



Instituto Tecnológico
del Valle de Oaxaca



MATERIA: TALLER DE BASE DE
DATOS

APLICACIÓN CON CONEXIÓN A BASE DE DATOS

30/05/2025

ALUMNA: KAREN NOEMI MARCIAL
GARCÍA

CARRERA: INGENIERIA EN TECNOLOGÍAS DE LA
INFORMACIÓN Y COMUNICACIONES

4ª SEMESTRE

CARRERA: INGENIERIA EN TECNOLOGÍAS DE LA
INFORMACIÓN Y COMUNICACIONES

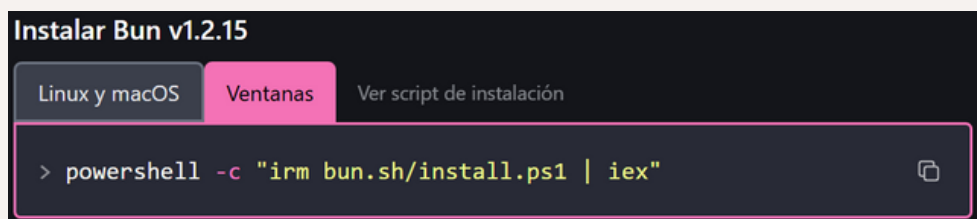
INSTALACION DEL BUN

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.

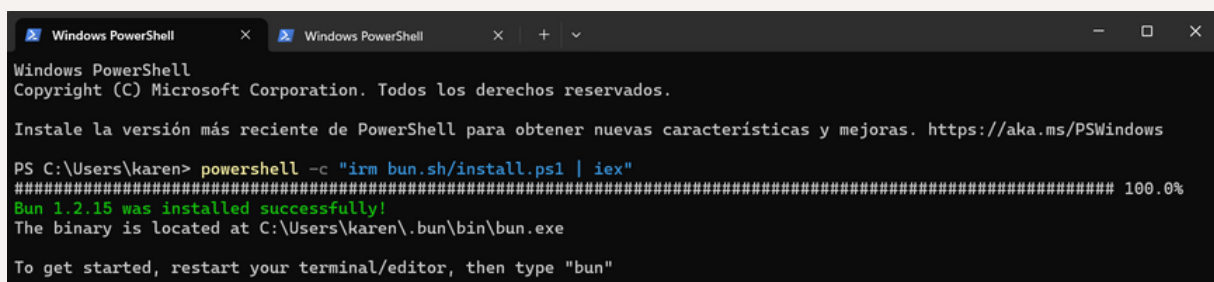
Entramos a la pagina oficial de “Bun”



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:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows

PS C:\Users\karen> bun --version
1.2.15
PS C:\Users\karen> |
```

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

```
cmd C:\Users\karen\Downloads\javascript_postgres-main
```

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

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.26100.4061]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\karen\Downloads\javascript_postgres-main>
C:\Users\karen\Downloads\javascript_postgres-main>bun init

✓ Select a project template: Blank

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

To get started, run:

  bun run index.ts

C:\Users\karen\Downloads\javascript_postgres-main>
C:\Users\karen\Downloads\javascript_postgres-main>
C:\Users\karen\Downloads\javascript_postgres-main>bun init
note: package.json already exists, configuring existing project
C:\Users\karen\Downloads\javascript_postgres-main>
```

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.

```
C:\Users\karen\Downloads\javascript_postgres-main>bun add drizzle-orm pg bun dotenv
bun add v1.2.15 (df017990)

installed drizzle-orm@0.44.1
installed pg@8.16.0
installed bun@1.2.15 with binaries:
- bun
- bunx
installed dotenv@16.5.0

19 packages installed [106.20s]
C:\Users\karen\Downloads\javascript_postgres-main>
```

