SDI – Sistemas Distribuidos e Internet

ENUNCIADO PRÁCTICA 2 – NODE.Js & Servicios Web

INFORME

Grupo 604-607

Nombre:	Thalía
Apellidos:	Cuetos Fernández
Email:	UO264545@uniovi.es
Cód. ID GIT	1920-604
Participación	50%
Nombre:	Sonia
Apellidos:	García Lavandera
Email:	UO263536@uniovi.es
Cód. ID GIT	1920-607
Participación	50%

Índice

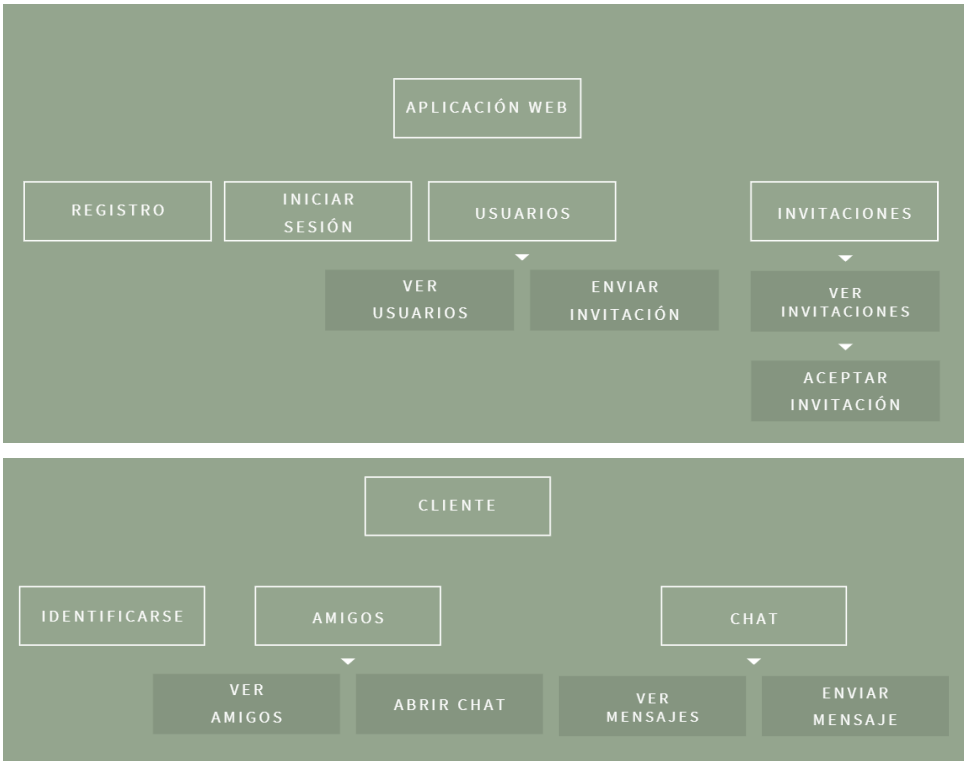
Introducción

Nuestra aplicación consta únicamente de usuarios estándar al contrario que lo que se había en el proyecto anterior que se disponía de un usuario administrador. A parte de eso, los usuarios que son ajenos a la app sólo podrán registrarse para poder usarla. Una vez inicias sesión tienes varias opciones puedes listar a todos los usuarios de la aplicación, listar a tus amigos, enviar invitaciones de amistad a otros usuarios, aceptar/rechazar las invitaciones recibidas que podrás listar... También se da la opción de poder buscar entre los usuarios de la aplicación mediante un cuadro de búsqueda y tanto los amigos, usuarios como invitaciones están paginados para acceder de forma más fácil.

A continuación, pasaremos a detallar cada uno de los casos de implementación para que queden claro la forma en la que enfocamos su desarrollo.

Nota: realizamos tanto los requisitos obligatorios como los opcionales, así como sus pruebas.

Mapa de navegación



Aspectos técnicos y de diseño relevantes

Caso de uso W1 Perfil público: registrarse como usuario

A la hora de registrar un usuario en la aplicación se le piden una serie de datos entre los que se encuentran el nombre, los apellidos, email y la contraseña que debe repetirse para confirmarse. Aquí como tuvimos que hacer varias comprobaciones de los datos que el usuario introduce, para hacer más legible el código, decidimos apoyarnos en dos funciones creadas por nosotras.

