Escola Tècnica Superior d'Enginyeria

Electrònica i Informàtica La Salle

Trabajo final de Master

Master Universitario en Ingeniería de Telecomunicaciones

DESARROLLO DE UNA APLICACIÓN WEBRTC

Alumno

Alex Fabian Oviedo Tinoco

Profesor Ponente

David Vernet

ACTA DEL EXAMEN

DEL TRABAJO FINAL DE MASTER

Reunido el Tribunal calificador en el día de la fecha, el alumno

D. Alex Fabian Oviedo Tinoco

Expuso el Trabajo final de Master, el cual trató sobre el tema siguiente:

Desarrollo de una aplicación WebRTC

Terminada la exposición y contestadas por parte del alumno las objeciones formuladas por los Srs. Miembros del tribunal, esta valoró el mencionado Trabajo con una calificación de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENTE DEL TRIBUNAL

Abstract

El objetivo de este proyecto es el estudio y desarrollo de una aplicación multiplataforma para comunicaciones en tiempo real con el fin de ofrecer servicios de videoconferencia, mensajería instantánea e intercambio de información a través de una pizarra virtual, utilizando una de las nuevas tecnologías de este sector llamada WebRTC. Dicha plataforma puede ser de mucha utilidad como soporte en las tutorías online y/o para aquellos centros que desarrollen sus funciones de forma remota. Los resultados han sido favorables, por una parte el componente de videoconferencia presenta una alta respuesta en la comunicación, así mismo los componentes de mensajería instantánea y pizarra virtual. Como conclusiones, ha sido un proyecto muy extenso tanto a nivel teórico como práctico debido a que se han aplicado diversas tecnologías las cuales hacen que el proyecto haya sido de alta motivación personal y profesional, con un resultado final que cumple con las expectativas planteadas, dejando también preparado el sistema para futuras mejoras o implementaciones.

Palabras clave: WebRTC, videoconferencia, tiempo real, aplicaciones multiplataforma, programación funcional.

Abstract

L'objectiu d'aquest projecte és l'estudi i desenvolupament d'una aplicació multiplataforma per a comunicacions en temps real per tal d'oferir serveis de videoconferència, missatgeria instantània i intercanvi d'informació a través d'una pissarra virtual, utilitzant una de les noves tecnologies d'aquest sector anomenada WebRTC. Aquesta plataforma pot ser de molta utilitat com a suport a les tutories en línia i / o per a aquells centres que desenvolupin les seves funcions de forma remota. Els resultats han estat favorables, d'una banda el component de videoconferència presenta una alta resposta en la comunicació, així mateix els components de missatgeria instantània i pissarra virtual. Com a conclusions, ha estat un projecte molt extens tant a nivell teòric com pràctic a causa que s'han aplicat diverses tecnologies les quals fan que el projecte hagi estat d'alta motivació personal i professional, amb un resultat final que compleix amb les expectatives plantejades, deixant també preparat el sistema per a futures millores o implementacions.

Paraules clau: WebRTC, videoconferència, temps real, aplicacions multiplataforma, programació funcional.

Abstract

The main goal of this project is the study and development of a multiplatform application for real-time communications in order to offer videoconference, instant messaging and a virtual whiteboard, using a new technology called WebRTC. This platform can be very useful as support for online classes and / or for those centers that perform their functions remotely. The results have been favorable, on the one hand the videoconference component presents a high response in the communication, as well instant messaging and virtual whiteboard components. As conclusions, it has been a very extensive project both at a theoretical and practical, because it contains various technologies which make the project has been highly personal and profesional motivation, with a final result that achieved the expectations raised, also the system is ready for future improvements or implementations.

Keywords: WebRTC, videoconferencing, real time, cross-platform applications, functional programming.

Contenido

Acrónimos

AES: *Advanced Encryption Standard.*

API : *Application Programming Interface.*

AWS: *Amazon Web Services.*

CPU: *central processing unit*

CSS: *Cascading Style Sheets.*

DB: Base de Datos*.*

DNS: *Domain Name System.*

DOM: *Document Object Model.*

EC2: *Elastic Compute Cloud.*

HTML: *HiperText Murkup Lenguage.*

HTTP: *HiperText Transport Protoco*l.

ICE: *Interactive Connectivity Establishment.*

IMEI: *International Mobile Station Equipment Identity.*

IP: *Internet Protocol*.

JSEP: *JavaScript Session Establishment Protocol*.

JSON: *JavaScript Object Notation.*

JWT: *Json Web Token.*

MAC: *Media Access Control*.

NAT: *Network Address Translation*.

NPM: *Node Package Manager.*

P2P: *Peer to Peer*.

PBX: *Private Branch Exchange.*

RDBMS: *Relational Database Management System***.**

RDS: *Relational Database Service*.

REST*: Representactional State Transfer.*

RTCP: *Real-time Transport Control Protocol.*

SDK: *Software Development Kit*.

SDP: *Session Description Protocol.*

SQL: *Structured Query Language*.

SSL*: Secure Sockets Layer*.

TCP: *Transmission Control Protocol*.

UDP: *User Datagram Protocol*

URL: *Uniform Resource Locator.*

UX: *User Experience.*

VoIP: *Voice over IP.*

WebRTC: *Web Real Time Comunication*

1 Introducción

Actualmente las comunicaciones en tiempo real han crecido considerablemente debido al aumento de dispositivos móviles con altas capacidades y por la alta demanda de plataformas, capaces de comunicar a los usuarios con este sistema, podemos destacar algunos casos de éxito como Uber, Glovo y Messenger.

Entre estos avances es necesario comentar la irrupción de proyectos de código abierto como es el caso de WebRTC, que presenta grandes capacidades para poder conseguir una alta calidad, escalabilidad y seguridad en aplicaciones de este tipo.

El siguiente proyecto plantea el estudio y desarrollo de una aplicación multiplataforma para ofrecer servicios de videoconferencia, mensajería instantánea y una pizarra virtual, aplicando una nueva tecnología llamada WebRTC, además el proyecto debe ser altamente escalable por lo que se utilizan varias tecnologías para conseguir dichos objetivos.

Este proyecto se enfoca especialmente en el campo docente, debido a que se ha detectado una falta de plataformas que ofrezcan este tipo de servicios o las existentes no son adecuadas para este fin.

En la mayoría de los casos para poder realizar alguna tutoría o clase virtual, se utilizan herramientas como Skype, la cual necesita una instalación y configuración específica para su funcionamiento, algunas otras necesitan de algún *plugin* extra como es el caso de BigBlueButton.

Así mismo, existen proyectos a través de la tecnología SIP, que al funcionar sobre internet reducen los costes que presenta la telefonía tradicional, sin embargo, el gran problema es la necesidad de un software y equipo especializado.

WebRTC soluciona todos estos problemas, debido a que la comunicación se establece directamente entre los navegadores, dispositivos móviles e incluso entre dispositivos IoT sin necesidad de instalar ningún software o equipo especial. De entrada ya es una gran ventaja respecto a las tecnologías SIP o Skype.

Para empezar, se comentará de forma general la evolución las comunicaciones y los diferentes tipos de plataformas que han resaltado en los últimos años, seguidamente se explicará la propuesta presentada denominada **Shardu**, se explicará la metodología que se ha seguido durante el proyecto, la arquitectura y el desarrollo, donde se destacarán los diferentes tipos de tecnologías utilizadas para finalmente terminar con el despliegue, escalabilidad y los resultados obtenidos.

2 Estado del arte

Las comunicaciones han estado en constante evolución a lo largo de los años, comenzando por el teléfono tradicional, la tecnología ha ido creciendo hasta conseguir cambiar completamente el sistema de comunicación con la aparición de tecnologías como VoIP, la cual transmite la Voz sobre IP, donde se puede destacar el protocolo **SIP**.

SIP es la tecnología más utilizada por los proveedores de telefonía tradicional, añadiendo muchas ventajas y funcionalidades a los sistemas de comunicación actuales, como por ejemplo la reducción de costes. Sin embargo un gran problema es la necesidad de comprar software y equipo especializado, por lo que es necesaria una gran inversión inicial.

Recientemente ha surgido una nueva tecnología capaz de solventar todos estos problemas debido a que la inversión inicial puede llegar a ser nula y no se necesita instalar algún software especial para poder consumir sus proyectos, esta tecnología se denomina **WebRTC**.

WebRTC está ganando terreno rápidamente y se propone revolucionar los estándares de comunicaciones, por ese motivo ha sido de gran interés para llevar a cabo este proyecto.

El objetivo principal será utilizar la tecnología WebRTC, junto con otras más, para crear una aplicación multiplataforma en la cual se pueda realizar videoconferencias, mensajería instantánea y compartir información mediante una pizarra virtual.

Tras un exhaustivo análisis se ha detectado la falta de plataformas que brinden estos servicios en el campo docente, por esa razón este proyecto se va a enfocar a dicho campo, proporcionando un recurso extra a los docentes, alumnos y plataformas *E-learning* actuales.

Evidentemente a estas alturas existen plataformas similares a la que se ha propuesto, pero presentan algunas desventajas las cuales se comentarán a continuación.

2.1 Asterisk

SIP es una las tecnologías principales en las comunicaciones de voz sobre IP, surgiendo así proyectos como Asterisk, un programa de código abierto que proporciona funcionalidades de una central telefónica PBX, soportando muchos protocolos VoIP, tales como SIP, el funcionamiento básico de Asterisk se lo puede observar en la Imagen 1.

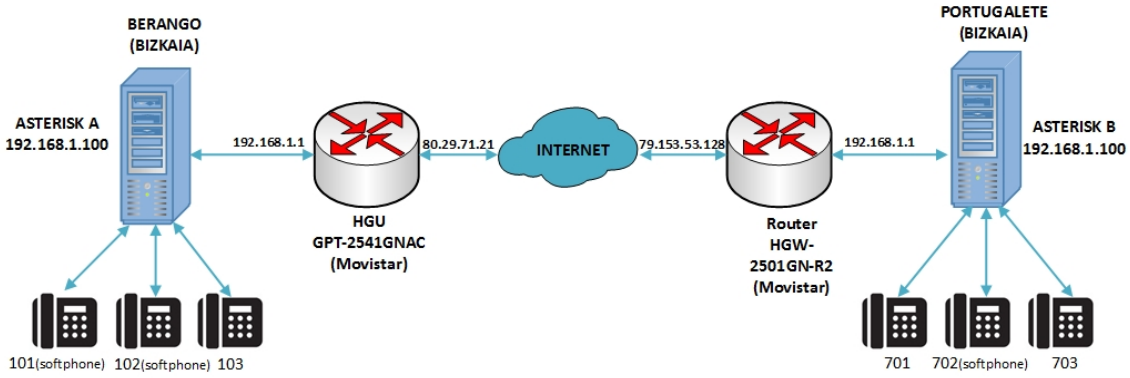

Imagen 1. Funcionamiento de Asterisk

Asterisk necesita de una compleja instalación y requiere de equipos especiales para poner en marcha su funcionamiento, aunque sea una mejora respecto a la telefonía tradicional, no es una las principales opciones a la hora de realizar un proyecto de comunicación a través de IP.

2.2 WebRTC

WebRTC es un proyecto de código abierto que está siendo elaborado por la W3C, y permite a las aplicaciones del navegador realizar llamadas de voz, chat de video y uso compartido de archivos P2P, sin la necesidad de instalar plugins, se puede observar un ejemplo básico de comunicación en la Imagen 2.

Imagen 2. Funcionamiento básico de WebRTC.

Algunas de las ventajas respecto a SIP que se puede destacar son las siguientes:

- Mayor flexibilidad, debido a que puede ser ejecutada en múltiples dispositivos sin depender de alguno en especial.

- Sin inversión, no se necesita comprar un equipo adicional y tampoco es necesaria la instalación de un software.
- Alta calidad.

Actualmente es soportado por Google Chrome, Firefox, Opera, Android, Safari, iOS.

2.3 Plataformas actuales

Las plataformas actuales reducen mucho la complejidad y los costes que se puedan generar con proyectos como Asterisk, a continuación se hablará de las más destacadas.

2.3.1 Skype

Skype es un *software* propietario distribuido por Microsoft tras haber comprado la compañía. Permite comunicaciones de texto, voz y vídeo sobre Internet, opera con base en el modelo P2P y una de sus principales ventajas respecto a SIP, es que solamente se necesita descargar e instalar el *software* en el ordenador, reduciendo así mucha complejidad al usuario final, es una aplicación multiplataforma como se puede observar en la Imagen 3.

Imagen 3. Skype multiplataforma.

Pero para el enfoque del proyecto, sigue siendo un problema, debido a que los alumnos y profesores tienen que compartir cuentas personales para poder comunicarse, por otra parte se añade la necesidad de instalar la aplicación en cada máquina.

2.3.2 Bigbluebutton

BigBlueButton es una aplicación web open source para videoconferencia, plataformas *E-Learning* o educación a distancia. Es un programa distribuido bajo licencia GNU y que ha surgido de la reutilización de proyectos varios como Asterisk, Flex SDK, Red5, MySQL y otros.

Es una plataforma similar a la que se ha propuesto, con la diferencia de que la integración de VoIP se hace mediante freeSWITCH, un proyecto que surge con la idea de tener una mejor plataforma que Asterisk, a continuación se puede observar el funcionamiento de la plataforma en la Imagen 4.

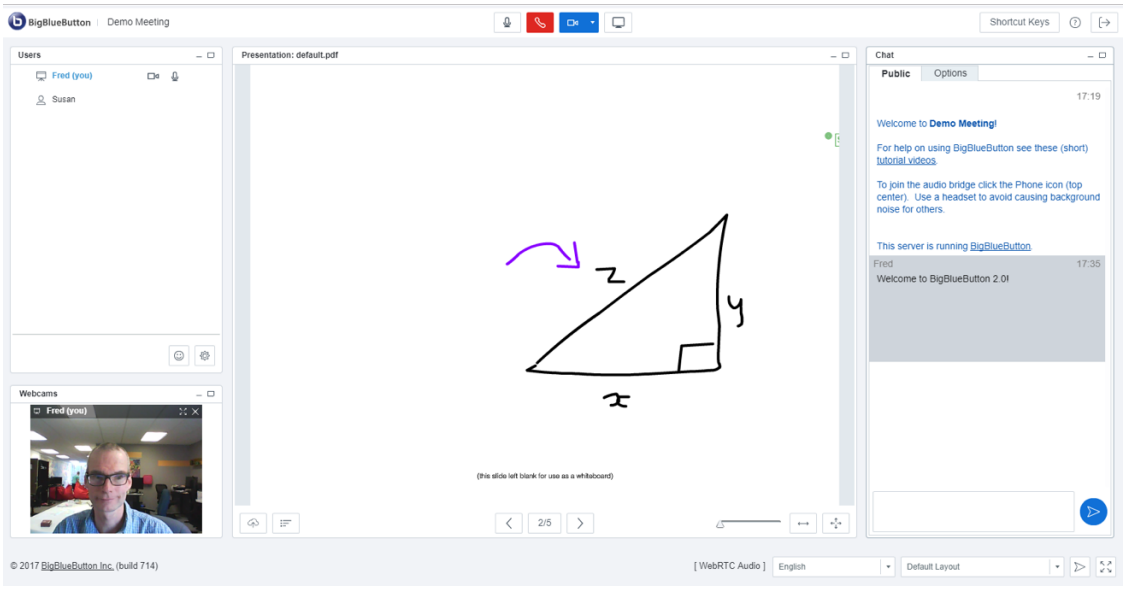

Imagen 4. Funcionamiento de Bigbluebutton.

La principal desventaja que presenta es que el cliente necesita instalar un plugin en el navegador para la plataforma de Adobe flash.

2.3.3 Classgap

Classgap es una plataforma *online* en la que se pueden realizar clases particulares de forma remota, las características principales son videoconferencia, mensajería instantánea y una pizarra virtual, es sin duda el proyecto más parecido al propuesto debido a que presentan las misma características, además de que ambas plataformas están desarrolladas sobre WebRTC, el funcionamiento de Classgap se puede observar en la Imagen 5.

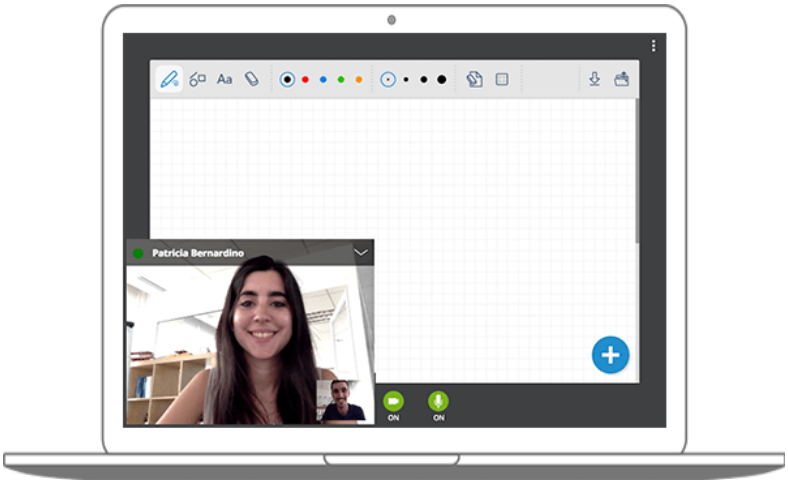

Imagen 5. Funcionamiento de Classgap.

La principal diferencia de Classgap respecto al proyecto propuesto, es que utiliza SimpleWebRTC, una plataforma que implementa las API's de WebRTC para ofrecer a las empresas una integración sencilla pero con ciertas limitaciones, el coste de las prestaciones de esta puede llegar a alcanzar hasta los 900€ por mes.

3 Shardu

La propuesta del proyecto se denomina **Shardu** *(**Sha**re e**du**cation),* una aplicación multiplataforma enfocada al campo docente, donde los usuarios comparten un espacio para poder enviar y recibir mensajes, realizar videoconferencias y utilizar una pizarra virtual, tal y como se puede observar en la Imagen 6.

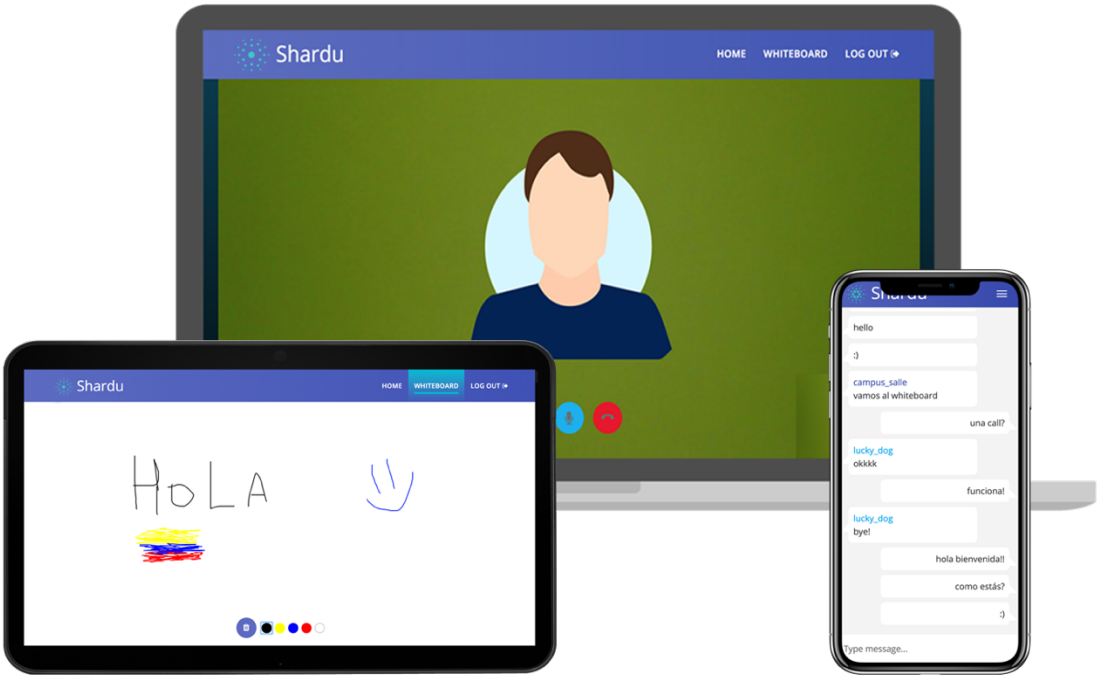

Imagen 6. Shardu App.

La idea surge a través de un análisis sobre aplicaciones en tiempo real, donde se ha visto un aumento considerable y gran interés por parte de las empresas, se pueden destacar casos de éxito como Cabify, Uber y Messenger, que están constantemente enviando y recibiendo datos en tiempo real con los usuarios.

La característica principal de este proyecto es la comunicación en tiempo real, por ese motivo tras una búsqueda exhaustiva se decidió utilizar **WebRTC** como una de las tecnologías base, debido a las múltiples ventajas que presenta respecto a tecnologías similares como SIP. El proyecto se enfoca principalmente en el campo docente, se encontraron plataformas similares pero con algunas desventajas que se intentarán solventar.

Las principales ventajas de **Shardu** respecto a las similares son:

- No se requiere de un equipo o software especial debido a que la aplicación está desarrollada para ser ejecutada en los navegadores de cualquier dispositivo móvil o escritorio.
- WebRTC de forma nativa, es decir no se utilizan servicios externos como SimpleWebRTC u otros, teniendo así mayor flexibilidad para futuras mejoras o implementaciones, ahorrando en costes que puedan generar dichos servicios.

4 Resumen comparativo

En la Tabla 1 se puede observar un resumen comparativo, el cual muestra claramente las principales ventajas respecto a las plataformas existentes.

Tabla 1. Comparativa de plataformas existentes

Plataformas	Sin Instalación/Plugins	Sin servicios externos	WebRTC
BigBlueButton	NO	NO	NO
classgap	SI	NO	SI
Shardu	SI	SI	SI
skype	NO	SI	NO

Pero Shardu no solo implica nuevas funcionalidades respecto a las plataformas existentes, también presenta un *Stack* de nuevas tecnologías capaces de hacer un sistema completo y altamente escalable.

5 Metodología

Una vez establecida la propuesta y los objetivos del proyecto, es indispensable saber cómo se ha llevado a cabo y las herramientas que han ayudado a cumplir los objetivos.

La primera fase se basa en un periodo de búsqueda y aprendizaje sobre el tema base a tratar, analizando todas las restricciones que se puedan tener para poder encaminar el proyecto de forma correcta.

La segunda fase consiste en crear un diseño, definiendo claramente las directrices a seguir, los elementos que se van a incluir y el orden para ser implementados, esta fase es muy importante debido a que es recomendable iterar en el diseño a que hacerlo directamente en el desarrollo, de esta forma se consigue cumplir con los tiempos propuestos.

La tercera fase consiste en el desarrollo del producto mínimo viable, el cual debe cumplir con todos los objetivos definidos en la fase de diseño, este producto debe funcionar correctamente, dejándolo dispuesto a futuras mejoras e implementaciones.

Algunas de las herramientas que han sido de gran utilidad para cumplir con todas estas fases han sido las siguientes:

- Diseño: Sketch + InVision.
- Desarrollo: Repositorios privados, Atom y GIT.

Es necesario destacar que se ha utilizado GitFlow [1] en todo el proceso de desarrollo, esta herramienta se basa en el modelo de ramificaciones que facilita el trabajo en el desarrollo de nuevas *features*, sin modificar una versión ya estable, se puede observar cómo funciona en la imagen 7.

