TEC | Tecnológico de Costa Rica

# Implementación de Servidor HTTP

Integrantes
Diego Alonso Mora Montes 2022104866
Daniel Cornejo Granados 2019018538
José Eduardo Gutiérrez Conejo 2019073558

Instituto Tecnológico de Costa Rica

Escuela de Computación

Principios de Sistemas Operativos

Semestre II

4 de Octubre de 2024

Prof. Kenneth Obando Rodríguez

**Diseño del servidor..... 3**
1. Servidor HTTP..... 3
2. ThreadPool..... 3
3. Operaciones HTTP y Manipulación del Archivo JSON..... 3
4. Manejo de Archivos y Serialización..... 4
**Implementación de la concurrencia..... 4**
1. Creación del ThreadPool..... 4
2. Comunicación entre Hilos..... 5
Sender y Receiver:..... 5
Código relevante:..... 5
3. Ejecución de Tareas Concurrentes..... 5
4. Hilos Trabajadores (Workers)..... 6
5. Gestión de Conexiones..... 7
6. Sincronización y Seguridad..... 7
**Manejo de cookies..... 8**
**Instrucciones para ejecutar y probar el servidor..... 9**
**Estructura y directorios..... 11**
**Resultados de pruebas..... 11**
**Manual de usuario..... 13**
Prerrequisitos..... 13
Dependencias..... 13
Instalación..... 13
Uso del Servidor HTTP..... 14
Pruebas..... 15

# Diseño del servidor

Este proyecto está basado en un **servidor HTTP concurrente** escrito en Rust, utilizando un enfoque modular y eficiente para gestionar múltiples peticiones de manera simultánea. A continuación se describen los principales componentes de la arquitectura:

## 1. Servidor HTTP

El servidor HTTP es el núcleo del proyecto. Escucha las conexiones entrantes en un puerto definido y las distribuye a diferentes hilos para su procesamiento. Cada conexión es tratada como una tarea independiente que será manejada por uno de los hilos en el `ThreadPool`.
Este servidor maneja peticiones como `GET`, `POST`, `PUT`, `PATCH` y `DELETE`, que se corresponden con operaciones sobre un archivo JSON.

## 2. ThreadPool

El servidor implementa un modelo de **ThreadPool** para gestionar la concurrencia de las conexiones. Este enfoque permite que el servidor maneje múltiples peticiones simultáneamente sin bloquear el proceso principal.

- **Creación del ThreadPool**: Cuando el servidor se inicia, se crea un pool de hilos con un número definido de trabajadores. Estos hilos están siempre en espera de recibir nuevas tareas desde el canal de comunicación.
- **Distribución de tareas**: Cada vez que el servidor recibe una nueva conexión, envía la tarea correspondiente (como procesar una solicitud HTTP) a uno de los hilos del pool.
- **Ejecución de las tareas**: Cada hilo recibe una tarea y la ejecuta. Una vez completada, el hilo queda libre para recibir una nueva tarea.

Esta estrategia mejora significativamente la escalabilidad del servidor, permitiendo gestionar múltiples clientes de manera eficiente.

## 3. Operaciones HTTP y Manipulación del Archivo JSON

El servidor ofrece varias rutas HTTP que permiten realizar operaciones sobre un archivo JSON que almacena información de personajes. Estas operaciones se implementan en funciones específicas dentro del código:

- **`GET`**: Obtiene entradas del archivo JSON.
- **`POST`**: Añade nuevas entradas al archivo.
- **`PUT`**: Reemplaza una entrada completa basada en su ID.
- **`PATCH`**: Actualiza un campo específico de una entrada (por ejemplo, el nombre).
- **`DELETE`**: Elimina una entrada del archivo.

Cada una de estas operaciones interactúa con el archivo JSON de manera que se asegura la consistencia de los datos almacenados, leyendo y escribiendo los cambios de forma concurrente pero segura.

