Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Implementación

En las secciones siguientes se mencionarán los aspectos más relevantes de la implementación.

1.1. Implementación del primer incremento

El primer incremento fue la primera toma de contacto con la tecnología utilizada tras la formación.

Los requisitos implementados en el primer incremento son los asociados a la parte del cuestionario de emparejamiento:

- RF-005: Emparejamiento
- RF-006: Filtrado de los resultados en base a distintos criterios
- RNF-002: Tiempo de respuesta de asignación

Las tareas más trascendentales realizadas en este incremento son las descritas a continuación:

Configuración Angular Todos los archivos Angular utilizados deben importarse en la *root view* (app/views/index.html en el proyecto): Librerías JavaScript, app.js, route.js, controllers y factories.

Para indicar a la aplicación que se trata de una aplicación Angular se debe especificar en la etiqueta ¡html¿de la *root view* (app/views/index.html) de la siguiente forma:

```
1 <html ng-app="Emozio"> </html>
```

Se define el módulo principal de la aplicación en el archivo app/assets/javascript/app.js:

```
1 angular.module('Emozio', ['ngRoute', 'ngResource']);
```

Entre corchetes se encuentran las dependencias que tiene nuestro módulo Emozio:

- `ngRoute`: Permite a la aplicación convertirse en una SPA, permitiendo la navegación ente distintas páginas sin necesidad de recargar. W3 angular-routing
- `ngResource`: Permite crear objetos para interactuar con los datos RESTful del lado servidor. El objeto devuelto tiene métodos de acción que proporcionan comportamiento sin necesidad de interactuar con el servicio $http a bajo nivel. Las acciones por defecto son:

```
1 { 'get':    {method:'GET'},
2   'save':   {method:'POST'},
3   'query':  {method:'GET', isArray:true},
4   'remove': {method:'DELETE'},
5   'delete': {method:'DELETE'} };
```

Llamar a estos métodos invoca a $http con el método http especificado, destino y parámetros.

Para indicar al $routeProvider dónde mostrar las templates se utiliza la siguiente directiva: `<div ui-view></div>`

Template	***Controller***
pacientes/cuestionarioPacientes.html	pacientes/cuestionarioController.js
pacientes/pacientesResultados.html	pacientes/perfilController.js

Implementación de las *templates* y *controllers* de los requisitos. Éstos se encuentran dentro del directorio app/assets.

Creación de las *factories*, *routes* y *models* de: Paciente, Psicologo y Patologia.

```
1 var MONGO_URL = 'mongodb://localhost:27017/emozio';
```

Se define la URL de la base de datos a la que se conecta la app.

```
1   var options = {
2     useMongoClient: true,
3     autoIndex: false, // No crear index
4     reconnectTries: Number.MAX_VALUE, // Nunca para de reintentar
          conectarse
5     reconnectInterval: 500, // Reconexion cada 500ms
6     poolSize: 10, // Mantener una conexion de 10 sockets
7     // Si no se conecta, devuelve un error inmediatamente antes
         de tratar de reconectarse
8     bufferMaxEntries: 0
9   };
```

Se establecen las opciones de conexión.

```
1 mongoose.Promise = global.Promise;
```

Se declara que las *promise* que va a utilizar Mongoose son las globales proporcionadas por Bluebird. En la base de datos utilizamos las *promise* definidas e implementadas en la librería Bluebird.

```
1 mongoose.connect(MONGO_URL, options, function(err, res) {
2   if(err) {
3     console.log('ERROR: Reconectando a la BBDD. ' + err);
4   }else{
5     console.log("Conectado a la BBDD");
6   }
7 });
```

Función de conexión de Mongoose a la base de datos con las opciones especificadas.

```
1   var db = mongoose.connection;
2   /* Si sucede un error, mostrarlo */
3   db.on('error', console.error.bind(console, 'Error de conexion:'
        ));
4   db.once('open', function() {
5     console.log("Con exito");
6   });
```

Se abren las conexiones a la base de datos.

```
1 app.use("/", express.static("app/"));
```

Con express.static se especifican los directorios donde se encuentran los archivos que Express va a leer. Facilita el acceso a los *assets* (bienes) de la carpeta app desde el servidor. Lo que nos permite tener la parte cliente separada del servidor.

```
1 app.set('views', __dirname + '/../app/views');
```

Define que las routes a las templates se rendericen con el render method dentro del directorio views.

