UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

ArreglamiCiudad: crowdsourcing para el mantenimiento de las ciudades

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Pablo Almenar Margarit

Tutor: Pietro Manzoni

2014 - 2015

Resumen

En este documento se introduce un nuevo software destinado a satisfacer la necesidad de un sistema eficaz, centralizado, rápido y de fácil acceso para gestionar las incidencias y averías de una ciudad y su reparación, basado en la colaboración ciudadana.

También se detallará el diseño, estructura y funcionalidad del sistema, así como las metodologías y herramientas usadas para su desarrollo, los patrones de diseño que se han implementado, y los *frameworks*, tecnologías y lenguajes de programación integrados.

Por último, se explicarán los métodos y tecnologías que se han utilizado para las pruebas y se incluirán detalles de implementación y uso, así como la forma de acceso a una demostración en vivo del proyecto.

Palabras clave: Web service, Android, Crowdsourcing.

Abstract

In this document we will introduce a new software meant to meet the need of an efficient, centralized, fast and easy-to-use system for the management of a city's incidents or breakdowns and their reparation, that relies on crowdsourcing.

We will also go into detail about the design, structure and functionality of the system, as well as the methodologies and tools used for its development, the design patters that have been implemented, and the integrated frameworks, technologies and programming languages.

Finally, we will explain the methods and technologies that have been used for testing and include implementation and usage details, as well as how to access a live demo.

Keywords : Web service, Android, Crowdsourcing.

Tabla de contenidos

1. Introducción

En las ciudades actuales, nos encontramos que, en muchas ocasiones en las que se produce un accidente o imprevisto, surgen dificultades en cuanto a su reporte y gestión por parte de las autoridades o personal competente.

Tecnologías actuales como los teléfonos móviles o Internet facilitan dicha tarea y ayudan a mejorar la situación, pero todavía no existe una plataforma unificada destinada a la gestión de dichas situaciones.

Utilizando una plataforma de este tipo, basada en la colaboración ciudadana y en la infraestructura y medios que la facilitan, como la conectividad de los teléfonos móviles actuales, se podrían reportar y gestionar dichas situaciones de una manera mucho más fiable.

Ante esta necesidad, se plantea la creación de ArreglamiCiudad, una plataforma basada en *crowdsourcing* para el mantenimiento y gestión de las ciudades.

Con ArreglamiCiudad, todos los ciudadanos podrán ver las incidencias de su ciudad, así como reportar ellos mismos incidencias que hayan descubierto desde cualquier sitio con su *smartphone*, o confirmar la veracidad de un reporte realizado por otro usuario, añadiendo opcionalmente un comentario. Posteriormente, el personal autorizado confirmará que en efecto la incidencia es real y desde el sistema informará a los ciudadanos y gestionará la reparación.

Para tal fin, se ha creado un servicio web para la gestión informatizada de las alertas de la ciudad. La aplicación consistirá en el servicio web que gestionará y almacenará la información de usuarios e incidencias, una aplicación móvil para realizar los reportes, y un portal desde el que visualizar más información y realizar gestiones.

Los usuarios podrán ver todas las alertas de su ciudad, su localización, descripción y una foto, así como colaborar añadiendo información a una incidencia ya reportada. El personal competente podrá confirmar la veracidad de los reportes y ordenar su gestión, así como eliminarlos del sistema cuando ya se hayan gestionado correctamente.

1.2 Antecedentes

A prop teu Cornellà

A prop teu Cornellà es una aplicación diseñada para proveer una via de comunicación con el ayuntamiento más cercana, ágil y eficaz. Se utiliza para reportar incidencias así como plantear quejas, sugerencias o propuestas.

A prop teu aprovecha nuevas tecnologías como Facebook, twitter, correo electrónico o SMS, pero, a diferencia de ArreglamiCiudad, no cuenta con una aplicación móvil para facilitar el reporte de incidencias.

Para reportar una incidencia en A prop teu es necesario indicar un canal de comunicación como una red social, correo electrónico o SMS y rellenar un reporte con lugar y descripción de la incidencia, y garantiza un tiempo de respuesta inferior a 24 horas.

Valladolid en tu mano

'Valladolid en tu mano' es una aplicación para smartphone oficial del ayuntamiento de Valladolid para reportar incidencias o sugerencias desde dispositivos móviles. Está diseñada para crear una vía sencilla y eficaz de colaboración con el ayuntamiento de Valladolid.

'Valladolid en tu mano' ofrece a la ciudadanía la posibilidad de comunicar sugerencias e incidencias sobre la vía pública, mobiliario urbano, infraestructuras, parque y jardines, etc. indicando para cada una de ellas diversos aspectos como tipología, coordenadas de localización, fotografías y descripción de la incidencia/sugerencia.

2. Herramientas utilizadas

2.1 Repositorio GitHub

Vamos a utilizar GitHub como repositorio para la gestión de versiones de nuestro proyecto, debido a su eficacia y sencillez de uso. GitHub es un servicio web para la gestión de repositorios mediante el sistema de control de versiones Git de manera remota, y también permite desplegar aplicaciones en Heroku, otra plataforma que vamos a utilizar, de forma directa.

Creación de un repositorio

Es posible crear un repositorio desde la propia web de GitHub, *github.com*, de la siguiente manera:

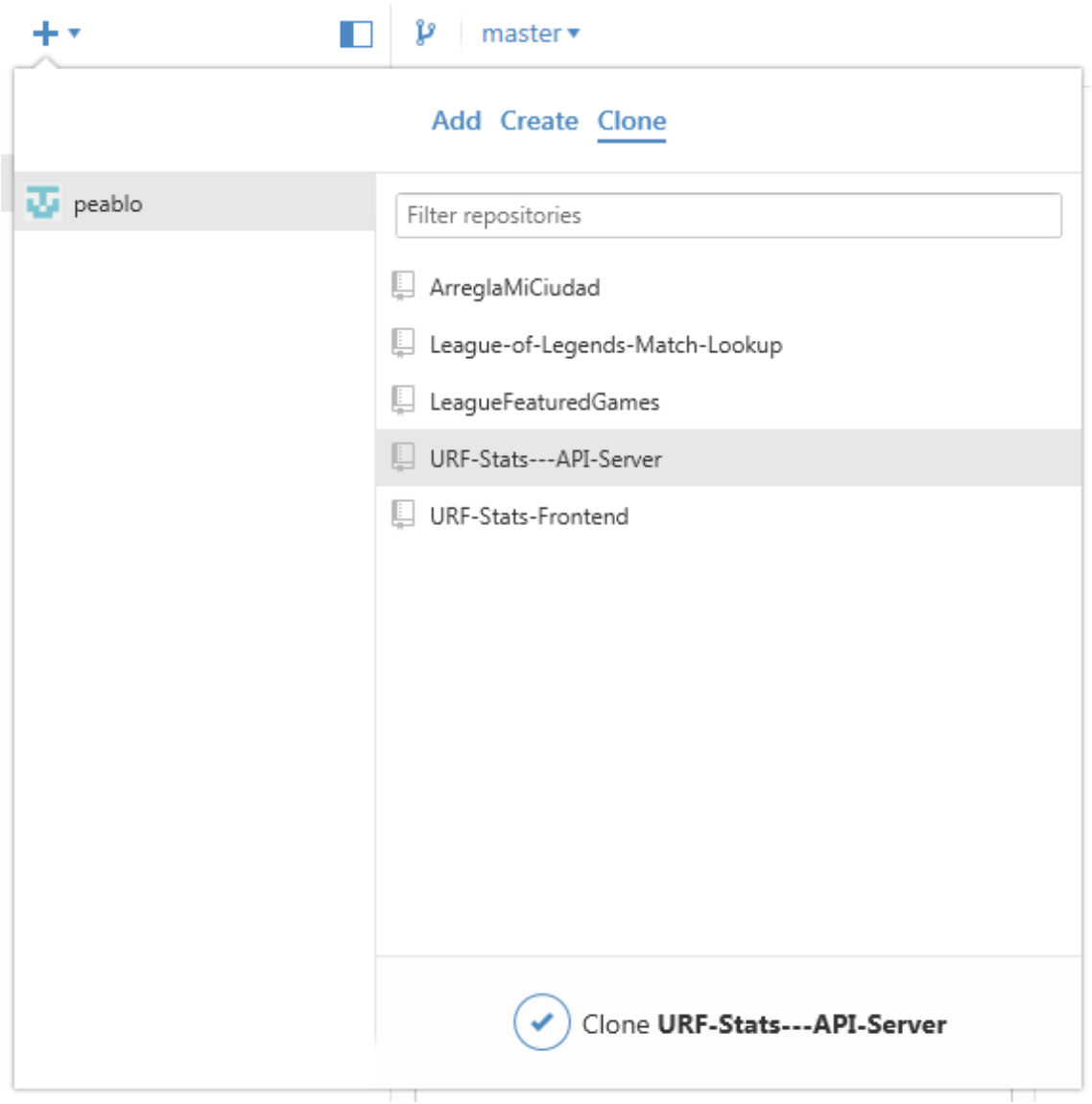
1 En la esquina superior izquierda de la pantalla, hacer clic en '**+**' y luego en *New repository***.**
2 Introducir el nombre del repositorio. Por ejemplo, *ArreglamiCiudad.*
3 Introducir la descripción del repositorio.
4 Elegir si el repositorio es privado o público, el nuestro será publico debido a la facilidad de configuración y a la necesidad de que el proyecto esté disponible para consulta e incluso mejora para todo el mundo.
5 Seleccionar *Initialize this repository with a README*
6 Hacer clic en *Create repository***,** de esta manera habremos creado el repositorio, sin código pero con un archivo llamado README.md que puede contener una descripción del proyecto así como detalles sobre su uso o instrucciones de implementación.

Sincronizar repositorio con nuestro código

Para trabajar localmente desde Windows, hemos usado la herramienta *GitHub for Windows* así como la consola de comandos de Git.

