



<Do!T/>

Proyecto final de Grado Superior en Desarrollo de
Aplicaciones Web. 2016 - 2017

Cristina Rodríguez
Iván Graña
Julián Egea

Tutores:
Alfredo Rueda
Fran Pérez

ÍNDICE

1. INTRODUCCIÓN	3
1.1. Descripción del proyecto	3
1.2. Presentación del equipo	4
1.3. Justificación del proyecto.....	5
1.4. Objetivos mínimos y máximos	5
1.5. Funcionalidades del proyecto	5
1.6. Evaluación de las tecnologías utilizadas.....	6
1.7. Evaluación del back-end.....	6
1.8. Evaluación del front-end	7
1.9. Planificación del proyecto	7
1.10. Diagrama de tecnologías	8
1.11. Diagrama de casos de uso	9
1.12. Diagrama de la aplicación.....	10
1.13. Diagrama de clases	11
2. DESARROLLO DEL PROYECTO	12
2.1. Registro.....	13
2.2. Login.....	19
2.3. Perfil de usuario	21
2.4. Retos.....	23
2.5. Eventos	26
2.6. Buscador de usuarios	33
2.7. Solicitudes de amistad.....	36
3. RESULTADOS.....	43
4. CONCLUSIONES Y LÍNEAS FUTURAS.....	44
5. WEBGRAFÍA Y BIBLIOGRAFÍA.....	45

1. INTRODUCCIÓN

1.1. *Descripción del proyecto*

Dolt es una aplicación web dividida en dos sectores. Primeramente, encontramos Dolt como red social, con un chat para que los usuarios interactúen entre ellos y un muro de actividad de tus amigos. El chat permitirá el envío de diferentes elementos de tipo multimedia como imágenes, vídeos y audios. Estos archivos multimedia se podrán mostrar mediante el sistema de gestión que la aplicación lleva incorporado, con tal de no tener que acceder a alguno ajeno a la aplicación. Además, también podremos compartir nuestros retos mediante el chat para lograr que tus contactos te voten.

El otro sector de Dolt es el ocio. Aquí, encontraremos una serie de retos semanales en los cuales nuestros usuarios podrán participar. Estos retos serán principalmente subir una imagen, un vídeo o un audio realizando aquello que el reto describa. Todos los usuarios que participen recibirán un logro y, al acabar el período disponible para participar (7 días), se hará un “top3” de las mejores participaciones en los retos para entregar a los usuarios que resulten ganadores las bonificaciones que correspondan. Con estos logros, los usuarios podrán ir formando su perfil.

Los retos tendrán una funcionalidad añadida, con la que los usuarios podrán crear eventos para realizar esos eventos e invitar a sus contactos. El reto constará de un mapa en el que señalaremos donde se quedará con los contactos y de una lista de asistentes. Los usuarios invitados podrán indicar si asistirán o no y si han llegado al lugar indicado.

Para poder crear estos hilos entre usuarios, estos también tendrán en su mano una herramienta de búsqueda que funcione con todos los atributos de un usuario disponibles (principalmente la búsqueda se hará con el correo electrónico de éste).

1.2. Presentación del equipo

Este proyecto ha sido realizado por un grupo de alumnos de CFGS de DAW con el objetivo de poner a prueba todas las habilidades adquiridas durante los dos años de ciclo. Antes de empezar a describir el proyecto se presenta a los miembros del grupo.

Julian Egea ([LinkedIn Julián](#))

Estudiante de 2º de Desarrollo de Aplicaciones Web, titulado en Desarrollo de Aplicaciones Multiplataforma. Apasionado de los videojuegos y la tecnología. Trabajé en Tiendeo, una empresa de marketing-digital, de web crawler en PHP. En un futuro próximo me gustaría dedicarme al desarrollo de videojuegos.

Iván Graña ([LinkedIn Ivan](#))

Actual estudiante de 2º de Desarrollo de Aplicaciones Web, titulado en Desarrollo de Aplicaciones Multiplataforma y Sistemas Microinformáticos y Redes. Me apasiona la tecnología y la informática, y por eso decidí tirar por los estudios de vocación informática. Mi experiencia profesional se basa en las prácticas curriculares de los propios cursos y en las prácticas realizadas en empresas durante los cursos correspondientes. Al terminar este curso, me gustaría encontrar un trabajo que me permita crecer como programador.

Cristina Rodríguez ([LinkedIn Cristina](#))

Vinculada desde que cursaba bachillerato a la programación, decidí empezar a estudiar un ciclo superior dedicado a ello, con lo que llegué a sacarme la titulación en Desarrollo de Aplicaciones Multiplataforma. Actualmente me encuentro estudiando 2º de Desarrollo de aplicaciones web con la finalidad de extender mi conocimiento y poder abarcar un área de trabajo mayor y así poder tener oportunidades laborales más amplias.

1.3. Justificación del proyecto

Actualmente, las aplicaciones web de este tipo están muy diversas. Por ello la creación de Dolt significará la unificación de todas esas aplicaciones ya existentes en el mercado. De éstas, se extraerán las funcionalidades más usadas por sus usuarios y, al juntarlas todas en una misma aplicación, haremos que todo el mercado quede más unido y sea más fácil para los usuarios realizar aquellas tareas que más le gustan.

1.4. Objetivos mínimos y máximos

Los objetivos mínimos planteados para el proyecto son: tener un apartado de login y tener un registro de usuarios. Una vez dentro de la aplicación, el usuario será capaz de añadir nuevos usuarios/contactos, un buscador de usuarios/contactos y un chat entre los propios usuarios de la aplicación.

Los objetivos máximos planteados para el proyecto son: todos los objetivos mínimos comentados anteriormente, un sistema de retos donde los usuarios puedan inscribirse a dicho reto y el gestor de contenido multimedia, en el cual los usuarios puedan visualizar sus fotografías de los retos, canciones, vídeos.

1.5. Funcionalidades del proyecto

La mayor funcionalidad de nuestro proyecto es la gestión de usuarios. Englobando esta funcionalidad encontramos el buscador de usuarios, la función de enviar solicitudes de amistad a los usuarios, aceptar y negar estas solicitudes y además borrar las amistades. Además, los usuarios también podrán modificar su propio perfil y eliminarlo.

Por orden de utilidad encontramos después la gestión de multimedia, con lo que podremos participar en retos, subir una imagen a un evento y añadir una imagen a nuestro perfil para darle más personalidad. Por último, aunque no menos importante, también tendremos la funcionalidad de gestionar eventos, crearlos, editarlos e incluso invitar a amigos para que se apunten a nuestro evento.

1.6. Evaluación de las tecnologías utilizadas

El proyecto de Dolt se generará primero usando JDL-Studio para el diseño del UML del proyecto, definiendo las clases y sus atributos correspondientes.

Una vez realizado esto, el back-end se generará con JHipster, mediante el archivo jd, lo que nos dará un back-end en java, con una estructura definida para poder empezar a trabajar.

En la base de datos del proyecto se usará MySQL. Y el front-end del proyecto se realizará usando Angular y Bootstrap.

En el apartado del back-end, se recurrirá a API's de terceros como la de Google Maps, para la localización de los retos en el mapa. Así el usuario podrá visualizar rápidamente, donde se realizará el evento sin necesidad de otra aplicación externa.

También se utilizó Jhipster, que nos permitió generar mediante su consola las entidades y sus correspondientes campos, y así ir generando nuestra base datos. En su apartado gráfico, nos permitía ir viendo si las peticiones entre el backend y el frontend se iban ejecutando, observar si las respuestas eran correctas.

1.7. Evaluación del back-end

El back-end de Dolt, se compondrá de la estructura generada a partir del UML, realizado mediante JDL-Studio y JHipster. De base, estará dividido entre las clases y los atributos. Además, encontraremos los resources, que serán aquellos archivos que contendrán las API de nuestros proyectos y los repositorios que serán los archivos que gestionarán todas las peticiones a la base de datos que usaremos en los resources.

La base de datos será de tipo relacional, MySQL, y se encargará de gestionar todos los datos de nuestra aplicación, desde los usuarios registrados hasta la seguridad de estos.

1.8. Evaluación del front-end

El front-end de Dolt estará desarrollado usando como base Bootstrap para el diseño y Angular para la gestión de datos. Usando Javascript conectaremos las API creadas en back-end para usarlas desde la parte cliente de la aplicación y poder acceder a los datos para mostrarlos y usarlos.

El front-end se distribuye en varios tipos de archivos: el service será el archivo que definirá las peticiones a back-end. Para que funcionen deberemos decir a qué url queremos conectar y qué gestión de datos haremos. Después encontramos el controller, un archivo js con el cual recogeremos los datos usando las funcionalidades definidas en el service y que nos permitirá finalmente mostrar los datos deseados en el último archivo, que sería el front-end en sí, el archivo html que contiene el diseño de la página indicada.

1.9. Planificación del proyecto

Nuestro grupo de trabajo está formado por 3 integrantes, lo que hará que el desarrollo del proyecto sea mucho más fácil. Para dividirnos el proyecto hemos dividido el trabajo entre front-end y back-end.

En el primer apartado tenemos a Julián Egea, encargado tanto del diseño como de la parte de Javascript y, en la parte de back-end estarían Iván Graña y Cristina Rodríguez, encargados de la programación de la parte servidor del proyecto. Pese a haber esta división, los integrantes también saldrán de su zona de trabajo para poder prestar apoyo al otro lado del proyecto.

1.10. Diagrama de tecnologías

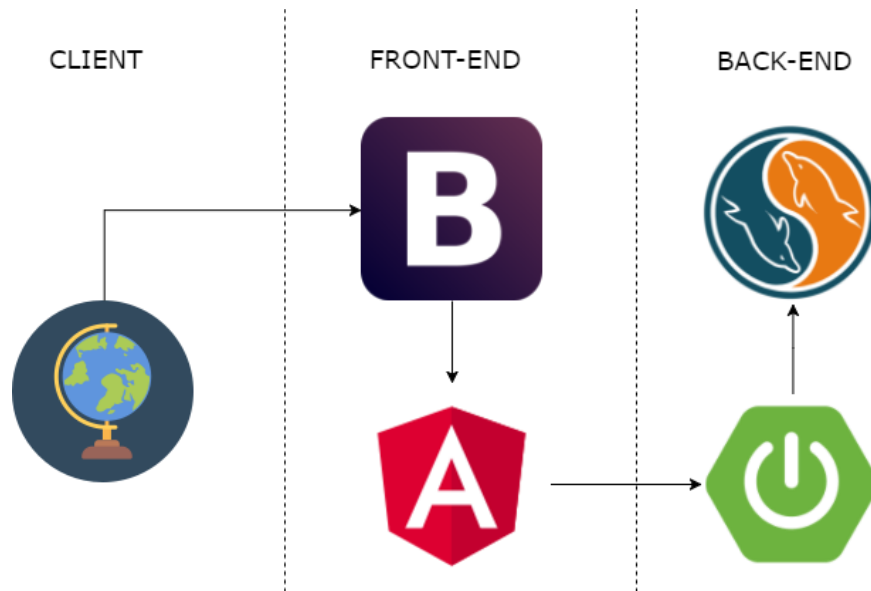


Diagrama de tecnologías

En este diagrama encontramos las diferentes tecnologías que utilizaremos y la zona del proyecto en la que lo usaremos. Por ejemplo, en front-end usaremos Bootstrap como diseño y Angular para la gestión de datos y en back-end tendremos Spring para el servidor y MySQL para gestión de datos.

