



PYTHON + ANGULAR



INTRODUCCIÓN

La arquitectura implementada se fundamenta en una estructura moderna y escalable, utilizando Python en el backend, Angular en el frontend y MySQL como sistema de gestión de bases de datos. Este enfoque permite desarrollar aplicaciones web robustas, seguras y con un alto rendimiento, garantizando una experiencia de usuario fluida y una administración eficiente de la información.

En el lado del servidor, Python proporciona un entorno flexible y eficiente para la creación de APIs REST, facilitando la lógica de negocio, la validación de datos y la comunicación con la base de datos. Como interfaz de usuario, Angular permite construir aplicaciones dinámicas y reactivas, manteniendo una estructura organizada y modular que mejora la mantenibilidad del proyecto. Finalmente, MySQL asegura un almacenamiento confiable y estructurado, permitiendo consultas optimizadas y una gestión precisa de los datos.

Este conjunto tecnológico permite desarrollar soluciones web completas, integrando un frontend intuitivo, un backend sólido y una base de datos estable, logrando aplicaciones profesionales y listas para producción.

OBJETIVO DEL PROYECTO

El objetivo principal de este proyecto es fortalecer mis competencias en el desarrollo web utilizando una arquitectura completa basada en **Python con Flask API** como tecnología backend, **Angular** como framework frontend y **MySQL** como sistema de gestión de bases de datos.

A través de esta implementación, busco dominar la creación de APIs REST con Python, integrar correctamente los servicios del backend con un frontend dinámico desarrollado en Angular y gestionar datos de manera eficiente utilizando MySQL. Este proceso de aprendizaje me permitirá comprender el flujo completo de una aplicación web moderna, desde la lógica del servidor y la interacción con la base de datos hasta la presentación final para el usuario.

Prerrequisitos y Opciones de Instalación

Para desarrollar el proyecto utilizando **Angular como frontend, Python (Flask API) como backend** y **MySQL** como base de datos, es necesario contar con los siguientes prerequisitos instalados en el equipo:

1. Node.js

Angular requiere Node.js para ejecutar sus herramientas internas y gestionar dependencias.

- Si el usuario no tiene Node.js instalado, puede buscar “**Node.js LTS**” en su navegador y descargar la versión **LTS (Long Term Support)** desde el sitio oficial.
- La versión LTS es la más recomendada por su estabilidad y soporte extendido.

2. Python

Python es esencial para desarrollar el backend utilizando **Fastapi**.

- Se recomienda instalar Python en su versión más reciente compatible y verificar que el comando python o python3 funcione correctamente en la terminal.

3. MySQL

Para la base de datos se requiere MySQL, que puede instalarse de distintas formas según la preferencia del usuario:

- **Instalación tradicional** (MySQL Server).
- **XAMPP**, que incluye MySQL dentro de un paquete de herramientas fácil de administrar.
- **Docker**, descargando la imagen oficial de MySQL para una instalación más rápida y aislada del sistema.

4. Angular CLI

El Angular CLI es la herramienta que permite crear, compilar y gestionar proyectos de Angular.

- Se instala mediante el comando:
- `npm install -g @angular/cli`

5. Editor de Código

El desarrollador puede utilizar el editor de su preferencia. Algunas opciones recomendadas son:

- **Visual Studio Code**
 - **JetBrains WebStorm**
 - **Sublime Text**
 - **Cursor**
-

Preparación del Entorno de Desarrollo

Una vez que contamos con todo el stack instalado (Node.js, Python, Angular CLI y MySQL), procedemos a preparar nuestro entorno de trabajo. Para ello, abrimos una terminal — puede ser **CMD**, **PowerShell**, **Git Bash**, **Warp**, o cualquier otro terminal de preferencia del usuario.

