Informe de Proyecto Unificado: Desarrollo y Evolución de una API REST para Gestión de Tareas

Materia: Desarrollo de Aplicaciones Web

Estudiante: José Daniel Avendaño Morales

Docente: Diego Fernando Zárate Pineda

Programa: Ingeniería de Sistemas

Fecha de Consolidación: 22 de enero de 2026

Índice de Contenidos

Introducción General del Proyecto

Objetivo y Alcance

El presente informe consolida el desarrollo de un proyecto de software realizado en dos fases, cuyo objetivo principal fue la creación y evolución de una interfaz de programación de aplicaciones (API) bajo el estilo arquitectónico REST. Esta API fue diseñada para la administración integral de tareas pendientes, permitiendo a los usuarios interactuar con sus datos de manera programática y estandarizada [1].

La primera fase del proyecto se centró en la construcción del núcleo funcional de la API, implementando las operaciones básicas de Crear, Leer, Actualizar y Eliminar (CRUD) [1]. La segunda fase se enfocó en extender las capacidades de la API, mejorando la experiencia del usuario mediante la adición de funcionalidades avanzadas como búsquedas dinámicas, una gestión de estados más flexible y la implementación de validaciones robustas del lado del servidor para garantizar la integridad de los datos [2].

Historia de Usuario

El desarrollo del proyecto se guio por la siguiente historia de usuario, que encapsula la necesidad fundamental del cliente final [1]:

> *"Como usuario de un sistema de gestión de tareas, quiero poder gestionar mis tareas a través de una API RESTful para poder realizar operaciones como crear, ver, editar y eliminar mis tareas".*

Arquitectura y Pila Tecnológica

La base tecnológica del proyecto se definió desde la primera fase para asegurar un desarrollo moderno, escalable y mantenible. La selección de tecnologías se mantuvo consistente a lo largo de ambas actividades. A continuación, se detalla la pila tecnológica empleada [1]:

- **Framework:** ASP.NET Core (Web API), elegido por su alto rendimiento, su naturaleza multiplataforma y su robusto soporte para la creación de servicios RESTful.
- **Lenguaje de Programación:** C# 12.0, aprovechando las últimas características del lenguaje para un código más limpio y seguro.
- **Servidor Web:** Kestrel, el servidor web multiplataforma y de alto rendimiento incluido por defecto en ASP.NET Core. Para el entorno de desarrollo, la API se ejecutó en el puerto local 5063.
- **Estrategia de Persistencia:** Almacenamiento volátil en memoria. Se utilizó una colección estática List<TaskModel> para simular una base de datos, una decisión pragmática para las fases iniciales del proyecto que priorizaba la rapidez en el desarrollo sobre la persistencia de datos a largo plazo.
- **Cliente de Pruebas:** Postman (v10+), herramienta fundamental para la validación manual de cada uno de los endpoints desarrollados, permitiendo la construcción y envío de peticiones HTTP complejas.

Fase 1: Implementación Inicial de la API (Actividad 1)

La primera fase del proyecto se concentró en establecer la funcionalidad básica de la API, implementando un conjunto completo de operaciones CRUD que sirvieron como cimiento para futuras mejoras [1].

Modelo de Datos (TaskModel.cs)

Se definió una clase TaskModel para representar la entidad principal del sistema: la tarea. Este modelo de datos se diseñó para ser robusto, incluyendo la inicialización de propiedades de tipo cadena para evitar advertencias de nulabilidad y el uso de *Data Annotations* para validaciones, como hacer el título obligatorio [1][2]. Las propiedades del modelo son:

- Id **(int):** Identificador único y numérico de la tarea, gestionado de forma auto-incremental por la lógica del controlador.
- Title **(string):** Nombre o título de la tarea. Es un campo obligatorio.
- Description **(string):** Detalle o descripción extendida de la actividad a realizar.
- IsCompleted **(bool):** Estado lógico que indica si la tarea ha sido completada (true) o está pendiente (false).

```
using System.ComponentModel.DataAnnotations; // Librería para validar datos

namespace GestionTareasApi.Models
{
  public class TaskModel
  {
    // El ID será como la cédula de la tarea, único e irrepetible
    public int Id { get; set; }

    // Con [Required] hacemos que el sistema no deje crear tareas sin nombre
    [Required(ErrorMessage = "¡Oye! No puedes dejar el título vacío.")]
    public string Title { get; set; } = string.Empty;

    // Una descripción breve de lo que hay que hacer
    public string Description { get; set; } = string.Empty;

    // Por defecto, toda tarea nueva empieza como "no completada" (false)
    public bool IsCompleted { get; set; } = false;
  }
}
```

Controlador y Operaciones CRUD (TareasController.cs)

