Red Social

# DOCUMENTACIÓN SDI PRÁCTICA 2

Carlos Sanabria Miranda UO250707

# Tabla de contenido

# Despliegue

La base de datos se trata de una base de datos **MongoDB en la nube**. No hace falta arrancar nada, pero es necesario tener conexión a Internet.

La aplicación se ejecuta en el **puerto 8081**.

Para lanzar los tests, se debe ejecutar la clase **Tests.java**.

A la hora de probar los tests, **habrá que cambiar la línea correspondiente al Path de Firefox**, para que corresponda con la ruta donde se encuentra dicho navegador en la máquina en la que se van a ejecutar.

```
static String PathFirefox = "<ruta_navegador_firefox_46.0>”
```

**Los tests están pensados para ejecutarse en el orden en el que vienen**, debido a que en algunos de ellos se crean invitaciones y relaciones de amistad que se necesitan para ejecutar tests posteriores. He considerado que al crearse documentos que se utilizan en tests posteriores, queda más claro que las pruebas están funcionando correctamente, ya que no se tiene acceso a la base de datos para comprobar el contenido de la misma.

Aunque algunos tests, como por ejemplo el de logearse, no necesitan ejecutar otros tests previamente.

# Descripción de los Casos de Uso (Web)

## 1. Público: registrarse como usuario

Se dispone de una vista "**signup.html**" con un formulario con los campos: email, name, lastName, password y passwordConfirm. Dicho formulario hace una petición POST a la URL **"/signup"**.

En **rusers.js** hay un método que recibe la petición **GET "/signup"** y devuelve la vista anterior, y otro que recibe la petición **POST "/signup"**.

Este último, comprueba que las passwords coincidan, encripta la password, y crea un objeto "user" con los campos recibidos del formulario. Por último, llama al método **insertUser**, del módulo **gestorBD**, para insertar el objeto "user" en la base de datos. Si se produjo un error se redirige a "/signup" y se muestra un mensaje de error, mientras que, si se insertó correctamente, se redirige a "/login" y se muestra un mensaje de éxito.

El método **insertUser** comprueba que no exista ya un documento con ese mismo email en la colección **'users'**, y en caso de no existir, lo inserta.

## 2. Público: iniciar sesión

Se dispone de una vista **"login.html"** con un formulario con los campos: email y password. Dicho formulario hace una petición POST a la URL **"/login"**.

En **rusers.js** hay un método que recibe la petición **GET "/login"** y devuelve la vista anterior, y otro que recibe la petición **POST "/login"**.

Este último, encripta la password y crea un objeto "criterio" con el email y la password encriptada. Por último, llama al método **getUsers**, del módulo **gestorBD**, pasándole el criterio. Si el resultado de la búsqueda anterior devuelve un array vacío, es que no existe un usuario con esas credenciales, por lo que se muestra un mensaje de error en las credenciales. Si el array no es vacío, el usuario existe, por lo que **se guarda su email en un atributo de la sesión** (para indicar que está autenticado), y se le redirige al listado de todos los usuarios (**"/user/list"**).

El método **getUsers** simplemente retorna los documentos de la colección **'users'** que coincidan con el criterio que se le pasa.

## 3. Usuario registrado: listar todos los usuarios de la aplicación

Se dispone de una vista **"user/list.html"** con una tabla. La vista recorre una lista de usuarios, y por cada uno de ellos muestra su nombre y su email en una fila de la tabla.

La paginación se ha implementado como un fragmento. Al final la vista anterior, se realiza un include del fragmento **"/fragments/pagination.html"**, al que se le pasa la URL "user/list", para que sea la que utilicen los enlaces de paginación (en este punto se explica más en detalle la paginación).

En **rusers.js** hay un método que recibe la petición **GET "/user/list"** y devuelve la vista anterior. En dicho método, se obtiene el número de página de la petición (si no existe se pone a 1) y se llama al método **getUsersPg** del módulo **gestorBD**, pasándole el número de página y un criterio vacío (suponemos que no hay parámetro de búsqueda, el cuál se explicará en el punto 4). Si no se produjo ningún error, calculamos cuál es la última página y obtenemos el array con los usuarios de la página actual, y pasamos ambos, junto a la página actual, a la vista.

El método **getUsersPg** es similar a **getUsers**, pero devuelve sólo los usuarios de la página indicada.

Para establecer el número de elementos que se muestra en cada página se ha definido una variable de aplicación **'itemsPerPage'** con valor 5 ( app.set('itemsPerPage', 5) ).