1.11. Diagrama de casos de uso

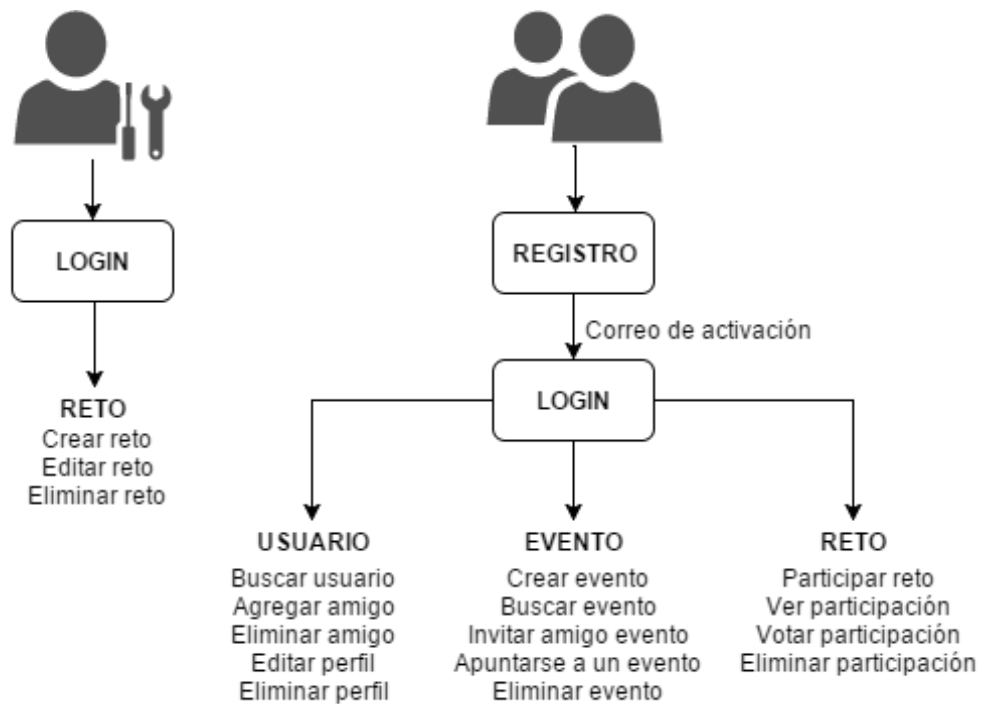


Diagrama de casos de uso

En los casos de uso vemos la diferencia entre un usuario administrador y un usuario básico. En el caso de ser usuario administrador tendremos unos pocos privilegios más de los que tiene un usuario básico.

Además, también está la opción de ser un usuario básico que sea administrador de un evento, lo que hará que tenga privilegios nuevos como el de editar ese evento o incluso borrarlo, cosa que alguien que no sea administrador del evento no podrá hacer.

1.12. Diagrama de la aplicación

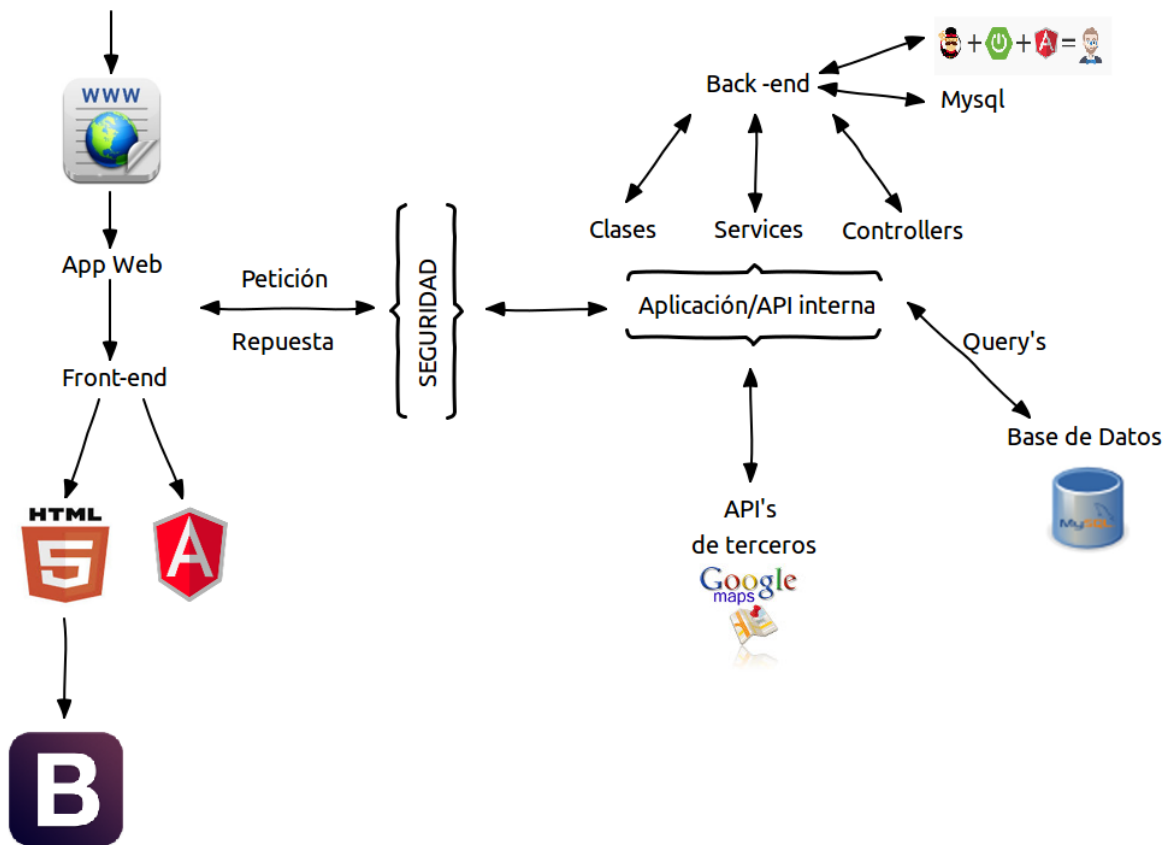


Diagrama de la aplicación

En el siguiente diagrama de nuestra aplicación se muestra el funcionamiento interno de la misma.

La aplicación está basada en el modelo MVC (Model View Controller). El cliente entra a la aplicación, en la que verá la vista adecuada. A la hora de consumir datos, la vista mediante el controller se comunicará con un servicio que le proporciona la dirección que debe seguir el controller para consumir una API adecuada. La API le proporcionará al controller los datos necesarios a petición de la vista para que, a su vez, el mismo controller modifique la vista para la visualización de los datos necesarios en ese momento.

1.13. Diagrama de clases

Al tener claro el concepto del proyecto y teniendo en cuenta el comunicar el frontend con el backend, empezamos a diseñar las primeras funcionalidades de la aplicación, eso nos llevó a plantearnos la estructura de nuestra base de datos.

La mejor manera para tener una base de datos óptima, es previamente establecer que opciones ofrece el proyecto y qué acciones podrá realizar el usuario. Con estas ideas claras se empezó con el diseño del UML de la base de datos.

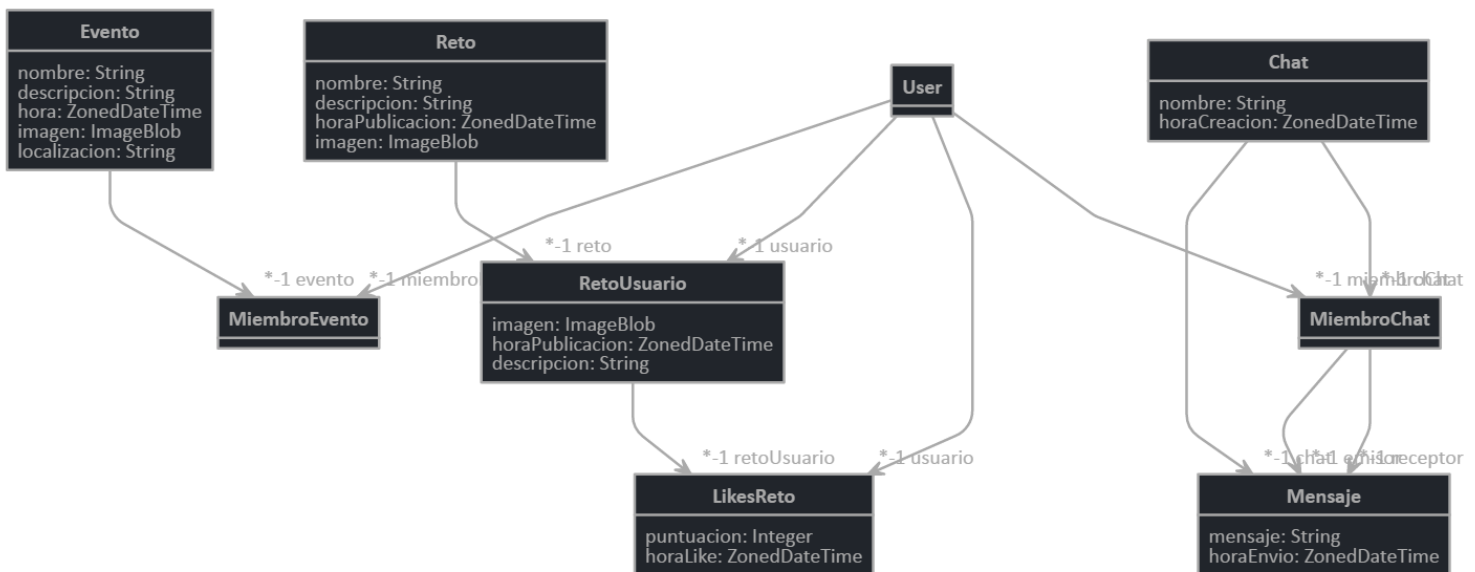


Diagrama de clases

2. DESARROLLO DEL PROYECTO

Para desarrollar el proyecto dividimos las funcionalidades entre front-end y back-end y nos asignamos cada una a un participante del grupo. Para ello, usamos la issue de github. Con ella gestionamos los tiempos de entrega y los commits que cada integrante hace sobre esa funcionalidad.