1. Verificación de Python y PIP

Antes de continuar, validamos que Python esté correctamente instalado ejecutando los siguientes comandos:

```
python --version
```

En sistemas Linux o macOS, también puede utilizarse:

```
python3 --version
```

Luego, verificamos la instalación de **pip**, el gestor de paquetes de Python:

```
pip --version
```

Si ambos comandos muestran una versión válida, podemos continuar.

2. Crear un Entorno Virtual

Para mantener aisladas las dependencias del proyecto, creamos un entorno virtual. El comando varía ligeramente según el sistema operativo:

- **Windows:**
 - `python -m venv .venv`
- **Linux / macOS:**

- `python3 -m venv .venv`

Una vez creado, activamos el entorno virtual:

- **Windows (CMD):**
 - `.venv\Scripts\activate`
- **Windows (PowerShell):**
 - `.venv\Scripts\Activate.ps1`
- **Linux / macOS:**
 - `source .venv/bin/activate`

3. Instalación de Librerías para el Servicio REST

Con el entorno virtual activo, procedemos a instalar las librerías necesarias para construir nuestra API REST con FastAPI y conectarnos a MySQL:

`pip install fastapi uvicorn mysql-connector-python pydantic`

- **FastAPI:** framework para desarrollar la API.
- **Uvicorn:** servidor ASGI de alto rendimiento.
- **mysql-connector-python:** conector oficial para comunicarnos con MySQL.
- **Pydantic:** validación de datos y modelos.

Conexión a la Base de Datos y Creación de la Estructura Inicial

Una vez instaladas todas las librerías necesarias para el backend, procederemos a establecer la conexión con nuestra base de datos **MySQL** utilizando una herramienta gráfica como **MySQL Workbench**.

Esto nos permitirá administrar la base de datos de forma visual y ejecutar consultas SQL con mayor facilidad.

1. Levantar los Servicios de MySQL

Dependiendo de la instalación que estemos utilizando, seguiremos uno de los siguientes métodos:

Si utilizamos XAMPP

1. Abrimos el panel de control de XAMPP.
2. Iniciamos los módulos **Apache** y **MySQL**.
3. Luego abrimos **phpMyAdmin** o MySQL Workbench para gestionar la base de datos.

Si utilizamos Docker en Linux

Ejecutamos el siguiente comando para crear y levantar un contenedor de MySQL:

```
docker run --name mysql \
-e MYSQL_ROOT_PASSWORD=dariozayas \
-p 3306:3306 \
-d mysql:8
```

Este comando:

- Crea un contenedor llamado mysql.
- Asigna la contraseña del usuario root.
- Expone el puerto 3306.
- Ejecuta la versión 8 de MySQL.

Si utilizamos Docker en Windows

El comando es exactamente el mismo, ejecutado desde **PowerShell**, **CMD** o **Git Bash**:

```
docker run --name mysql ` 
-e MYSQL_ROOT_PASSWORD=dariozayas ` 
-p 3306:3306 ` 
-d mysql:8
```

(En PowerShell se pueden usar los acentos invertidos ` para dividir líneas, o se puede escribir el comando todo en una sola línea).

2. Acceder a la Base de Datos con MySQL Workbench

Cuando el servicio esté levantado correctamente:

1. Abrimos **MySQL Workbench**.
 2. Nos conectamos utilizando los datos configurados (host, puerto, usuario y contraseña).
 3. Una vez dentro, abrimos un nuevo **Query Tab** para ejecutar nuestros comandos SQL.
-

3. Creación de la Base de Datos y Tablas Iniciales

Dentro del archivo de consulta (Query), escribimos las siguientes sentencias SQL para crear la base de datos y la tabla principal del proyecto:

```
CREATE DATABASE user;
```

```
USE user;
```

```
CREATE TABLE user (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(200),
    description VARCHAR(400),
    creado_en TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    actualizado_en TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