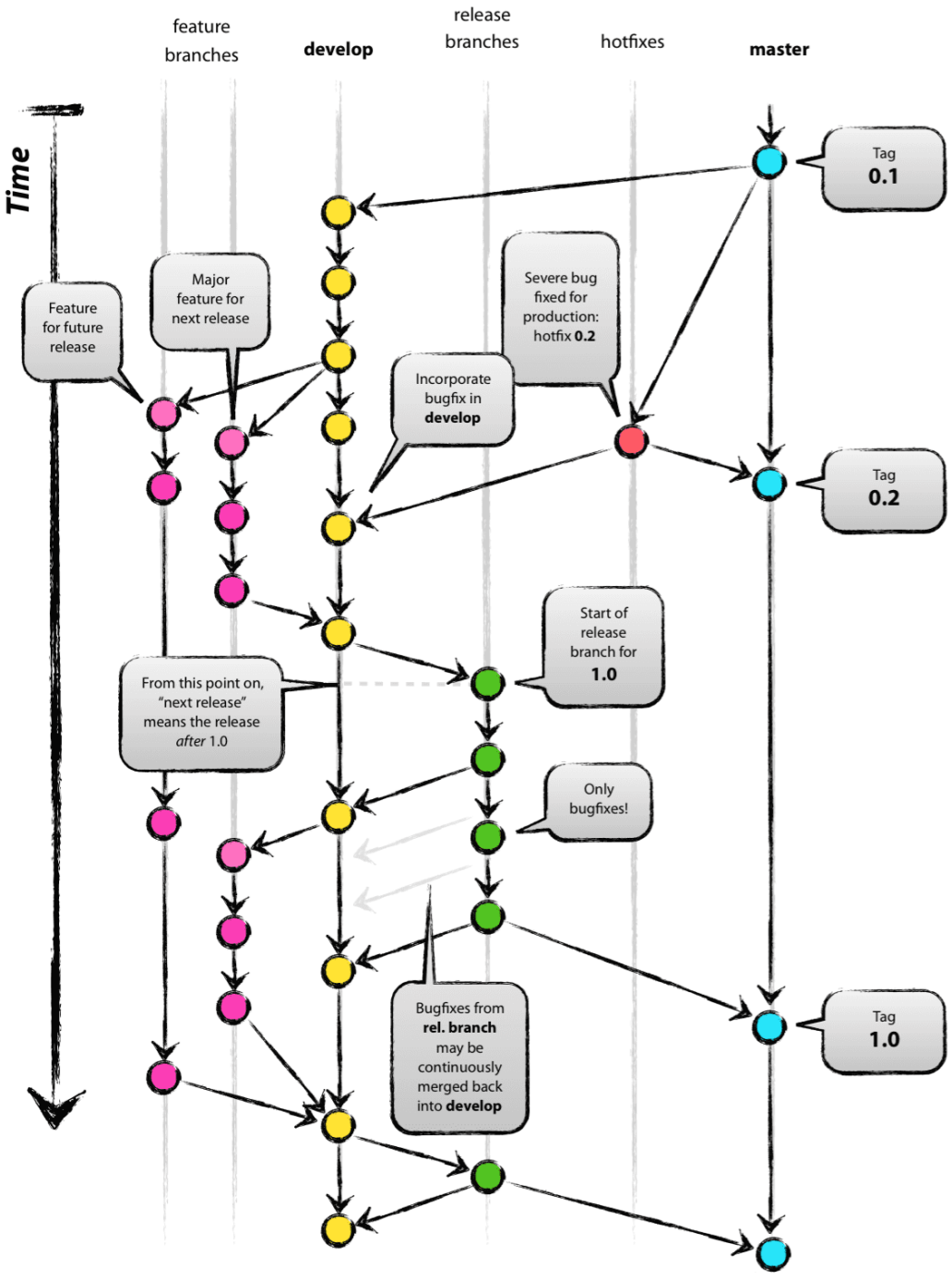

Imagen 7. Funcionamiento de GitFlow.

6 Tecnologías utilizadas

Detrás de la aplicación existen varias tecnologías y métodos que han hecho posible que el proyecto se encamine de forma correcta, dejando un sistema estable y altamente escalable para poder seguir mejorándolo o aplicando nuevas funcionalidades, a continuación se explica lo más destacado del proyecto.

6.1 JavaScript

JavaScript es un lenguaje de programación que permite realizar actividades complejas en una página web, como por ejemplo permitir al usuario interactuar con la página.

Es un lenguaje de programación interpretado por lo que:

- No es necesario compilar los programas para ejecutarlos.
- Los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

Proviene de un dialecto del estándar ECMAScript, es necesario destacar que Java y JavaScript no están relacionados, tienen semánticas y propósitos diferentes.

Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en el lado del cliente en servicios Web, implementado como parte de un navegador web permitiendo crear interacción con el usuario y páginas web dinámicas, aunque actualmente es posible ejecutar JavaScript en el servidor mediante Node.JS.

En este proyecto JavaScript es un lenguaje de programación fundamental, debido a que se utilizan múltiples librerías desarrolladas sobre este lenguaje, tanto para el cliente como para el servidor.

6.2 React.js

React.js es la tecnología clave para desarrollar los distintos módulos de la aplicación y de manera general, para estructurar todo lo relacionado con la parte del Frontend.

Es una librería JavaScript de código abierto para facilitar la creación de componentes interactivos, reutilizables, para interfaces de usuario. Es mantenido por Facebook, la comunidad de desarrolladores y varias compañías.

Uno de sus puntos más destacados, es que no sólo se utiliza en el lado del cliente, sino que también se puede representar en el servidor, y trabajar ambos de manera conjunta.

React.js, intenta facilitar el trabajo a los desarrolladores al momento de construir aplicaciones con datos que están en constante cambio. Su objetivo es ser sencillo, declarativo y fácil de combinar, su gran utilidad es manipular la interfaz de usuario en una aplicación.

A diferencia de otros *frameworks* como Angular.js, que tiene que renderizar el DOM a cada cambio, React.js mantiene un virtual DOM propio, en lugar de confiar solamente en el DOM del navegador. Esto deja a la biblioteca determinar qué partes del DOM han cambiado comparando contenidos entre la versión nueva y la almacenada en el virtual DOM, y utilizando el resultado para determinar cómo actualizar eficientemente el DOM del navegador, consiguiendo mayor rapidez a la hora de hacer los cambios debido a que no tiene cambiar el DOM completamente, se puede observar el funcionamiento en la imagen 8.

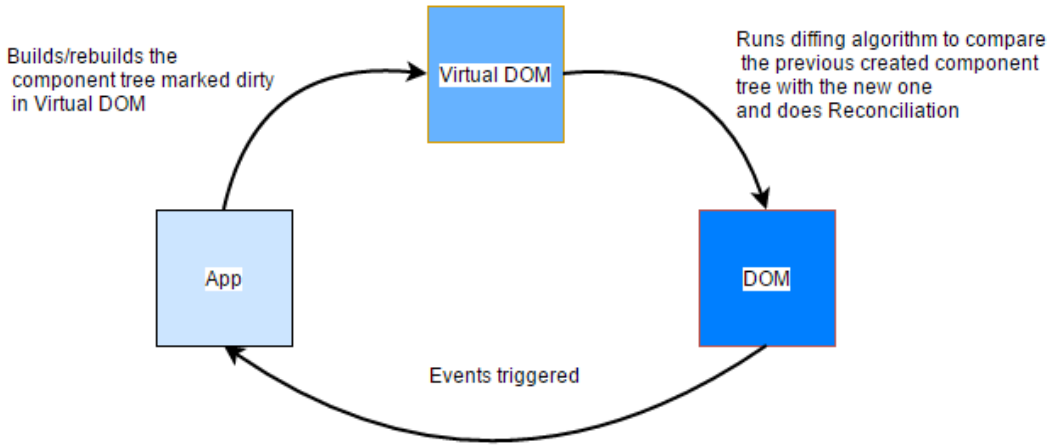

Imagen 8. Funcionamiento de React.js.

6.3 Redux

Para gestionar el estado de la aplicación se ha utilizado Redux, una librería escrita en JavaScript que se encarga de desacoplar en una aplicación el estado y la parte visual.

Redux está en gran parte influenciado por la arquitectura Flux propuesta por Facebook, está muy pensando para React.js pero también es posible utilizarla con Angular.js, Backbone.js, entre otras.

El estado de esta la aplicación se va a definir a través de los datos recibidos de la API, como por ejemplo datos del usuario o *token* de inicio de sesión, además del estado de la UI en un determinado momento.

Sin Redux, sería difícil de gestionar el estado de la aplicación, debido a que si un componente hace un cambio y este cambio afecta a otros, el estado tendría que pasar como parámetro por cada uno de los componentes hasta llegar al destino, a efectos prácticos no es del todo eficiente debido a la complejidad que se genera si la aplicación pasa a ser grande.

Con Redux, solucionamos todos estos problemas debido a que todos los componentes pasan a consumir los datos de un "*store*", se puede observar un ejemplo de ambos casos en la Imagen 9.

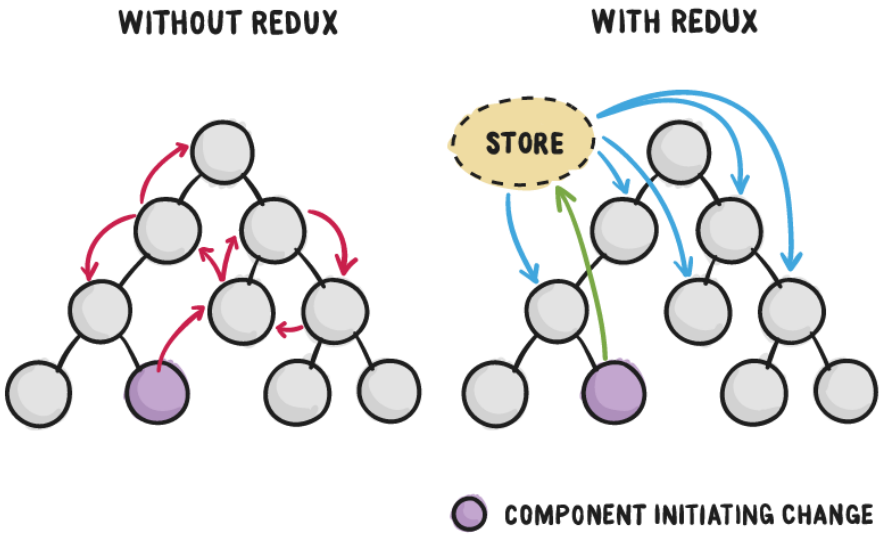

Imagen 9. Diferencia entre usar o no Redux.

Los conceptos clave de Redux son los siguientes:

- Una sola fuente de la verdad, es decir todo el estado de la aplicación está contenida en un único *store*.
- El estado es de solo lectura, no se puede modificar el estado directamente, solo se puede leer para representarlo en la vista y se quiere modificar algo del estado, se tiene que hacer a través de acciones.
- Los cambios se hacen mediante **funciones puras**, para gestionar los cambios del estado se utilizan *reducers*, que son funciones puras que reciben dos parámetros, el estado actual y la acción a realizar y devuelven un nuevo estado, sin modificar el estado actual.

Se puede observar el funcionamiento de Redux en la imagen 10.

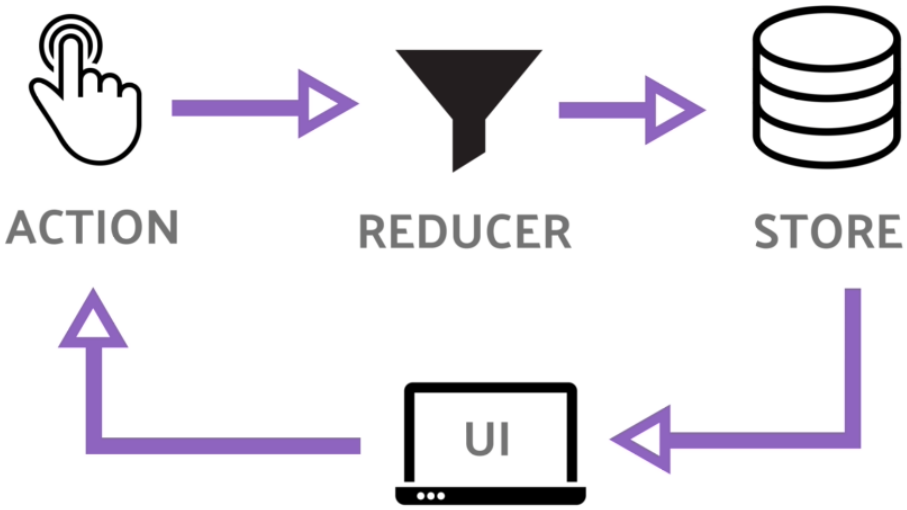

Imagen 10. Funcionamiento de Redux.

6.4 Ramda.js

Se ha decidido desarrollar este proyecto utilizando la **programación funcional** que es un paradigma de la programación declarativa, de esta forma el código escrito es más robusto y fácil de mantener.

A diferencia de otros paradigmas como la **programación imperativa,** que se basa en crear algoritmos que describen los pasos necesarios para solucionar un problema, en la programación declarativa las sentencias que se utilizan describen el problema que se quiere solucionar.

El proyecto se basará en el paradigma declarativo pero utilizando la programación funcional. Para ello se aprovechará una librería muy potente llamada **Ramda.js**.

Ramda.js contiene múltiples funciones y una documentación detallando claramente cada una de ellas.

Para ver la diferencia de Ramda.js respecto a lo que se suele hacer con la programación declarativa, se va a realizar algunos ejemplos prácticos. En la Imagen 11 se puede observar un ejemplo de programación declarativa.

```
let hasRed = false
let colors = [ "blue", "red", "black" ]

for(let i; i <= colors.length; i++) {
    if( i === "red" ) {
        hasRed = true
        return
    }
}

console.log( 'Existe el color rojo?', hasRed )
```

Imagen 11. Programación Declarativa.

Esta forma es un poco verbosa y tenemos que depender de una variable para saber si hemos encontrado el color dentro de la lista.

Con Ramda.js se puede programar lo mismo de la siguiente manera, tal y como se puede observar en la Imagen 12.

```
let colors = [ "blue", "red", "black" ]
console.log( 'Existe el color rojo?', R.contains( 'red', colors ) );
```

Imagen 12. Programación con Ramda.js

Como se puede observar, esta solución es más clara, concisa y fácil de entender por cualquier desarrollador.

6.5 WebRTC

La tecnología principal para realizar una videoconferencia es WebRTC, un proyecto de código abierto que actualmente está siendo estandarizado por la W3C, que permite a las aplicaciones del navegador realizar llamadas de voz, chat de video e intercambiar datos sin la necesidad de *plugins* externos, es decir directamente entre los navegadores gracias al conjunto de estándares que presenta dicha tecnología.

Es un proyecto en progreso que tiene implementaciones avanzadas principalmente en Google Chrome y Firefox. La API se basa en el trabajo previo realizado en la WHATWG.

Actualmente es soportado por los siguientes navegadores y plataformas:

- Google Chrome.
- Opera.
- Mozilla.
- Android.
- Safari.
- Microsoft Edge.
- iOS.

WebRTC está ganando terreno rápidamente y se propone revolucionar los estándares de comunicaciones.

6.5.1 Arquitectura

La arquitectura de WebRTC se compone de tres partes principales, tal y como se puede observar en la Imagen 12.

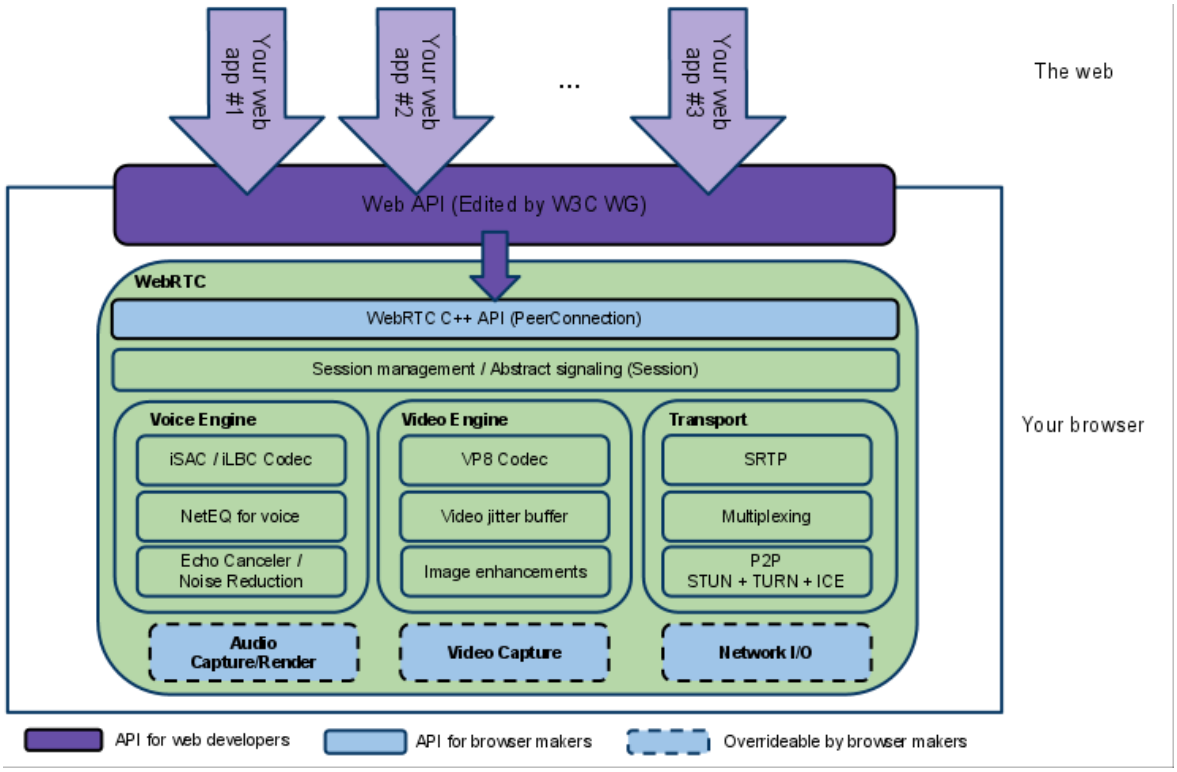

Imagen 12. Arquitectura de WebRTC.

- *API for web developers*, API para los desarrolladores web que deseen crear aplicación con WebRTC.
- *API for browser makers*, API para los desarrolladores de navegadores.
- *Overrideale by browser makers*, software que puede ser reemplazado por los desarrolladores de navegadores.

6.5.2 API

WebRTC está compuesto principalmente de tres API's que se encuentran implementadas en JavaScript, las cuales son:

- **getUserMedia**, permite obtener la cámara y micrófono del navegador.
- **RTCPeerConnection**, establece las llamadas de audio y video entre *peers*, realiza el procesado de señal, gestiona los códecs, la comunicación P2P, seguridad y el gestión del ancho de banda.
- **RTCDataChannel**, permite la comunicación bidireccional de datos arbitrarios entre *peers*.

6.5.2.1 getUserMedia (MediaStream)

Este método pide permiso al usuario para usar los dispositivos de vídeo o audio del navegador, si el usuario concede el permiso, se resuelve la promesa devolviendo el *stream* obtenido, en caso contrario la promesa devuelve error, tal y como se puede observar en la Imagen 13.

```
let mediaConstraints = {
    video: {
        facingMode: 'user',
        height: { min: 360, ideal: 720, max: 1080 }
    },
    audio: true
};

return (
    navigator.mediaDevices
    .getUserMedia( mediaConstraints )
    .then( localMediaStream => {

        this.setState( { callWindow: 'active', localSrc: localMediaStream } )
        return this.pc.addStream( localMediaStream )
    })
    .catch( err => {
        console.log('generateLocalMediaStream ');

        console.log('err', err)
    })
)
```

Imagen 13. Obtener MediaStreams.

Además, existen unas restricciones que se las puede pasar como parámetro al método getUserMedia(), tales como la resolución o el tipo de media que se quiere obtener.

6.5.2.2 RTCPeerConnection

RTCPeerConnection representa una conexión WebRTC entre el equipo local y remoto, proporciona varios métodos para conectar un equipo remoto, mantener y monitorear la conexión y cerrarla una vez se termine.

El constructor del objeto RTCPeerConnection se inicializa de la siguiente manera:

```
let rtcPeerConf = {
    iceServers: [
        { 'url'           : 'stun:stun.l.google.com:19302' },
        { 'url'           : 'stun:stun.services.mozilla.com' }
    ]
};

this.pc                 = new RTCPeerConnection( rtcPeerConf )
```

Imagen 14. Constructor RTCPeerConnection.

Como se puede observar en la Imagen 14, se le pasa un parámetro de configuración que hace referencia a los servidores ICE.

Los métodos principales son los siguientes [6]:

- **addIceCandidate(),** añade los ICE *candidates*.
- **getLocalStreams(),** devuelve un *array* de MediaStream asociados a la conexión local.
- **getRemoteStreams(),** devuelve un *array* de MediaStream asociados a la conexión remota.
- **getStreamById()**, devuelve el MediaStream asociado al Id dado, ya sea local o remoto, si no existe devuelve null.
- **removeTrack()**, elimina un MediaStream local de audio o video.
- **createDataChannel()**, crea un nuevo canal sobre el cual se puede transmitir cualquier clase de datos.
- **addStream(),** añade un objeto MediaStream como recurso local de audio o video. El recurso que se añade puede ser el *stream* local o remoto.
- **createOffer(),** inicializa la creación de una oferta SDP con el objetivo de inicializar una nueva conexión WebRTC a un *peer* remoto.
- **createAnswer(),** crea una respuesta SDP a una oferta recibida desde un *peer* remoto durante la negociación de una conexión WebRTC.
- **setLocalDescription(),** especifica las propiedades de la conexión local como poder ser el códec, recibe como parámetro el objeto RTCSessionDescription.
- **setRemoteDescription(),** especifica las propiedades de la conexión remota como poder ser el códec, recibe como parámetro el objeto RTCSessionDescription.
- **close(),** cierra la actual conexión.

Por otro lado los manejadores de eventos son los siguientes:

- **onaddstream**, este evento es llamado cuando se recibe el evento addStream, dicho evento es enviado cuando el MediaStream es añadido a la conexión por el *peer* remoto, este evento se envía inmediatamente después de llamar al setRemoteDescription(), y no espera por el resultado de la negociación SDP.
- **onicecandidate,** este evento es llamado cuando se recibe un evento icecandidate, y sucede cada vez que el agente local ICE necesita entregar un mensaje al otro *peer* a través del servidor de señalización.
- **ondatachannel,** este evento es llamado cuando se recibe el evento datachannel.
- **onicecandidatestateeschange,** este evento es llamada cuando cambia el estado de la conexión del agente ICE.
- **onnegotiationneeded,** este evento es llamado cuando se produce un cambio que requiere una negociación se sesión.
- **onremovestream,** este evento es llamado cuando el objeto MediaStream es eliminado de la conexión.
- **onsignalstatechange,** este evento es llamado cuando se recibe el evento signalingstatechange.

6.5.2.3 RTCDataChannel

RTCDataChannel representa un canal de red que puede utilizarse para la transferencia bidireccional de datos. Cada canal es asociado a una RTCPeerConnection, este *peer* puede tener como máximo 65534 canales de datos.

Para crear un canal de datos y pedirle a un *peer* remoto que se una a dicho canal, se necesita llamar al método createDataChannel() de RTCPeerConnection. El *peer* que se invita a intercambiar datos recibe un evento llamado datachannel para informarle que el canal de datos ha sido agregado a la conexión.

6.5.3 Señalización

Es necesario un mecanismo de señalización por el cual los *peers* se envían mensajes de control entre ellos, con el propósito de establecer el protocolo, canal y método de comunicación. WebRTC no define los protocolos o métodos que se tienen que utilizar, por esa razón el desarrollador es libre de utilizar el que desea [3].

La señalización es usada para intercambiar tres tipos de información:

- Mensajes de control de sesión, para inicializar o cerrar la comunicación e informar errores.
- Información de red, para saber la IP y puerto.
- Formato de los medios, para saber que códecs y resoluciones puede manejar el navegador local y remoto.

La adquisición e intercambio de información de la red y el formato de los medios, puede realizarse de forma simultánea, pero ambos procesos deben haberse completado antes de comenzar la transmisión de audio y video entre *peers*.

La arquitectura *Offer/Answer* que se realiza en WebRTC se llama JSEP y se puede observar en la Imagen 15.