La primera función se llama **validarDatosRegistroUsuario** y recibe en un objeto usuario los datos que nos llegan del formulario. Comprobamos que esos datos no vengan vacíos y en el caso de la contraseña que coincida con la confirmación de esta. También, a la hora de comprobar si el email con el que se quiere registrar el usuario en la aplicación existe, creamos una nueva función llamada **comprobarEmailRepetido**

al que se le pasa como criterio el email del usuario y se busca en la base de datos dicho email usando el método obtenerUsuarios del módulo gestorBD. Ambas funciones se llaman en el método post /usuario y en caso de que se produzca algún error se informará al usuario en la vista mediante el envío de mensajes con res.redirect().

Caso de uso W2 Perfil público: iniciar sesión

Para iniciar sesión en la web es necesario aportar un email y contraseña válidos. Al formulario se accede a través de /identificarse y consta de dos campos: uno para el email y otro para la contraseña.

Para comprobar que el usuario existe y la contraseña es correcta, se busca en la base de datos un usuario con los datos proporcionados y, si no se encuentra ninguno, el inicio de sesión es incorrecto y se muestra un mensaje de error. En caso de que sea correcto se guarda el email del usuario en *session* y se le redirige a la lista de usuarios.

Caso de uso W3 Usuario registrado: Fin de sesión

El fin de sesión consiste únicamente en establecer el usuario como null en session y redirigirlo a la vista de identificarse. Esta opción de menú solo es accesible para usuarios en sesión, por lo que es necesario pasar en cada vista el atributo usuario y comprobar si éste es null o tiene un valor.

Caso de uso W4 Usuario registrado: listar todos los usuarios de la aplicación

En base de datos los usuarios están guardados en una colección en la cual cada elemento de esa colección es un objeto usuario por decirlo así que contiene los datos que se pidieron en el registro. De esta forma mediante el método **obtenerUsuarios** del módulo gestorDB sacamos los usuarios de la aplicación. Además, contamos con otro método que empleamos en este caso que nos permite sacar los usuarios paginados para mostrarlos en la vista.

Cuando mandamos los amigos a la vista también aprovechamos para obtener los amigos del usuario en sesión y así no mostrar el enlace de "Añadir amigo".

En el enunciado de la práctica se pide que controlemos que se muestre una opción de menú que sea visible únicamente para los usuarios en sesión, por lo que a la vez que mandamos a la vista de listar usuarios la lista de usuarios y lo necesario para la paginación, también enviamos el email del usuario en sesión que nos permite comprobar si estamos en sesión y así mostrar o no esta opción de menú desde la vista.

Para poder mantener la búsqueda en la paginación, que es uno de los problemas que se comentó en clase, mandamos a la vista de listar usuarios la búsqueda también. De esta forma se trata ahí junto con la paginación para que se mantengan ambas.

Caso de uso W5 Usuario registrado: buscar entre todos los usuarios de la aplicación

Cuando devolvemos a la vista los usuarios de la aplicación se hace una pequeña comprobación del contenido del campo de búsqueda para saber si está vacío o no. En el caso de que esté vacío se devolverán todos los usuarios eso si paginados como se pide. En caso contrario de que no esté vacío se coge como criterio que el texto introducido en el campo de búsqueda esté contenido en el nombre, apellidos o email mediante una consulta de mongoDB con el operador **$or**.

Aquí tuvimos que pasar a la vista además de la lista con los usuarios, la búsqueda también para que al hacer la paginación no se pierda la búsqueda. Además, también mandamos el email para mostrar algunas opciones del menú de navegación para saber si hay usuario en sesión.

Caso de uso W6 Usuario registrado: enviar una invitación de amistad a un usuario

En la lista de usuarios de la aplicación aparece una opción al lado de cada uno que permite enviarle una invitación de amistad*. Este proceso tarda un poco ya que hace varias comprobaciones que requieren el acceso a base de datos para asegurar que:

- El usuario en sesión no se invite a sí mismo (no requiere acceso a base de datos ya que almacenamos el email del usuario en sesión).

- El usuario en sesión y al que se le invita no sean amigos ya. Se obtiene el usuario en sesión a partir del email y se busca el email de la persona invitada en el array de amigos.