Para añadir una opción de menú que permita acceder al listado, se ha incluido en la vista **"base.html"** un dropdown-menu con una opción que tiene un enlace a dicho listado. Dicho menú solo aparece si la variable **"email"** que se la pasa a **todas las vistas** NO es null. El valor de esa variable se obtiene del atributo email guardado en sesión, que indica si hay un usuario autenticado o no (si tiene valor es que el usuario con ese email está autenticado).

## 4. Usuario registrado: buscar entre todos los usuarios de la aplicación

Se parte de la vista y el método que recibe la petición **GET "/user/list"** indicados en el punto anterior.

En la vista se añade un **formulario** con un único campo, que es el texto a buscar, y que realiza una petición **GET** a la URL **"/user/list"**, que es recogida por el método anterior.

Dicho método, cuando recibe una petición con el parámetro "searchText", crea un criterio que busque coincidencias parciales (con insensibilidad de mayúsculas o minúsculas) del texto indicado, tanto en el email, como en el nombre de los usuarios. El criterio es el siguiente:

```
criterio = {$or: [
// Opcion i: Case insensitivity to match upper and lower cases
            {"email" : {$regex : ".*"+searchText+".*", $options: "i" }},
            {"name"  : {$regex : ".*"+searchText+".*", $options: "i" }}
] };
```

El resto del funcionamiento es similar al punto anterior.

## 5. Usuario registrado: enviar una invitación de amistad a un usuario

En la vista **"user/list.html"** se añade un botón "Agregar amigo" por cada usuario, siempre y cuando el usuario no sea el mismo que el usuario en sesión (esto se sabe gracias al campo **"canInvite"** que se le añade a cada objeto "user" pasado a la vista, en el método **addCanInviteToUsers**, llamado desde dentro del método que recibe la petición **GET "/user/list").**

Cada uno de esos botones es un enlace a la URL **"/user/invitate/<email>"**, siendo <email> el email del usuario de esa fila de la tabla.

En **rinvitations.js** hay un método que recibe la petición **GET "/user/invitate/:email"**. En él, se comprueba primero que el email pasado en la URL no sea el mismo que el email del usuario en sesión. Se crea un objeto "invitation" con los dos emails mencionados (el del usuario en sesión es el email "emisor" y el pasado como parámetro es el email "receptor") y se le pasa al método **insertInvitation** del módulo **gestorBD**.

El método **insertInvitation** comprueba:

- Que exista el usuario con ese "email receptor"
- Que no exista ya una invitación con esos emails "emisor" y "receptor"
- Que no exista la invitación inversa (es decir, que el usuario al que le quieres enviar una invitación no te haya enviado ya una a ti)
- Que los usuarios con esos emails no sean amigos

Una vez realizadas todas las comprobaciones, pasa a insertar el objeto en la colección **'invitations'**.

## 6. Usuario registrado: listar las invitaciones de amistad recibidas

Se dispone de una vista **"user/invitations.html"** con una tabla. La vista recorre una lista de invitaciones, y por cada una de ellas muestra el nombre del usuario que la ha enviado (a través del campo "senderUser" de la invitación, que se explica más adelante de donde ha salido, puesto que las invitaciones en base de datos sólo tienen los emails "emisor" y "receptor").

Al final de dicha vista, se incluye el fragmento de paginación, de forma similar a la explicada en el punto 3.

En **rinvitations.js** hay un método que recibe la petición **GET "/user/invitations"** y devuelve la vista anterior. En dicho método, se obtiene el número de página de la petición (si no existe se pone a 1) y se llama al método **getInvitationsPg** del módulo **gestorBD**, pasándole el número de página y un criterio (que el "email receptor" sea el email del usuario en sesión). Con esto, obtenemos todas aquellas invitaciones que han sido enviadas al usuario en sesión, en la página indicada.

Con dichas invitaciones, se realizan los siguientes pasos:

1. Se sacan los emails de los usuarios que las han enviado
2. Se obtienen de la base de datos los usuarios con esos emails
3. A cada invitación de la lista de invitaciones se le añade un nuevo atributo "senderUser", que va a ser el usuario que tiene el mismo email que el campo "senderEmail" de dicha invitación

Por último, si no se produjo ningún error, le pasamos a la vista la última página, la página actual y la lista de invitaciones (que ahora tienen el nuevo campo "senderUser").

El método **getInvitationsPg** es similar a **gerUsersPg**.

## 7. Usuario registrado: aceptar una invitación recibida

En la vista **"user/invitations.html"**, para cada invitación de la tabla se muestra también un botón para aceptar la invitación, cada uno de los cuales es un enlace a la URL **"/user/accept/:idInvitation"**, siendo <idInvitation> el id de la invitación de esa fila de la tabla.