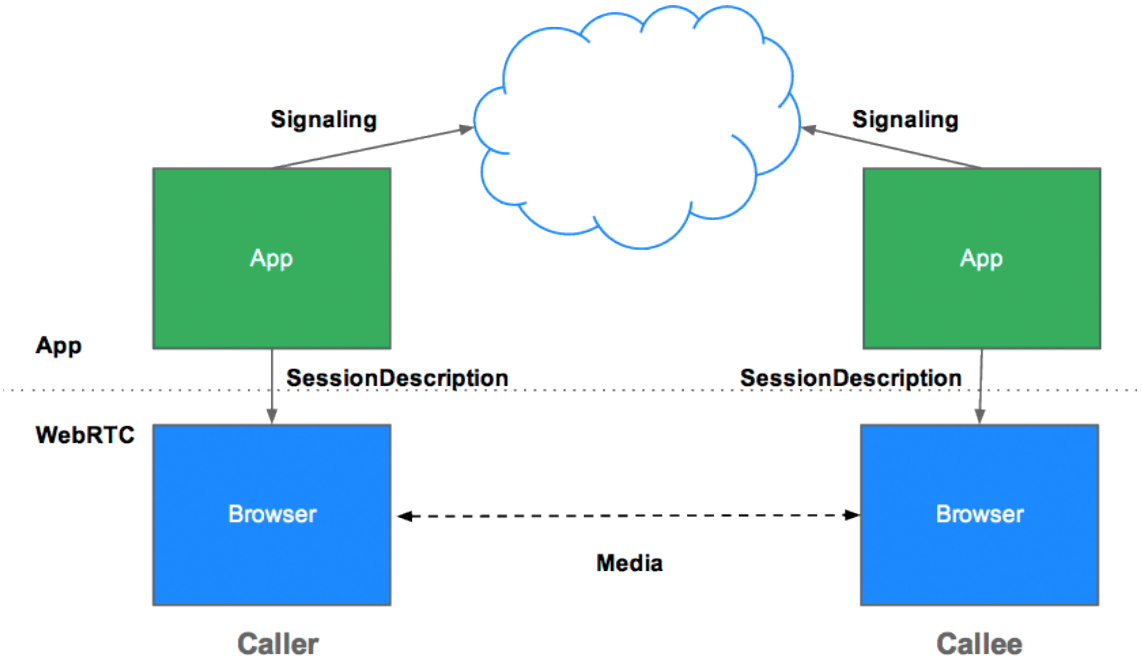

Imagen 15. Arquitectura JSEP.

Una vez que el proceso de señalización se ha completado con éxito, los datos se pueden transmitir directamente P2P, es decir entre el que realiza la llamada y el que la recibe. El encargado de transmitir estos datos es RTCPeerConnection.

6.5.4 ICE Framework

El Framework ICE permite que se prueben distinta rutas para comunicar dos terminales ente sí acordando una en común.

En un mundo ideal, cada cliente WebRTC tiene una dirección única que puede intercambiar con otros *peers* para realizar una comunicación directa, tal y como se puede observar en la Imagen 16.

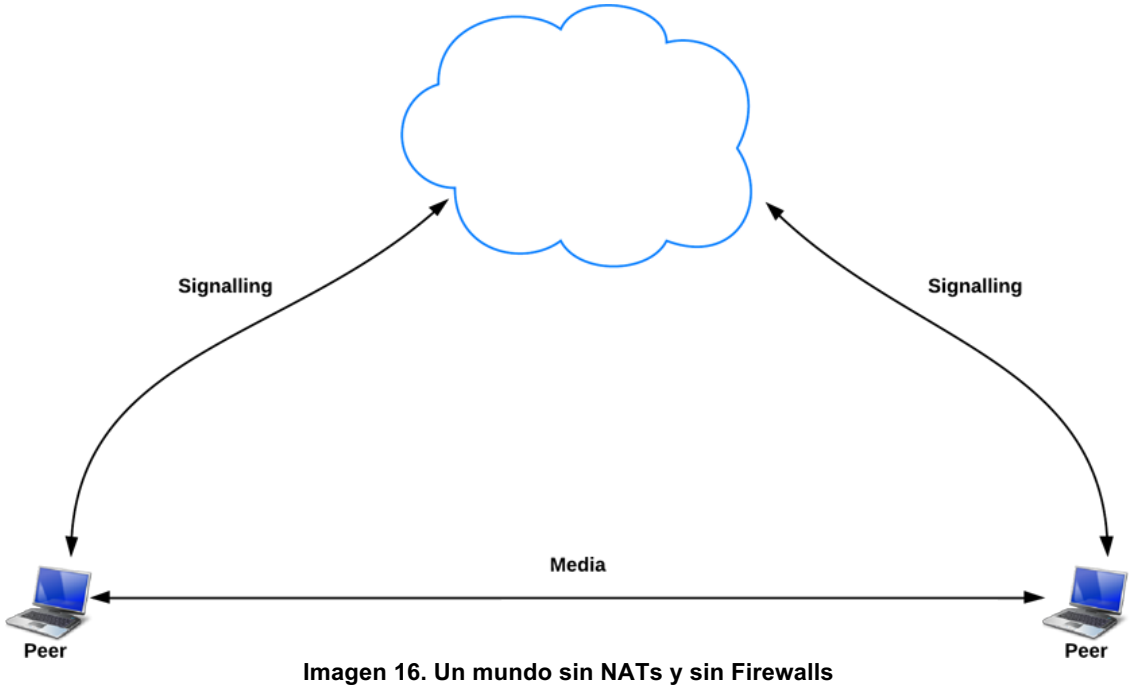

Imagen 16. Un mundo sin NATs y sin Firewalls

Pero el mundo real es diferente, muchos de los dispositivos viven detrás de un NAT, algunos pueden tener antivirus, software que bloquea ciertos puertos y protocolos, o también pueden estar detrás de *proxies* o *firewalls*, tal y como se puede observar en la Imagen 17.

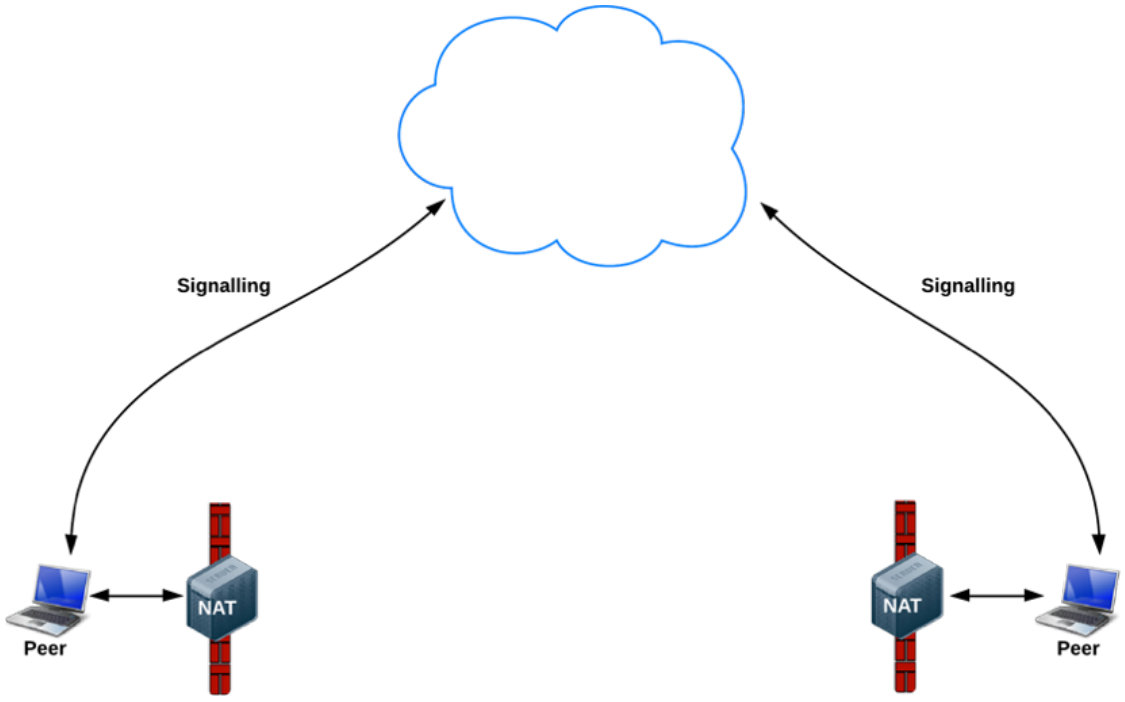

Imagen 17. Mundo Real.

Las aplicaciones de WebRTC pueden usar el ICE Framework para superar todas estas complejidades que presenta el mundo real. Para que esto sea posible se debe proporcionar los servidores ICE como parámetro al RTCPeerConnection, como se puede observar en la Imagen 18.

```
let rtcPeerConf = {
    iceServers: [
        { 'url'           : 'stun:stun.l.google.com:19302' },
        { 'url'           : 'stun:stun.services.mozilla.com' }
    ]
};

this.pc                 = new RTCPeerConnection( rtcPeerConf )
```

Imagen 18. Servidores ICE

ICE intentará encontrar el mejor camino posible para conectar a los *peers*, intenta todas las posibilidades en paralelo y elige la opción más eficiente.

Inicialmente, ICE intenta establecer una conexión utilizando la dirección del *host* que obtiene de la tarjeta de red, si esto falla debido a que se encuentra detrás de un NAT, ICE obtiene una dirección externa usando un servidor STUN, si esto continúa fallando, el tráfico se enruta a través de un TURN *relay server*.

En otras palabras:

- Un servidor STUN es usado para obtener una dirección de red externa.
- Un servidor TURN es usado para retransmitir el tráfico si falla la conexión directa P2P.

6.5.5 Servidores STUN

Los NATs proporcionan un dispositivo con una dirección IP para poder usar dentro de una red local, pero esta dirección no se puede usar en el exterior. Sin una dirección pública, no hay manera de que los *peers* de WebRTC se comuniquen, para evitar este problema WebRTC usa los servidores STUN.

El servidor STUN sirve para descubrir la dirección IP y puerto desde el punto de vista público. Este proceso permite a un *peer* WebRTC obtener una dirección de acceso público para sí mismo, que luego puede enviarla a otro *peer* a través del mecanismo de señalización, con el fin de configurar un enlace directo, tal y como se puede observar en la Imagen 19.

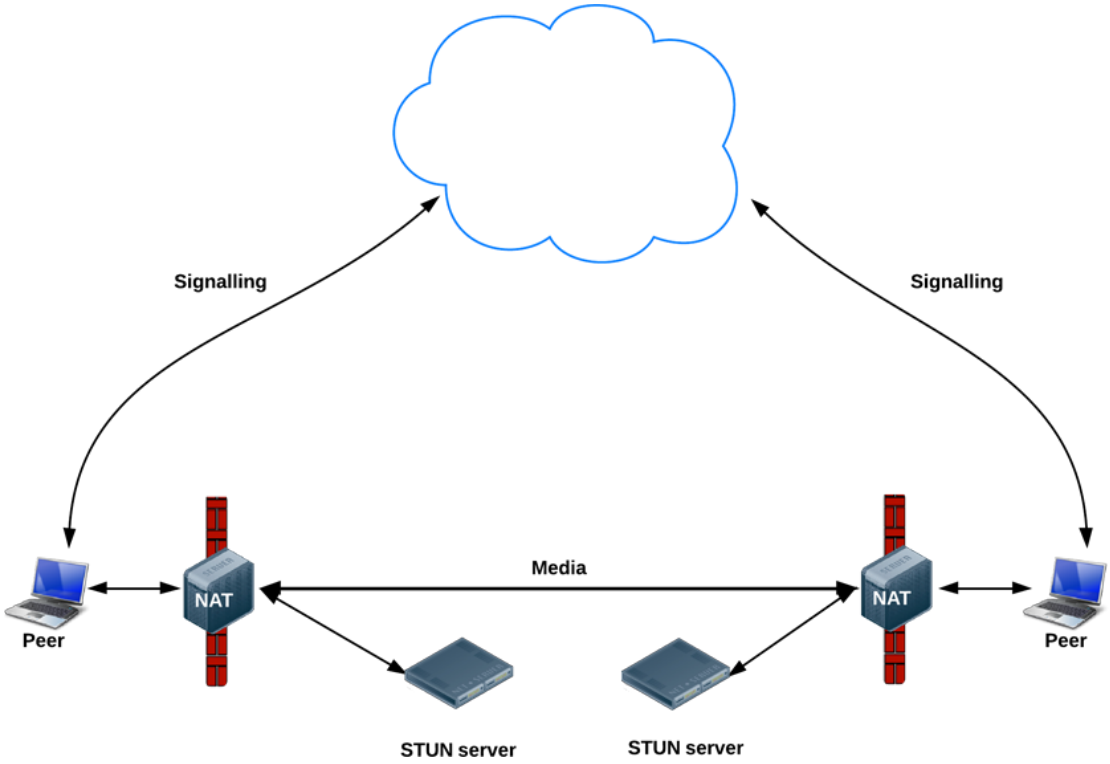

Imagen 19. Servidor STUN

6.5.6 Servidores TURN

RTCPeerConnection intenta establecer la comunicación directa entre *peers* a través de UDP, si falla la comunicación se recurre al protocolo TCP, y si esta falla nuevamente, utiliza los servidores TURN retrasmitiendo los datos entre los puntos finales, tal y como se puede observar en la Imagen 20.

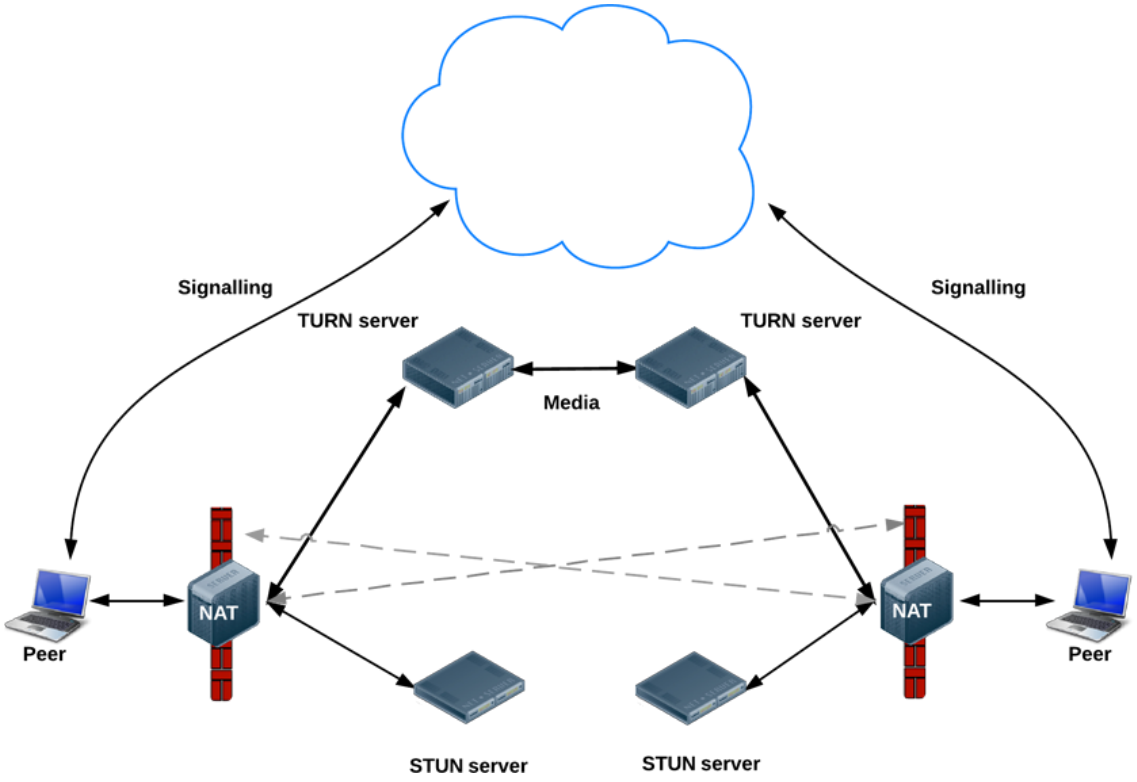

Imagen 20. Servidores TURN.

A diferencia de los servidores STUN, estos consumen mucho ancho de banda, y necesitan ser más robustos.

6.6 Node.js

Node.js es un entorno JavaScript de lado del servidor, utiliza el motor V8 JavaScript de Google para interpretar y ejecutar el código, soporta protocolos TCP, DNS y HTTP.

Uno de los principales objetivos de Node.js es proporcionar una manera fácil para construir aplicaciones de red rápidas y escalables.

Utiliza un modelo orientado a eventos sin bloqueo que hace que sea ligero y eficiente. Ideal para aplicaciones en tiempo real con un gran volumen de datos.

Node.js es de código abierto, puede ser ejecutado en Linux, OS X y Microsfot Windows, por otra parte existen muchos módulos en la comunidad, de manera que no hace falta escribir todo desde cero.

En conclusión, Node.js en realidad son dos cosas: Un entorno de ejecución y Una librería.

- Node.js = *Runtime Environment* + *JavaScript Library*

Las siguientes son algunas de las características más importantes de esta tecnología, siendo en la mayoría de los casos, la primera opción de los arquitectos de *software*.

- Asíncrono y controlado por eventos.
    - Todas las API's de la librería de Node.js son asíncronas y no bloqueantes, esto significa que un servidor basado en Node.js nunca va a esperar a que termine una petición para devolver los datos o para pasar a la siguiente petición.
    - El servidor puede trasladarse a la siguiente petición (cuando sea llamada) y un mecanismo de notificación de eventos de Node.js ayuda al servidor a obtener la respuesta de la petición anterior mediante un *Callback*.
- Con un único *Thread* pero altamente escalable.
    - Node.js utiliza un único modelo de *Thread* con *Event looping*.
    - Los mecanismos de eventos ayudan al servidor a responder de forma no bloqueante y hacen que el servidor sea altamente escalable.
    - El mismo Thread puede servir un gran número de peticiones que los servidores tradicionales como Apache HTTP *Server*.
- Muy rápido.

A continuación se enumeran las áreas en las que Node.js está demostrando ser muy potente:

- Aplicaciones con E/S de datos.
- Aplicaciones con datos en *Streaming*.
- Aplicaciones en tiempo real con un gran volumen de datos.
- Aplicaciones basadas en API's JSON.
- Aplicaciones de una sola página.

Node.js presenta múltiples módulos que se pueden instalar vía NMP, uno de los módulos principales para este proyecto es el *Framework* Express.

6.6.1 Framework Express

Es un *Framework* minimalista y flexible que facilita el desarrollo de aplicaciones web basadas en Node.js.

Las características principales del *Framework* son:

- Permite configurar middlewares para responder a las peticiones HTTP.
- Define el enrutamiento que se utiliza para llevar a cabo las diferentes acciones, basadas en el método HTTP.
- Permite representar dinámicamente páginas HTML con el paso de argumentos en las plantillas.

6.7 Socket.io

Es una librería JavaScript para las aplicaciones en tiempo real, habilita la comunicación bidireccional entre los clientes web y servidor basada en eventos.

Funciona en todos los navegadores y dispositivos centrándose por igual en la fiabilidad y velocidad.

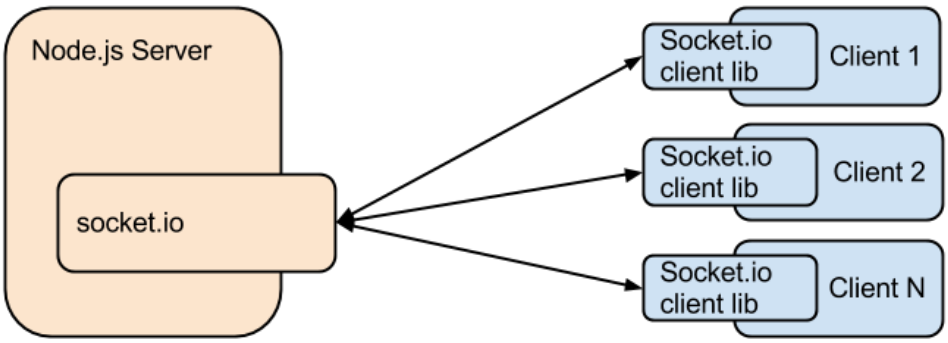

Imagen 21. Conexión cliente y servidor Socket.io.

Presenta dos partes importantes:

- Una librería en el lado del cliente.
- Una librería en el lado del servidor para Node.js

Ambas librerías puedes ser instaladas a través de la herramienta NPM.

Socket.io utiliza principalmente el protocolo WebSocket, aunque se puede usar simplemente como un *Wrapper*, proporciona más características que incluyen, *broadcast* a múltiples sockets, guardar datos asociados de cada cliente y E/S asíncronas.

Es una librería muy importante de cara al desarrollo de la aplicación, debido a que la comunicación en tiempo real es una las características importantes de este proyecto.

6.8 MySQL

MySQL es un sistema de gestor de base de datos relacional, es *open source* y su estructura está basada en el lenguaje SQL.

Sirve para almacenar y administrar los datos en una bases de datos relacional, utilizando tablas, vistas, funciones y muchas más características.

Se ejecuta virtualmente en casi todas las plataformas, como Linux, Unix y Windows, su uso habitual es en aplicaciones web.

Es basado en el modelo cliente-servidor, MySQL server es quien manipula toda las instrucciones o comandos que se realicen a la base de datos, también se puede utilizar un MySQL *client*, como por ejemplo MySQL Workbench.

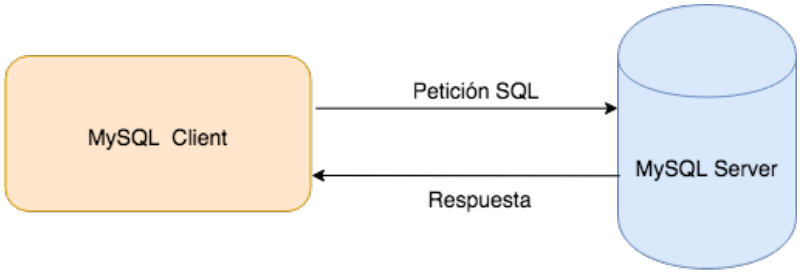

Imagen 22. Arquitectura Cliente-Servidor MySQL.

6.9 HTML5

HTML5 es la última versión de HTML, establece una nueva serie de elementos y atributos que actualmente se adaptan a los nuevas páginas Web, algunos de los elementos nuevos que se puede destacar son los siguientes:

- <audio>
- <video>
- <canvas>
- <header>
- <footer>
- <section>
- <nav>

Uno de los objetivos principales en este nuevo diseño es soportar la transmisión multimedia en dispositivos móviles, por ese motivo se han añadido los elementos <audio> , <video> y <canvas>, además añade nuevas características que permite a los usuarios interactuar con los documentos, tales como:

- Nuevas reglas de *parsing* para mejorar la flexibilidad.
- Nuevos atributos.
- Elimina atributos obsoletos o redundantes.
- Capacidades de *Drag and Drop* de un documentos HTML5 a otro.
- Edición sin conexión.
- Un estándar común para almacenar datos en una base de datos SQL (Web SQL).

6.10 Sass/CSS3

Sass es un preprocesador de CSS que añade características muy potentes y elegantes a este lenguaje de estilos, algunas de las características principales que presenta son:

- Permite crear variables.
- Reglas CSS anidadas.
- Mixins.
- Importación de hojas de estilos .

El objetivo principal es hacer un código simple y más eficiente, es compatible con todas la versiones de CSS, el único requerimiento para poder utilizarlo es tener instalado la librería Ruby.

Para un diseño *responsive* se utilizan en mayor parte las *media queries* de CSS y junto con Sass, hacen que el diseño multiplataforma sea muy sencillo de realizar.

6.10.1 Media Query

Una *media query* consiste en un tipo de consulta que limita las hojas de estilo utilizando características del medio como ancho, alto y color.

6.11 Docker

Docker es una herramienta *open source* diseñada para facilitar la creación, despliegue y ejecución de aplicaciones mediante el uso de contenedores.

Permite crear contenedores ligeros y portables para que las aplicaciones se puedan ejecutar en cualquier máquina con Docker instalado. Estos permiten al desarrollador empaquetar una aplicación con todas las librerías o dependencias que esta necesite, así el desarrollador se asegura de que la aplicación es ejecutada correctamente independientemente de las configuraciones personalizadas del sistema operativo.

Docker funciona parecido a una máquina virtual, pero la diferencia es que en lugar de crear un sistema operativo virtual completo, permite que las aplicaciones usen el mismo Kernel de Linux que el sistema en el que se ejecutan, esto proporciona un impulso significativo en el rendimiento y reduce el tamaño de la aplicación.

Es una herramienta diseñada para facilitar el trabajo a los desabolladores y administradores de sistemas, para los desarrolladores es útil debido a que se pueden enfocar en escribir código sin preocuparse en el sistema donde se va a ejecutar, en cambio para el administrador de sistemas le brinda flexibilidad y reduce la cantidad de sistemas necesarios debido a su tamaño y menor sobrecarga.

El problema que presenta este sistema es que, normalmente una aplicación necesita de múltiples servicios para poder funcionar, eso implica crear un contenedor para cada servicio y ejecutar cada uno de ellos, pero este enfoque no es muy eficiente.

Para resolver este problema Docker tiene una herramienta llamada Docker Compose, que permite utilizar varios servicios y comunicarlos entre ellos.

Docker Compose se basa en un fichero yml donde se indican todos los servicios que se desea desplegar, las dependencias y alguna configuración especial, finalmente solo se necesita ejecutar un comando para poner en marcha la aplicación.

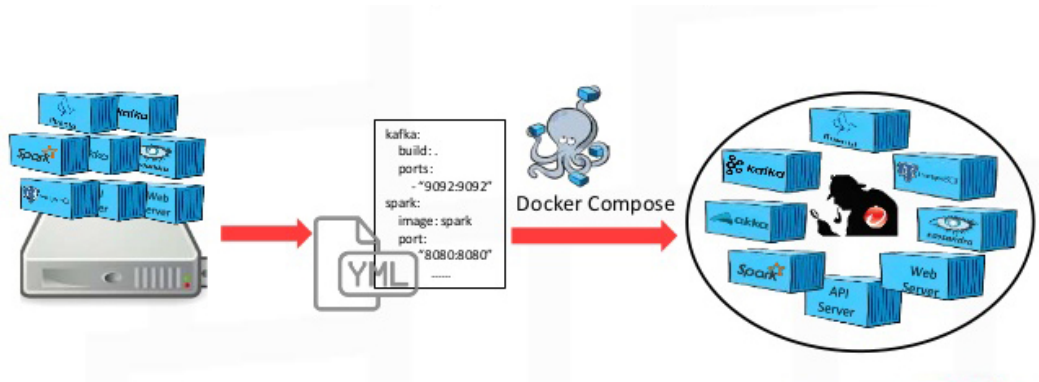

Imagen 23. Docker Compose.

7 Arquitectura de la aplicación

En esta sección se explica la arquitectura general del sistema, los diferentes tipos módulos que se han creado para llevar a cabo la comunicación cliente-servidor, las características y sus relaciones.

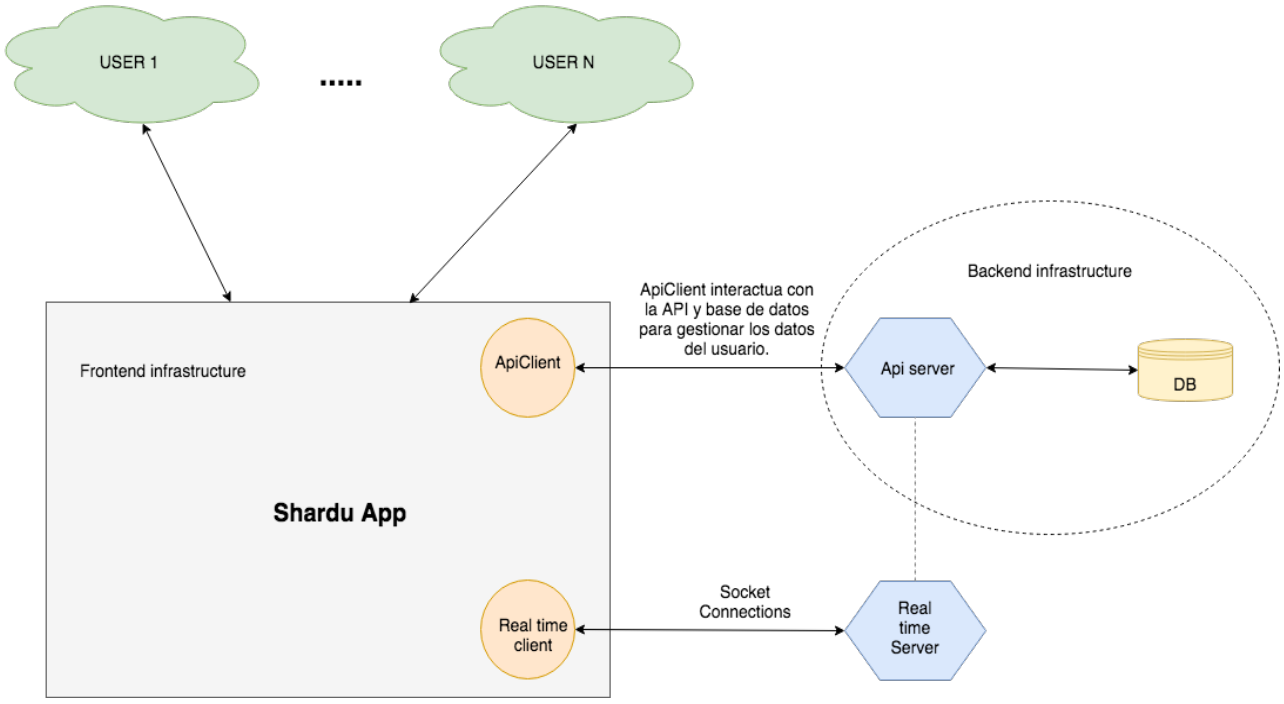

Imagen 24. Arquitectura general de la aplicación.

Se ha pensado en desarrollar un sistema flexible y altamente escalable, por ese motivo se proporciona una arquitectura con n-capas, y módulos independientes que gestionan una función específica.

De esta forma simplifica la compresión y organización del desarrollo y se reduce también las dependencias entre capas.

Los módulos que conforman esta arquitectura son los siguientes:

- **Infraestructura de Frontend**
  - Shardu App, capa de presentación y lógica.
  - ApiClient, módulo encargado de la comunicación con la API.
  - Real Time Client, módulo encargado de la comunicación con el Real Time Server.
- **Infraestructura del Backend**
  - API, capa lógica.
  - Base de datos, capa de datos.
- **Real Time Server**

7.1 Infraestructura del Frontend

7.1.1 Shardu App

Todo el frontend de la Aplicación se basa en React.js y se ha establecido una estructura poco común a lo visto en la mayoría de aplicaciones desarrolladas en esta tecnología, en la Imagen 25 se puede observar un desglose de la infraestructura del Frontend.

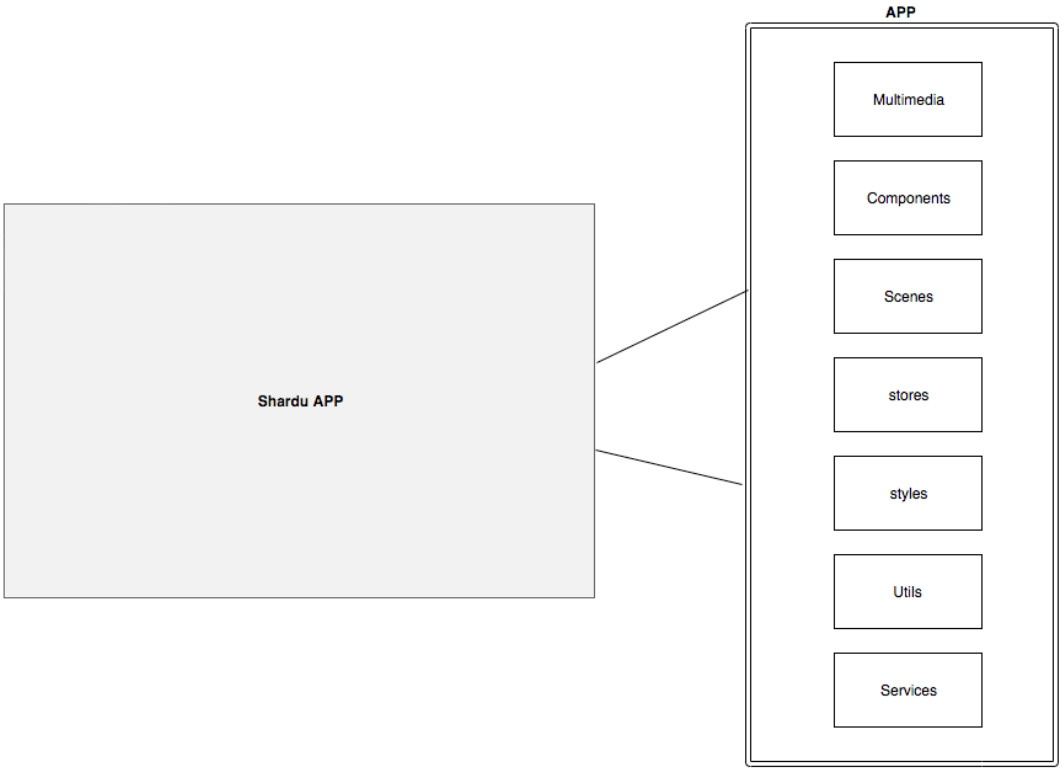

Imagen 25. Arquitectura del frontend de la aplicación.

Se ha utilizado un concepto nuevo denominado Scenes [5], que sirve para representar cada una de las páginas de la aplicación, como por ejemplo la pizarra virtual. Cada *scene* puede tener lo necesario para trabajar por sí misma, es decir estilos, componentes e imágenes.

A continuación se hablará con más detalles sobre la arquitectura de la aplicación.

7.1.1.1 Services

Contiene los servicios usados por más de un componente o *scene*, es decir servicios globales, como por ejemplo:

- API
- Analytics
- Utils

7.1.1.2 Components

Contiene los componentes que son usados en más de un componente o *scene*, por ejemplo:

- Botón personalizado
- TopBar

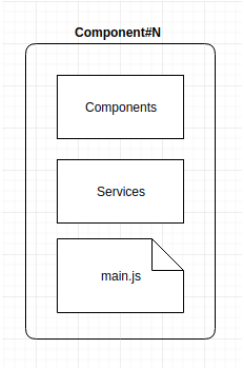

Imagen 26. Estructura de los componentes.

Como se puede observar en la Imagen 26, cada componente puede tener otros componentes, que son exclusivamente para ellos.

También pueden contener *scenes*, pero eso va a depender de si existe *routing* dentro del componente.

7.1.1.3 Scenes

Cada *scene* es mapeada a una ruta y puede contener otras *scenes*, componentes o servicios específicos a cada una de ellas.

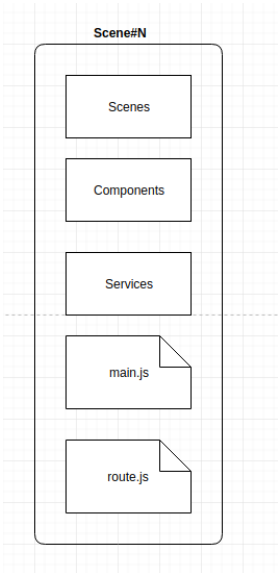

Imagen 27.Estrcutura de las Scenes.

route.js es opcional y solamente se necesita cuando una *scene* contiene otras *scenes* dentro.

7.1.1.4 Styles

Se ha seguido el *7-1 pattern*, que separa todo lo relacionado a estilos de cada *scene* o componente, teniendo así una mayor flexibilidad de cara a modificar o añadir estilos.

Una de las grandes ventajas que presenta este *pattern* respecto al típico estilo en línea de React.js, es que se genera un único archivo y es independiente del código JS principal, con esto se consigue obtener mayor rapidez al momento de cargar la página debido a que el navegador lo carga solo la primera vez y para futuras peticiones este se guarda en caché.

7.1.1.5 Store

Contiene la **lógica de negocio** y la estructura de datos de la aplicación, está muy ligado a redux.

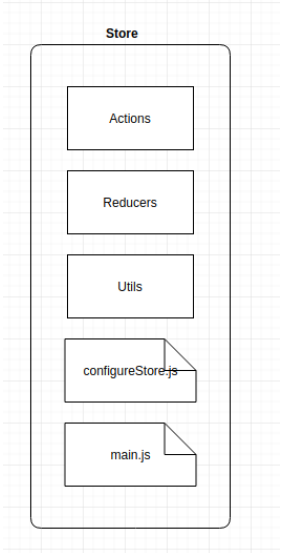

Imagen 28. Estructura del Store.

- **Utils**, contiene módulos extras como redux-persitor.
- **Actions**, contiene la lista de acciones, con cada *namespace*, siguiendo la siguiente estructura:

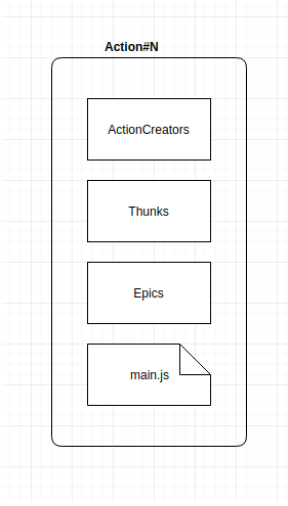

Imagen 29. Estructura de las Acciones.

- **Reducers**, contiene el *namespace* de los reducers, la estructura es la siguiente:

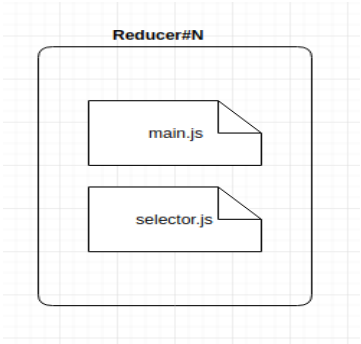

Imagen 30. Estructura de los Reducers.

Main contiene la estructura de los datos y el selector proporciona acceso a ellos.

7.1.1.6 Multimedia

Contiene las imágenes, logos e iconos de la aplicación.

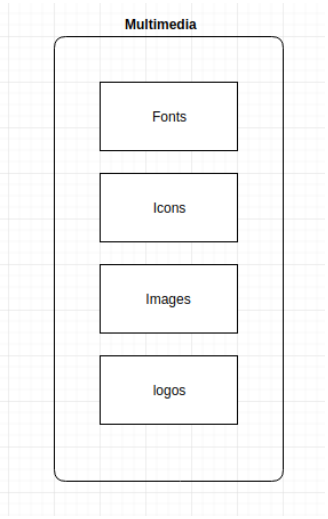

Imagen 31. Estructura de Multimedia.

De esta forma se consigue entender fácilmente la estructura de la aplicación, pudiendo hacer cambios y futuras *features* de forma rápida.

Para que funcione correctamente la estructura planteada, se tienen que cumplir algunas reglas:

- Un componente puede tener solamente componentes y servicios anidados, nunca se deben usar *scenes* dentro de los componentes.
- Una scene puede tener componentes, servicios y nuevas *scenes*.
- Un servicio puede tener solamente servicios, nunca componentes o *scenes*.

Siguiendo estas reglas, el proyecto está preparado para poder ser modificado o se puedan crear nuevas funcionalidades, sin tener que preocuparse plenamente en la estructura.

7.1.2 ApiClient

Para la comunicación entre el cliente y servidor se ha creado un módulo privado independiente llamado ApiClient, con el fin de gestionar de forma correcta las peticiones que se hacen desde el cliente al servidor.

Así, conseguimos añadir una capa extra de seguridad a nuestro sistema y flexibilidad para poder comunicarnos con la API, debido a que este módulo puede ser instalado en cualquier cliente vía NPM.

Por otra parte también es importante para todo el proceso de actualización de un **accesToken**.

Para poder usar este módulo, se deberá instanciar desde el cliente y pasarle los siguientes parámetros:

- **getAceesToken**, función que proporciona el accesToken.
  - Parámetros: ninguno.
  - Devuelve: accesToken.
- **setAccessToken**, función para pasar el accesToken.
  - Parámetros: accessToken.
  - Devuelve: nada.
- **getRefreshToken**, función que proporciona los datos del refreshToken.
  - Parámetros: ninguno
  - Devuelve: refreshToken
- **setRefreshToken**, función para pasar el refreshToken
  - Parámetros: refreshToken.
  - Devuelve: nada.
- **apiVersion**, un número que representa la versión de la API.
- **apiSubdomain**, subdominio para gestionar los entornos.
- **domain**, dominio principal de la API.

7.1.3 Real Time Client

Para la comunicación con el *Real Time Server*, se ha creado un módulo denominado *Real Time Client,* es un módulo privado independiente y puede ser instalado vía NPM, se basa en el cliente de la librería Socket.io.

La función principal de este módulo es definir los eventos de forma personalizada, teniendo en cuenta el *namespace* que le corresponde, para posteriormente enviárselos al *Real Time Server* para que los gestione, tal y como se puede observar en la Imagen 32.

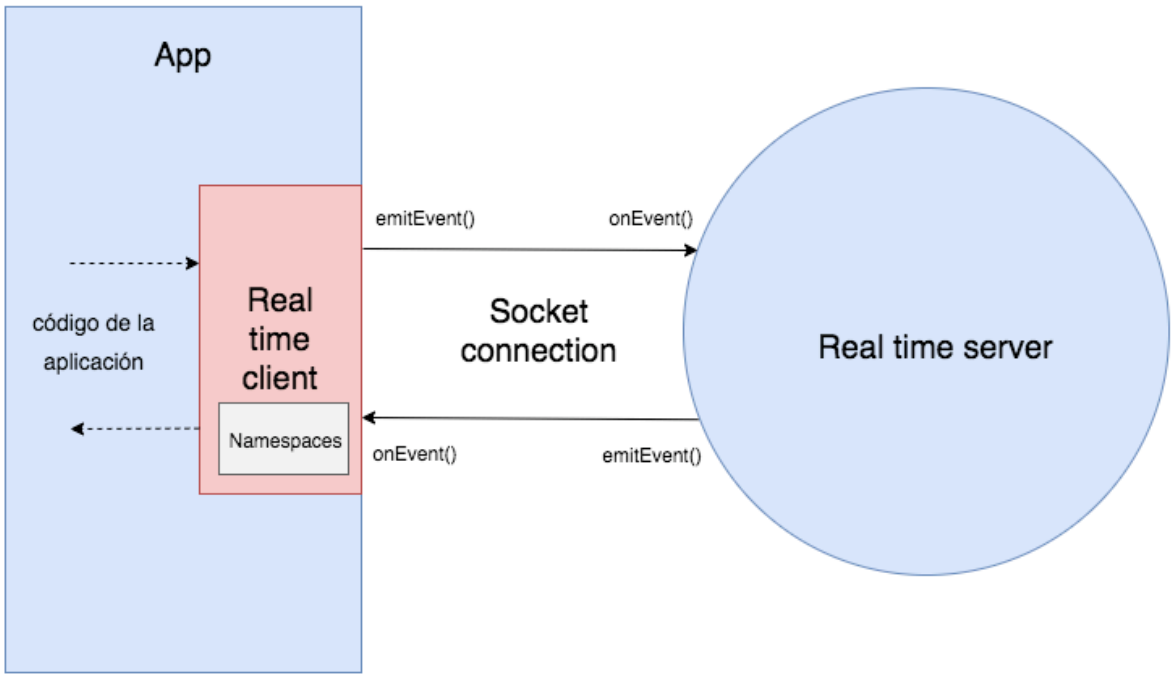

Imagen 32. Emitiendo eventos personalizados.

La conexión entre *Real Time Client* y *Server* se hace a través de Sockets, ambos deben trabajar conjuntamente para procesar los datos de forma correcta.

Un *namespace* es básicamente un diferente *endpoint* o *path*, que nos proporciona Socket.io, para minimizar los recursos o conexiones TCP y para aplicar una separación de los diferentes tipos de servicios que se creen en el sistema de *Real Time*, como por ejemplo, WebRTC, Notificaciones y Mensajes.

Los parámetros que recibe este módulo son los siguientes:

- **URL**, host que hace referencia al *Real Time Server*.
- **Port**, puerto en el que el *Real Time Server* está escuchando.
- ***conecctiontOps***, datos opcionales como, conexión SSL.
- **gestAccessToken**, *accessToken* del usuario.

Seguidamente, para definir un namespace se deben proporcionar los siguientes parámetros:

- **Id**, *namespace* Id, por ejemplo webrtc.
- **nsEvents**
  - **id**, parámetro que define el nombre del evento, por ejemplo INIT_CALL.
  - **canEmit**, parámetro para definir a un evento la posibilidad de emitir, por ejemplo TRUE.
  - **canListen**, parámetro para definir a un evento la posibilidad de escuchar, por ejemplo FALSE.

Finalmente, los eventos generados para poder utilizar en la aplicación son de este tipo:

- Partiendo del Id del evento : EventId
    - emitEvenId()
    - onEventid()

Así, se consigue una mayor flexibilidad y escalabilidad cuando se tengan que crear nuevos servicios en *Real Time* para consumir en la aplicación.

7.2 Infraestructura del *Backend*

7.2.1 API

En este apartado se explica la API de la aplicación, hemos utilizado la arquitectura API REST, es un tipo de arquitectura de desarrollo web que se apoya totalmente al estándar HTTP.

REST se compone de una lista de reglas que se deben cumplir en el diseño de una API:

- Interfaz uniforme.
- Peticiones sin estado.
- Cacheable.
- Separación de cliente y servidor.
- Sistema de Capas.
- Código bajo demanda.

El formato de intercambio de información de datos en este sistema se hace a través de JSON.

7.2.1.1 Funcionamiento de REST

Para los diferentes tipos de llamadas que se realicen a la API, existen un verbo especifico, los principales verbos para este sistema son los siguientes:

- GET: Obtener datos.
- PUT: Actualizar datos.
- POST: Crear un nuevo recurso.
- DELETE: Borrar el recurso.
- PATCH: Actualización parcial de datos.

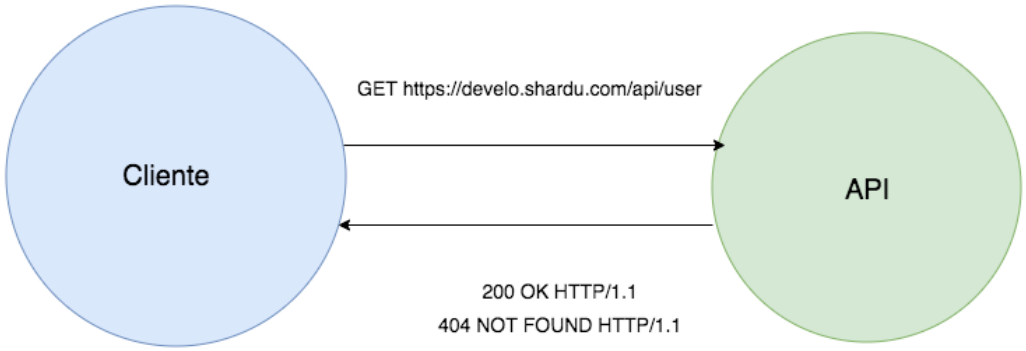

Imagen 33. Petición de un recurso con el método GET.

En la imagen 33, podemos observar un ejemplo de cómo obtener un recurso mediante el método GET.

- La URL, **https://develop.shardu.com/api/user**, representa el recurso.
- El método **GET**, representa la operación.

7.2.1.2 Idempotencia

El sistema sigue el principio de idempotencia [2] a los verbos HTTP, es decir que la ejecución repetida de una petición con los mismos parámetros sobre un mismo recurso, tendrá el mismo efecto en el estado de nuestro recurso si se ejecuta 1 o N veces.

A continuación se puede observar en la Tabla 2, cuales verbos HTTP, son o no idempotentes:

Tabla 2. Métodos HTTP idempotentes

Método HTTP	Idempotente
OPTIONS	SI
GET	SI
HEAD	SI
PUT	SI
POST	NO
DELETE	SI
PATCH	NO

Es muy importante este principio para definir de forma correcta los *endpoints* que gestionarán los diferentes tipos de recursos del sistema.

Un gran error que se comete en la mayoría de los casos, es utilizar el método PUT para la actualización parcial de un recurso, al ser este idempotente siempre espera todos los parámetros para actualizar dicho recurso, como se ve es ineficiente y se puede solucionar utilizando el método PATCH en su lugar, el cual nos permite la actualización parcial del recurso y no se espera todos los parámetros para poder actualizarlo.

7.2.2 Base de datos

En este sistema se utiliza una base de datos relacional, concretamente se basará en el sistema de gestión de base de datos MySQL.

En la imagen 34 se puede observar un ejemplo básico del flujo de una petición desde el cliente hacía la API y DB.

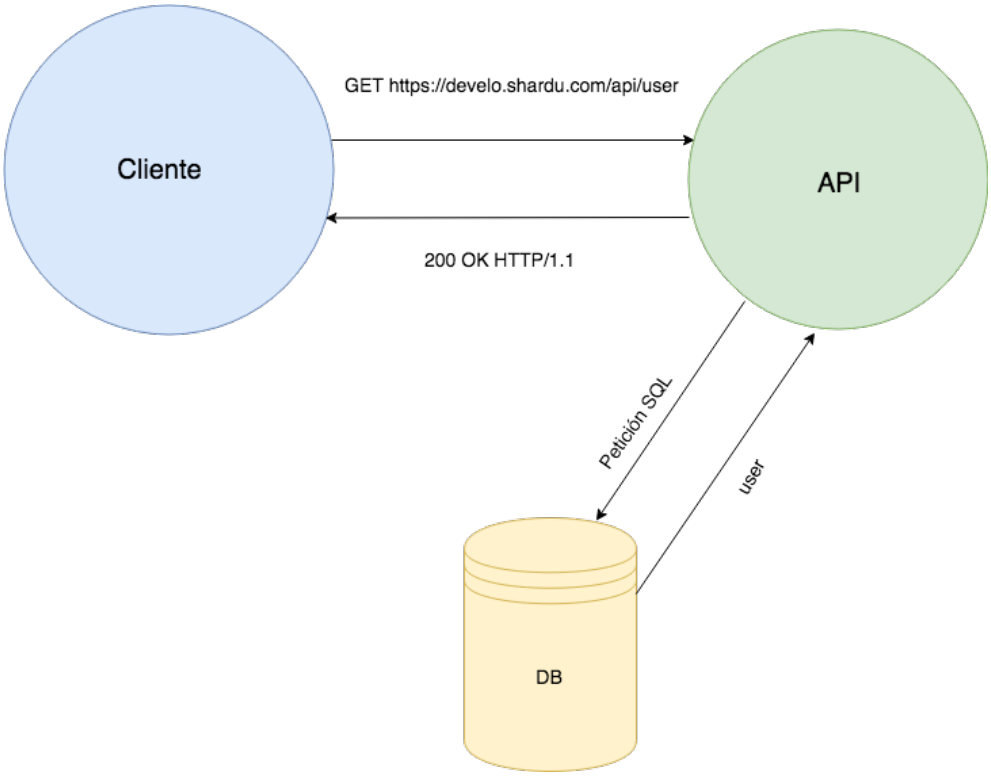

Imagen 34. Obtener un recurso de la base de datos.

7.3 Real Time Server

Real Time Server es un módulo independiente de la infraestructura del Backend de la aplicación, el motivo de hacer independiente este módulo se debe a la flexibilidad y escalabilidad que se quiere lograr de cara a futuras mejoras o implementaciones en el sistema de *Real Time*.

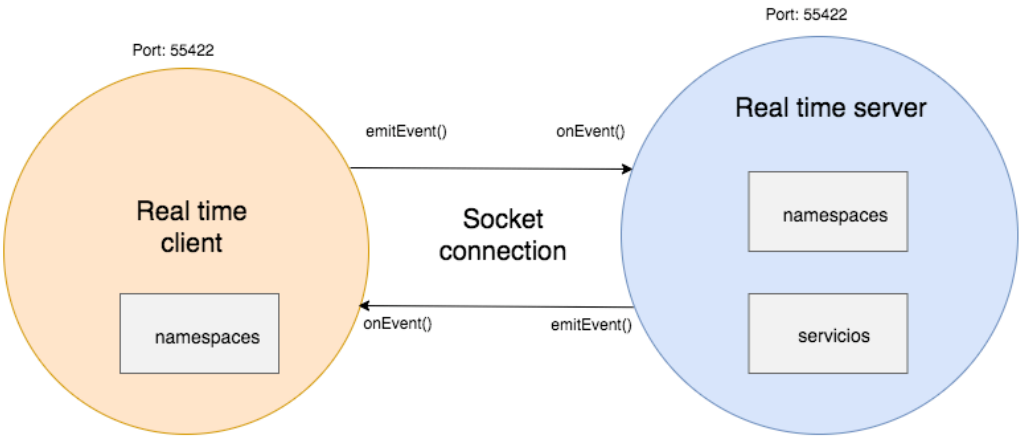

Imagen 35. Comunicación con el de cliente Socket.io.

Se basa en la librería Socket.io en el lado del servidor y es ejecutada sobre Node.js como un servidor independiente, eso quiere decir que se ha creado un nuevo servidor con un puerto diferente al de la API, desacoplando ambos servicios.

Para establecer la comunicación con el cliente, este debería pasarle el host y puerto correspondiente.

El módulo está desarrollado para soportar múltiples *namespaces* y servicios, tal y como se puede observar en la Imagen 36.

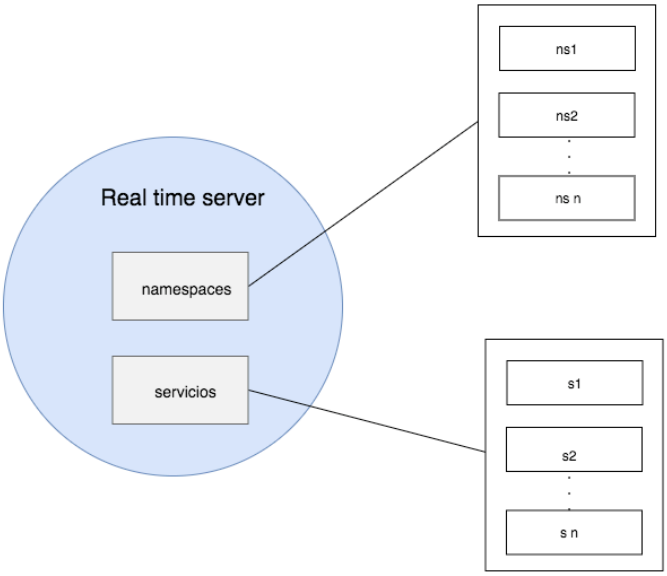

Imagen 36. Namespaces y servicios.

7.3.1 Namespaces

El servidor gestiona cada namespace definido en la aplicación con sus respectivos eventos.

7.3.2 Servicios

Por otra parte, los servicios son aquellos que sirven como soporte al *Real Time Server*, como por ejemplo, un servicio para establecer la conexión con la API y utilizar algún recurso.

8 Desarrollo e implementación

En esta sección se explica el desarrollo de los diferentes tipos de componentes de la aplicación y la estrategia que se tiene en cuenta es la arquitectura previamente planteada.

8.1 Cliente

Las partes principales del Frontend de la aplicación se pueden representar de la siguiente manera:

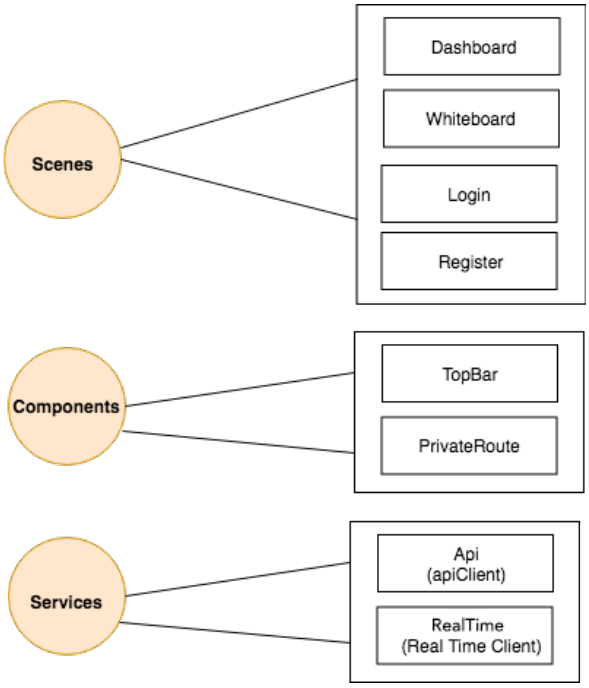

Imagen 37. Principales partes del Frontend.

Shardu, está compuesta por cuatro *Scenes* importantes:

- Dashboard.
- Whiteboard.
- Login.
- Register.

No todas las *Scenes* son públicas, es necesario tener una autorización para poder acceder a ellas, por esa razón las *scenes* se descomponen en públicas y privadas, tal y como se puede observar en la Imagen 38.

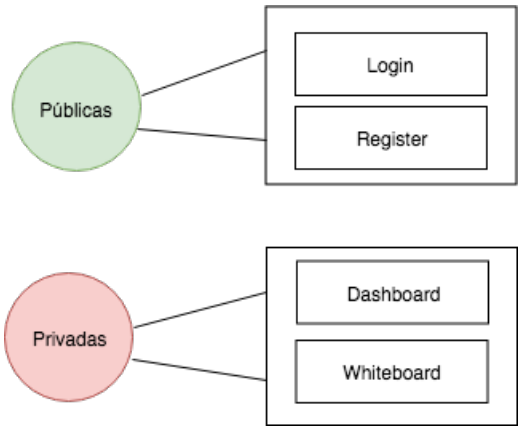

Imagen 38. Scenes públicas y privadas.

Las *Scenes* públicas son aquellas en las que el usuario puede acceder sin necesidad de autenticarse, es decir el *Login* y *Register*.

Por otra parte las *Scenes* privadas son aquellas en las que el usuario tiene que autenticarse obligatoriamente para poder acceder a ellas, es decir para el *Dashboard* y *Whiteboard*.

Los componentes principales de la aplicación son los siguientes:

- *TopBar*, presente en toda la aplicación
- *PrivateRoute*, para gestionar las rutas privadas o públicas.

Finalmente, los servicios que presenta la aplicación son:

- Api, servicio para gestionar la conexión con la API.
- *RealTime*, servicio para gestionar la conexión con el *Real Time Server*.

8.1.1 Autorización

Para gestionar la autorización del sistema entre el cliente y la API, se utiliza el sistema de autenticación basado en *token*, además es un papel fundamental en la seguridad de la aplicación.

Para este sistema se utiliza JSON Web Token [7],es un estándar que ha ganado popularidad debido a su tamaño compacto que permite que el *token* pueda ser fácilmente transmitido vía *query string*, a través de los atributos de las cabeceras o dentro del *body* de una petición POST.

Un JSON Web Token se compone de tres partes:

- **Header,** consiste de metadatos el cual incluye el tipo de *token* y el algoritmo hash usado para firmar el *token*.
- **Payload,** contiene los datos que el *token* está codificando.
- **Signature,** contiene la firma para verificar que el *token* es válido, para firmarlo es necesario una "se*cret-key*".

La representación de las tres partes conjuntas se hace la siguiente forma: **Header.Payload.Signature,** tal y como se puede observar en la Imagen 39.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Imagen 39, Representación de un JWT

El ciclo de vida de un *token* se puede representar de la siguiente manera:

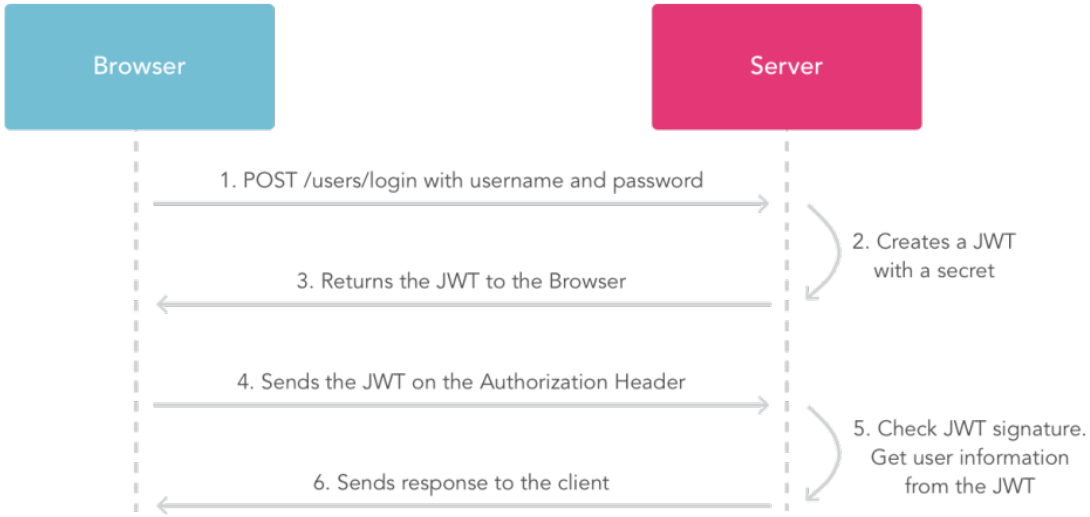

Imagen 40. Ciclo de vida un JWT.

El funcionamiento es tal que, a cada petición que se hace desde el cliente, este debe proporcionar un *token* que el servidor posteriormente tiene que verificar, si el *token* es válido el servidor responde a dicha petición.

Algunos comentarios a destacar:

- "Authorization Bearer", es la técnica que se usará para enviar el *token* a través de las cabeceras de la petición.
- Se crea un servicio de autenticación optimizado y desacoplado de la API.
- Se delega al cliente la gestión del estado del usuario.
- Con este sistema no se utilizan las cookies.

8.1.2 accessToken y refreshToken

Es muy importante contar con un *token* de actualización o refreshToken [8], para poder obtener un nuevo accessToken cuando este haya expirado.

El refreshToken puede ser usado hasta que se haya puesto en una zona llamada *BlackList*, debe mantenerse en una zona segura ya que permite al usuario estar autenticado siempre y si llegase a ser extraído por terceras personas, puede presentar un gran problema en la seguridad del sistema.

En este sistema se utilizan dos diferentes tipos de *tokens*:

- **accessToken**, basados en JSON Web Token, se encargan de la autorización de las peticiones hacía la API.
- **refreshToken**, es el encargado de obtener un nuevo accessToken cuando este haya expirado.

Cada vez que un usuario se registra o inicia sesión en la aplicación, se generan los dos tipos de *tokens*.

Cabe destacar que el sistema distingue entre dos tipos de usuario, GUEST y LOGGED_IN, se lo ha definido así por cuestiones de escalabilidad, para que de esta forma el sistema pueda tener una versión de invitado, así el usuario GUEST puede tener un *token* distinto al de uno ya registrado.

Para generar el accesToken, el *payload* lleva consigo los siguiente parámetros:

- **user_id**
- **scopes**, hace referencia a los diferentes tipos de peticiones que el usuario GUEST o LOGGED_IN pueden realizar, como *endpoints* y métodos.

user_token_sco...	url_pattern	http_method
GUEST	/favicon	GET
GUEST	/forgotPassword	POST
GUEST	/session	POST
GUEST	/session/facebook	POST
GUEST	/session/guest_mode_session	GET
GUEST	/signup	POST
GUEST	/signup/facebook	POST
LOGED_IN	/favicon	GET
LOGED_IN	/forgotPassword	POST
LOGED_IN	/session	POST
LOGED_IN	/session/facebook	POST
LOGED_IN	/session/guest_mode_session	GET
LOGED_IN	/signup	POST
LOGED_IN	/signup/facebook	POST
LOGED_IN	/token	POST
LOGED_IN	/token/:id	DELETE
LOGED_IN	/user	GET
LOGED_IN	/user	PATCH
LOGED_IN	/user/password	PUT

imagen 41. Scopes.

- **expiration_date**, fecha de expiración del token a 15 minutos.

Para explicar cómo se genera un refreshToken y los parámetros que debe llevar, es necesario destacar que, a diferencia del accessToken, este no siempre tiene porque expirar, existen casos en los que actualizar un *token* no tiene mucho sentido, como es el caso de los dispositivos móviles, normalmente una aplicación móvil nunca expira la sesión de un usuario, solamente se la puede revocar desde otro dispositivo.

Por ese motivo se tiene que distinguir entre dispositivos Móviles y Desktop, a continuación se explica los parámetros para generar un refreshToken:

- **Id**, identificador uuid().
- **user_id**, id del usuario.
- **expiration_date**, fecha de expiración de 6 meses para dispositivos de escritorio o null para dispositivos móviles.
- **user_token_scope**, GUEST o LOGGED_IN.
- **device_account_id**, parámetro para distinguir los diferentes dispositivos, debe seguir la siguiente estructura: <encryptedDeviceType>:<deviceAccountId>.

Finalmente, el proceso de refrescar un accessToken se hace automático a través del módulo ApiClient y sigue el siguiente flujo que se puede observar en la Imagen 42.

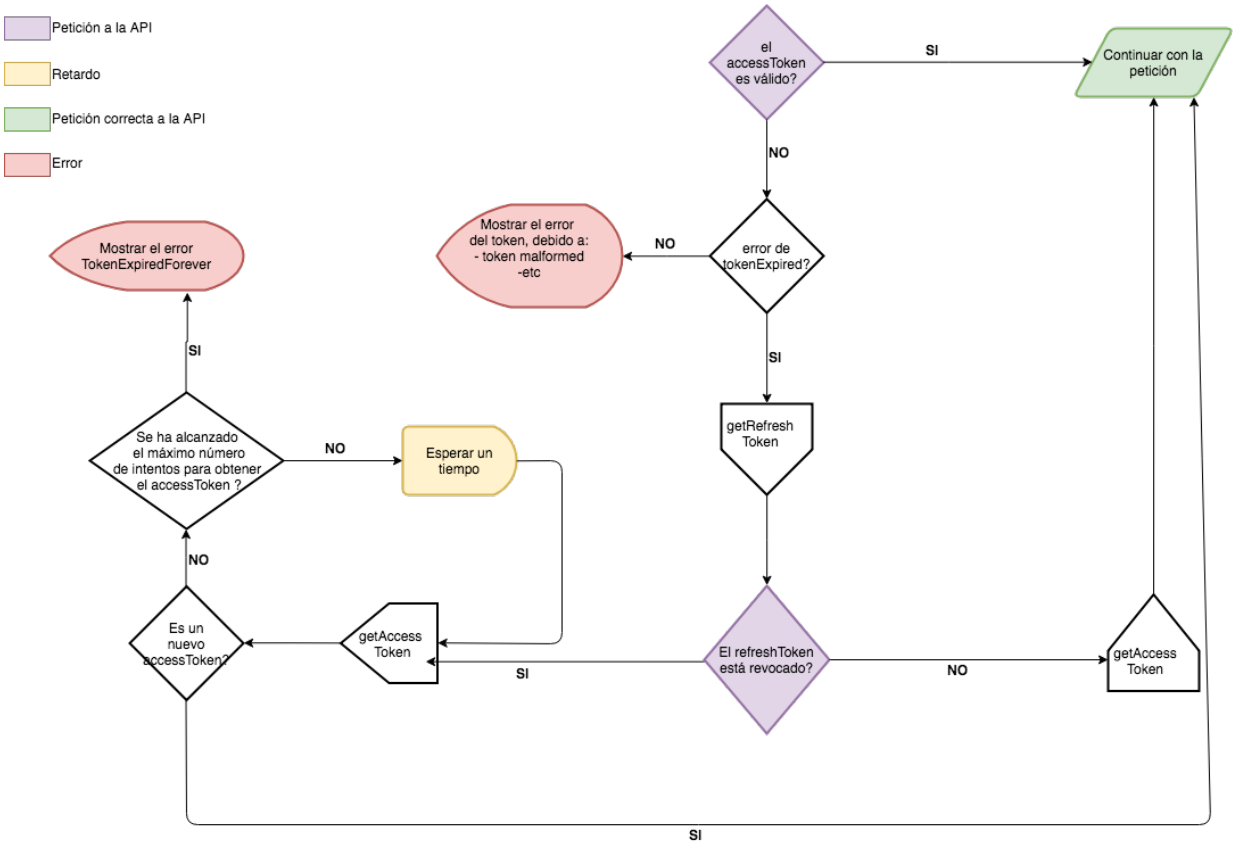

Imagen 42. Refrescar un AccessToken.

8.2 Store de la aplicación

Una vez el usuario inicia la sesión, es decir obtiene autorización para acceder al sistema, se utiliza Redux para guardar en el *store* datos importantes de cara a las acciones futuras que el usuario pueda realizar, como por ejemplo solicitar un nuevo recurso de la API.

Es necesario destacar, que todos los datos pasan al estado inicial cuando el usuario cierra la sesión, es decir:

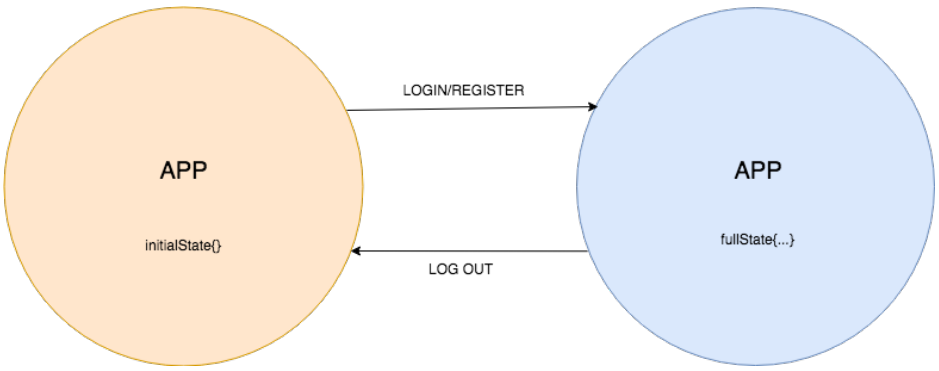

Imagen 43. Reset del estado.

A continuación se puede observar la estructura del *store* y los diferentes tipo de datos que almacena:

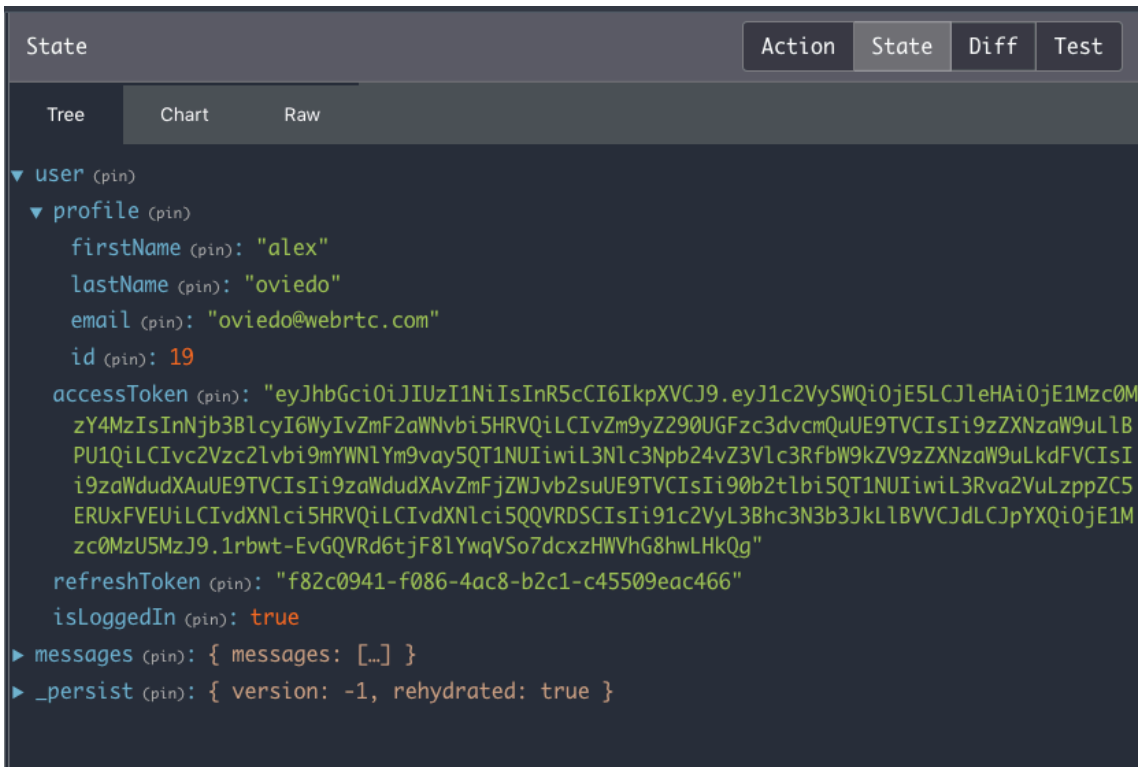

Imagen 44. Estructura del Store.

En este caso, los datos que se almacenan para poder comunicarse con la API son el accessToken y refreshToken.

Por otra parte, también es fundamental guardar datos que se van a reutilizar en todo el sistema, como pueden ser los datos básicos del usuario.

8.2.1 Perfil

Datos básicos del usuario principalmente utilizados en la *Scene Dashboard*:

- firstName
- lastName
- email
- id

8.2.2 accessToken y refreshToken

Datos de gran utilidad para la comunicación con la API.

- accessToken
- refreshToken

8.2.3 Mensajes

Redux también guarda datos referentes a los mensajes que se intercambian entre los usuarios, de esta forma si el usuario cierra el navegador o cambia de *Scene*, los mensajes se van a persistir hasta que el usuario cierre sesión, que es cuando se limpia todo el *store*.

