UNIVERSIDAD AUTÓNOMA METROPOLITANA
Casa abierta al tiempo
Unidad Iztapalapa

Primer proyecto usando NetBeans, Maven y SpringBoot

Alumno:
Pedro Misael Rodríguez Jiménez

Profesor:
Alfonso Martínez Martínez

8 de julio de 2025

Índice

1. Introducción

El desarrollo de aplicaciones web modernas exige herramientas que simplifiquen la gestión de dependencias, la construcción del proyecto y la configuración de los servicios necesarios para exponer la lógica de negocio. En este contexto, el presente trabajo describe la creación de una **API REST mínima para la gestión de tareas** empleando, por primera vez, tres tecnologías clave que se complementan entre sí:

1. **Maven**: el sistema de construcción y gestor de dependencias que, mediante su archivo `pom.xml`, descarga de forma automática todas las bibliotecas requeridas (starters de Spring, Hibernate, H2, Lombok, etc.).
2. **NetBeans IDE**: la plataforma de desarrollo integrada que ofrece edición de código. NetBeans cuenta con soporte nativo para proyectos Maven, por lo que cada acción del IDE (*Run Project*, *Clean & Build*) se traduce en la ejecución de comandos Maven.
3. **Spring Boot**: el *framework* que aporta un conjunto de *starters* para configurar automáticamente un contenedor Tomcat embebido, serialización JSON mediante Jackson, persistencia relacional con Spring Data JPA e Hibernate, y validación de datos con Bean Validation. Gracias a su filosofía *convention over configuration*, basta añadir anotaciones como `@Entity` y `@RestController` para disponer de endpoints funcionales sin necesidad de configuración manual extensa.

Objetivo: Levantar una API REST básica para el manejo de tareas, capaz de exponer algunas operaciones CRUD sobre la entidad `Task`, almacenar los datos en una base H2 embebida y permitir su inspección mediante la consola web correspondiente.

Con este objetivo alcanzado, el proyecto sienta las bases para evolucionar hacia funcionalidades de nivel intermedio (paginación, validación global, seguridad HTTP Basic y documentación OpenAPI), manteniendo la coherencia entre las herramientas Maven, NetBeans y Spring Boot que sustentan todo el proceso de desarrollo.

2. Creación de proyecto con Spring Initializr

Spring ofrece un sitio web intuitivo para generar un proyecto; **Spring Initializr** te entrega una plantilla lista para compilar y ejecutar una aplicación Spring Boot. Genera la estructura de carpetas y el archivo de construcción (pom.xml para Maven en este caso), incluye dependencias declaradas y alineadas con la versión exacta de Spring Boot que eliges y

permite definir metadatos (grupo, artefacto, nombre, descripción, versión de Java, etc). Para crear nuestro primer proyecto, seguiremos los siguientes pasos:

1. Visitar Spring Initializr

2. Completar:

   - Project: `Maven`
   - Language: `Java`
   - Spring Boot: `3.5.3`
   - Group: `com.example`
   - Articaft: `taskmanager`
   - Java: `17`

3. Agregar dependencias:

   - **Spring WEB** Incluye Spring MVC, Jackson y un Tomcat embebido. Permite exponer controladores anotados con @RestController, recibir y devolver JSON. Es la pieza que convierte el código en una API REST accesible en http://localhost:8080 o cualquier otro puerto deseado.
   - **Spring Data JPA** Aporta la capa de persistencia: Spring Data genera los repositorios (JpaRepository) y delega en Hibernate para convertir entidades Java en tablas SQL, evitando lidiar directamente con SQL.
   - **H2 Database** es un motor de base de datos relacional, escrito íntegramente en Java, embebible y ligero. Funciona en memoria o sobre archivo, arranca en milisegundos y trae una consola web (/h2-console). puede comportarse como PostgreSQL, MySQL, etc. sin necesidad de instalarlos, ideal para prototipos y tests.
   - **Validation** Activa anotaciones como `@NotBlank, @Size`, en la entidades. Spring las ejecuta automáticamente cuando se recibe un `@Valid @RequestBody`, devolviendo HTTP 400 si los datos no cumplen las reglas.
   - **Lombok** Genera en compilación, getters, setters, constructores y métodos mediante anotaciones. Reduce drásticamente el código repetitivo que no aporta lógica de negocio nueva, pero que hay que escribir para que el programa funcione.

4. **Descargar** la plantilla dando click en **GENERATE**.

3. Manejo de proyecto con NetBeans

Una vez descargado el proyecto generado en Spring Initializr, descomprimimos la carpeta y en el IDE NetBeans abrimos directamente el proyecto, y en el directorio `src/main/resources/application` encontraremos el archivo properties, donde haremos una configuración básica agregando las siguientes líneas:

- **Bloque H2:** Consola WEB embebida

  1. Activar la pequeña aplicacion WEB que H2 trae incorporada para poder abrirla en el navegador e inspeccionar tablas, hacer consultas SQL y verificar que los datos realmente se estan guardando. Por defecto está desactivada (`false`) porque no conviene exponer una consola SQL, solamente para desarrollo y pruebas locales.

  ```
  spring.h2.console.enabled=true
  ```

  2. Indicar la ruta en la que se servirá la consola, al arrancar la app y visitar `http://localhost:8080/h2-console`, veremos el login de H2

  ```
  spring.h2.console.path=/h2-console
  ```

- **Bloque JPA:** Comportamiento de Hibernate

  1. Indicarle a Hibernate que hacer con el esquema de la base de datos cada vez que arranca la aplicación. `validate` Crea tablas que no existan y altera las que sí existen.

  ```
  spring.jpa.hibernate.ddl-auto=update
  ```

  2. Imprime cada sentencia SQL que Hibernate ejecuta en la consola.

  ```
  spring.jpa.show-sql=true
  ```

- **Servidor WEB:** Definición de puerto

  1. Definir el puerto en el que el Tomcat embebido escucha:

  ```
  server.port=8080
  ```

3.1. Backend (Java + Spring Boot)

Dentro del paquete principal `src/main/java/com/example/taskmanager/`, haciendo click derecho y seleccionando *new Package* sobre crearemos los siguientes subpaquetes:

- **model/** Paquete para entidades JPA
- **repository/** Paquete para interfaces JpaRepository
- **controller/** Paquete para controladores REST

Model

La responsabilidad de este paquete es albergar todas las clases que describen la lógica de negocio (entidades JPA, enums, etc.).

1. **Entidad *Task***
   Crearemos una nueva clase dando click derecho en el subpaquete Model y seleccionando *new Java Class*

```
package com.example.taskmanager.model;

import jakarta.persistence.*;             //Mapear a BD
import jakarta.validation.constraints.*; //Rechazar datos incorrectos
import lombok.*;                          //Eliminar codigo repetitivo

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Task {
```

Anotación	Propósito en el proyecto
`@Entity`	Marca la clase para que JPA/Hibernate la mapee a una tabla SQL.
`@Getter`, `@Setter`	Genera getters y setters públicos para todos los atributos.
`@NoArgsConstructor`	Genera un constructor público y sin cuerpo, es obligatorio para cualquier clase que JPA gestione.
`@AllArgsConstructor`	Genera un constructor con todos los campos como parámetros, en el mismo orden en que aparecen. Combinada con @Builder agiliza la freación de objetos.
`@Builder`	Facilita creación legible de objetos.

```
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

Anotación	Propósito en el proyecto
`@Id`	Marca la clave primaria.
`@GeneratedValue(IDENTITY)`	pide a la BD que autoincremente.

```
    @NotBlank
    private String title;
```

Anotación	Propósito en el proyecto
`@NotBlank`	Exige que title no sea null, ni vacío, ni sólo espacios.

```
    @Size(max = 1000)
    private String description;
```

Anotación	Propósito en el proyecto
`@Size(max = 1000)`	Restringe la longitud de la cadena de caracteres.

```
    @Enumerated(EnumType.STRING)
    @NotNull
    private Priority priority;

    @Enumerated(EnumType.STRING)
    @NotNull
    private Status status;
}
```

Anotación	Propósito en el proyecto
`@Enumerated(EnumType.STRING)`	Le dice a JPA que guarde el nombre del enum, no el ordinal.
`@NotNull`	Forzosamente se debe especificar un valor válido.

2. **Enums *Priority* y *Status***
   Crearemos 2 Enums en el subpaquete Model, dando click derecho sobre el y seleccionando *new Java Enum*. En la lógica del gestor de tareas sólo existirán tres prioridades y tres estados.

```
package com.example.taskmanager.model;

public enum Status { PENDING,  IN_PROGRESS, DONE }
```

```
package com.example.taskmanager.model;

public enum Priority { LOW, MEDIUM, HIGH }
```

Repository

Este subpaquete concentra las interfaces que acceden a la base de datos. `JpaRepository` es una interfaz de Spring Data que ya incluye toda la lógica CRUD. Como parametros necesita la *entidad* que va a manejar, y el *tipo* de la clave primaria.
Crearemos esta interfaz dando click derecho sobre el subpaquete *repository* y seleccionando *new Java Interface*

```
package com.example.taskmanager.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.taskmanager.model.Task;

public interface TaskRepository extends JpaRepository<Task, Long> { }
```

Controller

Aquí se concentran todas las clases que exponen la lógica de negocio a través de HTTP. Los controladores reciben peticiones, validan, delegan a los servicios/repositorios y construyen la respuesta. De esta manera, el controlador no conoce SQL ni lógica de mapeo; solo orquesta el flujo web.
Crearemos esta clase dando click derecho sobre el subpaquete `controller` seleccionando `new Java Class`.

```
package com.example.taskmanager.controller;

import com.example.taskmanager.model.Task;
import com.example.taskmanager.repository.TaskRepository;
import jakarta.validation.Valid;
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
import java.util.List;
```

Iniciamos importando las librerías necesarias para poder usar las siguientes anotaciones

```
@RestController
@RequestMapping("/api/tasks")
public class TaskController {
private final TaskRepository repo;

public TaskController(TaskRepository repo) {
    this.repo = repo;
}
```

Anotación	Propósito en el proyecto
`@RestController`	Cada método devuelve directamente el cuerpo HTTP (Jackson serializa a JSON).
`@RequestMapping(/api/tasks")`	Prefijo para todos los métodos: las rutas reales serán `/api/tasks/....`

Endpoint 1: Listar todas las tareas

```
@GetMapping
public List<Task> all() {
    return repo.findAll();
}
```

Anotación	Propósito en el proyecto
`@GetMappping`	Atiende `GET /api/tasks`.

- Regresa un `List<Task>`; Jackson la serializa automáticamente a formato JSON.

- `repo.findAll()` es un metodo heredado de Jpa Repository

Enpoint 2: Crear una tarea

```
@PostMapping
public ResponseEntity<Task> create(@Valid @RequestBody Task task) {
    Task saved = repo.save(task);
    return ResponseEntity.status(HttpStatus.CREATED).body(saved);
}
```

Anotación	Propósito en el proyecto
`@PostMappping`	Atiende `POST /api/tasks`.
`@RequestBody`	Des-serializa el JSON recibido a un Objeto `Taks`.
`@Valid`	Aplica Bean Validation: si `title` está vacío, Spring devuelve HTTP 400 antes de llegar al método.

- `repo.save(task)` Persiste la tarea; Hibernate asigna `id` y lo devuelve ya rellenado.
- `ResponseEntity.status(HttpStatus.CREATED)` Envía código `201 Created` con la tarea recién guardada en el cuerpo.

Teniendo ya el **backend** listo; donde el subpaquete **model** describe el dominio persistente, **repository** materializa el acceso a la base de datos mediante Spring Data JPA y **controller** expone la lógica a través de endpoints REST. El siguiente paso es demostrar como estas capacidades pueden ser consumidas desde el cliente: es decir, construir un **frontend** mínimo que interactúe con los endpoints para crear y listar tareas. A partir de aquí se explicará la carpeta `static/`, la pagina `index.html` y el pequeño bloque de JavaScript responsable de emitir peticiones `fetch`, para así, cerrar el flujo completo `navegador - API - Base de datos` que da vida al gestor de Tareas.

3.2. Frontend (HTML + CSS + JS

Todo lo que esté dentro del directorio `src/main/resources/static/` se envía sin pasar por ningun `@Controller`, haciéndolo ideal para:

- Archivos HTML sueltos
- Código JavaScript
- Hoja de CSS
- Imagenes, iconos, etc

HTML

Dibuja la interfaz mínima (formulario + tabla) y delega toda la lógica en el archivo `main.js`.

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8" />
  <link rel="stylesheet" href="css/styles.css">
  <title>Gestor de Tareas Spring Boot</title>
</head>
<body>
  <h1>Gestor de tareas</h1>
  <h3>Nueva tarea</h3>
  <form id="NuevaTarea">
    <input  id="titulo" type="text" placeholder="Titulo" required />
    <select id="priority">
      <option value="LOW">LOW</option>
      <option value="MEDIUM" selected>MEDIUM</option>
      <option value="HIGH">HIGH</option>
    </select>
    <button>Agregar</button>
  </form>

  <h2>Lista de tareas</h2>
  <table id="tabla">
    <thead>
      <tr><th>ID</th><th>Titulo</th><th>Prioridad</th><th>Estatus</th></tr
         >
    </thead>
    <tbody></tbody>
  </table>

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

CSS

La hoja CSS sólo da un aspecto limpio y legible: tipografía sencilla, fondo claro, tabla con bordes finos y botones oscuros con efecto *hover*.

```
/* Estilos basicos para nuestra pagina */
body {
    font-family: sans-serif;
    background-color: #f0f0f0;
    text-align: center;
    margin: 50px;
}
table {
    border-collapse: collapse;
    width: 100%;
}

th, td {
    border: 1px solid #ccc;
    padding: .4rem;
    text-align: left;
}
th {
    background: #f3f3f3;
}
form {
    margin-bottom: 1rem;
}
input, select {
    margin-right: .5rem;
}
h1 {
    color: #333;
}
p {
    font-size: 1.2em;
    margin: 20px 0;
}
button,select{
    padding: 10px 20px;
    font-size: 1em;
    background-color: #2c2c2c;
    color: white;
    border: none;
```

```
    border-radius: 5px;
    cursor: pointer;
}
button:hover{
    background-color: #1a1a1a;
}
```

JavaScript

El script usa fetch para hablar con tu API Spring Boot; Ademas de operaciones DOM básicas para mostrar o añadir filas en la tabla.

```
const api = '/api/tasks';

// Insertar una fila en la tabla
function pintaFila(t) {
    const fila = document.createElement('tr');
    fila.innerHTML = `
        <td>${t.id}</td>
        <td>${t.title}</td>
        <td>${t.priority}</td>
        <td>${t.status}</td>`;
    document.querySelector('#tabla tbody').appendChild(fila);
}

// Cargar lista al entrar o recargar la pagina
async function cargaTareas() {
    const res  = await fetch(api);
    console.log(res);
    const data = await res.json();
    console.log(data);
    data.forEach(pintaFila);
}

// Manejar el submit del formulario
document.getElementById('NuevaTarea').addEventListener('submit', async (e)
    => {
    e.preventDefault();
    const titulo   = document.getElementById('titulo').value;
    const prioridad= document.getElementById('priority').value;

    const res = await fetch(api, {
```

```
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ title: titulo, priority: prioridad })
    });

    if (res.ok) {
        const nueva = await res.json();
        pintaFila(nueva);
        e.target.reset();
    } else {
        alert('Error al crear tarea');
    }
 });

// Arranque inicial
cargaTareas();
```

`fetch` es la API de JavaScript moderno para peticiones HTTP:

- **GET:** `fetch('api/tasks')` devuelve una *Promise* con la respuesta que despues se deserealiza a onjeto JavaScript con `.json()`
- **POST**: Se especifican `method, headers, body` (en formato JSON).

4. Conclusión

Se levantó un **backend** funcional y un mini **frontend** que lo consume. Con Maven y NetBeans descargamos y compilamos los starters de Spring Boot; definimos la entidad Task, su repositorio JPA y un controlador REST con 2 operaciones CRUD. H2 proporcionó la base embebida y Bean Validation garantizó la entrada de datos correcta, mientras Lombok eliminó el código repetitivo. Del lado cliente, un HTML muy sencillo en la carpeta static/, diseño con estilos básicos y un script fetch bastaron para mostrar y crear tareas.
El objetivo de la fase quedó cubierto: la aplicación compila, arranca sin errores, persiste datos y ofrece una interfaz mínima para probar el flujo completo `navegador - API - Base de datos`. A partir de aquí la base es sólida para añadir mas operaciones CRUD, validación global o seguridad en próximas etapas.