

Integración de características de tiempo real en aplicaciones web

Trabajo de fin de grado

GRADO EN INGENIERÍA INFORMÁTICA



VNiVERSiDAD
D SALAMANCA

CAMPUS DE EXCELENCIA INTERNACIONAL

Septiembre de 2014

Autora

Alicia Boya García

Tutora

Ángeles M^a Moreno Montero

D.^a Ángeles M^a Moreno Montero, profesora del Departamento de Informática y Automática de la Universidad de Salamanca.

CERTIFICA:

Que el trabajo titulado “*Integración de características de tiempo real en aplicaciones web*” ha sido realizado por D. Alicia Boya García, con DNI 71038815A y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado de la Titulación Grado en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 02 de enero de 2021.

D^a Ángeles M^a Moreno Montero
Dpto. de Informática y Automática
Universidad de Salamanca

Resumen

Este documento describe el desarrollo de un nuevo framework para facilitar el desarrollo de aplicaciones web con capacidades de tiempo real basadas en patrones. A este framework se le ha dado el nombre de *Snorky*.

En este contexto entenderemos tiempo real como la actualización inmediata de contenidos en aplicaciones web y la comunicación entre los diferentes procesos en red participantes con latencia mínima.

Se ha desarrollado un protocolo para intercambiar notificaciones entre un servidor web tradicional y un servidor *WebSocket*. Siguiendo este protocolo los cambios pueden ser distribuidos a las aplicaciones clientes de forma instantánea. La implementación del protocolo se ha encapsulado en varias bibliotecas (*conectores*) que lo utilizan requiriendo al programador de aplicaciones web la escritura de una cantidad mínima de código.

El sistema dispone de una interfaz de programación extensible en la cual el programador puede definir *servicios* que ofrezcan funcionalidad adicional específica de su aplicación, basadas en modelos como Publicación-Suscripción o comunicación cliente a cliente. Estos servicios se han diseñado siguiendo un esquema de llamadas a procedimientos remotos basado en el lenguaje JSON (*JavaScript Object Notation*) para organizar la funcionalidad en forma de métodos y parámetros en el lado del servidor y utilizarla a través de sencillas llamadas en el lado del cliente.

Se ha integrado el protocolo de notificaciones en la plataforma extensible sistema para poder ofrecer toda la funcionalidad desde un único servidor y se ha elaborado documentación junto con varias aplicaciones de ejemplo para validar y demostrar el uso del sistema.

Para el desarrollo de este trabajo se ha utilizado el lenguaje *Python*, utilizando *Tornado* para la conectividad en red, así como *JavaScript* para el conector del lado del cliente.

Palabras clave: *Aplicaciones web, WebSocket, JSON, framework, Python.*

Abstract

This document describes a new real-time web development framework based on patterns, Snorky.

In this context, we will understand real time as instant data updates and minimal-latency communication.

A protocol has been developed in order to exchange notifications between a traditional web server and a WebSocket server. Following this protocol, change notifications can be distributed to the client applications instantly. Several libraries (*connectors*) have been developed in order to allow using this protocol requiring the least effort to the programmer.

An extensible programming interface has been developed, allowing the programmer to define *services* which can offer application-specific functionality, leveraging models like publish-subscribe or client to client communication. These services have been designed following a remote procedure call scheme based on the JSON (*JavaScript Object Notation*) language, organizing functionality in form of methods and parameters on the server side and simple calls on the client side.

The data notification protocol has been added to this extensible platform, offering all the functionality mentioned above in a single server. Documentation has been redacted and several example applications have been developed.

The *Python* language and the *Tornado* networking library are used in the server part, whilst JavaScript is used in the client-side connectors.

Keywords: *Web development, WebSocket, JSON, framework, Python.*

1. Introducción	1
2. Objetivos	3
3. Antecedentes del desarrollo de aplicaciones web en tiempo real	5
3.1. El protocolo HTTP sin estado	5
3.2. Sondeo con <i>Meta Refresh</i>	6
3.3. Sondeo AJAX	7
3.4. El sondeo largo	8
3.5. El problema C10K	8
3.6. Programación asíncrona	9
3.7. WebSocket	10
3.8. Los frameworks del lado del cliente	11
3.9. Este trabajo de fin de grado	12
4. Metodología de desarrollo	13
4.1. Proponer un objetivo corto	13
4.2. Dibujar primero	14
4.3. Codifica cuando te sientas cómodo	14
4.4. Probar a fondo	14
4.5. Reflexionar y buscar otro problema	15
4.6. Conclusión	16
5. Herramientas utilizadas	17
5.1. Lenguajes de programación	17
5.2. Bibliotecas y frameworks	17
5.3. Software de desarrollo	18
5.4. Software libre y gratuito	18
6. Notación UML	19
6.1. Tipos	19
6.2. Anotaciones en plantillas	20
6.3. Asociaciones tipadas	20
6.4. Código de colores	21
6.5. Propiedades en lugar de atributos	22
6.6. Visibilidad no estricta	23

7. Aspectos relevantes	25
7.1. Sincronización con bases de datos	25
7.1.1. Requisitos	25
7.1.2. Análisis	26
7.1.3. Diseño	30
7.1.4. Implementación y pruebas	32
7.1.5. La primera demostración	34
7.1.6. El conector JavaScript	34
7.2. La aplicación de incidencias	37
7.2.1. Requisitos	37
7.2.2. Autenticación	37
7.2.3. Registro	38
7.2.4. Modelos de datos	39
7.2.5. ORM de Django	40
7.2.6. Integración con el ORM de Django	41
7.2.7. Django REST Framework	42
7.2.8. Integración con Django REST Framework	43
7.2.9. Reportar una incidencia	43
7.3. Servicios personalizados	46
7.3.1. Patrones	47
7.3.2. Requisitos	51
7.3.3. Análisis	52
7.3.4. Diseño del protocolo	53
7.3.5. Diseño de clases del servidor	55
7.3.6. Signatura de llamadas RPC	55
7.3.7. Implementación del servidor	58
7.3.8. Conector JavaScript	59
7.3.9. Servicios de ejemplo	63
7.3.10. El servicio de mensajería	65
7.3.11. Integración con AngularJS	67
7.3.12. El servicio Pub Sub y la interfaz de backend	68
7.4. Integración	69
7.4.1. Campos renombrados	71
7.4.2. Diseño	71
7.4.3. Implementación del servidor	71
7.4.4. Conector JavaScript	74
7.5. Documentación	74
7.5.1. La herramienta de documentación	75
7.5.2. Estilo	75
7.5.3. Contenidos	77
7.5.4. Página web	77
8. Conclusiones y líneas de trabajo futuro	81
9. Bibliografía	87
9.1. Referencia técnica	87
9.2. Libros	88

Introducción

Lejos queda el día en el que la web era un sistema de distribución de documentos, donde la comunicación entre cliente y servidor se limitaba al intercambio de páginas.

La aparición de JavaScript en 1995 introdujo una nueva tendencia: los navegadores adquirieron la capacidad de ejecutar código para alterar el aspecto de las páginas y realizar tareas que hasta entonces se consideraban responsabilidad exclusiva del servidor, tales como la validación de datos.

Si bien en aquellos años el uso de JavaScript era limitado, la tendencia de mover funcionalidad al lado del cliente continuaría de forma lenta pero constante en años posteriores.

Una nueva API llamada XMLHttpRequest surgiría cuatro años más tarde, permitiendo a estas nuevas aplicaciones del lado del cliente comunicarse con servidores remotos.

Actualmente JavaScript es una parte integral de la plataforma web. Habiendo dejado de ser la compatibilidad el problema que era hace pocos años, los desarrolladores tratan de explotar al máximo esta tecnología para ofrecer mejores interfaces de usuario con las mínimas esperas posibles, maximizando la eficiencia de los servidores y de la red.

WebSocket, introducido como estándar a finales de 2011, ha dado un paso más para la comunicación en tiempo real en aplicaciones web, entendiendo como tiempo real la comunicación con latencia mínima. Este nuevo protocolo permite que la parte servidor y la parte cliente de una aplicación web se comuniquen enviando mensajes en cualquier orden, no siendo dependientes del tradicional modelo *pregunta-respuesta*.

En una Internet con cada vez más usuarios, las aplicaciones web colaborativas son una tendencia importante. A menudo, para que estas aplicaciones sean eficientes y realmente permitan a los usuarios ayudarse de forma mutua sin que la máquina suponga un inconveniente para ello, es necesario que estén dotadas de capacidades de comunicación en tiempo real tales como mensajería instantánea o sincronización con modelos de datos remotos.

Este trabajo trata de proveer herramientas para la construcción de este tipo de aplicaciones, basándose en las tecnologías desarrolladas durante todo este tiempo.

El resto del documento se estructura de la siguiente forma. En el siguiente capítulo se enumeran los objetivos del proyecto. Para continuar se enuncian los antecedentes que han motivado el desarrollo de este trabajo. A continuación se explica la metodología que se ha utilizado, desde el planteamiento de requisitos hasta la implementación y en el capítulo siguiente se enuncian las herramientas técnicas que se han utilizado para la realización del trabajo. Para continuar se explica la notación utilizada en los diagramas UML del proyecto. Los temas posteriores cubren el desarrollo de las diferentes etapas del proyecto: construcción de un servidor prototipo que permita sincronizar vistas con bases de datos, desarrollo de una aplicación basada en el prototipo, establecimiento de una arquitectura que permita incorporar nuevos servicios de tiempo real, integración del prototipo de sincronización en la nueva arquitectura y documentación del proyecto. Finalmente se enuncian las conclusiones y posibles líneas futuras del proyecto.

En el CD adjunto a esta memoria se incluyen varios anexos, así como los artefactos producidos durante el desarrollo del trabajo.

- Anexo I: Documentación de análisis.
- Anexo II: Documentación de diseño.
- Anexo III: Manual del usuario.
- Anexo IV: Referencia del programador.
- Código fuente del framework, así como de todos sus conectores.
- Aplicaciones de ejemplo desarrolladas.

Objetivos

El objetivo principal de este trabajo es el desarrollo de un framework para facilitar el desarrollo de aplicaciones web con capacidades de tiempo real.

Siendo más concretos los objetivos que tiene que cumplir el sistema son:

- Construir un servidor WebSocket que permita sincronización de vistas en una aplicación web con bases de datos remotas, ofreciendo actualizaciones en tiempo real.
- Permitir la ampliación del framework de manera que el programador pueda incorporar nuevas funcionalidades específicas para su aplicación en caso de necesitarlas.
- Incluir conectores JavaScript para utilizar la funcionalidad de tiempo real de forma fácil en el desarrollo de aplicaciones web del lado del cliente.
- Incluir un conector para comunicación entre servidores que demuestre tanto el uso de las interfaces básicas del servidor como las ventajas de construir este tipo de software adicional.
- Respalidar el funcionamiento de las funcionalidades del framework con ejemplos de su utilización en prototipos de aplicaciones web.

Antecedentes del desarrollo de aplicaciones web en tiempo real

El protocolo HTTP sin estado

HTTP es el protocolo que utilizan los servidores web para transmitir las páginas a los navegadores. HTTP fue diseñado como un protocolo sin estado: cada vez que un cliente envía una petición el servidor procesa toda la información requerida – habitualmente leyendo o escribiendo en una base de datos y envía una página web como respuesta. Una vez hecho esto, la comunicación se da por finalizada y cualquiera de las dos partes puede cerrar la conexión.

Todas las peticiones HTTP son tratadas de manera independiente por el servidor, incluso si proceden del mismo cliente. Esta naturaleza sin estado hace el protocolo simple y escalable, facilitando el desarrollo de proxies, balanceadores de carga y otras tecnologías importantes.

Sin embargo, la carencia de estado de HTTP también es uno de sus primeras debilidades:

- Debido a que de forma predeterminada no existe un mecanismo para el seguimiento de clientes, cuando es necesario usar autenticación, cada petición debe contener una prueba para el servidor de que el mensaje ha sido enviado en virtud de la voluntad de un usuario legítimo. Esta es la razón por la que las cookies existen.
- Una vez que una página ha sido servida, la comunicación se acaba. No hay ningún método definido en el protocolo para que el servidor pueda enviar información actualizada una vez que la página ha sido servida.

En este capítulo se explican algunas soluciones que a lo largo de la historia del desarrollo web se han usado para solventar el segundo problema y cómo este proyecto encaja en ellas.

Sondeo con *Meta Refresh*

La primera solución utilizada ampliamente para obtener información actualizada de un servidor web es el sondeo, es decir, repetir la petición a intervalos periódicos. Cuando esto se hace con una página web en un navegador, se le suele conocer con el nombre de *refresco* o *recarga*.

Esta solución es con diferencia la más sencilla y todavía se utiliza en algunas aplicaciones (ej. el gestor de integración continua² Jenkins¹). El sondeo también es utilizado ocasionalmente en otros protocolos sin estado, como POP3 e IMAP.

HTTP hace muy fácil implementar esta estrategia. El siguiente código basta para hacer que una página web se recargue cada 3 segundos:

```
<meta http-equiv="refresh" content="3">
```

Sin embargo, hay que considerar serios inconvenientes con este método:

- No es comunicación en tiempo real: Puesto que estamos sondeando, sólo obtenemos información actualizada cuando el refresco ocurre. Hasta entonces esa información se queda sólo en el servidor.
- Elección del intervalo de refresco: A más corto sea el intervalo, antes recibiremos la información actualizada, pero también sufriremos los problemas descritos a continuación:
 - Rendimiento: Cada refresco invoca una nueva petición para obtener la página entera desde el servidor. Esto provoca una cantidad de tráfico considerable para el cliente, especialmente en conexiones lentas o de transferencia limitada (ej. conexiones móviles).

Puesto que la primera carga y sucesivos refrescos producen exactamente la misma petición, incurren en aproximadamente la misma carga para el servidor, que puede convertirse en un problema dependiendo del número de usuarios, el tiempo medio que un usuario mantiene la página abierta y la velocidad con la que las peticiones son atendidas.

Si un servidor recibe 10.000 peticiones de carga por hora, un usuario medio deja abierta la página 10 minutos y hacemos un refresco cada 3 minutos, el servidor necesitará soportar 40.000 peticiones por hora (una para la carga inicial más tres refrescos por cada usuario).

- Mala experiencia de usuario: Cada recarga de la página a menudo ocasiona que el navegador parpadee mientras los contenidos de la página son colocados y actualizados, convierte el icono de la pestaña en un reloj durante el tiempo de descarga y puede producir saltos si el usuario movió la barra de desplazamiento

² Un gestor de integración continua es un software que ejecuta tareas rutinarias relacionadas con el desarrollo de software, tales como ejecutar tests unitarios o compilar versiones. Generalmente este tipo de sistemas se utilizan para que los desarrolladores puedan saber si una versión de desarrollo es fiable.

¹ <http://jenkins-ci.org/>

durante ese tiempo. Debido al parpadeo, a menudo es difícil que el usuario se dé cuenta de qué ha cambiado realmente, causando confusión y frustración.

Sondeo AJAX

Empezando con Internet Explorer 5.0 en 1999 y pronto seguido por todos los demás navegadores, las aplicaciones web ganaron la capacidad de enviar peticiones arbitrarias al servidor web y procesar las respuestas en el lado del cliente.

Esta característica, popularizada pocos años después, es la base de las aplicaciones web modernas. Las peticiones enviadas con esta tecnología son llamadas *peticiones AJAX*, siendo *AJAX* un acrónimo de *Asynchronous Javascript And XML*, a pesar de que no es necesario que la respuesta a la petición esté en formato XML (*Extensible Markup Language*). De hecho, en la mayoría de aplicaciones web recientes el formato de serialización XML está siendo reemplazado por JSON (*JavaScript Object Notation*), un lenguaje alternativo más simple.

Las aplicaciones AJAX pueden enviar peticiones especializadas al servidor adecuadas a sus necesidades específicas. Para realizar un sondeo periódico no es necesario pedir toda la página completa, sino que es posible utilizar peticiones especializadas que requieran menos tráfico para el cliente y menos sobrecarga de procesamiento para el servidor.

Por ejemplo, la aplicación web de un periódico on-line, en lugar de preguntar cada 5 minutos *¿Cuáles son las noticias de Salamanca?*, podría preguntarlo sólo la primera vez que la página es cargada, y el resto del tiempo, cada cinco minutos, preguntar simplemente *¿Hay alguna noticia nueva en Salamanca que haya ocurrido desde las 11:00?*

Mientras que la primera consulta – *¿Cuáles son las noticias de Salamanca?* – requiere consultar un conjunto de artículos en la base de datos y volcarlos en la respuesta, la segunda consulta sólo requiere hacer una consulta mucho más limitada, que supone menor carga a la base de datos y que producirá una respuesta mucho más pequeña, consumiendo mucho menos tráfico.

AJAX no especifica cómo los datos recibidos de una respuesta se traducirán en los elementos de la interfaz. Para este propósito es necesario escribir código JavaScript específico. A menudo la misión de este código consiste en decodificar la respuesta y crear, actualizar o borrar elementos HTML.

El sondeo AJAX soluciona muchos de los problemas del refresco básico: la eficiencia se incrementa considerablemente tanto para el cliente como para el servidor y la experiencia de usuario mejora gratamente. Sin embargo, el intervalo de sondeo sigue siendo un problema importante.

Además, aparece un nuevo problema: la manipulación de elementos HTML en el lado del cliente se vuelve más compleja. Las aplicaciones AJAX tienden a ser muy complejas en el lado del cliente, requiriendo a menudo más código en la parte cliente que en el servidor.

El sondeo largo

Para tratar de solucionar el problema del intervalo de sondeo y enviar información actualizada en tiempo real a los navegadores han ido surgiendo muchas propuestas.

Una de esas propuestas es conocida como *Sondeo Largo (Long Polling)*. El sondeo largo funciona de forma similar al sondeo AJAX, con una diferencia importante: cuando no hay datos interesantes que enviar al cliente, el servidor se abstiene de enviar una respuesta al cliente, esperando a que realmente aparezcan datos interesantes para resolverla. La petición será resuelta una vez que dichos datos sean creados o después de un tiempo máximo de espera (*timeout*).

El lado del cliente envía una petición de sondeo largo una vez que la página ha cargado y la repite cada vez que la petición anterior finaliza.

De esta manera realmente es posible obtener información con mínima latencia: la petición es respondida en cuanto hay datos disponibles, sin necesidad de esperar por el intervalo de sondeo. Si la petición no es respondida después del tiempo máximo de espera será necesario enviar otra, pero la elección de este tiempo de espera no penaliza la latencia con la que los datos son recibidos.

Un caso de uso habitual del sondeo largo son los chats on-line. Por ejemplo, *Meebo*³ fue una de las primeras aplicaciones web en popularizar el uso de esta técnica. El estándar de mensajería instantánea XMPP definió una extensión oficial con el nombre de BOSH¹⁵ (*Bidirectional-streams Over Synchronous HTTP*) como alternativa a TCP básico, extendiendo el protocolo a la web e incluso a clientes de escritorio en redes con políticas estrictas que sólo permiten tráfico HTTP.

Por otro lado, HTTP no fue pensado para tecnologías como el sondeo largo y en consecuencia son bastante difíciles de implementar. La situación se complica más cuando es necesario hacer peticiones a un servidor en otro dominio. Por ejemplo, una de las técnicas para realizar sondeo largo contra otro dominio consiste en cargar dentro de un marco invisible un documento HTML infinito donde el servidor envía una etiqueta `<script>` para cada mensaje que quiere enviar al cliente y deja la conexión abierta después. Funciona, pero no es muy elegante.

El problema C10K

La mejora de latencia de las técnicas de sondeo largo introdujo a cambio un incremento de la complejidad del desarrollo del lado del servidor. Cuando se hacía un verdadero sondeo, el servidor consultaba la base de datos en cuanto recibía la petición y contestaba inmediatamente. El sondeo largo requiere en lugar de eso un esquema publicación-subscripción (a menudo llamado *Pub Sub*).

³ *Meebo* fue un cliente de mensajería instantánea que funcionaba como una aplicación web, hasta que en 2012 fue descontinuado después de que Google comprara la compañía.

¹⁵ <http://xmpp.org/extensions/xep-0124.html>

Un servidor *Pub Sub* realiza el seguimiento de un gran número de conexiones desde usuarios que *suscriben* notificaciones. Cada vez que ocurre un evento relevante, como un cambio en la base de datos, se envía una notificación a los clientes suscritos.

Los servidores *Pub Sub* necesitan mantener un gran número de conexiones simultáneas, las cuales se mantendrán inactivas durante la mayor parte del tiempo esperando notificaciones.

Esto difiere bastante del comportamiento que los servidores HTTP tradicionales esperan. A menudo estos servidores asignan un hilo del sistema a cada petición, que debe resolverla tan rápido como sea posible. Esta es la manera en la que funcionan Apache, PHP, Tomcat y muchos otros sistemas web.

Aunque un servidor *Pub Sub* puede ser construido basado en este esquema tradicional mediante esperas y primitivas de sincronización como el paso de mensajes, la eficiencia se ve seriamente afectada.

Los servidores web tradicionales no fueron pensados para tener tantas peticiones ejecutándose simultáneamente, sino sólo unas pocas, mientras el resto estarían esperando, encoladas. Como consecuencia, además de necesitar cambios de configuración, cada petición activa consume más recursos de los deseados, principalmente memoria y primitivas del sistema operativo.

Los hilos también se convierten fácilmente en un recurso escaso. Ejecutar demasiados hilos a menudo lleva a los sistemas operativos a volverse muy lentos y dejar de responder con celeridad. Este problema es conocido como el problema C10K⁴ debido a un popular escrito acerca de la dificultad de escribir un servidor que soporte 10.000 conexiones utilizando hilos.

Desde que el artículo del problema C10K fue escrito en 2003, los sistemas operativos actuales soportan hilos de una forma más eficiente y los ordenadores se han vuelto más potentes, pero aun así la solución más adoptada del artículo es no usar hilos en absoluto o limitarse a un número específico de ellos, habitualmente uno por núcleo del procesador para sacarle el máximo provecho a la máquina.

Programación asíncrona

Para manejar conexiones concurrentes sin hilos, se utiliza el modelo de programación asíncrona en lugar del tradicional modelo de programación síncrona. En él, el núcleo del sistema operativo es liberado de la tarea de cambiar de contexto entre el procesamiento de varias peticiones. En su lugar, este trabajo es asumido por la aplicación servidora, que puede controlar con gran precisión cuántos recursos necesita para procesar cada petición (a menudo sólo unos pocos bytes en estructuras de datos).

Los servidores asíncronos tienen un bucle de eventos que se utiliza para elegir la siguiente petición en ser atendida, de manera similar al trabajo que hace el planificador del sistema operativo en un sistema síncrono.

⁴ <http://www.kegel.com/c10k.html>

Para ser notificados de las conexiones entrantes y de cuándo éstas conexiones reciben datos, los servidores asíncronos hacen uso de APIs eficientes de selección de sockets del sistema operativo, como *epoll* (Linux), *kqueue* (FreeBSD) o *IOCP* (Windows).

El servidor comienza el procesamiento de una petición en cuanto la decodifica. Durante el tiempo que está procesando, el servidor no hace otra cosa, por lo tanto es crucial que no haga ninguna llamada al sistema bloqueante durante ese tiempo, ya que de lo contrario bloquearía todo el servidor.

Sin embargo, el procesamiento de una petición sí puede ser suspendido en cualquier momento, siendo suficiente con no enviar ningún dato y volver al bucle de eventos. Las peticiones suspendidas pueden continuar su procesamiento como respuesta a eventos tales como intervalos de tiempo, mensajes de otros clientes o eventos del sistema operativo.

La programación asíncrona está siendo utilizada con éxito en muchos desarrollos recientes, incluyendo los servidores HTTP *nginx*⁵ y *lighttpd*⁶, que ofrecen gran rendimiento para uso como servidores de archivos y proxies inversos. Nuevas plataformas de programación asíncrona como *Tornado*⁷ y *Node.js*⁸ establecen una base sólida para crear servidores escalables con relativa facilidad.

Nota: En este trabajo de fin de grado se usa programación asíncrona, basada en *Tornado*.

La programación asíncrona es diferente de la programación síncrona en muchos aspectos. Por un lado, el código asíncrono es más determinista ya que, al usar un solo hilo, podemos asegurar que el servidor no está ejecutando más código simultáneamente. Esto evita necesitar primitivas de sincronización tales como bloqueos o mutexes, fuentes habituales de complejidad y problemas de rendimiento.

Por otro lado, algunas operaciones son más difíciles de realizar sin bloquear el servidor, lo cual es necesario para que el servidor asíncrono funcione de forma eficiente. Las operaciones de disco son un ejemplo de esto¹⁶.

WebSocket

Aunque la técnica del sondeo largo funciona, se comenzó a pensar en la definición de un verdadero estándar eficiente de comunicación bidireccional. Como resultado emergió *WebSocket*, estandarizado en 2011.

El protocolo *WebSocket* maneja comunicación bidireccional entre un cliente y un servidor web con sobrecarga mínima. *WebSocket* funciona sobre TCP o TLS (si se requiere cifrado)

⁵ <http://nginx.org/en/>

⁶ <http://www.lighttpd.net/>

⁷ <http://www.tornadoweb.org/en/stable/>

⁸ <http://nodejs.org/>

¹⁶ <http://bert-hubert.blogspot.com.es/2012/05/on-linux-asynchronous-file-io.html>

y también incluye un protocolo para marcar el comienzo y fin de los mensajes (*framing*).

Una conexión WebSocket se inicia con una petición HTTP, habitualmente dirigida a los puertos estándares de HTTP, 80 o 443, lo que le permite funcionar incluso en entornos con cortafuegos restrictivos.

A pesar de que existe la necesidad de que el usuario disponga de un navegador reciente para que pueda utilizar WebSocket, la mayoría de navegadores en uso ya lo soportan de forma nativa. Internet Explorer 8 y 9 son los únicos navegadores con un uso significativo que todavía no soportan WebSocket.