El corazón de la API reside en el TareasController, que utiliza enrutamiento basado en atributos ([Route("api/tareas")]) para exponer los endpoints. Este controlador es el "cerebro" que recibe las peticiones y decide qué acción realizar. A continuación se presenta el código completo que integra las funcionalidades de ambas fases del proyecto [1][2]:

```
using Microsoft.AspNetCore.Mvc;
using GestionTareasApi.Models;

namespace GestionTareasApi.Controllers
{
  [ApiController]
  [Route("api/tareas")] // La dirección base será http://localhost:5063/api/tareas
  public class TareasController : ControllerBase
```

```
{
  // Creamos una lista en memoria para guardar las tareas mientras el programa corra
  private static List<TaskModel> _tareas = new List<TaskModel>();
  private static int _nextId = 1; // Contador para asignar IDs automáticamente

  // --- MÉTODOS PARA RECLAMAR DATOS (GET) ---

  [HttpGet] // Trae toda la lista
  public IActionResult Listar()
  {
    return Ok(new {
      mensaje = "Datos reclamados con éxito",
      datos = _tareas
    });
  }

  [HttpGet("buscar")] // Busca tareas por una palabra clave
  public IActionResult Buscar([FromQuery] string termino)
  {
    // Buscamos en la lista si el título contiene lo que escribió el usuario
    var resultados = _tareas.Where(t =>
      t.Title.Contains(termino ?? "", StringComparison.OrdinalIgnoreCase)).ToList();

    return Ok(new {
      mensaje = $"Búsqueda terminada. Encontré {resultados.Count} coincidencias.",
      datos = resultados
    });
  }

  // --- MÉTODO PARA ENVIAR DATOS (POST) ---

  [HttpPost]
  public IActionResult Crear([FromBody] TaskModel nuevaTarea)
  {
    // Le asignamos un número de ID y lo sumamos a la lista
    nuevaTarea.Id = _nextId++;
    _tareas.Add(nuevaTarea);

    // Respondemos que todo salió bien y mostramos qué se guardó
    return CreatedAtAction(nameof(Listar), new { id = nuevaTarea.Id }, new {
      mensaje = "Dato enviado con éxito",
      datoEnviado = nuevaTarea
```

```
    });
}

// --- MÉTODOS PARA ACTUALIZAR (PUT Y PATCH) ---

[HttpPut("{id}")] // Cambia toda la información de una tarea
public IActionResult Editar(int id, [FromBody] TaskModel actualizada)
{
    var tarea = _tareas.FirstOrDefault(t => t.Id == id);
    if (tarea == null) return NotFound(new { mensaje = "Esa tarea no existe, no la puedo editar" });

    tarea.Title = actualizada.Title;
    tarea.Description = actualizada.Description;

    return Ok(new { mensaje = "Dato actualizado con éxito", dato = tarea });
}

[HttpPatch("{id}/completar")] // Solo cambia el estado a completado (true)
public IActionResult Completar(int id)
{
    var tarea = _tareas.FirstOrDefault(t => t.Id == id);
    if (tarea == null) return NotFound(new { mensaje = "ID no encontrado" });

    tarea.IsCompleted = true;
    return Ok(new { mensaje = "¡Tarea terminada!", dato = tarea });
}

[HttpPatch("{id}/pendiente")] // Vuelve la tarea a pendiente (false)
public IActionResult MarcarPendiente(int id)
{
    var tarea = _tareas.FirstOrDefault(t => t.Id == id);
    if (tarea == null) return NotFound(new { mensaje = "ID no encontrado" });

    tarea.IsCompleted = false;
    return Ok(new { mensaje = "La tarea ahora está pendiente de nuevo", dato = tarea });
}

// --- MÉTODO PARA BORRAR (DELETE) ---

[HttpDelete("{id}")]
public IActionResult Eliminar(int id)
{
```

```
        var tarea = _tareas.FirstOrDefault(t => t.Id == id);
        if (tarea == null) return NotFound(new { mensaje = "No encontré qué borrar" });

        _tareas.Remove(tarea);
        return Ok(new { mensaje = "Dato eliminado con éxito", idEliminado = id });
    }
  }
}
```

Desafíos Técnicos y Soluciones (Fase 1)

Durante la puesta en marcha inicial, se enfrentaron y resolvieron varios obstáculos técnicos clave que son comunes en el desarrollo con ASP.NET Core [1].

3.3.1. Corrección de Error 404 (Not Found)

Se detectó que el servidor devolvía errores 404 al intentar acceder a los endpoints del controlador. La causa raíz fue una configuración por defecto de "Minimal API" en el archivo Program.cs, que no registra automáticamente los controladores basados en clases. La solución consistió en añadir explícitamente los servicios de controladores y mapear las rutas correspondientes mediante las siguientes líneas:

```
builder.Services.AddControllers();
app.MapControllers();
```

3.3.2. Corrección de Error 405 (Method Not Allowed)

Al probar el endpoint PUT, se recibía un error 405, indicando que el método no estaba permitido para la URL solicitada. El problema se originó por una discrepancia entre el parámetro de ruta definido en el decorador [HttpPut("{id}")] y el nombre de la variable en la firma del método. Se solucionó asegurando que ambos coincidieran, permitiendo así que el framework de enrutamiento mapeara correctamente la petición PUT a la acción del controlador.

