



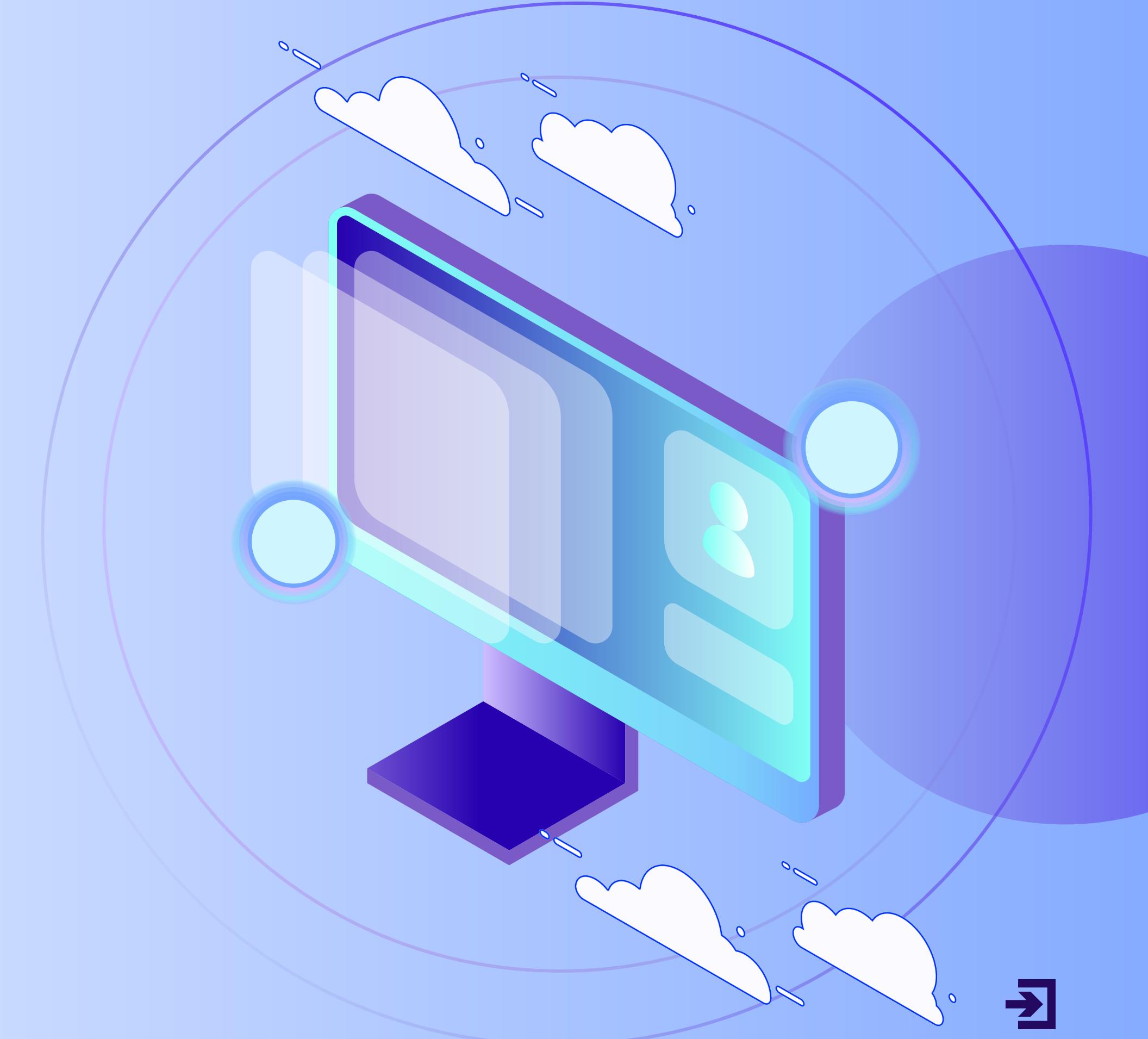
Ingeniería de Software I

HOST RUNNING

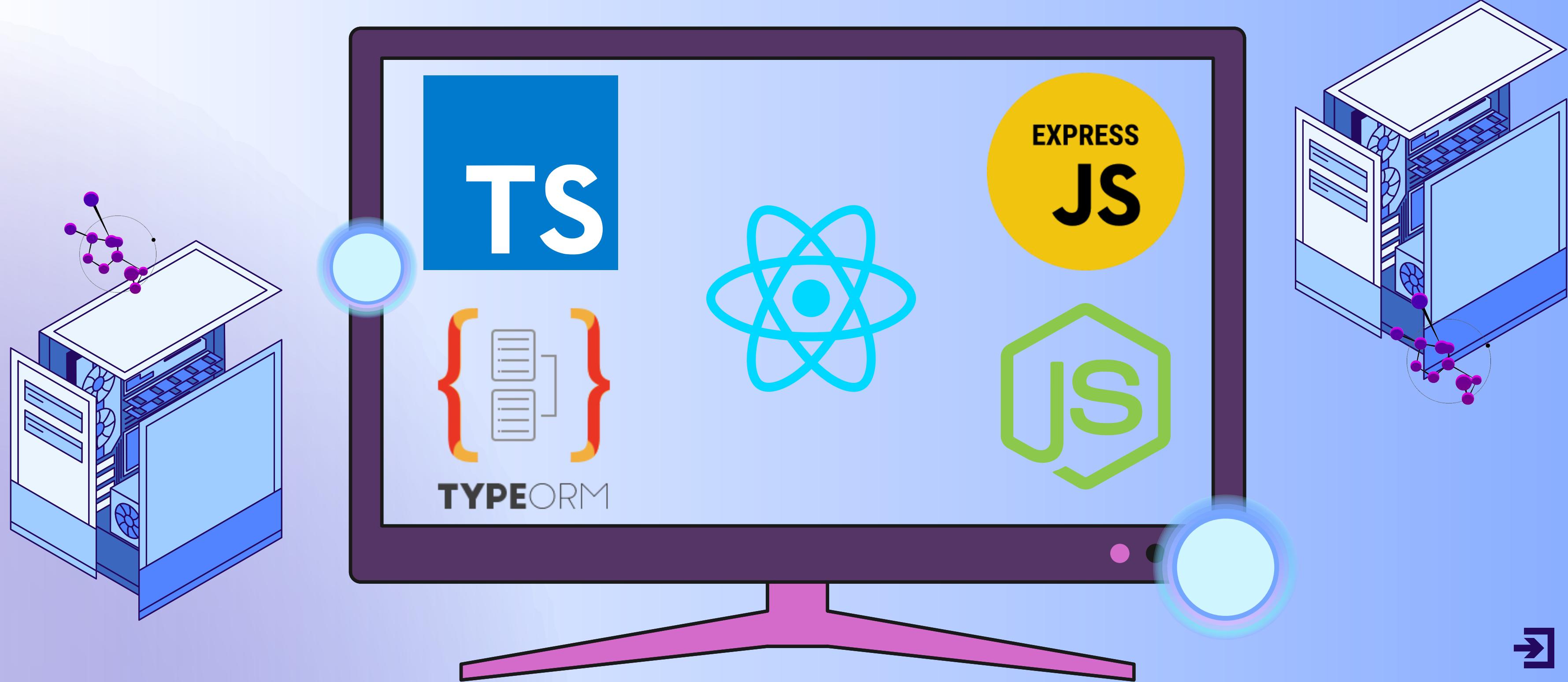
Tutorial:
Construcción desde 0

Diego Cabrera
Jauri Cortes

Ángel Gómez
Pablo Luna



LENGUAJES, FRAMEWORKS Y ORM



PROPÓSITO DE LIBRERÍAS



- **@types/node**: definiciones de tipo para Node.js para utilizar funciones y módulos de Node.js con tipado estático en TypeScript.
- **typescript**: Escritura de código en TypeScript. Es un superconjunto de JavaScript con tipado estático. Esto ayuda a detectar errores en tiempo de compilación.
- **ts-node**: Compilar archivos .ts sin necesidad de compilarlos primero en JavaScript.
- **nodemon**: Reinicio automático de la aplicación cuando detecta cambios en los archivos.
- **pg**: Cliente PostgreSQL para Node.js. Conexión y operaciones en bases de datos PostgreSQL.