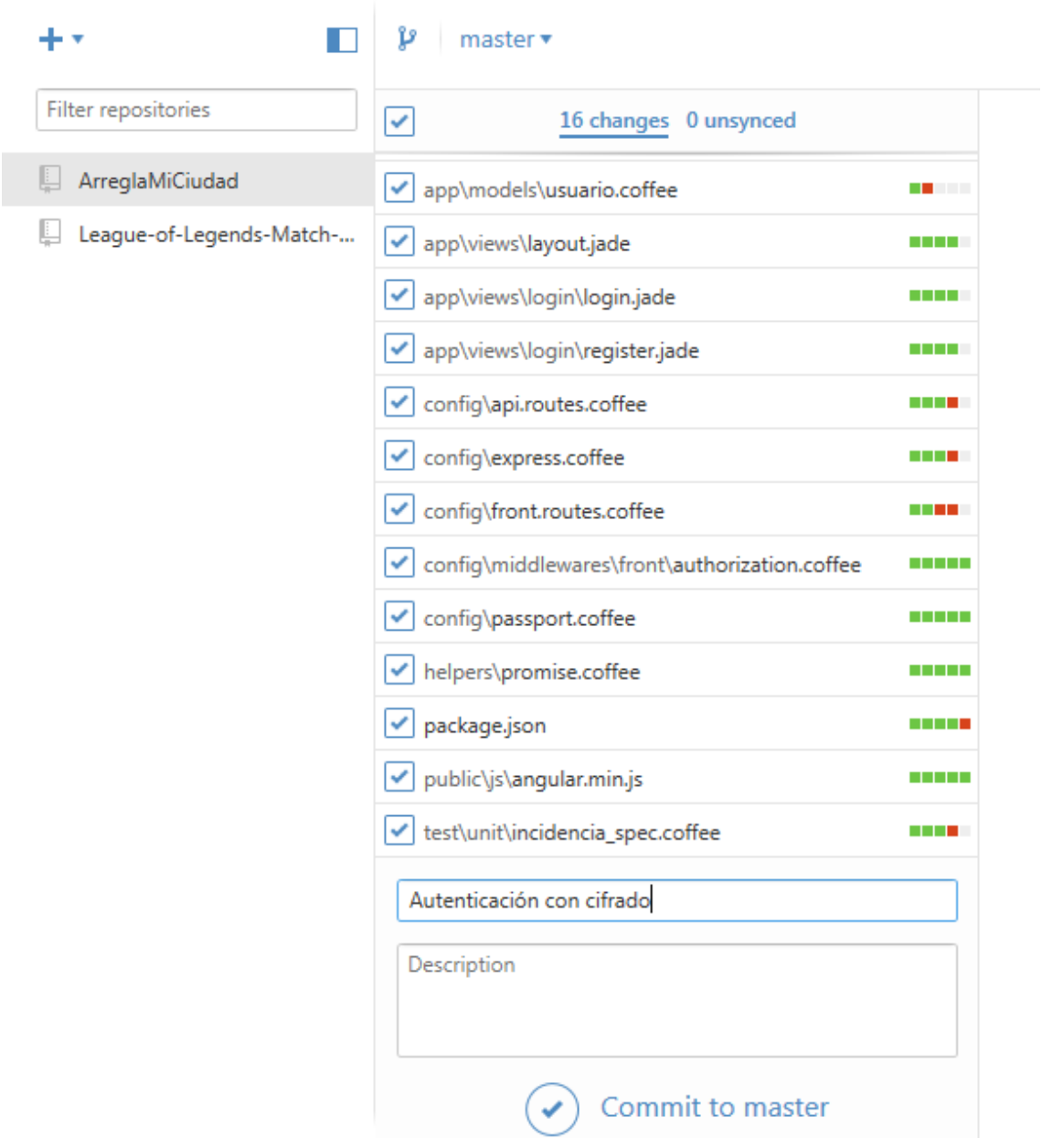
Para empezar a trabajar con el repositorio en nuestro ordenador:

1 Hacer clic en el signo + de la esquina superior izquierda de la ventana y seleccionar *clone*.
2 Elegir el nombre del repositorio y pulsar *Create <nombre>*.



A partir de ahora, Git registrará qué archivos del proyecto hemos creado, eliminado o modificado, y en cualquier momento podremos consultar un informe sobre los cambios realizados. Cuando hayamos terminado de trabajar en una modificación o mejora en concreto, conviene realizar *commit* para que Git registre los cambios de cada acción a lo largo del tiempo, así como para poder revertir las modificaciones si algo va mal. Para hacer *commit*:

1 Abrir *Github for Windows*; En la pantalla de inicio veremos un informe con los cambios que hemos realizado, así como un historial de *commits*.
2 Introducir un título breve y descriptivo de los cambios que hemos realizado así como una descripción opcional, y hacer clic en *Commit*.



También lo hemos hecho en algunas ocasiones desde la consola de comandos de Git, mediante los siguientes comandos:

- *Git status* – Para consultar los archivos que se han modificado, añadido o eliminado del repositorio
- *Git add* – Para añadir archivos que se actualizarán en el próximo *commit*
- *Git commit –m "<mensaje>"* – Para realizar *commit* añadiendo un título descriptivo.
- *Git push* – Para actualizar el repositorio remoto, en nuestro caso, en github.com.

Para este proyecto no es necesario utilizar la funcionalidad de ramas que ofrece *Git* ya que se trata de un trabajo individual.

2.2 Heroku

Vamos a utilizar Heroku para el despliegue de la demostración en vivo de nuestro proyecto, ya que se trata de una plataforma en la que se pueden alojar aplicaciones web de forma gratuita, que estarán disponibles las veinticuatro horas del día sin límite de tiempo. También dispone de tutoriales de desarrollo e implantación, y posibilidades como la de desplegar directamente desde un repositorio de GitHub.

Para alojar una aplicación web en Heroku basta con subir los archivos a un repositorio Git de Heroku, para éste proyecto lo hemos hecho directamente desde un repositorio GitHub, de la siguiente manera:

1 Desde nuestro *Dashboard* de Heroku, disponible en *heroku.com* al loguearse, hacer clic en el símbolo + en la esquina superior derecha de la página.
2 Elegir la opción *Connect to GitHub*.
3 Introducir las credenciales de nuestra cuenta *GitHub* y permitir el acceso.
4 Introducir el nombre del repositorio y pulsar en *Search* y *Connect*
5 En el final de la página, hacer clic en *Deploy branch*

2.3 Mongolab

Se ha utilizado Mongolab por su suporte técnico y sencillez de uso para el despliegue de una base de datos no relacional MongoDB de tamaño de archivo de hasta 500MB de forma gratuita.

2.4 npm Node Package Manager

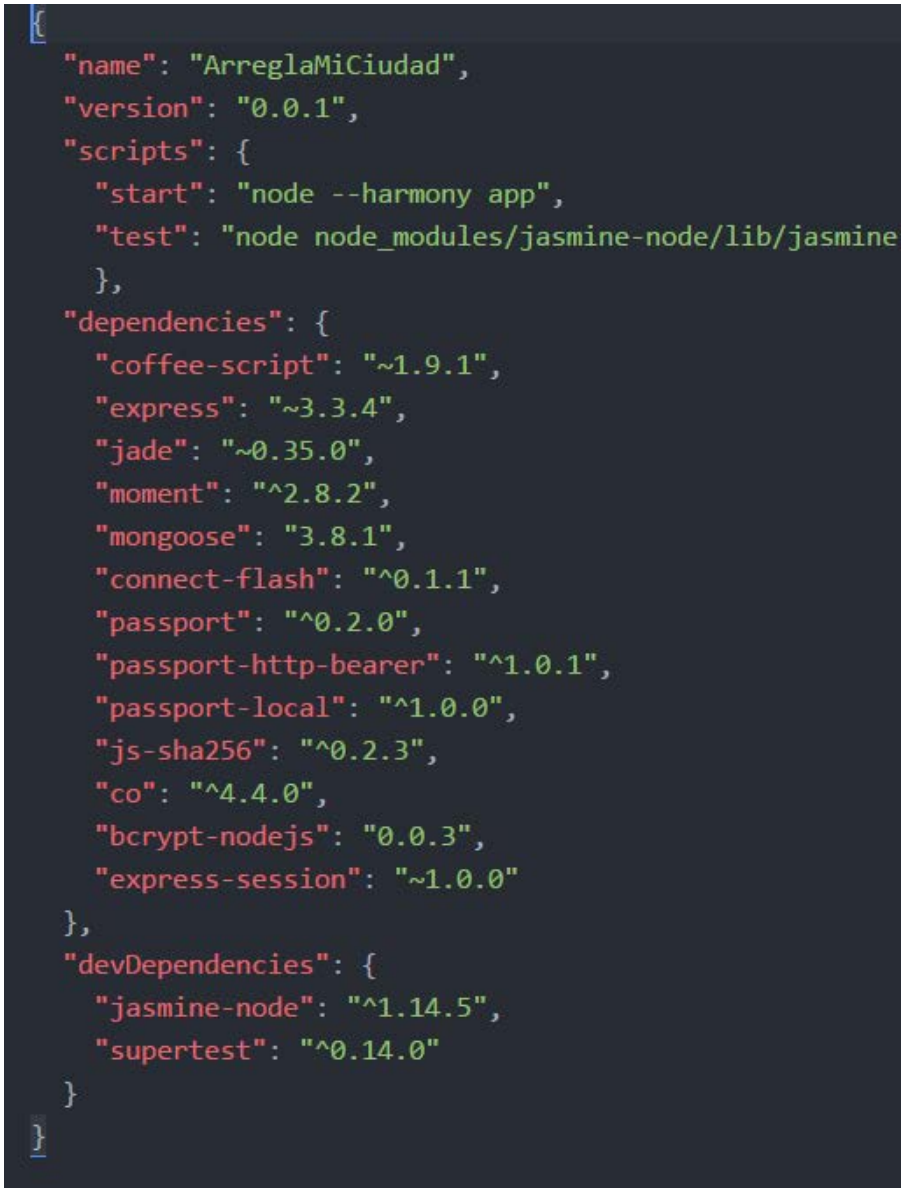
npm es una herramienta de gestión de paquetes para JavaScript, que hemos utilizado para la instalación de módulos que luego utilizará nuestra aplicación web.

npm se instala automáticamente con Node.js, la plataforma con la que hemos programado nuestra aplicación web.

Para instalar un módulo con npm, basta con situarse con la consola de comandos en la carpeta del proyecto y ejecutar *npm install <nombre>*, y se instalará en la carpeta *node_modules*.

En nuestro proyecto hemos utilizado otra funcionalidad de npm mediante la cual se permite incluir en el proyecto un archivo llamado *package.json* con información sobre los módulos que nuestra aplicación necesita para funcionar y su versión, y luego ejecutar el comando *npm install* para instalarlos todos automáticamente. Ésta funcionalidad también facilita el despliegue del proyecto de forma remota.