3.3.3. Estrategia de Persistencia en Memoria

Un desafío fundamental fue que la lista de tareas se reiniciaba con cada petición HTTP. Esto se debía al ciclo de vida transitorio de los controladores en ASP.NET Core, que se instancian y destruyen por cada solicitud. Para lograr una persistencia de datos durante la vida útil del proceso del servidor, se aplicó el modificador static a la declaración de la lista List<TaskModel>. Esto la convierte en una variable de clase, compartida por todas las instancias del controlador y persistente mientras la aplicación esté en ejecución [1].

Fase 2: Evolución y Mejoras de la API (Actividad 2)

La segunda fase del proyecto se centró en enriquecer la API con funcionalidades que mejoran la usabilidad y la robustez, partiendo de la base sólida establecida en la Actividad 1 [2].

Endpoint de Búsqueda Dinámica

Para mejorar la localización de recursos, se añadió un endpoint especializado para búsquedas [2]:

- **Endpoint:** GET /api/tareas/buscar
- **Funcionamiento:** Permite al cliente enviar una palabra clave a través de un parámetro de consulta (query parameter), por ejemplo: /api/tareas/buscar?termino=clase.
- **Lógica de Negocio:** El controlador filtra la lista completa de tareas y devuelve únicamente aquellas que contienen la palabra clave proporcionada en su campo Title o Description.
- **Criterio de Calidad:** Para una mayor precisión y una mejor experiencia de usuario, la búsqueda se implementó de forma insensible a mayúsculas y minúsculas (*case-insensitive*).

Gestión de Estado Avanzada: Reversión a Pendiente

Se identificó la necesidad de poder revertir el estado de una tarea. Para ello, se implementó una nueva operación parcial [2]:

- **Endpoint:** PATCH /api/tareas/{id}/pendiente
- **Funcionamiento:** Este endpoint permite que una tarea previamente marcada como "completada" pueda regresar a su estado original de "pendiente".
- **Lógica de Negocio:** Al recibir una petición a esta ruta, el controlador busca la tarea por el id especificado y establece su propiedad IsCompleted en false.

Robustez y Validación de Datos Obligatorios

Para mejorar la integridad de los datos y la robustez general de la API, se introdujeron validaciones del lado del servidor [2].

- **Acción:** Se estableció que el campo Title de una tarea es estrictamente obligatorio.
- **Implementación:** Se utilizaron *Data Annotations* directamente en el modelo TaskModel.cs, decorando la propiedad Title con el atributo [Required]. Gracias a la integración nativa de ASP.NET Core, el framework ahora rechaza automáticamente cualquier petición POST (creación) o PUT (edición) que no incluya un valor para el título, respondiendo con un código de estado 400 Bad Request y un detalle de los errores de validación.

Resolución de Incidencias (Fase 2)

Durante la compilación del proyecto con el comando dotnet run tras implementar las nuevas funcionalidades, se encontró un error crítico que impedía el arranque de la aplicación [2].

- **Problema:** El compilador de C# reportaba errores de sintaxis debido a la presencia de etiquetas de texto extrañas, como cite_start y cite_end, dentro del código fuente.
- **Causa:** El error se debió a un copiado y pegado de código que accidentalmente incluyó metadatos o marcadores de formato no pertenecientes al lenguaje C#.
- **Solución:** Se realizó una limpieza manual y exhaustiva de los archivos afectados (TaskModel.cs y TareasController.cs), eliminando todas las etiquetas ajenas al código y restaurando la sintaxis correcta de los atributos y la lógica del programa. Tras esta corrección, el proyecto compiló y se ejecutó exitosamente.

Conclusiones y Resumen de Mejoras

La finalización de las dos fases del proyecto permite extraer varias conclusiones significativas sobre el proceso de desarrollo y el producto final:

1. Desarrollo Iterativo Exitoso: El enfoque de dividir el proyecto en dos actividades (implementación inicial y evolución) demostró ser altamente efectivo. Permitió construir una base funcional sólida (Actividad 1) y luego enriquecerla con características de valor añadido (Actividad 2) de una manera organizada y manejable.
2. Dominio de ASP.NET Core: El proyecto sirvió como una aplicación práctica profunda del framework ASP.NET Core para la creación de APIs RESTful. Se abordaron aspectos clave como el enrutamiento, el ciclo de vida de los controladores, la configuración de servicios, el model binding y la validación automática.
3. Importancia de la Depuración y Resolución de Problemas: Los desafíos técnicos encontrados, como los errores HTTP 404/405 o los problemas de sintaxis, fueron cruciales para el aprendizaje. Su resolución reforzó la comprensión del funcionamiento interno del framework y la importancia de un diagnóstico metódico.
4. Robustez a través de la Validación: La introducción de validaciones del lado del servidor en la Fase 2 marcó un paso importante hacia la madurez de la API. Asegurar que los datos entrantes cumplan con las reglas de negocio (como un título obligatorio) es fundamental para prevenir datos corruptos y mejorar la fiabilidad del servicio.
5. Limitaciones y Futuras Mejoras: La estrategia de persistencia en memoria, si bien fue adecuada para el alcance de este proyecto académico, representa la principal limitación del sistema actual, ya que todos los datos se pierden al detener el servidor. Un paso lógico a futuro sería reemplazar la lista estática por una solución de persistencia real, como una base de datos SQL (usando Entity Framework Core) o una base de datos NoSQL, para garantizar la durabilidad de los datos.

