Sprint 2: Gestionar el Estado y OAuth

Procesos de Ingeniería del Software (curso 2025-2026)

Resumen del sprint

El objetivo del sprint es resolver el problema de mantener el estado de una aplicación SaaS que, por su propia naturaleza, no comparte un mismo espacio de direcciones y, por lo tanto, no resulta sencillo mantener el estado.

Otro objetivo del sprint es introducir el inicio de sesión de terceros mediante OAuth. Se trata de mejorar el acceso de los usuarios a nuestra aplicación de modo que, en vez de solicitar un nick sin ni siquiera pedir una contraseña, ahora exigiremos iniciar sesión con Google.

El siguiente paso para completar nuestra arquitectura es introducir la capa de acceso a datos.

Así pues, los bloques de este sprint son los siguientes
2.1 Gestionar el estado de la aplicación
2.2 Implementar autorización de terceros con Google
2.3 Implementar la capa de acceso a datos
2.4 Implementar Google One Tap
2.5 Implementar registro de usuarios locales
2.6 Implementar inicio de sesión de usuarios locales

El sprint termina con unas tareas adicionales como el cifrado de la clave del usuario (2.7), el aseguramiento de las rutas del API Rest (2.8) e implementar Salir (2.9).

2.1. Gestionar el Estado de la Aplicación

2.1.1. Introducción

Las aplicaciones Web son, por su propia naturaleza, distribuidas. La mayoría de ellas se estructuran siguiendo la arquitectura cliente-servidor. Esta arquitectura dificulta la tarea de mantener el estado de la aplicación, de ahí que se suele decir que las aplicaciones Web son sin estado (stateless). En una aplicación de escritorio, mantener el estado no es un reto ya que todos los componentes se ejecutan o al menos se controlan desde un mismo dispositivo. En una aplicación cliente-servidor (y por extensión el resto de arquitecturas distribuidas), mantener el estado requiere un esfuerzo adicional.

Mantener el estado no es un capricho, es el modo de proporcionar al usuario y clientes una experiencia armónica, unificada de nuestra aplicación.

Existen diferentes mecanismos para mantener el estado, algunos mecanismos se pueden emplear con diferentes tecnologías y otros mecanismos son exclusivos.

De forma general se denomina cookie al mecanismo que permite mantener el estado en una aplicación distribuida, aunque propiamente es uno de los mecanismos posibles, como veremos a continuación.

Una cookie es una porción pequeña de información que se almacena en el navegador y que se puede gestionar desde el servidor.

Las cookies se pueden emplear para diferentes propósitos:
- Gestionar el estado: inicio de sesión, carrito de la compra, etc.
- Almacenar preferencias del usuario
- Seguimiento (tracking) y análisis del comportamiento del usuario

Las cookies pueden definirse en el navegador cliente mediante JavaScript o en respuesta a una petición en el servidor. En nuestro proyecto vamos a utilizar los dos modos de definir las cookies. Comenzaremos definiendo cookies en el navegador.

Desde la aparición de HTML5, tenemos 3 posibilidades de almacenar información de la aplicación en el navegador:
- Cookies: es el primer mecanismo que apareció. No pueden almacenar mucha información (4KB), se definen a nivel de navegador, pueden caducar, almacena la información como cadena. Las cookies viajan en la petición.
- Local Storage: gran capacidad (10MB), se definen a nivel navegador, no caducan, almacenan cadenas, no viajan en la petición.
- Session Storage: capacidad media (5MB), a nivel de pestaña, desaparecen cuando se cierra la pestaña, almacena cadenas y no viajan en la petición.

Los tres mecanismos utilizan un array asociativo (clave-valor) para almacenar la información.

Puedes comprobar los diferentes compartimentos en las DevTools del navegador (en la pestaña Application, sección Storage).

2.1.2. Implementar localStorage en nuestra solución

La idea de usar cookies es que el usuario que ya ha iniciado sesión, no tenga que volver a hacerlo en caso de que refresque la página.

Vamos a implementar las cookies utilizando localStorage.

Para ello, nuestra aplicación cliente lo primero que tiene que hacer, antes de mostrar el formulario de inicio, es comprobar la cookie. Si existe, le muestra

directamente el mensaje de bienvenida, y si no existe, le muestra el formulario de inicio.

El primer paso es almacenar la cookie una vez el usuario haya conseguido registrarse. Observa la función de callback de agregarUsuario en el objeto ClienteRest (véase Código 1).

```
this.agregarUsuario=function(nick){
    var cli=this;
    $.getJSON("/agregarUsuario/"+nick,function(data){
        let msg="El nick "+nick+" está ocupado";
        if (data.nick!=-1){
            console.log("Usuario "+nick+" ha sido registrado");
            msg="Bienvenido al sistema, "+nick;
            localStorage.setItem("nick",nick);
        }
        else{
            console.log("El nick ya está ocupado");
        }
        cw.mostrarMensaje(msg);
    });
}
```

Código 1. Método “agregarUsuario()” modificado para almacenar la cookie “nick”. Observa la nueva línea “localStorage…”.

El siguiente paso es crear un método que compruebe la cookie. Este método lo definimos en ControlWeb (véase Código 2).

```
this.comprobarSesion=function(){
    let nick=localStorage.getItem("nick");
    if (nick){
        cw.mostrarMensaje("Bienvenido al sistema, "+nick);
    }
    else{
        cw.mostrarAgregarUsuario();
    }
}
```

Código 2. Nuevo método comprobarSesion() de ControlWeb.

Ahora tenemos que modificar el “index.html” para cambiar el primer método al que llamamos. En vez de llamar directamente a “agregarUsuario()”, llamaremos a “comprobarSesion()” en primer lugar (véase Código 3).

```
<script>
    cw=new ControlWeb();
    rest=new ClienteRest(cw);
    //cw.mostrarAgregarUsuario();
    cw.comprobarSesion();
</script>
```

Código 3. Cambio en el archivo “index.html” para llamar a “comprobarSesion()” en vez de “mostrarAgregarUsuario”.

Comprueba el resultado ejecutando la aplicación.

Comprueba los cambios operados en DevTools (Application, sección Storage). En la Figura 2.1 puedes ver un ejemplo de lo que debería mostrarse.

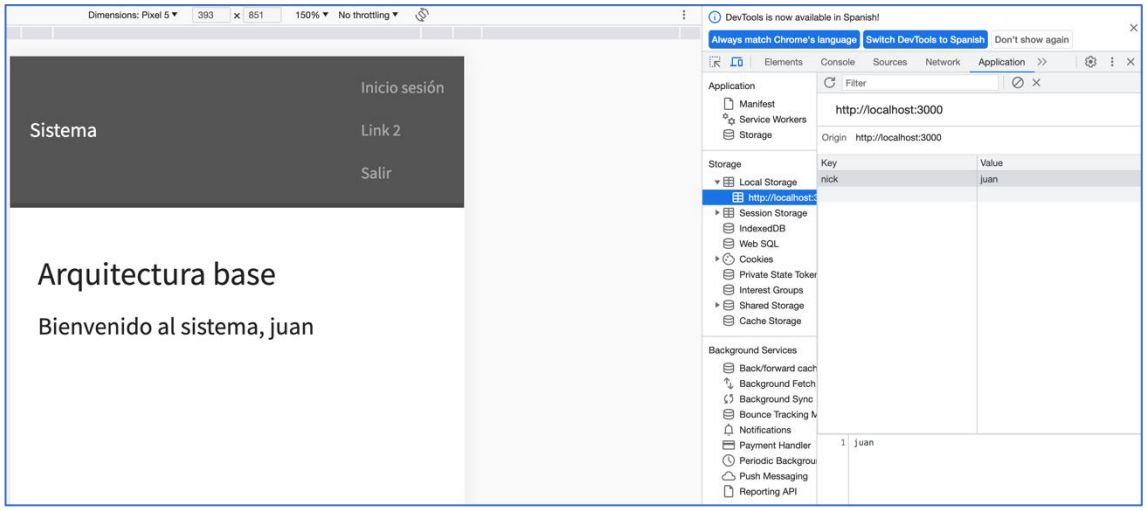

Figura 2.1. Comprobar localStorage de nuestra aplicación.

