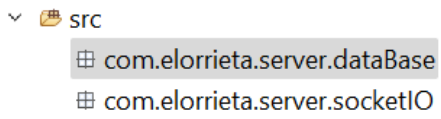
Socket.io - Trucos

Vamos a contar un par de ideas para facilitar la programación del Servidor de Socket.io para el Reto, o para cuando tienes que añadir más funcionalidades a servidor.

Estructura de clases y carpetas

Para ahorrarnos problemas, recomendable que sigas la estructura presentada en los ejemplos anteriores. En concreto, lo ideal sería que tuvieses por separado la parte de comunicación del socket.io con la base de datos.



De esta manera, varios programadores pueden trabajar simultáneamente sobre partes diferentes del código. A demás, esto favorece aislar y arreglar los posibles errores.

Claridad

A nada que tengas seis (o más) mensajes diferentes, es fácil perderse si colocáis todo el código en la clase que gestiona los eventos (en los ejemplos, **SocketIOModule**). Lo más aconsejable es delegar la responsabilidad de tratar los eventos a otras clases. Mira el código siguiente.

```
public ExampleModule(SocketIOServer server) {
    super();

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

    // Custom events
    server.addEventListener(Events.ON_GET_ALL_CLIENT_REGISTERED_ROOMS.value, JSONMensaje.class,
            this.onGetAllCientRegisteredRooms());
}

private DataListener<JSONMensaje> onGetAllCientRegisteredRooms() {
    return ((client, data, ackSender) -> {
        SocketIOManager.getInstance().onGetAllCientRegisteredRooms(client, data);
    });
}
```

Al recibir el evento ON_GET_ALL_CLIENT_REGISTERED_ROOMS llamamos al método **onGetAllCientRegisteredRooms ()**. Este método en lugar de procesar el mensaje, se limita a llamar a la clase **SocketIOManager**. Es esta clase la que se encarga de procesar el mensaje. Así conseguimos que la clase importante nos quede limpia y legible.

```
public class SocketIOManager {

    private static final SocketIOManager instance = null;

    private ClientManager clientManager = null;

    /**
     * Returns the sole instance of SocketIOManager
     *
     * @return The sole SocketIOManager instance
     * @throws RuntimeException
     */
    public static SocketIOManager getInstance() throws RuntimeException {
        return instance == null ? new SocketIOManager() : instance;
    }
```

En el ejemplo, la clase **SocketIOManager** utiliza patrón singleton para asegurarnos de que siempre está disponible para atender las peticiones del cliente. No es obligatorio hacerlo. A demás, si quieres dividir el código en más clases para facilitarte la gestión de los mensajes, mejor. Por ejemplo: **SocketIOManagerLogin**, **SocketIOManagerMessages**, etc.

Excepciones

Hacer un try con muchos catch disminuye la legibilidad del código. Como tengas una clase con docenas de funciones y docenas de try-catch, entonces te vas a marear buscando las cosas. Hay una forma sencilla de solucionar esto. En vez de hacer una sola función enorme, la dividimos en dos.

```
public void onGetAllCientRegisteredRooms(SocketIOClient client, JSONMensaje data) {
    System.out.println("Client " + client.getRemoteAddress() + " request all rooms he is registered in at DDBB");
    try {
        processGetAllCientRegisteredRooms(client, data);
    } catch (JsonSyntaxException e) {
        client.sendEvent(Events.ON_GET_ALL_CLIENT_REGISTERED_ROOMS_ERROR.value,
                new JSONMensaje(new Gson().toJson("Received JSON error")));
    } catch (Exception e) {
        client.sendEvent(Events.ON_GET_ALL_CLIENT_REGISTERED_ROOMS_ERROR.value,
                new JSONMensaje(new Gson().toJson("Server Error")));
    }
    System.out.println("Client " + client.getRemoteAddress() + " end request");
}
```

Hacemos una función que básicamente tiene el try-catch y una llamada a otra función que será que se encarga de procesar el evento. De esta forma, la primera función se encarga solamente de procesar las excepciones y la otra, de trabajar con los JSON.

```
private void processGetAllCientRegisteredRooms(SocketIOClient client, JSONMensaje data) {
    String clientID = "";
    List <Room> rooms = null;

    // Obtenemos clientID de data..

    // Accedemos a la BBDD...

    if (ifSuccess()) {
        client.sendEvent(Events.ON_GET_ALL_CLIENT_REGISTERED_ROOMS_RESPONSE.value,
                new JSONMensaje(new Gson().toJson(rooms)));
    } else {
        client.sendEvent(Events.ON_GET_ALL_CLIENT_REGISTERED_ROOMS_ERROR.value,
                new JSONMensaje(new Gson().toJson("No hay rooms")));
    }
}
```

Sí… esta segunda función también la puedes meter en otra clase aparate. Pero ya me parece excesivo…