- **reflect-metadata**: Permite que la herramienta TypeORM use metadatos en TypeScript para definir entidades y sus relaciones.
- **typeorm**: ORM: “Object-Relational Mapper” que permite interactuar con la base de datos con lenguaje de TypeScript en lugar realizar consultas en el lenguaje SQL.
- **@types/express**: Definiciones de tipo para el Framework Express.js.

INSTALACIÓN DE LIBRERÍAS



Paso	Comando	Propósito
1. Inicializar el Proyecto	<code>npm init -y</code>	Crea el archivo <code>package.json</code> para definir el proyecto Node.js.
2. Instalar Express	<code>npm install express</code>	Instala el <i>framework</i> principal del servidor web (Backend).
3. Instalar TypeScript	<code>npm install -g typescript ts-node</code>	Instala el compilador de TypeScript y la herramienta para ejecutar archivos <code>.ts</code> directamente.
4. Instalar TypeORM/Postgres	<code>npm install typeorm pg reflect-metadata</code>	Instala el ORM principal (TypeORM), el controlador de PostgreSQL (<code>pg</code>) y el <i>polyfill</i> necesario para los decoradores de TypeScript.
5. Instalar Tipos de TypeScript	<code>npm install --save-dev @types/node @types/express</code>	Instala los archivos de definición de tipos para Node.js y Express.
6. Archivo de Ejecución	<i>Configurar package.json :</i> <code>"start": "ts-node src/index.ts"</code>	Define el <i>script</i> para ejecutar el servidor.



ESTRUCTURA BACKEND

```
Backend/
└── src/
    ├── config/
    │   └── dataSource.ts      # Única configuración DB (TypeORM)
    ├── entity/
    │   └── User.ts           # Entidades TypeORM
    ├── controller/
    │   └── UserController.ts
    ├── scripts/
    │   └── create-db.js       # Solo para setup inicial
    └── index.ts              # Entry point
```

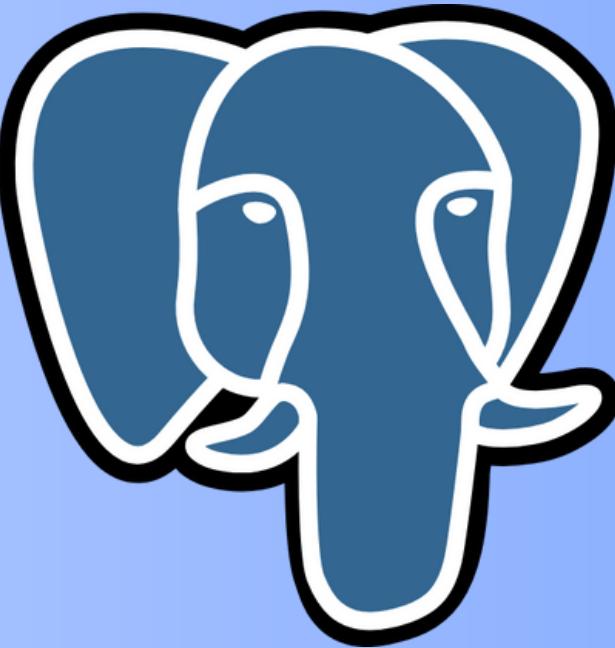
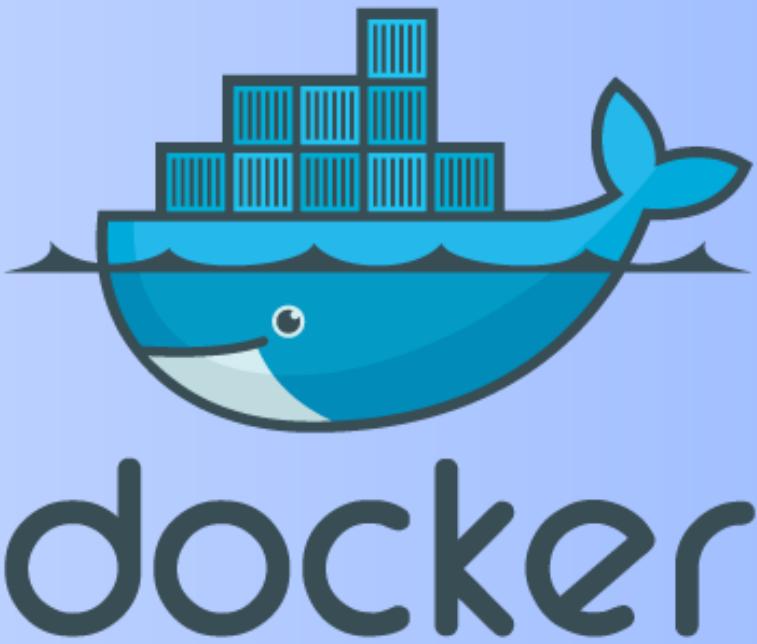