En **rinvitations.js** hay un método que recibe la petición **GET "/user/accept/:idInvitation"**. En dicho método se realizan los siguientes pasos:

1. Se obtiene la invitación con ese id
2. Se comprueba que el usuario en sesión es realmente el receptor de la invitación (que el campo "receiverEmail" de la invitación coincida con el email del usuario en sesión)
3. Se crea un objeto friendship con los emails de la invitación y se inserta en la base de datos mediante el método **insertFriendship** del módulo **gestorBD**
4. Se elimina la invitación de la base de datos mediante el método **removeInvitation** del módulo **gestorBD**

El método **insertFriendship** inserta el objeto pasado como parámetro dentro de la colección **'friends'**.

El método **removeInvitation** elimina las invitaciones que cumplan el criterio pasado como parámetro de la colección **'invitations'**.

Por último, si no se produjo ningún error, se redirige a **"user/friends"**, mostrando un mensaje de éxito.

## 8. Usuario registrado: listar los usuarios amigos

Se dispone de una vista **"user/friends.html"** con una tabla. La vista recorre una lista de usuarios amigos, y por cada uno de ellos muestra su nombre y su email.

Al final de dicha vista, se incluye el fragmento de paginación, de forma similar a la explicada en el punto 3.

En **rusers.js** hay un método que recibe la petición **GET "/user/friends"** y devuelve la vista anterior. En dicho método, se obtiene el número de página de la petición (si no existe se pone a 1) y se llama al método **getFriendshipsPg** del módulo **gestorBD**, pasándole el número de página y un criterio (que el email del usuario en sesión sea uno de los dos emails de un documento de la colección **'friends'**). Con esto, obtenemos todos los amigos del usuario en sesión, en la página indicada.

Con dichos objetos de la colección 'friends', se realizan los siguientes pasos:

1. Se sacan los emails de los amigos del usuario en sesión
2. Se obtienen de la base de datos los usuarios con esos emails

Por último, si no se produjo ningún error, le pasamos a la vista la última página, la página actual y la lista de usuarios amigos.

El método **getFriendshipsPg** es similar a **gerUsersPg**.

## Paginación

Para que los enlaces de paginación funcionen correctamente, es necesario, al incluir el fragmento de paginación, pasarle la URL de la página, de la siguiente forma:

```
{% include "../fragments/pagination.html" with {url: "/user/list"} %}
```

Dentro del fragmento, en caso de que exista una búsqueda, se obtiene su valor y se concatena con un string para añadirlo como parámetro en la url de los enlaces de paginación, de la siguiente forma:

```
{% if searchText != null %}
{% set stGetParam = "&searchText="+searchText %}
{% endif %}
```

Así, por ejemplo, para el enlace de paginación de la primera página, el valor del atributo href es el siguiente:

```
href="{{ url }}?pg=1{{ stGetParam }}"
```

De este modo, en aquellos casos en los que haya una búsqueda, se añadirá al final del enlace el parámetro con el texto de búsqueda, y cuando no lo haya no.

# Descripción de los Casos de Uso (API REST)

## S.1 Identificarse con usuario – token

Se define la URI "/api/autenticar", en la cual, mediante el método POST, se enviarán en el body de la petición los parámetros **email** y **password**. El SW comprueba si existe un usuario con esas credenciales, buscando en la colección de usuarios.

En caso de existir, crea un token encriptando el email del usuario y la fecha actual en segundos, y lo retorna, junto al código 200 OK.

En caso de no existir, devuelve el código 401 Unauthorized, y el mensaje "Inicio de sesión no correcto".

## Comprobar token de seguridad válido

Se define un router en app.js, llamado routerUsuarioToken, que comprueba si hay un token válido (y no caducado, es decir, que no hayan pasado mas de 8 min desde que se expidió) en la petición.

En caso de haberlo, se guarda el email del usuario que estaba encriptado en el token en la respuesta y se deja pasar la petición.

En caso de no haberlo, se devuelve un mensaje de error y el código 403 Forbidden.

Este router se va a aplicar a las URLS: "/api/friend", "/api/message" y "/api/user".

## S.2 Usuario identificado: listar todos los amigos

Se define la URI "/api/friend", a la cual se le realiza una petición GET. Recupera el email del usuario que almacenó el router en la respuesta y realiza en la BD una búsqueda de sus amigos.

Si se produce un error, se devuelve un mensaje de error y el código 500 Server Internal Error.

Si no, se obtiene el email de los amigos y se retorna una lista con ellos, junto al código 200 OK.

## S.3 Usuario identificado: Crear un mensaje

Se define la URI "/api/message", en la cual, mediante el método POST, se enviarán en el body de la petición los parámetros **destino** (email del usuario al que se envía el mensaje) y **texto** (contenido del mensaje).

El SW añade al mensaje el campo **emisor** (email del usuario autenticado, que almacenó el router en la respuesta) y el campo **leído** con valor false.

Si los datos de entrada no son validos se devuelve un mensaje de error y el código 400 Bad Request.

Antes de insertar el mensaje en la BD se comprueba:

- Que exista el destinatario del mensaje (si no existe se devuelve un mensaje de error y el código 404 Not Found)
- Que el usuario autenticado y el destinatario sean amigos (si no, se devuelve un mensaje de error y el código 403 Forbidden)

Si todo es correcto, se inserta y se devuelve el id del mensaje insertado, junto al código 201 Created.

Si se produce algún error durante todo el proceso, se devuelve un mensaje de error y el código 500 Server Internal Error.

## S.4 Usuario identificado: Obtener mis mensajes de una conversación

Sobre la URI "/api/message", se le realiza una petición GET, pasándole como parámetros dos emails, por ejemplo: /api/message?user1=user10@gmail.com&user2=user11@gmail.com

Si los datos de entrada no son validos se devuelve un mensaje de error y el código 400 Bad Request.

Antes de devolver los mensajes se comprueba:

- Que existan los dos usuarios con esos emails (si no existe alguno de los dos, se devuelve un mensaje de error y el código 404 Not Found)
- Que el email del usuario autenticado coincida con uno de esos dos emails (si no, se devuelve un mensaje de error y el código 403 Forbidden)

Si todo es correcto, se recuperan de la BD los mensajes entre los usuarios con esos dos emails, y se cambian los campos **emisor** y **destino** de los mensajes por **/api/user/emisor** y **/api/user/destino** respectivamente, para cumplir el principio **HATEOAS**. Se devuelven dichos mensajes, junto al código 200 OK.

Si se produce algún error durante todo el proceso, se devuelve un mensaje de error y el código 500 Server Internal Error.

## S.5 Usuario identificado: Marcar mensaje como leído

Se define la URI "/api/message/:id", en la cual, mediante el método PUT, se enviará en el body de la petición el parámetro **leido** a true.

El SW comprueba que exista el campo leído con valor true en el body de la petición, si no, se devuelve un mensaje de error y el código 400 Bad Request.

Se obtiene el mensaje con ese id de la BD (si no existe se devuelve un mensaje de error y el código 404 Not Found) y, antes de actualizar el mensaje con el campo leído a true, se comprueba:

- Que el usuario autenticado sea el receptor del mensaje (si no, se devuelve un mensaje de error y el código 403 Forbidden)

Si todo es correcto, se actualiza el mensaje y se devuelve el id del mensaje actualizado, junto a un mensaje de éxito y el código 200 OK.

Si se produce algún error durante todo el proceso, se devuelve un mensaje de error y el código 500 Server Internal Error.

Inicialmente se había diseñado la URI "/api/message/:id/leido", que actualizaba solamente el campo leído del mensaje a true, sin necesidad de pasarle nada en el body, pero, tras debatirlo con el profesor, se llegó a la conclusión de que la forma elegida es más genérica y permite ampliar en un futuro la funcionalidad de la API más fácilmente; aunque ambas valdrían.

## Obtener los datos de un usuario, dado su email

Se ha definido también la URI "/api/user/:email", en la cual, mediante el método GET, se obtiene toda la información del usuario con ese email.

Si se produce un error, se devuelve un mensaje de error y el código 500 Server Internal Error.
Si el usuario con ese email no existe, se devuelve un mensaje de error y el código 404 Not Found.
Si existe, se devuelve el usuario y el código 200 OK.

# Logger

Para llevar un log de lo que se realiza en la aplicación, se ha utilizado el módulo **log4js**.

Se ha configurado para que la salida se vaya mostrando por consola con colores en función del tipo de mensaje de log, y para que también se vaya guardando dicha salida en un fichero, llamado **uo250707.log**, dentro del directorio **logs**.

Se han creado los módulos **gestorLog** y **gestorLogApi**, que contienen una referencia al logger y una serie de métodos para crear un mensaje de log por cada operación que se puede realizar en la aplicación (el primer módulo para la parte de la aplicación web y el segundo para la parte de la API Rest).

# Descripción de los Casos de Uso (Cliente REST)

Se ha separado el código JavaScript del código HTML y creado un fichero /css/cliente.css para modificar algunos aspectos del css de Bootstrap.

## C.1. Autenticación del usuario

El widget **widget-login.html** contiene dos campos para introducir el email y la contraseña y un botón.

