

Escuela de Informática

Curso: Sistemas Distribuidos

EIF 508

Profesor: Armando Arce

Programa para consultas web a base de datos

Bryan Murillo Rodríguez

Bidkar Solís Pérez

II Ciclo, 2019

**Introducción**

En un mundo cada vez mas competitivo se exige que las operaciones sean más simples, agiles e innovadoras. En el trabajo constante de paso de mensajes para sistemas, el tener un recurso al servicio de los clientes es de mucha importancia y es por el cual los sistemas distribuidos entran en acción para formar parte de esa solución.

Una de las aplicaciones de sistemas distribuidos en es la sección del manejo de bases de datos, en la que una cantidad deseada de base de datos son usadas para almacenar información de los clientes, como toda funcionalidad normal. Pero con la aplicación de sistemas distribuidos permite agilizar las consultas debido a que no hay dependencia de una sola base de datos y un solo sistema que controle las consultas.

Con un sistema distribuidos, si hay un cliente que envía miles de consultas, y ocupa que se almacenen de forma ágil, rápida y eficiente el sistema distribuido le permite que sus procesos de consulta de inserción sean rápidos, ya que distribuye el trabajo en un grupo de programas que controlan la base de datos para almacenar la información en múltiples bases de datos al mismo tiempo.

A continuación se presenta el proyecto en el cual vamos a tener un conector que nos permitirá traducir las consultas desde la consulta de tipo HTTP, hasta un segundo programa que controla y distribuye las consultas a un grupo de programas que cada uno tiene asignada una base de datos para realizar las consultas enviadas por los clientes.

**Descripción del problema**

**Segundo Proyecto**

El objetivo de este proyecto es la creación de un conector Web hacia un servidor de bases de datos particionada. Una partición es una división de una base de datos lógica o sus elementos constitutivos en distintas partes independientes. Para ello se utilizará el protocolo TreeQL disponible en: https://treeql.org Se deben crear tres programas, primero un conector que capturará todas las solicitudes HTTP y las pasará a un coordinador de procesos a través de ZMQ.

Luego se debe programar el servidor de bases de datos que recibe solicitudes desde el coordinador y accede a una base de datos SQLite. Los datos serán manipulados usando el formato JSON, tal como se describe en TreeQL.

**Protocolo TreeQL**

TreeQL es un lenguaje de consulta para API y un tiempo de ejecución para completar esas consultas con sus datos existentes. TreeQL proporciona una descripción completa y comprensible de los datos en su API, brinda a los clientes el poder de pedir exactamente lo que necesitan y nada más, facilita la evolución de las API a lo largo del tiempo y habilita herramientas familiares para desarrolladores.

GET /records/{table} - list records

POST /records/{table} - create a record

GET /records/{table}/{id} - read a record

PUT /records/{table}/{id} - update a record's values

DELETE /records/{table}/{id} - delete a record

Se deben implementar únicamente las opciones básicas, no es necesario programar las opciones: include, join, filtros, ordenamientos, paginación o límite de registros.

**Particionamiento de la base de datos**

La base de datos se debe particionar en cuatro partes, para lo que se deben ejecutar cuatro procesos de acceso a bases de datos SQLite (aparte del proceso conector y el proceso coordinador). El coordinador de procesos que recibe las solicitudes desde el conector HTTP debe enviar cada consulta de datos a todos los servidores (patrón PUBLICAR/SUBSCRIBIR) para que alguno de ellos conteste la solicitud al coordinador y este a su vez responda al conector HTTP. Sin embargo, las operaciones de creación de registros se asignarán a un único servidor de datos (note que el patrón PUSH/PULL asigna las solicitudes en forma automática). Es importante resaltar que cada proceso accede a una base de datos SQLite diferente.

**Descripción detallada y explicación de los componentes principales**

**Conector web:**

El conector web es el programa encargado de aceptar las consultas HTTP que vienen del servidor localhost bajo el puerto destino que recibe el mensaje del request. Para el control de este tipo tenemos el patrón de desarrollo para ZMQ de Adaptador.

Con el patron adaptador tenemos dos sockets, los cuales uno es el encargado para el envió de la consulta y el otro para el recibo del resultado. En este caso se recibirán los request enviados desde tcp://\*:8080. Y de manera interna para procesar las consultas recibidas se enviarán mensajes mediante tcp://localhost:5555. Por ese puerto se enviará la información necesaria que el programa coordinador posteriormente envía a los programas de servicio de la base de datos.

El servidor se mantiene escuchando las consultas, al estar en un bucle while que mantiene vivo al servidor para las recepciones de los mensajes y así enviarlas al coordinador para ser posteriormente enviados a los que se suscriban al servicio.

A continuación, el código resultante:

// conectorweb.c

// HTTP adapter in 0MQ

#include "zhelpers.h"

int main(void)

{

  //Contexto para el manejo de sockets

  void \*ctx = zmq\_ctx\_new();

  //Servicio que permite recibir de los request web

  void \*hserver = zmq\_socket(ctx, ZMQ\_STREAM);

  zmq\_bind(hserver, "tcp://\*:8080");

  //Socket para la comunicacion interna con los servicios para el envio de la informacion necesaria por parte de

  //los programas internos para procesar las consultas

  void \*tserver = zmq\_socket(ctx, ZMQ\_REQ);

  zmq\_connect(tserver, "tcp://localhost:5555");

  //Parametros para la configuracion de los sockets

  uint8\_t id[256];

  size\_t id\_size = 256;

  char \*line;

  int more;

  size\_t more\_size = sizeof(more);

  int length = 0;

  //Mensaje para indicar que el servidor se encuentra esperando consultas

  printf("Servidor iniciado para recibir consultas\n");

  while (1)

  {

    // first frame has ID, the next the request.

    id\_size = zmq\_recv(hserver, id, 256, 0);

    // Get HTTP request

    char \*request = s\_recv(hserver);

    printf("1: valor del request %s\n", request);

    free(request);

    request = s\_recv(hserver);

    printf("2: valor del request %s\n", request);

    free(request);

    request = s\_recv(hserver);

    printf("3: valor del request %s\n", request);

    char http\_response[] =

        "HTTP/1.0 200 OK\n"

        "Connection: close\n"

        "Content-Type: text/html\n"

        "Content-Length:          ";

    //Recepcion del request para ser enviado a la programacion interna de la consulta

    s\_send(tserver, request);

    line = s\_recv(tserver);

    sscanf(line, "%d", &length);

    sprintf(http\_response, "%d\n\n", length);

    //Envio de la informacion para los subscriptiores del servicio

    zmq\_send(hserver, id, id\_size, ZMQ\_SNDMORE);

    zmq\_send(hserver, http\_response,

             strlen(http\_response), ZMQ\_SNDMORE);

    free(line);

    while (1)

    {

      //Mantiene el servicio de conector vivo para las demas consultas

      line = s\_recv(tserver);

      zmq\_send(hserver, id, id\_size, ZMQ\_SNDMORE);

      zmq\_send(hserver, line, strlen(line), ZMQ\_SNDMORE);

      free(line);

      zmq\_getsockopt(tserver, ZMQ\_RCVMORE, &more, &more\_size);

      if (!more)

      {

        // Send a zero to close connection to client

        zmq\_send(hserver, id, id\_size, ZMQ\_SNDMORE);

        zmq\_send(hserver, NULL, 0, ZMQ\_SNDMORE);

        break;

      }

    }

    free(request);

  }

  // Cierre de los servicios y sockets

  zmq\_close(hserver);

  zmq\_close(tserver);

  zmq\_ctx\_term(ctx);

  return 0;

}

**Coordinador**

El programa de coordinador es el encargado de recibir la información enviada desde el connector web, para la consulta que se desea llevar a cabo en las bases de datos. Esta información es capturada para obtener el tipo de consulta HTTP (Get, Post, Put,Delete), la base de datos donde se realiza las consultas y la información necesaria para realizar el string de la consulta o query para la base de datos.

La lógica del código a continuación trata de obtener la información necesaria que ocupa el programa servidor de base de datos del request enviado por el conector web, de esta información obtenemos el tipo de request, la base de datos y la consulta a ejecutar o los parámetros necesarios usados en el programa servidor de base de datos

A continuación, el código desarrollado:

// coordinador.c

// Publisher coordinador server

// Binds REP socket to tcp://\*:5555

#include <zhelpers.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

// Function to replace a string with another

// string

char \*replaceWord(const char \*s, const char \*oldW,

                  const char \*newW)

{

    char \*result;

    int i, cnt = 0;

    int newWlen = strlen(newW);

    int oldWlen = strlen(oldW);

    // Counting the number of times old word

    // occur in the string

    for (i = 0; s[i] != '\0'; i++)

    {

        if (strstr(&s[i], oldW) == &s[i])

        {

            cnt++;

            // Jumping to index after the old word.

            i += oldWlen - 1;

        }

    }

    // Making new string of enough length

    result = (char \*)malloc(i + cnt \* (newWlen - oldWlen) + 1);

    i = 0;

    while (\*s)

    {

        // compare the substring with the result

        if (strstr(s, oldW) == s)

        {

            strcpy(&result[i], newW);

            i += newWlen;

            s += oldWlen;

        }

        else

            result[i++] = \*s++;

    }

    result[i] = '\0';

    return result;

}

int main(void)

{

    //Context

    void \*context = zmq\_ctx\_new();

    //Socket para comunicar con el adapter

    void \*responder = zmq\_socket(context, ZMQ\_REP);

    zmq\_bind(responder, "tcp://\*:5555");

    //Socket Publisher

    void \*contextPub = zmq\_ctx\_new();

    void \*publisher = zmq\_socket(contextPub, ZMQ\_PUB);

    int rc = zmq\_bind(publisher, "tcp://\*:5556");

    rc = zmq\_bind(publisher, "ipc://weather.ipc");

    char length[20];

    //Bucle para mantener vivo el servicio

    while (1)

    {

        //Variale para recibir la informacion de la consulta

        char \*request;

        request = s\_recv(responder);

        /\*

        request = replaceWord(request, "%7B", "");

        request = replaceWord(request, "%7D", "");

        \*/

        if (request == NULL)

            continue;

        // Imprime lo que trae la consulta

        printf("Received in coordinator: %s\n", request);

        //Metodo utilizado para obtener el tipo de request y se almacena en la variable requestType

        char \*url = strtok(request, " ");

        char \*requestType;

        char \*id;

        //Variable para almacenar el tipo de request HTTP

        requestType = url;

        //A partir de aca seguimos filtrando la informacion para obtener el nombre de la base de datos y la informacion necesaria

        //para formatear el query para la ejecucion en la base de datos por medio del servidor web

        printf("Request despues del primer strtok: %s\n", url);

        url = strtok(NULL, " ");

        printf("Tipo de request: %s\n", requestType);

        url = strtok(url, "/");

        printf("Request despues del segundo strtok: %s\n", url);

        url = strtok(NULL, "/");

        printf("Request despues del tercer strtok: %s\n", url);

        id = strtok(NULL, "/");

        printf("Valor de ID: %s\n", id);

        printf("Request despues del strtok con NULL: %s\n", url);

        //Concatenamos la informacion necesaria para enviarla a los subscriptores

        strcat(requestType, ",");

        strcat(requestType, url);

        // Algunas veces vienen los ID para consultas individiales, se obtinen aca y se hace la concatenacion si

        //la variable id obtine algun valor

        if (id != NULL)

        {

            strcat(requestType, ",");

            strcat(requestType, id);

        }

        //Imprimimos el url y el request que vamos enviar al servidor web

        printf("Url despues de replace: %s\n", url);

        printf("Statement: %s\n", requestType);

        //Envia la informacion a los subscriptores para que realicen el trabajo dependiendo de la informacion enviada

        s\_send(publisher, requestType);

        free(request);

        //Reply que recibe el conector web cuando finaliza las consultas

        char \*reply = "Hello World";

        sprintf(length, "%d", strlen(reply));

        s\_sendmore(responder, length);

        s\_send(responder, reply);

    }

    //Cierre de los recursos para la los sockets

    zmq\_close(responder);

    zmq\_close(publisher);

    zmq\_ctx\_destroy(context);

    zmq\_ctx\_destroy(contextPub);

    return 0;

}

**Servidor de base de datos**

El servidor de base de datos es el programa el cual recibe la información ya en formato necesario para la elaboración de la consulta SQL que permite ejecutar con la librería de SQLLite 3. El servidor tomo el string de información y lo divide en las partes necesarias que se ocupa para formular la sentencia de SQL para ejecutar en la base de datos.

A continuación, el código resultante:

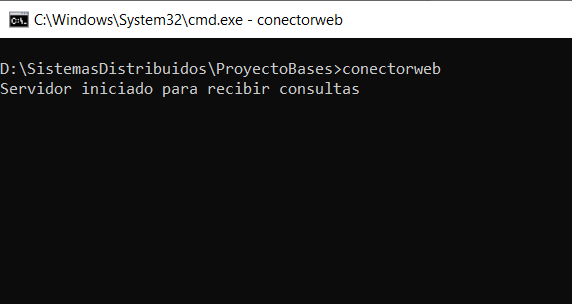
**Análisis de resultados de pruebas**

A continuación, se presentan los escenarios de los request HTTP de GET, POST , PUT and DELETE ejecutados mediante los tres programas anteriores. Se tomara de ejemplo la table de profesor de las base de datos a las que están asociados los programas servidor de base de datos.

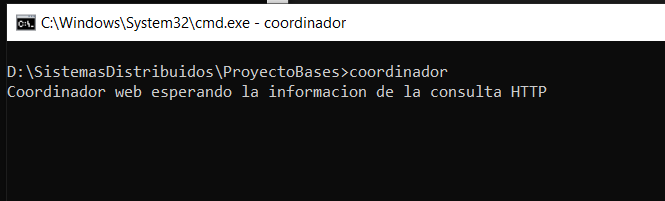
Las consultas serán ejecutadas desde el programa PostMan para enviar las consultas desde el servidor de Localhost formateadas acorde a Treeql.

El primer servicio a correr es el conector web, seguido del coordinador y por ultimo el programa de base de datos para la ejecución de las consultas.

Conector web:

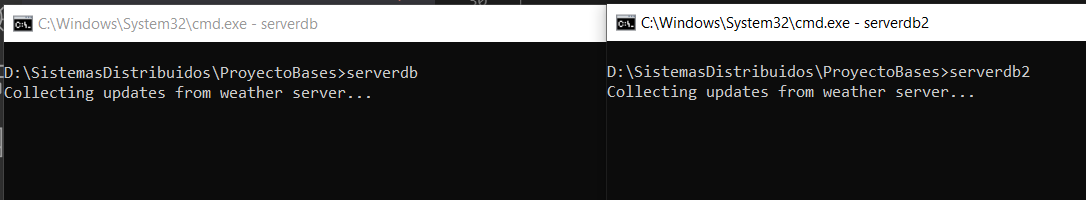


Coordinador:



Servidor de base de datos:

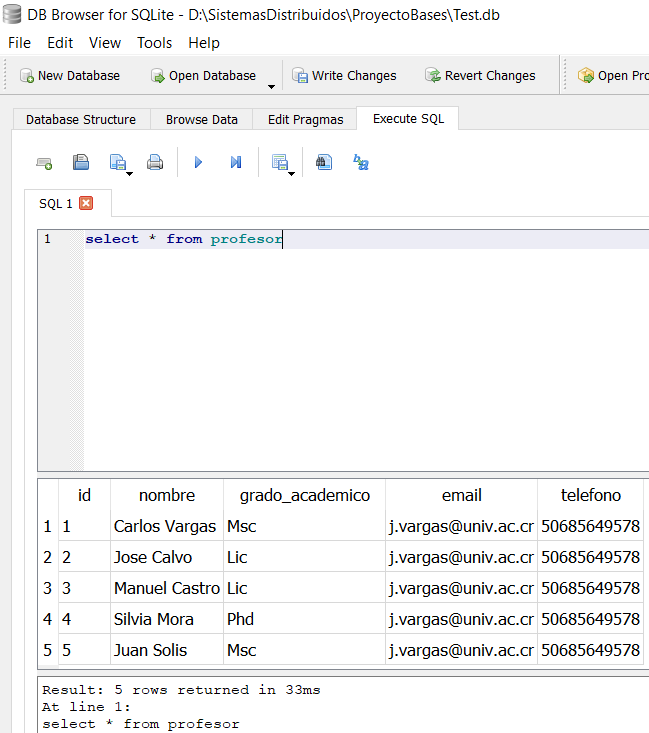
En este caso vamos a poner ejecutar dos programas de servidor base de datos.



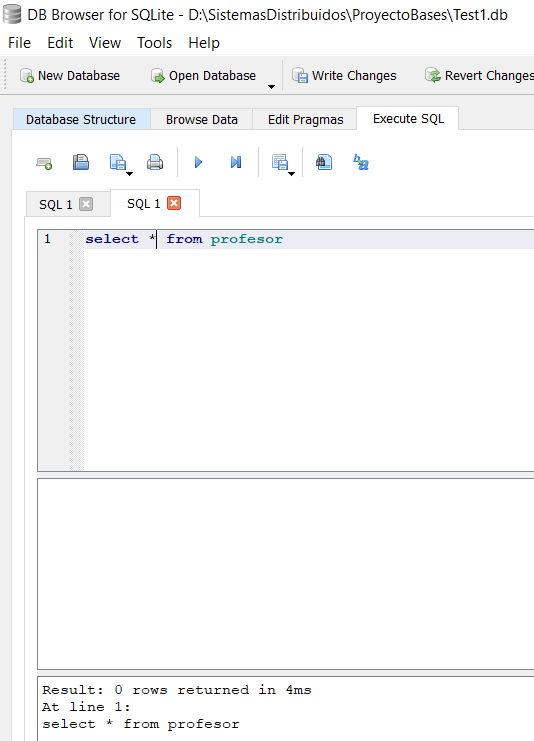
A continuación, las consultas http, y su resultado:

A continuación, un par de imágenes que nos indican los datos actuales en las bases de datos:

Test:



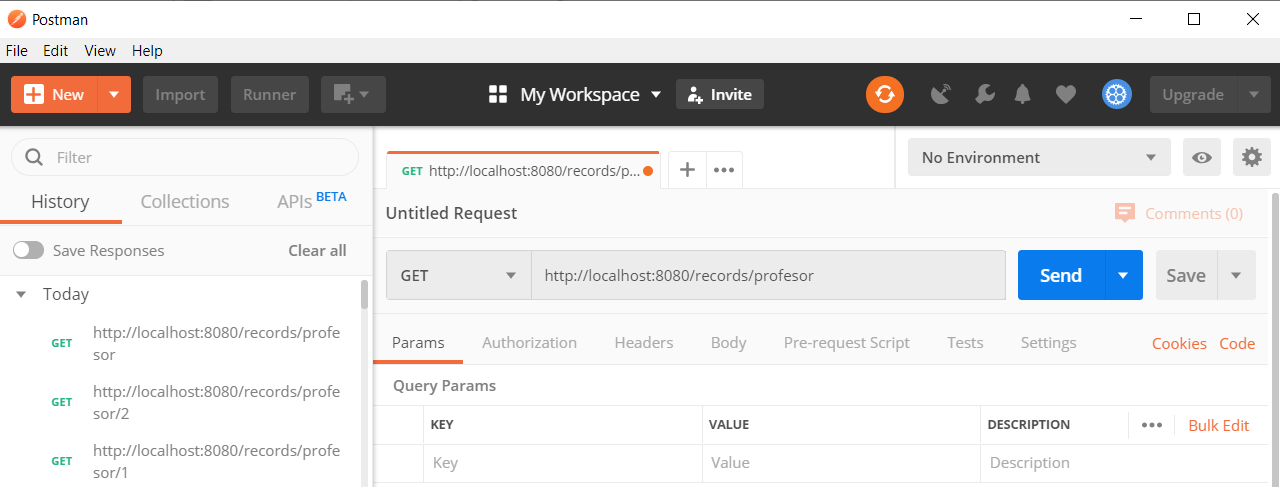
Test1:



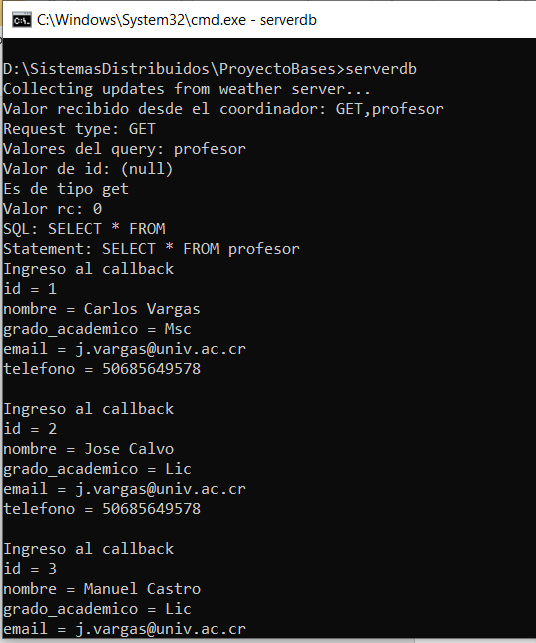
Como se observa solo la base de datos Test tiene datos de profesores, mientras que la de Test1 no tiene nada de datos. Por lo tanto, se espera que solo el servidor unido a la base de datos Test sea la que nos devuelva resultados de la consulta.

Get todos los profesores:

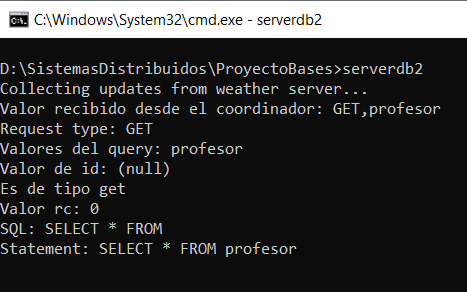
Ejecución con Postma:



Get de serverdb, unido a Test.db:



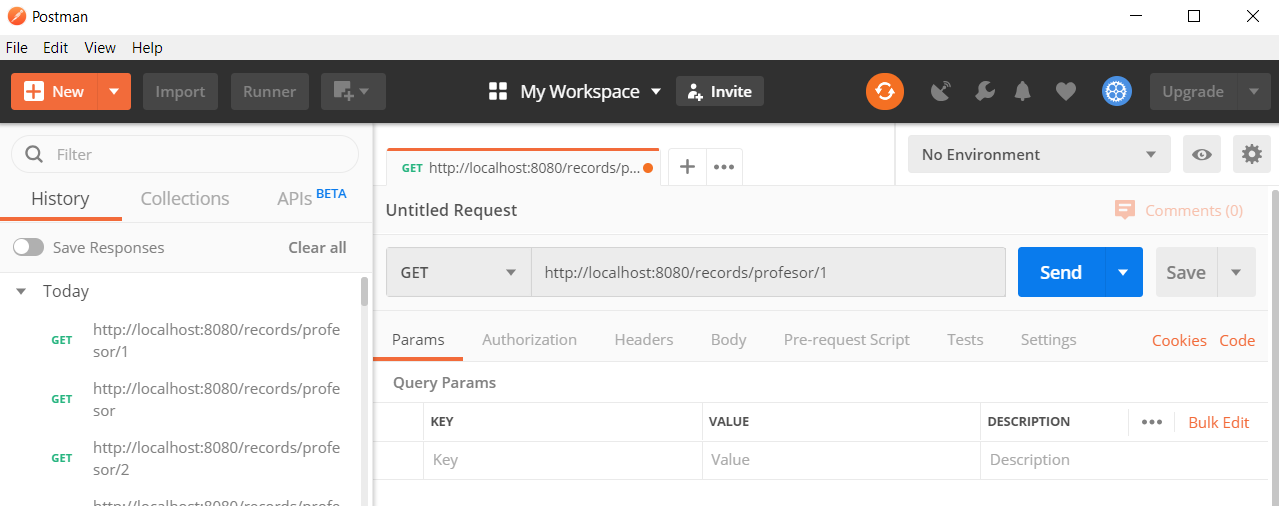
Get de serverdb2, unido a Test1.db:



Como se observa en las imágenes solo serverdb, es el único que obtuvo resultados de la consulta a la base de datos. Mientras que el programa asociado a la base Test1.db que no tiene registros solo indica la consulta enviada.

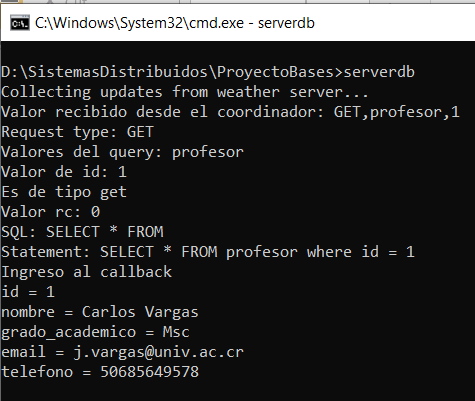
Get un profesor en específico:

Ejecución de la consulta en Postman:

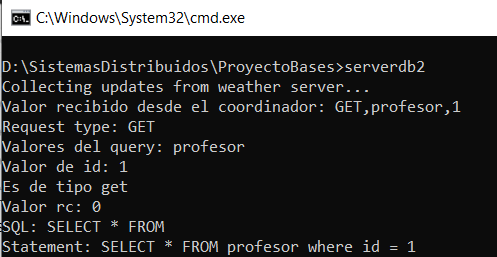


En la consulta estamos tratando de obtener la información del profesor cuyo ID es igual a 1. Por lo tanto, esperamos solo respuesta del programa con nombre de ejecución serverdb.

Resultado get serverdb:



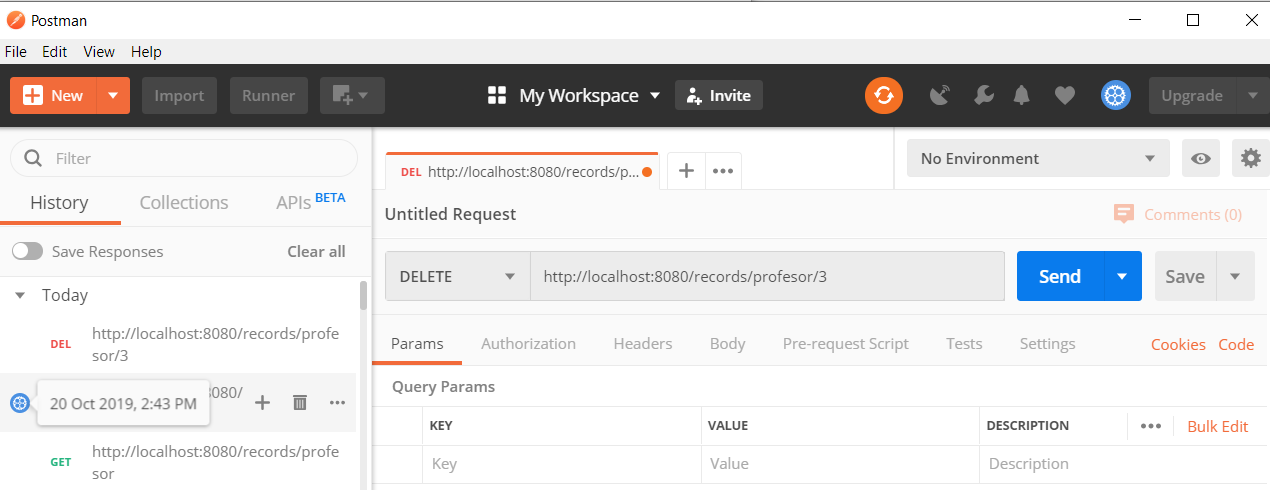
Resultado get serverdb2



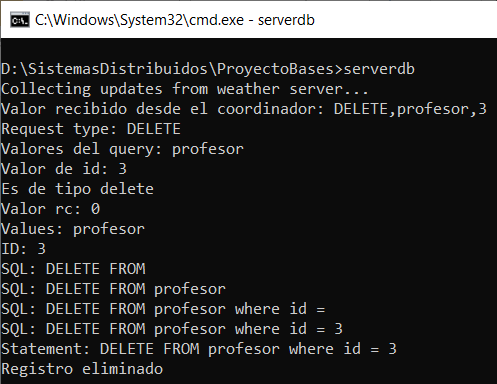
Delete de un registro:

A continuación, realizaremos la eliminación de un registro de la base de datos con el ID 3, el cual nos eliminara de la base de datos que contenga ese valor registrado en su base de datos mediante su servidor de base de datos asociado.

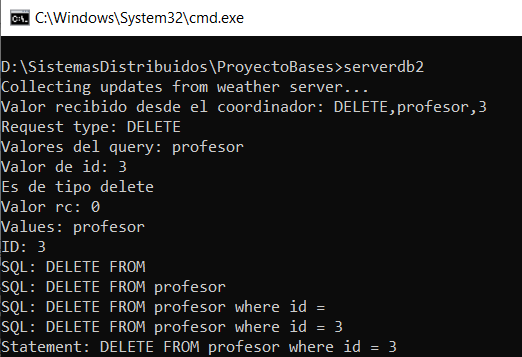
Consulta en Postman:



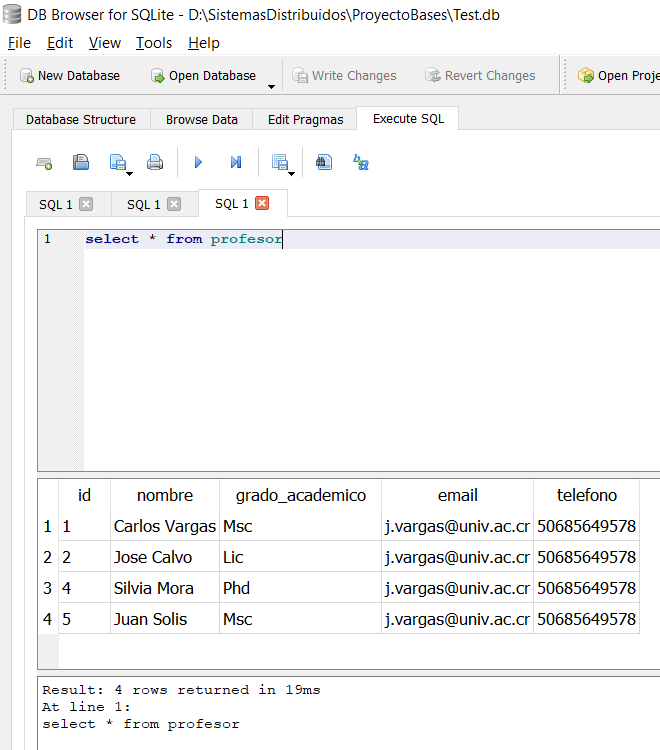
Resultado en serverdb:



Resultado en serverdb2:



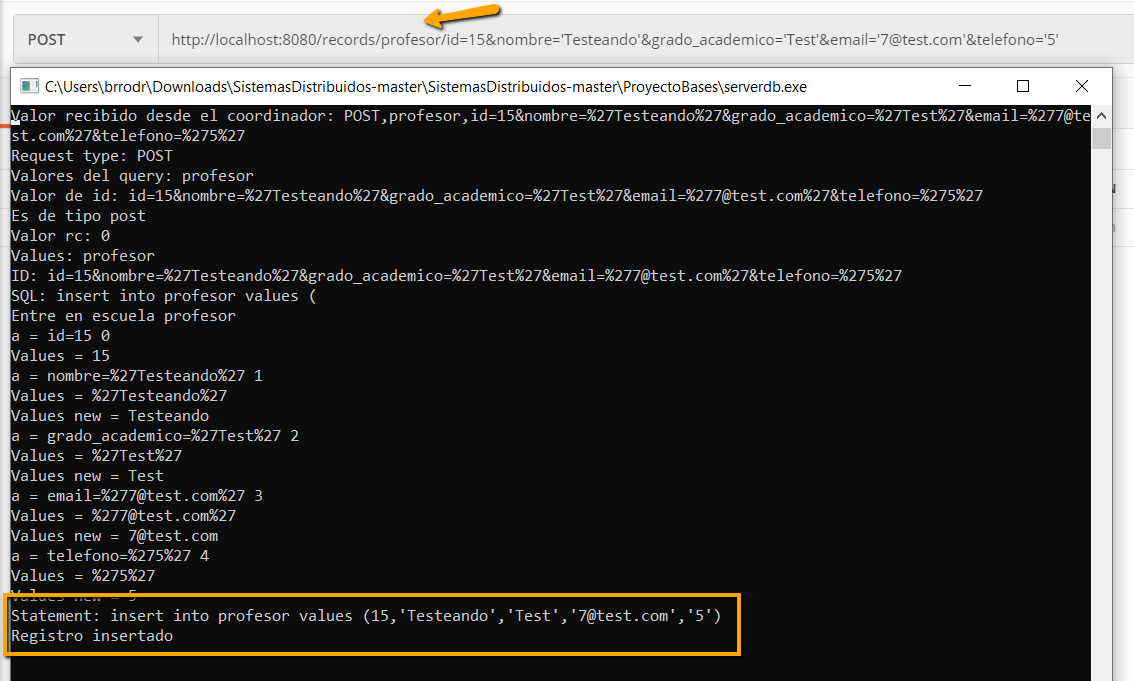
Registros en la base de datos Test.bd:

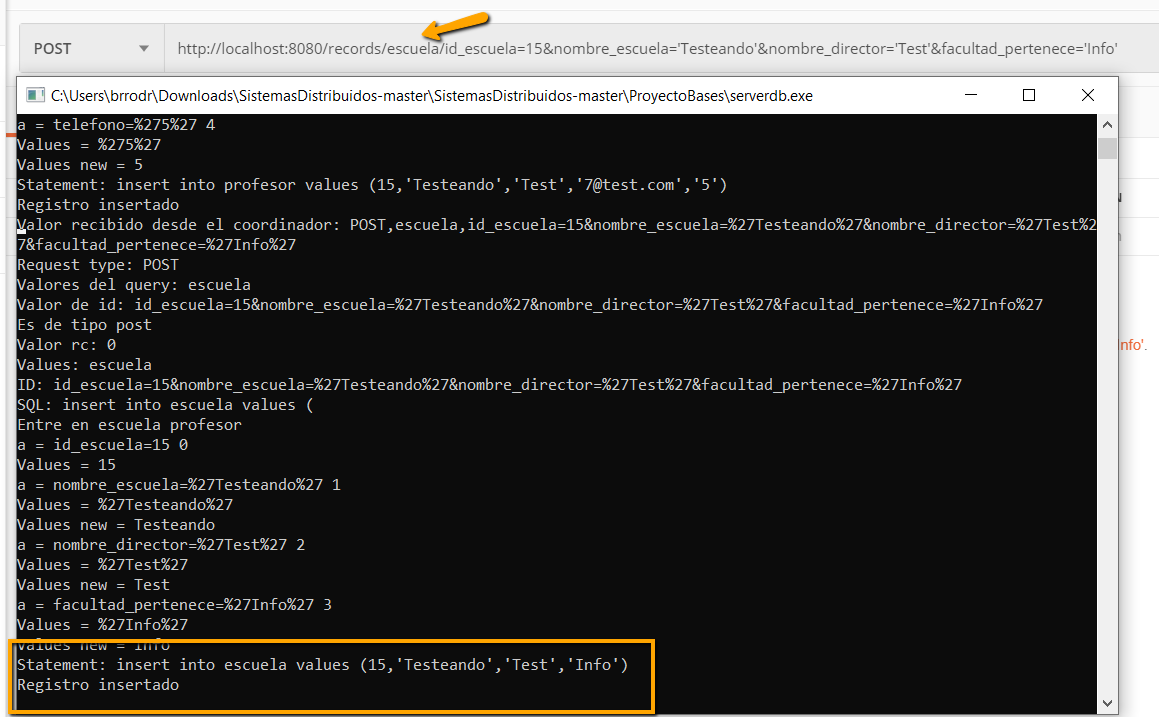


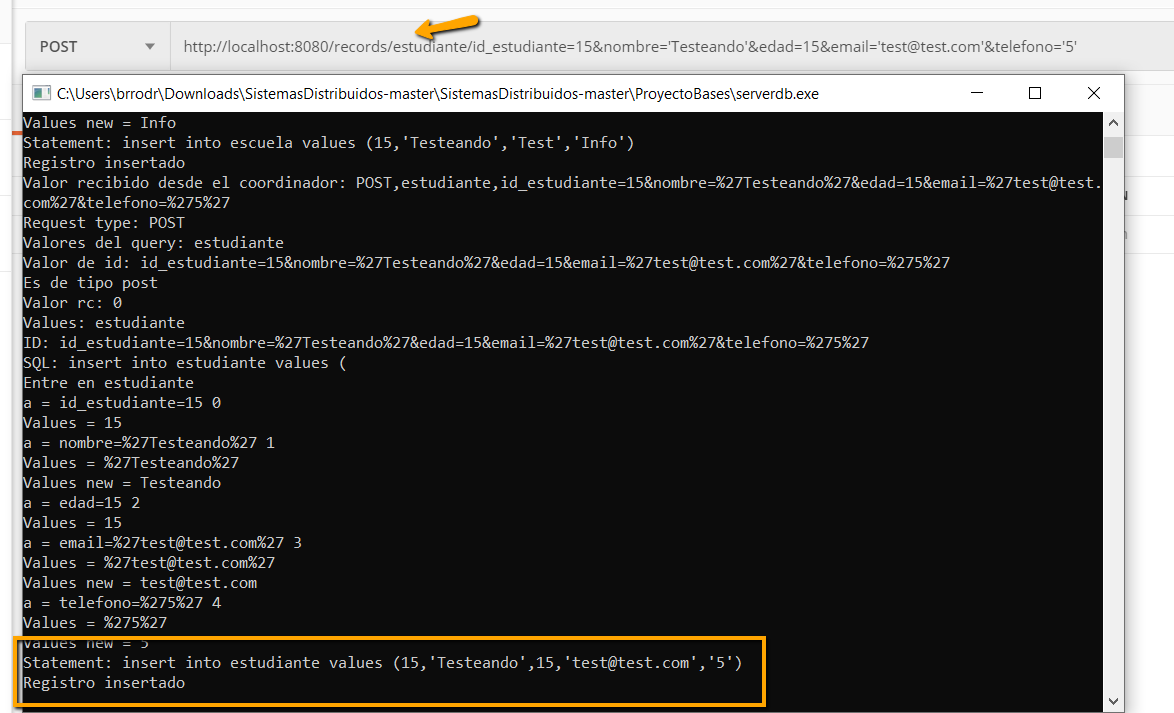
Como podemos observar el registro a la base de datos asociada con el programa serverdb con el ID igual a 3 fue eliminado de la base de datos, mientras que el segundo servidor de base de datos solo muestra la consulta solicitada.

Insert Testing:  
Curso







  
Update Testing:

