Universidad ORT Uruguay

Facultad de Ingeniería

Programación de Redes
Obligatorio

Santiago Toscanini (232566)
Sofía Tejerina (209239)

Entregado como requisito de la materia Programación de Redes

3 de diciembre de 2020

Índice

Alcance del sistema

El sistema consta de un Servidor TCP que permite:

- Aceptar conexiones de varios clientes TCP
- Permite consultar los clientes conectados
- Permite realizar alta, baja y modificación de clientes

A los clientes TCP con los que estableció conexión:

- Registrarse o ingresar con una cuenta ya creada al sistema
- Obtener el listado de fotos de un usuario
- Subir fotos asociadas al propio usuario
- Comentar una foto asociada a un usuario
- Consultar los usuarios registrados en el sistema
- Obtener los comentarios de una imagen del usuario
- Desconectarse

Los clientes TCP proveen una interfaz para acceder a estas funcionalidades brindadas por el servidor.

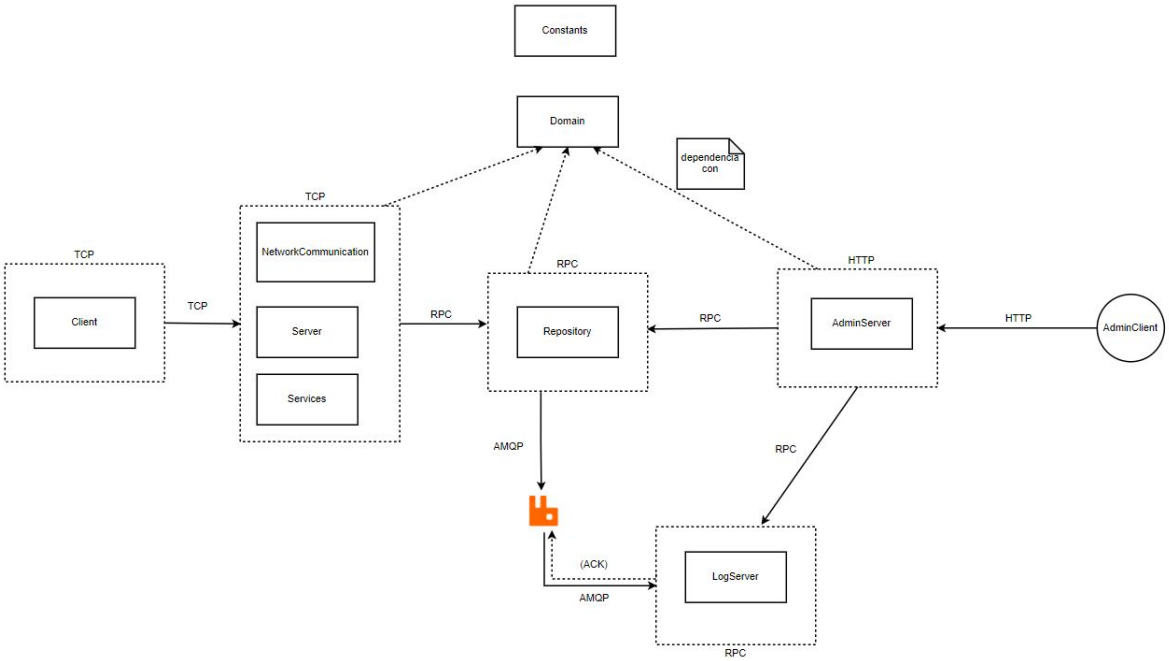
Se cuenta con un servidor de Logs, que brinda una comunicación pasiva para la recepción de los realizados por los eventos del Servidor y los Administradores y para la obtención de los logs por parte de los últimos.

Por último se cuenta con un Servidor Administrador que permite a los clientes administradores realizar:

- Alta, baja y modificación de los usuarios
- Obtención de los logs registrados

Descripción de la Arquitectura

Diagrama de la solución propuesta



Cliente TCP

El cliente TCP utiliza sockets para establecer una conexión con el servidor TCP a través de un puerto y una IP establecidas. Utiliza un hilo separado para escuchar lo que el servidor responda y el principal para escribirle mensajes al servidor.

