Pregunta:

¿Podrías proporcionarme cada uno de los pasos e instrucciones para instalar crear el entorno inicial de trabajo y los ficheros básicos de configuración de los que partir tanto en front-end como en el back-end?

Respuesta:

Claro que sí. A continuación te presento una guía detallada paso a paso para crear el entorno de desarrollo inicial tanto para el *backend* con NestJS como para el *frontend* con Next.js, incluyendo los archivos de configuración básicos.

### **Prerrequisitos (Instalación única)**

Antes de empezar, asegúrate de tener instalado **Node.js** y **Docker**. Node.js es el entorno de ejecución para JavaScript/TypeScript, y Docker nos permitirá gestionar la base de datos y la aplicación de forma aislada y consistente.

1. **Instalar Node.js:** Descarga e instala la versión LTS (Long-Term Support) desde la [página oficial de Node.js](https://nodejs.org/). Esto también instalará npm, el gestor de paquetes de Node.
2. **Instalar Docker:** Descarga e instala **Docker Desktop** desde su [página oficial](https://www.docker.com/products/docker-desktop/). Esto te proporcionará Docker y Docker Compose.
3. **Instalar NestJS CLI:** Abre tu terminal o línea de comandos y ejecuta el siguiente comando para instalar la interfaz de línea de comandos de NestJS de forma global.  
   Bash  
   npm install -g @nestjs/cli

### **Parte 1: Creación del Entorno de Backend (NestJS)**

Vamos a crear el proyecto para tu API, configuraremos la conexión a la base de datos MySQL existente y añadiremos la documentación automática de la API con Swagger.

#### **Paso 1.1: Crear el Proyecto NestJS**

1. En tu terminal, navega hasta el directorio donde quieres crear tus proyectos.
2. Ejecuta el siguiente comando para crear un nuevo proyecto NestJS llamado mi-api-gestion.  
   Bash  
   nest new mi-api-gestion  
     
   Cuando te pregunte qué gestor de paquetes prefieres, puedes elegir **npm**. Esto creará una carpeta mi-api-gestion con toda la estructura inicial.
3. Navega dentro del nuevo directorio:  
   Bash  
   cd mi-api-gestion

#### **Paso 1.2: Instalar Dependencias para MySQL y Configuración**

Ahora instalaremos los paquetes necesarios para conectar NestJS con tu base de datos MySQL usando TypeORM y para gestionar las variables de entorno (como las credenciales de la base de datos).

Bash

npm install @nestjs/typeorm typeorm mysql2 @nestjs/config

* @nestjs/typeorm: Integración oficial de TypeORM para NestJS.
* typeorm: El ORM en sí.
* mysql2: El driver para conectar con bases de datos MySQL.
* @nestjs/config: Para gestionar variables de entorno de forma segura.

#### **Paso 1.3: Configurar la Conexión a la Base de Datos**

1. **Crear el archivo de variables de entorno:** En la raíz del proyecto mi-api-gestion, crea un archivo llamado .env y añade los datos de tu conexión a MySQL.  
   **.env**  
   # Database Configuration  
   DB\_HOST=localhost  
   DB\_PORT=3306  
   DB\_USERNAME=root  
   DB\_PASSWORD=tu\_contraseña\_de\_mysql  
   DB\_DATABASE=nombre\_de\_tu\_bd\_existente  
   * **Importante:** Si vas a usar Docker, es posible que el DB\_HOST deba cambiarse más adelante al nombre del servicio de Docker (por ejemplo, mysql\_db).
2. Configurar el Módulo Principal (app.module.ts):  
   Abre el archivo src/app.module.ts y modifícalo para importar ConfigModule y TypeOrmModule y establecer la conexión.  
   **src/app.module.ts**  
   TypeScript  
   import { Module } from '@nestjs/common';  
   import { ConfigModule, ConfigService } from '@nestjs/config';  
   import { TypeOrmModule } from '@nestjs/typeorm';  
   import { AppController } from './app.controller';  
   import { AppService } from './app.service';  
     
   @Module({  
    imports: [  
    // 1. Cargar variables de entorno del archivo .env  
    ConfigModule.forRoot({  
    isGlobal: true, // Hace que ConfigModule esté disponible en toda la aplicación  
    }),  
     
    // 2. Configurar la conexión a la base de datos usando las variables de entorno  
    TypeOrmModule.forRootAsync({  
    imports: [ConfigModule],  
    inject: [ConfigService],  
    useFactory: (configService: ConfigService) => ({  
    type: 'mysql',  
    host: configService.get<string>('DB\_HOST'),  
    port: parseInt(configService.get<string>('DB\_PORT'), 10),  
    username: configService.get<string>('DB\_USERNAME'),  
    password: configService.get<string>('DB\_PASSWORD'),  
    database: configService.get<string>('DB\_DATABASE'),  
    entities: [\_\_dirname + '/\*\*/\*.entity{.ts,.js}'], // Busca automáticamente las entidades  
    synchronize: false, // ¡MUY IMPORTANTE! Poner en 'false' en producción para no sobreescribir tu BD existente.  
    }),  
    }),  
    ],  
    controllers: [AppController],  
    providers: [AppService],  
   })  
   export class AppModule {}

#### **Paso 1.4: Configurar la Documentación con Swagger (OpenAPI)**

Esto creará automáticamente una página web interactiva con la documentación de tu API.