```
INSERT INTO user (name, description)
VALUES ('Juan Pérez', 'Esta es la primera prueba');
```

Estas instrucciones crean:

- Una base de datos llamada **user**.
 - Una tabla llamada **user** con campos para ID, nombre, descripción y fechas de creación/actualización.
 - Una inserción inicial de prueba.
-

Creación del Archivo principal del Backend (main.py)

Una vez que hemos configurado nuestro entorno virtual e instalado todas las librerías necesarias, procederemos a crear la estructura base del backend con **FastAPI**.

Para ello, ingresamos a la carpeta del proyecto y la abrimos en nuestro editor de preferencia, como **VS Code**.

Dentro del proyecto, creamos un archivo llamado **main.py**, el cual contendrá toda la lógica inicial de nuestra API REST. En este archivo definimos:

1. Importaciones principales

El código inicia importando las herramientas esenciales:

- **FastAPI**, para construir la API.
- **CORSMiddleware**, para permitir solicitudes desde el frontend (Angular).
- **Pydantic**, para la validación de datos.
- **mysql.connector**, para conectarnos a la base de datos MySQL.
- **asynccontextmanager**, utilizado para manejar el ciclo de vida de la aplicación.

Estas importaciones permiten estructurar un servicio REST completo, seguro y validado.

2. Configuración de la base de datos

Se define un diccionario DB_CONFIG con:

- Host de la base de datos
- Usuario y contraseña
- Nombre de la base de datos

Luego, se crea la función **get_db_connection()**, encargada de abrir la conexión y manejar posibles errores.

Esta función será utilizada por los endpoints que necesiten interactuar con MySQL.

3. Modelos de datos (Pydantic Models)

El archivo define varios modelos con Pydantic:

- **UsuarioBase**: estructura base con name y description.
- **UsuarioCreate**: para creación de nuevos usuarios.
- **UsuarioUpdate**: para actualización parcial.
- **Usuario**: modelo completo que incluye id.

Estos modelos aseguran que la API reciba y devuelva información validada y consistente.

4. Inicialización de la base de datos

A través del decorador `@asynccontextmanager`, se configura un ciclo de vida que:

- Al iniciar la aplicación, crea la tabla user si no existe.
- Evita errores de estructura y facilita el despliegue.
- Cierra la aplicación correctamente al finalizar.

Esto permite que el backend esté listo sin necesidad de crear manualmente la tabla.

5. Configuración de la aplicación FastAPI

Se crea la instancia principal:

```
app = FastAPI(  
    title="API CRUD de Usuarios",  
    description="API REST para gestionar usuarios con operaciones CRUD completas",  
    version="1.0.0",  
    lifespan=lifespan  
)
```

Además, se configura **CORS** para permitir solicitudes desde Angular.

6. Endpoints CRUD completos

El archivo incluye todos los endpoints necesarios para un CRUD funcional:

1. **GET /** → mensaje de bienvenida.

2. **POST /usuarios/** → crear usuario.
3. **GET /usuarios/** → obtener todos los usuarios.
4. **GET /usuarios/{id}** → obtener usuario por ID.
5. **PUT /usuarios/{id}** → actualizar usuario.
6. **DELETE /usuarios/{id}** → eliminar usuario.

Cada endpoint gestiona errores, valida datos y usa sentencias SQL seguras.

7. Ejecución del servidor

Para iniciar el backend, utilizamos:

```
uvicorn main:app --reload
```

- main → hace referencia al archivo main.py
 - app → es la instancia FastAPI
 - --reload → reinicia el servidor automáticamente al cambiar el código
-

```

# main.py
# Importa las clases y funciones necesarias de FastAPI, Pydantic y otras bibliotecas.
# FastAPI: El framework para construir APIs.
# HTTPException: Para manejar errores HTTP.
# Depends: Para la inyección de dependencias (no se usa directamente aquí, pero es común en FastAPI).
# CORSMiddleware: Para permitir solicitudes de diferentes orígenes (Cross-Origin Resource Sharing).
# BaseModel, Field, ConfigDict: De Pydantic, para la validación y configuración de modelos de datos.
# Optional, List: De typing, para definir tipos de datos.
# mysql.connector: El conector oficial de Python para MySQL.
# Error: Clase de excepción de mysql.connector.
# asynccontextmanager: Para gestionar recursos de forma asíncrona (en este caso, la base de datos).
from fastapi import FastAPI, HTTPException, Depends
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field, ConfigDict
from typing import Optional, List
import mysql.connector
from mysql.connector import Error
from contextlib import asynccontextmanager