Resumen de Cambios y Mejoras

El proyecto consolidado integra mejoras clave que elevan la calidad y funcionalidad de la API:

- **Unificación de Actividades:** Se integró el CRUD básico de la Actividad 1 con la lógica de búsqueda y reversión de estados de la Actividad 2 en un solo controlador funcional y cohesivo.
- **Validación de Datos:** Se implementó el uso de DataAnnotations para que el sistema rechace automáticamente cualquier tarea que no tenga título, enviando un mensaje claro al usuario y garantizando la integridad de los datos.
- **Retroalimentación del Servidor:** A diferencia de la versión inicial, ahora todos los métodos devuelven un objeto JSON con un campo `mensaje`. Esto se hizo para que el usuario sepa exactamente qué pasó con su petición (si se envió, si se reclamó o si hubo un error), mejorando significativamente la experiencia de depuración y consumo de la API.
- **Correcciones Técnicas:** Se eliminaron errores de sintaxis provocados por caracteres extraños en el editor y se ajustó el enrutamiento para que sea consistente en Postman, asegurando un funcionamiento estable y predecible.

Referencias

- [1] Avendaño Morales, J. D. (2026). *Informe de Implementación: API REST de Gestión de Tareas (Actividad 1)*. Universidad [Nombre de la Universidad, si se conociera].
- [2] Avendaño Morales, J. D. (2026). *Informe de Proyecto: Actividad 2 - Evolución de la API REST*. Universidad [Nombre de la Universidad, si se conociera].

Anexos: Guía de Validación y Pruebas con Postman

Esta sección detalla los endpoints disponibles y los resultados de las pruebas realizadas con Postman para validar el correcto funcionamiento de la API en sus dos fases.

Anexo A: Resumen de Endpoints y Pruebas

La siguiente tabla consolida todos los endpoints desarrollados a lo largo del proyecto, su método HTTP, propósito y el resultado esperado en una prueba exitosa [1].

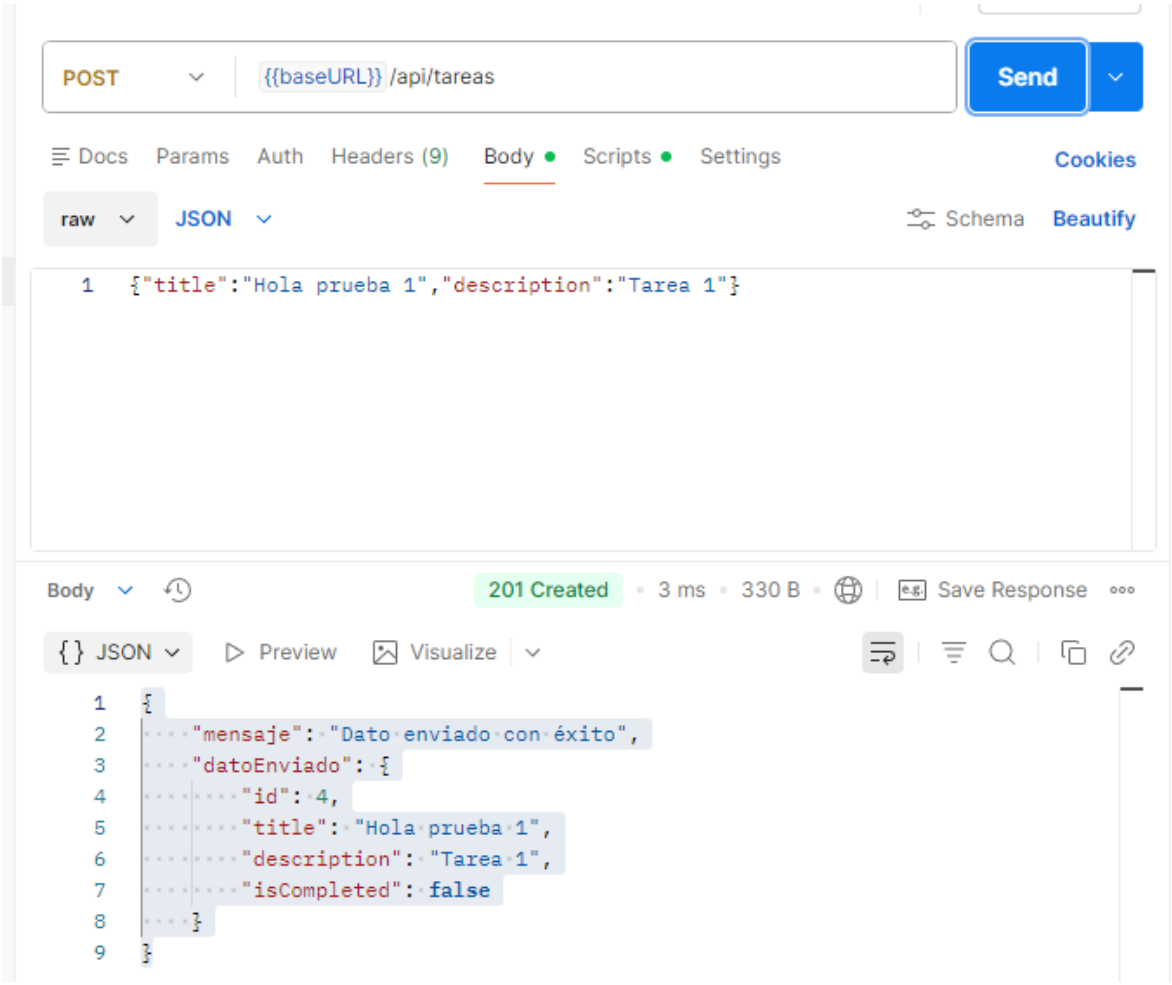
Método	Endpoint	Headers	Body (Raw JSON)	Resultado
POST	http://localhost:5063/api/tareas	Content-Type: application/json	{"title": "Tarea 1", "description": "Prueba"}	201 Created
GET	http://localhost:5063/api/tareas	N/A	N/A	200 OK
PUT	http://localhost:5063/api/tareas/1	Content-Type: application/json	{"title": "Editada", "description": "Nueva"}	200 OK
PATCH	http://localhost:5063/api/tareas/1/completar	N/A	N/A	200 OK
DELETE	http://localhost:5063/api/tareas/1	N/A	N/A	204 No Content
GET	/api/tareas/buscar	N/A	N/A (Query: ?termino=...)	200 OK
PATCH	/api/tareas/{id}/pendiente	N/A	N/A	200 OK

