Instituto Tecnológico del Valle de Oaxaca

TECNM TECNOLOGICO NACIONAL DE MÉXICO

MATERIA:TALLER DE BASE DE DATOS

APLICACIÓN CON CONEXIÓN A BASE DE DATOS

30/05/2025

ALUMNA: KAREN NOEMI MARCÍAL GARCÍA

CARRERA: INGENIERIA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES

4ªSEMESTRE

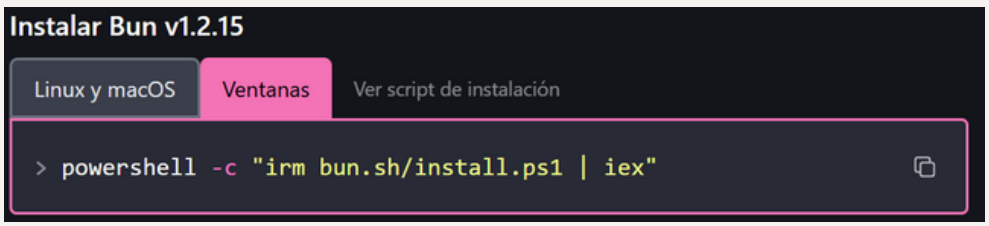
CARRERA: INGENIERIA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES

INSTALACION DEL BUN

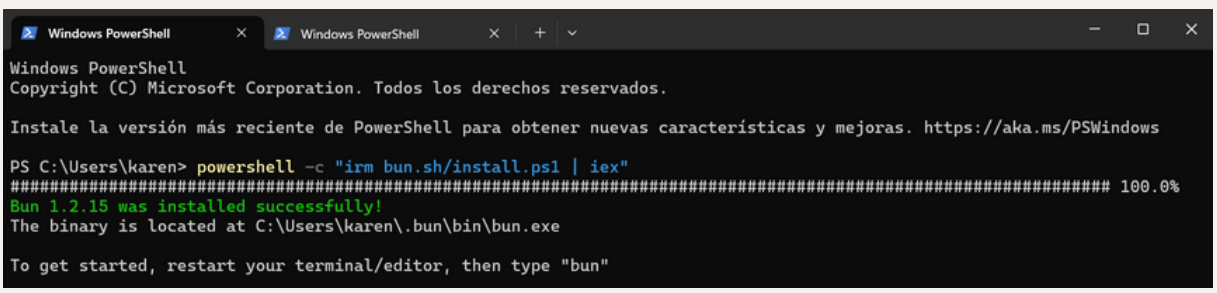
Desarrolla, prueba, ejecuta y agrupa proyectos de JavaScript y TypeScript: todo con Bun. Bun es un entorno de ejecución y kit de herramientas de JavaScript todo en uno, diseñado para la velocidad, que incluye un empaquetador, un ejecutor de pruebas y un gestor de paquetes compatible con Node.js. Bun busca una compatibilidad total con Node.js.

Entramos a la pagina oficial de “Bun”

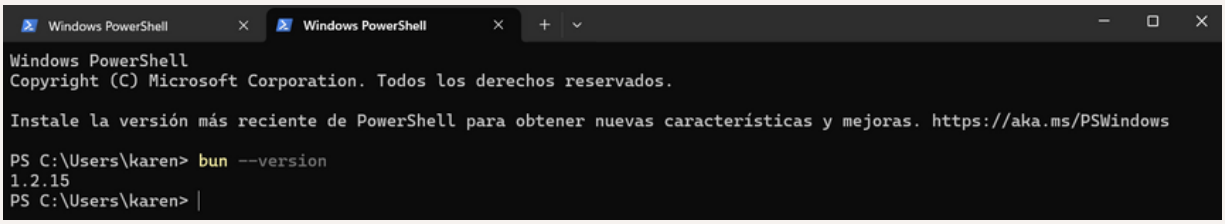
Bun se instala mediante un script automatizado que descarga e instala el entorno en el sistema del usuario. Para ello, se debe ejecutar el siguiente comando desde la terminal:



Este script descargará la última versión estable de Bun, la instalará en el sistema y agregará automáticamente las variables de entorno necesarias. En algunos casos, puede requerirse reiniciar la terminal para que los cambios surtan efecto.