### 4. Manejo de Archivos y Serialización

El archivo `one_piece2.json` es la base de datos del proyecto. Todas las operaciones se realizan sobre este archivo, leyendo y escribiendo en formato JSON. Para facilitar estas operaciones, el proyecto utiliza las bibliotecas `serde` y `serde_json` que permiten la serialización y deserialización de los datos entre estructuras de Rust y JSON.

# Implementación de la concurrencia

La concurrencia en este proyecto se gestiona mediante un patrón de **ThreadPool** (o pool de hilos), lo que permite al servidor HTTP procesar múltiples peticiones simultáneamente de manera eficiente. A continuación se detalla cómo se ha implementado este sistema y cómo gestiona las tareas concurrentes:

### 1. Creación del ThreadPool

El **ThreadPool** es un conjunto de hilos (workers) pre-creados que permanecen en espera para procesar las tareas asignadas. Al iniciar el servidor, se crea una instancia de `ThreadPool` con un número fijo de hilos, determinado por el tamaño que se le pasa al constructor (`ThreadPool::new(5)` en este caso).

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);
        let (sender, receiver) = mpsc::channel();
        let receiver = Arc::new(Mutex::new(receiver));
        let mut workers = Vec::with_capacity(size);
```

```
        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }
}
```

Aquí, el pool de hilos se crea con una cantidad específica de hilos (`size`), y cada hilo se asocia con un receptor que está a la espera de recibir tareas (trabajos o `Job`). El uso de `mpsc::channel` permite enviar trabajos desde el servidor principal a los hilos.

## 2. Comunicación entre Hilos

El canal **mpsc** (multiple-producer, single-consumer) se utiliza para la comunicación entre el servidor principal (que recibe las conexiones HTTP) y los hilos del pool. Este canal permite que el servidor envíe trabajos (`Job`), que son las tareas que representan las conexiones entrantes, a los hilos disponibles en el pool.

**Sender y Receiver:**

- **Sender**: Permite al servidor enviar trabajos (funciones `FnOnce`) al pool de hilos.
- **Receiver**: Cada hilo recibe los trabajos enviados por el servidor a través de este canal.

**Código relevante:**
rust
Copiar código
```
let (sender, receiver) = mpsc::channel();
let receiver = Arc::new(Mutex::new(receiver));
```

Para garantizar la seguridad en entornos concurrentes, el `receiver` se comparte entre múltiples hilos utilizando **`Arc`** (Contador de Referencias Atómicas) y **`Mutex`** (exclusión mutua), lo que asegura que solo un hilo pueda acceder al canal a la vez.

## 3. Ejecución de Tareas Concurrentes

Cada vez que llega una conexión al servidor, se envía una tarea (`Job`) a uno de los hilos del pool para que la procese. Esto se logra con el método `execute` del `ThreadPool`, que toma una función de tipo `FnOnce` (una tarea que se ejecutará una sola vez) y la envía a través del canal:

```
pub fn execute<F>(&self, f: F)
where
    F: FnOnce() + Send + 'static,
{
    let job = Box::new(f);
    self.sender.send(job).unwrap();
}
```

La función `execute` encapsula la tarea dentro de un `Box<dyn FnOnce()>`, que es un puntero dinámico a la función que se va a ejecutar. Esta tarea se envía a través del canal `sender`, que será recibida por uno de los hilos en el `ThreadPool`.

### 4. Hilos Trabajadores (Workers)

Cada hilo del pool se representa con una estructura `Worker`. Los hilos están en un bucle infinito a la espera de recibir nuevas tareas a través del canal `receiver`. Cuando un hilo recibe una tarea, la ejecuta y luego vuelve a esperar nuevas tareas.