Servidor TCP

El servidor TCP utiliza sockets para el intercambio de información con el cliente, por cada cliente que se conecta se abre un nuevo hilo desde el lado del servidor.

La comunicación se realiza con streams que permiten el intercambio de bytes con los datos, luego de que el usuario ingresa a su cuenta su email queda guardado junto con su “Communication” (una clase que guarda el stream creado a partir del cliente TCP y se encarga de realizar las operaciones de lectura y escritura para la comunicación entre ambos), para esta comunicación se implementó el siguiente protocolo: antes de enviar los datos se envía el tamaño de los datos que el receptor recibirá. Esto se debe a que la información se envía en un arreglo de bytes, que debe tener el largo para instanciar, crear arreglos del largo máximo para asegurarnos de que no se pierda ningún datos sería un desperdicio de recursos, por lo que se opta por enviar primero el datos del largo y a partir de este crear el arreglo para recibirlos. El

largo, al igual que toda la información también se envía en un arreglo de bytes, pero el tamaño máximo posible para un array de bytes que traiga un valor numérico es 4, un desperdicio menor en el peor caso que la alternativa planteada.

Para el envío de archivos utilizamos el siguiente protocolo: primero se envía el largo del nombre del archivo, luego el nombre. Una vez enviado el nombre mandamos el largo del archivo y comenzamos a enviar el archivo en paquetes. Del otro lado, para leerlo se va calculando según el valor del largo del archivo, que recibió previamente, cuando ya no quedan más tramas por ser recibidas. A la vez que se van recibiendo las tramas se van enviando para por File stream.

El intercambio de archivos también se da a través de Stream, pero en este caso se utiliza la clase FileStream, mientras que para el intercambio de mensajes utilizamos NetworkStream.

El Servidor utiliza un servicio que se encarga de la obtención y guardado de datos en un proyecto repositorio, los datos se transfieren entre estos dos a través de RPC con gRPC que utiliza un protocol buffer (mecanismo extensible, neutral en cuanto al lenguaje y la plataforma, de Google para serializar datos estructurados). Esto es mejor que una REST API porque estas serializan a un json mientras que esto serializa a binario, de esa forma es más eficiente en cuanto a ancho de banda. La deserialización y serialización es mucho menos pesada a nivel de CPU que si estuviéramos utilizando Json.

Repository

En este proyecto se guardan en local los datos de los usuarios. Es un proyecto gRPC por lo que para guardar los datos en él se utiliza RCP con los protocol buffers, una forma de serializar los datos creada por Google que es más eficiente en comparación con Json (utilizado en las REST APIs).

En los archivos .proto declaramos los servicios con sus métodos rpc, los cuales recibirán y devolverán “Messages” que declararemos en el archivo. Al compilar se generarán clases (en nuestro caso serán .cs) que nos permitirán trabajar con el servicio y los mensajes declarados.

Para utilizar las clase autogenerada a partir de .proto creamos un servicio que implementa la clase del servicio que creamos y sobreescribirá los métodos. Este servicio será el que utilicen para guardar y obtener los datos de los otros proyectos. Para utilizar los métodos se puede optar por la versión asíncrona de estos (el cual tiene el sufijo “Async” en el nombre). Nosotros utilizamos las versiones asíncronas para no trancar el hilo de ejecución sin hacer uso de otro hilo.

Al guardar los datos localmente, para que estos no se pierdan utilizamos el patrón Singleton aprovechando el uso de inyección de dependencias que provee el proyecto.

LogServer

El servidor de log brinda una comunicación pasiva con Repository y AdminServer, el primero escribe los eventos del sistema (ya que todos pasan por él), los Logs registrados se guardan en un repositorio de logs, similar a como se guardan los datos en Repository; haciendo uso de un Singleton e inyección de dependencias.

Los logs se envían a través AMQP (Advanced Message Queuing Protocol) para esto utilizamos RabbitMQ (un software que implementa el protocolo mencionado). Con Rabbit tendremos el Repository que manda a un exchange (utilizando un routing key, del mismo nombre que la cola en nuestro caso) la información, este la manda a las colas (en nuestro caso una única cola que se llama LogQueue). Dentro de este servidor tendremos un servicio consumiendo los mensajes que lleguen a la cola y guardando los logs el re repositorio de logs.