Nuestro archivo *package.json:*



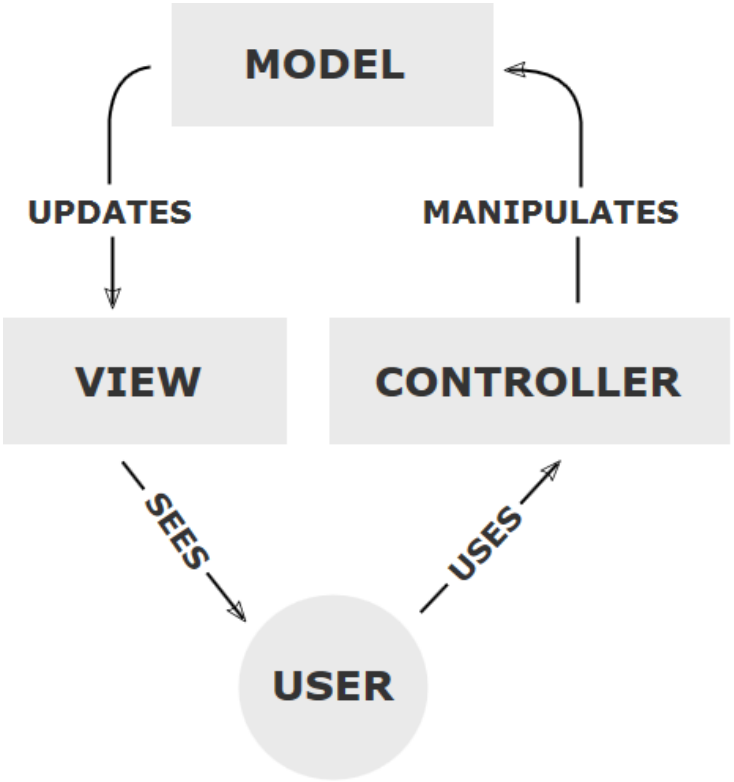
```
{
  "name": "ArreglaMiCiudad",
  "version": "0.0.1",
  "scripts": {
    "start": "node --harmony app",
    "test": "node node_modules/jasmine-node/lib/jasmine
    },
  "dependencies": {
    "coffee-script": "~1.9.1",
    "express": "~3.3.4",
    "jade": "~0.35.0",
    "moment": "^2.8.2",
    "mongoose": "3.8.1",
    "connect-flash": "^0.1.1",
    "passport": "^0.2.0",
    "passport-http-bearer": "^1.0.1",
    "passport-local": "^1.0.0",
    "js-sha256": "^0.2.3",
    "co": "^4.4.0",
    "bcrypt-nodejs": "0.0.3",
    "express-session": "~1.0.0"
  },
  "devDependencies": {
    "jasmine-node": "^1.14.5",
    "supertest": "^0.14.0"
  }
}
```

3. Patrones de diseño y metodologías

3.1 MVC – Modelo-Vista-Controlador

MVC (Modelo-Vista-Controlador) es un patrón de diseño de arquitectura de software para implementar interfaces de usuario. Divide una aplicación en tres partes interconectadas, que separan las representaciones internas de la información de la forma en que dicha información se presenta al usuario.

En ArreglamiCiudad hemos implementado el patrón MVC para dar a la aplicación una estructura mejor organizada, debido a la naturaleza del proyecto, cuya finalidad es registrar y gestionar información de incidencias.



En el caso de nuestro proyecto, se utiliza el *framework AngularJS* en el lado del cliente, capaz de manejar parte de la lógica del controlador además de la vista. *AngularJS* recibe también información del modelo desde el lado del servidor y adapta la vista de la página web a dicha información, así como a las acciones del usuario.

Ejemplo de código del *script* Angular de una de las vistas de la aplicación:

```
$scope.getIncidencias = function () {

  $http({
    method: 'GET',
    url: '/api/incidencias'
  }).success(function(data, status, headers, config) {

    $scope.incidencias = data;

    agregarIncidencias(data);

  }).error(function(data, status, headers, config) {
    alert(status);
  });

}

var agregarIncidencias = function(incidencias) {

  incidencias.forEach(function (i) {

    var myLatlng = new google.maps.LatLng(i.lat ,i.lng);

    var marker = new google.maps.Marker({
      position: myLatlng,
      title:i.titulo
    });

    marker.setMap(map);

    google.maps.event.addListener(marker, 'click', function() {
      abrirInfo(marker, i);
    });
```

En cuanto al controlador, en nuestro proyecto consiste en una capa que se ejecuta en el lado del servidor y contiene todos los métodos necesarios para manejar la información e interactuar con la base de datos.

Ejemplo de código del controlador de la API que gestiona las incidencias:

```
exports.update = (req, res) ->
  incidencia = req.body
  incidencia._id = req.params.id

  query =
    _id: incidencia._id

  params = {$set: {}}

  params.$set.titulo = incidencia.titulo if incidencia.titulo
  params.$set.descripcion = incidencia.descripcion if incidencia.descripcion
  params.$set.urlImagen = incidencia.urlImagen if incidencia.urlImagen

  Incidencia.findOneAndUpdate query, params, (err, result) ->
    return res.status(500).send(err) if err
    return res.status(404).end() unless result
    res.send result



exports.delete = (req, res) ->
  query =
    _id: req.params.id

  Incidencia.findOne query, (err, incidencia) ->
    return res.status(404).end() unless incidencia
    Incidencia.remove query, (err, result) ->
      res.status(200).end()
```

Para el modelo, hemos utilizado *Mongoose*, una herramienta que facilita la creación de modelos y su utilización para bases de datos no relacionales MongoDB

Modelo de *Incidencia*:

```
mongoose = require('mongoose')

Schema = mongoose.Schema

IncidenciaSchema = new Schema(
  titulo:
    type: String
    required: true
  descripcion:
    type: String
    required: false
  urlImagen:
    type: String
    required: false
  aprobada:
    type: Boolean
    default: false
  votos: [{
    usuario:
      type: Schema.Types.ObjectId
      ref: 'Usuario'
    texto:
      type: String
      default: ''}]
  date:
    type: Date
    require: true
    default: Date.now
  lat: Number
  lng: Number
)

Incidencia = mongoose.model('Incidencia', IncidenciaSchema)

module.exports = Incidencia
```

3.2 Desarrollo guiado por pruebas (*Test Driven Development*)

El desarrollo guiado por pruebas o *Test Driven Development* se basa en en el siguiente ciclo:

1 El programador escribe un caso de prueba automático que define una mejora o funcionalidad deseada para la aplicación.
2 Se desarrolla el código justo para que el test pase con éxito.
3 Se *refactoriza* el código desarrollado.

En nuestro proyecto utilizamos desarrollo guiado por pruebas para contar con un sistema fácil, fiable y rápido que nos indique, con cada mejora o cambio que se realiza en la aplicación, si sigue funcionando según lo esperado.

Los casos de prueba que hemos implementado comprueban el funcionamiento de la gestión de incidencias y la autenticación, mediante la API a la que acceden las vistas de la página web y la aplicación móvil.

Se ha utilizado *Jasmine* como herramienta para las pruebas, así como el módulo npm *supertest* para realizar peticiones a la API desde los casos de prueba.

Ejemplo de caso de prueba, que comprueba el funcionamiento de la opción de votar en una incidencia reportada por otro usuario y añadir un comentario:

```
it "añade un voto con comentario", (done) ->
  Incidencia.findOne titulo: 'prueba', (err, incidencia) ->
      return done(err) if err
      voto =
        texto: 'comentario'
      request(app).put("#{apiUrl}/#{incidencia._id}/votar").set("Authorization", "Bearer testKey").send(voto).end (err, res) ->
        return done(err) if err
        expect(res.statusCode).toBe 200
        request(app).get("#{apiUrl}/#{incidencia._id}").set("Authorization", "Bearer testKey").end (err, res) ->
          incidencia = res.body
          expect(incidencia.votos?.length).toBe 2
          expect(incidencia.votos[1]?.texto).toEqual 'comentario'
          expect(incidencia.votos[1]?.usuario?.name).toEqual 'test'
          done()
```

4. Diseño

La infraestructura de ArreglamiCiudad consistirá en un servicio web que almacenará y gestionará la información de incidencias y usuarios y servirá mediante una API pública a la aplicación Android y al portal web.

La aplicación Android estará diseñada para el reporte rápido de incidencias: No obligará al usuario a escribir texto más allá de un breve título descriptivo, y permitirá tomar una foto y enviarla al servidor de manera rápida desde una misma pantalla.

El portal web permitirá visualizar todas las incidencias en un mapa, ver los detalles de cualquiera de ellas con sólo hacer clic en el marcador correspondiente, corroborar una incidencia creada por otro usuario o añadir información.

Para el personal del ayuntamiento también estarán disponibles opciones de borrado de reportes falsos, confirmación de los verdaderos, y el registro de gestión y resolución de incidencias en el sistema.

En cuanto a la estructura del sistema, como se ha explicado antes está basado en el patrón MVC, siendo la capa de modelado y persistencia un sistema Mongoose/MongoDB en el que se modelan con Mongoose los esquemas para las incidencias y usuarios y se guardan como documentos JSON en la base de datos.

La capa controlador está basada principalmente en dos ficheros separados de código en los que se centralizan y abstraen todas las comunicaciones con la base de datos referentes a la lectura o modificación de información referente a incidencias y usuarios.

La capa de vistas está basada en una serie de archivos y plantillas JADE que se compilan a HTML en tiempo de ejecución y que ofrecen toda la funcionalidad del portal web, utilizando la API pública para todas las comunicaciones con el servidor.

4.1 Herramientas, *frameworks* y lenguajes

Herramientas de desarrollo

Atom: Atom es un editor de texto diseñado para las tareas de programación, con funcionalidades básicas como la diferenciación mediante colores de las diferentes partes y palabras clave del código. Además, permite la instalación de plugins creados por otros usuarios para extender su funcionalidad.

Android Studio: Android Studio es un IDE diseñado específicamente para Android, con diversas funcionalidades destinadas al desarrollo y compilación de aplicaciones. Cabe destacar que también cuenta con previsualización y editor gráfico para el desarrollo de interfaces.

Lenguajes de programación

JavaScript: JavaScript es un lenguaje de script que se ha utilizado a lo largo de todas las capas de nuestro proyecto, ya que se aplica tanto a la vista de la página web, ejecutándose en el navegador del cliente, como al controlador y enrutamiento del servidor, al modelo y la base de datos, que al estar basada en MongoDB funciona mediante objetos JSON (*JavaScript Object Notation)*.

CoffeeScript: Para conseguir una superior simplicidad del código y facilidad de desarrollo y mantenimiento, se ha utilizado *CoffeeScript*, un lenguaje basado en JavaScript que simplifica el desarrollo, aumenta las posibilidades, y se compila automáticamente a código JavaScript en tiempo de ejecución, gracias al módulo *coffee-script* de npm.

