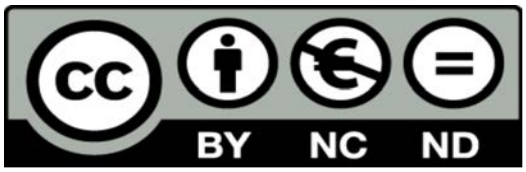
UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título
Despliegue en IaaS de API REST y aplicación móvil de explotación de la API
Autor/es
Víctor Lanchares Fernández
Director/es
Jesús María Aransay Azofra y Eloy Javier Mata Sotés
Facultad
Titulación
Máster universitario en Tecnologías Informáticas
Departamento
Curso Académico
2015-2016



UNIVERSIDAD DE LA RIOJA

Facultad de Ciencias, Estudios Agroalimentarios e Informática

TRABAJO FIN DE MÁSTER

Grado en Ingeniería Informática

Despliegue en IaaS de API REST y aplicación móvil de explotación de la API

Alumno:

Víctor Lanchares Fernández

Tutores:

Jesús María Aransay Azofra

Eloy Javier Mata Sotés

Logroño, Junio,2016

Resumen

El objetivo de este proyecto consta de 2 fases: la primera, hacer el despliegue de una aplicación móvil desarrollada en Phonegap e Ionic compatible con Android para la gestión y visualización de alarmas generadas por un software automatizado; la segunda fase consistirá en el estudio de las dificultades en la traducción de aplicaciones desarrolladas por medio de las tecnologías multiplataforma Ionic y Phonegap de Android a IOs y si es posible hacer esa traducción.

Abstract

The main objective of this project has 2 phases: the first, display an app developed in Phonegap and Ionic compatible with Android for the management and visualization of generated alarms by an automated software; the second will be a study of the difficulties in translating aplications developed by the multiplatform technologies Ionic and Phone in Android to iOS and if posible make that translation.

Agradecimientos

Por todo el apoyo durante la realización de este trabajo me gustaría agradecer su esfuerzo a:

- Jesús Aransay y Eloy Mata: Por su dedicación y buenos consejos.
- Familia y amigos: Por su apoyo en los buenos y malos momentos pero sobre todo a mi padre.

Índice

Introducción

Quantitas Energy http://quantitasenergy.com/ es una empresa tecnológica dedicada a la optimización y mejora de los procesos industriales. También realiza medidas y ensayos eléctricos como Laboratorio de Ensayos acreditado por ENAC. Entre sus productos cuentan con un software de medición y monitorización de diferentes datos y parámetros donde también hay datos meteorológicos.

En la monitorización de estos datos a veces hay algún valor fuera de lo común, que no ha sido bien medido o contiene un valor nulo. Cuando esto pasa se produce una alarma que queda registrada en sus bases de datos en la tabla Alarma con su id, fecha, tipo y campos. Estas alarmas se producen para un usuario, ya que los parámetros que se miden son independientes según el usuario.

Por eso en Quantitas pensaron que sería útil para los clientes hacer una aplicación para móvil que pudiera avisar si se produce una alarma en tiempo real, y que además en la propia aplicación se pudiera ver las alarmas que se han producido e incluso tener gráficos para tener una visión más global de todas ellas.

Planificación

	Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo
8:00-9:00							
9-10							
10-11							
11-12							
12-13							
13-14							
14-15							
15-16							
16-17							
17-18							
18-19							
19-20							

Tabla 1

Horas de trabajo en el TFG

Horas de trabajo en el TFG opcionales, en el caso de que las necesitara

Metodología a seguir

La metodología que se seguirá durante este Trabajo de Fin de Máster no será la típica dividida en 3 fases sino que será una basada en un estudio previo de cómo hacer cada tarea y ver los caminos posibles para hacerlo y su posterior ejecución. En la fase de estudio se analizará la tarea, la viabilidad de su ejecución, todas las maneras posibles y la elección de la más eficiente. Y en la fase de ejecución se pondrán en práctica todas las decisiones tomadas en la fase anterior.

Iteraciones	Duración	Horas planificadas
Tarea 1	3 semanas	110 horas
Tarea 2	5 semanas	150 horas
Reuniones	20 horas	20 horas
Memoria	30 horas	30 horas

Tabla 2

Descripción de las tereas

Tarea 1

Esta tarea consiste en desplegar la aplicación móvil para que sea puesta a producción. Para ello habrá que hacer bastantes cambios tanto en la app como en la zona del servidor ya que todos los cambios que se han ido haciendo en el servidor para completar la funcionalidad de la app se hicieron en una réplica en el entorno de pruebas, y todos esos cambios tendrán que ser hechos también en el servidor de producción. Además de los cambios en el servidor, la app también necesita actualizaciones como mejorar las interfaces, cambiar la IP a la que se tiene que conectar (ya que el servidor cambia) y “resolver problemas” como hacer que una serie de procedimientos se hagan de manera automática.

Tarea 2

Con la idea de que la app fuera accesible a más usuarios se decidió desarrollar la app también en el sistema operativo iOS y ese es el propósito de esta tarea. La app al haberse hecho con Ionic y ser multiplataforma no debería ser tan complicado traducirla como hacerla desde cero. Trataremos de plantear la terea de manera abstracta como una metodología de traducción de aplicaciones de Android a iOS.

Actores del TFM

Víctor Lanchares: Realizador del TFG

Raúl Bonachía Castillo: Cliente y tutor de la empresa.

Jesús María Aransay y Eloy Mata Sotés: Tutor de la universidad.

Análisis de la situación actual

Este Trabajo de Fin de Máster surge a partir de un contrato realizado con la fundación ADER para la empresa Quantitas Energy, en el cual mi tarea asignada era crear una aplicación móvil compatible con Android de gestión y visualización de unas alarmas generadas por una aplicación web desarrollada por ellos. Para poder desarrollar la aplicación creé en una máquina virtual una réplica de su servidor con la aplicación web y todas sus bases de datos. La aplicación web está desarrollada en Symfony y las bases de datos que utilizan son MariaDB y las gestionan con PhpMyAdmin. La app móvil está desarrollada con Ionic que es un framework de creación de aplicaciones móviles multiplataforma mediante HTML 5 y Javascript. Principalmente la funcionalidad de esta aplicación es pedir datos a una API de tipo REST, desarrollada por el alumno de Doctorado Carlos Sáenz Adán. La aplicación a su vez es parte del servidor Symfony, de las alarmas producidas, de todas o de las que aún no han sido vistas, y en la parte gráfica mostrarlas a modo de lista y que a medida que se va deslizando la pantalla hacia abajo se vayan pidiendo más alarmas al servidor. También cuenta con un menú en el que poder elegir ver solo alarmas de un determinado tipo por lo que la vista se actualizará pero la funcionalidad será la misma. Para esta funcionalidad hubo que hacer algunos cambios en la parte del servidor, en la parte de persistencia y en la de lógica de negocio. Otra pequeña funcionalidad es un panel de ajustes en el que básicamente se puede elegir qué información quieres que se muestre en el listado de alarmas. También en el listado si seleccionas una alarma puedes ver esta con más detalle (hora producida, tipo de alarma, fecha…). También se añadió un servicio de login para así añadir algo más de seguridad a la app y que cada usuario tuviera su propia sesión en el servidor. Y la última funcionalidad añadida fue la de visualización de las alarmas por medio de gráficos. Para poder mostrarlos se utilizó el plugin tc-angular-charts el cual era muy fácil de instalar a través de los comando de Cordova y fácil de utilizar ya que cuenta con su propia página web y un tutorial en el que explica todo lo necesario para su uso. Se añadieron 4 tipos de gráficos, uno de barras, otro de sectores, otro lineal y otro discreto. El de barras y el lineal muestran para cada día de un mes elegido el número de alarmas que se han producido independientemente del tipo de alarma. El de sectores representa el porcentaje de alarmas producidas de cada tipo desde que se tienen datos y el discreto muestra para un mes elegido si para cada día se ha producido un tipo de alarma o no diferenciando por el tipo de alarma.

Después de estas funcionalidades también se añadió un botón de compartir el listado de alarmas con distintas redes sociales. Se añadió a través de un plugin que detecta las aplicaciones que tienes instaladas en tu dispositivo y te las muestra y cuando entras en una de ellas en el mensaje a mandar ya está copiada toda la información del listado de alarmas.

Primera fase

Fase de estudio

Para desplegar una aplicación móvil en producción primero tenía que cumplir una serie de condiciones: la primera que cumpliera todos los requisitos exigidos por el cliente, que aparte de cumplir estos no tuviera fallos y su ejecución se hiciera correctamente, que el servidor pudiera responder a las peticiones de la aplicación, que este también pudiera enviar notificaciones push, que además contara con las bases de datos actualizadas para poder guardar información adicional sobre los usuarios y que tuviera implementados los métodos para poder hacer pruebas de generación de alarmas y así poder enviar notificaciones push.

Una vez definidas las condiciones a cumplir por la aplicación y el servidor hubo que pensar la manera de cómo hacer este despliegue. Estuvimos barajando varias opciones pero con la que nos quedamos fue con la siguiente. Lo primero que habría que hacer sería una automatización de varias tareas de la aplicación como el registro de los dispositivos para que así pudieran recibir notificaciones push. Una vez hecho esto lo siguiente que se haría sería volcar los cambios hechos en mi servidor de pruebas al servidor de producción de Quantitas Energy (esto se haría con un merge de sus ramas alojadas en BitBucket.org). Luego habría que actualizar las bases de datos y con el servidor de producción actualizado se pasaría a hacer las pruebas de funcionamiento. Estas primero se hicieron desde un emulador y cuando funcionaran correctamente se generaría una APK para comprobar que la aplicación funcionase fluidamente en un dispositivo real y que las pruebas se superaban con éxito. Una vez que estas tuvieron la aprobación del cliente lo que se decidió hacer fue pasar a trabajar otra vez al servidor de pruebas, ya que no es una buena práctica desarrollar en el servidor en producción, mientras se iban corrigiendo los errores que tuviera la aplicación y nuevas mejoras que iba aportando el cliente.

Para cada una de estas mejoras una vez que se terminaran lo primero que se haría sería una prueba en el emulador. Si esta era exitosa se hacía un commit del proyecto, se guardaría en un software de control de versiones, se generaría una APK para probarla en un dispositivo real, se harían las pruebas pertinentes y de ser así se daría por terminada esa mejora.

Una vez que estuvieran terminadas todas las mejoras y corregidos todos los fallos, se haría otro merge con la rama del servidor en producción en el caso de que se hubiera hecho algún cambio y se generaría una APK, pero esta vez quitando todos los logs que ayudaban a su debug. Esta APK pasaría unas pruebas más rigurosas, en más y distintos dispositivos y se haría estando conectada desde Wifi y desde datos móviles para ver si aún con una conexión media la aplicación fuera bien. Y si todas estas pruebas funcionasen correctamente se podría dar por terminada esta primera fase.

Puesta en práctica

Como se dijo en la fase de estudio lo primero por lo que se empezaría sería corrigiendo los errores que pudiera haber y automatizando algunas tareas de la aplicación. Pero como errores

importantes no se encontraron se empezó por automatizar el guardado de las claves de los dispositivos móviles.

Para explicar esto creo que antes debería explicar el funcionamiento del envío de notificaciones push ya que es una parte importante de la aplicación. Las notificaciones se mandan a través de Google Cloud Message que proporciona una plataforma de envío de mensajes push. Lo primero que hay que hacer es registrar la aplicación que tenemos en el servidor en el servicio de Google Cloud Message. Una vez que la hemos registrado nos da 2 claves, una que se guarda en nuestro servidor que será para autenticar a nuestra aplicación y otra que la utilizarán los clientes, en nuestro caso la app móvil, que sirve para conectarse con nuestra aplicación. Una vez hecho esto la app, cada vez que se vuelve a ejecutar, se tiene que registrar en el Google Cloud Message y este la devolverá un ID de registro que identifica únicamente a ese dispositivo y, para que la notificación enviada desde el servidor llegue a nuestro móvil hay que decirle a qué dispositivos queremos que lo mande, de ahí la importancia de que el servidor tenga las claves de registro de los dispositivos. Con esto explicado la tarea que tenía que automatizar era que cuando el dispositivo obtuviera la id de registro enviase la clave en una petición post al servidor y en el servidor identificar qué usuario ha hecho esa petición y en la base de datos guardar la id de registro en su fila correspondiente. Pero un usuario puede tener más de un dispositivo al que enviarle las peticiones push, por eso al almacenar en la base de datos compruebo que si el registro es nulo se añade sin más, pero si ya hay algún id se concatenan los 2, separados con una coma.