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

Especifica que los datos recogidos de un formulario se pasen a través del método *post*.

```
app.use(bodyParser.json());
```

Mediante el paquete Body-parser, podemos tratar los objetos en formato JSON sin necesidad de manipularlos, o cambiar su tipo.

```
var app = express();
```

Se inicializa el servidor express.

```
  app.get('/', function(req, res){
    res.sendfile('index.html', {root: app.settings.views});
  });
```

Express establece cuál va a ser la *view* raíz (*root*) enviada tras el inicio del servidor.

Además, se importan los archivos de server/routes que sirven como *endpoint* (punto medio) entre la parte cliente y servidor, de esta forma durante la implementación, logramos mantener la parte del *frontend* independiente del *backend* haciéndola funcional.

1.2. Implementación del segundo incremento

Especificación del arranque del servidor y se establece cuál es la vista raíz en el archivo routes.js. También, se vinculan todos los archivos que va utilizar el servidor: Archivos de configuración, de conexiones al modelo... Los requisitos implementados en el segundo incremento son los asociados a la parte de la gestión de usuarios:

- RF-001: Acceso usuarios
- RF-002: Registro
- RF-003: Baja
- RF-004: Modificación de los datos
- RNF-001: Encriptado de datos

Template	***Controller***
inicio.html	pacientes/pacientesAccesoController.js
pacientes/registroPacientes.html	pacientes/pacientesRegistroController.js
pacientes/pacientesModificar.html	pacientes/pacientesModificarController.js
psicologos/psicologosModificar.html	psicologos/psicologosModificarController.js
psicologos/registroPsicologo.html	psicologos/psicologosRegistroController.js

Gestión de sesiones de usuario con PassportJS y Express-session. *La configuración de PassportJS se realizó en el directorio server/config/passport.js*

Las credenciales utilizadas para autenticar a un usuario sólo son transmitidas durante la petición de acceso (*login request*). Si la autenticación se realiza con éxito, la sesión será establecida y mantenida vía una cookie en el navegador del usuario. Cualquier petición posterior no contendrá las credenciales, únicamente la *cookie* que identifica la sesión. Para poder gestionar las sesiones de acceso (*login sessions*), Passport *serialize* y *deserialize* instancias de usuario.

```
1 passport.serializeUser(function(usuarios, done){
2   done(null, usuarios._id);
3 })
```

Sólo el ID de usuario es "creado" en la sesión, manteniendo mínima la cantidad de datos guardados. Cuando las siguientes peticiones sean recibidas, este ID será el utilizado para encontrar al usuario, el cual fue guardado en `req.user`.

```
1  passport.deserializeUser(function(id, done){
2    Paciente.findById(id, function(error, usuario){
3      if(usuario!=null){
4        done(null, usuario);
5      } else {
6        Psicologo.findById(id, function(error, usuario){
7          done(null, usuario);
8        });
9      }
10   });
11 })
```

`deserializeUser()` es invocado en cada petición por `passport.session`. Permite cargar información adicional a la información de usuario en cada petición; este objeto está asociado a la petición como `req.user` haciéndolo accesible en la gestión de peticiones.

```
1 passport.use(new LocalStrategy(
2   {
3     usernameField: 'email',
4     passwordField: 'password'
5   },
6   function (username, password, done) {
```

```
7     Paciente.findOne({email: username}, function(error, paciente)
          {
8       if(!paciente){
9         //                return done(null, false, {message: '
              Este email: '+email+'no esta registrado'});
10        Psicologo.findOne({email: username}, function(error,
              psicologo){
11          if(!psicologo) {
12            return done(null, false, {message: 'Este email: '+
                  username+'no esta registrado'});
13          } else {
14            psicologo.compararPassword(password, function(error,
                  sonIguales){
15              if(sonIguales){
16                return done(null, psicologo);
17              } else {
18                return done(null, false, {message: 'La contrasena
                      no es valida'});
19              }
20            });
21          }
22        });
23      } else {
24        paciente.compararPassword(password, function(error,
              sonIguales){
25          if(sonIguales){
26            return done(null, paciente);
27          } else {
28            return done(null, false, {message: 'La contrasena no
                  es valida'});
29          }
30        });
31      }
32    });
33  }
34 ));
```

Para poder utilizar la autenticación por `username` y `password`, Passport utiliza el mecanismo proporcionado por su módulo passport-local.

- `UsernameField` y `PassworfField` son los obtenidos del cuerpo de la petición (`req.body`) recibida cuando un usuario quiere acceder.
- Se busca al paciente que posea esos datos; y se pueden dar dos casos:
  - Si no existe, se busca al psicólogo que posea esos datos:
    - Si no existe: El usuario no está registrado.
    - Si existe: Se comprueba que la contraseña sea la misma a la introducida. Si son iguales, el acceso es correcto. Si no, la contraseña no es válida.

