
5.1 Aplicación con conexión a Base de Datos

Asignatura

TALLER DE BASE DE DATOS

UNIDAD

“5”

Docente:

CARDOSO JIMÉNEZ AMBROSIO

Alumno:

BRYAN JIMENEZ CRUZ

Link:

<https://github.com/zurcjb/base-de-datos-.git>

INGENIERÍA INFORMÁTICA

SEMESTRE: IV

GRUPO: “A”

PERIODO: ENERO-JUNIO

Oaxaca, Oaxaca de Juárez a 30 de mayo de 2025

CONFIGURACIÓN E INSTALACIÓN DEL “BUN”

🌈 INSTALACION Y CONFIGURACION DEL RUN'POSTGRES "BUN":

Bun es un entorno de ejecución de JavaScript, similar al de Node.js pero con enfoque en velocidad y simplicidad, es un kit de herramientas que incluye un gestor de paquetes, un empaquetador, un ejecutor de pruebas, lo que ayuda a un desarrollador web

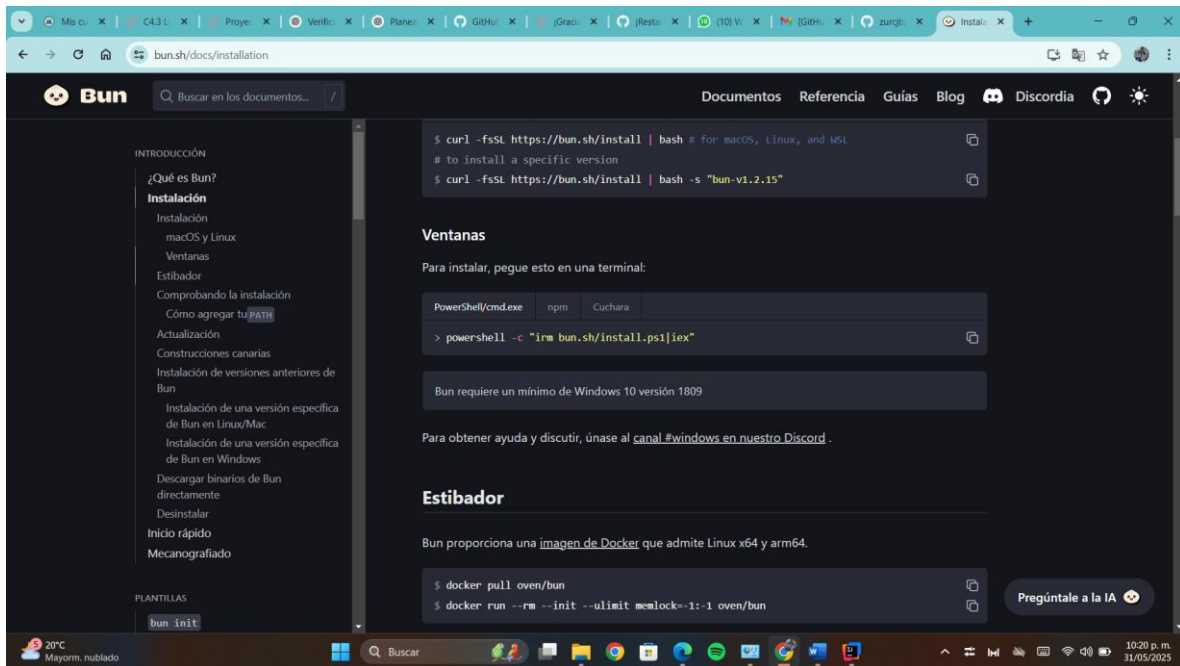
Pasos para la instalacion de “Bun”:

- ✓ Nos dirigimos hacia la pagina de doumentacion de instalacion “Bun” :
<https://bun.sh/docs/installation> (Revise imagen 1).