1. **Instalar el paquete de Swagger:**  
   Bash  
   npm install @nestjs/swagger
2. Inicializar Swagger en main.ts:  
   Abre el archivo src/main.ts y añade el código para configurar Swagger.  
   **src/main.ts**  
   TypeScript  
   import { NestFactory } from '@nestjs/core';  
   import { AppModule } from './app.module';  
   import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';  
     
   async function bootstrap() {  
    const app = await NestFactory.create(AppModule);  
     
    // Configuración de CORS para permitir peticiones desde el frontend  
    app.enableCors({  
    origin: 'http://localhost:3000', // La URL de tu frontend Next.js  
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',  
    credentials: true,  
    });  
     
    // Configuración de Swagger  
    const config = new DocumentBuilder()  
    .setTitle('API de Gestión de Personal')  
    .setDescription('Documentación de la API para la aplicación de gestión.')  
    .setVersion('1.0')  
    .build();  
    const document = SwaggerModule.createDocument(app, config);  
    SwaggerModule.setup('api-docs', app, document); // La documentación estará en /api-docs  
     
    await app.listen(3001); // La API se ejecutará en el puerto 3001  
   }  
   bootstrap();

#### **Paso 1.5: Arrancar y Verificar**

1. **Arranca la aplicación en modo desarrollo:**  
   Bash  
   npm run start:dev
2. **Verifica que todo funciona:**
   * La terminal no debería mostrar errores de conexión a la base de datos.
   * Abre tu navegador y ve a **http://localhost:3001/api-docs**. Deberías ver la interfaz de Swagger con el controlador AppController por defecto.

¡Tu backend está listo para empezar a construir los módulos y la lógica de negocio!

### **Parte 2: Creación del Entorno de Frontend (Next.js)**

Ahora crearemos la aplicación de frontend con Next.js, TypeScript y Tailwind CSS.

#### **Paso 2.1: Crear el Proyecto Next.js**

1. En tu terminal, sitúate en el mismo directorio donde creaste el backend (no dentro de la carpeta mi-api-gestion).
2. Ejecuta el siguiente comando para crear una nueva aplicación Next.js.  
   Bash  
   npx create-next-app@latest mi-app-gestion  
     
   Durante la instalación, te hará varias preguntas. Responde así:
   * Would you like to use TypeScript? > **Yes1**
   * Would you like to use ESLint? > **Yes23**
   * Would you like to use Tailwind CSS? > **Yes45**
   * Would you like to use \src/` directory?` > **Yes67**
   * Would you like to use App Router? > **Yes** (Es la nueva y recomendada forma de trabajar en Next.js).
   * Would you like to customize the default import alias? > **No** (presiona Enter).
3. Esto creará una carpeta mi-app-gestion con la estructura del proyecto. Entra en ella:  
   Bash  
   cd mi-app-gestion

#### **Paso 2.2: Instalar Librerías Adicionales**

Instalaremos **TanStack Query** para gestionar los datos de la API y **Axios** como cliente HTTP para hacer las peticiones.

Bash

npm install @tanstack/react-query axios

#### **Paso 2.3: Configurar TanStack Query (React Query)**

Para poder usar React Query en toda la aplicación, necesitamos envolver nuestra aplicación en un proveedor (Provider).

1. Crea un nuevo archivo en src/app/providers.tsx.  
   **src/app/providers.tsx**  
   TypeScript  
   'use client'; // Este archivo debe ser un componente de cliente  
     
   import { QueryClient, QueryClientProvider } from '@tanstack/react-query';  
   import { useState } from 'react';  
     
   export default function Providers({ children }: { children: React.ReactNode }) {  
    const [queryClient] = useState(() => new QueryClient());  
     
    return (  
    <QueryClientProvider client={queryClient}>  
    {children}  
    </QueryClientProvider>  
    );  
   }
2. Ahora, abre el archivo src/app/layout.tsx (el layout principal de tu aplicación) y usa este Provider para envolver el contenido.  
   **src/app/layout.tsx**  
   TypeScript  
   import type { Metadata } from 'next';  
   import { Inter } from 'next/font/google';  
   import './globals.css';  
   import Providers from './providers'; // Importamos nuestro nuevo Provider  
     
   const inter = Inter({ subsets: ['latin'] });  
     
   export const metadata: Metadata = {  
    title: 'Gestión de Personal',  
    description: 'Aplicación de gestión para Servicios Operativos',  
   };  
     
   export default function RootLayout({  
    children,  
   }: Readonly<{  
    children: React.ReactNode;  
   }>) {  
    return (  
    <html lang="es">  
    <body className={inter.className}>  
    <Providers> {/\* Envolvemos los children con el Provider \*/}  
    {children}  
    </Providers>  
    </body>  
    </html>  
    );  
   }

#### **Paso 2.4: Ejemplo de Conexión con la API**

Vamos a modificar la página principal para que haga una llamada a nuestro backend.