JADE: Para las vistas web se ha utilizado JADE en lugar de HTML. JADE es un sistema que, además de mejorar la simplicidad y posibilidades del código HTML, ofrece un sistema de plantillas gracias al que hemos podido programar la base de las vistas web en un archivo separado que importa todos los scripts y estilos y monta las barras superior y lateral, pudiendo así centrar el código de cada vista únicamente en su contenido principal. Esto se traduce a código HTML en tiempo de ejecución mediante el motor de vistas JADE.

Tecnologías y frameworks

AngularJS: AngularJS es un *framework* capaz de aumentar las posibilidades de las vistas de la página web en el lado del cliente, ejecutando parte de la lógica de la capa controlador y reaccionando a las acciones del usuario como lo haría una aplicación de escritorio.

NodeJS: Node.JS es una plataforma capaz de ejecutar JavaScript en el lado del servidor, con lo que se mejora y simplifica la integración con las vistas. El código ejecutado por NodeJS funciona asíncronamente sin necesidad de hilos, mediante eventos que desencadenan la ejecución de determinadas funciones, y funciones *callback* para organizar la ejecución asíncrona sin condiciones de carrera.

Express: Express es un framework para node.js que simplifica el desarrollo de aplicaciones con mucho código y provee una estructura más organizada de las diferentes rutas y funciones.

EcmaScript6 - Harmony: Se han aprovechado algunas de las funciones del nuevo estándar EcmaScript6, implementado por JavaScript, que NodeJS es capaz de ejecutar siempre que se incluya la opción *–harmony* al iniciar la aplicación. Incluye nuevas funcionalidades como funciones generadoras (*generators),* objetos *Promise* para la ejecución asíncrona, y nuevas funciones matemáticas y de iteración de listas.

MongoDB: MongoDB es un sistema de bases de datos no relacional cuyos registros y consultas consisten en objetos JavaScript, motivo por el que se ha elegido para nuestro proyecto.

Mongoose: Mongoose es un herramienta de modelado para MongoDB que permite crear un modelo para los objetos utilizados, como *incidencia* y *usuario* en nuestro caso, y facilita su creación, consulta, actualización y borrado.

4.2 Cifrado

Para garantizar la seguridad de la aplicación y la privacidad de los usuarios, se cifran sus contraseñas en tiempo de ejecución, antes de guardarlas en la base de datos, utilizando el paquete de npm *bcrypt-nodejs*, una implementación para node.js de bcrypt. En la capa controlador, al registrar un usuario, se realiza un cifrado procesando la información durante 8 rondas antes de guardar la contraseña, y al intentar hacer login se compara la contraseña introducida con el hash almacenado para un usuario.

También se cifra la clave API de los usuarios, una variable necesaria para un método alternativo de autenticación por *token*, mediante la función *hash* SHA-256.

Cifrado:

```
exports.generateHash = (password) ->
  bcrypt.hashSync password, bcrypt.genSaltSync(8), null
```

Comprobación:

```
exports.isValidPassword = (password, passwordToCompare) ->
  bcrypt.compareSync password, passwordToCompare
```

4.3 Autenticación

En este proyecto hemos utilizado el módulo *Passport* para la autenticación en node.js, definiendo dos posibles tipos de autenticación, por *cookies* y por *token*, esta última es útil principalmente para los tests automatizados.

La autenticación por cookies funciona de la siguiente manera:

1. El usuario introduce su usuario y contraseña en la pantalla de login, en la ruta raíz de la aplicación o '/login'.
2. Durante el enrutamiento, se ejecuta el *Passport* como middleware que bloquea el acceso a las partes de la aplicación que requieren autenticación.
3. El módulo *Passport* procesa la petición de autenticación con la estrategia que hemos definido, que comprueba que el nombre de usuario existe y si la contraseña introducida es la que le corresponde:

```
passport.use 'local-login', new LocalStrategy(
  usernameField : 'username',
  passwordField : 'password',
  passReqToCallback : true
, (req, username, password, done) ->

  process.nextTick ->
    co ->
      user = yield userController.getByUsername username

      return done(null, false, req.flash('loginMessage', 'User or password incorrect')) unless user
      return done(null, false, req.flash('loginMessage', 'User or password incorrect')) unless userController.isValidPassword password, user.password

      req.session.apiKey = user.apiKey

      done null, user

    .catch(done)
```

4. Se devuelve un valor según el resultado, con lo que se admite o bloquea el acceso al usuario, mostrando un mensaje de error en caso de que se deniegue el acceso. Un usuario no autenticado no puede acceder a determinadas url.

Registro de un usuario:

1. El usuario introduce su nombre de usuario y contraseña en la pantalla de registro, en la ruta '/register'
2. La información se envía una vez más a '/register', esta vez en forma de petición POST
3. El módulo *Passport* procesa la petición de autenticación con la estrategia que hemos definido para la creación de usuarios, que comprueba si el nombre de usuario está disponible :

```
passport.use 'local-signup', new LocalStrategy(
  usernameField : 'username',
  passwordField : 'password',
  passReqToCallback : true
, (req, username, password, done) ->

  process.nextTick ->
    co ->
      user = yield userController.getByUsername username
      return done(null, false, req.flash('signupMessage', 'That username is already taken.')) if user

      newUser =
        name: username

      createdUser = yield userController.create newUser, password

      return done null, createdUser
    .catch(done)
)
```

4. Se devuelve un valor según el resultado, que el enrutamiento utilizará para determinar si devolver al usuario a la pantalla de registro mostrando un mensaje de error o llevarle directamente a la aplicación. La variable *createdUser* en el código de ejemplo contiene la información que se agregará a las cookies del usuario para mantener su sesión.

Autenticación por *token*:

1. El usuario envía una petición a la aplicación web, con un *token* en la cabecera de la petición en lugar de una sesión en las cookies.
2. El middleware de autenticación, al ver que el usuario no está autenticado de la forma tradicional, pasa a comprobar mediante el método por *token*:

```
passport = require 'passport'

exports.isLogged = (req, res, next) ->
  if req.isAuthenticated()
    return next()
  passport.authenticate('bearer', session:false)(req, res, next)
```

3. Con esto, el módulo *Passport* comprueba si el token corresponde a algún usuario de la base de datos:

```
passport.use 'bearer', new BearerStrategy(
  passReqToCallback: true
, (req, token, done) ->

  process.nextTick ->
    co ->
      user = yield userController.getByApiKey token

      if user
        done null, user
      else
        done null, false

    .catch(done)

)
```

4. Si el *token* provisto coincide con algún usuario, se produce la autenticación y se permite el acceso con los privilegios de dicho usuario, si no, se deniega el acceso a la ruta solicitada.

4.4 Enrutamiento y *middleware*

En este proyecto vamos a utilizar el framework *Express* para la diferenciación de respuestas y funciones según la ruta a la que accede el usuario, con posibilidad de *middleware*, que en nuestro caso se ha usado principalmente para la autenticación.

Contamos con dos archivos de configuración de rutas, uno para la API y otro para las vistas:

```
#controlador
incidencias = require '../app/controllers/api/incidencia'

app.get '/api/incidencias/:id?', incidencias.get

app.post '/api/incidencias', incidencias.create

app.put '/api/incidencias/:id', Auth.isLogged, incidencias.update
app.put '/api/incidencias/:id/votar', Auth.isLogged, incidencias.votar
app.put '/api/incidencias/:id/aprobar', Auth.isLogged, incidencias.aprobar

app.delete '/api/incidencias/:id', Auth.isLogged, incidencias.delete
```

Ejemplo de código de rutas API, donde se observa tanto la inclusión de parámetros como el uso de middleware (Auth)

```
module.exports = (app, config, passport) ->

  #controlador
  incidencias = require '../app/controllers/front/incidencia'

  app.get '/', (req, res) ->
    res.render 'login/login'

  # Authentication
  app.get '/login', (req, res) ->
    res.render 'login/login', {message: req.flash('loginMessage') }

  app.post '/login', passport.authenticate('local-login', {failureRedirect: "/login", failureFlash: true}) , (req, res) ->
    res.redirect 'index'

  app.post '/register', passport.authenticate('local-signup', {
    successRedirect : "/index", # redirect to the secure profile section
    failureRedirect : "/register", # redirect back to the signup page if there is an error
    failureFlash : true # allow flash messages
  })

  app.get '/logout', (req, res) ->
    req.logOut()
    res.redirect 'login/login'

  app.get '/register', (req, res) ->
    res.render 'login/register', {message: req.flash('signupMessage') }

  app.get '/index', Auth.isLogged, (req, res) ->
    res.render 'index'
```

Ejemplo de código de rutas de las vistas, se observa el uso de middleware así como la configuración de Passport y el paso de mensajes de error.

4.5 Capa de vistas

Para la capa de vistas del proyecto hemos utilizado JADE para mejorar el desarrollo HTML con funcionalidades como las plantillas o la simplificación del código.

Mediante el sistema de plantillas de JADE, se permite definir una plantilla base con las dependencias y estructura que tienen en común todas las páginas de la aplicación, para luego centrarse únicamente en el contenido al programar el resto:

```
  title ArreglaMiCiudad

  //- Bootstrap core CSS
  link(href='assets/css/bootstrap.css', rel='stylesheet')
  //- external css
  link(href='assets/font-awesome/css/font-awesome.css', rel='stylesheet')
  //- Custom styles for this template
  link(href='assets/css/style.css', rel='stylesheet')
  link(href='assets/css/style-responsive.css', rel='stylesheet')

  block head

body
  section#container
    include bars/nav
    include bars/side

    section#main-content
      section.wrapper.site-min-height
        block content

    footer.site-footer
      block footer

  script(src='assets/js/jquery.js')
  script(src='assets/js/bootstrap.min.js')
  script(src='assets/js/jquery-ui-1.9.2.custom.min.js')
  script(src='assets/js/jquery.ui.touch-punch.min.js')
  script.include(type='text/javascript', src='assets/js/jquery.dcjqaccordion.2.7.js')
  script(src='assets/js/jquery.scrollTo.min.js')
  script(src='assets/js/jquery.nicescroll.js', type='text/javascript')
  // common script for all pages
  script(src='assets/js/common-scripts.js')

  //angular
  script(src='js/angular.min.js')
```