UTILIZANDO DOCKER

Ghost Running es, en esencia, una aplicación móvil. Debido a las restricciones de nuestro contexto, todo debe ejecutarse de manera local.

Docker introduce una capa adicional de consumo de RAM y procesamiento, conocida como *overhead* en inglés, que en nuestro caso debemos minimizar para asegurar un rendimiento óptimo en el dispositivo.

Por esta razón, para ejecutar nuestra base de datos PostgreSQL decidimos instalar el servidor directamente en el dispositivo, en lugar de usar un contenedor. De esta forma evitamos añadir carga innecesaria al sistema y aprovechamos mejor los recursos disponibles.



DATABASE DESIGN

```
DB_HOST=localhost  
DB_PORT=5432  
DB_USER=postgres  
DB_PASSWORD=tu_contraseña  
DB_NAME=ghost_running_db
```

1. Creación de archivo de variables de entorno (archivo ".env") para configurar las credenciales de la conexión con la base de datos

```
1 import { Client } from 'pg'; // importa cliente PostgreSQL para conexión directa  
2 import dotenv from 'dotenv'; // Habilita variables de entorno  
3  
4 dotenv.config(); // Carga variables de entorno  
5  
6 const createDatabase = async () => {  
7   const client = new Client({ // Configura conexión inicial a PostgreSQL  
8     user: process.env.DB_USER,  
9     host: process.env.DB_HOST,  
10    password: process.env.DB_PASSWORD,  
11    port: Number(process.env.DB_PORT)  
12  });  
13  
14  try { // Logica de verificación  
15    await client.connect(); // Establece la conexión  
16    const dbName = process.env.DB_NAME;  
17  
18    // Verificar si la base de datos existe  
19    const checkResult = await client.query(  
20      `SELECT 1 FROM pg_database WHERE datname = $1`,  
21      [dbName]  
22    );  
23  
24    if (checkResult.rows.length === 0) {  
25      // Crear base de datos si no existe  
26      await client.query(`CREATE DATABASE ${dbName}`);  
27      console.log(`Base de datos ${dbName} creada exitosamente`);  
28    } else {  
29      console.log(`Base de datos ${dbName} ya existe`);  
30    }  
31  } catch (error) {  
32    console.error('Error al crear la base de datos:', error);  
33  } finally {  
34    await client.end();  
35  }  
36};  
37  
38 createDatabase();
```

2. Escribir script ".ts" para la creación de la base de datos

ts create-db.ts

DATABASE DESIGN

4.