The screenshot shows the GitHub interface for the repository 'CristinaRodPalm / Dolt'. At the top, there are tabs for Code, Issues (7), Pull requests (0), Projects (0), Wiki, Settings, and Insights. Below the tabs, there's a search bar with the text 'is:issue is:open' and buttons for 'Filters', 'Labels', 'Milestones', and a 'New issue' button. The main content area displays a list of 7 open issues. The issue 'Modificar el HTML y CSS de la App' is selected and highlighted. The list includes issues like 'Editar perfil usuario', 'HTML mensajes', 'Controlar tamaño imagen', 'Buscador de eventos', 'Invitar amigo a un evento', and 'Usuarios puntúan la participación al reto'.

Issue Title	Author	Open Date	Status
Editar perfil usuario	CristinaRodPalm	opened an hour ago	Open
Modificar el HTML y CSS de la App	Julianegpo	opened 18 days ago	Open
HTML mensajes	CristinaRodPalm	opened 23 days ago	Open
Controlar tamaño imagen	CristinaRodPalm	opened on Apr 20	Open
Buscador de eventos	CristinaRodPalm	opened on Mar 20	Open
Invitar amigo a un evento	CristinaRodPalm	opened on Feb 23	Open
Usuarios puntúan la participación al reto	CristinaRodPalm	opened on Feb 17	Open

Issues en Github del proyecto

Para poder explicar cómo acabamos desarrollando el proyecto vamos a explicar funcionalidad a funcionalidad la integración de front-end y de back-end y como se relacionan entre ellas. Además, adjuntaremos fragmentos de código y los commits relacionados para que la explicación sea más completa y se pueda comprender mejor la integración del código.

El hilo de la explicación será el mismo que el formato de uso de un usuario, empezando por el registro y el login de los usuarios y seguido de las funcionalidades completas como la totalidad de los eventos, los retos...

2.1. Registro

La aplicación consta de un registro inicial en el cual un nuevo usuario puede rellenar un formulario con sus datos para poder acceder a la aplicación. Comentar que el registro se generó de forma automática con JHipster, lo que nos dió unos campos por defecto.

Como para nuestro proyecto necesitabamos otros campos, generamos una entidad auxiliar que los gestionase y le hicimos una relación 1-1 para poder acceder a estos desde el usuario y al revés el teléfono, la fecha de nacimiento y la imagen. Una vez añadidos, el resultado final será este.

The image shows a web registration form titled "Registro". It contains several input fields for user information: "Nombre de usuario", "Tu nombre", "Tus apellidos", "Tu correo electrónico", "Tu telefono", and a date picker for "Fecha de nacimiento". There are also password fields for "Nueva contraseña" and "Confirmación de la nueva contraseña", with a password strength indicator below the first. At the bottom, there is an "Imagen" section with a button labeled "Añadir imagen".

Registro

Usuario

Nombre de usuario

Nombre

Tu nombre

Apellidos

Tus apellidos

Correo electrónico

Tu correo electrónico

Teléfono

Tu telefono

Fecha de nacimiento

Nueva contraseña

Nueva contraseña

Seguridad de la contraseña:

Confirmación de la nueva contraseña

Confirmación de la nueva contraseña

Imagen

Añadir imagen

registro.html

A continuación, explicaremos detalladamente cómo han sido añadidos los campos adicionales.

[Commit 5b44a4e211e062fc55502b405d542742ac7f4bb2](#)

[Commit a7844e4951a057151d2ba0ca7450867b198bdfc0](#)

En la siguiente captura, mostramos los atributos de la nueva clase userExt, clase que guardará los nuevos campos del usuario. Para conseguirlo, crearemos una relación con el usuario definido por JHipster para que se conecten automáticamente.

```
@Entity
@Table(name = "user_ext")
public class UserExt implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "fecha_nacimiento")
    private ZonedDateTime fechaNacimiento;

    @Lob
    @Column(name = "imagen")
    private byte[] imagen;

    @Column(name = "imagen_content_type")
    private String imagenContentType;

    @NotNull
    @Column(name = "telefono", nullable = false)
    private String telefono;

    //@Past
    @Column(name = "nacimiento")
    private LocalDate nacimiento;

    @OneToOne(optional = false)
    @NotNull
    @JoinColumn(unique = true)
    private User user;
```

UserExt.java

Para que el usuario se pueda registrar con los campos auxiliares también, debemos añadir éstos al formulario de registro. En el html añadimos los inputs de la fecha de nacimiento, imagen y teléfono y además también gestionamos los posibles errores que se pueden encontrar en un registro.

```
<!-- telefono usuario -->
<div class="form-group">
  <label class="control-label" data-translate="global.form.telefono">Teléfono</label>
  <input type="text" class="form-control" id="phone" name="phone" placeholder="{{global.form.telefono.placeholder | translate}}"
    ng-model="vm.registerAccount.phone" ng-minlength=9 ng-maxlength=9 ng-pattern="/^[0-9]/" >
</div>
<!-- fecha nacimiento -->
<div class="form-group">
  <label class="control-label" data-translate="global.form.nacimiento" for="field_nacimiento">Nacimiento</label>
  <div class="input-group">
    <input id="field_nacimiento" type="text" class="form-control" name="nacimiento" uib-datepicker-popup="{{dateformat}}"
      ng-model="vm.registerAccount.nacimiento" is-open="vm.datePickerOpenStatus.nacimiento"
    />
    <span class="input-group-btn">
      <button type="button" class="btn btn-default" ng-click="vm.openCalendar('nacimiento') "><i class="glyphicon glyphicon-calendar"></i></button>
    </span>
  </div>
</div>
```

register.html

La gestión de errores en el teléfono se puede hacer directamente desde html pero para controlar la edad debemos esperar a que el resource de crear usuario lo gestione para ver si son correctos o no.

```
if(managedUserVM.getNacimiento().isAfter(LocalDate.now())) {
  return new ResponseEntity<>({ body: "afternow", textPlainHeaders, HttpStatus.BAD_REQUEST});
} else if(managedUserVM.getNacimiento().getYear()+16 > LocalDate.now().getYear()) {
  return new ResponseEntity<>({ body: "menor", textPlainHeaders, HttpStatus.BAD_REQUEST});
}
```

AccountResource.java

Cuando ese resource devuelva su opinión al controller, éste dirá si se debe mostrar el error o no.

```
} else if(response.status === 400 && response.data === 'menor') {
  vm.menorDeEdad = 'ERROR';
} else if(response.status === 400 && response.data === 'afternow') {
  vm.afternow = 'ERROR';
```

register.controller.js

Para que el error se muestre en la parte superior con un alert de bootstrap, añadimos las alertas a la parte superior, pero haciendo que no sean visibles.

```
<div class="alert alert-danger" ng-show="vm.menorDeEdad" data-translate="register.messages.error.menorDeEdad">
  <strong>Registration failed!</strong> Eres menor de 16 años
</div>

<div class="alert alert-danger" ng-show="vm.afternow" data-translate="register.messages.error.afternow">
  <strong>Registration failed!</strong> Tu fecha de nacimiento no puede ser posterior a la actual
</div>
```

register.html

Para poder guardar los campos, accederemos a ellos con el ng-model del formulario de registro. En el controller recogeremos todos los datos y los enviaremos al service que guardará el usuario. Cuando esos datos lleguen a la función de crear ese usuario, crearemos a la vez que el usuario de JHipster, un userExt relacionado con los datos introducidos.

```
public User createUser(String login, String password, String firstName, String lastName, String email,
    String imageUrl, String langKey, String phone, LocalDate nacimiento, byte[] imagen, String imagenContentType) {

    User newUser = new User();
    Authority authority = authorityRepository.findOne(AuthoritiesConstants.USER);
    Set<Authority> authorities = new HashSet<>();
    String encryptedPassword = passwordEncoder.encode(password);
    newUser.setLogin(login);
    // new user gets initially a generated password
    newUser.setPassword(encryptedPassword);
    newUser.setFirstName(firstName);
    newUser.setLastName(lastName);
    newUser.setEmail(email);
    newUser.setImageUrl(imageUrl);
    newUser.setLangKey(langKey);
    // new user is not active
    newUser.setActivated(false);
    // new user gets registration key
    newUser.setActivationKey(RandomUtil.generateActivationKey());
    authorities.add(authority);
    newUser.setAuthorities(authorities);
    userRepository.save(newUser);
    log.debug("Created Information for User: {}", newUser);

    //el nuevo usuario con los campos adicionales junto con el usuario de jhipster original
    UserExt newUserExtra = new UserExt();
    newUserExtra.setUser(newUser);
    newUserExtra.setTelefono(phone);
    newUserExtra.setNacimiento(nacimiento);
    newUserExtra.setImagen(imagen);
    newUserExtra.setImagenContentType(imagenContentType);
    userExtRepository.save(newUserExtra);
    log.debug("Created Information for Extra User: {}", newUserExtra);

    return newUser;
}
```

UserService.java

Cuando los datos ya han sido gestionados por el service, ya tendremos creado el usuario y su usuario auxiliar relacionado.

Podemos ver que los datos se han guardado correctamente en la gestión de usuario.

ID	doitApp.userExt.imagen	doitApp.userExt.telefono	Fecha de nacimiento	User	
1	 image/jpeg, 4 542 034 bytes	123456789	27 abr. 1981	julianegpo	Ver Editar Eliminar
3	 image/jpeg, 2 198 906 bytes	123456789	12 sept. 1961	ricard	Ver Editar Eliminar
4	 image/jpeg, 2 840 922 bytes	195685748	9 may. 1993	alan	Ver Editar Eliminar
6	 image/jpeg, 4 048 560 bytes	656958477	27 dic. 1996	crisbarna	Ver Editar Eliminar

user-exts.html

[Commit 82212ef9587da33f36a22e44e9051f6aff90803a](#)