Además, la simplicidad de la API de WebSocket, que sólo tiene tres métodos (*open*, *send* y *close*) y tres eventos (*on open*, *on message*, *on close*) ha propiciado un esfuerzo para crear capas de compatibilidad que permitan utilizar algo similar a WebSocket en navegadores que no lo soporten. Una de estas capas de compatibilidad es SockJS⁹, el cual utiliza varias de las técnicas alternativas anteriormente comentadas para proveer un canal de comunicación alternativo a WebSocket en navegadores viejos.

Nota: Este trabajo se diseñará para soportar SockJS como alternativa a WebSocket.

Los frameworks del lado del cliente

A medida que las aplicaciones web han ido incorporando más y más código en la parte del cliente la complejidad de las mismas se ha incrementado de forma notoria.

Nos estamos acercando lentamente al punto donde los servidores ya no envían páginas formateadas con datos incluidos, sino que proveen por un lado los datos y por otro el código para dar formato a esos datos y mostrarlos en una interfaz de usuario. Las interfaces resultantes son mucho más ágiles y requieren muchas menos esperas, ofreciendo una experiencia similar a las aplicaciones de escritorio.

Varios proyectos han surgido con el objetivo de manejar la complejidad de estas interfaces. Habitualmente proveen algún mecanismo de plantillas que permite expresar de forma declarativa dónde se colocarán los datos dentro de la interfaz, e implementan patrones basados en la arquitectura Modelo-Vista-Controlador para responder a la interacción del usuario y actualizar la interfaz con poco código.

Algunos de esos proyectos son AngularJS¹⁰, Ember.js¹¹, React¹² y Knockout.js¹³, aunque hay muchos otros.

⁹ <https://github.com/sockjs>

¹⁰ <https://angularjs.org/>

¹¹ <http://emberjs.com/>

¹² <http://facebook.github.io/react/>

¹³ <http://knockoutjs.com/>

Este trabajo de fin de grado

Los frameworks del lado del cliente han cambiado las reglas de la programación web. Ser capaz de mostrar y actualizar datos en la interfaz de forma declarativa limpia el camino para construir mejores interfaces que ofrezcan mejores experiencias de usuario.

Si un usuario carga una página con mensajes que debe moderar en un foro, está esperando a que cierre una subasta o espera con impaciencia un mensaje, a menudo la única forma que tiene de obtener información actualizada de forma rápida es pulsar la tecla F5 muchas veces, simplemente porque la sincronización en tiempo real lleva mucho esfuerzo y pocos programadores invierten tiempo en implementarla.

Este trabajo trata de cerrar la brecha, ofreciendo sincronización de datos entre servidores y clientes y al mismo tiempo requiriendo el mínimo esfuerzo posible a los desarrolladores.

Metodología de desarrollo

El desarrollo de software es un problema difícil: Nunca puedes saber con seguridad qué problemas surgirán y a menudo es difícil saber si una solución es realmente buena hasta que ha sido implementada.

Este mal afecta a la mayoría de proyectos, sin embargo este está especialmente en una mala posición debido a que la complejidad del objetivo es alta, y no hay mucha experiencia en el entorno que pueda ser tomada como referencia.

Por otro lado, el objetivo es algo vago: *Hacer el desarrollo de aplicaciones web con características de tiempo real más fácil*, ¿pero cómo?

Este capítulo explica la metodología que se ha utilizado en el desarrollo de este trabajo.

Proponer un objetivo corto

El primer paso es preguntarse cómo podemos acercarnos al objetivo. Es difícil saberlo con seguridad, pero pensando un rato es fácil dar con unos cuantos problemas que solucionar.

Si tenemos varios, debemos elegir aquellos que parezcan más relacionados con el éxito del proyecto, preferiblemente si incurrir en riesgo para el mismo. Si el proyecto no puede terminar exitosamente, cuanto antes se llegue a esa conclusión, mejor. Por otro lado, si el problema planteado es solucionado con éxito, no sólo conseguimos un importante progreso para el proyecto, sino que además ganamos confianza para seguir con el resto del desarrollo.

El primer problema objetivo del proyecto es lograr la sincronización de datos entre un servidor WebSocket y una base de datos. Este problema es crítico, no ser capaz de lograr esta sincronización significaría que el software resultante no proveería mucha más ayuda que las soluciones existentes.

Por este motivo, la sincronización con bases de datos fue el primer objetivo del proyecto, descrito en Sincronización con bases de datos.

Dibujar primero

Convertir ideas en código no es nada fácil. Las ideas ocultan muchos detalles del mundo real que se acaban traduciendo en muchos cambios de código, razonamiento y rediseño. Aquí es donde UML sirve de ayuda.

Sin importar si dibujas diagramas o no, casi siempre descubrirás fallos y omisiones de diseño que necesitas solucionar para que el sistema funcione: falta una asociación, cierta funcionalidad debería moverse a una nueva clase, no hay suficientes estructuras de datos para limpiar las suscripciones cuando un cliente desconecta...

Todos esos fallos se pueden detectar y arreglar tanto en código como en diagramas. Sin embargo, normalmente es mucho más fácil y rápido arreglar esos problemas cuando no hay código de por medio.

Además, ser capaz de capturar todas las entidades y relaciones en diagramas visuales introduce una excelente fuente de documentación, especialmente en fases tempranas del proyecto, así como a la hora de introducir nuevos desarrolladores.

Para utilizar los diagramas de forma efectiva en los lenguajes de programación propuestos se introducirán algunas extensiones al lenguaje UML. Estas extensiones son descritas en Notación UML.

Codifica cuando te sientas cómodo

Una vez que hay suficientes diagramas para cubrir la implementación y han sido revisados lo suficiente como para pensar que no falta nada importante, es el momento de empezar a escribir código.

Para el proceso de codificación se abrirá un editor de texto, se escogerá una clase que no tenga dependencias y se codificará empezando por ella y siguiendo por aquellas que dependan de ella.

Es probable que durante el diseño fueran olvidados algunos detalles importantes. En estos casos, los errores deben ser subsanados primero en los diagramas y después en el código. Afortunadamente, los problemas más graves ya habrán sido detectados en la etapa de diseño, por lo que generalmente esta etapa no será demasiado problemática.

Probar a fondo

Codificar el software entero e intentar depurar después todos los problemas que surjan al final es simplemente no productivo. Depurar un proyecto entero requiere mucho esfuerzo por cada problema que se quiere tratar, y no es común que se den muchas omisiones por las que el sistema funciona en algunos casos pero falla en otros.

Aquí es donde las pruebas unitarias entran en juego. Se debe codificar un caso de prueba para cada clase no trivial. Los casos de prueba deben ser unitarios: cada caso de prueba debe probar la funcionalidad de una sola clase, o como mucho de una clase y de otras pocas muy relacionadas con ella.

Para probar relaciones entre clases se utilizarán clases sustituto (*mock*). Estas clases se definen en los mismos archivos que los casos de prueba y tienen sólo los métodos y propiedades necesarios para ejecutar el caso de prueba. JavaScript y Python hacen esta práctica muy fácil al ser lenguajes dinámicos sin comprobación estricta de tipos.

Cada caso de prueba se escribirá después de escribir el código de la clase correspondiente. Este método es muy efectivo ya que recibes retroalimentación inmediata sobre la calidad del código que acabas de escribir.

En la medida de lo posible, las pruebas se escribirán teniendo en cuenta todos los posibles casos límite en los argumentos de los métodos. El código fuente de la clase se revisará en busca de condicionales y excepciones y se comprobará que todas son probadas. Esta metodología de prueba es conocida como *pruebas de caja gris* (*gray box testing*) y es mucho más fácil de seguir cuando las pruebas son escritas después de las clases.

A menudo se encontrarán casos límite no tratados en el código fuente mientras se preparan las pruebas. En algunos casos esos errores pueden provenir incluso del diseño. Puesto que las clases se codifican de forma incremental, el esfuerzo necesario para arreglar este tipo de problemas es reducido.

Además, una batería de pruebas automática es una herramienta extremadamente valiosa cuando se quieren hacer cambios en el código o en el diseño, ya que te dan la garantía de que el código funciona antes y después del cambio. Esto es especialmente importante en lenguajes de tipado dinámico como los que se utilizan en este proyecto.

Como conclusión, las pruebas automáticas son una gran medida de control de calidad, eliminando tiempo de depuración, verificando código y diseño e identificando problemas cuando menos cuesta arreglarlos.

Reflexionar y buscar otro problema

Una vez que el problema planteado ha sido solucionado es tiempo de reflexionar acerca de la calidad de la solución. ¿Funciona bien la solución propuesta? ¿La API es fácil de usar? ¿Quizás el diseño podría haber sido mejor? ¿Falta algo?

Si el problema fue resuelto en cierto grado, entonces se ha demostrado que una solución es posible e incluso se ha desarrollado una implementación funcional.

Si el problema no fue resuelto correctamente, es momento de buscar alternativas. ¿Es el problema tan importante? ¿Hay otras maneras de lidiar con él?

En este momento podemos volver al primer paso y buscar otro problema que necesite una

solución. Los artefactos generados en esta iteración podrán ser reutilizados en iteraciones siguientes.

Conclusión

La metodología de desarrollo es un aspecto fundamental del desarrollo de un proyecto. Mediante este esquema iterativo e incremental podemos desarrollar software de complejidad elevada minimizando los riesgos implícitos en el proyecto. Las pruebas automáticas nos permiten escribir código de forma cómoda y asegurar la calidad del proyecto desde los primeros diseños hasta tener una implementación funcional.

Herramientas utilizadas

Para el desarrollo de este trabajo se han utilizado una serie de herramientas de terceros. Este capítulo trata de enumerar las más importantes.

Lenguajes de programación

Python es el lenguaje de programación escogido para desarrollo de la parte servidor. Las razones para su elección han sido:

- Experiencia probada en el desarrollo web.
- Gran cantidad de bibliotecas y frameworks con excelente documentación.
- Experiencia previa del autor con el lenguaje.

Para la parte cliente, se ha elegido JavaScript por necesidad: es el lenguaje que soportan los navegadores.

Bibliotecas y frameworks

- Tornado¹⁷: Framework para desarrollo web y manejo asíncrono de redes.
- unittest¹⁸: Sistema para pruebas unitarias de facto en Python.
- Jasmine¹⁹: Sistema de pruebas para JavaScript.
- Grunt²⁰: Ejecutor de tareas para JavaScript.

¹⁷ <https://github.com/tornadoweb/tornado/>

¹⁸ <https://docs.python.org/2/library/unittest.html>

¹⁹ <https://jasmine.github.io/>

²⁰ <http://gruntjs.com/>

- Sphinx²¹: Generador de documentación en Python. También se ha utilizado para realizar este documento.

Software de desarrollo

- CPython, el intérprete predeterminado de Python.
- Vim²² como editor de texto, con varios añadidos y plantillas para escribir código de forma eficiente en varios lenguajes. La configuración utilizada por el autor está accesible públicamente en GitHub²³.
- El navegador Chrome²⁴, que incluye consola JavaScript y depurador.
- git²⁵ como sistema de control de versiones.
- Visual Paradigm²⁶ como software de modelado UML.
- El sistema operativo Arch Linux²⁷, el escritorio KDE²⁸ y las herramientas de consola GNU.

Software libre y gratuito

Todo el software expuesto en este capítulo, excepto Visual Paradigm, es gratuito y de código abierto.

Que las herramientas de desarrollo sean gratuitas es deseable ya que garantiza que nuevos contribuyentes al proyecto no se encontrarán dificultades al intentar modificar el código y potencialmente unirse al mismo.

Que las herramientas no sean sólo gratuitas sino también libres en el sentido definido por organizaciones como Open Source Initiative²⁹ reduce el riesgo de que decisiones externas de negocio supongan un problema para el desarrollo del proyecto en el futuro.

A pesar de ello, tras haber probado diferentes programas de modelado UML, ninguna de las alternativas libres mostró calidad suficiente para ser adoptada. Visual Paradigm mostró suficiente calidad como para servir para el trabajo y los precios parecieron razonables, así que finalmente se ha elegido en lugar de alternativas más abiertas pero menos maduras.

²¹ <http://sphinx-doc.org/>

²² <http://www.vim.org/>

²³ <https://github.com/ntrrgc/dotfiles>

²⁴ <https://www.google.com/chrome/browser/>

²⁵ <http://git-scm.com/>

²⁶ <http://visual-paradigm.com/>

²⁷ <https://www.archlinux.org/>

²⁸ <http://kde.org/>

²⁹ <http://opensource.org/>

Notación UML

En los diagramas de este trabajo se utilizarán ciertas convenciones específicas no contempladas en el lenguaje UML básico.

El propósito de este capítulo es describir estas convenciones.

Tipos

Tanto Python como JavaScript son lenguajes dinámicos que no requieren interfaces de tipos: cada método puede aceptar cualquier tipo de dato en sus argumentos y puede devolver cualquier objeto de cualquier tipo como valor de retorno o no devolver nada. Esta libertad se utiliza a menudo en la realización de las pruebas.

A pesar de ello, con vistas a la documentación y al diseño es importante conocer con seguridad qué tipo de objetos pueden ser devueltos como respuesta de una llamada, por lo que todos los diagramas de diseño del proyecto usarán tipado fuerte.

Los nombres de los tipos utilizados están basados en los nombres originales de los tipos de Python:

- `str`: Una cadena de caracteres. Puede contener caracteres Unicode fuera del ASCII.
- `bytes`: Una cadena de bytes. No siempre contendrá texto, puede tener datos binarios arbitrarios.
- `object`: Cualquier cosa. Nótese que en Python todos los tipos de datos son objetos, incluso los números y el `None` (`null` en otros lenguajes).
- `void`: Usado como valor de retorno de una función para hacer explícito que nunca devuelve nada.

Algunos tipos son plantillas, utilizando notación símbolos de menor y mayor:

- `list<Type>`: Una lista (array en JavaScript) compuesta de elementos de la clase `Type`.

- `dict<Key, Value>`: Un diccionario (conjunto asociativo) cuyas claves son objetos de la clase `Key` y cuyos valores son objetos de la clase `Value`.
- `set<Key>`: Un *conjunto* de elementos de la clase `Key`. Un conjunto es similar a un diccionario, pero sin valores, sólo con claves.
- `iter<Type>`: Un objeto generador o iterable. Si se coloca este objeto en un bucle `for` devolverá un objeto de la clase `Type` en cada iteración. Listas, diccionarios y conjuntos son iterables. En Python es posible crear nuevos objetos iterables adicionales.

Anotaciones en plantillas

Los tipos especificados en las plantillas pueden añadir anotaciones opcionales para explicar su propósito. Por ejemplo, si tenemos un diccionario que indiza objetos `Subscription` por el campo `token`, podríamos escribir la siguiente declaración de tipo:

```
dict<token : str, Subscription>
```

Cualquier cantidad de argumentos puede llevar anotaciones.

Asociaciones tipadas

UML incluye una notación simple para especificar la multiplicidad de las asociaciones. Los valores más utilizados en el campo de multiplicidad son los siguientes:

- `1`: Cada instancia está asociada con exactamente una instancia de la otra clase.
- `0..1`: Cada instancia puede o no estar asociada con una instancia de la otra clase.
- `0..*`: Cada instancia puede estar asociada con cualquier número de instancias de la otra clase.

Al transformar los diagramas en código, las dos primeras cadenas de multiplicidad no dan muchos problemas: se implementarán como una referencia al tipo de la otra clase.

`0..*` es algo más ambiguo, en cambio. UML acepta algunos modificadores como `{ordered}` y `{unique}` que pueden evitar cierta confusión, pero no toda.

Una solución común es utilizar el tipo lista o array para las asociaciones `0..*`, a veces tratándolas como un detalle menor de implementación.

Sin embargo, en este proyecto se hacen necesarias abundantes asociaciones, a menudo provistas por diccionarios para que las búsquedas sean tan rápidas como sea posible (a menudo con complejidad logarítmica $O(1)$). Esto no es un detalle de implementación, sino un problema de diseño muy importante: el rendimiento está más relacionado con la elección de las estructuras de datos adecuadas que con ninguna otra cosa. Además,

escoger buenas estructuras de datos hace que el código sea más corto, más directo y más fácil de entender (ej. evitando bucles for para hacer búsquedas).

Para expresar las relaciones entre clases de forma precisa se hará uso de la notación de tipos descrita en los puntos anteriores.

Por ejemplo, en el ejemplo mostrado en la 6.1, la clase `ServiceManager` está asociada con un conjunto de servicios, indizados en un diccionario por el campo `name` de tipo `str`.

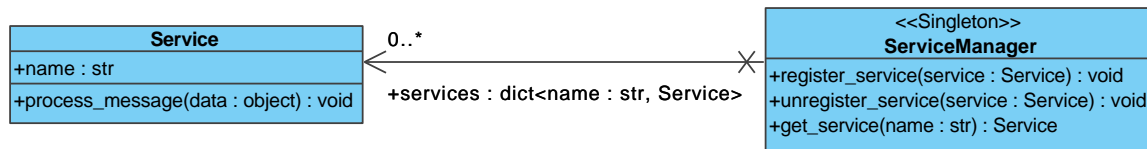


Figura 6.1: Una relación tipada.

Una asociación puede ser de cualquier tipo siempre que permita de alguna manera llegar al objeto de la otra clase de alguna forma. En el ejemplo mostrado en la 6.2, la clase `ClientMessage` tiene una asociación con `Service` a través de un campo de tipo cadena de texto.

A pesar de que no tiene una referencia explícita al objeto `Service`, puede conseguir una de forma trivial pasándole `service_name` al método `get_service()` de su colaboradora, `ServiceManager`.

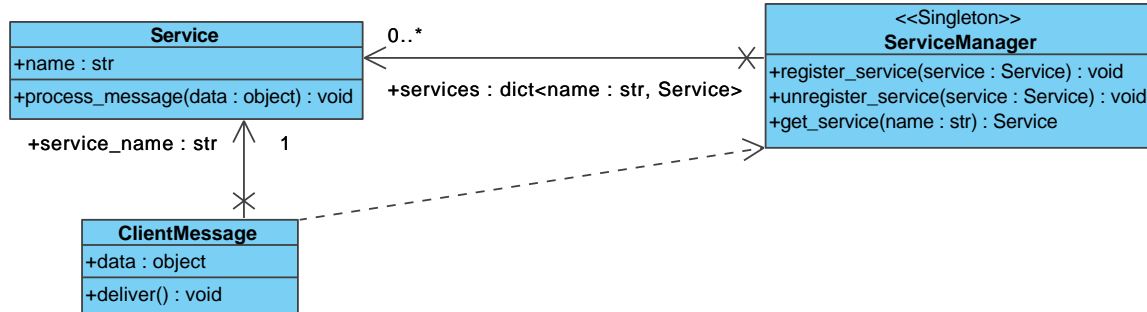


Figura 6.2: Una relación tipada menos obvia.

Código de colores

Durante el desarrollo se utilizará un código de colores.

El color de fondo de las clases en los diagramas transmitirá un significado adicional:

- **Azul:** La clase no se ha implementado en código.
- **Amarillo:** La clase ha sido implementada.
- **Verde:** Esta clase representa una clase definida en un framework externo. La clase

real puede tener más métodos o propiedades no representados en ésta. Sólo aquellos métodos y propiedades utilizados en este proyecto serán representados en la clase verde. La 6.3 muestra un ejemplo de esto.

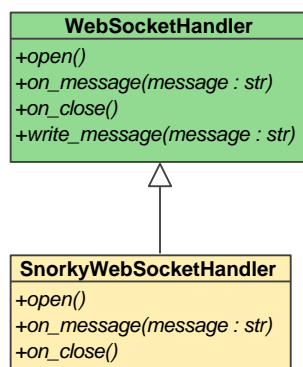


Figura 6.3: SnorkyWebSocketHandler extiende la clase WebSocketHandler de Tornado.

Propiedades en lugar de atributos

En algunos lenguajes, como C++ y Java, los atributos públicos se consideran como un mal diseño, promoviendo en su lugar métodos de acceso `get()` y `set()`.

Esta posición está codificada en algunas métricas de calidad del software como el *Factor de Atributos Ocultos*³⁰ (*Attribute Hiding Factor*) cuyos autores afirman que debería ostentar un valor muy cercano al 100%.

Los métodos de acceso tienen una desventaja: introducen código adicional que raramente es importante para el lector, ya que rara vez hay código interesante en los métodos de acceso.

Por otro lado, la razón para utilizar métodos de acceso reside en que si no se hiciera y luego en un punto futuro del desarrollo del sistema se quisiera ejecutar código adicional cuando un valor del objeto es leído o escrito, no se podría hacer sin romper la interfaz. Sería necesario pues encontrar todos los usos de atributos y reemplazarlos por los correspondientes métodos de acceso.

Aunque ésta es una argumentación válida para C++ y Java, no es suficiente para ciertos lenguajes. Algunos lenguajes como Python, JavaScript y C# tienen el concepto de propiedades, sustituyendo o complementando a los atributos.

El uso de una propiedad es sintácticamente idéntico al de un atributo. Sin embargo las propiedades pueden implementar comportamiento adicional similar al de los métodos de acceso en cualquier momento sin romper la interfaz.

En este tipo de lenguajes, la métrica de atributos ocultos no tiene sentido porque no hay atributos como tal. Sin embargo, puesto que el uso de propiedades es muy similar al de atributos, las propiedades se representarán como atributos en los diagramas UML.

³⁰ <http://www.aivosto.com/project/help/pm-oo-mood.html>

Visibilidad no estricta

Python y JavaScript carecen de la noción de visibilidad de atributos y métodos. Todas las propiedades y métodos definidas por un objeto son accesibles a cualquiera que obtenga una referencia al objeto. Es responsabilidad del usuario hacer un uso responsable de los mismos.

Por esta razón, la visibilidad de métodos y atributos especificada en los diagramas UML del proyecto es meramente informativa y la mayoría de las propiedades y métodos se representarán como miembros públicos.

Cuando un cierto método o propiedad es para uso interno de la clase y no debería ser usado desde fuera, o al menos no sin verificar con cuidado las ramificaciones de hacerlo, se seguirá la convención de añadir un guión bajo (_) como prefijo al nombre de la propiedad o el método.

Aspectos relevantes

Sincronización con bases de datos

Como ya se ha comentado en el capítulo Objetivos, el primer objetivo del proyecto es construir un servidor WebSocket que permita de sincronización en tiempo real con bases de datos. Para este propósito se escribirá un servidor prototipo.

Requisitos

El objetivo en esta fase es construir un servidor WebSocket que reciba notificaciones de cambios de una base de datos y las distribuya entre los clientes interesados.

El servidor WebSocket podría estar situado en un proceso independiente a aquel del servidor que entrega las páginas HTML y hace consultas a la base de datos.

El sistema deberá:

- Recibir notificaciones de cambios de la base de datos.
- Permitir a los clientes WebSocket subscribirse a ciertos tipos de notificaciones.
- Enviar a los clientes WebSocket notificaciones de cambios adecuadas a sus subscripciones.

Adicionalmente, hay algunos requisitos no funcionales muy importantes:

- Realizar todas las operaciones de forma segura, no permitiendo a ningún usuario recibir notificaciones de cambios ocurridos sobre datos a los que no tiene derecho de acceso ni a ningún equipo hacerse pasar por el servidor principal para entregar notificaciones falsas.
- Funcionar sin condiciones de carrera bajo asunciones razonables.

Análisis

Partes implicadas

Como siempre pasa en el desarrollo web, el proyecto se compone de un sistema distribuido: diferentes procesos se comunican mediante paso de mensajes. Las siguientes partes están implicadas en el sistema:

- Navegadores.
- El servidor WebSocket.
- El servidor web principal. Este servidor es responsable del acceso a la base de datos. Habitualmente será un servidor síncrono.

Ya conocidos los participantes del sistema, el siguiente paso es analizar la interacción entre ellos.

Responsabilidades básicas

El cliente del navegador necesitará conectar a los dos notificaciones, tanto al servidor principal como al servidor WebSocket. Hay dos tipos de datos que el cliente recibirá:

- Los datos iniciales. Estos serán los datos que estaban en la base de datos cuando la petición del cliente fue recibida.
- Los datos notificados. Estos serán los datos que el cliente recibe del servidor WebSocket según los cambios van ocurriendo.

Se ha considerado apropiado que el servidor principal sea quien envíe los datos iniciales, por las siguientes razones:

- Los servidores web que no soportan funcionalidad de tiempo real ya hacen esto. No quitarles esta tarea facilitaría la adaptación de dichos sistemas.
- El soporte de conectores de bases de datos es algo mejor en servidores síncronos.
- Muchos frameworks para desarrollo web ya proveen de herramientas adecuadas para hacer este acceso de forma fácil, por ejemplo Django REST Framework³¹.

El problema de la sincronización

Un enfoque inicial para solucionar el problema podría ser el siguiente.

1. Conectar al servidor principal y enviarle una petición para obtener los datos actuales. Por ejemplo, *dime los vuelos disponibles desde Salamanca*.

³¹ <http://www.django-rest-framework.org/>

2. Una vez que los datos han sido recibidos, conectar al servidor WebSocket y pedirle las actualizaciones que se produzcan. Por ejemplo, por ejemplo *avísame cuando haya llegado un nuevo vuelo disponible desde Salamanca*.

Hay un problema de sincronismo con este método. Una nueva unidad de datos, ej. un vuelo disponible desde Salamanca, puede aparecer desde el instante en el que la petición 1 es resuelta pero antes de que la petición 2 lo sea. El usuario perdería su vuelo.

Para evitar este problema, tanto los datos iniciales como los datos notificados deben estar **sincronizados**, de manera que no haya huecos entre la información de ambas peticiones. De alguna manera los cambios ocurridos entre ambas peticiones deben guardarse en algún sitio para ser enviados al cliente cuando conecte.

El problema de la autenticación

Las aplicaciones web pueden contar con complejos controles de acceso. Deberían aplicarse los mismos controles de acceso en el servidor WebSocket que en el servidor principal para evitar fugas de datos no autorizados a través de las notificaciones de cambios.

