

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Prácticas Iniciales Sección F
Ing. Herman Véliz



MANUAL TÉCNICO PRÁCTICA: DESARROLLO WEB

NOMBRE DE LOS INTEGRANTES DEL GRUPO:

1. 202111783_Samuel Barrera_58402853
2. 202212209_Juan Jose Almengor Tizol_51998837
3. 202202429_Diego Andrés Dubón Samayoa_30355643
4. 202201213_Juan Pablo Zuncar Aquino_55664767
5. 202200331_Eric David Rojas de León_50532549

NOMBRE DE LOS TUTORES:

1. 202003894_Estuardo Gabriel Son Mux_49849095
2. 202003959_Angel Eduardo Marroquin Canizales_42018782

Requisitos para poder utilizar el programa:

Sistema Operativo: tenemos que utilizar windows, o que Angular y Visual Studio Code funcionan sin problemas en cualquiera de los sistemas.

Procesador: Necesitarás un procesador de al menos dos núcleos, aunque cuanto más potente sea tu procesador, mejor será la experiencia.

Memoria RAM: Te recomiendo tener al menos 8 GB de RAM. .

Navegador web: Tendrás que contar con un navegador web actualizado, como Google Chrome o Mozilla Firefox, para probar la aplicación web.

Visual Studio Code: No te olvides de instalar Visual Studio Code en tu PC. Es una herramienta de desarrollo muy útil y altamente recomendada para proyectos de Angular y JavaScript.

Node.js y npm: Es imprescindible que tengas Node.js y npm instalados en tu sistema. Puedes pensar en ellos como el motor que impulsa tu desarrollo web. Puedes descargarlos fácilmente desde el sitio web oficial de Node.js.

Conexión a Internet: Una conexión de alta velocidad facilitará la descarga rápida de paquetes y dependencias.


Herramientas de control de versiones: No olvides utilizar una herramienta de control de versiones como Git y un servicio como GitHub o GitLab para gestionar el código fuente.

Software de gestión de bases de datos: MySQL, necesitarás tener un cliente de MySQL instalado en tu PC para administrar y consultar la base de datos.

Angular: Angular es un popular framework de desarrollo de aplicaciones web y móviles creado por Google. Proporciona un conjunto de herramientas y bibliotecas que facilitan la creación de aplicaciones web dinámicas y de una sola página (SPA), donde la mayor parte del procesamiento se realiza en el navegador del cliente.

Instalar Angular, pero el primer paso es tener instalado Node Js, para luego por comandos instalar angular.

Backend:

 `.gitignore``node_modules`

Esta carpeta alberga las dependencias del proyecto, que son bibliotecas y módulos que el proyecto necesita para funcionar correctamente. Sin embargo, no es necesario incluir esta carpeta en el repositorio de GitHub.

Para que otros desarrolladores puedan utilizar el proyecto, simplemente deben clonar el repositorio y luego ejecutar el comando “npm instal” en la consola. Este comando lee el archivo package.json y descarga automáticamente todas las dependencias necesarias, asegurando que tengan las mismas versiones que tengo en mi proyecto, pero sin cargar una carpeta pesada en GitHub, si no se descargan no se podrá utilizar el proyecto.

```
class ConsultasController {
  index(req, res) {
    res.json({ "Controlador": 'Funciona la API' });
  }
  holamundo(req, res) {
    res.json({ "Controlador": 'Hola Mundo' });
  }
  getuser(req, res) {
    return __awaiter(this, void 0, void 0, function* () {
      try {
        const conexion = yield promise_1.default.createConnection(config);
        const consulta = `select * from usuarios`;
        const result = yield conexion.query(consulta);
        const respuesta = result[0];
        yield conexion.end();
        return res.status(200).json({ respuesta });
      }
      catch (error) {
        return res.status(500).json({ "error": 'ya no sale' });
      }
    });
  }
}
exports.consultasController = new ConsultasController();
```

creamos un controlador para una API web en Node.js utilizando Express. Utiliza el módulo "mysql2/promise" para interactuar con una base de datos MySQL de forma asincrónica. La clase ConsultasController define tres métodos:

index(req, res): Devuelve un mensaje JSON que dice "Funciona la API".

holamundo(req, res): Devuelve un mensaje JSON que dice "Hola Mundo".

getuser(req, res): Consulta una base de datos MySQL para obtener todos los usuarios y devuelve los resultados como JSON. Si hay un error, devuelve un mensaje de error.