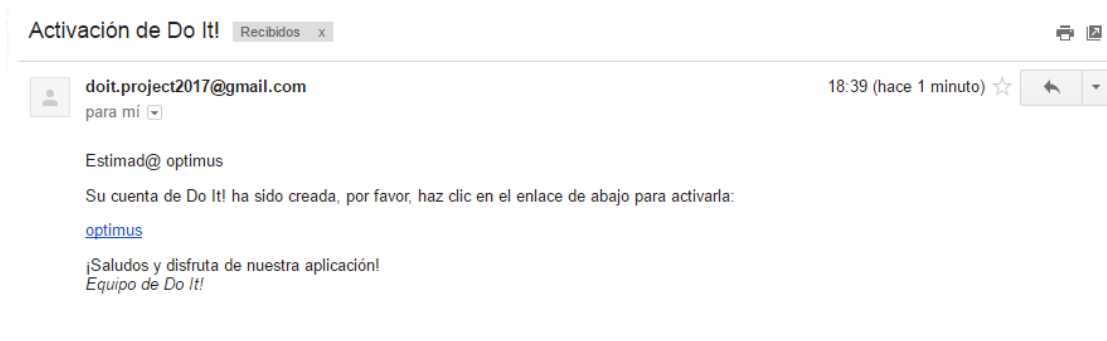
Una vez registrado el usuario en la base de datos queda la segunda parte del registro: la validación mediante correo electrónico. El nuevo usuario recibirá en el correo que haya añadido al registro un mensaje en el que encontrará un link que le permitirá activar su cuenta con solo un click.

```
<title th:text="#{email.activation.title}">JHipster creation</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
  <p th:text="#{email.activation.greeting(${user.login})}">
    Dear
  </p>
  <p th:text="#{email.creation.text1}">
    Your JHipster account has been created, please click on the URL below to access it:
  </p>
  <p>
    <a th:href="@{|${baseUrl}|#/reset/finish?key=${user.resetKey}|"
      th:text="|${user.login}|">login</a>
  </p>
  <p>
    <span th:text="#{email.activation.text2}">Regards, </span>
    <br/>
    <em th:text="#{email.signature}">JHipster.</em>
  </p>
</body>
```

creationEmail.html

```
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title th:text="#{email.activation.title}">JHipster creation</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <p th:text="#{email.activation.greeting(${user.login})}">
      Dear
    </p>
    <p th:text="#{email.creation.text1}">
      Your JHipster account has been created, please click on the URL below to access it:
    </p>
    <p>
      <a th:href="@{|${baseUrl}|#/reset/finish?key=${user.resetKey}|"
        th:text="|${user.login}|">login</a>
    </p>
    <p>
      <span th:text="#{email.activation.text2}">Regards, </span>
      <br/>
      <em th:text="#{email.signature}">JHipster.</em>
    </p>
  </body>
</html>
```

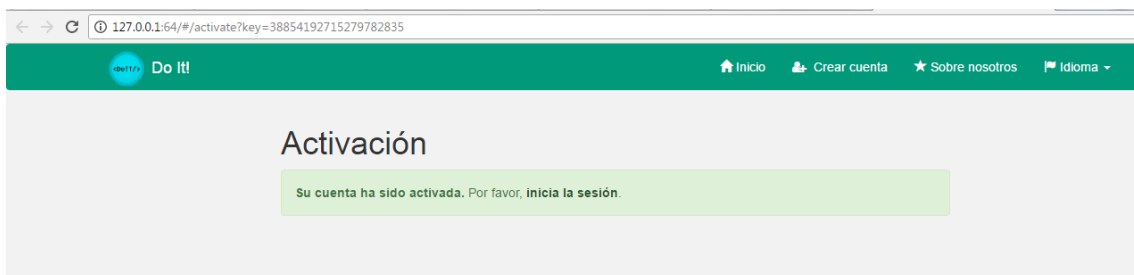
activationEmail.html



Correo de activación

[Commit e182456032ec3a642e4dd9778053e389bb620258](#)

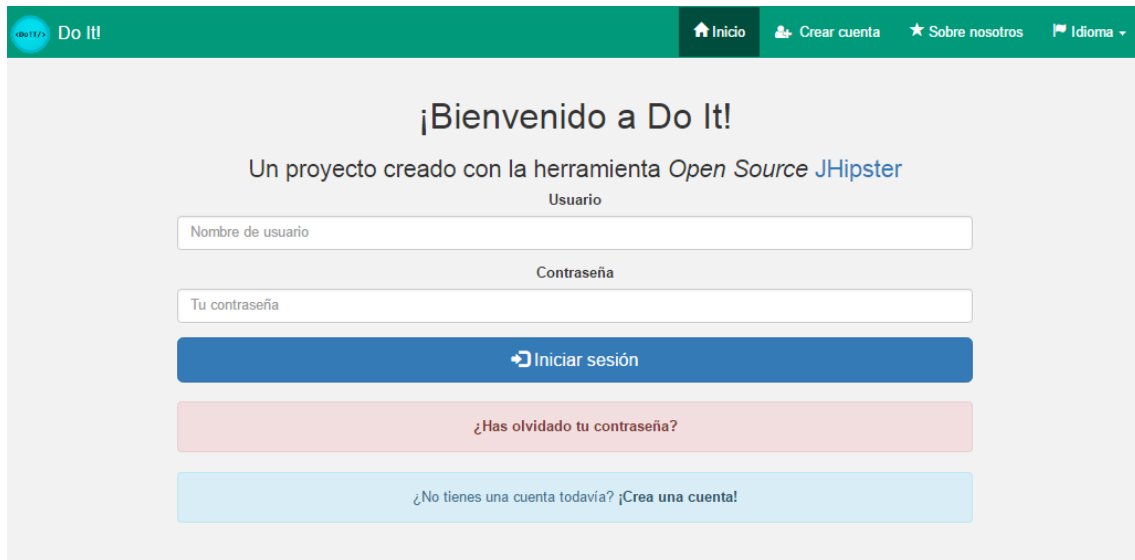
Una vez el usuario haya confirmado el correo, el proceso de registro en Dolt habrá acabado y ya podrá proceder a logearse en la plataforma.



Inicio con el mensaje de activación

2.2. Login

Cuando el usuario accede a la plataforma se encuentra con un formulario de acceso. Para poder usar las funcionalidades del proyecto, este deberá dar sus credenciales de acceso.

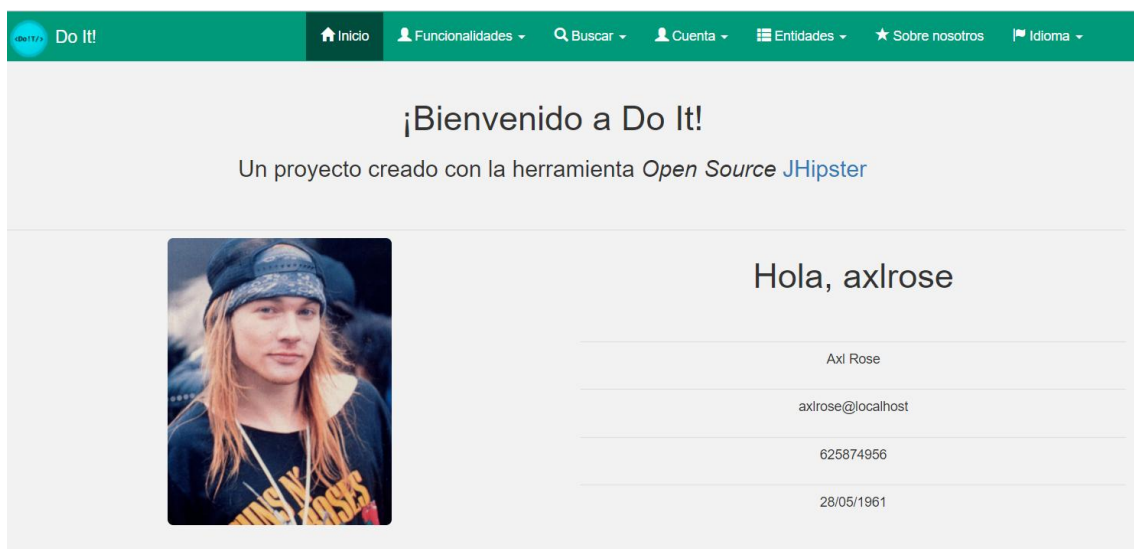


Formulario de acceso

[Commit b7d927aedc539765ecdf705aecf8601a5eaa1845](#)

Una vez accedemos, se nos mostrará una pantalla inicial con nuestros datos y todas las funcionalidades del proyecto a las que podemos acceder.

La página que se muestra de ejemplo no será la pantalla final, ya que todavía necesita mucha edición.

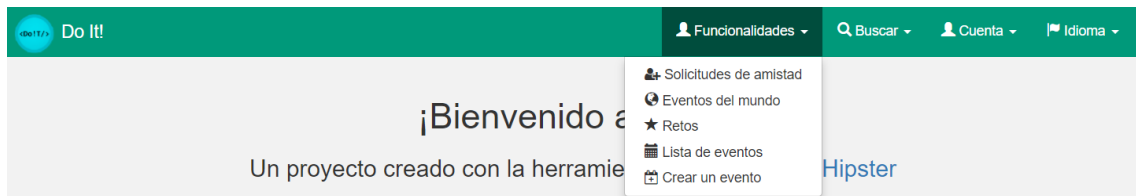


Página principal del usuario

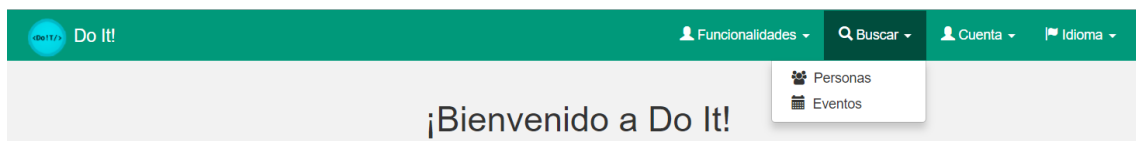
Durante toda la navegación del usuario se encontrará en la parte superior una navbar con las funcionalidades de la plataforma.



Navbar



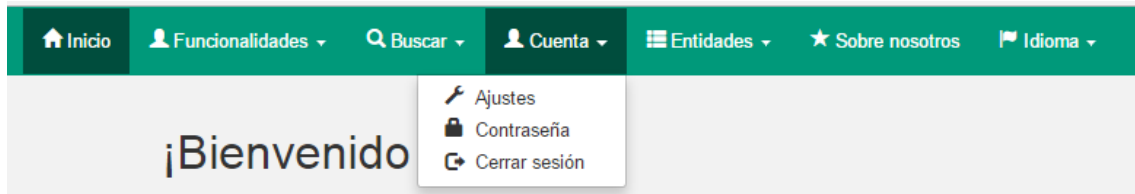
Dropdown de funcionalidades



Dropdown de buscador

2.3. Perfil de usuario

Los usuarios de la plataforma podrán gestionar sus datos con funcionalidades como la de editar su perfil, ver sus datos e incluso eliminar su cuenta.



Dropdown de cuenta

En ajustes, el usuario podrá ver sus datos y, si lo desea, modificarlos. Tales cambios se efectúan al instante. Los cambios que realice el usuario, siempre están bajo control de errores, es decir, se controlará si los campos requeridos están vacíos, si se ha introducido bien el número de teléfono o si la fecha de nacimiento es válida entre otros casos, igual que se hace en el registro.

En la página de editar los datos de la cuenta deben ser agregados todavía los campos del usuario auxiliar que guarda entre otros datos el teléfono y la imagen del usuario. También debe ser añadida la funcionalidad de modificar contraseña ya que actualmente se hace en una pantalla diferente. Por ahora, la funcionalidad está disponible con los datos que gestiona el usuario de JHipster por defecto.