8.3 Servidor

La API se puede representar de la siguiente manera:

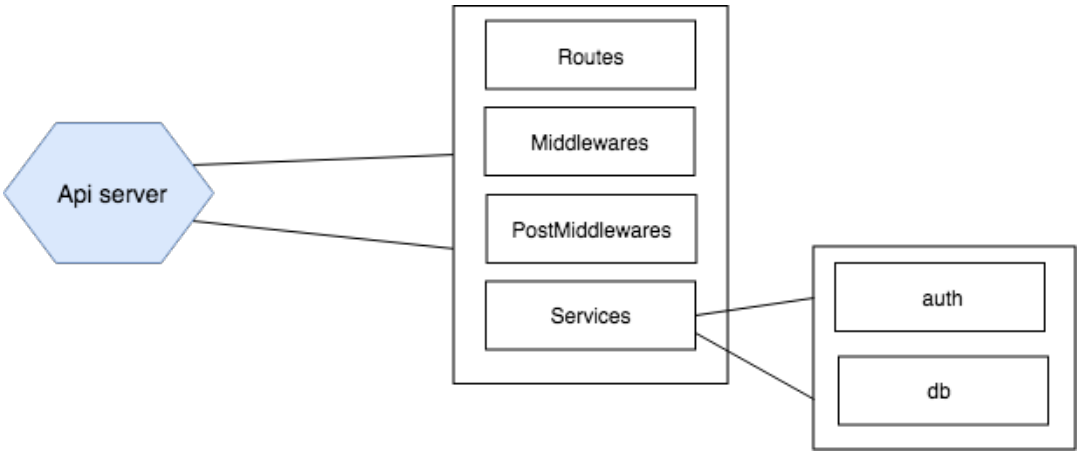

Imagen 45. Representación de la API.

Esta API utiliza el Framework expreses para facilitar todo el manejo de Rutas, Middlewares y más.

8.3.1 Rutas o Endpoints

La API cuenta con varias rutas para diferentes tipos de recursos, pero se diferencian en que algunas necesitan un accessToken para poder consumir de ellas, a continuación se explicará los dos tipos de rutas:

8.3.1.1 Con token

En la Tabla 3 se puede observar las rutas que necesitan un *token* para poder solicitar un recurso de ellas.

Tabla 3. Rutas con token.

URL	Método HTTP	Objetivos
/user	GET	Obtener información del usuario
/user	PATCH	Realizar un cambio parcial
/user/password	PUT	Cambiar la contraseña
/token/:id	DELETE	Revocar un *Refresh token*

La rutas con *token*, son aquellas que necesitan un accessToken válido para poder consumir de ellas.

8.3.1.2 Sin token

En la Tabla 4 se puede observar las rutas que no necesitan un *token* para poder solicitar un recurso de ellas.

Tabla 4. Rutas sin token.

URL	Método HTTP	Objetivo
/signup	POST	Crear un nuevo usuario
/session	POST	Crear una nueva sesión
/token	POST	Actualizar el accessToken
/docs	GET	Obtener la documentación
/guest_mode_session	GET	Obtener la sesión en modo invitado

Las rutas sin *token*, son aquellas que no necesitan un accessToken para consumir de ellas, en otras palabras son rutas públicas.

8.3.2 Middleware

El Middleware es en el encargado de gestionar las peticiones que entran en el servidor, actúa como un mediador antes de que la API devuelva un resultado, en este caso sus funciones principales son:

- Gestión de las Cabeceras HTTP.
- Gestión de la autorización a las rutas.
- Parsear las *queryString* de los peticiones.

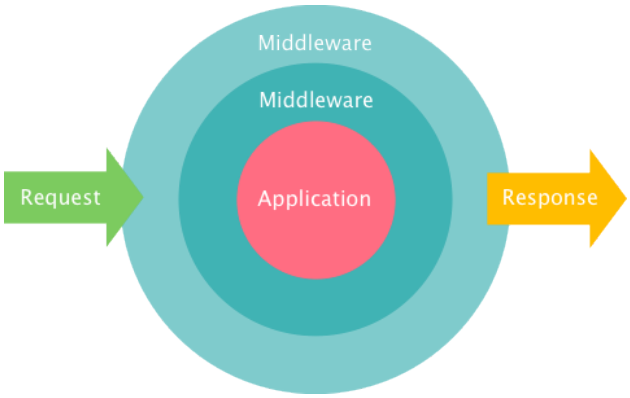

Imagen 46. Middleware.

8.3.3 PostMiddleware

Por otro lado, el PostMiddleware es el encargado de gestionar la respuesta del servidor al cliente, es decir una vez el Middleware comprueba que la petición cumple con todos los requerimientos de entrada que tiene el sistema, el PostMiddleware se encarga de enviar el resultado, ya sea error de un recurso no encontrado o la respuesta a dicho recurso.

8.3.4 Gestión accessToken y refreshToken

Como se ha comentado anteriormente existen dos tipos de *token* accessToken y refreshToken, cada *token* tiene una función distinta y se los crea de la siguiente manera:

- accessToken, se ha creado un servicio en la API enfocado a JWT.
- refreshToken, librería uuid.

8.3.4.1 Creación de accessToken

Cuando se genera un accessToken es necesario destacar que valores conlleva el *scope* para que al momento de descifrarlo se pueda determinar a qué recurso está autorizado, es decir los *scopes* representan las rutas a las que puede acceder el accessToken.

Debido a que el sistema presenta dos tipos de usuarios, GUEST Y LOGED_IN, los *scopes* generados para cada tipo son los siguientes:

user_token_scope	url_pattern	http_method
LOGED_IN	/session	POST
LOGED_IN	/session/guest_mode_session	GET
LOGED_IN	/signup	POST
LOGED_IN	/token	POST
LOGED_IN	/token/:id	DELETE
LOGED_IN	/user	GET
LOGED_IN	/user	PATCH
LOGED_IN	/user/password	PUT
GUEST	/session	POST
GUEST	/session/guest_mode_session	GET
GUEST	/signup	POST

Imagen 47. Scopes del usuario.

8.3.4.2 Creación de refreshToken

El flujo en la creación del refreshToken se puede observar en la Imagen 48:

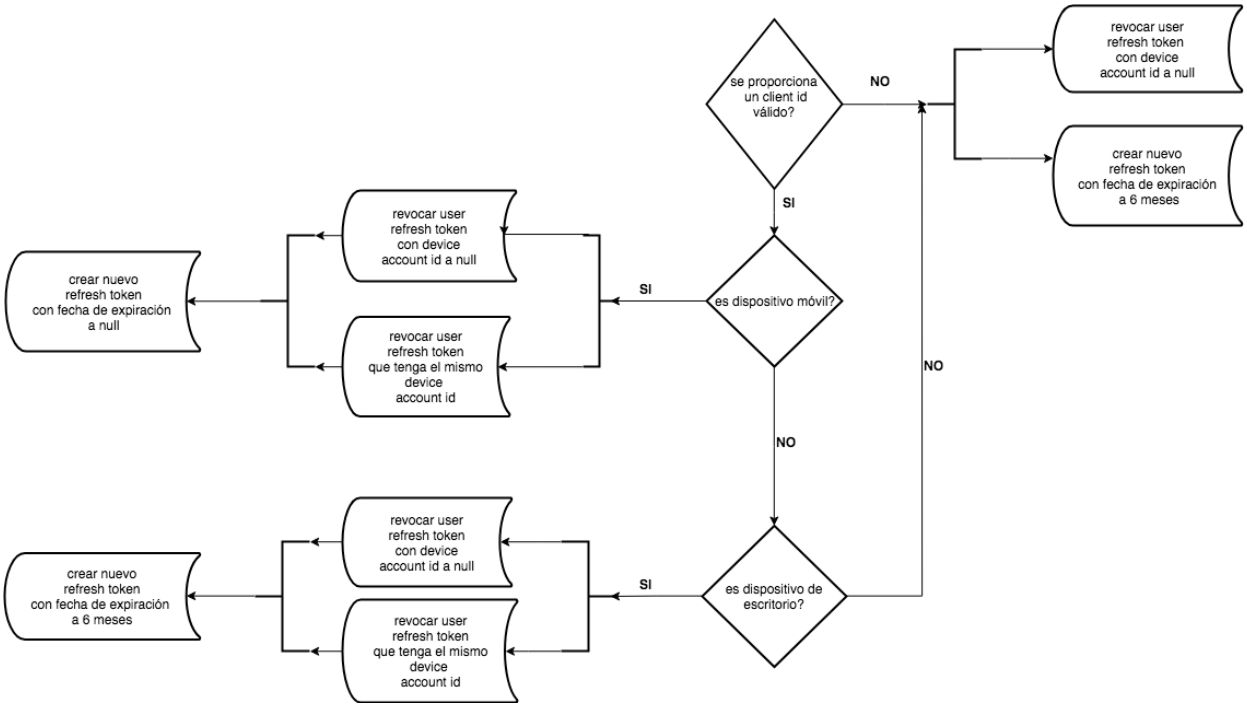

Imagen 48. Creación del Refresh Token.

Hay que tener en cuenta que la fecha de expiración del refreshToken va a depender mucho del dispositivo en el que el usuario se conecta, ya que existe mucha diferencia en acceder al sistema a través de dispositivo móvil o desktop, por temas de seguridad y UX.

Para que el servidor sepa diferenciar ambos dispositivos se debe pasar un CLIENT_ID a través del body del *sesión* o *register,* y se genera de la siguiente forma:

- **<encryptedDeviceType>:<deviceAccountId>,** el encryptedDeviceType es un string cifrado mediante AES con el texto MOBILE o DESKTOP, y el deviceAccountId es el identificador único del dispositivo ya sea IMEI o la dirección MAC.

Una vez se ha descifrado de forma correcta el texto enviado por el cliente ya se conoce el dispositivo por el cual el usuario accede y se procede a establecer una fecha de expiración:

- MOBILE, fecha de expiración NUNCA.
- DESKTOP, fecha de expiración 6 Meses.

Cuando el refreshToken caduca o es revocado, pasa a formar parte de la lista de *BlackList*.

8.3.5 Base de datos

La base de datos presenta seis tablas:

- **users**, esquema de la tabla de usuarios.
- **api_enpoints**, esquema de las rutas presentes en la API.
- **users_refresh_tokens,** esquema de los refreshTokens generados en el sistema.
- **users_refresh_tokens_blackilist,** esquema de los refreshTokens expirados o revocados.
- **users_token_scopes_types,** esquema de los diferentes tipos de usuarios en la aplicación, GUEST o LOGED_IN
- **users_grants_types,** esquema de los diferentes tipos de rutas a las que un usuario, GUEST o LOGEDIN tiene permitido acceder.

A continuación se muestra el diagrama de entidad relación donde se puede observar con mayor claridad el diseño de la base de datos:

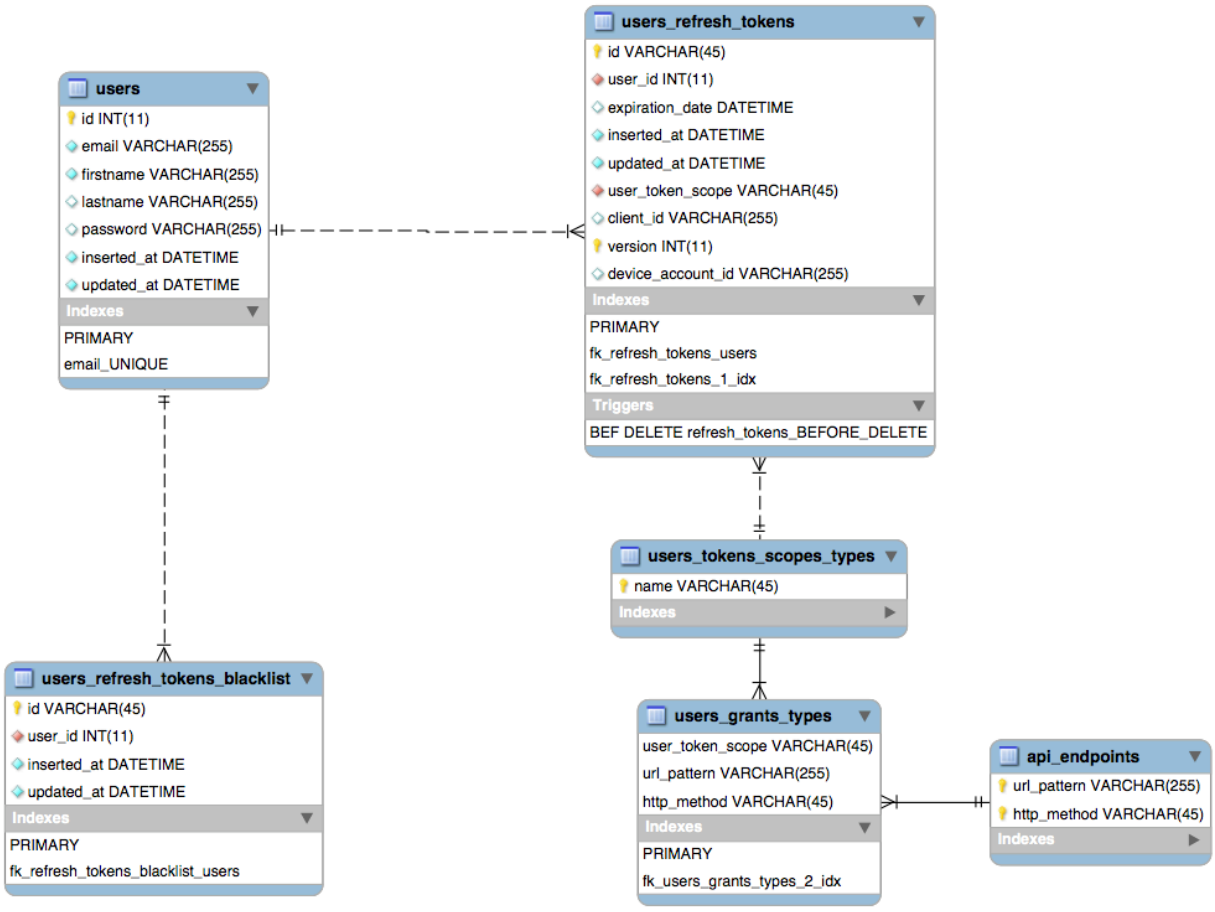

Imagen 49. Diagrama de entidad relación.

8.3.6 Json Schema Flow

El sistema presenta tres herramientas para ayudar a mantener y mejorar la API, estas comparten un vocabulario denominado Json Schema, el cual se divide en tres partes fundamentales que son:

- *End to end testing*
- Validación
- Documentación

Las tres herramientas comparten todos los esquemas definidos en el formato JSON para cada *endpoint* o recurso de la API.

8.3.6.1 End to end testing

Esta herramienta es útil para el testing de la API, se utiliza principalmente para testear todos los *endpoints* y así evitar posibles errores en la lógica.

Utiliza Mocha y Chai, dos librerías muy completas para las pruebas en entornos JavaScript.

```
  INIT TESTING
------------ Running testing system -----------
    SESSION Email
url http://localhost:3051/api/session
      ✓ POST api/session Successful (269ms)
url http://localhost:3051/api/session
******************************** handleError ***************************
Unknow api error
{ description: 'Check the email or password field',
  httpCode: 400,
  code: 'WRONG_CREDENTIALS',
  info:
   'https://develop.shardu.com/api/docs#errors/WRONG_CREDENTIALS',
  payload: {} }
      ✓ POST api/session WRONG_CREDENTIALS (114ms)
url http://localhost:3051/api/session/111111x
******************************** handleError ***************************
Unknow api error
{ description: 'The resource is not found',
  httpCode: 404,
  code: 'RESOURCE_NOT_FOUND',
  info:
   'https://develop.shardu.com/api/docs#errors/RESOURCE_NOT_FOUND',
  payload: {} }
      ✓ POST api/session/<something> RESOURCE_NOT_FOUND
    SESSION Guest mode
url http://localhost:3051/api/guest_mode_session
      ✓ GET api/guest_mode_session Successful
----------- Finish testing ------------


  4 passing (415ms)
```

Imagen 50. Resultados del testeo en la API.

Cada vez que se realiza un *test*, se ha establecido una regla para que se ejecute sobre todos los *endpoints* presentes en la aplicación, la forma correcta de testear cada uno es aplicando lo siguiente:

- Camino de la izquierda, es el encargado de testear todos los posibles errores de un recurso, obteniendo el error que le corresponde.

- Camino de la derecha, es el encargado de testear de forma correcta un recurso, obteniendo el resultado que le corresponde.

8.3.6.2 Validación

Esta herramienta funciona como un Middleware extra para comprobar que el formato de los datos que se pasan en el body de los métodos POST, PATCH y PUT son los correctos.

Las ventajas que presenta esta herramienta son las siguientes:

- La comprobación se realiza justo antes de procesar los datos en la API, por lo tanto no se consumen recursos si los datos de entrada son errores.
- Gran utilidad para los desarrollares de Frontend, debido a que los posibles errores se pueden chequear en *runtime*.

Un posible error de formato se lo puede observar en la Imagen 51.


```
"data": {
  "description": "Some format validation error in the body or url request",
  "httpCode": 400,
  "code": "FORMAT_VALIDATION_ERROR",
  "info": "https://develop.shardu.com/api/docs#errors/FORMAT_VALIDATION_ERROR",
  "payload": {
    "schemaErrors": [
      {
        "keyword": "format",
        "dataPath": "/email",
        "schemaPath": "#/properties/email/format",
        "params": {
          "format": "email"
        },
        "message": "should match format \"email\""
      }
    ]
  }
}
```


Imagen 51. Validación de datos de entrada.

En este ejemplo al momento de iniciar sesión el email no cumple con el formato que le corresponde, por lo tanto el resultado que devuelve es exactamente indicando que el formato email es erróneo, además se añade información extra para comprobarlo en la documentación.

8.3.6.3 Documentación

La documentación es una herramienta que funciona como una página independiente y se encarga de mostrar toda la información referente a la API.

Es de gran ayuda a los desarrolladores para ver toda la información de los datos de entrada y salida de cada recurso con las respectivas restricciones y tipo de datos.

A continuación se puede ver un ejemplo en la Imagen 52.

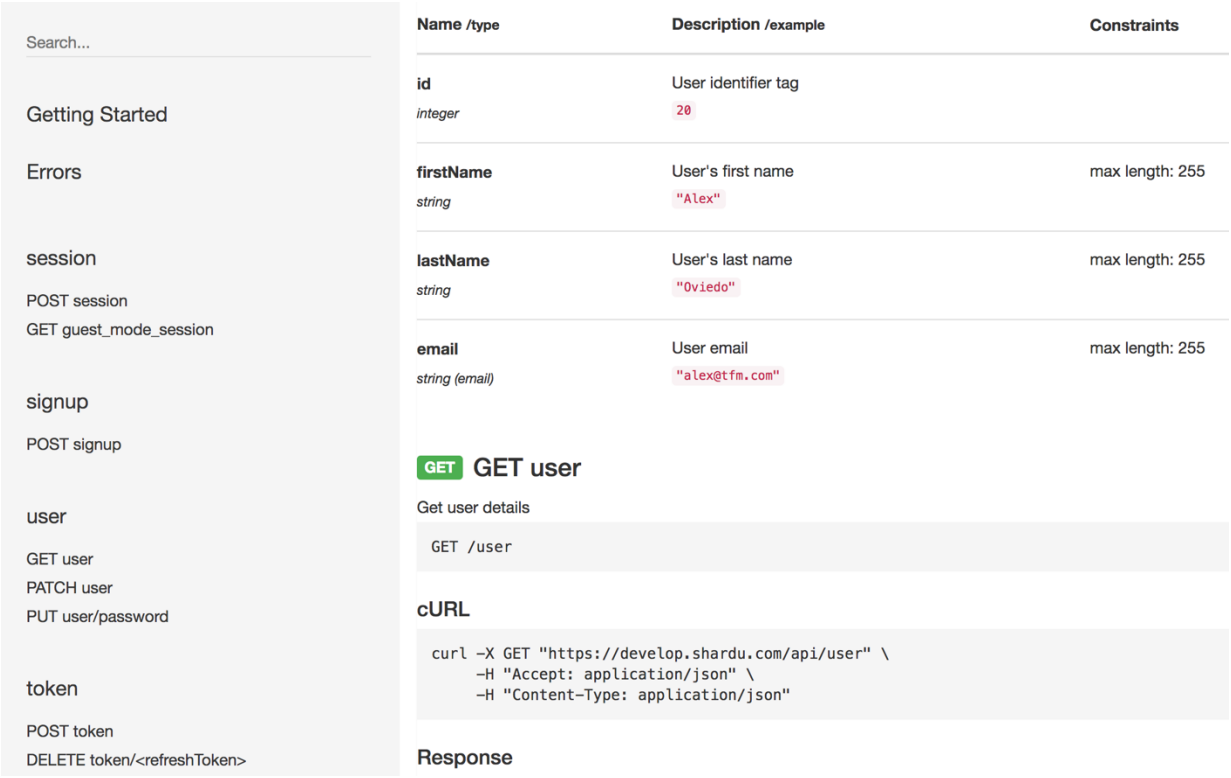

Imagen 52. Documentación de la API.

8.4 Real Time Client

Como se ha comentado anteriormente, el sistema de comunicación en tiempo real se hace a través de dos módulos, *Real Time Client* y *Real Time Server*.

El *Real Time Client* se encarga de definir los diferentes eventos y *namespaces* que se van a utilizar en la aplicación.

8.4.1 Namespaces

El sistema presenta tres *namespaces* enfocados a cada servicio en tiempo real que presenta la aplicación:

- webrtc, *namespace* para el servicio de videoconferencia.
- messages, *namespace* para el servicio de mensajería instantánea.
- whiteboard, *namespace* para el servicio de la pizarra virtual.

8.4.2 Gestión de Emitters and Listeners

Para cada *namespace* se establecen los eventos que estos van a realizar con sus respectiva función, es decir emitir, escuchar o ambas, y posteriormente el módulo crea los eventos personalizados.

La estructura de cada namespace se la puede observar en la Tabla 5:

Tabla 5. Estructura de los Namespaces.

Namespace	Eventos	Funciones
webrtc	INIT_REQUEST_CALL INIT_CALL END_CALL	Escucha y Emite Escucha y Emite Escucha y Emite
messages	SEND_MESSAGE RECEIVE_MESSAGE	Emite Escucha
whiteboard	DRAWING	Escucha y Emite

Finalmente los eventos personalizados para cada namespace se los puede observar en la Tabla 6:

Tabla 6. Eventos para cada namespace.

Namespace	Eventos
webrtc	onInitRequestCall() y emitRequestCall() onInitCall() y emitInitCall() onEndCall() y emitEndCall()
messages	emitSendMessage() onReceiveMessage()
whiteboard	emitDrawing() y onDrawing()

Estos eventos personalizados se generan en el mismo módulo, haciendo una transformación del nombre del evento recibido a un nombre sencillo de usar en el Frontend de la aplicación.

Normalmente para capturar un evento en socket.io se hace lo siguiente:

```
socket.on('chat message', function( msg ) {
    $('#messages').append($('<li>').text( msg ))
})
```

Imagen 53. Escuchar un evento simple.

Con la transformación generada por el módulo podemos escribir lo mismo de la siguiente manera:

```
socket.onChatMessage( msg => {
    $('#messages').append($('<li>').text( msg ))
})
```

Imagen 54. Escuchar un evento generado por un módulo propio..

8.5 Real Time Server

El *Real Time Server* trabaja conjuntamente con el *Real Time Client* para procesar los diferentes tipos de eventos que se ha definido en cada *namespace*, además contiene algunos servicios útiles como por ejemplo establecer conexión con la API.

Cada vez que un cliente se conecta, se crea un socket que posteriormente es utilizado por los diferentes tipos de *namespaces* de la aplicación.

Además a cada Socket se le establece un USER_NAME correspondiente a cada usuario.

8.5.1 Namespaces

Los *namespaces* son los mismos que se han definido en el cliente, en este caso:

- webrtc.
- messages.
- whiteboard.

Cada *namespace* gestiona sus eventos de forma independiente.

8.5.2 Servicios

Los servicios definidos son los siguientes:

- Api, para establecer comunicación con la API.
- Auth, para gestionar el accessToken.