- No existe una invitación del usuario a la otra persona o viceversa. Se realiza una consulta mediante el operador $or y si no devuelve nada es correcto

```
// Comprobar que no se ha enviado ya la petición o que no se ha recibido una petición de la otra persona
gestorBD.obtenerInvitaciones( criterio: {
    $or : [
        {"email_emisor" : req.session.usuario, "email_receptor" : usuarios[0].email },
        {"email_receptor" : req.session.usuario, "email_emisor" : usuarios[0].email }
    ]}, funcionCallback: function(result) {
    if(result != null && result.length > 0){
        app.get("logger").info("Ya existe una invitación de amistad");
        res.redirect("/usuarios?mensaje=Ya existe una invitación a/de " + usuarios[0].nombre + ".");
```

Si todo lo anterior se cumple se crea una invitación con el email y nombre del emisor y el email del receptor y se procede a insertarla en la base de datos. Tanto si todo sale bien como si se produce algún error durante el proceso se muestra un mensaje informativo en la misma vista de usuarios.

*En caso de que los usuarios ya sean amigos se muestra el texto "Ya sois amigos". Esto es posible ya que al obtener los usuarios a listar añadimos un parámetro a la vista que es el array de amigos del usuario en sesión. También se podría añadir que no muestre la opción de agregar a amigos a usuarios a los que ya se ha enviado una invitación o de los cuales nos ha llegado, pero supondría una consulta más a la hora de listarlos y puede tardar mucho. El enlace para enviar invitación tampoco aparece para el usuario en sesión.

Lista usuarios

Nombre	Apellidos	Email	
Sonia	Garcia	sonia@email.com	Ya sois amigos
Thalía	Cuetos	thalia@email.com	
Edward	Núñez	edward@email.com	Añadir amigo

Caso de uso W7 Usuario registrado: listar las invitaciones de amistad recibidas

Listar las invitaciones de amistad solo requiere una consulta a base de datos. Guardar el nombre del usuario que envía la invitación nos ahorra una consulta más para obtener el usuario emisor.

La vista muestra el email y nombre del usuario que ha enviado la invitación y dos enlaces: aceptar y rechazar invitación. Además, cuenta con una paginación de 5 invitaciones por página para facilitar la navegación por la aplicación.

Caso de uso W8 Usuario registrado: aceptar una invitación de amistad recibida

Al aceptar una invitación de amistad primero se debe comprobar que dicha invitación existe con una consulta a base de datos. Si existe tenemos que añadir al usuario A a los amigos del usuario B y al revés.

Para eso creamos la función **añadirAmigos** en **rinvitaciones.js**. Esta función añade primero el email del usuario A al array de amigos del usuario B mediante la llamada al gestorBD, que a su vez utiliza la función de mongo para añadir elementos a un array: update con $push. Después se hace lo mismo con el usuario A en los amigos de B.

```
/**
 * Añade un usuario a la lista de amigos
 * @param criterio
 * @param amigo
 * @param funcionCallback
 */
añadirAAmigos(criterio, amigo, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('db'), options: function(err, db) {
        if (err) {
            funcionCallback(null);
        } else {
            let collection = db.collection('usuarios');
            collection.update(criterio, document: {$push: { "amigos" : amigo }}, options: function(err, result) {
                if (err) {
                    funcionCallback(null);
                } else {
                    funcionCallback(result);
                }
                db.close();
            });
        }
    });
```

Si todo sale bien se elimina la invitación de la base de datos y se muestra un mensaje informativo, al igual que si se produce algún error. Al igual que enviar una invitación, aceptarla también tarda por las consultas a base de datos.

Caso de uso W9 Usuario registrado: listar los usuarios amigos

A la hora de listar los usuarios amigos del usuario en sesión nosotras pensamos en la forma más fácil de llevarlo a cabo. En nuestro caso decidimos que, en base de datos, cada usuario iba a tener una lista de amigos y dentro de esa lista íbamos a almacenar los emails de aquellos usuarios que son nuestros amigos.

Escogimos los emails ya que representan a cada usuario de forma única puesto que a la hora de registrarse no permitimos emails repetidos. De esta forma no almacenamos datos innecesarios ya que los datos de cada amigo se podrían obtener de otra consulta, además, no sabemos si van a ser siempre necesarios.

En este caso sería coger de req.session.usuario el usuario en sesión y mediante la función **obtenerUsuarios** del módulo gestorBD sacamos los datos de ese usuario y devolvemos únicamente los amigos que serían el array de emails. A partir de ahí si necesitamos los datos de esos amigos con otra consulta usando la misma función anterior los obtendríamos fácilmente, pasándoselos a la vista de visualizar los amigos eso si esta vez paginados ya que es lo que se pide.