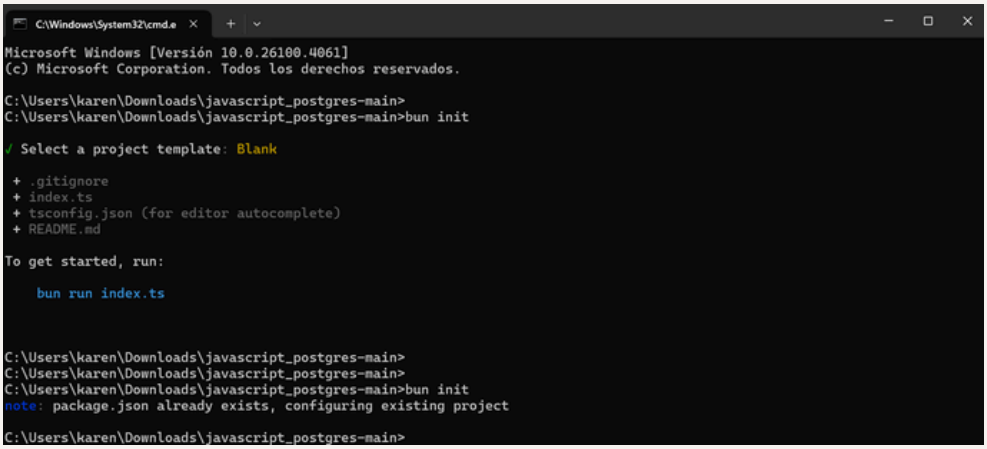


Para verificar que Bun se ha instalado correctamente, se utiliza el siguiente comando:

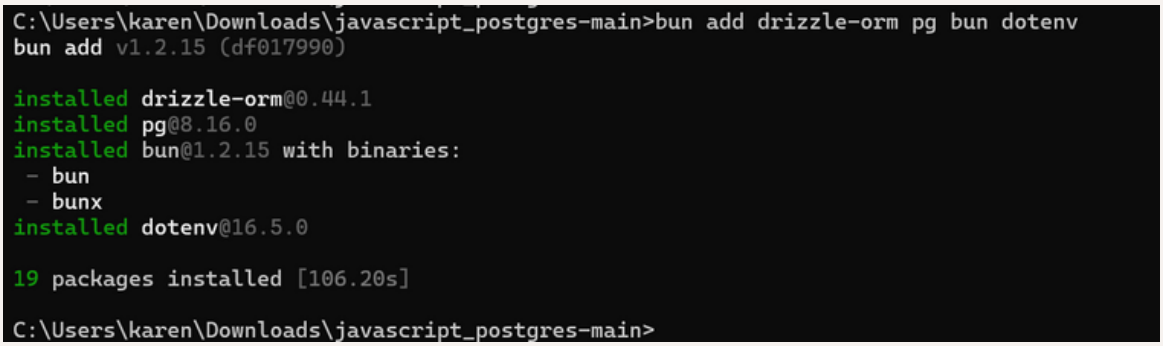


Nos dirigimos a la ruta de nuestro trabajo y al inicio escribimos “cmd” para que nos abra una venta de consola

En el cmd del proyecto escribimos “bun init” este es un comando el cual se utiliza para crear un nuevo proyecto en blanco



Ya creado el proyecto agregaremos otro comando =bun add drizzle-orm pg bun dotenv este descargara e instalara estas librerías desde el registro de paquetes y las añada a las dependencias de tu proyecto, para que puedas utilizarlas en tu código.



CONEXION DE JAVASCRIPT-POSTGRES

Para la creacion de la conexion utilizaremos IntelliJ IDEA que es un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos. Es desarrollado por JetBrains (anteriormente conocido como IntelliJ), y está disponible en dos ediciones: edición para la comunidad y edición comercial

Estructuracion de carpetas

- src: Carpeta que tiene el codigo fuente
- controllers:Contiene la lógica que maneja las peticiones HTTP. Aquí se reciben las solicitudes del cliente, se validan, y se llama a los servicios correspondientes.
- db:Se encarga de la configuración de la base de datos. Aquí puedes definir la conexión a PostgreSQL, MySQL, SQLite u otro motor.
- middlewares: Contiene funciones que se ejecutan antes o después de los controladores, como validaciones, autenticación, manejo de errores o logging.
- repositories: Aquí se encapsula el acceso a la base de datos, separando la lógica de negocio del acceso a datos.
- routes: Define las rutas del servidor, y asocia cada ruta con su controlador correspondiente.
- schemas: Aquí van los esquemas de validación (por ejemplo, usando Zod, Joi, Yup) o los modelos de la base de datos si estás usando ORM como Prisma o Sequelize.
- services:Contiene la lógica de negocio de la aplicación. Aquí se hacen procesos, cálculos, validaciones internas o coordinación entre múltiples repositorios.
- utils:Almacena funciones auxiliares reutilizables, como generación de tokens, funciones de formateo, encriptación, etc.