- Inicializa la conexión a la base de datos
- Configura middleware y rutas
- Inicia el servidor Express
- Maneja errores de conexión

```
1 // src/index.ts
2 import "reflect-metadata"; // Debe ser la primera importación - necesaria para TypeORM
3 import express from "express"; // Framework web
4 import * as dotenv from "dotenv"; // Manejo de variables de entorno
5 import * as path from "path"; // Manejo de rutas de archivos
6 import { AppDataSource } from "./config(dataSource"; // Configuración de TypeORM
7 import { getFirstUser } from "./controller/UserController"; // Controlador de usuario
8
9
10 // Cargar variables de entorno
11 dotenv.config({ path: path.resolve(__dirname, '../.env') });
12 // ¡¡¡LÍNEA DE DEBUG!!!
13 console.log(`DEBUG: DB_PASSWORD leída: [${process.env.DB_PASSWORD}]`);
14
15 const app = express();
16 const PORT = process.env.PORT || 3000;
17
18 // Inicializar la Conexión a la Base de Datos
19 AppDataSource.initialize()
20 .then(() => {
21     console.log("✓ Conexión a la Base de Datos establecida con éxito.");
22
23     app.use(express.json()); // Middleware para parsear JSON
24
25     // Ruta de prueba
26     app.get("/api/hello-user", getFirstUser);
27
28     // Iniciar el Servidor
29     app.listen(PORT, () => {
30         console.log(`🚀 Servidor Express corriendo en el puerto ${PORT}`);
31         console.log(`Endpoint de prueba: http://localhost:${PORT}/api/hello-user`);
32     });
33 }
34 .catch((error) => console.error("✗ Error al conectar la base de datos:", error));
```

DATABASE DESIGN

```
1 import { DataSource } from "typeorm";
2 import { User } from "../entity/User";
3 import dotenv from 'dotenv';
4
5 dotenv.config();
6
7 export const AppDataSource = new DataSource({
8   type: "postgres",
9   host: process.env.DB_HOST,
10  port: Number(process.env.DB_PORT),
11  username: process.env.DB_USER,
12  password: process.env.DB_PASSWORD,
13  database: process.env.DB_NAME,
14  entities: [User], // Array de entidades a gestionar
15  synchronize: true, //Sincronización automática del esquema
16  logging: true // Registro de consultas
17});
```

```
// Ejemplo de uso en un controlador
import { AppDataSource } from "../config(dataSource";

const userRepository = AppDataSource.getRepository(User);

// Consultas a la base de datos
const users = await userRepository.find();
```

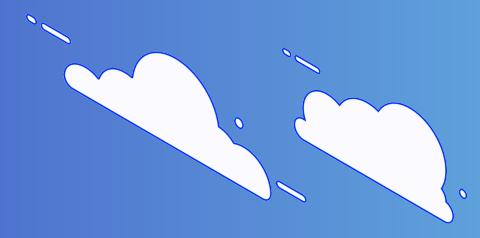
3. Configuración de TypeORM para:
- Crear tablas automáticamente.
 - Mantener sincronización entre código y base de datos.
 - Gestionar relaciones entre entidades.

DATABASE

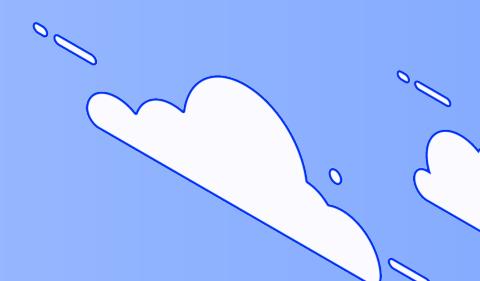
DESIGN

4.