El AdminServer se comunicará por gRCP con un Servicio (que implementa un servicio del proto correspondiente) que le proveerá los logs guardados (paginados).

Para los logs utilizamos un formato similar a log4j.

AdminServer

El AdminServer es una API con la cual se comunicaran los clientes administradores a través de HTTP.

Los services utilizados para comunicarse por gRCP para hacer el alta, baja y modificación de los usuarios (con el Repository) y la obtención de los logs (con el LogServer) utilizan el patrón singleton y la inyección de dependencias, al igual que hicimos con los proyectos gRPC.

Para el manejo de errores utilizamos un Filter que hace el catch de las excepciones que llegan al controller y devuelve la respuesta que corresponda junto con el código de estado adecuado según el caso.

Domain

Es el proyecto utilizado para determinar las entidades de negocio que se persisten en local: Usuarios,Comentarios e Imágenes.

Constants

La finalidad de este proyecto es guardar constantes que sirven para todas las soluciones, tales como los puertos de los servidores, y el formato de los logs.

Funcionamiento de la aplicación

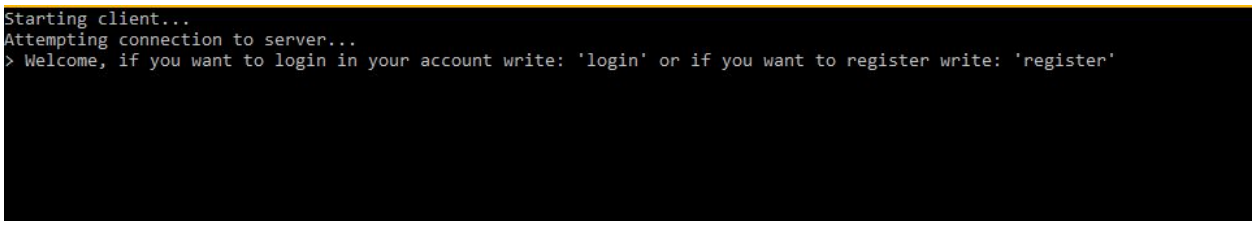
La aplicación consta de cinco proyectos que deben estar levantados para el correcto funcionamiento de todas las funcionalidades, estos proyectos son:

- AdminServer
- Client
- Server
- LogServer
- Repository

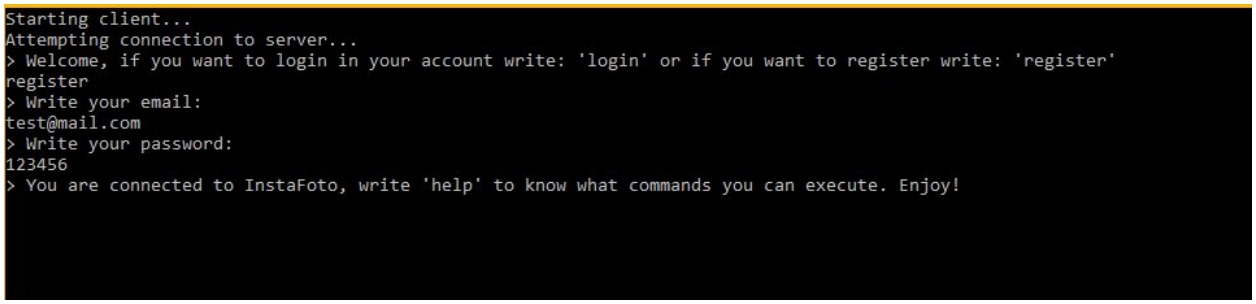
Cliente TCP y Servidor TCP

El Cliente TCP y el Servidor TCP fueron implementados como aplicaciones de consola, que como dijimos anteriormente, se comunican utilizando sockets. Es importante aclarar que para que funcione el Cliente debe estar previamente funcionando el Servidor, sino dará un error al intentar establecer la conexión.

