Taller Arquitectura Cliente-Servidor: Sockets, WebServer y API Rest

Julio Ariel Hurtado

Junio 13 de 2022

1 Introducción

En este taller haremos un recorrido por una serie de recursos que permitirán ir ampliando el panoramaa y conocimientos de soluciones basadas en el estilo cliente servidor. Los códigos se encuentran disponibles en el repositorio `https://github.com/arielhurtado/client-server-evolution.git`

Siga el repaso de este documento, descargue los códigos, revíselos y entiéndalos. Ejecute el servidor y el cliente, revise que el cliente encuentre el servidor en el puerto adecuado. Si el servidor es web debe crear un cliente web o usar una herramienta como postman Descargar Postman. Primero van a trabajar con el servidor genérico hecho por completo en casa, luego se trabajará con uno específico para la agencia de viajes, luego la agencia de viajes con un servidor web básico (HttpServer) y finalmente una API Rest que utiliza un servidor java empresarial JEE más completo como GlassFish o Payara Server los cuales vienen incorporados en netbeans. Finalmente se les pedirá una pequeña implementación sobre el código de la agencia de viajes para que cuente con una API Rest.

2 Servidor Multihilo basado en Sockets TCP/IP Genérico

La programación de sockets se refiere a la escritura de programas que se ejecutan en varias computadoras en las que todos los dispositivos están conectados entre sí mediante una red.

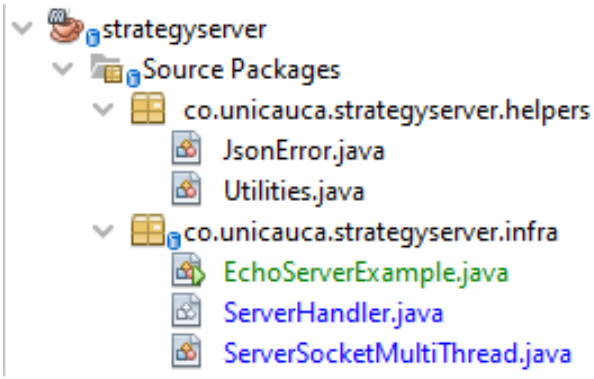
Hay dos protocolos de comunicación disponibles para la programación de sockets: protocolo de datagramas de usuario (UDP) y protocolo de control de transferencia (TCP).

La principal diferencia entre los dos es que UDP no tiene conexión, lo que significa que no hay sesión entre el cliente y el servidor, mientras que TCP está orientado a la conexión, lo que significa que primero se debe establecer una conexión exclusiva entre el cliente y el servidor para que se produzca la comunicación.

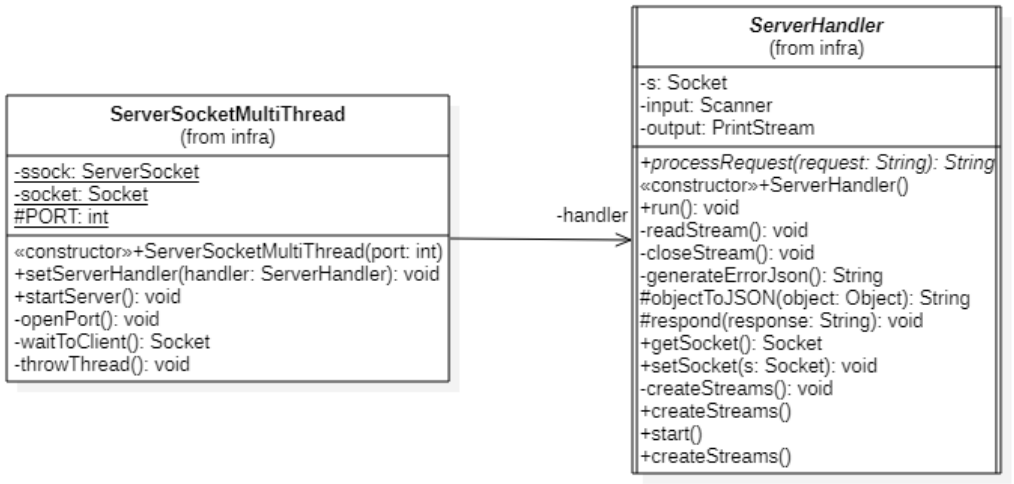
Java proporciona una colección de clases e interfaces que se encargan de los detalles de comunicación de bajo nivel entre el cliente y el servidor. Para ello se necesita trabajar con los paquetes java.net (ServerSocket, Socket) y java.io (InputStream, OutputStream).

Ahora revisemos el proyecto strategyserver que programa un servidor genérico. En la siguiente Figura podemos ver las clases que conforman la solución: ServerSocketMultiThread y ServerHandler.

La clase EchoServerExample es un servidor de ejemplo que devuelve lo que se solicita. La estructura de clases se puede ver en la gráfica UML. ServerSocketMultiThread representa un servidor multihilo genérico que inicia un servidor al recibir el mensaje startServer(). El método startServer



abre el puerto y luego en un ciclo infinito escucha la solictud de un cliente y la atiende de manera concurrente, disparando(throw) un hilo para manejar la comunicación con el cliente.



Fragmento de Código Java de la Clase Servidor

```
public class ServerSocketMultiThread{
/**
 * El hilo manejador de la peticion del cliente
 */
private ServerHandler handler;

/**
 * Pone en funcionamiento la logica central del servidor
 */
public void startServer() {
    openPort();
    while (true) {
        waitToClient();
        throwThread();
        }
    }
...
}
```

A continuación la implementación del método que atiende la solitud del cliente (método trowThread).

Fragmento de Código Java de la Clase Servidor

```
/**
 * Lanza el hilo
 */
private void throwThread() {
    try {
        handler = (ServerHandler) handler.getClass().newInstance();
        handler.setSocket(socket);
        handler.start();
    } catch (InstantiationException | IllegalAccessException ex) {
        Logger.getLogger(ServerSocketMultiThread.class.getName())
        .log(Level.SEVERE, null, ex);
    }
}
```

En este método, el servidor delega (usando el patrón estrategia) en un Manejador de Petición (Server Handler) el procesamiento de la respuesta en su propio hilo de ejecución. La clase Server-Handler a su vez hereda de la clase Thread y sobreescribe el método run(), el cual se utiliza de método plantilla . En la implementación del método run() se puede observar que utiliza varios pasos, uno de ellos es abstracto: el método processRequest() y otros actúan como ganchos (o métodos hook), como por ejemplo el método respond(). De esta forma una clase derivada sólo deberá sobreescribir el método processRequest().

Fragmento de Código Java de la Clase Servidor

```
public abstract class ServerHandler extends Thread{

    /**
     * Metodo abstracto para procesar la peticion
     * @param request corresponde a la peticion en forma de String
     * @return response que corresponde a la respuesta para el cliente
     */
    public abstract String processRequest(String request);


    /**
     * Metodo hook que envia la respuesta al cliente
     * @param response
     */
    protected void respond(String response){
         getOutput().println(response);
    }


    /**
     * Constructor del manejador de la peticion
     */
    public ServerHandler(){
    }

    /**
     * Codigo de ejecucion del hilo de atencion al cliente
     */
    @Override
    public void run(){
        try {
            createStreams();
            String request = readStream();
            if(!request.equals("")){
                processRequest(request);
            }
            closeStream();

        } catch (IOException ex) {
            Logger.getLogger(this.getClass().getName()).
            log(Level.SEVERE, "Eror al leer el flujo", ex);
        }
    }
    ...
    }

\end{tcolorbox}
```

Veamos un pequeño ejemplo, un servidor que hace eco sobre lo que se le solicita, es decir

devuelve la misma petición:

Haciendo un pequeño Servidor tipo Eco

```
package co.unicauca.strategyserver.infra;

/**
 *
 * @author ahurtado
 */
public class EchoServerExample{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        ServerSocketMultiThread server;
        server = new ServerSocketMultiThread(5000);
        ServerHandler echo = new ServerHandler(){
            @Override
            public String processRequest(String request) {
                return request;
            }
        };
        server.setServerHandler(echo);
        server.startServer();
    }
}
\end{tcolorbox}
```

3 El servidor de clientes de viaje

Este servidor instancia el servidor multihilo, asocia un objeto de la clase TravelAgencyHandler, fija aspectos de persistencia y da inicio al servidor con el método startServer().

La sobreescritura del método getLine() en EncryptedParagraph

```
public class TravelAgencyServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        ServerSocketMultiThread myServer =
                    new ServerSocketMultiThread(5001);
        TravelAgencyHandler myHandler = new TravelAgencyHandler();
        myHandler.setService(
            new CustomerService(new CustomerRepositoryImplArrays()));
        myServer.setServerHandler(myHandler);
        myServer.startServer();
    }
```

La clase TravelAgencyHandler sobreescribe el método processRequest() para procesar la petición del lado del servidor. Si el recurso solicitado es Customer entonces verifica el Action, si es get envía el mensnaje processGetCustomer() y si es post envía el mensaje processPostCustomer().

La sobreescritura del método getLine() en EncryptedParagraph

```
1
public class TravelAgencyHandler extends ServerHandler {
3   @Override
      public String processRequest(String requestJson) {
5         // Convertir la solicitud a objeto Protocol para poderlo procesar
          Gson gson = new Gson();
7         Protocol protocolRequest;
          protocolRequest = gson.fromJson(requestJson, Protocol.class);
9         String response="";
          switch (protocolRequest.getResource()) {
11            case "customer":
                  if (protocolRequest.getAction().equals("get")) {
13                    // Consultar un customer
                      response = processGetCustomer(protocolRequest);
15                }

17                if (protocolRequest.getAction().equals("post")) {
                      // Agregar un customer
19                    response = processPostCustomer(protocolRequest);

21                }
                  break;
23            }
          return response;
25        }
          ...
27    }
```

4 Servidor Web con HttServer, HttpHandler y HttpExchange

Ahora vamos a volver el servidor de la agencia de viajes, un servidor web haciendo uso de la librería com.sun.net.httpserver. Es muy similar a la solución brindada anteriormente sólo que su creación varía y el el servidor debe crearse a partir de las abstracciones: HttpServer(Servidor Web), HttpHandler(Manejador de la petición web) y HttpExchange (Petición/Respuesta).

La sobreescritura del método getLine() en EncryptedParagraph

```
public class WebServer {
    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(
        new InetSocketAddress(8003), 0);
        TravelAgencyWebHandler webhandler = new TravelAgencyWebHandler();
        webhandler.setService(new CustomerService(
            new CustomerRepositoryImplArrays()));
        server.createContext("/client", webhandler);
        // crea un ejecutor por defecto
        server.setExecutor(Executors.newCachedThreadPool());
        server.start();
        System.out.println("Servidor inicializado en el puerto 8003");
    }
}
```

Ahora la clase TravelAgencyWebHandler hereda de la clase HttpHandler y sobre escribe el método handle(). El interior luce igual que el anterior sólo que ahora debemos hacernos responsable de armar una respuesta siguiendo el protocolo http, por eso se define un método respond adecuado al protocolo. Tanto el método handle como el método respond trabaja con un objeto HttpExchange que permite leer información proveniente del cliente, así como escribirle la respuesta.

La sobreescritura del método getLine() en EncryptedParagraph

```
public class TravelAgencyWebHandler implements HttpHandler  {

    private CustomerService service;


    @Override
    public void handle(HttpExchange t) throws IOException {
            Scanner scanner = new Scanner(t.getRequestBody());
            String requestJson = scanner.nextLine();
            Gson gson = new Gson();
            System.out.println(requestJson);
            Parameters parameters = gson.fromJson(requestJson,
            Parameters.class);
            String response="";
            if (t.getRequestMethod().equals("GET"))
               response = processGetCustomer(parameters);
            if (t.getRequestMethod().equals("POST"))
                response = processPostCustomer(parameters);
            respond(response, t);
    }

    private void respond(String response,
                         HttpExchange t) throws IOException{
            t.sendResponseHeaders(200, response.getBytes().length);
            t.getResponseBody().write(response.getBytes());
            t.getResponseBody().close();

    }
```

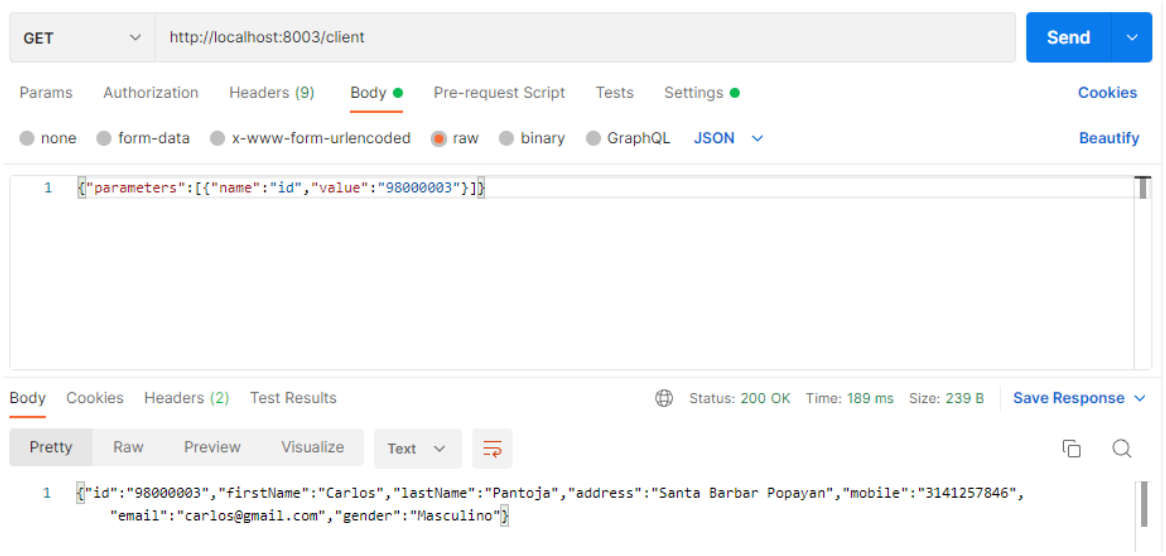
Para probar este nuevo servidor debe Procesa la petición de un cliente, se le debe pasar esta información a través de la herramiente postman como una fila (raw)
"parameters":["name":"id","value":"98000003"]

Esto es debido a que ya no manejamos la petición sino una lista de parámetros.

5 Servicios REST

Una API REST es una aplicación web en el lado del back-end. Tiene una serie de rutas (resources) y métodos que interactúan con la base de datos y hacen la parte lógica. Por ejemplo, un método que autentique un usuario, otro método que registre un usuario, otro método que liste productos, etc. Estos métodos pueden ser consumidos por cualquier tipo de cliente, ya sea web, móvil, de escritorio o cualquier otro tipo que se comunique a través del protocolo HTTP. Cada método es invocado por cada uno de los verbos de HTTP respectivos:

- GET (consulta y lectura)
- POST (crear)
- DELETE (eliminar)



- PUT (editar).

El back-end y los clientes se comunican enviando mensajes ya sea XML o JSON. JSON es más recomendable actualmente por ser un formato simple. La idea para entender, es que desarrollen el ejercicio de la siguiente práctica de laoratorio:

Laboratorio API Rest con Spring Boot

6 Tareas

1. Descargar, instalar y correr la clase EchoServerExample que viene en el proyecto strategy-server. Valor 0.5
2. Descargar, instalar y correr el AgencyTravelServer como servidor tcp/ip y el AgencyTravel-Client (infra.tcpip). Valor 0.5
3. Correr el Web Server(infra.web) que vienen en el mismo proyecto AgencyTravelServer. Usar un cliente postman para hacer la consulta, recuerden que el protocolo cambió y ahora sólo van los parámetros. Ver la consulta postman. Valor 0.5
4. Realizar el taller de la API Rest y probar las consultas a través de postman o un Jersey Client. Sigan el paso a paso del taller. Valor 1.5
5. Hacer una API Rest para el AgencyTravelServer. Probarla desde un Jersey Client o a través de posman. Valor Valor 2.0.

Deberán entregar un link de la solución que debe tener el código del útlimo punto y los pantallazos que evidencien la ejecución de las aplicaciones.

La fecha de entrega 21 de junio de 2022 a través del sitio del curso en Moodle.