Archivos en la raiz

- **app.ts:** Es el cerebro de tu aplicación. Aquí defines cómo se comporta tu servidor web
- **server.ts:** Este archivo es el "interruptor de encendido". Su única tarea es poner en marcha el servidor y decirle en qué "puerta" (el puerto 3000)
- **.env:** Imagina esto como una caja fuerte para tus secretos y configuraciones. Aquí guardas información sensible o que cambia según el entorno.
- **package.json:** Enumera todos los paquetes externos (librerías) que tu aplicación necesita para funcionar, y también te permite definir "atajos" (scripts) para tareas comunes

Db/schema: Una tabla actor y lo que parece ser una tabla film o movie, definiendo las columnas, sus tipos de datos y si son claves primarias. Esto es fundamental para que tu aplicación interactúe con la base de datos de forma tipada y segura.

actor_schema.ts

```ts
import {pgTable, seriat, varchar, integer, serial} from "drizzle-orm/pg-core";


export const actors  = pgTable('actor', {  no usages
    actor_id: serial("actor_id").primaryKey(),
    first_name: varchar("first_name", {length: 100}),
    last_name: varchar("last_name", {length: 100})
});

export const films =pgTable('fila',{  no usages
    film_id: serial(  "film_id").primaryKey(),
    title: varchar(  "title", { length: 255 }),
    description: varchar(  "description", { length: 500 }),
    release_year: integer( "release_year"),
    language_id:integer(  "lamguage_id"),
    rental_duration: integer (  "rentel_duration"),
    length: integer( "Lemgth"),
})
```

repositories: Una tabla actor y una tabla film, definiendo las columnas, sus tipos de datos y si son claves primarias. Esto es fundamental para que la aplicación interactúe con la base de datos de forma tipada y segura.

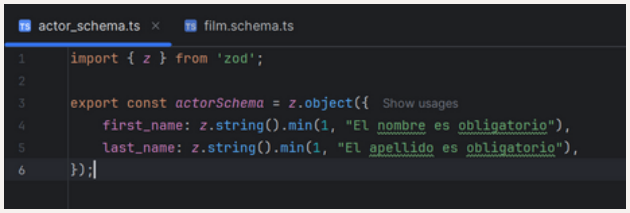
actor.repos.ts

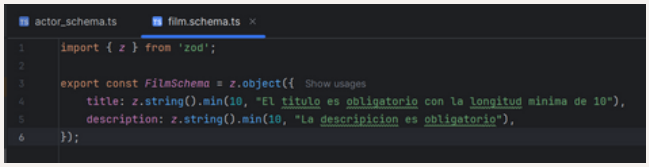
```ts
import { db } from '../db';
import { actors } from '../db/schema.ts';
import { eq } from "drizzle-orm/sql/expressions/conditions";

export const ActorRepository = {  Show usages
    findAll: async () => db.select().from(actors),
    findById: async (id: number) => {
        const [actor] = await db
            .select()
            .from(actors)
            .where(eq(actors.actor_id, id));
        return actor;
    },
    add: async (data: { first_name: string; last_name: string }) =>
        db.insert(actors).values(data).returning(),
};
```

film.repos.ts

```ts
import { db } from '../db';
import { films } from '../db/schema.ts';
import { eq } from "drizzle-orm/sql/expressions/conditions";

export const filmRepository = {  Show usages
    findAll: async () => db.select().from(films),
    findById: async (id: number) => {
        const [film] = await db
            .select()
            .from(films)
            .where(eq(films.film_id, id));
        return film;
    },
    add: async (data: { title: string; description: string, rental_duration: number }) =>
        db.insert(films).values(data).returning(),
};
```

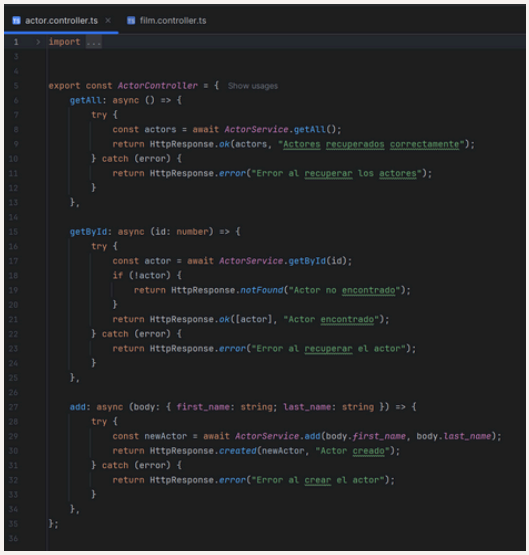
schema: Esta carpeta contiene esquemas de validación de datos. Define reglas para campos como nombre, apellido, título y descripción, asegurando que la información cumpla con los formatos y longitudes requeridos antes de ser utilizada.