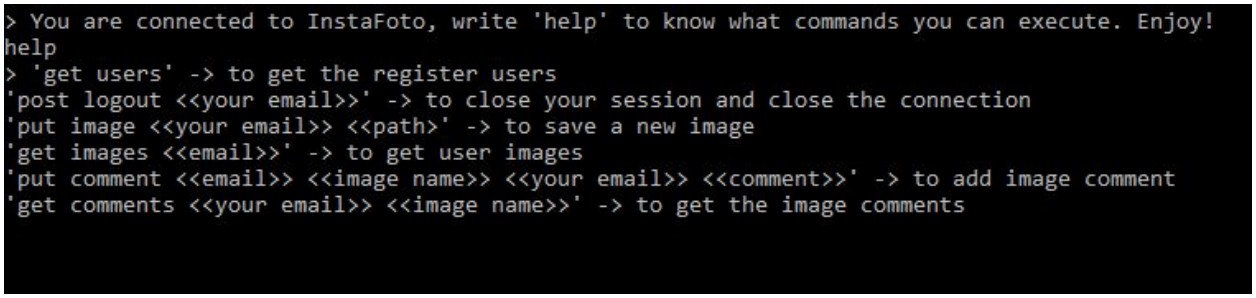
Al iniciar un cliente obtendremos el siguiente mensaje:



Como el mensaje indica al escribir login te permitirá ingresar con una cuenta que ya exista, y si todavía no tiene cuenta puede optar por registrarse:



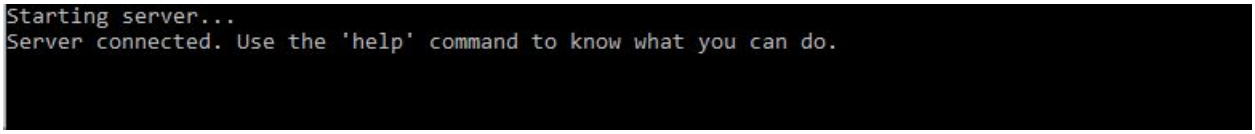
Para hacer más fácil su uso implementamos un comando “help” que el cliente puede utilizar para saber con qué funcionalidades dispone y cómo utilizarlas:



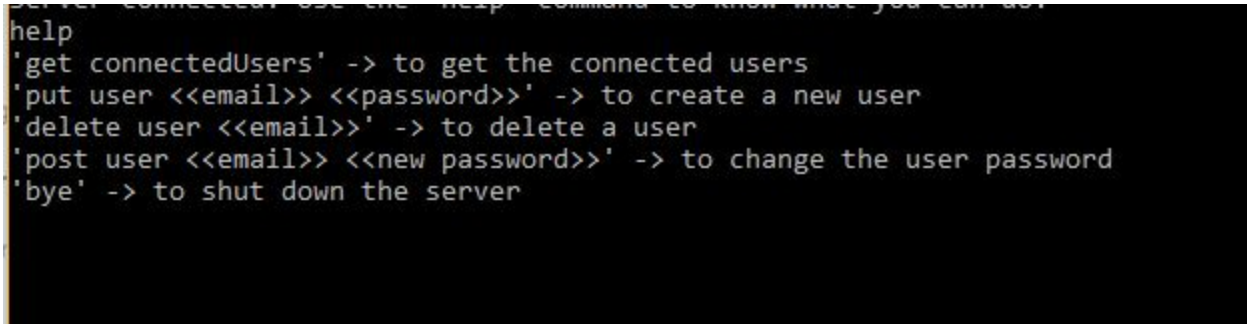
Como muestra la lista el cliente poderá:

- Consultar qué clientes registrados hay
- Cerrar su sesión
- Agregar una imagen
- Obtener las imágenes de un usuario
- Agregar un comentario a la imagen de un usuario
- Obtener los comentarios de una de sus imágenes

Desde la consola del Servidor se puede observar el siguiente mensaje:



Como se puede ver en el mensaje, también contamos con un comando “help” desde el lado del servidor para consultar las funcionalidades brindadas:

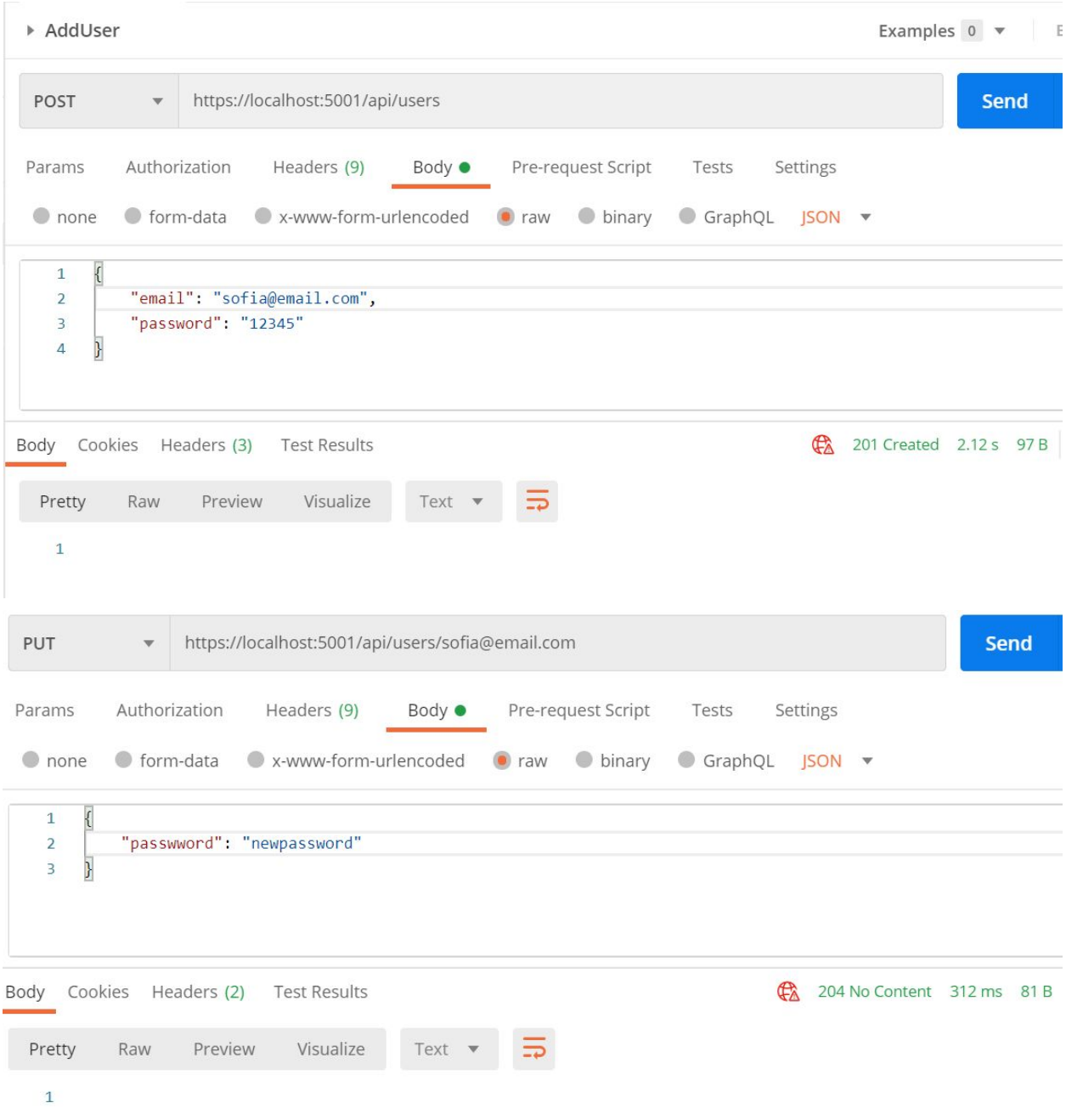


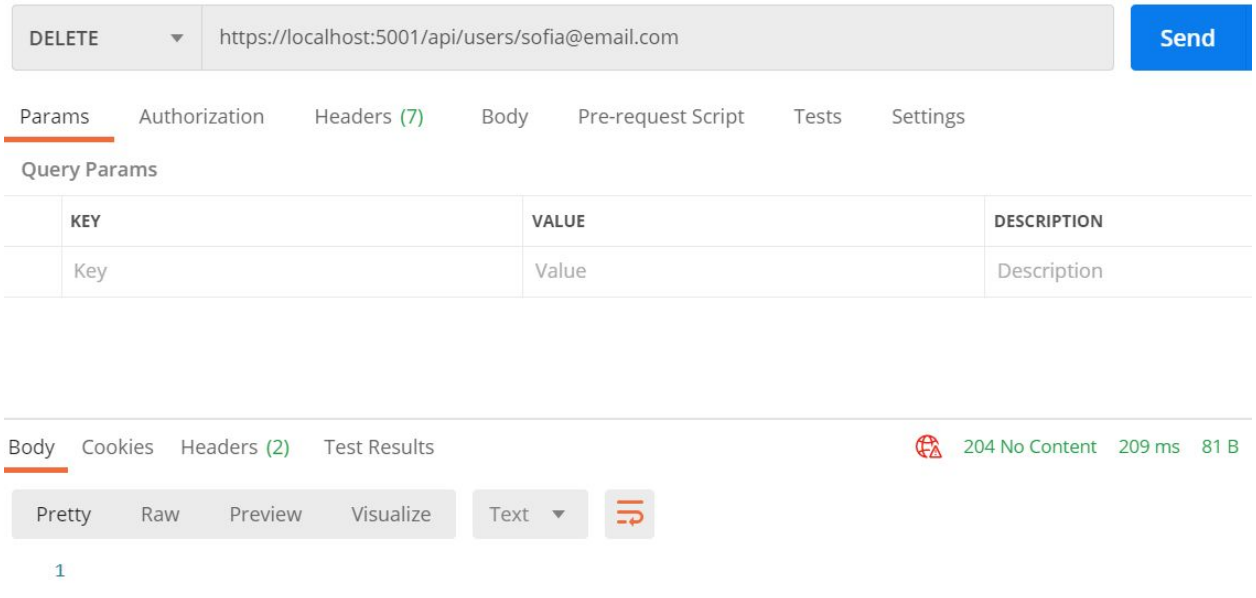
Desde la consola del servidor se poderá:

- Obtener los usuarios que se encuentren conectados
- Crear un usuario
- Eliminar un usuario registrado
- Modificar un usuario existente (actualmente solo se podrá cambiar la contraseña)
- Cerrar la sesión del servidor para que no acepte más conexiones (tampoco se podrán ejecutar las funcionalidades del servidor)

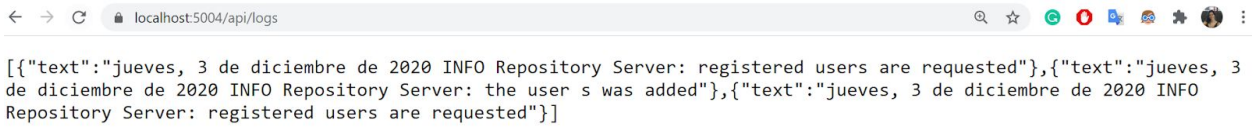
Servidor de administración y clientes de administración

El servidor de administración es una API web de la cual sus clientes son browsers que realizarán consultas:

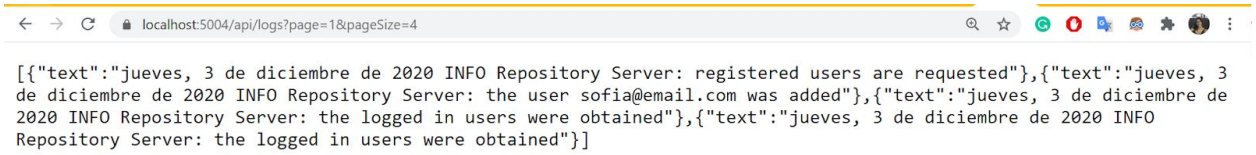




En los ejemplos anteriores se utilizó la aplicación Postman para realizar las consultas como cliente administrativo, también se podría hacer desde cualquier navegador.



En las consultas de los logs registrados utilizamos paginación, por defecto las paginación comienza en la página 1 y trae 3 elementos:



A continuación dejaremos un ejemplo de consulta de logs, es la última página y tiene un único elemento: