
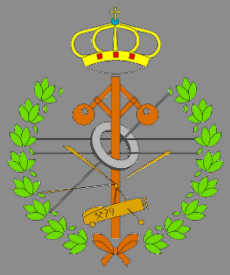

Escuela de Ingenierías
Industrial, Informática y
Aeroespacial

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:

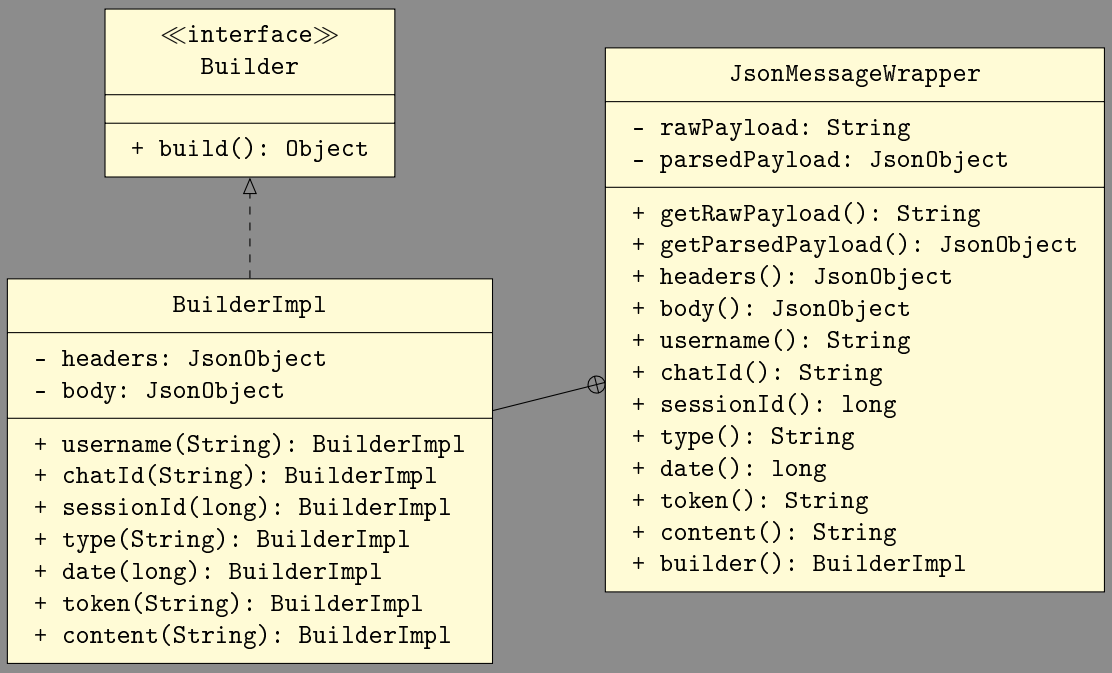


Diagrama UML 2.1: Patrón Builder empleado en la aplicación.

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` |

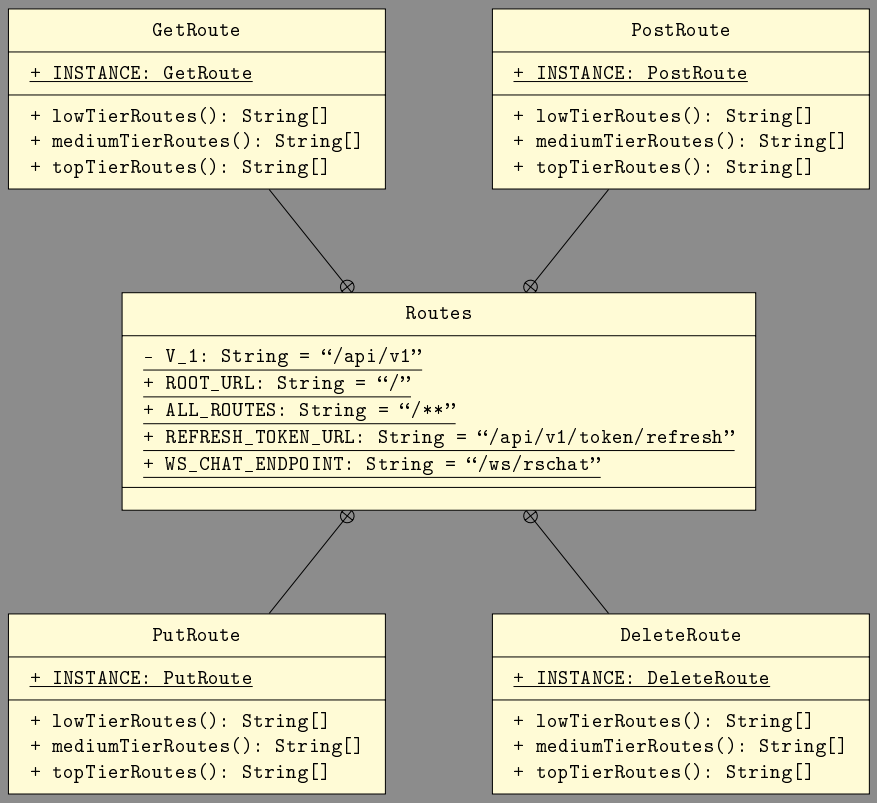


Diagrama UML 2.2: Patrón Singleton empleado en la aplicación.

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.

Las clases que implementan esta interfaz son las siguientes:

- `ActiveUsersStrategy`
- `ErrorMessageStrategy`
- `GenericMessageStrategy` - De la cual heredan:
  - `AudioMessageStrategy`
  - `ImageMessageStrategy`
  - `TextMessageStrategy`
  - `VideoMessageStrategy`
- `GetHistoryStrategy`
- `PingStrategy`
- `UserJoinedStrategy`
- `UserLeftStrategy`

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

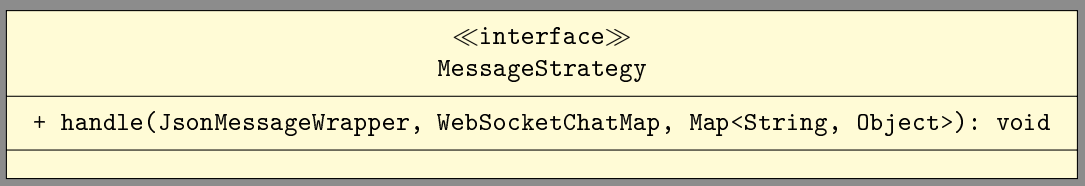


Diagrama UML 2.3: Interfaz MessageStrategy.

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.

# Bibliografía

[1] R. Guru, “Classification of patterns.”

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