La siguiente tarea a automatizar está muy relacionada con la anterior ya que es que en el envío de notificaciones push se cojan las id de registro del dispositivo móvil del usuario correspondiente. Ya que en la fase de desarrollo esto era estático, tenía unos id de registro ya definidos y siempre se utilizaban estos independientemente del usuario que estuviera en la sesión. Para hacer esto cuando se produce una nueva alarma se obtiene el usuario de la sesión y con este recuperamos el registro de la tabla con los id de registro. Lo que obtenemos es una cadenas de id´s separados por comas, pero el método de envío de notificaciones necesita un array de id´s de registro, por lo que utilizamos la orden “explode(‘,’,$idregistro)” que lo que hace es separar una cadena por el carácter indicado y los almacena en una array, y es esta la que pasamos a la clase de envío de la notificación push, además de toda la información correspondiente a la alarma producida.

También se barajaron otras posibilidades para el almacenamiento de los id´s de registro y el posterior envío de las notificaciones con los correspondientes id´s. Aparte de la opción elegida también se pensó en que cada vez que se arrancara la app móvil se enviase la petición post con el id de registro y se guardara en una variable de sesión para que luego pudiera ser utilizada cuando se produjera una alarma para ese usuario. Pero había 2 problemas, el primero era que si el usuario arrancaba la aplicación y no ocurría o no realizaba ninguna acción la sesión podía caducar por lo que se perdería el id de registro. El segundo problema hacía que esta solución no fuera viable ya que aunque inicies sesión con el mismo usuario en la app móvil y en la aplicación del servidor el scope* no es compartido, son sesiones diferentes, es decir tienen id´s diferentes por lo que si al iniciar la sesión en la app móvil y guardar el id de registro en una variable de sesión y luego desde la sesión del servidor se intentaba acceder a esa variable sería imposible ya que no comparten el scopey no veríamos nada.

*scope: Almacenamiento de datos o variables solo visibles por ese controlador.

Con las tareas a automatizar ya terminadas pasamos a resolver algunos problemas que tenía la app. El primero fue mejorar las interfaces de las pantallas de los gráficos añadiendo una

pequeña leyenda y haciendo que tuviera una apariencia más agradable. Para el gráfico de sectores se creó un listado con el tipo de alarma y su número de ocurrencias. Para el listado de los tipos junto con sus ocurrencias se utilizó un tag HTML de Ionic muy útil el cual te permite iterar a través de una variable e ir mostrando su contenido en función del tag HTML en el que esté. Este sería el código correspondiente:

```
<ion-item ng-if="muestra" ng-repeat="numtype in numtypes">
        <p>Tipo de alarma {{numtype["id"]}} : {{numtype["1"]}}</p>
  </ion-item>
```

Se recorre la variable numtypes y para cada uno de sus elementos se muestra su contenido en un elemento HTML <p> donde se ve su id y su atributo “1” ya que de donde se extraen es una variable de tipo JSON. El tag “ng-repeat” es muy útil en este tipo de casos ya que facilita mucho el trabajo y ahorra muchas líneas de código.