A screenshot of a web form titled 'Ajustes del usuario [optimus]'. The form contains several input fields: 'Nombre' with the value 'Optimus', 'Apellidos' with the value 'Prime', 'Correo electrónico' with the value 'ivangraham@gmail.com', and 'Idioma' with a dropdown menu showing 'Español'. At the bottom of the form is a blue button labeled 'Guardar'.

Editar perfil del usuario

Ajustes del usuario [optimus]

Nombre

Apellidos

Se requiere que ingrese sus apellidos.

Correo electrónico

Se requiere un correo electrónico.

Idioma

Guardar

Control de errores

Contraseña de [optimus]

¡La contraseña ha sido cambiada!

Nueva contraseña

Seguridad de la contraseña: ■ ■ ■ ■ ■ ■

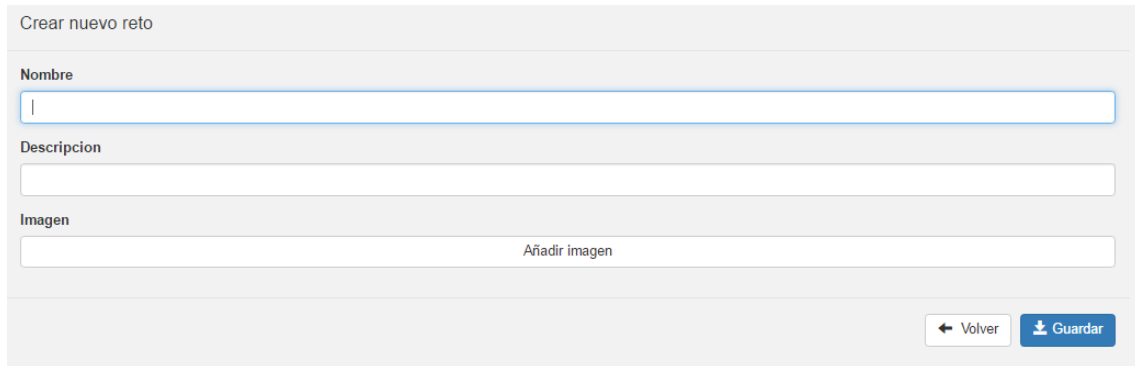
Confirmación de la nueva contraseña

Guardar

Modificación de la contraseña

2.4. Retos

La funcionalidad de retos está destinada principalmente a los administradores del proyecto ya que tan solo ellos tienen la capacidad de acceder al CRUD de esta entidad. Los usuarios de la plataforma sólo podrán ver los retos y participar en ellos.



Crear nuevo reto

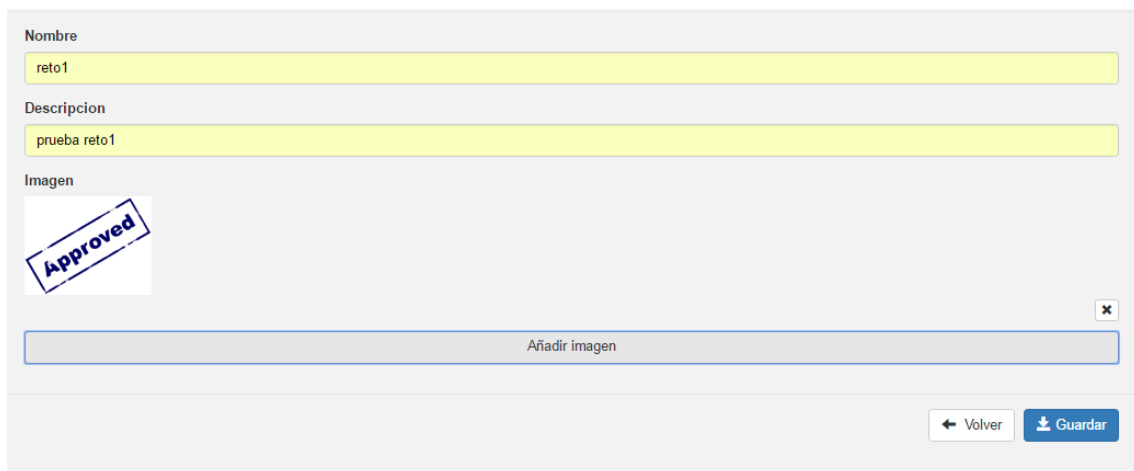
Nombre

Descripción

Imagen

[← Volver](#) [Guardar](#)

Reto dialog



Nombre

Descripción

Imagen

[← Volver](#) [Guardar](#)

Reto dialog con los datos introducidos

Una vez rellenos los datos podríamos ver el evento creado con los botones para hacer el CRUD, como eliminar y editar.

*Retos para un administrador*

En cambio, cuando visitamos la misma página de retos desde un usuario básico, vemos que los botones de edición y borrado ya no existen. Eso es porque se controla en el html que sólo los usuarios con rol de administrador puedan verlos.

*Retos para un usuario común*

Para gestionar que sólo los administradores puedan crear y editar los eventos, debemos hacer que en la lista de retos esos botones tan solo se muestren si el usuario tiene el rol de administrador y no de usuario general.

```

<button type="button" ui-sref="reto-detail.edit({id:vm.reto.id})" class="btn btn-primary"
  ng-if="vm.currentAccount.authorities[0] == 'ROLE_ADMIN' || vm.currentAccount.authorities[1] == 'ROLE_ADMIN'">
  <span class="glyphicon glyphicon-pencil"></span>
  <span class="hidden-sm-down" data-translate="entity.action.edit"> Edit</span>
</button>
<button type="button" ui-sref="reto.delete({id:vm.reto.id})" class="btn btn-danger"
  ng-if="vm.currentAccount.authorities[0] == 'ROLE_ADMIN' || vm.currentAccount.authorities[1] == 'ROLE_ADMIN'">
  <span class="glyphicon glyphicon-ban-circle"></span>
  <span class="hidden-sm-down" data-translate="entity.action.delete"> Delete</span>
</button>
</div>

```

retos-list.html

[Commit 4a25b090b23cee0d4b6d3581be9682e0d5b23790](#)

Recalcar, que los retos se muestran de mayor antigüedad a menor, es decir, que se muestran primero los retos más antiguos y así hasta los más nuevos.

```

@GetMapping("/retosOrder")
@Timed
public List<Reto> getAllRetosOrder() {
  // para invertir Comparator.comparing(Reto::getHoraPublicacion).reversed()
  return retoRepository.findAll().parallelStream().sorted(Comparator.comparing(Reto::getHoraPublicacion)).collect(Collectors.toList());
}

```

La vista de los retos, tiene un diseño responsive de HTML, para su vista en múltiples dispositivos.

[Commit d1f8a9a422d7224c73966a6f1d0c0b96cfbd43d6](#)

[Commit 4cbd1dafde06dc837b58aeceb4102b95d5a1a5d9](#)

2.5. Eventos

En Dolt, los usuarios también podrán crear y gestionar eventos. La creación de estos eventos se basa en un formulario con una serie de campos en el que encuentran, entre otras cosas, un input para introducir la dirección del evento, que se mostrará marcada en un mapa que hay en la parte inferior del formulario.


Crear nuevo evento

Nombre

Descripción

Dirección


☐ **Stucom Centre d'Estudis** Calle de Pelayo, Barcelona, España
☐ **StuComm B.V.** Hoofdveste, Houten, Países Bajos



Fecha del evento

evento-dialog.html

Dirección



evento-dialog.html

Este input se autocompleta de forma automática gracias al uso del módulo ng-maps importado al proyecto, lo que hará que se guarde automáticamente la latitud y longitud de la dirección en la base de datos y se pueda mostrar en el mapa.

```
angular
    .module('doitApp', [
        'ngStorage',
        'tmh.dynamicLocale',
        'pascalprecht.translate',
        'ngResource',
        'ngCookies',
        'ngAria',
        'ngCacheBuster',
        'ngFileUpload',
        'ui.bootstrap',
        'ui.bootstrap.datetimepicker',
        'ui.router',
        'infinite-scroll',
        // jhipster-needle-angularjs-add-module JHipster will add new module here
        'angular-loading-bar',
        'ngMap'
    ])
    .run(run);
```

app.module.js

Este módulo se conecta a la API de Google Maps para recoger los datos solicitados como serían las direcciones para autocompletar o las coordenadas.

[Commit c1ec751a542ea1be8036f1390269fe40495a9277](#)

[Commit 533780c743e6549d20eb4e5e098663624e61ac1c](#)

En las siguientes capturas mostramos el controller específico para el mapa del registro de eventos, en este controller hay diferentes funciones encargadas de mostrar y recoger los valores del mapa.

```
vm.placeChanged = function() {
    vm.place = this.getPlace();
    console.log('location', vm.place.geometry.location);
    vm.map.setCenter(vm.place.geometry.location);
    vm.evento.latitud = vm.place.geometry.location.lat();
    vm.evento.longitud = vm.place.geometry.location.lng();
    vm.map.setZoom(15);
}

NgMap.getMap().then(function(map) {
    vm.map = map;
    vm.map.setZoom(6);
    vm.map.setCenter(new google.maps.LatLng(40.4378698, -3.8196217));
});

setTimeout(function () {
    angular.element('.form-group:eq(1)>input').focus();
});
```

evento-dialog.controller.js

Al acabar con el registro del evento, se muestran todos los eventos creados en la plataforma al usuario y él podrá elegir al cual apuntarse.

Eventos

¡Aquí puedes ver una lista de eventos que han creado otros usuarios!

Administrador	Nombre	Descripción	Creado	Dirección	Fecha del evento	Reto
admin	test1	asdasdasdasdas	1 may. 2017 13:32:00	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	1 may. 2017 13:31:57	+ Apuntarse ✎
admin	prueba2	adasdsadad	4 may. 2017 18:45:19	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	4 may. 2017 18:45:17	+ Apuntarse ✎
admin	test1	ASsSsAS	8 may. 2017 20:16:18	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	8 may. 2017 20:16:17	+ Apuntarse ✎
admin	Prueba Prueba	evento no apuntado	15 may. 2017 17:09:42	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	15 may. 2017 17:09:40	+ Apuntarse ✎
admin	234234324	weweawe	15 may. 2017 17:19:21	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	15 may. 2017 17:19:20	+ Apuntarse ✎

[eventos.html](#)

Destacar que la funcionalidad de apuntarse a un evento, está controlada en todo momento, para no mostrar eventos repetidos, que el usuario no se pueda apuntar a un evento dos veces. Automáticamente cuando un usuario se apunta a un evento, el botón de apuntarse desaparece y aparece el de invitar amigos a ese evento. Eso controla que ese evento está guardado en la lista de eventos de ese usuario.

Eventos

¡Aquí puedes ver una lista de eventos que han creado otros usuarios!