Como consecuencia, es preferible dejar la tarea de autenticación al servidor principal para evitar codificarla dos veces, lo cual sería propenso a errores, incluidos fallos de seguridad.

Suscripciones autorizadas

Se propone la siguiente técnica que soluciona ambos problemas antes mencionados.

En vez de que el cliente dirija la petición de consulta al servidor WebSocket, la enviará al servidor principal, reutilizando la interfaz utilizada para pedir datos, pero añadiendo un parámetro para indicar que el cliente quiere también una suscripción.

El servidor principal comprobará los permisos del cliente, rechazando la petición si no son suficientes para la petición. Si los permisos son correctos, enviará un mensaje `authorizeSubscription` al servidor WebSocket especificando sobre qué clase de datos el cliente debe recibir notificaciones. El servidor WebSocket devolverá un código aleatorio en respuesta, el *testigo de suscripción* o *subscription token*, que el servidor principal enviará al cliente junto con los datos iniciales.

El cliente visualizará los datos iniciales y conectará con el servidor WebSocket, entregándole el testigo de la suscripción. Una vez hecho esto, el cliente recibirá desde este servidor las notificaciones que el servidor principal autorizó.

Este sistema permite que el servidor WebSocket almacene las notificaciones recibidas para el cliente hasta que finalmente conecte.

Si el servidor principal no procesa más peticiones al mismo tiempo, es fácil de entender que esto efectivamente soluciona el problema de sincronización.

Si el servidor principal procesa varias peticiones simultáneamente, puede ocurrir que ocurra un cambio durante el intervalo de tiempo entre que el servidor autoriza la suscripción y consulta la base de datos – por ejemplo porque el servidor procesara una petición de actualización durante ese tiempo. En este caso el cliente recibirá dos veces el dato nuevo: una vez como respuesta de la consulta y otra como notificación. El cliente deberá estar preparado para este tipo de casos, no señalando una notificación de cambio no aplicable como un error, puesto que puede ser consecuencia de esta situación.

Filtrar notificaciones

Por razones de eficiencia y seguridad, el servidor WebSocket no debería enviar todas las notificaciones a todos los clientes. En lugar de ello, las suscripciones deberían funcionar como filtros, permitiendo al servidor principal especificar qué datos deben recibir los clientes y permitiendo repartirlas de forma eficiente.

Las clases responsables de este reparto son conocidas como *repartidores (dealers)*

Canal de backend

La comunicación entre el servidor principal y el servidor WebSocket se hará sobre un canal distinto del que utilizan los clientes finales para comunicarse con el servidor WebSocket.

A este canal se le denomina *backend* porque compone la parte no visible detrás de la interfaz que ofrece el servidor WebSocket al público.

La seguridad en este canal es importante, ya que un intruso podría enviar notificaciones falsas afectando de forma negativa a los usuarios finales o realizar autorizaciones de suscripciones ilegítimas para obtener acceso a datos privilegiados.

Por este motivo, y como primera medida de seguridad, este canal puede estar situado dentro de la red interna de una organización, no siendo accesible desde Internet.

Como medida de seguridad adicional, todas las peticiones realizadas a través de este canal deberán incluir una clave compartida por el servidor principal y el servidor WebSocket. El objetivo de esta clave es controlar de forma más fina el acceso a las características protegidas del servidor, no dejando el servidor desprotegido frente al resto de máquinas de la red, así como evitar que un ataque a un software no relacionado en la máquina que ejecuta el servidor WebSocket pueda propagarse a éste.

Puesto que a menudo los clientes de esta interfaz de *backend*, por diseño de sus respectivas tecnologías, no podrán mantener conexiones por tiempo prolongado, este canal funcionará mediante el protocolo HTTP clásico en vez de WebSocket.

Nótese que por defecto este canal no va cifrado. Esto no es problema cuando tanto el servidor principal como el servidor WebSocket están alojados en la misma máquina física o en una red segura. Si esto no puede garantizarse, debe usarse cifrado. Una posibilidad es añadir soporte de TLS (*Transport Layer Security*) al canal. Sin embargo, con

el objetivo de reducir el tiempo empleado en la negociación de parámetros de seguridad, se recomienda utilizar soluciones alternativas que permitan reutilizar conexiones, por ejemplo el reenvío de puertos de OpenSSH³².

Diagramas

El diagrama en la figura 7.1 muestra de manera simplificada las clases definidas en este proceso de análisis.

- **Delta** es un nombre corto para las notificaciones de cambios de datos. En Informática, no es raro asociar esta letra griega con las actualizaciones incrementales, como puede verse en términos tales como *delta RPM* o *delta backup*.
- **Client** representa una conexión al servidor WebSocket.
- **Subscription** representa una petición para recibir notificaciones de cambios de datos para un potencial cliente.
 - Cada suscripción es asociada con uno o más repartidores, para los cuales especifican un campo con el tipo de datos que los clientes quieren.
 - Cada suscripción puede estar asociada con un cliente, si ese cliente ha conectado y adquirido el testigo de suscripción.
- **Dealer** implementa un filtrado de las notificaciones de cambio de datos, repartiéndolas a los clientes que poseen suscripciones adecuadas.

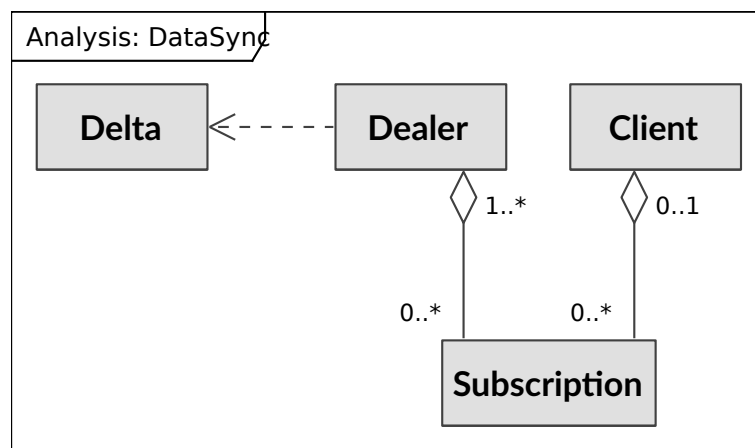


Figura 7.1: Las clases definidas en el análisis.

El diagrama de secuencia en la figura 7.2 muestra la interacción general entre el navegador, el servidor principal y el servidor WebSocket:

1. El cliente envía una petición al servidor principal pidiéndole los datos e indicando que quiere una suscripción.

³² <http://www.openssh.com/>

- a) Siguiendo la colaboración descrita en *Suscripciones autorizadas*, el servidor principal envía un mensaje de autorización al servidor WebSocket, obteniendo un testigo de suscripción como respuesta.
 - b) El servidor principal consulta la base de datos para obtener los datos que el cliente ha pedido.
 - c) El servidor principal responde al cliente enviando como respuesta tanto los datos iniciales que ha consultado en la base de datos como el testigo de la suscripción.
2. El cliente negocia una conexión con el servidor WebSocket.
 3. Una vez establecida la conexión con el servidor WebSocket, el cliente le entrega el testigo de suscripción. El cliente podría tener varios testigos si hubiera hecho diferentes consultas al servidor principal.
 4. El servidor WebSocket confirma la adquisición del testigo, haciéndole entrega a partir de este momento de todas las notificaciones asociadas a su suscripción.
 5. Cada vez que ocurre un cambio en la base de datos se envía una notificación al servidor WebSocket.
- Esta funcionalidad puede conseguirse añadiéndola manualmente a los métodos de manipulación de la base de datos o de forma más automatizada con escuchando eventos de un framework de acceso a datos.
6. En cualquier momento el cliente puede cancelar una suscripción para dejar de recibir notificaciones asociadas a ella. Si el cliente tiene varias suscripciones, sólo aquella suscripción cancelada dejará de emitir notificaciones.
 7. Una vez que la conexión entre el cliente y el servidor WebSocket es finalizada, el servidor WebSocket cancela todas las suscripciones que quedaran activas y elimina la información relativa a la conexión de sus estructuras de datos.

Diseño

Partiendo de las entidades definidas en el análisis, se refinan las clases, se añaden métodos y propiedades y se definen las colaboraciones de forma más concreta.

Esta parte ha presentado una notable dificultad, ya que omisiones sutiles a menudo han hecho que el sistema diseñado no sea implementable y haya tenido que ser repensado. Los diagramas de secuencia han resultado ser muy útiles para probar que todas las clases ofrecían toda la funcionalidad necesaria.

Aun así, algunas omisiones han llegado latentes a la fase de implementación, requiriendo volver temporalmente al diseño.

El diagrama de clases en la figura 7.3 muestra el resultado del diseño. Algunos puntos

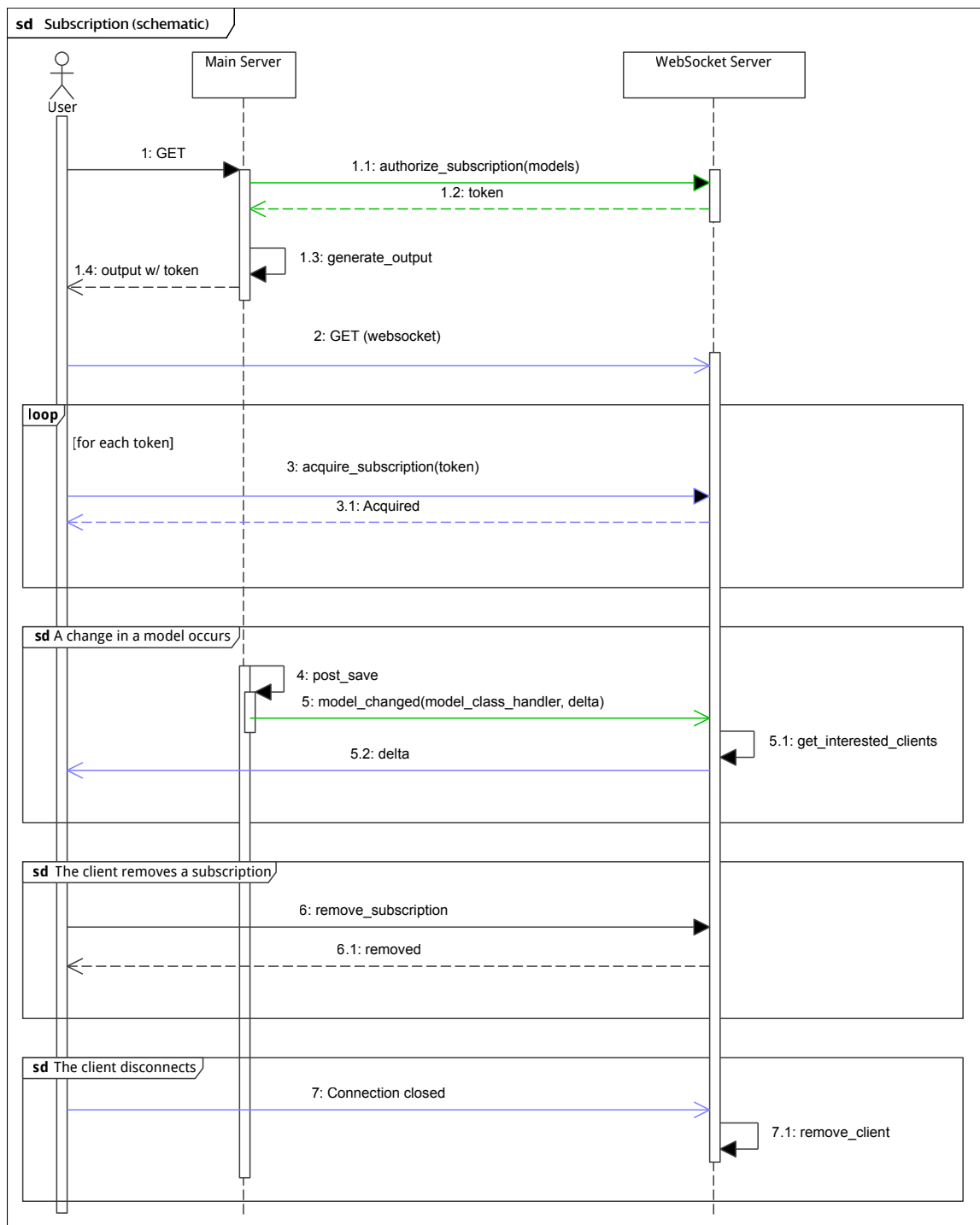


Figura 7.2: La interacción entre el cliente, el servidor principal y el servidor WebSocket, en un diagrama de secuencia UML.

destacables respecto del análisis son:

- La clase `Subscription` se ha separado en dos: `Subscription` y `SubscriptionItem` para soportar suscripciones a varios repartidores.
- Dos repartidores base se han implementado para ofrecer filtros simples:
 - `SimpleDealer` permite a los clientes suscribirse a cambios sobre modelos de datos filtrados por el valor de un campo concreto, ej. *vuelos filtrados por origen*.

La clase `Dealer` provee una interfaz básica para desarrollar repartidores que ofrezcan filtros más complejos.

- Algunas interacciones complejas se han abstraído en una clase `Facade`.
- Los clientes `WebSocket` conectan a la clase `FrontendWebSocketHandler` o `FrontendSockJSHandler`. El servidor principal conecta a las subclases de `BackendRequestHandler`.

Implementación y pruebas

Una vez completados los diagramas de diseño se procedió a realizar la implementación.

La implementación y las pruebas se han hecho de manera simultánea. Empezando con una clase sin dependencias, se ha implementado la misma en código e inmediatamente después se comenzaron a escribir las pruebas unitarias para la misma. Una vez terminada y probada la clase, se eligió otra, y así sucesivamente.

La implementación se ha escrito en Python, teniendo cuidado de que el resultado funcionara tanto en Python 2 como en Python 3. Las pruebas unitarias se han sido escritas con el módulo de pruebas por defecto de Python, `unittest`.

Las pruebas unitarias fueron extremadamente importantes en esta etapa ya que el software no podría ser probado con una aplicación web real hasta que todas o casi todas las clases estuvieran implementadas.

Algunas clases de diseño fueron descartadas en la implementación:

- `ClientManager` permitiría realizar un seguimiento de clientes conectados para, por ejemplo, implementar características tales como consultar el número de clientes activos. Puesto que al momento de terminar el resto de la implementación, ninguna funcionalidad dependía de esta clase, se acabó descartando. No obstante, se dejó colocada en los diagramas de diseño por si resultara de útil en posteriores iteraciones.
- `Server` era una clase ficticia que servía para explicar qué entidades el programador necesitaría manejar para escribir un servidor `WebSocket` basado en este prototipo. Sin embargo, no es necesario que todas estas entidades se controlen desde una clase.



Figura 7.3: Clases de diseño del prototipo. Las clases externas tienen un color diferente, tal como se definió en *Código de colores*.

La primera demostración

Eventualmente todo el código estaba escrito y todos las pruebas se ejecutaban correctamente, pero no había demostración visual de que el software funcionaba.

Para este propósito, se han codificado varias aplicaciones de ejemplo. La captura de pantalla en la figura 7.4 muestra la primera de ellas. Consta de una simple tabla, que recuerda a la pantalla previa a iniciar una partida en un videojuego multijugador. En realidad es una simple tabla cuyos datos no tienen ningún significado concreto, pero que sirve para comprobar el funcionamiento de la sincronización.

Los datos son almacenados en memoria por un servidor web escrito en python con el framework Flask³³. Este servidor tiene el rol de *servidor principal* definido en el análisis. Se comunica con el servidor WebSocket para enviar las notificaciones de cambios y autorizar las subscripciones.

Hay dos repartidores:

- AllPlayers. Hereda de BroadcastDealer y envía notificaciones de todos los cambios ocurridos en todos los objetos Player.
- PlayersWithColor. Hereda de SimpleDealer y envía notificaciones de cambios para los objetos Player cuyo campo color coincida con el que los usuarios suscriban.

El siguiente fragmento de código muestra el código requerido para estas clases repartidoras:

```
class AllPlayers(BroadcastDealer):
    name = "all_players"
    model_class_name = "Player"

class PlayersWithColor(SimpleDealer):
    name = "player_with_color"
    model_class_name = "Player"

    def get_key_for_model(self, model):
        return model["color"]
```

El conector JavaScript

Para escribir la demostración de los jugadores ha sido necesario no solo código en el lado del servidor, sino también en el lado del cliente, en JavaScript.

Comunicarse con el servidor WebSocket directamente utilizando solamente la API de WebSocket o SockJS es demasiado bajo nivel. En consecuencia, se ha tratado de aislar el comportamiento relacionado con sincronizar las vistas de la aplicación en patrones reutilizables.

³³ <http://flask.pocoo.org/>



Figura 7.4: La demo de jugadores. Cada vez que una entrada es añadida, actualizada o borrada, el cambio se refleja en todas las ventanas de todos los clientes, en tiempo real.

Para empezar, en la demostración de jugadores se utiliza un framework del lado del cliente, AngularJS. Esto ha ayudado a separar la vista (la interfaz que ve el usuario) del modelo (los datos).

La vista es una plantilla HTML procesada por AngularJS. Por ejemplo, el siguiente ejemplo es el código necesario para mostrar una fila de la tabla de jugadores. El código para mostrar el formulario de edición ha sido omitido.

```
<div class="tr-group" ng-repeat="player in players">
  <form class="tr" ng-if="!thereIsPlayerEditForm(player)">
    <div class="td wide">
      {{ player.name }}
    </div>
    <div class="td wide" ng-style="{ 'color': player.color }">
      {{ player.color }}
    </div>
    <div class="td"><button ng-click="showEditPlayerForm(player)">Edit</button></div>
    <div class="td"><button ng-click="deletePlayer(player)">Delete</button></div>
  </form>
</div>
```

El modelo es un array de JavaScript que contiene la lista de objetos jugador disponibles. Este array se rellena como resultado de solicitar una consulta al servidor principal.

```
// This makes an AJAX call to GET /players/{args}
Restangular.all('players').getList(args)
  .then(function(players) {
```

```
// This function is called when the server replies  
  
// Assign the received data to players model  
$scope.players = players.data;  
});
```

La variable `$scope.players` usada ahí es la misma que es usada en la primera línea del código de la plantilla HTML, `ng-repeat="player in players"`. El bloque con esa anotación será repetido para cada objeto en el array `$scope.players`. Se crea una nueva variable ceñida al ámbito del bloque repetido con el nombre `player`, haciendo referencia en cada caso a un objeto diferente del array.

AngularJS monitorizará este array para detectar cambios, añadiendo, actualizando o eliminando filas en la vista según las entradas del array son añadidas, actualizadas o borradas.

Esto significa que mientras que utilicemos un framework que permita separar el modelo de la vista como AngularJS, añadir un nuevo elemento visible será tan sencillo como añadir un nuevo elemento a un array.

Por este motivo, un patrón evidente será aplicar las notificaciones recibidas del servidor WebSocket sobre este array: añadir los elementos nuevos recibidos, así como encontrar los elementos que han cambiado y reemplazarlos con aquellos recibidos por el servidor o borrarlos si procediera.

Este patrón se ha codificado, junto con el código para enviar y recibir peticiones a través del canal WebSocket, resultando en una biblioteca JavaScript, `miau.js`.

Las partes del código específicas de AngularJS se han separado en clases individuales y se ha diseñado una interfaz *colección* para permitir que el conector funcione con colecciones de datos que no necesariamente sean arrays JavaScript.

El siguiente fragmento muestra el código requerido para obtener la lista de jugadores con notificaciones actualizadas automáticamente y reflejadas en la vista.

```
// This makes an AJAX call to GET /players/{args} with an 'X-Miau'  
// header set to Subscribe.  
Restangular.all('players').getList(args, {'X-Miau': 'Subscribe'})  
  .then(function(players) {  
    // This function is called when the server replies  
  
    // Assign the received data to players model  
    $scope.players = players.data;  
  
    // Adapt the players model (which is just a JS array) into a  
    // Collection.  
    var collection = new ArrayCollection($scope.players);  
  
    // Tell the CollectionDeltaProcessor to process deltas over the  
    // collection  
    deltaProcessor.collections["Player"] = collection;
```

```
// Acquire the subscription with the token received
miau.acquireSubscription(players.token);
});
```

La aplicación de incidencias

Se ha considerado conveniente desarrollar una aplicación más compleja para probar la integración del prototipo de sincronización con un framework de desarrollo web.

El framework de desarrollo web elegido es Django³⁴, basado en Python. Este framework utiliza programación síncrona y servirá como servidor principal en la arquitectura definida en el prototipo.

Requisitos

Se implementará un pequeño sistema de gestión de incidencias. El sistema debe contemplar los siguientes casos de uso:

- Los usuarios deben poder iniciar sesión con una dirección de correo electrónico.
- Todos los usuarios deben poder abrir incidencias.
- Los usuarios con permisos de administración deberán poder ver y responder las incidencias de todos los usuarios.
- Los usuarios deben poder escribir respuestas en los hilos de las incidencias.
- Tanto la página de incidencia como la lista de incidencias deben actualizarse en tiempo real.

Autenticación

Para permitir a los usuarios iniciar sesión autenticándose con una dirección de correo electrónico se ha utilizado un sistema de autenticación basado en Mozilla Persona³⁵.

Mozilla Persona es un sistema de autenticación descentralizado, basado en el protocolo *BrowserId*. Este sistema permite a los usuarios iniciar sesión con cualquier dirección de correo electrónico, siempre y cuando sean capaces de demostrar su identidad frente a su sistema de correo electrónico. En caso afirmativo, el servidor de correo electrónico emitirá un certificado temporal al cliente, el cual será enviado al servidor principal. El servidor principal comprobará la validez del certificado, asegurándose de que está firmado con la

³⁴ <https://www.djangoproject.com/>

³⁵ <https://login.persona.org/>

clave pública del servidor de correo electrónico, finalizando el proceso de autenticación y dando la sesión por iniciada.

Para implementar este sistema de autenticación en la aplicación se ha utilizado una biblioteca externa, `django-browserid`³⁶, que realiza todo el proceso de autenticación antes descrito, integrándolo con el framework Django.

En la figura 7.5 puede verse la pantalla de inicio de sesión.

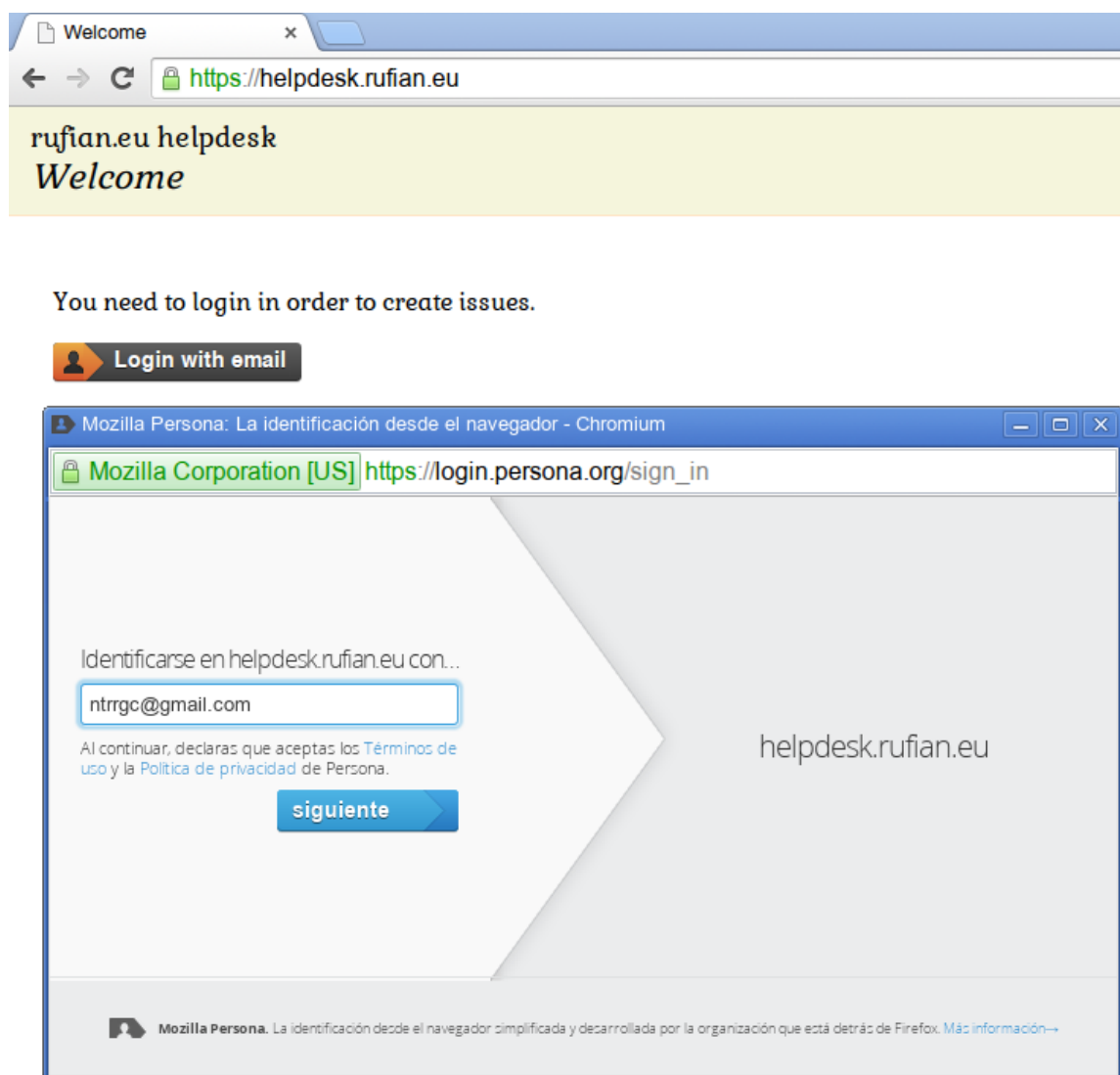


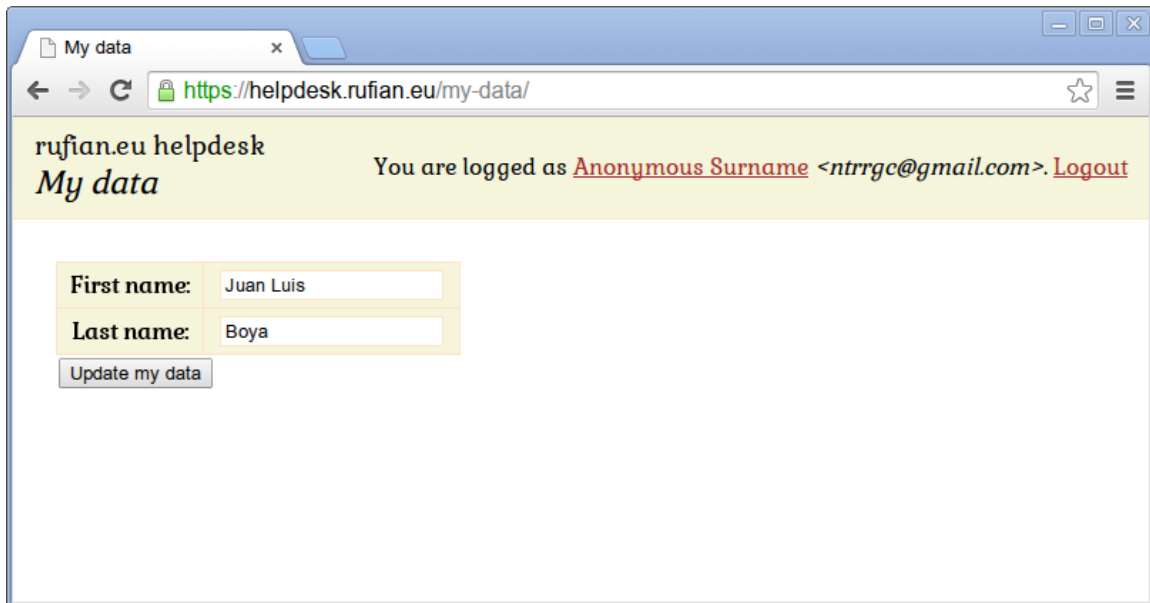
Figura 7.5: Pantalla de inicio de sesión.

Registro

La autenticación basada en *Mozilla Persona* no requiere un formulario de registro. Sin embargo, para que los usuarios puedan crear incidencias deben introducir su nombre.

³⁶ <http://django-browserid.readthedocs.org/en/latest/>

Por este motivo, cuando el sistema recibe un acceso de usuario que no está registrado en la base de datos, le da de alta y muestra un formulario para que introduzca su nombre. En la figura 7.6 se puede ver este formulario.



The screenshot shows a web browser window with the title 'My data'. The address bar displays 'https://helpdesk.rufian.eu/my-data/'. The page content includes the text 'rufian.eu helpdesk' and 'My data'. A message indicates the user is logged as 'Anonymous Surname <ntrrgc@gmail.com>' with a 'Logout' link. Below this, there is a form with two input fields: 'First name:' containing 'Juan Luis' and 'Last name:' containing 'Boya'. A button labeled 'Update my data' is positioned below the form fields.

Figura 7.6: Esta pantalla sustituye al formulario de registro en un sistema de autenticación tradicional.

Modelos de datos

Se contemplan tres modelos de datos con diferentes campos de datos:

- Usuarios.
 - Nombre.
 - Apellidos.
 - Dirección de correo electrónico.
 - Es administrador (verdadero o falso).
- Incidencias. Poseen los siguientes campos:
 - Autor.
 - Título.
 - Fecha.
 - Contenido de la incidencia.
 - Solucionada (verdadero o falso).

- Necesita atención (verdadero o falso).

El campo *necesita atención* se establece automáticamente a verdadero cuando la incidencia es nueva o la última respuesta es del usuario que la creó.

- Respuestas de incidencias. Poseen los siguientes campos:

- Autor.
- Incidencia.
- Fecha.
- Contenido.

ORM de Django

Django incluye un sistema ORM (*Object-Relational Mapping*) que permite especificar los modelos de datos de forma declarativa y realizar la mayoría consultas con una API objetual fácil de usar, sin requerir en ningún momento el uso del lenguaje SQL más que para consultas u operaciones muy específicas que no estuvieran soportadas en el ORM.

El siguiente fragmento de código muestra la clase *Issue*, que describe una incidencia.

```
from django.db import models
from django.contrib.auth import get_user_model
User = get_user_model()

class Issue(models.Model):
    initiator = models.ForeignKey(User, related_name='reported_issues')
    title = models.CharField(max_length=150)
    date = models.DateTimeField(auto_now_add=True)
    solved = models.BooleanField(default=False)
    content = models.TextField()
    needs_attention = models.BooleanField(default=True)
```

El siguiente fragmento de código muestra cómo localizar y actualizar un objeto *Issue*:

```
my_issue = Issue.objects.get(id=15) # Get issue with id = 15

print(my_issue.title) # Read a value

my_issue.solved = True
my_issue.save() # Update in database
```

Integración con el ORM de Django

Para integrar las actualizaciones en tiempo real en un sistema web se debe conseguir que se envíen notificaciones cada vez que ocurre un cambio en la base de datos.

El ORM de Django representa una interfaz centralizada para el acceso a datos. Además, este ORM permite manejar eventos tales como `pre_save`, `post_save`, `pre_delete` y `post_delete`. Estos eventos se generados antes y después de los eventos de actualización (incluyendo en este caso inserción) y borrado.

Se ha desarrollado un conector para Django que ofrece una interfaz simplificada para capturar los eventos del ORM y enviar notificaciones al servidor WebSocket.

Para hacer uso del conector el programador debe incluirlo un par de líneas en la configuración de Django para especificar la URL donde se deben enviar las notificaciones, así como la clave que se utilizará para autenticarse con el servicio.

```
MIAU_URL_BASE = 'http://localhost:5001/'
MIAU_SECRET_KEY = 'JkdXZCQgsCipFAA7GsPY'
```

Una vez hecho esto, para hacer que un modelo de datos envíe notificaciones automáticamente basta con colocarle el decorador `@subscriptable` y definir un método `jsonify()` que devuelva la representación del modelo que se enviará al servidor WebSocket codificada en lenguaje JSON.

El siguiente fragmento de código muestra el mismo modelo `Issue`, ahora con envío automático de notificaciones.

```
from miau.backend.django import subscriptable
from django.db import models
from django.contrib.auth import get_user_model
User = get_user_model()

@subscriptable
class Issue(models.Model):
    initiator = models.ForeignKey(User, related_name='reported_issues')
    title = models.CharField(max_length=150)
    date = models.DateTimeField(auto_now_add=True)
    solved = models.BooleanField(default=False)
    content = models.TextField()
    needs_attention = models.BooleanField(default=True)

    def jsonify(self):
        return {
            "initiator": self.initiator,
            "title": self.title,
            "date": self.date.isoformat(),
            "solved": self.solved,
            "content": self.content,
            "needs_attention": self.needs_attention,
        }
```

Cada vez que se ejecuta el método `save()` de un modelo decorado con `@subscriptable` el conector captura los eventos `pre_save` y `post_save` para detectar el tipo de cambio y notificarlo adecuadamente.

Django REST Framework

Una aplicación web a menudo debe exponer al menos parte de su base de datos a los usuarios. Cuando las peticiones de acceso a los datos se hacen desde el lado del cliente, una manera habitual de solucionar este problema es mediante una API REST.

REST es un conjunto de reglas para diseñar APIs basadas en el protocolo HTTP especializadas en el acceso a datos.

Dependiendo de la tecnología que se utilice en el resto del proyecto, existen algunas componentes para facilitar la construcción de APIs REST. Uno de estos componentes es Django REST Framework³⁷.

Django REST Framework permite definir funcionalidades propias de una API REST en pocas líneas a partir de propiedades configurables y un amplio catálogo de clases base. Por ejemplo, la clase mostrada en el siguiente fragmento de código permite , listar todos los elementos de tipo `Issue`.

```
from . import models, serializers, permissions
from rest_framework import mixins, viewsets, generics

class IssueSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.Issue
        fields = ('id', 'title', 'date', 'solved', 'content',
                  'needs_attention')

class ListIssuesView(generics.ListAPIView):
    serializer_class = IssueSerializer
    permission_classes = [permissions.ManipulateIssuesPermission]
    model = models.Issue
```

Adicionalmente puede verse cómo `IssueSerializer` define los campos que serán accesibles desde esta vista de la API y `permission_classes` permite especificar controles de acceso para los diferentes puntos de entrada de la API.

Se pueden consultar todos los detalles de la API de Django REST Framework en su documentación oficial³⁸.

³⁷ <http://django-rest-framework.org/>

³⁸ <http://www.django-rest-framework.org/api-guide/generic-views>

Integración con Django REST Framework

Dado que Django REST Framework hace tan fácil definir APIs para permitir y controlar el acceso a datos, se ha desarrollado una extensión para permitir autorizar suscripciones en el servidor WebSocket.

Esta extensión funciona extendiendo las clases de Django REST Framework para que busquen una cabecera X-Miau: Subscribe. Si la cabecera es encontrada, se autorizará una suscripción contra el servidor WebSocket. La clase debe especificar la clase repartidora (*dealer*) a la que se enviará la petición, y en su caso qué valor se usará como filtro.

El siguiente fragmento de código muestra cómo codificar un punto de entrada en la API que permita listar las incidencias del usuario con una sesión activa, así como obtener una suscripción para recibir actualizaciones de manera continua si el usuario lo pide:

```
from . import models, serializers, permissions
from rest_framework import mixins, viewsets, generics
import miau.backend.django.rest_framework as miau

class MyIssuesView(miau.ListSubscribeAPIView):
    serializer_class = serializers.IssueSerializer
    dealer_name = 'IssuesByUser'
    model = models.Issue

    def get_model_key(self):
        return self.request.user.email

    def get_queryset(self):
        return models.Issue.objects.filter(initiator=self.request.user)
```

Reportar una incidencia

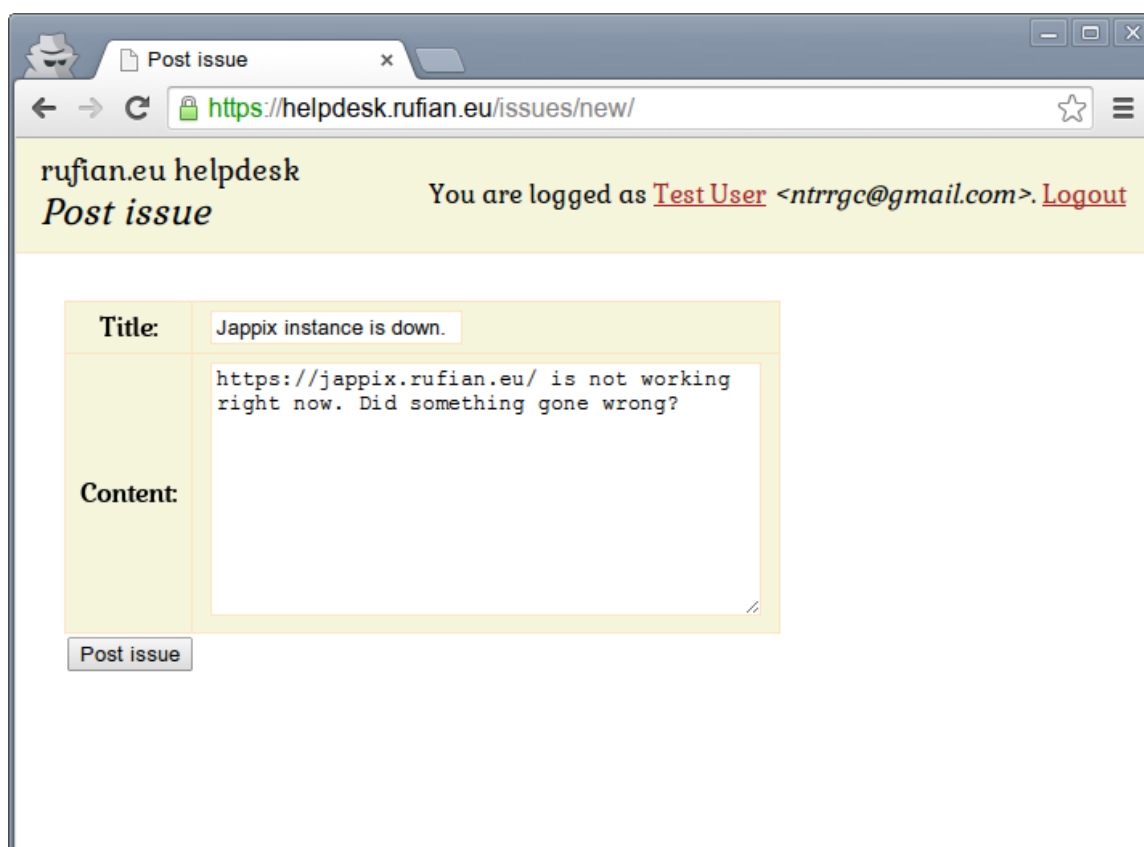
Una vez la sesión ha sido iniciada, un usuario puede crear una incidencia mediante el formulario de la figura 7.7.

Si un usuario con permisos de administrador estaba viendo su lista de incidencias en ese momento, verá aparecer una nueva entrada, como se puede ver en la figura 7.8.

Al hacer clic en la incidencia se muestra una página con el mensaje del usuario que la creó así como un formulario para responderla.

Esta página también utiliza la sincronización con bases de datos. Gracias a ello las respuestas son recibidas en tiempo real. La figura 7.9 muestra la vista de detalle de la incidencia.

Todos los cambios se reflejan en las dos vistas. Marcar una incidencia como solucionada ocasiona que de forma automática aparezca representada como no necesitando atención en la lista de incidencias, tal como puede verse en 7.10.



Post issue

https://helpdesk.rufian.eu/issues/new/

rufian.eu helpdesk

You are logged as **Test User** <ntrrgc@gmail.com>. [Logout](#)

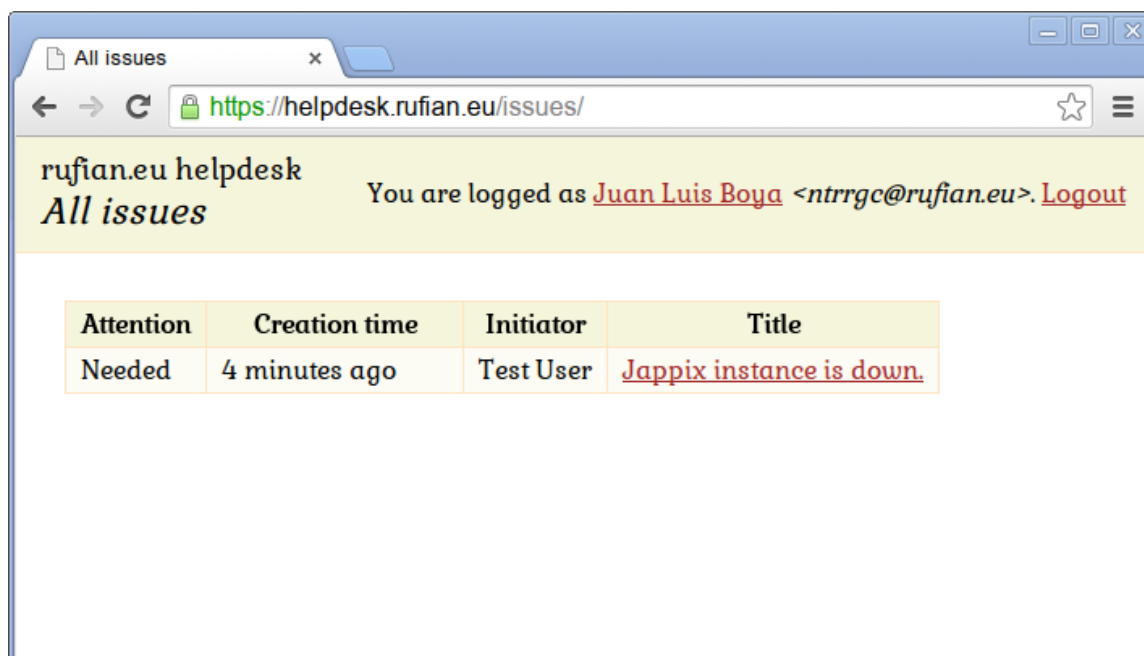
Post issue

Title: Jappix instance is down.

Content: https://jappix.rufian.eu/ is not working right now. Did something gone wrong?

Post issue

Figura 7.7: Formulario de creación de incidencias.



All issues

https://helpdesk.rufian.eu/issues/

rufian.eu helpdesk

You are logged as **Juan Luis Boya** <ntrrgc@rufian.eu>. [Logout](#)

All issues

Attention	Creation time	Initiator	Title
Needed	4 minutes ago	Test User	Jappix instance is down.

Figura 7.8: Lista de incidencias, actualizada en tiempo real.

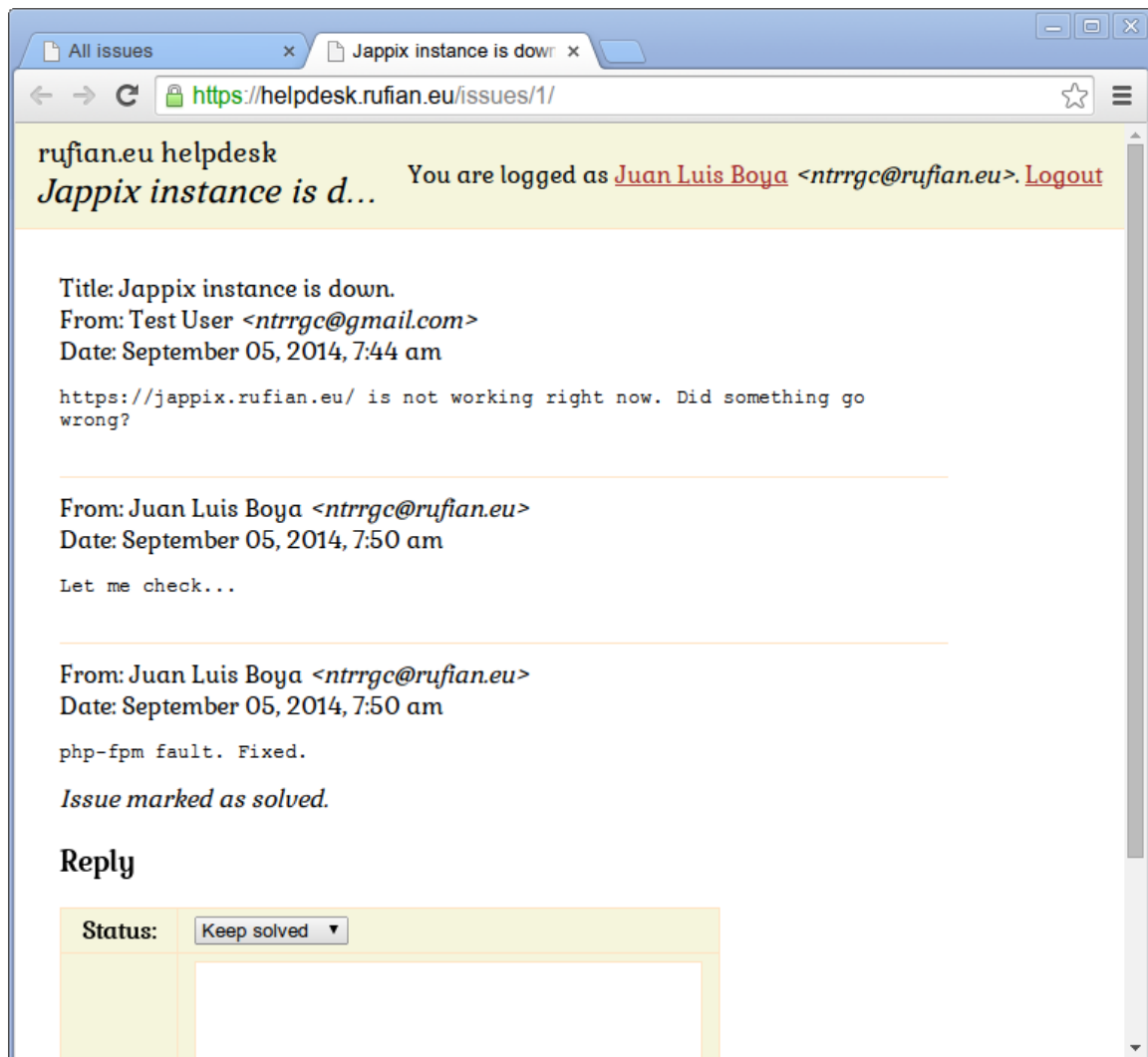


Figura 7.9: Vista de detalle de la incidencia.

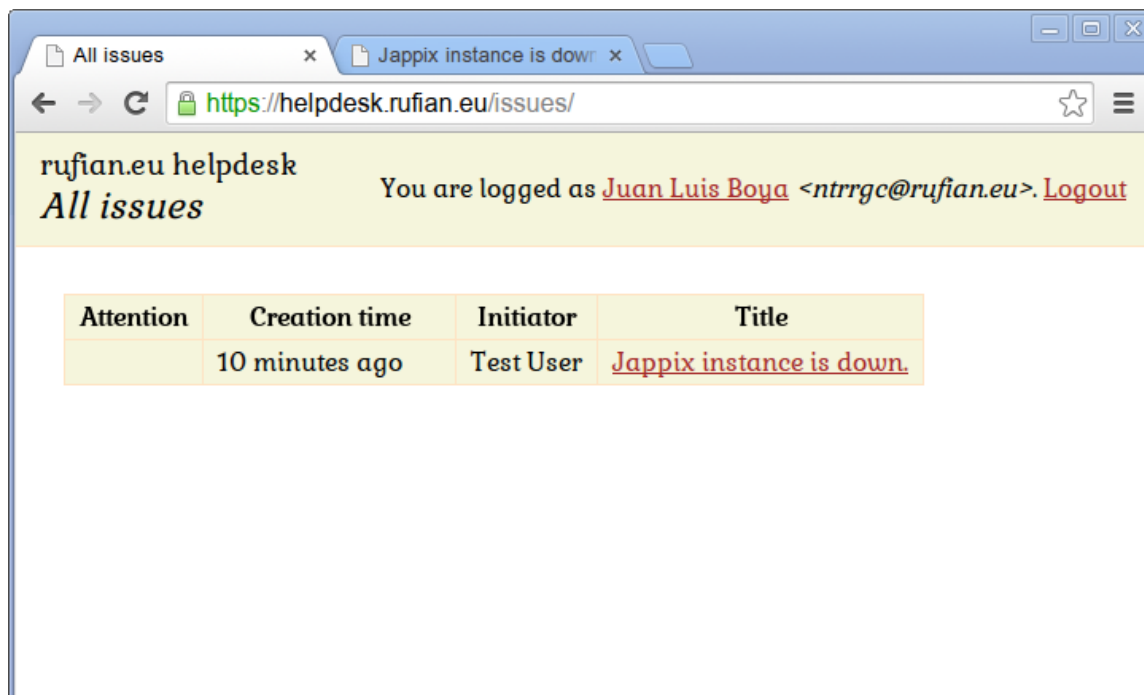


Figura 7.10: La lista de incidencias se actualiza automáticamente en todo momento.

El código de la parte del cliente es muy similar al mostrado en la demostración de jugadores. En esta aplicación también se utiliza el framework AngularJS para actualizar el modelo JavaScript con la vista del navegador. El siguiente fragmento muestra el código de la tabla de incidencias.

```
<table ng-show="issues.length > 0">
  <tr>
    <th>Attention</th>
    <th style="width: 140px">Creation time</th>
    <th>Initiator</th>
    <th>Title</th>
  </tr>
  <tr ng-repeat="issue in issues | orderBy:['-needs_attention','-date']">
    ng-class="{attention: issue.needs_attention}"
    <td>{{ formatAttention(issue.needs_attention) }}</td>
    <td><span am-time-ago="issue.date"></span></td>
    <td>{{ issue.initiator.first_name }} {{ issue.initiator.last_name }}</td>
    <td><a target="_blank" ng-href="/issues/{{issue.id}}/">{{ issue.title }}</a></td>
  </tr>
</table>
```

Servicios personalizados

La sincronización con bases de datos es sólo uno de los problemas que las aplicaciones web en tiempo real encaran. Incluso siendo quizás uno de los más importantes, sería deseable hacer que el servidor WebSocket fuera tan extensible como fuera posible,

permitiendo al desarrollador añadir nuevas funcionalidades de tiempo real o desarrollar las suyas propias específicas para su aplicación.

Desafortunadamente, el prototipo descrito en Sincronización con bases de datos está demasiado orientado a la sincronización con bases de datos y no ayudó en este aspecto.

El objetivo pretendido es el desarrollo de una plataforma para servidores WebSocket, basados en patrones, intentando que dichos servidores lleven menos esfuerzo de programación que si se escribieran desde cero. Esta plataforma se conocerá con el nombre de *Snorky*.

Eventualmente deberá ser posible reescribir la funcionalidad de sincronización con bases de datos sobre esta plataforma.

Patrones

En este apartado se describirá el uso de varios patrones que se han considerado útiles para la implementación de servidores WebSocket.

Servicios independientes

El objetivo principal de esta plataforma es extender el prototipo de sincronización de datos para ofrecer nueva funcionalidad.

Puesto que cuando el prototipo fue escrito, la única funcionalidad que tenía el servidor era la sincronización con bases de datos, no se creó ningún espacio de nombres para agrupar las peticiones de los clientes según la funcionalidad que solicitaran.

En Snorky las diferentes funcionalidades se separarán en *servicios*. Un servidor Snorky tendrá uno o más *servicios* los cuales son clases que atienden peticiones de los clientes y ocasionalmente les envían mensajes.