Como se puede ver, con esta nueva versión, el usuario que inició sesión mantiene su sesión abierta. El problema que nos toca resolver ahora es que tenemos que posibilitar que cierre la sesión. Esto se consigue borrando la cookie. Esto lo incluimos en un nuevo método de ControlWeb que llamamos “salir()” (véase Código 4).

```
this.salir=function(){
    localStorage.removeItem("nick");
    location.reload();
}
```
Código 4. Método “salir()” de ControlWeb.

Puedes modificar el método “salir()” para que muestre algún mensaje de despedida al usuario que cierra la sesión.

Comprueba el resultado.

2.1.3 Implementar cookies

En esta sección, vamos a modificar nuestra solución para que utilice cookies en vez de localStorage.

Los cambios que vamos a introducir son mínimos ya que vamos a aplicar cookies a resolver el mismo problema que en la sección anterior. El objetivo es que el usuario que inició sesión, no tenga que volver a hacerlo en caso de que refresque la página, cierre la pestaña o incluso cierre el navegador.

Lo primero que hacemos es incluir una librería que nos permite usar JQuery para gestionar las cookies.

Como hemos hecho en otras ocasiones, vamos a localizar la url que nos permita cargar la librería desde Internet. Para ello, ponemos en el buscador:
cdnjs jquery-cookie

El primer resultado debería ser la página de cdnjs. Copiamos el enlace pulsando en </> y lo pegamos en el "index.html". Lo podemos ubicar justo antes de nuestros archivos (controlWeb.js, clienteRest.js).

Lo siguiente que hacemos es modificar las líneas donde aparece localStorage y las sustituimos por jquery-cookie, siguiendo esta equivalencia:

localStorage.setItem(clave,valor) → $.cookie(clave, valor)
localStorage.getItem(clave) → $.cookie(clave)
localStorage.removeItem(clave) → $.removeCookie(clave)

Realiza los cambios necesarios y comprueba el resultado. Comprueba también la zona Cookies de las DevTools (Application).

2.2. Implementar Autorización de Terceros (Google)

2.2.1. Introducción a OAuth y PassportJS

En qué consiste OAuth (véase https://auth0.com/es/intro-to-iam/what-is-oauth-2).

En qué consiste PassportJS (véase https://www.passportjs.org/)

2.2.2. Implementar OAuth con PassportJS

La implementación de OAuth comenzará por el cliente web.

Nuestra primera tarea será localizar las imágenes a utilizar para implementar el acceso con Google. Debemos seguir el estilo marcado por Google ya que es un modo de no confundir al usuario. Para localizar las imágenes que conviene utilizar deberás escribir en el buscador algo como esto: Plataforma de identidad de Google para implementar el "acceso con Google".

Una vez en la página de Plataforma de identidad de Google, localiza y descarga un archivo zip que contiene las imágenes en diversos formatos y tamaños.

Crear la carpeta "img" dentro de la carpeta "cliente" de nuestra solución. Ubica en esa carpeta la imagen que hayas elegido de entre las diferentes opciones.

El siguiente paso consiste en incluir la imagen en el formulario que tenemos en ControlWeb para mostrar agregar usuario (véase Código 5).

```
this.mostrarAgregarUsuario=function(){
	$('#bnv').remove();
	$('#mAU').remove();
	let cadena='<div id="mAU">';
	cadena = cadena + '<div class="card"><div class="card-body">';
	cadena = cadena +'<div class="form-group">';
	cadena = cadena + '<label for="nick">Nick:</label>';
	cadena = cadena + '<p><input type="text" class="form-control" id="nick" placeholder="introduce un
nick"></p>';
	cadena = cadena + '<button id="btnAU" type="submit" class="btn btn-primary">Submit</button>';
	cadena=cadena+'<div><a href="/auth/google"><img
src="./cliente/img/btn_google_signin_light_focus_web@2x.png" style="height:40px;"></a></div>';
	cadena = cadena + '</div>';
	cadena = cadena + '</div></div></div>';
.....
```

Código 5. Parte del método “mostrarAgregarUsuario”, en el que hemos incluido una nueva línea con un href y una imagen.

Observa el atributo “href” de la etiqueta <a href…/>. En este caso, cuando el usuario pulse la imagen, el navegador buscará la ruta “/auth/google” en el servidor.

A partir de este momento, tenemos que seguir en el servidor, en concreto en nuestra capa Rest que está en el archivo index.js.

Para poder implementar OAuth en NodeJS vamos a utilizar un middleware que se llama PassportJS (http://www.passportjs.org). Con esta librería podemos implementar accesos OAuth de diferentes proveedores.

En el archivo index.js, cargamos la librería de PasspporJS (la ubicamos justo antes de cargar nuestro archivo “modelo.js”):

```
	const passport=require("passport");
```

PassportJS utiliza cookies de servidor para mantener la sesión. Necesitamos cargar la librería “cookie-session”. Lo ubicamos justo después del que acabamos de incluir:

```
const cookieSession=require("cookie-session");
```

El siguiente paso es incluir un archivo donde vamos a centralizar las diferentes estrategias que podemos emplear con PassportJS.

```
require("./servidor/passport-setup.js");
```

El archivo “passport-setup.js” lo completaremos más tarde. Por ahora vamos a seguir definiendo la inicialización del middleware PassportJS.

La inicialización del middleware se realiza con la siguiente directiva (lo ubicamos justo después del app.use que define el directorio raíz):

```
app.use(cookieSession({
  name: 'Sistema',
  keys: ['key1', 'key2']
}));
```

La inicialización continua con estas dos directivas (que ubicamos a continuación del anterior “app.use”):

```
app.use(passport.initialize());
app.use(passport.session());
```

Ahora vamos a incluir el manejador de la ruta que ya teníamos definida en el cliente (“/auth/google”):

```
app.get("/auth/google",passport.authenticate('google', { scope: ['profile','email'] }));
```

Si seguimos la secuencia de la petición (controlWeb.js→ clienteRest.js→index.js), en este momento entra en juego el archivo que habíamos dejado pendiente (passport-setup.js).

En la carpeta de “servidor”, crea un nuevo archivo de nombre “passport-setup.js”. En ese archivo ponemos el contenido del Código 6.

```
const passport=require("passport");
const GoogleStrategy = require('passport-google-oauth20').Strategy;

passport.serializeUser(function(user, done) {
  done(null, user);
});

passport.deserializeUser(function(user, done) {
    done(null, user);
});

passport.use(new GoogleStrategy({
    clientID: "xxxxxxxxxxx",
    clientSecret: "xxxxxxxxxxx",
    callbackURL: "http://localhost:3000/google/callback"
  },
  function(accessToken, refreshToken, profile, done) {
      return done(null, profile);
  }
));
```

Código 6. Contenido del archivo “passport-setup.js”

Como podemos ver en el Código 6, tenemos dos “require”, uno de PassportJS, que ya habíamos incluido en “index.js”, y la librería que usamos para implementar la estrategia Google.

Para poder utilizar el acceso Google, necesitamos obtener dos valores (clientID y clientSecret) en la zona de credenciales de Google Cloud Platform.

Abre la consola de GCP: console.cloud.google.com

En el menú principal (tres líneas arriba a la izquierda), elige la opción “API y servicios”, y luego elige “credenciales” (véase Figura 2.2).

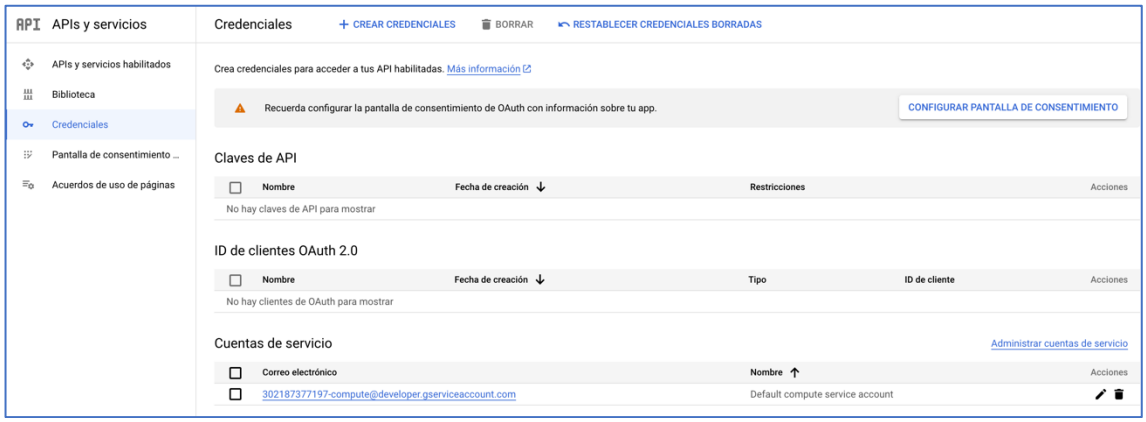

Figura 2.2. Panel de GCP para crear las credenciales.

Para crear las credenciales, pulsamos “crear credenciales” y elegimos “ID de cliente de OAuth”.

El siguiente paso es “Configurar pantalla de consentimiento”. Nos aparecen dos opciones, elegimos “externo”. Introducimos los campos obligatorios (nombre app, email en dos lugares). En el resto de formularios, podemos elegir las opciones por defecto. En un momento dado elegimos “Publicar” la aplicación.

Una vez configurada la pantalla de consentimiento, volvemos al panel de credenciales (Figura 2.2) y volvemos a pulsar “crear credenciales” y elegimos “ID de cliente de OAuth”.

En Tipo de Aplicación elegimos “Aplicación web”. En “sistema-local” ponemos un identificador de credenciales para probarlo en local (localhost).

En “Orígenes autorizados de JavaScript”, definimos una URI: “http://localhost:3000”.

En “URI de redireccionamiento autorizados”, definimos la ruta de callback: http://localhost:3000/google/callback.

Si todo ha ido bien, nos aparecerá un diálogo con las dos claves que necesitamos (ID de cliente, Secreto del cliente). Esas dos cadenas las tenemos que incluir en el archivo “passport-setup.js”.

Ya tenemos definida la secuencia que va desde nuestra app hasta Google. Por ahora quedaría así:

controlWeb.js→ clienteRest.js→index.js→passport-setup.js→Google

Una vez Google termina de autenticar al usuario, devolvería el control a nuestra app en la ruta “/google/callback” en el archivo “index.js” (véase Código 7).

```
app.get('/google/callback',
  passport.authenticate('google', { failureRedirect: '/fallo' }),
  function(req, res) {
    res.redirect('/good');
});
```

Código 7. Ruta de callback con la respuesta de Google.

Si observamos el “app.get” de “/google/callback”, el middleware controla tanto la posibilidad de que el usuario no haya podido autenticarse y para eso utiliza la ruta “/fallo”, y también la opción de éxito a través de la ruta “/good”.

Código 8 muestra el código que implementa la ruta “/good”, y Código 9 muestra la implementación de la ruta “/fallo”.

```
app.get("/good", function(request,response){
  let nick=request.user.emails[0].value;
  if (nick){
    sistema.agregarUsuario(nick);
  }
  //console.log(request.user.emails[0].value);
  response.cookie('nick',nick);
  response.redirect('/');
});
```

Código 8. El usuario ha podido iniciar sesión con Google.

```
app.get("/fallo",function(request,response){
  response.send({nick:"nook"})
});
```

Código 9. El usuario no ha podido iniciar sesión.

Para poder probar el funcionamiento, tenemos que instalar las dependencias que necesita nuestra solución:

npm install passport
npm install cookie-session
npm install passport-google-oauth20

En el package.json instalaremos la version 0.5.3 de Passport.

Una vez instaladas las dependencias, ya podemos probar la autorización Google.

NOTA 1: conviene crear otro par de credenciales (clientID y clientSecret) para la versión desplegada (o también llamada producción).

NOTA 2: no conviene publicar en repos públicos ninguna clave privada de acceso.

2.3. Implementar la capa de acceso a datos (CAD)

2.3.1 Configurar la BBDD en Mongo Atlas

Crear credenciales en Mongo Atlas

Registrarse e iniciar sesión en Mongo Atlas.

Crear un nuevo proyecto

Elegimos un nombre.

Crear un deployment

Elegimos M0 Free

El resto de opciones dejamos las que hay por defecto.

Pulsamos crear.

Security Quickstart

Elegimos Username and Password

Escribimos un Username y una Password (debemos copiarlas porque las necesitaremos luego)

Pulsamos Create User

En las opciones “Where would you like to connect from?“ elegimos Cloud Environment.

Agregamos una nueva entrada 0.0.0.0 con etiqueta todos

Eliminamos la entrada de nuestra IP local

Pulsamos Finish & Close

En la pantalla Overview, pulsamos el botón CONNECT y nos aparece el diálogo que se muestra en la Figura 2.3.

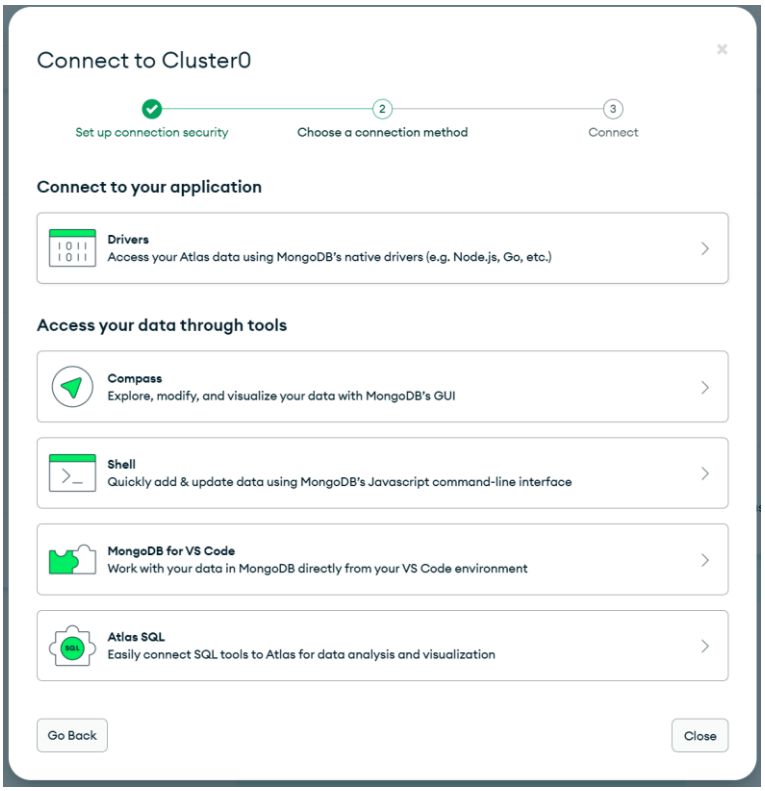

Figura 2.3. Diálogo para elegir el tipo de conexión a la BBDD.

Elegimos la opción Drivers (Connect to your application).

Sigue las instrucciones que se muestran:

- Elegimos el driver de Node.js
- Instalamos el driver con npm install mongodb
- Copiamos la cadena de conexión (connection string)

Nota: la versión de mongodb que vamos a usar es la 4.4.0 (modifica la versión de mongodb en el archivo package.json y ejecutar npm install).

2.3.2 Conectar con la base de datos

Toda la comunicación con la base de datos, en nuestro caso Mongo Atlas, la vamos a centralizar en un componente que almacenaremos el archivo “cad.js”.

Crea un nuevo archivo llamado “cad.js” en la carpeta “servidor”.

En ese archivo creamos la estructura general de nuestro objeto CAD (véase Código 10).

```
function CAD(){


}

module.exports.CAD=CAD;
```
Código 10. Estructura del objeto CAD, el componente de acceso a datos.

Este componente utilizará el driver de “mongodb” para realizar las operaciones sobre la base de datos Mongo.

Para poder utilizar el driver, debemos incluir las dependencias en el archivo “cad.js”:

```
const mongo=require("mongodb").MongoClient;
const ObjectId=require("mongodb").ObjectId;
```

A continuación incluimos la colección:

```
    this.usuarios;
```

El método que nos permitirá conectarnos:

```
    this.conectar=async function(callback){
        let cad=this;
        let client= new
mongo("mongodb+srv://xxxx:xxxx@xxxx.xxxxx.mongodb.net/?retryWrites=true&w
=majority");
        await client.connect();
        const database=client.db("sistema");
        cad.usuarios=database.collection("usuarios");
        callback(database);
    }
```

Para poder conectar con la base de datos, necesitamos la cadena de conexión que obtenemos del panel de control de Mongo Atlas. Cada uno debe incluir el usuario y la password (las cadenas xxx antes del símbolo @ y separados por :) y la url que nos proporciona Mongo Atlas.

Obsérvese que en el método “conectar(callback)” utilizamos el mecanismo async-await en vez de callbacks, para gestionar las llamadas asíncronas.

Nótese que nuestra intención es usar una base de datos denominada “sistema” y como primera colección vamos a definir “usuarios. La base de datos y la colección se creará en el momento en que hagamos la primera inserción de un registro.

Ahora vamos a conectar la capa lógica (el objeto “Sistema()” que está definido en el archivo “modelo.js”) con la capa de acceso a datos. Esto lo hacemos incluyendo el siguiente “require” en el archivo “modelo.js”:

```
const datos=require("./cad.js");
```

La dependencia entre capas se implementa como un atributo en el objeto "Sistema()":

```
this.cad=new datos.CAD();
```

Además, debemos lanzar la conexión a la base de datos en el mismo momento en que instanciemos el objeto "Sistema()":

```
    this.cad.conectar(function(db){
        console.log("Conectado a Mongo Atlas");
    });
```

Comprueba que la conexión funciona correctamente. Si todo funciona correctamente, obtendremos el mensaje "Conectado a Mongo Atlas".

2.3.3 Realizar inserciones a la BBDD

Nuestra primera operación en nuestra base de datos será insertar el usuario autenticado de Google.

MongoDB nos proporciona diferentes métodos para realizar inserciones de documentos. El primero que vamos a utilizar tiene la particularidad de que nos permite insertar el documento SI y SOLO SI no existe.

Observa el código mostrado en Código 11, que comentaremos posteriormente.

```
this.buscarOCrearUsuario=function(usr,callback){
        buscarOCrear(this.usuarios,usr,callback);
    }

    function buscarOCrear(coleccion,criterio,callback)
    {
        coleccion.findOneAndUpdate(criterio, {$set: criterio}, {upsert:
true,returnDocument:"after",projection:{email:1}}, function(err,doc) {
           if (err) { throw err; }
           else {
                console.log("Elemento actualizado");
                console.log(doc.value.email);
                callback({email:doc.value.email});
            }
         });
    }
```

Código 11. Operación pública "buscarOCrearUsuario", para buscar o crear un usuario.

El método “buscarOCrearUsuario(email,callback)” es un método que nos permite insertar un nuevo usuario {“email”:email} siempre y cuando no haya sido insertado anteriormente.

Este método utiliza una función “privada” que se llama “buscarOCrear”. Esta función la utilizaremos con otras colecciones que necesitemos gestionar.

Inserta el código de Código 11 en el objeto “CAD()”.

El siguiente paso es crear un nuevo método en “Sistema()” (archivo “modelo.js”) que nos permita realizar la inserción en la BBDD, que se llamará “buscarOCrearUsuario(email, callback)”:

```
this.usuarioGoogle=function(usr,callback){
        this.cad.buscarOCrearUsuario(usr,function(obj){
            callback(obj);
        });
    }
```

Finalmente, modificamos la capa Rest, para poder realizar la inserción, una vez el usuario haya sido autenticado por Google.

Para ello, tenemos que modificar la ruta “/good” del archivo “index.js”:

```
app.get("/good", function(request,response){
  let email=request.user.emails[0].value;
  sistema.usuarioGoogle({“email”:email},function(obj){
    response.cookie('nick',obj.email);
    response.redirect('/');
  });
});
```

Nótese que el usuario que ha utilizado inicio con Google, no está incluido en la lista de usuarios “vivos” de Sistema. Habría que revisar los dos tipos de inicios para que el resultado sea similar.

2.4. Implementar Google One Tap

El inicio con One Tap de Google permite a los usuarios iniciar sesión en una aplicación con un solo click, siempre y cuando hayan iniciado sesión en Google (véase Figura 2.4).

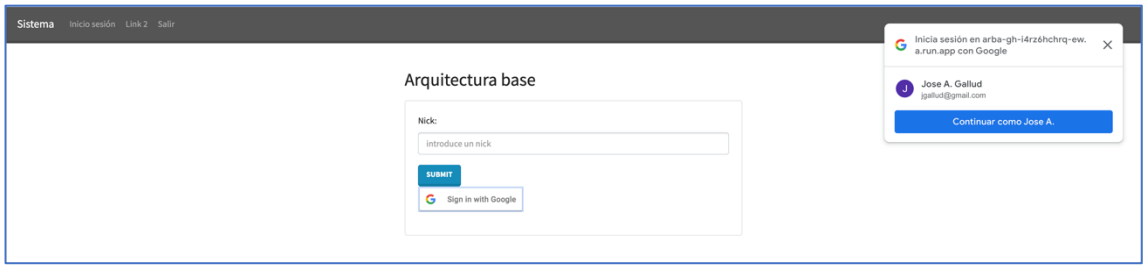

Figura 2.4. Diálogo One Tap.

Antes de presentar la implementación de One Ta que vamos a utilizar en nuestra solución, puede resultar conveniente consultar la documentación de Google al respecto, donde se explican las diferentes opciones.

En esta solución vamos a implementar One Tap desde nuestro cliente Web.

Para poder hacer peticiones al API de autenticación de Google, necesitaremos incluir la correspondiente librería (en el archivo “index.html”):

```
<script src="https://accounts.google.com/gsi/client" async defer></script>
```

Además, tenemos que incluir en el archivo “index.html” una etiqueta necesaria para que nuestro cliente se conecte a Google y lance el diálogo One Tap:

```
<div
  id="g_id_onload"
  data-client_id="YOUR_GOOGLE_CLIENT_ID"
  data-login_uri=https://url/oneTap/callback
  data-skip_prompt_cookie="nick"
></div>
```

Esta vez necesitaremos solamente un ID de cliente. En la consola de GCP, en Credenciales, puedes crear una clave (o dos, si quieres usar una para local y otra para producción). También funciona utilizando las mismas credenciales que utilizamos para implementar la estrategia Google con PassportJS.

Para realizar las pruebas en local, tenemos que agregar dos URI de origen: http://localhost:3000 y http://localhost. Para la URI de redireccionamiento pondremos http://localhost:3000/oneTap/callback (para las credenciales de producción, pondremos la URL de la app desplegada en lugar de http://localhost:3000).

Ahora pasamos al servidor.

Lo primero que tenemos que hacer es incluir una nueva referencia en el package.json. Para conseguirlo, abrimos un terminal y nos ubicamos en el directorio raíz de nuestra solución (justo donde se encuentra el archivo “package.json”) y ejecutamos el siguiente comando:

npm install passport-google-one-tap

Nos aparecerá un nueva entrada en el archivo “package.json”.

La nueva estrategia PassportJS la integramos en el archivo “passport-setup.js”. En primer lugar, debemos incluir la referencia a la nueva librería:

```
const GoogleOneTapStrategy = require("passport-google-one-
tap").GoogleOneTapStrategy;
```

También agregamos una nueva sección “app.use” para la nueva estrategia:

```
passport.use(
  new GoogleOneTapStrategy(
    {
    //client_id:"xxxxxxx.apps.googleusercontent.com", //local
      client_id:"xxxxxx.apps.googleusercontent.com", //prod-oneTap
      //clientSecret: "xxxx", //local
      clientSecret:"xxxxxxxxxxx, // prod-oneTap
      verifyCsrfToken: false, // whether to validate the csrf token or
not
    },
    function (profile, done) {
      return done(null, profile);
    }
  )
);
```

Código 12. Fragmento de código que gestiona la petición Google One Tap.

Si recordamos la etiqueta que hemos incluido en el “index.html”, veremos que uno de los valores indicados es la URI de redireccionamiento. Esta es una ruta nueva que debemos gestionar en nuestra capa REST, ya que es el lugar al que conectará Google una vez haya validado al usuario.

Así pues, debemos incluir esa nueva ruta “/oneTap/callback” en el archivo “index.html”:

```
app.post('/oneTap/callback',
  passport.authenticate('google-one-tap', { failureRedirect: '/fallo' }),
  function(req, res) {
    // Successful authentication, redirect home.
    res.redirect('/good');
});
```

Como se puede notar, en caso de fallo, reutilizamos la ruta que ya teníamos "/fallo". Lo mismo hacemos en caso de validación correcta, reutilizamos la ruta "/good".

Para poder recoger la información contenida en el BODY de la petición necesitamos importar una librería que nos facilite el acceso a esa información. Esa librería se llama "body-parser" y debemos instalar esa dependencia (npm install body-parser). Incluye el siguiente "require" en la zona correspondiente, al princpio, del archivo "index.js":

```
const bodyParser=require("body-parser");
```

Debemos indicar a Express que debe usar esa librería:

```
app.use(bodyParser.urlencoded({extended:true}));
app.use(bodyParser.json());
```

Con esa librería ya podemos recuperar la información contenida en las peticiones tipo POST y similares.

Comprueba el funcionamiento.

En el cliente debería aparecer el diálogo One Tap de Google (siempre y cuando tengamos una sesión de Google iniciada), en la esquina superior derecha. Una vez validada la cuenta, nuestro sistema debería iniciar sesión de forma correcta.

2.5. Implementar Registro de usuarios locales

Hasta el momento, nuestra aplicación permite la entrada a usuarios que se autentican con Google (facilitando dos modos de acceso), y proporciona una estructura fácilmente extensible que permite incluir otros accesos OAuth (GitHub, Facebook, X, etc).

Junto a estas opciones de acceso, también nos hemos preocupado por almacenar, en una BBDD documental como Mongo Atlas, la información (por ahora el email) de los usuarios que acceden a nuestro sistema.

Además de los usuarios OAuth, que son los que se autentican utilizando un proveedor de identidades, tenemos los usuarios locales. Los usuarios locales son aquellos que se registran directamente en nuestro sistema sin utilizar ningún proveedor de identificación (proveedor OAuth).

En este apartado se describe cómo realizar el registro e inicio de sesión de usuarios locales de modo que participen los componentes de nuestra arquitectura.

Para describir esta funcionalidad comenzaremos desde el cliente.

Crea un archivo nuevo en la carpeta “cliente” que se llame “registro.html”.

Localiza un formulario HTML que quieras utilizar como base. A modo de ejemplo, se puede revisar la sección BS4 Forms de w3schools y elegir alguno de los ejemplos que se presentan. En este tutorial se adapta el primero de ellos (véase Código 13).

```
<div id="fmRegistro">
    <form>
        <div class="form-group">
            <label for="apellidos">Apellidos:</label>
            <input type="text" class="form-control"
placeholder="Apellidos" id="apellidos">
        </div>
        <div class="form-group">
            <label for="nombre">Nombre:</label>
            <input type="nombre" class="form-control"
placeholder="Nombre" id="nombre">
        </div>
        <div class="form-group">
            <label for="email">Email address:</label>
            <input type="email" class="form-control"
placeholder="Introduce email" id="email">
        </div>
        <div class="form-group">
            <label for="pwd">Password:</label>
```

```
            <input type="password" class="form-control"
placeholder="Introduce password" id="pwd">
        </div>
        <button type="submit" id="btnRegistro" class="btn btn-
primary">Registrar</button>
    </form>
</div>
```

Código 13. Formulario de registro tomado como ejemplo.

El siguiente paso es mostrar el formulario en nuestra página. Para ello tenemos que crear un método en el objeto "ControlWeb()". En Código 14 se muestra este método.

```
    this.mostrarRegistro=function(){
        $("#fmRegistro").remove();
        $("#registro").load("./cliente/registro.html",function(){
            $("#btnRegistro").on("click",function(e){
                e.preventDefault();
                let email=$("#email").val();
                let pwd=$("#pwd").val();
                if (email && pwd){
                    //rest.registrarUsuario(nick);
                    console.log(email+" "+pwd);
                }
            });
        });

    }
```

Código 14. Método "mostrarRegistro()" de "ControlWeb()".

Como se puede ver en Código 14, necesitamos una etiqueta HTML en el "index.html" que sirva de enganche donde vamos a insertar el formulario. Observa que los "id" del enganche y del formulario son diferentes, y que nunca borramos las etiquetas que usamos como "enganche".

El resto del código del método se explica por sí solo.

Veamos esa nueva etiqueta HTML en "index.html" donde insertaremos el formulario nuevo:

```
<div class="container">
        <h3>Sistema</h3>
        <div id="au"></div>
        <div id="registro"></div>
        <div id="msg"></div>
      </div>
```

Lo siguiente que tenemos que hacer es cambiar la invocación de "mostrarAgregarUsuario()" por la de "mostrarRegistro()" en el método "comprobarSesion()" de "ControlWeb()".

Comprueba el resultado consultando la consola del navegador.

El siguiente paso es enviar la información del formulario al servidor. Para ello vamos a implementar una petición de tipo POST. Como sabemos, las peticiones REST las gestionamos a través del objeto "ClienteRest()".

