universidad de león

Escuela de Ingenierías Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

TÍTULO DEL TRABAJO

TITLE OF THE WORK

Autor: Samuel Castrillo Domínguez
Tutor: Eva María Cuervo Fernández

Junio, 2023

UNIVERSIDAD DE LEÓN

Escuela de Ingenierías Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

ALUMNO: Samuel Castrillo Domínguez

TUTOR: Eva María Cuervo Fernández

TÍTULO: Título del trabajo

TITLE: Title of the work

CONVOCATORIA: Junio, 2023

RESUMEN:
El resumen reflejará las ideas principales de cada una de las partes del trabajo, pudiendo incluir un avance de los resultados obtenidos. Constará de un único párrafo y se recomienda una longitud no superior a 300 palabras. En cualquier caso, no deberá superar una página de longitud.

ABSTRACT:
Abstract will reflect the main ideas of each part of the work, including an advance of the results obtained. It will consist of a single paragraph and it is recommended a length not superior to 300 words. In any case, it should not exceed a page of length.

Palabras clave: Lorem, ipsum, dolor, sit, amet.

Firma del alumno:	**VºBº Tutor/es:**

Índice de contenidos

Índice de figuras

Índice de cuadros y tablas

Índice de bloques de código

Índice de diagramas UML

1. Introducción

2. Contenido

2.1. PATRONES DE DISEÑO

Un patrón de diseño es una solución que se puede aplicar a diferentes contextos y que se puede reutilizar en diferentes proyectos.
Hay varias categorías de patrones de diseño, cada una con una finalidad diferente[1]:

- **Patrones de creación**: se utilizan para crear objetos de una forma flexible y reutilizando código existente.
- **Patrones estructurales**: se utilizan para convertir clases y objetos en estructuras más complejas.
- **Patrones de comportamiento**: se utilizan para definir la interacción entre objetos.

En este proyecto se han utilizado varios patrones de diseño, para permitir una mejor escalabilidad, mantenibilidad y reutilización del código. A continuación se detalla cierta información de los patrones utilizados:

- Definición / categoría.
- Explicación de cómo se ha implementado en el proyecto.
- Diagrama UML.
- Justificación de su uso en la aplicación.

2.1.1. BUILDER

Es un patrón de **creación** que permite instanciar objetos complejos de una forma sencilla. Se ha creado una interfaz (`Builder`) genérica para su reutilización en caso de ser necesaria para cualquier otra clase. Esta interfaz define el método `build()` que devolverá la instancia de un objeto con los datos establecidos previamente. El diagrama UML del patrón implementado es el siguiente:

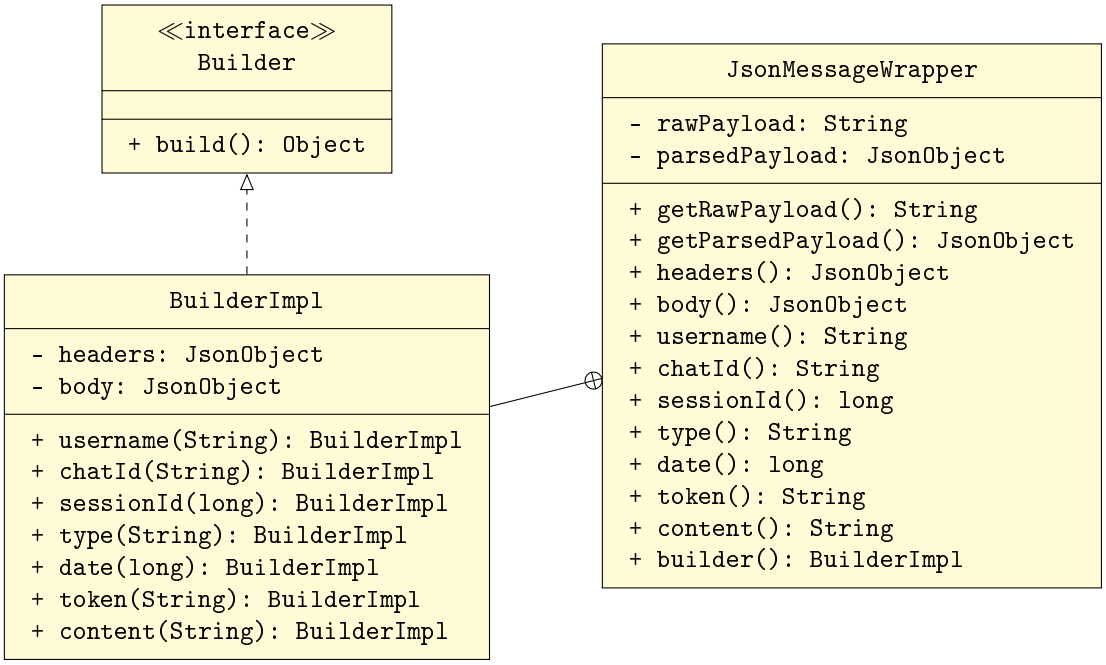

Diagrama UML 2.1: Patrón Builder empleado en la aplicación.

El patrón builder se ha usado en la aplicación para simplificar la creación de objetos de tipo `JsonMessageWrapper`, por el momento. Esta clase es la encargada de encapsular los mensajes que se envían a través de la red en formato JSON.

2.1.2. SINGLETON

También es un patrón de **creación** y se emplea para garantizar que una clase concreta tenga una única instancia y proporciona un punto de acceso global a ella [2]. En el contexto de esta aplicación, se utiliza en ciertas clases de utilidad y en las clases que asocian rutas a un método HTTP (por ejemplo `/login` con el método POST). Estas últimas son clases internas de `Routes.java` y los nombres dependen del método HTTP que se debe utilizar para realizar una petición a una ruta específica.

Cuadro 2.1: Relación entre método HTTP y ruta.

Método HTTP	Clase de la ruta
GET	`GetRoute`
POST	`PostRoute`
PUT	`PutRoute`
DELETE	`DeleteRoute`

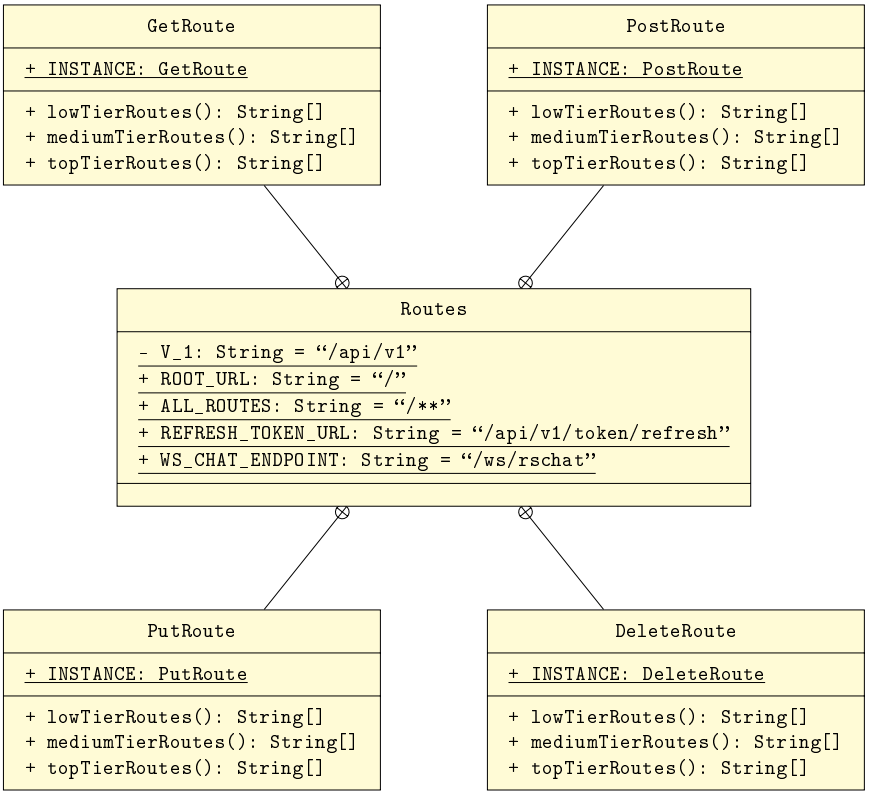

Diagrama UML 2.2: Patrón Singleton empleado en la aplicación.

Se han utilizado diferentes formas de acceso a la instancia de las clases. En el caso de las clases que asocian rutas a métodos HTTP, el modificador de acceso a la instancia es público. En otros casos, se provee un método estático para obtener la instancia de la clase, como se muestra en el siguiente ejemplo:

```
public class Routes {
    private Routes() {}
    ...
    public static class GetRoute {
        public static final GetRoute INSTANCE = new GetRoute();

        private GetRoute() {}

        public static final String USERS_URL = V_1 + "/users";

        /* Array containing the routes allowed by the low tier user */
        public String[] lowTierRoutes() {...}

        /* Array containing the routes allowed by the medium tier user */
        public String[] mediumTierRoutes() {...}

        /* Array containing the routes allowed by the top tier user */
        public String[] topTierRoutes() {...}
    }
    ...
}
```

Código 2.1: Rutas utilizadas para simplificar el proceso de autorización de los usuarios.

Todo: Quitar el código y dejar solo el diagrama

2.1.3. STRATEGY

Este patrón de **comportamiento** se utiliza para encapsular un algoritmo dentro de una clase, de forma que pueda ser intercambiado por otro algoritmo en tiempo de ejecución. En la aplicación, se emplea para encapsular el proceso de manejo de los mensajes que se envían entre los usuarios. Existen varias clases que se encargan de realizarlo, por lo que todas ellas implementan la interfaz `MessageStrategy`, que define el método `handle(...)`, encargado de realizar el procesamiento del mensaje y recibe 3 parámetros:

- El mensaje que se debe manejar.
- La lista con todos los chats para determinar al que se debe enviar el mensaje.
- Otros datos que pueden ser necesarios para el manejo del mensaje. Dependiendo de la clase, este parámetro puede contener más o menos datos.

La relación entre el tipo de mensaje y las clases encargadas de procesarlo, que implementan esta interfaz, son las siguientes:

Cuadro 2.2: Relación Mensaje - Estrategia

Mensaje	**Clase de la estrategia**
`USER_JOINED`	`UserJoinedStrategy`
`USER_LEFT`	`UserLeftStrategy`
`TEXT_MESSAGE`	`TextMessageStrategy`
`IMAGE_MESSAGE`	`ImageMessageStrategy`
`AUDIO_MESSAGE`	`AudioMessageStrategy`
`VIDEO_MESSAGE`	`VideoMessageStrategy`
`ACTIVE_USERS_MESSAGE`	`ActiveUsersStrategy`
`GET_HISTORY_MESSAGE`	`GetHistoryStrategy`
`ERROR_MESSAGE`	`ErrorMessageStrategy`
`PING_MESSAGE`	`PingStrategy`

A continuación se muestra el diagrama UML de la interfaz `MessageStrategy`:

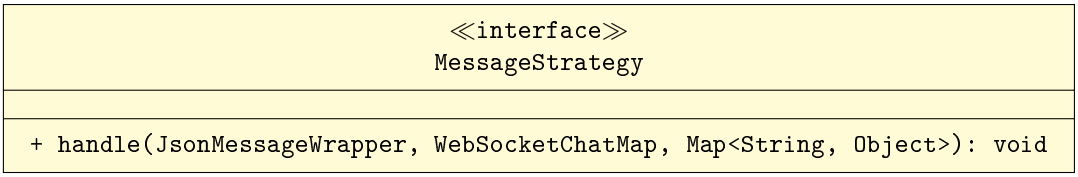

Diagrama UML 2.3: Interfaz MessageStrategy.

Este patrón se ha utilizado para simplificar el manejo de los mensajes recibidos en el servidor y para que sea más fácil de extender. En el caso de que se desee agregar un nuevo tipo de mensaje, se debe crear una nueva clase que implemente la interfaz `MessageStrategy` y agregarla a la lista de estrategias que se encuentran en la clase `WebSocketHandler`. Esta clase es la encargada de determinar qué estrategia se debe utilizar para manejar el mensaje. Para esto, se utiliza el método `decideStrategy(receivedMessageType: WSMessage)` que recibe como parámetro el tipo mensaje y devuelve la estrategia que se debe utilizar para manejar el mensaje.

2.2. PROCESAMIENTO DE LOS MENSAJES

Como hemos visto en la sección anterior, los mensajes que se reciben en el servidor pueden ser de diferentes tipos.

2.3. CICLO DE VIDA DE LA CONEXIÓN DE USUARIOS

Cuando un usuario accede a un chat de la aplicación, se inicia una conexión entre el cliente y el servidor a través del protocolo de comunicación bidireccional **WebSocket** [3]. Esta conexión se mantiene abierta mientras el usuario esté en el chat, y se cierra cuando el usuario abandona el chat. La secuencia de eventos que ocurren durante la conexión de un usuario al chat es la siguiente

2.3.1. FRONTEND

1. Se realiza una solicitud de conexión WebSocket al servidor.
2. Se realiza una petición HTTP para comprobar que el usuario puede acceder al chat. Esto se realiza para que, en caso de que el usuario introduzca la URL de un chat al que no tiene acceso, se le redirija a la página de inicio de la aplicación.
3. Si se confirma que el usuario **puede acceder** al chat:
    3.1. Se establece la conexión WebSocket.
    3.2. Se envía un mensaje de tipo `USER_JOINED` al servidor.
    3.3. Se consultan los últimos 65 mensajes del historial de mensajes del chat con el mensaje de tipo `GET_HISTORY_MESSAGE`.
    3.4. Se realiza una petición de la lista de usuarios activos con un mensaje de tipo `ACTIVE_USERS_MESSAGE`.
    3.5. Se configura un temporizador para mandar un mensaje de tipo `PING_MESSAGE` cada 30 segundos. Esto se realiza para que el servidor no cierre la conexión por inactividad.
4. Si el usuario **no puede acceder** al chat:
    4.1. Se cierra la conexión WebSocket.
    4.2. Se redirige al usuario a la página principal.

2.3.2. BACKEND

Cuando comienza la ejecución del programa, se indica a Spring Boot que el manejador de mensajes a través de WebSocket del servidor es una instancia de la clase `WebSocketHandler`, en la ruta '/ws/rschat'. Al instanciar esta clase, se crea un objeto **chatMap** de tipo `WebSocketChatMap`, que contiene el atributo `chats` (ver Diagrama UML 2.4). Es una tabla de dispersión que contiene los chats activos en la aplicación. Cada entrada asocia a una cadena de texto (identificador del chat) la instancia de un objeto de tipo `Chat`. El proceso de **conexión** al servidor sigue el siguiente flujo de eventos:

1. Se establece la conexión WebSocket con el usuario. Esto ocurre de forma transparente al programador debido a que la implementación se realiza en el framework de Spring Boot.

2. Se recibe el mensaje `USER_JOINED` del cliente y se crea un objeto de tipo `WSClient` (formado por la instancia de `WebSocketSession` y el `WSClientID` del usuario). Este nuevo objeto se añade a la lista de usuarios conectados al chat.

    2.1. Si el usuario es el primero que se ha conectado al chat, se crea uno nuevo, guardándose en la lista de chats.

    2.2. Si no, se añade al chat correspondiente de la lista de chats.

3. Se recibe el mensaje `GET_HISTORY_MESSAGE` del cliente y se envían como respuesta los últimos 65 mensajes del historial de mensajes del chat.

4. Se recibe el mensaje `ACTIVE_USERS_MESSAGE` del cliente y se devuelve la lista con los usuarios activos en el chat.

Y el proceso de **desconexión** se realiza como sigue:

1. Se recibe el mensaje `USER_LEFT` del cliente.

2. Se informa al resto de los usuarios del chat de la desconexión del usuario.

3. Al eliminar un usuario del chat se pueden dar 2 casos:

3.1. Si el usuario es el último que se ha desconectado del chat, se elimina el chat de la tabla de dispersión `chats`. Cuando esto ocurre, el historial de mensajes del chat que se haya registrado desde que se inició, se envía al almacenamiento en la nube.

3.2. Si no, se elimina el usuario de la lista de usuarios del chat.

4. Se cierra la conexión WebSocket con el usuario, de forma transparente al programador (al igual que la conexión).

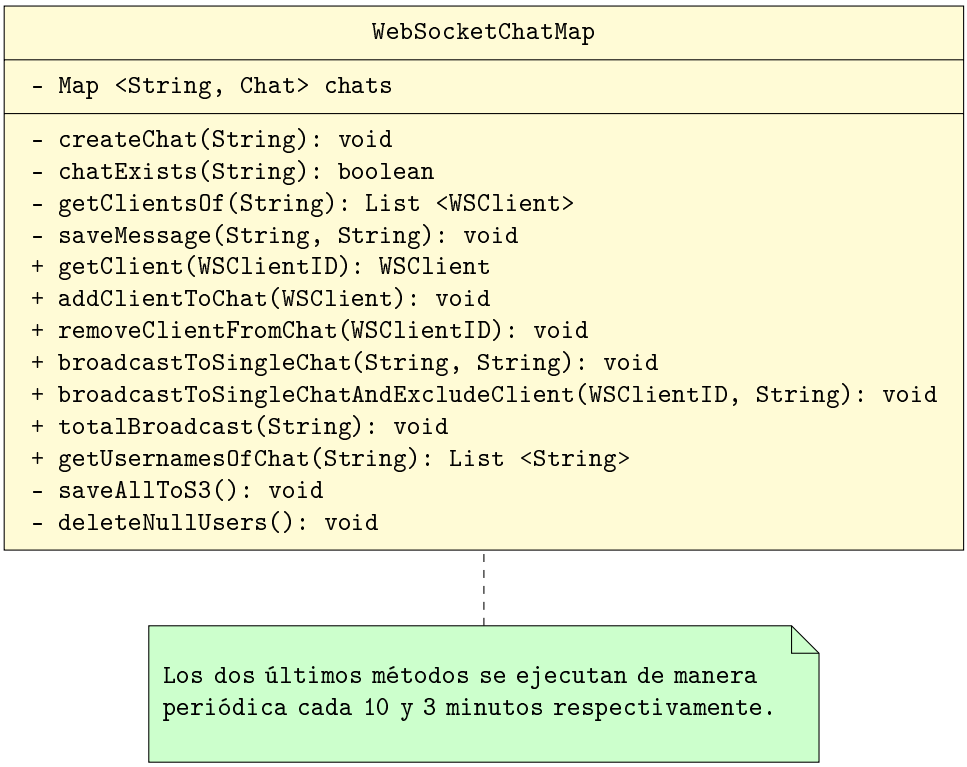

Diagrama UML 2.4: Clase `Chat` para almacenar los usuarios activos

Bibliografía

[1] R. Guru, "Classification of patterns."

[2] V. Sarcar, *Java Design Patterns: A Hands-On Experience with Real-World Examples*. Apress, 2018.

[3] I. Fette and A. Melnikov, "The websocket protocol," 12 2011.