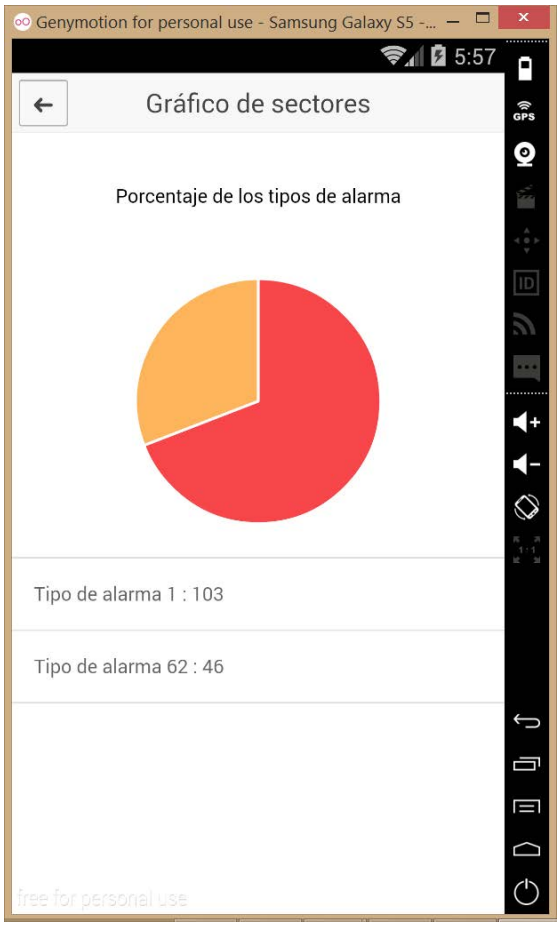


Imagen 1

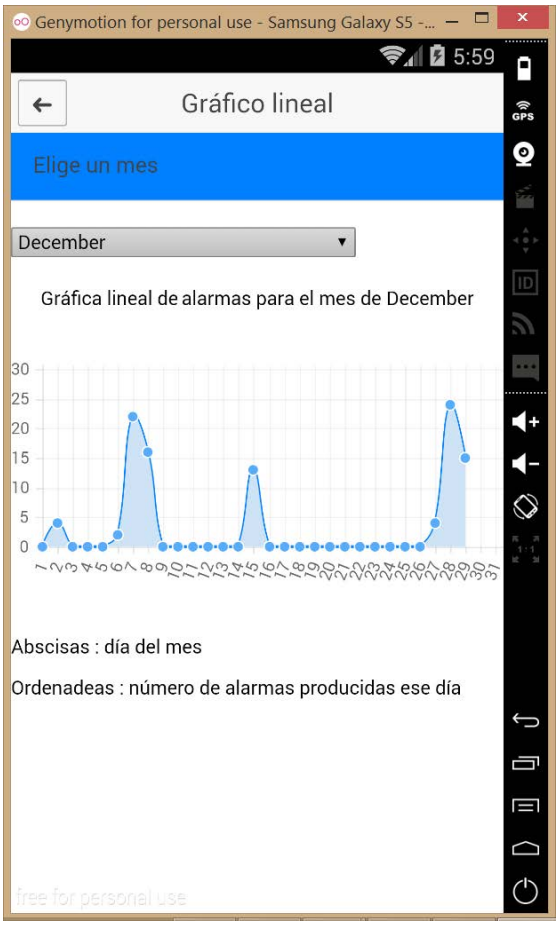


Imagen 2

El siguiente problema a solucionar fue mejorar la interfaz de ajustes ya que esta tenía funcionalidad pero según el cliente no era para nada agradable. Por eso decidí utilizar los componentes que ofrece Ionic que se pueden encontrar en: http://ionicframework.com/docs/components/ y que son muy sencillos de usar además de tener un estilo adecuado, y como los llevaba utilizando en toda la aplicación, encajaban perfectamente. Se añadió una lista con los tipos de alarma y a la derecha de cada elemento de la lista un checkbox para poder elegir si se quisiera recibir notificaciones de alarmas de ese tipo. Se añadieron 4 select menús para poder elegir el rango de horas entre los que se quería recibir notificaciones. Esto que se añadió no contaba con funcionalidad, a petición del cliente, solo está la parte gráfica. Este sería el resultado de la pantalla de ajustes:

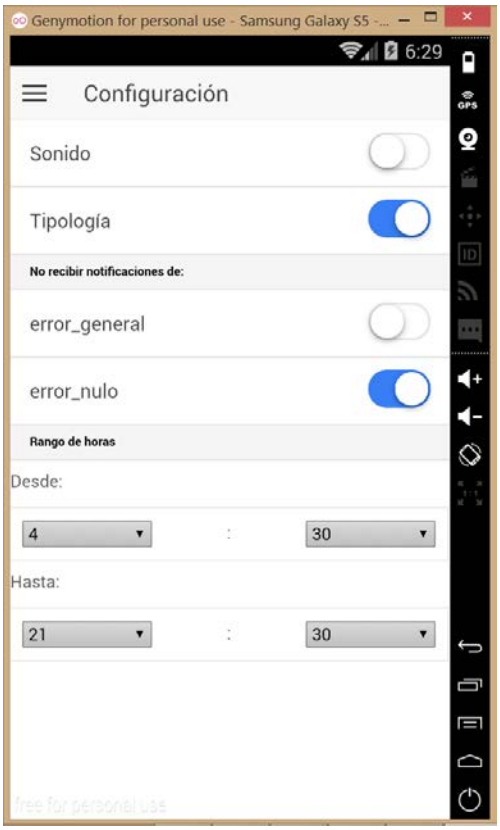


Imagen 3

La siguiente tarea que hubo que hacer fue probar la app en un dispositivo real ya que solo se había probado en 2 emuladores .En uno iba bien, pero en el otro iba bastante lento y nada fluido, por lo que teníamos miedo de que consumiera demasiados recursos y fuera lento en un móvil.