Los servicios son de gran ayuda para futuras implementaciones, un ejemplo práctico sería el caso de los mensajes, de momento solo se almacenan en local a través de Redux, si en el futuro se desea guardar cada mensaje del usuario en la base de datos, se podrían utilizar estos servicios para comunicarse con la API y guardar los mensajes en la base de datos.

8.6 Componentes de la aplicación

En esta sesión se explica cómo se ha desarrollado cada componente, teniendo en cuenta el desarrollo de todos los módulos que se ha comentado previamente. Shardu cuenta con tres componentes principales:

- Videoconferencia
- Mensajería instantánea
- Pizarra virtual

8.6.1 Mensajería instantánea

La función principal de este componente es intercambiar mensajes de texto en tiempo real con todos los usuarios que acceden al sistema y está situado en la *Scene Dashboard*.

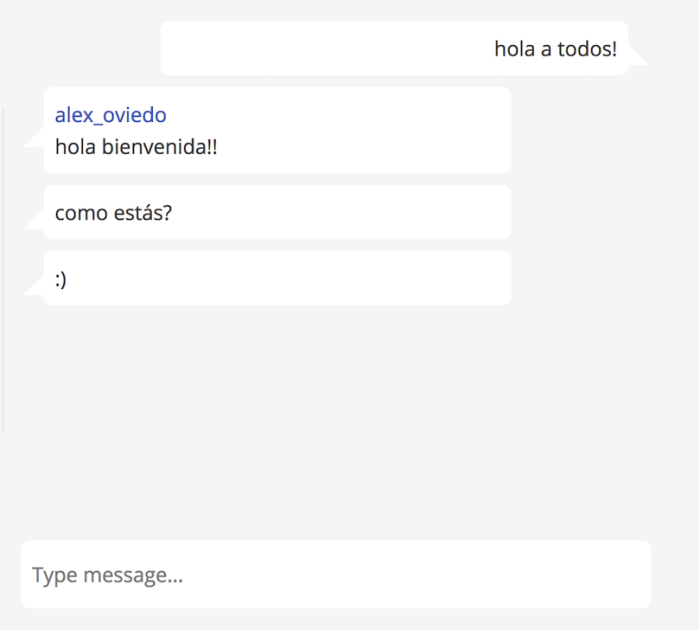

Imagen 55. Mensajería Instantánea.

Para hacer esto posible se utiliza principalmente los módulos de *Real Time Client* y *Real Time Server*, en el *namespace messages*.

Los principales eventos para realizar el intercambio de mensajes entre los dos módulos son:

- emitSendMessage(…), envía el texto y el autor del mensaje.
- onReceiveMessage(...), recibe el texto y autor del mensaje.

Todos los mensajes se guardan en el *store* de Redux, de esta forma el usuario siempre los tendrá de manera persistente en la aplicación y los puede consultar hasta que se cierre la sesión.

Cada vez que se recibe un mensaje se dispara la acción updateMessages(), y a través de un Reducer se actualiza el estado con el último mensaje recibido.

Se tiene en cuenta si el nuevo mensaje recibido es hecho por el mismo usuario del anterior mensaje, si pasa esto no se muestra el nombre si no, solo el mensaje, esto ayuda mucho a la experiencia de usuario en la aplicación.

8.6.2 Videoconferencia

El componente videoconferencia se encarga de llevar a cabo la comunicación en tiempo real de audio y video, está situado en la *Scene Dashboard.*

La videoconferencia funciona diferente a los mensajes, es decir no funciona en modo *broadcast*, por esa razón el usuario necesita especificar a qué USER_NAME del sistema desea realizar la llamada, al igual que una llamada telefónica.

Como se ha comentado anteriormente cada USER_NAME tiene asociado un socket que se crea cuando un usuario se conecta a la aplicación, este USER_NAME se puede encontrar en el Dashboard de cada usuario.

Está desarrollado de esta manera para evitar un caos entre todos los usuarios conectados en la aplicación, si se desea realizar una videoconferencia grupal, el sistema está preparado para crear un namespace que haga referencia a una *Room*, de momento al ser un producto mínimo viable, no se ha pensado en esta funcionalidad y queda pendiente para futuras implementaciones.

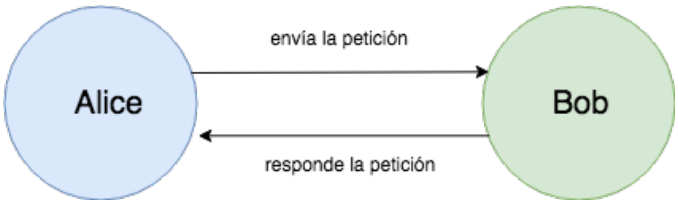

Imagen 56. Petición de una llamada.

Para conseguir esta comunicación se utiliza principalmente WebRTC y los módulos *Real Time Client* y *Real Time Server*, en el *namespace* webrtc.

Este componente se divide en tres estados:

- Petición de videoconferencia.
- Iniciar la videoconferencia.
- Terminar la videoconferencia.

8.6.2.1 Petición de videoconferencia

En este estado inicial es necesario especificar el USER_NAME al que se desea realizar la videoconferencia, una vez insertado y se realiza la acción de llamar (como se puede observar en la Imagen 57) el sistema envía el primer evento denominado emitInitRequestCall(), dicho evento conlleva el USER_NAME del usuario que envía la petición de videoconferencia y el USER_NAME del usuario a quien va dirigida.

El sistema no permite realizar una videoconferencia consigo mismo, si pasa esto se mostrará el error correspondiente antes de ejecutar la acción.

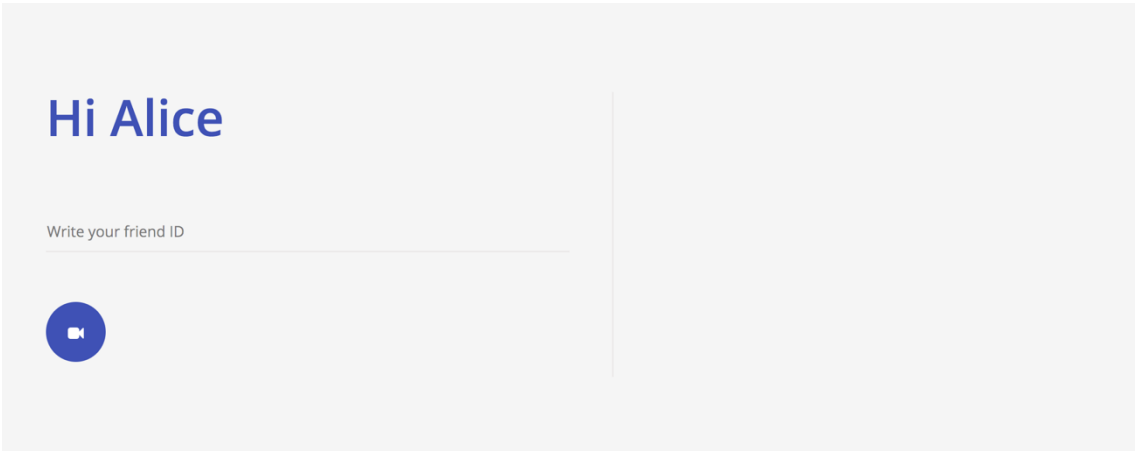

Imagen 57. Interfaz petición de llamada.

Partiendo de dos usuarios:

- Alice, envía la petición de llamada.
- Bob, recibe la petición de llamada.

Cuando Alice envía la petición de videoconferencia, Bob recibe una notificación indicando que Alice lo está llamando, tal y como se puede observar en la Imagen 58, dicha notificación contiene dos acciones:

- Iniciar la videoconferencia.
- Rechazar la videoconferencia.

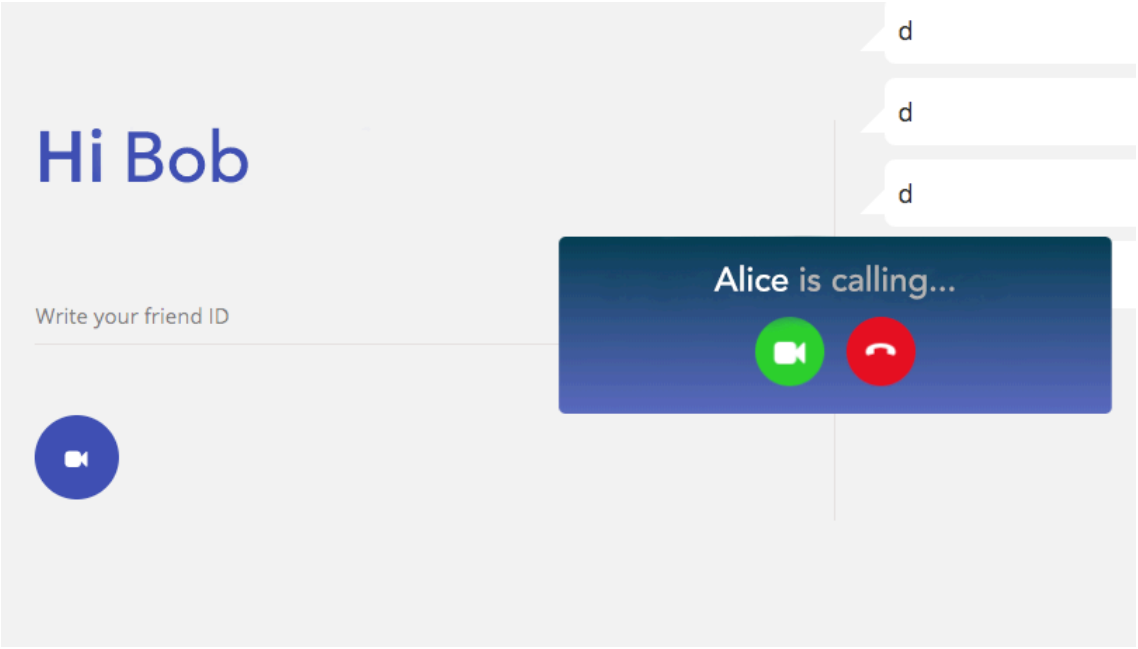

Imagen 58. Notificación de llamada.

Si Bob rechaza la videoconferencia el sistema vuelve al estado inicial, en caso contrario se procede a iniciar la videoconferencia.

8.6.2.2 Iniciar la videoconferencia

Cuando Bob acepta la llamada, se procede a iniciar la comunicación, en este punto WebRTC y el servidor de señalización son los encargados de gestionar todo este flujo de datos, como se puede observar en la Imagen 59.

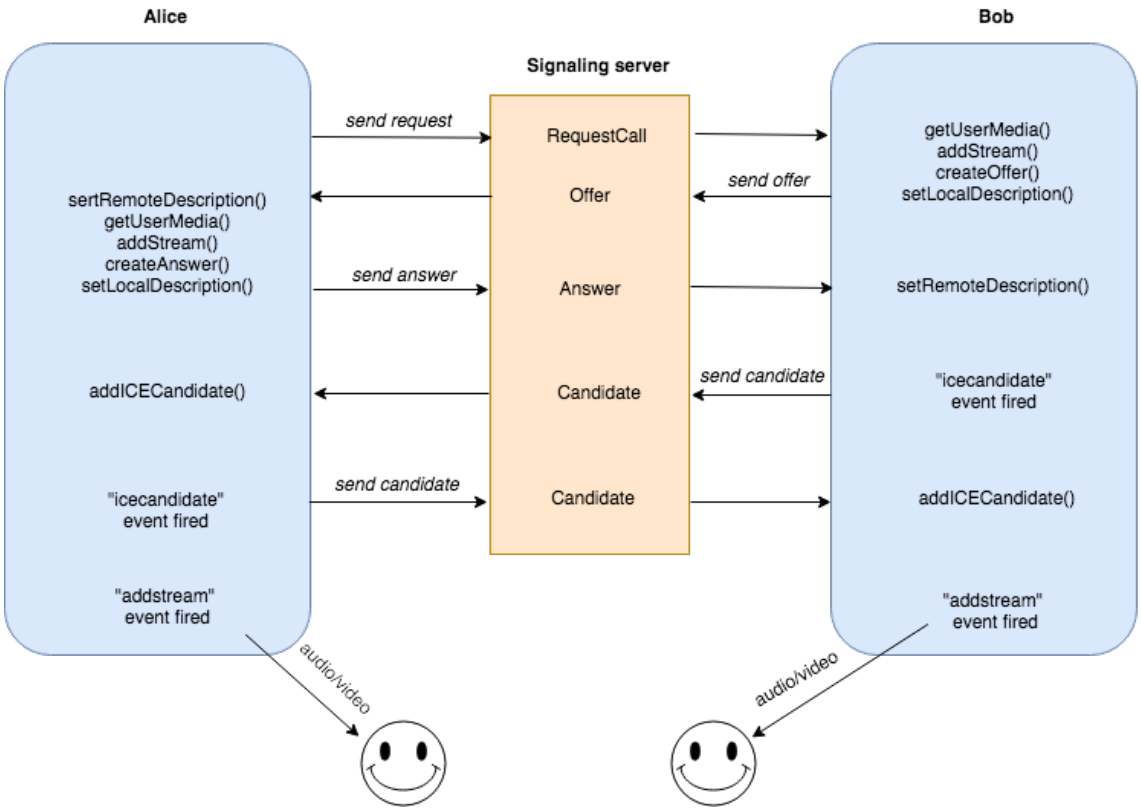

Imagen 59. Comunicación entre pares.

Bob acepta la petición de videoconferencia y procede a realizar las siguientes acciones:

- **getUserMedia(),** para obtener los permisos del usuario para utilizar la cámara y el micrófono, si el usuario acepta se sigue con el proceso, en caso contrario la llamada devuelve un error.
- **addStream(),** añade el stream obtenido en getUserMedia(), para luego establecer el stream remoto.
- **createOffer(),** crea una oferta para Alice.
- **setLocalDescription()**, establece la oferta en la descripción local de su máquina.
- Finalmente se envía la oferta mediante el servidor de señalización con el evento emitInitcall() hacía Alice.

Alice, recibe la oferta y ejecuta las siguientes acciones:

- **setRemoteDescription(),** añade la oferta recibida en la descripción remota de su máquina.
- **getUserMedia(),** para obtener los permisos del usuario para utilizar la cámara y el micrófono, si el usuario acepta se sigue con el proceso, en caso contrario la llamada devuelve un error.
- **addStream(),**añade el stream obtenido en getUserMedia(), para luego establecer el stream remoto.
- **createAnswer(),** crea la respuesta.
- **setLocalDescription()**, establece la respuesta en la descripción local de su máquina.

- Finalmente se envía la respuesta mediante el servidor de señalización con el evento emitInitcall() hacía Bob.

Bob recibe la respuesta y ejecuta la siguiente acción:

- **setRemoteDescription(),** añade la respuesta recibida en la descripción remota de su máquina.

Una vez se obtiene la descripción remota en Bob, se activa el protocolo ICE para obtener y enviar los ICECandidates y realizar la conexión WebRTC.

Es necesario destacar, que para cada usuario se establece el localStream creado con getUserMedia(), de esta forma se puede observar también el *stream* del usuario que realiza la videoconferencia.

Finalmente se establece el *stream* remoto y local en los *tags* <video/> de cada usuario y se puede disfrutar de la comunicación, tal y como lo podemos observar en la Imagen 60.

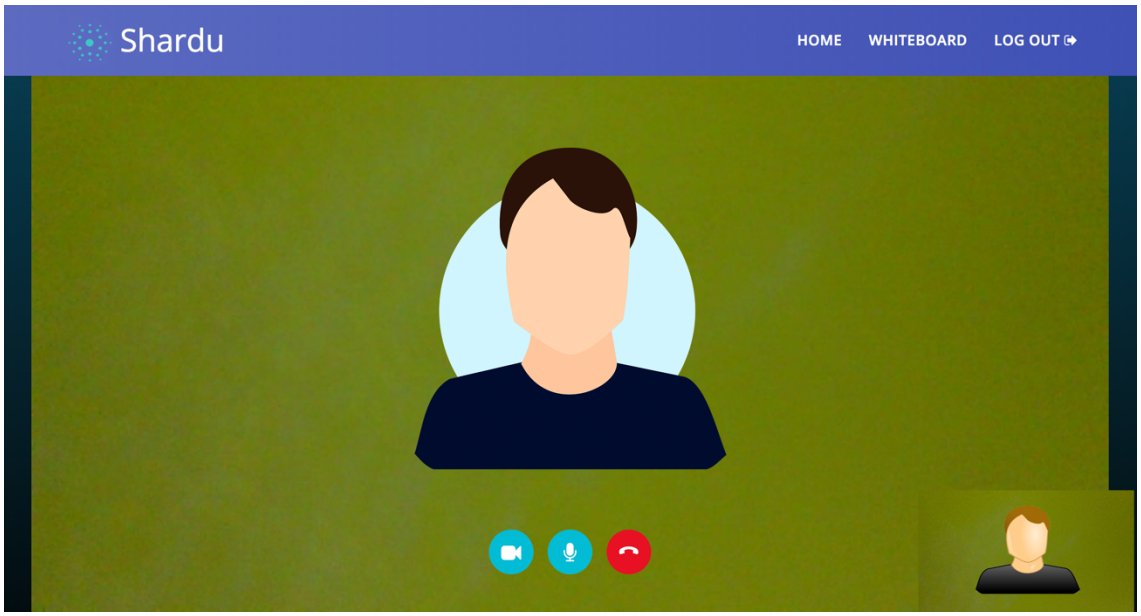

Imagen 60. Establecimiento de la videoconferencia.

Se han añadido algunas funciones extras, como permitir desconectar la cámara o micrófono y terminar la videoconferencia en cualquier momento de la comunicación.

8.6.2.3 Terminar la videoconferencia

En cualquier punto de la comunicación se puede ejecutar la acción de terminar la videoconferencia, si Alice o Bob realizan dicha acción, esta es ejecutada en ambos lados. Para hacer esto posible se utiliza el evento emitEndCall() y cuando este se ejecuta se hace un *reset* general de todos los componentes que están en ejecución, y se vuelve al estado inicial en ambos usuarios.

8.6.3 Pizarra virtual

La función principal de este componente es intercambiar datos en tiempo real mediante una pizarra virtual que está desarrollada principalmente con el elemento canvas y se encuentra en la *Scene* Whiteboard.

Los datos escritos en la pizarra se envían en modo *broadcast*, es decir todos los usuarios conectados podrán verlos.

Para conseguir este objetivo, se utilizan principalmente los módulos *Real Time Client* y *Real Time Server*, con en el *namespace* whiteboard.

Imagen 61. Pizarra virtual.

Se utiliza el mouse del ordenador como pincel para trazar el dibujo en la pizarra, cuando el usuario empieza a dibujar se toman las posiciones del pincel y se tienen en cuenta los diferentes eventos realizados la pizarra respecto al mouse, como son *mousedown*, mouseup, *mouseout* y *mousemove*, de esta forma estamos capturando cada movimiento para posteriormente dibujarlo en la pizarra utilizando las funciones moveTo y lineTo de canvas.

Para propagar estos datos a todos los usuarios, se utiliza el evento emitDrawing(), los datos enviados a través del evento son los siguientes:

- Posiciones del pincel (mouse), por defecto (0,0,0,0)
- Color del pincel, por defecto es *black*.

Cuando se envían los datos, el evento onDrawing() los recibe y llama a la función de dibujar, pasándole posiciones y el color del pincel, de esta forma cada vez que un usuario escribe en la pizarra estos datos se replican a todos los usuarios.

Además, presenta algunas herramientas extras, como:

- **Limpiar la pizarra**, cuando se ejecuta está acción, se llama a una función que establece el contexto del canvas al estado inicial, esta acción es enviada a todos

los usuarios a través del evento emitDrawing(), con el parámetro *clear* igual a true.

- **Cambiar el color del pincel**, es posible cambiar el color del pincel con la lista proporcionada en la interfaz y todos estos cambios se propagan a todos los usuarios.

9 Manual de uso

Para utilizar el sistema y todos los componentes que lo conforman es necesario registrase con un e-mail y contraseña, el sistema está pensado para ser muy fácil de usar, por ese motivo se ha tenido en cuenta cada detalle para que la experiencia del usuario del usuario sea positiva.

9.1 Registrar

El registro es obligatorio debido a que se plantea un sistema altamente escalable, y de cara a futuras implementaciones es probable que se necesite un usuario para gestionar los datos que le pertenecen, aunque la API soporta un usuario en modo invitado de momento no está implementado en el lado del cliente.

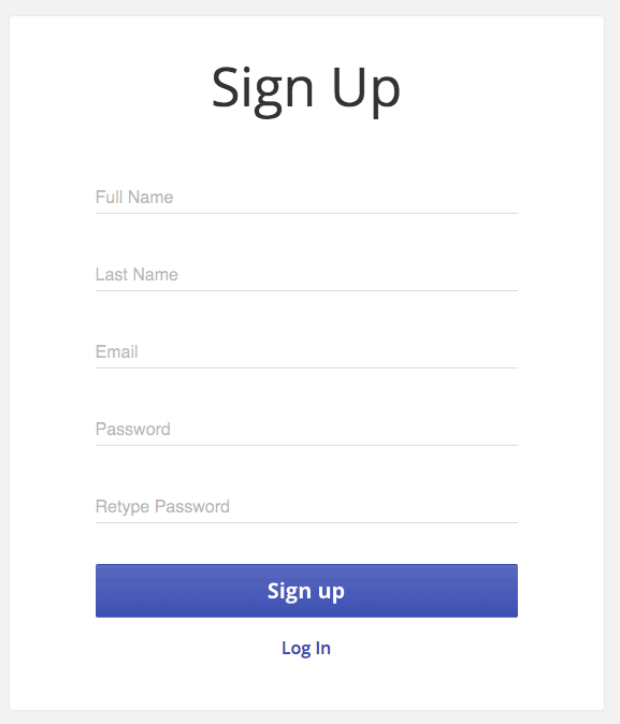

Imagen 62. Interfaz de registrar.

Se tiene cuenta todos los posibles errores al momento de crear un nuevo usuario, como pueden ser el email incorrecto, el usuario ya existe o la contraseña es muy corta. La mayor parte de problemas se solucionan con Json Schema Flow, específicamente con la herramienta validación, en el caso de un email duplicado, la API es el encargada de enviar el error para que el cliente proceda a mostrarlo, tal y como se puede observar en la imagen 63.

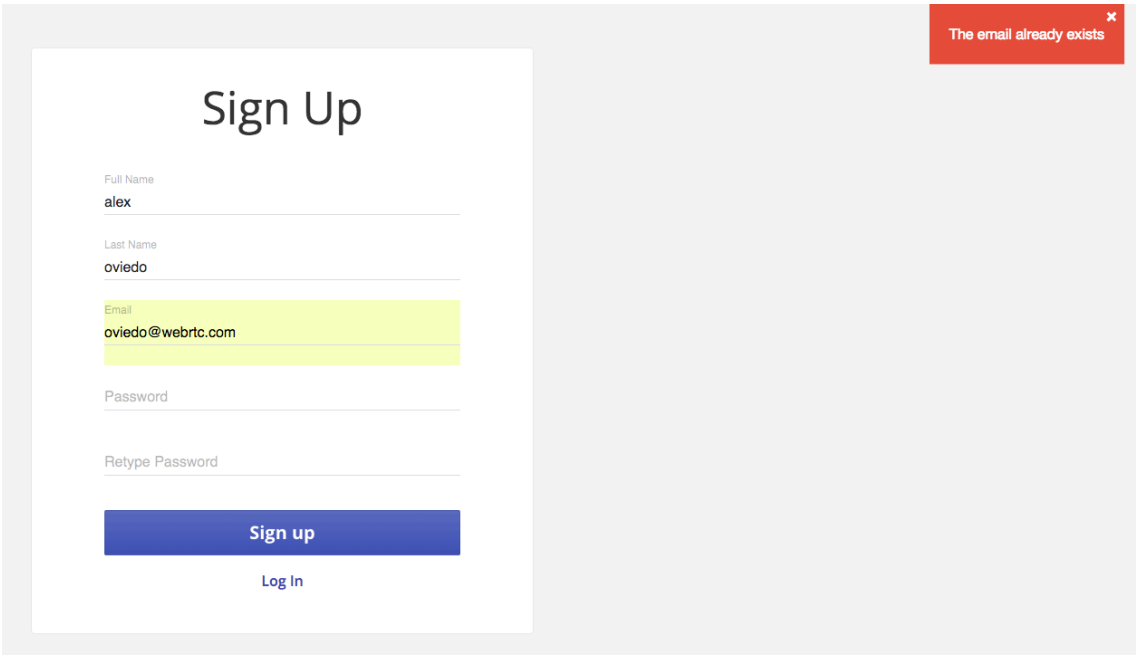

Imagen 63. Errores en el registro.

