



UNIVERSIDAD
NACIONAL
DE COLOMBIA

UNIVERSIDAD NACIONAL DE COLOMBIA

Facultad de Ingeniería

Ingeniería de Software I

Grupo: 1

Profesor: Oscar Eduardo Álvarez Rodríguez

Integrantes:

Nicolás Rodríguez Tapia
Juan David Alarcón Sanabria
José Leonardo Pinilla Zamora
David Nicolás Urrego Botero

24 de octubre de 2025

Crea tu aplicación: Configuración inicial y “Hola Mundo” con Java, Spring Boot y PostgreSQL.

1. Introducción

En este tutorial aprenderás a crear, paso a paso, una aplicación web con Java y Spring Boot que guarda y muestra un mensaje sencillo, como "Hola Mundo". El objetivo es entender cómo una aplicación puede conectarse a una base de datos (PostgreSQL), guardar información y luego mostrarla en una página web.

Para lograr esto, usaremos Spring Boot, un framework que facilita mucho el desarrollo con Java. Verás en tiempo real cómo se comunican sus diferentes partes: desde el navegador que hace una solicitud, hasta el servidor que consulta la base de datos y devuelve la respuesta.

Al final del tutorial, tendrás una aplicación web completa y funcional que muestra en el navegador un mensaje almacenado en la base de datos, construida desde cero.

2. Objetivos de aprendizaje

Al finalizar este tutorial habrás aprendido a:

- Crear un nuevo proyecto con **Spring Boot**, la herramienta que usaremos para construir nuestra aplicación web.
- Instalar y ejecutar una base de datos **PostgreSQL** usando **Docker**, para que todo funcione fácilmente sin configuraciones complicadas.
- Indicarle a la aplicación cómo conectarse a esa base de datos mediante un archivo de configuración.
- Crear una **entidad** llamada “Mensaje”, que representará la información que se guardará en la base de datos.
- Entender cómo se comunican las distintas **capas** del programa: el controlador que recibe las solicitudes, el servicio que maneja la lógica y el repositorio que accede a los datos.
- Probar la aplicación completa desde el navegador, presionando un botón que mostrará el mensaje guardado en la base de datos.

3. Conceptos clave

Antes de empezar, repasemos brevemente algunas herramientas que usaremos a lo largo del tutorial:

- **JDK (Java Development Kit):** Es el conjunto de herramientas que permite crear y ejecutar programas en Java. Sin él, el computador no sabría cómo interpretar el código Java que escribimos.
- **Maven:** Es una herramienta que nos ayuda a organizar el proyecto y

manejar sus dependencias (las librerías externas que necesita nuestro código, como Spring Boot). También se encarga de compilar y ejecutar la aplicación.

- **Spring Boot:** Es un framework (una base lista para trabajar) que simplifica mucho el desarrollo de aplicaciones web con Java. Nos permite crear un servidor web que reciba solicitudes, conectarlo a una base de datos y enviar respuestas, todo sin tanta configuración complicada.
- **PostgreSQL:** Es una base de datos donde se guarda la información de nuestra aplicación, por ejemplo, el mensaje "Hola Mundo".
- **Docker y Docker Compose:** Son herramientas que permiten levantar servicios fácilmente en contenedores, es decir, entornos listos para usar. Gracias a ellas podremos tener PostgreSQL funcionando sin instalarlo manualmente.
- **ORM (JPA / Hibernate):** Son tecnologías que permiten conectar las clases de Java con las tablas de la base de datos. Así, podemos guardar o leer objetos Java sin escribir consultas SQL complicadas.

4. Prerrequisitos y cómo instalarlos

Java 17 (JDK 17)

Java será el lenguaje principal de nuestra aplicación. El JDK (Java Development Kit) permite compilar y ejecutar el código que escribiremos. Para comprobar si ya lo tienes instalado, abre una terminal o consola y escribe:

```
java -version
```

Si aparece algo como lo siguiente entonces Java está listo:

```
java version "17.0.10" 2024-01-16 LTS
Java(TM) SE Runtime Environment (build 17.0.10+11-LTS-240)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.10+11-LTS-240, mixed mode, sharing)
```

Si no, puedes descargarlo desde el sitio oficial donde también encontrarás guías de instalación:

<https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>

Docker y Docker Compose

Docker nos permitirá ejecutar PostgreSQL fácilmente sin instalaciones complicadas. Docker Compose nos ayuda a levantar servicios con un solo comando.

Para comprobar Docker puedes escribir:

```
docker --version
...

**Salida esperada:**
...

Docker version 24.x.x ...
```

Para comprobar Docker Compose:

```
docker-compose --version
...

**Salida esperada:**
...

Docker Compose version 2.x.x ...
```

Si no lo tienes instalado:

Windows/Mac: Descarga Docker Desktop desde <https://www.docker.com/products/docker-desktop> (incluye Docker Compose)
Linux: Sigue las instrucciones en <https://docs.docker.com/engine/install/>

Puede que tengas el plugin instalado, en ese caso debes probar con: `docker compose version`

Maven (opcional)

El proyecto que crearemos incluye Maven Wrapper, un script que descarga Maven automáticamente la primera vez que lo uses. Esto lo veremos más adelante en la sección de ejecución, sin embargo, si deseas instalarlo globalmente puedes encontrarlo en su página oficial:

<https://maven.apache.org>

Editor de código

Necesitarás un editor para escribir el código Java. Recomendamos cualquiera de los dos siguientes:

Visual Studio Code: <https://code.visualstudio.com/> (ligero y gratuito)

IntelliJ IDEA Community: <https://www.jetbrains.com/idea/download/> (más potente, específico para Java)

Navegador web

Cualquier navegador moderno servirá (Chrome, Firefox, Edge, Safari) para ver la interfaz de la aplicación.

5. Crear el proyecto con Spring Initializr (paso a paso)

Como primera tarea para la creación de nuestro proyecto, vamos a generar la estructura base de la aplicación usando Spring Initializr. Esta herramienta (oficial de Spring) permite crear un proyecto Spring Boot ya configurado, con todas las dependencias necesarias y una estructura lista para comenzar a programar.

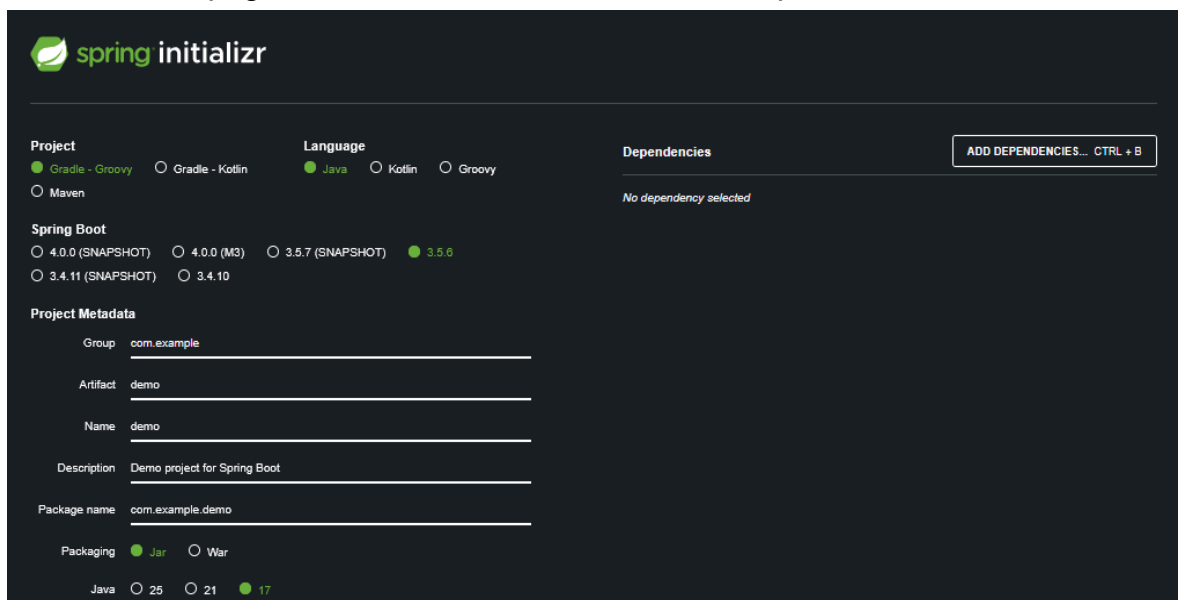
Podemos hacerlo desde la página web o directamente desde la línea de comandos; en este caso, usaremos primero la opción web por ser la más visual y sencilla.

Paso 1: Entrar a la web oficial

Abre tu navegador y visita la página: <https://start.spring.io>

Paso 2: Configurar el proyecto

Una vez en la página, verás un formulario con varias opciones.



The screenshot shows the Spring Initializr web interface. At the top left is the 'spring initializr' logo. Below it, the form is organized into sections: 'Project' with radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; and 'Dependencies' with a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'. The 'Spring Boot' section has radio buttons for versions: '4.0.0 (SNAPSHOT)', '4.0.0 (M3)', '3.5.7 (SNAPSHOT)', '3.5.6' (selected), '3.4.11 (SNAPSHOT)', and '3.4.10'. The 'Project Metadata' section contains input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). At the bottom, the 'Packaging' section has radio buttons for 'Jar' (selected) and 'War', and a 'Java' version selector with '25', '21', and '17' (selected).

Configura las opciones de la siguiente manera:

Campo	Valor	Descripción
Project	Maven	Indica que usamos Maven como herramienta para gestionar dependencias y compilación
Language	Java	El lenguaje de programación principal de nuestro proyecto
Spring Boot	3.5.x (la versión recomendada)	Versión del framework Spring Boot
Group	com.ejemplo	Define el nombre base del paquete raíz del proyecto
Artifact	holamundo (o el que gustes)	Nombre del proyecto y del archivo principal que se generará.
Name	holamundo (o el que gustes)	El nombre legible del proyecto, normalmente es igual al Artifact
Description	Aplicación Hola Mundo con Spring Boot (o la descripción que desees)	Una breve descripción de qué hace tu proyecto, aparecerá en el <code>pom.xml</code>
Package name	Se generará automáticamente como com.ejemplo.holamundo	Es la ruta donde estarán tus clases Java, combina Group y Artifact
Packaging	Jar	El formato en que se va a empaquetar tu aplicación, JAR es un archivo ejecutable que contiene todo el código compilado
Java	17	La versión de Java que usará el proyecto, debe coincidir con la

		versión que instalaste en tu PC
--	--	---------------------------------

Paso 3: Agregar las dependencias necesarias

En la sección de la derecha verás un botón que dice "ADD DEPENDENCIES". Haz clic en él y busca las siguientes dependencias una por una:

- **Spring Web:** Para crear controladores y servicios web (por ejemplo, endpoints o APIs).
- **Spring Data JPA:** Permite conectarse a bases de datos y trabajar con entidades sin escribir SQL directamente.
- **PostgreSQL Driver:** Es el conector necesario para comunicarse con la base de datos PostgreSQL.

Estas dependencias aseguran que la aplicación pueda responder peticiones HTTP y guardar datos en la base de datos. A medida que las agregues, aparecerán listadas en la sección de dependencias seleccionadas.

Paso 4: Generar y descargar el proyecto

Una vez configurado todo, haz clic en el botón verde "GENERATE" en la parte inferior de la página.

Esto descargará un archivo ZIP con el nombre `holamundo.zip` (o el nombre que hayas puesto en Artifact)

Busca el archivo descargado (normalmente en tu carpeta de Descargas), haz clic derecho sobre `holamundo.zip`, Selecciona "Extraer aquí" o "Extract here", Mueve la carpeta extraída a un lugar de trabajo cómodo, por ejemplo:

Windows: `C:\Users\TuNombre\proyectos\holamundo`

Mac/Linux: `/Users/tunombre/proyectos/holamundo`

Paso 5: Abrir y verificar la estructura

Abre la carpeta del proyecto, deberías ver una estructura similar a esta:



6. Estructura de carpetas y rutas exactas

Cuando generamos el proyecto con Spring Initializr, se creó una estructura básica de carpetas.

Spring Boot utiliza una organización estándar para mantener el código ordenado: el programa, las configuraciones y los recursos visibles desde el navegador están separados. A continuación veremos cómo está organizada la aplicación, qué archivos ya existen y cuáles se crearán más adelante.

Archivos que ya existen desde el inicio:

- **HolamundoApplication.java**: clase principal que inicia la aplicación.
- **application.properties**: archivo de configuración vacío (más adelante lo reemplazaremos por **application.yml**).
- **pom.xml**: define las dependencias y configuraciones de Maven.

A medida que avancemos en el tutorial, agregaremos nuevos archivos en las carpetas mostradas abajo. Cada uno cumple un propósito dentro de la arquitectura de la aplicación.

Tipo de archivo	Ruta relativa	Propósito
Entidad	src/main/java/com/ejemplo/holamundo/model/Mensaje.java	Representa una tabla de la base de datos.
Repositorio	src/main/java/com/ejemplo/holamundo/repository/MensajeRepository.java	Acceso a la base de datos (guardar y leer mensajes).
Servicio	src/main/java/com/ejemplo/holamundo/service/MensajeService.java	Lógica intermedia entre controlador y repositorio.
Controlador	src/main/java/com/ejemplo/holamundo/controller/MensajeController.java	Expone endpoints que el navegador puede llamar.
Data Loader (opcional)	src/main/java/com/ejemplo/holamundo/config/DataLoader.java	Inserta datos iniciales automáticamente.

Archivos de configuración y recursos:

Archivo o carpeta	Propósito
application.yml	Define la conexión a la base de datos y otras configuraciones.
data.sql (<i>opcional</i>)	Contendrá un registro inicial de ejemplo ("Hola Mundo").
static/index.html	Página web principal, mostrará el mensaje almacenado en la base de datos.
static/js/main.js	Contendrá el código JavaScript que llama al backend y muestra el resultado.

Dentro del proyecto hay una carpeta llamada **src/main/resources/static**, que Spring Boot utiliza para guardar los archivos visibles desde el navegador, como páginas HTML, imágenes o scripts. En pocas palabras, todo lo que coloques allí puede abrirse directamente escribiendo su nombre en el navegador, por ejemplo: <http://localhost:8080/index.html>. Esto ocurre porque Spring Boot incluye un pequeño servidor web interno que se encarga de mostrar automáticamente los archivos que estén dentro de esa carpeta.

Una vez completados los siguientes pasos del tutorial, la estructura de carpetas del proyecto se verá así:

```
holamundo/
├── pom.xml
└── src/
    ├── main/
    │   ├── java/com/ejemplo/holamundo/
    │   │   ├── HolamundoApplication.java
    │   │   ├── model/Mensaje.java
    │   │   ├── repository/MensajeRepository.java
    │   │   ├── service/MensajeService.java
    │   │   ├── controller/MensajeController.java
    │   │   └── config/DataLoader.java (opcional)
    │   ├── resources/
    │   │   ├── application.yml
    │   │   ├── data.sql (opcional)
    │   │   └── static/
    │   │       ├── index.html
    │   │       └── js/main.js
    └── test/
```

7. Arquitectura de capas y flujo de datos (Controller → Service → Repository)

La arquitectura de capas y el flujo de datos que el proyecto va a tener es el siguiente:

- Controller: El controlador recibe la petición HTTP y devuelve la respuesta.
- Service: Contiene la lógica del negocio (aquí mínima: leer/crear mensajes).
- Repository: Maneja operaciones con la base de datos (JPA/Hibernate).
- Entidad / Model: clase Java que representa una fila en la tabla SQL.

Por lo tanto, el flujo para el GET /hola sería el siguiente:

Cliente (navegador) → Controller(/hola) → Service (getMensaje) → Repository (consulta DB/Entidad) → Service → Controller → Cliente (navegador)

8. Levantar PostgreSQL con Docker (archivo y comandos)

Ahora, para crear la base de datos con PostgreSQL con Docker, crearemos un archivo llamado **docker-compose.yml** en la carpeta holamundo del tutorial.

El archivo debe contener lo siguiente:

```
version: '3.8'

services:
  db:
    image: postgres:15
    environment:
      POSTGRES_DB: hola_mundo
      POSTGRES_USER: hm_user
      POSTGRES_PASSWORD: hm_pass
    ports:
      - '5432:5432'
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```

Los comandos que debes escribir son los siguientes:

Recuerda empezar el comando con `sudo` si estás usando Linux y que tu terminal o consola se encuentre en el mismo directorio que tu archivo `docker-compose.yml`.

```
docker compose up -d
```

Si está correcto, deberías ver algo como esto:

```
[+] Running 15/15
✓ db Pulled 72.0s
  ✓ 38513bd72563 Pull complete 32.8s
  ✓ 4d7e468bab95 Pull complete 33.2s
  ✓ 9f4f471e8793 Pull complete 34.0s
  ✓ 0b1dc6e268e3 Pull complete 34.5s
  ✓ d7714ec68f7b Pull complete 35.7s
  ✓ 46ad9eec8822 Pull complete 36.3s
  ✓ e9148cac9d4d Pull complete 36.7s
  ✓ 47cf08e1e182 Pull complete 37.2s
  ✓ 18484353fd3e Pull complete 66.6s
  ✓ b994166fc0d8 Pull complete 67.1s
  ✓ d51fd24b08df Pull complete 67.5s
  ✓ 02d092d78b17 Pull complete 67.9s
  ✓ 8df7d2b03eac Pull complete 68.3s
  ✓ 4beb76b7345c Pull complete 68.8s
[+] Running 3/3
✓ Network holamundo_default Created 1.8s
✓ Volume holamundo_db-data Created 0.2s
✓ Container holamundo-db-1 Started 4.8s
```

Podemos verificar los contenedores en ejecución así:

```
# Verificar contenedores en ejecución
docker ps
```

Deberías ver algo como:

CONTAINER ID	IMAGE	COMMAND NAMES	CREATED	STATUS	PORTS
1504939067d9	postgres:15	"docker-entrypoint.s..." /tcp, [::]:5432->5432/tcp	5 minutes ago	Up 5 minutes	0.0.0.0:5432->5432

9. Configurar Spring Boot ([application.yml](#))

Para configurar Spring Boot (necesario para la construcción, configuración y despliegue de nuestra aplicación!) lo primero que necesitamos hacer es crear o editar el archivo [application.yml](#) en la siguiente ruta:

[src/main/resources/application.yml](#)

Este archivo debe tener el siguiente contenido:

```
server:
  port: 8080

spring:
  datasource:
    url: jdbc:postgresql://${DB_HOST:localhost}:${DB_PORT:5432}/${DB_NAME:hola_mundo}
    username: ${DB_USER:hm_user}
    password: ${DB_PASS:hm_pass}
  jpa:
    hibernate:
      ddl-auto: update
      show-sql: true
  sql:
    init:
      mode: always
```

Explicación sencilla: **-url:** dónde está la base de datos (usamos variables de entorno con valores por defecto). **-ddl-auto: update:** permite que JPA cree/actualice tablas automáticamente (útil en desarrollo). **-show-sql: true:** muestra las consultas SQL en la consola para que verifiques que la app habla con la BD.

(opcional) También, en el mismo directorio, crea (si no existe) un archivo [application.properties](#) con el siguiente contenido:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/hola_mundo
spring.datasource.username=hm_user
spring.datasource.password=hm_pass
spring.datasource.driver-class-name=org.postgresql.Driver
```

```
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true
```

Sin embargo, esto es opcional en algunos casos donde más adelante al ejecutar la Spring Boot aparece un error, y es importante tener en cuenta que si se crea y establece la configuración del archivo `application.yml`, el archivo `application.properties` debe quedar vacío o debe ser eliminado.

(Opcional) Si usas Docker, puedes exportar las variables con estos comandos en tu terminal:

```
export DB_HOST=localhost  
export DB_PORT=5432  
export DB_NAME=hola_mundo  
export DB_USER=hm_user  
export DB_PASS=hm_pass
```

(En Windows PowerShell usa `setx` o configura en las opciones de Docker)

Estos comandos definen las variables de entorno en tu sesión de shell. Spring Boot las leerá automáticamente cuando empiece.