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 raíz

- **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
1
2 import { pgTable, serial, varchar, integer, serial } from "drizzle-orm/pg-core";
3
4
5 export const actors = pgTable('actor', { no usages
6   actor_id: serial("actor_id").primaryKey(),
7   first_name: varchar("first_name", {length: 100}),
8   last_name: varchar("last_name", {length: 100})
9 });
10
11 export const films = pgTable('film', { no usages
12   film_id: serial("film_id").primaryKey(),
13   title: varchar("title", {length: 255}),
14   description: varchar("description", {length: 500}),
15   release_year: integer("release_year"),
16   language_id: integer("language_id"),
17   rental_duration: integer("rental_duration"),
18   length: integer("length"),
19 });
20
```

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
1 import { db } from '../db';
2 import { actors } from '../db/schema.ts';
3 import { eq } from "drizzle-orm/sql/expressions/conditions";
4
5 export const ActorRepository = { Show usages
6   findAll: async () => db.select().from(actors),
7   findById: async (id: number) => {
8     const [actor] = await db
9       .select()
10      .from(actors)
11      .where(eq(actors.actor_id, id));
12     return actor;
13   },
14   add: async (data: { first_name: string; last_name: string }) =>
15     db.insert(actors).values(data).returning(),
16 };
17
```

```
film_repos.ts
1 import { db } from '../db';
2 import { films } from '../db/schema.ts';
3 import { eq } from "drizzle-orm/sql/expressions/conditions";
4
5 export const FilmRepository = { Show usages
6   findAll: async () => db.select().from(films),
7   findById: async (id: number) => {
8     const [film] = await db
9       .select()
10      .from(films)
11      .where(eq(films.film_id, id));
12     return film;
13   },
14   add: async (data: { title: string; description: string; rental_duration: number }) =>
15     db.insert(films).values(data).returning(),
16 };
17
```

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.

```
1 import { z } from 'zod';
2
3 export const actorSchema = z.object({ Show usages
4   first_name: z.string().min(1, "El nombre es obligatorio"),
5   last_name: z.string().min(1, "El apellido es obligatorio"),
6 });
```

```
1 import { z } from 'zod';
2
3 export const FilmSchema = z.object({ Show usages
4   title: z.string().min(10, "El título es obligatorio con la longitud mínima de 10"),
5   description: z.string().min(10, "La descripción es obligatoria"),
6 });
```

```
1 import { ActorRepository } from '../repositories/actor.repos.ts';
2
3 export const ActorService = { Show usages
4   getAll: () => ActorRepository.findAll(),
5   getById: (id: number) => ActorRepository.findById(id),
6   add: (first_name: string, last_name: string) =>
7     ActorRepository.add({ first_name, last_name }),
8 };
```

```
1 import { FilmRepository } from '../repositories/film.repos.ts';
2 export const FilmService = { Show usages
3   getAll: () => FilmRepository.findAll(),
4   getById: (id: number) => FilmRepository.findById(id),
5   add: (title: string, description: string, rental_duration: number) =>
6     FilmRepository.add({ title, description, rental_duration }),
7 };
8
```

```
1 > import zod
2
3 export const ActorController = { Show usages
4   getAll: async () => {
5     try {
6       const actors = await ActorService.getAll();
7       return HttpResponse.ok(actors, "Actores recuperados correctamente");
8     } catch (error) {
9       return HttpResponse.error("Error al recuperar los actores");
10    },
11  },
12
13   getById: async (id: number) => {
14     try {
15       const actor = await ActorService.getById(id);
16       if (!actor) {
17         return HttpResponse.notFound("Actor no encontrado");
18       }
19       return HttpResponse.ok([actor], "Actor encontrado");
20     } catch (error) {
21       return HttpResponse.error("Error al recuperar el actor");
22     }
23   },
24
25   add: async (body: { first_name: string; last_name: string }) => {
26     try {
27       const newActor = await ActorService.add(body.first_name, body.last_name);
28       return HttpResponse.created(newActor, "Actor creado");
29     } catch (error) {
30       return HttpResponse.error("Error al crear el actor");
31     }
32   },
33 };
34
```

```
1 import (HttpResponse) from '../utils/http_response.ts';
2 import (FilmService) from '../services/film.service.ts';
3
4 export const FilmController = { Show usages
5   getAll: async () => {
6     try {
7       const actors = await FilmService.getAll();
8       return HttpResponse.ok(actors, "Películas recuperados correctamente");
9     } catch (error) {
10      return HttpResponse.error("Error al recuperar las películas");
11    },
12  },
13
14   getById: async (id: number) => {
15     try {
16       const actor = await FilmService.getById(id);
17       if (!actor) {
18         return HttpResponse.notFound("Película no encontrado");
19       }
20       return HttpResponse.ok([actor], "Película encontrado");
21     } catch (error) {
22       return HttpResponse.error("Error al recuperar el actor");
23     }
24   },
25
26   add: async (body: { title: string; description: string, rental_duration: number }) => {
27     try {
28       const newFilm = await FilmService.add(body.title, body.description, body.rental_duration);
29       return HttpResponse.created(newFilm, "Película creada");
30     } catch (error) {
31       return HttpResponse.error("Error al crear la película");
32     }
33   },
34 };
35
```