9.2 Iniciar sesión

Si el usuario ya se ha registrado previamente en el sistema, simplemente tiene que acceder al sistema con el e-mail y contraseña, como se puede observar en la Imagen 64.

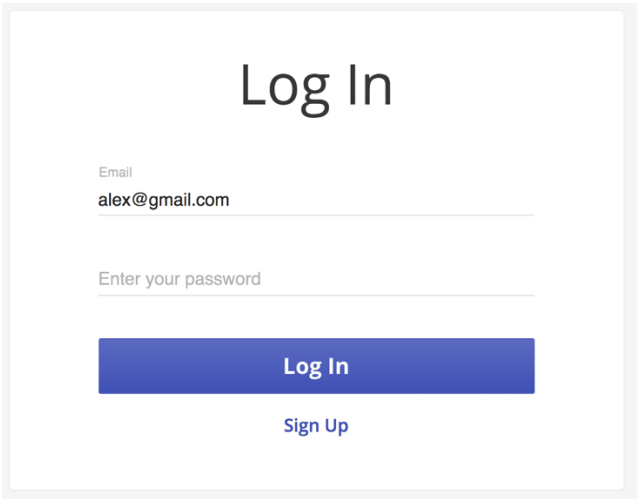

Imagen 64. Interfaz de login.

Una vez el usuario se ha registrado o ha iniciado sesión, es enviado automáticamente a la *Scene* Dashboard, esta contiene los componentes Videoconferencia y Mensajería instantánea, además se puede observar elementos nuevos en la TopBar, como Whiteboard y Logout, se debe a que la mayor parte de los componentes son privados, y cuando el usuario inicia sesión, estos se activan, tal y como se puede observar en la Imagen 65.

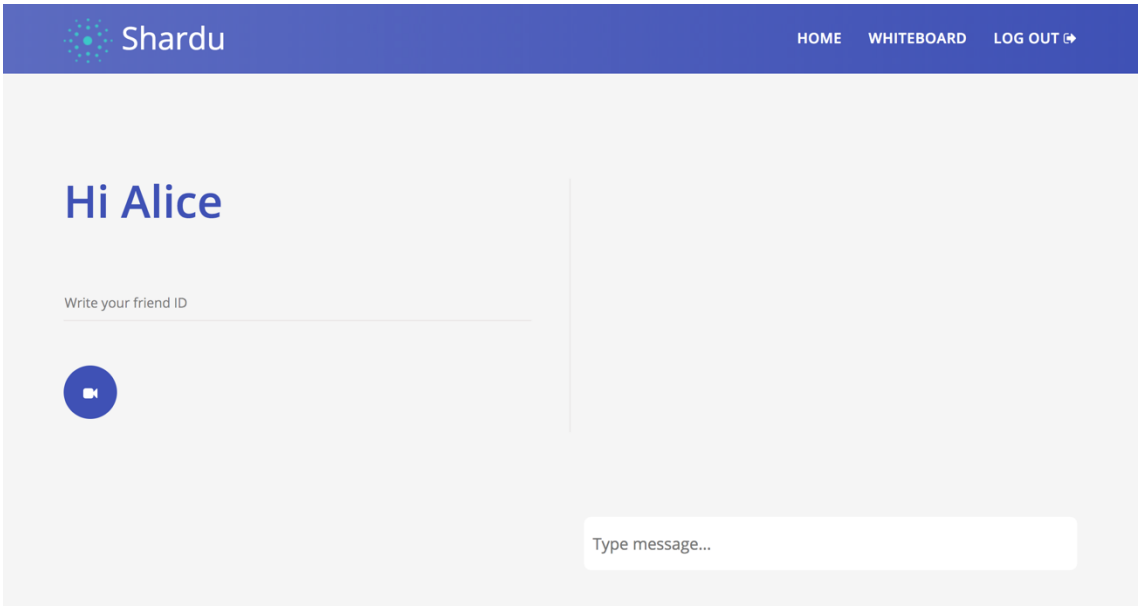

Imagen 65. Dashboard.

9.3 Videoconferencia

Para utilizar este componente, es necesario escribir el *friend* Id al que se desea llamar, seguidamente se envía la petición presionando el botón de llamar, el *friend* Id recibe la notificación de llamada, la acepta y se establece la comunicación, en caso contrario se cierra la llamada. Cuando la comunicación se ha establecido de forma correcta, el usuario puede ejecutar varias acciones como ocultar el video, apagar el micrófono o terminar la llamada.

9.4 Mensajería instantánea

Este componente es muy sencillo de usar, simplemente se escribe en el input el mensaje que se desea enviar, se presiona *enter* y el sistema lo envía en tiempo real a todos los usuarios conectados.

9.5 Pizarra virtual

Para utilizar este componente es necesario dirigirse a la *Scene* Whiteboard, una vez allí, el usuario tiene que escribir con el pincel lo que desee, posteriormente estos datos se comparten en tiempo real con todos los usuarios conectados en el sistema, el usuario puede ejecutar algunas acciones que se encuentran en la parte inferior como son, borrar la pizarra y cambiar el color del pincel.

10 Despliegue y Escalabilidad

Para el despliegue y escalabilidad del sistema se han utilizado varias herramientas con un propositivo diferente.

El despliegue va a depender del tipo de entorno en donde se ejecuta la aplicación, en este caso el sistema se divide en tres entornos:

- Local, entorno para desarrolladores donde trabajan directamente con el código fuente.
- Devel, entorno intermedio que se utiliza mayormente para testear y solventar problemas antes de pasar al entorno de producción.
- Prod, entorno de producción listo para ser utilizado por el usuario final.

10.1 AWS

La plataforma Amazon Web Services es el principal servicio de almacenamiento y gestión de la aplicación, cada módulo tiene una instancia asociada:

- Infraestructura del Frontend, instancia EC2 máquina Ubuntu.
- API, instancia EC2 máquina Ubuntu.
- DB, instancia RDS.
- *Real Time Server*, instancia EC2 máquina Ubuntu.

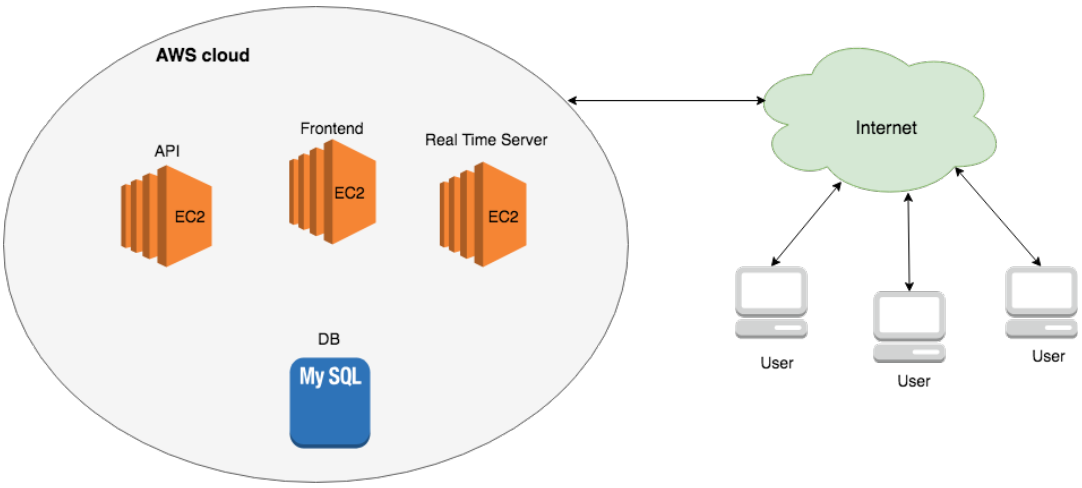

Imagen 66. Shardu en la nube.

10.2 Docker

Se ha creado un multicontenedor con Docker Compose para la Infraestructura del Backend y el *Real Time Server*, los servicios que están presentes son los siguientes:

- Base de datos.
- API.
- *Real Time Server*.

Cada servicio tiene una configuración específica, como la imagen a usar, el puerto donde se va a ejecutar, el volumen que presenta en la máquina local y las variables globales de entorno, se puede observar la configuración en la Imagen 67.

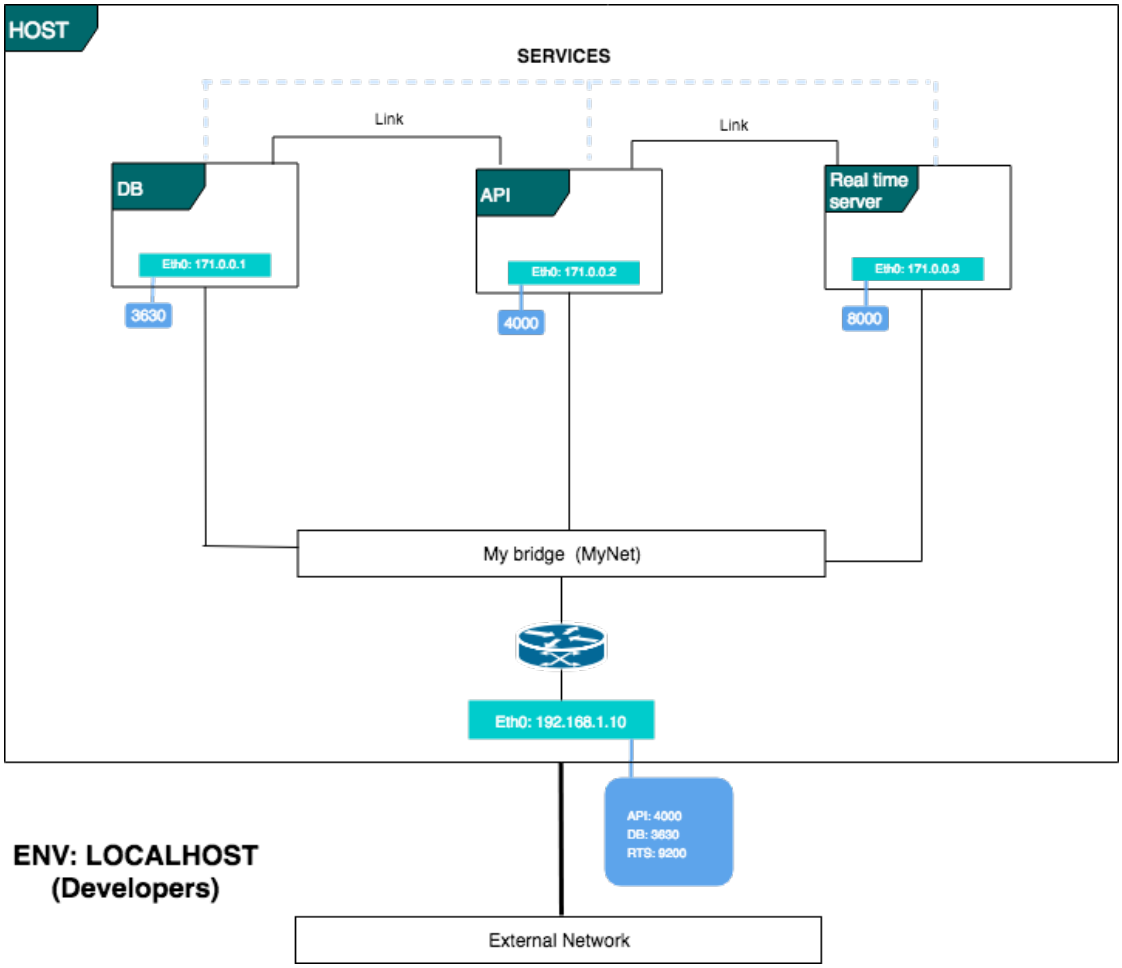

Imagen 67. Microservicios.

Existen dependencias entre servicios, en este caso la API depende del servicio DB para consumir los datos, por esa razón se crea un enlace entre ambos, así mismo el *Real Time Server* puede consumir los recursos de la API, para hacer esto posible se crea otro enlace entre este servicio y la API.

Todos los servicios se ejecutan sobre una red interna llamada MyNet y el tipo de red de Docker Compose se denomina *Bridge*.

Para poder utilizar cada servicio fuera del contenedor, se debe especificar el puerto de cada uno:

- API: 4000
- DB: 3630
- Real Time Server: 9200

Este multicontenedor se utiliza principalmente en el entorno local, y es útil para los desarrolladores, debido a que tan solo necesitan ejecutar el Docker Compose y tendrán en su máquina la Infraestructura del Backend y el *Real Time Server* montada, sin preocuparse de todas las dependencias que estos módulos presentan.

10.3 Kubernetes

Para el despliegue en los entornos Devel o Prod, se utiliza Kubernetes, es un sistema *open source* desarrollado por Google para la gestión de aplicaciones en contenedores, especialmente para contenedores Docker, también llamado “Orquestador de Docker”.

Las acciones principales que permite Kubernetes son [9]:

- Despliegue y Rollback, Kubernetes despliega los cambios y en caso de fallo hace un rollback automático
- Escalado y auto-escalado, en función del uso de CPU permite el escalado vertical de la aplicación de manera manual o automática.
- Monitorización, permite el monitoreo a través de una interfaz gráfica.
- Auto-reparación, en caso de fallo en un contenedor, puede reiniciarlo automáticamente.
- Gestión y configuración de *secrets*, los datos sensibles como *password* o claves ssh, se almacenan en kubernetes ocultas en un *secret*.

Con Kubernetes se puede programar y ejecutar la aplicaciones de los contendores que están situados en diferentes máquinas virtuales o físicas.

En este caso la aplicación cuenta con tres máquinas virtuales deferentes, DB, API y el *Real Time Server*, tal y como se puede observar en la imagen 68.

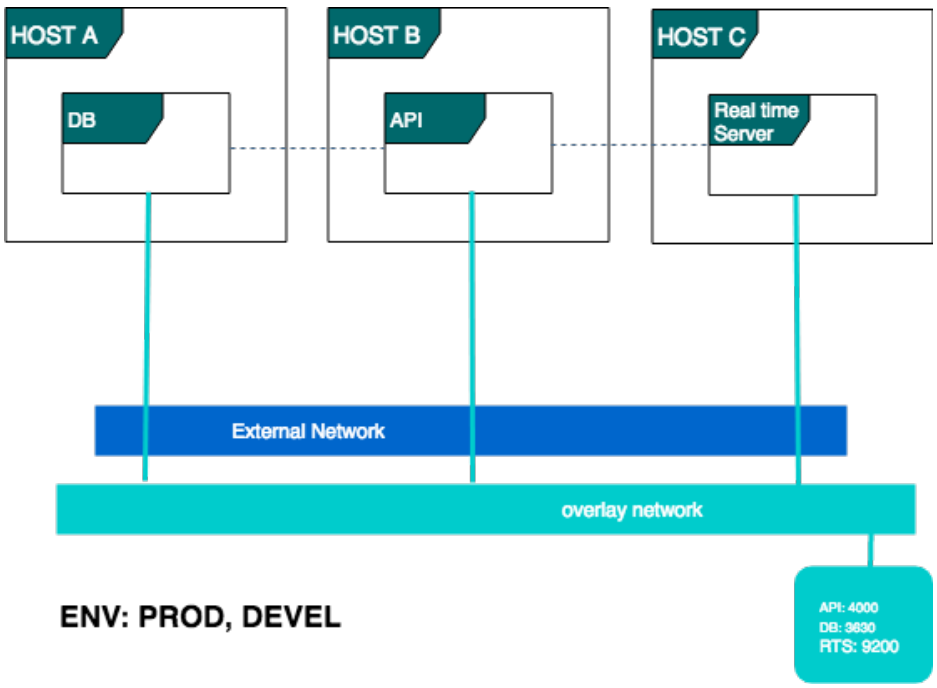

Imagen 68. Kubernetes

Kubernetes proporciona la infraestructura para construir un entorno de desarrollo centrado en el contenedor con varias funcionalidades que ayudan al sistema a corregir los posibles errores debido a la sobrecarga de usuarios, o errores específicos de cada servicio.

11 Resultados

Los resultados para cada componente han sido favorables, se ha conseguido cumplir con los objetivos fijados y por otra parte se ha dejado el sistema preparado para crear nuevas funcionalidades o mejorar las ya existentes.

Se han realizado varias pruebas en diferentes entornos y navegadores, consiguiendo los mismos resultados favorables, debido a que todas las tecnologías son compatibles con prácticamente todos los navegadores.

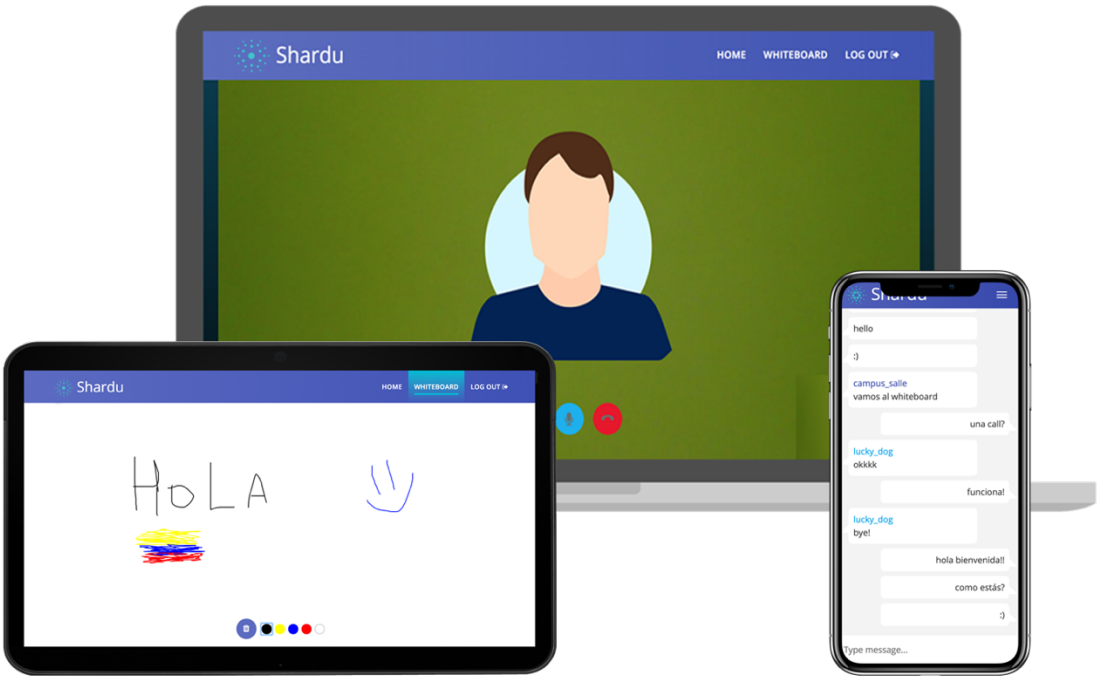

Imagen 69. Resultado final.

El sistema es muy fácil de utilizar y no requiere de extensos tutoriales, se ha pensado en cada detalle para que el usuario no tenga problemas al utilizar la aplicación, obteniendo así una experiencia de usuario muy favorable.

El diseño *responsive* desarrollado en la aplicación es muy importante para poder utilizar en múltiples dispositivos.

12 Estudio Económico

La mayoría del software utilizado en este proyecto es de código abierto, es decir no se paga para su uso. Excepto Sketch, el software de diseño que se ha utilizado para crear el prototipo. El valor de la licencia de este software es de 85€. También se ha comprado curso para el aprendizaje de Kubernetes [10] por un valor de 9,5€.

El proyecto se ha realizado en un MacBook pro y dos monitores de la marca Asus MX279H, con un valor de 1500€ y 280€ respectivamente. Con una amortización lineal del 25% anualmente sobre el precio inicial de ambos productos, tenemos:

- MackBook pro:

$$1500€ * 0{,}25 * \frac{6}{12} = 187{,}5€$$

- Monitor Asus:

$$280€ * 0{,}25 * \frac{6}{12} = 35€$$

Teniendo en cuenta el mercado laboral actual, el salario medio de un programador full-stack Semi senior ronda por los 40000€ al año en jornada completa.

La duración del proyecto ha sido de 6 meses a media jornada, partiendo del análisis anterior se puede llegar a la conclusión que el coste del programador es de 9996€.

Tabla7. Costes del proyecto.

Concepto	Cantidad	Valor (€)
Sketch	1	85
Monitor Asus	2	70
MackBook pro	1	187,5
Curso Udemy	1	9,5
Full-Stack Developer	1	9996
TOTAL		**10348**

El desglose en horas de todo el desarrollo del proyecto lo podemos ver en la Tabla 8.

Tabla 8. Desglose total de horas del proyecto.

Tarea	Tiempo (horas)
Planificación del proyecto	20
Diseño	20
Desarrollo del Frontend	140
Desarrollo del Backend	100
Desarrollo de los Módulos	80
Integración	20
Testeo y *bug fixing*	20
Documentación	80
TOTAL	**480**

13 Conclusiones

Ha sido un proyecto muy extenso tanto a nivel teórico como práctico debido a que se han aplicado diversas tecnologías las cuales hacen que el proyecto haya sido de alta motivación personal y profesional, con un resultado final que cumple con las expectativas planteadas, dejando al sistema listo para futuras mejoras o implementaciones.

La metodología planteada ha hecho que el desarrollo sea fácil de seguir y realizar, al ser un proyecto individual es muy importante saber gestionar el tiempo y buscar las herramientas adecuadas que ayuden a cumplir los objetivos.

Como se ha mostrado la tecnología WebRTC está en constante crecimiento y pretende ser una de las más importante en el sector de las comunicaciones, presenta múltiples ventajas respecto a sistemas actuales que realizan las mismas funciones, pero la principal ventaja es que no necesita de ningún software o plugin extra, lo cual transmite confianza al usuario ya que solamente tiene que acceder al navegador web para utilizar un servicio.

El proyecto está enfocado al campo docente, pero al tener mucha flexibilidad podría encajar perfectamente en las empresas de cualquier sector, algunos casos de uso podrían ser reuniones de empresa, mostrar el producto a un cliente o entrevistas a futuros empleados.

La videoconferencia se limita a ser ejecutada de forma *one-to-one,* por ese motivo el punto débil del proyecto, desde mi punto de vista, es no haber podido terminar el desarrollo de las videoconferencias en grupo, esta característica no entraba en el objetivo planteado pero pienso que podría hacer que el proyecto tenga un gran alcance.

Por otra parte, un punto fuerte es el Stack tecnológico, debido a que la mayoría de tecnologías son relativamente nuevas y respaldadas por una gran comunidad, teniendo así un gran soporte en el desarrollo.

Algunas de la líneas futuras del proyecto podrían ser:

- Terminar el desarrollo de las videoconferencias en grupo, que de momento solo está presente en la infraestructura del Backend y el en el sistema de Real Time.
- Crear un sistema de mensajería privada entre cada usuario.
- Compartir archivos.
- Implementar el modo de usuario invitado.

14 Referencias

[1] Vincent Driessen. *A successful Git branching model*. Recuperado de https://nvie.com/posts/a-successful-git-branching-model/.

[2] Restcookbook. *What are idempotent and/or safe methods?.* Recuperado de http://restcookbook.com/HTTP%20Methods/idempotency/.

[3] Sam Dutton. *Getting Started with WebRTC*. Recuperado de https://www.html5rocks.com/en/tutorials/webrtc/basics/.

[4] Alex Castrounis. *What is WebRTC and How Does it Work?.* Recuperado de https://www.innoarchitech.com/what-is-webrtc-and-how-does-it-work/.

[5] Alexis Mangin. *How to better organize your React applications?.* Recuperado de https://medium.com/@alexmngn/how-to-better-organize-your-react-applications-2fd3ea1920f1.

[6] Mozilla. *Web Api Reference*. Recuperado de https://developer.mozilla.org/es/docs/Web/API.

[7] Auth0. *Introduction to JSON Web Tokens*. Recuperado de https://jwt.io/introduction/.

[8] Auth0. *Understanding Refresh Tokens*. Recuperado de https://auth0.com/learn/refresh-tokens/.

[9] Juan María Fiz, Por qué todo apuesta por kubernetes. Recuperado de https://www.paradigmadigital.com/techbiz/por-que-todos-apuestan-por-kubernetes/.

[10] Level Up Kubernetes Program, Basit Mustafa, Tao W., James Lee. Kubernetes Cuse from a DevOps guru. Recuperado de https://www.udemy.com/kubernetes-from-a-devops-kubernetes-guru/.