Sección de código de la plantilla base

```
header.header.black-bg
  .sidebar-toggle-box
    .fa.fa-bars.tooltips(data-placement='right', data-original-title='Toggle Navigation')
  //- logo start
  a.logo(href='index.html')
    b ARREGLA MI CIUDAD
  //- logo end
  #top_menu.nav.notify-row
    //- notification start
    ul.nav.top-menu
      //- settings start
      li.dropdown
        a.dropdown-toggle(data-toggle='dropdown', href='index.html#')
          i.fa.fa-tasks
          span.badge.bg-theme 4
        ul.dropdown-menu.extended.tasks-bar
          .notify-arrow.notify-arrow-green
          li
            p.green You have 4 pending tasks
          li
            a(href='/index')
              .task-info
                .desc DashGum Admin Panel
                .percent 40%
              .progress.progress-striped
```

Sección de código de la barra de navegación

```
extends ../layout

block head
  script(src="/js/controllers/incidencias/mapa.js")
  script(src="https://maps.googleapis.com/maps/api/js?key=AIzaSyBEK84gbA-jv
  script.
    var map, lat = 39.4609105, lng = -0.3604354, zoom = 13;

    function initialize() {
      var mapOptions = {
        center: { lat: lat,
                  lng: lng
                },
        zoom: zoom
      };
      map = new google.maps.Map(document.getElementById('map-canvas'),
          mapOptions);
    }
    google.maps.event.addDomListener(window, 'load', initialize);


block content
  div(ng-controller-"mapaCtrl")
```

Sección de código de una de las secciones de la página web

También se utiliza el framework AngularJS para mostrar la información al usuario y realizar algunas operaciones de la capa de controlador en el lado del cliente:

```
  var abrirInfo = function(marker, incidencia){

    var contentString = '<div id="content">'+
      '<div id="siteNotice">'+
      '</div>'+
      '<h1 id="firstHeading" class="firstHeading">'+ incidencia.titulo +'</h1>'+
      '<div id="bodyContent">'+
      '<img src="'+incidencia.urlImagen+'" width="140">'+
      '<p>'+ incidencia.descripcion +'</p>'+
      '<a href="/incidencias/'+incidencia._id+'">'+
      'Gestionar</a>'+
      '</div>'+
      '</div>';

      var infowindow = new google.maps.InfoWindow({
        content: contentString
      });

      infowindow.open(map, marker);

  }

  $scope.getIncidencias();

});

arreglaMiCiudad.controller ('verIncidenciaCtrl', function ($scope, $http, $window) {
```

Fragmento de código AngularJS que, haciendo uso del modelo, utiliza la información para mostrar el detalle de una incidencia en el mapa

4.6 Capa de controlador

En ArreglamiCiudad, la capa controlador consiste principalmente en una serie de funciones, definidas en archivos de código separados, para la gestión, consulta y almacenamiento de la información sobre incidencias y usuarios.

Aunque la vista también utilice el modelo, nunca se comunica con la base de datos si no es a través de esta capa.

Las operaciones del controlador de incidencias permiten crear, acceder, borrar y modificar (incluyendo votar y confirmar/rechazar)

Las operaciones del controlador de usuario, además de las operaciones CRUD, también cuentan con funcionalidades de cifrado. Este controlador utiliza la tecnología de generadores ES6.

4.7 Capa de modelo

Para el modelado hemos utilizado Mongoose, una herramienta que complementa MongoDB permitiendo crear modelos de datos y abstrayendo las operaciones de lectura, acceso, escritura y borrado. Hemos creado modelos para la información de las incidencias y de los usuarios, indicando para cada campo el tipo de archivo junto con otros parámetros, como referencias a otros esquemas y si un campo es obligatorio:

```
Schema = mongoose.Schema

UsuarioSchema = new Schema(
  name:
    type: String
    required: true
  password:
    type: String
    required: true
  operator:
    type: Boolean
    default: false
  apiKey:
    type: String
    required: true
)

Usuario = mongoose.model('Usuario', UsuarioSchema)

module.exports = Usuario
```

Esquema del modelo de usuario.

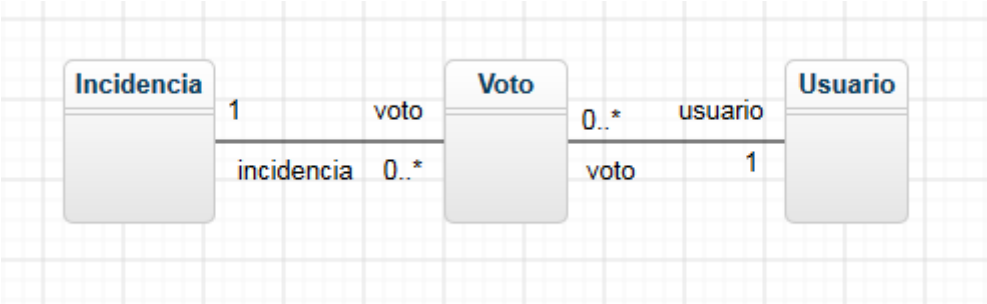
```
Schema = mongoose.Schema

IncidenciaSchema = new Schema(
  titulo:
    type: String
    required: true
  descripcion:
    type: String
    required: false
  urlImagen:
    type: String
    required: false
  aprobada:
    type: Boolean
    default: false
  votos: [{
    usuario:
      type: Schema.Types.ObjectId
      ref: 'Usuario'
    texto:
      type: String
      default: ''}]
  date:
    type: Date
    require: true
    default: Date.now
  lat: Number
  lng: Number
)

Incidencia = mongoose.model('Incidencia', IncidenciaSchema)

module.exports = Incidencia
```

Esquema del modelo de incidencia, la lista de votos incluye una referencia al esquema de usuario, que puede poblarse durante las consultas con la opción 'populate' de Mongoose



Modelo de datos

```coffeescript
exports.connect = (config) ->
  mongoose = require('mongoose')
  fs = require('fs')

  mongoose.connect config.db, (err) ->
    if (!err)
      console.log 'Conectado a MongoDB'

  db = mongoose.connection
  db.on 'error',  (err) ->
        console.log(err)
        throw new Error('Fallo de conexión a MongoDB, base de datos: ' + config.db)

  modelsPath = __dirname + '/../models'
  fs.readdirSync(modelsPath).forEach (file) ->
    require(modelsPath + '/' + file) if file.indexOf('.js') >= 0 or file.indexOf('.coffee') >= 0
```

Conexión a la base de datos, se observa como se realiza la conexión con el servidor de Mongolab utilizando la información de un fichero de configuración, y posteriormente se cargan todos los modelos de Mongoose

4.8 API pública

Para el acceso a la información por parte de las vistas de la página web, el entorno de pruebas automatizado y la aplicación Android, se utiliza una API mediante la cual la base de datos y capa controlador se comunican con el resto de aplicaciones de manera sencilla y desacoplada.

Las rutas de la API pública están protegidas mediante *middleware* para impedir el acceso por parte de un usuario no autenticado.

Autenticación

- '/login' (Petición POST) – Realizando una petición post a la url /login del servidor, se pide al servidor que compruebe nuestras credenciales y nos autentique, guardando la información de sesión en nuestras cookies en caso de que proceda.
- '/register' (Petición POST) – Con una petición post a '/register' se envían un usuario nuevo y contraseña al servidor para que los registre en la base de datos, comprobando antes si el nombre de usuario está disponible y devolviendo en ese caso un mensaje de error en la respuesta, que se mostrará en la página si estamos utilizando un navegador.

Incidencias

- '/api/incidencias/:id?' (Petición GET) – Realizando una petición get a '/api/incidencias' con el parámetro opcional 'id', se obtiene una lista en formato JSON de todas las incidencias registradas en el servidor, o bien la incidencia a la que corresponde la id introducida. Si no existe, el servidor devolverá una respuesta con código de error 404.
- '/api/incidencias' (Petición POST) – Haciendo una petición post a la url de la API de incidencias, se envía a la aplicación la información necesaria para reportar una nueva incidencia, que se registrará en el sistema y podrá ser vista y votada por los demás usuarios.
- '/api/incidencias/:id' (Petición PUT) – Mediante una petición PUT con el parámetro obligatorio id, se envía la información necesaria para modificar la incidencia a la que corresponde la ID, por ejemplo para mejorar una descripción.
- '/api/incidencias/:id/votar' (Petición PUT) – Añadiendo 'votar' a la url anterior, lo que se haría es añadir un voto a la incidencia a la que corresponde la id, siempre que se esté logueado con un usuario que no haya votado la misma anteriormente.
- '/api/incidencias/:id/aprobar' (Petición PUT) – Añadiendo 'aprobar a la url anterior, se confirmaría que el reporte de incidencia es verídico y se pasaría a la gestión de su reparación. Es necesario estar logueado como operador para realizar esta acción.

4.9 Entorno de pruebas

Para las pruebas automáticas de la API de la aplicación, se ha utilizado la herramienta Jasmine, que permite configurar los tests de forma sencilla e intuitiva.

Se ha realizado un caso de prueba para cada detalle de la funcionalidad de la aplicación, para poder comprobar que sigue funcionando correctamente después de cada cambio o mejora, así como para confirmar cuando una modificación deseada está terminada y funcionando correctamente, ya que en la metodología que hemos utilizado para este proyecto, el test se desarrolla antes que la propia funcionalidad.

Para mejorar la simplicidad y mantenibilidad del entorno de pruebas y permitir la ejecución de determinados tests individualmente, se ha diseñado de forma que ningún test influye en el estado de los demás, reiniciando la información de la base de datos antes y después de cada ejecución de un caso de prueba.

Cada caso de prueba utiliza la API para realizar una acción determinada y después hace diversas comprobaciones sobre los resultados de la petición o el estado de la base de datos, informando de un error y dando el test por inválido si alguna de las condiciones no se cumple.

```
prepararDB = (done) ->
  usuario = new Usuario
  usuario.name = 'test'
  usuario.password = 'testPassword'
  usuario.operator = false
  usuario.apiKey = 'testKey'
  usuario.save (err, result) ->
    admin = new Usuario
    admin.name = 'admin'
    admin.password = 'adminPassword'
    admin.operator = true
    admin.apiKey = 'adminKey'
    admin.save (err, result) ->
      incidencia = new Incidencia
      incidencia.titulo = 'prueba'
      incidencia.descripcion = 'prueba'
      incidencia.urlImagen = 'prueba'
      incidencia.lat = 1
      incidencia.lng = 2
      incidencia.save (err, result) ->
        done()

limpiarDB = (done) ->
  Usuario.remove {}, (err, result) ->
    Incidencia.remove {}, (err, result) ->
      done()

beforeEach prepararDB
afterEach limpiarDB
```

Limpieza y preparación de la base de datos antes de cada ejecución

```
it "Devuelve 404 si la incidencia no existe", (done) ->
  request(app).get("#{apiUrl}/54465a699a0d2fcd11000059").set("Authorization", "Bearer testKey").end (err, res) ->
    return done(err) if err
    expect(res.statusCode).toBe 404
    done()
```

Ejemplo de caso de prueba

5. Aplicación Android

Debido a que está basada en la colaboración ciudadana, ArreglamiCiudad contará con una aplicación Android disponible para todo el mundo, para garantizar que las personas que quieran participar puedan reportar las incidencias que descubran desde cualquier sitio, de manera sencilla y rápida.