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.

```
actor.routes.ts x film.routes
1 > import ... // este es el nuevo middleware
2
3 const actorRouter = new Hono();
4
5 actorRouter.get('/actors', async () => Promise<Response> => {
6   const { status, body } = await ActorController.getAll();
7   return new Response(JSON.stringify(body), {
8     status: status,
9     headers: { 'Content-Type': 'application/json' }
10  });
11 });
12
13 actorRouter.get('/actors/:id', async (c) => {
14   const id = Number(c.req.param('id'));
15   const { status, body } = await ActorController.getById(id);
16   return new Response(JSON.stringify(body), {
17     status: status,
18     headers: { 'Content-Type': 'application/json' }
19  });
20 });
21
22 actorRouter.post(
23   '/actors',
24   validateBody(actorSchema), // validación personalizada
25   async (c) => {
26     const bodyValidated = c.get('validatedBody'); // ya está validado
27     const { status, body } = await ActorController.add(bodyValidated);
28     return new Response(JSON.stringify(body), {
29       status: status,
30       headers: { 'Content-Type': 'application/json' }
31     });
32   }
33 );
34
35 export default actorRouter;
```

```
actor.routes.ts x film.routes
1 import { Hono } from 'hono';
2 import { FilmSchema } from '../schemas/film.schema.ts';
3 import { FilmController } from '../controllers/film.controller.ts';
4 import { validateBody } from '../middlewares/validate.ts';
5 import { FilmRepository } from '../repositories/film.repo.ts'; // este es el nuevo middleware
6
7 const filmRouter = new Hono();
8
9 filmRouter.get('/films', async () => Promise<Response> => {
10   const { status, body } = await FilmController.getAll();
11   return new Response(JSON.stringify(body), {
12     status: status,
13     headers: { 'Content-Type': 'application/json' }
14  });
15 });
16
17 filmRouter.get('/films/:id', async (c) => {
18   const id = Number(c.req.param('id'));
19   const { status, body } = await FilmController.getById(id);
20   return new Response(JSON.stringify(body), {
21     status: status,
22     headers: { 'Content-Type': 'application/json' }
23  });
24 });
25
26 filmRouter.post(
27   '/films',
28   validateBody(FilmSchema), // validación personalizada
29   async (c) => {
30     const bodyValidated = c.get('validatedBody'); // ya está validado
31     const { status, body } = await FilmController.add(bodyValidated);
32     return new Response(JSON.stringify(body), {
33       status: status,
34       headers: { 'Content-Type': 'application/json' }
35     });
36   }
37 );
38
39 export default filmRouter;
```

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

```
ts app.ts x
1 > import ...
2
3 const app = new Hono();
4
5 app.use('*', errorHandler); // Aplica a todas las rutas
6 app.route('/', actorRouter);
7 app.route('/', filmRouter);
8
9 export default app;
```

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
```

```
localhost:3000/films

"success": true,
"message": "Películas recuperados correctamente",
"data": [
  {
    "film_id": 133,
    "title": "Chamber Italian",
    "description": "A Fateful Reflection of a Moose And a Husband who must Overcome a Monkey in Nigeria",
    "release_year": 2006,
    "language_id": 1,
    "rental_duration": 7,
    "length": 117
  },
  {
    "film_id": 384,
    "title": "Grosse Wonderful",
    "description": "A Epic Drama of a Cat And a Explorer who must Redeem a Moose in Australia",
    "release_year": 2006,
    "language_id": 1,
    "rental_duration": 5,
    "length": 49
  }
]
```

```
{
  "success": true,
  "message": "Actores recuperados correctamente",
  "data": [
    {
      "actor_id": 1,
      "first_name": "Penelope",
      "last_name": "Guinness"
    },
  ],
}
```