1. Primero, vamos a crear una instancia de Axios para no tener que escribir la URL base de la API cada vez. Crea un archivo en src/lib/api.ts.  
   **src/lib/api.ts**  
   TypeScript  
   import axios from 'axios';  
     
   const api = axios.create({  
    baseURL: 'http://localhost:3001', // La URL de tu backend NestJS  
    withCredentials: true,  
   });  
     
   export default api;
2. Ahora, modifica la página principal en src/app/page.tsx para usar React Query y Axios y obtener el mensaje "Hello World!" de la API.  
   **src/app/page.tsx**  
   TypeScript  
   'use client'; // Necesario para usar hooks como useQuery  
     
   import { useQuery } from '@tanstack/react-query';  
   import api from '@/lib/api';  
     
   // Función que obtendrá los datos de la API  
   const getHelloMessage = async (): Promise<string> => {  
    const { data } = await api.get('/');  
    return data;  
   };  
     
   export default function Home() {  
    // Usamos el hook useQuery para llamar a la API  
    const { data, isLoading, isError, error } = useQuery({  
    queryKey: ['helloMessage'], // Clave única para esta consulta  
    queryFn: getHelloMessage, // Función que se ejecutará  
    });  
     
    return (  
    <main className="flex min-h-screen flex-col items-center justify-center p-24">  
    <div className="z-10 w-full max-w-5xl items-center justify-center font-mono text-sm text-center">  
    <h1 className="text-4xl font-bold mb-8">Aplicación de Gestión</h1>  
    <div className="p-6 border rounded-lg bg-gray-50 dark:bg-gray-800">  
    <p className="text-lg">Mensaje desde la API:</p>  
    {isLoading && <p className="text-blue-500">Cargando...</p>}  
    {isError && <p className="text-red-500">Error: {error.message}</p>}  
    {data && (  
    <p className="text-2xl font-semibold text-green-600 mt-2">  
    "{data}"  
    </p>  
    )}  
    </div>  
    </div>  
    </main>  
    );  
   }

#### **Paso 2.5: Arrancar y Verificar**

1. Abre **una nueva terminal** (mantén la del backend en ejecución).
2. Navega a la carpeta del frontend: cd mi-app-gestion.
3. Ejecuta el comando de desarrollo:  
   Bash  
   npm run dev
4. Abre tu navegador y ve a **http://localhost:3000**. Deberías ver el título, un mensaje de "Cargando..." y, después de un momento, el mensaje "Hello World!" obtenido de tu API.

### **Resumen del Entorno de Trabajo**

¡Felicidades! Ahora tienes:

1. Un **proyecto de backend** (mi-api-gestion) en el puerto 3001, conectado a tu base de datos MySQL y con una API documentada en http://localhost:3001/api-docs.
2. Un **proyecto de frontend** (mi-app-gestion) en el puerto 3000, listo para consumir la API con React Query y estilizado con Tailwind CSS.
3. Ambos proyectos están configurados con **TypeScript** desde el inicio.

Tu siguiente paso será empezar a definir las entidades (tablas) en el backend con TypeORM, crear los módulos, controladores y servicios para una funcionalidad (ej. "Personal"), y consumir esos nuevos endpoints desde tu frontend.

**Fuentes**

1. <https://stackoverflow.com/questions/68610452/error-when-connecting-nesjs-app-to-sql-server-express-failed-to-connect-to-loc?rq=1>

2. <https://github.com/Tony133/nestjs-api-boilerplate-jwt> sujeto a licencia (MIT)

3. <https://github.com/Med-Aymen99/helpet-app>

4. <https://github.com/liemlylac/kots-nestjs> sujeto a licencia (MIT)

5. <https://github.com/KevinRivera1/Nesjs-Rest-API>

6. <https://www.contentful.com/blog/build-blog-next-js-tailwind-css-contentful/>

7. <https://ably.com/blog/informational-dashboard-with-nextjs-and-recharts>

8. <https://github.com/flytaly/feedmailu>

9. <https://plainenglish.io/community/next-js-app-router-graphql-codegen-and-tanstack-query>

10. <https://www.omarileon.me/blog/react-promises>

11. <https://github.com/PaginaIbero/iberoofficial>

12. <https://dapp-world.com/smartbook/how-to-login-and-fetch-lens-profile-or-lens-handle-lWN6>