Instalaremos express-session, mysql, útil y md5 . A excepción de md5, el resto ya las hemos utilizado en ejemplos previos y conocemos su funcionamiento. md5 cumple la función de encriptar cadenas de texto usando el mismo algoritmo que usamos para encriptar la contraseña de nuestro usuario. En este caso la usaremos para encriptar lo que el usuario ingrese en el formulario de login y de esa manera comparar las 2 cadenas encriptadas para asegurarnos que el usuario sea quien dice ser.

Completados todos los pasos de preparación estamos listos para comenzar con la programación del login de nuestro panel de administración.

Paso 1

Crearemos el archivo routes/admin/login.js donde iremos incorporando las rutas relacionadas al proceso. Por el momento solo incluiremos este código.

```
var express = require('express');
var router = express.Router();


router.get('/', function (req, res, next) {
  res.render('admin/login', {
    layout: 'admin/layout',
  });
});

module.exports = router;
```

Paso 2

Importamos nuestro manejador de rutas en app.js y lo configuramos para que se encargue de las rutas que comienzan con /admin/login

```
var loginRouter = require('./routes/admin/login');


app.use('/admin/login', loginRouter);
```

Paso 3

Creamos el archivo views/admin/layout.hbs que será el nuevo layout base para nuestro administrador.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.
css"
        integrity="sha384-
JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z"
crossorigin="anonymous">
    <link href="https://fonts.googleapis.com/css?
family=Alfa+Slab+One|Open+Sans:400,700&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="/stylesheets/font-awesome.min.css">
    <link rel="stylesheet" href="/stylesheets/style.css">
    <title>Cerveceria X</title>
</head>
<body>
    <nav class="navbar fixed-top navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="/">BEER X</a>
    </nav>

    {{{body}}}
```

```
    <footer>
        <p class="text-center">Diseñado por: Flavia Ursino año</p>
    </footer>
    <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"
        integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
        crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
        integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
        crossorigin="anonymous"></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js
"
        integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6"
        crossorigin="anonymous"></script>
    <script
        src="https://cdn.jsdelivr.net/gh/cferdinandi/smooth-
scroll@15.0/dist/smooth-scroll.polyfills.min.js"></script>
</body>
</html>
```

Paso 4

Creamos el archivo views/admin/login.hbs que albergará el formulario de login para acceder al panel.

```
<div class="container">
  <div class="row" style="margin:200px 0">
    <div class="col">
      <form action="/admin/login" method="post">
        <div class="form-group">
          <input type="text" class="form-control" placeholder="Usuario"
                 name="usuario">
        </div>
        <div class="form-group">
          <input type="password" class="form-control"
                 placeholder="Password" name="password">
          </div>
        <button type="submit" class="btn btn-primary">Entrar</button>
      </form>
    </div>
  </div>
</div>
```

Paso 5

Para este paso es importante que tengamos configurados los archivos .env y bd.js con la configuración

correcta para conectarnos a nuestro servidor de MySQL local.

Debido a que vamos a empezar a trabajar con más archivos y a realizar más operaciones involucrando a la base de datos, y dichas operaciones se utilizarán en varios archivos, vamos a emplear el concepto de modelos de la arquitectura MVC.

Los modelos serán archivos donde crearemos funciones que nos permitan consultar, crear, actualizar o eliminar un tipo de dato en particular. De esta forma, al importar o requerir estas funciones en otros archivos como nuestros controladores tendremos acceso a la información sin necesidad de tipear varias veces lo mismo.

El primer archivo que crearemos con esta finalidad es models/usuariosModel.js, cuyo contenido será el siguiente:

```
var pool = require('./bd');
var md5 = require('md5');


async function getUserByUsernameAndPassword(user, password) {
    try {
        var query = "select * from usuarios where usuario = ? and password = ? limit
1";
        var rows = await pool.query(query, [user, md5(password)]);
        return rows[0];
    } catch (error) {
        throw error;
    }
}


module.exports = { getUserByUsernameAndPassword }
```

En este archivo vemos empleado el uso de la estructura try/catch que sirve para interceptar excepciones y que estas no lleguen al usuario. Las excepciones son errores que nuestra aplicación arroja y que podemos ir interceptando con distintos bloques try/catch y poder actuar en consecuencia.

La función getUserByUsernameAndPassword recibe como parámetros el nombre de usuario y la contraseña y devuelve, en caso de encontrar una coincidencia en la tabla usuarios de nuestra base de datos, la fila correspondiente como un objeto.

Paso 6

En el archivo routes/admin/login.js agregaremos el controlador necesario para capturar los datos enviados por el formulario. También vamos a importar el archivo usuariosModel para poder utilizar la función que creamos.