En una fase futura se creará un servicio de sincronización de datos. Otros servicios interesantes pueden ser chat, Pub Sub, bloqueos – ej. para evitar que dos usuarios trabajen en el mismo objeto al mismo tiempo y uno pise los cambios de otros... La extensibilidad es la clave.

Llamadas RPC

La mayoría de la interacción con el servidor WebSocket en el prototipo era de la forma Petición-Respuesta. Hacer un patrón alrededor de esto permitiría simplificar el código de forma considerable.

Una interfaz RPC es aquella que permite a su usuario invocar funciones o métodos en un servidor remoto, proporcionando en cada caso un nombre de método y una serie de parámetros. La característica más importante de una interfaz RPC es que debe

ocultar los detalles de cómo está codificada esa información, enviada a través del canal y posteriormente la respuesta es identificada y decodificada.

Esto es beneficioso tanto para el servidor como para el cliente:

- El servidor contará con un lenguaje mucho más cómodo para definir los métodos RPC, no requiriendo al programador gastar mucho tiempo procesando los datos del usuario.
- El cliente podrá comunicarse con el servidor a través de una interfaz sencilla. Usando un conector JavaScript similar al realizado en el prototipo anterior, el programador podrá escribir llamadas remotas con poco más código que si fueran llamadas locales.

No REST

REST es un acrónimo de *REpresentation State Transfer*. Es un conjunto de reglas sobre cómo diseñar servicios web interoperables.

Particularmente, REST establece que las URLs deben ser asociadas a *recursos* (entidades de datos del sistema) mientras que los métodos HTTP utilizados deben representar la operación que se quiere hacer sobre el recurso seleccionado.

REST funciona muy bien cuando las operaciones se centran alrededor del almacenamiento de datos en bases de datos. Por ejemplo, crear un objeto Task podría hacerse enviando los campos del nuevo objeto en una petición POST dirigida al recurso `/tasks`. Eliminar el primer objeto Task podría hacerse enviando una petición DELETE a `/tasks/1`.

El conjunto de métodos disponibles (como POST y DELETE) está limitado a aquellos métodos definidos por el protocolo HTTP. Aunque REST funciona muy bien para los casos de uso Crear-Consultar-Actualizar-Borrar, añadir funcionalidad fuera de esos métodos habitualmente requiere utilizar URLs poco consistentes (poco *RESTful*).

Esto representa un problema para el prototipo anterior. Hay dos posibles peticiones en el backend, `authorize_subscription` y `notify_deltas`. No está muy claro cómo deberían encajar en una API REST. POST `/subscription` podría tener sentido para crear una nueva suscripción, pero realmente no es muy acorde a la filosofía de REST ya que no devuelve ningún objeto que se pueda consultar después con la misma interfaz. Si bien podrían haberse añadido métodos a la interfaz para implementar dicha funcionalidad, no tendría sentido hacerlo sin que fuera un requisito expreso, sólo por ser *RESTful*.

Se hizo evidente que una interfaz RPC tiene mucho más sentido para operaciones de backend.

Sería deseable utilizar la misma codificación tanto para peticiones de frontend como de backend, puesto que así se requeriría menos código y el programador sólo tendría que aprender a escribir servicios RPC una vez, en vez de aprender a escribirlos primero para la parte frontend y después para la parte backend.

Como beneficio adicional, aislar el protocolo RPC hace la comunicación en el backend independiente de HTTP, permitiendo desarrollar interfaces de backend sobre nuevos protocolos con facilidad. Esto podría ser útil para sistemas basados en colas de mensajes como RabbitMQ³⁹ o ZeroMQ⁴⁰, lo cual podría convertirse en un punto esencial para escalar Snorky a varios servidores en versiones futuras.

Promesas

Las peticiones a servidores remotos acostumbran a tardar un tiempo significativo en ser respondidas. Esto es especialmente cierto en navegadores, ya que algunos usuarios pueden tener conexiones considerablemente lentas.

Por este motivo, el diseño de interfaces de usuario debe ser asíncrono: la interfaz no debe quedarse bloqueada porque una petición está pendiente de ser resuelta. Durante este tiempo la interfaz debe mostrar un reloj o incluso permitir al usuario seguir trabajando normalmente, dependiendo de la naturaleza de la petición y de la aplicación.

La manera tradicional de conseguir esto en JavaScript es con *callbacks*. Al hacer una petición se especifica una función como argumento – el callback, que será ejecutada cuando la respuesta sea recibida. Si la petición devuelve algún valor, esta función será quien lo reciba.

Los callbacks funcionan, pero pueden ser una amenaza a la mantenibilidad en interfaces de usuario. A menudo los callbacks desencadenan nuevas peticiones, que en respuesta ejecutan más callbacks, y así sucesivamente.

Un efecto típico de los callbacks es un antipatrón conocido como *Pirámide de la Perdición* (*Pyramid of Doom*), donde el código es difícil de seguir ya que el orden en el que se ejecuta difiere del orden en el que se escribe, y cada tarea asíncrona adicional requiere un nuevo nivel de sangrado.

El fragmento de código siguiente muestra un ejemplo de una *Pirámide de la Perdición*:

```
asyncThing1(args1, function() {
  asyncThing2(args2, function() {
    asyncThing3(args3, function() {
      console.log("Pyramid of Doom!");
    });
    // not asynchronous things that happen between thing 3 and 4
  });
  // not asynchronous things that happen between thing 2 and 3
});
// not asynchronous things that happen between thing 1 and 2
```

Hay más problemas desagradables que ocurren como consecuencia de usar callbacks. Una necesidad recurrente en interfaces pero difícil de realizar con callbacks es la de ejecutar una tarea cuando dos tareas (dependencias) se han completado, pero no se sabe

³⁹ <http://www.rabbitmq.com/>

⁴⁰ <http://zeromq.org/>

a priori cuál de ellas terminará antes. Este es un caso de uso común cuando se mezclan peticiones AJAX con animaciones, como puede verse en la figura 7.11.

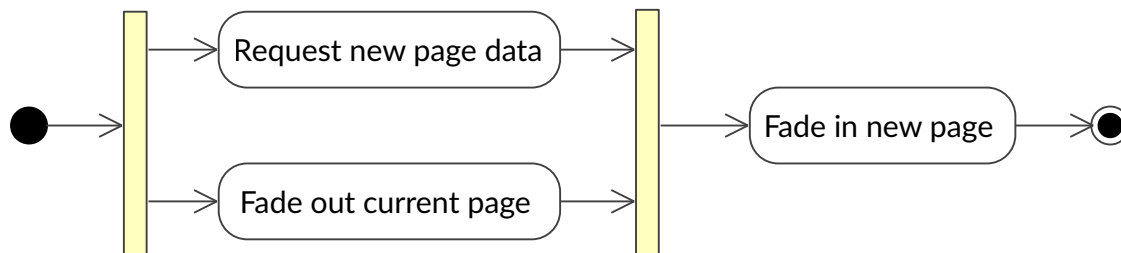


Figura 7.11: La nueva página no debe empezar a mostrarse progresivamente hasta que (1) la petición AJAX haya sido respondida, y por lo tanto haya datos para mostrar en la nueva página y (2) la animación de desvanecer de la página vieja haya terminado.

Finalmente, los errores son muy difíciles de tratar, puesto que hay muchos puntos de fallo, y cada uno de ellos requiere tratamiento individual. El siguiente código muestra cómo se podría hacer un tratamiento de errores con reintentos en una *Pirámide de la Perdición*.

```
function retry1() {
  asyncThing1(args1, function retry2() {
    asyncThing2(args2, function retry3() {
      asyncThing3(args3, function() {
        console.log("Pyramid of Doom!");
      }, function() {
        // Error caught in asyncThing3
        retry3();
      });
      // not asynchronous things that happen between thing 3 and 4
    }, function() {
      // Error caught in asyncThing2
      retry2();
    });
    // not asynchronous things that happen between thing 2 and 3
  }, function() {
    // Error caught in asyncThing1
    retry1();
  });
  // not asynchronous things that happen between thing 1 and 2
}
```

Hay varias técnicas para evitar la *Pirámide de la Perdición* y escribir código asíncrono más legible y mantenible. Una de esas soluciones consiste en el uso de *promesas*. Un objeto promesa o *Promise* contiene callbacks para las condiciones de éxito y de fallo.

Los callbacks para las condiciones de éxito y fallo son establecidos con los métodos `then()` y `catch()` respectivamente.

La característica principal de las promesas es que pueden ser encadenadas. Cada llamada a `then()` o `catch()` devuelve una nueva promesa que será resuelta con el valor de

retorno del callback proporcionado. Una resolución exitosa puede tornarse en resolución con fallo y viceversa.

Como demostración, el siguiente fragmento de código muestra la pirámide anterior reescrita con promesas.

```
asyncThing1
.then(function() {
  return asyncThing2();
})
.then(function() {
  return asyncThing3();
})
.then(function() {
  console.log("Done!");
})
.catch(function(error) {
  // Error caught in any asynchronous function.
  return error.retry();
});
```

Las bibliotecas de promesas también soportan agregación de promesas, solucionando de manera elegante el problema de esperar por varios eventos que pueden ocurrir en cualquier orden.

El patrón promesa ha recibido mucha atención últimamente, surgiendo un gran número de implementaciones. Para disminuir las diferencias entre ellas y hacerlas interoperables, una interfaz de promesa estándar ha sido publicada, conocida como Promises/A+⁴¹.

La próxima versión del estándar de JavaScript, ECMAScript 6, contendrá una implementación de este estándar. Hasta entonces este patrón puede ser explotado a través de diferentes bibliotecas.

Siguiendo con esta tendencia, es deseable que el conector de Snorky utilice promesas en aquellos métodos que realicen llamadas remotas.

Requisitos

Varias cosas tendrán que ser establecidas durante el desarrollo de esta fase.

- Una capa de servicios que permita alojar y utilizar varios servicios.
- Una interfaz RPC que los clientes puedan utilizar para comunicarse con los servicios.
- Un lenguaje para codificar las peticiones y respuestas RPC en la red.

⁴¹ <http://promisesaplus.com/>

Análisis

La fase de análisis consistió en identificar las clases clave en las que se basaría el diseño. La figura 7.12 muestra estas clases.

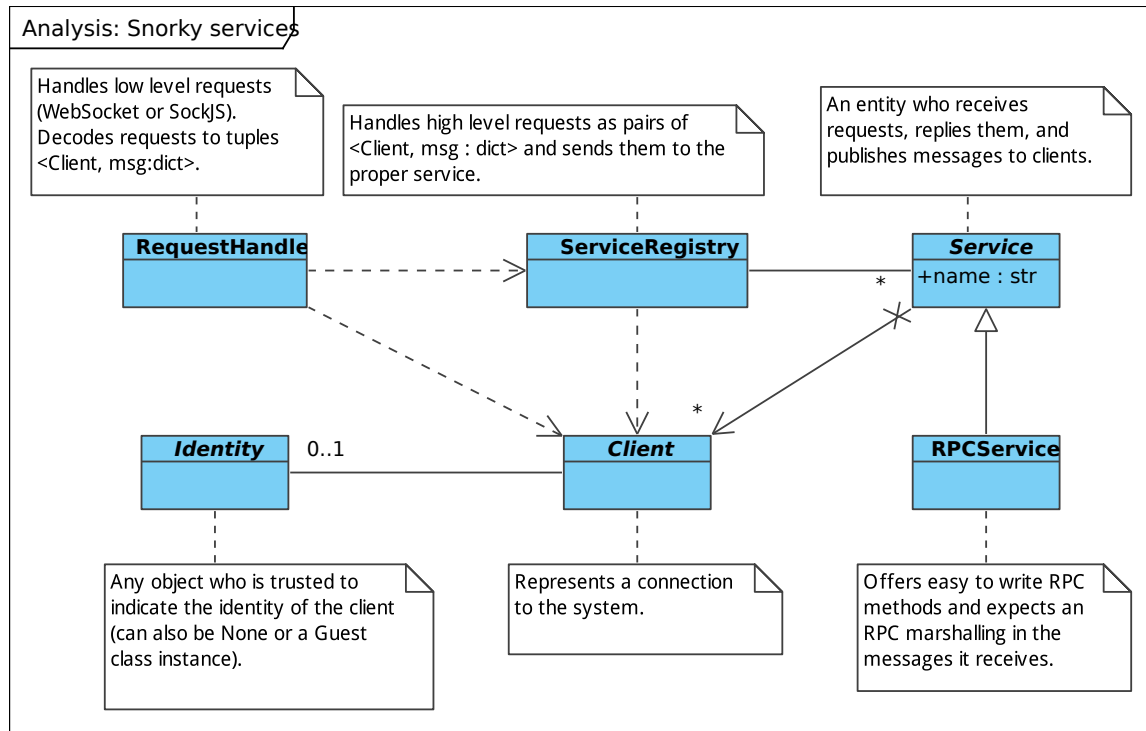


Figura 7.12: Las clases de análisis identificadas para la plataforma Snorky.

- **Service** recibirá y enviará mensaje, bien en formato Petición-Respuesta o en cualquier orden.
- **RPCService** especializará **Service** ofreciendo métodos de ayuda para escribir servicios RPC.
- **Client** abstraerá los detalles de la conexión de un cliente, sin importar el protocolo a través del que se ha conectado.
- Un objeto **Client** puede o no estar asociado a un objeto **Identity** conteniendo información de identificación acerca del usuario final. Esta información de identidad puede ser establecida por un servicio de autenticación después de que el cliente la haya demostrado de alguna manera, por ejemplo enviando al servidor WebSocket un testigo de sesión que puede comprobar con el servidor principal.
- **ServiceRegistry** agrega un conjunto de servicios, identificados por nombre, y se encarga de que reciban los mensajes de los clientes.
- Las clases **RequestHandler** atenderían protocolos específicos como **WebSocket** o **SockJS**, decodificando los mensajes que reciben y enviándolos a **ServiceRegistry**. Con cada mensaje deben enviar también una instancia de **Client**, que debe proveer

un método para enviar un mensaje de vuelta al cliente.

Diseño del protocolo

El protocolo a utilizar en la comunicación entre el servidor WebSocket y el resto se basará en el lenguaje JSON. JSON es un protocolo adecuado por las siguientes razones:

- Los tipos de dato de JSON se traducen fácilmente en tipos de dato comunes de la mayoría de lenguajes de programación usados para desarrollo web. Esos tipos son listas, diccionarios, números, cadenas y el valor nulo. En consecuencia, trabajar con JSON es a menudo más fácil que con XML, que requiere utilizar nuevas construcciones del lenguaje para acceder a los datos (ej. clases *nodo* o el lenguaje *XPath*).

Esto es especialmente cierto para Python (lado del servidor) y JavaScript (lado del cliente), los cuales soportan todos los tipos necesarios sin bibliotecas adicionales. Muchos otros lenguajes, como Java, Ruby, C#, PHP o ASP.Net también soportan todos estos tipos sin trabajo adicional.

- JSON es un formato muy popular para representación de modelos de datos, siendo utilizado como formato de respuesta en la mayoría de las interfaces REST desarrolladas actualmente. Este es un punto muy importante puesto que es deseable usar el mismo lenguaje en el futuro servicio de sincronización con bases de datos.
- JSON soporta composición: un diccionario o lista puede contener otros diccionarios o listas. Esto hace fácil desarrollar protocolos multicapa basados en JSON.
- JSON ya se utilizó con éxito en el prototipo de sincronización con bases de datos.

Como no podría ser de otra forma dada la arquitectura de Internet, el protocolo de Snorky dependerá de varias capas de protocolos, cada cual proveyendo servicios a la capa superior.

La siguiente lista explica las capas necesarias en una conexión WebSocket de abajo arriba; a más baja la capa, más cerca está del hardware de red.

- El protocolo IP, sea versión 4 o versión 6 (*IPv6*). Este protocolo permite mover mensajes arbitrarios de tamaño limitado (datagramas) a través de Internet.

Cada datagrama incluye sendas direcciones de origen y destino. Cada vez que una máquina de Internet recibe un datagrama IP con una dirección de destino perteneciente a una red externa, redirige el mensaje a través de la red externa con dirección a la máquina que forme del camino más corto hacia el destino solicitado..

- El protocolo TCP se asienta encima del protocolo IP para solucionar algunas limitaciones del mismo, ofreciendo un transporte fiable.
 - TCP permite que cada máquina en Internet ofrezca diferentes servicios,

identificados por *números de puerto*. Esto hace posible que una sola máquina ofrezca al mismo tiempo, por ejemplo, servicios web y de correo al mismo tiempo.

Se podría decir que mientras que IP posibilita la comunicación entre equipos, TCP la posibilita entre procesos.

- TCP utiliza técnicas de fragmentación para poder intercambiar mensajes de tamaño mayor que un datagrama.
- TCP incorpora un sistema de confirmación de mensajes, encolado e integridad. Éstas capacidades son importantes para brindar un servicio de transporte fiable.
 - Permite la comunicación entre equipos de diferente velocidad sin pérdida de mensajes. Si la otra parte no ha confirmado la recepción de los últimos n bytes, TCP esperará antes de enviar más mensajes.
 - Cualquiera de los participantes puede descubrir que faltan mensajes y pedir su reenvío. Los mensajes duplicados son ignorados de forma automática.
 - Alteraciones no intencionadas en el contenido de los datagramas serán detectadas automáticamente, solicitando el reenvío de forma automática y transparente para las aplicaciones.
- El protocolo TLS provee una capa de seguridad ofreciendo cifrado y autenticación. Esta capa es opcional. Es utilizada cuando el cliente conecta a una URL `https://` o `wss://` en vez de `http://` o `ws://`.
- El protocolo WebSocket se asienta por encima de TCP o TLS. Provee negociación en HTTP, lo que le permite ser utilizado desde los navegadores siguiendo las políticas de seguridad desplegadas en la web, tales como la política de mismo origen⁴².

Adicionalmente WebSocket especifica un formato para marcar la longitud de los mensajes (*framing*), lo que permite intercambiar mensajes binarios o de texto de cualquier longitud sin que las aplicaciones tengan que preocuparse de cómo delimitarlos.

Encima de esta pila de protocolos, Snorky ofrecerá las siguientes capas:

- La capa JSON, que será responsable de decodificar la información recibida a través del protocolo WebSocket, traduciendo los mensajes en estructuras tales como diccionarios, listas, números, etc.

Esta capa también permitirá enviar mensajes en esas estructuras, encargándose ella de la serialización a JSON.

Si un mensaje no puede ser decodificado como JSON, el fallo será procesado por esta capa.

⁴² https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

- La capa de servicios, que será responsable de entregar cada mensaje al servicio que le corresponda.

Si un mensaje está dirigido a un servicio desconocido, esta capa será la responsable de señalar el problema.

- La capa RPC. Esta capa estará presente en los servicios RPC y se encargará de decodificar las peticiones, convirtiéndolas en llamadas a métodos.

También es función de esta capa asegurar que cada petición y respuesta están correctamente identificadas para que el cliente pueda asociar cada respuesta con la petición correcta, incluso habiendo varias simultáneas.

Si el método pedido no está existe o los parámetros son inválidos, no siendo posible efectuar la llamada, esta capa notificará el fallo.

Diseño de clases del servidor

La fase de diseño tiene por objetivo evolucionar las clases de análisis antes vistas hacia versiones más concretas, listas para la implementación.

La figura 7.13 muestra cómo los manejadores de peticiones (*request handlers*) abstraen los detalles de sus conexiones en clases conformes a la interfaz *Client* y delegan a una clase independiente la entrega de mensajes.

Anteriormente en el prototipo de sincronización con bases de datos, las clases manejadoras de peticiones solucionaban este problema con herencia: La clase *FrontendHandler* tenía un método *on_message()* que decodificaba la petición e invocaba la operación adecuada, junto con otros métodos como *on_close()* para manejar las desconexiones. Aunque esta solución ha funcionado tanto para el transporte *WebSocket* como para *SockJS*, se ha considerado problemático en este nuevo diseño, ya que futuras clases manejadoras de peticiones podrían utilizar ya esos nombres. En consecuencia, se ha considerado que usar delegación era una solución mejor para esta nueva arquitectura.

La figura 7.14 muestra las clases que implementarán las capas de servicios y RPC. No hay mucho detalle en el diseño de clases acerca de cómo será la implementación de los servicios RPC, ya que la implementación de esta funcionalidad dependerá mucho de la funcionalidad del lenguaje de programación, no siendo fácil de representar en UML.

Signatura de llamadas RPC

Las interfaces RPC tienen el propósito de que las invocaciones remotas parezcan invocaciones locales hechas en el lenguaje de programación del usuario. Sin embargo, diferentes lenguajes de programación tienen diferentes formas de llamar a sus funciones.

Un aspecto donde se diferencian algunos lenguajes es en la especificación de parámetros de funciones:

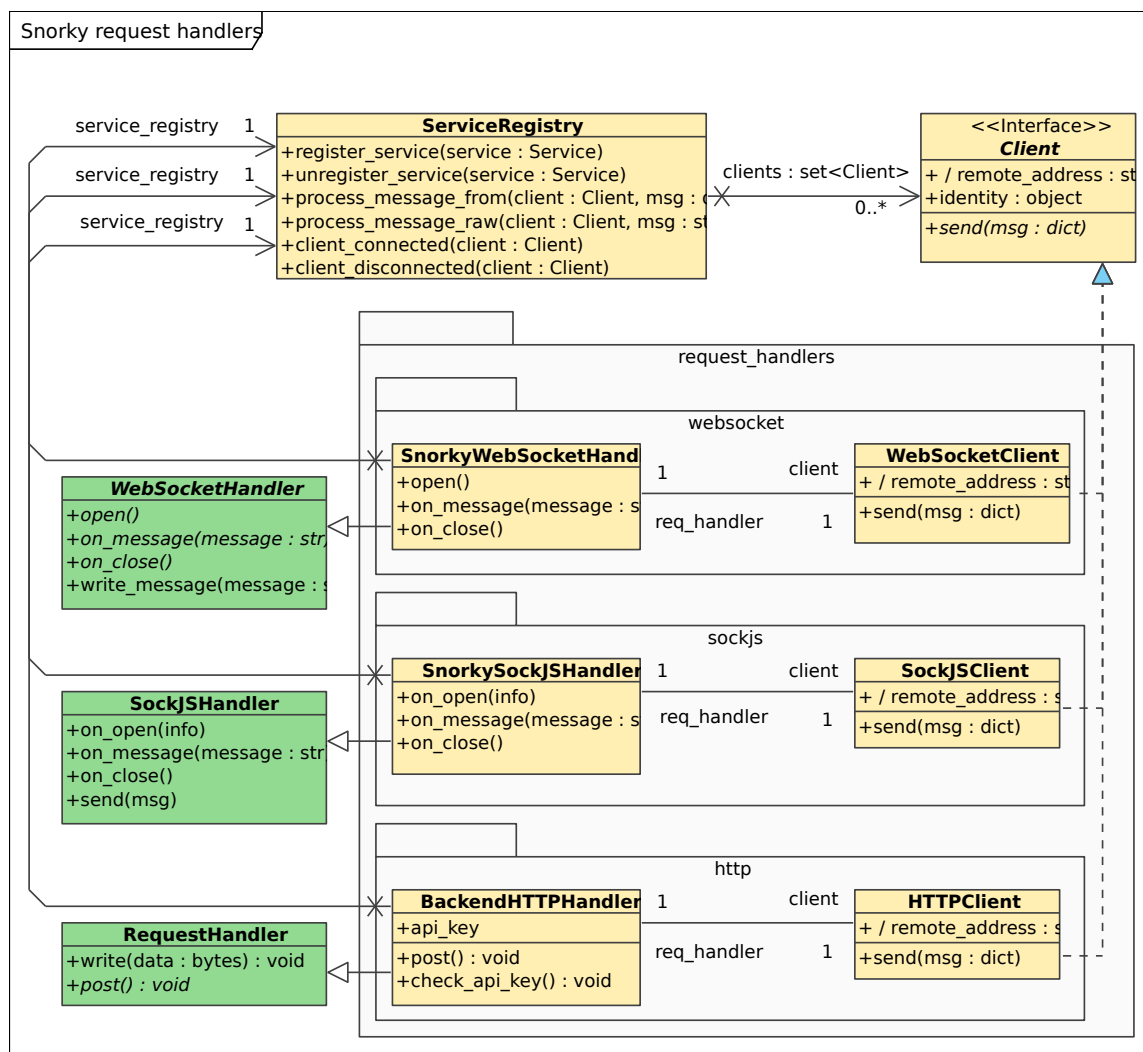


Figura 7.13: Las clases manejadoras de peticiones reciben mensajes de distintos protocolos y se los entregan a **ServiceRegistry**, quien decodifica el JSON y entrega los mensajes decodificados a los servicios correspondientes. Cada protocolo tiene su propia clase cliente.