Anexo B: Evidencias de Pruebas (Actividad 1)

A continuación se presentan las evidencias visuales de las pruebas realizadas para los endpoints de la primera fase [1].

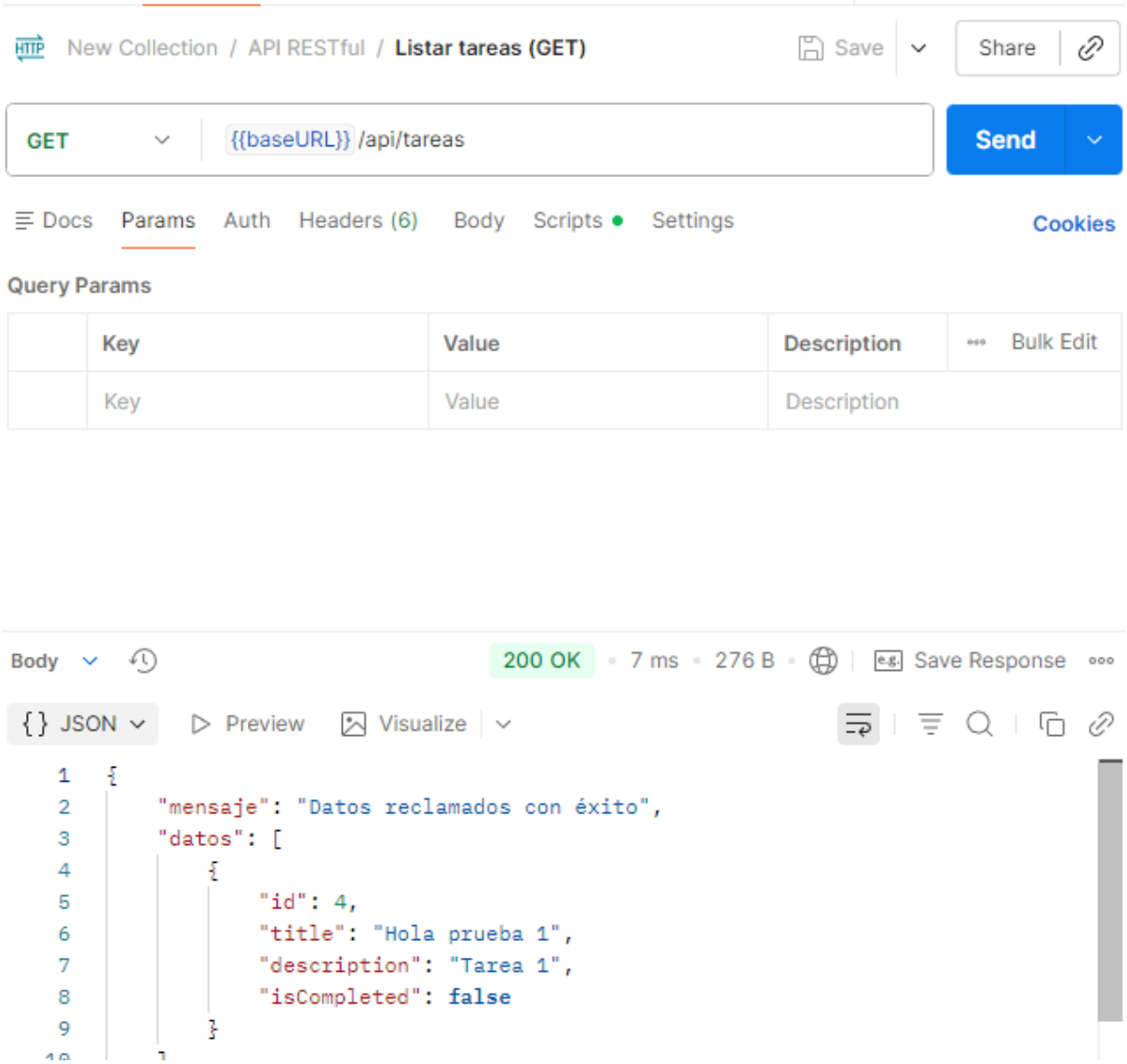
Prueba POST

Evidencia de la creación de una nueva tarea mediante una petición