- Inicializa la conexión a la base de datos
- Configura middleware y rutas
- Inicia el servidor Express
- Maneja errores de conexión



```
1 // src/entity/User.ts
2 import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";
3
4 @Entity()
5 export class User {
6     @PrimaryGeneratedColumn() // Define clave primaria auto incremental
7     id!: number;           // !: indica que el atributo no puede ser null
8     // Definición de atributos
9     @Column({ unique: true }) // Evita duplicados en la base de datos
10    username!: string;
11
12    @Column({ unique: true })
13    email!: string;
14
15    @Column({ select: false }) // Oculta atributo. Medida de seguridad
16    password_hash!: string;    // dada la naturaleza del atributo
17 }
```



DATABASE DESIGN

En este archivo se implementa la función donde se aprecia el flujo completo de la API “Express” con la base de datos.

```
1 // src/controller/UserController.ts
2 import { Request, Response } from 'express'; // Tipos de objetos de Express para
3 // manejar peticiones HTTP
4 import { AppDataSource } from '../config/dataSource'; // Importamos la conexión
5 import { User } from '../entity/User'; // Importamos la entidad
6
7 /**
8 * Endpoint del tutorial "Hola Mundo".
9 * Garantiza que existe un registro de usuario y lo devuelve,
10 * demostrando la comunicación API -> Express -> TypeORM -> PostgreSQL.
11 */
12 export const getFirstUser = async (req: Request, res: Response) => {
13     try {
14         // 1. Acceder al repositorio de la entidad User
15         const userRepository = AppDataSource.getRepository(User);
16
17         // 2. Intentar buscar el primer registro con ID = 1
18         let firstUser = await userRepository.findOne({ where: { id: 1 } });
19
20         // 3. Si no existe, lo insertamos (setup de demostración)
21         if (!firstUser) {
22             const newUser = userRepository.create({
23                 username: "Diego Cabrera",
24                 email: "diegoghost@running.com",
25                 password_hash: "ruunerporlaUN"
26             });
27             await userRepository.save(newUser);
28             firstUser = newUser;
29         }
30
31         // 4. Devolver el registro de la BD
32         res.status(200).json({
33             message: "Hello World: User record fetched successfully",
34             data: {
35                 id: firstUser.id,
36                 username: firstUser.username,
37                 email: firstUser.email
38             }
39         });
40
41     } catch (error) {
42         console.error("Error fetching user:", error);
43         res.status(500).json({ message: "Internal server error" });
44     }
45 };
46
```

VERIFICACIÓN EJECUCIÓN

BACKEND

```
# Instalar dependencias si no lo hiciste Todo en el directorio /Backend
npm install
# Iniciar servidor
npm start
```

```
✓ Conexión a la Base de Datos establecida con éxito.
🚀 Servidor Express corriendo en el puerto 3000
Endpoint de prueba: http://localhost:3000/api/hello-user
```

ESTRUCTURA FRONTEND

```
ingesoft-i
└── Proyecto
    ├── Backend
    └── Frontend
        └── ghost-running-app
            ├── node_modules
            ├── src
            │   └── screens <-- Aquí pondremos el componente
            ├── App.tsx      <-- Archivo principal
            └── package.json
            ...

```

FRONTEND



- *Inicializa el proyecto React Native con Framework Expo. Usamos un nombre simple para la app.*

Ejecutar en terminal y directorio /Proyecto:

```
npx create-expo-app ghost-running-app --template expo-template-blank-typescript
```

- *Entrar a la carpeta del proyecto recién creado*
cd ghost-running-app

- *Instalar dependencias y Axios. Instala la librería para hacer peticiones HTTP*
npm install axios

- *Crear estructuras de carpetas. Desde la terminal en la carpeta ghost-running-app*
mkdir src
mkdir src/screens

FRONTEND

- Crea el archivo `src/screens/TestScreen.tsx` y pega el siguiente código:

```
1 // Proyecto/Frontend/ghost-running-app/src/screens/TestScreen.tsx
2
3 import React, { useState, useEffect } from 'react';
4 import { View, Text, StyleSheet, ActivityIndicator } from 'react-native';
5 import axios from 'axios';
6
7 // --- AJUSTA ESTA IP SEGÚN TU ENTORNO (10.0.2.2 es para Emulador Android) ---
8 const API_BASE_URL = 'http://192.168.5.102:3000';
9
10 interface UserData {
11   id: number;
12   username: string;
13   email: string;
14 }
15
16 const TestScreen = () => {
17   const [userData, setUserData] = useState<UserData | null>(null);
18   const [loading, setLoading] = useState(true);
19   const [error, setError] = useState('');
20
21   useEffect(() => {
22     fetchData();
23   }, []);
24
25   const fetchData = async () => {
26     try {
27       // Llama al endpoint de prueba que creamos en el Backend
28       const response = await axios.get(`${API_BASE_URL}/api/hello-user`);
29
30       if (response.data && response.data.data) {
31         setUserData(response.data.data);
32       } else {
33         setError('Respuesta inesperada del servidor');
34       }
35
36     } catch (err: any) {
37       console.error('Error fetching data:', err);
38       // Esto ayuda a diagnosticar si el error es de red (no encuentra la IP)
39       setError(`Fallo la conexión o el backend: ${err.message}`);
40     } finally {
41       setLoading(false);
42     }
43   };
44 }
```

```
44   if (loading) {
45     return (
46       <View style={styles.container}>
47         <ActivityIndicator size="large" color="#007bff" />
48         <Text style={{ marginTop: 10 }}>Cargando datos del Backend...</Text>
49       </View>
50     );
51   }
52
53   return (
54     <View style={styles.container}>
55       <Text style={styles.header}>✓ RNF 1: Conexión Frontend/Backend Exitosa</Text>
56       {error ? (
57         <Text style={styles.errorText}>Error: {error}</Text>
58       ) : userData ? (
59         <View style={styles.card}>
60           <Text style={styles.label}>Usuario de prueba (Obtenido de PostgreSQL):</Text>
61           <Text style={styles.value}>ID: {userData.id}</Text>
62           <Text style={styles.value}>Username: {userData.username}</Text>
63           <Text style={styles.value}>Email: {userData.email}</Text>
64         </View>
65       ) : (
66         <Text style={styles.errorText}>No se pudo cargar la información del usuario.</Text>
67       )
68     </View>
69   );
70 }
71
72
73 const styles = StyleSheet.create({
74   container: { flex: 1, justifyContent: 'center', alignItems: 'center', padding: 20, backgroundColor: '#f5f5f5' },
75   header: { fontSize: 18, fontWeight: 'bold', marginBottom: 30, textAlign: 'center', color: '#007bff' },
76   card: { backgroundColor: '#ffffff', padding: 20, borderRadius: 10, shadowColor: '#0000', shadowOpacity: 0.1, shadowRadius: 4, elevation: 5, width: '90%' },
77   label: { fontSize: 14, fontWeight: '600', marginBottom: 5, color: '#333' },
78   value: { fontSize: 16, marginBottom: 5, color: '#555' },
79   errorText: { fontSize: 16, color: 'red', textAlign: 'center', marginTop: 20 },
80 });
81
82 export default TestScreen;
```

FRONTEND

- Modificar archivo App.tsx

```
1 // Proyecto/Frontend/ghost-running-app/App.tsx
2
3 import React from 'react';
4 // Importa el componente de prueba
5 import TestScreen from './src/screens/TestScreen';
6
7 export default function App() {
8   return <TestScreen />;
9 }
```

- En este directorio ejecutar

d:\ingesoft-i\proyecto\Frontend>



npm start



EMULADORES



VERIFICACIÓN EJECUCIÓN

- En este directorio ejecutar



```
\ingesoft-i\Proyecto\Frontend\ghost-running-app> npm start
```