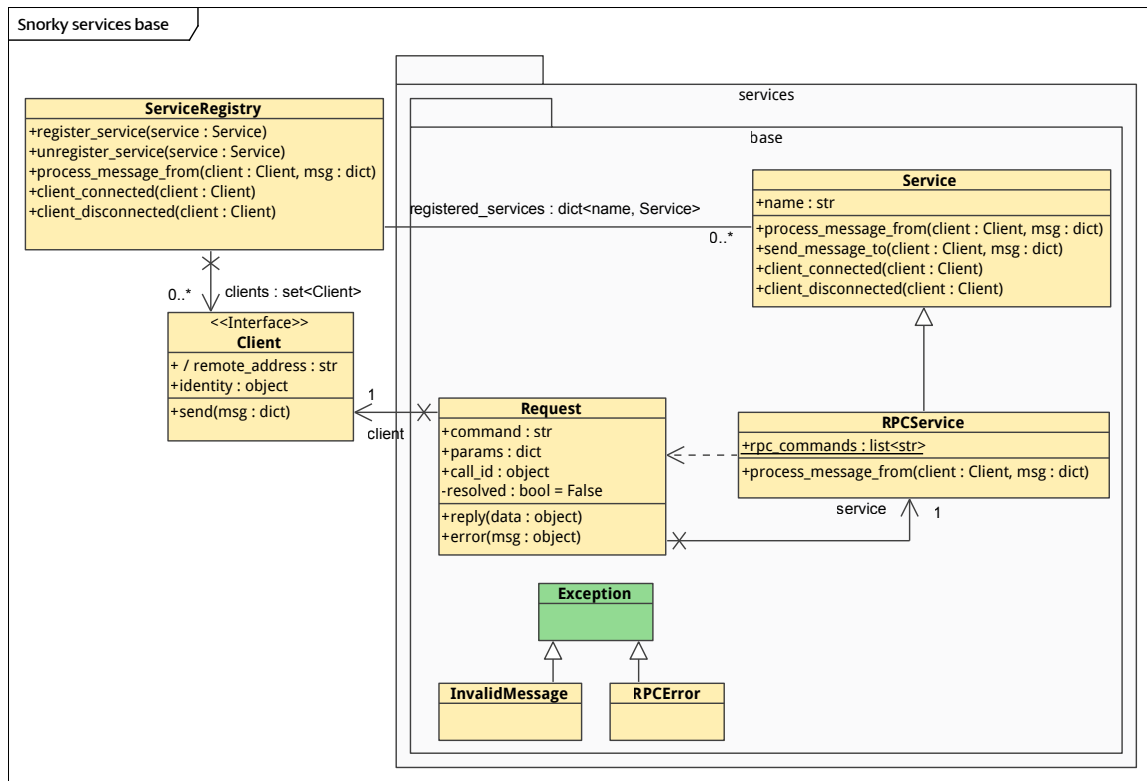


Figura 7.14: Los servicios proveen una interfaz simple para recibir y enviar mensajes de clientes. Los servicios RPC disponen de una clase *Request* que envuelve cada petición.

- Algunos lenguajes, como C, C++ o Java, utilizan argumentos posicionales. Las funciones esperan recibir sus parámetros en un orden muy específico y la posición de un parámetro determina su significado. Por ejemplo, *el primer parámetro debe ser la ruta a abrir, el segundo debe ser el modo*.
- Otra posibilidad es utilizar argumentos nombrados. Cada argumento incluiría el nombre del parámetro, seguido de su valor. Esto hace muy fácil añadir nuevos parámetros sin romper la interfaz. El orden de los parámetros no suele ser importante.

Pocos lenguajes de programación implementan este tipo de especificación de parámetros – Objective C es uno de ellos. Sin embargo, esta forma de especificar parámetros sí es muy común en las interfaces de servicios web, debido a que se ajusta bien a las funcionalidades del protocolo HTTP y permite extensión manteniendo compatibilidad hacia atrás.

- Otros lenguajes, como Python, C#, Visual Basic o Ada, permiten mezclar ambos estilos en la misma llamada. Los argumentos no nombrados se ajustarán a los primeros parámetros de la función, mientras que los argumentos nombrados se repartirán a los parámetros cuyo nombre coincida con el especificado por el usuario, sin que importe el orden.

El tercer estilo es el más versátil. Sin embargo, este estilo de llamadas resultaría poco natural en JavaScript, que es el lenguaje donde más llamadas RPC se harían.

Aunque JavaScript no soporta argumentos nombrados, algunas bibliotecas usan un enfoque parecido: Declaran un único argumento, que se espera que sea un diccionario de parámetros.

De ésta manera se obtiene el beneficio de hacer parámetros opcionales de forma arbitraria sin especificar orden y es posible añadir nuevos parámetros en cualquier momento.

La desventaja es que este estilo requiere especificar el nombre de un argumento incluso cuando sólo hay un parámetro posible.

A pesar de ello, se ha decidido utilizar el estilo de argumentos nombrados en Snorky. Puede que sea un poco más verboso a veces, por ejemplo `acquireSubscription({ token: token })` en vez de `acquireSubscription(token)`, pero ésto no es un grave problema comparado con la ventaja de poder añadir parámetros nuevos a la interfaz en cualquier momento y la probabilidad reducida de utilizar mal una API por colocar los argumentos de forma desordenada.

Implementación del servidor

La fase de implementación de esta fase consiste en implementar las clases antes mostradas así como los añadidos del lenguaje para las llamadas RPC.

Para declarar un método RPC se utiliza el decorador `@rpc_command`. Este decorador añade el método al conjunto `rpc_commands` de la clase `RPCService`. Cuando se recibe una petición, el método manejador del evento de mensaje recibido buscará el nombre del método solicitado en `rpc_commands`. Si no se encuentra, la petición fallará con el mensaje de error `Unknown command`.

Los métodos RPC se invocan con los parámetros especificados por las llamadas RPC junto con un parámetro adicional, `req` que contendrá el objeto `Request` con métodos para resolver la petición satisfactoriamente (`reply()`) o señalar un error (`error()`).

Para hacer la programación de métodos RPC más cómoda, por defecto se hará que el valor de retorno de los mismos sea enviado al cliente como respuesta. Si en vez de retornar satisfactoriamente se genera una excepción del tipo especial `RPCError`, se señalará al usuario el error, que podrá especificarse como argumento. El resto de tipos de excepciones serán tratadas como errores internos.

Como respuesta a un error interno se enviará un mensaje de error con el mensaje `Internal error` al cliente y se registrará de una traza de depuración.

El siguiente ejemplo de la documentación muestra cómo hacer un servicio RPC que calcula algunas operaciones matemáticas sencillas.

```
import math
from snorky.services.base import RPCService, rpc_command

class CalculatorService(RPCService):
    @rpc_command
    def sum(self, req, number1, number2):
```

```

        return number1 + number2

@rpc_command
def log(self, req, number, base=2.718):
    return math.log(number, base)

```

Un decorador adicional, `@rpc_asynchronous`, permite a los métodos que lo portan desactivar el envío de respuesta automática, permitiendo así diferir la resolución de una petición. El siguiente código muestra un ejemplo de un método RPC asíncrono que permite al cliente autenticarse utilizando un servidor HTTP externo como autoridad.

```

from snorky.services.base import RPCService, rpc_command, rpc_asynchronous
from tornado.httpclient import AsyncHTTPClient, HTTPRequest
from functools import partial

class BasicAuthService(RPCService):
    @rpc_asynchronous
    @rpc_command
    def authenticate(self, req, username, password):
        http_client = AsyncHTTPClient()
        request = HTTPRequest("http://localhost:5481/api/v2.0/account",
                              auth_username=username, auth_password=password)
        callback = partial(self.auth_response_received, req, username)
        http_client.fetch(request, callback)

    def auth_response_received(self, req, username, auth_response):
        if auth_response.code == 200:
            # Set identity
            req.client.identity = username
            # Successful authentication
            req.reply(None)
        elif auth_response.code == 403:
            req.error("Invalid credentials")
        else:
            req.error("Error in authentication server")

```

Conector JavaScript

Usar la nueva interfaz RPC con JavaScript sin añadidos no sería demasiado productivo. En lugar de eso, se seguirá el ejemplo del prototipo anterior y se construirá un conector JavaScript para utilizar las nuevas interfaces de forma más sencilla.

El ejecutor de tareas

El nuevo conector deberá ser más modular que el anterior, puesto que necesitará soportar nuevos servicios, incluso algunos creados por los usuarios.

Para conseguir esta modularidad se ha decidido separar el código en varios archivos.

Puesto que no es deseable incluir varios archivos para utilizar la biblioteca, la decisión de separar en varios archivos requiere introducir un ejecutor de tareas.

Un ejecutor de tareas es una aplicación que ejecuta determinadas órdenes relacionadas con el desarrollo de proyectos. En este caso, la primera tarea será concatenar los ficheros en un único archivo. Otras tareas típicas son ejecutar servidores de desarrollo o ejecutar la batería de pruebas.

El ejecutor de tareas escogido es Grunt⁴³, principalmente por experiencia del autor.

Pruebas unitarias

Las pruebas unitarias son de gran importancia para el nuevo conector. Hay varios frameworks para escribirlas. El escogido para este proyecto ha sido Jasmine 2.0.

Grunt ha demostrado ser muy útil para ejecutar las pruebas unitarias, especialmente porque tiene una opción para ejecutarlas cada vez que se modifica un archivo de código fuente. Esto permite trabajar de forma ágil sin cambios de ventana ni esperas. La figura 7.15 muestra la salida resultante de ejecutar los tests unitarios con el monitor de Grunt.

El siguiente fragmento de código muestra un caso de prueba con la sintaxis de Jasmine 2.0.

```
describe("ArrayCollection", function() {
    var array, collection;

    beforeEach(function() {
        array = ["red", "blue"];
        collection = new DataSync.ArrayCollection(array);
    });

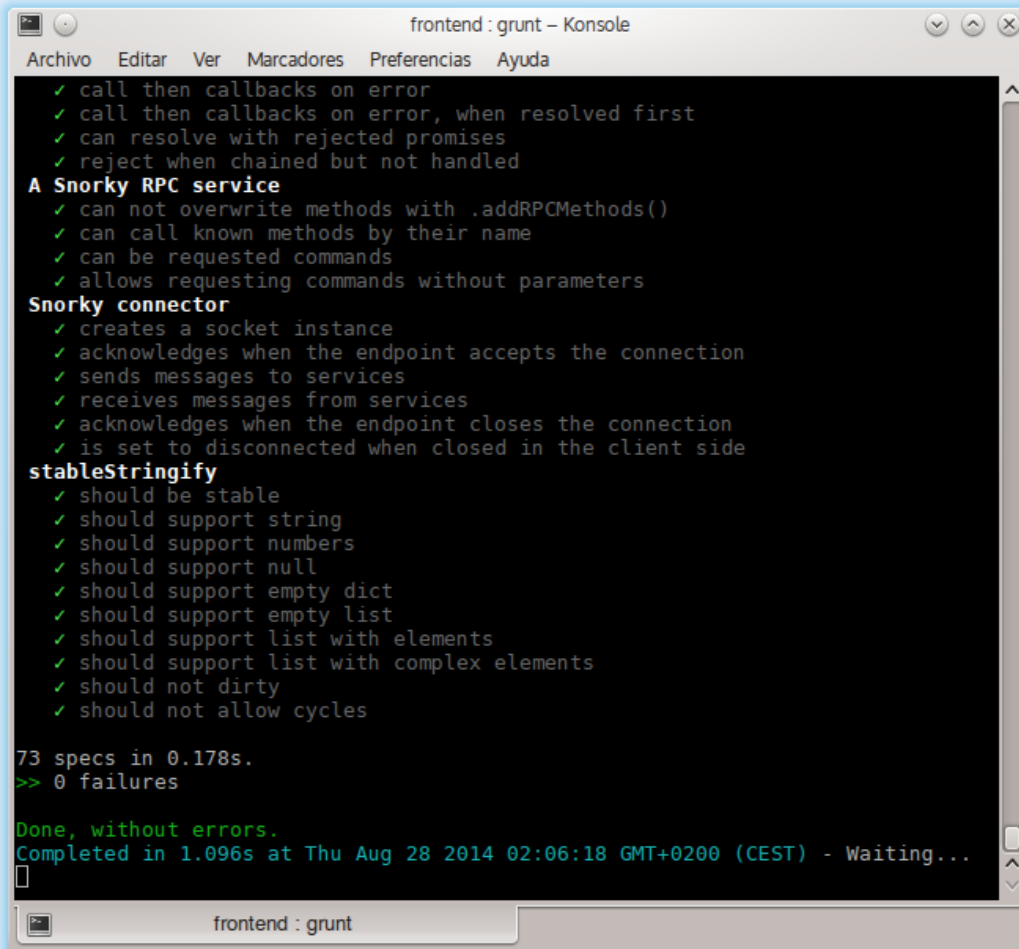
    it("can insert elements", function() {
        collection.insert("orange");
        expect(array).toEqual(["red", "blue", "orange"]);
    });
});
```

La biblioteca de clases

Aunque pueda parecer extraño, a pesar de que JavaScript hace un uso extensivo del concepto de objeto, carece de una construcción del lenguaje para declarar clases. Pueden crearse clases, pero requiere más código del que sería necesario en otros lenguajes y los mecanismos que hacen que funcionen son algo complejos.

El siguiente fragmento de código muestra una simple jerarquía de clases en JavaScript sin añadidos.

⁴³ <http://gruntjs.com/>



```
frontend : grunt - Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda

✓ call then callbacks on error
✓ call then callbacks on error, when resolved first
✓ can resolve with rejected promises
✓ reject when chained but not handled
A Snorky RPC service
✓ can not overwrite methods with .addRPCMethods()
✓ can call known methods by their name
✓ can be requested commands
✓ allows requesting commands without parameters
Snorky connector
✓ creates a socket instance
✓ acknowledges when the endpoint accepts the connection
✓ sends messages to services
✓ receives messages from services
✓ acknowledges when the endpoint closes the connection
✓ is set to disconnected when closed in the client side
stableStringify
✓ should be stable
✓ should support string
✓ should support numbers
✓ should support null
✓ should support empty dict
✓ should support empty list
✓ should support list with elements
✓ should support list with complex elements
✓ should not dirty
✓ should not allow cycles

73 specs in 0.178s.
>> 0 failures

Done, without errors.
Completed in 1.096s at Thu Aug 28 2014 02:06:18 GMT+0200 (CEST) - Waiting...

frontend : grunt
```

Figura 7.15: La batería de tests se ejecuta cada vez que se produce un cambio en el código fuente.

```
function Vehicle(owner) {  
  // Constructor  
  this.owner = owner;  
}  
  
// Method declaration  
Vehicle.prototype.move = function() {  
  console.log("I'm moving!");  
};  
  
function Car(owner, color) {  
  // Superclass constructor  
  Vehicle.call(this, owner);  
  
  this.color = color;  
}  
  
// Set superclass  
Car.prototype = new Vehicle();  
Car.prototype.constructor = Car;  
  
// Method declaration  
Car.prototype.honk = function() {  
  console.log("Too slow!");  
};
```

Puesto que Snorky usa el concepto de clase de forma recurrente, es importante encontrar una manera más expresiva de utilizar estas construcciones.

Se han probado varias bibliotecas, siendo finalmente escogida la biblioteca `my.Class.js`⁴⁴. No es la más potente, pero su reducido tamaño (425 bytes comprimida) y su naturaleza no intrusiva la convirtió en una opción adecuada.

El siguiente fragmento de código muestra el ejemplo anterior utilizando `my.Class.js`:

```
var Vehicle = new Class({  
  constructor: function(owner) {  
    this.owner = owner;  
  },  
  move: function() {  
    console.log("I'm moving!");  
  }  
});  
  
var Car = new Class(Vehicle, {  
  constructor: function(owner, color) {  
    Car.Super.call(this, owner);  
    this.color = color;  
  },  
  honk: function() {  
    console.log("Too slow!");  
  }  
});
```

⁴⁴ <https://github.com/jiem/my-class>


```
}  
});
```

Biblioteca de señales

Los usuarios del conector a menudo necesitan escuchar eventos como `messageReceived` o `deltaReceived`. El mecanismo más básico para permitir esto en JavaScript es una propiedad `callback`. La clase servicio tendría métodos reemplazables donde un usuario podría escribir código para el manejo de eventos.

Así fue como los eventos se manejaban en el prototipo anterior, y también fue como se manejaron al principio en esta nueva versión.

Sin embargo, según el desarrollo ha ido avanzando se ha notado que este método no es suficientemente flexible para algunas aplicaciones, ya que hace difícil añadir varios manejadores para un mismo evento.

En lugar del método clásico, se introdujeron objetos `Signal`, que permiten añadir y quitar cualquier número de funciones manejadoras para cada evento.

La clase `Signal` procede de la biblioteca `js-signals`⁴⁵.

Se ha dejado un punto de extensión para poder modificar el comportamiento de la clase `Signal` dentro del conector, pudiendo, por ejemplo, ejecutar código adicional cada vez que se atiende un evento.

Diseño de clases del conector

La figura 7.16 muestra las clases definidas en el conector. Hay una clase `Snorky` que contiene una referencia a la clase encargada de la conexión (`WebSocket` o `SockJS`) y varios *conectores de servicios* que ofrecen una interfaz cómoda para comunicarse con los servicios de `Snorky` del lado del servidor.

Servicios de ejemplo

Se han añadido algunos servicios para probar que el framework funciona:

- `MessagingService` permite a los usuarios identificarse con un nombre y enviar mensajes a otros usuarios.
- `PubSubService` permite a los usuarios unirse a canales (identificados por una entidad JSON, como una cadena o un diccionario) y compartir mensajes con el resto de los usuarios del canal.

Para cada uno de estos servicios se ha implementado una aplicación de demostración.

⁴⁵ <https://millermedeiros.github.io/js-signals/>

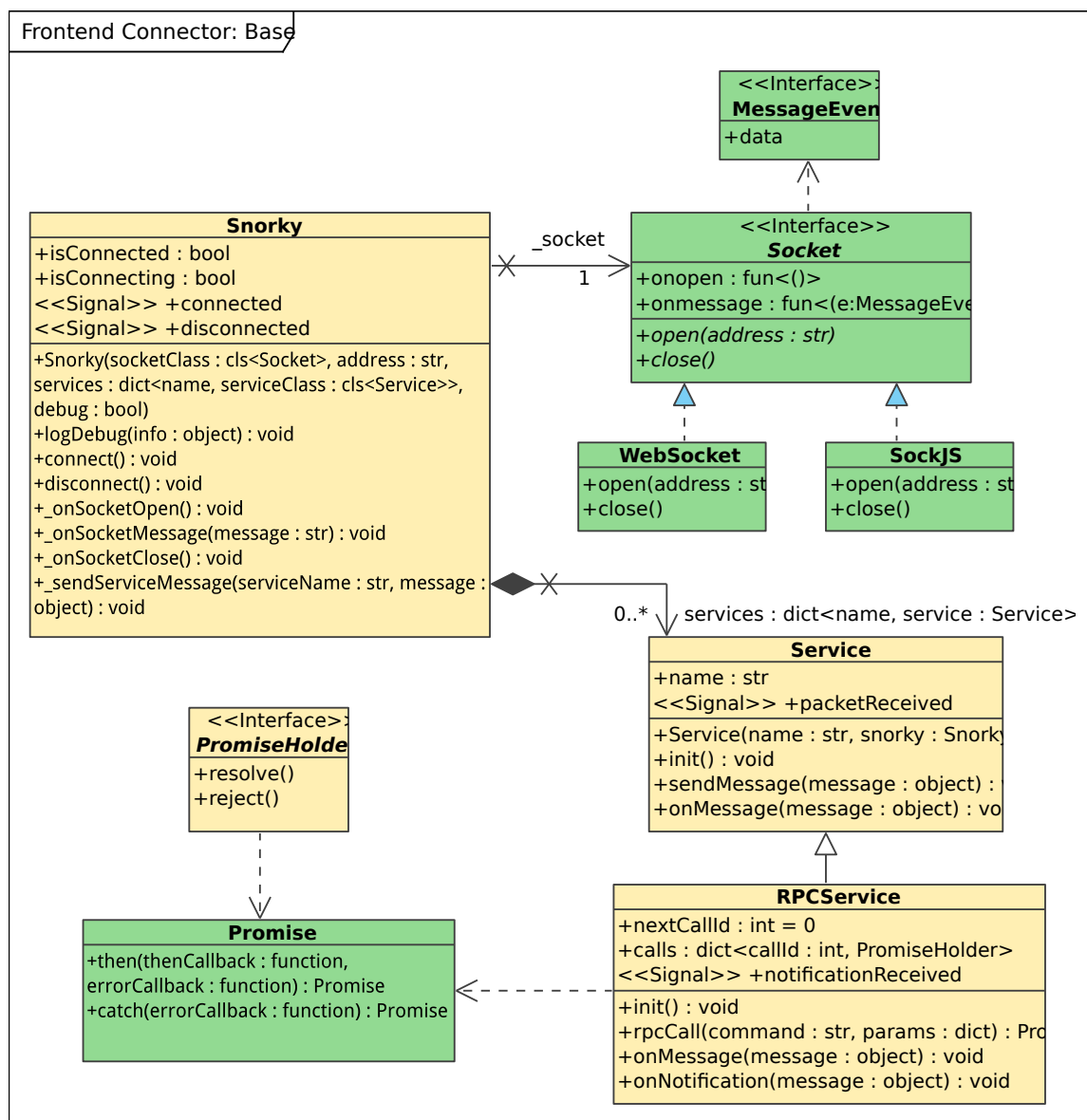


Figura 7.16: Las clases básicas de Snorky.js.

El servicio de mensajería

Como demostración de un servicio de Snorky se ha creado el servicio de mensajería, `MessagingService`.

Este servicio permite a los usuarios identificarse con un nombre y enviar mensajes a otros usuarios.

La figura 7.17 muestra cómo el servicio ha sido modelado en UML. Una clase extendiendo de `RPCService` que maneje instancias de objetos `Client` es suficiente para definir un servicio.

La figura 7.18 muestra cómo se ha modelado el conector del lado del cliente.

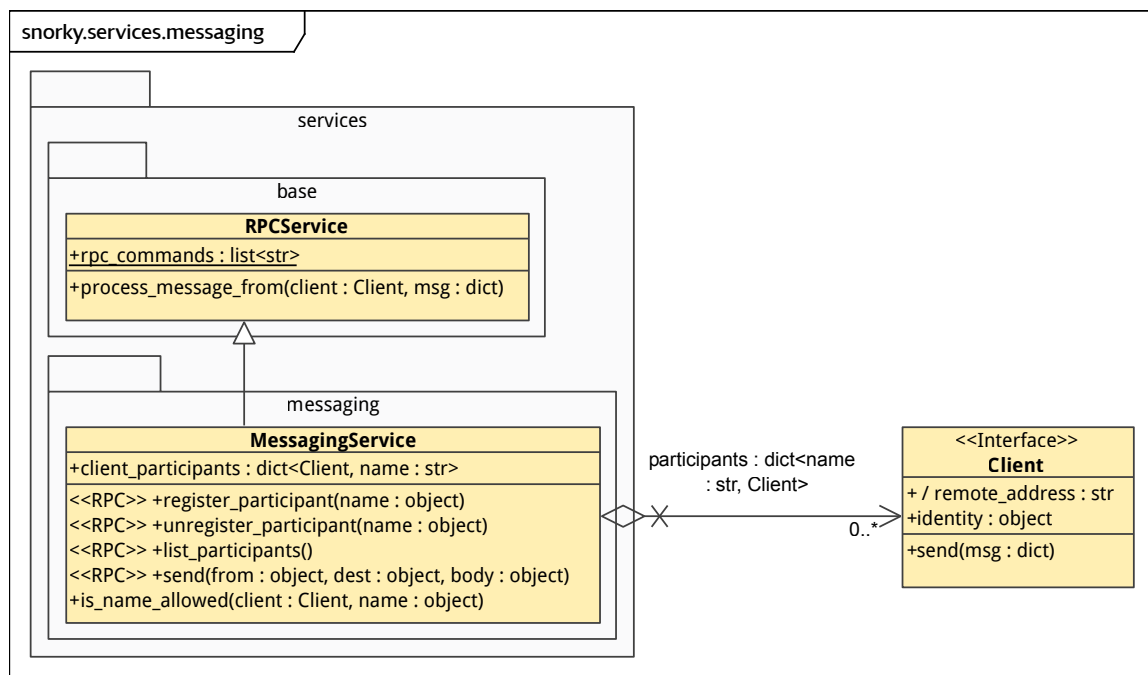


Figura 7.17: Este diagrama muestra cómo los servicios de Snorky pueden ser modelados en UML.

Se ha realizado una aplicación web sencilla para mostrar este servicio en acción, como puede verse en la figura 7.19. Esta aplicación cuenta con un formulario que cubre las funcionalidades del servicio, permitiendo registrar un nombre y enviar mensajes a otros usuarios.

Puesto que probar esta demo en varios navegadores resultaba bastante incómodo, se ha utilizado un sistema de prueba automatizado basado en Robot Framework⁴⁶.

Este sistema inicia automáticamente varios navegadores a elegir y automáticamente interactúa con la interfaz de la forma que se especifique en los ficheros de prueba, rellenando campos de texto, haciendo clic en botones y comprobando que aparecen la

⁴⁶ <http://robotframework.org/>

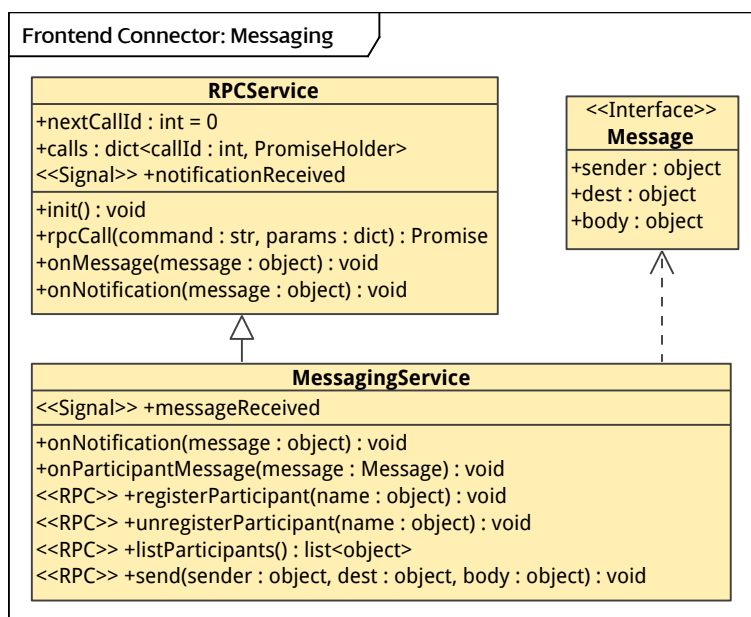


Figura 7.18: El conector de MessagingService soporta los mismos métodos que su contraparte en el lado del servidor y define un evento para los mensajes recibidos.