- Si existe: Se comprueba que la contraseña sea la misma a la introducida. Si son iguales, el acceso es correcto. Si no, la contraseña no es válida.

```
exports.estaAutenticado = function (req, res, next){
  if(req.isAuthenticated()){
    return next();
  }
  req.session.error = 'Please sign in!';
  res.redirect('/');
}
```

Función que comprueba si el usuario que está realizando una petición, está autenticado.

En la route de Pacientes del directorio server/routes/paciente.js:

```
app.route('/pacientes/acceso')
    .post(function(req, res, next){
    setTimeout(function(){
      passport.authenticate('local', function(error, paciente,
          info){
        if(error){
          return next(error);
        }
        if(!paciente) {
          return null;
        }else{
          req.login(paciente, {}, function(err) {
            if (err) {
              return null;
            };
            return res.json(paciente);
          });
        }
      })(req, res, next); //Funcion que devuelve passport y que
          debe ser invocada de esta forma
    }, 50);
  });
```

El formulario de acceso es enviado por el servidor a través del método POST. Utilizando `authenticate()` es como gestionamos la petición de acceso: En el caso de que el paciente exista, hacemos el `login`.

```
  app.use(session({
    /* Se utiliza para firmar el ID de sesion de la cookie*/
    secret: 'ESTO ES SECRETO',
    /* Por cada llamada realizada al servidor, la sesion se
       guardara en la BBDD */
    resave: true,
```

```
6     /* Cuando se realiza la llamada por primera vez, guarda un
          objeto vacio con informacion de esa session */
7     saveUninitialized: true,
8     store: new MongoStore({
9       url: MONGO_URL,
10      /* Si sucede un error, trata de volver a conectarse */
11      autoReconnect: true
12    })
13  }));
```

Se establecen las opciones de la sesión que será utilizada en la aplicación. Cada sesión es guardada en la base de datos a modo de registro, aunque por el momento no es un funcionamiento relevante para nuestra aplicación.

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

Se inicializa PassportJS.

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

Se determina que passport utilice las sesiones.

Encriptación bcrypt de las contraseñas En la aplicación, el paquete bcrypt es utilizado para encriptar las contraseñas de usuario.

Uno de los casos donde es necesario utilizar bcrypt es cuando es necesaria la comparación de contraseñas en el inicio de sesión. Se hace por medio de la siguiente función:

```
1   /* Comprobar una contrasena con su hash */
2   bcrypt.compare(password, this.password, function(error,
        sonIguales){
3     if(error){
4       return cb(error);
5     }
6     cb(null, sonIguales);
7   })
```

Compara la contraseña con la que está tratando de acceder el usuario después de cifrarla con el hash que existe actualmente en la base de datos. El hash que se encuentra en la base de datos tiene almacenados el coste, la sal y la contraseña, por lo que bcrypt sabe cómo cifrar la nueva contraseña introducida.

Un ejemplo práctico: El hash introducido en la base de datos podría ser el siguiente: `$2a$10$vI8aWBnW3fID.ZQ4/zo1G.q1lRps.9cGLcZEiGDMVr5yUP1KUOYTa`

Este hash contiene tres cadenas concatenadas con el símbolo "$":

- `2a` identifica la versión del algoritmo bcrypt utilizado.
- `10` es el factor de coste: Se han utilizado 2^{10} iteraciones de la función de derivación de la key.

- `vI8aWBnW3fID.ZQ4/zo1G.q1lRps.9cGLcZEiGDMVr5yUP1KUOYTOa` son la sal y la contraseña cifrados, concatenados y codificados en Base64. Los 22 primeros caracteres codificados en un valor para la sal de 16-byte. El resto de caracteres son la contraseña cifrada que va a ser comparada durante la autenticación.

Otro de los casos donde es necesario utilizar bcrypt es al dar de alta a un usuario.

```
1  bcrypt.genSalt(10, function(error, salt){
2    if(error) {
3      console.log("Error en la sal");
4    }
5    bcrypt.hash(paciente.password, salt, null, function(error,
        hash){
6      if(error) {
7      console.log("Error en el hash");
8      }
9    });
10 });
```

Primero se genera la sal con un factor de coste 10, y después, se crea el hash con la contraseña que ha introducido el paciente y la sal. Este hash será almacenado en la base de datos.

Google Maps JavaScript API para el autocompletado de la localización en los formularios de registro y modificación. Para autocompletar las localizaciones introducidas por el usuario tanto en el formulario de registro como en el de modificación, se ha utilizado la Autocompletado para direcciones y términos búsqueda de Google Maps JavaScript API.