En resumen, este código configura un controlador para una API que puede proporcionar información de usuarios desde una base de datos MySQL.

```

"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
const express_1 = require("express");
const ConsultasController_1 = require("../Controllers/ConsultasController");
class IndexRoutes {
    constructor() {
        this.router = (0, express_1.Router)();
        this.config();
    }
    config() {
        this.router.get('/', ConsultasController_1.consultasController.index);
        this.router.get('/mundohola', ConsultasController_1.consultasController.holamundo);
        this.router.get('/ejemplo', ConsultasController_1.consultasController.getuser);
    }
}
const indexRoutes = new IndexRoutes();
exports.default = indexRoutes.router;

```

use strict: Esta línea habilita el "modo estricto" en JavaScript, que impone reglas más estrictas para escribir un código más limpio y seguro.

Object.defineProperty(exports, "__esModule", { value: true });: Esta línea parece estar relacionada con la exportación de módulos en CommonJS y señala que el módulo es un módulo ES6 válido.

Se importan varios módulos de Express y un controlador llamado ConsultasController que se encuentra en la ruta "../Controllers/ConsultasController".

Se define una clase llamada IndexRoutes que inicializa un enrutador Express en su constructor y configura las rutas en el método config().

En el método config(), se especifican tres rutas para la API:

/: Dirigirá las solicitudes a ConsultasController.index.

/mundohola: Dirigirá las solicitudes a ConsultasController.holamundo.

/ejemplo: Dirigirá las solicitudes a ConsultasController.getuser.

Finalmente, se crea una instancia de IndexRoutes llamada indexRoutes, y se exporta el enrutador de esta instancia como el valor predeterminado del módulo.

```

{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon build/index",
    "build": "tsc -w"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parse": "^0.1.0",
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "morgan": "^1.10.0",
    "mysql2": "^3.6.1",
    "nodemon": "^3.0.1",
    "typescript": "^5.2.2"
  },
  "devDependencies": {
    "@types/body-parser": "^1.19.2",
    "@types/cors": "^2.8.14",
    "@types/express": "^4.17.17",
    "@types/morgan": "^1.9.5"
  }
}

```

"name": Define el nombre del proyecto como "backend".

"version": Establece la versión del proyecto en "1.0.0".

"description": Puede incluir una descripción del proyecto, pero en este caso, está vacía.

"main": Especifica el archivo principal del proyecto como "index.js".

"scripts": Aquí se definen varios scripts que se pueden ejecutar desde la línea de comandos:

"test": Un script que simplemente imprime un mensaje de error si no se especifican pruebas y luego sale con un código de error.

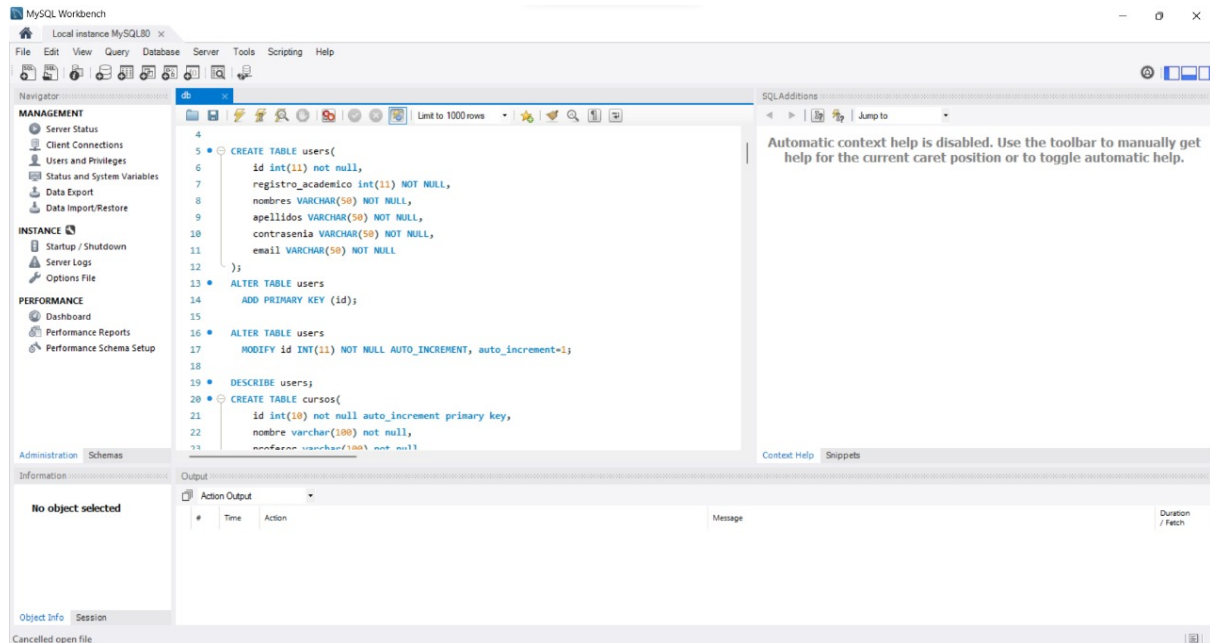
"start": Este script inicia la aplicación utilizando nodemon y ejecuta el archivo "build/index" para mantener la aplicación en ejecución durante el desarrollo.