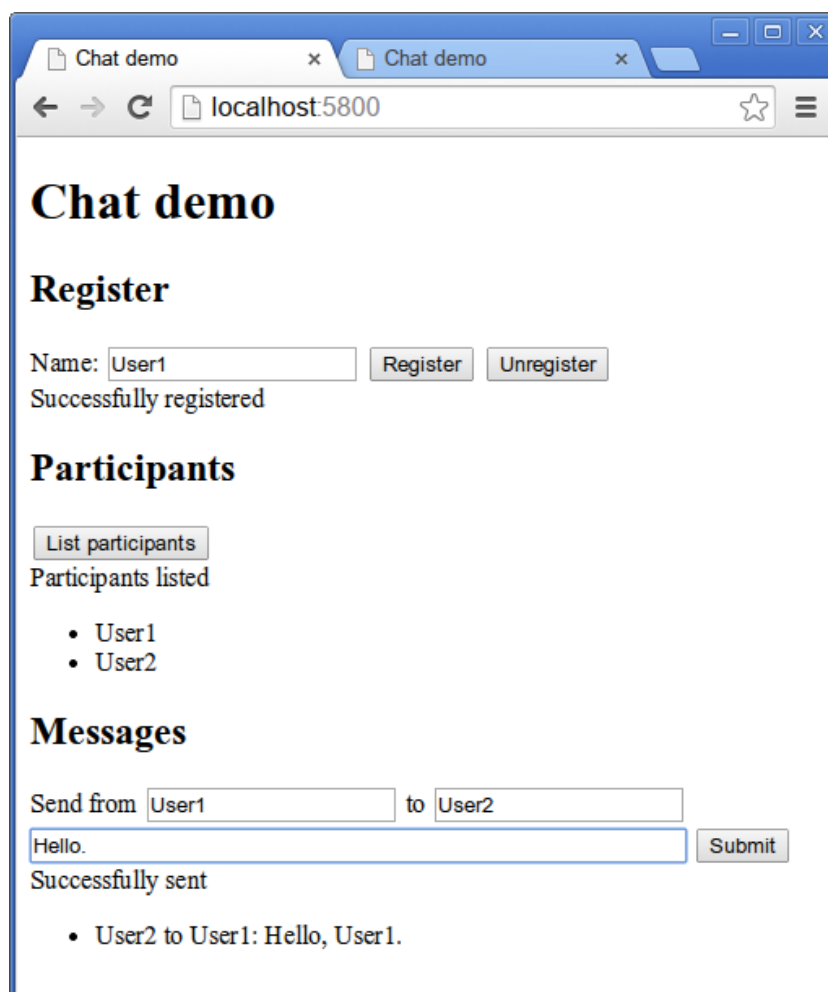
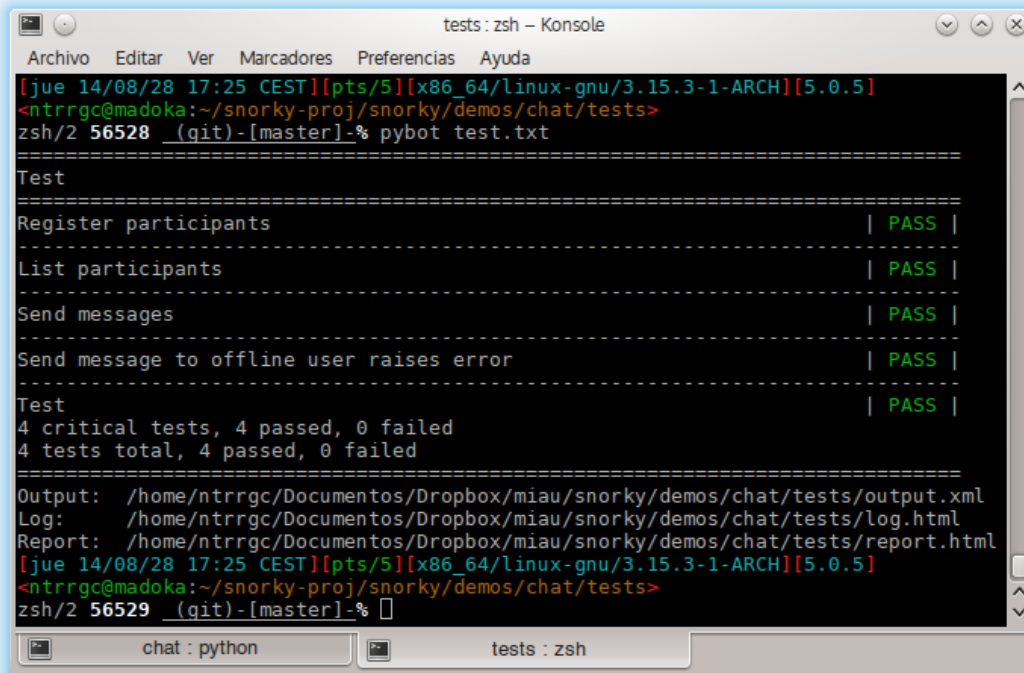


Figura 7.19: Esta aplicación web demuestra el uso del servicio de mensajería.

secciones de texto adecuadas. La figura 7.20 muestra el resultado de ejecutar la batería de pruebas de Robot.



```

tests : zsh - Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
[jue 14/08/28 17:25 CEST][pts/5][x86_64/linux-gnu/3.15.3-1-ARCH][5.0.5]
<ntrrgc@madoka:~/snorky-proj/snorky/demos/chat/tests>
zsh/2 56528 _(git)-[master]-% pybot test.txt

=====
Test
=====
Register participants                                     | PASS |
-----
List participants                                         | PASS |
-----
Send messages                                             | PASS |
-----
Send message to offline user raises error                 | PASS |
-----
Test                                                     | PASS |
4 critical tests, 4 passed, 0 failed
4 tests total, 4 passed, 0 failed
=====
Output:  /home/ntrrgc/Documentos/Dropbox/miau/snorky/demos/chat/tests/output.xml
Log:     /home/ntrrgc/Documentos/Dropbox/miau/snorky/demos/chat/tests/log.html
Report:  /home/ntrrgc/Documentos/Dropbox/miau/snorky/demos/chat/tests/report.html
[jue 14/08/28 17:25 CEST][pts/5][x86_64/linux-gnu/3.15.3-1-ARCH][5.0.5]
<ntrrgc@madoka:~/snorky-proj/snorky/demos/chat/tests>
zsh/2 56529 _(git)-[master]-%

```

Figura 7.20: Salida resultante de ejecutar la batería de pruebas de Robot.

Integración con AngularJS

Una vez construida la aplicación de demostración de mensajería en JavaScript sin añadidos a parte de Snorky, se ha implementado otra versión basada en el framework AngularJS, con el objetivo de probar la interacción entre ambos frameworks.

AngularJS, anteriormente mencionado en el apartado *El conector JavaScript* de Sincronización con bases de datos, es un framework basado en la arquitectura Modelo-Vista-Controlador para el desarrollo de aplicaciones web del lado del cliente.

Se ha desarrollado una extensión que ha hecho posible hacer que los dos interactúen sin problema.

La extensión desarrollada se denomina `angular-snorky.js`. Es un módulo de AngularJS que parchea Snorky en dos aspectos:

- Sustituye la clase `Promise` de Snorky por un adaptador para que utilice las promesas de AngularJS.
- Parchea el método `dispatch()` de `Signal` para que se produzca un ciclo `$digest` después de procesar eventos. Esta es la manera de solicitarle a AngularJS

que compruebe si ha habido cambios en el modelo y actualice la interfaz en consecuencia.

La figura 7.21 muestra la nueva aplicación de demostración.

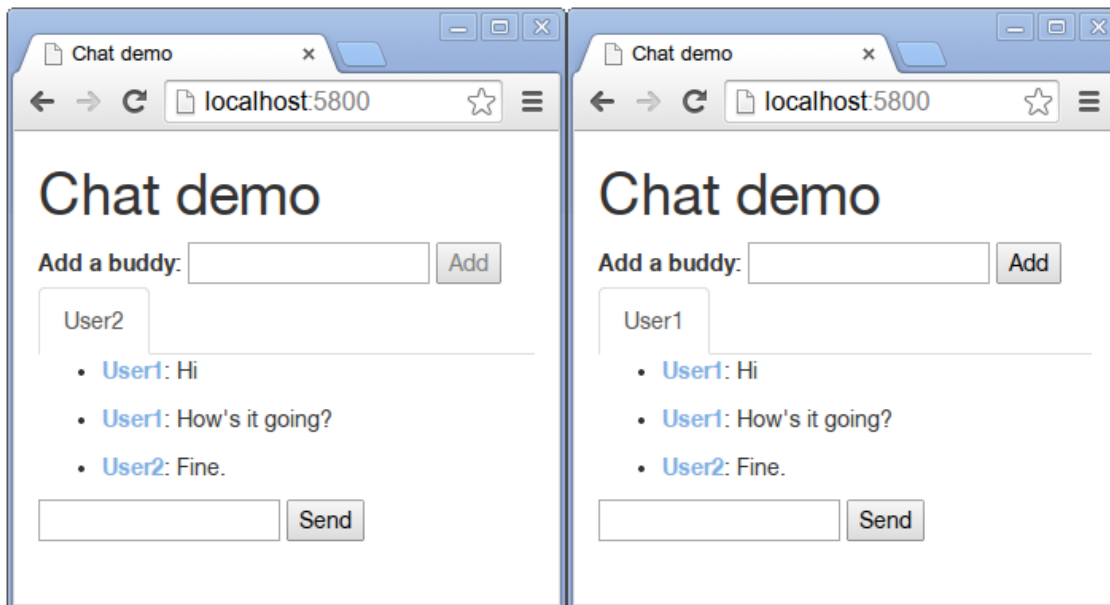


Figura 7.21: Aplicación demostrando la integración de Snorky con AngularJS.

El servicio Pub Sub y la interfaz de backend

Uno de los patrones más recurrentes en el desarrollo de aplicaciones web en tiempo real es el patrón de *publicación-suscripción*, comúnmente denominado por el acrónimo inglés *Pub Sub*.

De forma genérica, un servicio *Pub Sub* es aquel que permite a sus usuarios suscribirse a uno o más canales y desde ese momento recibir mensajes que otros usuarios *publican* en el mismo canal.

Se ha desarrollado una implementación de este patrón en Snorky, con el nombre `PubSubService`: los clientes pueden publicar y suscribirse a un canal identificado por una cadena u otra entidad JSON. El servicio permite configurar bajo qué condiciones un cliente puede o no publicar en un canal.

Si bien una posibilidad de uso del servicio *Pub Sub* está en la comunicación de cliente a cliente, otro caso incluso más frecuente es el de la comunicación de servidor a cliente, donde sólo un servidor con permisos especiales puede publicar.

Para cubrir este caso de uso se ha añadido un servicio adicional, `PubSubBackend`. Este servicio se inicializa enlazándolo con una instancia de `PubSubService`.

`PubSubBackend` no está pensado para ser expuesto por `WebSocket`, sino en una interfaz

HTTP protegida, de manera que sólo sea accesible por otros equipos o servidores autorizados de la misma red. Este servicio ofrece sólo un método RPC, `publish`, el cual envía una publicación a un canal en el servicio asociado, sin restricciones.

La figura 7.22 muestra la interacción descrita.

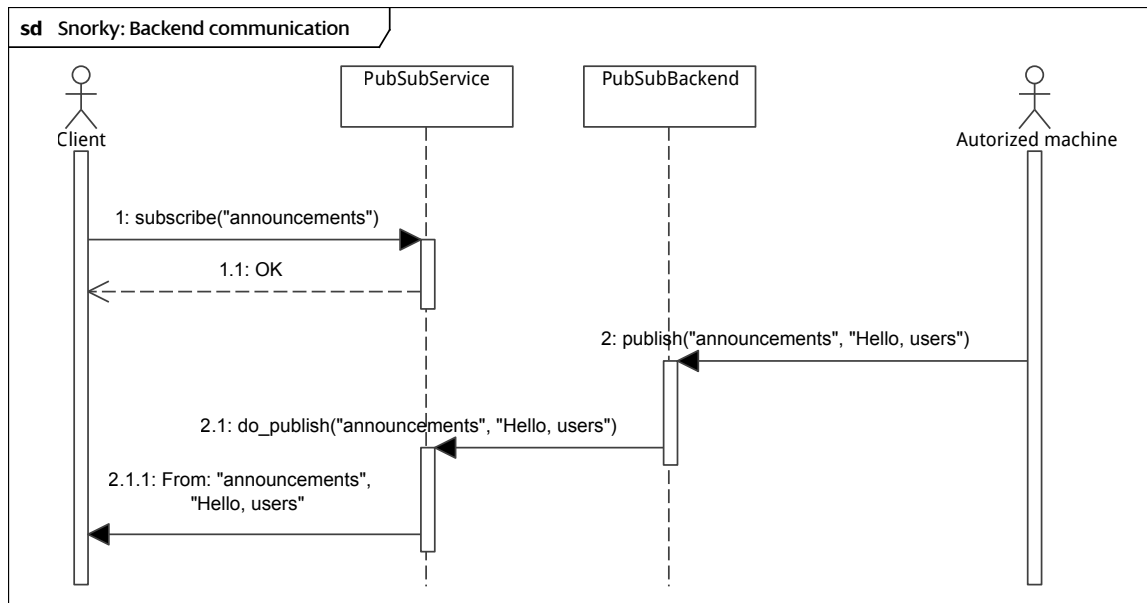


Figura 7.22: Unos servicios pueden llamar a otros. En este caso el servicio `PubSubBackend` atiende peticiones de equipos autorizados para publicar mensajes en una instancia de `PubSubService` que no permite publicar mensajes de otra forma.

Para demostrar el funcionamiento de ambos servicios se ha creado una aplicación de ejemplo, mostrada en la figura 7.23. En ella los clientes web conectan a `Snorky` y se suscriben a un canal del servicio `Pub Sub`, pero no pueden publicar. Un proceso con acceso al canal de backend sí puede enviar un mensaje al servicio `PubSubBackend` para publicar un mensaje, como se demuestra con la orden `publish.py`.

Conviene destacar que esta posibilidad de comunicar diferentes servicios expuestos en diferentes interfaces es clave para añadir la funcionalidad de sincronización con bases de datos a este nuevo sistema.

Integración

Una vez bien definida la arquitectura de la fase anterior, es momento de adaptar el anterior prototipo de sincronización con bases de datos para que funcione en el nuevo sistema.

Adicionalmente, ya que será necesario reescribir código viejo, se aprovechará la situación para reflexionar sobre decisiones de diseño pasadas y hacer cambios que se crean convenientes.

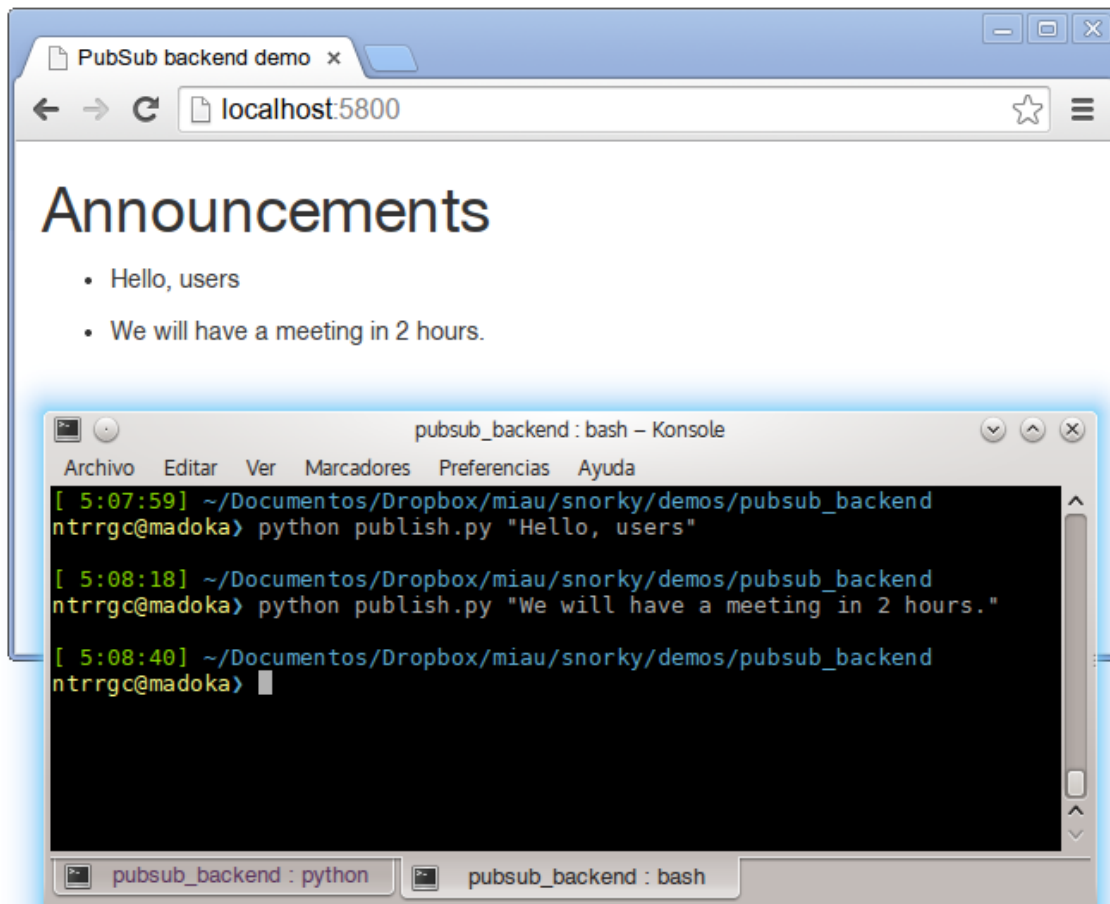


Figura 7.23: Con una orden de consola se envía un mensaje a los usuarios. La orden de consola necesita tener acceso al canal *backend* para funcionar.

Campos renombrados

- Algunos campos de la API de Snorky fueron renombrados para hacerlos más breves. Por ejemplo, `model_class_name` se convirtió en `model`.
- El campo `model_key` que se enviaba a las clases repartidoras era un permitía especificar un filtro que éstas usarían para determinar qué clase de notificaciones debe recibir cada cliente. Recibía este nombre porque en `SimpleDealer` se utilizaba el valor de este campo como clave de un índice. Sin embargo, este nombre tenía menos sentido con las otras clases repartidoras por lo que ha sido renombrado a `query`.

Diseño

El diseño del prototipo ha sido reutilizado en su mayor parte, aunque ha habido que hacer algunos cambios para acomodar la nueva arquitectura.

- La clase base `FrontendHandler` ha sido reemplazada por un servicio de Snorky, `DataSyncService`. Los clientes envían peticiones a este servicio para adquirir y cancelar suscripciones.
- La vieja API REST ha sido reemplazada con otro servicio RPC, `DataSyncBackend`. El servidor principal envía peticiones a este servicio para autorizar nuevas suscripciones y para enviar notificaciones de cambios.
- La vieja clase `Facade` ha sido eliminada pues ya no era necesaria en la nueva arquitectura.
- La vieja clase `Client` ha sido reemplazada con la nueva clase `Client` de Snorky.

La clase vieja guardaba alguna información relacionada con la sincronización con bases de datos, tal como el conjunto de suscripciones que el cliente ha adquirido.

En la nueva arquitectura de Snorky no es recomendable añadir nuevos métodos a la clase `Client` porque es compartida con más servicios y podría dar lugar a conflictos. En lugar de eso, la información de suscripciones se ha movido a un diccionario dentro de `SubscriptionManager`, acompañando a otros similares.

Las figuras 7.24 y 7.25 muestran el nuevo diseño de clases.

Implementación del servidor

Unas partes del código han sido reescritas y otras han sido reutilizadas, haciendo balance en cada caso del coste de adaptar contra el coste de reescribir.

En los casos en que se escogió reescribir, se ha utilizado la doble vista del editor para trabajar viendo la versión nueva y la vieja, de manera que fuera más rápido copiar

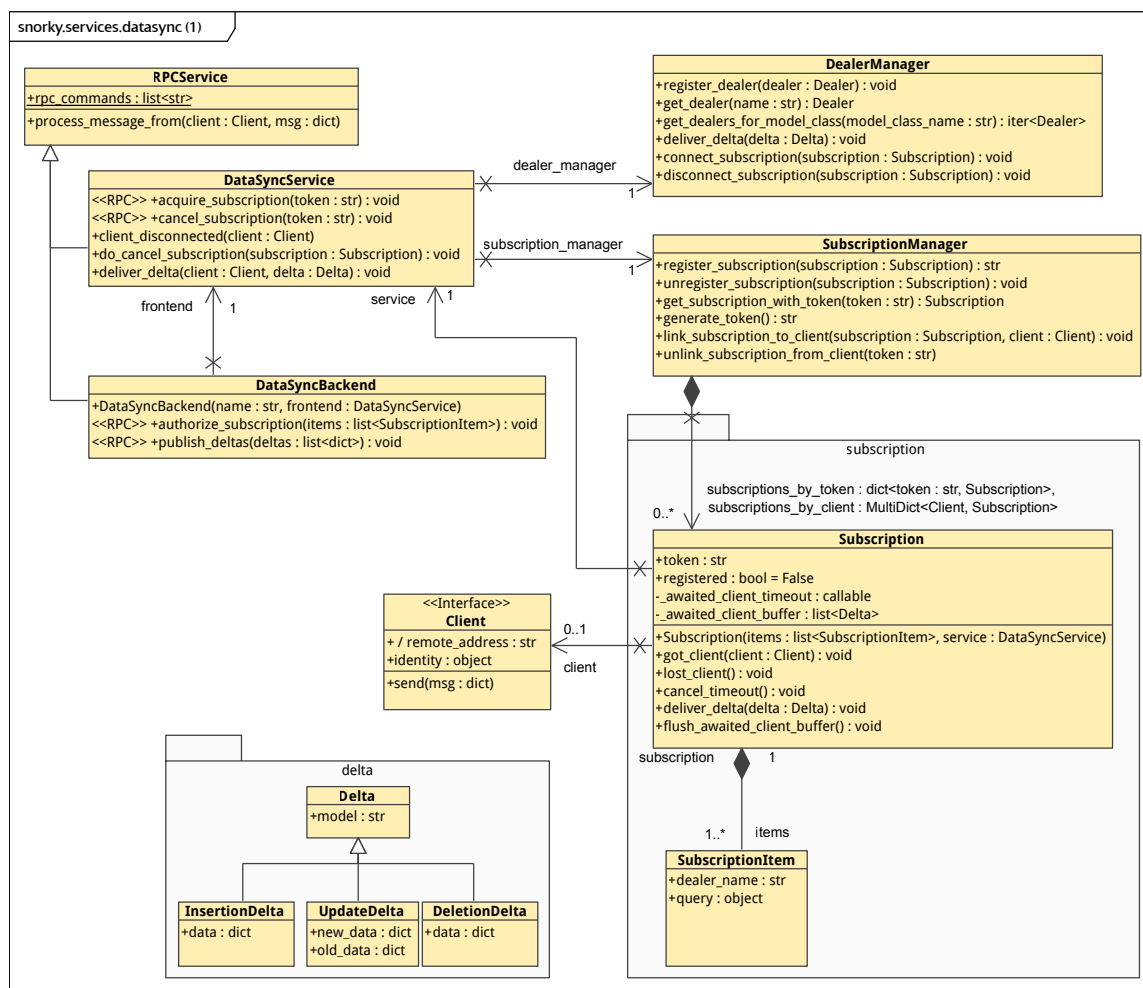


Figura 7.24: El diseño de Snorky DataSync (Parte 1).

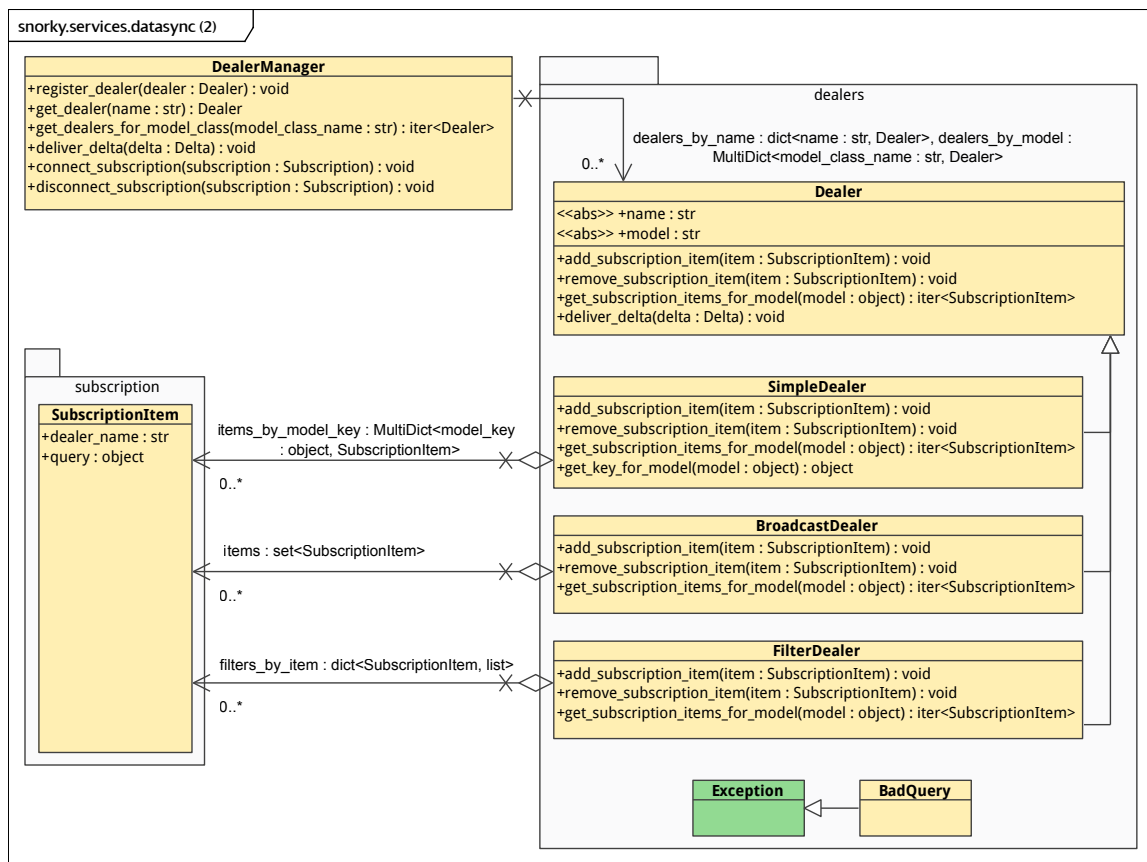


Figura 7.25: El diseño de Snorky DataSync (Parte 2).

algunas cosas y comprobar que no se ha producido ninguna omisión que pudiera provocar fallos posteriormente.