A continuación, podemos ver cómo se almacenan los usuarios en base de datos:

```
_id: ObjectId("5eb8efd782145662182285d30")
email: "sonia@email.com"
nombre: "Sonia"
apellidos: "Garcia"
password: "6fabd6ea6f1518592b7348d84a51ce97b87e67902aa5a9f86beea34cd39a6b4a"
amigos: Array
    0: "thalia@email.com"
    1: "rut@email.com"
    2: "edward@email.com"
```


Caso de uso W10 Seguridad

Para controlar la seguridad y comprobar que los usuarios no puedan acceder mediante URL a partes de la aplicación que no deberían sin estar autenticados lo que hacemos es usar el **routerUsuarioSession** que nos permite controlar si hay usuario en sesión. En caso afirmativo dejará correr la petición, en caso contrario se lanzará un mensaje de advertencia y nos devolverá al login. De esta forma controlamos que no se pueda acceder a la lista de usuarios, amigos, peticiones de amistad e invitaciones sin estar autenticado en la aplicación.

La **prueba 22** no la pudimos realizar ya que nosotras obtenemos los amigos del usuario en sesión de esta forma: cogemos el usuario en sesión de req.session.usuario no lo tenemos como parámetro en la URL por lo que la forma en la que se pide acceder a la lista de amigos de otro usuario es imposible ya que mediante URL no se podría llevar a cabo. Para ello tendrías que entrar en la sesión de otro usuario con su contraseña y listar los amigos, pero no es eso lo que se está pidiendo tampoco.

Con el resto de las pruebas de la parte de seguridad no tuvimos ningún problema.

Caso de uso S1 Identificarse con usuario – token

La autenticación desde la API funciona de forma similar al inicio de sesión a través de la aplicación web. La diferencia está en el token: cuando el inicio de sesión es correcto se crea un token mediante el módulo **jsonwebtoken**. Este token se crea a partir del email del usuario encriptado, lo que nos permitirá posteriormente determinar qué usuario es el que está en sesión.

Caso de uso S2 Usuario identificado: listar todos los amigos

Aquí el enunciado de la práctica pedía devolver los identificadores de los amigos, en nuestro caso devolvemos los emails de esos amigos, ya que lo tenemos almacenado así en base de datos. Además, consideramos que los emails son identificadores de los amigos puesto que son únicos y no se repiten por la comprobación que hacemos cuando un usuario se registra.

Haciéndolo de esa forma sólo haríamos una consulta a base de datos ya que en una primera consulta obtenemos los datos del usuario en sesión entre los que se encuentra el array de amigos.

Para controlar que el usuario esté identificado en la aplicación cuando solicita listar los amigos usamos el routerUsuarioToken que permite comprobar si hay token y verifica ese token si no es válido entonces se denegará el acceso sino se almacenará en res.usuario el usuario que viene en el token una vez se desencripta y así no habrá que desencriptarlo nuevamente y se permitirá el acceso a esa URL.

Caso de uso S3 Usuario identificado: Crear un mensaje

Para crear un mensaje es necesario que los dos usuarios sean amigos. Primero se realiza una consulta para obtener el usuario en sesión del que conocemos su email gracias al routerUsuarioToken. Después se comprueba que el email del receptor del mensaje (pasado en el cuerpo de la petición) se encuentra en su lista de amigos.

Si esto se cumple se crea y almacena un mensaje (función **enviarMensaje**) cuyos datos son:

- emisor: email del usuario en sesión.

- receptor: email del usuario destino.

- texto: texto del mensaje, también pasado en el body.

- leido: indica si el usuario receptor ha leído el mensaje. Se establece como false.

- fecha: fecha y hora a la que se envía el mensaje. Útil más adelante para ordenar los amigos por mensaje más reciente.