Administrador	Nombre	Descripción	Creado	Dirección	Fecha del evento	Reto
admin	test1	asdasdasdasdas	1 may. 2017 13:32:00	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	1 may. 2017 13:31:57	+ Invitar amigos ✎
admin	prueba2	adasdsadad	4 may. 2017 18:45:19	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	4 may. 2017 18:45:17	+ Invitar amigos ✎
admin	test1	ASsSsAS	8 may. 2017 20:16:18	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	8 may. 2017 20:16:17	+ Apuntarse ✎
admin	Prueba Prueba	evento no apuntado	15 may. 2017 17:09:42	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	15 may. 2017 17:09:40	+ Apuntarse ✎
admin	234234324	weweawe	15 may. 2017 17:19:21	Stucom Centre d'Estudis, Calle de Pelayo, Barcelona, España	15 may. 2017 17:19:20	+ Apuntarse ✎

[eventos.html](#)

Para controlar el funcionamiento de apuntarse, recogemos el id del evento al cual quiere apuntarse ese usuario, al acabar de recoger el id, volvemos a recargar la lista de los eventos.

```
vm.apuntarse = function (id) {
    InvitacionEvento.participar({'id': id}, {});
    console.log(id);
    $state.go('eventos', null, {reload: 'eventos'});
}
```

[evento.controller.js](#)

[Commit 57eec1f257e130ef9d03765005729692cb9e7b63](#)

En el back-end, recogemos este id del evento, y lo sumamos al id del usuario logeado, en ese momento vamos recorriendo si dentro de los campos de evento, para saber si el id del usuario logeado coincide con algún id de los campos mencionados.

```
@PostMapping("/invitacion-eventos/{id}/apuntarse")
@Timed
public ResponseEntity<InvitacionEvento> apuntarse(@PathVariable Long id) throws URISyntaxException {
    // EL ID ES DEL EVENTO
    //tenemos el usuario logeado
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    //tenemos el evento clicado por el usuario logeado
    Evento evento = eventoRepository.findOne(id);
    List<InvitacionEvento> inviEventoList = invitacionEventoRepository.findEventosSigned(userLogin.getId());

    for(InvitacionEvento inviEvento: inviEventoList){
        if(id.equals(inviEvento.getInvitado().getId()) || id.equals(inviEvento.getMiembroEvento().getId())){
            return ResponseEntity.badRequest().headers(HeaderUtil.
                createFailureAlert(ENTITY_NAME, "invitationexists", "Ya te has apuntado a este evento!")).
                body(null);
        }
    }

    //control de error
    //comprobar que la id_evento no sea null
    //comprobar que la invitacion del evento no sea null

    InvitacionEvento invitacion = new InvitacionEvento();
    invitacion.setHoraInvitacion(ZonedDateTime.now());
    invitacion.setMiembroEvento(userLogin);
    invitacion.setEvento(evento);
    invitacion.setInvitado(userLogin);
    invitacion.setHoraResolucion(ZonedDateTime.now());
    invitacion.setResolucion(true);

    InvitacionEvento result = invitacionEventoRepository.save(invitacion);
    return ResponseEntity.created(new URI("/api/invitacion-eventos/apuntarse" + result.getId()))
        .headers(HeaderUtil.createEntityCreationAlert(ENTITY_NAME, result.getId().toString()))
        .body(result);
}
```

EventoResource.java

En el controller del evento, se controla con dos funciones diferentes, que son las siguientes que se muestran en la siguiente captura.

```
function loadApuntados() {
    InvitacionEvento.eventosApuntado(function (result) {
        vm.eventosApuntado = result;
        vm.searchQuery = null;
        console.log("Apuntado: "+vm.eventosApuntado);
    });
}

function loadNoApuntados() {
    InvitacionEvento.eventosNoApuntado(function (result) {
        vm.eventosNoApuntado = result;
        vm.searchQuery = null;
        console.log("No apuntado: "+vm.eventosNoApuntado);
    });
}
```

evento.controller.js

Estas funciones son controladas directamente desde el back-end, con dos API's programadas específicamente para ese cometido. Estas API's recogen los valores de los arrays de eventos (apuntados y no apuntados) y los procesan en las API's, para devolver los diferentes resultados.

```
@GetMapping("/invitacion-eventos/eventosUsuarioApuntado")
@Timed
public List<InvitacionEvento> getEventosUsuario() {
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    List<InvitacionEvento> invitacionEvento = invitacionEventoRepository.findEventosSigned(userLogin.getId());
    //System.out.println(invitacionEvento);
    return invitacionEvento;
    //return ResponseUtil.wrapOrNotFound(Optional.ofNullable(invitacionEvento));
}

@GetMapping("/invitacion-eventos/eventosUsuarioNoApuntado")
@Timed
public List<Evento> getEventosNoUsuario() {
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    List<Evento> eventos = invitacionEventoRepository.findEventosNotSigned(userLogin.getId());
    return eventos;
    //return ResponseUtil.wrapOrNotFound(Optional.ofNullable(invitacionEventos));
}
```

EventoResource.java


```

    @Query("select invitacionEvento from InvitacionEvento invitacionEvento where " +
        "invitacionEvento.horaResolucion is not null and invitacionEvento.resolucion = true " +
        "and invitacionEvento.invitado.id =:currentUser")
    List<InvitacionEvento> findEventosSigned(@Param("currentUser") Long currentUser);

    @Query("select evento from Evento evento where " +
        "evento not in (select invitacionEvento.evento from InvitacionEvento invitacionEvento where " +
        "invitacionEvento.horaResolucion is not null and invitacionEvento.resolucion = true " +
        "and invitacionEvento.invitado.id =:currentUser)")
    List<Evento> findEventosNotSigned(@Param("currentUser") Long currentUser);

```

EventoRepository.java

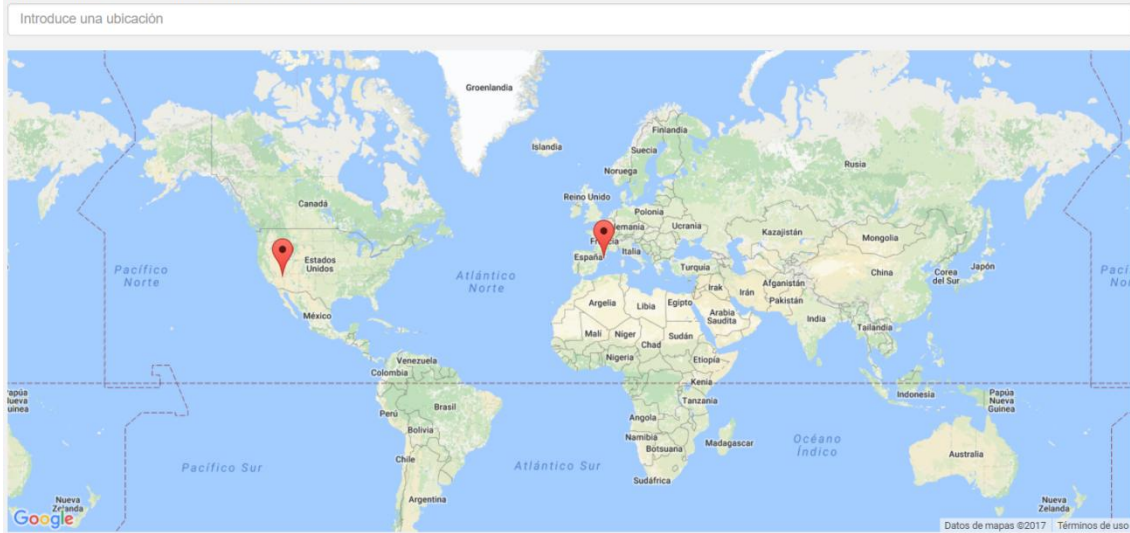
[Commit 35023f57aafef3d4e9c41a980c2216ef37799b34](#)

Los diferentes eventos que vayan creando los usuarios de la plataforma, también pueden ser visualizados en un apartado llamado "Eventos del mundo". Esta funcionalidad complementaría a los eventos, ayuda a los usuarios a buscar eventos a partir de una dirección en concreto.

En la página principal encontraremos un mapa del mundo y un input que también implementa el módulo ng-maps. Gracias a este input podremos buscar los eventos que hay en una zona determinada ya que, según la dirección que introduzcamos, el mapa se centrará en esa zona para mostrar más de cerca qué hay cerca de allí.

Eventos en el mundo

Introduce una dirección para buscar eventos en la zona



Eventos mundo

```

<div class="row">
  <h2 translate="doitApp.evento.world-events.title">World events</h2>
  <div class="form-group">
    <label class="control-label" data-translate="doitApp.evento.buscarDireccion"
      for="field_direccion">Introduce una dirección para buscar eventos en la zona</label>
    <input type="text" class="form-control" name="direccion"
      id="field_direccion" places-auto-complete
      ng-model="vm.evento.direccion" on-place-changed="vm.placeChanged()" />
  </div>
</div>
<div class="row">
  <div class="text-center">
    <ng-map id="mapa" class="map-track"
      disable-default-ui="true"
      style="..."
    >
      <marker ng-repeat="evento in vm.eventos" position="{{(evento.latitud)},{{(evento.longitud)}}">
      </marker>
    </ng-map>
  </div>
</div>

```

eventos-mundo.html

Para que funcione el input debemos establecer la función en el controlador que se realice cuando cambiamos la dirección del input.

```

vm.placeChanged = function() {
  vm.place = this.getPlace();
  console.log('location', vm.place.geometry.location);
  vm.map.setCenter(vm.place.geometry.location);
  vm.evento.latitud = vm.place.geometry.location.lat();
  vm.evento.longitud = vm.place.geometry.location.lng();
  vm.map.setZoom(8);
}

```














map.controller.js

2.6. Buscador de usuarios

Se trata de una funcionalidad que permite la búsqueda de usuarios, creando así un ámbito social entre ellos y la plataforma. Gracias a poder buscarlos, el usuario podrá agregar amigos y ver aquellos que ya tiene agregados en una lista de contactos. Esta lista será utilizada por otras funcionalidades de la plataforma como por ejemplo el envío de solicitudes de amistad. El usuario receptor de esa solicitud de amistad podrá verla en un apartado de la aplicación programado para mostrar las solicitudes recibidas y aceptarlas o rechazarlas.

El buscador permite buscar por diversos campos de la entidad usuario como serían por nombre de usuario, nombre y apellidos reales, correo electrónico, teléfono...

Busca personas por nombre, teléfono, fecha de nacimiento...