Esta nueva petición será similar a las que ya tenemos definidas, como puede ser "enviarJwt(jwt)". En cuanto a la función de callback, tendremos que hacer algo parecido a lo que hace "agregarUsuario(nick)".

En Código 14 se muestra la implementación de la petición utilizando Jquery AJAX.

```
this.registrarUsuario=function(email,password){
        $.ajax({
            type:'POST',
            url:'/registrarUsuario',
            data: JSON.stringify({"email":email,"password":password}),
            success:function(data){
                if (data.nick!=-1){
                    console.log("Usuario "+data.nick+" ha sido
registrado");
                    $.cookie("nick",data.nick);
                    cw.limpiar();
                    cw.mostrarMensaje("Bienvenido al sistema,
"+data.nick);
                    //cw.mostrarLogin();
                }
                else{
                    console.log("El nick está ocupado");
                }
             },
             error:function(xhr, textStatus, errorThrown){
                console.log("Status: " + textStatus);
                console.log("Error: " + errorThrown);
             },
            contentType:'application/json'
        });
    }
```

Código 14. Petición AJAX para registrar usuario.

En el campo "data" creamos un objeto JSON con algunos de los campos del formulario, cuyos valores los recogemos en el método "mostrarRegistro()" de

ControlWeb(). El objeto JSON lo convertimos en cadena para poder enviarlo en el cuerpo de la petición (a diferencia de la QueryString que se usa en peticiones GET).

Si continuamos recorriendo la arquitectura de nuestra solución, el siguiente componente sería nuestro servidor REST, que está contenido en el archivo "index.js". En ese archivo definimos el endpoint de la petición (véase Código 15).

```
app.post("/registrarUsuario",function(request,response){
    sistema.registrarUsuario(request.body,function(res){
      response.send({"nick":res.email});
    });
});
```

Código 15. Manejador de la ruta "/registrarUsuario".

El componente REST delega en la capa de lógica, en concreto en el objeto "Sistema()".

```
this.registrarUsuario=function(obj,callback){
        let modelo=this;
        if (!obj.nick){
            obj.nick=obj.email;
        }
        this.cad.buscarUsuario(obj,function(usr){
            if (!usr){
                modelo.cad.insertarUsuario(obj,function(res){
                    callback(res);
                });
            }
            else
            {
                callback({"email":-1});
            }
        });
    }
```

Código 16. Método "registrarUsuario(obj)" en la capa lógica.

El método "registrarUsuario(obj)" de "Sistema()" se encarga de realizar las comprobaciones que veamos conveniente. En nuestro caso se asegura que definir un nuevo campo del usuario que se llame "nick" en caso de que no esté definido. Lo siguiente que hace es pedirle a la CAD que se encargue de hacer una inserción en caso de que el usuario no exista.

Como vemos, necesitamos dos nuevos métodos en la capa de acceso a datos: "buscarUsuario(usr)" e "insertarUsuario(usr)". Ambos métodos son métodos que consideramos y tratamos como públicos. Cada uno de ellos hará uso de su correspondiente método "privado".

Veamos el contenido de los dos métodos públicos (y vinculados a entidades, en esta ocasión están vinculados a la colección “usuarios”):

```
this.buscarUsuario=function(obj,callback){
        buscar(this.usuarios,obj,callback);
    }
```

```
this.insertarUsuario=function(usuario,callback){
        insertar(this.usuarios,usuario,callback);
    }
```

Y ahora vamos a ver el contenido de los métodos que consideramos privados.

```
function buscar(coleccion,criterio,callback){
        coleccion.find(criterio).toArray(function(error,usuarios){
            if (usuarios.length==0){
                callback(undefined);
            }
            else{
                callback(usuarios[0]);
            }
        });
    }
```

```
function insertar(coleccion,elemento,callback){
        coleccion.insertOne(elemento,function(err,result){
            if(err){
                console.log("error");
            }
            else{
                console.log("Nuevo elemento creado");
                callback(elemento);
            }
        });
    }
```

Comprueba el resultado. Se debe comprobar tanto que es posible registrar un nuevo usuario, como que no se puede registrar un usuario previamente registrado.

Tareas adicionales: realiza los cambios oportunos en tu aplicación para mostrar los mensajes adecuados a cada caso, el mensaje de registro realizado y también el mensaje en caso de error.

Ejercicio: Tomando como referencia la implementación del registro de usuarios que se acaba de describir, realiza la implementación del inicio de sesión. Se sugiere comenzar desde el cliente.

Los métodos a implementar son los siguientes:

- mostrarLogin() en el objeto ControlWeb()
- loginUsuario(usr) en el objeto ClienteRest()
- endpoint “/loginUsuario” en “index.js”
- loginUsuario(usr) en el objeto Sistema()

Comprueba el resultado. Es importante tener en cuenta que cuando un usuario se registre con nuestro formulario, en caso de que el registro sea correcto le mostramos el formulario de inicio de sesión.

Una vez realizado el ejercicio, puedes comparar tus métodos con los que hay disponibles en la Zona de Código.

2.6 Implementar Confirmación de cuenta

2.6.1 Elegir un proveedor de envío de correos

Para confirmar la cuenta de un usuario local, el proceso normal es que nuestra aplicación le envíe un correo a la dirección que el usuario ha introducido en le registro, incluyendo en el correo un enlace que le permita al usuario verificar la cuenta.

El envío de correos desde una aplicación era una tarea sencilla hace unos años pero, debido a esa sencillez, se convirtió en un problema de seguridad. El sistema de envio de correos fue utilizado por los hackers y piratas para entrar en los sistemas, infectarlos con virus, o apropiarse de información.

Este fenómeno motivó el aumento de las medidas de seguridad en las aplicaciones informáticas y en el uso de APIs, entre las que se encuentran las que facilitan el envío de correos.

La mejor opción, en cuanto a que es la más profesional, para permitir el envío de correos es utilizar un proveedor, como por ejemplo Sendgrid.

Los pasos a seguir en el caso de usar Sendgrid serían los siguientes:
- Crear cuenta de sendgrid
- Crear un API key para la aplicación
- Verificar email remitente y verificar el dominio

Sin embargo, para el desarrollo de nuestro prototipo que tiene como objetivo principal el aprendizaje y que tiene unos requisitos muy estrictos en cuanto al uso de herramientas gratuitas, vamos a utilizar una opción menos profesional que es el acceso a Gmail con contraseña de aplicación.

Los pasos a seguir para conseguir la contraseña de aplicación de Gmail son los siguientes:

- Con tu cuenta de Gmail iniciada, abre las opciones de Gestionar tu cuenta

- Selecciona Seguridad
- En la opción "Iniciar session con Google," selecciona Verificación en dos pasos
- Al final de esa página, selecciona Contraseñas de aplicación
- Introduce un nombre que te sirva para recordar la aplicación que va a usar esa clave
- Selecciona Generar y copia la cadena generada en alguna parte (la usaremos más adelante)

Crear el componente de envío de correos

En la carpeta “servidor” crea un nuevo archivo que se llame “email.js”.

Incluye una referencia a ese archivo en “modelo.js”. Las referencias a módulos externos hay que colocarlas al principio del archivo:

```
const correo=require("./email.js");
```

En el nuevo archivo “email.js” incluye el siguiente código:

```
const nodemailer = require('nodemailer');
const url="http://localhost:3000/";
//const url="tu-url-de-despliegue";

const transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
        user: 'tu-cuenta@gmail.com',
        pass: 'tu-clave-generada-en-Gmail'
    }
});

//send();

module.exports.enviarEmail=async function(direccion, key,men) {
    const result = await transporter.sendMail({
        from: 'tu-cuenta@gmail.com',
        to: direccion,
        subject: men,
        text: 'Pulsa aquí para confirmar cuenta',
        html: '<p>Bienvenido a Sistema</p><p><a
href="'+url+'confirmarUsuario/'+direccion+'/'+key+'">Pulsa aquí para
confirmar cuenta</a></p>'
    });
}
```

Observa que el código tiene una dependencia de un middleware que se llama "nodemailer". Como es conocido, las nuevas dependencias deben ser instaladas. En este caso, hay que abrir un terminal y ejecutar "npm install nodemailer".

Observa también que al crear la variable "transporter" se debe incluir la cuenta de Gmail que utilices para tu aplicación, así como la clave generada en el apartado anterior. Tu dirección de email también debe figurar en el campo del remitente (from) del método "enviarEmail".

2.6.3 Modificar el flujo de registro e inicio de sesión

El siguiente paso es modificar el registro e inicio de sesión para contemplar la nueva funcionalidad de confirmación de cuenta.

Al registrar una cuenta, incluimos "key" (Date.now().toString()) y "confirmada" a false. Además enviamos un correo al supuesto nuevo usuario.

El nuevo método "registrarUsuario" de "Sistema()" quedaría de la siguiente manera:

```
this.registrarUsuario=function(obj,callback){
        let modelo=this;
        if (!obj.nick){
            obj.nick=obj.email;
        }
        this.cad.buscarUsuario(obj,function(usr){
            if (!usr){
                //el usuario no existe, luego lo puedo registrar
                obj.key=Date.now().toString();
                obj.confirmada=false;
                modelo.cad.insertarUsuario(obj,function(res){
                    callback(res);
                });
                correo.enviarEmail(obj.email,obj.key,"Confirmar cuenta");
            }
            else
            {
                callback({"email":-1});
            }
        });
    }
```

El siguiente paso es incluir una nueva ruta que permita al usuario confirmar la cuenta en cuanto le llegue el correo electrónico.

Comenzamos en la capa REST (archivo "index.js"):

```
app.get("/confirmarUsuario/:email/:key",function(request,response){
  let email=request.params.email;
  let key=request.params.key;
  sistema.confirmarUsuario({"email":email,"key":key},function(usr){
    if (usr.email!=-1){
        response.cookie('nick',usr.email);
    }
    response.redirect('/');
  });
})
```

Como se puede ver, este método delega en “Sistema()”. Debemos implementar un nuevo método “confirmarUsuario” en ese objeto. Este nuevo método deberá buscar en la base de datos un registro que tenga el “email” y “key” que aparecían en el correo electrónico, y que además tenga el atributo “confirmada” a “false”:

```
 this.confirmarUsuario=function(obj,callback){
        let modelo=this;
        this.cad.buscarUsuario({"email":obj.email,"confirmada":false,"key
":obj.key},function(usr){
            if (usr){
                usr.confirmada=true;
                modelo.cad.actualizarUsuario(usr,function(res){
                    callback({"email":res.email}); //callback(res)
                })
            }
            else
            {
                callback({"email":-1});
            }
        })
    }
```

Este método utilizar un nuevo método de “CAD()” que se llama “actualizarUsuario”, que a su vez, utiliza un nuevo método llamado “actualizar”. Veamos ambos métodos (corresponden al objeto “CAD()” que está en el archivo “cad.j”):

```
this.actualizarUsuario=function(obj,callback){
        actualizar(this.usuarios,obj,callback);
    }
```

```
function actualizar(coleccion,obj,callback){
        coleccion.findOneAndUpdate({_id:ObjectId(obj._id)}, {$set: obj},
{upsert: false,returnDocument:"after",projection:{email:1}},
function(err,doc) {
            if (err) { throw err; }
```

```
            else {
                 console.log("Elemento actualizado");
                 callback({email:doc.value.email});
             }
          });
    }
```

A continuación, vamos a modificar el modo de iniciar sesión. Esta modificación es necesaria para impedir que inicie sesión un usuario registrado que no haya confirmado su cuenta mediante el correo electrónico.

Para implementar el cambio, lo único que tenemos que hacer es buscar un usuario con el "email" indicado y que además tenga el atributo "confirmada" a "true".

Veamos como quedaría el método "loginUsuario" de "Sistema()" (archivo "modelo.js"):

```
this.loginUsuario=function(obj,callback){
        this.cad.buscarUsuario({"email":obj.email,"confirmada":true},func
tion(usr){
            if(usr && usr.password==obj.password)
            {
                callback(usr);
            }
            else
            {
                callback({"email":-1});
            }
        });
    }
```

Por último, vamos a utilizar Passport como middleware para controlar también el inicio de sesión de cuentas locales.

Para ello, debemos hacer algunos cambios en la capa REST (archivo "index.js").

En primer lugar debemos incluir una nueva dependencia para la estrategia local de PassportJS (la incluimos justo antes del require de "passport-setup.js":

```
const LocalStrategy = require('passport-local').Strategy;
```

El siguiente paso es pedirle a PassportJS que utilice también la estrategia local. Esto lo hacemos con el siguiente código (lo incluimos justo después de inicializar la sesión de Passport):

```
passport.use(new
LocalStrategy({usernameField:"email",passwordField:"password"},
```

```
  function(email,password,done){
    sistema.loginUsuario({"email":email,"password":password},function(use
r){
      return done(null,user);
    })
  }
));
```

Con este código estamos incluyendo nuestro inicio de sesión "sistema.loginUsuario" en el proceso de verificación de PassportJS.

Esto exige que modifiquemos nuestra ruta "/loginUsuario" para incluir PassportJS en la gestión de ese inicio de sesión (además del de Google, y otros que se pudieran implementar).

```
app.post('/loginUsuario',passport.authenticate("local",{failureRedirect:"
/fallo",successRedirect: "/ok"})
);

app.get("/ok",function(request,response){
  response.send({nick:request.user.email})
});
```

No olvides comentar el "post" de "/loginUsuario" que teníamos implementado.

Observa que la ruta "/fallo" es la misma que usamos para el caso de autenticación Google.

Comprobar:

- Al registrar usuario, comprobar los campos key y confirmada
- Confirmar cuenta desde el correo recibido
- Intentamos iniciar sesión con una cuenta no confirmada: no permitir
- Iniciar sesión con cuenta confirmada

2.7 Cifrar la clave del usuario

Una tarea importante es cifrar las claves en la base de datos. No se deben dejar las claves con texto plano de modo que cualquiera las pueda ver.

Para cifrar las claves se suele recurrir a alguna librería que proporcione métodos de cifrado y comparación de claves cifradas.

El cifrado de las claves aparece en dos momentos del proceso registro e inicio de sesión:

- Cifrar la clave en el momento del registro (método “registrarUsuario” de “Sistema()”)
- Comparar la clave que nos llega en el inicio de sesión con la de la base de datos (método “loginUsuario” de “Sistema()”

En nuestra solución se sugiere utilizar la librería Bcrypt.

El modo de usar los métodos de cifrado y comparación se explican en el siguiente tutorial:

https://www.makeuseof.com/nodejs-bcrypt-hash-verify-salt-password/

Comprueba que las nuevas claves de los usuarios registrados aparecen cifradas en la base de datos.

2.8 Asegurar las rutas

Otro paso importante relacionado con la seguridad de nuestra aplicación consiste en securizar las rutas del API Rest para que sólo puedan se utilizadas por los usuarios registrados.

