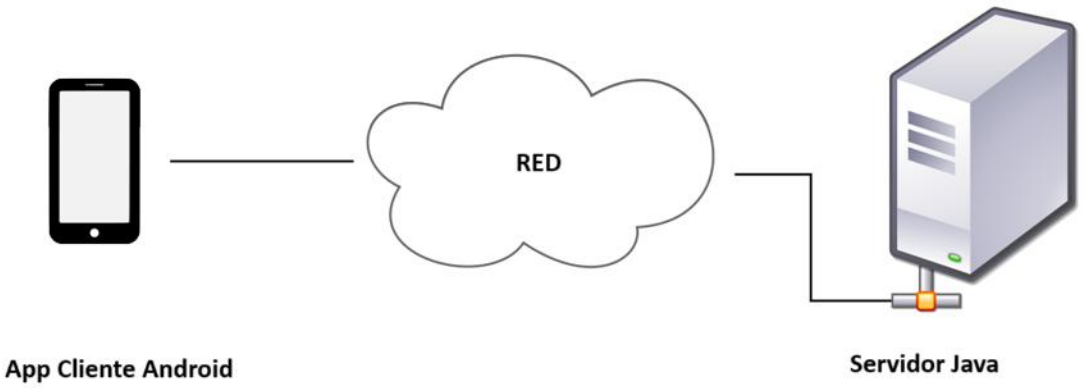
Socket.io

Socket.IO es una biblioteca basada en eventos para aplicaciones web en tiempo real. Permite la comunicación bidireccional entre clientes y servidores web. Consta de dos componentes: un cliente y un servidor. Socket.IO también es un protocolo, donde diferentes implementaciones compatibles del protocolo pueden comunicarse entre sí. Por ejemplo, hay implementaciones para Node.js, Deno (JavaScript), C++, Java, Python, Swift y por supuesto, Kotlin.

Socket.IO utiliza principalmente el protocolo WebSocket, que a su vez es un protocolo de comunicaciones que proporciona un canal bidireccional y simultáneo a través de una única conexión de tipo TCP.

Comunicación Cliente - Servidor

Vamos a ver un ejemplo simple que nos permita comunicar un cliente y un servidor mediante Socket.io. Aunque ambos utilizarán la misma biblioteca, verás que la implementación será diferente para ambos (aunque muy parecida). En el ejemplo, vamos a suponer que disponemos de una **App Cliente Android** que solicita al **Servidor Java** la información referente a unos alumnos. El esquema general de la aplicación será:



Primeros Pasos: Los Eventos

Antes de ponerse a picar código, primero hay que pararse a diseñar las operaciones vamos a realizar contra el servidor. En nuestro ejemplo vamos a suponer que nuestra **App Cliente Android** necesita obtener del **Servidor Java** la información de un alumno (login); el listado de todos los alumnos (selectAll); y comunicar que se ha deslogueado (logout). No nos pararemos mucho a pensar si estas operaciones son realmente adecuadas o no para completar este ejemplo, dado que están planteadas a modo de prueba.

Partiendo de esta base, podemos empezar a diseñar la lógica detrás de estas operaciones.

Login	
Nombre del evento:	"onLogin"
Parámetros enviados	userName
Parámetros devueltos:	Alumno
Propósito:	El servidor devuelve el alumno de Base de Datos cuyo userName coincide con el solicitado.

GetAll	
Nombre del evento:	"onGetAll"
Parámetros enviados	-
Parámetros devueltos:	Listado de Alumnos
Propósito:	El servidor devuelve todos los alumnos de la Base de Datos.

Logout	
Nombre del evento:	"onLogout"
Parámetros enviados	userName
Parámetros devueltos:	-
Propósito:	El servidor toma nota de que el cliente se ha deslogueado.

El apartado **nombre del evento** es importante. Socket.io utiliza ese nombre para distinguir entre diferentes solicitudes al servidor. Por tanto, utilizar el nombre del evento incorrecto puede llevar a errores.

El Cliente

Vamos a comenzar con la parte del **Servidor Java**. En Eclipse, creamos un **Proyecto Maven**. Le tendremos que indicar que el nivel de compilación sea Java 20 y el de ejecución sea el mismo. Esto se debe a que estaremos utilizando operaciones lambda que requieres una jre 1.8 o superior. A continuación, le añadimos las siguientes dependencias al proyecto en el **pom.xml** y actualizamos el proyecto. Esto nos dejará las librerías descargadas para su uso.

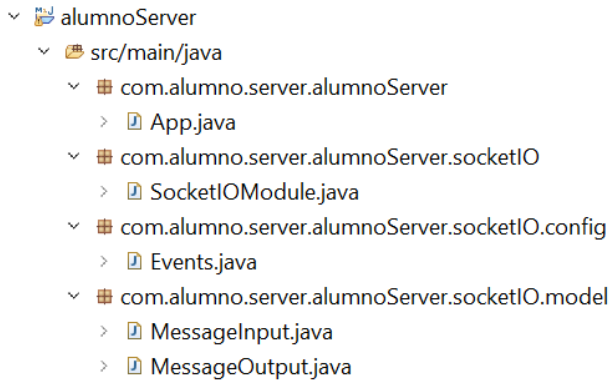
```
<!-- Socket.io dependencies -->

<dependency>
    <groupId>com.corundumstudio.socketio</groupId>
    <artifactId>netty-socketio</artifactId>
    <version>2.0.3</version>
</dependency>

<!-- GSON dependencies -->

<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.1</version>
</dependency>
```

Netty es una implementación open-source de socket-io; y gson es una librería de Google para trabajar con JSON.

Nota: en el código de ejemplo verás que hay una serie de dependencias adicionales para intentar evitar que se genere un warning molesto al arrancar el servidor. No son necesarias.

El código

Vamos a empezar a generar la estructura de clases. Generamos las siguientes clases distribuidas en los paquetes tal y como ves en la imagen.



Las clases a grandes rasgos sirven para lo siguiente:

- **App**: Contiene el main de la aplicación
- **SocketIOModule**: Contiene el código principal de comunicación
- **Events**: Listado de los eventos que escuchará y emitirá el servidor.
- **MessageInput** / **MessageOutput**: Lo que se envía y recibe el servidor.

Vamos a comenzar por la parte sencilla: los Mensajes.

PASO 1 - MessageInput y **MessageOutput** son las clases que definen lo que enviamos y recibimos en nuestro servidor. En nuestro caso es exactamente lo mismo: un texto plano (un JSON). Por tanto, ambas clases serán idénticas. Para simplificar las cosas, creamos una clase **AbstractMessage** con los campos necesarios de la que heredarán ambas.

Nosotros no nos tendremos que preocupar de convertir un JSON en un Objeto Java (o viceversa) porque esa conversión la hace automáticamente la librería netty-socket.io

```java
public abstract class AbstractMessage {

    private String message = null;

    public AbstractMessage(String message) {
        super();
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

```java
public class MessageInput extends AbstractMessage {

    public MessageInput(String message) {
        super(message);
    }
}
```

```java
public class MessageOutput extends AbstractMessage{

    public MessageOutput(String message) {
        super(message);
    }

}
```

PASO 2 – Events es una enumeración en la que tendremos que indicar los eventos que genera o responde nuestra app. En nuestro caso son tres, indicados más arriba, y dos respuestas al cliente. Por tanto:

```java
public enum Events {

    ON_LOGIN ("onLogin"),
    ON_GET_ALL ("onGetAll"),
    ON_LOGOUT ("onLogout"),
    ON_LOGIN_ANSWER ("onLoginAnswer"),
    ON_GET_ALL_ANSWER ("onGetAllAnswer");

    public final String value;

    private Events(String value) {
        this.value = value;
    }
}
```

PASO 3 – SocketIOModule es la clase que va a definir el comportamiento general de la comunicación entre cliente y servidor. Dado que la comunicación se base en eventos, en pocas palaras puede decirse que cada vez que el cliente 'hace algo' el servidor 'hace algo al respecto'. Si decide hacer A el server hace A. En nuestro caso, si el cliente decide hacer login el servidor ejecutará el método login ().

¿Vamos a ello? Podemos decir que hay dos tipos de eventos: los predefinidos y los que nosotros podemos definir. Los predefinidos son **onConnect** y **onDisconnect**. No hay que indicarlos en la clase Events porque siempre existen. Lo único que habrá que hacer es definir lo que queremos que haga el Server cuando los recibe. Los otros son, como habrás adivinado, los que hemos definido en la clase Events.

```
public class SocketIOModule {

    // The server
    private SocketIOServer server = null;

    public SocketIOModule(SocketIOServer server) {
        super();
        this.server = server;

        // Default events (for control the connection of clients)
        server.addConnectListener(onConnect());
        server.addDisconnectListener(onDisconnect());
    }

    private ConnectListener onConnect() {
        return (client -> {
            client.joinRoom("default-room");
            System.out.println("New connection, Client: " + client.getRemoteAddress());
        });
    }

    private DisconnectListener onDisconnect() {
        return (client -> {
            client.leaveRoom("default-room");
            System.out.println(client.getRemoteAddress() + " disconected from server");
        });
    }
}
```

El siguiente código lo que hace es definir el comportamiento del Sever cuando un cliente se conecta y se desconecta. Si te fijas, lo único que hacemos es añadir o eliminar al cliente de una **Room** y escribir una traza del suceso. Las Room (salas) son agrupaciones de clientes. Esto permite enviar y recibir mensajes entre ellos, de forma similar a una sala de chat. No vamos a usar esta característica en este proyecto.

Ahora, hay que añadir el comportamiento para los eventos que nosotros hemos definido. La forma de hacerlo es similar: en el constructor definimos los eventos y los métodos que se lanzan, y después definimos dichos métodos. Obviamente, no incluimos las respuestas.

Añadimos al constructor:

```
// Custom events
server.addEventListener(Events.ON_LOGIN.value, MessageInput.class, this.login());
server.addEventListener(Events.ON_GET_ALL.value, MessageInput.class, this.getAll());
server.addEventListener(Events.ON_LOGOUT.value, MessageInput.class, this.logout());
```

Y añadimos los métodos vacíos con una traza para hacer el seguimiento:

```
private DataListener<MessageInput> login() {
    return ((client, data, ackSender) -> {
        System.out.println("Client from " + client.getRemoteAddress() + " wants to login");
    });
}
```

Ahora podemos empezar a añadir el código del comportamiento de cada evento.

PASO 4 – Para el **Login** se ha decidido que el usuario va a enviarnos el **userName** y tendremos que devolverle el Alumno correspondiente desde Base de Datos. La App Cliente Android sabrá lo que tiene que hacer para averiguar si el usuario puede o no loguearse, pero eso no es cosa nuestra.

Para este ejemplo, que NO USA BBDD, vamos añadir en un paquete aparte la clase **Alumno**, que obviamente es una Entidad, y luego simularemos que la BBDD funciona correctamente.

```
public class Alumno {

    private int id = 0;
    private String name = null;
    private String surname = null;
    private String pass = null;
    private int edad = 0;
```

Vamos ahora al método login () y completamos el código. **Data** contiene la información que nos ha llegado del cliente, y es un **MessageInput**. De él podemos extraer el mensaje, que está en formato JSON. Para poder usarlo, hay que parsearlo mediante GSON. Obviamente, nos tocará averiguar exactamente qué nos está llegando desde el cliente para poder parsear correctamente. En nuestro caso asumimos que llega esto: {"userName": "patata"}.

Extraemos el userName y vamos a base de datos en busca de ese Alumno. Esto nos lo saltamos, y devolvemos cualquier cosa. Finalmente, convertimos Alumno en un JSON y se lo mandamos de vuelta al cliente dentro de un **MessageOutput**. Fíjate que el evento de vuelta es un **loginAnswer**. No lo implementamos de evento en el Server porque el Cliente NO NOS ENVÍA un 'answer' nunca.

Por cierto… para responder con errores simplemente se añade un if () y se envía un evento **errorAnswer** con otro JSON diferente. Por si te lo preguntabas.

El resultado es:

```
private DataListener<MessageInput> login() {
    return ((client, data, ackSender) -> {
        System.out.println("Client from " + client.getRemoteAddress() + " wants to login");

        // The JSON message from MessageInput
        String message = data.getMessage();

        // We parse the JSON into an JsonObject
        // The JSON should be something like this: {"userName": "patata"}
        Gson gson = new Gson();
        JsonObject jsonObject = gson.fromJson(message, JsonObject.class);
        String userName = jsonObject.get("userName").getAsString();

        // We access to database and...
        // Let's say it answers with this...
        Alumno alumno = new Alumno(0, userName, "potato", "pass", 20);

        // We parse the answer into JSON
        String answerMessage = gson.toJson(alumno);

        // ... and we send it back to the client inside a MessageOutput
        MessageOutput messageOutput = new MessageOutput(answerMessage);
        client.sendEvent(Events.ON_LOGIN_ANSWER.value, messageOutput);
    });
}
```

PASO 6 – Para el **Logout** la cosa es mucho más simple porque no vamos a hacer nada. Simplemente recogemos el userName y sacamos una traza. No hay que enviar respuestas siempre al cliente.

```
private DataListener<MessageInput> logout() {
    return ((client, data, ackSender) -> {
        // This time, we simply write the message in data
        System.out.println("Client from " + client.getRemoteAddress() + " wants to logout");

        // The JSON message from MessageInput
        String message = data.getMessage();

        // We parse the JSON into an JsonObject
        // The JSON should be something like this: {"userName": "patata"}
        Gson gson = new Gson();
        JsonObject jsonObject = gson.fromJson(message, JsonObject.class);
        String userName = jsonObject.get("userName").getAsString();

        // We do something on dataBase? ¯\_(ツ)_/¯

        System.out.println(userName + " loged out");
    });
}
```

PASO 6 – Finalmente, vamos a hacer el getAll. Que, en realidad, es básicamente lo mismo que hemos hecho en el Login, sólo que en este caso devolvemos un lista de Alumnos.

```
private DataListener<MessageInput> getAll() {
    return ((client, data, ackSender) -> {
        // This time, we simply write the message in data
        System.out.println("Client from " + client.getRemoteAddress() + " wants to getAll");

        // We access to database and... we get a bunch of people
        List<Alumno> alumnos = new ArrayList<Alumno>();
        alumnos.add(new Alumno(0, "patata", "potato", "pass", 20));
        alumnos.add(new Alumno(1, "patata2", "potato2", "pass2", 22));
        alumnos.add(new Alumno(2, "patata3", "potato3", "pass3", 23));

        // We parse the answer into JSON
        String answerMessage = new Gson().toJson(alumnos);

        // ... and we send it back to the client inside a MessageOutput
        MessageOutput messageOutput = new MessageOutput(answerMessage);
        client.sendEvent(Events.ON_GET_ALL_ANSWER.value, messageOutput);
    });
}
```

PASO 7 – Solo queda añadir el código necesario al main () para arrancar todo el tinglado. Añadimos a la clase los métodos de arranque y parada; y después en el main:

```
// Server control

public void start() {
    server.start();
}

public void stop() {
    server.stop();
}
```

```
public class App {

    private static final String HOST_NAME = "localhost";
    private static final int PORT = 3000;

    public static void main(String[] args) {

        // Server configuration
        Configuration config = new Configuration ();
        config.setHostname(HOST_NAME);
        config.setPort(PORT);

        // We start the server
        SocketIOServer server = new SocketIOServer(config);
        SocketIOModule module = new SocketIOModule(server);
        module.start();
    }
}
```

Y se acabó… ya puedes levantar el server y lanzarle peticiones. Por cierto… ignora este tipo de warnings de la consola.

```
App (1) [Java Application]  [pid: 7084]
log4j:WARN No appenders could be found for logger (io.netty.util.internal.logging.InternalLoggerFactory).
log4j:WARN Please initialize the log4j system properly.
```