POST. Se observa el cuerpo de la petición en formato JSON y la respuesta del servidor con el código 201 Created, devolviendo la tarea recién creada con su ID asignado.



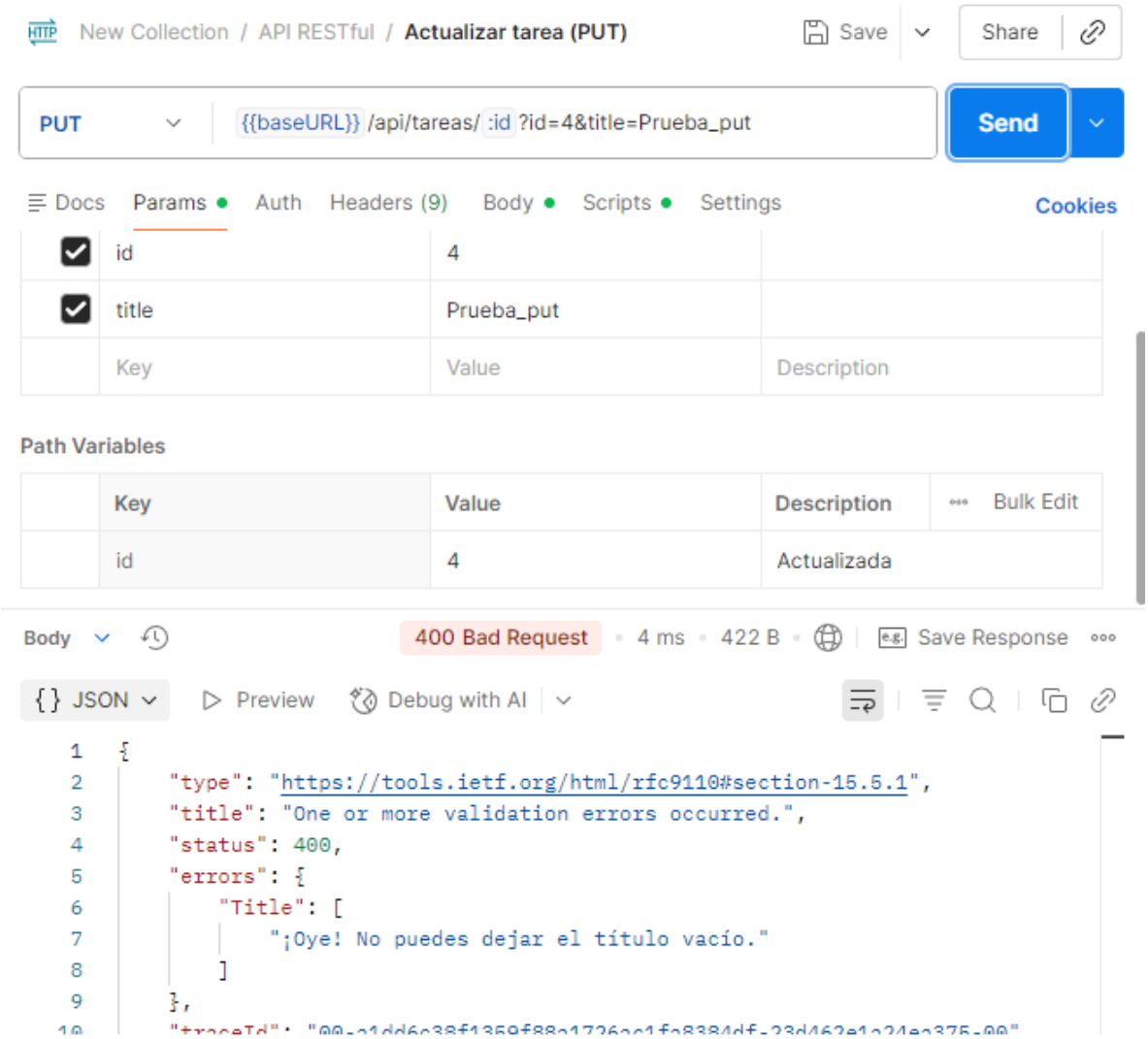
Prueba GET

Resultado de la obtención de todas las tareas almacenadas en memoria.