# Define un diccionario con los parámetros de conexión a la base de datos MySQL.
DB_CONFIG = {
    "host": "localhost", # La dirección del servidor de la base de datos.
    "user": "root", # El nombre de usuario para la conexión.
    "password": "xxxx", # La contraseña del usuario.
    "database": "user" # El nombre de la base de datos a la que se conectará.
}

# Define una función para obtener una conexión a la base de datos.
def get_db_connection():
    try:
        # Intenta establecer una conexión usando la configuración definida en DB_CONFIG.
        conn = mysql.connector.connect(**DB_CONFIG)
        # Devuelve el objeto de conexión si tiene éxito.
        return conn
    except Error as e:
        # Si ocurre un error de conexión, lanza una excepción HTTP 500 con un mensaje de error.
        raise HTTPException(status_code=500, detail=f"Error de conexión a la base de datos: {str(e)}")
# Define un modelo Pydantic base para un usuario.
class UsuarioBase(BaseModel):
    """
    Modelo base con los campos comunes de un usuario.
    """
    # Define el campo 'name' como una cadena de texto, obligatorio, con longitud entre 1 y 100.
    name: str = Field(..., min_length=1, max_length=100, description="Nombre del usuario")
    # Define el campo 'description' como una cadena de texto, obligatorio, con longitud entre 1 y 100.
    description: str = Field(..., min_length=1, max_length=100, description="Descripción del usuario")
    # Configuración del modelo Pydantic.
    model_config = ConfigDict(
        from_attributes=True, # Permite que el modelo se cree a partir de atributos de objeto (ORM).
        json_schema_extra={ # Define un ejemplo para la documentación de la API.
            "example": {
                "name": "Juan Pérez",
                "description": "juan@example.com",
            }
        }
    )

```

```

# Define un modelo para la creación de un usuario, que hereda de UsuarioBase.
class UsuarioCreate(UsuarioBase):
    """Modelo para crear un usuario"""
    # No añade campos adicionales, utiliza los de la clase base.
    pass

# Define un modelo para la actualización de un usuario.
class UsuarioUpdate(BaseModel):
    """
    Modelo para actualizar un usuario (todos los campos son opcionales).
    """
    # Define el campo 'name' como opcional (puede ser None).
    name: Optional[str] = Field(None, min_length=1, max_length=100)
    # Define el campo 'description' como opcional (puede ser None).
    description: Optional[str] = Field(None, min_length=1, max_length=100)

# Define el modelo completo de un usuario, incluyendo el ID.
class Usuario(UsuarioBase):
    """
    Modelo completo de usuario incluyendo ID.
    """
    # Define el campo 'id' como un entero obligatorio.
    id: int = Field(..., description="ID único del usuario")

    # Configuración del modelo Pydantic.
    model_config = ConfigDict(
        from_attributes=True, # Permite la creación desde atributos de objeto.
        json_schema_extra={ # Define un ejemplo para la documentación.
            "user": {
                "name": "Juan Pérez",
                "description": "juan@example.com"
            }
        }
    )