Para poder hacer esto lo primero que hubo que hacer fue hacer que la máquina virtual donde estaba alojado el servidor de pruebas fuera visible desde la red wifi para que el móvil se pudiera conectar. Para ello solo hubo que cambiar el tipo de adaptador de red a una conexión de tipo puente.

Lo siguiente que hubo que hacer fue generar la APK. Android Studio te permite generar un APK firmada pero como no contaba con las claves necesarias para la firma y no iba a ser la versión

que fuera a subir a producción decidí optar por otra opción, ya que Android Studio cada vez que haces un cambio en el código y ejecutas la aplicación, el propio entorno genera una APK sin firmar que se guarda en: Tuaplicación/platforms/Android/build/outputs/apk:

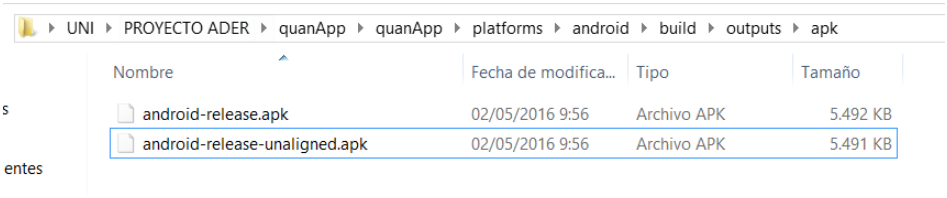


Imagen 4

La APK que genera puede ser “release” o “debug”. Esto depende de cómo tengas configurado Android Studio en el apartado de Build Variants donde puedes elegir entre los 2 tipos:

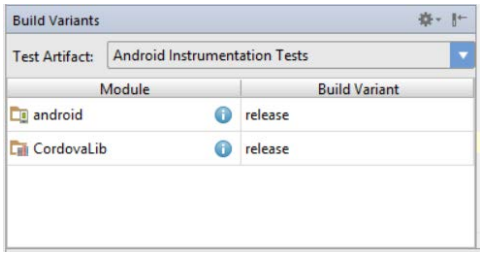


Imagen 5

Cuando ya teníamos la APK solo quedaba instalarla en el dispositivo pero al no estar firmada el teléfono no nos dejaba. Para arreglar esto (en cada móvil cambia, pero de manera genérica) en “Ajustes” suele estar la opción de aceptar APK´s sin firmar o aceptar APK´s de terceros. Una vez que hicimos eso ya pudimos probar la app en un dispositivo móvil y los resultados fueron mucho mejor de los esperados ya que se pudo probar en más de un móvil y en todos la aplicación iba muy bien. Los datos se descargaban muy rápido, los gráficos aparecían de manera muy fluida (todo lo anterior mencionado fallaba en uno de los emuladores) y en general la interfaz se adaptaba muy bien a una pantalla real. Así que pudimos comprobar que en el emulador iba lento por la falta de memoria RAM, ya que al fin y al cabo es una máquina virtual.

Con los problemas resueltos y las pruebas pertinentes realizadas el siguiente paso fue actualizar el código del servidor de producción al que tenía yo en mi servidor de pruebas para poder empezar a probar la aplicación en un entorno real.

Lo primero que hubo que hacer para actualizar el código del servidor fue hacer un commit del código que tenía mi servidor de pruebas y hacer un push a mi rama de Bitbucket, para luego hacer un merge con la rama Master que es la que utilizan en producción.

En general este proceso se realizó sin demasiados problemas. Los más comunes fueron el cambio de algún path y errores en las consultas a las bases de datos, ya que al hacer el commit

y el merge los cambios hechos en las bases de datos de mi servidor no se reflejaban en el commit, por lo que hubo que hacer los cambios a mano a medida que iban surgiendo errores.

Lo siguiente que hubo que cambiar fueron las direcciones a las que se tenía que conectar la app. Ahora ya no hacía falta poner la ip del servidor de pruebas, ahora había que poner la dirección web de Quantitas Energy "http://quantic.quansw.com/app/*". Con estos cambios hechos empezamos las pruebas para testear el funcionamiento de la app en un entorno real, es decir, conectarse o utilizar la app desde cero para ver que el login funcionara correctamente, una vez logeado comprobar que el id del dispositivo se enviaba correctamente y que recibía las notificaciones push. Y todas estas pruebas se hicieron conectado a una red wifi pero también estando conectado solo con los datos móviles del dispositivo, para ver si con una velocidad más baja de internet los datos se seguían viendo tan rápidos y fluidos como antes y efectivamente no había diferencia en el funcionamiento de estar conectado al wifi o estar con datos móviles.