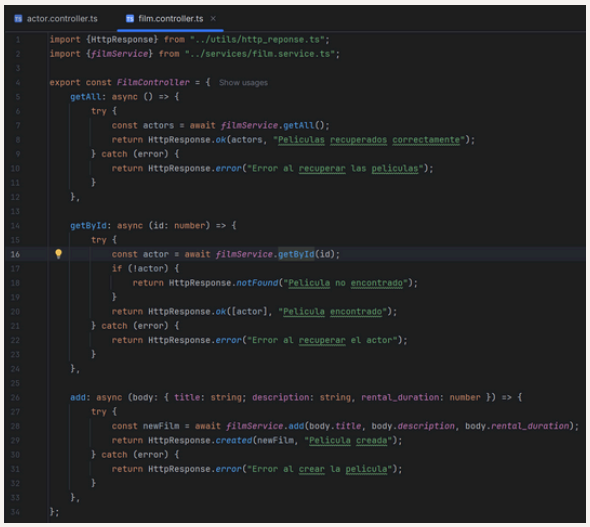




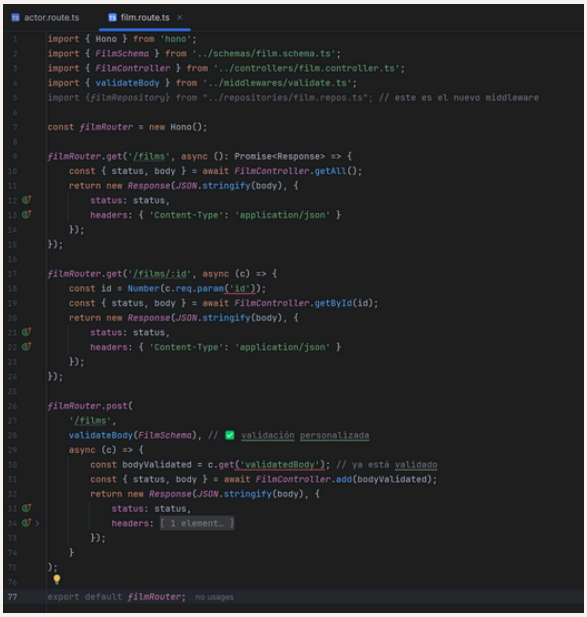
services: Los archivos muestran servicios que gestionan la lógica de negocio para actores y películas. Estos servicios importan y utilizan repositorios para realizar operaciones de acceso a datos como obtener, buscar por ID y añadir nuevos registros.

- **controllers:** Consulta exitosa Los archivos actor.controller.ts y film.controller.ts manejan las solicitudes HTTP. Utilizan servicios para obtener o agregar datos y envían respuestas HTTP al cliente, incluyendo mensajes personalizados en español.

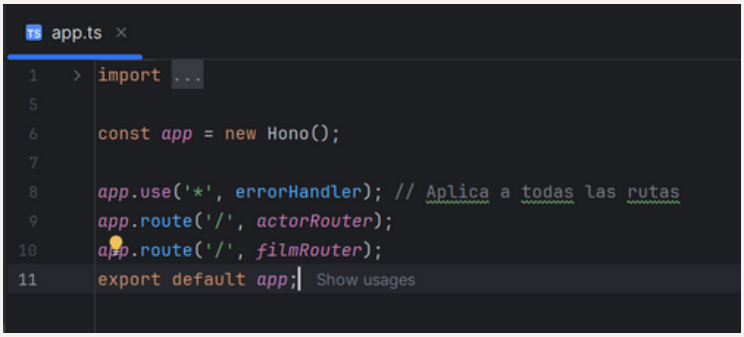




routers: Los archivos muestran cómo una aplicación Hono.js maneja rutas y valida datos. Definen las rutas API para actores y películas, utilizando un "middleware" para validar los datos de entrada con esquemas de Zod.



app: Inicializa la aplicación, aplica un manejador de errores global y registra las rutas para actores y películas.



Despliegue: Para ejecutar el proyecto, desde la carpeta raíz, usa bun server.ts. Luego, en tu navegador, accede a localhost/actors o localhost/films para ver los datos.

```
C:\Users\karen\Downloads\javascript_postgres-main>bun server.ts
🚀 Servidor escuchando en http://localhost:3000
```