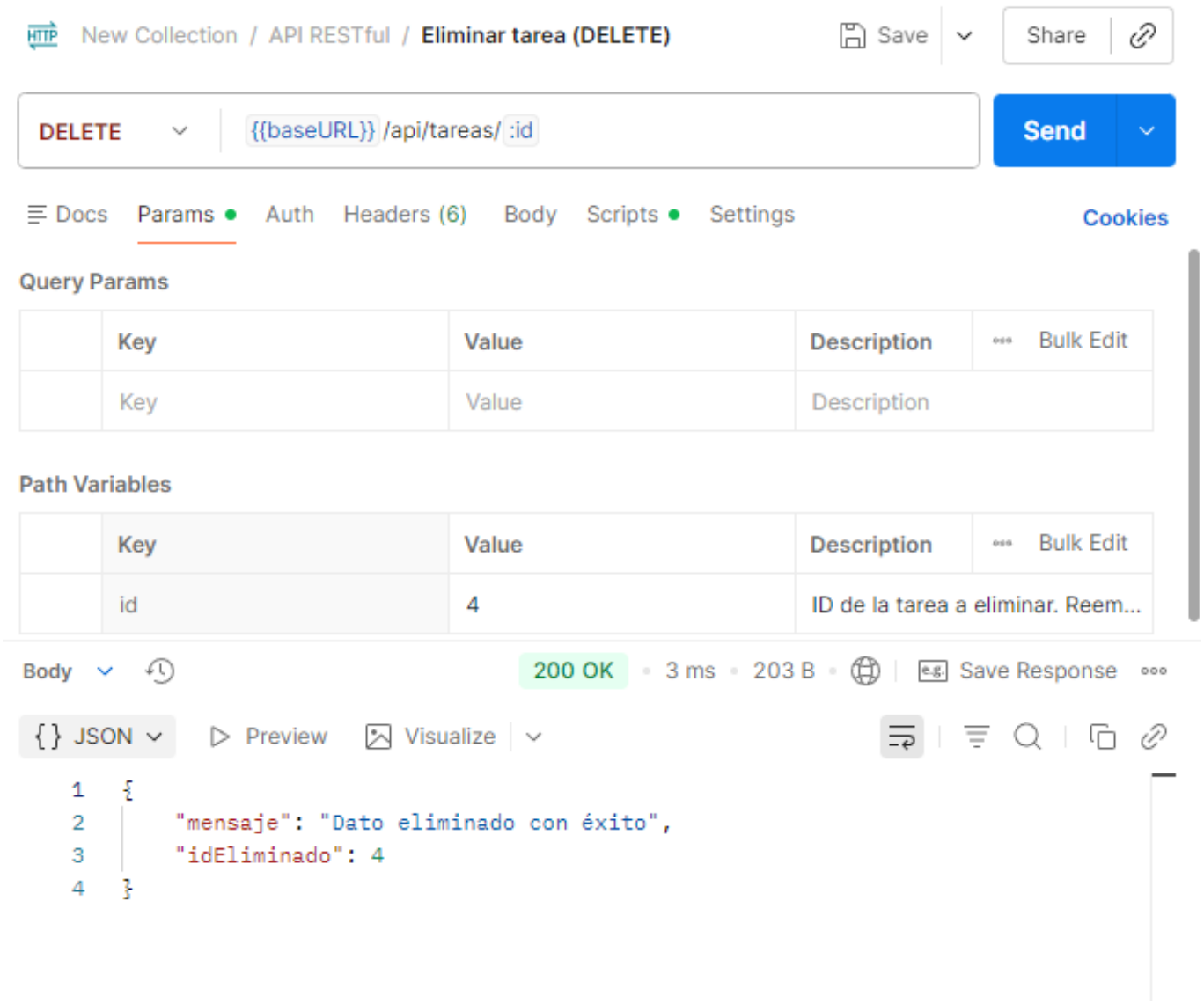
Prueba PUT

Captura de la actualización completa de una tarea existente. Se envía un cuerpo JSON con los nuevos datos para el título y la descripción.