	Nombre de usuario	Nombre real	Teléfono	E-mail	Fecha nacimiento	
	ricard	Ricard Huelamo	123456789	ricard@localhost	12/09/1961	 Ya es tu amigo
	julianegpo	Julian Egea	123456789	julianegpo@localhost	27/04/1981	 Solicitud enviada
	crisbama	Cristina Rodriguez	656958477	cristinarodriguez.palmer@gmail.com	27/12/1996	 Solicitud pendiente
	ivangraherm	Ivan Grana	987654321	ivangrana@localhost	10/08/1994	 Agregar a amigos
	alan	Alan Valve	195685748	alan@localhost	09/05/1993	 Agregar a amigos
	alex	Alex Palmer	987654325	alexpalmer@localhost	15/11/1995	 Agregar a amigos
	optimus	optimus optimus	656958477	jotabono92@gmail.com	18/05/1961	 Agregar a amigos

user-ext-search

El buscador ha sido programado usando un filter de angular en el que establecemos una variable ng-model al input que se le aplicará al ng-repeat que muestra los usuarios. La directiva filter hará que solo los usuarios que tengan algún dato en su lista de información igual al introducido sean mostrados.

```

<div class="input-group">
  <span class="input-group-addon"> <span class="glyphicon glyphicon-search"></span> </span>
  <input type="text" class="form-control" ng-model="criteria"
    placeholder="{{ 'global.form.search.placeholder' | translate }}">
</div>

<tbody>
  <!-- LOS AMIGOS -->
  <tr ng-repeat="friend in vm.accepted|filter:criteria" ng-if="vm.currentAccount.id!=user.user.id">
    ...
  </tr>
</tbody>

```

user-ext-search.html

[Commit 325fb8c051e44d2269640fefad0551143c118db1](#)

Para que los diferentes usuarios se muestren con botones diferentes hemos hecho peticiones diversas tanto para los que ya son amigos, los que tienen la solicitud enviada, las solicitudes pendientes (tú las tienes que aceptar o rechazar) y el resto de usuarios que no tienen solicitud ni son amigos.

En el service hemos definido las diferentes peticiones que haremos al servidor y después los métodos desde los que las usaremos en el controller.

```
return $resource(resourceUrl, {}, {
  'query': {method: 'GET', isArray: true},
  'getAllByCurrentUser': {method: 'GET', isArray: true, url: 'api/amistades'},
  'getSolicitudesAceptadas': {method: 'GET', isArray: true, url: 'api/usersSolAceptadas'},
  'getSolicitudesPendientes': {method: 'GET', isArray: true, url: 'api/usersSolPendientes'},
  'getSolicitudesPendientesEmisor': {method: 'GET', isArray: true, url: 'api/usersSolPendientesEmisor'},
  'getSolicitudesPendientesReceptor': {method: 'GET', isArray: true, url: 'api/usersSolPendientesReceptor'},
```

amistad.service.js

```
// users amigos al actual
Amistad.getSolicitudesAceptadas(function (result) {
  vm.accepted = result;
});
```

user-ext.controller.js usuarios amigos

```
// resto de usuarios
UserExt.query(function(result) {
  vm.userExts = result;
  vm.otherUsers = result;
  vm.searchQuery = null;
  comprobarNoAmigos();
});
```

user-ext.controller.js usuarios no amigos

```
// solicitud pendiente --> eres el emisor
Amistad.getSolicitudesPendientesEmisor(function(result) {
  vm.resultado = result;
  for(var i = 0; i < vm.resultado.length; i++){
    if(vm.resultado[i].user != null){
      vm.pendingEmisor.push(vm.resultado[i]);
    }
  }
});

// solicitud pendiente --> eres el receptor
Amistad.getSolicitudesPendientesReceptor(function(result) {
  vm.resultado = result;
  for(var i = 0; i < vm.resultado.length; i++){
    if(vm.resultado[i].user != null){
      vm.pendingReceptor.push(vm.resultado[i]);
    }
  }
});
```

user-ext.controller.js usuarios con solicitudes pendientes



Una vez tenemos los datos cargados hacemos los diferentes ng-repeat para mostrar los tipos de contactos y los botones determinados para cada uno.

Como en todos será muy parecido y más adelante se explicará mejor esta funcionalidad, expondremos sólo el caso de un tipo de amistad, el de solicitudes pendientes de aceptar.

```
<!-- PENDING EMISOR -->
<tr ng-repeat="pendingFriend in vm.pendingEmisor|filter:criteria" ng-if="vm.currentAccount.id!=user.user.id">
  <td...>
  <td><span><a ui-sref="user-ext-detail({id:pendingFriend.id})">{{pendingFriend.user.login}}</a></span></td>
  <td><span>{{pendingFriend.user.firstName}} {{pendingFriend.user.lastName}}</span></td>
  <td><span>{{pendingFriend.telefono}}</span></td>
  <td><span>{{pendingFriend.user.email}}</span></td>
  <td><span>{{pendingFriend.nacimiento | date:'dd/MM/yyyy'}}</span></td>
  <td class="text-right">
    <div class="btn-group">
      <button type="submit" class="btn btn-default" disabled="true">
        <span class="glyphicon glyphicon-user" aria-hidden="true"></span>
        <span translate="doitApp.userExt.sended"></span>
      </button>
    </div>
  </td>
</tr>
```

user-ext-search.html

En este caso tenemos un botón deshabilitado que nos dice que la solicitud ha sido enviada.

	julianegpo	Julian Egea	123456789	julianegpo@localhost	27/04/1981	 Solicitud enviada
---	------------	-------------	-----------	----------------------	------------	---

user-ext-search

2.7. Solicitudes de amistad

Esta funcionalidad permite a los usuarios enviar solicitudes de amistad a los diferentes usuarios de la plataforma, mediante el buscador anteriormente explicado. El sistema de funcionamiento es similar al de otras redes sociales que están por Internet actualmente. Controlar en todo momento que id de usuario será el receptor de dicha solicitud.

Para enviar la solicitud seleccionamos el id del que será el receptor de esta para que el back-end gestione la creación de la solicitud.

```
<div class="btn-group">
  <button type="submit"
    ng-click="vm.sendFriendRequest(user.user.id)"
    class="btn btn-primary">
    <i class="fa fa-user-plus" aria-hidden="true"></i>
    <span translate="doitApp.userExt.add"></span>
  </button>
</div>
```

user-ext-search.html

Ese id será enviado a un método del controller que usará la conexión entre back-end y front-end creada en el service para que se cree la solicitud en back-end.

```
vm.sendFriendRequest = function(id) {
  Amistad.sendFriendRequest({'id': id}, {});
  $state.go('user-search', null, {reload: 'user-search'});
}
```

user-ext.controller.js

```
function Amistad($resource, DateUtils) {
  var resourceUrl = 'api/amistads/:id';

  return $resource(resourceUrl, {}, {
    'sendFriendRequest': {method: 'POST', isArray: false, url: 'api/amistad/:id/emisor'},
  });
}
```

user-ext.service.js

El service crea una conexión a la url “api/amistad/:id/emisor” que referencia a un método creado en el back-end, el que será el encargado de crear esa solicitud. Este método de back-end recibirá el id del receptor y cogerá el del emisor desde servidor ya que será el usuario que está registrado en ese momento.

```
// Enviar solicitud amistad desde buscar amigo
@PostMapping("/amistad/{id}/emisor")
@Timed
public ResponseEntity<Amistad> sendFriendRequest(@PathVariable Long id) throws URISyntaxException {
    log.debug("REST request to save Amistad : {}");
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    User userReceptor = userRepository.findOne(id);
    List<Amistad> amigos = amistadRepository.findAllFriends(userLogin.getId());
    for (Amistad amigo: amigos) {
        if(id.equals(amigo.getReceptor().getId()) || id.equals(amigo.getEmisor().getId())){
            return ResponseEntity.badRequest().headers(HeaderUtil.createFailureAlert(ENTITY_NAME, "friendexists", defaultMessage: "Este usuario ya es tu amigo!")).body(null);
        }
    }
    Amistad result = amistadRepository.save(
        new Amistad(ZonedDateTime.now(), userLogin, userReceptor)
    );
    return ResponseEntity.created(new URI("str: "/api/amistads/" + result.getId()))
        .headers(HeaderUtil.createEntityCreationAlert(ENTITY_NAME, result.getId().toString()))
        .body(result);
}
```

AmistadResource.java

Una vez hecho, la solicitud queda creada, pero la amistad todavía no se genera ya que hasta que el receptor no acepta dicha solicitud, la amistad no está asegurada.

Para ello, también se cuenta con dichas funcionalidades, el control de aceptar o denegar solicitudes de amistad. Todas las solicitudes de amistad que tenga ese usuario, se le irán mostrando en una lista, que se irá actualizando a partir de la decisión del usuario receptor de tales.

Solicitudes de amistad

Aquí se muestran las solicitudes de amistad que te han enviado otros usuarios.

admin

✓

✗

Solicitudes de amistad

Cada botón tiene una función esperando una respuesta, es decir, que dependiendo de la elección del usuario de aceptar o denegar, se enviará un resultado u otro.

```

<div class="btn-group flex-btn-group-container">
  <button class="btn btn-primary"
    ng-click="vm.acceptFriend(amistad.id)">
    <span class="glyphicon glyphicon-ok"></span>
  </button>
  <button class="btn btn-danger"
    ng-click="vm.denyFriend(amistad.id)">
    <span class="glyphicon glyphicon-remove"></span>
  </button>
</div>

```

solicitudes-amistad.html

Las dos funciones programadas en el controller, esperan como respuesta un id para poder actualizar el estado de esa solicitud según si la quiere aceptar o no. Para que los cambios se generen también en back-end, creamos en el service unas conexiones con éste.

```

'accept': { method: 'PUT', url: 'api/amistads/:id/accept' },
'deny': { method: 'PUT', url: 'api/amistads/:id/deny' },

```

amistad.service.js

```

vm.acceptFriend = function(id) {
  Amistad.accept({'id':id}, {});
  $state.go('amistades', null, {reload: 'amistades'});
}

vm.denyFriend = function(id) {
  Amistad.deny({'id':id}, {});
  $state.go('amistades', null, {reload: 'amistades'});
}

```

amistad.controller.js

En cuanto a la programación de las API's en el back-end, al recibir el resultado en forma de id, estas se encargan de actualizar los valores de la solicitud y mandarlos a la base de datos, para que quede constancia.

```

@PutMapping("/amistads/{id}/accept")
@Timed
public ResponseEntity<Amistad> accept(@PathVariable Long id) {
    log.debug("REST request to update Amistad : {}", id);
    if (id == null) {
        return ResponseEntity.badRequest().headers(HeaderUtil.createFailureAlert(
            ENTITY_NAME, errorKey: "noexists", defaultMessage: "La amistad no existe")).body(t null);
    }
    Amistad amistad = amistadRepository.findById(id);
    amistad.setHoraRespuesta(ZonedDateTime.now());
    amistad.setAceptada(true);
    Amistad result = amistadRepository.save(amistad);
    return ResponseEntity.ok()
        .headers(HeaderUtil.createEntityUpdateAlert(ENTITY_NAME, amistad.getId().toString()))
        .body(result);
}

```

AmistadResource.java

```


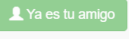

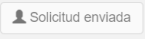



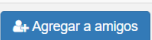
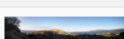
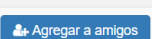

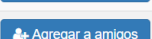
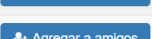
@PutMapping("/amistads/{id}/deny")
@Timed
public ResponseEntity<Amistad> deny(@PathVariable Long id){
    log.debug("REST request to update Amistad : {}", id);
    if (id == null) {
        return ResponseEntity.badRequest().headers(HeaderUtil.createFailureAlert(
            ENTITY_NAME, errorKey: "noexists", defaultMessage: "La amistad no existe")).body(t null);
    }
    Amistad amistad = amistadRepository.findById(id);
    amistad.setHoraRespuesta(ZonedDateTime.now());
    amistad.setAceptada(false);
    Amistad result = amistadRepository.save(amistad);
    return ResponseEntity.ok()
        .headers(HeaderUtil.createEntityUpdateAlert(ENTITY_NAME, amistad.getId().toString()))
        .body(result);
}

```

AmistadResource.java

[Commit 205ef3c066ea78be5f29a500043034c4ca57ef31](#)

Recalcar que las funcionalidades anteriormente comentadas relacionadas con las solicitudes de amistad, controlan y muestran en todo momento, el estado de éstas, es decir, se le informa al usuario, de si ya ha enviado una solicitud a un usuario y el estado de esta, si un usuario ya es su amigo, si otro usuario no es su amigo o de si tiene solicitudes de otros usuarios.

