

peeker

Aplicación web de videoconferencia

18 de junio de 2018

Mario garcía ramos

Desarrollo de aplicaciones web – IES Virgen de la Paz (2017 – 2018)



ÍNDICE

[ÍNDICE 1](#_Toc516863703)

[Justificación del proyecto y objetivos 3](#_Toc516863704)

[Introducción 4](#_Toc516863705)

[Tecnologías 4](#_Toc516863706)

[Metodología y desarrollo del proyecto 6](#_Toc516863707)

[Establecimiento de requisitos 6](#_Toc516863708)

[Planificación del proyecto 8](#_Toc516863709)

[Software empleado 10](#_Toc516863710)

[Estructura de ficheros 10](#_Toc516863711)

[Comunicación WebRTC 10](#_Toc516863712)

[Análisis del modelo (M) 13](#_Toc516863713)

[Usuario 13](#_Toc516863714)

[Análisis de la interfaz (V) 14](#_Toc516863715)

[Página principal 14](#_Toc516863716)

[Página de videollamada 15](#_Toc516863717)

[Página de llamada entrante 16](#_Toc516863718)

[Página principal en dispositivos móvil 18](#_Toc516863719)

[Página de video llamada en dispositivos móviles 19](#_Toc516863720)

[Página de llamada entrante en dispositivos móviles 20](#_Toc516863721)

[Análisis de clases (C) 21](#_Toc516863722)

[Chat y contactos 21](#_Toc516863723)

[Autenticación 28](#_Toc516863724)

[App 33](#_Toc516863725)

[Comunicación cliente – servidor mediante web sockets 35](#_Toc516863726)

[Arquitectura Angular 4 47](#_Toc516863727)

[Resultados y discusión 48](#_Toc516863728)

[Conclusiones 50](#_Toc516863729)

[Bibliografía y referencias 51](#_Toc516863730)

[Anexos/ Otros 53](#_Toc516863731)

[Programación funcional 53](#_Toc516863732)

[Gestión de base de datos con Mongoose 54](#_Toc516863733)

[Autenticación mediante tokens con JWT.io 55](#_Toc516863734)

[Manejo de seguridad con BCrypt JS 56](#_Toc516863735)

[Resultado estético 56](#_Toc516863736)

[Especificación del plan de pruebas 56](#_Toc516863737)

[TODOs: 59](#_Toc516863738)

[En Contacts Service añadir método get contacts para obtener el array, distinto del getcontacts de acceso a la bd 59](#_Toc516863739)

[Quitar todas las listas de usuarios y métodos que no tengan que ver de subcomponentes de contactscomponent 59](#_Toc516863740)

[Lo mismo con los atributos y métodos de auth 59](#_Toc516863741)

[En Eliminar de contactos eliminar de la BD también 59](#_Toc516863742)

[Actualizar diagramas con cambios 59](#_Toc516863743)

[Pasar lo del localstorage de signincomponent al authservice 59](#_Toc516863744)

[Guardar módulos y app en sí para producción y documentarlo 59](#_Toc516863745)

# Justificación del proyecto y objetivos

Escogí este proyecto gracias a mi inquietud por la tecnología que hace posible la comunicación entre las personas de forma remota y por la versatilidad que posee el desarrollo de aplicaciones web. En cuanto al diseño de la parte final que ve el usuario es sencillo situar los componentes o elementos a mostrar donde se desea:

En los últimos años, con el auge de los *smartphones* ha surgido la necesidad de adaptar el contenido de las webs a cantidades ingentes de tamaños de pantalla. Desde que separar las aplicaciones en una versión de escritorio y otra móvil ya no es una opción viable por la diversidad de dispositivos que nos rodean en la actualidad, han surgido varias tecnologías como CSS3, Bootstrap, IONIC que hacen posible la centralización de la aplicación web en un único software de manera que el contenido se puede adaptar a las dimensiones de la pantalla donde se va a mostrar la aplicación.

De esta curiosidad y de otras inquietudes como el aprendizaje de tecnologías o *frameworks* como Angular y Node JS ha nacido este proyecto.

En este proyecto se pretenden llevar a cabo las siguientes funcionalidades:

* Comunicación por videoconferencia
* Sistema de autenticación de usuarios
* Lista de amigos por usuario
* Integración de la aplicación de forma nativa para iOS y Android mediante IONIC.

De las cuales estarán disponibles para la presentación:

* La comunicación por videoconferencia, que tendrá la posibilidad de pausar la cámara web, silenciar el micrófono y adaptarse a la pantalla del usuario.
* El sistema de autenticación de usuarios mediante encriptación de contraseñas y tokens web.
* La funcionalidad relacionada con la posibilidad de tener una lista de contactos por usuario.

# Introducción

En este apartado se explica a grandes rasgos en qué consiste el proyecto y las tecnologías que emplea.

Este proyecto consiste en el desarrollo de una aplicación **MVC** (Modelo Vista Controlador) que permite la comunicación por video y audio entre dos personas la cual se basa en la combinación de varias tecnologías con el fin de demostrar hasta qué punto es posible conectarlas por distintas que sean.

## Tecnologías

En este proyecto se van a emplear las siguientes tecnologías:

* Como lenguajes de programación se emplearán JavaScript y TypeScript
* Angular (Angular 4) – Google. Para gestionar la aplicación en el lado del cliente
* Node JS – Node JS Foundation. Para emplearse a modo de servidor web.
* Express JS – Node JS Foundation. Para simplificar el desarrollo de las rutas.
* Ionic Framework – Para integrar la aplicación de forma nativa en los sistemas operativos iOS y Android.
* WebRTC – Para la comunicación multimedia que hace posible la videoconferencia.
* Socket.io – Para la comunicación entre clientes (intercambio de información sobre los ICE Candidates, sobre desconexiones, llamadas entrantes…).
* MongoDB – Como base de datos.
* Mongoose – Para gestionar la base de datos desde Node JS.
* BCryptJS – Para la encriptación de las contraseñas
* Jwt.io (JSON Web Token) – Para la gestión de las sesiones.
* Local Storage – Para almacenar el token en el lado del cliente

Muchas de ellas son módulos que se pueden instalar gracias al gestor de paquetes de una de las más importantes de las que se usan en el proyecto (**Node JS**).

Se trata de combinar por un lado tecnologías para la gestión del lado del cliente como **Angular 4**, la cual permite gestionar las rutas de la aplicación, los controladores y las vistas mediante el modelo que recibe del lado del servidor mediante Node JS y, más concretamente **Express JS** (ver apartado [Arquitectura Angular 4](#_Arquitectura_Angular_4) ).

Esta librería se empleará para administrar las rutas de la aplicación para aumentar la seguridad de la aplicación, así como controlar las llamadas asíncronas realizadas por el cliente a la hora de la obtención o el procesamiento de datos.

Express JS se sirve de la base de datos **MongoDB** para gestionar el modelo y la persistencia en el tiempo del mismo. Al no poder usarla de forma directa, se requiere de la ayuda de un módulo, del que gracias a Node JS, es posible hacer uso: **Mongoose**.

Mongoose es una librería que nos permite acceder a la base de datos MongoDB deseada y poder realizar cualquier tipo de operación que se pueda realizar sobre ella (ver [Gestión de base de datos con Mongoose](#_Gestión_de_base)).

Ya que se menciona la persistencia de datos, cabe destacar el modo en el que se van a tratar las contraseñas, puesto que es un tema que siempre hay que tener en cuenta cuando se pretende establecer seguridad en el mundo del software. A la hora de almacenar las contraseñas, estas son encriptadas mediante la librería **BCryptJS**, que nos facilita un conjunto de funcionalidades para afrontar el problema del robo de claves las de los usuarios.

Gracias a esta librería, las contraseñas son grabadas en la base de datos de forma que no es tan fácil acceder al conocimiento de las mismas, puesto que el inicio de sesión lo único que hace es encriptar la contraseña introducida, y solamente en el caso de que coincida con la asociada al usuario mediante el que se intenta acceder a la aplicación (que ya se halla encriptada) se le permitirá iniciar la sesión, para más información ver [Manejo de seguridad con BCrypt JS](#_Manejo_de_seguridad).

En lo relacionado con la autenticación, por muy segura que pueda llegar a ser la gestión de contraseñas, cuanto menos viajen entre el cliente y el servidor, menos probabilidades habrá de recibir ataques. Por lo que para cada inicio de sesión se crea un token web gracias a la librería **JWT.io**. De esta manera, el usuario mantendrá su sesión a lo largo del día sin tener que preocuparse de tener que iniciar sesión. Este token creado se almacena en el almacenamiento local o ***local storage*** del navegador y el servidor verifica por cada intento de acceso, si este ha expirado o sigue en vigencia (ver [Autenticación mediante tokens con JWT.io](#_Autenticación_mediante_tokens)).

Para la comunicación entre usuarios se va a emplear WebRTC, cuya lógica se explica en el apartado [Comunicación WebRTC](#_Comunicación_WebRTC).

Por último, para la implementación de la aplicación en dispositivos móviles iOS y Android de forma nativa, se hará uso de la herramienta **IONIC**.

# Metodología y desarrollo del proyecto

En este apartado se describen las metodologías y tecnologías empleadas en el desarrollo del proyecto.

## Establecimiento de requisitos

En esta sección se procede a analizar los requisitos del sistema. Para se definen los siguientes tipos de requisitos:

1. Funcionales (RF): aquellos requisitos dedicados exclusivamente a definir funcionales del sistema, es decir, cómo va a funcionar nuestro programa
2. Técnicos (RT): aquellos requisitos que no definen como funciona el programa sino cómo ha de estar hecho.
3. De interfaz (RI): aquellos requisitos que especifican como ha de ser la interfaz de usuario.

El catálogo de requisitos obtenido es el siguiente, estando en negrita los resueltos en el prototipo a presentar:

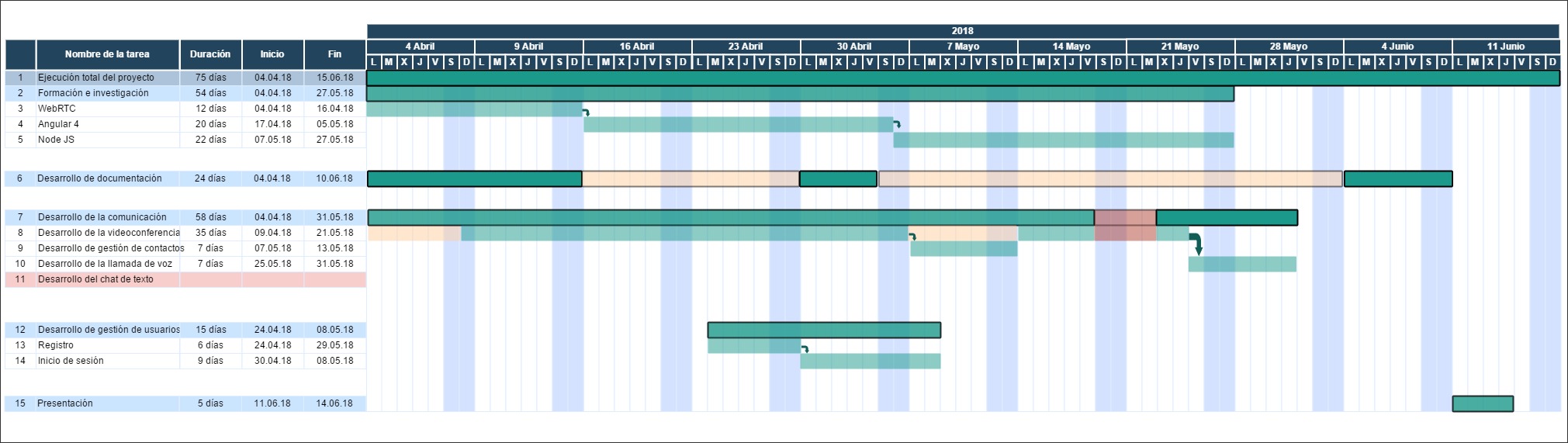
**Requisitos funcionales (RF)**

* + **RF1.** Se deben poder realizar videoconferencias.
  + **RF2.** Se deben poder realizar conferencias de audio.
  + **RF3.** Se deben poder realizar comunicación por medio de mensajes instantáneos de texto.
  + **RF4.** Los usuarios podrán darse de alta e ingresar en la aplicación.
  + **RF5.** Cada usuario podrá tener una lista de amigos con los que podrá contactar.
  + **RF6.** Se debe poder colgar la llamada una vez se ha establecido.
  + **RF7.** Se debe poder pausar la cámara una vez se ha establecido la llamada.
  + **RF8.** Se debe poder silenciar el micrófono una vez se ha establecido la llamada.
  + **RF8.** Se debe poder colgar la llamada cuando un usuario llama a otro, antes de que se haya establecido.
  + **RF9.** Se debe poder colgar la llamada cuando un usuario es llamado por otro, antes de que se haya establecido.
  + **RF10.** Se deberá poder registrar en un fichero el log de lo que ocurre en el servidor (conexiones, llamadas, mensajes…).
* **Requisitos técnicos (RT)**
  + **RT1.** En toda videoconferencia se deben percibir ambas cámaras.
  + **RT2.** Sólo se recibirá el audio del usuario remoto tanto en videoconferencias como en llamadas de audio.
  + **RT3.** Los mensajes instantáneos deben estar limitados a 200 caracteres.
  + **RT4.** Los usuarios accederán a la funcionalidad de la aplicación una vez hayan iniciado sesión.
  + **RT5.** Una vez en la llamada, los usuarios podrán silenciar su micrófono, de manera que el otro usuario no pueda escucharle.
  + **RT6.** Una vez en la llamada, los usuarios podrán pausar su cámara, de manera que el otro usuario no reciba datos de vídeo.
* **Requisitos de interfaz (RI)**
  + **RI1.** Al cargar la aplicación se debe mostrar una animación en la que el logo esté rodeado de un borde circular ambos de color gris con un color recorriendo este último.
  + **RI2.** La aplicación debe ser “***responsive***”, flexible al tamaño del dispositivo en el que se va a mostrar.
  + **RI3.** La disposición de los controles en la videoconferencia debe ser de aspecto simétrico.
  + **RI4.** Los controles de la videoconferencia deben estar siempre presentes en la versión de móvil.
  + **RI5.** En caso de no caber ambos vídeos por separado (el remoto y el local) se deberán superponer el uno al otro, quedando por encima el local, puesto que es notablemente más pequeño.

## Planificación del proyecto

En este apartado se describe la planificación del proyecto a lo largo del tiempo en función de las tareas. En color **verde** se representan las tareas realizadas con éxito, en **ámbar** las tareas pausadas y en **rojo** las tareas en las que se han tenido problemas y, por tanto, se ha producido algún retraso en el periodo de tiempo que abarca.

Para la planificación del proyecto he utilizado una planificación básica que me permite visualizar de forma rápida la duración de cada una de las tareas. Para esto he utilizado la herramienta draw.io, mediante la cual he podido representar de forma gráfica la planificación del proyecto. De esta manera queda al alcance de un vistazo el poder ver un retraso, una pausa o un resultado exitoso en la tarea realizada.



## Software empleado

El software empleado para la realización de este prototipo es el siguiente:

* Microsoft Office 2016 (Word y Power Point): Para la redacción de este documento y la elaboración de la presentación.
* Node JS
* Angular 4
* Visual Studio Code. Para la codificación de la página y la gestión de los ficheros de la misma.
* Express. Para facilitar el uso del desarrollo en el lado del servidor.
* Google Chrome, Mozilla Firefox, Microsoft Edge y Opera para la visualización del contenido web y para la resolución de errores mediante el debugging.
* Windows 10 como sistema operativo en el que se ha desarrollado el prototipo.
* Draw.io para la ilustración de diagramas.

## Estructura de ficheros

En este apartado se explica en detalle la estructura de carpetas y de ficheros que conforma el proyecto.

<https://webpack.js.org/configuration/>

y bundle.js

## Comunicación WebRTC

WebRTC es una tecnología de comunicación en tiempo real (Real-Time Communications o RTC) que hace posible la comunicación entre clientes mediante llamadas de voz, chat de vídeo, compartición de mensajes o incluso de archivos ***P2P*** (*Peer to peer*).

Para que la comunicación sea posible, primero han de intercambiar algunos datos como información sobre el dato a transferir (dimensiones y resolución de la cámara, códec de audio, si se transmite audio y video o sólo uno de los dos…), además de las direcciones IPs para la posterior conexión *P2P*.

A continuación, se explica la secuencia de acontecimientos que hacen posible la comunicación en este proyecto (ver también la figura 1).

Se presenta el siguiente escenario: Un **usuario 1** llama a un **usuario 2**:

1. Al conectarse un usuario a la aplicación, se guardan en una lista de usuarios online su nombre junto con un objeto ***socket*** asociado al flujo de datos que apunta al cliente.
2. Cuando el usuario 1 llama al usuario 2, se le solicita permiso para poder obtener el flujo de datos de la videocámara y el micrófono que estén conectados.
3. Al aceptar, el cliente del usuario 1 envía mediante la librería sockets.io un mensaje al servidor, el cual comunica con el usuario 2 en caso de estar presente en la lista de usuarios online.
4. Una vez el usuario 2 ha sido notificado, y coge la llamada, se realiza el mismo procedimiento que con los datos de cámara y vídeo del usuario 1.
5. Tras esto, el usuario 2 envía un mensaje al servidor para informar de que ya ha cogido la llamada, y este les comunica a los dos usuarios que ambos están listos para comunicar la comunicación.
6. Ambos usuarios crean un objeto **PeerConnection** y el usuario 1, al ser el que llamó, crea una oferta (**offer**) y se la envía al usuario 2 a través del servidor mediante el mismo sistema de mensajes basado en web sockets que se ha mencionado antes.
7. El usuario 1 genera un **ICE Candidate** (**Interactive Connectivity Establishment**) y es enviado a través del servidor al usuario 2, cuyo fin es establecer la conexión P2P.
8. Cuando el usuario 2 recibe el ICE Candidate, este envía el suyo a través de la creación de una respuesta (**answer**), la cual es enviada al usuario 1 a través del servidor.
9. Una vez ambos tienen los datos de conexión, la comunicación P2P comienza e intercambian entre ellos los datos y se pueden ver y escuchar el uno al otro.

En el apartado [Comunicación cliente – servidor mediante web sockets](#_Comunicación_cliente_–) se profundiza más en el tema a nivel técnico.

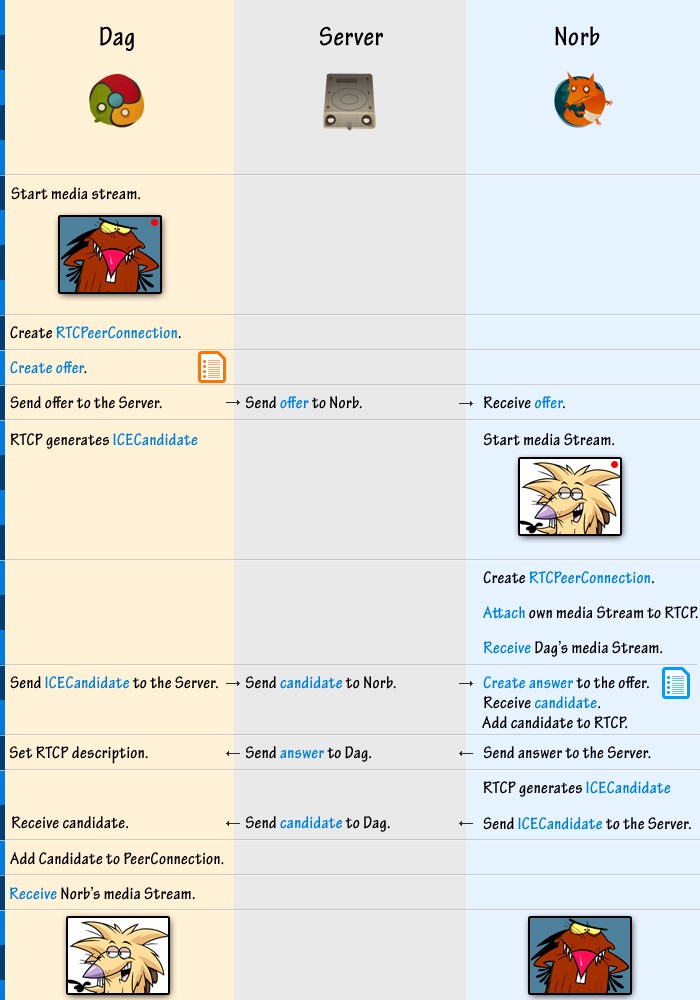


Figura 1. Fuente: <https://medium.com/@kanecohen/basics-of-webrtc-peer-connections-c1ed743de2f6>

## Análisis del modelo (M)

En este apartado se pretenden describir cada uno de los modelos que van a intervenir en la aplicación.

### Usuario

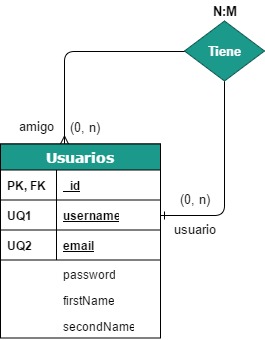
El usuario es el único modelo de la aplicación actualmente. Por eso y por fines estéticos, a continuación, se muestra un modelo lógico combinado con un modelo entidad relación de la tabla o colección **Usuarios**.

Figura 2

Al ser MongoDB la base de datos escogida, las claves primarias las escoge de forma automática, y no es necesario preocuparse demasiado por ese tema, por lo que la ***primary key*** o clave primaria será un atributo id.

Posee dos índices únicos, por lo que no se pueden repetir en ningún punto de la aplicación:

* **Username**: Es el atributo destinado a hacer las veces de nombre del usuario.
* **Email**: Atributo destinado a almacenar la dirección de correo electrónico mediante la cual el usuario accederá a la aplicación.

Además de esos índices, tendrá tres atributos:

* **Password**: Esta propiedad hace referencia a la contraseña del usuario. Una vez se registra un usuario, su contraseña se almacena encriptada, por lo que no será tan fácil acceder a ella.
* **FirstName**: Propiedad que porta el nombre de pila del usuario.
* **SecondName**: Propiedad que porta el apellido o los apellidos del usuario.

La única relación del modelo de la aplicación es una relación reflexiva sobre el propio usuario. Esta relación consiste en la posesión de contactos o amigos por parte de un usuario.

Por lo que, si se sigue el diagrama, se puede apreciar que un usuario puede llegar a tener entre ninguno y muchos amigos. O si se lee a la inversa, varios usuarios pueden ser contactos de ningún usuario o muchos.

## Análisis de la interfaz (V)

En este apartado se describen los aspectos principales de la interfaz que se diseñaron en el comienzo del proyecto. Si se desea ver el resultado final, ver [Resultado estético](#_Resultado_estético).

En primer lugar, se pensó en las vistas de la versión de escritorio. Todas ellas tienen unas características en común como pueden ser los componentes con los que el usuario puede interactuar para realizar todo tipo de acciones relacionadas con la videoconferencia como puede ser la lista de contactos y el menú principal.

### Página principal

En esta vista se muestran los componentes que se comparten entre las versiones cuyas dimensiones se equiparan a las de un monitor o pantalla de ordenador, como ya se ha mencionado antes. Lo único que varía es el componente principal del centro, donde se muestra el logo hasta que se cambie a otra vista.

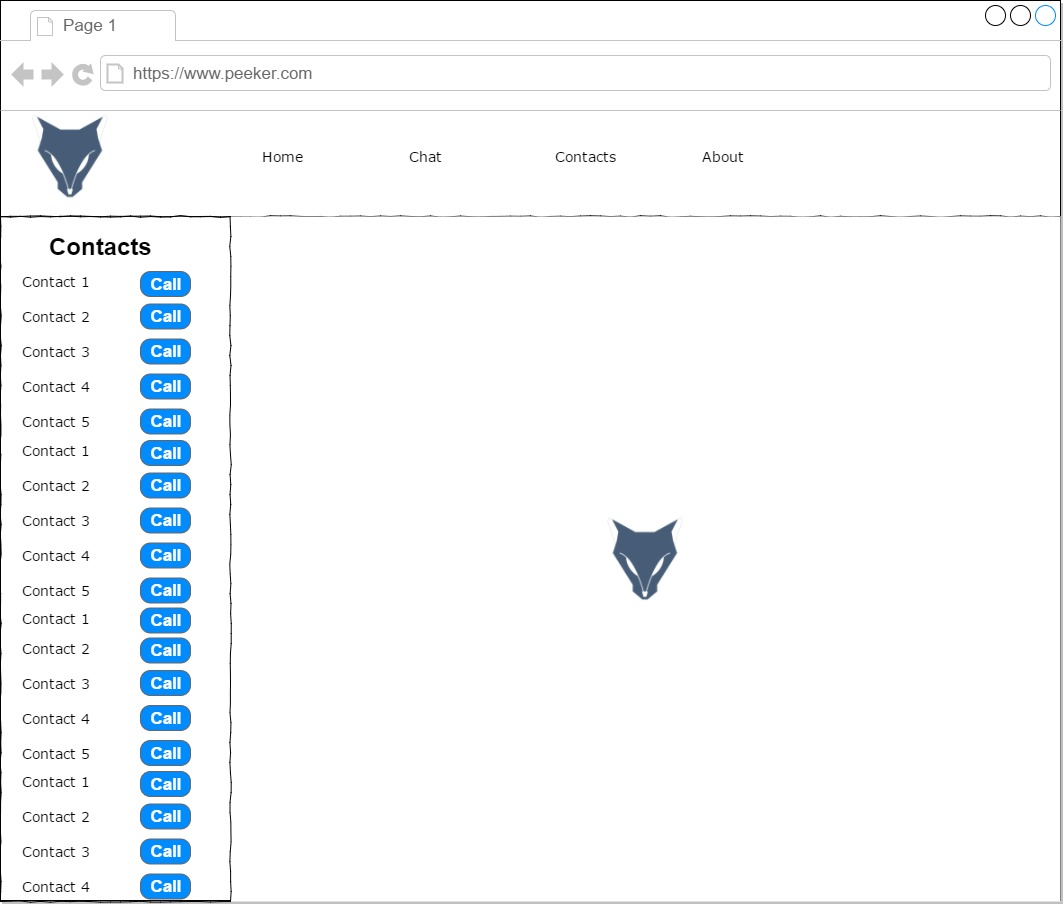


Figura 3

### Página de videollamada

En esta página se muestran los elementos de la videoconferencia. Por lo tanto, será accesible sólo cuando se esté en una videollamada o llamando a un usuario (con sus respectivas variantes, como el no ver el vídeo del usuario a llamar hasta que se acepta la llamada).

Se puede apreciar en la figura 4 cómo están distribuidos estos elementos:

Por un lado, tenemos los cuadros de vídeo en los cuales se van a ver reflejados el flujo de datos que capta la videocámara web. Se presenta como el más grande el vídeo **remoto**, pues es el que requiere mayor atención. Mientras que el video local (el propio del usuario) se muestra notablemente más pequeño. En caso de ser necesario por motivos del tamaño de la pantalla, este se superpondrá al del usuario remoto, puesto que tiene un tamaño inferior.

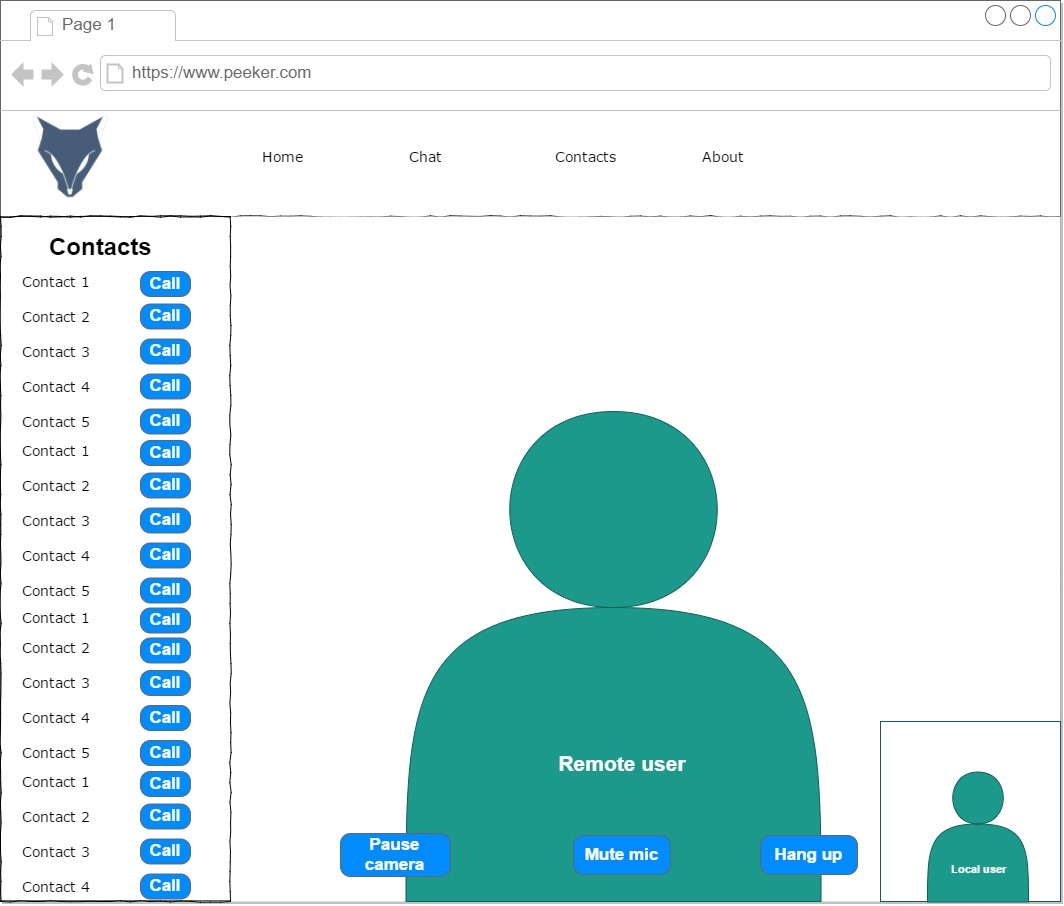
Por otro lado, se puede apreciar en la parte inferior de la misma figura un grupo de botones: el de **pausa** (que permite pausar la transmisión de datos proveniente de la propia cámara de vídeo), el de **silencio** (que deshabilita la grabación de audio por parte del propio micrófono) y el de **colgar** llamada (que, en caso de estar llamando, la cancelará).

Figura 4

### Página de llamada entrante

Esta vista se muestra cuando un usuario cualquiera nos está llamando. Está compuesta por la imagen de perfil del usuario llamante, una frase que indica el nombre del usuario que está llamando y un par de botones:

* **Aceptar**: Este botón permite descolgar el teléfono o coger la llamada, de manera que la siguiente vista a mostrar sería la anterior mostrada, pues el estado actual es en videoconferencia.
* **Rechazar**: Este botón le da la capacidad al usuario para colgar la llamada sin haberla cogido, o, dicho de otra manera, rechazar la llamada entrante.

Es posible que diste ligeramente de la versión final, puesto que en la versión final se incluyen accesorios e iconos mediante herramientas de terceros.

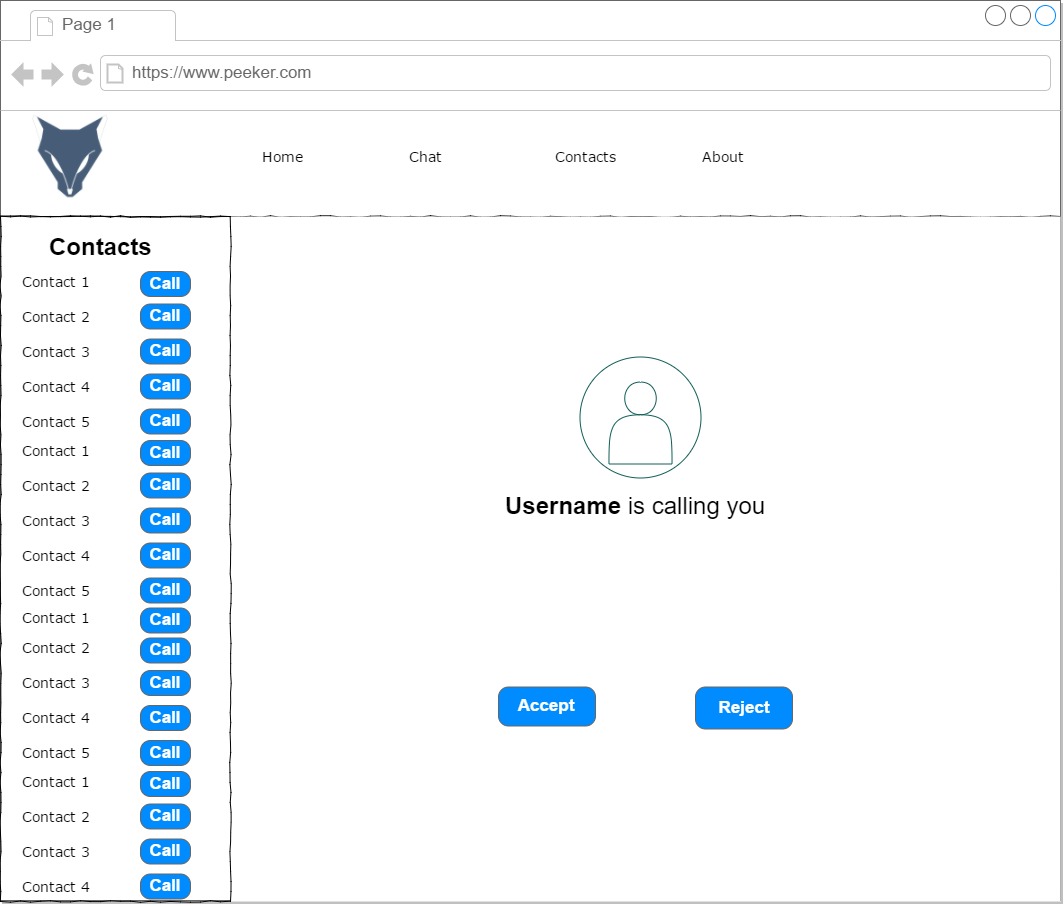


Figura 5

En segundo lugar, se pensó en adaptar el contenido a pantallas de menor tamaño, como pueden ser las de los dispositivos móviles y tablets. Por lo que a continuación se muestra cómo se verían las vistas principales en un dispositivo móvil actual.

### Página principal en dispositivos móvil

En la figura 6 se puede ver cómo ha desaparecido el menú principal para contraerse en un botón de “bocadillo”, que, al ser presionado, desplegará cada una de sus opciones.

La pantalla principal se basa en la lista de contactos, porque al fin y al cabo la acción que en más ocasiones se va a llevar a cabo es la de realizar videoconferencias o charlas de texto.

En futuras versiones se mostrarán los chats abiertos con los contactos con los que se ha mantenido alguna conversación de cualquier tipo (video conferencia, llamada de audio o chat de texto).



Figura 6

### Página de video llamada en dispositivos móviles

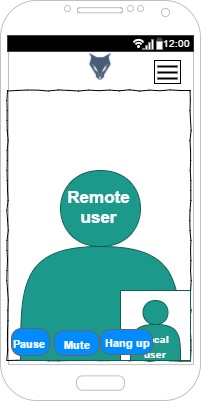
En la figura 7 se presenta la vista que representa la aplicación una vez se ha iniciado una videoconferencia con un contacto.

Figura 7

Comparte la característica del menú contraído con la vista anterior y distribuye los distintos elementos en la sección principal de la pantalla de la siguiente manera:

* **Video remoto**. Como se puede apreciar en la imagen, ocupa la totalidad de la sección principal de la vista, tanto de ancho como de alto. Debido al grande tamaño de las dimensiones, está en un último plano, pues no se pierde gran parte de la información y es necesario para poder apreciar el resto de los elementos.
* **Video local**. Está situado en un segundo plano, puesto que, si estuviera en una capa inferior a la del video remoto, sería imposible de ver. Es por esto, y que la información que contiene no es tan relevante como la del otro vídeo, que se muestra superpuesto respecto al remoto.
* **Panel de botones**. En este panel se encuentran los botones ya explicados en la página de videoconferencia de escritorio, por lo que simplemente cabe destacar que han de estar dispuestos simétricamente en la pantalla (en el *mockup* o maqueta no cumple la norma para que se visualice bien la cámara local).

### Página de llamada entrante en dispositivos móviles

En esta vista se puede apreciar el minimalismo que las versiones móviles obligan a alcanzar. Como ya se ha visto en la vista anterior, en los dispositivos móviles la lista de contactos desaparece por completo, puesto que la atención en el preciso instante en que se está mostrando la pantalla, va dirigida al propósito de la vista en sí.

En este caso la idea es indicar de una forma clara y sencilla que hay un usuario intentando establecer contacto audiovisual con el usuario que porta el dispositivo visualizado en la figura 8. Por lo que la complejidad no tiene cabida en este tipo de interfaces gráficas. Consta de los mismos componentes que la anterior versión de escritorio mencionada: la foto de perfil del usuario, su nombre y dos botones para descolgar el teléfono o rechazar la llamada.

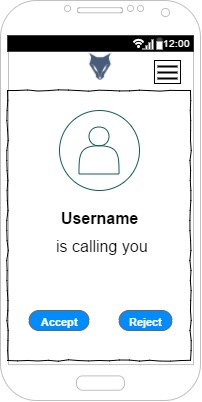
En un futuro se implementará un nuevo diseño que permita acceder a cada una de las funcionalidades (dentro de lo posible) desde una sola pantalla con una simple distinción de gestos ejercidos sobre la pantalla del dispositivo. Como puede ser deslizar de izquierda a derecha y desplegar la lista de contactos, y una vez en ella si se realiza el mismo gesto sobre un contacto de la lista se abriría un chat.

Figura 8

## Análisis de clases (C)

En este apartado se explican en detalle las clases más relevantes, que están escritas en el lenguaje ***TypeScript*** que son empleadas a modo de ***controladores***. Son también denominadas componentes de Angular, aunque entre ellas se van a mostrar también otros tipos de controladores, como pueden ser ficheros de rutas o ficheros de servicios.

Para comprender mejor cómo funciona Angular 4 y el modo en el que está integrado, ver apartado [Interactividad con Angular 4](#_Interactividad_con_Angular).

### Chat y contactos

En primer lugar, se presentan los controladores del ámbito de los contactos. En la figura 9 se muestran los controladores relacionados con el chat.

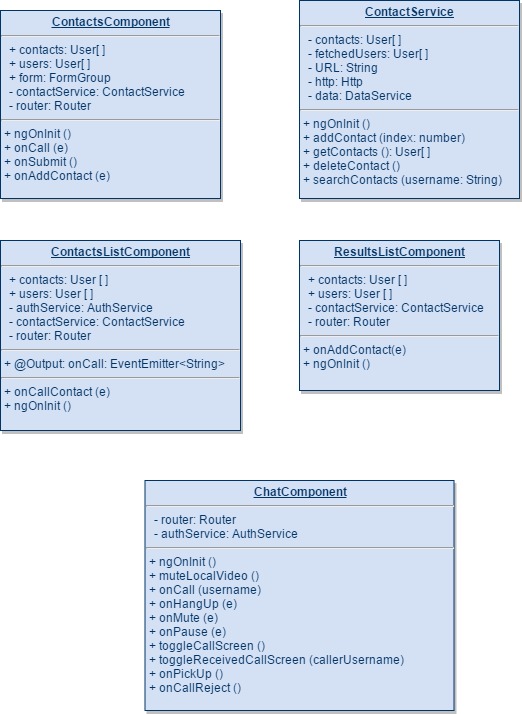


Figura 9

A continuación, se muestra en detalle la información mostrada en el diagrama anterior.

#### ResultsListComponent

Este componente contiene un listado de los usuarios resultantes de una búsqueda en el campo habilitado para ello. Cada uno de los elementos se asocia a un resultado de la consulta hecha. Esta consulta consiste en la selección de registros de la tabla de los usuarios cuyo campo nombre (***username***) contenga la cadena introducida. En la siguiente tabla se muestran las propiedades o atributos:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *ResultsListComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *users* | Lista de usuarios resultantes de la búsqueda. | User [] |

Tabla

A continuación, se muestran los métodos del componente.

|  |  |  |
| --- | --- | --- |
| **Métodos de *ResultsListComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *onAddContact* | Se ejecuta cuando el usuario hace clic sobre el botón de agregar perteneciente a un resultado de la búsqueda. Emite un evento para transmitírselo al componente padre final que es el ***ContactsComponent***, donde debe ir toda la funcionalidad relacionada con los contactos. | Vacío |

Tabla

#### ChatComponent

Este componente se encarga de la gestión de todas las acciones relacionadas con el chat. Como se puede apreciar en la figura 9, sus métodos están dedicados al manejo de los eventos y todo tipo de acciones relacionadas con los componentes que abarca. Se pueden ver las propiedades pertenecientes al controlador en la tabla 3.

|  |  |  |
| --- | --- | --- |
| **Propiedades de *ChatComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *router* | Permite la navegación entre las rutas establecidas en el fichero controlador de rutas. | Router |
| *authService* | Servicio que permite realizar acciones relacionadas con la autenticación, como pueden ser cerrar la sesión actual, obtener el token de sesión, obtener el nombre de usuario… | AuthService |

Tabla

Caben destacar los métodos de la clase que ejercen las acciones mencionadas, están presentes en la tabla 4.

|  |  |  |
| --- | --- | --- |
| **Métodos de *ChatComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *muteLocalVideo* | Silencia el vídeo local para escuchar sólo al remoto. | Vacío |
| *onCall* | Se ejecuta cuando el usuario hace clic sobre el botón de llamar de un contacto. Inicia la llamada con dicho usuario. | Vacío |
| *onHangUp* | Se ejecuta cuando el usuario hace clic sobre el botón de colgar la llamada. Finaliza la llamada actual o cancela la llamada que aún está en espera de que el usuario remoto responda. | Vacío |
| *onMute* | Se ejecuta cuando el usuario hace clic sobre el botón de silenciar el micrófono. Mutea el micrófono, por lo que el usuario remoto deja de escuchar al que ejerció la acción. | Vacío |
| *onPause* | Se ejecuta cuando el usuario hace clic sobre el botón de pausar la videocámara web. Pausa el vídeo a emitir, por lo que el usuario remoto deja de ver al usuario que ejerció la acción. | Vacío |
| *toggleCallScreen* | Activa o desactiva la pantalla de videoconferencia. Si está activada la oculta, y si está desactivada la muestra. | Vacío |
| *toggleReceivedCallScreen* | Activa o desactiva la pantalla de llamada entrante. Si está activada la oculta, y si está desactivada la muestra. | Vacío |
| *onPickUp* | Se ejecuta cuando el usuario hace clic sobre el botón de coger la llamada entrante. Descuelga la llamada. | Vacío |
| *onCallReject* | Se ejecuta cuando el usuario hace clic sobre el botón de rechazar la llamada entrante. Rechaza la llamada. | Vacío |

Tabla

#### ContactsComponent

Este componente es el conjunto formado otros dos: ***ContactsListComponent*** y ***ResultsListComponent***. Es el que contiene la lista de contactos y la búsqueda de estos.

#### ContactsListComponent

Este componente contiene un listado de contactos actual del usuario. Cada uno de los elementos se asocia a un usuario añadido a la lista, en los cuales se muestran el nombre del usuario y un botón para llamarle. Sus propiedades son las siguientes:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *ContactsListComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *onCall* | Evento que es disparado cuando el usuario hace clic sobre el botón de llamar a un contacto a través de la función onCallContact. Contiene el nombre del usuario al que se desea llamar y, una vez llega al componente adecuado (ChatComponent) se realiza la acción de la llamada en sí. | EventEmitter<String> |
| *contacts* | Lista de usuarios que son contactos del usuario. | User[] |
| *contactService* | Servicio que permite realizar acciones relacionadas con la gestión de contactos, como pueden ser buscar, agregar, eliminar y obtener el listado. | ContactService |
| *router* | Permite la navegación entre las rutas establecidas en el fichero controlador de rutas. | Router |
| *authService* | Servicio que permite realizar acciones relacionadas con la autenticación, como pueden ser cerrar la sesión actual, obtener el token de sesión, obtener el nombre de usuario… | AuthService |

Tabla

Su único método figura en la tabla 6.

|  |  |  |
| --- | --- | --- |
| **Métodos de *ContactsListComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *onCallContact* | Se ejecuta cuando el usuario hace clic sobre el botón de llamar de un contacto. Emite un evento para transmitírselo al componente padre final que es el chat, donde debe ir toda la funcionalidad relacionada con la comunicación. | Vacío |

Tabla

#### ContactService

Este controlador funciona a modo de servicio. Permite gestionar todas las acciones relacionadas con los contactos como se puede ver a continuación, en la tabla nº 7.

|  |  |  |
| --- | --- | --- |
| **Métodos de *ContactService*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *addContact* | Añade el contacto recibido por parámetro a la lista actual de contactos almacenada en memoria. A su vez realiza una llamada asíncrona al servidor de manera que este contacto persiste en la base de datos. | Vacío |
| *getContacts* | Obtiene los contactos actualmente almacenados en memoria. | User[] |
| *selectContacts* | Obtiene los contactos actualmente almacenados en la base de datos. | User[] |
| *deleteContact* | Elimina el usuario recibido por parámetro de la lista de contactos y de la base de datos. | Vacío |
| *searchContacts* | Realiza una llamada asíncrona que ataca al servidor con el fin de obtener un listado de usuarios cuyo campo nombre (***username***) contenga la cadena que es enviada. | Observable |

Tabla

Una vez vistos los métodos del servicio, se presentan las propiedades de este en la siguiente tabla:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *ContactService*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *contacts* | Lista que contiene los contactos del usuario actual. Está situada en el servicio para acceder a ella desde cualquier componente que lo necesite para realizar operaciones de consulta, inserción, modificación o eliminación de los contactos. | User[] |
| *fetchedUsers* | Lista que contiene los usuarios resultantes de la búsqueda del componente hijo ResultsListComponent. | User[] |
| *URL* | Dirección URL base que contiene el dominio y el puerto en caso de tenerlo obtenida del servivio DataService. | String |
| *http* | Esta propiedad habilita al servicio a realizar llamadas asíncronas a direcciones URL. En este caso, se empleará para atacar al servidor y realizar operaciones de consulta, inserción, modificación o eliminación de los contactos sobre la base de datos. | Http |
| *Data* | Esta propiedad es un servicio común a todos los componentes que deseen compartir los mismos datos. En este caso es empleado para acceder en tiempo real a la URL donde se aloja el servidor (para que todos los servicios que hagan uso de llamadas asíncronas accedan siempre a la misma URL y, en caso de variar, que el código sea más mantenible y escalable). | DataService |

Tabla

### Autenticación

Una vez presentados los controladores relacionados con el chat y los contactos, se van a explicar cada uno de los que componen el ámbito de la autenticación. En la figura 10 se puede ver el diagrama de clases que lo representa de forma gráfica.

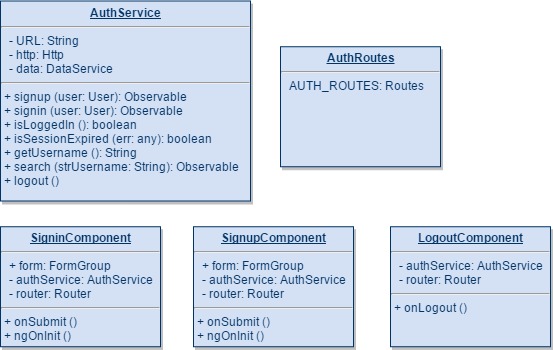


Figura 10

A continuación, se explica específicamente lo visto en la figura 10.

#### SignupComponent

Este componente se encarga del registro de nuevos usuarios en el sistema. A continuación, se muestran algunos de sus métodos.

|  |  |  |
| --- | --- | --- |
| **Métodos de *SignupComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *onSubmit* | Este método es ejecutado cuando el usuario hace clic sobre el botón de registro o envío de datos. Realiza la llamada al servidor mediante el atributo ***authService***, que se encarga de realizar todo este tipo de servicios. | Vacío |

Tabla

Una vez vistas los métodos del componente, se pueden ver sus propiedades:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *SignupComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *form* | Objeto que permite la gestión del formulario: desde la inicialización de valores por defecto, pasando por la validación de cada uno de los datos que contiene hasta la obtención de los mismos. | FormGroup |
| *authService* | Servicio que permite realizar acciones relacionadas con la autenticación, como pueden ser cerrar la sesión actual, obtener el token de sesión, obtener el nombre de usuario… En este caso será empleado única y exclusivamente para el alta de usuarios y comprobar si hay una sesión ya existente. | AuthService |
| *router* | Permite la navegación entre las rutas establecidas en el fichero controlador de rutas. | Router |

Tabla

#### SigninComponent

Este controlador es el componente que gestiona el inicio de sesión. En la tabla 11 se pueden ver algunos de sus métodos:

|  |  |  |
| --- | --- | --- |
| **Métodos de *SigninComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *onSubmit* | Este método es ejecutado cuando el usuario hace clic sobre el botón de inicio de sesión. Realiza la llamada al servidor mediante el atributo ***authService***, que se encarga de realizar todo este tipo de servicios. | Vacío |

Tabla

Tras ver el único método que tiene este componente, pasamos a ver algunas de sus propiedades:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *SigninComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *form* | Objeto que permite la gestión del formulario: desde la inicialización de valores por defecto, pasando por la validación de cada uno de los datos que contiene hasta la obtención de los mismos. | FormGroup |
| *authService* | Servicio que permite realizar acciones relacionadas con la autenticación, como pueden ser cerrar la sesión actual, obtener el token de sesión, obtener el nombre de usuario… En este caso será empleado única y exclusivamente para el alta de usuarios y comprobar si hay una sesión ya existente. | AuthService |
| *router* | Permite la navegación entre las rutas establecidas en el fichero controlador de rutas. | Router |

Tabla

#### LogoutComponent

El cometido de este componente es finalizar la sesión del usuario logeado que ejerce la acción. Sus métodos son los siguientes:

|  |  |  |
| --- | --- | --- |
| **Métodos de *LogoutComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *onLogout* | Este método es ejecutado cuando el usuario hace clic sobre el botón de cerrar sesión. Realiza la llamada al servidor mediante el atributo ***authService***, que se encarga de realizar todo este tipo de servicios. En este caso, finaliza la sesión del usuario eliminando sus datos del almacenamiento local o ***local storage*** (entre ellos se encuentra el token a través del cual se mantiene la sesión abierta). | Vacío |

Tabla

A continuación, se muestran las propiedades del componente:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *LogoutComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *authService* | Servicio que permite realizar acciones relacionadas con la autenticación, como pueden ser cerrar la sesión actual, obtener el token de sesión, obtener el nombre de usuario… En este caso sirve para cerrar la sesión del usuario mediante la función onLogout de este componente. | AuthService |
| *router* | Permite la navegación entre las rutas establecidas en el fichero controlador de rutas. | Router |

Tabla

#### AuthRoutes

Este archivo contiene las rutas relacionadas con la autenticación. Todas ellas tienen en común el prefijo “***auth/***”. Por ejemplo, la ruta al inicio de sesión sería tal que así: “***/auth/signin***”.

#### AuthService

Este controlador presta servicios relacionados con la autenticación, como pueden ser el registro, el inicio de sesión o la finalización de esta. Para ver en detalle sus métodos, basta con echar un vistazo a la tabla 15.

|  |  |  |
| --- | --- | --- |
| **Métodos de *AuthService*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *signup* | Este método está pensado para ser ejecutado cuando el componente de alta de usuarios requiere sus funciones. Realiza una llamada asíncrona al servidor para registrar el usuario recibido por parámetro. | Observable |
| *signin* | Este método se ejecuta cuando el componente de alta de usuarios requiere sus funciones. Realiza una llamada asíncrona al servidor para registrar el usuario recibido por parámetro. | Observable |
| *isLoggedIn* | Verifica si el usuario actual ha iniciado sesión. | Boolean |
| *isSessionExpired* | Verifica si la sesión del usuario actual ha caducado en función del token que se encuentra en el almacenamiento local del navegador. | Boolean |
| *getUsername* | Obtiene el nombre de usuario del almacenamiento local del navegador. | String |
| *search* | Realiza una llamada asíncrona al servidor para obtener los usuarios cuyo campo ***username*** contiene la cadena recibida (la que el usuario introduce en el campo de búsqueda). | Observable |
| *logout* | Pone fin a la sesión del usuario actual eliminando del almacenamiento local del navegador los datos relacionados con esta. | Vacío |

Tabla

A continuación, se muestran las propiedades del servicio:

|  |  |  |
| --- | --- | --- |
| **Propiedades de *AuthComponent*** | | |
| **Nombre** | **Descripción** | **Tipo** |
| *URL* | Dirección URL base que contiene el dominio y el puerto en caso de tenerlo obtenida del servivio DataService. | String |
| *http* | Esta propiedad habilita al servicio a realizar llamadas asíncronas a direcciones URL. En este caso, se empleará para atacar al servidor y realizar operaciones de consulta, inserción, modificación o eliminación de los contactos sobre la base de datos. | Http |
| *Data* | Esta propiedad es un servicio común a todos los componentes que deseen compartir los mismos datos. En este caso es empleado para acceder en tiempo real a la URL donde se aloja el servidor (para que todos los servicios que hagan uso de llamadas asíncronas accedan siempre a la misma URL y, en caso de variar, que el código sea más mantenible y escalable). | DataService |

Tabla

### App

Por último, caben destacar el resto de los controladores que componen la aplicación en el lado del cliente.

Los componentes pertenecientes a este ámbito carecen de funcionalidad, pues son componentes que atribuyen solamente la parte visual, por lo que la explicación de estos será breve.

#### AppComponent

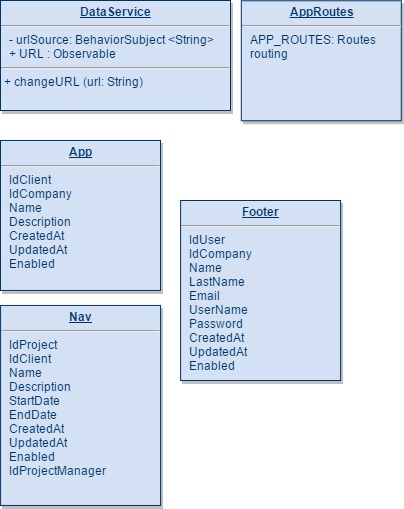
Componente que contiene la aplicación. Los componentes que abarca de forma directa, es decir, que son hijos directos de este, son el ***NavComponent*** y el ***FooterComponent***. La parte que varía de la aplicación se encuentra en el componente basado en la ruta actual.

#### NavComponent

Componente que contiene la barra superior de navegación, lo que compone el menú principal de la aplicación.

#### FooterComponent

Componente que contiene el pie de página donde figura el logo y datos de uso, términos, condiciones de uso, etc.



Figura

#### DataService

Este controlador ofrece el servicio de mantener cada uno de los datos que deben ser comunes para todos los componentes en los que se necesiten. Un ejemplo ya mencionado anteriormente es la dirección URL asociada al dominio y puerto por el que se accede de forma asíncrona. Este servicio hace la aplicación mucho más mantenible y escalable.

#### AppRoutes

Este fichero contiene las rutas de la aplicación, que son asociadas a componentes o a vistas concretas. En función de las rutas establecidas en él, la parte dinámica de la aplicación contenida en el componente AppComponent, ***router-outlet,*** mostrará un componente u otro.

#### AppModule

En este archivo se declaran todos los controladores que se van a usar en la aplicación para que sean accesibles desde cualquiera de ellos a la hora de importarlos. Por un lado, están las ***declarations***, donde se incluyen los componentes, y por otro lado están los ***providers***, que contienen los servicios o inyectables.

## Comunicación cliente – servidor mediante web sockets

La comunicación entre el cliente y el servidor relacionada con las videoconferencias está basada en *web-sockets.*

Ambos pueden ponerse a la escucha gracias a la librería socket.io (tanto del lado del cliente, como del servidor).

Como ya se menciona en el apartado anterior [Estructura de ficheros](#_Estructura_de_ficheros), el fichero *www*, contiene la gestión de los sockets y más código referente a otras secciones (todo del lado del servidor).

Lo primero a realizar es importar las dependencias para poder usar los objetos deseados (ver figura x).



/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Code for Socket.io handling

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var os = require('os');

var nodeStatic = require('node-static');

var socketIO = require('socket.io');

var fileServer = new(nodeStatic.Server)();

var io = socketIO.listen(server);

var users = [];

console.log('Server listening on port ' + port);

Figura 12 – Fichero bin/www

1. La librería ***os*** es requerida para obtener la dirección IPv4 para mandar un mensaje al socket cliente en caso de que pregunte por la dirección IP actual (ver figura x).



/\*\*

\* On ippaddr

\*/

socket.on('ipaddr', function() {

var ifaces = os.networkInterfaces();

for (var dev in ifaces) {

ifaces[dev].forEach(function(details) {

if (details.family === 'IPv4' && details.address !== '127.0.0.1') {

socket.emit('ipaddr', details.address);

}

});

}

});

});

Figura 13 - Fichero bin/www

1. El módulo de ***node*** ***static*** permite crear una ruta estática virtual existente o no en el sistema que facilita el acceso a los ficheros a la hora de servirlos.
2. Por último, el módulo que contiene la librería de web sockets ***socket.io***. Gracias a él, se podrá establecer la comunicación entre los clientes y el servidor. Crea una ruta propia que es de donde se sirve el cliente a la hora de solicitar la API desde el navegador.

Esta librería basa su funcionalidad en **listeners**. Se pone a la escucha una instancia de ***http.Server*** que es empleada tanto en el lado del servidor como en el lado del cliente y en función del evento que se emita en un lado, en el lado contrario se realizará una función u otra.

Una vez dicho esto. ¿Cómo gestiona la aplicación la comunicación del chat con este mecanismo? De la siguiente manera:

Al inicializarse el componente chat, se ejecuta la función ***initClient***, con sus respectivas **funciónes *callback*** pasadas por parámetro para poder habilitar o inhabilitar las pantallas de llamada.

Lo primero que hace el usuario al iniciarse la aplicación es conectarse al servidor mediante el método connect del objeto de tipo http.Server “**io**” de la librería socket, la cual se obtiene del archivo ***socket.io.js*** (importado en el fichero ***views/index.hbs***).



<script src="/socket.io/socket.io.js"></script>

Figura 14 - Fichero views/index.hbs

Además, se obtiene el nombre de usuario (una vez el usuario ha iniciado sesión se registra en el local storage del navegador) para la posterior presentación al servidor.



var socket;

var username;

var calleeUsername;

function initClient(newToggleCallScreenCallback,

newToggleReceivedCallScreenCallback) {

//Initializing variables

socket = io.connect();

username = localStorage.getItem('username');

Figura 15 - Fichero public/main.js

Una vez se ha obtenido el nombre de usuario del local storage, mediante un mensaje “**presentation**” enviamos el nombre de usuario al servidor.



//Telling the server who the user is

socket.emit('presentation', username);

Figura 16 - Fichero public/main.js

Este hará que se ejecute la función que se ve a continuación en la figura X, gracias al listener asociado a ese evento.



/\*\*

\* On user presentation.

\* When user connects to the app, he presents itself to

\* get stored in the current online user's list of the server.

\*/

socket.on('presentation', function(username) {

var user = { username: username, socket: socket };

console.log('User ' + username + ' connected.');

users.push(user);

});

Figura 17 - Fichero bin/www

Esta función crea un nuevo objeto usuario que tendrá el nombre del mismo y el objeto de tipo socket que utilizará para ponerse en contacto con él.

**Nota**: Tanto el nombre de usuario, como la id que contiene el objeto socket son únicos en la aplicación.

Cuando un usuario llama a otro que está presente en su lista de contactos, la función ***call*** es ejecutada gracias al evento que se dispara debido al listener establecido en la plantilla HTML del componente contacts-list (en el apartado [Interactividad con Angular 4](#_Interactividad_con_Angular) se explican los componentes y cada una de sus partes en detalle).



/\*\*

\* Función que establece el valor de iniciador de llamada a verdadero,

\* establece el valor de la variable asociada al nombre del usuario

\* al que se está llamando con el valor recibido por parámetro y

\* solicita los permisos de obtención del flujo de datos de la

\* cámara y del micrófono.

\* @param {\*} newCalleeUsername

\*/

function call(newCalleeUsername) {

isInitiator = true;

calleeUsername = newCalleeUsername;

getUserMedia(constraints, handleUserMedia, handleUserMediaError);

}

Figura 18 - Fichero public/js/libs/main.js

Esta función lo que hace en líneas generales es solicitar los datos del usuario y establecer varios datos de manera que el usuario que inicia la llamada es el mismo que la ha ejecutado.

Una vez se han solicitado los permisos mediante la función ***getuserMedia***, si el usuario acepta se ejecutará la función callback ***handleUserMedia***, de lo contrario, si cancela o carece de dispositivos media, se ejecutará la función callback ***handleUserMediaError***.



function handleUserMedia(stream) {

console.log('Adding local stream.');

try {

localVideo.srcObject = stream;

} catch (error) {

localVideo.src = URL.createObjectURL(stream);

}

localStream = stream;

if (isInitiator) {

//Once we've got the user media, we can call the other user.

socket.emit('call', calleeUsername, username);

} else {

socket.emit('pickup', room, callerId);

}

}

Figura 19 - Fichero public/js/libs/main.js

Esta función obtiene el ***stream*** o flujo de datos de la videocámara y del micrófono y en caso de haber sido el que inició la llamada emite un mensaje socket “***call***” mediante el cual indica que va a llamar al usuario pasado por parámetro y su propio nombre (para evitar buscar por id en el servidor).

De lo contrario, si no es el que llamó, emitirá un mensaje “***pickup***” al servidor. Esto se debe a que esta función se ejecuta cada vez que se solicita el flujo de datos de la cámara, por lo que será llamada cada vez que un usuario llame a alguien o coja una llamada.

En el servidor la siguiente función callback hará el trabajo de llamar al usuario en cuestión.

Para ello primero se realiza la búsqueda del usuario a través de la función ***findUserByName***, que la gestiona mediante la programación funcional (ver apartado [Programación funcional](#_Programación_funcional)). Si el usuario no se encuentra online, se enviará un mensaje “***userdisconnected***” al usuario que llamó, para indicarle que el usuario al que está llamando está desconectado (esto provocaría el final de la ejecución de la función).



socket.on('call', function (calledName, username) {

//Buscamos en la lista de usuarios el primer usuario con ese nombre (deben ser únicos)

var user = findUserByName(calledName),

room;

if (!user) {

socket.emit('userdisconnected', calledName, 'The user you\'re trying to call is disconnected');

return;

}

Figura 20 - Fichero bin/www

Si por lo contrario el usuario está en línea, se introduce al que llamó en la habitación mediante la cual se va a establecer una especie de comunicación local.



//Establecemos el string de la habitación concatenando la id del socket called con la del caller

room = user.socket.id + "" + socket.id;

console.log('User with ID:' + socket.id + ' is calling ' + calledName + '(' + user.socket.id + ')');

//Introducimos al usuario en la habitación creada para el chat entre los usuarios.

socket.join(room);

Figura 21 - Fichero bin/www

Tras unirse se informa mediante el mensaje “***calling***” al usuario llamador que se está llamando al usuario llamado, mientras que a este último se le informa de que está siendo llamado mediante el mensaje “***called***” de manera que reciba el nombre de la habitación (que está formado por la concatenación de las ids del llamador y el llamado).



//Notificamos al usuario que ha llamado de que se está llamando al usuario

socket.emit('calling' , room, socket.id);

//Notificamos al otro usuario de que se le está llamando

user.socket.emit('called' , room, socket.id, username);

});

Figura 22 - Fichero bin/www

El usuario llamado recibe el mensaje y establece una serie de parámetros como pueden ser la variable booleana que indica si es el usuario que **inició la llamada**, la **habitación** en la que se encuentra el usuario llamante y la **id** de este.

Finalmente se ejecuta la función callback que hace posible la visualización de la pantalla que notifica al usuario que se le está llamando y que le ofrece la posibilidad de coger o rechazar la llamada.



//Un usuario está llamando a este cliente, recibiendo por parámetro el string de la habitación

socket.on('called', function(serverRoom, serverCallerId, callerUsername) {

isInitiator = false;

room = serverRoom;

callerId = serverCallerId;

//Executing callback function from chat component.

onToggleReceivedCallScreenCallback(callerUsername);

});

Figura 23 - Fichero public/js/libs/main.js

Tras presentarle la pantalla de llamada entrante al usuario, pueden ocurrir tres cosas: el usuario que llamó cancela la llamada, el llamado descuelga o la rechaza.

Si el usuario llamante se arrepiente y cancela la llamada, se envía un mensaje “***cancelcall***” al servidor junto con el nombre del usuario al que se llamó.

/\*\*

\* Función que envía un mensaje al servidor de cancelación de llamada

\*/

function cancelCall() {

socket.emit('cancelcall', calleeUsername);

}

Figura 24 - Fichero public/js/libs/main.js

Una vez el servidor recibe el mensaje, si encuentra al usuario en cuestión, le notifica de que la llamada se ha cancelado.



/\*\*

\* When the user stop calling someone

\*/

socket.on('cancelcall', function(calleeUsername) {

var user = findUserByName(calleeUsername);

//If user is disconnected, nothing to notify

if (!user) {

return;

}

user.socket.emit('canceledcall');

});

Figura 25 - Fichero bin/www

Cuando el usuario recibe el mensaje se inhabilita la pantalla de llamada entrante.

//Una vez el otro usuario ha rechazado la llamada

socket.on('canceledcall', function(room) {

onToggleReceivedCallScreenCallback();

});

Figura 26 - Fichero public/js/libs/main.js

Si rechaza la llamada, la función ***rejectCall*** es ejecutada.

/\*\*

\* Función que envía un mensaje al servidor de rechazo de llamada

\*/

function rejectCall() {

socket.emit('rejectcall', callerId);

}

Figura 27 - Fichero public/js/libs/main.js

Mediante la cual se envía un mensaje “***rejectcall***” al servidor, que buscará al usuario y en caso de encontrarse en línea le notificará a través de un mensaje “***rejectedcall***”.



/\*\*

\* When the user reject when someone is calling

\*/

socket.on('rejectcall', function(callerId) {

var user = findUserById(callerId);

//If user is disconnected, nothing to notify

if (!user) {

return;

}

user.socket.emit('rejectedcall');

});

Figura 28 - Fichero bin/www

Cuando el usuario llamante recibe el mensaje de vuelta de que la llamada ha sido rechazada, se inhabilita la pantalla de llamada en curso (la cual se explica más en detalle junto con todo lo relacionado de ***Angular*** en la sección [Interactividad con Angular 4](#_Interactividad_con_Angular)).



//Una vez el otro usuario ha rechazado la llamada

socket.on('rejectedcall', function(room) {

onToggleCallScreenCallback();

});

Figura 29 - Fichero public/js/libs/main.js

Por otro lado, en el caso de que coja la llamada se le pedirán los datos de la cámara y del micrófono.



//Una vez nos llaman podemos pulsar el botón de coger la llamada

function pickUp() {

getUserMedia(constraints, handleUserMedia, handleUserMediaError);

}

Figura 30 - Fichero public/js/libs/main.js

De manera que, como ya se ha mostrado anteriormente se realiza mediante la función ***getUserMedia***, que recibe la función callback que se ejecuta en caso de éxito ***handleUserMedia*** y la que se encarga de gestionar el error o cancelación ***handleUserMediaError***.

Se hizo mención de que en función de si inició la llamada o no haría una cosa u otra, lo cual se decide en el caso del usuario llamado en el listener “***called***” del socket visto anteriormente, donde la variable ***isInitiator*** se establece como falsa.

Esto determina que para este usuario se emitirá al servidor un mensaje “***pickup***” en la función que se ejecuta en caso de éxito.



if (isInitiator) {

//Once we've got the user media, we can call the other user.

socket.emit('call', calleeUsername, username);

} else {

socket.emit('pickup', room, callerId);

}

Figura 31 – Fragmento de función *handleUserMedia* del fichero public/js/libs/main.js

El servidor recibe el mensaje y busca un usuario por la id recibida en el mismo. En caso de no encontrarse conectado se emitirá un mensaje “***userdisconnected***" al usuario que llamado.

De lo contrario se le mete en la habitación junto con el usuario llamante y envía un mensaje “***ready***” a la habitación que recibirán todos los participantes de esta.



/\*\*

\* Una vez el usuario ha sido llamado y lo ha cogido se une a la habitación en cuestión

\*/

socket.on('pickup', function (room, callerId) {

var user = findUserById(callerId);

//If user is disconnected, nothing to pick up

if (!user) {

socket.emit('userdisconnected', callerId, 'The user you\'re trying to call is disconnected');

return;

}

//Joining the room where the calling user is

socket.join(room);

//Notifying all participants in the room that the communication is ready to be established

io.sockets.in(room).emit('ready');

});

Figura 32 - Fichero bin/www

Cuando el servidor les indica a los usuarios que la comunicación está lista para realizarse mediante el mensaje mencionado, se establece el valor de la variable booleana ***isChannelReady*** a verdadero y se ejecuta la función ***maybeStart***, que finalmente establece la comunicación mediante objetos de tipo PeerConnection (ver sección [Comunicación WebRTC](#_Comunicación_WebRTC)).



// Cuando el servidor es notificado de que el usuario llamado ha cogido la llamada

socket.on('ready', function(room) {

// Establecemos la variable isChannelReady como verdadera porque ya está listo para la comunicación

isChannelReady = true;

maybeStart();

});

Figura 33 - Fichero public/js/libs/main.js

## Arquitectura Angular 4

Angular es un framework que permite a los desarrolladores construir aplicaciones sostenibles y escalables debido a que asegura el uso de algunas de las mejores prácticas que han de realizarse para este cometido.

El elemento básico de una aplicación Angular es el NgModule, que proporciona el contexto de compilación a los componentes, es decir, hacen presentes las dependencias y librerías necesarias para que estén listas para ser empleadas por la aplicación y puedan ser accedidas desde cualquier parte del software.

¿Que a cuento de qué viene todo esto? La explicación es que dicho elemento básico es lo que dota a los componentes de la aplicación de la funcionalidad necesaria para su correcto funcionamiento.

Los componentes son elementos visuales de la vista que se componen generalmente de tres partes:

* La plantilla, basada en HTML, que establece los elementos de la vista. Está dotada de una serie de directivas basadas en TypeScript, que permiten renderizar la vista de una manera más óptima o en base a los datos de la parte del “controlador” de la vista haciendo uso de las directivas. Las directivas son fragmentos de código incrustados en la vista que permiten añadir cierta lógica al documento.
* Los estilos, basada en CSS aplica los estilos a la plantilla o vista.
* El controlador, no es más que el archivo TypeScript que pone en conjunto todas las partes y añade cierta funcionalidad. Sería como el JavaScript de las páginas convencionales, pero separado de una manera que permite la mejor estructuración de cada uno de los elementos de la página.

Este es el cometido de Angular, aislar cada bloque de la página semánticamente y a su vez separar cada aspecto de este (vista, estilos y funcionalidad como puede ser reaccionar a eventos o funcionalidad específica de la vista).

Cuando se trata de una funcionalidad más compleja y que tiene un uso más allá de la parte estética, se hace uso de los servicios, los cuales comienzan a estar disponibles para los componentes desde el momento en el que se inyectan en ellos. Por ejemplo, para realizar llamadas asíncronas al servidor, gestión de cookies o manejo del almacenamiento local del navegador.

A continuación, en la figura 34 se puede ver la representación gráfica de la arquitectura de Angular 4.



Figura . Fuente: <https://angular.io/guide/architecture>

# Resultados y discusión

Este apartado contrasta la planificación del proyecto (la cual figura en la sección [Planificación del proyecto](#_Planificación_del_proyecto)) con la realidad y las causas de cada una de las diferencias.

En primer lugar, la intención del proyecto era abarcar todas las tecnologías posibles, de manera que este proyecto sirviera como aprendizaje a la hora de combinarlas y de aprender cada una de ellas en profundidad. Pero, por otro lado, también ofrecer un prototipo de calidad. Por lo que fue necesario buscar el equilibrio entre ambos extremos para comenzar el desarrollo cuanto antes.

La propuesta inicial fue el desarrollo de una aplicación web que permitiera la videoconferencia. Por ambición, se quisieron abarcar varias tecnologías más, de entre las cuales se descartó ***IONIC***.

Esto es debido a que, como ya se ha mencionado en el párrafo anterior, la otra parte de la balanza era ofrecer un prototipo de calidad, por lo que la decisión final fue añadir funcionalidad de gestión de usuarios, a cambio de la implementación de esta tecnología. En cierto modo, las páginas web ya son aplicaciones accesibles desde los navegadores de los dispositivos móviles, por lo que no era una pérdida de conocimiento excesiva.

A cambio, se pudo profundizar más en el uso de la librería Mongoose, para la gestión de la autenticación y la lista de contactos por cada uno de los usuarios.

Otro de los motivos fue, que durante el curso ya se ha hecho uso de la herramienta ***Apache Cordova***, por lo que en cierto modo ese ámbito de conversión de aplicación web a aplicación nativa para iOS y Android ya estaba cubierto.

Al fin y al cabo, por falta de tiempo, se ha dejado de hacer uso de la tecnología anteriormente mencionada, además del nulo desarrollo de la funcionalidad relacionada con el envío de mensajes de texto instantáneos. Esto cambiará en un futuro no muy lejano, como se comenta en la siguiente sección [Conclusiones](#_Conclusiones).

# Conclusiones

La labor principal de este proyecto sin duda, ha sido la **investigación**. Más de la mitad del tiempo dedicado a este proyecto ha sido invertida en la búsqueda por foros conocidos y no tan conocidos, páginas web de tutoriales y de algún particular que aportaba su granito de arena. He visitado cantidades ingentes de páginas web y fuentes de información como libros para contrastar información, soluciones y maneras de pensar, de las cuales, por no mostrar el historial completo de mi navegador, se presentan las más relevantes para el proyecto en el apartado [Bibliografía y referencias](#_Bibliografía_y_referencias).

Con WebRTC he sido capaz de, siguiendo unos breves tutoriales, construir un sencillo chat de vídeo. Que se complicó a la hora de implementarlo en la aplicación, puesto que está basada en Angular 4 y Node JS. Entonces, con un poco de empeño y de razonamiento conseguí pasar ese primer gran bache.

Una vez ya tenía una aplicación como tal, tuve que dar el siguiente paso: añadir una lista de contactos. Esto se me complicó más de lo que pensaba, puesto que el código ejecutable estaba en un componente Angular, y la lista estaba anidada en el mismo, por lo que no era capaz de notificar al componente padre de cada uno de los eventos. Pero otra vez, fui persistente y conseguí hacerlo.

Según pasaban las semanas, el proyecto iba avanzando cada vez más rápido. En una reunión se llegó a la conclusión de que, al ser una presentación en un servidor local, no se podría hacer uso de un dominio fijo. Por lo que se decidió habilitar un botón para establecerlo de forma manual en caso de que no se pudiera acceder a la dirección por defecto. Por lo que, tras investigar un poco, conseguí dar con la mejor solución: crear un servicio Angular que comunicase todos los componentes deseados de la aplicación, de manera que tuvieran datos en común, y fueran notificados si se ejercían cambios sobre ellos.

Sin duda, lo más satisfactorio es haber recopilado varias tecnologías y haber conseguido una combinación inusual.

El mensaje que más ha estado presente en mi cabeza al desarrollar este proyecto ha sido el siguiente:

“*La cuestión no es conseguir que todo funcione, eso no es difícil. La complejidad de todo esto es ser capaz de realizar lo propuesto, de la mejor manera posible. Y eso requiere ser metódico y paciente para aplicar las mejores prácticas y ser profesional en lo que se hace.”*

*Mario García Ramos*

Por último, cabe destacar la dirección de este proyecto, la cual apunta a una aplicación que permita la comunicación entre dos o más usuarios por vídeo, audio o mensajes de texto instantáneos. Con funcionalidades nuevas como personalización de perfil con foto, autenticación mediante teléfono móvil y SMS o correo electrónico…

Y lo que más me entusiasma: la adición de realidad aumentada en la videoconferencia.

