

UNIVERSIDAD DE OVIEDO



Universidad de Oviedo

Universidá d'Uviéu

University of Oviedo



Escuela de
Ingeniería
Informática

Universidad de Oviedo

ESCUELA DE INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

Diseño y desarrollo de biblioteca WebSocket para APIs HTTP

TUTOR: Ángel José Riesgo Martínez

AUTOR: Luis Miguel Gómez del Cueto

Agradecimientos

Quiero aprovechar estas líneas para dar las gracias a todas las personas que, de una forma u otra, han estado presentes durante estos años.

En primer lugar, a mi familia, por darme la libertad de elegir mi camino y apoyarme, confiando siempre en mis decisiones. También a mis amigos más cercanos, con quienes he compartido tantos momentos de descanso, risas y desconexión, que han sido igual de importantes que el estudio.

A mis compañeros y compañeras, por estar dentro y fuera del aula, ayudándome en trabajos, resolviendo dudas y haciendo mucho más llevadera la experiencia universitaria. Y, por supuesto, a todos los profesores que, de una forma u otra, han contribuido a que hoy tenga los conocimientos y herramientas que tengo.

A todos, gracias.

Resumen

El presente Trabajo de Fin de Grado tiene como objetivo ofrecer una solución flexible para **mejorar la comunicación en tiempo real entre aplicaciones web y servidores**. En muchos entornos (como sistemas de monitorización, plataformas colaborativas o aplicaciones financieras) es fundamental que la información se actualice al instante de forma eficaz, sin necesidad de recargar la página o realizar peticiones continuas al servidor.

Para dar respuesta a esta necesidad, se ha desarrollado una **biblioteca reutilizable** que permite ampliar de forma sencilla cualquier API REST (interfaz que permite la comunicación entre sistemas a través del protocolo HTTP) incorporando un canal de comunicación en tiempo real basado en WebSocket (protocolo que permite intercambiar datos de manera bidireccional y persistente entre cliente y servidor). Esta solución evita las limitaciones de técnicas tradicionales como el *polling* o *long polling* (consultas reiteradas), mejorando significativamente la eficiencia de la comunicación.

A diferencia de una implementación *ad hoc* centrada en un único caso de uso, este proyecto propone una **infraestructura genérica y modular**, orientada a ser fácilmente integrada en distintos contextos. Su diseño facilita que cualquier API REST pueda ser adaptada, sin modificar su lógica interna, para emitir notificaciones en tiempo real cuando cambian determinados estados o recursos.

Para validar el funcionamiento de la biblioteca, se ha implementado un adaptador específico para la API pública de **Yahoo Finance**, que proporciona cotizaciones bursátiles en tiempo real. No obstante, la biblioteca está diseñada para funcionar con cualquier otra fuente de datos externa compatible.

Desde el punto de vista técnico, el servidor se ha desarrollado en **Node.js**, utilizando **WebSocket** nativo, con una arquitectura orientada a objetos que divide la lógica en diferentes módulos bien definidos. El diseño aplica varios **patrones de diseño clásicos** (como *Adapter*, *Observer*, *Singleton*, *Proxy* y *Abstract Factory*) con el objetivo de mejorar la claridad, la extensibilidad y la reutilización del código. Estos patrones están documentados e ilustrados mediante diagramas UML en la memoria del proyecto.

El sistema permite que múltiples clientes se conecten al servidor WebSocket, se suscriban dinámicamente a distintos “estados” y reciban notificaciones automáticas solo cuando esos valores cambian. Esta funcionalidad es útil en numerosos ámbitos como la monitorización financiera, sistemas de alerta, interfaces colaborativas o aplicaciones de datos en vivo.

El código se ha publicado en GitHub y gestionado mediante el sistema de control de versiones **Git**, en línea con las buenas prácticas de desarrollo de software moderno. Aunque no está destinado a un cliente real, el proyecto ha sido concebido desde el principio con una **visión profesional y práctica**, haciendo hincapié en la calidad del diseño, la documentación y la posibilidad de reutilización futura.

En resumen, este trabajo ofrece una solución técnica flexible y actual para dotar de capacidad de actualización en tiempo real a APIs REST tradicionales, aportando tanto valor práctico como fundamentos teóricos relevantes en el contexto del desarrollo de software moderno.

Palabras Clave

- WebSocket
- API REST
- Comunicación en tiempo real
- Node.js
- Arquitectura cliente-servidor
- Biblioteca reutilizable
- Diseño modular
- Patrones de diseño
 - Observer
 - Adapter
 - Singleton
 - Abstract Factory
 - Proxy
 - Flyweight
 - Iterator
 - Facade
- Yahoo Finance
- JavaScript
- Chart.js
- Tailwind CSS
- Actualización dinámica de datos
- Suscripción de eventos
- Interoperabilidad
- Desarrollo web
- Git y GitHub
- Ingeniería de software
- Pruebas de usabilidad
- Visualización de datos
- Proxy inverso
- Arquitectura orientada a objetos
- Escalabilidad
- Latencia

Abstract

The aim of this Final Degree Project is to offer a flexible solution to **improve real-time communication between web applications and servers**. In many environments (such as monitoring systems, collaborative platforms, or financial applications) it is essential that information is updated instantly effectively, without the need to reload the page or make continuous requests to the server.

To meet this need, a **reusable library** has been developed that allows any REST API (an interface that enables communication between systems via the HTTP protocol) to be easily extended by incorporating a real-time communication channel based on WebSocket (a protocol that enables bidirectional and persistent data exchange between client and server). This solution avoids the limitations of traditional techniques such as polling or long polling (repeated queries), significantly improving communication efficiency.

Unlike an ad hoc implementation focused on a single use case, this project proposes a **generic and modular infrastructure**, oriented to be easily integrated in different contexts. Its design facilitates that any REST API can be adapted, without modifying its internal logic, to issue real-time notifications when certain states or resources change.

To validate the operation of the library, a specific adapter has been implemented for the **Yahoo Finance** public API, which provides real-time stock quotes. However, the library is designed to work with any other supported external data source.

From the technical point of view, the server has been developed in **Node.js**, using native **WebSocket**, with an object-oriented architecture that divides the logic into different well-defined modules. The design applies several **classic design patterns** - such as Adapter, Observer, Singleton, Proxy and Abstract Factory - with the goal of improving code clarity, extensibility and reusability. These patterns are documented and illustrated by UML diagrams in the project memory.

The system allows multiple clients to connect to the WebSocket server, dynamically subscribe to different "states" and receive automatic notifications only when those values change. This functionality is useful in numerous domains such as financial monitoring, alerting systems, collaborative interfaces or live data applications.

The code has been published on GitHub and managed using the **Git** version control system, in line with modern software development best practices. Although not intended for a real customer, the project has been conceived from the beginning with a **professional and practical vision**, emphasizing design quality, documentation and future reusability.

In summary, this work offers a flexible and current technical solution to provide real-time update capability to traditional REST APIs, providing both practical value and relevant theoretical foundations in the context of modern software development.

Keywords

- WebSocket
- REST API
- Real-time communication
- Node.js
- Client-server architecture
- Reusable library
- Modular design
- Design patterns
 - Observer
 - Adapter
 - Singleton
 - Abstract Factory
 - Proxy
 - Flyweight
 - Iterator
 - Facade
- Yahoo Finance
- JavaScript
- Chart.js
- Tailwind CSS
- Dynamic data updates
- Event subscription
- Interoperability
- Web development
- Git and GitHub
- Software engineering
- Usability testing
- Data visualization
- Reverse proxy
- Object-oriented architecture
- Scalability
- Latency

Declaración de Originalidad

Yo, **Luis Miguel Gómez del Cueto**, estudiante de la **Escuela de Ingeniería Informática** de la Universidad de Oviedo, declaro que el presente Trabajo de Fin de Grado titulado:

“Diseño y desarrollo de biblioteca WebSocket para APIs HTTP”

es una obra original, realizada por mí, y que:

- No ha sido copiada ni total ni parcialmente de trabajos ajenos.
- No ha sido presentada previamente para la obtención de ningún otro título o certificación académica.
- Todas las fuentes y recursos utilizados han sido debidamente citados y referenciados.

Índice General

1	PRESENTACIÓN	13
2	ASPECTOS TEÓRICOS	13
2.1	PATRONES DE DISEÑO.....	14
2.1.1	<i>Introducción.....</i>	14
2.1.2	<i>Observer</i>	15
2.1.3	<i>Adapter.....</i>	16
2.1.4	<i>Singleton.....</i>	16
2.1.5	<i>Abstract Factory</i>	17
2.1.6	<i>Proxy.....</i>	18
2.1.7	<i>Flyweight.....</i>	18
2.1.8	<i>Iterator</i>	19
2.1.9	<i>Facade</i>	20
2.2	CONCEPTOS CLAVES.....	21
2.2.1	<i>Comunicación en tiempo real en la Web.....</i>	21
2.2.2	<i>Patrón Observer aplicado a WebSocket</i>	21
2.2.3	<i>Concepto de Proxy y su aplicación en el diseño.....</i>	21
2.3	HERRAMIENTAS DE DESARROLLO	23
2.3.1	<i>Protocolo WebSocket</i>	23
2.3.2	<i>Node.js y Express</i>	23
2.3.3	<i>Herramientas del Cliente Web.....</i>	23
2.4	METODOLOGÍA Y NOTACIONES.....	24
2.4.1	<i>Proceso Unificado de Desarrollo (RUP).....</i>	24
2.4.2	<i>Notación UML.....</i>	24
2.4.3	<i>Justificación y aplicación práctica</i>	24
3	MEMORIA DEL PROYECTO	25
3.1	HOJA DE IDENTIFICACIÓN.....	25
3.2	INTRODUCCIÓN	26
3.3	OBJETO	26
3.4	ANTECEDENTES	28
3.5	DESCRIPCIÓN DE LA SITUACIÓN ACTUAL	28
3.6	NORMAS Y REFERENCIAS	30
3.6.1	<i>Disposiciones legales y normas aplicadas</i>	30
3.6.2	<i>Métodos, Herramientas, Modelos, Métricas y Prototipos.....</i>	30
3.6.3	<i>Mecanismos de control de calidad aplicados durante la redacción del proyecto</i>	32
3.7	DEFINICIONES Y ABREVIATURAS	33
3.7.1	<i>Definiciones</i>	33
3.7.2	<i>Abreviaturas</i>	33
3.8	REQUISITOS INICIALES	34
3.8.1	<i>Requisitos funcionales</i>	34
3.8.2	<i>Requisitos no funcionales</i>	34
3.9	ALCANCE.....	35
3.10	HIPÓTESIS Y RESTRICCIONES	36
3.10.1	<i>Hipótesis.....</i>	36
3.10.2	<i>Restricciones</i>	36
3.11	ESTUDIOS DE ALTERNATIVAS Y VIABILIDAD.....	37

3.11.2	<i>Conclusión del análisis de viabilidad</i>	39
3.12	DESCRIPCIÓN DE LA SOLUCIÓN PROPUESTA	40
3.13	ANÁLISIS DE RIESGOS	41
3.14	ORGANIZACIÓN Y GESTIÓN DEL PROYECTO	42
3.15	PLANIFICACIÓN TEMPORAL	43
3.16	RESUMEN DEL PRESUPUESTO.....	44
4	ANÁLISIS Y DISEÑO DEL SISTEMA	46
4.1	DOCUMENTACIÓN DE ENTRADA.....	46
4.2	ANÁLISIS.....	47
4.2.1	<i>Identificación de Subsistemas</i>	47
4.2.2	<i>Interfaces entre subsistemas</i>	47
4.2.3	<i>Casos de uso principales</i>	48
4.2.4	<i>Ánálisis de Clases en la Fase de Análisis</i>	50
4.2.5	<i>Ánálisis de Interfaces de Usuario</i>	54
4.2.6	<i>Especificación del Plan de Pruebas</i>	56
4.3	DISEÑO	58
4.3.1	<i>Diseño de Casos de Uso Reales</i>	58
4.3.2	<i>Diseño de Clases</i>	60
4.3.3	<i>Arquitectura del Sistema</i>	64
4.3.4	<i>Diseño de la Interfaz</i>	67
4.3.5	<i>Especificación Técnica del Plan de Pruebas</i>	70
4.4	ESTIMACIÓN DEL TAMAÑO Y ESFUERZO	74
4.4.1	<i>Métricas seleccionadas</i>	74
4.5	PLANES DE GESTIÓN DEL PROYECTO.....	76
4.6	PLAN DE SEGURIDAD	77
4.6.1	<i>Objetivos de seguridad</i>	77
4.6.2	<i>Medidas organizativas</i>	78
4.6.3	<i>Medidas técnicas</i>	78
4.6.4	<i>Puntos críticos y normativa</i>	78
4.6.5	<i>Herramientas y buenas prácticas</i>	79
4.6.6	<i>Conclusión</i>	79
4.7	OTROS DOCUMENTOS QUE JUSTIFIQUEN Y ACLAREN CONCEPTOS EXPRESADOS EN EL PROYECTO	79
4.7.1	<i>Catálogo de componentes desarrollados</i>	79
4.7.2	<i>Listados de código y fragmentos destacados</i>	80
4.7.3	<i>Información en soporte digital</i>	80
4.7.4	<i>Referencias externas utilizadas</i>	80
5	ESPECIFICACIÓN DEL SISTEMA.....	81
5.1	ESPECIFICACIÓN DE REQUISITOS	81
5.1.1	<i>Requisitos funcionales</i>	81
5.1.2	<i>Requisitos no funcionales</i>	82
5.1.3	<i>Especificación de Casos de Uso</i>	82
5.1.4	<i>Ánálisis de Escenarios</i>	82
5.1.5	<i>Actores del Sistema</i>	84
6	PRESUPUESTO	84
6.1	CUADRO DE PRECIOS DE UNIDADES DE MEDIDA	84
6.2	COSTES DE UNIDADES LÓGICAS CON ENTIDAD PROPIA.....	85
6.3	VALORACIÓN ECONÓMICA GLOBAL.....	85
6.4	BASES PARA LA CONFECIÓN DEL PRESUPUESTO.....	86

6.5	PLANIFICACIÓN TEMPORAL RESUMIDA.....	86
7	ESTUDIOS CON ENTIDAD PROPIA.....	87
7.1	LEGISLACIÓN SOBRE SEGURIDAD Y PROTECCIÓN DE DATOS	87
7.2	LEGISLACIÓN SOBRE PROPIEDAD INTELECTUAL E INDUSTRIAL	87
7.3	PREVENCIÓN DE RIESGOS LABORALES	88
7.4	IMPACTO AMBIENTAL.....	88
8	IMPLEMENTACIÓN DEL SISTEMA	89
8.1	ESTÁNDARES Y NORMAS SEGUIDOS.....	89
8.2	LENGUAJE DE PROGRAMACIÓN.....	90
8.3	HERRAMIENTAS Y PROGRAMAS USADOS PARA EL DESARROLLO.....	91
8.3.1	<i>Visual Studio Code</i>	91
8.3.2	<i>Node.js</i>	91
8.3.3	<i>Navegador Web (Google Chrome)</i>	91
8.3.4	<i>Git y GitHub</i>	91
8.3.5	<i>Chart.js</i>	92
8.4	CREACIÓN DEL SISTEMA	92
8.4.1	<i>Problemas Encontrados</i>	93
8.4.2	<i>Descripción Detallada de las Clases</i>	94
9	DESARROLLO DE LAS PRUEBAS	98
9.1	PRUEBAS UNITARIAS	98
9.2	PRUEBAS DE INTEGRACIÓN Y DEL SISTEMA	99
9.3	PRUEBAS DE USABILIDAD	100
9.3.1	<i>Participantes</i>	100
9.3.2	<i>Resultados Cuestionario de Evaluación (Usuario Final)</i>	100
9.3.3	<i>Observaciones y Comentarios de Usuarios</i>	100
9.3.4	<i>Resultados Cuestionario para el Responsable de las Pruebas</i>	101
9.3.5	<i>Conclusiones</i>	101
9.4	PRUEBAS DE RENDIMIENTO	101
10	MANUALES DEL SISTEMA	102
10.1	MANUAL DE INSTALACIÓN.....	102
10.2	MANUAL DE EJECUCIÓN.....	102
10.3	MANUAL DE USUARIO	103
10.4	MANUAL DEL PROGRAMADOR.....	104
11	CONCLUSIONES Y AMPLIACIONES.....	106
11.1	CONCLUSIONES	106
11.2	AMPLIACIONES.....	106
12	BIBLIOGRAFÍA.....	108
12.1	LIBROS Y MANUALES TÉCNICOS	108
12.2	DOCUMENTACIÓN OFICIAL Y ESTÁNDARES.....	108
12.3	NORMATIVA ACADÉMICA Y LEGAL	108
12.4	RECURSOS WEB Y ARTÍCULOS ESPECIALIZADOS	108
13	ANEXOS	109
13.1	CONTENIDO ENTREGADO EN EL CD-ROM.....	109
13.1.1	<i>Contenidos</i>	109
13.1.2	<i>Código Ejecutable e Instalación</i>	110

13.1.3	<i>Ficheros de Configuración</i>	110
13.2	CÓDIGO FUENTE	111
13.2.1	<i>Paquete client:</i>	111
13.2.2	<i>Paquete server:</i>	131

Índice de Figuras

Figura 1.	Portada del libro “Design Patterns” (Gamma et al., 1994).....	14
Figura 2.	Patrón Observer.....	15
Figura 3.	Patrón Adapter	16
Figura 4.	Patrón Singleton (1)	17
Figura 5.	Patrón Singleton (2)	17
Figura 6.	Patrón Abstract Factory	18
Figura 7.	Patrón Flyweight	19
Figura 8.	Patrón Iterator	19
Figura 9.	Patrón Facade	20
Figura 10.	Esquema conceptual del uso de un proxy entre cliente y servidor	22
Figura 11.	Ejemplo de arquitectura de comunicación basada en WebSocket	27
Figura 12.	UML del sistema.....	50
Figura 13.	Boceto de la interfaz	55
Figura 14.	Clase YahooooFinanceLibraryAdapter	61
Figura 15.	Clase LibraryStateCache	61
Figura 16.	Clase LibraryStateSubscription	61
Figura 17.	Clase LibraryStateSubscriptionIterator	61
Figura 18.	Clase LibraryStateSubscriptionManager	62
Figura 19.	Clase WebsocketManager	62
Figura 20.	Clase Settings	62
Figura 21.	Diagrama de clases global	63
Figura 22.	Diagrama de Arquitectura	65
Figura 23.	Prototipo Real (modo oscuro)	68
Figura 24.	Prototipo Real (modo claro)	69
Figura 25.	Servidor iniciado	103
Figura 26.	Vista del cliente	104

1 Presentación

El desarrollo de aplicaciones web modernas exige cada vez más soluciones que permitan una comunicación eficiente y en tiempo real entre cliente y servidor. Tradicionalmente, esta necesidad se ha cubierto mediante técnicas como *polling* o *long polling*, pero estas presentan limitaciones en cuanto a latencia y consumo de recursos. En este contexto, el protocolo **WebSocket** se ha consolidado como una alternativa más eficiente para la transmisión de datos bidireccional y persistente.

El presente proyecto parte de esa necesidad generalizada, pero no se limita al desarrollo de una solución concreta. Su objetivo principal es el diseño y la implementación de una **biblioteca reutilizable** que facilite la incorporación de capacidades de actualización en tiempo real, mediante WebSocket, a cualquier interfaz de programación de aplicaciones (API; application programming interface) de tipo REST basada en el protocolo HTTP propio de la transmisión de datos en Internet. Esta biblioteca se ha concebido como una herramienta genérica, extensible y fácilmente integrable en distintos contextos.

Para alcanzar este objetivo, el trabajo se ha abordado con un enfoque orientado a objetos, estructurando el código en distintos módulos especializados y aplicando varios **patrones de diseño** reconocidos (*Adapter*, *Observer*, *Proxy*, entre otros) que favorecen la reutilización, claridad y mantenibilidad del código. Asimismo, se ha implementado un adaptador concreto sobre la API de **Yahoo Finance** como caso de uso de demostración, sin que el proyecto esté destinado a un cliente específico.

La memoria se estructura en varios capítulos. En primer lugar, se introducen los conceptos teóricos clave relacionados con WebSocket, la arquitectura REST y los patrones de diseño utilizados. A continuación, se describe en detalle el proceso de diseño e implementación de la biblioteca, incluyendo diagramas UML y decisiones técnicas relevantes. Finalmente, se presentan las conclusiones extraídas del trabajo y las posibles líneas de evolución futura.

En definitiva, esta memoria no solo documenta el desarrollo de una solución técnica funcional, sino que también ofrece una visión general sobre cómo diseñar software modular, reutilizable y alineado con las necesidades actuales del desarrollo web en tiempo real.

2 Aspectos Teóricos

En este apartado se presentan los **conceptos, tecnologías y herramientas** que sirven de base para el desarrollo del proyecto. Se ofrece una visión general de su origen y propósito, destacando su papel dentro del ecosistema actual de desarrollo web y su relevancia para la construcción de sistemas eficientes, escalables y mantenibles.

Además de describir cada elemento, se justifica su **selección e integración** en la solución propuesta, mostrando cómo contribuyen a habilitar la comunicación en tiempo real sobre una arquitectura tradicional basada en APIs REST. De este modo, esta sección proporciona el marco conceptual necesario para comprender las decisiones técnicas adoptadas en las fases posteriores del trabajo.

2.1 Patrones de diseño

2.1.1 Introducción

En el ámbito de la ingeniería del software, los **patrones de diseño** constituyen un conjunto de soluciones probadas para problemas que aparecen de manera recurrente durante el proceso de diseño y desarrollo de sistemas. Su importancia radica en dos aspectos fundamentales:

En primer lugar, los patrones de diseño proporcionan **técnicas estructuradas para la resolución de problemas comunes**. Funcionan como una “caja de herramientas” conceptual que permite reconocer situaciones similares en distintos contextos y aplicar soluciones bien definidas, facilitando así la reutilización de ideas y la mejora de la calidad del diseño.

En segundo lugar, aportan un **lenguaje común** para la comunicación entre desarrolladores. Gracias a ellos, equipos de trabajo pueden discutir ideas complejas de forma concisa y precisa: si un compañero sugiere, por ejemplo, aplicar un *Observer*, el resto del equipo comprende de inmediato la solución propuesta sin necesidad de largas explicaciones. Este vocabulario compartido favorece la colaboración, la claridad y la estandarización en el desarrollo de software.

La difusión moderna de los patrones de diseño está estrechamente ligada al libro “Design Patterns: Elements of Reusable Object-Oriented Software” (1994), escrito por **Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides**. Esta obra se considera un **clásico de la programación orientada a objetos** y ha ejercido una influencia decisiva en la ingeniería del software. A sus autores se les conoce informalmente como la “Gang of Four” (GoF, Banda de los Cuatro), en alusión humorística a un grupo de dirigentes políticos leales a Mao Zedong en China durante la época de la Revolución Cultural (1966-76). Desde su publicación, este libro se ha convertido en una referencia fundamental para profesionales y estudiantes de informática en todo el mundo.

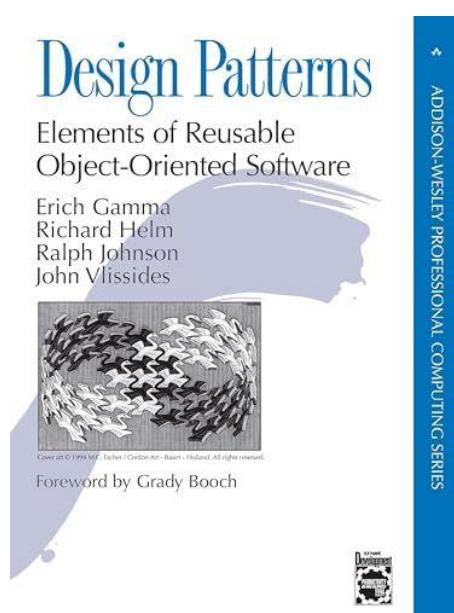


Figura 1. Portada del libro “Design Patterns” (Gamma et al., 1994)

2.1.2 Observer

Descripción: El patrón **Observer** [1] permite que un objeto notifique a otros objetos sobre cambios de estado. Se implementa como una forma de suscripción y notificación de eventos.

Una analogía posible sería un sistema de notificación meteorológica. Un objeto central (el “sujeto”) representa la estación meteorológica y varios dispositivos (aplicaciones, paneles, relojes inteligentes) actúan como observadores que reciben actualizaciones automáticas cuando cambian la temperatura o la humedad.

Implementación: Este patrón se compone de dos roles principales: el sujeto (subject), que mantiene una lista de observadores y los notifica ante cualquier cambio, y los observadores (observers), que reaccionan al evento. La comunicación suele implementarse mediante una interfaz común que define el método de actualización (update()).

Ejemplo: Un ejemplo típico del patrón Observer es el de un widget o elemento visual que, en una aplicación gráfica, muestre en un campo de texto el nombre de algún tipo de objeto, como podría ser un nodo de una escena. Si el nombre de ese objeto puede cambiar por mecanismos diversos, el widget tendrá que “observarlo” para mantener sincronizados el nombre que se muestra y el nombre interno del objeto.

Uso en el proyecto: En *LibraryStateSubscription.js* y *LibraryStateSubscriptionManager.js* se llevan las suscripciones a cambios en valores de estados que se monitorizan. En *WebSocketManager.js* se reciben mensajes de suscripción y se notifican los cambios a los clientes conectados. Nótese que este uso del patrón Observer se manifiesta aquí en el nivel de la arquitectura de red del sistema y no en la interacción entre clases dentro de un programa aislado, a diferencia de los casos de uso canónicos.



Figura 2. Patrón Observer

2.1.3 Adapter

Descripción: El patrón **Adapter** [1] permite que interfaces incompatibles colaboren, adaptando la API externa a la que se espera internamente.

Una analogía posible sería un adaptador de corriente eléctrica: el dispositivo espera un tipo de enchufe y voltaje específico, mientras que el adaptador convierte la señal para que ambos puedan funcionar juntos.

Implementación: Este patrón se implementa creando una clase intermedia que traduce las llamadas entre dos interfaces diferentes. El adaptador implementa la interfaz esperada por el cliente y delega internamente las operaciones al objeto adaptado, transformando los datos o métodos según sea necesario.

Ejemplo: Un ejemplo posible podría ser: dada una API en lenguaje C que se quiera utilizar en C++, se podrían escribir funciones adaptadoras que utilicen parámetros típicos de C++, como el tipo “`std::string`” para cadenas de texto, que llamen a las funciones del API traduciendo los parámetros a los tipos utilizados en la API en C como “`const char*`”. De ese modo, el código en C++ será más limpio, evitando las traducciones entre tipos, que quedarán encapsuladas en las funciones adaptadoras.

Uso en el proyecto: En `YahooFinanceLibraryAdapter.js`, donde se adapta la API de yahoo-finance2 para que cumpla con la interfaz definida (métodos `GetStateNames()` y `GetStateValue()`).

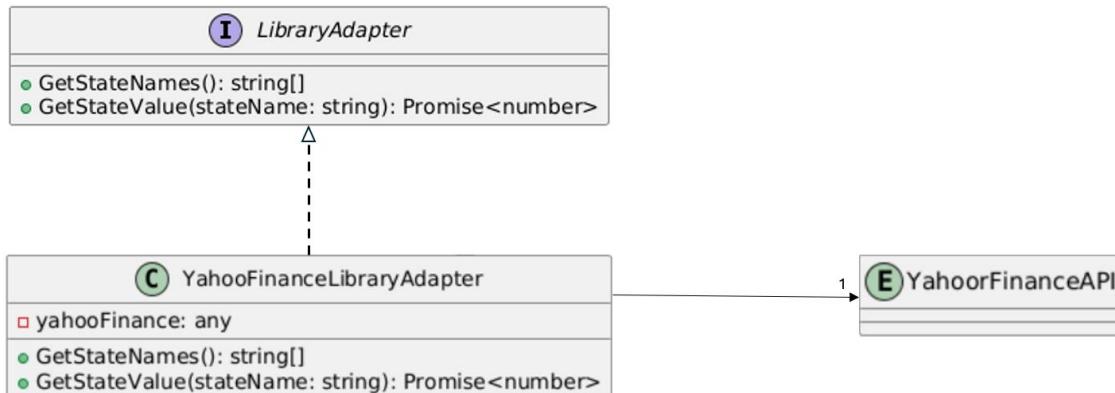


Figura 3. Patrón Adapter

2.1.4 Singleton

Descripción: El patrón **Singleton** [1] garantiza que una clase tenga una única instancia y proporciona un punto global de acceso a ella.

Implementación: Se implementa normalmente declarando un constructor privado que impide crear instancias directamente. La clase expone un método estático, generalmente denominado `getInstance()`, que devuelve siempre la misma instancia única.

Ejemplo: Un ejemplo clásico este patrón es la gestión de la configuración de una aplicación. Solo debe existir un objeto que almacene las preferencias globales, de modo que cualquier componente del sistema consulte y modifique las mismas opciones compartidas.

Uso en el proyecto: Las clases *Settings.js* y *WebSocketSubscriptionManager.js* controlan su construcción mediante un método *GetInstance()* y un flag para prevenir instanciaciones directas.

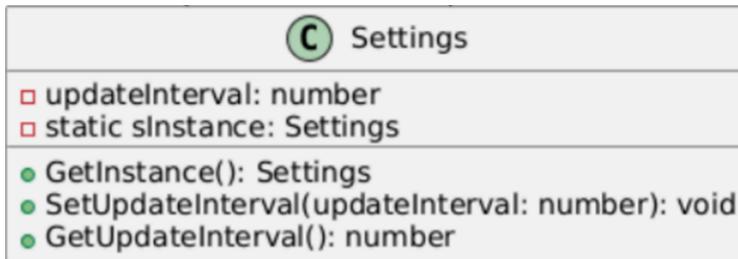


Figura 4. Patrón Singleton (1)

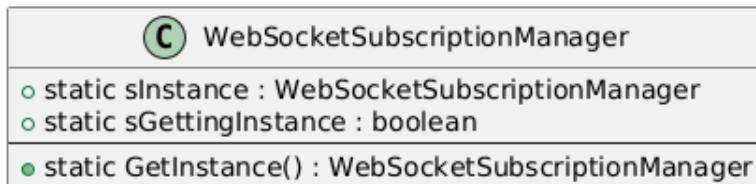


Figura 5. Patrón Singleton (2)

2.1.5 Abstract Factory

Descripción: El patrón **Abstract Factory** [1] permite delegar la creación de objetos a una clase central que se encarga de construir instancias específicas, según ciertas condiciones o parámetros. Esto promueve la separación de responsabilidades y facilita la extensibilidad.

Implementación: Define una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Cada “fábrica concreta” implementa esa interfaz y devuelve objetos compatibles entre sí. Este patrón se apoya frecuentemente en otros, como *Factory Method* o *Singleton*, para gestionar la creación centralizada.

Ejemplo: Un ejemplo clásico es un sistema de interfaz gráfica multiplataforma. Una fábrica abstracta puede definir métodos para crear botones y ventanas; las fábricas concretas generan los componentes correspondientes a Windows, macOS o Linux, garantizando la coherencia visual entre ellos.

Uso en el proyecto: En este proyecto se ha implementado una factoría para instanciar y encapsular el comportamiento de los componentes encargados de gestionar las suscripciones al estado del sistema.

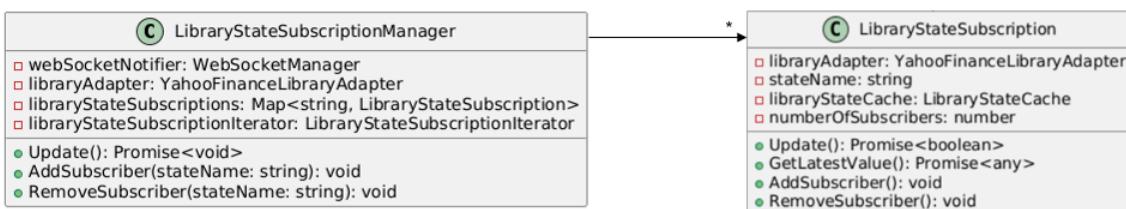


Figura 6. Patrón Abstract Factory

2.1.6 Proxy

Descripción: El patrón **Proxy** [1] proporciona un sustituto o representante de otro objeto para controlar su acceso. Este patrón permite añadir una capa intermedia que puede encargarse de gestionar la creación, el control de acceso, la comunicación remota o la optimización del uso de recursos antes de que el objeto real sea invocado.

Implementación: El patrón se implementa definiendo una interfaz común entre el objeto real (RealSubject) y su representante (*Proxy*). El proxy mantiene una referencia al objeto real y redirige las llamadas del cliente hacia él, añadiendo lógica adicional según sea necesario (control de permisos, registro de uso, almacenamiento en caché, etc.).

Ejemplo: Un ejemplo típico es el acceso remoto a un servicio. En lugar de que el cliente se comunique directamente con el servidor, utiliza un objeto Proxy local que implementa la misma interfaz y se encarga de establecer la conexión, enviar las peticiones y devolver las respuestas.

Uso en el proyecto: La biblioteca desarrollada actúa conceptualmente como un proxy entre el cliente y una API REST existente. En lugar de que el cliente se comunique directamente con la API original, lo hace a través de la biblioteca, que recibe las peticiones, y las reenvía al servidor de destino. Nuestro uso de este patrón de diseño tiene lugar en el nivel de la arquitectura de red, como en el caso del patrón *Observer*, comentado anteriormente.

2.1.7 Flyweight

Descripción: El patrón **Flyweight** [1] se utiliza para reducir el consumo de memoria y mejorar el rendimiento cuando se manejan grandes cantidades de objetos similares. Su principio básico consiste en compartir el estado común entre varios objetos en lugar de duplicarlo, almacenando solo la información que es verdaderamente variable en cada instancia.

Implementación: El patrón se implementa dividiendo el estado de los objetos en dos partes: el estado intrínseco, que es inmutable y puede compartirse entre instancias, y el estado extrínseco, que varía y se gestiona externamente.

Ejemplo: Un ejemplo clásico es el de los caracteres en un procesador de texto: cada letra de la fuente (por ejemplo, la forma visual de la “a”, “b”, “c”) se representa con un único objeto compartido, mientras que la posición o color de cada carácter se guarda de forma externa. Esto permite manejar miles de caracteres sin crear miles de objetos redundantes.

Uso en el proyecto: Se aplica para optimizar la gestión de memoria y evitar duplicación de estados compartidos. Cada instancia de *LibraryStateCache.js* actúa como un objeto *Flyweight*,

que almacena el estado común (por ejemplo, el valor más reciente de un recurso monitorizado). Las clases *LibraryStateSubscription.js* funcionan como contextos que asocian a cada cliente con dicho estado, reutilizando la misma información base. Este enfoque reduce significativamente la creación de objetos redundantes y mejora la eficiencia del sistema cuando múltiples clientes se suscriben a los mismos datos.



Figura 7. Patrón Flyweight

2.1.8 Iterator

Descripción: El patrón **Iterator** [1] proporciona un mecanismo uniforme para recorrer los elementos de una colección sin exponer su estructura interna. Permite acceder secuencialmente a los elementos de una colección (como listas, árboles o mapas) sin que el cliente tenga que conocer los detalles de implementación.

Implementación: El patrón se implementa definiendo una interfaz con métodos como *next()* y *isDone()*. La colección concreta (por ejemplo, una lista o conjunto) proporciona su propio iterador, que mantiene un puntero o referencia al elemento actual. Esto separa la lógica de iteración de la estructura de almacenamiento, favoreciendo la extensibilidad y la encapsulación.

Ejemplo: Un ejemplo clásico es el recorrido de una lista de reproducción musical. El iterador permite pasar de una canción a otra sin que el usuario conozca cómo se almacenan internamente (array, base de datos, flujo en red, etc.).

Uso en el proyecto: En la biblioteca desarrollada, el patrón Iterator se refleja en la gestión de colecciones de suscripciones activas. El *LibraryStateSubscriptionManager.js* utiliza la clase *LibraryStateSubscriptionIterator.js* [nueva clase] para recorrer las suscripciones registradas (*LibraryStateSubscription.js*). Esto se hace mediante una estructura interna que maneja la secuencia y notifica a cada cliente suscrito cuando un estado cambia, evitando exponer directamente la estructura de datos interna del Manager y permitiendo un control granular de la iteración, como limitar el número de actualizaciones por ciclo (*kMaximumNumberOflterations*).

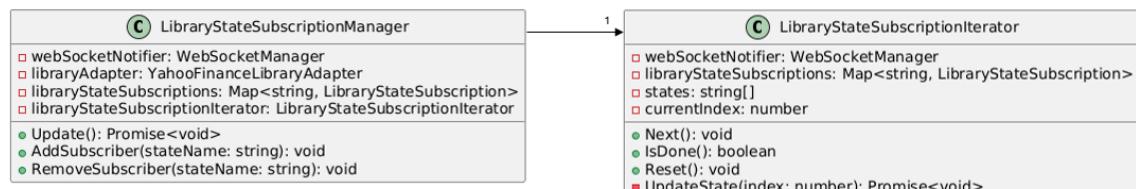


Figura 8. Patrón Iterator

2.1.9 Facade

Descripción: El patrón **Facade** [1] proporciona una interfaz unificada y simplificada a un conjunto de interfaces en un subsistema complejo. Define un punto de entrada de alto nivel que hace que el subsistema sea más fácil de usar, ocultando la complejidad subyacente.

Implementación: La Fachada es una clase única (o un número reducido de clases) que contiene una referencia a las distintas clases del subsistema complejo. Expone un conjunto limitado de métodos que delegan las llamadas en los objetos apropiados del subsistema.

Ejemplo: Una clase *MultimediaConverter* que expone un único método *Convert(file, format)*, ocultando las interacciones complejas con los subsistemas de códecs, buffers y manejo de ficheros.

Uso en el proyecto: La clase *WebSocketSubscriptionManager* actúa como la Fachada principal para el subsistema de suscripciones del cliente. Su rol es proporcionar una interfaz de alto nivel y muy simple al desarrollador, a través de los métodos *AcquireSubscription()* y *ReleaseSubscription()*, que ocultan la complejidad de:

- La lógica de creación del objeto *WebSocketSubscription*.
- El manejo interno del ciclo de vida y estado de la única conexión WebSocket.

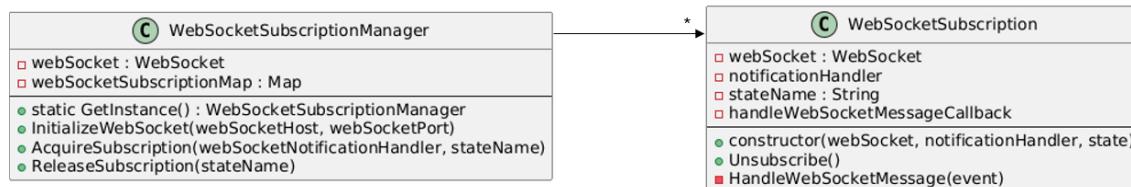


Figura 9. Patrón Facade

2.2 Conceptos Claves

2.2.1 Comunicación en tiempo real en la Web

Desde sus inicios, la comunicación en la web se ha basado en el protocolo HTTP, que opera bajo un esquema de solicitud-respuesta. Sin embargo, este modelo presenta limitaciones para aplicaciones que requieren actualizaciones en tiempo real. Para mitigar esta problemática, han surgido diversas soluciones como el *polling*, el *long polling* y, más recientemente, el protocolo WebSocket, que ofrece una comunicación bidireccional persistente entre cliente y servidor.

2.2.2 Patrón Observer aplicado a WebSocket

En el contexto de WebSocket, el patrón *Observer* se aplica de forma natural: el servidor actúa como sujeto, gestionando el estado compartido, mientras que los clientes se comportan como observadores, recibiendo notificaciones automáticas cuando se producen cambios. Esto elimina la necesidad de que cada cliente realice peticiones periódicas al servidor, reduciendo la carga de red y mejorando la latencia de actualización.

Este enfoque resulta especialmente útil en aplicaciones que requieren sincronización inmediata, como sistemas de chat, paneles de monitorización o plataformas colaborativas en tiempo real.

2.2.3 Concepto de Proxy y su aplicación en el diseño

En el contexto de las comunicaciones entre cliente y servidor, este proyecto representa un caso particular de aplicación del patrón de diseño Proxy en el nivel de configuración y arquitectura de servidores. En lugar de que los clientes web se comuniquen directamente con la API REST de destino, la biblioteca desarrollada actúa como un proxy inverso, intermediando entre ambos extremos.

Este enfoque permite centralizar la gestión de peticiones, controlar la lógica de negocio y añadir una capa adicional de comunicación en tiempo real mediante WebSocket, sin necesidad de modificar la API original. Además, facilita la reutilización de la biblioteca con distintas fuentes de datos externas, mejorando la escalabilidad y el mantenimiento del sistema.

En términos generales, un proxy puede recibir peticiones del cliente y reenviarlas al servidor (o viceversa), modificando, controlando o simplemente registrando el tráfico según sea necesario.

Existen dos tipos principales de proxy:

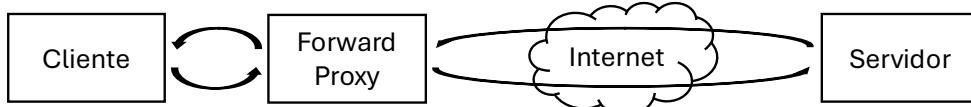
- **Proxy directo (forward proxy)**: es un intermediario que el cliente configura voluntariamente para acceder a Internet u otros servicios. Este tipo de proxy se utiliza habitualmente para filtrar tráfico saliente, mejorar el rendimiento mediante cachés o anonimizar peticiones.
- **Proxy inverso (reverse proxy)**: se sitúa entre el servidor y los clientes, pero es gestionado por el servidor o por la infraestructura del *backend*. Su función habitual es distribuir

carga, gestionar la seguridad o transformar peticiones antes de que lleguen al servidor real.

Caso abstracto:



Caso concreto: proxy directo



Caso concreto: proxy inverso

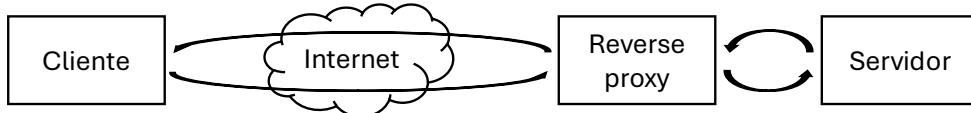


Figura 10. Esquema conceptual del uso de un proxy entre cliente y servidor

En este proyecto, la **biblioteca desarrollada actúa como un proxy inverso**. En lugar de que los clientes web se comuniquen directamente con una API REST existente (por ejemplo, la API de Yahoo Finance), lo hacen a través de nuestra biblioteca, que recibe las peticiones, las retransmite al servidor de destino, y adicionalmente gestiona una conexión WebSocket para enviar notificaciones en tiempo real cuando se detectan cambios en los datos.

Este enfoque ofrece múltiples ventajas:

- **Reduce la carga de peticiones repetidas** en la API original, recluyendo la lógica de *polling* dentro del servidor proxy.
- **Facilita el control del tráfico** y la lógica de negocio desde una única capa intermedia.
- **Hace posible reutilizar la biblioteca** con distintas APIs REST sin necesidad de modificar el cliente o la API final.

Este uso del patrón proxy, y en particular del proxy inverso, es una estrategia potente para adaptar sistemas existentes a nuevas necesidades sin alterar su arquitectura interna.

2.3 Herramientas de Desarrollo

2.3.1 Protocolo WebSocket

WebSocket es un protocolo de comunicación bidireccional y persistente que permite el intercambio de datos entre cliente y servidor a través de una única conexión TCP. Fue estandarizado por el IETF (Internet Engineering Task Force) en 2011 (RFC 6455) [4] y forma parte de las APIs del navegador definidas por el W3C.

A diferencia del modelo tradicional basado en HTTP, donde cada solicitud debe ser iniciada por el cliente, WebSocket permite que el servidor envíe datos de forma proactiva en tiempo real. Esto lo convierte en una solución ideal para aplicaciones donde la actualización inmediata de los datos es crítica (sistemas financieros, juegos online, chat, etc.) y es el servidor el que sabe cuándo cambian los datos.

En este proyecto, WebSocket se utiliza como base para la comunicación de eventos de cambio de estado entre el servidor y los clientes web, mejorando el rendimiento frente a alternativas como el *polling*.

2.3.2 Node.js y Express

Para la implementación del servidor WebSocket, se ha utilizado **Node.js**, un entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome. La elección de Node.js se debe a su capacidad para manejar operaciones de E/S de manera asíncrona, lo que lo hace ideal para aplicaciones en tiempo real. Junto con Node.js, se ha empleado **Express.js**, un *framework* minimalista que facilita la creación de API REST.

2.3.3 Herramientas del Cliente Web

Para el desarrollo de la interfaz de usuario se han utilizado dos bibliotecas ligeras y ampliamente adoptadas en el ecosistema *frontend*:

- **Tailwind CSS:** Se ha incluido mediante CDN (<script src="https://cdn.tailwindcss.com">) y ha permitido construir una interfaz moderna y responsive sin necesidad de escribir hojas de estilo personalizadas. Tailwind se basa en clases de utilidad que agilizan el desarrollo visual y aseguran una mayor coherencia estética a lo largo del sistema.
- **Chart.js:** Incorporada también mediante CDN (<script src="https://cdn.jsdelivr.net/npm/chart.js">), esta biblioteca ha sido clave para representar gráficamente la evolución de los precios monitorizados. Su integración facilita la creación de gráficos dinámicos que se actualizan automáticamente cuando se reciben nuevos datos vía WebSocket.

El término **CDN** (*Content Delivery Network*, o *red de distribución de contenidos*) hace referencia a una infraestructura de servidores distribuidos geográficamente que permite entregar archivos estáticos (como bibliotecas CSS o JavaScript) de forma más rápida y eficiente. Al utilizar un CDN,

el navegador descarga los recursos desde el servidor más cercano, reduciendo la latencia y mejorando el rendimiento de la aplicación. Este mecanismo es ampliamente utilizado en la web moderna y está disponible a través de proveedores como **Cloudflare** [18] y **Amazon Web Services (AWS)** [19].

Ambas herramientas han contribuido significativamente a la experiencia de usuario final, permitiendo una visualización clara e intuitiva de la información en tiempo real.

2.4 Metodología y Notaciones

2.4.1 Proceso Unificado de Desarrollo (RUP)

El proyecto sigue una variante del **Proceso Unificado de Desarrollo de Software (Rational Unified Process, RUP)**, un marco metodológico ampliamente utilizado en ingeniería del software que permite una gestión iterativa e incremental del desarrollo. RUP estructura el ciclo de vida del proyecto en fases y entregas parciales, facilitando la validación progresiva de los resultados y la adaptación a cambios durante el desarrollo.

Para este Trabajo de Fin de Grado se han priorizado los entregables más relevantes, adaptando la documentación y las actividades a las necesidades académicas específicas, manteniendo al mismo tiempo los principios fundamentales de planificación, iteración y control de calidad característicos de RUP.

2.4.2 Notación UML

Para la representación de los diagramas del sistema, se ha utilizado **UML (Unified Modeling Language)**, un estándar ampliamente usado en ingeniería de software. Diagramas como el **diagrama de clases**, **diagrama de secuencia** y **diagrama de estados** serán utilizados para describir la arquitectura y comportamiento del sistema.

2.4.3 Justificación y aplicación práctica

La elección del modelo **RUP** se ha basado en su capacidad para adaptarse a proyectos de tamaño medio, como un Trabajo de Fin de Grado, permitiendo una planificación clara, entregas parciales y validación progresiva. Su enfoque iterativo ha facilitado la incorporación de mejoras durante el desarrollo, especialmente en fases como el diseño de la arquitectura y la implementación de pruebas.

Por otro lado, UML ha sido seleccionado como notación por su amplia aceptación en el ámbito de la ingeniería del software y su capacidad para representar de forma precisa tanto la estructura como el comportamiento del sistema. Los diagramas de clases, secuencia y estados han sido utilizados no solo como documentación, sino también como herramienta de análisis previo a la codificación, asegurando coherencia entre diseño y desarrollo.

3 Memoria del Proyecto

3.1 Hoja de identificación

Título del proyecto:

Diseño y desarrollo de biblioteca WebSocket para APIs HTTP

Código identificador: [TFG-UO27730]

Cliente: Escuela de Ingeniería Informática de la Universidad de Oviedo

C.I.F.: Q3318001I Dirección: C/ Valdés Salas, 33007 Oviedo

Teléfono: 985 10 27 96 Fax: 985 10 27 96

Suministrador: Luis Miguel Gómez del Cueto

N.I.F.: [n.i.f.] Dirección: [dirección suministrador]

Teléfono: [teléfono]

Fecha y firma del cliente:

[Espacio para la fecha]

[Espacio para la firma]

Fecha y firma del suministrador:

[Espacio para la fecha]

[Espacio para la firma]

Resumen:

Diseño e implementación de una API WebSocket que complementa una API HTTP existente para mejorar la comunicación en tiempo real entre cliente y servidor, reduciendo latencia y optimizando recursos en aplicaciones web modernas.

Duración estimada: 8,5 meses (300 horas)

Coste: 3.750 €

3.2 Introducción

La evolución de las tecnologías web ha estado impulsada por la necesidad de establecer una comunicación más eficiente y dinámica entre clientes y servidores. Aunque el protocolo HTTP sigue siendo el estándar predominante, su modelo de solicitud-respuesta no está optimizado para aplicaciones que requieren actualizaciones frecuentes o en tiempo real. En estos casos, el uso repetido de peticiones genera una carga innecesaria tanto para la red como para el servidor.

Para resolver esta limitación, surgieron técnicas como el *polling* y el *long polling*, que permiten al cliente comprobar periódicamente si existen cambios en los datos. Sin embargo, estas soluciones siguen siendo ineficientes en términos de consumo de recursos. Como alternativa más eficaz, como se explicó en el resumen, el protocolo WebSocket permite establecer una conexión persistente y bidireccional entre cliente y servidor, lo que posibilita el envío inmediato de actualizaciones sin necesidad de nuevas solicitudes.

Este Trabajo de Fin de Grado tiene como objetivo el diseño y desarrollo de una biblioteca reutilizable que facilite la integración de WebSocket en cualquier API REST ya existente. De este modo, se busca simplificar la adición de comunicación en tiempo real en aplicaciones web modernas, sin tener que reescribir grandes partes del código ya implementado.

El enfoque adoptado se apoya en principios de diseño orientado a objetos y en la aplicación de varios patrones de diseño de software (como el patrón Observer, entre otros), lo que mejora la modularidad, extensibilidad y mantenibilidad del código. La solución resultante pretende ser lo suficientemente genérica como para adaptarse a diferentes contextos y necesidades, sirviendo como herramienta base para futuros desarrollos.

La memoria está estructurada de forma que se introducen primero los fundamentos teóricos y tecnológicos empleados, seguidos por el diseño de la biblioteca, su implementación práctica y las pruebas realizadas. Finalmente, se recogen las conclusiones y se proponen posibles líneas de mejora futuras. El trabajo se ha desarrollado conforme a la norma UNE 157801 y dentro del marco metodológico adaptado para proyectos de fin de grado.

3.3 Objeto

En la actualidad, muchas aplicaciones web requieren una comunicación constante y en tiempo real entre el servidor y los distintos usuarios conectados. Este es el caso de plataformas de mensajería, sistemas de monitorización de datos en directo, aplicaciones colaborativas o servicios de atención al cliente. En todos estos casos, es fundamental que los cambios en el sistema se reflejen de inmediato en la interfaz del usuario, sin necesidad de que este tenga que refrescar la página o realizar una nueva petición.

Tradicionalmente, este tipo de funcionalidad se resolvía mediante técnicas como el *polling* o el *long polling*, que implican que el cliente consulte al servidor de forma periódica para ver si hay novedades. Estas técnicas funcionan, pero consumen muchos recursos y generan una gran cantidad de tráfico innecesario, especialmente cuando hay muchos usuarios simultáneos.

Este proyecto surge para dar respuesta a esa necesidad, mediante el desarrollo de una biblioteca de software reutilizable que permite ampliar cualquier API REST tradicional (basada en el protocolo HTTP) con un canal de comunicación en tiempo real utilizando WebSocket. Esta biblioteca ha sido diseñada para integrarse fácilmente en proyectos ya existentes, permitiendo a los desarrolladores añadir soporte para actualizaciones instantáneas sin tener que reescribir todo el sistema.

A diferencia de soluciones más específicas o *ad hoc*, esta herramienta se ha construido siguiendo principios de diseño modular y orientado a objetos, lo que facilita su mantenimiento y adaptación a distintos contextos. Además, incorpora patrones de diseño ampliamente reconocidos que permiten gestionar de forma ordenada y escalable las suscripciones y notificaciones de eventos.

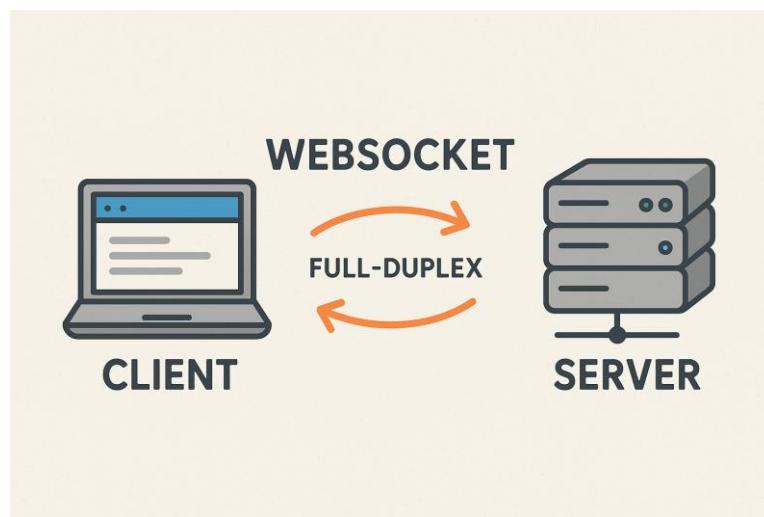


Figura 11. Ejemplo de arquitectura de comunicación basada en WebSocket

El resultado es una solución más eficiente, moderna y flexible que las alternativas tradicionales, que mejora significativamente la experiencia del usuario y reduce la carga sobre los servidores. Esta biblioteca está pensada para ser usada en todo tipo de proyectos web que necesiten actualizaciones en tiempo real, y su desarrollo contribuye a fomentar la reutilización de código y la mejora continua de la calidad del software.

La biblioteca desarrollada puede ser aplicada en múltiples contextos, como:

- Sistemas de notificación en tiempo real.
- Dashboards financieros o industriales.
- Aplicaciones colaborativas (editores, pizarras, chats).
- Monitorización de sensores IoT.
- Juegos en línea con sincronización de estado.

3.4 Antecedentes

La comunicación entre cliente y servidor en aplicaciones web ha evolucionado significativamente en las últimas décadas. Inicialmente, el modelo dominante era el de solicitud-respuesta basado en el protocolo HTTP, donde el cliente realiza peticiones puntuales y el servidor responde con los datos solicitados. Este enfoque, aunque eficaz para muchas aplicaciones, presenta limitaciones evidentes en entornos donde se requiere una **actualización constante de información**, como en sistemas financieros, plataformas colaborativas o aplicaciones de monitorización.

Para paliar estas carencias, se han utilizado técnicas como el *polling*, que consiste en realizar peticiones periódicas al servidor para comprobar si hay nuevos datos. Aunque funcional, esta técnica genera una carga innecesaria en el servidor y en la red, además de introducir latencia en la recepción de los datos. El *long polling*, una variante más eficiente, mantiene la conexión abierta hasta que el servidor tiene información nueva que enviar, pero sigue sin ser óptimo en términos de escalabilidad.

La necesidad de una solución más eficiente llevó al desarrollo del **protocolo WebSocket**, estandarizado por el IETF en el RFC 6455 [4]. Esta tecnología ha sido adoptada ampliamente en aplicaciones modernas como chats, videojuegos multijugador, plataformas de trading y sistemas de notificación.

En paralelo, el desarrollo de APIs REST ha sido una constante en la arquitectura de aplicaciones web. Estas APIs, basadas en HTTP, ofrecen una forma estructurada y escalable de acceder a recursos, pero no están diseñadas para emitir actualizaciones en tiempo real. Integrar WebSocket en este tipo de APIs requiere soluciones específicas que respeten su diseño original y permitan extender su funcionalidad sin comprometer su estabilidad.

Este proyecto surge precisamente de esa necesidad: **dotar a las APIs REST tradicionales de capacidades de comunicación en tiempo real**, mediante una biblioteca reutilizable que actúe como capa intermedia entre el cliente y el servidor. La solución propuesta se apoya en principios de diseño orientado a objetos y en patrones arquitectónicos que permiten una integración limpia, escalable y mantenible.

3.5 Descripción de la situación actual

En el desarrollo de aplicaciones web modernas, la arquitectura basada en API REST sobre HTTP sigue siendo el estándar predominante. Este modelo, aunque robusto y ampliamente adoptado, presenta limitaciones cuando se requiere una actualización continua de datos entre cliente y servidor. En particular, el patrón de solicitud-respuesta no permite una comunicación eficiente en tiempo real, lo que obliga a recurrir a soluciones como el *polling* o el *long polling*, que generan una carga innecesaria en el servidor y aumentan la latencia.

Actualmente, muchas APIs públicas (como la de **Yahoo Finance**) ofrecen datos actualizados con frecuencia, pero no están diseñadas para emitir notificaciones automáticas ante cambios. Esto obliga a los clientes a realizar peticiones periódicas para comprobar si ha habido variaciones, lo que resulta ineficiente tanto desde el punto de vista técnico como de experiencia de usuario.

Por otro lado, como se expuso en la introducción, el protocolo WebSocket es una solución ideal para entornos donde la inmediatez en la transmisión de datos es crítica. Sin embargo, su

integración en sistemas ya existentes suele requerir modificaciones profundas en la arquitectura, lo que dificulta su adopción en proyectos consolidados.

El punto de partida de este proyecto es la existencia de una API HTTP funcional, sobre la cual se desarrollará una API complementaria basada en WebSocket. La integración de esta nueva tecnología busca optimizar la comunicación en tiempo real sin reemplazar completamente el modelo basado en HTTP, sino combinando ambas soluciones para maximizar eficiencia y compatibilidad.

Desde el punto de vista de los recursos, el desarrollo del proyecto se enfrenta a los siguientes condicionantes:

- **Recursos humanos:** El equipo de desarrollo está compuesto por un programador con conocimientos en desarrollo web, arquitectura cliente-servidor y protocolos de comunicación en red.
- **Infraestructura hardware:** Se cuenta con un servidor de desarrollo y un entorno de pruebas adecuado para simular múltiples conexiones concurrentes.
- **Software y licencias:** Se emplearán tecnologías de código abierto para la implementación del servidor WebSocket.

En este contexto, se identifica una **carenza técnica**: la falta de herramientas que permitan incorporar WebSocket de forma sencilla y no invasiva en APIs REST ya desarrolladas. Esta situación plantea la necesidad de una solución que actúe como **capa intermedia**, capaz de interceptar peticiones HTTP, gestionar suscripciones y emitir actualizaciones en tiempo real sin alterar la lógica interna del sistema original.

El presente proyecto surge como respuesta a esta necesidad, proponiendo el desarrollo de una **biblioteca reutilizable** que facilite dicha integración. Esta herramienta permitirá dotar a cualquier API REST de capacidades de comunicación en tiempo real, mejorando la eficiencia del sistema y la experiencia del usuario final.

3.6 Normas y referencias

3.6.1 Disposiciones legales y normas aplicadas

Durante el desarrollo de este Trabajo de Fin de Grado se han tenido en cuenta diversas disposiciones legales y normativas técnicas relacionadas con el ámbito del software, la documentación técnica y el uso de tecnologías de terceros:

- **Ley de Propiedad Intelectual** (Real Decreto Legislativo 1/1996, de 12 de abril) [12]: se ha respetado la titularidad de los contenidos ajenos utilizados en el proyecto (documentación técnica, bibliotecas de terceros, imágenes, etc.), citando adecuadamente las fuentes y haciendo uso exclusivo de software con licencias de código abierto.
- **Reglamento General de Protección de Datos (RGPD)** (Reglamento UE 2016/679) [13]: si bien el sistema desarrollado no gestiona información personal, se ha prestado atención al diseño responsable y a la posibilidad de futuras adaptaciones que cumplan con la normativa vigente en materia de protección de datos.
- **Norma UNE 157801:2008** [14]: sobre la estructura de la documentación técnica en proyectos de software, aplicada como guía para organizar y redactar esta memoria, garantizando su coherencia y claridad.
- **RFC 6455 - The WebSocket Protocol** [4]: especificación técnica oficial del protocolo WebSocket, utilizada como base para la implementación de la biblioteca desarrollada.
- **RFC 9112 - HTTP/1.1** [5]: normativa técnica del protocolo HTTP/1.1, relevante para la integración con APIs REST tradicionales.

Además, se ha seguido un enfoque basado en buenas prácticas de ingeniería del software, haciendo uso del sistema de control de versiones Git y respetando los principios de diseño modular, reutilización de componentes y separación de responsabilidades.

3.6.2 Métodos, Herramientas, Modelos, Métricas y Prototipos

Durante el desarrollo del proyecto se han utilizado diversos métodos, herramientas y modelos que han guiado tanto la implementación técnica como la organización del trabajo. A continuación se describen los principales elementos aplicados en este apartado:

3.6.2.1 Métodos

- **Desarrollo Iterativo e Incremental**: siguiendo los principios del Proceso Unificado (RUP), el trabajo se ha estructurado en fases con entregables parciales, priorizando la funcionalidad base antes de implementar mejoras y ampliaciones.

- **Control de versiones con Git:** se ha utilizado el sistema de control de versiones Git para gestionar el código fuente, lo que ha permitido un seguimiento ordenado de los cambios, así como retrocesos seguros ante errores.

3.6.2.2 Herramientas

- **Node.js:** entorno de ejecución de JavaScript para el servidor, elegido por su eficiencia en aplicaciones en tiempo real y su ecosistema de paquetes. <https://nodejs.org>
- **Express.js:** framework ligero de Node.js para definir endpoints HTTP de forma sencilla y estructurada. <https://expressjs.com>
- **ws:** biblioteca para la implementación del servidor WebSocket. <https://github.com/websockets/ws>
- **Chart.js:** biblioteca de gráficos en JavaScript para la visualización en tiempo real de los datos monitorizados. <https://www.chartjs.org>
- **Tailwind CSS:** framework de diseño por clases utilitarias para el desarrollo rápido de interfaces responsivas. <https://tailwindcss.com>
- **Visual Studio Code:** entorno de desarrollo utilizado durante todo el proyecto. <https://code.visualstudio.com>
- **GitHub:** plataforma de alojamiento del repositorio del proyecto, donde se ha publicado el código siguiendo buenas prácticas de código abierto. <https://github.com>

3.6.2.3 Modelos

- **Modelado UML:** se han utilizado diagramas UML (de clases, de secuencia y de estados) para representar la estructura del sistema y sus interacciones internas.
- **Patrones de diseño de software:** Observer, Adapter, Proxy, Singleton, Abstract Factory Flyweight e Iterator han sido aplicados como modelos estructurales y de comportamiento para organizar el sistema de forma extensible y mantenible.

3.6.2.4 Métricas

- **Número de clases y líneas de código:** como métricas básicas para estimar la complejidad del sistema y su modularidad.
- **Conexiones concurrentes soportadas:** se ha realizado una estimación y pruebas básicas para asegurar que el sistema puede manejar al menos 1000 conexiones WebSocket simultáneas sin degradación significativa.
- **Tiempo de actualización en cliente:** medido como la latencia media entre un cambio detectado y la visualización del nuevo dato en la interfaz (aproximadamente 100-200 ms en entorno local).

3.6.2.5 Prototipos

- **Interfaz Web de prueba:** se ha desarrollado una interfaz web simple como prototipo funcional para comprobar la integración de la biblioteca con el cliente, permitiendo realizar suscripciones y visualizar los datos actualizados.
- **Caso de uso con API de Yahoo Finance:** se ha utilizado esta API como prototipo de integración real, permitiendo demostrar el funcionamiento completo del sistema en un contexto práctico.

3.6.3 Mecanismos de control de calidad aplicados durante la redacción del proyecto

Para garantizar la calidad de la documentación del proyecto, se han seguido una serie de prácticas y revisiones orientadas a asegurar la coherencia, precisión técnica y claridad expositiva de la memoria:

- **Uso de plantilla oficial del TFG:** en la medida de lo posible, se ha respetado la estructura y el formato establecidos por la plantilla elegida, lo que ha facilitado la homogeneidad y la organización de los contenidos.
- **Revisões periódicas:** el documento ha sido revisado en varias fases, tanto por el autor como con el acompañamiento del tutor del proyecto, para detectar errores, redundancias o carencias en la exposición de ideas técnicas y metodológicas.
- **Control de versiones con Git:** al igual que con el código, se ha mantenido el control de versiones del documento mediante Git, permitiendo realizar cambios incrementales y conservar un historial claro de modificaciones.
- **Corrección ortográfica y gramatical:** se han aplicado herramientas automáticas de revisión lingüística y se ha realizado una revisión manual final para evitar errores formales en la redacción.
- **Verificación de referencias:** todas las fuentes bibliográficas y técnicas han sido revisadas para garantizar su correcta citación y validez, evitando errores de atribución o enlaces rotos.
- **Coherencia entre código y documentación:** se ha prestado especial atención a que los fragmentos de código, diagramas y explicaciones técnicas estén alineados entre sí y con el funcionamiento real del sistema.

Estas medidas han permitido elaborar una memoria clara, rigurosa y conforme a los estándares académicos y técnicos exigidos para un Trabajo de Fin de Grado.

3.7 Definiciones y abreviaturas

3.7.1 Definiciones

- **API (Application Programming Interface)**: Conjunto de reglas y especificaciones que permiten la comunicación entre diferentes aplicaciones de software.
- **WebSocket**: Protocolo que permite la comunicación bidireccional y en tiempo real entre cliente y servidor a través de una única conexión persistente.
- **REST (Representational State Transfer)**: Estilo de arquitectura para el diseño de servicios web basado en recursos identificados por URLs y operaciones HTTP.
- **Cliente-servidor**: Modelo de arquitectura en el que las tareas se reparten entre proveedores de recursos o servicios (servidores) y demandantes (clientes).
- **Patrón de diseño**: Solución reutilizable a un problema común en el contexto del diseño de software.
- **Adaptador**: Componente que permite la integración de interfaces incompatibles, facilitando la comunicación entre sistemas distintos.
- **Suscripción**: Mecanismo mediante el cual un cliente se registra para recibir notificaciones de cambios en un recurso.
- **Notificación**: Mensaje enviado automáticamente para informar de un cambio de estado o evento relevante.
- **Latencia**: Tiempo que transcurre desde que se envía una solicitud hasta que se recibe la respuesta.
- **Escalabilidad**: Capacidad de un sistema para manejar un aumento en la carga de trabajo sin perder rendimiento.
- **Git**: Sistema de control de versiones distribuido, creado originalmente por Linus Torvalds para el desarrollo de Linux.
- **Prototipo**: Versión preliminar de un sistema o componente, utilizada para validar conceptos y funcionalidades.
- **Repositorio**: Espacio centralizado donde se almacena y gestiona el código fuente de un proyecto.
- **Proxy inverso (reverse proxy)**: Servidor que actúa como intermediario entre los clientes y los servidores de destino, gestionando las peticiones y respuestas.
- **Node.js**: Entorno de ejecución para JavaScript en el servidor
- **Express.js**: Framework para la creación de APIs REST en Node.js
- **Chart.js**: Biblioteca para la visualización de datos en la web
- **Tailwind CSS**: Framework de utilidades para el diseño de interfaces web

3.7.2 Abreviaturas

- **TFG**: Trabajo de Fin de Grado
- **API**: Application Programming Interface
- **HTTP**: HyperText Transfer Protocol
- **UML**: Unified Modeling Language
- **RUP**: Rational Unified Process
- **JS**: JavaScript
- **CSS**: Cascading Style Sheets
- **JSON**: JavaScript Object Notation
- **WS**: WebSocket (en el contexto de la biblioteca utilizada en Node.js)
- **RUP**: Rational Unified Process

3.8 Requisitos iniciales

El presente proyecto tiene como objetivo principal la implementación de una API basada en el protocolo WebSocket que complemente una API HTTP existente, optimizando la comunicación en tiempo real entre cliente y servidor. Para ello, se establecen los siguientes requisitos iniciales, tanto funcionales como no funcionales, que servirán de referencia para el desarrollo y validación del sistema:

3.8.1 Requisitos funcionales

- **RF1.** El sistema debe permitir la conexión de más de un cliente a través de WebSocket.
- **RF2.** El servidor debe emitir mensajes en tiempo real a los clientes conectados cuando se produzcan eventos.
- **RF3.** El sistema debe mantener la comunicación HTTP para operaciones tradicionales como autenticación y consulta de datos.
- **RF4.** La API WebSocket debe permitir el envío y recepción de mensajes estructurados en formato JSON.
- **RF5.** El sistema debe registrar los eventos de conexión y desconexión de los clientes.
- **RF6.** El sistema debe permitir representar gráficamente los datos recibidos en el cliente web mediante Chart.js.
- **RF7.** La API debe manejar correctamente errores y excepciones en la comunicación WebSocket, notificando al cliente en caso de fallo.

3.8.2 Requisitos no funcionales

- **RNF1.** El servidor debe ser capaz de gestionar al menos 200 conexiones WebSocket simultáneas sin que la latencia media de notificación supere los 500 ms en un entorno controlado (entorno local de desarrollo), para demostrar la viabilidad y eficiencia del diseño.
- **RNF2.** El código fuente debe estar debidamente documentado y seguir los estándares de buenas prácticas de desarrollo, facilitando su mantenimiento y evolución futura.
- **RNF3.** El sistema debe funcionar correctamente en los principales navegadores modernos (Google Chrome, Mozilla Firefox, Microsoft Edge), asegurando la compatibilidad multiplataforma.
- **RNF4.** La solución debe ser fácilmente desplegable en entornos compatibles con Node.js, permitiendo su integración en diferentes infraestructuras.
- **RNF5.** El código y la documentación deben estar gestionados mediante un sistema de control de versiones (Git), alojados en un repositorio público (GitHub) para garantizar la trazabilidad y colaboración.

3.9 Alcance

El presente proyecto tiene como objetivo principal el diseño e implementación de una biblioteca reutilizable que permita dotar de capacidades de comunicación en tiempo real a cualquier API basada en HTTP, mediante el uso del protocolo WebSocket. El desarrollo se realiza en un entorno académico, no vinculado a un cliente real, con el propósito de demostrar la viabilidad técnica y la calidad del diseño de la solución propuesta.

El sistema desarrollado se integra como una capa adicional que actúa como proxy inverso entre el cliente y una API REST ya existente, permitiendo que eventos relevantes se emitan en tiempo real a través de un canal WebSocket. Esta aproximación ofrece una mejora notable en eficiencia respecto a técnicas tradicionales como el *polling*, especialmente en contextos donde se requiere actualización constante de datos.

Elementos incluidos en el alcance del proyecto:

- Desarrollo de una biblioteca modular en Node.js capaz de exponer un canal WebSocket complementario a cualquier API HTTP.
- Implementación de un adaptador específico para la API de Yahoo Finance, como caso de uso demostrativo.
- Aplicación de patrones de diseño como Observer, Adapter, Abstract Factory y Singleton, para garantizar escalabilidad y mantenibilidad.
- Integración de la biblioteca con herramientas de *frontend* como Chart.js y Tailwind CSS para la visualización en tiempo real.
- Elaboración de documentación técnica detallada (manual de instalación, uso, y arquitectura del sistema).

Elementos excluidos del alcance del proyecto:

- Desarrollo de una interfaz gráfica completa orientada al usuario final.
- Implementación de medidas avanzadas de seguridad (como cifrado SSL/TLS, autenticación OAuth, etc.).
- Soporte para entornos distribuidos o balanceo de carga en producción.
- Integración con bases de datos persistentes o almacenamiento en nube.
- Compatibilidad con navegadores antiguos o tecnologías obsoletas.

Con este alcance, se busca entregar un sistema funcional, flexible y fácilmente integrable, que sirva como base para proyectos más amplios o aplicaciones en entornos reales. El foco está en la arquitectura del servidor WebSocket y su interoperabilidad con APIs existentes, priorizando la calidad técnica, la documentación y la posibilidad de reutilización.

3.10 Hipótesis y restricciones

El desarrollo del presente Trabajo de Fin de Grado se ha realizado bajo una serie de hipótesis razonables y restricciones asumidas que han condicionado tanto el diseño como la implementación del sistema. Estas suposiciones permiten acotar el alcance y contextualizar las decisiones técnicas adoptadas.

3.10.1 Hipótesis

H1. Se parte de la existencia de una API HTTP funcional y pública (como Yahoo Finance), que sirve de base para la integración con la biblioteca WebSocket.

H2. Los usuarios acceden a la solución desde navegadores modernos compatibles con tecnologías como WebSocket, JavaScript, HTML5 y CSS3.

H3. El entorno de ejecución del sistema será un entorno de desarrollo o pruebas controlado (por ejemplo, localhost o red local), sin necesidad de despliegue en producción.

H4. Se dispone de un equipo de desarrollo compuesto por una sola persona con conocimientos en desarrollo web, arquitectura cliente-servidor y patrones de diseño software.

H5. El código se gestiona mediante control de versiones (Git) y se publica en un repositorio privado o público (GitHub) para su seguimiento y mantenimiento.

H6. Se asume que los usuarios de la biblioteca (es decir, los desarrolladores que la integren en sus proyectos) cuentan con conocimientos básicos de programación web y del entorno Node.js, lo que les permite instalar y ejecutar la solución sin necesidad de asistencia adicional.

H7. Se considera que la infraestructura de red utilizada permite conexiones persistentes y bidireccionales sin bloqueos por parte de firewalls o proxies intermedios.

3.10.2 Restricciones

R1. El desarrollo debe realizarse de forma individual y en un tiempo limitado, correspondiente al calendario académico del Trabajo de Fin de Grado.

R2. Solo se utilizarán herramientas y bibliotecas de código abierto, evitando cualquier dependencia de software con licencias comerciales.

R3. No se implementarán mecanismos de seguridad de nivel de producción, como el uso de cifrado TLS/WSS ni sistemas de autenticación basados en tokens (JWT), ya que el proyecto se ejecuta en un entorno de prueba controlado (localhost).

R4. No se contempla la escalabilidad horizontal del sistema ni su despliegue en entornos distribuidos con balanceo de carga.

R5. El sistema debe ser funcional, pero no se exige una interfaz gráfica final orientada al usuario general, ya que el foco está en la lógica del servidor y su capacidad de integración.

R6. El soporte para navegadores antiguos o no estándar no está garantizado.

R7. La solución no contempla la integración con bases de datos externas ni la persistencia de datos a largo plazo.

R8. El sistema está diseñado para funcionar en entornos compatibles con Node.js v18+ y no se garantiza su funcionamiento en otros entornos de ejecución.

R9. La biblioteca se ha validado principalmente con la API de Yahoo Finance, por lo que su integración con otras APIs puede requerir adaptaciones adicionales.

3.11 Estudios de alternativas y viabilidad

Para abordar el diseño de una solución que permita la integración de comunicación en tiempo real con una API REST existente, se estudiaron diversas alternativas tecnológicas y arquitectónicas. El objetivo fue identificar el enfoque que ofreciera el mejor equilibrio entre simplicidad, rendimiento, extensibilidad y facilidad de integración.

3.11.1.1 *Alternativa 1: Polling tradicional*

Descripción:

Consiste en realizar peticiones periódicas al servidor (por ejemplo, cada N segundos) para comprobar si existen cambios en los datos. Es una técnica simple, ampliamente utilizada en aplicaciones web.

Ventajas:

- Fácil de implementar tanto en cliente como en servidor.
- Compatible con cualquier servidor HTTP.
- No requiere cambios significativos en la infraestructura existente.

Inconvenientes:

- Genera un gran número de peticiones innecesarias si no hay cambios.
- Aumenta la carga del servidor y el consumo de ancho de banda.
- Introduce latencia artificial entre eventos y notificaciones.
- Escalabilidad limitada con muchos usuarios concurrentes.

3.11.1.2 *Alternativa 2: Long Polling*

Descripción:

El cliente realiza una petición que se mantiene abierta hasta que el servidor tiene datos disponibles para responder. Al completarse, el cliente lanza inmediatamente una nueva petición.

Ventajas:

- Menor latencia que el *polling* tradicional.
- Aumenta la eficiencia de la red al reducir el número de peticiones vacías.

Inconvenientes:

- Sigue siendo un enfoque basado en HTTP, con necesidad de mantener múltiples conexiones abiertas.
- Complica la lógica en servidor y cliente.
- No es ideal para escenarios con actualizaciones muy frecuentes o muchos clientes.

3.11.1.3 *Alternativa 3 (Seleccionada): WebSocket*

Descripción:

Protocolo que establece una conexión persistente y bidireccional entre cliente y servidor, permitiendo el intercambio de mensajes en tiempo real sin necesidad de nuevas solicitudes HTTP.

Ventajas:

- Comunicación eficiente, persistente y en tiempo real.
- Bidireccional: el servidor puede enviar datos de forma proactiva y recibir comandos del cliente.
- Bajo consumo de recursos y excelente escalabilidad.
- Ideal para arquitecturas modernas que requieren interactividad y respuesta inmediata.

Inconvenientes:

- Requiere una infraestructura adicional en el servidor para gestionar el canal WebSocket.
- No se adapta directamente al modelo RESTful, por lo que se requiere una capa adicional o proxy.

3.11.1.4 Alternativa 4: Server-Sent Events (SSE)

Descripción:

SSE (*Server-Sent Events*) es una tecnología que permite que un servidor envíe actualizaciones automáticas a los clientes a través de una única conexión HTTP unidireccional. A diferencia del *polling*, el cliente no tiene que realizar peticiones continuas: la conexión permanece abierta y el servidor puede enviar datos en tiempo real cuando haya cambios.

Ventajas:

- Simplicidad de implementación en el lado del cliente (compatible con la interfaz EventSource [19] en navegadores modernos).
- Utiliza HTTP estándar, lo que facilita su integración con infraestructuras existentes.
- Requiere menos sobrecarga que el *polling* tradicional.

Inconvenientes:

- Es unidireccional: solo el servidor puede enviar mensajes al cliente (no hay canal de retorno).
- Menor flexibilidad que WebSocket en aplicaciones que requieren comunicación interactiva bidireccional [20].
- Compatibilidad limitada con algunos navegadores o escenarios corporativos con proxies restrictivos.

3.11.2 Conclusión del análisis de viabilidad

Tras comparar las distintas alternativas, se seleccionó WebSocket como la base para la solución propuesta debido a su eficiencia, flexibilidad y capacidad de comunicación bidireccional en tiempo real.

Aunque su integración requiere más esfuerzo inicial que técnicas como *polling*, *long polling* o Server-Sent Events (SSE), ofrece un rendimiento superior y una mejor experiencia de usuario en aplicaciones con requisitos dinámicos e interacción en ambas direcciones.

SSE, si bien es una opción ligera y adecuada para notificaciones unidireccionales, presenta limitaciones importantes cuando se requiere comunicación interactiva cliente-servidor.

La implementación mediante una biblioteca modular basada en WebSocket, que actúe como proxy inverso adaptable a múltiples APIs HTTP, ofrece una solución genérica y reutilizable, alineada con los principios de escalabilidad y mantenimiento del desarrollo moderno de software.

3.12 Descripción de la solución propuesta

La solución propuesta consiste en el desarrollo de una **biblioteca reutilizable basada en WebSocket** que actúa como una capa de comunicación en tiempo real para complementar cualquier API REST tradicional basada en HTTP. Esta biblioteca permite que los sistemas existentes (sin modificar su lógica interna) puedan emitir notificaciones automáticas a los clientes cuando ciertos recursos o estados cambian.

La arquitectura planteada se basa en los siguientes principios clave:

- **Modularidad y orientación a objetos**

El sistema está estructurado en módulos independientes con responsabilidades bien definidas. Esto facilita la mantenibilidad, la reutilización del código y la extensión del sistema a nuevos contextos o APIs sin necesidad de reescribir la lógica principal.

- **Patrón Reverse Proxy**

La biblioteca actúa como intermediaria entre el cliente y una API REST preexistente. Esta capa se encarga de redirigir las peticiones HTTP estándar y, de forma paralela, gestionar una conexión WebSocket persistente desde la que se emiten notificaciones cuando se detectan cambios.

- **Suscripción dinámica a estados**

Los clientes pueden conectarse al servidor WebSocket y suscribirse a ciertos "estados" o recursos (por ejemplo, cotización de una acción bursátil). Cuando el valor asociado a ese estado cambia, se envía automáticamente una notificación al cliente suscrito.

- **Adaptador para fuentes externas**

Para facilitar la integración con APIs heterogéneas, se utiliza el patrón Adapter. Este patrón permite adaptar cualquier fuente de datos externa (como la API de Yahoo Finance) al formato y métodos esperados por la biblioteca (`getStateNames()`, `getStateValue()`).

- **Interfaz de visualización básica**

Aunque el foco del proyecto es el *backend*, se ha desarrollado una interfaz web mínima con **Chart.js** y **Tailwind CSS** que permite visualizar los datos actualizados en tiempo real, comprobando el funcionamiento completo de la solución.

- **Soporte a múltiples clientes concurrentes**

El sistema está diseñado para permitir la conexión simultánea de múltiples clientes, con gestión eficiente de suscripciones, conexiones y desconexiones. Las notificaciones se envían exclusivamente a los clientes interesados en un determinado estado.

- **Pruebas, documentación y código abierto**

La solución incluye pruebas unitarias e integradas para validar su funcionamiento. El código se encuentra disponible en un repositorio público de GitHub y está acompañado de manuales de instalación, uso y descripción técnica

Resumen de características clave:

- Comunicación bidireccional y persistente mediante WebSocket.
- Compatible con cualquier API REST HTTP existente.
- Subsistema de suscripción y notificación de eventos en tiempo real.
- Modularidad basada en patrones de diseño: Observer, Adapter, Singleton, Abstract Factory, Flyweight, Iterator, Facade.
- Adaptador de ejemplo para Yahoo Finance.
- Arquitectura extensible, reutilizable y documentada.

3.13 Análisis de Riesgos

En todo proyecto de desarrollo software, especialmente cuando se trabaja de forma individual y con recursos limitados, es fundamental identificar los riesgos potenciales que pueden afectar al resultado final. Este análisis permite anticiparse a posibles problemas y definir estrategias de mitigación que reduzcan su impacto en los objetivos del proyecto.

A continuación se presenta una tabla con los principales riesgos identificados, su evaluación en términos de **probabilidad** e **impacto**, así como las medidas previstas para su mitigación:

ID	Riesgo	Probabilidad	Impacto	Medida de Mitigación
R1	Dificultad técnica en la integración de WebSocket con APIs externas	Media	Alta	Comenzar con una API pública sencilla (Yahoo Finance) como prueba de concepto; uso del patrón Adapter para desacoplar lógica.
R2	Limitaciones de tiempo para completar todas las fases del proyecto	Alta	Media	Definir tareas prioritarias, aplicar metodología incremental e iterativa (RUP) y establecer hitos realistas.
R3	Cambios en la API externa durante el desarrollo (por ejemplo, Yahoo Finance)	Media	Media	Diseñar el sistema para ser fácilmente configurable y sustituible con nuevas fuentes de datos.
R4	Possible inestabilidad o fallos del servidor durante pruebas de concurrencia	Media	Alta	Realizar pruebas de carga tempranas y aplicar mecanismos de control de errores y reconexión en WebSocket.
R5	Complejidad en la gestión de múltiples conexiones concurrentes	Media	Alta	Usar estructuras de datos eficientes y patrones como Observer y Singleton para centralizar el estado.
R6	Problemas de compatibilidad con navegadores antiguos	Baja	Baja	Definir desde el inicio que el sistema está orientado a navegadores modernos con soporte de WebSocket.
R7	Fallos durante la documentación técnica del sistema	Baja	Media	Documentar desde el inicio cada módulo y aplicar herramientas de control de versiones (Git) para seguimiento continuo.

Conclusión:

La mayoría de los riesgos identificados pueden gestionarse adecuadamente mediante planificación anticipada, buenas prácticas de desarrollo y un diseño modular del sistema. Al ser un proyecto académico sin cliente externo, el impacto económico es limitado, pero se ha dado especial importancia a la calidad técnica y a la completitud de la documentación para garantizar el valor formativo del trabajo.

3.14 Organización y gestión del proyecto

La organización del proyecto se ha basado en una planificación individual, estructurada en fases bien definidas y adaptada a los requisitos y plazos establecidos por el Trabajo de Fin de Grado. La gestión se ha llevado a cabo de forma iterativa, siguiendo principios del modelo RUP, con entregables parciales que han permitido validar progresivamente los avances realizados.

Roles y responsabilidades

- **Autor del proyecto:** responsable del análisis, diseño, implementación, pruebas, documentación y gestión integral del proyecto.
- **Tutor académico:** encargado de supervisar el trabajo, realizar revisiones periódicas y proponer mejoras tanto en el enfoque técnico como en la redacción de la memoria.

Directrices de trabajo

- **Gestión de cambios:** cualquier modificación significativa en los objetivos o la estructura del proyecto ha sido previamente discutida con el tutor y justificada en el marco del plan original.
- **Seguimiento:** se han mantenido reuniones periódicas con el tutor para evaluar el progreso, resolver dudas técnicas y ajustar prioridades.
- **Recopilación y distribución de información:** toda la documentación técnica, bibliografía y notas de trabajo se han organizado de forma estructurada en repositorios locales y en la plataforma GitHub.
- **Comunicación:** la comunicación entre tutor y estudiante se ha realizado principalmente por correo electrónico y reuniones presenciales o virtuales en fechas clave del calendario académico.
- **Validación de entregables:** cada bloque funcional del proyecto (análisis, desarrollo, pruebas, documentación) se ha revisado de forma individual para garantizar su calidad antes de avanzar a la siguiente fase.

Entorno de trabajo

El desarrollo se ha llevado a cabo en un entorno local controlado, utilizando un único equipo personal con las herramientas necesarias preinstaladas. Todo el trabajo se ha desarrollado de manera individual, dentro del marco temporal asignado al Trabajo de Fin de Grado.

Tabla 1. Roles y responsabilidades del proyecto

Rol	Persona asignada	Responsabilidades principales
Autor	Luis Miguel Gómez del Cueto	Análisis, diseño, implementación, pruebas, redacción de la memoria, gestión de versiones.
Tutor académico	Ángel José Riesgo Martínez	Revisión técnica, orientación metodológica, propuestas de mejora, seguimiento del proyecto.

3.15 Planificación temporal

El proyecto se ha planificado desde el 20 de febrero hasta el 1 de noviembre, con una primera etapa intensiva de trabajo técnico hasta julio, y una segunda etapa centrada en revisiones, correcciones y preparación de la entrega. El total estimado es de 300 horas, distribuidas entre las distintas fases del proyecto de forma proporcional a su complejidad y carga de trabajo.

Distribución de fases y horas

Fase	Fecha estimada	Duración aproximada	Horas asignadas
Análisis y documentación inicial	20 feb - 10 mar	3 semanas	30 h
Diseño de la arquitectura	11 mar - 24 mar	2 semanas	30 h
Implementación de la biblioteca (servidor)	25 mar - 21 abr	4 semanas	50 h
Implementación del cliente + integración	22 abr - 24 may	5 semanas	60 h
Pruebas y validación técnica	25 may - 14 jun	3 semanas	30 h
Redacción de la memoria	15 jun - 1 jul	2 semanas	80 h
Correcciones, revisión final y entrega	2 jul - 1 nov	4 meses	20 h
Total	20 feb - 1 nov	8,5 meses	300 h

Enfoque de trabajo

La planificación ha buscado concentrar el desarrollo técnico en el primer semestre, de modo que el sistema esté completamente funcional y documentado al finalizar junio. Desde julio hasta la entrega en noviembre, el tiempo se ha reservado para la revisión profunda de la memoria, integración de sugerencias del tutor, pulido del código y preparación de la presentación final.

Hitos clave

- **20 de febrero** - Inicio del proyecto
- **1 de julio** - Proyecto terminado (código y memoria completa)
- **Julio-Octubre** - Correcciones, mejoras y entrega final
- **1 de noviembre** - Entrega del TFG

Tabla 2. Diagrama de Gantt del proyecto

Fase	Feb	Mar	Abr	May	Jun	Jul	Ago	Sep	Oct	Nov
Análisis y documentación inicial	■	■								
Diseño de la arquitectura		■								
Implementación (biblioteca servidor)			■							
Implementación (cliente e integración)				■						
Pruebas y validación					■					
Redacción de la memoria					■	■				
Correcciones y revisión final					■	■	■	■	■	■

Leyenda: ■ = fase activa

3.16 Resumen del Presupuesto

Dado que el presente proyecto ha sido desarrollado en el marco de un Trabajo de Fin de Grado, no ha requerido de una inversión económica directa por parte de ninguna organización externa. Sin embargo, a efectos de estimar su viabilidad en un contexto real (por ejemplo, si se decidiera evolucionar esta solución hacia una biblioteca profesional para uso comercial), se presenta a continuación una estimación razonada de los costes asociados a su ejecución.

Coste total estimado: **3.750 €**

Desglose por partidas:

Concepto	Descripción	Coste estimado (€)
Horas de desarrollo	300 horas × 10 €/h (tarifa reducida para estimación académica)	3.000 €
Infraestructura	Uso de equipo personal, electricidad, internet (coste amortizado)	400 €
Software y licencias	Herramientas gratuitas (VS Code, Git, bibliotecas open source)	0 €
Servicios online	Alojamiento del repositorio en GitHub (versión gratuita)	0 €
Formación y documentación técnica	Consulta de documentación, tutoriales, artículos técnicos	0 €
Gastos imprevistos y material auxiliar	Posibles costes asociados a impresión, transporte, almacenamiento, etc.	350 €
TOTAL		3.750 €

Consideraciones

- El coste de desarrollo se ha estimado con una tarifa simbólica de 10 €/hora, representativa del valor de una dedicación técnica básica.
- No se han incluido licencias comerciales, ya que todo el software empleado es de código abierto o gratuito para uso académico.
- En un entorno profesional, este coste podría duplicarse o triplicarse dependiendo de los requisitos adicionales (seguridad, soporte, integración, mantenimiento, etc.).

Esta estimación permite valorar que el proyecto, en su forma actual, puede ser ejecutado con un presupuesto moderado, y sería perfectamente viable para una organización que desee desarrollar una biblioteca similar para uso interno o distribución libre. En caso de ampliarse hacia una versión 2.0 con nuevas funcionalidades o mayor escalabilidad, se requeriría una planificación presupuestaria adicional.

4 Análisis y Diseño del sistema

4.1 Documentación de entrada

La documentación de entrada reúne toda la información preliminar que ha servido de base para la planificación, diseño e implementación del proyecto. Esta documentación incluye tanto los requisitos generales del Trabajo de Fin de Grado como referencias técnicas y fuentes externas utilizadas para definir el contexto, justificar decisiones tecnológicas y orientar el desarrollo de la solución propuesta.

Documentos institucionales y académicos

- **Normativa del Trabajo de Fin de Grado de la Escuela de Ingeniería Informática** de la Universidad de Oviedo: documento que define los requisitos formales, estructura, plazos y objetivos del TFG.
- **Guía de estilo y plantilla para la memoria técnica del TFG**: utilizada como referencia para la organización y redacción del presente documento.
- **Norma UNE 157801**: especificación aplicada a la estructura y calidad de la documentación técnica en proyectos de ingeniería del software.

Fuentes técnicas y referencias externas

- **RFC 6455 - The WebSocket Protocol**: especificación oficial del protocolo WebSocket, publicada por el IETF (Internet Engineering Task Force).
- **Documentación oficial de Node.js, Express.js**: utilizada para la implementación del servidor y la API WebSocket.
- **Documentación de la API Yahoo Finance (yahoo-finance2)**: fuente de datos seleccionada como caso de uso para la demostración del funcionamiento de la biblioteca.
- **Manual de patrones de diseño “Design Patterns: Elements of Reusable Object-Oriented Software” (GoF)**: referencia para la implementación de los patrones Adapter, Observer, Abstract Factory y Singleton en el proyecto.

Herramientas y configuraciones previas

- Instalación y configuración del entorno de desarrollo (Node.js, VS Code, Git).
- Inicialización de un repositorio en GitHub para control de versiones.
- Configuración de un entorno de pruebas en localhost para la ejecución y verificación del sistema.

Resumen:

La documentación de entrada ha sido clave para orientar correctamente el planteamiento del proyecto, garantizar el cumplimiento de los requisitos académicos, y seleccionar las herramientas más adecuadas desde el punto de vista técnico.

4.2 Análisis

El presente apartado describe el análisis funcional del sistema propuesto, su estructura modular y los principales elementos de diseño empleados en la construcción de la solución. Se parte de los requisitos definidos previamente para identificar los componentes clave del sistema, sus interacciones y su organización en subsistemas.

4.2.1 Identificación de Subsistemas

El sistema ha sido dividido en los siguientes subsistemas principales:

- **Subsistema WebSocket:** encargado de gestionar las conexiones persistentes con los clientes, recibir suscripciones y enviar notificaciones.
- **Subsistema REST/HTTP:** actúa como proxy inverso para redirigir las peticiones del cliente a la API original (Yahoo Finance).
- **Subsistema de Adaptación:** implementa el patrón Adapter para traducir la interfaz de la API externa a la esperada por el sistema.
- **Subsistema de Gestión de Estado:** controla los valores monitorizados y notifica a los clientes cuando hay cambios relevantes.
- **Subsistema Cliente Web (simplificado):** permite la visualización en tiempo real de los datos utilizando tecnologías *frontend* modernas (Chart.js, Tailwind CSS).

4.2.2 Interfaces entre subsistemas

- El Subsistema WebSocket **comunica con el Subsistema de Gestión de Estado mediante eventos y observadores.**
- El Subsistema de Adaptación **accede a datos externos (Yahoo Finance) y los transforma para el uso interno.**
- El Subsistema Cliente Web **se conecta a WebSocket y muestra los datos recibidos, sin acceso directo a la API REST.**

4.2.3 Casos de uso principales

A continuación se describen los casos de uso principales del sistema desde la perspectiva del cliente que interactúa con la biblioteca a través de WebSocket. Estos casos están orientados a la suscripción a recursos dinámicos, como precios de acciones, y su actualización en tiempo real.

4.2.3.1 Suscripción a estado

Descripción:

El cliente solicita al servidor comenzar a monitorizar un recurso identificado (por ejemplo, un activo financiero como AAPL). El servidor añade al cliente a la lista de suscriptores asociados a ese recurso.

Flujo básico:

1. El usuario selecciona un recurso desde la interfaz.
2. El cliente envía un mensaje WebSocket al servidor con una acción de tipo "subscribe" y el nombre del estado.
3. El servidor registra al cliente como interesado en ese recurso.
4. El servidor realiza un *polling* periódico a una API externa (por ejemplo, cada 5 segundos).
5. Si se detecta un cambio en el valor del recurso, se notifica inmediatamente a todos los clientes suscritos mediante mensajes WebSocket.

Precondiciones:

- El recurso solicitado debe estar disponible y monitorizable.
- El cliente debe tener conexión WebSocket activa.

Postcondiciones:

- El cliente comienza a recibir actualizaciones automáticas cuando cambia el valor del recurso.

Ejemplo: Suscripción exitosa

1. El cliente abre la interfaz web y elige un símbolo (por ejemplo, TSLA) desde el selector.
2. Se envía al servidor el siguiente mensaje WebSocket:
`{ "action": "subscribe", "stateName": "TSLA" }`
3. El servidor añade al cliente a la lista de suscriptores de TSLA.
4. El servidor consulta la API externa cada 5 segundos.
5. Cuando el valor de TSLA cambia, el servidor envía una notificación:
`{ "stateName": "TSLA", "newValue": 248.90 }`
6. El cliente actualiza automáticamente la interfaz con el nuevo valor.

4.2.3.2 Anulación de suscripción a estado

Descripción:

El cliente indica que ya no desea recibir actualizaciones de un determinado recurso. El servidor lo elimina de la lista de suscriptores.

Flujo básico:

1. El usuario deselecciona el recurso desde la interfaz.
2. El cliente envía un mensaje WebSocket al servidor con una acción de tipo "unsubscribe" y el nombre del estado.
3. El servidor elimina al cliente de la lista de suscriptores.
4. El cliente deja de recibir notificaciones de ese recurso.

Precondiciones:

- El cliente debe estar previamente suscrito al estado indicado.

Postcondiciones:

- El cliente ya no recibirá actualizaciones para el recurso cancelado.

Ejemplo: Cancelación exitosa

1. El usuario pulsa el botón de cancelar en un recurso previamente suscrito (TSLA).
2. Se envía al servidor el siguiente mensaje:
{ "action": "unsubscribe", "stateName": "TSLA" }
3. El servidor elimina al cliente de la lista de suscriptores de TSLA.
4. A partir de ese momento, no se envían más notificaciones al cliente sobre dicho estado.

4.2.4 Análisis de Clases en la Fase de Análisis

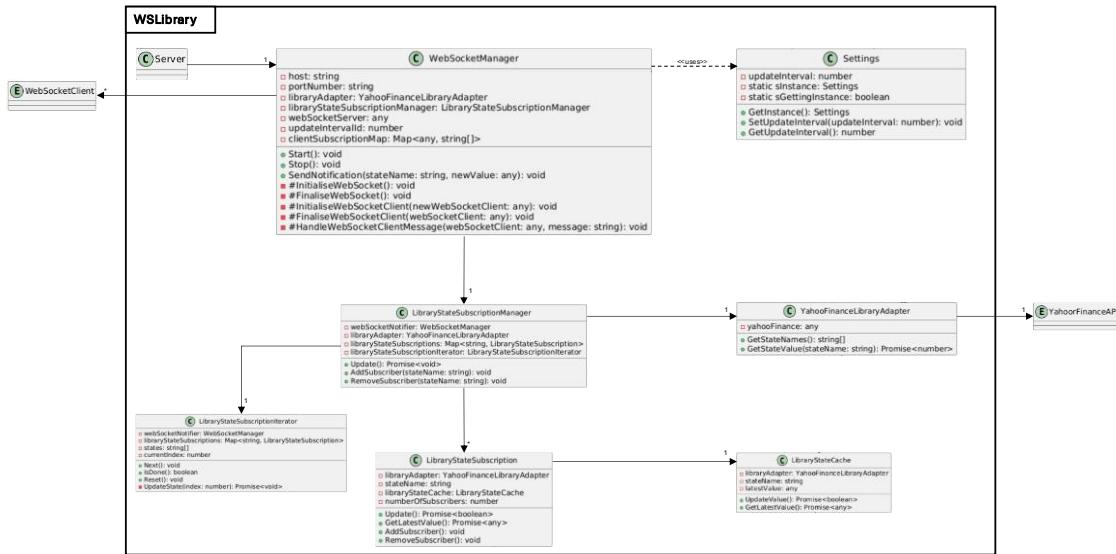


Figura 12. UML del sistema

Este sistema se compone de dos paquetes principales: **client**, que implementa la interfaz gráfica para el usuario, y **server**, que proporciona la lógica de *backend* para gestionar suscripciones WebSocket y comunicar con Yahoo Finance.

4.2.4.1 Paquete 1: client

Contiene el *frontend* web de la aplicación, con lógica JavaScript, estructura HTML y estilos CSS. Permite suscribirse en tiempo real a precios de acciones, criptomonedas o divisas, y mostrarlos gráficamente.

4.2.4.1.1 index.html

Descripción:

Define la estructura de la interfaz: menús desplegables de categorías y símbolos, secciones para gráficos y una consola visual de eventos WebSocket.

Responsabilidades:

- Presentar al usuario los símbolos disponibles organizados por categoría.
- Visualizar los precios actualizados y gráficos interactivos.
- Permitir cambio de tema (claro/oscuro).

4.2.4.1.2 styles.css

Descripción:

Define estilos personalizados para elementos clave como precios y consola.

Responsabilidades:

- Aplicar colores según tendencia del precio.

- Estilizar la consola WebSocket.
- Asegurar coherencia entre modo claro y oscuro.

4.2.4.1.3 app.js

Descripción:

Controla la lógica del cliente, incluyendo conexión WebSocket, suscripciones, gráficos con Chart.js, y respuesta a eventos.

Responsabilidades:

- Conectarse al servidor WebSocket.
- Gestionar suscripciones a símbolos y actualizar la interfaz.
- Dibujar y actualizar gráficos.
- Sincronizar estado visual con modo de color.
- Mostrar consola de mensajes WebSocket.

Elementos clave:

- subscribedSymbols: Lista activa de suscripciones.
- priceHistory: Histórico por símbolo para actualizar los gráficos.
- toggleTheme(), subscribe(), unsubscribe().

4.2.4.2 Paquete 2: server

4.2.4.2.1 WSLibrary.js

Descripción:

Define un espacio de nombres global como contenedor para todas las clases del servidor.

Responsabilidad:

- Prevenir colisiones de nombres.
- Actuar como punto de organización del código.

4.2.4.2.2 YahooFinanceLibraryAdapter.js

Descripción:

Adaptador para conectarse con la API yahoo-finance2. Expone los símbolos disponibles y permite obtener precios actuales.

Responsabilidades:

- Obtener los nombres de estado (GetStateNames()).
- Consultar el valor actual de un símbolo (GetStateValue(stateName)).

4.2.4.2.3 LibraryStateCache.js

Descripción:

Gestiona la caché local de un símbolo concreto. Solo actualiza si cambia el valor.

Campos clave:

- #latestValue: Último valor conocido.
- #libraryAdapter: Acceso a datos en Yahoo Finance.

Responsabilidades:

- Consultar el valor actual.
- Detectar cambios de precio (*UpdateValue()*).

4.2.4.2.4 LibraryStateSubscription.js

Descripción:

Representa una suscripción concreta a un símbolo. Guarda el número de suscriptores y decide cuándo se debe mantener o liberar la caché.

Campos clave:

- #numberOfSubscribers: Cuenta de suscriptores activos.
- #libraryStateCache: Referencia al objeto *LibraryStateCache*.

Responsabilidades:

- Añadir o eliminar suscriptores (*AddSubscriber*, *RemoveSubscriber*).
- Consultar o actualizar valor solo si hay suscriptores.

4.2.4.2.5 LibraryStateSubscriptionIterator.js

Descripción:

Clase que implementa el patrón *Iterator* para recorrer las suscripciones (*LibraryStateSubscription*) gestionadas por el Manager. Desacopla la lógica de iteración del *LibraryStateSubscriptionManager* y optimiza la actualización en ciclos controlados.

Responsabilidades:

- Recorrer de forma segura y eficiente el mapa de suscripciones, invocando la actualización de cada estado y notificando los cambios a través del *WebSocketManager*.

4.2.4.2.6 LibraryStateSubscriptionManager.js

Descripción:

Gestor central de todas las suscripciones. Contiene una instancia por cada símbolo.

Responsabilidades:

- Mantener el mapa { símbolo → *LibraryStateSubscription* }.

- Notificar a los clientes vía WebSocket cuando cambia un valor.
- Añadir o quitar suscriptores a través del método correspondiente.

4.2.4.2.7 Settings.js

Descripción:

Clase *Singleton* que contiene parámetros globales del sistema.

Responsabilidades:

- Definir el intervalo de actualización (*GetUpdateInterval()*).
- Permitir ajustes seguros mediante una única instancia global.

4.2.4.2.8 WebSocketManager.js

Descripción:

Controla el servidor WebSocket, gestiona clientes conectados y su relación con los símbolos suscritos.

Responsabilidades:

- Recibir conexiones entrantes (*#InitialiseWebSocketClient*).
- Procesar mensajes *subscribe* y *unsubscribe*.
- Orquestar la invocación periódica a *LibraryStateSubscriptionManager.Update()* para la detección de cambios de estado.
- Invocar periódicamente la actualización de estados.

Estructura de datos clave:

- *#clientSubscriptionMap*: Mapa cliente → símbolos suscritos.
- *#libraryStateSubscriptionManager*: Interfaz con el sistema de suscripciones.

4.2.4.2.9 Server.js

Descripción:

Punto de entrada de la aplicación. Inicializa el adaptador, la configuración y el *WebSocketManager*.

Responsabilidades:

- Cargar todos los módulos del servidor.
- Iniciar el servidor en el puerto 3000.

4.2.5 Análisis de Interfaces de Usuario

4.2.5.1 Descripción de la Interfaz

El sistema cuenta con una **interfaz web responsive** desarrollada con HTML, Tailwind CSS y JavaScript. El objetivo principal es permitir a los usuarios visualizar de forma clara y organizada los precios en tiempo real de activos financieros, agrupados en tres categorías: acciones, criptomonedas y divisas.

La interfaz se divide en las siguientes zonas de trabajo:

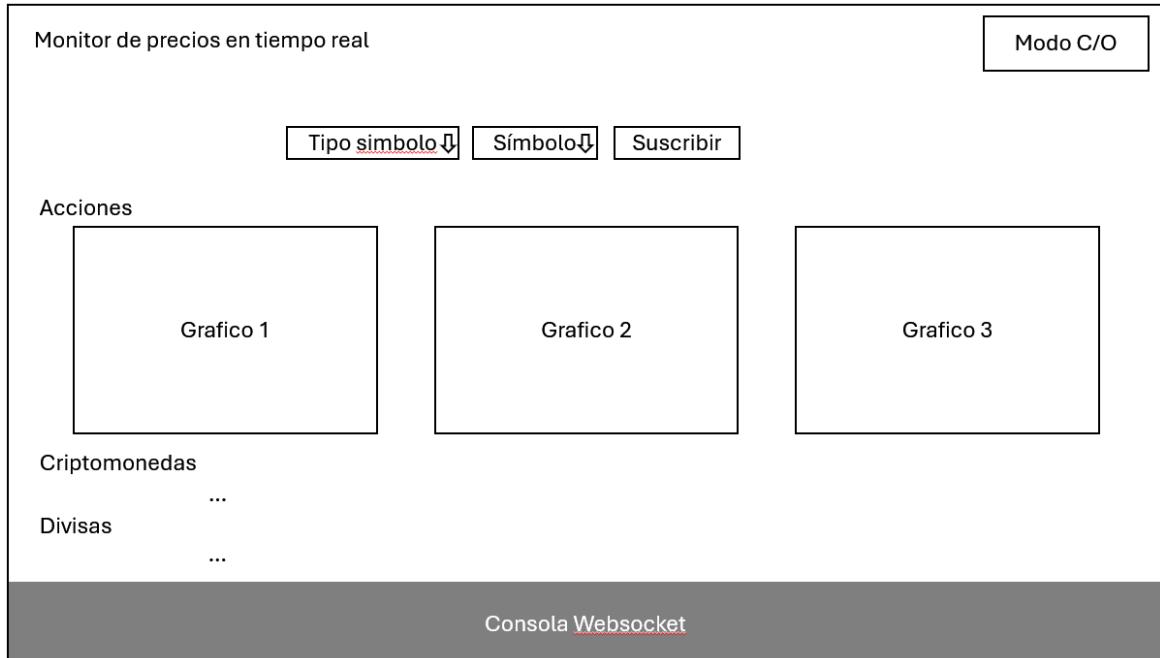
- **Menú superior con selector de tema** (claro/oscuro).
- **Selector de categoría** (stocks, crypto, forex) y selector de símbolo.
- **Botón de suscripción** que permite añadir nuevos símbolos.
- **Sección de visualización de precios** para cada símbolo suscrito:
 - Nombre del símbolo.
 - Precio actual con color codificado según la tendencia (verde, rojo, gris).
 - Gráfico de evolución temporal generado con Chart.js.
 - Botón para desuscribirse.
- **Consola WebSocket** que muestra eventos recibidos en tiempo real para depuración o interés del usuario avanzado.

Grupos de usuarios previstos:

- Usuarios interesados en el seguimiento de precios financieros en tiempo real.
- Estudiantes o docentes que desean entender el funcionamiento de sistemas WebSocket.
- Usuarios técnicos que deseen monitorizar múltiples activos sin necesidad de recargar la página.

Normas de usabilidad:

- Toda acción es reversible (puede desuscribirse en cualquier momento).
- Se emplean **colores consistentes y accesibles** para representar tendencias.
- El modo oscuro y claro asegura **comodidad visual** en distintos entornos.
- La interfaz **no requiere conocimientos técnicos**: es intuitiva, inmediata y visual.
- Se previene el error mediante la desactivación de botones si no hay símbolos disponibles.

Boceto de la interfaz:**Figura 13. Boceto de la interfaz****4.2.5.2 Descripción del Comportamiento de la Interfaz**

- Al cargar la página, el sistema se conecta automáticamente al servidor WebSocket.
- Se inicializan las listas de símbolos agrupadas por categoría.
- El usuario puede seleccionar un símbolo y suscribirse. El símbolo desaparece del selector tras ser suscrito.
- Por cada símbolo suscrito:
 - Se muestra una **sección individual** con su precio en tiempo real y un gráfico.
 - El gráfico se actualiza en vivo sin intervención del usuario.
- Si el usuario se desubscribe, la sección se elimina y el símbolo vuelve a estar disponible en el selector.
- Si se desconecta el WebSocket, la consola informará de ello.
- Se puede alternar entre temas claro/oscuro en todo momento.
- Se muestra una **consola visual** donde se registran mensajes de suscripción, actualización y desuscripción.

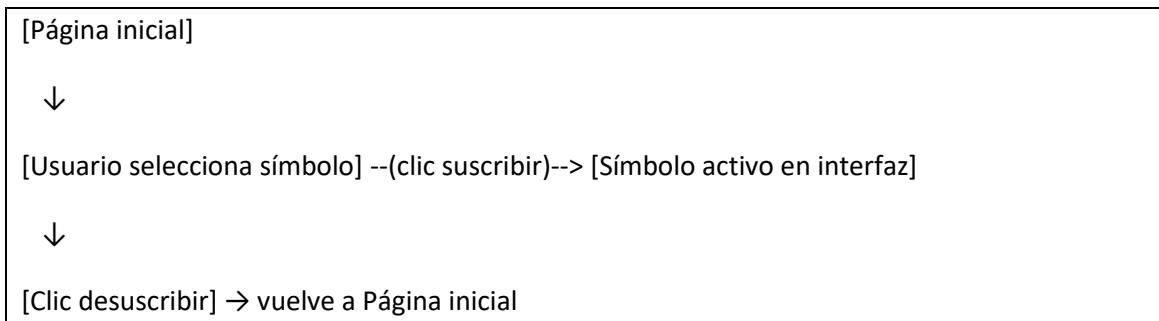
Validación y errores:

- No es posible suscribirse dos veces al mismo símbolo.
- No se permite seleccionar símbolos si ya están todos suscritos.

- Los mensajes JSON malformados desde el cliente no se procesan.
- El sistema informa mediante logs si ocurre un error de conexión o actualización.

Diagrama de navegación entre pantallas:

Como se trata de una aplicación de una sola página (SPA), no hay navegación entre múltiples vistas. Sin embargo, el comportamiento puede modelarse con el siguiente diagrama de transiciones entre estados de interfaz:



4.2.6 Especificación del Plan de Pruebas

El sistema se ha validado mediante pruebas de tres niveles:

4.2.6.1 Pruebas Unitarias

Se han probado de forma aislada los siguientes módulos clave:

Clase	Método	Entrada	Resultado Esperado
YahooFinanceLibraryAdapter	GetStateValue("AAPL")	Símbolo válido	Devuelve valor numérico
LibraryStateCache	UpdateValue()	Nuevo valor de precio	Devuelve true si ha cambiado, false si no
LibraryStateSubscription	AddSubscriber(), Remove()	Invocaciones repetidas	Mantiene recuento correcto de suscriptores
Settings	SetUpdateInterval(-1)	Valor inválido	Lanza excepción

4.2.6.2 Pruebas de Integración

Escenario	Entrada	Resultado Esperado
Cliente se suscribe a un símbolo válido	Mensaje JSON {"action":"subscribe"}	El servidor añade suscriptor y envía actualizaciones
Cliente se desuscribe de un símbolo	Mensaje JSON {"action":"unsubscribe"}	El servidor lo elimina del mapa y detiene notificación
Se actualiza el precio de un símbolo activo	Nuevo valor desde Yahoo Finance	Todos los clientes suscritos reciben update

4.2.6.3 Pruebas del Sistema

Caso de Prueba	Entrada	Resultado Esperado
Cliente inicia sesión en modo oscuro	Recarga con tema guardado	Se mantiene tema desde localStorage
Todos los símbolos suscritos	Usuario suscribe todos los activos posibles	El sistema bloquea nuevos intentos y desactiva botón
Desconexión del WebSocket	Cliente se desconecta	Se limpia su entrada en clientSubscriptionMap
Símbolo inválido	{"action":"subscribe", "stateName":"XYZ"}	Se ignora y se loguea advertencia

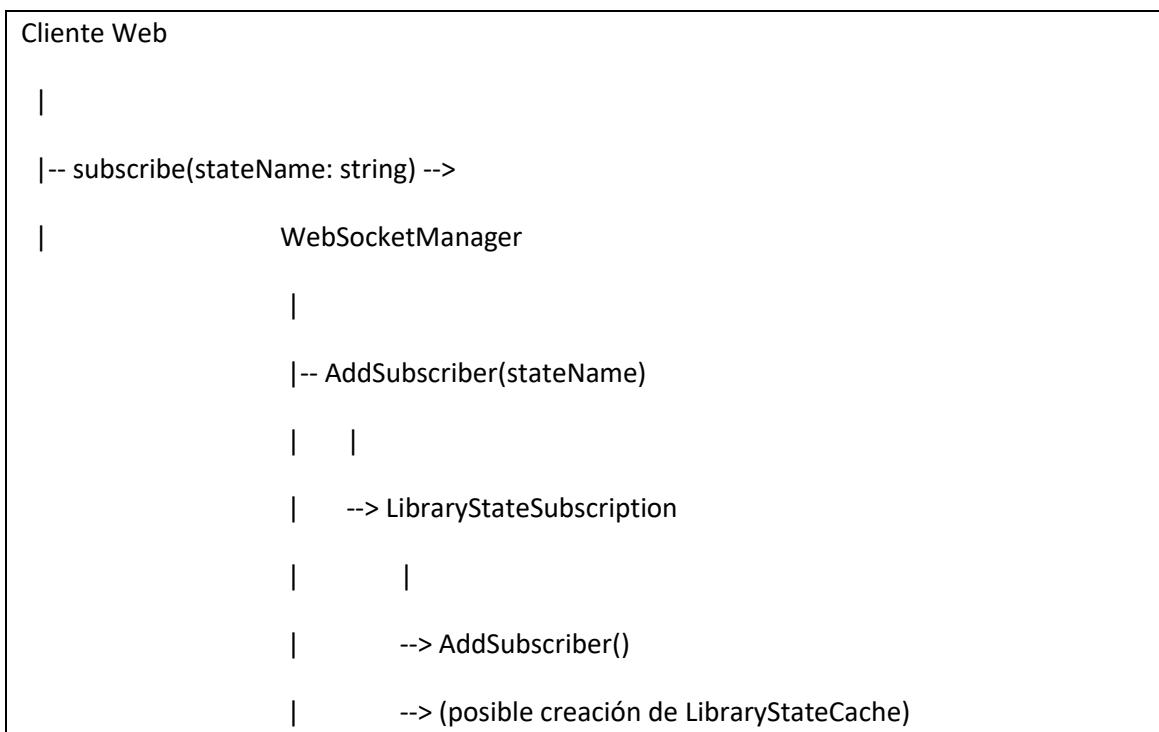
4.3 Diseño

4.3.1 Diseño de Casos de Uso Reales

4.3.1.1 Caso de Uso 1: Suscripción a un símbolo

4.3.1.1.1 Escenario 1.1: El usuario selecciona un símbolo y se suscribe a él para recibir actualizaciones de precio en tiempo real.

Diagrama de Secuencia: Suscripción



Descripción paso a paso:

1. **Cliente Web (JavaScript)** ejecuta la función subscribe(), que envía un mensaje JSON por WebSocket:

```
{ "action": "subscribe", "stateName": "AAPL" }
```

2. **WebSocketManager** recibe el mensaje, lo interpreta y llama a AddSubscriber(stateName) en LibraryStateSubscriptionManager.
3. En **LibraryStateSubscriptionManager**, se obtiene la instancia correspondiente de LibraryStateSubscription y se invoca AddSubscriber().
4. Si es el **primer suscriptor**, se crea internamente una instancia de LibraryStateCache para ese estado.
5. A partir de aquí, ese cliente recibirá notificaciones periódicas si cambia el valor del símbolo.

Diagrama de Estados: LibraryStateSubscription

```

[Sin subscriptores]
  |
AddSubscriber()
  ↓
[Con subscriptores]
  |
RemoveSubscriber()
  ↓
[Sin subscriptores] (y se elimina el cache)

```

4.3.1.2 Caso de Uso 2: Notificación de cambio de precio**4.3.1.2.1 Escenario 2.1: El precio de un símbolo ha cambiado, y el sistema debe notificarlo solo a los clientes suscritos.****Diagrama de Secuencia: Notificación de Cambio**

```

[Timer: cada X ms]
  |
--> WebSocketManager.#Update()
  |
--> LibraryStateSubscriptionManager.Update()
  |
--> LibraryStateSubscriptionIterator.Reset() (si IsDone())
  |
--> LibraryStateSubscriptionIterator.Next()
  |
--> LibraryStateSubscription.Update()
  |
--> LibraryStateCache.UpdateValue()
  |
--> YahooFinanceLibraryAdapter.GetStateValue()
  |
(sí ha cambiado)
--> WebSocketManager.SendNotification()

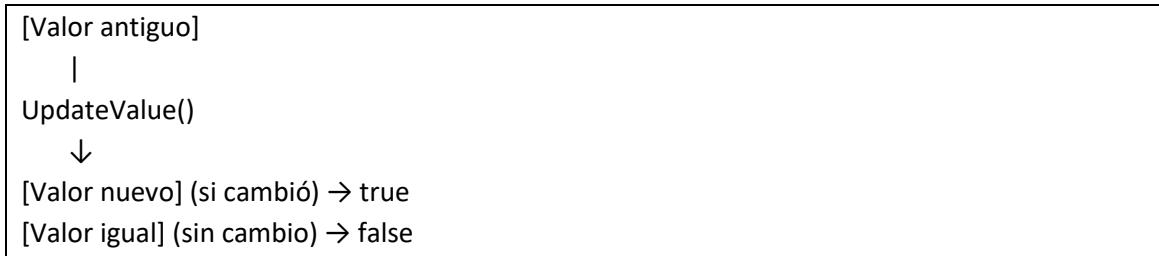
```

Descripción paso a paso:

1. Cada cierto tiempo (configurado en Settings, por defecto 5000 ms), WebSocketManager ejecuta #Update().
2. Este método invoca Update() sobre el LibraryStateSubscriptionManager, que a su vez recorre todas las suscripciones.
3. Para cada símbolo, si hay suscriptores, se llama a Update() en LibraryStateSubscription, que llama a UpdateValue() en LibraryStateCache.

4. LibraryStateCache consulta el valor con YahooFinanceLibraryAdapter.GetStateValue() y lo compara con el último.
5. Si el valor ha cambiado, se invoca SendNotification() en WebSocketManager, que envía el nuevo precio solo a los clientes suscritos a ese símbolo.

Diagrama de Estados: LibraryStateCache



Casos no desarrollados con diagrama

Los siguientes se comportan de forma similar al caso de uso 1 y 2, por lo que no requieren nuevos diagramas:

- **Desuscripción de símbolo:** Equivalente a unsubscribe → RemoveSubscriber() → eliminación de la suscripción si ya no quedan clientes.
- **Cambio de tema oscuro/claro:** Lógica solo de cliente; no requiere interacción con el servidor.
- **Conexión/desconexión WebSocket:** Ya está gestionada automáticamente al establecer la conexión en WebSocketManager.

4.3.2 Diseño de Clases

El diseño de clases del sistema es una evolución directa del análisis. Se ha mantenido una estructura organizada y modular, separando responsabilidades por clase, y dividiendo el sistema en dos paquetes principales: server y client.

El lado servidor está implementado en JavaScript con Node.js y contiene toda la lógica de negocio, suscripciones, caché y comunicación con Yahoo Finance.

El lado cliente es una interfaz web (HTML + JS + Chart.js + Tailwind) que permite al usuario visualizar en tiempo real los precios y gráficos de distintos activos financieros.

4.3.2.1 Diagrama de Clases

A continuación se muestra cada una de las clases. Todas las clases pertenecen al espacio de nombres WSLibrary (en el servidor).

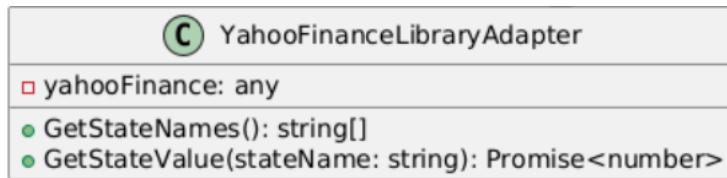


Figura 14. Clase YahooFinanceLibraryAdapter

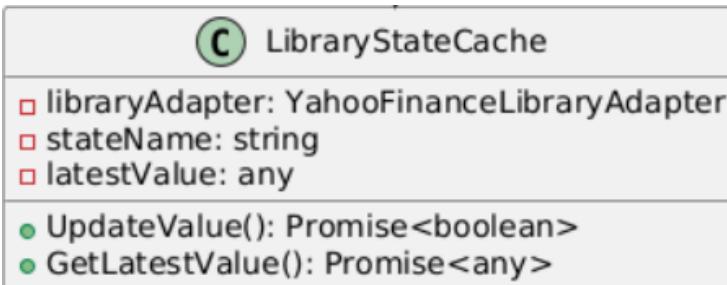


Figura 15. Clase LibraryStateCache

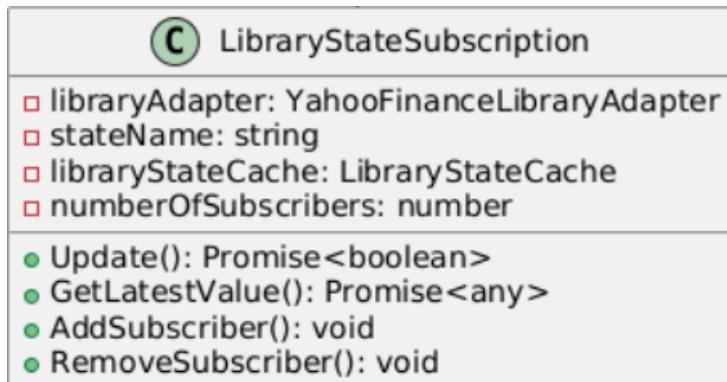


Figura 16. Clase LibraryStateSubscription

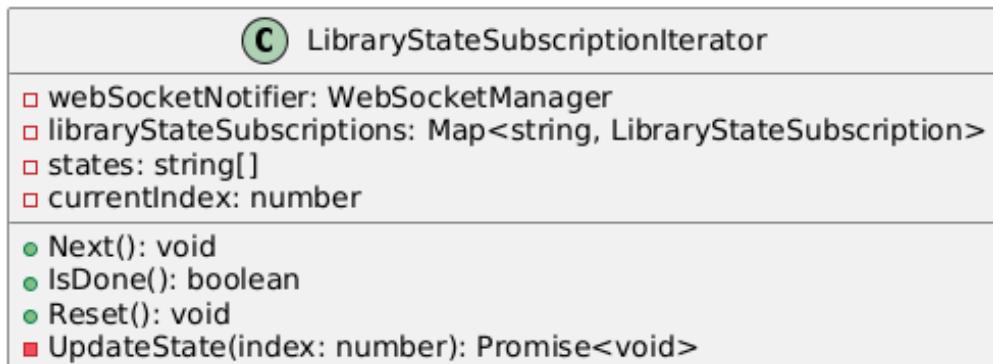


Figura 17. Clase LibraryStateSubscriptionIterator

C	LibraryStateSubscriptionManager
□	webSocketNotifier: WebSocketManager
□	libraryAdapter: YahooFinanceLibraryAdapter
□	libraryStateSubscriptions: Map<string, LibraryStateSubscription>
□	libraryStateSubscriptionIterator: LibraryStateSubscriptionIterator
●	Update(): Promise<void>
●	AddSubscriber(stateName: string): void
●	RemoveSubscriber(stateName: string): void

Figura 18. Clase LibraryStateSubscriptionManager

C	WebSocketManager
□	host: string
□	portNumber: string
□	libraryAdapter: YahooFinanceLibraryAdapter
□	libraryStateSubscriptionManager: LibraryStateSubscriptionManager
□	webSocketServer: any
□	updateIntervalId: number
□	clientSubscriptionMap: Map<any, string[]>
●	Start(): void
●	Stop(): void
●	SendNotification(stateName: string, newValue: any): void
■	#InitialiseWebSocket(): void
■	#FinaliseWebSocket(): void
■	#InitialiseWebSocketClient(newWebSocketClient: any): void
■	#FinaliseWebSocketClient(webSocketClient: any): void
■	#HandleWebSocketClientMessage(webSocketClient: any, message: string): void

Figura 19. Clase WebsocketManager

C	Settings
□	updateInterval: number
□	static sInstance: Settings
□	static sGettingInstance: boolean
●	GetInstance(): Settings
●	SetUpdateInterval(updateInterval: number): void
●	GetUpdateInterval(): number

Figura 20. Clase Settings

Relaciones entre clases

- **YahooFinanceLibraryAdapter** es usado por:
 - **LibraryStateCache**
 - **LibraryStateSubscription**
 - **LibraryStateSubscriptionManager**
- **LibraryStateCache** es compuesto por **LibraryStateSubscription** (agregación).
- **LibraryStateSubscriptionManager** contiene un mapa de objetos **LibraryStateSubscription**.

- **WebSocketManager** tiene una **referencia directa** a **LibraryStateSubscriptionManager** y a cada conexión de cliente (WebSocket).
- **WebSocketManager** actúa como **notificador** (implementa el método `SendNotification` llamado por el manager de suscripciones).
- **Settings** se accede como **singleton global**.

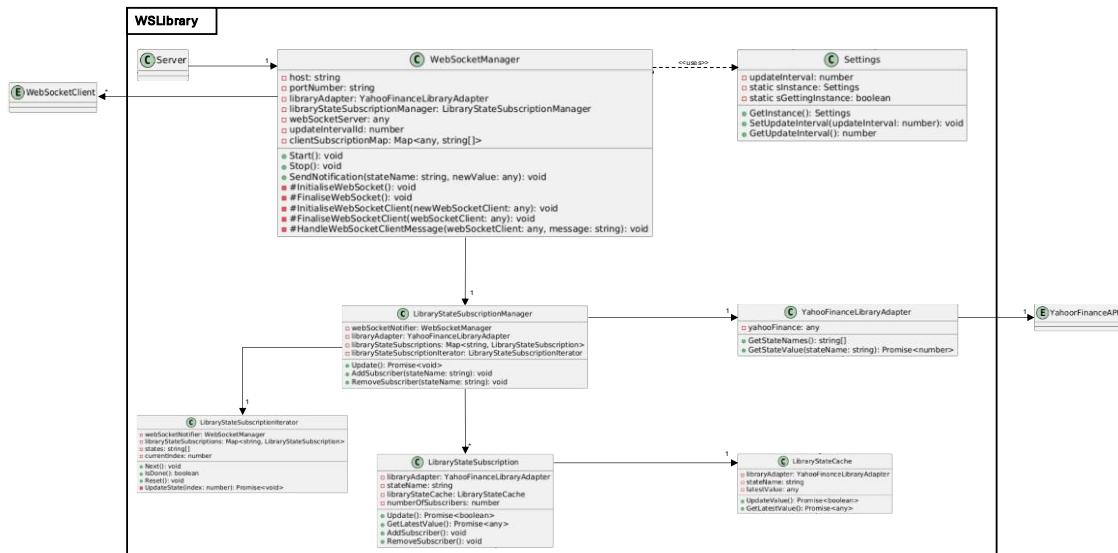
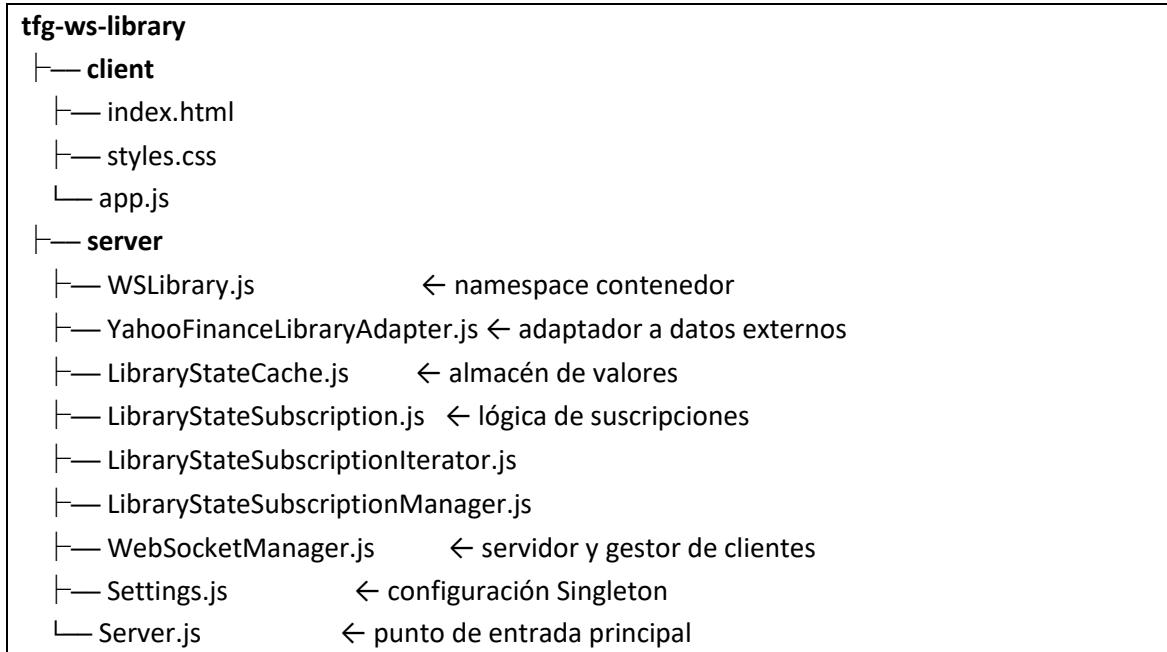


Figura 21. Diagrama de clases global

4.3.2.2 Jerarquía de Paquetes



Explicacion:

- El paquete **server** encapsula toda la lógica de *backend*. Cada fichero representa una clase.
- El paquete **client** contiene una interfaz web interactiva orientada al usuario final.
- El namespace **WSLibrary** actúa como envoltorio común para las clases del servidor.

Notas adicionales:

- Se ha optado por una estructura **modular y extensible**, en la que agregar nuevas fuentes de datos (por ejemplo, otra API de cotizaciones) requeriría únicamente implementar una nueva clase adaptador.
- El patrón **observer** se implementa implícitamente a través de suscripciones WebSocket.
- El diseño se ajusta a los principios de responsabilidad única, separación de capas y bajo acoplamiento.

4.3.3 Arquitectura del Sistema

El sistema se ha diseñado bajo una arquitectura cliente-servidor reactiva, basada en la comunicación asíncrona a través de WebSockets. Esto permite que el servidor actúe como publisher de información financiera en tiempo real, y el cliente como subscriber, evitando el uso de *polling* o peticiones periódicas.

La arquitectura se divide en dos grandes subsistemas:

4.3.3.1 Subsistema 1: Servidor WebSocket

Este subsistema está compuesto por el conjunto de clases del paquete server, desarrolladas en Node.js. Su función es:

- Mantener conexiones WebSocket activas con clientes.
- Interactuar con la API de Yahoo Finance mediante un adaptador (YahooFinanceLibraryAdapter).
- Gestionar suscripciones y cachear valores (LibraryStateCache, LibraryStateSubscription).
- Detectar cambios de precio y notificar únicamente a los clientes suscritos (WebSocketManager).

Clases implicadas:

- WebSocketManager: orquestador general del servidor.
- LibraryStateSubscriptionManager: coordina las suscripciones.
- LibraryStateSubscription: gestiona una suscripción concreta.
- LibraryStateCache: almacena último valor para cada símbolo.
- YahooFinanceLibraryAdapter: conecta con datos externos.
- Settings: almacén central de configuración.

4.3.3.2 Subsistema 2: Cliente Web

Este subsistema está compuesto por el paquete client, que representa la interfaz gráfica para el usuario. Permite suscribirse a activos, ver los precios actualizados y visualizar gráficamente su evolución.

Responsabilidades:

- Conectarse al servidor vía WebSocket.
- Enviar mensajes de suscripción/desuscripción.
- Recibir y visualizar precios en tiempo real.
- Dibujar gráficos con Chart.js.
- Adaptar la interfaz al tema claro u oscuro.

Archivos implicados:

- app.js: lógica de cliente, comunicación y gráficos.
- index.html: estructura visual.
- styles.css: diseño visual complementario (colores, consola).

4.3.3.3 Diagrama de la Arquitectura

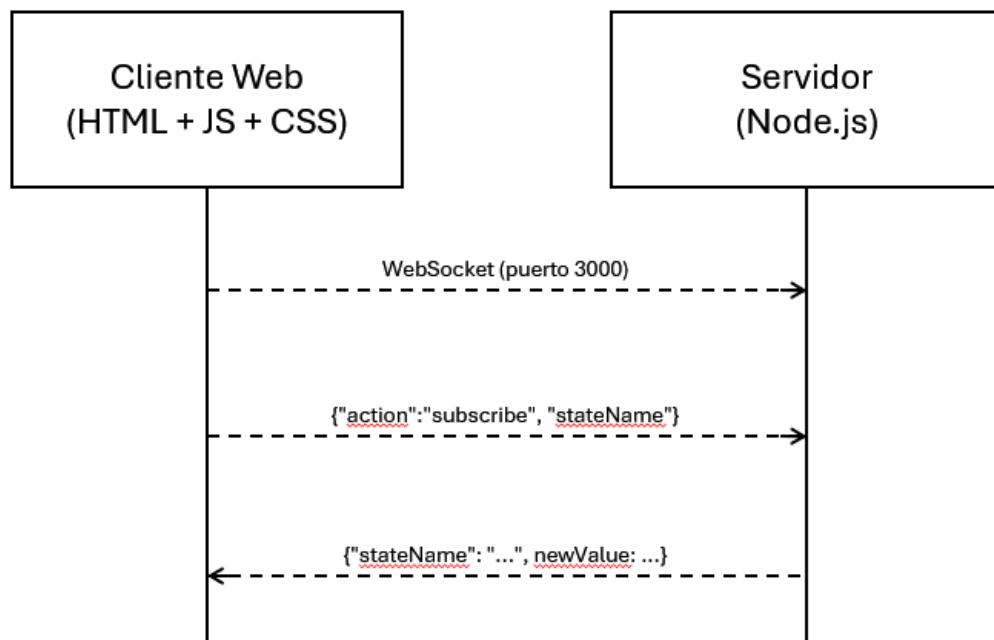
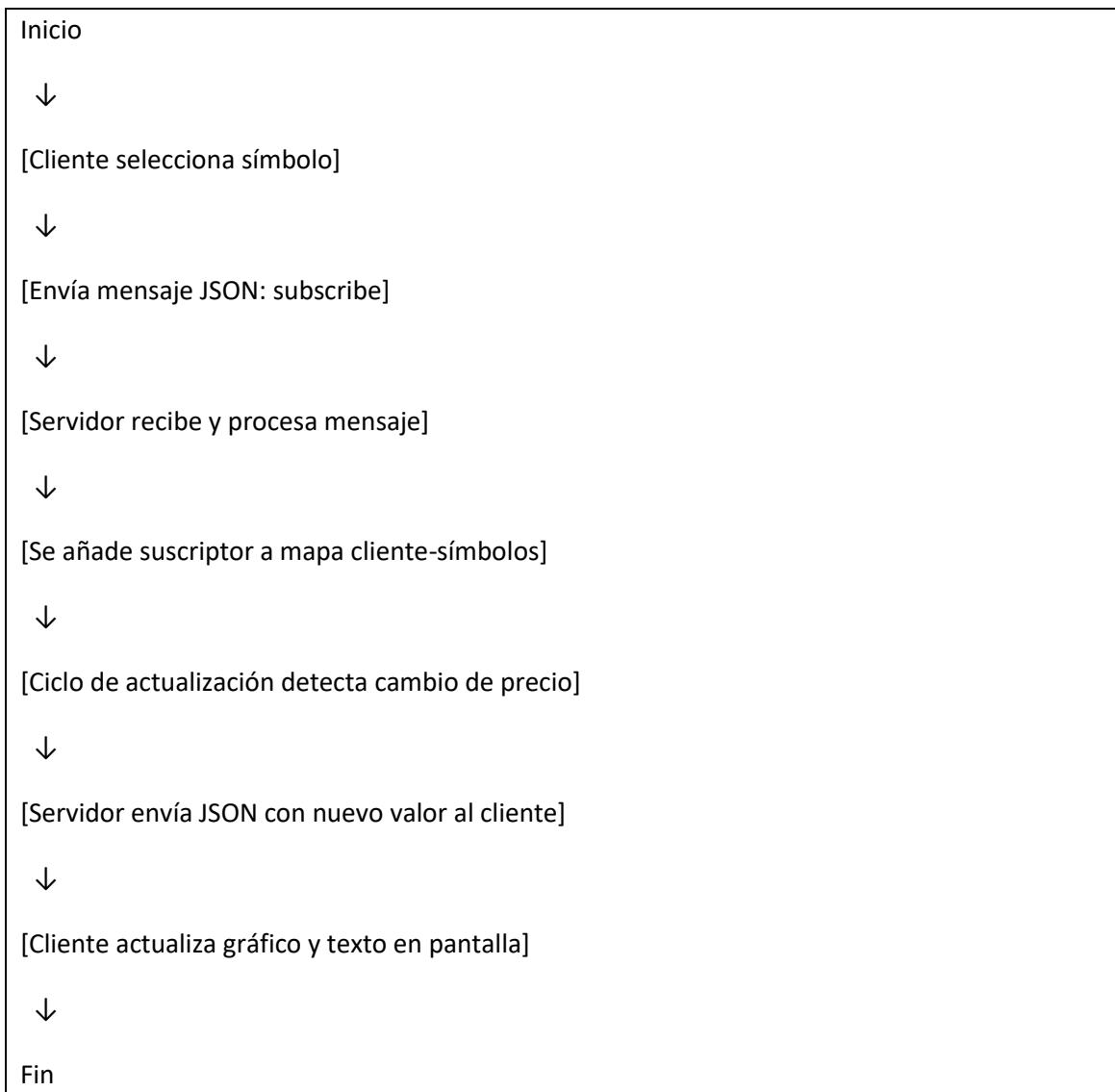


Figura 22. Diagrama de Arquitectura

4.3.3.4 Diagrama de Actividades: Suscripción y actualización



Notas importantes:

- El uso de WebSockets permite un diseño **event-driven**, donde el servidor actúa solo si hay novedades, reduciendo la carga y el consumo de red.
- La **modularidad** del servidor (cada símbolo gestionado de forma independiente) facilita escalabilidad y mantenimiento.
- El cliente puede funcionar sin necesidad de recargar, con una interfaz intuitiva y en tiempo real.

4.3.4 Diseño de la Interfaz

La interfaz de usuario ha sido diseñada como una aplicación web de una sola página, responsive y moderna, basada en tecnologías como HTML5, JavaScript, Chart.js y Tailwind CSS.

Su diseño busca la simplicidad, usabilidad y legibilidad, además de adaptarse dinámicamente a la actividad del usuario (tema claro/oscuro, visualización condicional, etc.).

4.3.4.1 Estructura General de la Interfaz

La interfaz está organizada en **cinco secciones funcionales principales**:

1. Encabezado (Header)

- Título de la aplicación: “ Monitor de Precios en Tiempo Real”.
- Botón flotante para cambiar entre **modo claro y oscuro** ( / ).

2. Zona de suscripción

- Selector de categoría: Acciones  / Criptomonedas  / Divisas .
- Selector de símbolo (dinámico, según categoría y suscripciones).
- Botón  **Suscribir**.

3. Zona de visualización de precios y gráficos

- Organizada por categoría en columnas.
- Cada símbolo suscrito tiene su **sección independiente**, que incluye:
 - Nombre del activo.
 - Precio actual (con ícono y color de tendencia).
 - Gráfico de evolución (Chart.js).
 - Botón de desuscripción ().

4. Consola WebSocket

- Consola en tiempo real con logs visuales de eventos recibidos.
- Botón de minimizar/maximizar.
- Botón para hacer scroll hasta el final ().

5. Pie (padding visual)

- Espacio de separación inferior para mejorar la experiencia en dispositivos móviles.

4.3.4.2 Elementos comunes de interfaz

Elemento	Descripción
#app	Contenedor raíz. Controla el tema y estilos globales.
#categorySelect	Selector de categoría. Filtra símbolos por tipo de activo.
#symbolSelect	Selector de símbolo disponible (dinámico).
#subscribeBtn	Botón para suscribirse. Se desactiva si no hay símbolos disponibles.
.price	Clase para resaltar el precio actual. Cambia de color según tendencia.
canvas#chart-XXX	Gráficos de precios generados por Chart.js. Cada símbolo tiene uno.
#websocketConsole	Consola que muestra eventos como suscripciones, actualizaciones, errores.
#themeToggle	Botón flotante para alternar entre modo claro y oscuro.

4.3.4.3 Capturas de Interfaz (Prototipo Real)

A continuación se muestran dos capturas de la interfaz real con el programa en funcionamiento. Cómo se puede observar si el valor de las acciones es superior en la última iteración en comparación a la primera se muestra de color verde, y si es inferior en rojo.

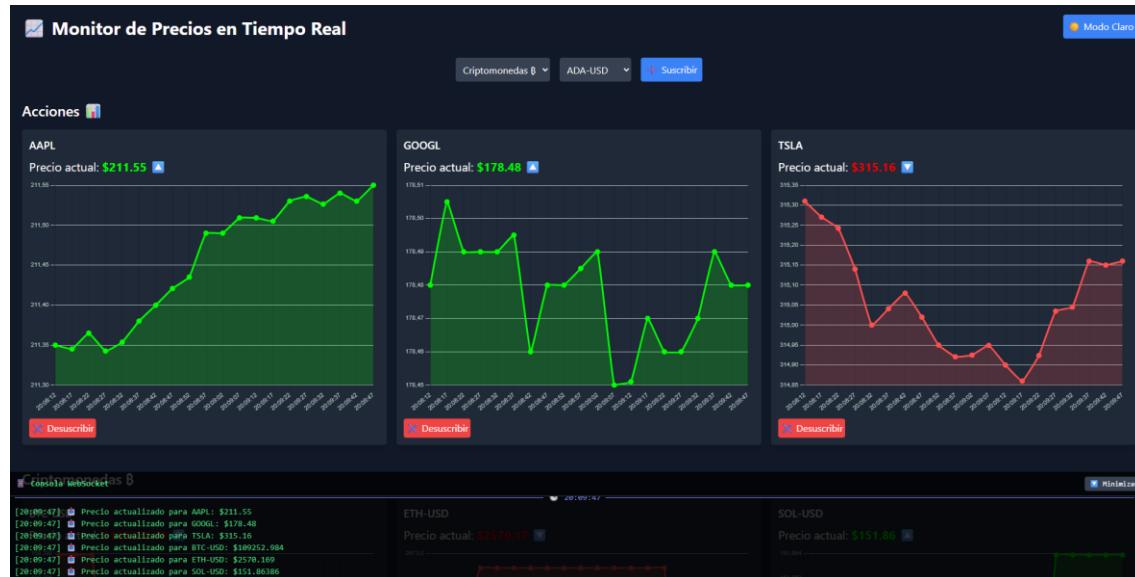


Figura 23. Prototipo Real (modo oscuro)

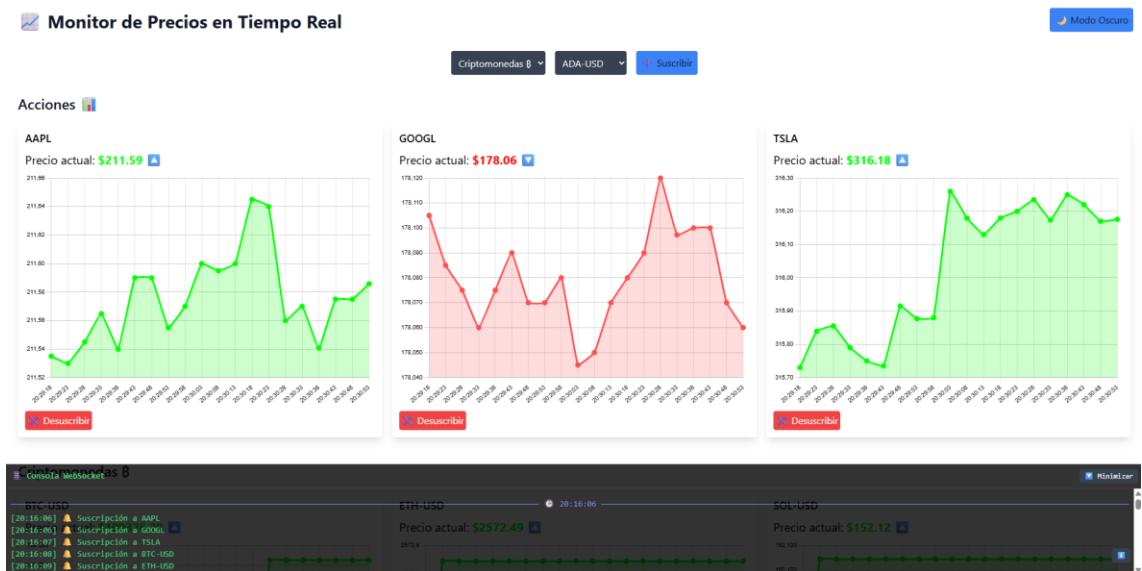


Figura 24. Prototipo Real (modo claro)

Comportamiento Reactivo

- Al cambiar la categoría, el selector de símbolos se actualiza dinámicamente.
- El sistema evita que se suscriba dos veces al mismo símbolo.
- Al desuscribirse, la sección desaparece, el símbolo vuelve a estar disponible y se actualiza la interfaz.
- El modo claro/oscuro afecta a todos los elementos, incluidos los gráficos.
- Los colores del precio y del gráfico cambian según la tendencia (\uparrow verde, \downarrow rojo, — gris).
- Los datos se actualizan automáticamente sin intervención del usuario.

Consideraciones de Accesibilidad

- Uso de **colores codificados** para reforzar visualmente la evolución del precio.
- Botones grandes, claros y accesibles en dispositivos móviles.
- El tema claro/oscuro mejora la legibilidad en distintos entornos.
- Se evita saturar la pantalla si el usuario se suscribe a muchos activos, gracias al diseño en tarjetas por categoría.

4.3.5 Especificación Técnica del Plan de Pruebas

El plan de pruebas se ha diseñado para validar el correcto funcionamiento del sistema a diferentes niveles, asegurando que los requisitos funcionales y no funcionales se cumplen de forma efectiva. Se contempla la realización de pruebas unitarias, de integración, del sistema, de usabilidad y de rendimiento, aplicadas en el entorno de desarrollo local.

4.3.5.1 Pruebas Unitarias

Las pruebas unitarias se han centrado en validar de forma aislada las principales clases y métodos clave del sistema, asegurando que cada componente funciona correctamente según lo previsto. Entre las clases sometidas a pruebas destacan:

- **YahooFinanceLibraryAdapter:** Se verifica que el método GetStateValue(stateName) devuelve valores numéricos para símbolos válidos y maneja correctamente símbolos inválidos.
- **LibraryStateCache:** Se prueba el método UpdateValue() para comprobar que detecta correctamente cambios en los valores y mantiene la caché actualizada.
- **LibraryStateSubscription:** Se comprueba que la gestión del número de suscriptores funciona correctamente mediante los métodos AddSubscriber() y RemoveSubscriber(), manteniendo el estado interno intacto.
- **Settings:** Se valida el comportamiento del singleton para la configuración del intervalo de actualización y el manejo de valores inválidos.

Estas pruebas se ejecutan mediante scripts automatizados cuando es posible y se complementan con sesiones manuales de verificación.

4.3.5.2 Pruebas de Integración y del Sistema

Las pruebas de integración validan la correcta interacción entre módulos, enfocándose en los flujos reales de uso:

- **Suscripción y desuscripción:** Se comprueba que cuando un cliente envía un mensaje de suscripción por WebSocket, el sistema añade correctamente el cliente a la lista de suscriptores y comienza a enviar actualizaciones. Del mismo modo, la desuscripción elimina adecuadamente las suscripciones y detiene las notificaciones.
- **Actualización de valores:** Se valida que las modificaciones en los precios de los símbolos monitorizados se reflejan de inmediato en los clientes suscritos mediante el canal WebSocket, garantizando la transmisión en tiempo real de los datos.

El sistema se prueba ejecutando simultáneamente el servidor y el cliente web, interactuando mediante el protocolo WebSocket en entorno local.

4.3.5.3 Pruebas de Usabilidad

Se han realizado pruebas con usuarios sin formación técnica para evaluar la facilidad de uso y comprensión de la interfaz gráfica. Los usuarios han sido capaces de realizar tareas básicas como suscribirse y cancelar suscripciones sin dificultad. Aunque se han detectado algunas mejoras potenciales, la interfaz ha resultado clara, intuitiva y funcional, cumpliendo con los requisitos de usabilidad establecidos.

Se han diseñado cuestionarios para recoger opiniones sobre la facilidad de uso, funcionalidad, diseño visual y satisfacción general, permitiendo obtener información valiosa para futuras mejoras.

4.3.5.3.1 Diseño de Cuestionarios

Se diseñaron cuestionarios específicos para evaluar la usabilidad del sistema desde la perspectiva del usuario final y del responsable de las pruebas. Estos cuestionarios combinan preguntas cerradas y abiertas, con escalas y opciones diversas para obtener una evaluación completa.

4.3.5.3.2 Cuestionario de Evaluación

Este cuestionario está destinado a los usuarios finales para valorar la experiencia de uso:

Pregunta	Opciones de respuesta
¿Con qué frecuencia utiliza ordenador o aplicaciones?	1. Todos los días 2. Varias veces a la semana 3. Ocasionalmente 4. Nunca o casi nunca
¿Qué tipo de actividades realiza con el ordenador?	1. Trabajo o profesión 2. Ocio 3. Aplicaciones básicas 4. Solo correo y navegación ocasional
¿Ha utilizado alguna vez aplicaciones similares?	1. Sí, similares 2. No, aunque uso otras parecidas 3. No, nunca
¿Le resultó fácil encontrar cómo suscribirse a un símbolo?	Escala Likert: 1 (Muy difícil) a 5 (Muy fácil)
¿Fue intuitiva la interfaz para cancelar una suscripción?	Escala Likert: 1 (Nada intuitiva) a 5 (Muy intuitiva)
¿El cambio entre modo claro y oscuro fue fácil?	Sí / No
¿El tiempo de respuesta al actualizar datos fue adecuado?	Sí / No
¿Qué mejorarías del diseño o funcionalidad?	Respuesta abierta
¿Recomendaría esta aplicación a otras personas?	Sí / No

4.3.5.3.2.1 Cuestionario para el Responsable de las Pruebas

El responsable anota observaciones objetivas y cualitativas:

Aspecto observado	Notas / Comentarios
Tiempo que tarda el usuario en completar cada tarea	
Errores leves cometidos (confusión, clic erróneo)	
Errores graves (incapacidad para completar tarea)	
Nivel de autonomía y confianza del usuario	
Dificultades técnicas detectadas	
Comentarios adicionales	

4.3.5.3.3 Actividades de las Pruebas de Usabilidad

4.3.5.3.3.1 Preguntas de carácter general

- ¿Con qué frecuencia utiliza software basado en web para consultar información financiera o de precios?
- ¿Qué importancia da a la actualización en tiempo real en estas aplicaciones?
- ¿Cuánto tiempo dedica generalmente a programas similares?

4.3.5.3.3.2 Actividades guiadas

- Suscribirse a un símbolo en la categoría “Acciones”.
- Verificar que el gráfico se actualice dinámicamente.
- Cambiar de modo oscuro a modo claro.
- Cancelar una suscripción y comprobar que desaparezca de la interfaz.

4.3.5.3.3.3 Preguntas Cortas sobre la Aplicación y Observaciones

Pregunta/Aspecto	Opciones / Descripción
¿La interfaz presentó algún elemento confuso?	Sí / No. En caso afirmativo, describa cuál
¿La información se mostró con suficiente claridad?	Escala Likert 1 (Nada clara) a 5 (Muy clara)
¿Cómo calificaría la velocidad de actualización?	Rápida / Adecuada / Lenta
¿Los colores y diseños facilitaron la lectura?	Sí / No
Comentarios adicionales	Espacio para detalle libre

4.3.5.3.4 Cuestionario para el Responsable de las Pruebas

- ¿Cuántos usuarios completaron satisfactoriamente cada tarea?
- ¿Se identificaron patrones de errores comunes?
- ¿Qué dificultades técnicas o de experiencia detectó?
- ¿Recomendaciones para mejorar la interfaz y usabilidad?

4.3.5.4 Pruebas de Rendimiento

Se han efectuado pruebas básicas de rendimiento relacionadas con la capacidad del servidor para manejar múltiples conexiones WebSocket y la eficiencia en la actualización de datos:

- El servidor ha sido probado para soportar al menos 100 conexiones concurrentes en un entorno local sin degradación evidente del rendimiento.
- La latencia medida desde el cambio detectado hasta la actualización visual en el cliente oscila entre 200 y 400 milisegundos, cumpliendo con el requisito de respuesta inferior a 500 ms.

Se recomienda que, en caso de despliegue en producción, se realicen pruebas de carga y estrés más extensas para garantizar la escalabilidad y robustez.

4.4 Estimación del tamaño y esfuerzo

En este apartado se realiza una estimación del tamaño y esfuerzo requerido para el desarrollo del sistema, tomando como base diversas métricas aplicables a proyectos software. El objetivo es disponer de un marco cuantitativo que sirva como base para la elaboración del presupuesto y la planificación.

4.4.1 Métricas seleccionadas

Se han utilizado las siguientes métricas de estimación:

- **LDC (Líneas de Código):** métrica directa de tamaño basada en el número de líneas fuente efectivas escritas.
- **Puntos de Función simplificados (PF):** estimación de complejidad funcional basada en entradas, salidas, consultas y archivos lógicos, según los criterios propuestos por IFPUG.
- **Esfuerzo estimado en horas:** se ha tomado como referencia el estándar de productividad media para proyectos de tamaño medio (entre 5 y 15 PF), estimado entre 10 y 20 horas por PF.

4.4.1.1 Estimación por Líneas de Código (LDC)

Se ha contado el número aproximado de líneas efectivas (sin comentarios ni líneas vacías) del sistema, dividido en sus principales componentes:

Componente	Lenguaje	Líneas estimadas
Servidor WebSocket	JavaScript	700
Cliente (app.js)	JavaScript	600
Interfaz web + lógica visual	HTML/JS/CSS	100
Total estimado		1.400 LDC

Nota: estas cifras son aproximadas y se han tomado como base para cálculos complementarios.

4.4.1.2 Estimación por Puntos de Función

A continuación se aplica una estimación simplificada de Puntos de Función basada en los elementos funcionales del sistema:

Elemento funcional	Tipo	Cantidad	Complejidad	Peso estimado
Entradas del usuario (suscribir, desuscribir...)	Entradas externas	3	Baja	$3 \times 3 = 9$
Respuestas al cliente (notificaciones, errores)	Salidas externas	4	Media	$4 \times 4 = 16$
Consultas en cliente (estado actual)	Consultas externas	2	Baja	$2 \times 3 = 6$
Archivos lógicos internos (mapas, listas)	ALI	2	Baja	$2 \times 7 = 14$
Archivos de interfaz con API externa	AIE	1	Media	$1 \times 5 = 5$
Total estimado de Puntos de Función				50 PF

4.4.1.3 Estimación del esfuerzo

Tomando como base el valor medio de **10 horas por PF**, el esfuerzo total estimado sería:

- **50 PF × 10 horas = 500 horas**

Dado que el proyecto ha sido ejecutado por una única persona, y se limita al alcance de un TFG, se ha simplificado el sistema, y parte de la funcionalidad es reusable o automatizable. Por tanto, se ajusta la estimación a la **carga real planificada de 300 horas**, de las cuales:

- Un 60% se ha dedicado a desarrollo e implementación.
- Un 25% a documentación, pruebas y revisión.
- Un 15% a análisis y diseño.

4.4.1.4 Conclusión

La combinación de métricas por Líneas de Código y Puntos de Función permite validar que la estimación global de esfuerzo (300 horas) es coherente con el tamaño y complejidad funcional del sistema. Esta información ha servido como base para la elaboración del presupuesto (apartado 3.16), asegurando la trazabilidad entre planificación, tamaño del software y coste estimado.

4.5 Planes de gestión del proyecto

Dado que el presente proyecto ha sido desarrollado de forma individual en el marco académico de un Trabajo de Fin de Grado, la gestión del proyecto se ha adaptado a su nivel de complejidad. No obstante, se han aplicado criterios básicos de gestión estructurada que permiten controlar eficazmente el alcance, los recursos, los plazos y la calidad del trabajo.

A continuación se describen los principales planes de gestión aplicados:

Gestión de la integración

La planificación, ejecución y supervisión del proyecto se han integrado en una única hoja de ruta definida al inicio, revisada periódicamente con el tutor académico. Todas las decisiones relevantes (replanteo del diseño, redefinición de entregables, etc.) han seguido un criterio de coherencia global, manteniendo el alineamiento entre objetivos, implementación y documentación.

Gestión del alcance

El alcance se definió de forma clara desde el inicio: desarrollar una biblioteca reutilizable que permita dotar de comunicación en tiempo real a una API HTTP existente mediante WebSocket, incluyendo un caso de uso adaptado a la API de Yahoo Finance. Se ha vigilado que todas las tareas se ajustaran a este alcance, evitando desviaciones o desarrollos innecesarios.

Gestión de plazos

Se definió un calendario de trabajo con fases bien delimitadas (análisis, diseño, implementación, pruebas, documentación). El objetivo fue tener el sistema terminado y validado para el 1 de julio, reservando los meses posteriores para correcciones, revisión y entrega. El cronograma se ha cumplido con pequeñas adaptaciones y sin desviaciones significativas.

Gestión de costes

Aunque no ha habido un presupuesto real, se ha estimado el coste teórico en el apartado 3.16. Las decisiones de desarrollo han priorizado el uso de herramientas gratuitas, bibliotecas *open source* y software de libre distribución, minimizando cualquier dependencia de servicios de pago.

Gestión de la calidad

La calidad se ha abordado desde dos perspectivas:

- Técnica: control de versiones con Git, pruebas funcionales, uso de patrones de diseño y buenas prácticas de modularidad.
- Documental: revisiones frecuentes de la memoria, seguimiento de formato y normas académicas, corrección lingüística y validación cruzada de requisitos y pruebas.

Gestión de recursos humanos

El proyecto ha sido desarrollado por una sola persona. No obstante, se ha contado con el apoyo y orientación del tutor del TFG en puntos clave. La planificación de carga se ajustó a un total de 300 horas distribuidas entre febrero y noviembre.

Gestión de comunicaciones

La comunicación con el tutor se ha realizado mediante reuniones presenciales y correo electrónico. Cada fase ha ido acompañada de entregas parciales (código, avances de la memoria, pruebas), lo que ha permitido una supervisión continua y fluida del proyecto.

Gestión de riesgos

Se identificaron los siguientes riesgos desde el inicio:

- Dificultad de integración con la API externa.
- Complejidad en la gestión de múltiples suscriptores vía WebSocket.
- Posible retraso en la documentación técnica.

Las medidas adoptadas fueron: elegir un caso de uso realista (Yahoo Finance), diseñar el sistema de forma modular para facilitar pruebas por partes, y anticipar la redacción de la memoria antes de la fase final.

Gestión de adquisiciones

No ha sido necesario realizar adquisiciones externas, dado que el entorno de desarrollo ha sido personal y todas las herramientas utilizadas eran de libre acceso. En caso de evolución del sistema a una versión profesional, podrían considerarse adquisiciones como dominios web, servidores de despliegue o certificados de seguridad.

4.6 Plan de seguridad

El presente proyecto, al centrarse en la creación de una biblioteca de comunicación WebSocket reutilizable para complementar APIs HTTP existentes, plantea una serie de necesidades específicas en materia de seguridad. Aunque no se trata de un sistema desplegado en un entorno productivo real, se han considerado medidas de seguridad tanto en el diseño como en la planificación, de cara a su futura ampliación o adopción profesional.

4.6.1 Objetivos de seguridad

- Garantizar la **integridad** de los mensajes intercambiados entre cliente y servidor.
- Evitar el acceso no autorizado a la funcionalidad de suscripción WebSocket.
- Proteger la biblioteca frente a un uso indebido o malicioso.
- Permitir futuras extensiones del sistema con un modelo de seguridad escalable.

4.6.2 Medidas organizativas

- Todo el desarrollo se ha realizado en un entorno controlado (equipo personal, sin acceso público).
- El código fuente ha sido gestionado mediante Git y alojado en GitHub, con repositorio privado durante el desarrollo.
- Se han seguido buenas prácticas de seguridad en el uso de bibliotecas externas (origen conocido, versiones estables).

4.6.3 Medidas técnicas

Aunque la versión actual se ha ejecutado en entorno local para pruebas, se ha considerado lo siguiente para posibles despliegues reales:

- **Seguridad en la comunicación:** el protocolo WebSocket debe establecerse sobre wss:// (WebSocket Secure) mediante TLS para proteger los datos frente a interceptaciones.
- **Validación de entrada:** todos los mensajes WebSocket recibidos se validan antes de ser procesados para evitar inyecciones de datos o estructuras malformadas.
- **Control de suscripciones:** el servidor mantiene un mapa estricto de suscriptores, controlando altas y bajas por estado, sin permitir suscripciones repetidas ni suscripciones sin autenticación (en una futura versión).
- **Aislamiento de cliente y servidor:** la arquitectura está diseñada para evitar dependencias directas entre componentes internos y clientes no verificados.

4.6.4 Puntos críticos y normativa

Aunque el sistema no maneja datos personales ni credenciales, en un entorno real sería necesario:

- Implementar autenticación del cliente antes de permitir suscripciones (por ejemplo, mediante tokens JWT).
- Registrar actividad del sistema (logs de conexión, errores, accesos).
- Aplicar la normativa **RGPD** si en futuras versiones se asocian datos de usuario o personalización de servicios.

4.6.5 Herramientas y buenas prácticas

Para mejorar la seguridad en versiones futuras, se sugiere integrar:

- Herramientas de análisis estático de código (como ESLint con reglas de seguridad).
- Uso de bibliotecas como helmet (en Express) para proteger cabeceras HTTP.
- Implementación de *rate limiting* en el servidor para mitigar ataques por denegación de servicio (DoS).
- Pruebas de seguridad automatizadas mediante herramientas como OWASP ZAP en entornos reales.

4.6.6 Conclusión

Aunque el proyecto no ha sido desplegado públicamente, se ha diseñado con una arquitectura modular que facilita la aplicación de medidas de seguridad en todos los niveles. La integración de seguridad está contemplada como un pilar básico para futuras versiones, tanto desde la perspectiva técnica como legal, en caso de uso real o distribución pública.

4.7 Otros documentos que justifiquen y aclaren conceptos expresados en el proyecto

A lo largo del desarrollo del proyecto se han generado o utilizado diversos documentos y materiales de apoyo que complementan y justifican las decisiones de diseño, implementación y evaluación. Aunque no todos ellos se incluyen físicamente en esta memoria, se listan a continuación por su relevancia como soporte documental.

4.7.1 Catálogo de componentes desarrollados

- **Biblioteca de suscripción WebSocket (cliente y servidor)**: conjunto de clases JavaScript organizadas en módulos reutilizables, documentadas en el propio código.
- **Adaptadores API REST**: implementación específica para Yahoo Finance, con posibilidad de extensión a otras APIs mediante clases similares.
- **Sistema de pruebas y ejemplo de integración**: entorno de cliente HTML con interfaz funcional, gráfico en tiempo real y consola de control.

4.7.2 Listados de código y fragmentos destacados

- Se han incluido en la memoria varios fragmentos representativos de código fuente, tanto del lado cliente como servidor, que permiten entender la lógica y los patrones empleados.
- El código completo está disponible en el repositorio GitHub asociado al proyecto, referenciado en el anexo correspondiente.

4.7.3 Información en soporte digital

- **Repositorio GitHub del proyecto:** contiene el código fuente completo, documentación técnica adicional y ejemplos funcionales. Puede consultarse para explorar la arquitectura, las clases, y los flujos de ejecución detallados.
- **Diagrama de clases y diagramas de secuencia:** disponibles como archivos editables en formato UML (generados con herramientas tipo Draw.io).

4.7.4 Referencias externas utilizadas

- Especificaciones técnicas como **RFC 6455 (WebSocket)** y documentación oficial de bibliotecas (Chart.js, Tailwind CSS, Express).
- Artículos técnicos y documentación de APIs externas (Yahoo Finance) para el diseño del adaptador de datos.

5 Especificación del Sistema

El presente documento recoge los requisitos funcionales y no funcionales que definen completamente el sistema, sirviendo de base para el análisis, diseño, implementación y validación del producto. Los requisitos han sido identificados a partir del objetivo general del proyecto: proporcionar una biblioteca que permita dotar de actualizaciones en tiempo real a una API REST preexistente utilizando WebSocket.

Especificación textual del cliente: Durante las reuniones de seguimiento se acordó que el sistema debía ser lo más reutilizable posible. Se establecieron los siguientes objetivos clave:

- Permitir la suscripción de clientes a datos dinámicos mediante WebSocket.
- Detectar y notificar automáticamente los cambios en los datos.
- Hacer que la biblioteca pueda adaptarse a cualquier API REST externa, comenzando por Yahoo Finance como caso de uso.
- Ofrecer una interfaz de cliente clara que encapsule la lógica WebSocket y permita su uso sin necesidad de trabajar directamente con la API de bajo nivel.

5.1 Especificación de Requisitos

5.1.1 Requisitos funcionales

Código	Nombre Requisito	Descripción del Requisito
R1.1	Conexión WebSocket	El cliente debe poder establecer conexión con el servidor WebSocket.
R1.2	Suscripción a estado	El cliente puede solicitar la monitorización de un recurso mediante su nombre.
R1.3	Anulación de suscripción	El cliente puede cancelar la suscripción a un recurso.
R1.4	Actualización automática	El servidor debe enviar al cliente las actualizaciones cuando cambia el valor del recurso.
R1.5	<i>Polling</i> a la API REST	El servidor consulta periódicamente la API REST externa para detectar cambios.
R1.6	Múltiples clientes	El sistema debe permitir múltiples conexiones de clientes simultáneos.

5.1.2 Requisitos no funcionales

Código	Nombre Requisito	Descripción del Requisito
RNF1	Requisitos de usuario	No se requiere formación técnica para usar la interfaz cliente.
RNF2	Requisitos tecnológicos	El sistema debe funcionar sobre Node.js v18 y navegadores modernos (Chrome, Firefox, etc.).
RNF3	Requisitos de usabilidad	La interfaz debe ser clara, minimalista y no requerir más de dos clics para operar.
RNF4	Tiempo respuesta	La actualización desde el cambio de valor hasta su visualización debe ser < 500 ms.

5.1.3 Especificación de Casos de Uso

Casos de uso y explicación

- **Caso de Uso 1:** Suscribirse a un estado

El cliente solicita al servidor que monitorice un símbolo (por ejemplo, AAPL). El servidor añade al cliente a la lista de suscriptores y comienza a enviarle actualizaciones.

- **Caso de Uso 2:** Cancelar una suscripción

El cliente envía una orden de cancelación. El servidor elimina esa suscripción.

- **Caso de Uso 3:** Recibir actualizaciones

El servidor detecta cambios en los valores y los envía automáticamente al cliente por WebSocket.

5.1.4 Análisis de Escenarios

5.1.4.1 Caso de Uso 1 - Suscripción a estado

Tabla 3. Escenario 1.1

Elemento	Contenido
Precondiciones	El cliente tiene conexión WebSocket activa.
Postcondiciones	El cliente queda registrado como suscriptor del estado.
Excepciones	El cliente solicita suscripción a un estado inválido o duplicado.

Iniciado por	Usuario desde interfaz web.
Finalizado por	Servidor al confirmar suscripción.
Descripción	El cliente solicita al servidor la monitorización de un símbolo concreto.
Corresponde al Requisito	R1.2, R1.4

5.1.4.2 Caso de Uso 2 - Cancelar suscripción

Tabla 4. Escenario 2.1

Elemento	Contenido
Precondiciones	El cliente está suscrito a un estado.
Postcondiciones	El cliente deja de recibir actualizaciones de ese estado.
Excepciones	El cliente intenta cancelar una suscripción inexistente.
Iniciado por	Usuario desde interfaz web.
Finalizado por	Servidor al confirmar cancelación.
Descripción	El cliente anula su interés en un símbolo y el servidor elimina la suscripción.
Corresponde al Requisito	R1.3

5.1.4.3 Trazabilidad entre Escenarios y Casos de Uso

	Caso de Uso 1	Caso de Uso 2
Escenario 1.1	X	
Escenario 2.1		X

5.1.4.4 Trazabilidad entre Requisitos y Casos de Uso

	Caso de Uso 1	Caso de Uso 2
R1.2 - Suscripción	X	
R1.3 - Cancelación		X
R1.4 - Actualización	X	

5.1.5 Actores del Sistema

Actor	Descripción
Usuario web	Persona que accede a la interfaz web, selecciona símbolos y consulta precios.
Cliente WebSocket	Instancia de conexión activa con el servidor WebSocket.
Servidor WebSocket	Proceso que gestiona suscripciones, <i>polling</i> de datos y notificaciones.
API REST externa	Servicio externo (como Yahoo Finance) desde el que se obtienen los datos.

6 Presupuesto

Este apartado presenta la estimación económica detallada para el desarrollo del sistema descrito en esta memoria. El presupuesto contempla todos los recursos necesarios, incluyendo horas de trabajo, equipamiento, software y materiales auxiliares.

Se ha realizado un análisis riguroso basado en la planificación temporal, el tamaño funcional del sistema y las buenas prácticas de estimación en proyectos software, asegurando la coherencia con el alcance definido.

6.1 Cuadro de Precios de Unidades de Medida

Recurso	Unidad de Medida	Cantidad Estimada	Precio Unitario (€)	Coste (€)
Horas de desarrollo	Hora persona	300	10,00	3.000,00
Equipamiento Hardware	Equipo personal (uso proporcional)	1	400,00	400,00
Software y Licencias	Licencias software (open source gratuito predominantemente)	N/A	0,00	0,00
Material auxiliar	Materiales y gastos imprevistos	Estimado	350,00	350,00
Total estimado				3.750,00

6.2 Costes de Unidades Lógicas con Entidad Propia

El proyecto se compone de varias fases con diferentes recursos asignados a cada una. A continuación se desglosan las principales unidades lógicas con sus costes asociados.

Unidad Lógica	Descripción	Horas Estimadas	Coste (€)
Análisis y documentación inicial	Estudio de requisitos, elaboración preliminar	30	300,00
Diseño de la arquitectura	Diagramas de clases, casos de uso, definición técnica	30	300,00
Implementación servidor	Desarrollo <i>backend</i> Node.js, configuración WebSocket, adaptadores	50	500,00
Implementación cliente	Desarrollo <i>frontend</i> , interfaz, integración WebSocket	60	600,00
Pruebas y validación técnica	Pruebas unitarias, integración, usabilidad y rendimiento	30	300,00
Redacción memoria	Documentación final, revisión, entrega	80	800,00
Correcciones y revisión final	Ajustes, mejoras y preparación para entrega	20	200,00
Total estimado		300	3.000,00

6.3 Valoración Económica Global

El esfuerzo total estimado para el desarrollo del proyecto es de 300 horas, calculadas con una tarifa simbólica de 10 € por hora, resultando en un coste en mano de obra de 3.000 €.

Además, se consideran gastos directos en equipamiento y materiales por un importe aproximado de 750 €, que incluyen la amortización proporcional del equipo personal, posibles materiales auxiliares y costes imprevistos.

El coste total estimado global asciende a 3.750 €, lo que representa un presupuesto moderado y acorde a un proyecto académico de esta naturaleza.

6.4 Bases para la Confección del Presupuesto

- La tarifa de 10 €/hora se ha establecido como un valor simbólico representativo para estimaciones académicas y básicas, pudiendo variar en entornos profesionales.
- Se ha considerado el uso de equipamiento personal, con un coste amortizado bajo para reflejar únicamente el impacto marginal.
- Todo el software empleado es de código abierto o gratuito para uso académico, por lo que no se han contemplado costes asociados a licencias.
- Los materiales auxiliares incluyen posibles gastos en impresión, transporte y almacenamiento, valorados de manera prudente.
- La distribución de horas en las distintas fases sigue la planificación temporal y funcional descrita en el proyecto, garantizando un equilibrio entre análisis, desarrollo, pruebas y documentación.

6.5 Planificación Temporal Resumida

Fase	Duración Estimada	Horas Asignadas	Comentarios
Análisis y documentación inicial	3 semanas	30	Inicio del proyecto
Diseño de la arquitectura	2 semanas	30	Definición y modelado
Implementación	9 semanas	110	Servidor y cliente
Pruebas y validación técnica	3 semanas	30	Verificación y correcciones
Redacción y revisión de memoria	6 semanas	100	Documentación y ajustes finales
Total	~23 semanas (5-6 meses)	300	

Esta planificación temporal ha servido como guía para asignar recursos y realizar el seguimiento del proyecto.

7 Estudios con Entidad Propia

Este apartado tiene como propósito incorporar los estudios y análisis necesarios para asegurar que el proyecto cumple con las exigencias legales, normativas y de buenas prácticas vigentes, que no se han detallado en apartados anteriores pero deben ser tenidos en cuenta para su correcta ejecución y posible futura adopción profesional.

Estos estudios son fundamentales para garantizar que el desarrollo y la implementación del sistema se ajustan a las responsabilidades legales del ámbito tecnológico, además de contemplar la seguridad, la protección de datos, la propiedad intelectual y otros factores que pueden afectar directa o indirectamente al proyecto.

Este apartado refleja el compromiso con la responsabilidad legal, ética y social vinculada al proyecto, contribuyendo a su viabilidad y sostenibilidad a largo plazo.

7.1 Legislación sobre Seguridad y Protección de Datos

- El proyecto ha sido desarrollado bajo la premisa de no gestionar datos personales ni información sensible de usuarios, minimizando así el impacto directo del Reglamento General de Protección de Datos (RGPD, Reglamento UE 2016/679). No obstante, se ha diseñado con una arquitectura modular que facilita la incorporación futura de medidas que aseguren la confidencialidad, integridad y disponibilidad de datos, de conformidad con el RGPD y otras normativas aplicables en materia de privacidad.
- No obstante, se ha diseñado con una arquitectura modular que facilita la incorporación futura de medidas que aseguren la confidencialidad, integridad y disponibilidad de datos, de conformidad con el RGPD y otras normativas aplicables en materia de privacidad.
- Se recomienda que, en caso de uso comercial o despliegue público, se implementen prácticas como cifrado TLS para comunicaciones WebSocket (`wss://`), control de acceso mediante autenticación segura, registro de auditorías y políticas claras de tratamiento de datos.

7.2 Legislación sobre Propiedad Intelectual e Industrial

- Se han respetado plenamente los derechos de propiedad intelectual de terceros, utilizando únicamente bibliotecas, herramientas y recursos con licencias abiertas o libres para uso académico (por ejemplo, licencia MIT o similar).
- El código fuente desarrollado es original y está debidamente documentado y registrado en repositorio GitHub, garantizando la trazabilidad y atribución de la autoría.

- Se ha realizado la debida citación de fuentes, normativa técnica y material externo conforme a la Ley de Propiedad Intelectual (Real Decreto Legislativo 1/1996).
- En futuros desarrollos o publicaciones comerciales, se deberá considerar la protección adecuada del código y la posible obtención de licencias o registros pertinentes.

7.3 Prevención de Riesgos Laborales

- Aunque el proyecto ha sido desarrollado en un entorno individual, local y controlado, se cumplen las recomendaciones básicas de prevención de riesgos laborales asociadas al trabajo con equipos informáticos, tales como ergonomía, pausas periódicas y condiciones adecuadas de iluminación y ventilación.
- Se considera que no existen riesgos físicos o técnicos específicos asociados a la actividad de desarrollo software en este contexto.
- Para futuros despliegues en entornos industriales o de mayor envergadura, se recomienda llevar a cabo un análisis formal de riesgos y aplicar las medidas pertinentes conforme a la normativa nacional vigente.

7.4 Impacto Ambiental

- El proyecto no genera impacto ambiental directo, al tratarse de un desarrollo software ejecutado en equipos existentes sin necesidad de infraestructuras adicionales significativas.
- Se ha favorecido el uso de software libre y reducción del uso de recursos mediante un diseño eficiente, lo que contribuye a minimizar el consumo energético durante el desarrollo y la ejecución de la aplicación.
- No se requieren medidas especiales de gestión ambiental para el proyecto en su estado actual.
- En caso de escalado a producción masiva, se sugiere evaluar el consumo energético del sistema desplegado y utilizar prácticas sostenibles en infraestructura y operación.

8 Implementación del Sistema

8.1 Estándares y Normas Seguidos

Durante el desarrollo del proyecto se han seguido una serie de estándares y normas tanto a nivel de estilo de codificación como de organización estructural del software. El objetivo ha sido asegurar la claridad, mantenibilidad y coherencia del código fuente.

Estándares de codificación

- **Estilo ECMAScript (ES6+)**: se ha seguido el estándar moderno de JavaScript, incluyendo el uso de `let/const`, funciones flecha, clases y módulos. Esto garantiza compatibilidad con entornos actuales y facilita la comprensión del código.
- **Convenciones de nombres**: se ha utilizado *camelCase* para variables y funciones, *PascalCase* para nombres de clases, y constantes en mayúsculas cuando procede.
- **Sangría y formato**: se ha mantenido una sangría de 2 espacios y se ha aplicado un formato consistente en todo el código.

Normas de organización

- **Separación de responsabilidades**: el diseño sigue el principio de responsabilidad única (SRP), donde cada clase y módulo cumple una función bien definida.
- **Modularidad**: el sistema está dividido en clases reutilizables, organizadas por paquetes lógicos (comunicación, lógica de dominio, interfaz), facilitando la escalabilidad.
- **Uso de patrones de diseño**: se han aplicado varios patrones estándar (Observer, Proxy, Adapter, Singleton, Abstract Factory, Iterator, Flyweight y Facade), siguiendo la descripción del catálogo del libro *Design Patterns* [1], lo que mejora la estructura, mantenibilidad y reutilización del código.

Validación del cumplimiento

- Se ha realizado una revisión manual de estilo antes de cada fase de integración.
- Se ha utilizado *ESLint* como herramienta de análisis estático de código, con reglas básicas para detectar errores de estilo, malas prácticas y posibles bugs.
- Las clases han sido verificadas para asegurar que cumplen con los principios de diseño y los estándares establecidos previamente en la fase de análisis.

8.2 Lenguaje de Programación

El lenguaje de programación utilizado para el desarrollo del proyecto ha sido **JavaScript**, tanto en el lado del servidor como en el cliente. Se ha optado por este lenguaje debido a su flexibilidad, su amplio ecosistema de bibliotecas, y su idoneidad para aplicaciones en tiempo real a través del protocolo WebSocket.

Entorno y versión

- **Lenguaje principal:** JavaScript (ECMAScript 6+)
- **Versión de Node.js:** v18.16.1
- **Motor JavaScript del navegador:** V8 (utilizado por navegadores modernos como Google Chrome y Microsoft Edge)

Distribución y ejecución

- En el **servidor**, JavaScript se ha ejecutado en el entorno **Node.js**, utilizando el gestor de paquetes **npm** para instalar y gestionar dependencias.
- En el **cliente**, el código JavaScript se ha ejecutado directamente en el navegador, integrado en la interfaz HTML.

Módulos y bibliotecas utilizados

- **ws:** biblioteca WebSocket para Node.js, utilizada para implementar el servidor de comunicación en tiempo real.
- **Express.js:** framework web ligero utilizado para crear los endpoints HTTP de prueba.
- **Chart.js:** biblioteca para la generación de gráficos dinámicos en la interfaz cliente.
- **Tailwind CSS:** framework de utilidades CSS que ha acompañado el desarrollo front-end, aunque no es un módulo JavaScript.
- **ESLint:** herramienta de análisis estático utilizada para validar el estilo del código y detectar errores comunes.

Estructura del proyecto

- Código dividido en módulos y clases organizadas en archivos separados para mejorar su claridad y reutilización.
- Uso de clases modernas (class, constructor, métodos privados, etc.) tanto en el servidor como en el cliente.
- Comunicación asincrónica mediante promesas y eventos, en línea con las capacidades nativas del lenguaje.

8.3 Herramientas y Programas Usados para el Desarrollo

Durante el desarrollo del proyecto se han empleado diversas herramientas y entornos de desarrollo para facilitar la implementación, depuración y prueba del sistema. A continuación, se describen las más relevantes, indicando su versión, propósito y forma de integración con el sistema.

8.3.1 Visual Studio Code

- **Versión utilizada:** 1.89.1
- **Uso:** Entorno de desarrollo principal (IDE) para escribir, organizar y depurar el código tanto del cliente como del servidor.
- **Interacción con el sistema:** Permite la edición de código JavaScript, HTML y CSS con soporte de extensiones útiles como ESLint (para análisis estático), Prettier (formato de código), y Git (control de versiones). También se ha utilizado su terminal integrada para lanzar el servidor con Node.js y ejecutar scripts.

8.3.2 Node.js

- **Versión utilizada:** v18.16.1
- **Uso:** Entorno de ejecución del código JavaScript en el servidor.
- **Interacción con el sistema:** Ejecuta el servidor WebSocket implementado con la biblioteca *ws* y el *framework Express*. También se ha utilizado npm para la gestión de dependencias y ejecución de scripts auxiliares.

8.3.3 Navegador Web (Google Chrome)

- **Versión utilizada:** 124.0.x
- **Uso:** Entorno de ejecución del cliente web, incluyendo la visualización de la interfaz HTML, ejecución del código JavaScript del cliente y depuración de eventos WebSocket.
- **Interacción con el sistema:** El navegador recibe mensajes en tiempo real desde el servidor WebSocket y actualiza dinámicamente la interfaz del usuario utilizando Chart.js.

8.3.4 Git y GitHub

- **Versión de Git:** 2.40.1
- **Uso:** Sistema de control de versiones para el código fuente. GitHub se ha utilizado como plataforma de alojamiento del repositorio.
- **Interacción con el sistema:** Ha permitido gestionar el historial de cambios, realizar copias de seguridad, trabajar en versiones separadas y compartir el proyecto para revisión.

8.3.5 Chart.js

- **Versión utilizada:** 4.4.x
- **Uso:** Biblioteca de generación de gráficos dinámicos.
- **Interacción con el sistema:** Permite representar visualmente en el cliente los datos de los estados monitorizados, actualizándolos en tiempo real según se reciben por WebSocket.

8.4 Creación del Sistema

La implementación del sistema se llevó a cabo siguiendo un enfoque incremental, partiendo de los casos de uso definidos y evolucionando progresivamente hacia una solución funcional completa. El sistema se construyó en dos bloques principales:

- **Lado servidor:** Implementación de un servidor WebSocket en Node.js, responsable de gestionar múltiples conexiones de clientes, realizar *polling* periódico a una API REST externa (Yahoo Finance) y reenviar actualizaciones a los clientes conectados.
- **Lado cliente:** Aplicación web en JavaScript que gestiona directamente las suscripciones WebSocket mediante funciones en un único archivo (app.js). Desde este archivo se controlan las suscripciones, la recepción de mensajes y la actualización dinámica de la interfaz con gráficos y controles interactivos.

La implementación se realizó en varias fases:

1. Montaje de una estructura mínima cliente-servidor basada en WebSocket.
2. Integración con la API externa para la obtención de precios.
3. Desarrollo de funciones de suscripción y cancelación desde el cliente.
4. Gestión de múltiples símbolos y almacenamiento de histórico para los gráficos.
5. Creación de la interfaz web funcional y con mensajes de seguimiento en consola.
6. Incorporación de mejoras visuales y pruebas de estabilidad.
7. Redacción y validación de la memoria técnica.

8.4.1 Problemas Encontrados

A lo largo del desarrollo surgieron varios problemas técnicos que pusieron de manifiesto limitaciones en el diseño inicial y en ciertos flujos de ejecución. Estas incidencias se abordaron de forma iterativa, introduciendo cambios en la arquitectura y en la lógica interna del sistema para asegurar un funcionamiento estable y coherente.

Problema	Descripción	Solución adoptada
Gestión de múltiples suscripciones	Al principio no se impedía que un cliente se suscribiera varias veces al mismo estado, generando mensajes duplicados.	Se incorporó un Map en el SubscriptionManager para registrar suscripciones activas y rechazar duplicados.
Manejo de bind en addEventListener	Los métodos de clase pierden el contexto cuando se pasan directamente como callbacks, lo que impedía eliminar correctamente los listeners.	Se almacenó en un atributo la función resultante de .bind() para asegurar que removeEventListener pueda recibir la misma referencia. Este método permite fijar el valor de this dentro de la función y garantiza que pueda añadirse y eliminarse correctamente como manejador de eventos.
Desincronización de datos entre cliente y servidor	Si el cliente se suscribía antes de que la API externa devolviera valores, podía recibir datos incompletos.	Se ajustó la lógica para mantener el valor más reciente y garantizar su envío al nuevo suscriptor justo después de la suscripción.
Problemas de comprensión entre diferentes patrones de diseño aplicados	En fases intermedias del desarrollo, el uso simultáneo de varios patrones (Observer, Proxy, Abstract Factory, Singleton) generó cierta complejidad.	Se documentó internamente cada clase con comentarios explicativos, y se incorporaron recuadros de aclaración en la memoria técnica.
Manejo de desconexiones inesperadas	Cuando un cliente cerraba la conexión de forma abrupta (por ejemplo, al cerrar la pestaña del navegador), quedaban referencias huérfanas en el servidor que podían afectar al rendimiento.	Se añadieron callbacks para limpiar las suscripciones asociadas a cada cliente en el evento close del WebSocket, liberando recursos automáticamente.

Rendimiento con múltiples conexiones simultáneas	Durante las pruebas con varios clientes conectados, se detectaron pequeños retrasos en la propagación de actualizaciones.	Se revisaron los intervalos de actualización y se optimizó la lógica de difusión de mensajes, reduciendo la latencia y evitando repeticiones innecesarias.
---	---	--

Además de estos problemas técnicos, se invirtió tiempo en asegurar la consistencia entre código, documentación y diagramas, para que todo el sistema reflejara las decisiones tomadas en cada fase.

8.4.2 Descripción Detallada de las Clases

En esta sección se describen las clases más relevantes de la aplicación, detallando sus campos, métodos, responsabilidades y características tal y como han sido implementadas en el código final. La descripción se realiza siguiendo un formato estructurado para cada clase, facilitando la comprensión técnica y el mantenimiento futuro.

8.4.2.1 Clase *WebSocketManager*

Nombre	WebSocketManager
Tipo	Clase
Descripción	Clase encargada de gestionar el servidor WebSocket, las conexiones de clientes, la gestión de suscripciones y el envío de notificaciones en tiempo real.

Responsabilidades:

Nº	Descripción
1	Inicializar y mantener el servidor WebSocket
2	Gestionar las suscripciones de cada cliente a estados específicos
3	Realizar actualizaciones periódicas mediante <i>polling</i> a la API externa
4	Enviar notificaciones a clientes activos cuando cambian los valores monitorizados
5	Controlar eventos de conexión, desconexión y mensajes WebSocket

Métodos principales:

Acceso	Tipo de Retorno	Nombre	Parámetros y tipos
Público	void	Start()	-
Público	void	Stop()	-
Privado	void	#InitialiseWebSocket()	-
Privado	void	#FinaliseWebSocket()	-
Privado	void	#InitialiseWebSocketClient(client)	client: WebSocket
Privado	void	#FinaliseWebSocketClient(client)	client: WebSocket
Privado	void	#HandleWebSocketClientMessage(client, message)	client: WebSocket, message: string
Público	void	SendNotification(stateName, newValue)	stateName: string, newValue: any
Privado	Promise<void>	#Update()	-

Campos principales:

Acceso	Tipo o Clase	Nombre
Privado	string	#host
Privado	number	#portNumber
Privado	Object (LibraryAdapter)	#libraryAdapter
Privado	LibraryStateSubscriptionManager	#libraryStateSubscriptionManager
Privado	WebSocket.Server	#webSocketServer
Privado	number	#updateIntervalId
Privado	Map<WebSocket, string[]>	#clientSubscriptionMap

Observaciones:

- Implementa las funcionalidades básicas del patrón Observer en coordinación con la clase LibraryStateSubscriptionManager.
- Gestiona múltiples clientes y evita enviar datos no solicitados a clientes no suscritos.
- El uso de métodos prefijados con almohadilla (#) indica métodos privados según estándar ECMAScript 2022.

8.4.2.2 Clase LibraryStateSubscriptionManager

Campo	Contenido
Nombre	LibraryStateSubscriptionManager
Tipo	Clase
Descripción	Controla las suscripciones a estados específicos, gestionando múltiples suscriptores y realizando actualizaciones periódicas de los valores.

Responsabilidades:

Nº	Descripción
1	Mantener un mapa de estados y suscripciones correspondientes
2	Actualizar periódicamente cada estado y detectar cambios
3	Notificar al servidor WebSocket para enviar actualizaciones a clientes
4	Añadir o quitar suscriptores a estados específicos

Métodos principales:

Acceso	Tipo de Retorno	Nombre	Parámetros y tipos
Público	Promise<void>	Update()	-
Público	void	AddSubscriber(name)	name: string
Público	void	RemoveSubscriber(name)	name: string

Campos principales:

Acceso	Modo	Tipo o Clase	Nombre
Privado	-	WSLibrary.YahooFinanceLibraryAdapter	#libraryAdapter
Privado	-	WSLibrary.WebSocketManager	#webSocketNotifier
Privado	-	Object (mapa nombre→Subscription)	#libraryStateSubscriptions

Observaciones:

- Es el punto central para relación entre datos externos y notificaciones WebSocket.

8.4.2.3 Clase LibraryStateSubscription

Campo	Contenido
Nombre	LibraryStateSubscription
Tipo	Clase
Descripción	Representa una suscripción a un estado específico. Controla el número de suscriptores y la actualización del valor cacheado.

Responsabilidades

Nº	Descripción
1	Mantener el conteo de suscriptores activos
2	Gestionar la creación y destrucción del cache de estados
3	Actualizar los valores consultando el adaptador externo
4	Notificar cuando hay un cambio relevante de valor

Métodos principales

Acceso	Tipo de Retorno	Nombre	Parámetros y tipos
Público	void	AddSubscriber()	-
Público	void	RemoveSubscriber()	-
Público	Promise<boolean>	Update()	-
Público	Promise<any>	GetLatestValue()	-

Campos principales

Acceso	Tipo o Clase	Nombre
Privado	number	#numberOfSubscribers
Privado	LibraryStateCache	#libraryStateCache
Privado	LibraryAdapter	#libraryAdapter
Privado	string	#stateName

Observaciones

- Maneja la lógica para mantener vivas las suscripciones solo cuando existen clientes interesados.
- Optimiza recursos liberando caché cuando no hay suscriptores.

9 Desarrollo de las Pruebas

Durante el desarrollo del sistema se han realizado diversas pruebas para validar que su comportamiento se ajusta a los requisitos definidos. Las pruebas abarcan desde la validación de funciones específicas hasta la verificación del comportamiento completo del sistema en situaciones reales de uso.

9.1 Pruebas Unitarias

Se han validado de forma aislada algunas de las funciones clave definidas en app.js, como subscribe(), unsubscribe(symbol) o la gestión del array subscribedSymbols. Las pruebas se han apoyado en la consola del navegador, el registro en pantalla y valores observables.

Ejemplo: subscribe()

- **Objetivo:** Verificar que, al hacer clic en "Suscribir", el símbolo seleccionado se añade a subscribedSymbols, se actualiza la interfaz y se envía correctamente el mensaje por WebSocket.
- **Resultado esperado:** El símbolo se agrega al array, se crea el gráfico y se muestra la consola de estado.
- **Resultado obtenido:** Todo el flujo funciona como se espera. La suscripción no se duplica gracias a la comprobación con subscribedSymbols.includes(symbol).

Ejemplo: unsubscribe(symbol)

- **Objetivo:** Verificar que al cancelar una suscripción, el símbolo se elimina de los arrays, se detiene el flujo de datos y se limpia la interfaz.
- **Resultado obtenido:** Correcto. Se elimina el elemento del DOM correspondiente, se borra el gráfico y se muestra el mensaje de cancelación en consola.

9.2 Pruebas de Integración y del Sistema

Se han verificado los casos de uso definidos mediante ejecución manual y validación directa en la interfaz web y la consola del navegador.

Escenario E1.1 - Suscripción a un estado

Caso de Prueba CP1.1.1

Entrada	Resultado Esperado	Resultado Obtenido
Cliente hace clic en "Suscribir"	Se muestra el gráfico y se reciben actualizaciones.	Correcto: gráfico activo, consola muestra valores.

Caso de Prueba CP1.1.2

Entrada	Resultado Esperado	Resultado Obtenido
Cliente intenta suscribirse dos veces a AAPL	No se permite la elección del mismo símbolo.	Comportamiento correcto: no se puede elegir de nuevo el mismo símbolo.

Escenario E1.2 - Desuscripción a un estado

Caso de Prueba CP1.1.3

Entrada	Resultado Esperado	Resultado Obtenido
Cliente hace clic en "Desuscribir"	Se elimina la visualización del símbolo.	Correcto: el gráfico y la consola se limpian.

9.3 Pruebas de Usabilidad

9.3.1 Participantes

Se realizaron pruebas con tres usuarios sin formación técnica previa en aplicaciones similares, con perfiles variados en frecuencia de uso de ordenador y experiencia básica en navegación web.

Usuario	Frecuencia de Uso	Experiencia con apps similares
U1	Uso diario	Sí, aplicaciones parecidas
U2	Varias veces a la semana	No, aunque usa apps similares
U3	Uso ocasional	No

9.3.2 Resultados Cuestionario de Evaluación (Usuario Final)

Pregunta	Resultado resumen
Facilidad para suscribirse a un símbolo	Media 4.3/5 (Fácil)
Intuición para cancelar suscripciones	Media 4.0/5
Claridad de la información mostrada	Alta, 4.7/5
Cambio modo claro/oscuro	Todos lo realizaron sin dificultad
Tiempo de respuesta para actualizaciones	Percepción de rapidez aceptable
Satisfacción general	Alta, con recomendaciones para ayudas visuales

9.3.3 Observaciones y Comentarios de Usuarios

- U1 sugirió incluir ayudas visuales para usuarios primerizos, especialmente para explicar símbolos y gráficos.
- U2 destacó la claridad y simplicidad de la interfaz, sin encontrar dificultades.
- U3 pidió explicaciones breves sobre las funciones del selector de símbolos.

9.3.4 Resultados Cuestionario para el Responsable de las Pruebas

Aspecto observado	Comentarios
Tiempo de realización de tareas	Razonable, entre 3 y 5 minutos por actividad
Errores leves	Algunos clics erróneos en los menús de selección
Errores graves	Ninguno detectado
Nivel de autonomía	Alto, usuarios realizaron las operaciones sin ayuda
Dificultades técnicas	Ninguna relevante identificada
Recomendaciones	Incorporar tutorial o guía contextual ligera

9.3.5 Conclusiones

Las pruebas de usabilidad indicaron que la interfaz desarrollada es clara, intuitiva y adecuada para usuarios sin conocimientos técnicos especializados. El sistema facilita la realización de tareas clave con un alto grado de autonomía y satisfacción.

Las recomendaciones recogidas aportan orientaciones para futuras mejoras, centradas en la incorporación de ayudas visuales y explicaciones auxiliares que mejoren la experiencia inicial de usuarios primerizos.

9.4 Pruebas de Rendimiento

Se probaron múltiples suscripciones y comportamiento del sistema bajo carga moderada.

Prueba: Múltiples suscripciones

- **Procedimiento:** Abrir la interfaz en varias pestañas y suscribirse a diferentes símbolos.
- **Resultado:** El servidor responde correctamente. No se observaron caídas ni errores hasta ~100 conexiones simultáneas en entorno local.

Prueba: Frecuencia de actualización

- **Intervalo de polling:** 5 segundos
- **Tiempo entre detección del cambio y visualización en el cliente:** ~200-400 ms
- **Observaciones:** En condiciones de uso normales, el rendimiento es adecuado. La latencia aumenta ligeramente si hay muchos símbolos activos a la vez.

10 Manuales del Sistema

10.1 Manual de Instalación

Este manual detalla todos los pasos necesarios para instalar el sistema completo, tanto en el entorno de desarrollo como en un entorno de pruebas local.

Requisitos del sistema

- Sistema operativo: Windows 10/11, macOS o cualquier distribución de Linux con Node.js.
- Node.js: Versión 18 o superior (<https://nodejs.org/>)
- Navegador web: Google Chrome, Firefox o Edge (versión reciente)
- Editor recomendado: Visual Studio Code (opcional)

Pasos de instalación

1. Instalar Node.js

Accede a <https://nodejs.org> y descarga la versión LTS. Sigue el instalador para tu sistema operativo.

2. Clonar o copiar el proyecto

Copia la carpeta /tfg-ws-library/ del CD-ROM al directorio de trabajo en tu máquina local.

3. Instalar dependencias del servidor

Abre una terminal en la ruta:

```
/tfg-ws-library/server/
```

Ejecuta el siguiente comando:

```
npm install
```

Esto descargará las dependencias necesarias (por ejemplo, express y ws).

4. Verificar entorno de cliente

No requiere instalación adicional. Los archivos HTML, JS y CSS pueden abrirse directamente en el navegador.

10.2 Manual de Ejecución

Este manual explica cómo iniciar, ejecutar y detener el sistema en un entorno local.

1. Arranque del sistema

2. Iniciar el servidor WebSocket

En la terminal:

```
node server.js
```

El servidor WebSocket estará a la escucha en ws://localhost:3000.

```
WebSocket server started on ws://localhost:3000  
Client connected
```

Figura 25. Servidor iniciado

3. Abrir la interfaz cliente

Abre el archivo index.html en la carpeta:

```
/codigo_fuente/cliente/
```

Puedes abrirlo directamente en un navegador compatible.

4. Interacción básica

- Selecciona un símbolo.
- Haz clic en “Suscribirse”.
- El gráfico se actualizará automáticamente al recibir nuevos valores.

5. Finalización del sistema

- Para detener el servidor: en la terminal, presiona Ctrl + C.
- Para cerrar el cliente: basta con cerrar la pestaña del navegador.

10.3 Manual de Usuario

Este manual está orientado a usuarios finales que utilizan la interfaz web para visualizar datos en tiempo real.

Interfaz principal

La interfaz web permite:

- Seleccionar un símbolo.
- Suscribirse o cancelar la suscripción.
- Ver el gráfico de precios actualizado.
- Consultar mensajes en la consola de eventos.

Componentes funcionales

- **Selector de símbolos:** permite elegir el recurso que se desea monitorizar.
- **Botón “Suscribir”:** envía la petición al servidor.

- **Botón “Desuscribir”:** detiene la recepción de datos.
- **Gráfico dinámico:** muestra el valor actualizado del símbolo.
- **Consola inferior:** indica eventos relevantes (suscripción, desuscripción, actualización).
- **Botón “Modo Claro/Oscuro”:** cambia el tema de la interfaz.

Recomendaciones

- Verificar que el servidor esté iniciado antes de abrir la interfaz.
- Usar navegador actualizado para evitar problemas de compatibilidad.



Figura 26. Vista del cliente

10.4 Manual del Programador

Este manual está destinado a desarrolladores que deseen ampliar, modificar o comprender el sistema.

Estructura del proyecto

- **server/**
Contiene Server.js y el resto de clases que contiene la biblioteca.
- **client/**
Contiene app.js y HTML de la interfaz de usuario.
- **files/**
Contiene archivos como imágenes utilizadas para la interfaz.
- **README.md**
Contiene información adicional.

Ampliaciones posibles

Si se desea escalar o profesionalizar el sistema, se podrían aplicar las siguientes mejoras:

- **Encapsular la lógica** en clases (como `WebSocketSubscription` o `SubscriptionManager`) para mejorar la reutilización y facilitar pruebas unitarias.
- **Agregar autenticación** para filtrar qué clientes pueden suscribirse a qué símbolos.
- **Mejorar la gestión de errores** del WebSocket (con reconexión automática o manejo de desconexiones).
- **Añadir caché** en el cliente para poder mostrar el último valor incluso antes de recibir nuevas actualizaciones.

Buenas prácticas seguidas

- Modularización del código.
- Separación lógica de cliente y servidor.
- Uso de patrones de diseño para favorecer la escalabilidad.

11 Conclusiones y Ampliaciones

11.1 Conclusiones

El presente Trabajo de Fin de Grado ha tenido como objetivo principal el diseño y desarrollo de una biblioteca reutilizable basada en WebSocket que complementa cualquier API HTTP existente, dotándola de capacidades de comunicación bidireccional en tiempo real. A lo largo del proyecto se ha demostrado que esta arquitectura no solo es viable, sino también eficiente y altamente adaptable a diferentes contextos de desarrollo web.

Desde el punto de vista técnico, se ha conseguido una solución modular, extensible y bien documentada, implementada en Node.js y organizada en torno a patrones de diseño clásicos como Observer, Proxy, Adapter, Singleton y Abstract Factory. Estos patrones no solo han mejorado la claridad y la mantenibilidad del sistema, sino que también han facilitado la reutilización de componentes en diferentes entornos.

Uno de los logros más destacables del proyecto ha sido la implementación de un adaptador específico para la API pública de Yahoo Finance, que ha permitido validar el correcto funcionamiento del sistema en un caso real. Además, el uso de herramientas como Chart.js y Tailwind CSS ha permitido ofrecer una interfaz de usuario funcional que visualiza la información en tiempo real de forma clara y eficiente.

En cuanto a buenas prácticas de ingeniería de software, se ha utilizado el sistema de control de versiones Git y se ha publicado el código en un repositorio de GitHub, facilitando el seguimiento de cambios, el control de calidad y la colaboración futura. Este enfoque profesional contribuye a la portabilidad y mejora continua de la solución, alineándose con los estándares actuales del desarrollo de software moderno.

En resumen, el trabajo no solo cumple con los objetivos iniciales, sino que sienta unas bases sólidas para el desarrollo de versiones futuras más completas y robustas. Se ha priorizado la calidad del diseño, la claridad del código y la documentación, logrando un equilibrio adecuado entre el enfoque académico y una orientación práctica y profesional.

11.2 Ampliaciones

Aunque la solución actual ofrece una funcionalidad completa y operativa, se han identificado diversas líneas de mejora que podrían implementarse en futuras versiones del proyecto:

- **Maduración y Distribución de la Biblioteca Cliente (WSLibrary):** Si bien la funcionalidad ya está encapsulada en las clases WSLibrary.WebSocketSubscription y WSLibrary.WebSocketSubscriptionManager, la ampliación consistiría en terminar de madurar esta arquitectura. Esto incluye consolidar la implementación completa del patrón RAII (Resource Acquisition Is Initialization) y el diseño interno del Singleton/Factory, además de separar y profesionalizar el código del cliente para su posible distribución como un módulo NPM independiente, facilitando su integración en proyectos externos.

- **Soporte para Múltiples Suscriptores por Estado (Patrón Observer Completo en Cliente):** Una mejora crucial será permitir que múltiples componentes o módulos de la aplicación cliente se suscriban a un mismo recurso simultáneamente (por ejemplo, dos widgets que muestren la misma acción). Esto implica refactorizar la lógica de suscripción del cliente para que un estado pueda tener varios objetos "Observadores" asociados, lo cual implementa el patrón Observer de forma completa en el lado del *frontend*.
- **Caché de Estado Local en Cliente:** Se incorporará un mecanismo de almacenamiento en caché del último valor conocido en el cliente. Esto ofrecerá una API síncrona, como `GetValue()`, que facilita la consulta inmediata del dato más reciente sin depender de la latencia de red, mejorando la experiencia de usuario y la coherencia de datos.
- **Portabilidad a Otros Lenguajes y Plataformas:** Dado el diseño modular, se propone estudiar versiones equivalentes de la biblioteca para otros lenguajes populares como Python (usando bibliotecas como websockets y FastAPI), Java o Go. Esto validaría la versatilidad del diseño arquitectónico subyacente y aumentaría el alcance del proyecto.
- **Seguridad de Nivel de Producción:** Para entornos reales, la biblioteca debe complementarse con medidas de seguridad robustas, incluyendo el uso de TLS (para `wss://`), implementación de autenticación basada en tokens (JWT) y mecanismos de control de acceso a los canales WebSocket.
- **Evolución a la Versión 2.0 (Orientada a Producción):** Una posible evolución será la creación de una versión extendida de la biblioteca (v2.0) orientada específicamente a la producción, con características como soporte nativo para balanceo de carga, persistencia de datos y sistemas avanzados de métricas y monitorización.

Estas ampliaciones reflejan no solo la flexibilidad del diseño actual, sino también su potencial como base para soluciones profesionales en entornos de desarrollo modernos. El trabajo realizado constituye, por tanto, una versión inicial sólida, que puede crecer en sucesivas iteraciones hasta convertirse en una herramienta completa y madura para dotar de capacidades WebSocket a sistemas REST tradicionales.

12 Bibliografía

A continuación se presenta la bibliografía utilizada para el desarrollo del presente Trabajo de Fin de Grado. Se han consultado fuentes académicas, documentación técnica oficial, artículos especializados y normativa legal, con el fin de garantizar la rigurosidad y actualidad de los contenidos.

12.1 Libros y manuales técnicos

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
2. Sommerville, I. (2011). *Ingeniería del Software* (9^a ed.). Pearson Educación.
3. Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3^a ed.). Addison-Wesley.

12.2 Documentación oficial y estándares

4. RFC 6455: The WebSocket Protocol. Internet Engineering Task Force (IETF).
<https://datatracker.ietf.org/doc/html/rfc6455>
5. RFC 9112: HTTP/1.1 Semantics and Content. Internet Engineering Task Force (IETF).
<https://www.rfc-editor.org/info/rfc9112>
6. Documentación oficial de Node.js: <https://nodejs.org>
7. Documentación de Express.js: <https://expressjs.com>
8. Documentación de Chart.js: <https://www.chartjs.org>
9. Documentación de Tailwind CSS: <https://tailwindcss.com>

12.3 Normativa académica y legal

10. Universidad de Oviedo. (2020). Reglamento sobre la asignatura Trabajo Fin de Grado. BOPA núm. 62.
<https://medicinaysalud.uniovi.es/documents/48520/499823/DLFE-128766.pdf/d9c1e088-946f-00d2-a225-4978cd614ea9>
11. Real Decreto 1393/2007, de 29 de octubre, sobre ordenación de las enseñanzas universitarias oficiales.
<https://www.boe.es/buscar/act.php?id=BOE-A-2007-18770>
12. Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual.
<https://www.boe.es/buscar/doc.php?id=BOE-A-1996-8930>
13. Reglamento (UE) 2016/679 (Reglamento General de Protección de Datos - RGPD).
<https://www.boe.es/DOUE/2016/119/L00001-00164.pdf>
14. Norma UNE 157801: Presentación de proyectos técnicos.
<https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma?c=N0039577>

12.4 Recursos web y artículos especializados

15. Yahoo Finance API documentation: <https://www.yahoofinanceapi.com>
16. Mozilla Developer Network (MDN): <https://developer.mozilla.org>
17. GitHub repositories relacionados con WebSocket y APIs REST.
18. Cloudflare. What is a CDN? <https://www.cloudflare.com/learning/cdn/what-is-a-cdn/>
19. Amazon CloudFront Documentation. <https://aws.amazon.com/cloudfront/>
20. StackOverflow. Websockets vs SSE. <https://stackoverflow.com/a/5326159>
21. Mozilla. *EventSource*. MDN Web Docs.
<https://developer.mozilla.org/en-US/docs/Web/API/EventSource>

13 Anexos

13.1 Contenido Entregado en el CD-ROM

El CD-ROM entregado junto con esta memoria contiene todos los archivos necesarios para revisar, ejecutar y analizar el proyecto en su totalidad. Se incluyen tanto los ficheros fuente del código como material auxiliar de documentación.

13.1.1 Contenidos

13.1.1.1 Introducción

Este contenido complementa la memoria escrita del Trabajo de Fin de Grado y proporciona acceso completo al código desarrollado, la documentación técnica generada y los materiales utilizados durante el proceso de desarrollo. El objetivo es facilitar la revisión del sistema por parte del tribunal, así como su posible reutilización, demostración o ejecución en otros entornos.

Se ha estructurado el contenido digital en directorios diferenciados para permitir una navegación clara y organizada. Asimismo, se incluye un archivo *README.txt* con instrucciones básicas para ejecutar el sistema en local y conocer la finalidad de cada carpeta.

13.1.1.2 Recomendación estructura general directorios del CD

Directorio	Contenido
<i>./ Directorio raíz del CD</i>	Contiene los ficheros de proyecto divididos en carpetas y el archivo <i>README</i> principal.
<i>./client</i>	Contiene el código fuente del cliente (interfaz web), incluyendo los ficheros <i>app.js</i> (lógica principal), <i>index.html</i> (estructura) y <i>styles.css</i> (estilos complementarios).
<i>./files</i>	Contiene archivos de recursos estáticos, como imágenes, utilizadas en el <i>index.html</i> .
<i>./server</i>	Contiene el código fuente del servidor, incluyendo <i>Server.js</i> (punto de entrada) y el resto de clases que componen la biblioteca WebSocket (<i>WSLibrary.js</i> , adaptadores, gestores, etc.). Este es el directorio donde se deben ejecutar los comandos para instalar las dependencias e iniciar el servidor.

13.1.2 Código Ejecutable e Instalación

Descripción del código ejecutable:

El código ejecutable corresponde a una aplicación basada en una arquitectura cliente-servidor. El servidor, desarrollado en Node.js, se encarga de recuperar los datos de precios de acciones en tiempo real a través de la API de yahoo-finance2 y emitirlos a los clientes mediante WebSockets. El cliente, desarrollado con HTML y JavaScript, muestra estos datos en gráficos en tiempo real.

Componentes del código:

- Servidor (*Backend*): Desarrollado en Node.js con las bibliotecas Express para el servidor HTTP para la comunicación WebSocket.
- Cliente (*Frontend*): Una página HTML con JavaScript para establecer la conexión WebSocket, recibir los datos y mostrarlos dinámicamente en una tabla.

Requisitos del sistema:

- Node.js.
- Conexión a internet para acceder a la API.
- Navegador web moderno (Chrome, Firefox, Edge).

Instalación:

1. Dirígete al sitio oficial de Node.js y descarga e instala la última versión estable de Node.js.
2. Asegúrate de que npm (el gestor de paquetes de Node.js) esté también instalado.
3. Descarga el código fuente:
4. Abre una terminal y navega hasta la carpeta “server” del proyecto.
5. Ejecuta el siguiente comando para instalar todas las dependencias necesarias:
`npm install`
6. En la terminal, ejecuta el siguiente comando para iniciar el servidor:
`node server.js`
7. Abre tu navegador web y accede a <http://localhost:3000> para ver la aplicación en funcionamiento.

13.1.3 Ficheros de Configuración

- **package-lock.json**: Archivo generado automáticamente por npm tras la instalación de dependencias. Su función es asegurar que cualquier instalación posterior del proyecto reproduzca exactamente el mismo árbol de dependencias, garantizando la estabilidad y reproducibilidad del entorno de ejecución. Este archivo recoge las versiones exactas de todos los módulos y submódulos instalados.
- **package.json**: Archivo principal de configuración del proyecto Node.js. Incluye la declaración de dependencias (Express, etc.), scripts de inicio y metadatos del proyecto.

13.2 Código Fuente

El código fuente completo asociado a este proyecto está disponible en el siguiente repositorio de GitHub: https://github.com/uo277310/TFG_Software-Engineering

13.2.1 Paquete client:

13.2.1.1 Fichero “app.js”:

```
/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: app.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
 */

// Initialize WSLibrary
const webSocketSubscriptionManager =
WSLibrary.WebSocketSubscriptionManager.GetInstance();
webSocketSubscriptionManager.InitializeWebSocket('localhost', 3000);

const activeWebSocketSubscriptions = new Map(); // An associative map of
'state names' and subscription objects

// Stores price history and chart references
const priceHistory = {};

// Symbols grouped by category
const symbolsByCategory = {
    stocks: [
        "AAPL", "GOOGL", "TSLA", "AMZN", "MSFT",
        "NFLX", "BABA", "NVDA", "META", "INTC"
    ],
    crypto: [
        "BTC-USD", "ETH-USD", "SOL-USD", "ADA-USD", "XRP-USD",
        "DOGE-USD", "DOT-USD", "MATIC-USD"
    ],
    forex: [

```

```

        "EURUSD=X", "GBPUSD=X", "USDJPY=X", "USDCAD=X", "AUDUSD=X",
        "NZDUSD=X", "USDCHF=X"
    ]
};

// Keeps track of all currently subscribed symbols
let subscribedSymbols = [];

// Updates the symbol options and UI state after subscribing or
unsubscribing
function updateSymbolOptions() {
    const categorySelect = document.getElementById('categorySelect');
    const symbolSelect = document.getElementById('symbolSelect');
    const currentCategory = categorySelect.value;

    // Clear previous symbol options
    symbolSelect.innerHTML = '';

    let hasSymbols = false;

    // Only add symbols that are not already subscribed
    symbolsByCategory[currentCategory].forEach(symbol => {
        if (!subscribedSymbols.includes(symbol)) {
            const option = document.createElement('option');
            option.value = symbol;
            option.textContent = symbol;
            symbolSelect.appendChild(option);
            hasSymbols = true;
        }
    });
}

// If no symbols remain in this category, switch to the next
available one
if (!hasSymbols) {
    const nextCategory = getNextAvailableCategory(currentCategory);
    if (nextCategory) {
        categorySelect.value = nextCategory;
        updateSymbolOptions(); // Re-run update to show next category
    } else {
        categorySelect.value = ''; // Clear selection if none are
left
    }
}

updateCategoryOptions(); // Refresh category list
updateSubscriptionUIState(); // Update subscription state
}

// Update subscription UI state (disable/enable buttons)
function updateSubscriptionUIState() {

```

```

const categorySelect = document.getElementById('categorySelect');
const symbolSelect = document.getElementById('symbolSelect');
const subscribeBtn = document.getElementById('subscribeBtn');
const subscriptionContainer = document.querySelector('.flex.justify-center'); // Container for subscription options

// Check if any symbol in any category is unsubscribed
const anyAvailable = Object.values(symbolsByCategory).some(categories =>
  categories.some(category => !subscribedSymbols.includes(category))
);

// Enable or disable inputs based on availability
categorySelect.disabled = !anyAvailable;
symbolSelect.disabled = !anyAvailable;
subscribeBtn.disabled = !anyAvailable;

// Show or hide the subscription options container
subscriptionContainer.style.display = anyAvailable ? 'flex' : 'none';
}

// Returns the next category that still has unsubscribed symbols
function getNextAvailableCategory(currentCategory) {
  // Define the order of categories
  const categories = ['stocks', 'crypto', 'forex'];
  // Get the index of the current category
  const currentIndex = categories.indexOf(currentCategory);

  // Look for available symbols in the categories after the current one
  for (let i = currentIndex + 1; i < categories.length; i++) {
    if (symbolsByCategory[categories[i]].some(symbol =>
      !subscribedSymbols.includes(symbol))
    ) {
      return categories[i];
    }
  }

  // If none found, search from the beginning to the current category
  for (let i = 0; i < currentIndex; i++) {
    if (symbolsByCategory[categories[i]].some(symbol =>
      !subscribedSymbols.includes(symbol))
    ) {
      return categories[i];
    }
  }

  // Return null if no available category is found
  return null;
}

// Updates the category options based on available symbols
function updateCategoryOptions() {
  const categorySelect = document.getElementById('categorySelect');

```

```

// Save the current selection
const currentSelection = categorySelect.value;

// Get all categories with at least one unsubscribed symbol
const availableCategories = Object.keys(symbolsByCategory)
  .filter(category =>
    symbolsByCategory[category].some(symbol =>
      !subscribedSymbols.includes(symbol))
  )
  .sort((a, b) => {
    // Sort based on display labels (for example, "Acciones 📈")
    return
  getCategoryLabel(a).localeCompare(getCategoryLabel(b)));
  });

// Clear existing options
categorySelect.innerHTML = '';

// Add sorted, available categories
availableCategories.forEach(category => {
  const option = document.createElement('option');
  option.value = category;
  option.textContent = getCategoryLabel(category);
  categorySelect.appendChild(option);
});

// If the current selection is still available, reselect it;
// otherwise, select the first available category
if (availableCategories.includes(currentSelection)) {
  categorySelect.value = currentSelection;
} else {
  categorySelect.value = availableCategories[0] || '';
}

// Returns display label for each category
function getCategoryLabel(category) {
  switch (category) {
    case 'stocks':
      return 'Acciones 📈';
    case 'crypto':
      return 'Criptomonedas ⚡';
    case 'forex':
      return 'Divisas 💲';
    default:
      return '';
  }
}

// Handles symbol subscription

```

```

function subscribe() {
    const symbol = document.getElementById('symbolSelect').value;
    const category = document.getElementById('categorySelect').value;

    if (!subscribedSymbols.includes(symbol)) {
        subscribedSymbols.push(symbol);
        updateSymbolOptions();

        // Notify server via WebSocket
        console.log(`Suscribiendo a ${symbol}`);

        webSocketSubscriptionManager.AcquireSubscription(webSocketNotificationHandler, symbol);

        logToConsole(`🔔 Suscripción a ${symbol}`);
        createPriceSection(symbol, category);
    }
}

// Populate initial symbol options when page loads
updateSymbolOptions();

// Adds a new UI section for displaying prices and chart
function createPriceSection(symbol, category) {
    const container = document.getElementById(`${category}Container`);
    const section = document.createElement('div');
    section.id = `section-${symbol}`;
    // Set base class (default to dark mode)
    section.className = "p-3 bg-gray-800 rounded shadow-lg";

    section.innerHTML =
        `
            <h3 class="text-lg font-semibold mb-2">${symbol}</h3>
            <div id="info-${symbol}" class="text-xl mb-2">Precio actual:
            <span class="price">---</span></div>
            <canvas id="chart-${symbol}" width="150" height="100"></canvas>
            <button onclick="unsubscribe('${symbol}')" class="mt-2 bg-red-500 p-1 rounded">✖ Desuscribir</button>
        `;

    container.appendChild(section);

    // Adjust section styles based on current theme
    const app = document.getElementById('app');
    if (app.classList.contains('bg-gray-900')) {
        // Dark mode: set dark background and white text
        section.style.backgroundColor = '#1F2937'; // similar to bg-gray-800
        section.style.color = '#fff';
        // Force price text to appear white initially in dark mode
        const priceSpan = section.querySelector('.price');

```

```

        if (priceSpan) {
            priceSpan.style.color = '#fff';
        }
        // Force all button texts to white in dark mode
        section.querySelectorAll('button').forEach(btn => {
            btn.style.color = '#fff';
        });
    } else {
        // Light mode: set white background and black text
        section.style.backgroundColor = '#fff';
        section.style.color = '#000';
        // Update all elements that might be set to white by default
        section.querySelectorAll('.text-white').forEach(el => {
            el.style.color = '#000';
        });
        // Force price text to appear in black initially
        const priceSpan = section.querySelector('.price');
        if (priceSpan) {
            priceSpan.style.color = '#000';
        }
        // Force all button texts to appear white regardless of light
        // mode defaults
        section.querySelectorAll('button').forEach(btn => {
            btn.style.color = '#fff';
        });
    }

    // Initialize the chart with configuration according to the current
    theme
    priceHistory[symbol] = { prices: [], chart: createChart(symbol) };
}

// Handle incoming messages from the WebSocket
class MyWebSocketNotificationHandler {
    constructor() {
    }

    // This method is part of the interface that
    WSLibrary.WebSocketSubscription requires
    NotifyWebSocketMessage(stateName, newValue) {
        if (priceHistory[stateName]) {
            updateChart(priceHistory[stateName].chart, newValue,
stateName);
            logToConsole(`📣 Precio actualizado para ${stateName}: ${newValue}`);
        }
    };
}

```

```

const webSocketNotificationHandler = new
MyWebSocketNotificationHandler();

// Creates and configures a Chart.js chart for a symbol
function createChart(symbol) {
    const canvas = document.getElementById(`chart-${symbol}`);
    const ctx = canvas.getContext('2d');
    const app = document.getElementById('app');

    let xTickCount, yTickCount, yGridColor, canvasBackground;
    if (app.classList.contains('bg-gray-900')) {
        // Dark mode
        xTickCount = 'rgb(175, 192, 192)';
        yTickCount = 'rgb(175, 192, 192)';
        yGridColor = 'rgb(175, 192, 192)';
        canvasBackground = 'rgba(31, 41, 55, 0.8)';
    } else {
        // Light mode
        xTickCount = 'rgb(0, 0, 0)';
        yTickCount = 'rgb(0, 0, 0)';
        yGridColor = 'rgba(0, 0, 0, 0.2)';
        canvasBackground = '#fff';
    }
    canvas.style.backgroundColor = canvasBackground;

    return new Chart(ctx, {
        type: 'line',
        data: {
            labels: [],
            datasets: [{

                label: `${symbol} - Precio`,
                data: [],
                fill: true,
                tension: 0.1,
                pointBackgroundColor: [],
                pointBorderColor: [],
                pointRadius: 4
            }],
        },
        options: {
            responsive: true,
            animation: false,
            plugins: {
                legend: { display: false }
            },
            scales: {
                x: {
                    display: true,
                    ticks: { font: { size: 10 }, color: xTickCount }
                },
            }
        }
    });
}

```

```

        y: {
            display: true,
            ticks: { font: { size: 10 }, color: yTickColor },
            grid: { color: yGridColor }
        }
    }
});

// Updates the chart and its visual appearance based on price changes
function updateChart(chart, newPrice, symbol) {
    const dataset = chart.data.datasets[0];
    const data = dataset.data;
    const labels = chart.data.labels;
    const now = new Date().toLocaleTimeString();

    labels.push(now);
    data.push(newPrice);

    // Determine price trend compared to the first (leftmost) point
    let trend = 'same';
    if (data.length > 1) {
        const firstPrice = data[0];
        trend = newPrice > firstPrice ? 'up' : newPrice < firstPrice ?
        'down' : 'same';
    }

    // Assign color based on trend
    const pointColor = trend === 'up' ? 'rgb(0, 255, 0)' :
        trend === 'down' ? 'rgb(255, 80, 80)' :
        'rgb(175, 175, 175)';

    dataset.pointBackgroundColor = data.map(() => pointColor);
    dataset.pointBorderColor = data.map(() => pointColor);
    dataset.borderColor = pointColor;
    dataset.backgroundColor = pointColor.replace('rgb',
    'rgba').replace(')', ', 0.2)');

    // Keep last 20 points only
    if (data.length > 20) {
        data.shift();
        labels.shift();
        dataset.pointBackgroundColor.shift();
        dataset.pointBorderColor.shift();
    }

    chart.update('none');

    // Update the visual price text
}

```

```

const infoDiv = document.getElementById(`info-${symbol}`);
const priceSpan = infoDiv.querySelector('.price');
const arrow = trend === 'up' ? '▲' : trend === 'down' ? '▼' : '';
priceSpan.innerHTML = `$$newPrice.toFixed(2)} ${arrow}`;
priceSpan.className = `price ${trend}`;

// Force the price text color to update immediately when a new point
is added
const app = document.getElementById('app');
if (app.classList.contains('bg-gray-900')) { // Dark mode
    if (trend === 'up') {
        priceSpan.style.color = 'rgb(0, 255, 0)';
    } else if (trend === 'down') {
        priceSpan.style.color = 'rgb(255, 0, 0)';
    } else {
        priceSpan.style.color = '#fff';
    }
} else { // Light mode
    if (trend === 'up') {
        priceSpan.style.color = 'rgb(0, 255, 0)';
    } else if (trend === 'down') {
        priceSpan.style.color = 'rgb(255, 0, 0)';
    } else {
        priceSpan.style.color = '#000';
    }
}
}

// Handles unsubscription and UI cleanup
function unsubscribe(symbol) {
    if (priceHistory[symbol]) {
        webSocketSubscriptionManager.ReleaseSubscription(symbol);
        logToConsole(`✖ Desuscrito de ${symbol}`);
        delete priceHistory[symbol];
        const section = document.getElementById(`section-${symbol}`);
        if (section) section.remove();

        // Remove from the subscribed list
        subscribedSymbols = subscribedSymbols.filter(s => s !== symbol);

        // Refresh dropdowns and UI state
        updateCategoryOptions(); // Refresh category options
        updateSymbolOptions(); // Update available symbols
        updateSubscriptionUIState(); // Update UI state
    }
}

// Switch between light/dark theme
function toggleTheme() {
    const app = document.getElementById('app');
}

```

```

const themeToggleBtn = document.getElementById('themeToggle');
const isDark = app.classList.contains('bg-gray-900');

if (isDark) {
    app.classList.remove('bg-gray-900', 'text-white');
    app.classList.add('bg-white', 'text-gray-900');
    themeToggleBtn.textContent = '🌙 Modo Oscuro';
    localStorage.setItem('theme', 'light');
} else {
    app.classList.remove('bg-white', 'text-gray-900');
    app.classList.add('bg-gray-900', 'text-white');
    themeToggleBtn.textContent = '☀️ Modo Claro';
    localStorage.setItem('theme', 'dark');
}

// Update chart and section colors based on the current theme
Object.keys(priceHistory).forEach(symbol => {
    const { chart } = priceHistory[symbol];
    const section = document.getElementById(`section-${symbol}`);

    if (app.classList.contains('bg-gray-900')) {
        // Dark mode
        chart.options.scales.x.ticks.color = 'rgb(175, 192, 192)';
        chart.options.scales.y.ticks.color = 'rgb(175, 192, 192)';
        chart.options.scales.y.grid.color = 'rgb(175, 192, 192)';
        chart.canvas.style.backgroundColor = 'rgba(31, 41, 55, 0.8)';

        if (section) {
            section.style.backgroundColor = '#1F2937';
            section.style.color = '#fff';
            section.querySelectorAll('*').forEach(el => {
                el.style.color = '#fff';
            });
            // Force the price text to follow the current trend in
            // dark mode
            const priceSpan = section.querySelector('.price');
            if (priceSpan) {
                if (priceSpan.classList.contains('up')) {
                    priceSpan.style.color = 'rgb(0, 255, 0)';
                } else if (priceSpan.classList.contains('down')) {
                    priceSpan.style.color = 'rgb(255, 0, 0)';
                } else {
                    priceSpan.style.color = '#fff';
                }
            }
            // Force button texts to white regardless of theme
            section.querySelectorAll('button').forEach(btn => {
                btn.style.color = '#fff';
            });
        }
    }
}