"build": Este script utiliza el compilador TypeScript (tsc) en modo observador ("-w") para compilar automáticamente el código TypeScript en JavaScript.

"keywords": Puede incluir palabras clave relacionadas con el proyecto, pero en este caso, está vacío.

"author": El campo del autor está vacío en este caso.

"license": El proyecto utiliza la licencia "ISC", que es una licencia de código abierto.



DROP DATABASE IF EXISTS database_links;: Esta línea verifica si existe una base de datos llamada "database_links" y, si existe, la elimina. Esto se hace para asegurarse de que no haya ninguna base de datos con ese nombre antes de crear una nueva.

CREATE DATABASE database_links;: Esta línea crea una nueva base de datos llamada "database_links". La base de datos es donde se almacenarán las tablas y los datos relacionados.

USE database_links;: Esta línea selecciona la base de datos "database_links" como la base de datos activa en la que se realizarán las operaciones posteriores.

CREATE TABLE users(...);: Aquí se está creando una tabla llamada "users" con varias columnas. Esta tabla se utilizará para almacenar información sobre usuarios. Las columnas incluyen "id" (identificador único), "registro_academico" (número de registro académico), "nombres", "apellidos", "contrasenia" (contraseña) y "email".

ALTER TABLE users ADD PRIMARY KEY (id);: Esta línea establece la columna "id" como la clave primaria de la tabla "users". La clave primaria garantiza que cada registro en la tabla tenga un valor único en la columna "id".

ALTER TABLE users MODIFY id INT(11) NOT NULL AUTO_INCREMENT, auto_increment=1;: Esto modifica la columna "id" para que sea de tipo INT (entero) y le asigna la propiedad AUTO_INCREMENT. Esto significa que cada vez que se inserta un nuevo registro en la tabla, el valor de "id" se incrementará automáticamente en 1 y no puede ser nulo.

DESCRIBE users;: Esta línea muestra la descripción de la estructura de la tabla "users", lo que significa que mostrará información sobre las columnas, sus tipos de datos y restricciones.

CREATE TABLE cursos(...);: Similar a lo que se hizo con la tabla "users", aquí se crea una tabla llamada "cursos" con las columnas "id" (clave primaria), "nombre" (nombre del curso), "profesor" (nombre del profesor del curso), "practicante" (nombre del practicante del curso) y "detalles" (detalles del curso).

INSERT INTO cursos(...) VALUES();: Esta línea es un intento de insertar un nuevo registro en la tabla "cursos". Sin embargo, falta la especificación de los valores que se deben insertar entre los paréntesis en "VALUES()". Debes proporcionar los valores que correspondan a las columnas de la tabla para que esta inserción sea exitosa.

Frontend:

```
1  <app-navbar></app-navbar>
2  <br/>
3  <div class="container">
4    <div class="card-body text-center">
5      <h3 class="card-title">Bienvenido Usuario</h3>
6      <br/>
7      <h4 class="card-title">Usuarios</h4>
8      <p class="card-text">Vista general de los Usuarios en el sistema.</p>
9    </div>
10   <div class="card">
11     <table class="table table-hover">
12       <thead>
13         <tr>
14           <th scope="col">ID</th>
15           <th scope="col">Email</th>
16           <th scope="col">Contraseña</th>
17         </tr>
18       </thead>
19       <tbody>
20         <tr *ngFor="let user of usuarios">
21           <td>{{ user.id }}</td>
22           <td>{{ user.correo }}</td>
23           <td>{{ user.psw }}</td>
24         </tr>
25       </tbody>
26     </table>
27   </div>
28 </div>
29 <br/>
```

<div class="container">:

- Este <div> con la clase "container" se utiliza para encapsular y aplicar estilos de diseño específicos al contenido. En muchas bibliotecas de CSS, como Bootstrap, la clase "container" se usa para centrar y limitar el ancho del contenido en la pantalla.

```
<div class="card-body text-center">
```

- Este `<div>` con las clases "card-body" y "text-center" se utiliza para crear un contenedor que tiene un estilo de tarjeta y el contenido centrado en el centro de la tarjeta.