# Define un gestor de ciclo de vida asíncrono para la aplicación FastAPI.
@asynccontextmanager
async def lifespan(app: FastAPI):
    # Código que se ejecuta al iniciar la aplicación (startup).
    try:
        # Obtiene una conexión a la base de datos.
        connection = get_db_connection()
        # Crea un cursor para ejecutar comandos SQL.
        cursor = connection.cursor()
        # Ejecuta una consulta SQL para crear la tabla 'user' si no existe.
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS user (
                id INT AUTO_INCREMENT PRIMARY KEY,
                name VARCHAR(200),
                description VARCHAR(400),
                creado_en TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                actualizado_en TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
            );
        """)
        # Confirma los cambios en la base de datos.
        connection.commit()
        # Cierra el cursor.
        cursor.close()
        # Cierra la conexión.
        connection.close()
        # Imprime un mensaje de éxito en la consola.
        print("Base de datos inicializada correctamente")
    except Exception as e:
        # Si ocurre un error, imprime un mensaje de error.
        print(f"Error al inicializar base de datos: {e}")

    # La aplicación se ejecuta después de este 'yield'.
    yield

    # Código que se ejecuta al apagar la aplicación (shutdown).
    print("Cerrando aplicación...")

# Crea una instancia de la aplicación FastAPI.
app = FastAPI(
    title="API CRUD de Usuarios", # Título de la API.
    description="API REST para gestionar usuarios con operaciones CRUD completas", #
    version="1.0.0", # Versión de la API.
    lifespan=lifespan # Asigna el gestor de ciclo de vida.
)

```

```

# Añade el middleware de CORS a la aplicación.
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Permite solicitudes de cualquier origen.
    allow_credentials=True, # Permite el envío de credenciales (cookies, etc.).
    allow_methods=["*"], # Permite todos los métodos HTTP (GET, POST, etc.).
    allow_headers=["*"], # Permite todas las cabeceras HTTP.
)

# Define un endpoint para la ruta raíz de la API.
@app.get("/", tags=["Root"])
async def root():
    # Devuelve un mensaje JSON de bienvenida.
    return {"mensaje": "API CRUD de Usuarios funcionando correctamente"}

# Define un endpoint para crear un nuevo usuario (método POST).
@app.post("/usuarios/", response_model=Usuario, status_code=201, tags=["Usuarios"])
async def crear_usuario(usuario: UsuarioCreate):
    try:
        # Obtiene una conexión a la base de datos.
        connection = get_db_connection()
        # Crea un cursor.
        cursor = connection.cursor()

        # Define la consulta SQL para insertar un nuevo usuario.
        query = "INSERT INTO user (name, description) VALUES (%s, %s)"
        # Define los valores a insertar, tomados del modelo de entrada.
        values = (usuario.name, usuario.description)

        # Ejecuta la consulta con los valores.
        cursor.execute(query, values)
        # Confirma la transacción.
        connection.commit()

        # Obtiene el ID del último registro insertado.
        usuario_id = cursor.lastrowid
        # Cierra el cursor.
        cursor.close()
        # Cierra la conexión.
        connection.close()

        # Devuelve el nuevo usuario creado, incluyendo su ID.
        return Usuario(id=usuario_id, **usuario.dict())
    except mysql.connector.IntegrityError:
        # Si ocurre un error de integridad (ej. 'description' duplicado), lanza un error 400.
        raise HTTPException(status_code=400, detail="El description ya está registrado")
    except Error as e:
        # Para otros errores de base de datos, lanza un error 500.
        raise HTTPException(status_code=500, detail=f"Error al crear usuario: {str(e)}")

# Define un endpoint para obtener todos los usuarios (método GET).
@app.get("/usuarios/", response_model=List[Usuario], tags=["Usuarios"])
async def obtener_usuarios():
    try:
        # Obtiene una conexión a la base de datos.
        connection = get_db_connection()
        # Crea un cursor que devuelve filas como diccionarios.
        cursor = connection.cursor(dictionary=True)

        # Ejecuta la consulta para seleccionar todos los usuarios.
        cursor.execute("SELECT * FROM user")
        # Obtiene todos los resultados.
        usuarios = cursor.fetchall()

        # Cierra el cursor.
        cursor.close()
        # Cierra la conexión.
        connection.close()

        # Devuelve la lista de usuarios.
        return usuarios
    except Error as e:
        # Si hay un error, lanza una excepción HTTP 500.
        raise HTTPException(status_code=500, detail=f"Error al obtener usuarios: {str(e)}")

```

```
# Define un endpoint para obtener un usuario por su ID (método GET).
@app.get("/usuarios/{usuario_id}", response_model=Usuario, tags=["Usuarios"])
async def obtener_usuario(usuario_id: int):
    try:
        # Obtiene una conexión a la base de datos.
        connection = get_db_connection()
        # Crea un cursor que devuelve filas como diccionarios.
        cursor = connection.cursor(dictionary=True)

        # Ejecuta la consulta para seleccionar un usuario por su ID.
        cursor.execute("SELECT * FROM user WHERE id = %s", (usuario_id,))
        # Obtiene un solo resultado.
        usuario = cursor.fetchone()

        # Cierra el cursor.
        cursor.close()
        # Cierra la conexión.
        connection.close()

        # Si no se encuentra el usuario, lanza un error 404.
        if not usuario:
            raise HTTPException(status_code=404, detail="Usuario no encontrado")

        # Devuelve el usuario encontrado.
        return usuario

    except Error as e:
        # Si hay un error, lanza una excepción HTTP 500.
        raise HTTPException(status_code=500, detail=f"Error al obtener usuario: {str(e)}")

# Define un endpoint para actualizar un usuario por su ID (método PUT).
@app.put("/usuarios/{usuario_id}", response_model=Usuario, tags=["Usuarios"])
async def actualizar_usuario(usuario_id: int, usuario: UsuarioUpdate):
    try:
        # Obtiene una conexión a la base de datos.
        connection = get_db_connection()
        # Crea un cursor que devuelve filas como diccionarios.
        cursor = connection.cursor(dictionary=True)

        # Listas para construir la consulta de actualización dinámicamente.
        campos = []
        valores = []

        # Si se proporciona un nuevo nombre, lo añade a la consulta.
        if usuario.name is not None:
            campos.append("name = %s")
            valores.append(usuario.name)
        # Si se proporciona una nueva descripción, la añade a la consulta.
        if usuario.description is not None:
            campos.append("description = %s")
            valores.append(usuario.description)

        # Si no se proporcionó ningún campo para actualizar, lanza un error 400.
        if not campos:
            raise HTTPException(status_code=400, detail="No hay campos para actualizar")

        # Añade el ID del usuario al final de la lista de valores.
        valores.append(usuario_id)

        # Construye la consulta SQL de actualización.
        query = f"UPDATE user SET {', '.join(campos)} WHERE id = %s"
        # Ejecuta la consulta.
        cursor.execute(query, valores)
        # Confirma la transacción.
        connection.commit()

        # Si ninguna fila fue afectada, significa que el usuario no fue encontrado.
        if cursor.rowcount == 0:
            raise HTTPException(status_code=404, detail="Usuario no encontrado")

        # Vuelve a consultar la base de datos para obtener el usuario actualizado.
        cursor.execute("SELECT * FROM user WHERE id = %s", (usuario_id,))
        usuario_actualizado = cursor.fetchone()

        # Cierra el cursor.
        cursor.close()
        # Cierra la conexión.
        connection.close()

        # Devuelve el usuario actualizado.
        return usuario_actualizado

    except mysql.connector.IntegrityError:
        # Si hay un error de integridad, lanza un error 400.
        raise HTTPException(status_code=400, detail="El description ya está registrado")
    except Error as e:
        # Para otros errores, lanza un error 500.
        raise HTTPException(status_code=500, detail=f"Error al actualizar usuario: {str(e)}")
```

```
● ● ●

# Define un endpoint para eliminar un usuario por su ID (método DELETE).
@app.delete("/usuarios/{usuario_id}", status_code=204, tags=["Usuarios"])
async def eliminar_usuario(usuario_id: int):
    try:
        # Obtiene una conexión a la base de datos.
        connection = get_db_connection()
        # Crea un cursor.
        cursor = connection.cursor()