La aplicación permitirá tomar una foto, añadir título y descripción, y enviarla a ArreglamiCiudad para su gestión.

Captura del menú para generar un reporte de la aplicación Android.

Para la toma de fotos se ha utilizado la aplicación Cámara de Android, abriéndola desde nuestra aplicación y recuperando la foto tomada mediante *Intents*.

El envío de la foto a nuestro servidor se realiza utilizando el servicio Amazon S3, que aloja la foto, de manera que sólo es necesario enviar a la aplicación de ArreglamiCiudad la URL:

```
AmazonS3Client s3Client =   new AmazonS3Client( new BasicAWSCredentials( Id, Key ) );

PutObjectRequest por = new PutObjectRequest( Constants.getPictureBucket(),
        picName,
        new java.io.File( filePath) );

s3Client.putObject( por );
```

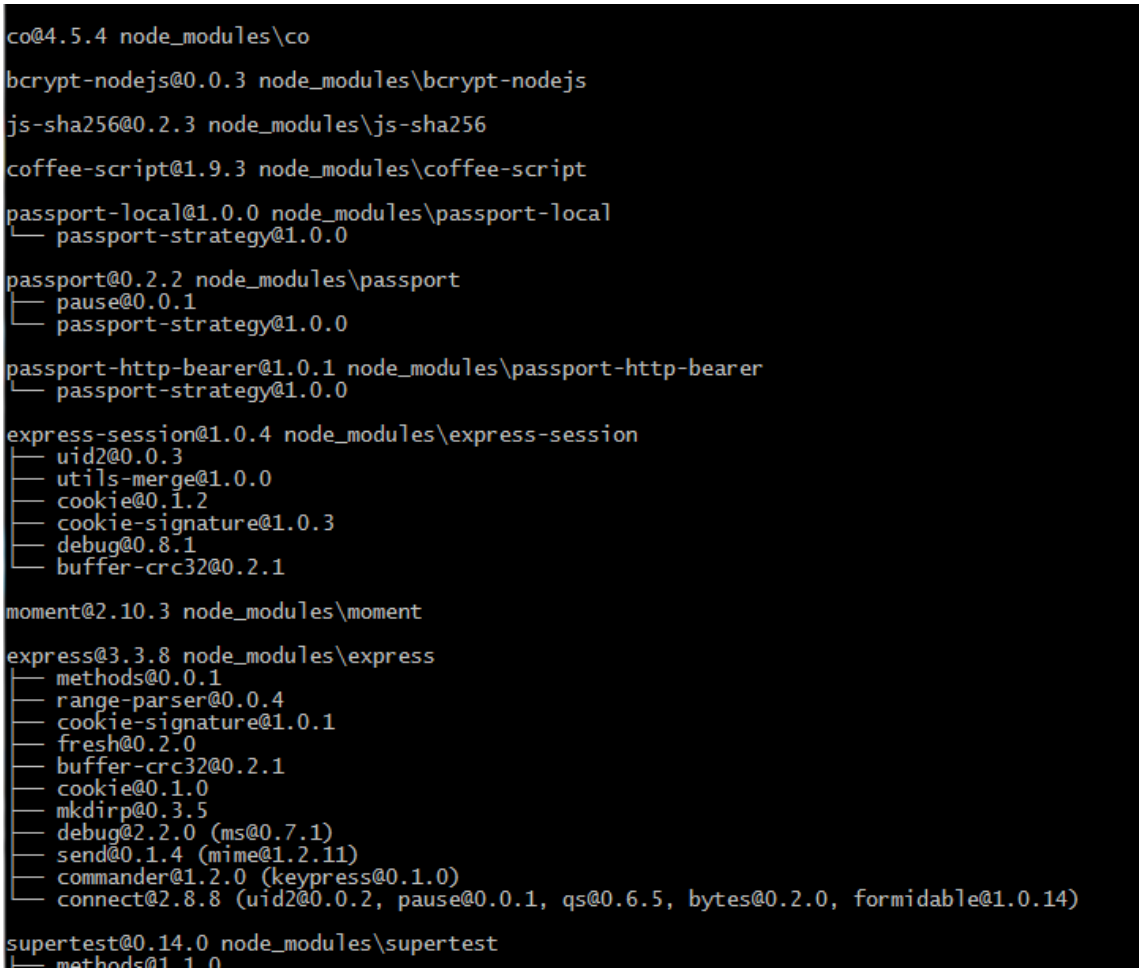
Al enviar el reporte generado al servidor, se registrará una nueva incidencia que se añadirá a la base de datos del sistema y los demás usuarios ya estarán avisados del problema, además de poder comentarla, corroborarla o desmentirla en caso de que fuera un reporte falso.

6. Implementación y demostración

Aunque es fácil desplegar la aplicación en Heroku mediante su funcionalidad para conectarse con un repositorio de GitHub directamente, en esta sección vamos a detallar cómo funciona realmente el despliegue de ArreglamiCiudad a partir del código fuente y qué pasos seguir para realizarlo.

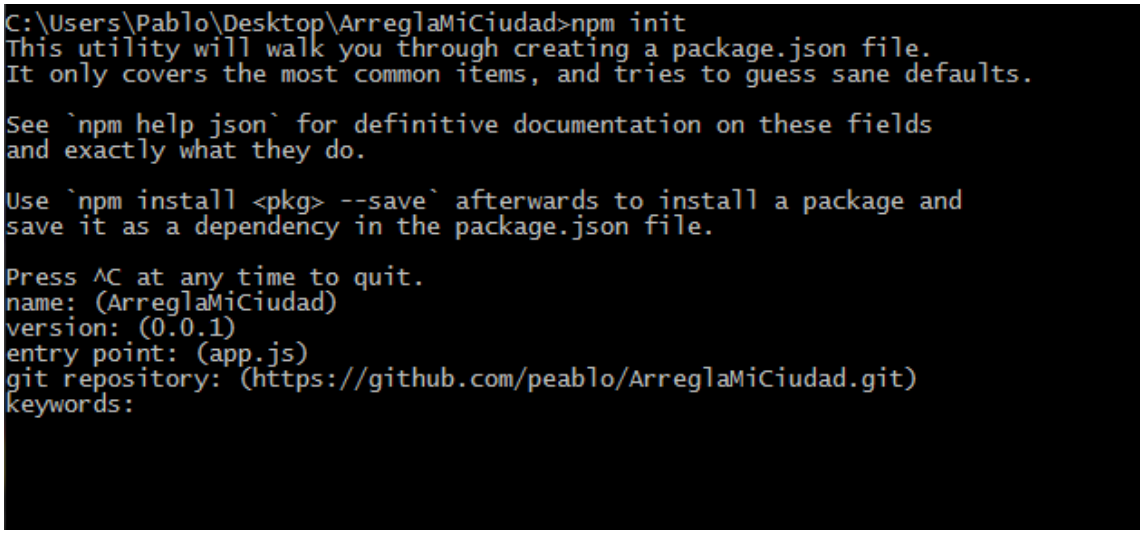
Para disminuir el peso del repositorio y no llenarlo de código innecesario, se ha configurado Git para ignorar los archivos de módulos de npm, por lo que ha de realizarse la instalación de los mismos antes del despliegue.

Npm automatiza muchas de las funciones necesarias, por ejemplo, con el comando 'npm install' se instalan automáticamente todos los módulos necesarios en su versión especificada en las secciónes de nuestro package.json 'dependencies' y 'devDependencies', ésta última reservada para dependencias que sólo se utilizarán para su desarrollo y pruebas y nunca en producción.



Instalación automática de las dependencias

El archivo package.json, a su vez, se puede generar automáticamente de forma guiada mediante el comando *npm init*



Generación del package.json

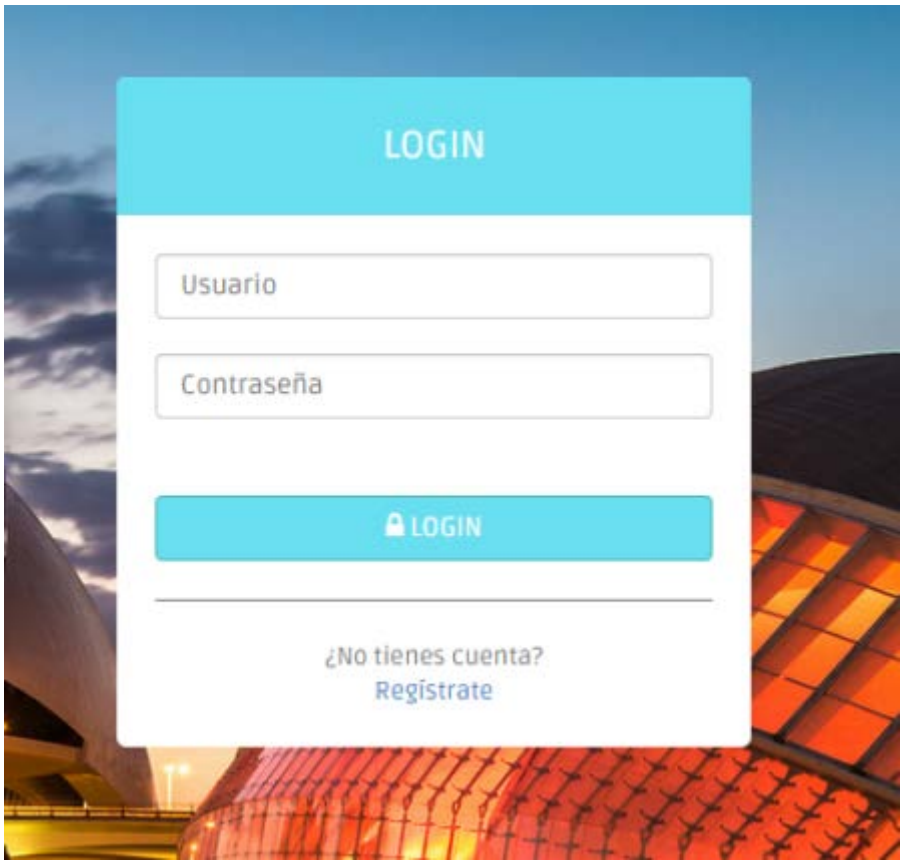
Hecho esto, basta con situarse en la ruta del proyecto y ejecutar el comando de inicio, *node –harmony app* en nuestro caso, ya que necesitamos la opción *harmony* para permitir las funcionalidades exclusivas de EcmaScript6, y el servidor estará ejecutándose en modo local, accesible a la url *localhost:3000* y reportando por la consola todas las peticiones realizadas así como cualquier mensaje de debug o error.