La clase **widget-login.js** define un escuchador del click del botón. Cuando se realiza el click, obtiene los valores de los campos anteriores (si alguno está vacío se muestra un mensaje) y, utilizando AJAX, realiza una petición POST a la URL **/api/autenticar**, enviando el email y el password en el body.

Si la petición tiene éxito, se guardan el email del usuario y el token en variables globales y en cookies y se carga el **widget-friends**.

Si no, se muestra un mensaje de error.

## C.2. Mostrar la lista de amigos

El widget **widget-friends.html** contiene un campo para filtrar por nombre los amigos y una tabla donde se irán mostrando el email y el nombre de los amigos.

La clase **widget-friends.js** realiza una petición GET a la URL **/api/friend**, enviando el token en la cabecera. Esto retorna la lista de emails, y para cada uno de ellos, se realiza una petición GET a la URL **/api/user/:email**, enviando el token en la cabecera, para obtener la información de cada amigo, que se va guardando en un array global y añadiendo a la tabla.

Define también un escuchador sobre el campo para filtrar los amigos. Cuando se escribe en dicho campo, se obtiene el valor escrito y se guardan en un array sólo aquellos amigos (del array global de amigos) en los que el texto escrito esté contenido en su nombre. Por último, se vacía la tabla y se añaden los usuarios guardados en dicho array.

## C.3. Mostrar los mensajes

La clase **widget-friends.js**, al insertar los datos de los amigos en la tabla, mete el nombre del usuario en un <a></a>, que realiza una llamada a una función para guardar el email del usuario seleccionado en una variable global y en una cookie y cargar el **widget-chat**.

El widget **widget-chat.html** contiene una tabla con dos columnas, en la que se van a insertar los mensajes, y un campo con un botón, para crear un nuevo mensaje.

La clase **widget- chat.js** realiza una petición GET a la URL **/api/message**, enviando como parámetros el email del usuario autenticado y el email del usuario seleccionado, además del token en la cabecera. Esto retorna la lista de mensajes entre los dos usuarios, y cada uno de ellos se añade a la tabla (si el receptor del mensaje es el usuario en sesión, se añade el texto del mensaje en la columna de la izquierda, y si no, en la de la derecha).

Se establece también un **setInverval()**, que cada ***UPDATE_TIME*** segundos (variable global definida en cliente.js) llama a la función que realiza una petición a la API Rest para obtener los mensajes y cargarlos en la tabla.

## C.4. Crear mensaje

La clase **widget-chat.js** tiene una función que es llamada cuando se hace click en el botón de enviar un mensaje.

Dicha función, comprueba que el texto a enviar no sea vacío, y en caso de no serlo, realiza una petición POST a la URL **/api/message**, enviando en el body los campos **destino** (cuyo valor es el email del usuario con el que chateas) y **texto** (cuyo valor es el texto obtenido del campo), además del token en la cabecera.

## C.5. Marcar mensajes como leídos de forma automática

En la clase **widget- chat.js**, al añadir un mensaje a la tabla, si el mensaje está leido se le añade al final el texto "<leido>".

Al añadirlo, cuando se comprueba si eres el receptor del mensaje para ponerlo en la columna de la izquierda, si el mensaje NO está leido, se realiza una petición **PUT** a la URL **/api/message/:id**, enviando en el body el campo **leido** con valor **true**, además del token en la cabecera. Con esto, estamos marcando dicho mensaje como leído.

Como el proceso que añade mensajes a la tabla se realiza cada ***UPDATE_TIME*** segundos, se van a marcar los mensajes como leídos de forma automática.

## C.6. Mostrar el número de mensajes sin leer

Cada ***UPDATE_TIME*** segundos se comprueba el número de mensajes sin leer en cada chat, se ordenan los amigos por antigüedad de ultimo mensaje en su chat, y se actualiza la tabla de amigos.

Para comprobar el número de mensajes sin leer en el chat de cada usuario, se realiza una petición **GET** a la URL **/api/message**, enviando como parámetros el email del usuario autenticado y el email del usuario seleccionado, además del token en la cabecera. Una vez obtenidos todos los mensajes con ese usuario, se cuenta cuantos tienen como destino el usuario en sesión y están sin leer, y se le añade ese valor a un campo del usuario actual del array de usuarios que se muestran actualmente.

A la hora de añadir un amigo a la tabla, se muestra junto a su nombre dicho número de mensajes sin leer, entre corchetes.

## C.7. Ordenar la lista de amigos por último mensaje

Como se comenta en el punto anterior, cada ***UPDATE_TIME*** segundos se ordenan los amigos por antigüedad de ultimo mensaje en su chat.