El mensaje se almacena en la colección mensajes.

```
/**
 * Envia un mensaje con los parámetros indicados
 * El mensaje se envia como no leído y con la fecha actual
 * @param emisor, email del emisor del mensaje
 * @param receptor, email del receptor del mensaje
 * @param texto, texto del mensaje
 */
function enviarMensaje(emisor, receptor, texto, funcionCallback) {
    var mensaje = {
        "emisor" : emisor,
        "receptor" : receptor,
        "texto" : texto,
        "leido" : false,
        "fecha" : new Date()
    }
    gestorBD.insertarMensaje(mensaje,  funcionCallback: function(result){
            funcionCallback(result);
        }
    );
}
```

Caso de uso S4 Usuario identificado: Obtener mis mensajes de una "conversación"

En base de datos los mensajes se guardarán en una colección mensajes en la que cada mensaje va a guardar los datos: emisor, receptor, mensaje, fecha, texto y leído.

```
_id: ObjectId("5eb8efd882145662182285d37")
emisor: "rut@email.com"
receptor: "sonia@email.com"
texto: "Comprobando que funciona"
leido: true
fecha: 2020-05-11T06:25:28.363+00:00
```


Para obtener los mensajes de una conversación lo que hacemos es comprobar que el usuario en sesión y la persona con la que mantenemos la conversación son emisor o receptor de esos mensajes mediante una consulta a mongoDB usando el operador $or. Para ello nos apoyamos en el módulo gestorBD que mediante el método **obtenerMensajes** nos devuelve los mensajes de esa conversación.

Caso de uso S5 Usuario identificado: Marcar mensaje como leído

Marcar un mensaje como leído supone establecer su atributo leído a true. Antes es necesario comprobar que el receptor de dicho mensaje es el usuario identificado, por lo que necesitamos de nuevo el routerUsuarioToken. Además de cambiar el atributo leído a true, se añade uno nuevo llamado releido que después nos permitirá buscar los mensajes que han sido leídos por la otra persona y actualizarlos en la conversación.

Caso de uso C1 Autenticación del usuario

El widget-login.html contiene, al igual que la vista de logueo de la aplicación web, dos inputs para el email y la contraseña, pero en este caso no se trata de un formulario. El botón de Aceptar ejecuta una función que se encarga de hacer una petición POST a api/autenticar (caso S1).

Si la petición tiene éxito se guarda el token devuelto en la variable token y en una Cookie para no perder la información al refrescar la página. Tras identificarse se carga el widget-amigos.html.

Si se produce un error se muestra un mensaje de alerta.

```
$("document").ready( function() {
   $("#boton-login").click(function() {
      $.ajax({
         url: URLbase + "/autenticar",
         type: "POST",
         data: {
            email: $("#email").val(),
            password: $("#password").val()
         },
         dataType: 'json',
         success: function (respuesta) {
            // Guardamos el token en el objeto token
            token = respuesta.token;
            // Lo añadimos a una cookie
            Cookies.set('token', respuesta.token);
            // Redirigimos al usuario a la vista de amigos
            $("#contenedor-principal").load("widget-amigos.html");
         },
         error: function(error) {
            $("#alerta")
               .html("<div class='alert alert-danger'>Usuario no encontrado</div>");
         }
      });
   })
});
```

Caso de uso C2 Mostrar la lista de amigos

A la hora de mostrar la lista de los amigos en la vista widget-amigos.html nos apoyamos en el widget-amigos.js para separar la parte de código de la vista. En este archivo js tenemos varias funciones para separar funcionalidades. Entre ellas se encuentran las funciones **cargarAmigos** que junto con **obtenerDatosAmigos** nos permite cargar los datos de los amigos para mostrar en la vista mediante el método **mostrarUsuarios**.

En esta vista también se muestra el número de mensajes no leídos al lado del nombre de los amigos gracias a **actualizarNoLeidos** que es la función que se ejecuta automáticamente mediante el **setInterval** y **obtenerNoLeidos** que inicialmente cargaría ese número de mensajes.

En esta vista como se requiere que se actualicen los mensajes de forma automática hacemos uso del **setInterval** cuyo valor se asigna a una variable para poder para la ejecución cuando se cambia de widget.

También se nos pide que cuando se pulsa sobre un amigo se abra el chat con este lo que logramos gracias al método **abrirChat** que nos permite cargar el html donde se muestra el chat.

A la hora de filtrar los amigos por nombre en la vista lo hacemos con un evento on sobre el cuadro de texto donde se introduce el filtro y se compara el nombre de todos los amigos con el del cuadro de búsqueda mirando si lo contiene. Para no limitar esa búsqueda decidimos pasar a minúsculas con el método toLowerCase() y así dar más posibilidades de encontrar de forma satisfactoria el nombre buscado.