También es posible desplegar nuestra aplicación en un servicio como Heroku y que el servidor remoto utilice la información sobre dependencias, comando de inicio y comando de testing para realizar las instalaciónes necesarias y la ejecución de la aplicación.

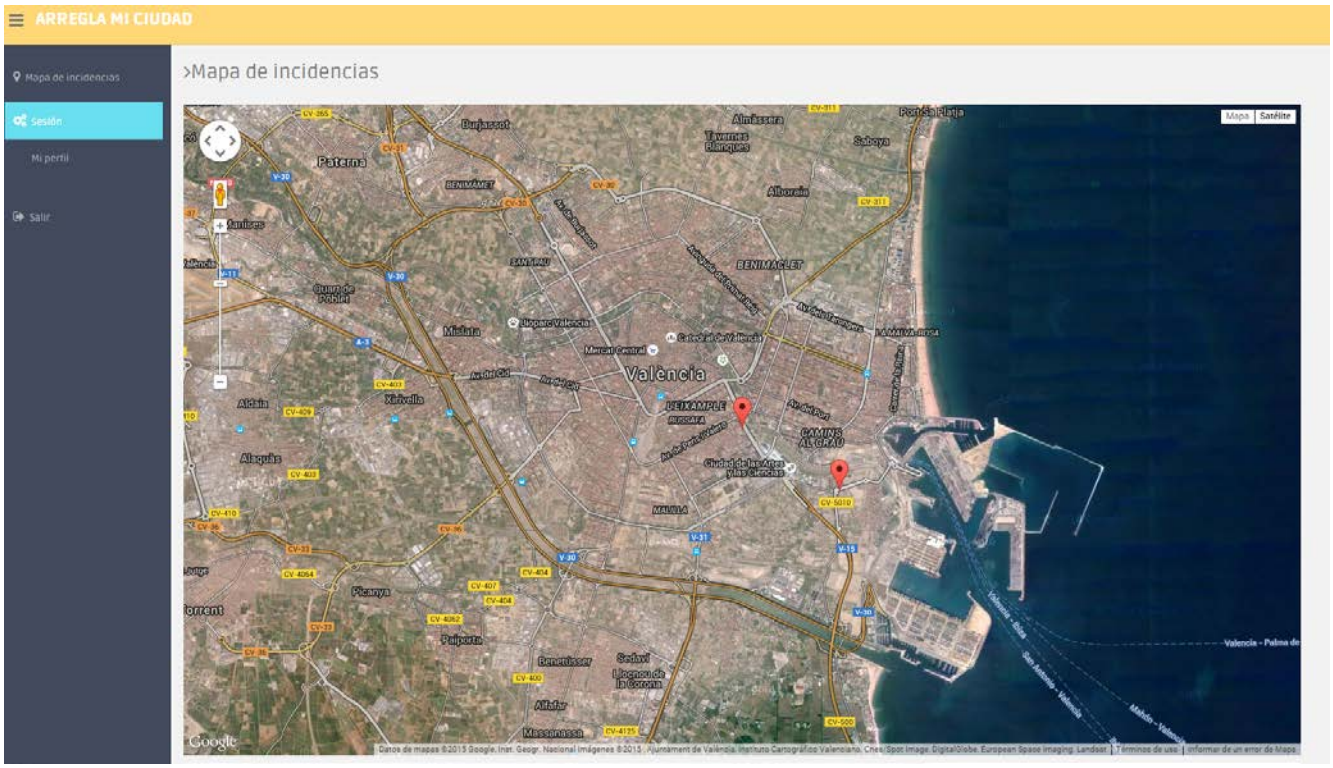
6.1 Demostración

La versión de demostración en vivo de ArreglamiCiudad se encuentra alojada en arreglamiciudad.herokuapp.com

Al acceder a la URL, al no estar logueados, se nos redirige a la página de autenticación:



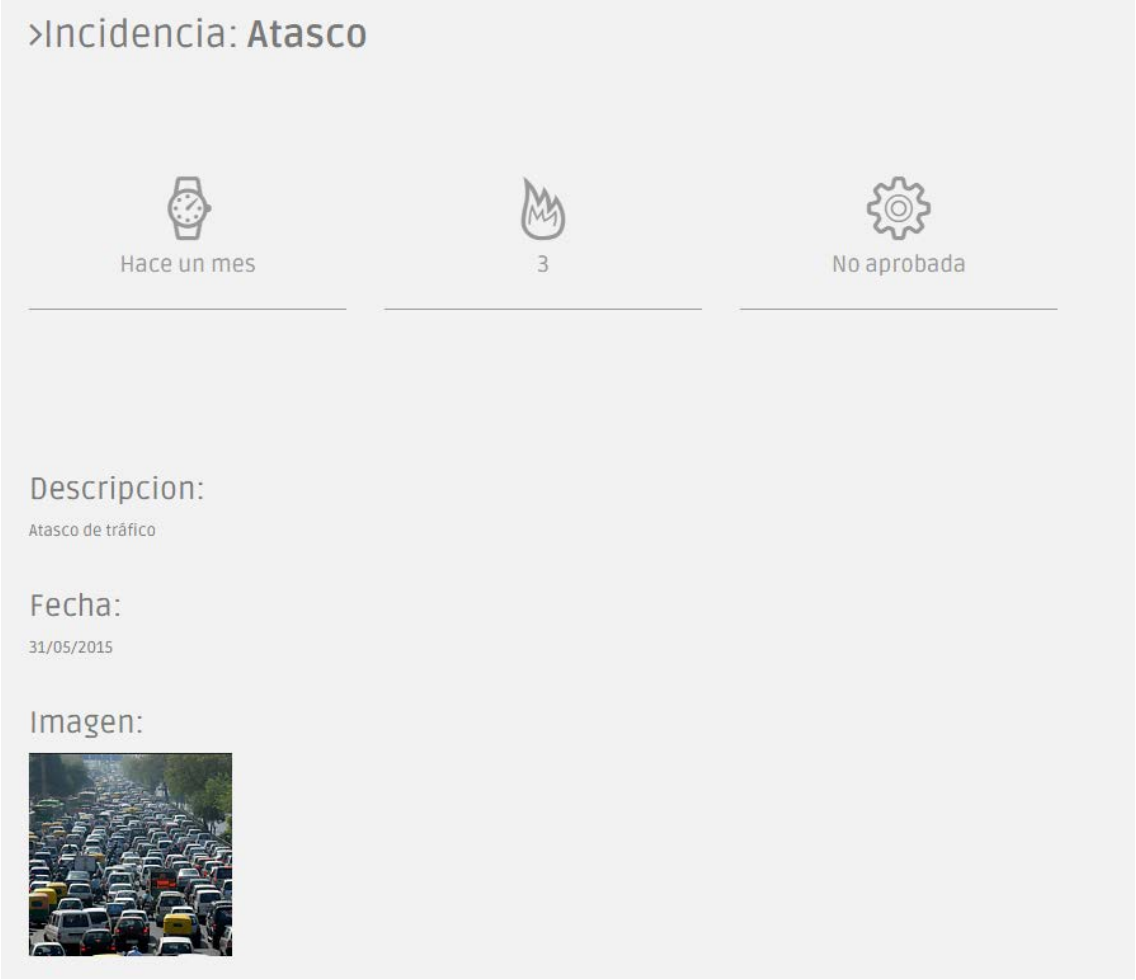
Tras autenticarnos, o registrar un nuevo usuario, se nos redirige a la página principal de la aplicación, un mapa mostrando todas las incidencias no solucionadas:



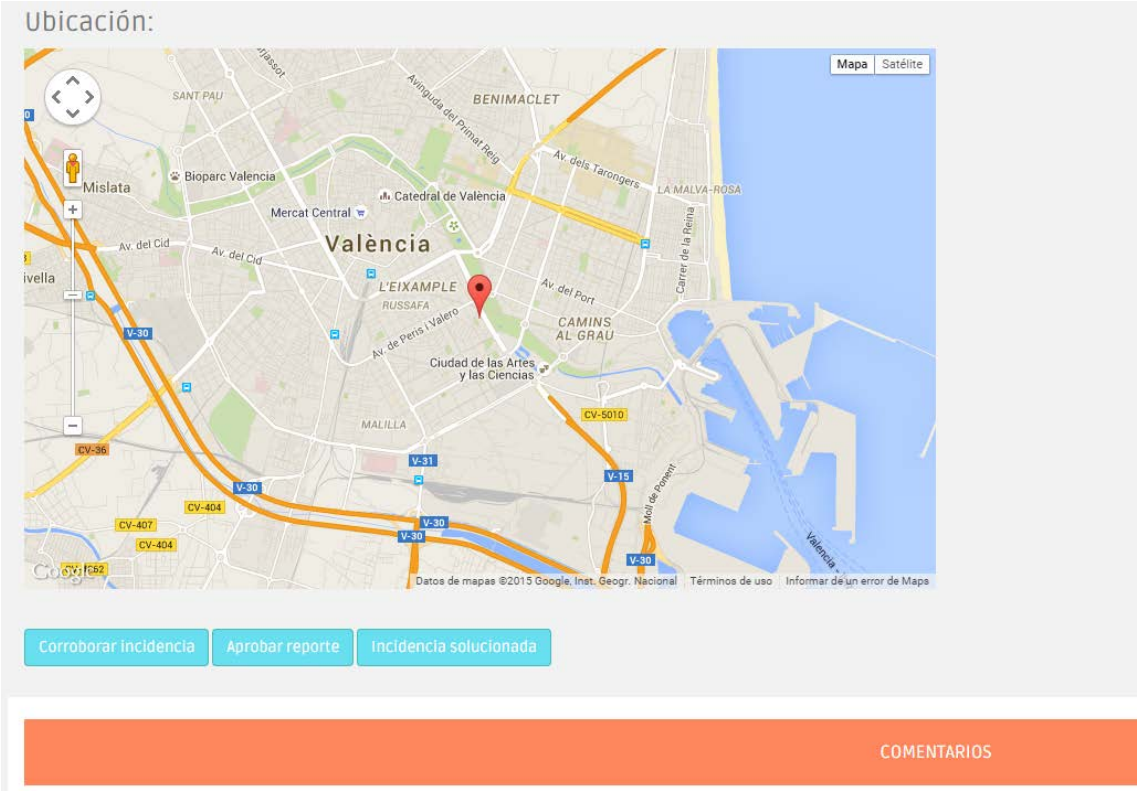
Al hacer clic en uno de los marcadores, se abre una ventana con información sobre la incidencia:

Podemos ir a una página con más información y opciones clicando en 'Ver detalle', desta página se muestran una descripción, fecha del reporte, imagen, mapa con la localización, y lista de comentarios.