FRONTEND

```
Starting Metro Bundler

> Metro waiting on exp://192.168.5.102:8081
> Scan the QR code above with Expo Go (Android) or the Camera app (iOS)

> Using Expo Go
> Press s | switch to development build

> Press a | open Android
> Press w | open web

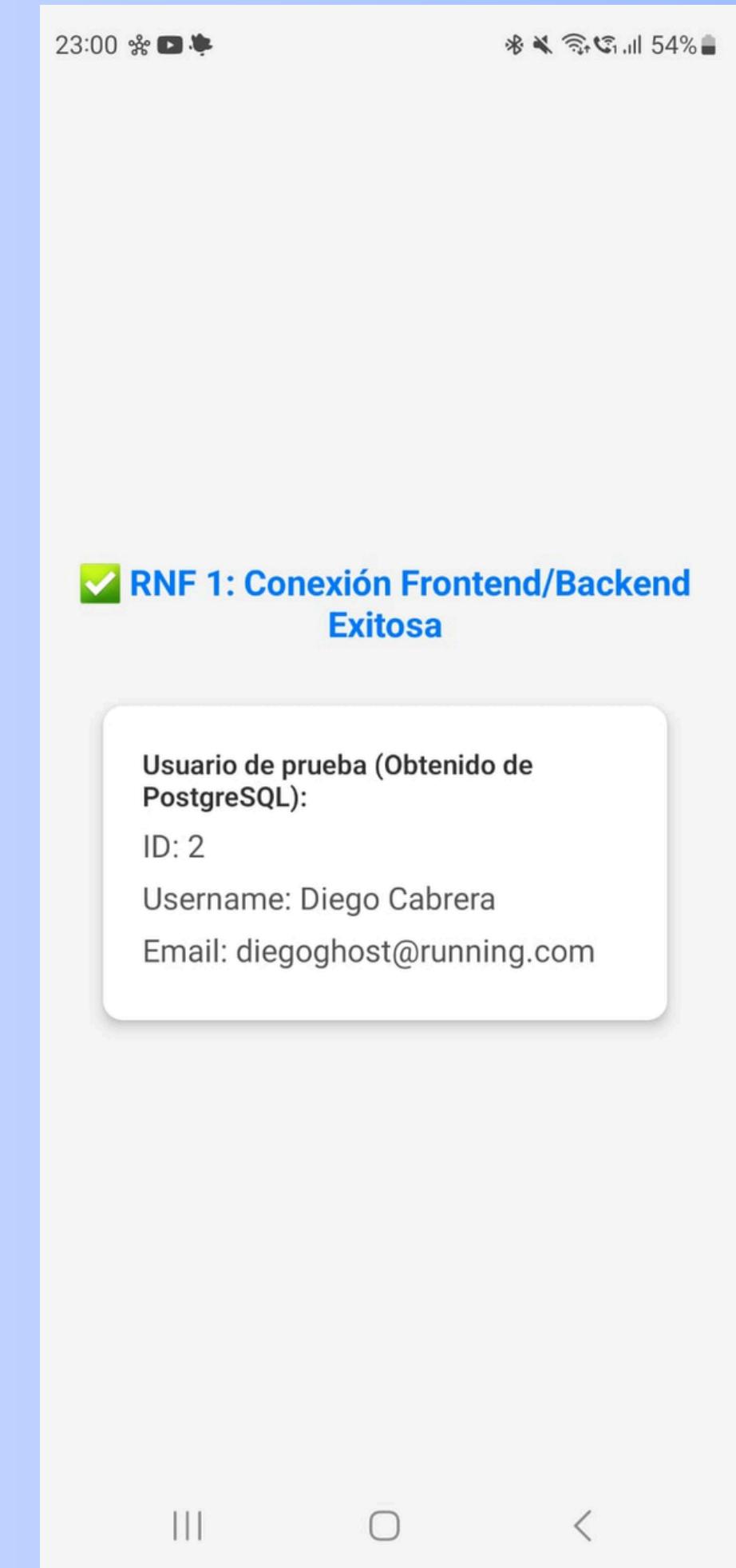
> Press j | open debugger
> Press r | reload app
> Press m | toggle menu
> shift+m | more tools
> Press o | open project code in your editor

> Press ? | show all commands

Logs for your project will appear below. Press Ctrl+C to exit.
```

FRONTEND

- *RESPUESTA*



THANK YOU!