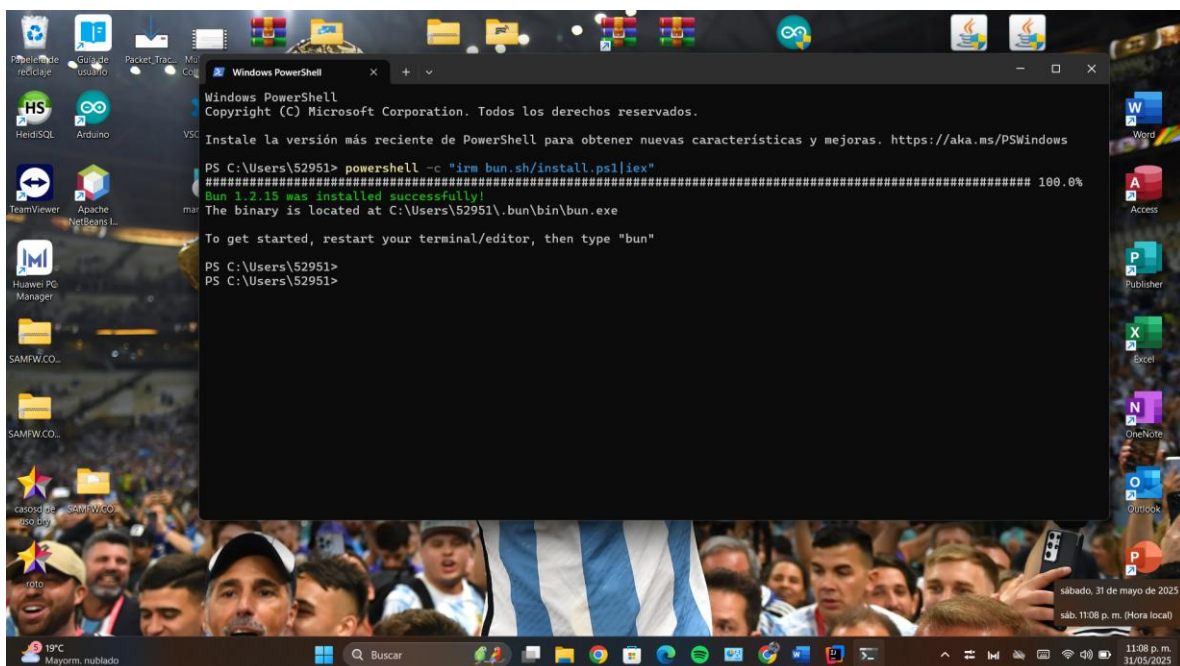
The screenshot shows the Bun website with a dark theme. At the top, there's a navigation bar with links to 'Documentos', 'Referencia', 'Guías', 'Blog', 'Discordia', and a GitHub icon. The main heading reads 'Bun es un rápido tiempo de ejecución'. Below this, a paragraph describes Bun as a JavaScript and TypeScript environment, toolchain, and package manager. To the right, there's a section titled 'Express.js "Hola mundo"' comparing Bun's performance to Node.js and Deno. A bar chart shows Bun leading with 59,026 requests per second, followed by Deno at 25,335 and Node.js at 19,039. At the bottom, there's a terminal snippet showing the installation command for Windows: `> powershell -c "irm bun.sh/install.ps1 | iex"`.

Run-time	Version	Requests per second (Linux x64)
Bun	v1.2.15	59,026
Deno	version 2.1.6	25,335
Node.js	version 18.12.0	19,039

- ✓ Una vez ya dentro, dependiendo del sistema operativo de la laptop, en caso de Windows, nos dirigiremos a los códigos que nos proporciona para su instalación desde Powershell o CMD.



- ✓ Copiaremos el código de powershell abriremos una nueva ventana de este mismo programa y pegaremos el código (Una vez hecho, comenzara a descargar “bun” deberá de esperar a que complete el 100%) una vez hecho, para revisar si se instaló la última versión metemos el siguiente comando.



- ✓ Una vez ya descargado, Nos dirigiremos a un directorio para instalarlo, en una carpeta donde viene un proyecto ya previsto de la base de datos
- ✓ Una vez ya abierto el cmd, escribiremos el comando “bun init” es un comando que se utiliza para crear un nuevo proyecto.

```
C:\Windows\System32\cmd.exe - bun add drizzle-orm pg bun dotenv
Microsoft Windows [Versión 10.0.19045.5737]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\koda\Documents\Javascript_postgres>bun init

Select a project template: Blank

+ .gitignore
+ index.ts
+ tsconfig.json (for editor autocomplete)
+ README.md

To get started, run:
```