10. Definir la entidad Mensaje

Para definir la entidad mensaje, tenemos que crear el archivo `Mensaje.java` en la siguiente ruta:

`src/main/java/com/ejemplo/holamundo/model/Mensaje.java`

Debes crear la carpeta `/model/` ya que aún no la tenemos.

El cual debe contener el siguiente código:

```
package com.ejemplo.holamundo.model;  
  
import jakarta.persistence.*;  
  
@Entity  
@Table(name = "mensaje")  
public class Mensaje {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @Column(nullable = false)  
    private String texto;
```

```
public Mensaje() {  
}  
  
public Mensaje(String texto) {  
    this.texto = texto;  
}  
  
public Long getId() {  
    return id;  
}  
  
public String getTexto() {  
    return texto;  
}  
  
public void setTexto(String texto) {  
    this.texto = texto;  
}  
}
```

Esta clase es una entidad JPA, lo que nos quiere decir que representará una tabla en una base de datos relacional, en este caso nuestra BD en PostgreSQL.

Una explicación simple de algunas de las partes del código:

[@Entity](#) indica que esta clase es una tabla de la BD.

[@Id](#) marca la columna llave primaria.

[@GeneratedValue\(strategy = GenerationType.IDENTITY\)](#) dice que la base de datos asigna el id automáticamente (comportamiento típico en Postgres para columnas autoincrementales).

[@Column\(nullable = false\)](#) obliga que el texto no sea nulo.

11. Definir el Repositorio, Servicio y Controlador

En esta sección definiremos las clases [MensajeRepository.java](#), [MensajeService.java](#) y [MensajeController.java](#). Estas clases definen tres de las capas principales que se utilizan en la arquitectura de un proyecto Spring Boot.

11.1 MensajeRepository

Para crear el repositorio, crearemos el archivo [MensajeRepository.java](#) en la ruta:

[src/main/java/com/ejemplo/holamundo/repository/MensajeRepository.java](#)

Debes crear la carpeta `/repository/`.

El archivo debe contener el siguiente código:

```
package com.ejemplo.holamundo.repository;

import com.ejemplo.holamundo.model.Mensaje;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface MensajeRepository extends JpaRepository<Mensaje,
Long> {
}
```

Puede que la clase parezca pequeña, pero esta define una interfaz de repositorio para la entidad `Mensaje`. Un repositorio es una parte de la capa de acceso de datos (*data access layer*) y su trabajo es manejar la lectura y escritura en tu base de datos.

Al extender la clase `JpaRepository`, heredamos un conjunto completo de operaciones CRUD, sin tener que escribir ninguna línea en SQL.

11.2 MensajeService

Crearemos la clase correspondiente al servicio de la entidad `Mensaje`, para esto debemos crear el archivo `MensajeService.java` en la ruta:

`src/main/java/com/ejemplo/holamundo/service/MensajeService.java`

Debemos crear la carpeta `/service/`.

El código que debe tener el archivo es el siguiente:

```
package com.ejemplo.holamundo.service;

import com.ejemplo.holamundo.model.Mensaje;
import com.ejemplo.holamundo.repository.MensajeRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class MensajeService {

    private final MensajeRepository repository;
```

```

public MensajeService(MensajeRepository repository) {
    this.repository = repository; // Constructor injection
}

public List<Mensaje> findAll() {
    return repository.findAll();
}

public Mensaje save(Mensaje m) {
    return repository.save(m);
}

public Mensaje findFirst() {
    return repository.findAll().stream().findFirst().orElse(null);
}
}

```

Esta clase es parte de la capa de servicio en la aplicación de Spring Boot, la capa intermedia entre el repositorio (Base de datos) y el controlador (Endpoints).

Sobre [@Autowired](#) : aquí usamos inyección por constructor en lugar de [@Autowired](#). Es la forma recomendada porque es más simple de testear y evita problemas con campos nulos.

11.3 MensajeController

Para la clase correspondiente el controlador, crearemos el archivo

[MensajeController.java](#) en la ruta:

[src/main/java/com/ejemplo/holamundo/controller/MensajeController.java](#)

De nuevo, necesitaremos crear la carpeta /controller/

El código debe ser el siguiente:

```

package com.ejemplo.holamundo.controller;
import com.ejemplo.holamundo.model.Mensaje;
import com.ejemplo.holamundo.service.MensajeService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/hola")
public class MensajeController {
    private final MensajeService service;

    public MensajeController(MensajeService service) {
        this.service = service;
    }
}

```



```

@GetMapping
public ResponseEntity<Mensaje> getHola() {
    Mensaje m = service.findFirst();
    if (m == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(m);
}

@PostMapping
public ResponseEntity<Mensaje> createHola(@RequestBody Mensaje
nuevo) {
    Mensaje saved = service.save(nuevo);
    return ResponseEntity.status(201).body(saved);
}
}

```

Esta clase define los Endpoints de nuestro REST API, la URL que los clientes pueden utilizar para interactuar con nuestra aplicación.