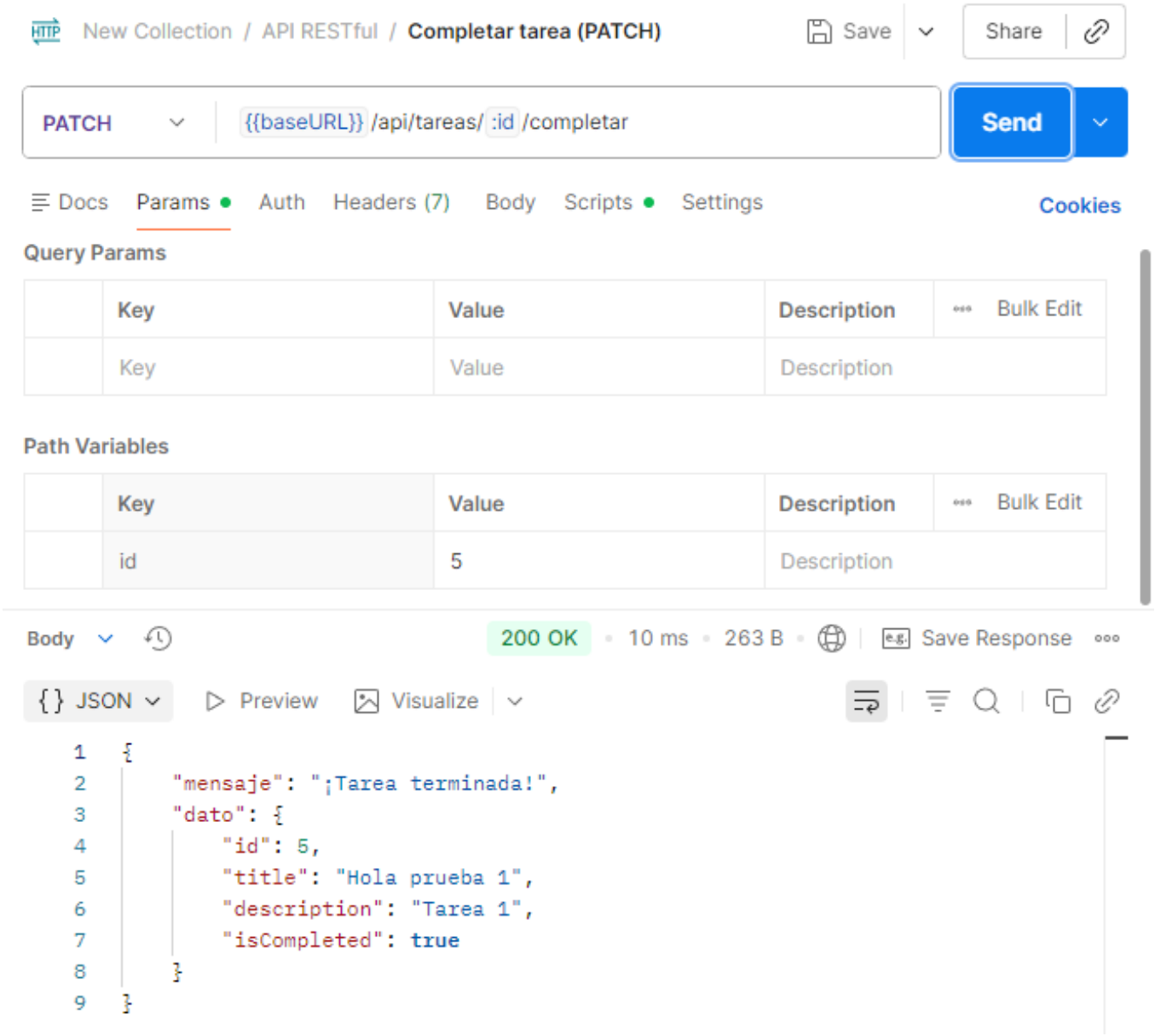


Prueba DELETE

Eliminación de una tarea específica a través de su ID.



Prueba PATCH
Actualización parcial para marcar una tarea como completada.



Anexo C: Evidencias de Pruebas (Actividad 2)
A continuación se describen los resultados de las pruebas para las nuevas funcionalidades implementadas en la segunda fase [2].

Prueba de Validación de Integridad
Intento de crear una tarea con un título vacío para verificar la validación.

- **Petición:** POST /api/tareas
- **Cuerpo de la Petición (Body):** {"title": "", "description": "Intento fallido"}
- **Resultado:** 400 Bad Request. La API rechaza la petición y devuelve un error de validación, confirmando que el campo Title es obligatorio.

Prueba de Búsqueda Funcional
Búsqueda de tareas que contengan el término "Tarea".

- **Petición:** GET /api/tareas/buscar?termino=clase
- **Resultado:** 200 OK. El cuerpo de la respuesta contiene un array con todas las tareas cuyo título o descripción incluye la palabra "clase", independientemente de si está en mayúsculas o minúsculas.

Prueba de Gestión de Estado (Reversión)
Cambio del estado de una tarea completada de vuelta a pendiente.

- **Petición:** PATCH /api/tareas/1/pendiente
- **Resultado:** 200 OK. Una posterior petición GET a /api/tareas/1 verifica que el campo isCompleted de la tarea con ID 1 ha cambiado su valor a false.