<input type="text" value="Busca personas por nombre, teléfono, fecha de nacimiento..."/>						
	Nombre de usuario	Nombre real	Teléfono	E-mail	Fecha nacimiento	
	ricard	Ricard Huelamo	123456789	ricard@localhost	12/09/1961	
	julianegpo	Julian Egea	123456789	julianegea@localhost	27/04/1981	
	crisbarna	Cristina Rodriguez	656958477	cristinarodriguez.palmer@gmail.com	27/12/1996	
	ivangraherm	Ivan Grana	987654321	ivangrana@localhost	10/08/1994	
	alan	Alan Valve	195685748	alan@localhost	09/05/1993	
	alex	Alex Palmer	987654325	alexpalmer@localhost	15/11/1995	
	optimus	optimus optimus	656958477	jotabono92@gmail.com	18/05/1961	

Para controlar y mostrar los amigos de dicho usuario, se creó una API específica que controlase los diferentes casos de id, tanto de emisor como receptor. El resultado de dicha API, se va guardando en una lista, donde sólo habrá los amigos de ese usuario en concreto.

```
@Query("select amistad from Amistad amistad where " +
    "amistad.horaRespuesta is not null and amistad.aceptada = true " +
    "and (amistad.receptor.id =:currentUser or amistad.emisor.id =:currentUser)")
List<Amistad> findFriendsAcceptedRequest(@Param("currentUser") Long currentUser);
```

AmistadRepository.java

```
//get solicitudes aceptadas (amigos)
@GetMapping("/usersSolAceptadas")
@Timed
public List<UserExt> getFriendsAccepted() throws URISyntaxException {
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    List<UserExt> amigos = amistadRepository.findFriendsAcceptedRequest(userLogin.getId());
    amigos.parallelStream().
        map(amistad -> {
            UserExt user;
            if(amistad.getEmisor().equals(userLogin)){
                user = userExtRepository.findById(amistad.getReceptor().getId());
            }else{
                user = userExtRepository.findById(amistad.getEmisor().getId());
            }
            return user;
        })
        .collect(Collectors.toList());

    return amigos;
}
```

AmistadResource.java

Otro caso a controlar eran las solicitudes de amistad con estado pendientes, es decir, que se habían enviado pero el usuario receptor de esta no la había aceptado ni denegado. Para eso se crearon dos nuevas API's, que controlasen los estados pendientes.

Estas API's controlaban el estado, mirando si el usuario era el emisor o receptor de esta, por eso había que ir controlando el id de usuario al cual iba esa solicitud.

```
// todas las amistades aceptadas donde el emisor y receptor es el user login --> QUE ESTÁN PENDIENTES
@Query("select amistad from Amistad amistad where " +
    "amistad.horaRespuesta is null and amistad.aceptada is null " +
    "and (amistad.receptor.id =:currentUser or amistad.emisor.id =:currentUser)")
List<Amistad> findFriendsPendingRequest(@Param("currentUser") Long currentUser);
```

AmistadRepository.java


```
// get solicitudes pendientes el current user es EMISOR --> enviaré los receptores
@GetMapping("/usersSolPendientesEmisor")
@Timed
public List<UserExt> getFriendsPendingRequestEmisor() throws URISyntaxException {
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    List<UserExt> amigos = amistadRepository.findFriendsPendingRequest(userLogin.getId()).
        parallelStream().
        map(amistad -> {
            if(amistad.getEmisor().equals(userLogin)){
                return userExtRepository.findByUserID(amistad.getReceptor().getId());
            }else return null;
        })
        .collect(Collectors.toList());

    return amigos;
}
```

AmistadResource.java

```
// get solicitudes pendientes el current user es RECEPTOR --> enviaré los emisores
@GetMapping("/usersSolPendientesReceptor")
@Timed
public List<UserExt> getFriendsPendingRequestReceptor() throws URISyntaxException {
    User userLogin = userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get();
    List<UserExt> amigos = amistadRepository.findFriendsPendingRequest(userLogin.getId()).
        parallelStream().
        map(amistad -> {
            if(amistad.getReceptor().equals(userLogin)){
                return userExtRepository.findByUserID(amistad.getEmisor().getId());
            }else return null;
        })
        .collect(Collectors.toList());

    return amigos;
}
```

AmistadResource.java

Las siguientes capturas, muestran la programación a nivel de front-end, de todas las API's explicadas anteriormente, las cuales tienen sus funciones en el front-end, las cuales atacan al service y controller, para ir capturando y mostrando los datos necesarios al usuario.

```
return $resource(resourceUrl, {}, {
    'query': {method: 'GET', isArray: true},
    'getAllByCurrentUser': {method: 'GET', isArray: true, url: 'api/amistades'},
    'getSolicitudesAceptadas': {method: 'GET', isArray: true, url: 'api/usersSolAceptadas'},
    'getSolicitudesPendientesEmisor': {method: 'GET', isArray: true, url: 'api/usersSolPendientesEmisor'},
    'getSolicitudesPendientesReceptor': {method: 'GET', isArray: true, url: 'api/usersSolPendientesReceptor'},

```

```

function loadAll() {
    // users amigos al actual
    Amistad.getSolicitudesAceptadas(function (result) {
        vm.accepted = result;
    });

    // solicitud pendiente --> eres el emisor
    Amistad.getSolicitudesPendientesEmisor(function(result) {
        vm.resultado = result;
        for(var i = 0; i < vm.resultado.length; i++){
            if(vm.resultado[i].user != null){
                vm.pendingEmisor.push(vm.resultado[i]);
            }
        }
    });

    // solicitud pendiente --> eres el receptor
    Amistad.getSolicitudesPendientesReceptor(function(result) {
        vm.resultado = result;
        for(var i = 0; i < vm.resultado.length; i++){
            if(vm.resultado[i].user != null){
                vm.pendingReceptor.push(vm.resultado[i]);
            }
        }
    });

    // resto de usuarios
    UserExt.query(function(result) {
        vm.userExts = result;
        vm.otherUsers = result;
        vm.searchQuery = null;
        comprobarNoAmigos();
    });
}

```

user-ext.controller.js

3. RESULTADOS

Como resultados referidos al proyecto, comentar que hay ciertos aspectos que podrían plantearse de diferentes maneras. Hay funcionalidades que fueron planteadas inicialmente de una manera y con unos usos determinados que finalmente no han abarcado tanto como esperábamos.

La aplicación en sí está bien planteada a nivel conceptual y bien organizada a nivel de programación, pero cuando se plantea un proyecto como este, se ve que las ideas iniciales tanto de orden como de dirección no se mantienen y que, durante el transcurso del proyecto, hemos visto que muchas de esas ideas no era posible llevarlas a cabo de la manera decidida inicialmente.

Vamos a adjuntar un [vídeo](#) que mostrará el resultado final de la aplicación tanto en diseño como en funcionalidades.

El recorrido que seguirá el vídeo es el de un nuevo usuario al entrar a la plataforma, empezando por el registro, la activación de la cuenta, login y las diferentes funcionalidades del proyecto.

En el vídeo podemos observar que hay varios aspectos a mejorar como el tiempo de depuración del código, las cargas de las páginas con imágenes... Otro ejemplo de cosas a mejorar en el código se ve en el reload de las páginas, ya que a nivel de programación está bien implementada, pero por falta de conocimientos y tiempo, su funcionamiento no es del todo preciso. Se puede observar esto ya que cuando el usuario envía las solicitudes de amistad, la vista tarda en volver a ser cargada.

Otro ejemplo sería la navegación de la plataforma. El cambio entre vistas de no es todo lo fluido y rápido que debería ser.

4. CONCLUSIONES Y LÍNEAS FUTURAS

Nuestro proyecto no ha alcanzado los objetivos máximos marcados, pero sí que ha cubierto de manera satisfactoria las funcionalidades mínimas que estaban establecidas.

Comentar que hay funcionalidades que no han podido implementarse ya sea por falta de tiempo, falta de recursos o falta de conocimiento.

Una de esas funcionalidades, la cual nos hubiese gustado implementar en el proyecto, ya que llegó a funcionar en un proyecto piloto, era el chat. La idea del chat era que los propios usuarios de la plataforma, pudiesen chatear entre ellos, de manera individual o en grupo, para organizar diferentes ideas dentro de la plataforma.

[IvanGraHer -> Chat-Sample-Dolt](#)

Otra de las funcionalidades que ha faltado por añadir es el buscador en Hibernate5, ya que pese a haber estado creado el back-end, no hubo tiempo suficiente para implementar el front-end y gestionar el funcionamiento. Además, el buscador estaba pensado tanto para usuarios como para eventos y era mucho más funcional que el finalmente aplicado.

Si hubiese habido más tiempo o una mejor organización en el grupo, una de las siguientes funcionalidades a añadir sería el chat ya que realmente era una de las opciones más potentes de la plataforma. También se haría una mejora de código y programación para hacerlo más óptimo ya que muchas veces esa optimización se ha perdido por dedicarle menos tiempo.

A nivel de programación de front-end, se podría hablar de un diseño nuevo y mejorado, utilizando mejor las tecnologías de las que disponíamos y haciendo que las funcionalidades fueran más vistosas y llamativas.

En cuanto a back-end, una reestructuración de algunas entidades, que fueron planteadas de una forma para darse un uso en concreto y al final por diversos motivos, no se han podido implementar en el proyecto.

5. WEBGRAFÍA Y BIBLIOGRAFÍA

Stackoverflow

JHipster

Ng-maps

Bootstrap

W3schools

AngularJS

Google developers API Google Maps