Caso de uso C3 Mostrar los mensajes

En este caso quisimos que nuestro chat fuera parecido a una app de mensajería por lo que nos guiamos por un css que encontramos en Internet y lo adaptamos a nuestro gusto. De esta forma, los mensajes que el usuario en sesión envía en un chat se mostrarán a la derecha y los que nos envía el otro usuario a la

izquierda. Además, decidimos añadir la imagen del ojo para cuando un mensaje es leído. En el fondo hay una cajita donde se puede introducir el mensaje que queremos mandar y pulsando sobre el botón enviar aparecerá en el chat.

Para conseguir esto volvimos a encapsular la parte del código JavaScript en un archivo a parte llamado widget-mensajes.js donde hacemos uso de las siguientes funciones:

- **leerMensajes:** que permitiría marcar como leídos los mensajes una vez se accede al chat
- **cargarMensajes:** que simplemente carga de base de datos los mensajes del amigo seleccionado y mediante el método **mostrarMensajes** permitirá que se visualicen en la vista.
- **actualizarMensajes:** carga los mensajes que no están leídos. Tuvimos que meter un booleano porque a veces se repetían los mensajes
- **comprobarMensajesLeidos:** encarga de actualizar los cambios en los estados leído
- **marcarComoLeidos:** inserta a los mensajes el símbolo de leídos, el ojo
- **cargarNombreAmigo:** nos muestra en el html el nombre de la persona con la que chatea el usuario en sesión
- **enviarMensaje:** se crea el mensaje y se muestra en el chat
- **updateScroll:** permite actualizar el scroll para visualizar mejor los mensajes

Caso de uso C4 Crear mensaje

En el widget-mensajes.html se incluye un input y un botón para enviar un mensaje a la conversación. El botón tiene establecida la función **enviarMensaje**, que se encarga de hacer una petición POST a /mensaje (caso S3) con el texto escrito en el input y el amigo con el que estamos manteniendo la conversación. Es necesario pasarle también el token ya que, como hemos comentado anteriormente, el routerUsuarioToken se encarga de comprobarlo para obtener el usuario identificado.

Tras enviar el mensaje lo añadimos también al html y vaciamos el input.

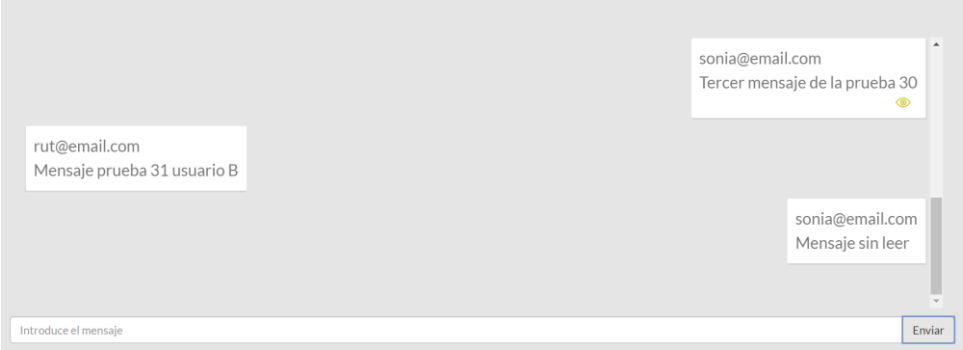
```
function enviarMensaje() {
    // Comprobamos que el mensaje no está vacío
    if($("#messageContent").val().length == 0 || !$("#messageContent").val()) {
        return;
    }
    $.ajax({
        url: URLbase + "/mensaje",
        type: "POST",
        data: {
            receptor: amigoSeleccionado, texto: $("#messageContent").val()
        },
        headers: { "token": token },
        dataType: 'json',
        success: function (respuesta) {
            var mensaje = [ {...}];
            mostrarMensajes(mensaje);
            $("#messageContent").val("");
            updateScroll();
        },
        error: function(error) {
            $("#alerta")
                .html("<div class='alert alert-danger'>Se ha producido un error al enviar el mensaje</div>");
        }
    });
```

Caso de uso C5 Marcar mensajes como leídos de forma automática

Al entrar en una conversación es necesario marcar como leídos todos los nuevos mensajes recibidos, ya que si no se cargan dos veces. Además, cada cierto tiempo los nuevos mensajes se marcan como leídos.