- ✓ Una vez creado colocaremos el siguiente comando: “bun add drizzle-orm pg bun dotenv” para instalar varias dependencias necesarias para trabajar con Drizzle ORM, PostgreSQL y variables de entorno.

```
C:\Users\koda\Documents\Javascript_postgres>bun add drizzle-orm pg bun dotenv
bun add v1.2.12 (32a47ae4)

Installed drizzle-orm@0.43.1
Installed pg@8.15.6
Installed bun@1.2.12 with binaries:
- bun
- bunx
Installed dotenv@16.5.0

3 packages installed [188.51s]

C:\Users\koda\Documents\Javascript_postgres>
```

```
C:\Windows\System32\cmd.exe - exit server.ts
Microsoft Windows [Versión 10.0.19045.5737]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\koda\Documents\Javascript_postgres-main>bun server.ts
Error: Cannot find module 'dotenv/config' from 'C:\Users\koda\Documents\Javascript_postgres-main\server.ts'

C:\Users\koda\Documents\Javascript_postgres-main>bun add dotenv
bun add v1.2.12 (32a47ae4)
drizzle-orm... warn: bun's postinstall cost you 30.2s

typescript@5.5.3
@types/bun@1.2.9
@types/node@22.14.0 (22.15.17 available)
@types/pg@8.15.1
@hono/zod-validator@0.4.3 (0.5.0 available)
bun@1.2.9
drizzle-orm@0.43.0
hono@4.7.9
pg@8.14.1
rod@1.24.4

Installed dotenv@16.5.0

3 packages installed [100.09s]

C:\Users\koda\Documents\Javascript_postgres-main>bun server.ts
```

Esta es la forma con la que se conecta hacia el localhost, más adelante se mostrara la conexión y creación de archivos en “IntelliJ IDEA” donde se mostrara el código de como se hace la ejecución de la conexión con la base de datos y podemos acceder a las tablas por medio de nuestro navegador preferido.

CONEXION DE BASE DE DATOS

"JAVASCRIPT - POSTGRES"

Para la conexión de base de datos, estaremos utilizando el programa de IntelliJ IDEA, es un IDE (Integrated Development Environment) o entorno de desarrollo integrado, creado por JetBrains especialmente para el desarrollo de software, especialmente en Java y Kotlin.

Primero veremos la estructura de las carpetas:

SRC: Carpeta principal que contiene todo el código fuente del proyecto

Controollers: Maneja la lógica de las solicitudes HTTP (req/res).

db: Contiene la configuración de la base de datos (PostgreSQL).

Middlewares: Funciones intermedias que procesan solicitudes antes de llegar al controlador (ej: autenticación, validación).

Repositories: Capa de acceso directo a la base de datos. Aquí se escriben las consultas SQL o se usan ORMs.

Routes: Define las rutas de la API y las asocia a los controladores.

schemas: Define la estructura de datos.

services: Procesar datos antes de guardarlos en la base de datos.

utils: Funciones auxiliares reutilizables

Archivos en la raíz:

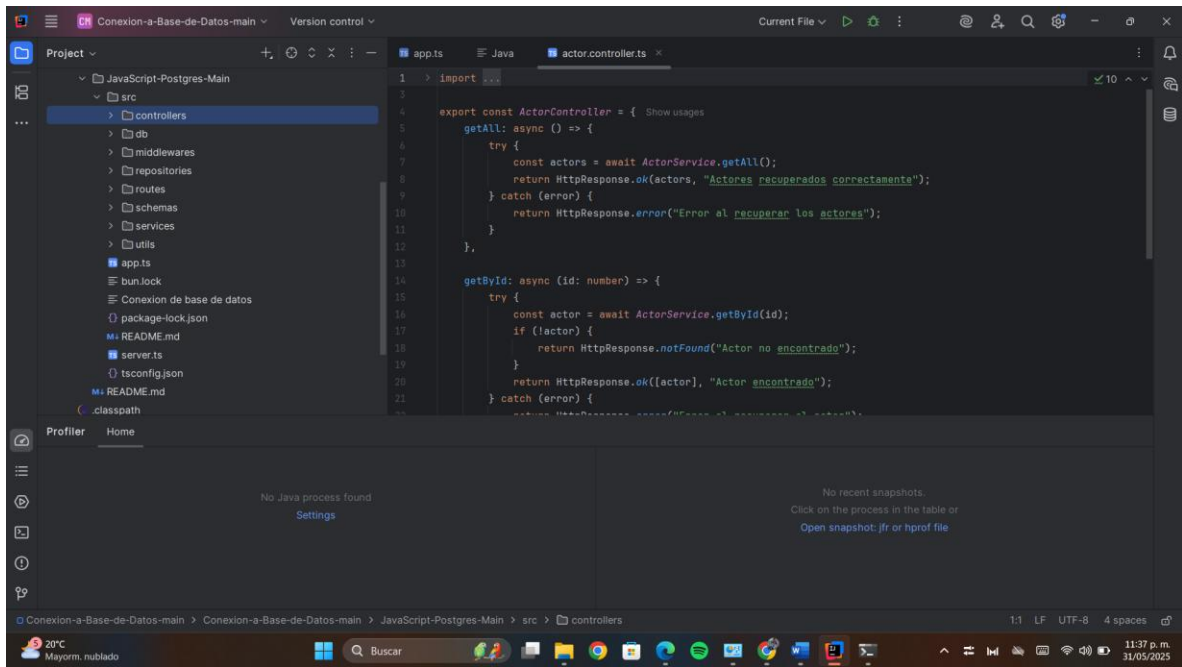
app.ts: Configuración de Express (middlewares globales, rutas base).

server.ts: Inicia el servidor (escucha en el puerto 3000).