También podemos ver estadísticas como cuándo se reportó la incidencia, cuánta gente se ha sumado a la queja y si está o no en proceso de reparación.



Estadísticas, descripción, fecha e imagen.



Mapa con la posición y opciones, los botones 'Aprobar reporte' e 'Incidencia solucionada' sólo aparecen en caso de estar autenticado como operador del sistema.

Lista de comentarios y formulario para añadir uno nuevo.

7. Conclusiones

El software ArreglamiCiudad es un sistema rápido, seguro y con una interfaz atractiva al usuario, y que provee la funcionalidad necesaria para mejorar notablemente el reporte y gestión de incidencias o averías en una ciudad, incentivando la participación ciudadana para tal fin.

En un futuro, se puede analizar si la situación del producto permitiría una ampliación del mismo, por ejemplo modificando la estructura para que una misma instancia de la plataforma pueda diferenciar y gestionar las incidencias de diferentes ciudades.

A modo de resumen, ArreglamiCiudad nos proporciona la capacidad de reportar incidencias o averías desde cualquier sitio, confirmar/votar y comentar las reportadas por otros, ver las incidencias de la ciudad en un mapa con todas y ver el detalle de una incidencia. También provee la posibilidad de que un usuario operador del sistema confirme un reporte y ordene la solución de una incidencia.

Cabe destacar que, por rapidez y sencillez de uso, en ningún caso se obliga al usuario a escribir, más allá de un breve título descriptivo al realizar un reporte.

Por tanto y para finalizar, este sistema, actual, moderno, y con una seguridad y ampliabilidad notables, nos proporciona utilidades que podrían mejorar enormemente la vida de las ciudades, previniendo imprevistos y acelerando la reparación de los problemas.

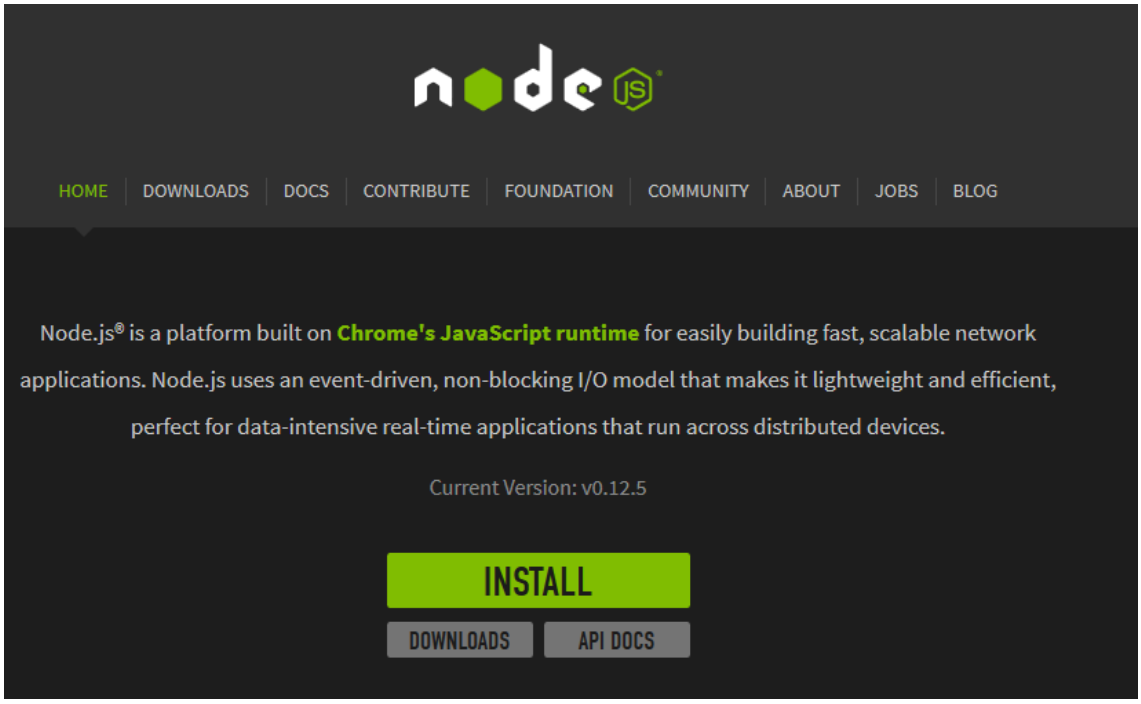
8. Anexo

8.1 Preparación del entorno de desarrollo

Antes de empezar a desarrollar, o mejorar, una aplicación como ArreglamiCiudad, hay que preparar el entorno e instalar todas las herramientas necesarias.

Instalación de node.js

Para realizar la instalación de Node.js, si estamos utilizando Windows, basta con acceder a su página web *nodejs.org* y hacer click en *Install*, la página también dispone de documentación (*API Docs*) con detalle de todas las funciones y posibilidades de node.



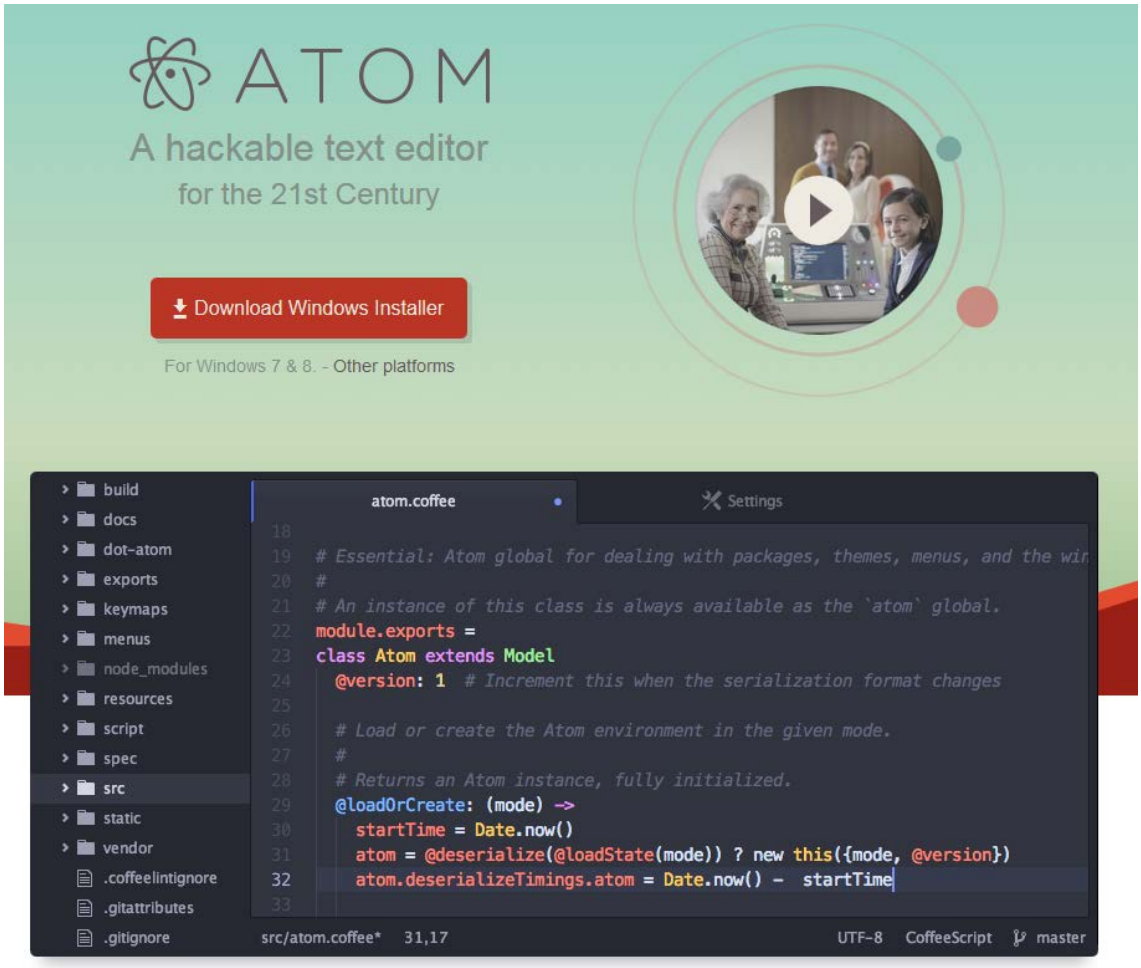
Nodejs.org

La instalación de node incluye el gestor de paquetes npm, por lo que con esta instalación ya tenemos prácticamente todo lo necesario para empezar a desarrollar aplicaciones web.

Editor de texto

Para este proyecto hemos utilizado el editor de texto Atom, debido a su simplicidad, diseño, funcionalidades para el desarrollo de software, y enormes posibilidades de personalización.

Está disponible para descarga en *atom.io:*



Atom.io

La instalación de plugins está disponible en el menú File – Settings – Install. Se recomienda utilizar los plugins *atom-jade* y *highlight-selected* para optimizar el desarrollo.

GitHub

Para realizar la gestión de versiones con un repositorio, como se ha explicado en la versión correspondiente, es necesario disponer del software de GitHub, disponible para su descarga en la página *windows.github.com:*

Windows.github.com

Esta instalación incluye, además de la herramienta con interfaz gráfica para Windows, el software Git y la consola de comandos de Git, con lo que ya estamos preparados para gestionar el repositorio y empezar a hacer *commits*.

Android Studio

Para el desarrollo de la aplicación Android, hemos utilizado Android Studio, la herramienta oficial y recomendada para el desarrollo Android, disponible para su descarga en *https://developer.android.com/sdk/index.html*

9. Bibliografía

StackOverflow.com – Stack Overflow es la mayor comunidad online de programadores, y una plataforma muy recomendada para aumentar la productividad, resolver cuestiones de programación y aprender técnicas nuevas.

https://developers.google.com/maps/documentation/javascript/?hl=es

https://www.npmjs.com/

https://nodejs.org/api/

http://expressjs.com/4x/api.html

https://docs.angularjs.org/api

http://jade-lang.com/api/

http://docs.mongodb.org/manual/

http://mongoosejs.com/docs/guide.html