En resumen, este código HTML y Angular se utiliza para mostrar una página web que incluye una barra de navegación, un mensaje de bienvenida, una descripción y una tabla de usuarios con información como ID, correo y contraseña. La tabla se genera dinámicamente a partir de una lista de usuarios utilizando la directiva `*ngFor`.

```
describe('DashboardcursosComponent', () => {  
  let component: DashboardcursosComponent;  
  let fixture: ComponentFixture<DashboardcursosComponent>;  
  |
```

Estas variables se declaran para almacenar una instancia del componente (`component`) y una instancia de la fixture de componente (`fixture`). La fixture se utiliza para interactuar con el componente durante las pruebas.

```
beforeEach(() => {  
  TestBed.configureTestingModule({  
    declarations: [DashboardcursosComponent]  
  });  
  fixture = TestBed.createComponent(DashboardcursosComponent);  
  component = fixture.componentInstance;  
  fixture.detectChanges();  
});
```

El bloque `beforeEach` se ejecuta antes de cada prueba en la suite y se encarga de configurar el entorno de prueba.

- `TestBed.configureTestingModule({ declarations: [DashboardcursosComponent] });` Aquí se configura el módulo de prueba utilizando `TestBed`. Se declara el componente `DashboardcursosComponent` para que esté disponible en el entorno de prueba.
- `fixture = TestBed.createComponent(DashboardcursosComponent);` Se crea una instancia de la fixture del componente `DashboardcursosComponent`. Esto proporciona un contexto de prueba para el componente.
- `component = fixture.componentInstance;` Se obtiene una referencia a la instancia del componente que se va a probar. Esto permite interactuar directamente con las propiedades y métodos del componente en las pruebas.

- `fixture.detectChanges()` ;: Detecta los cambios iniciales en el componente. Esto es necesario para asegurarse de que cualquier inicialización y enlace de datos se realice antes de las pruebas.

```
export class DashboardcursosComponent {  
  
  usuarios: any[] = [];  
  info: any;
```

Estas son las propiedades del componente:

- `usuarios` es un arreglo que se utilizará para almacenar los datos de los usuarios.
- `info` es una variable que contendrá la información obtenida del servicio.

```
constructor(private backend: BackendService, private router: Router, private ruta: ActivatedRoute) { } //crear constructor
```

Este es el constructor del componente. Recibe tres argumentos: `backend` (una instancia del servicio `BackendService`), `router` (para la navegación) y `ruta` (para acceder a los parámetros de la ruta actual).

```
describe('NavbarComponent', () => {  
  let component: NavbarComponent;  
  let fixture: ComponentFixture<NavbarComponent>;
```

Estas variables se declaran para almacenar una instancia del componente (`component`) y una instancia de la fixture de componente (`fixture`). La fixture se utiliza para interactuar con el componente durante las pruebas.

```
beforeEach(() => {  
  TestBed.configureTestingModule({  
    declarations: [NavbarComponent]  
  });  
  fixture = TestBed.createComponent(NavbarComponent);  
  component = fixture.componentInstance;  
  fixture.detectChanges();  
});
```

El bloque `beforeEach` se ejecuta antes de cada prueba en la suite y se encarga de configurar el entorno de prueba.

- `TestBed.configureTestingModule({ declarations: [NavbarComponent] });` Aquí se configura el módulo de prueba utilizando `TestBed`. Se declara el componente `NavbarComponent` para que esté disponible en el entorno de prueba.
- `fixture = TestBed.createComponent(NavbarComponent);` Se crea una instancia de la fixture del componente `NavbarComponent`. Esto proporciona un contexto de prueba para el componente.
- `component = fixture.componentInstance;` Se obtiene una referencia a la instancia del componente que se va a probar. Esto permite interactuar directamente con las propiedades y métodos del componente en las pruebas.
- `fixture.detectChanges();` Detecta los cambios iniciales en el componente. Esto es necesario para asegurarse de que cualquier inicialización y enlace de datos se realice antes de las pruebas.