Para hacerlo disponemos de la función **leerMensajes** del widget-mensajes.js. Esta función realiza una petición POST a /mensajes/marcarLeidos (caso S5) pasándole el amigo con el cual estamos hablando.

Por otra parte, los mensajes enviados y que han sido leídos por la otra persona muestran un icono de un ojo (lo que sería el doble tick azul en Whatsapp). No hemos considerado necesario que los mensajes recibidos muestren el estado de leído, ya que serían todos los de la conversación. Al mostrar los mensajes cargados al abrir la conversación comprobamos si han sido leídos para mostrar o no la imagen.



Pero para poder actualizar en tiempo real el estado es necesario comprobar cada cierto tiempo que mensajes han sido leídos y aún no han sido comprobados. Aquí es dónde utilizamos el atributo mencionado en el caso S5: releído. Si un mensaje enviado a la persona identificada aparece como leído y no releído se debe actualizar en la conversación y mostrar la imagen del ojo.

La función **comprobarMensajesLeidos** se encarga de actualizar los cambios en los estados leído. Para ello realiza una petición GET a una nueva función de la API: /mensajes/leidos. La API se encarga de buscar los mensajes leídos y no releídos enviados por el usuario del chat al usuario identificado y, además, de marcarlos como releídos.

En caso de éxito se actualizan todos los mensajes que nos ha devuelto la petición.

```
/**
 * Carga los mensajes leidos
 */
function comprobarMensajesLeidos() {
    if(amigoSeleccionado == "")
        return
    $.ajax({
        url: URLbase + "/mensajes/leidos/" + amigoSeleccionado,
        type: "GET",
        data: { },
        dataType: 'json',
        headers: {"token": token},
        success: function (respuesta) {
            marcarComoLeidos(JSON.parse(respuesta));
            actualizado = true;
        },
        error: function (error) {
            $("#contenedor-principal").load("widget-login.html");
        }
    });
}
```

```
/**
 * Permite marcar como leidos los mensajes mostrando la imagen del ojo
 * @param mensajes mensajes del chat
 */
function marcarComoLeidos(mensajes) {
    for(var i = 0; i < mensajes.length; i++) {
        $("#" + mensajes[i]._id.toString()).html("<img class='imagenVisto' src='/img/visto.png' />");
    }
}
```

Esta función se ejecuta cada cierto tiempo gracias al setInterval.

Caso de uso C6 Mostrar el número de mensajes sin leer

Para mostrar el número de mensajes sin leer que el usuario tiene en una conversación con un amigo es necesario realizar una petición GET a /mensajes/noLeidos para cada amigo que aparece en la lista, siendo el receptor el usuario identificado y el emisor el amigo. El número a mostrar es la longitud del array devuelto en la petición.

La función **actualizarNoLeidos** es la que se encarga de realizar este proceso y se ejecuta a la hora de cargar los amigos y cada cierto intervalo de tiempo (de nuevo setInterval) para que se actualice en tiempo real.

```
/**
 * Actualiza el numero mensajes no Leídos
 */
function actualizarNoLeidos() {
    $(".amigo").each( function (index : MongoError ) {
        var amigo =  $(this).text()
        $.ajax({
            url: URLbase + "/mensajes/noLeidos/" + amigo,
            type: "GET",
            data: { },
            dataType: 'json',
            headers: {"token": token},
            success: function (respuesta) {
                $("[name ='numNoLeidos" + amigo + "']").html(
                    JSON.parse(respuesta).length == 0 ? "" : JSON.parse(respuesta).length
                );
            },
            error: function (error) {
                $("#contenedor-principal").load("widget-login.html");
            }
        });
    });
}
```

Caso de uso C7 Ordenar la lista de amigos por último mensaje

A la hora de mostrar los amigos cuando accedemos a la lista estos se ordenarán en función de la antigüedad del último mensaje. Para llevar a cabo esta funcionalidad lo que hacemos es obtener primero el mensaje más reciente de cada amigo gracias a la función **obtenerUltimoMensajeDelAmigo** que lo que hace es listar los mensajes en los que el amigo es emisor o receptor y sacar el más reciente comparando las fechas de dichos mensajes. Tras esto mediante la función **ordenarAmigos** y usando el método sort de los arrays comparamos la fecha que nos devuelve el método de los mensajes para cada uno de los amigos de la lista y las vamos comparando. Una vez hecho esto ya tenemos ordenada la lista de amigos que se mostrará en la vista cuando se haga la petición get /api/amigos.