Para ello, cuando se obtienen todos los mensajes entre el usuario autenticado y cada amigo, se saca el ultimo mensaje del array de mensajes, y de su **_id** se obtiene la fecha de creación del mensaje (dicha fecha está almacenada en los 4 primeros bytes del id). Se establece esa fecha como tiempo del ultimo mensaje con ese amigo. Si no hay mensajes con ese amigo, se establece null como fecha.

Para ordenar los amigos, se utiliza la fecha del ultimo mensaje con dicho amigo, y en caso de que sea null, aparecerá abajo en la lista (para ello se usa la función sort de los arrays). Se ordenan tanto el array que tiene todos los amigos, como el array que tiene los amigos que se están mostrando actualmente en la tabla (filtrados por el nombre).

# Descripción de las Pruebas Unitarias (Web)

Antes de cada ejecución de los tests, se realiza una conexión a la base de datos para eliminar todos los documentos de las colecciones 'invitations', 'friends' y 'messages', y se eliminan los usuarios 'newUser@gmail.com' y 'notExists@gmail.com' para que puedan ejecutarse repetidas veces sin problemas.

## 1.1 [RegVal] Registro de Usuario con datos válidos.

Va a la página de signup, introduce datos válidos (entre ellos un email que no existe ya en la aplicación) y comprueba que le lleva a la página de login y se le muestra el mensaje "Usuario registrado correctamente".

## 1.2 [RegInval] Registro de Usuario con datos inválidos (repetición de contraseña inválida).

Va a la página de signup e introduce datos válidos salvo la contraseña, que al repetirla la introduce incorrectamente. Comprueba que le lleva a la página de signup, mostrándole el mensaje de error "Las contraseñas no coinciden".

## 2.1 [InVal] Inicio de sesión con datos válidos.

Va a la página de login, introduce datos válidos de un usuario existente en la aplicación (user01@gmail.com), y comprueba que inicia sesión correctamente, es decir, que le lleva a una página en la que aparece el siguiente texto "Usuario autenticado: <email>", donde <email> es el email del usuario que se acaba de autenticar.

Finalmente, se deslogea y comprueba que le lleva a la página de login (aparece el texto "Identifícate como usuario"), y aparece también el texto "Desconectado correctamente".

## 2.2 [InInVal] Inicio de sesión con datos inválidos (usuario no existente en la aplicación).

Va a la página de login, introduce datos inválidos (el usuario con esos datos no existe) y comprueba que le lleva de nuevo a la página de login (aparece el texto "Identifícate como usuario") y se muestra el mensaje de error "Email o password incorrecto".

## 3.1 [LisUsrVal] Acceso al listado de usuarios desde un usuario en sesión.

Va a la página de login, introduce datos válidos de un usuario existente en la aplicación (user01@gmail.com), y comprueba que inició sesión correctamente (como se indica en 2.1).

Hace click en el nav en la opción "Usuarios" y espera a que se despliegue el dropdown-menu y aparezca la opción "Ver Todos". Hace click en esa opción para ir a la página con el listado de todos los usuarios y comprueba que aparece el texto "Todos los usuarios".

(Se podría comprobar que aparece el texto “Todos los usuarios” directamente al iniciar sesión, ya que un inicio de sesión correcto nos redirige a la página con el listado de todos los usuarios, pero haciéndolo de esta forma nos aseguramos de que funciona correctamente la opción del nav para ir a dicha página.)

Comprobamos que hay 5 usuarios en la pagina actual (se parte de al menos 15 usuarios en la BD).

Finalmente, se deslogea (como se indica en 2.1).

## 3.2 [LisUsrInVal] Intento de acceso con URL desde un usuario no identificado al listado de usuarios desde un usuario en sesión.

**(Debe producirse un acceso no permitido a vistas privadas)**

Se intenta ir directamente a la URL "/user/list" desde un usuario no logeado.

Se comprueba que se envía al usuario a la página de login (aparece el texto "Identifícate como usuario") y se muestra el mensaje de error "Debes identificarte primero para acceder a esa página".

## 4.1 [BusUsrVal] Realizar una búsqueda valida en el listado de usuarios desde un usuario en sesión.

Va a la página de login, introduce datos válidos de un usuario existente en la aplicación (user01@gmail.com), y comprueba que inició sesión correctamente (como se indica en 2.1).

Se hace click en el campo de búsqueda, se borra su contenido y se introduce el texto "mar". Se hace click en el botón para realizar la búsqueda.

Se comprueba que aparezcan sólo 4 usuarios, con los siguientes nombres:

- "Juan Pérez Martínez"
- "Marta Roces Lara"
- "María Torres Viesca"
- "Álvaro Alonso Pérez" (su email es "user06@**mar**te.com")

(El resto de usuarios existentes no contienen la palabra "mar" ni en su nombre ni en su email)

Finalmente, se deslogea (como se indica en 2.1).

## 4.2 [BusUsrInVal] Intento de acceso con URL a la búsqueda de usuarios desde un usuario no identificado.

**(Debe producirse un acceso no permitido a vistas privadas)**

Se intenta ir directamente a la URL "/user/list?searchText=mar" desde un usuario no logeado.

Se comprueba que se envía al usuario a la página de login (aparece el texto "Identifícate como usuario") y se muestra el mensaje de error "Debes identificarte primero para acceder a esa página".

## 5.1 [InvVal] Enviar una invitación de amistad a un usuario de forma valida.

Nos logeamos con el usuario user01@gmail.com. Nos redirige automáticamente a la página del listado de usuarios, en la que buscamos el botón para enviar una petición de amistad al usuario user02@gmail.com, y clickamos en él. Comprobamos que aparece el mensaje "Invitación enviada correctamente". Nos deslogeamos (como se indica en 2.1).

Nos logeamos con el usuario user02@gmail.com y pinchamos en la opción del menú que nos lleva al listado de invitaciones, donde comprobamos que aparezca el nombre de user01@gmail.com, que es "Juan Pérez Martínez", lo cual quiere decir que ha llegado correctamente su invitación. Nos deslogeamos (como se indica en 2.1).

## 5.2 [InvInVal] Enviar una invitación de amistad a un usuario al que ya le habíamos invitado la invitación previamente.

**(No debería dejarnos enviar la invitación)**

Nos logeamos con el usuario user01@gmail.com. Nos redirige automáticamente a la página del listado de usuarios, en la que buscamos el botón para intentar volver a enviar una petición de amistad al usuario user02@gmail.com, y clickamos en él. Comprobamos que aparece el mensaje "Error al enviar la invitación. ¡Ya has enviado una invitación de amistad a ese usuario!".

Nos deslogeamos (como se indica en 2.1).

## 6.1 [LisInvVal] Listar las invitaciones recibidas por un usuario, realizar la comprobación con una lista que al menos tenga una invitación recibida.

Nos logeamos con el usuario user02@gmail.com y pinchamos en la opción del menú que nos lleva al listado de invitaciones. Comprobamos que aparece el texto "Solicitudes de amistad", que sólo haya una invitación, y que aparezca el nombre del usuario user01@gmail.com, que es "Juan Pérez Martínez", el cual le envió una invitación a user02@gmail.com previamente en un test anterior.

Nos deslogeamos (como se indica en 2.1).

## 7.1 [AcepInvVal] Aceptar una invitación recibida.

Nos logeamos con el usuario user02@gmail.com y pinchamos en la opción del menú que nos lleva al listado de invitaciones. Comprobamos que aparece el texto "Solicitudes de amistad" y clickamos en el botón de "Aceptar" de la invitación cuyo nombre de usuario es "Juan Pérez Martínez" (user01@gmail.com).

Comprobamos que nos lleva al listado de amigos (aparece el texto "Tus Amigos"), que se muestra el mensaje "Usuario agregado como amigo correctamente", y que aparece el nombre de "Juan Pérez Martínez", con lo que se ha añadido como amigo.

Nos deslogeamos (como se indica en 2.1).

## 8.1 [ListAmiVal] Listar los amigos de un usuario, realizar la comprobación con una lista que al menos tenga un amigo.

Nos logeamos con el usuario user02@gmail.com y pinchamos en la opción del menú que nos lleva al listado de amigos. Comprobamos que aparece el texto "Tus Amigos", que sólo tiene un amigo, y que su nombre es "Juan Pérez Martínez" y su email es "user01@gmail.com".

Nos deslogeamos (como se indica en 2.1).

# Descripción de las Pruebas Unitarias (SW Rest)

Antes de cada ejecución de los tests, se realiza lo indicado en el punto anterior y se crea relaciones de amistad entre user10 con: user11, user12 y user13. También se añaden 5 mensajes en la conversación entre user10 y user11.

## C1.1[CInVal] Inicio de sesión con datos válidos.

Va al widget de login, introduce datos válidos de un usuario existente en la aplicación (user10@gmail.com), y comprueba que inicia sesión correctamente, es decir, que aparecen los siguientes textos: "Listado de tus amigos" y "Usuario autenticado: <email>", donde <email> es el email del usuario que se acaba de autenticar.

## C1.2 [CInInVal] Inicio de sesión con datos inválidos (usuario no existente en la aplicación).

Va al widget de login, introduce datos de un usuario no existente en la aplicación (notExists@gmail.com), y comprueba que aparecen los siguientes textos: "Identifícate para acceder a los chats" y "Usuario no encontrado".