# Bibliografía y referencias

En este apartado se hace referencia a las fuentes de información que han aportado conocimiento durante la realización del proyecto.

|  |  |
| --- | --- |
| **Referencias genéricas** | |
| **Nombre** | **Enlace** |
| Angular | <https://angular.io/> |
| Node JS | <https://nodejs.org/es/> |
| Express JS | <http://expressjs.com/es/> |
| Ionic Framework | <https://ionicframework.com/> |
| WebRTC | <https://webrtc.org/> |
| MongoDB | https://www.[mongodb](https://www.mongodb.com/).com/ |
| Mongoose | <http://mongoosejs.com/> |
| Mongoose unique | <https://www.npmjs.com/package/mongoose-unique-validator> |
| BCrypt JS | <https://www.npmjs.com/package/bcryptjs> |
| JWT.io | <https://jwt.io/> |
| Fontawesome | <https://fontawesome.com/> |

Tabla

|  |  |
| --- | --- |
| **Referencias WebRTC** | |
| MediaStream Deprecations | [https://developers.google.com/web/updates/2015/07/mediastream-deprecations?hl=en#stop-ended-and-active](https://developers.google.com/web/updates/2015/07/mediastream-deprecations?hl=en%23stop-ended-and-active) |
| How to write video-conferencing application using WebRTC | <https://www.webrtc-experiment.com/docs/how-to-WebRTC-video-conferencing.html> |
| Getting Started with WebRTC | <https://www.html5rocks.com/en/tutorials/webrtc/basics/> |
| WebRTC in the real world: STUN, TURN and signaling | <https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/> |
| Building a WebRTC Video and Voice Chat Application | <https://www.pubnub.com/blog/2014-10-21-building-a-webrtc-video-and-voice-chat-application/> |
| Building a Video/Audio Chat Web App with WebRTC | <https://web-engineering.info/node/57> |
| WebRTC Video Chat in 20 Lines of JavaScript | <http://www.instructables.com/id/WebRTC-Video-Chat-in-20-Lines-of-JavaScript/> |
| Codelab WebRTC By Google | <http://io2014codelabs.appspot.com/static/codelabs/webrtc-file-sharing/> |
| PeerConnection: make a call in local network | <https://stackoverflow.com/questions/23498398/peerconnection-make-a-call-in-local-network> |
| Basics of WebRTC Peer Connections | <https://medium.com/@kanecohen/basics-of-webrtc-peer-connections-c1ed743de2f6> |
| Stop/Close webcam which is opened by navigator.getUserMedia | <https://stackoverflow.com/questions/11642926/stop-close-webcam-which-is-opened-by-navigator-getusermedia?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa> |
| Understanding Socket.IO | <https://nodesource.com/blog/understanding-socketio/> |

Tabla

|  |  |
| --- | --- |
| **Referencias Angular** | |
| Tutorial oficial Angular | <https://angular.io/tutorial/toh-pt4> |
| Angular 2 and Socket.io Chat App | <https://medium.com/@REPTILEHAUS/angular-2-and-socket-io-chat-app-f56afb9ceeb2> |
| How to use JavaScript libraries in Angular 2+ apps | <https://hackernoon.com/how-to-use-javascript-libraries-in-angular-2-apps-ff274ba601af> |
| Sharing Data Between Angular Components | <https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/> |
| How to bundle an Angular app for production | <https://stackoverflow.com/questions/37631098/how-to-bundle-an-angular-app-for-production?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa> |

Tabla

# Anexos/ Otros

En esta sección se muestran fragmentos y referencias de código de la aplicación.

## Programación funcional

En este apartado se muestran fragmentos de código en el que se han empleado los conocimientos de programación funcional.

En el caso de la figura X, se trata de funciones encargadas de la gestión de usuarios online.



function findUserById(id) {

return users.filter(user => user.socket.id === id)[0];

}

function findUserByName(username) {

return users.filter(user => user.username === username)[0];

}

function deleteUserById(id) {

return users.filter(user => user.socket.id !== id);

}

Figura . Fichero: bin/www

En cambio, en el lado del cliente, contemplamos la llamada síncrona a la ruta que se encarga de ello. En el apartado siguiente, [Gestión de base de datos con Mongoose](#_Gestión_de_base) se muestra el código que tiene detrás.



return this.http.post(this.URL +'/user/search', body, {headers: headers})

.map((response: Response) => response.json()).

catch((error: Response) => Observable.throw(error.json()));

Figura . Fichero: assets/app/auth/auth.service.ts

## Gestión de base de datos con Mongoose

En este apartado se muestran ejemplos del acceso a la base de datos, como pueden ser la búsqueda de usuarios cuando un usuario está en busca de contactos nuevos y decide emplea el componente de contactos para ello.

En este caso, en la figura X, se presenta un fragmento del código de manejo de rutas donde se gestiona la búsqueda de un usuario cuyo campo ***username*** contiene el recibido en el cuerpo de la solicitud (cuarta línea de código).



router.post('/search', function (req, res, next) {

//Looking for users whose username attribute contains the string received as "username"

var query = User.find({username: { $regex: '.\*' + req.body.username + '.\*' } })

.select('username firstName secondName');

query.exec(function (err, users) {

//If an error occurs (mostly validation) it will return a 500 error.

if (err) {

res.status(500).json({

title: 'An error ocurred',

error: err

});

}

//If no user is found it will return a 500 error "Wrong credentials".

if (!users) {

return res.status(401).json({

title: 'User not found',

error: {message: 'User not found'}

});

}

res.status(200).json({

message: 'User found',

users: users

});

});

});

Figura . Fichero: /routes/ contact.js



router.post('/', function (req, res, next) {

var user = new User ({

email: req.body.email,

password: bcrypt.hashSync(req.body.password, 10),

username: req.body.username,

firstName: req.body.firstName,

secondName: req.body.secondName

});

user.save(function (err, result) {

//If an error occurs (mostly validation) it will return a 500 error.

if (err) {

res.status(500).json({

title: 'An error ocurred',

error: err

});

}

//If no errors, it will return a json message "User created".

var token = jwt.sign({user: user}, 'secret', {expiresIn: 86400});//24hr

res.status(200).json({

message: 'Succesfully logged in',

token: token,

userId: user.\_id,

username: result.username

});

});

});

Figura . Fichero: /routes/user.js

## Autenticación mediante tokens con JWT.io

Como se muestra en la figura 36 del apartado anterior, al darse de alta un usuario, se crea un token para mantener la sesión del usuario. Este fenómeno se produce también cuando el usuario inicia sesión.



var token = jwt.sign({user: user}, 'secret', {expiresIn: 86400});//24hr

res.status(200).json({

message: 'Succesfully logged in',

token: token,

userId: user.\_id,

username: result.username

});

Figura . Fichero: /routes/user.js

## Manejo de seguridad con BCrypt JS

En el apartado [Gestión de base de datos con Mongoose](#_Gestión_de_base), más concretamente, en la figura X, se ha visto cómo se almacena el usuario. Si desviamos la atención a un fragmento concreto, podremos ver el lugar donde la contraseña es encriptada gracias al método ***hashSync*** que nos proporciona la librería ***BCrypt JS***:



var user = new User ({

email: req.body.email,

password: bcrypt.hashSync(req.body.password, 10),

username: req.body.username,

firstName: req.body.firstName,

secondName: req.body.secondName

});

Figura 37. Fichero: /routes/user.js

## Resultado estético

## Especificación del plan de pruebas

En esta actividad se desea realizar un análisis de las pruebas a las que se va a someter el software una vez esté finalizado el periodo de desarrollo de este. Puesto a que son varias las partes en las que se divide el proyecto

|  |  |  |
| --- | --- | --- |
| **Id** | **Descripción** | **Resultado obtenido** |
| **1** | Un usuario está llamando a otro. |  |
| **1.1** | Se muestra la pantalla de llamada con sus correspondientes botones |  |
| **1.2** | Se solicitan los permisos para la obtención de datos a través del micrófono y la videocámara. |  |
| **1.3** | Si el usuario acepta los permisos se procede a mostrar por pantalla el vídeo capturado en directo. |  |
| **1.4** | Una vez obtenidos los datos de vídeo y audio y mostrados por pantalla, se procede a llamar al usuario en cuestión. |  |
| **2 - A** | El usuario a llamar recibe la llamada. |  |
| **2.1** | Al descolgar se notifica sobre ello al otro usuario que llamó, después de haber aceptado los permisos para la obtención de la cámara. |  |
| **2.1.1** | El usuario que llamó recibe el mensaje y está listo para la videoconferencia. |  |
| **2.2** | Al descolgar, se solicitan los permisos de la cámara y micrófono |  |
| **2.3** | Al aceptar los permisos habiendo recibido una llamada se establece el vídeo y el audio locales correctamente. |  |
| **2.4** | Una vez se han aceptado los permisos y tenemos video y audio locales funcionando, se envía un mensaje al servidor y este responde a los dos usuarios con un estado de “listo”. |  |
| **2.5** | Una vez el canal de comunicación socket está listo se procede a crear la conexión p2p (peer to peer) en la que se incluye el flujo de datos de la cámara y micrófono y el usuario que llamó crea una oferta. |  |
| **2.6** | Una vez se crea la oferta, el servidor recibe el mensaje con los datos para reenviársela al otro usuario. |  |
| **2.7** | El otro usuario recibe la oferta con los datos deseados y crea una respuesta correctamente. |  |
| **2.8** | Una vez se crea la respuesta, el servidor recibe el mensaje con los datos para reenviársela al otro usuario. |  |
| **2.8** | Una vez establecida la comunicación peer to peer, los usuarios ven su propio video y ven y escuchan el vídeo del otro usuario. |  |
| **2 - B** | El usuario está desconectado. Se muestra un mensaje al que llamó y se para la llamada. |  |
| **3** | Habiendo una llamada en curso el usuario cuelga y se cierra el flujo de datos relativo a la cámara y el micrófono. |  |
| **3.1** | Habiendo una llamada en curso el usuario cuelga desaparece la pantalla de llamada. |  |
| **4** | Habiendo una llamada en curso un usuario cuelga y para el otro participante de la llamada desaparece la pantalla de llamada. |  |
| **4.1** | Habiendo una llamada en curso un usuario cuelga y el flujo de datos relativo a la cámara y el micrófono del otro participante finaliza. |  |
| **5** | En una llamada en curso se presiona el botón de silenciar micrófono y el flujo de datos se corta. El otro participante deja de escucharle. |  |
| **6** | En una llamada en curso se presiona el botón de pausar vídeo y el flujo de datos se corta. El otro participante deja de verle. |  |
| **7** | Tras realizar una llamada el usuario vuelve a llamar al mismo usuario y no se comparte el flujo de datos hasta que se acepta la llamada. |  |
| **8** | Si un usuario llama a otro y este rechaza la llamada, la pantalla de llamada desaparece |  |
| **8.1** | Si un usuario es llamado por otro y rechaza la llamada, la pantalla de llamada recibida desaparece. |  |
| **9** | Si un usuario es llamado por otro y el que llamaba cancela la llamada, la pantalla de llamada recibida desaparece. |  |
| **9.1** | Si un usuario llama a otro y el que llamaba cancela la llamada, la pantalla de llamada desaparece. |  |
| **10** | Cuando se muestra la pantalla de llamada al llamar a otro usuario, no se muestra el vídeo de la llamada anterior. |  |
| **11** | Cuando el otro participante de la llamada cierre la aplicación, la llamada finalizará |  |
| **12** | Cuando se coja una llamada desaparece el panel de llamada recibida y aparece el de llamada en curso. |  |
| **13** | Al haber eliminado el mensaje por habitación, se establece correctamente la llamada mediante la comunicación por el método emit. (se comunican mediante el sendmessage y a parte mediante el on call emit etc del socket) |  |
| **14** | Al haber silenciado el micrófono en la llamada anterior no se mantiene en la siguiente. |  |
| **15** | Al haber pausado la cámara en la llamada anterior no se mantiene en la siguiente. |  |
| **16** | En una llamada en curso, cuelga el usuario que llamó y la llamada finaliza para ambos. |  |
| **17** | En una llamada en curso, cuelga el usuario que ha sido llamado y la llamada finaliza para ambos. |  |

# TODOs:

## En Contacts Service añadir método get contacts para obtener el array, distinto del getcontacts de acceso a la bd

## Quitar todas las listas de usuarios y métodos que no tengan que ver de subcomponentes de contactscomponent

## Lo mismo con los atributos y métodos de auth

## En Eliminar de contactos eliminar de la BD también

## Actualizar diagramas con cambios

## Pasar lo del localstorage de signincomponent al authservice

## Guardar módulos y app en sí para producción y documentarlo