```
struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        let thread = thread::spawn(move || loop {
            let job = receiver.lock().unwrap().recv().unwrap();
            println!("Worker {id} executing");
            job();
        });

        Worker { id, thread }
    }
}
```

- **Recepción de tareas**: Los hilos utilizan `receiver.lock().unwrap().recv()` para recibir una tarea. El `Mutex` asegura que solo un hilo acceda al canal de trabajo en cada momento, evitando condiciones de carrera.
- **Ejecución**: Una vez recibida la tarea (`job()`), el hilo la ejecuta. En este caso, la tarea puede ser procesar una solicitud HTTP.

## 5. Gestión de Conexiones

El servidor escucha las conexiones entrantes y, por cada conexión recibida, crea una nueva tarea que será ejecutada por el `ThreadPool`. Esto asegura que el servidor no se bloquee mientras espera a que las solicitudes sean procesadas. En lugar de bloquearse, el servidor puede seguir aceptando nuevas conexiones mientras los hilos procesan las tareas en paralelo.

```
for stream in listener.incoming() {
    let stream = stream.unwrap();
    pool.execute(|| {
        handle_connection(stream);
    });
}
```

Por cada conexión entrante, el servidor llama a `pool.execute` y le pasa una función que manejará la conexión. Esta función es enviada a uno de los hilos del `ThreadPool` para su procesamiento.

## 6. Sincronización y Seguridad

La concurrencia en este servidor está bien gestionada gracias al uso de **`Arc`** y **`Mutex`**, que aseguran el acceso seguro a recursos compartidos entre los hilos. Estos mecanismos permiten compartir el receptor (`receiver`) del canal entre todos los hilos de manera segura y evitar que varios hilos intenten acceder al mismo recurso de forma concurrente.

- **`Arc`**: Permite la referencia compartida a un mismo recurso entre múltiples hilos de manera segura y sin la sobrecarga de bloquear el acceso para lectura.
- **`Mutex`**: Garantiza que solo un hilo acceda a un recurso compartido a la vez, previniendo condiciones de carrera.

# Manejo de cookies

Para el manejo de cookies primero se analiza las cookies de las solicitudes entrantes y luego se establecen cookies en las respuestas salientes.

1. Analizar Cookies de la Solicitud
   Cuando un cliente envía una solicitud al servidor, puede incluir cookies en el encabezado Cookie. Para extraer estas cookies, se necesita:

- Recuperar el encabezado Cookie de los encabezados de la solicitud.
- Dividir el valor del encabezado en cookies individuales.
- Analizar cada cookie como un par clave-valor y almacenarlas en un HashMap.

La función parse_cookies logra esto iterando sobre el encabezado Cookie, dividiéndolo por punto y coma para separar las cookies individuales, y luego dividiendo cada cookie por el signo igual para separar la clave y el valor.

2. Establecer Cookies en la Respuesta
   Para enviar cookies de vuelta al cliente, se debe incluir un encabezado Set-Cookie en la respuesta. Este encabezado especifica el nombre de la cookie, su valor y atributos adicionales, como la ruta y la bandera HTTP-only. La función set_cookie construye el valor del encabezado Set-Cookie y lo agrega a los encabezados de la respuesta.

Integración en la Función handle_connection
En la función handle_connection, se integra la lógica para manejar cookies de la siguiente manera:

- Analizar cookies: Se llama a parse_cookies con los encabezados de la solicitud para extraer las cookies.
- Establecer cookies: Se utiliza set_cookie para añadir cookies a los encabezados de la respuesta.

La expiración de una cookie se maneja mediante la función set_cookie, la cual permite establecer cookies con una fecha de expiración específica. Esta función toma cuatro parámetros: un vector mutable de cookies (cookies), el nombre de la cookie (name), el valor de la cookie (value) y una opción de cadena (expires) que representa la fecha de expiración.

Verificación de Expiración
Para verificar si una cookie ha expirado, se utiliza la función is_cookie_expired, que toma la fecha de expiración como una cadena y la compara con el tiempo actual. Si la fecha de expiración es anterior al tiempo actual, la cookie se considera expirada.

# Instrucciones para ejecutar y probar el servidor

Para asegurar el correcto funcionamiento del servidor, se utilizó **Postman** como herramienta para realizar pruebas HTTP sobre las distintas rutas y operaciones implementadas. A continuación, se detallan las instrucciones utilizadas para probar cada operación:

1. **GET** - Obtener las entradas desde el archivo JSON:
    - **Ruta**: `GET localhost:7878/entries`
    - Descripción: Este endpoint recupera todas las entradas almacenadas en el archivo `one_piece2.json`. Se puede limitar el número de resultados retornados agregando un parámetro opcional en la URL, como `localhost:7878/entries?limit=5`.
2. **POST** - Agregar una nueva entrada:
    - **Ruta**: `POST localhost:7878/submit`
    - Descripción: Este endpoint permite agregar una nueva entrada al archivo JSON. Se envía el objeto de la nueva entrada en formato JSON en el cuerpo de la solicitud. Ejemplo de cuerpo en JSON:

```
{

   "rank": "Pirate King",
   "trend": "Increasing",
   "season": 2,
   "episode": 5,
   "name": "Monkey D. Luffy",
   "start": 1999,
   "total_votes": "5000",
   "average_rating": 9.8

}
```

    -
3. **PUT** - Reemplazar todos los campos de una entrada existente:
    - **Ruta**: `PUT localhost:7878/put_entity`
    - Descripción: Utilizado para modificar una entrada existente en el archivo `one_piece2.json` según su `id`. El cuerpo de la solicitud debe contener todos los campos de la entrada que se desea actualizar. Ejemplo de cuerpo en JSON:

```
{

   "id": 1,
```

```
    "rank": "Admiral",
    "trend": "Decreasing",
    "season": 3,
    "episode": 12,
    "name": "Roronoa Zoro",
    "start": 2001,
    "total_votes": "3000",
    "average_rating": 8.9
}
```

4. **PATCH** - Actualizar solo el nombre de una entrada:
    - **Ruta**: `PATCH localhost:7878/patch_entity_name`
    - Descripción: Este endpoint permite modificar únicamente el campo `name` de una entrada existente. El cuerpo de la solicitud debe incluir el `id` de la entrada a modificar y el nuevo nombre. Ejemplo de cuerpo en JSON:
    - {

```
    "id": 2,
    "name": "Sanji Vinsmoke"
}
```

5. **DELETE** - Eliminar una entrada:
    - **Ruta**: `DELETE localhost:7878/delete_entity`
    - Descripción: Este endpoint elimina una entrada existente del archivo `one_piece2.json` según su `id`. El cuerpo de la solicitud debe contener el `id` de la entrada a eliminar. Ejemplo de cuerpo en JSON:

```
{

    "id": 3
}
```

Cada una de estas pruebas fue realizada exitosamente mediante **Postman**, asegurando que las distintas operaciones del servidor gestionen correctamente las peticiones y manipulen el archivo JSON conforme a lo esperado.

# Estructura y directorios

La estructura de archivos y directorios que se utilizó está organizada de manera que archivo cumple un rol dentro del servidor HTTP y la gestión de datos. A continuación se presenta esta estructura desglosada con su explicación:

`src/`: Directorio principal del código fuente de Rust.

- `main.rs`: Punto de entrada. Aquí se inicializa el servidor, se crean las conexiones, se maneja el flujo inicial del programa y maneja las solicitudes de las funciones a otros archivos. Este archivo de igual manera contiene la definición de las pruebas unitarias del proyecto, mediante las cuales se prueba el buen funcionamiento del servidor.

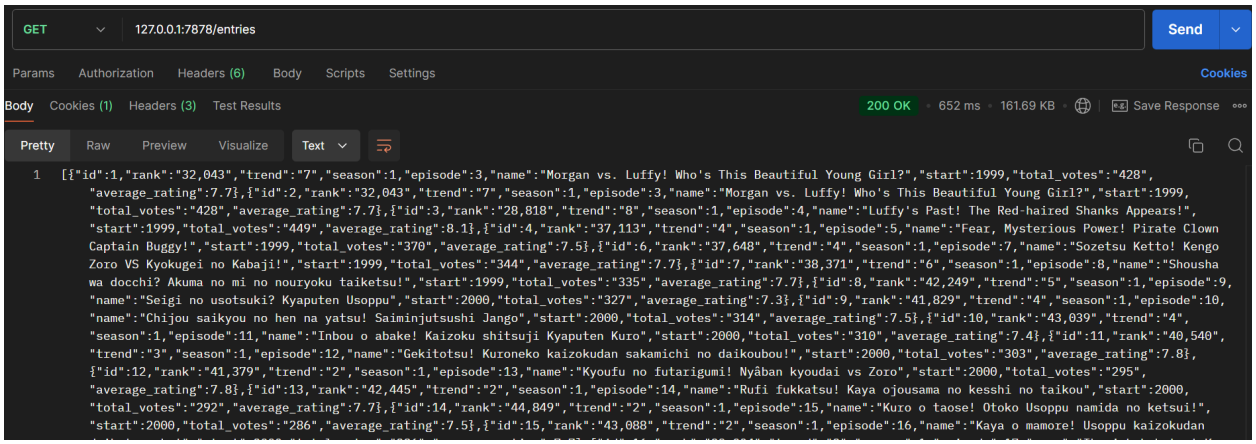
- `lib.rs`: Maneja la concurrencia del servidor mediante la estructura `ThreadPool`, mediante esta se permite procesar múltiples solicitudes concurrentemente utilizando hilos.

- `endpoints.rs`: Contiene la definicion de las operaciones CRUD del servidor HTTP, manejando datos en formato JSON, en este archivo se interactúa directamente con los datos del archivo `one_piece2.json`, en donde se hacen lectura, modificación, escritura y eliminación de los datos.