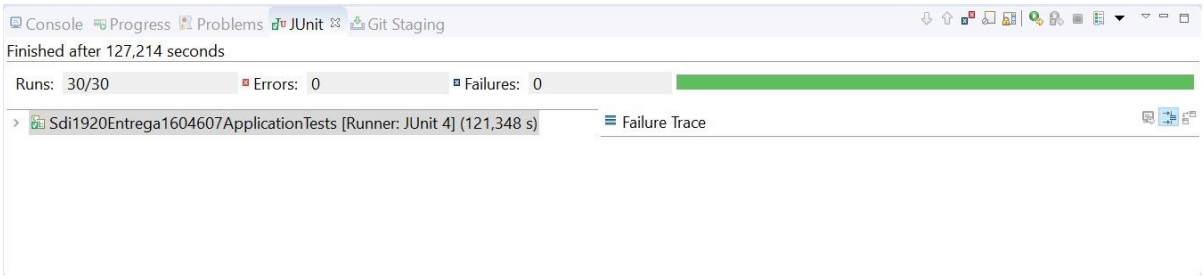
Información necesaria para el despliegue y ejecución

Simplemente decir que antes de arrancar el proyecto de pruebas se debería arrancar el proyecto de Node.Js. También nos gustaría recordar que la prueba 22 no la implementamos por las razones que comentamos anteriormente.

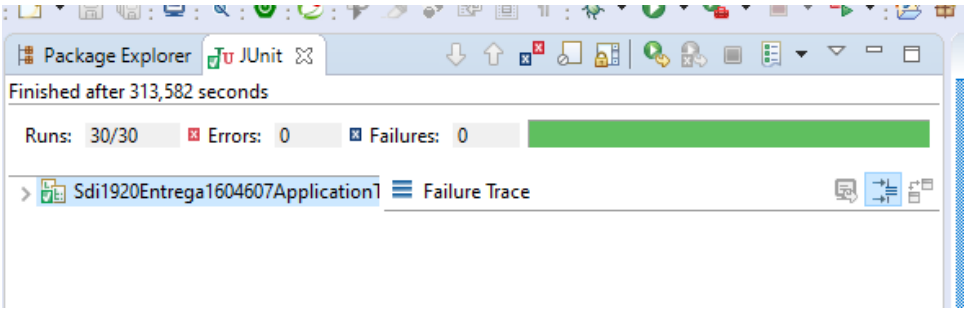
Además de un logger que imprime por pantalla también nuestro logger almacena la traza en un fichero que se guarda en una carpeta llamada logs.

Esperemos que las pruebas se ejecuten de forma correcta como nos pasó a nosotras cuando las ejecutamos. Además, cada vez que se ejecutan las pruebas se resetea la base de datos y se insertan los datos necesarios para la realización de estas, por lo que se podrían ejecutar las pruebas todas las veces que se quisiera.

A continuación, nos gustaría poner una captura para demostrar que nos pasaron las pruebas



Pruebas ejecutadas a 11/05/2020 20:52



Observaciones: a veces algunas pruebas como la 12, 13, 25, 26 o 31 no pasan porque esperan algún elemento más en la lista por ejemplo esperan 3 amigos y solo encuentran 2 y la 31 a veces no es capaz de ver que está bien hecha la ordenación cuando sí lo está. No entendemos muy bien a que se debe este comportamiento, pero suponemos que serán el número de peticiones que se hace a la base de datos que puede llegar a colapsar de alguna forma o a negarnos esos datos. De todas formas, podemos ver que el 95% de las veces las pruebas se ejecutan con éxito.

Conclusión

Nos gustaría comentar que trabajamos muy a gusto en equipo y creemos que el trabajo lo repartimos de forma equitativa entre las dos, además nos ayudamos continuamente y estuvimos en contacto en todo momento. Le dedicamos muchas horas a este proyecto a pesar de que fuimos algo justas de tiempo por las demás entregas que se nos juntaron todas en estas últimas semanas. Sufrimos un poco en algunos momentos con algunos problemas de programación como por ejemplo el setInterval que se nos volvió un poco loco y tuvimos que buscar una solución la cual fue usar el clearInterval una vez se salía del widget correspondiente. Por lo demás nos gustó trabajar con esta tecnología, aunque al principio costó un poco adaptarse a ella. Esperemos que nuestro trabajo de sus frutos.