Por ejemplo, en el formulario de registro de usuarios de la *template* pacientes/registroPacientes.html, fue incorporado al código HTML de la siguiente forma:

```
1  <div class="field" id="locationField">
2    <div class="ui left icon input">
3      <i class="marker icon"></i>
4      <input id="autocomplete" name="localizacion" placeholder="
          Localizacion" ng-focus="geolocate()" type="text"
          required>
5    </div>
6  </div>
```

Donde ng-focus ng-focus es una directiva que indica que cuando el input esté señalado, se ejecute la función `geolocate()`.

La función `geolocate` se encuentra en pacientes/pacientesRegistroController.js y viene definida por Google de la siguiente forma:

```
1  $scope.geolocate = function() {
2    if (navigator.geolocation) {
3      navigator.geolocation.getCurrentPosition(function(position)
           {
4        var geolocation = {
5          lat: position.coords.latitude,
6          lng: position.coords.longitude
7        };
8      });
9    }
10 }
```

Esta función, simplemente toma la ubicación donde se sitúa el usuario en ese momento. $scope se utiliza para pasar la función del *controller* a la *template*.

Por otra parte, la página se encuentra contenida bajo la siguiente etiqueta:

```
1 <div class='container' ng-init="initAutocomplete()">...</div>
```

La directiva ng-init ng-init evalúa `initAutocomplete()` al inicializar la aplicación.

La función `initAutocomplete` se encuentra en el *controller*.

```
1  $scope.initAutocomplete = function() {
2    autocomplete = new google.maps.places.Autocomplete(
3      (document.getElementById('autocomplete')),
4      { types: ['(cities)'], /* Se buscaran ciudades */
5       componentRestrictions: {country: "es"}}); /* Restringidas
           dentro de Espana */
6
7    /* Cuando el usuario selecciona una opcion del desplegable,
        se ejecuta la funcion indicada */
8    autocomplete.addListener('place_changed', fillInAddress);
9
10   /* Funcion que recupera el lugar escogido en el
        autocompletado */
11   function fillInAddress() {
12     /* Toma los detalles del lugar del objeto de autocompletado
           */
13     place = autocomplete.getPlace();
14   }
15 }
```

`google.maps.places.Autocomplete` crea un objeto tomando como argumentos el `input` donde se quiere autocompletar y una serie de opciones. Para la aplicación, se han restringido las opciones a ciudades españolas.

Nodemailer.js Cuando un psicólogo cubre el formulario de registro, sus datos son mandados al correo electrónico de Emozio para poder valorarlos antes de formar parte de nuestra base de datos. Para enviar los e-mails se ha utilizado Nodemailer. Algunas particularidades son descritas a continuación.

```
var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'emozio.info@gmail.com',
    pass: '********************'
  }
});
```

Se crea un objeto transporter utilizado en el trasporte por defecto SMTP. SMTP es el transporte utilizado en Nodemailer para el envío de mensajes, pero también, es el protocolo utilizado por diferentes *host* de *email*.

```
var mailOptions = {
  from: 'emozio.info@gmail.com',
  to: 'emozio.info@gmail.com',
  subject: 'Emozio Web - Nuevo psicologo',
  generateTextFromHTML: true,
  html: /*Aqui iria el codigo HTML */
};
```

Se especifican las opciones de *email* que se quieren enviar.

```
transporter.sendMail(mailOptions, (error, info) => {
  if (error) {
    return console.log(error);
  }
  console.log('Message sent: %s', info.messageId);
});
```

Se envía el *email* con la función `sendMail` a través del `transporter`.

Validación de formularios con SemanticUI Para la validación de formularios se utilizó SemanticUI que funciona de la siguiente forma:

En la *template*, se han de tener todos los campos de formulario corréctamente nombrados y añadir un identificador a la etiqueta `<form>`.

En el *controller*, existe un objeto especial al que se le pueden pasar la lista de elementos del formulario a validar, las reglas que ha de cumplir cada campo y los mensajes de error en cada caso.

En el código de la aplicación, un ejemplo podría ser:

```
$('#access_form').form({
  on : 'blur', /* Cada elemento se evalua por separado */
  inline: 'false', /* Los mensajes de validacion no se disponen
      en linea */
  /* Campos a validar: Identificador y reglas a evaluar */
  fields : {
    email : {
      identifier : 'email',
      rules : [
        {
```

```
10              type : 'empty',
11              prompt : 'Por favor, introduzca un e-mail.'
12            },
13            {
14              type: 'email',
15              prompt: 'El formato del e-mail es incorrecto.'
16            },
17            {
18              type: 'maxLength[50]',
19              prompt: 'Demasiados caracteres.'
20            }
21          ]
22        },
23        password: {
24          identifier: 'password',
25          rules: [
26            {
27              type: 'empty',
28              prompt: 'Por favor, introduzca una contrasena.'
29            },
30            {
31              type: 'maxLength[50]',
32              prompt: 'Demasiados caracteres.'
33            }
34          ]
35        }
36      }
37    });
```

Los campos a validar en el formulario de acceso a la plataforma son email y password.