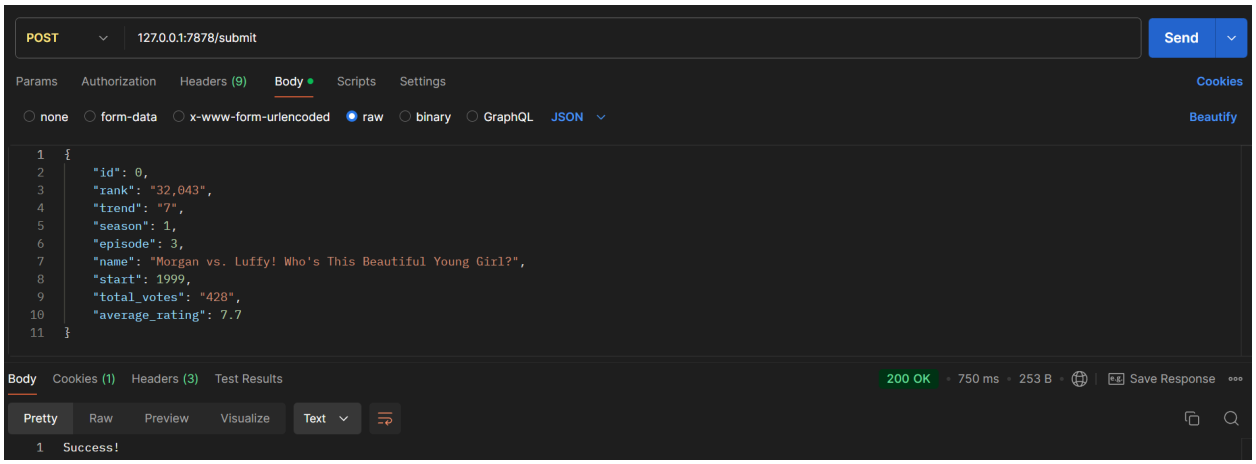
# Resultados de pruebas

Para asegurar el buen funcionamiento de los `endpoints` de nuestro servidor utilizamos **Postman**. A continuación se presentan imágenes de pruebas realizadas sobre las operaciones HTTP:

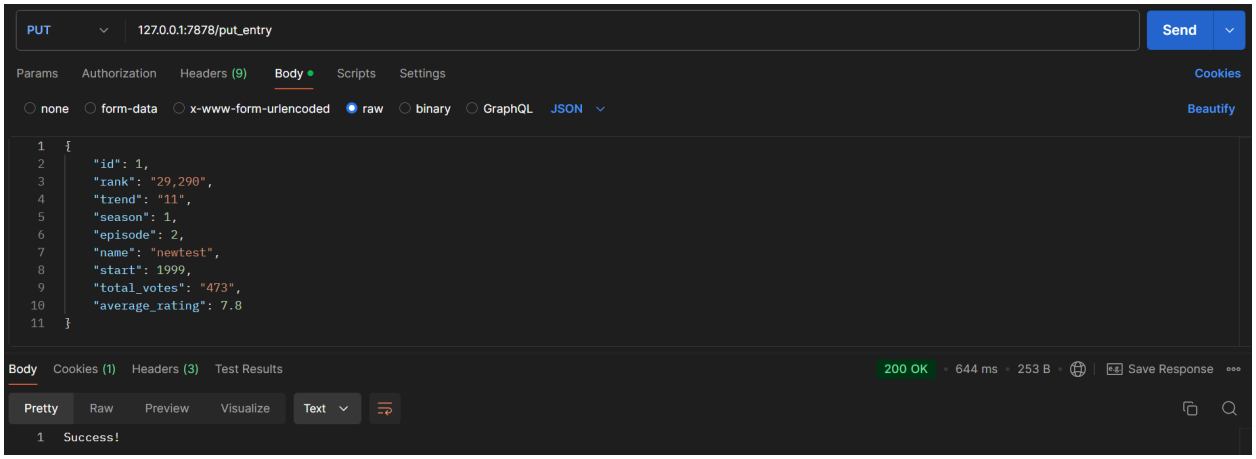
1. **GET** - Obtener las entradas desde el archivo JSON:



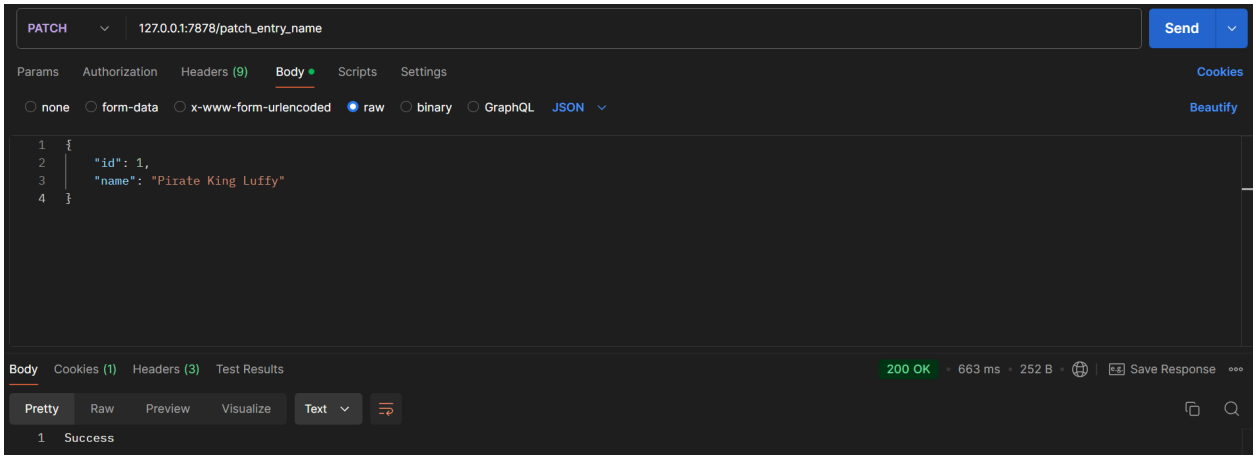
2. **POST** - Agregar una nueva entrada:



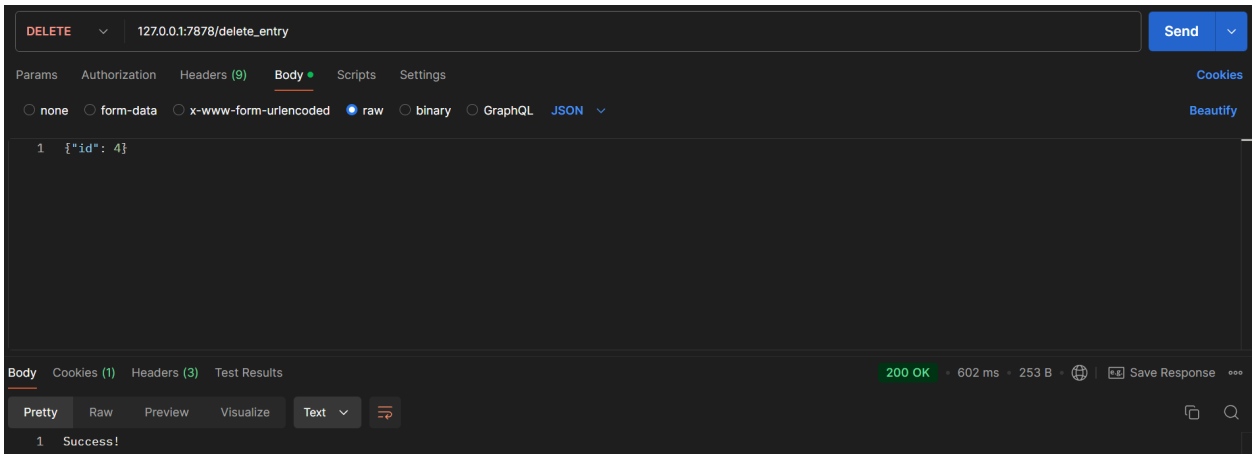

3. **PUT** - Reemplazar todos los campos de una entrada existente:




4. **PATCH** - Actualizar solo el nombre de una entrada:



5. **DELETE** - Eliminar una entrada:




.

# Manual de usuario

## Prerrequisitos

- Tener una versión estable de Rust instalada.
- Tener Cargo instalado, para la gestión de paquetes.

## Dependencias

- `serde` para la serialización/deserialización de JSON.
- `chrono` para la gestión de fechas.
- `threadpool` para el manejo de concurrencia.

## Instalación

1. **Clonar el repositorio**
   ```
   git clone https://github.com/danielcornejo14/rust-http-server.git
   cd servidor-http-rust
   ```

2. **Compilar el proyecto**
   ```
   cargo build
   ```

3. **Ejecutar el proyecto**
   ```
   cargo run
   ```