## C2.1 [CListAmiVal] Acceder a la lista de amigos de un usuario.

**(Que al menos tenga tres amigos)**

Va al widget de login y se identifica como user10@gmail.com.

Se comprueba que tiene 3 amigos (los que se añadieron al principio de los tests), con los siguientes nombres:

- "Diego Armando"
- "Peter Scholes"
- "Red Parker"

## C.2.2 [CListAmiFil] Acceder a la lista de amigos de un usuario, y realizar un filtrado para encontrar a un amigo concreto

**(El nombre a buscar debe coincidir con el de un amigo)**

Realizamos lo indicado en el punto anterior, e introducimos el texto "Red Parker" en el campo para filtrar.

Se comprueba que sólo aparece un amigo, y que su nombre es "Red Parker" y su email es "user13@gmail.com".

## C3.1 [CListMenVal] Acceder a la lista de mensajes de un amigo "chat".

**(La lista debe contener al menos tres mensajes)**

Va al widget de login y se identifica como user10@gmail.com.

Clickamos en el nombre de "Diego Armando" y comprobamos que aparecen los siguientes textos: "Chat con el usuario: Diego Armando" y "Usuario autenticado: <email>", donde <email> es el email del usuario autenticado. También se comprueba que la conversación tenga 5 mensajes (los que se añadieron al principio de los tests).

## C4.1 [CCrearMenVal] Acceder a la lista de mensajes de un amigo “chat” y crear un nuevo mensaje.

**(Validar que el mensaje aparece en la lista de mensajes)**

Va al widget de login y se identifica como user10@gmail.com.

Accedemos al chat con "Diego Armando" y comprobamos que la conversación tenga 5 mensajes.

Introducimos el texto "Bien, nos vemos. Chao!" en el campo para enviar un mensaje y hacemos click en el botón de enviar. Comprobamos que aparece el texto del mensaje creado y que ahora la conversación tiene 6 mensajes.

## C5.1 [CMenLeidoVal] Marcar mensajes como leídos de forma automática

**(Identificarse en la aplicación y enviar un mensaje a un amigo, validar que el mensaje enviado aparece en el chat. Identificarse después con el usuario que recibido el mensaje y validar que tiene un mensaje sin leer, entrar en el chat y comprobar que el mensaje pasa a tener el estado leído)**

Va al widget de login y se identifica como user11@gmail.com. Accedemos al chat con user10@gmail.com, le mandamos el mensaje "¡Hasta luego!", y comprobamos que aparece y no está leído.

Nos identificamos como user10@gmail.com. Comprobamos que tiene un mensaje sin leer de user11@gmail.com y accedemos a su chat. Comprobamos que aparece el mensaje "¡Hasta luego!" y que se muestra como leído.

## C6.1 [CListaMenNoLeidoVal] Mostrar el número de mensajes sin leer

**(Identificarse en la aplicación y enviar tres mensajes a un amigo, validar que los mensajes enviados aparecen en el chat. Identificarse después con el usuario que recibido el mensaje y validar que el número de mensajes sin leer aparece en la propia lista de amigos)**

Va al widget de login y se identifica como user11@gmail.com. Accedemos al chat con user10@gmail.com, le mandamos 3 mensajes ("Ah!", "Recuerda llevar las entradas" y "Están en la cocina"), y comprobamos que aparecen.

Nos identificamos como user10@gmail.com y comprobamos que tiene 3 mensajes sin leer de user11@gmail.com.

## C7.1 [COrdenMenVall] Ordenar la lista de amigos por último mensaje

**(Identificarse con un usuario A que al menos tenga 3 amigos, ir al chat del ultimo amigo de la lista y enviarle un mensaje, volver a la lista de amigos y comprobar que el usuario al que se le ha enviado el mensaje esta en primera posición. Identificarse con el usuario B y enviarle un mensaje al usuario A. Volver a identificarse con el usuario A y ver que el usuario que acaba de mandarle el mensaje es el primero en su lisa de amigos.)**

Va al widget de login y se identifica como user10@gmail.com. Accedemos al chat con el último usuario de la tabla de amigos y obtenemos su nombre, le mandamos el mensaje "Buenos dias", y comprobamos que aparece.

Volvemos a la lista de amigos y comprobamos que el primer amigo de la tabla tenga el nombre que obtuvimos en el paso anterior.

Nos identificamos como user12@gmail.com, accedemos al chat con user10@gmail.com, le mandamos el mensaje "Me gusta el ketchup", y comprobamos que aparece.

Nos identificamos de nuevo como user10@gmail.com y comprobamos que el primer amigo de la tabla sea user12@gmail.com.