```
var usuariosModel = require('./../../models/usuariosModel');


router.post('/', async (req, res, next) => {
  try {
    var usuario = req.body.usuario;
    var password = req.body.password;

    var data = await
usuariosModel.getUserByUsernameAndPassword(usuario, password);

    if (data != undefined) {
      req.session.id_usuario = data.id;
      req.session.nombre = data.usuario;
      res.redirect('/admin/novedades');
    } else {
      res.render('admin/login', {
        layout: 'admin/layout',
        error: true
      });
    }

  } catch (error) {
    console.log(error);

  }
})
```

Como podemos ver, estamos capturando los valores enviados por el formulario y los almacenamos en variables que después pasamos como argumentos del método que creamos en el modelo de usuarios. En caso de que la respuesta sea diferente a undefined (la combinación de usuario y contraseña existe) vamos a crear las variables de sesión id_usuario y nombre y asignaremos su valor a los devueltos por la consulta a la base de datos. Por último redirigimos al usuario a hacia la ruta /admin/novedades.

Si no llega a haber coincidencia, ya sea porque el usuario escribio mal algún dato o porque directamente no existe esa combinación, haremos un render del template de admin/login.hbs especificando el nuevo layout de admin/layout.hbs y le enviaremos la variable error seteada en true para poder notificar al usuario de que hubo un error en el proceso de autenticación, para lo cual incluiremos el siguiente código en el template:

```
{{#if error}}
    <p>Usuario o clave incorrecto.</p>
{{/if}}
```

Paso 7

Creamos el archivo routes/admin/novedades.js con el siguiente contenido para mostrar el template correspondiente:

```
var express = require('express');
var router = express.Router();

router.get('/', function (req, res, next) {
   res.render('admin/novedades', {
    layout: 'admin/layout',
    usuario: req.session.nombre,
    });
});

module.exports = router;
```

En este archivo pasamos a la vista el nombre de usuario registrado en la sesión que acabamos de iniciar.

Paso 8

Creamos la vista para novedades en views/admin/novedades.hbs e ingresamos este contenido.

```
<div class="container" style="margin:50px auto">
    <div class="row">
        <div class="col">
            <p class="text-right">Hola {{ usuario }} ! </p>
        </div>
    </div>
</div>
```

Logout

Paso 1

En el archivo routes/admin/login.js creamos la función que se encargará de terminar la sesión del usuario.

```
router.get('/logout', function (req, res, next) {
  req.session.destroy();
  res.render('admin/login', {
    layout: 'admin/layout'
  });
});
```

El método destroy() del objeto sesión es el encargado de eliminar por completo la sesión del usuario y todos los valores almacenados en ella.

Paso 2

Modificamos nuestro archivo views/admin/novedades.hbs e incluimos un link a nuestra nueva función de logout.

```
<div class="row">
    <div class="col">
        <p class="text-right">Hola {{ usuario }} ! <a href="/admin/login/logout"
     class="btn btn-sm btn-danger">Cerrar sesión<i class="fa fa-sign-out"></i>
        </a></p>
    </div>
</div>
```

Páginas privadas

El proceso de autorización es el cual en el que se verifica no solo que el usuario haya iniciado correctamente (proceso de autenticación) sino que además tendremos la posibilidad de verificar (en casos más avanzados) que el usuario logueado tenga los permisos necesarios para realizar la acción que esté solicitando.

En nuestro ejercicio simplemente verificaremos que el usuario haya pasado por el proceso de autenticación, el cual deja como resultado algunas variables de sesion que podemos, cuya existencia podemos chequear en nuestras rutas para determinar si está o no logueado.

Paso 1

En el archivo app.js vamos a crear una nueva función y vamos a asignarla a una variable llamada secured.

```
secured = async(req,res,next) => {
    try{
        console.log(req.session.id_usuario);
        if(req.session.id_usuario){
            next();
        } else {
            res.redirect('/admin/login');
        }

    }catch(error){
        console.log(error);

    }
}
```

Esta función, que utilizaremos como middleware en todas las rutas que deseemos proteger, será la encargada de verificar que exista la variable de sesion id_usuario. En caso de encontrarla ejecutará la función next() que pasa el control de la petición a la siguiente función.

Si un usuario ingresa a una ruta protegida sin haber iniciado sesión correctamente interrumpimos el proceso de la petición y lo redirigimos hacia la pantalla de login. Para implementar esta nueva función en nuestras rutas simplemente debemos ponerla antes del manejador de ruta al momento de declararla.

```
app.use('/admin/novedades',secured, adminNovedadesRouter);
```

Bibliografía:

- Google Cloud. Disponible desde la URL:

https://cloud.google.com/functions/docs/writing/specifying-dependencies-nodejs?hl=es-419

- PhpMyAdmin. Disponible desde la URL: https://www.phpmyadmin.net/
- npmjs. Disponible desde la URL: https://www.npmjs.com/