## Uso del Servidor HTTP

- El servidor esta asignado al puerto `"127.0.0.1:7878"`
- El número de hilos en el `ThreadPool` es 5

### Solicitudes HTTP

| | | |
|---|---|---|
| `GET` | `localhost:7878/entries` | Este endpoint recupera todas las entradas almacenadas en el archivo `one_piece2.json`. |
| `POST` | `localhost:7878/submit` | Este endpoint permite agregar una nueva entrada al archivo JSON. Se envía el objeto de la nueva entrada en formato JSON en el cuerpo de la solicitud. |
| `PUT` | `localhost:7878/put_entity` | Utilizado para modificar una entrada existente en el archivo `one_piece2.json` según su `id`. El cuerpo de la solicitud debe contener todos los campos de la entrada que se desea actualizar. |
| `PATCH` | `localhost:7878/patch_entity_name` | Este endpoint permite modificar únicamente el campo `name` de una entrada existente. El cuerpo de la solicitud debe incluir el `id` de la entrada a modificar y el nuevo nombre. |

| | | |
|---|---|---|
| `DELETE` | `localhost:7878/delete_entity` | Este endpoint elimina una entrada existente del archivo `one_piece2.json` según su `id`. El cuerpo de la solicitud debe contener el `id` de la entrada a eliminar. |

## Pruebas

En el proyecto se incluyeron múltiples pruebas unitarias para probar las funcionalidades.

Para ejecutar las pruebas:

```
cargo test
```