Se observa que muchas estructuras de datos son de la forma `dict<key, set<Object>>`, donde varios objetos se indexan y agrupan por una cierta cualidad. Por ejemplo, los objetos del tipo `Subscription` se pueden indexar por el campo `client` para poder consultar las suscripciones de un cliente.

Este patrón ha sido aprovechado en una nueva clase, `MultiDict`, que ofrece la misma estructura interna pero añade métodos adicionales para añadir y quitar elementos creando y eliminando conjuntos vacíos según sea necesario.

En el apartado de pruebas también se han reutilizado muchos casos de prueba del prototipo. En algunos casos adaptar los casos de prueba ha sido tan fácil como copiar y pegar y en otros ha requerido más esfuerzo manual, aunque los casos de prueba viejos han servido como guía.

Adicionalmente, se ha incorporado el paquete `mock`⁴⁷ para ayudar en la creación de de clases sustituto, ofreciendo funciones espía.

Una función espía es aquella que reemplaza a otra, registrando cuántas veces es llamada y con qué argumentos. Esto permite probar colaboraciones entre clases con menos código que reimplementando métodos manualmente.

Conector JavaScript

El nuevo servicio de sincronización con bases de datos se ha integrado en el conector de `Snorky`.

El sistema de colecciones también ha sido adaptado. El siguiente código de la documentación muestra el código requerido para escuchar a un evento `deltaReceived` y actualizar en consecuencia un array `comments`.

```
var collectionProcessor = new Snorky.DataSync.CollectionDeltaProcessor({
  "Comment": new Snorky.DataSync.ArrayCollection(comments)
});

// Delegate delta processing to the collection processor
snorky.services.datasync.deltaReceived.add(function(delta) {
  collectionProcessor.processDelta(delta);
});
```

Documentación

Una vez adaptado el nuevo servicio de sincronización con bases de datos a la nueva arquitectura y se considera suficientemente estable, se ha realizado un esfuerzo para

⁴⁷ <http://www.voidspace.org.uk/python/mock/>

documentar el funcionamiento del framework.

Una buena documentación es una parte fundamental de un sistema informático, especialmente cuando el objetivo de éste es que sea utilizado por otros programadores.

Por este motivo se ha utilizado una herramienta que facilita escribir documentación en un formato consistente y ofrece herramientas habitualmente esperadas en un sistema de este tipo, tales como búsqueda y consulta de código fuente.

La herramienta de documentación

Sphinx ha sido la herramienta de documentación escogida para este proyecto.

Sphinx es un generador de documentación. Lee archivos de código *fuentes* de documentación en el formato Restructured Markup (.rst) y los transforma en HTML, LaTeX y otros formatos.

El lenguaje Restructured Markup está basado en texto, tratando de ser legible tanto por humanos como por máquinas, así como poder ser editado en cualquier editor de texto convencional.

El siguiente fragmento de código muestra un ejemplo de la sintaxis de Restructured Markup tal como se ha utilizado en el capítulo Herramientas utilizadas.

```
Lenguajes de programación
~~~~~

Python fue el lenguaje de programación escogido para desarrollo de la parte
servidor. Las razones para su elección fueron.

* Experiencia probada en el desarrollo web.

* Gran cantidad de bibliotecas y frameworks con excelente documentación.

* Experiencia previa del autor con el lenguaje.

Para la parte cliente, JavaScript fue elegido por necesidad: es el lenguaje
que soportan los navegadores.
```

Elementos convencionales de texto plano tales como el asterisco o líneas de guiones son transformados en listas, encabezados, etc. Hay directivas adicionales para imágenes, fragmentos de código, enlaces, notas y más.

Estilo

Para la salida HTML se ha utilizado el tema *Read The Docs*. La figura 7.26 muestra una captura de pantalla de la documentación.

Ha sido necesario hacer algunos arreglos al código fuente del tema para que el estilo

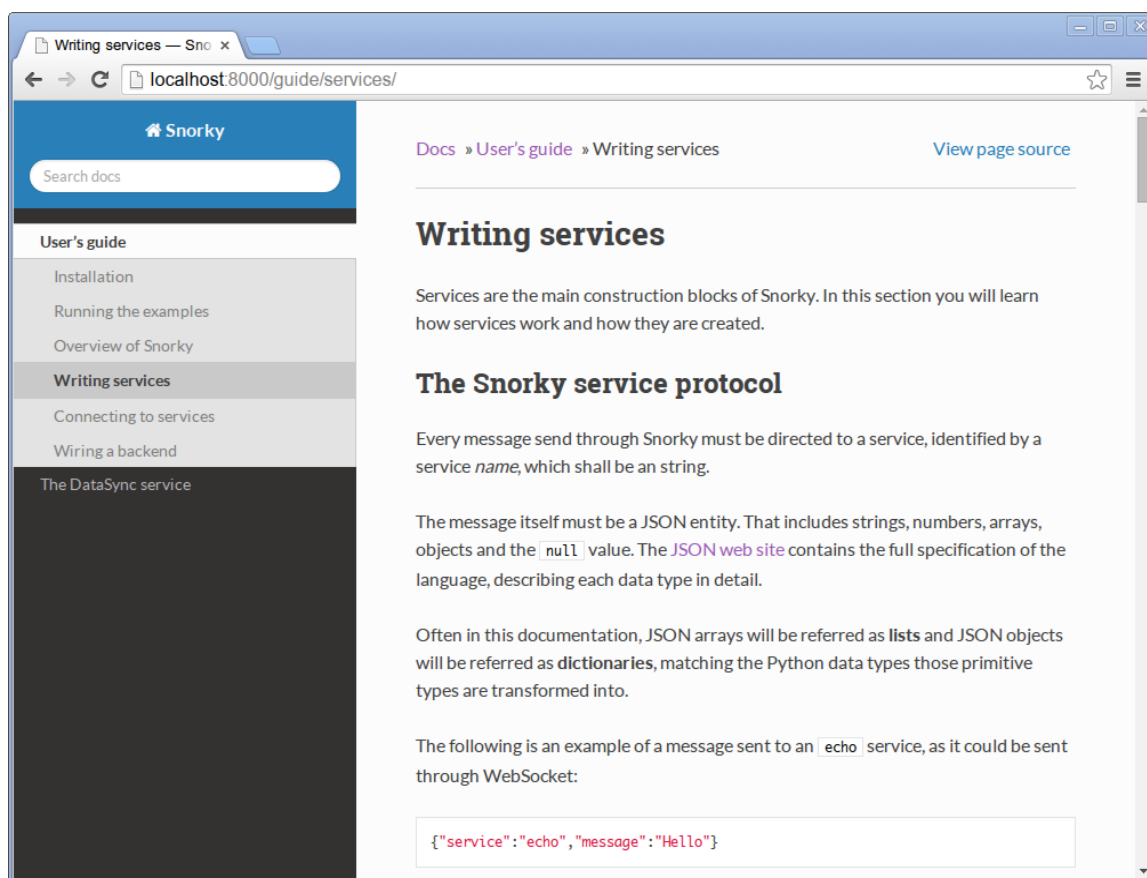


Figura 7.26: La documentación de Snorky.

fuera consistente.

Contenidos

Hay dos maneras opuestas de escribir documentación:

- Escribir una página para cada clase del sistema, explicando todos los métodos de la clase. Este es el enfoque con el que está escrita, por ejemplo, la documentación de la API de Java⁴⁸.
- Escribir como un libro: cada capítulo debe centrarse en explicar un conjunto de conceptos relacionados, de manera incremental y debe mostrar de ejemplos. Este enfoque es muy popular entre los frameworks desarrollados en Python, siendo usado de manera extensiva en proyectos como Tornado⁴⁹, Celery⁵⁰ o Django⁵¹ entre muchos otros.

Consultar una documentación escrita de la primera manera es más rápida cuando estás acostumbrado al software y sólo quieres buscar información sobre un método o una clase específica en la documentación.

Por otro lado, la segunda manera es mucho mejor para recién llegados que quieren adquirir un conocimiento básico sobre el funcionamiento del framework antes de buscar detalles más concretos.

Para la documentación de Snorky se ha escogido la segunda opción.

Se han escrito dos capítulos:

- Una guía de usuario básica que explica cómo instalar Snorky, cómo es su arquitectura y cómo crear servicios.
- Un capítulo acerca del servicio de sincronización con bases de datos, explicando que normas debe seguir un sistema que quiera implementarlo, qué servicios están implicados, cómo funcionan los repartidores y otros conceptos relacionados.

La documentación se ha compilado en formato PDF como anexo de esta memoria, así como en formato HTML en el directorio Documentación. Ambos documentos están incluidos en el CD adjunto.

Página web

Se ha diseñado y escrito una pequeña página web con el objetivo de que sirva como presentación para potenciales usuarios del framework, accesible desde la siguiente URL:

⁴⁸ <https://docs.oracle.com/javase/6/docs/api/>

⁴⁹ <http://www.tornadoweb.org/en/stable/guide/intro.html>

⁵⁰ <https://celery.readthedocs.org/en/latest/userguide/index.html>

⁵¹ <https://docs.djangoproject.com/en/1.6/>

<http://informatica.usal.es/snorky>

Se ha utilizado el framework Django para crear la estructura de la página, junto con una extensión para coloreado de código desde las plantillas HTML.

Las imágenes han sido realizadas con el editor vectorial Inkscape⁵².

Se ha diseñado con cuidado para que la página web sea funcional en diversas resoluciones. Las figuras 7.27, 7.28 y 7.29 muestran el resultado.

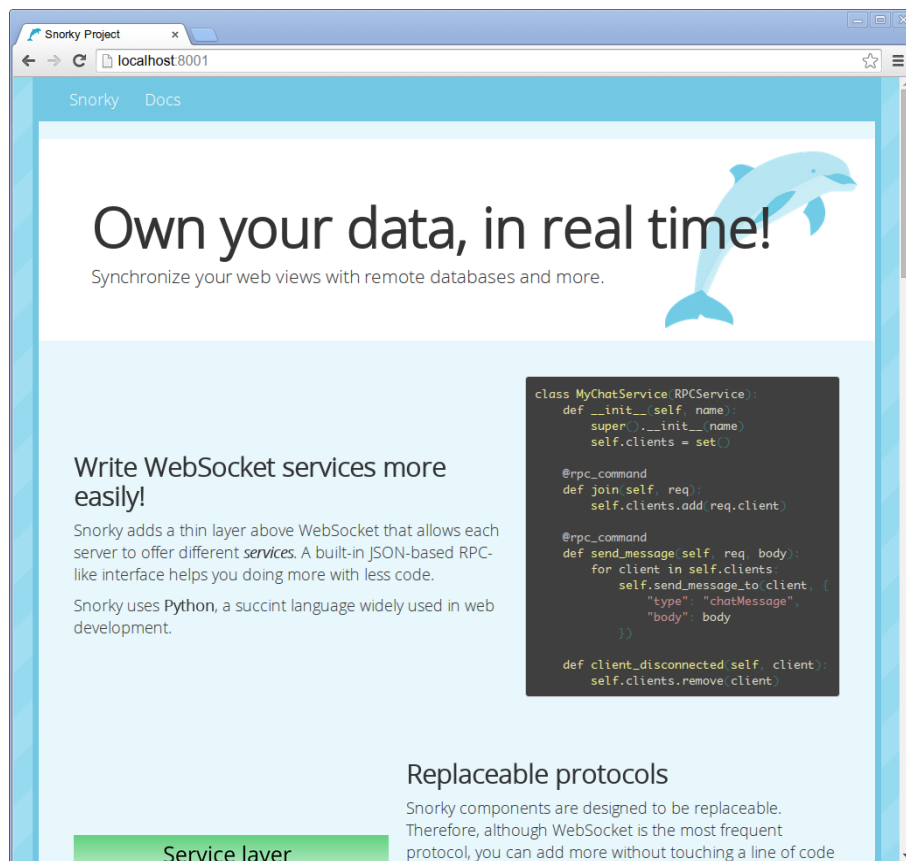


Figura 7.27: Página web de Snorky en una pantalla mediana.

⁵² <http://www.inkscape.org/en/>

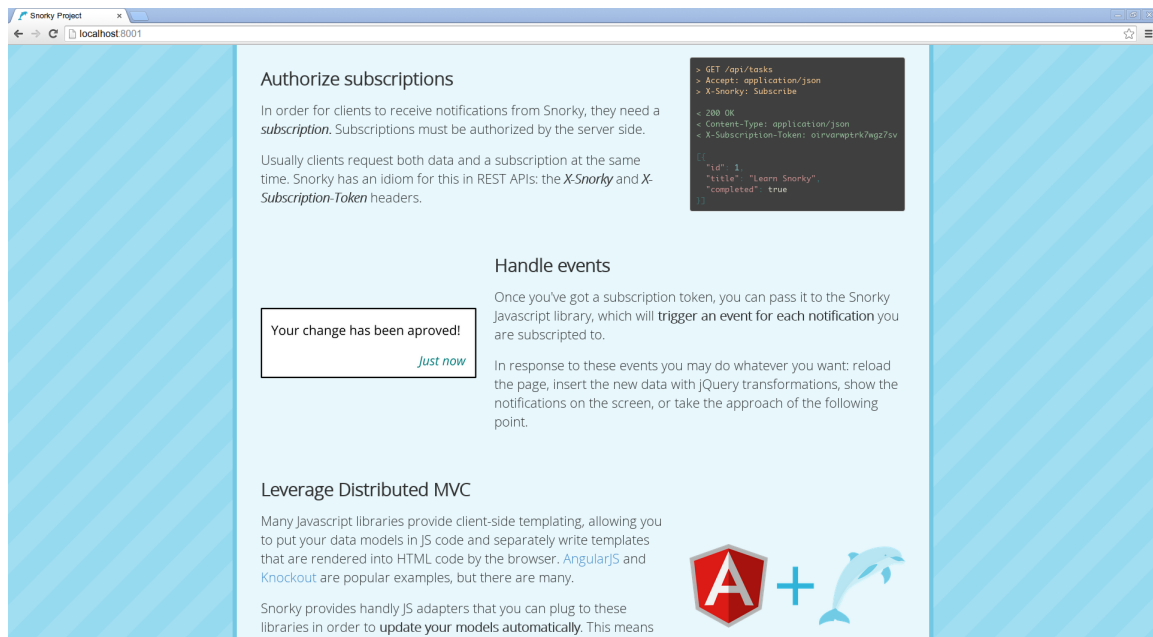


Figura 7.28: Página web de Snorky en una pantalla grande.

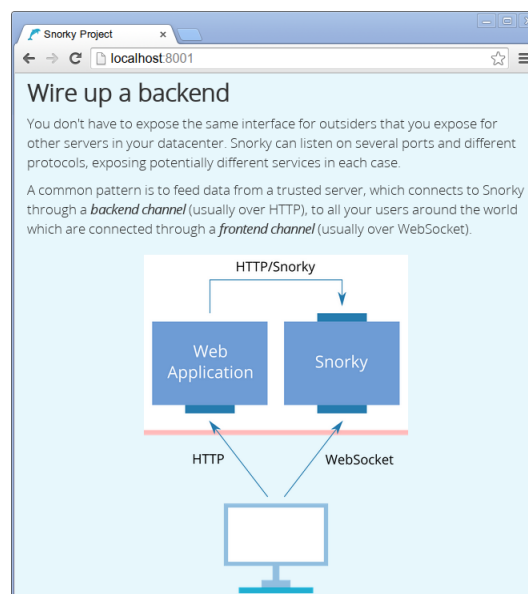


Figura 7.29: Página web de Snorky en una pantalla pequeña.

Conclusiones y líneas de trabajo futuro

Las principales conclusiones y aportaciones de este trabajo son las siguientes:

- Se ha diseñado, implementado y probado un framework para el desarrollo de aplicaciones web con características de tiempo real.

Este framework permite incorporar características tales como sincronización con bases de datos remotas de manera rápida y sencilla, facilitando la realización de aplicaciones colaborativas.

El sistema se compone de una parte servidor que se comunica con los clientes finales mediante el protocolo WebSocket y con otros servidores del sistema mediante HTTP.

En este sistema, los servidores responsables del almacenamiento de datos envían notificaciones al servidor WebSocket de todos los cambios ocurridos que puedan ser de relevancia para los usuarios de la aplicación web.

Se ha definido un protocolo de suscripción en el que los clientes, al realizar una consulta a la base de datos, reciben un código que les autoriza a recibir notificaciones sobre cambios en el conjunto de datos que han consultado. Este protocolo reutiliza todas las políticas de seguridad del servidor responsable del almacenamiento de datos.

Se ha definido una interfaz que permite al programador especificar cómo se repartirán las notificaciones de manera que los clientes sólo reciban datos que hayan suscrito.

- Se ha trabajado en hacer el framework extensible para permitir la adición de nuevas funcionalidades.

Algunas de las funcionalidades incorporadas son la comunicación entre usuarios por medio del protocolo publicación-suscripción y la mensajería instantánea.

El sistema desarrollado permite al desarrollador de aplicaciones web codificar nuevas funcionalidades con una interfaz de programación basada el modelo de

llamadas a procedimiento remoto.

- Se han desarrollado varias bibliotecas (*conectores*) para integrar el framework en diversas tecnologías tales como Python, Javascript, AngularJS, Django y Django REST Framework.

Estos conectores reducen al mínimo la cantidad de código necesitada para realizar tareas comunes como enviar notificaciones, enviar peticiones a servicios o actualizar la vista como respuesta a una notificación.

- Se ha puesto en práctica una metodología de desarrollo soportada por pruebas automáticas.

Se han implementado grandes baterías de pruebas para garantizar la estabilidad del servidor, así como de los distintos conectores.

Se han estudiado diferentes sistemas para escribir, ejecutar y mantener distintos tipos de pruebas, tanto unitarias, como de integración, como de sistemas, según ha sido conveniente en cada situación.

- Se han desarrollado varias de aplicaciones de ejemplo para comprobar el funcionamiento de las distintas funcionalidades del framework.

Estas aplicaciones también demuestran la integración del framework con varias tecnologías existentes, tales como los frameworks del lado del servidor Flask y Django y el framework del lado del cliente AngularJS.

- Se ha elaborado una completa documentación sobre el uso del framework en PDF y HTML.

Se ha cuidado el aspecto de la documentación para asegurar que el estilo es consistente.

Se ha desarrollado una página web con el objetivo de que sirva como presentación del framework, disponible en <http://informatica.usal.es/snorky>.

- Se han seguido las prácticas de seguridad adecuadas para garantizar el correcto control de acceso a los datos, centralizando las políticas de control de acceso en el servidor principal y protegiendo las suscripciones mediante códigos no predecibles calculados por un generador de números aleatorios de calidad criptográfica.

Además, desde una perspectiva personal del estudiante:

- Se han puesto en práctica los conocimientos adquiridos en el Grado en Ingeniería Informática mediante la realización de un proyecto de tamaño considerable, de forma satisfactoria.
- Se han aprendido y puesto en práctica los principios del desarrollo asistido por pruebas automáticas.

Se ha aprendido a diseñar clases software con vistas a que sean fáciles de probar

de forma aislada, facilitando el desarrollo de pruebas unitarias.

Se han aprendido técnicas para probar clases con colaboraciones, tales como las clases sustituto (*mock*) y las funciones espía (*spies*).

Se ha aprendido a desarrollar pruebas de sistema de manera que sea posible probar de manera conjunta la unión de todos los componentes del software implicados en un caso de uso.

- Se han estudiado numerosos conceptos del framework AngularJS con vistas tanto a la realización de aplicaciones web de todo tipo, como a su integración con Snorky.
- Se ha aprendido a construir y mantener bibliotecas JavaScript utilizando el ejecutor de tareas Grunt.
- Se ha adquirido experiencia en varios entornos de programación orientados a desarrollos web, tales como Django y Tornado.
- Se ha aprendido a usar el control de versiones git para la gestión de un proyecto de software.

Particularmente la característica de bisección ha sido de gran utilidad para encontrar y solucionar regresiones manifestadas sólo en ciertas plataformas.

Se han identificado varias áreas en las que este trabajo de fin de grado podría ser continuado.

- Publicar el framework bajo una licencia de software libre, haciéndolo accesible al resto del mundo.
- Desarrollar nuevos servicios reutilizables.
 - Un servicio de bloqueos podría evitar que dos usuarios trabajaran al mismo tiempo sobre un mismo campo, de manera que un usuario no pise accidentalmente el trabajo de otro.

Este servicio podría indicar también dónde está posicionado el cursor de cada usuario y si está o no editando.
 - Un servicio de cambios provisionales, de manera que en aquellas aplicaciones donde sea conveniente pueda verse lo que otros usuarios están tecleando antes de que sea registrado en la base de datos.
- Reducir el tamaño de los mensajes del protocolo.
- Realizar pruebas de rendimiento para comprobar el funcionamiento del sistema ante una carga elevada.
- Desarrollar una capa de gestión de sesiones.

Añadir una capa de gestión de sesiones permitiría a los usuarios reanudar

una conexión perdida o hacer *roaming* entre diferentes redes sin perder la sincronización.

El protocolo de mensajería instantánea XMPP define una extensión con funcionalidades similares: XEP-0198 Stream Management⁵³.

Los clientes al conectarse con el servidor por primera vez negociarían una *sesión*, identificada por un identificador que mantienen en secreto. A partir de ese momento todos los mensajes del protocolo se enviarían identificados con un número de secuencia y la otra parte deberá responderlos con un mensaje de confirmación (*ack*).

Si en algún momento el cliente pierde la conectividad con el servidor, puede iniciar una nueva conexión, pidiéndole restaurar la sesión anterior.

Para que la gestión de sesiones funcione es necesario que el servidor guarde en memoria todos los mensajes no confirmados, de manera que si se descubre que el usuario ha reconectado desde otra red sea posible enviárselos de nuevo. El cliente también debe hacer lo mismo para garantizar que los mensajes enviados a la aplicación son recibidos correctamente.

El tiempo durante el que un mensaje puede almacenarse en memoria esperando un mensaje de confirmación es limitado. Puede ser necesario guardar estos mensajes en disco para no llenar la memoria principal, o incluso para evitar ataques de denegación de servicio.

- Implementar un sistema no bloqueante de envío de notificaciones.

Hasta ahora las aplicaciones de demostración han utilizado las APIs de comunicación HTTP síncronas para enviar notificaciones.

Esto ocasiona que cuando un cliente envía una petición al servidor principal para hacer una modificación en la base de datos la petición no es respondida hasta que Snorky ha recibido la notificación.

Sin embargo, este retraso no es necesario. Hay varias formas posibles de evitarlo:

- Utilizar un servidor proxy local: Un servidor de este tipo ofrecería exactamente la misma interfaz que Snorky, pero no procesaría él mismo las peticiones, sino que las enviaría al servidor real.

Con una opción adicional este servidor no esperaría a recibir la respuesta de Snorky sino que le enviaría una respuesta vacía al servidor principal de forma inmediata.

Si bien la comunicación sigue siendo síncrona, con este esquema la latencia queda reducida al tiempo de la comunicación entre ambos procesos, no dependiendo de la latencia de la red que separa el servidor principal y Snorky.

- Utilizar un método de comunicación entre procesos sin confirmación. El

⁵³ <http://xmpp.org/extensions/xep-0198.html>

método anterior se podría extender para funcionar por un protocolo que no requiera confirmación del proceso destino, posiblemente reduciendo aun más la latencia, aunque quizás no de forma notable.

- Utilizar un sistema externo de cola de mensajes que maneje este tipo de situaciones.
- Permitir que el sistema sea escalable a varios servidores WebSocket.

Hasta ahora sólo se ha contemplado un único servidor WebSocket. Sin embargo, la potencia de los ordenadores es limitada, por lo que es posible llegar a un punto en el que este servidor no soporte la carga que recibe.

Para lidiar con este problema se podría convertir Snorky en un sistema distribuido que soportara varios servidores WebSocket.

Un sistema de este tipo plantearía muchos retos:

- El sistema tendría que conseguir que las tareas se repartieran de forma equitativa entre los distintos servidores.
- El sistema debería poder seguir funcionando si un servidor se cae.
- El sistema debería poder mover trabajo de unos servidores a otros cuando fuera necesario sin que los usuarios finales perdieran la conectividad. Esto es necesario para que los servidores puedan ser apagados o reiniciados por motivos de mantenimiento.

Si bien la dificultad es alta, éste podría ser un trabajo derivado interesante.

Bibliografía

Referencia técnica

- Documentación de Python
<https://docs.python.org/3/>
- Documentación de Tornado
<http://www.tornadoweb.org/en/stable/>
- Documentación de AngularJS
<https://docs.angularjs.org/guide>
- Documentación de Jasmine
<https://jasmine.github.io/2.0/introduction.html>
- Documentación de Django
<https://docs.djangoproject.com/en/>
- Documentación de Django REST Framework
<http://www.django-rest-framework.org/>
- Documentación de Flask
<http://flask.pocoo.org/docs/0.10/>

Libros

- Mastering Web Application Development with AngularJS⁵⁴

Peter Bacon Darwin, Pawel Kozlowski. August 2013, Packt Publishing.

⁵⁴ <https://www.packtpub.com/web-development/mastering-web-application-development-angularjs>