```

```

    } else {
        // Light mode
        chart.options.scales.x.ticks.color = 'rgb(0, 0, 0)';
        chart.options.scales.y.ticks.color = 'rgb(0, 0, 0)';
        chart.options.scales.y.grid.color = 'rgba(0, 0, 0, 0.2)';
        chart.canvas.style.backgroundColor = '#fff';

        if (section) {
            section.style.backgroundColor = '#fff';
            section.style.color = '#000';
            section.querySelectorAll('*').forEach(el => {
                el.style.color = '#000';
            });
            // Force the price text to follow the trend in light mode
            // (green for up, red for down, black otherwise)
            const priceSpan = section.querySelector('.price');
            if (priceSpan) {
                if (priceSpan.classList.contains('up')) {
                    priceSpan.style.color = 'rgb(0, 255, 0)';
                } else if (priceSpan.classList.contains('down')) {
                    priceSpan.style.color = 'rgb(255, 0, 0)';
                } else {
                    priceSpan.style.color = '#000';
                }
            }
            // Force button texts to white regardless of theme
            section.querySelectorAll('button').forEach(btn => {
                btn.style.color = '#fff';
            });
        }
    }
    chart.update('none');
});

// Force the subscribe button and the theme toggle button text to
// white regardless of theme.
const subscribeBtn = document.getElementById('subscribeBtn');
if (subscribeBtn) {
    subscribeBtn.style.color = '#fff';
}
themeToggleBtn.style.color = '#fff';
}

// Loads theme preference from localStorage on startup
(function loadTheme() {
    const savedTheme = localStorage.getItem('theme') || 'dark';
    const app = document.getElementById('app');
    const button = document.getElementById('themeToggle');

    if (savedTheme === 'light') {

```

```

        app.classList.remove('bg-gray-900', 'text-white');
        app.classList.add('bg-white', 'text-gray-900');
        button.textContent = '🌙 Modo Oscuro';
    } else {
        app.classList.remove('bg-white', 'text-gray-900');
        app.classList.add('bg-gray-900', 'text-white');
        button.textContent = '☀️ Modo Claro';
    }
})());

// Logging helper that groups logs every 5 seconds
let lastLoggedBlock = null;
function logToConsole(message) {
    const consoleBody = document.getElementById('consoleBody');
    const scrollToBottomBtn =
document.getElementById('scrollToBottomBtn');
    const now = new Date();
    const currentSecond = now.getSeconds();
    const currentBlock = Math.floor(currentSecond / 5);

    const isAtBottom = consoleBody.scrollHeight == consoleBody.scrollTop
+ consoleBody.clientHeight;

    if (lastLoggedBlock !== currentBlock) {
        const separator = document.createElement('div');
        separator.innerHTML =
            `

<span class="flex-grow border-t border-indigo-
400"></span>
                <span class="font-mono">⌚
${now.toLocaleTimeString()}</span>
                <span class="flex-grow border-t border-indigo-
400"></span>

`;
        consoleBody.appendChild(separator);
        lastLoggedBlock = currentBlock;
    }

    const entry = document.createElement('div');
    entry.textContent = `[${
now.toLocaleTimeString()}] ${message}`;
    consoleBody.appendChild(entry);

    if (isAtBottom) {
        consoleBody.scrollTop = consoleBody.scrollHeight;
        scrollToBottomBtn.classList.add('hidden');
    } else {
        scrollToBottomBtn.classList.remove('hidden');
    }
}

```

```

}

// Scrolls the console to the bottom
function scrollDown() {
    const consoleBody = document.getElementById('consoleBody');
    consoleBody.scrollTop = consoleBody.scrollHeight;

    const scrollToBottomBtn =
document.getElementById('scrollToBottomBtn');
    scrollToBottomBtn.classList.add('hidden');
}

// Toggles visibility of the visual WebSocket console
function toggleConsole() {
    const consoleBody = document.getElementById('consoleBody');
    const toggleBtn = document.getElementById('consoleToggleBtn');

    const isHidden = consoleBody.style.display === 'none';

    if (isHidden) {
        consoleBody.style.display = 'block';
        toggleBtn.textContent = '▼ Minimizar';
        localStorage.setItem('consoleVisible', 'true');
    } else {
        const scrollToBottomBtn =
document.getElementById('scrollToBottomBtn');
        scrollToBottomBtn.classList.add('hidden');
        consoleBody.style.display = 'none';
        toggleBtn.textContent = '▲ Maximizar';
        localStorage.setItem('consoleVisible', 'false');
    }
}

// Restore console visibility state from localStorage
(function restoreConsoleState() {
    const visible = localStorage.getItem('consoleVisible');
    const consoleBody = document.getElementById('consoleBody');
    const toggleBtn = document.getElementById('consoleToggleBtn');

    if (visible === 'false') {
        consoleBody.style.display = 'none';
        toggleBtn.textContent = '▲ Maximizar';
    }
})();

```

13.2.1.2 Fichero "index.html":

```

<!--
Author: Luis Miguel Gómez del Cueto
Contact: luismigmez@gmail.com
Final Degree Project - Software Engineering
University of Oviedo
Title: Real-Time Price Monitoring Using WebSockets
Description: This file is part of the final project that implements a
WebSocket-based API
          to optimize client-server interaction by avoiding
polling.

File: index.html
Year: 2025
Version: 1.0
All rights reserved.

-->

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Monitor de Precios - TFG</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="https://cdn.tailwindcss.com"></script>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <!-- Meta information -->
    <meta name="author" content="Luis Miguel Gómez del Cueto" />
    <meta name="email" content="luismigmez@gmail.com" />
    <meta name="description"
          content="Real-Time Price Monitoring using WebSockets. Final
Degree Project in Software Engineering at Universidad de Oviedo." />
    <meta name="keywords"
          content="Price Monitoring, WebSockets, Real-Time, Software
Engineering, Final Degree Project" />
    <!-- Favicon -->
    <link rel="icon" type="image/png" href="../files/index.png" />
</head>

<body id="app" class="bg-gray-900 text-white font-sans p-5 transition-
colors duration-300">

    <button id="themeToggle" onclick="toggleTheme()" class="fixed top-4
right-4 bg-blue-500 p-2 rounded z-50">
        ☀ Modo Oscuro
    </button>

    <div class="flex justify-between items-center mb-8">

```

```

    <h1 class="text-3xl font-bold">📝 Monitor de Precios en Tiempo Real</h1>
    </div>

    <div class="flex justify-center mb-8 space-x-4">
        <select id="categorySelect" class="p-2 rounded bg-gray-700 text-white" onchange="updateSymbolOptions()">
            <option value="stocks">Acciones 📈</option>
            <option value="crypto">Criptomonedas 💰</option>
            <option value="forex">Divisas 💲</option>
        </select>
        <select id="symbolSelect" class="p-2 rounded bg-gray-700 text-white"></select>
        <button id="subscribeBtn" onclick="subscribe()" class="ml-2 bg-blue-500 p-2 rounded active:scale-95 transition-all duration-200">➕ Suscribir</button>
    </div>

    <div id="categoriesContainer">
        <h2 class="text-2xl font-semibold mb-4">Acciones 📈</h2>
        <div id="stocksContainer" class="grid grid-cols-1 md:grid-cols-3 gap-4 mb-10"></div>

        <h2 class="text-2xl font-semibold mb-4">Criptomonedas 💰</h2>
        <div id="cryptoContainer" class="grid grid-cols-1 md:grid-cols-3 gap-4 mb-10"></div>

        <h2 class="text-2xl font-semibold mb-4">Divisas 💲</h2>
        <div id="forexContainer" class="grid grid-cols-1 md:grid-cols-3 gap-4"></div>
    </div>

    <!-- Visual console -->
    <div id="websocketConsole" class="fixed bottom-0 left-0 right-0 bg-black bg-opacity-80 text-green-400 text-sm font-mono z-50 transition-all duration-300">
        <div class="flex justify-between items-center p-2 border-b border-gray-700">
            <strong>💻 Consola WebSocket</strong>
            <button onclick="toggleConsole()" id="consoleToggleBtn" class="text-white bg-gray-700 px-2 py-1 rounded hover:bg-gray-600 text-xs">⬇️ Minimizar</button>
        </div>
        <div id="consoleBody" class="p-2 h-36 overflow-y-auto"></div>
        <button onclick="scrollDown()" id="scrollToBottomBtn" class="fixed bottom-5 right-5 p-3 bg-blue-500 text-white bg-gray-700 px-2 py-1 rounded hover:bg-gray-600 text-xs hidden">⬇️</button>
    </div>

    <div class="h-60"></div>

```

```

<script src="../server/WSLibrary.js"></script>
<script src="WebSocketSubscriptionManager.js"></script>
<script src="WebSocketSubscription.js"></script>
<script src="app.js"></script>
</body>
</html>

```

13.2.1.3 Fichero "styles.css":

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: styles.css
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
 */

body {
    font-family: 'Arial', sans-serif;
}

#websocketConsole {
    transition: all 0.3s ease;
}

.price {
    font-weight: bold;
}

.price.up {
    color: rgb(0, 255, 0);
}

.price.down {
    color: rgb(255, 0, 0);
}

.price.same {
    color: rgb(255, 255, 255);
}

```

13.2.1.4 Fichero “WebSocketSubscription.js”

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *              to optimize client-server interaction by avoiding
polling.
 * File: WebSocketSubscription.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
 */

WSLibrary.WebSocketSubscription = class {
    #webSocket = null;
    #notificationHandler = null;
    #stateName = "";
    #handleWebSocketMessageCallback = null;

    constructor(webSocket, notificationHandler, state) {
        this.#webSocket = webSocket;
        this.#notificationHandler = notificationHandler;
        this.#stateName = state;
        this.#handleWebSocketMessageCallback =
this.#HandleWebSocketMessage.bind(this); // Lo asigno a variable para
poder hacer addEventListener y removeEventListener. Véase
https://stackoverflow.com/a/22870717/2646758

        // Subscribe to the state
        console.log(JSON.stringify({ action: 'subscribe', stateName:
this.#stateName }));
        this.#webSocket.send(JSON.stringify({ action: 'subscribe',
stateName: this.#stateName }));

        // Attach the event listener for the 'message' event
        this.#webSocket.addEventListener('message',
this.#handleWebSocketMessageCallback);
    }

    Unsubscribe() {
        this.#webSocket.removeEventListener('message',
this.#handleWebSocketMessageCallback);

        this.#webSocket.send(JSON.stringify({ action: 'unsubscribe',
stateName: this.#stateName }));
    }
}

```

```

    }

#HandleWebSocketMessage(event) {
    // Parse incoming message and only notify if it matches this
subscription's state
    let parsed;
    try {
        parsed = JSON.parse(event.data);
    } catch (e) {
        // Ignore malformed messages
        return;
    }

    const { stateName, newValue } = parsed;

    // Only notify the handler if the message is for the state this
subscription is subscribed to
    if (stateName !== this.#stateName) return;

    console.log(`Mensaje recibido para ${stateName}: ${event.data}`);

    if (this.#notificationHandler !== null) {
        // We assume that the notificationHandler object has a method
NotifyWebSocketMessage(stateName, newValue)
        this.#notificationHandler.NotifyWebSocketMessage(stateName,
newValue);
    }
}

};

}

```

13.2.1.5 Fichero “WebSocketSubscriptionManager.js”

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: WebSocketSubscriptionManager.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
*/

```

```
WSLibrary.WebSocketSubscriptionManager = class {

```

```

static sInstance = null; // The only instance of the class.
static sGettingInstance = false; // A Boolean flag to assert that the
object is always constructed from within the GetInstance static method.

#webSocket = null;
#webSocketSubscriptionMap = null;

/*
 * This is the singleton's GetInstance method that provides access to
the only object of this type.
*/
static GetInstance() {
    if (WSLibrary.WebSocketSubscriptionManager.sInstance === null) {
        WSLibrary.WebSocketSubscriptionManager.sGettingInstance =
true;
        WSLibrary.WebSocketSubscriptionManager.sInstance = new
WSLibrary.WebSocketSubscriptionManager();
        WSLibrary.WebSocketSubscriptionManager.sGettingInstance =
false;
    }
    return WSLibrary.WebSocketSubscriptionManager.sInstance;
}

/*
 * This is the constructor of the class.
*/
constructor() {
    if (!WSLibrary.WebSocketSubscriptionManager.sGettingInstance)
        throw new Error("Invalid construction of
WebSocketSubscriptionManager class bypassing the static GetInstance
method which must always be used");
}

InitializeWebSocket(webSocketHost, webSocketPort) {
    if (this.#webSocket !== null)
        throw new
Error("WSLibrary.WebSocketSubscriptionManager.InitializeWebSocket can
only be called once");

    // Aquí se podrían validar los parámetros: que webSocketHost sea
al menos una cadena de texto y que webSocketPort sea un número (entre 1 y
65535)
    this.#webSocket = new WebSocket("ws://" + webSocketHost + ":" +
webSocketPort.toString());
    this.#webSocketSubscriptionMap = new Map();
}

/*

```

```

    * This method acquires a subscription to a given state name,
creating a new WebSocketSubscription object if necessary.
    */
    AcquireSubscription(webSocketNotificationHandler, stateName) {
        if (this.#webSocket === null)
            throw new
Error("WSLibrary.WebSocketSubscriptionManager.InitializeWebSocket must be
called once before acquiring any subscription");

        let newActiveWebSocketSubscription = null;

        if (this.#WebSocketSubscriptionMap.has(stateName)) {
            throw new Error("attempting to subscribe to a state twice;
this version of the WSLibrary does not support multiple subscriptions");
        }
        else {
            newActiveWebSocketSubscription = new
WSLibrary.WebSocketSubscription(this.#webSocket,
webSocketNotificationHandler, stateName);
            this.#WebSocketSubscriptionMap.set(stateName,
newActiveWebSocketSubscription);
        }
        return newActiveWebSocketSubscription;
    }

/*
 * This method releases a subscription to a given state name.
 */
    ReleaseSubscription(stateName) // Continuando con el "to do " de
arriba, el parámetro tendría que ser webSocketSubscription para admitir
suscriptores múltiples.
{
    if (this.#WebSocketSubscriptionMap.has(stateName)) {
        const webSocketSubscription =
this.#WebSocketSubscriptionMap.get(stateName);
        webSocketSubscription.Unsubscribe();
        this.#WebSocketSubscriptionMap.delete(stateName);
    }
}
};
```

13.2.2 Paquete server:

13.2.2.1 Fichero “WSLibrary.js”:

```
/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *          to optimize client-server interaction by avoiding
polling.
 * File: WSLibrary.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
*/

if (typeof globalThis.WSLibrary !== 'undefined')
    throw new Error("Unexpected error: WSLibrary namespace already
defined");

const WSLibrary = {}; // This is a dummy empty object used as a
namespace.
globalThis.WSLibrary = WSLibrary;
```

13.2.2.2 Fichero “LibraryStateCache.js”:

```
/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *          to optimize client-server interaction by avoiding
polling.
 * File: LibraryStateCache.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
*/

WSLibrary.LibraryStateCache = class {
```

```

#libraryAdapter = null; // The object that manages the interaction
with the external API.
#stateName = ""; // The text identifier of the state that this cache
object stores.
#latestValue = null; // The latest retrieved value for the state.

/*
 * This is the constructor of the class.
 */
constructor(libraryAdapter, stateName) {
    this.#libraryAdapter = libraryAdapter;
    this.#stateName = stateName;

    this.UpdateValue();
}

/*
 * Method that updates the value, replacing the latest value and
returning true if the value has changed.
*/
async UpdateValue() {
    try {
        const currentStateValue = await
this.#libraryAdapter.GetStateValue(this.#stateName);
        const valueChanged = (currentStateValue !==
this.#latestValue);
        if (valueChanged) {
            this.#latestValue = currentStateValue;
        }
        return valueChanged;
    } catch (error) {
        console.error(`Error updating state value for
${this.#stateName}:`, error);
        return false;
    }
}

/*
 * Method that returns the latest value.
*/
async GetLatestValue() {
    return this.#latestValue;
}
};

```

13.2.2.3 Fichero “*LibraryStateSubscription.js*”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *              to optimize client-server interaction by avoiding
polling.
 * File: LibraryStateSubscription.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
*/

WSLibrary.LibraryStateSubscription = class {

    #libraryAdapter = null; // The object that manages the interaction
with the external API.
    #stateName = ""; // The text identifier of the state that this cache
object stores.
    #libraryStateCache = null; // The object that manages the cached
value for the managed state.
    #numberOfSubscribers = 0; // The number of current subscribers.

    /*
     * This is the constructor of the class.
     */
    constructor(libraryAdapter, stateName) {
        this.#libraryAdapter = libraryAdapter;
        this.#stateName = stateName;
    }

    /*
     * Method that updates the cached value if there are any subscribers.
     * It returns true if there has been a change in value and the web
socket clients must be notified.
     */
    async Update() {
        let valueChanged = false;

        if (this.#numberOfSubscribers > 0)
            valueChanged = await this.#libraryStateCache.UpdateValue();

        return valueChanged;
    }
}

```

```

/*
 * Method that returns the latest value.
 */
async GetLatestValue() {
    let latestValue = null;
    if (this.#numberOfSubscribers > 0 && this.#libraryStateCache)
        latestValue = await this.#libraryStateCache.GetLatestValue();

    return latestValue;
}

/*
 * Method that adds a subscriber.
 */
AddSubscriber() {
    if (this.#numberOfSubscribers === 0)
        this.#libraryStateCache = new
WSLibrary.LibraryStateCache(this.#libraryAdapter, this.#stateName);

    ++this.#numberOfSubscribers;
}

/*
 * Method that removes a subscriber.
 */
RemoveSubscriber() {
    if (this.#numberOfSubscribers > 0) {
        --this.#numberOfSubscribers;
        if (this.#numberOfSubscribers === 0)
            this.#libraryStateCache = null;
    } else {
        console.warn(`There are no subscribers to remove in the
${this.#stateName} state`);
    }
}

};


```

13.2.2.4 Fichero “*LibraryStateSubscriptionIterator.js*”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.

```

```

* File: LibraryStateSubscriptionIterator.js
* Year: 2025
* Version: 1.0
* All rights reserved.
*/
WSLibrary.LibraryStateSubscriptionIterator = class {

    #webSocketNotifier = null; // The object that knows how to send
    notifications to the web socket clients.
    #libraryStateSubscriptions = null; // The map to iterate over
    #states = []; // An array that stores a cached copy of the keys in
    the map above (assumed to be constant while this iterator object is being
    used)
    #currentIndex = 0; // The current iteration index

    constructor(webSocketNotifier, libraryStateSubscriptions) {
        // Parameter validation
        if (typeof webSocketNotifier !== 'object' || webSocketNotifier
        === null)
            throw new TypeError("webSocketNotifier must be a non-null
        object that implements SendNotification(stateName, newValue)");

        if (typeof webSocketNotifier.SendNotification !== 'function')
            throw new TypeError("webSocketNotifier must implement a
        SendNotification(stateName, newValue) method");

        if (!(libraryStateSubscriptions instanceof Map))
            throw new TypeError("libraryStateSubscriptions must be a Map
        instance");

        if (libraryStateSubscriptions.size === 0)
            throw new Error("libraryStateSubscriptions must be a non-
        empty Map");

        this.#webSocketNotifier = webSocketNotifier;
        this.#libraryStateSubscriptions = libraryStateSubscriptions;
        this.#states =
        Array.from(this.#libraryStateSubscriptions.keys());
        this.#currentIndex = 0;
    }

    Next() {
        const kMaximumNumberOfIterations = 100;
        const maximumIndex = this.#currentIndex +
        kMaximumNumberOfIterations;
        for (let index = this.#currentIndex; this.#currentIndex <
        this.#states.length && index < maximumIndex; ++index) {
            this.#UpdateState(index);
            ++this.#currentIndex;
        }
    }
}

```

```

    }

    IsDone() {
        return this.#currentIndex === this.#states.length;
    }

    Reset() {
        this.#currentIndex = 0;
    }

    async #UpdateState(index) {
        const stateName = this.#states[index];
        const libraryStateSubscription =
this.#libraryStateSubscriptions.get(stateName);

        try {
            // Await the async Update() result so we only notify when the
value actually changed.
            const changed = await libraryStateSubscription.Update();
            if (changed) {
                const latestValue = await
libraryStateSubscription.GetLatestValue();
                this.#webSocketNotifier.SendNotification(stateName,
latestValue);
            }
        } catch (error) {
            console.error(`Error updating the ${stateName} state:`, error);
        }
    }
};

}

```

13.2.2.5 Fichero “*LibraryStateSubscriptionManager.js*”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: LibraryStateSubscriptionManager.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
*/

```

```

WSLibrary.LibraryStateSubscriptionManager = class {

    #webSocketNotifier = null; // The object that knows how to send
    notifications to the web socket clients.
    #libraryAdapter = null; // The object that manages the interaction
    with the external API.
    #libraryStateSubscriptions = new Map(); // The associative map of
    state identifiers and objects that support the subscriptions for each
    individual state.
    #libraryStateSubscriptionIterator = null; // Object that will handle
    the update logic of the iteration over states and subscribers

    /*
     * This is the constructor of the class.
     */
    constructor(webSocketNotifier, libraryAdapter) {
        this.#webSocketNotifier = webSocketNotifier;
        this.#libraryAdapter = libraryAdapter;

        const stateNames = this.#libraryAdapter.GetStateNames();

        for (const stateName of stateNames)
            this.#libraryStateSubscriptions.set(stateName, new
WSLibrary.LibraryStateSubscription(this.#libraryAdapter, stateName));

        this.#libraryStateSubscriptionIterator = new
WSLibrary.LibraryStateSubscriptionIterator(this.#webSocketNotifier,
this.#libraryStateSubscriptions);
    }

    /*
     * Method that updates all the values checking for changes and
     notifying the web socket clients when a change is detected.
     */
    async Update() {
        if (this.#libraryStateSubscriptionIterator.IsDone())
            this.#libraryStateSubscriptionIterator.Reset();
        this.#libraryStateSubscriptionIterator.Next();
    }

    /*
     * Method that adds a subscriber for a particular supported state.
     */
    AddSubscriber(stateName) {
        const libraryStateSubscription =
this.#libraryStateSubscriptions.get(stateName);

        if (libraryStateSubscription)
            libraryStateSubscription.AddSubscriber();
    }
}

```

```

        else
            throw new Error("Unknown state name in
LibraryStateSubscriptionManager.AddSubscriber");
    }

/*
 * Method that removes a subscriber.
*/
RemoveSubscriber(stateName) {
    const libraryStateSubscription =
this.#libraryStateSubscriptions.get(stateName);

    if (libraryStateSubscription)
        libraryStateSubscription.RemoveSubscriber();
    else
        throw new Error("Unknown state name in
LibraryStateSubscriptionManager.RemoveSubscriber");
}

};

```

13.2.2.6 Fichero “*Server.js*”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: Server.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
 */

require("./WSLibrary.js");
require("./Settings.js");
require("./LibraryStateCache.js");
require("./LibraryStateSubscription.js");
require("./LibraryStateSubscriptionIterator.js");
require("./LibraryStateSubscriptionManager.js");
require("./YahooFinanceLibraryAdapter.js");
require("./WebSocketManager.js");

const host = "localhost";
const port = 3000;

```

```

const yahooFinanceLibraryAdapter = new
WSLibrary.YahooFinanceLibraryAdapter();
const webSocketManager = new WSLibrary.WebSocketManager(host, port,
yahooFinanceLibraryAdapter);

try {
    webSocketManager.Start();
    console.log(`WebSocket server started on ws://${host}:${port}`);
} catch (error) {
    console.error("Error starting the WebSocket server:", error);
    process.exit(1);
}

```

13.2.2.7 Fichero “Settings.js”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: Settings.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.
*/

const DEFAULT_UPDATE_INTERVAL = 5000; // Update interval in milliseconds

WSLibrary.Settings = class {

    #updateInterval = DEFAULT_UPDATE_INTERVAL; // The update interval to
use when polling for state changes.
    static sInstance = null; // The only instance of the class.
    static sGettingInstance = false; // A Boolean flag to assert that the
object is always constructed from within the GetInstance static method.

    /*
     * This is the constructor of the class.
    */
    constructor() {
        if (!WSLibrary.Settings.sGettingInstance)
            throw new Error("Invalid construction of Singleton class
bypassing the static GetInstance method which must always be used");
    }
}

```

```

    }

    /*
     * This is the singleton's GetInstance method that provides access to
     * the only object of this type.
    */
    static GetInstance() {
        if (WSLibrary.Settings.sInstance === null) {
            WSLibrary.Settings.sGettingInstance = true;
            WSLibrary.Settings.sInstance = new WSLibrary.Settings();
            WSLibrary.Settings.sGettingInstance = false;
        }
        return WSLibrary.Settings.sInstance;
    }

    /*
     * Accessor that sets the update interval.
    */
    SetUpdateInterval(updateInterval) {
        if (typeof updateInterval !== 'number' || updateInterval <= 0)
            throw new Error("El update interval debe ser un número
positivo.");
        this.#updateInterval = updateInterval;
    }

    /*
     * Accessor that gets the update interval.
    */
    GetUpdateInterval() {
        return this.#updateInterval;
    }
}

```

13.2.2.8 Fichero “*WebSocketManager.js*”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: WebSocketManager.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.

```

```

*/
const WebSocket = require('ws');

WSLibrary.WebSocketManager = class {
    #host = ""; // The host for this web socket server.
    #portNumber = ""; // The port number for this web socket server.
    #libraryAdapter = null; // The object that manages the interaction
    with the external API.
    #libraryStateSubscriptionManager = null; // The object that manages
    the interaction with the external API to handle the web socket
    notifications.
    #webSocketServer = null; // The object that manages the web socket
    server connection.
    #updateIntervalId = -1; // The timer id returned by setInterval.
    #clientSubscriptionMap = null; // The map that associates each
    existing client connection to an array of state names.

    /*
     * This is the constructor of the class.
     */
    constructor(host, portNumber, libraryAdapter) {
        this.#host = host;
        this.#portNumber = portNumber;
        this.#libraryAdapter = libraryAdapter;
        this.#libraryStateSubscriptionManager = new
        WSLibrary.LibraryStateSubscriptionManager(this, this.#libraryAdapter);
        this.#clientSubscriptionMap = new Map();
    }

    /*
     * Method that starts the web socket server.
     */
    Start() {
        this.#InitialiseWebSocket();
    }

    /*
     * Method that stops the web socket server.
     */
    Stop() {
        this.#FinaliseWebSocket();
    }

    /*
     * Method required by the "WebSocketNotifier" interface that
     LibraryStateSubscriptionManager requires as a construction parameter.
     */
    SendNotification(stateName, newValue) {
}

```

```

        this.#clientSubscriptionMap.forEach((subscribedStateNames,
webSocketClient) => {
    // Verify that the client is open before shipping
    if (webSocketClient.readyState === WebSocket.OPEN &&
subscribedStateNames.includes(stateName))
        webSocketClient.send(JSON.stringify({ stateName, newValue
})));
    });
}

/*
 * Private method that updates the subscriptions.
*/
async #Update() {
    if (this.#libraryStateSubscriptionManager)
        await this.#libraryStateSubscriptionManager.Update();
}

/*
 * Private method that initialises the web socket server.
*/
#InitialiseWebSocket() {
    this.#FinaliseWebSocket(); // Just in case

    this.#webSocketServer = new WebSocket.Server({
        host: this.#host,
        port: this.#portNumber
    });

    this.#webSocketServer.on("connection",
this.#InitialiseWebsocketClient.bind(this));

    const settings = WSLibrary.Settings.GetInstance();
    const updateInterval = settings.GetUpdateInterval();

    this.#updateIntervalId = setInterval(this.#Update.bind(this),
updateInterval);
}

/*
 * Private method that finalises the web socket server.
*/
#FinaliseWebSocket() {
    if (this.#updateIntervalId !== -1) {
        clearInterval(this.#updateIntervalId);
        this.#updateIntervalId = -1;
    }

    // Close active connections if necessary
    if (this.#webSocketServer)

```

```

        this.#WebSocketServer.close();
this.#WebSocketServer = null;
}

/*
 * Private method that initialises a web socket client connection.
*/
#InitialiseWebSocketClient(newWebSocketClient) {
    this.#clientSubscriptionMap.set(newWebSocketClient, []);

    newWebSocketClient.on("close", () =>
this.#FinaliseWebSocketClient(newWebSocketClient));
    newWebSocketClient.on("message", (message) =>
this.#HandleWebSocketClientMessage(newWebSocketClient, message));
    console.log('Client connected');
}

/*
 * Private method that finalises a web socket client connection.
*/
#FinaliseWebSocketClient(webSocketClient) {
    this.#clientSubscriptionMap.delete(webSocketClient);
    console.log('Client disconnected');
}

/*
 * Private method that handles a subscribe/unsubscribe message from a
web socket client.
*/
#HandleWebSocketClientMessage(webSocketClient, message) {
    let parsed;
    try {
        // Attempt to parse the incoming message as JSON
        parsed = JSON.parse(message);
    } catch (e) {
        console.error("Error parsing message: ", e);
        return;
    }

    // Validate that message has the necessary properties
    const { action, stateName } = parsed;
    if (!action || !stateName) {
        console.warn("Received message is missing required fields: ",
parsed);
        return;
    }

    if (action === "subscribe") {
        // Avoid duplicates

```

```

        const subscriptions =
this.#clientSubscriptionMap.get(webSocketClient) || [];
    if (!subscriptions.includes(stateName)) {
        subscriptions.push(stateName);
        this.#clientSubscriptionMap.set(webSocketClient,
subscriptions);
            // Notify the subscription manager that a new state has
been subscribed
            this.#libraryStateSubscriptionManager.AddSubscriber(state
Name);
        }
    }
    else if (action === "unsubscribe") {
        const subscriptions =
this.#clientSubscriptionMap.get(webSocketClient) || [];
        const index = subscriptions.indexOf(stateName);
        if (index > -1) {
            subscriptions.splice(index, 1);
            this.#clientSubscriptionMap.set(webSocketClient,
subscriptions);
                // Notify the subscription manager that a state has been
unsubscribed
                this.#libraryStateSubscriptionManager.RemoveSubscriber(st
ateName);
            }
        }
        else {
            // Log an unknown action for debugging purposes
            console.warn("Unknown action: ", action);
        }
    }
};

}

```

13.2.2.9 Fichero “*YahooFinanceLibraryAdapter.js*”:

```

/*
 * Author: Luis Miguel Gómez del Cueto
 * Contact: luismigmez@gmail.com
 * Final Degree Project - Software Engineering
 * University of Oviedo
 * Title: Real-Time Price Monitoring Using WebSockets
 * Description: This file is part of the final project that implements a
WebSocket-based API
 *                  to optimize client-server interaction by avoiding
polling.
 * File: YahooFinanceLibraryAdapter.js
 * Year: 2025
 * Version: 1.0
 * All rights reserved.

```

```

/*
WSLibrary.YahooFinanceLibraryAdapter = class {

    #yahooFinance = null; // The object that manages the interaction with
    the Yahoo API.

    /*
     * This is the constructor of the class.
    */
    constructor() {
        this.#yahooFinance = require('yahoo-finance2').default;
        this.#yahooFinance.suppressNotices(['yahooSurvey']);
    }

    /*
     * Method that returns the available state names, as required by the
     LibraryAdapter interface.
     * In this case, we return the Yahoo quotes as the valid state names.
    */
    GetStateNames() {
        return ["AAPL", "GOOGL", "TSLA", "AMZN", "MSFT", "NFLX", "BABA",
        "NVDA", "META", "INTC",
            "BTC-USD", "ETH-USD", "SOL-USD", "ADA-USD", "XRP-USD", "DOGE-
        USD", "DOT-USD", "MATIC-USD",
            "EURUSD=X", "GBPUSD=X", "USDJPY=X", "USDCAD=X", "AUDUSD=X",
        "NZDUSD=X", "USDCHF=X"];
    }

    /*
     * Method that returns the available state names, as required by the
     LibraryAdapter interface.
    */
    async GetStateValue(stateName) {
        if (!this.GetStateNames().includes(stateName))
            throw new Error("Invalid state name: " + stateName);

        // Save the original stdout/stderr write functions for
        restoration later.
        const originalStdoutWrite = process.stdout.write;
        const originalStderrWrite = process.stderr.write;
        try {
            // Temporarily disable stdout and stderr to avoid warnings
            from yahoo-finance2
            process.stdout.write = () => { };
            process.stderr.write = () => { };

            const result = await this.#yahooFinance.quote(stateName);
            const price = result.regularMarketPrice;
        }
    }
}

```

```

        return price;
    } catch (error) {
        console.error("Error fetching state value for", stateName,
":", error);
        throw error;
    } finally {
        // Ensure that stdout and stderr are always restored
        process.stdout.write = originalStdoutWrite;
        process.stderr.write = originalStderrWrite;
    }
}

};

```

13.2.2.10 Fichero “package.json”

```
{
  "name": "server",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1",
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "axios": "^1.8.2",
    "dotenv": "^16.4.7",
    "express": "^4.21.2",
    "ws": "^8.18.1",
    "yahoo-finance2": "^2.13.3"
  }
}
```