Para ello vamos a utilizar una nueva función “haIniciado()” que insertaremos en las rutas que queramos securizar. Esta función la ubicamos en el archivo “index.js”:

```
const haIniciado=function(request,response,next){
  if (request.user){
    next();
  }
  else{
    response.redirect("/")
  }
}
```

Y un ejemplo de ruta securizada sería el siguiente:

```
app.get("/obtenerUsuarios",haIniciado,function(request,response){
    let lista=sistema.obtenerUsuarios();
    response.send(lista);
});
```

Para poder comprobar el correcto funcionamiento de esta funcionalidad es imprescindible que la opción “salir” de la aplicación funcione adecuadamente, es decir, que borre toda huella de nuestra aplicación en el cliente, y también en el servidor.

2.9 Implementar Salir

Vamos a modificar el modo que utilizan los usuarios para cerrar sesión en nuestra aplicación.

Comenzaremos en el cliente, en concreto en “controlWeb.js”:

```
    this.salir=function(){
        //localStorage.removeItem("nick");
        $.removeCookie("nick");
        location.reload();
        rest.cerrarSesion();
    }
```

Como se puede ver, ahora tenemos una nueva petición REST que se llama “cerrarSesion()”. Las peticiones REST se ubican en el componente “ClienteRest()” que está en el archivo “clienteRest.js”:

```
this.cerrarSesion=function(){
        $.getJSON("/cerrarSesion",function(){
            console.log("Sesión cerrada");
            $.removeCookie("nick");
        });
    }
```

El siguiente paso es gestionar la ruta “/cerrarSesion” en la capa REST, esto es, en el archivo “index.js”:

```
app.get("/cerrarSesion",haIniciado,function(request,response){
  let nick=request.user.nick;
  request.logout();
  response.redirect("/");
  if (nick){
    sistema.eliminarUsuario(nick);
  }
});
```

2.10 Mejoras adicionales

Robustez: Antes de realizar mejoras, se debe comprobar el mayor número posible de situaciones excepcionales que puedan comprometer la estabilidad del sistema.

Mejorar la experiencia de usuario

Mejora de los formularios:

- Control de campos de entrada: email, clave
- Mejorar el feedback mediante mensajes que informen al usuario del resultado de su interacción.

Mejorar el correo de confirmación (el formato del HTML).

Integrar la funcionalidad “agregarUsuario” en los diferentes inicios que hemos implementado en este Sprint 2.

Mejorar los mensajes de la consola de servidor (evitar mensajes que no aportan información al administrador de la aplicación).

Securizar la app con JWT:
https://soshace.com/securing-node-js-applications-with-jwt-and-passport-js/

Anexo A. Soluciones de los ejercicios propuestos

A.1 Implementar el inicio de sesión

A continuación se muestran los métodos necesarios para implementar el inicio de sesión, comenzando desde el cliente Web y recorriendo toda la arquitectura.

Archivo “login.html”:

```
<div id="fmLogin">
    <form>
        <div class="form-group">
            <label for="email">Dirección email:</label>
            <input type="email" class="form-control"
placeholder="Introduce email" id="email">
        </div>
        <div class="form-group">
            <label for="pwd">Password:</label>
            <input type="password" class="form-control"
placeholder="Introduce password" id="pwd">
        </div>
        <button type="submit" id="btnLogin" class="btn btn-
primary">Iniciar sesión</button>
    </form>
</div>
```

Método “mostrarLogin()” de ControlWeb():

```
 this.mostrarLogin=function(){
        if ($.cookie('nick')){
            return true;
        };
        $("#fmLogin").remove();
        $("#registro").load("./cliente/login.html",function(){
            $("#btnLogin").on("click",function(){
                let email=$("#email").val();
                let pwd=$("#pwd").val();
                if (email && pwd){
                    rest.loginUsuario(email,pwd);
                    console.log(email+" "+pwd);
                }
            });
        });
    }
```

Método “loginUsuario(email,password)” de ClienteRest():

```
 this.loginUsuario=function(email,password){
        $.ajax({
            type:'POST',
            url:'/loginUsuario',
            data: JSON.stringify({"email":email,"password":password}),
            success:function(data){
                if (data.nick!=-1){
                    console.log("Usuario "+data.nick+" ha sido
registrado");
                    $.cookie("nick",data.nick);
                    cw.limpiar();
                    cw.mostrarMensaje("Bienvenido al sistema,
"+data.nick);
                    //cw.mostrarLogin();
                }
                else{
                    console.log("No se pudo iniciar sesión");
                    cw.mostrarLogin();
                    //cw.mostrarMensajeLogin("No se pudo iniciar
sesión");
                }
             },
             error:function(xhr, textStatus, errorThrown){
                console.log("Status: " + textStatus);
                console.log("Error: " + errorThrown);
             },
            contentType:'application/json'
        });
    }
```

Endpoint tipo POST “/loginUsuario” en el archivo “index.js”

```
app.post('/loginUsuario',passport.authenticate("local",{failureRedirect:"
/fallo",successRedirect: "/ok"})
);

app.get("/ok",function(req,res){
  res.send({nick:req.user.email})
});
```

Método “loginUsuario(obj,callback)” de Sistema():

Este método asume que la password del usuario se ha cifrado en el proceso de registro (se incluye el método “registrarUsuario” de “Sistema” al final).

```
 this.loginUsuario=function(obj,callback){
        let modelo=this;
```

```
        this.cad.buscarUsuario({"email":obj.email,"confirmada":true},func
tion(usr){
            if (!usr)
            {
                callback({"email":-1});
                return -1;
            }
            else{
                bcrypt.compare(obj.password, usr.password, function(err,
result) {
                    if (result) {
                        callback(usr);
                        modelo.agregarUsuario(usr);
                    }
                    else{
                        callback({"email":-1});
                    }
                });
            }
        });
    }
```

Métodos “buscarUsuario” y “buscar” de CAD():

```
this.buscarUsuario=function(criterio,callback){
        buscar(this.usuarios,criterio,callback);
    }
```

```
 function buscar(coleccion,criterio,callback){
        let col=coleccion;
        coleccion.find(criterio).toArray(function(error,usuarios){
            if (usuarios.length==0){
                callback(undefined);
            }
            else{
                callback(usuarios[0]);
            }
        });
    }
```

Métodos “insertarUsuario” e “insertar” de CAD:

```
this.insertarUsuario=function(usuario,callback){
        insertar(this.usuarios,usuario,callback);
    }
```

```
function insertar(coleccion,elemento,callback){
        coleccion.insertOne(elemento,function(err,result){
```

```
            if(err){
                console.log("error");
            }
            else{
                console.log("Nuevo elemento creado");
                callback(elemento);
            }
        });
    }
```

A.2 Registro de usuarios cifrando la clave con la librería bcrypt

Este nuevo método “registrarUsuario” de “Sistema()” incluye el cifrado de la password del usuario utilizando la librería bcrypt.

```
this.registrarUsuario=function(obj,callback){
        let modelo=this;
        if (!obj.nick){
            obj.nick=obj.email;
        }
        this.cad.buscarUsuario({"email":obj.email},async function(usr){
            if (!usr){
                let key=Date.now().toString();
                obj.confirmada=false;
                obj.key=key;
                const hash = await bcrypt.hash(obj.password, 10);
                obj.password=hash;
                modelo.cad.insertarUsuario(obj,function(res){
                   callback(res);
                });
                correo.enviarEmail(obj.email,key,"Confirmar cuenta");
            }
            else
            {
                callback({"email":-1});
            }
        });
    }
```

Observa que la inserción del nuevo registro y el envío del email de confirmación de cuenta se realizan de manera asíncrona.