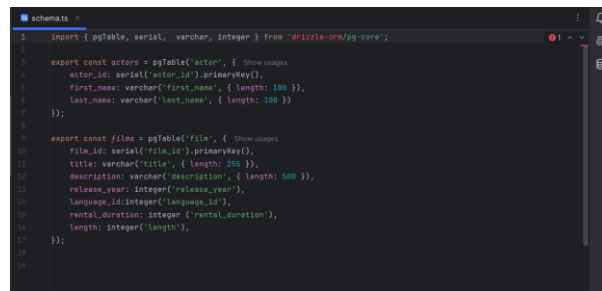
.env: Variables de entorno

package.json: Dependencias y scripts del proyecto (ej:

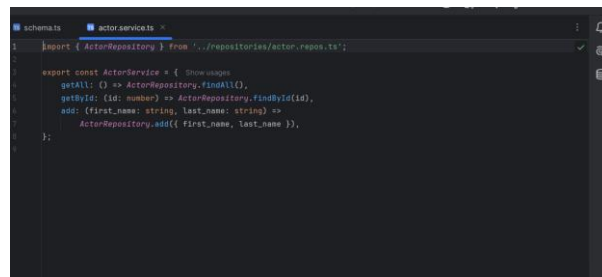
npm run dev para iniciar)



Schemas: En esta carpeta, se define las validaciones de cada atributo que asignamos anteriormente, por ejemplo: para los nombres seran .string, para los apellidos seran .string, todo creados por medio de objetos.



Services: Aquí deben de editarse como se debe de ver los resultados, en este caso como tipo JSON y como queremos que los visualice el usuario.



Controllers: En esta carpeta se crean los getters y los métodos de añadir para actores y para películas, además de que se añade al mismo BD cuando desea el usuario añadir películas o actores.

```
import express

export const ActorController = {
  getAll: async () => {
    try {
      const actors = await ActorService.getAll();
      return HttpResponse.ok(actors, "Actores recuperados correctamente");
    } catch (error) {
      return HttpResponse.error("Error al recuperar los actores");
    }
  },

  getById: async (id: number) => {
    try {
      const actor = await ActorService.getById(id);
      if (!actor) {
        return HttpResponse.notFound("Actor no encontrado");
      }
      return HttpResponse.ok([actor], "Actor encontrado");
    } catch (error) {
      return HttpResponse.error("Error al recuperar el actor");
    }
  },

  add: async (body: { first_name: string; last_name: string }) => {
    try {
      const newActor = await ActorService.add(body.first_name, body.last_name);
      return HttpResponse.created(newActor, "Actor creado");
    } catch (error) {
      return HttpResponse.error("Error al crear el actor");
    }
  },
}
```

Routers: En esta carpeta define las rutas y asocia el controlador. Además se importan el schema dependiendo de que TS se está escribiendo ya sea para el actor o para la película.

```
import express // esto es el nuevo middleware
import { ActorSchema } from 'schemas'

const actorRouter = new Router();

actorRouter.get('/actors', async () => {
  const { status, body } = await ActorController.getAll();
  return new HttpResponse(JSON.stringify(body), {
    status: status,
    headers: { 'Content-Type': 'application/json' }
  });
});

actorRouter.get('/actors/:id', async (c) => {
  const id = Number(c.req.param('id'));
  const { status, body } = await ActorController.getById(id);
  return new HttpResponse(JSON.stringify(body), {
    status: status,
    headers: { 'Content-Type': 'application/json' }
  });
});

actorRouter.post(
  '/actors',
  validateBody(ActorSchema), // validación personalizada
  async (c) => {
    const bodyValidated = c.get('validatedBody'); // ya está validado
    const { status, body } = await ActorController.add(bodyValidated);
    return new HttpResponse(JSON.stringify(body), {
      status: status,
      headers: { 'Content-Type': 'application/json' }
    });
  }
);
```

App: En esta carpeta define las rutas para su ejecutable y se pueda hacer en el navegador.

```
import express

const app = new Express();

app.use('*', errorHandler); // Aplica a todas las rutas
app.route('/', actorRouter);
app.route('/', filmRouter);
export default app;
```