Para email, las reglas de validación son:

- El campo no debe estar vacío.
- El formato debe ser de tipo email.
- La logitud máxima de carácteres permitidos es 50.

Para password, las reglas de validación son:

- El campo no debe estar vacío.
- La logitud máxima de carácteres permitidos es 50.

1.3. Implementación del tercer incremento

Los requisitos implementados en el segundo incremento son los asociados a la parte de la comunicación:

- RF-007: Contacto del paciente con el psicólogo
- RF-008: Valoración del psicólogo por parte del paciente

Además, aunque en primera instancia no estuviese contemplado, surgió la idea de gestionar la comunicación como un sistema de citas. Por ello, se incorporó a posteriori un calendario mensual donde el psicólogo pudiese ver las citas que tiene agendadas con una vista por mes, por semana y por día.

Las tareas más trascendentales realizadas en este incremento son las descritas a continuación:

Template	***Controller***
pacientes/pacientesMail.html	pacientes/pacientesMailController.js
psicologos/psicologosMail.html	psicologos/psicologosMailController.js
psicologos/psicologoCalendario.html	psicologos/psicologosCalendarioController.js

Implementación de las *templates* y *controllers* de los requisitos. Éstos se encuentran dentro del directorio app/assets.

Creación de la *factories*, *routes* y *models* de Mensaje.

Visualización de las citas agendadas por medio de FullCalendar En la *template* psicologoCalendario.html, el calendario es dispuesto de la siguiente forma:

```
<div id="calendar" ui-calendar ng-model="eventSources"></div>
```

Este elemento es gestionado por el psicologosCalendarioController.js.

```
$('#calendar').fullCalendar({
  header: { /* Opciones de la barra de herramientas */
      left: 'prev,next today', /* A la izquierda: Anterior,
         siguiente y hoy */
      center: 'title', /* En el centro: Nombre del mes */
      right: 'month,basicWeek,basicDay' /* A la derecha: Mes,
         semana y dia */
    },
    buttonText : { /* Nombre que aparece en las opciones de la
       barra de herramientas */
      today:    'hoy',
      month:    'mes',
      week:     'semana',
      day:      'dia',
      prev:   '<',
      next:   '>'
    },
```

```
15        firstDay : 1, /* Empieza el lunes */
16        dayNames : ['Domingo', 'Lunes', 'Martes', 'Miercoles', '
              Jueves', 'Viernes', 'Sabado'], /* Nombres de los dias */
17        dayNamesShort : ['Dom.', 'Lun.', 'Mart.', 'Mierc.', 'Juev.'
              , 'Vier.', 'Sab.'], /* Nombres abreviados de los dias */
18        monthNames: ['Enero','Febrero','Marzo','Abril','Mayo','
              Junio','Julio','Agosto','Septiembre','Octubre','
              Noviembre','Diciembre'], /* Nombres de los meses */
19        monthNamesShort: ['Ene','Feb','Mar','Abr','May','Jun','Jul'
              ,'Ago','Sep','Oct','Nov','Dic'], /* Nombres abreviados
              de los meses */
20        navLinks: true, /* Permite navegar entre las distintas
              vistas: Mes, semana y dia */
21        editable: true, /* Los eventos pueden ser modificados */
22        eventLimit: true, /* Numero de eventos mostrados en un dia
              limitado */
23        theme: true, /* Activa el tema */
24        themeSystem:'bootstrap3', /* Establece el tipo de tema */
25        eventColor: '#ffffff', /* Establece el color del fondo de
              los eventos */
26        eventTextColor: '#008080', /* Establece el color del texto
              de los eventos */
27        eventClick: function (calEvent) { . . . },
28
29        height: 600 /* Establece la altura del calendario */
30      });
```

Para meter los eventos en el calendario se utiliza la función fullcalendar con el argumento `renderEvent`.

```
1 $('#calendar').fullCalendar('renderEvent', cita, true);
```

- Se inicializa el calendario.
- Opciones mostradas en los botones del calendario.
- Traducción de los mensajes de los botones (que traen por defecto) al castellano.
- Indica el día donde comienza la semana.
- Traducción de los días, abreviaturas de los días, meses y abreviaturas de los meses al castellano.
- Función que se ejecuta cuando se hace *click* en un evento y lo muestra.