        # Ejecuta la consulta para eliminar el usuario.
        cursor.execute("DELETE FROM user WHERE id = %s", (usuario_id,))
        # Confirma la transacción.
        connection.commit()

        # Si ninguna fila fue afectada, el usuario no fue encontrado.
        if cursor.rowcount == 0:
            raise HTTPException(status_code=404, detail="Usuario no encontrado")

        # Cierra el cursor.
        cursor.close()
        # Cierra la conexión.
        connection.close()

        # Devuelve una respuesta sin contenido (código 204).
        return None

    except Error as e:
        # Si hay un error, lanza una excepción HTTP 500.
        raise HTTPException(status_code=500, detail=f"Error al eliminar usuario: {str(e)}")

    # Comentario para indicar cómo ejecutar la aplicación con uvicorn.
    # uvicorn: El servidor ASGI.
    # main: El nombre del archivo Python (main.py).
    # app: El objeto FastAPI dentro del archivo.
    # --reload: Hace que el servidor se reinicie automáticamente al detectar cambios en el código.
    # Ejecutar: uvicorn main:app --reload
```

Configuración del Frontend con Angular

Con el backend completamente funcional, procedemos a preparar el entorno del **frontend**, el cual desarrollaremos utilizando **Angular**. Este módulo será responsable de la interfaz gráfica y la interacción directa con los usuarios.

1. Crear el Proyecto en Angular

Abrimos una terminal (CMD, Git Bash, PowerShell o la de tu editor) y ejecutamos el siguiente comando para crear un nuevo proyecto:

```
ng new nombre-del-proyecto
```

Puedes reemplazar *nombre-del-proyecto* con el nombre que deseas.

Una vez generado el proyecto, ingresamos a la carpeta recién creada:

```
cd nombre-del-proyecto
```

2. Instalación de PrimeNG y el Sistema de Temas

Dentro del proyecto, instalamos **PrimeNG** y el paquete de temas de **PrimeUI**, los cuales nos permitirán contar con componentes modernos, accesibles y fáciles de usar:

```
npm install primeng @primeui/themes
```

Luego, localizamos el archivo:

```
src/app/app.config.ts
```

y pegamos la siguiente configuración:

```
import { ApplicationConfig } from '@angular/core';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';
import { providePrimeNG } from 'primeng/config';
import Aura from '@primeui/themes/aura';
```

```
export const appConfig: ApplicationConfig = {
```

```
  providers: [
```

```
    provideAnimationsAsync(),
```

```
    providePrimeNG({
```

```
      theme: {
```

```
        preset: Aura
```

```
      }
```

```
    })
```

```
  ]
```

```
};
```

Esta configuración habilita:

- Animaciones de Angular de forma asíncrona.
 - Configuración global de PrimeNG.
 - Aplicación del tema **Aura** para dar estilo a los componentes.
-

3. Instalación e Importación de PrimeIcons

Regresamos a la terminal y ejecutamos:

```
npm install primeicons
```

Luego, abrimos el archivo raíz de estilos:

```
src/styles.css
```

y añadimos:

```
@import "primeicons/primeicons.css";
```

Esto nos permitirá utilizar la amplia librería de íconos de PrimeIcons en toda la aplicación.

4. Instalación de TailwindCSS PrimeUI

Para potenciar aún más el diseño, instalamos el paquete:

```
npm install tailwindcss-primeui
```

Y en el mismo archivo styles.css, agregamos:

```
@import "tailwindcss";
```

```
@import "tailwindcss-primeui";
```

Con esto, integramos:

- TailwindCSS, el framework de utilidades más popular.
- PrimeUI + TailwindCSS, una combinación moderna que facilita la construcción de interfaces limpias y atractivas.