En esta clase definimos el [GET /hola](#), que devuelve el primer mensaje guardado (o 404 si no existe). [POST /hola](#) permite crear uno nuevo (opcional en este tutorial)

12. Insertar la instancia mínima (data.sql y CommandLineRunner)

Ahora ya tenemos la entidad y las capas de manejo de información, pero nuestra base de datos se encontrará vacía cuando iniciamos la aplicación, y estará así a menos de que manualmente le insertemos un mensaje. Una instancia mínima (o semilla inicial) nos sirve para insertar automáticamente un registro cuando la aplicación se inicia.

Tenemos dos opciones para crear el registro inicial [Hola Mundo](#).

1. Archivo data.sql

Podemos crear un archivo [data.sql](#), el cual es ejecutado por Spring Boot al iniciar la aplicación. Lo ubicaríamos en la ruta [src/main/resources/data.sql](#)

Este archivo tendría lo siguiente:

```
INSERT INTO mensaje (texto) VALUES ('Hola Mundo');
```

Asegúrate que el nombre de la tabla (mensaje) coincide con el [@Table](#)

(en la clase Java de la entidad) o con la convención generada.

2. CommandLineRunner

Como segunda opción, podemos crear una clase en Java que se ejecute cuando la aplicación inicie. Lo haríamos con un archivo `DataLoader.java` ubicado en la ruta `src/main/java/com/ejemplo/holamundo/config/DataLoader.java`. Como siempre, recuerda que tenemos que crear la carpeta `/config/`.

Esta clase tendría lo siguiente:

```
package com.ejemplo.holamundo.config;

import com.ejemplo.holamundo.model.Mensaje;
import com.ejemplo.holamundo.repository.MensajeRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DataLoader {

    @Bean
    CommandLineRunner initDatabase(MensajeRepository repo) {
        return args -> {
            if (repo.count() == 0) {
                repo.save(new Mensaje("Hola Mundo"));
            }
        };
    }
}
```

Esta forma es más útil si queremos expandir nuestra lógica al momento de insertar datos iniciales.

13. Algunos conceptos REST antes de probar

Rápidamente asegurémonos de reconocer algunos de los conceptos relacionados al API REST antes de seguir:

- **Endpoint:** URL que atiende peticiones (ej.: `/hola`).
- **GET:** pide información (no modifica). Usaremos para leer `Mensaje`.
- **POST:** crea un nuevo recurso (opcional en este tutorial).

- **JSON:** Es el formato de intercambio de datos que usualmente se utiliza en las API.
Ejemplo: ({"id":1,"texto":"Hola Mundo"}).
- **Códigos HTTP:** Estas son las respuestas a nuestras peticiones a los Endpoint.
Ejemplo: 200 OK (éxito), 201 Created (recurso creado), 404 Not Found, 500 Error interno.

14. Probar el endpoint

Ya tenemos lo necesario para probar nuestro Endpoint. Primero que todo, asegurémonos de que la app está corriendo. Para hacer esto verificaremos que nuestro proyecto aparezca en Maven y que la base de datos de PostgreSQL esté corriendo.

Para esto, ubicate en el directorio inicial (/holamundo/) y usa lo siguientes comandos:

(No olvides el sudo en Linux!)

```
docker ps
```

Deberías ver algo como:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS			NAMES	
1504939067d9	postgres:15	"docker-entrypoint.s..."	3 days ago	Up 8 minutes
0.0.0.0:5432->5432/tcp, [::]:5432->5432/tcp			holamundo-db-1	

y

```
mvn spring-boot:run
```

y en windows:

```
mvnw spring-boot:run
```

```

  ____
 /  __ \
/  /  \
/  /___\
/_____/

:: Spring Boot ::                (v3.5.6)

2025-10-24T20:12:05.408-05:00 INFO 33925 --- [           main] c.e.holamundo.Ho
lamundoApplication : Starting HolamundoApplication using Java 21.0.8 with
PID 33925 (/home/juan/Documents/holamundo/target/classes started by root in /hom
e/juan/Documents/holamundo)
2025-10-24T20:12:05.411-05:00 INFO 33925 --- [           main] c.e.holamundo.Ho
lamundoApplication : No active profile set, falling back to 1 default prof
ile: "default"
2025-10-24T20:12:06.045-05:00 INFO 33925 --- [           main] .s.d.r.c.Reposit
oryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT
mode.
2025-10-24T20:12:06.098-05:00 INFO 33925 --- [           main] .s.d.r.c.Reposit
oryConfigurationDelegate : Finished Spring Data repository scanning in 44 ms. Fo
und 1 JPA repository interface.
2025-10-24T20:12:06.549-05:00 INFO 33925 --- [           main] o.s.b.w.embedded

```

En otro terminal, probemos el **GET** curl:

```
curl -i http://localhost:8080/hola
```

HTTP/1.1 200 OK Content-Type: application/json {\"id\":1,\"texto\":\"Hola Mundo\"}
--

```
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sat, 25 Oct 2025 02:36:27 GMT

{"id":1,"texto":"Hola Mundo"}juan@g-old:~$
```

HTTP/1.1 404 Not Found

¡Felicidades! Si obtuviste una de estas respuestas significa que tus Endpoints sirven.

15. Crear la interfaz web: “index.html” y “main.js”

Hasta ahora, todo el trabajo que hemos hecho sucede "detrás de escena": el servidor, la base de datos y el código Java que los conecta. En este paso, crearemos una pequeña interfaz web para que el usuario pueda ver el mensaje "Hola Mundo" desde su navegador.

Esta interfaz será muy sencilla: una página con un título, un botón y un espacio donde se mostrará el texto que venga desde el servidor.

¿Dónde se guardarán los archivos visibles desde el navegador?

Recordemos que Spring Boot, además de manejar la lógica del servidor, también puede servir archivos estáticos como páginas HTML, imágenes o archivos JavaScript. Por convención, todos esos archivos deben guardarse dentro de la carpeta:

```
src/main/resources/static
```

Cuando la aplicación está en ejecución, Spring Boot busca automáticamente allí y permite acceder a esos archivos desde el navegador. Por ejemplo, si colocamos un archivo “index.html” en esa carpeta, podremos abrirlo visitando:

```
http://localhost:8080/index.html
```

La estructura de carpetas quedaría así:

```
src/main/resources/static/  
├── index.html  
└── js/  
    └── main.js
```

15.1 Crear la página principal “index.html”

1. En tu navegador, abre la carpeta: `src/main/resources/static`
2. Dentro de ella crea un nuevo archivo llamado “index.html”
3. Copia el siguiente código dentro del archivo:

```
<!DOCTYPE html>  
<html lang="es">
```

```
<head>
  <meta charset="utf-8" />
  <title>Hola Mundo - Spring Boot</title>
</head>
<body>
  <h1>Hola Mundo - Spring Boot + Postgres</h1>

  <button id="btn">Ver Hola</button>

  <div id="resultado"></div>

  <script src="/js/main.js"></script>
</body>
</html>
```

```

<!doctype html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Hola Mundo - Spring Boot</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      max-width: 600px;
      margin: 50px auto;
      padding: 20px;
    }
    button {
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
      background-color: #007bff;
      color: white;
      border: none;
      border-radius: 4px;
    }
    button:hover {
      background-color: #0056b3;
    }
    #resultado {
      margin-top: 20px;
      padding: 10px;
      border: 1px solid #ccc;
      background-color: #f9f9f9;
      min-height: 30px;
      border-radius: 4px;
    }
  </style>
</head>
<body>
  <h1>Hola Mundo - Spring Boot + PostgreSQL</h1>
  <button id="btn">Ver Hola</button>
  <div id="resultado"></div>
  <script src="/js/main.js"></script>
</body>
</html>

```

¿Qué hace este código?

- La etiqueta `<h1>` muestra un título principal.
- El botón con `id="btn"` es lo que se presionara para pedir al servidor el mensaje.

- El `<div>` con `id="resultado"` mostrará el texto que venga desde el backend.
- El `<style>` hace que la página se vea mejor (colores, espacios, etc.).
- Finalmente, `<script src="/js/main.js"></script>` carga el archivo JavaScript que crearemos a continuación.

15.2 Crear el archivo JavaScript llamado "main.js"

1. Dentro de la carpeta static, crea una carpeta llamada js, la ruta completa debe quedar así:

```
src/main/resources/static/js
```

2. Dentro de esa carpeta, crea un archivo llamado "main.js"
3. Copia el siguiente código en ese archivo:

```
document.getElementById('btn').addEventListener('click', () => {
  fetch('/hola')
    .then(res => {
      if (!res.ok) throw new Error('Respuesta no OK: ' + res.status);
      return res.json();
    })
    .then(data => {
      document.getElementById('resultado').innerText =
        `ID: ${data.id} - Texto: ${data.texto}`;
    })
    .catch(err => {
      document.getElementById('resultado').innerText =
        'Error al obtener mensaje: ' + err.message;
    });
});
```

¿Qué hace este código?

- `document.getElementById('btn')` Busca el botón que creamos en el HTML (el que tiene `id="btn"`).
- `.addEventListener('click', () => { ... })` Dice: "Cuando el usuario haga clic en este botón, ejecuta el código dentro de las llaves".
- `fetch('/hola')` Envía una SOLICITUD al servidor pidiendo el mensaje guardado en la BD. Es como llamar por teléfono y decir: "¿Cuál es el primer mensaje?"
- `.then(res => { ... })` Cuando el servidor RESPONDE, ejecuta este código. `res.ok` verifica que la respuesta fue correcta (sin errores). `return res.json()` convierte la respuesta a formato JSON que JavaScript pueda entender.
- `.then(data => { ... })` Ahora tienes los datos (`data.id` y `data.texto`). Los escribes en el `<div id="resultado">` para que el usuario los vea.
- `.catch(err => { ... })` Si algo falla (servidor caído, error, etc.), muestra un

mensaje de error.

Con estos dos archivos, la comunicación funciona de la siguiente manera:

1. El usuario abre `http://localhost:8080/index.html` en el navegador.
2. Ve la página con el botón "Ver Hola".
3. Presiona el botón.
4. JavaScript ejecuta `fetch('/hola')`.
5. El navegador envía solicitud al servidor (endpoint `/hola`).
6. El servidor busca en la BD: `"SELECT * FROM mensaje WHERE id = 1"`.
7. Obtiene: `{id: 1, texto: "Hola Mundo"}`.
8. Lo convierte a JSON y lo devuelve.
9. JavaScript recibe el JSON.
10. Extrae `data.texto = "Hola Mundo"`.
11. Lo escribe en la página: `"ID: 1 - Texto: Hola Mundo"`.
12. El usuario ve el mensaje en pantalla.

16. Ejecución final paso a paso

Llegó el momento de probar todo lo que construimos. Hasta este punto ya tenemos:

- La base de datos PostgreSQL configurada en Docker (desde el paso 8).
- La aplicación Spring Boot con todas sus capas.
- El archivo `data.sql` que inserta el mensaje "Hola Mundo".
- Y una pequeña interfaz web (`index.html`) que lo mostrará en pantalla.

En este paso uniremos todo para ejecutar la aplicación completa y comprobar que el flujo funciona de principio a fin: la base de datos guarda el mensaje, el servidor lo entrega y el navegador lo muestra.

Paso 1: Verificar que PostgreSQL esté en ejecución

Si seguiste el paso 8, tu base de datos ya debería estar levantada con Docker. Sin embargo, si reiniciaste el computador o cerraste Docker, puede que el contenedor se haya detenido.

Para verificarlo, abre una terminal y escribe:

```
docker ps
```

Si ves algo similar a esto:

CONTAINER ID	IMAGE	COMMAND NAMES	CREATED	STATUS	PORTS
1504939067d9	postgres:15	"docker-entrypoint.s..." holamundo-db-1	5 minutes ago	Up 5 minutes	0.0.0.0:5432->5432/tcp, [::]:5432->5432/tcp

Significa que el contenedor está funcionando correctamente, si no aparece, puedes iniciarlo nuevamente desde el mismo directorio donde tienes el archivo `docker-compose.yml` ejecutando:

```
docker compose up -d
```

Este comando arrancará el contenedor en segundo plano. Una vez esté en ejecución, la base de datos ya estará lista para que la aplicación pueda conectarse.

Paso 2: Ejecutar la aplicación Spring Boot

Ahora que la base de datos está lista, abre una NUEVA terminal y navega hasta la carpeta principal del proyecto (donde está el archivo pom.xml). Por ejemplo:

```
cd ruta/del/proyecto/holamundo
```

Allí escribiremos el siguiente comando para ejecutar la aplicación con Maven:

```
mvn spring-boot:run
```

Este comando se encargará de:

1. Compilar el proyecto (la primera vez tardará más, descargará dependencias).
2. Iniciar el servidor embebido de Spring Boot (Tomcat).
3. Conectarse automáticamente a la base de datos configurada en application.yml.
4. Crear las tablas necesarias y ejecutar el archivo data.sql para insertar el mensaje "Hola Mundo".

La primera ejecución puede tardar 1 o 2 minutos porque Maven va a descargar todas las dependencias. Las siguientes serán más rápidas.

Cuando todo esté listo verás en la consola algo como:

```
Started HolaMundoApplication in 3.2 seconds (JVM  
running for 5.1)  
Tomcat started on port(s): 8080 (http)
```

Paso 3: Probar el endpoint con curl (opcional pero recomendado)

Antes de ir al navegador, vamos a verificar que el servidor esté respondiendo.

Para ello, abre una tercera terminal y ejecuta el siguiente comando:

```
curl -i http://localhost:8080/hola
```

Deberías ver una respuesta similar a esta:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41

{"id":1,"texto":"Hola Mundo"}
```

Si ves esto significa que el servidor está funcionando correctamente, si no, verifica que el servidor sigue ejecutándose en la segunda terminal, si ves error “Connection refused” significa que el servidor no está corriendo.

Paso 4: Ver la interfaz en el navegador

Ahora abre tu navegador web (Chrome, Firefox, Edge o Safari) y escribe en la barra de direcciones:

`http://localhost:8080/index.html`

Deberías ver:

Hola Mundo - Spring Boot + PostgreSQL

Ver Hola

Paso 5: Presionar el botón

Ahora presiona el botón “Ver Hola”, deberías ver que aparece:

Hola Mundo - Spring Boot + PostgreSQL

Ver Hola

ID: 1 - Texto: Hola Mundo

Si ves esto, felicidades, tu aplicación funciona correctamente.

¿Qué acaba de pasar?

1. Presionaste el botón.
2. JavaScript envió: `fetch('/hola')`
3. El servidor recibió la solicitud en el endpoint `/hola`.
4. El controlador llamó al servicio.
5. El servicio llamó al repositorio.
6. El repositorio consultó la BD: `SELECT * FROM mensaje WHERE id = 1`.
7. La BD devolvió: `{id: 1, texto: "Hola Mundo"}`.
8. Todo volvió hacia arriba.
9. Se convirtió a JSON.
10. Se envió al navegador.
11. JavaScript lo recibió.
12. Lo mostró en la página.

Felicidades, acabas de crear tu primera aplicación web con `Java`, `Spring Boot` y `PostgreSQL` desde cero.