Con las bases de datos y el código del servidor actualizado y los cambios en las direcciones a las que se tenía que conectar la app ya se podía dar por terminado la fase de poner en producción la app, pero como había algunos detalles que querían mejorar y además estábamos cumpliendo los plazos planificados me sugirieron que añadiera un par de mejoras a la app.

Estas mejoras consistían en que cuando una notificación push de que se había producido una alarma llegase al dispositivo, al pulsar en ella fuera directamente a la pantalla en la que se muestra una alarma detallada, la otra mejora era que en las pantallas de gráficos y en la de alarma detallada se pudiera hacer una captura de pantalla para luego poder compartirla en redes sociales.

Para hacer estos cambios volví a usar mi servidor de pruebas ya que como las nuevas mejoras solo afectaban a la app no había diferencia entre utilizar uno u otro y además para mí sería mucho más cómodo, lo único que había que hacer era cambiar las direcciones a las que se conectaba la app.

Para que cuando llegara una notificación fuera directamente al detalle de esa alarma había que modificar el método de recepción de las notificaciones para que cuando se reciba haga la petición correspondiente para obtener la información para el id de esa alarma y con eso obtenido hacer una redirección a la pantalla de la alarma detallada guardando en su scope la información de la alarma.

Pero había un problema y era que cuando la aplicación no estaba corriendo en primer plano (la cierras pero no la paras) y llega una notificación, la aplicación no ejecuta el método para la recepción de esta, por lo que se optó por hacerlo de 2 maneras: si la aplicación estaba en background, no está corriendo en primer plano, cuando se recibiera la notificación se llamaría al método "resume", que lo que hace este método es ejecutarse cuando la aplicación pasa a estar en primer plano y lo que se haría en este método sería mostrar un "alert" indicando la información de la alarma y que fuera a revisarla.

Para el otro caso, cuando la aplicación sí estuviera corriendo en primer plano sí que utilizaríamos el método para la recepción de notificaciones push. El método engloba todos los eventos involucrados con las notificaciones, es decir que se ejecutará también al principio cuando el dispositivo tiene que registrarse para poder recibir las notificaciones push. Por eso hay que diferenciar cada uno de los casos y para hacerlo solo hay que consultar el parámetro "event" del evento que se ejecuta. A nosotros para este caso nos interesa cuando el valor de "event" sea "message", ya que querrá decir que es una notificación push. Este sería el aspecto que tendría el código:

```
window.onNotificationGCM = function(e){

  if(e.event == 'message'){

      <obtener información de la alarma y posterior redirección>

  }

}
```

Una vez que se entra en este método es cuando pasamos a obtener la id de la alarma generada. Esta es la única parte que hubo que tocar en el servidor pero solo fue cambiar un poco el mensaje que se envía cuando se produce una alarma para que en el propio mensaje vaya oculto el id de la alarma, y como es un cambio muy pequeño no habrá ningún problema cuando se suba a producción. Para obtener el id de la alarma solo que hay hacer una serie de operaciones bastante sencillas en el propio mensaje para obtenerlo:

```
      mensaje = e.message.split("/");

    alert("Alarma detectada con id:"+mensaje[1]+ "por favor revísela");

    mensajeAMostrar=mensaje[1];
```

Una vez que lo tenemos hay que obtener la información de la alarma correspondiente a ese id. Para ello hay que hacer una petición a través de los servicios que nos ofrece Ionic y PhoneGap por el cual puedes asignarle a una url un nombre para que así te sea más fácil manejarlo, y además cuenta con métodos para hacer las peticiones "query()". Por tanto nosotros ya teníamos creado nuestro servicio para obtener la información de esta alarma, este sería su aspecto:

```
.factory('Alarm', function ($resource) {

  return{

    queryParams : function(alarmId){ return
$resource('http://192.168.1.56/master_proyect/web/app_dev.php/app/alarms/'+alarmId);}

  }

})
```

En este caso el servicio (o “factory”) tiene un parámetro que luego se añade a la URL que hará la petición.

Una vez que tenemos el servicio creado solo queda llamarlo para obtener la información. Para hacerlo hay que llamar al método “query” mencionado anteriormente pero esta vez pasándole el parámetro del id:

```
$scope.alarmanotificada = Alarm.queryParams(mensaje).query();
```

El resultado de la consulta lo guardamos en la variable “alarmanotificada”.

Cuando ya lo tenemos, hay que hacer la redirección a la pantalla de alarma detallada, para eso tenemos que usar la orden “$state.go(‘app.alarm’)” que hará que la redirección se produzca. Y para pasarle la información al final opté por utilizar una variable global, visible por toda la aplicación, ya que era mucho más sencillo y al no haber concurrencia no iba a suponer ningún problema. Para esto basta con definir una variable como antes pero sin la palabra clave “$scope”.

Pero en todo esto hay una peculiaridad y es que la función “query()” no se ejecuta en el momento que se llama sino en el momento en que se necesitan los datos que esta recoge, por lo que para forzar que se ejecute en el momento de su llamada, la redirección se ejecuta dentro de la propia función a modo de callback. Para entenderlo mejor esta es la manera en la que está hecho:

```
$scope.alarmanotificada = Alarm.queryParams(mensaje).query(

  function(){

       alarmaAMostrar = $scope.alarmanotificada[0];

      $state.go('app.alarm');

   }

);
```

Como podemos ver dentro de la función “query()” definimos una función que lo que hace es obtener los datos de la consulta. Como devuelve un array con solo un JSON esa es la forma de obtenerlo “$scope.alarmanotificada[0]”. Y una vez que tenemos los datos se define la varible global alarmaAMostrar y se hace la redirección.

Encontramos algún problema, ya que mi primera opción fue definir el método de recepción de las notificaciones push en el archivo app.js el cual tiene un método llamado “$ionicPlatform.ready” que se ejecuta cuando la aplicación ya está lista y continúa escuchando todo el rato, por lo que era el lugar idóneo para definir el método de recepción. Pero los problemas venían por 2 razones, la primera era que el método “$ionicPlatform.ready” solo está escuchando cuando la aplicación se está ejecutando en primer plano, cuando deja de estarlo no hay manera de que capte si se recibe o no una notificación. Y el siguiente problema era que en el método “$ionicPlatform.ready” parecía no funcionar la orden “$state.go()” que

servía para la redirección. Busqué toda la información que pude sobre el tema pero no encontré ninguna explicación exacta del problema, aunque a mucha más gente le pasaba, por lo que tuve que buscar una solución.

Al final lo que se hizo fue poner el método para la recepción de las notificaciones push en el controlador de la pantalla de las alarmas no vistas, ya que es la pantalla por defecto y la que más tiempo está viéndose. Además por las pruebas realizadas pudimos comprobar que aunque estuviera en ese controlador si se producía una notificación también era recogida por el método de recepción.

Daba igual poner el método dentro de "$ionicPlatform.ready" o en el controlador de la pantalla de alarmas no vistas ya que si la aplicación no se estaba ejecutando en primer plano en ninguno de los 2 casos la notificación iba a ser detectada, y si sí se estaba ejecutando en primer plano en los 2 casos la notificación iba a ser recibida. Pero la ventaja de poner el método de recepción en el controlador de la pantalla de alarmas no vista es que desde ahí sí que se podía llamar al método de redirección.

La siguiente mejora que me sugirió mi tutor de la empresa fue que en algunas pantallas de la aplicación hubiera la opción de hacer una captura de pantalla para luego poder ser compartida con diferentes redes sociales.

Para comenzar esta tarea lo primero que hicimos fue buscar información de cómo hacer una captura de pantalla con el framework de Ionic y lo único que encontramos es que había que utilizar código nativo del sistema operativo que se estuviera utilizando por lo que seguimos buscando diferentes métodos de cómo hacerlo y encontramos un plugin "CordovaScreenShot" el cual hacía una captura de pantalla.

Este plugin lo que hace es hacer una captura de pantalla y guardarla en la carpeta por defecto donde se almacenan las imágenes según el sistema operativo, en el caso de Android las guardaba en la carpeta "Pictures". Para instalarlo se hacía de manera sencilla con la orden:

```
cordova plugin add https://github.com/gitawego/cordova-screenshot.git
```

La cual añade las dependencias necesarias según el sistema operativo y añade el código necesario para que esté listo para utilizarse. Solo había que añadir el servicio que recoge los parámetros y hace la llamada al propio método del código nativo, este sería el servicio:

```
.service('$cordovaScreenshot', ['$q', function ($q){

  return {

    capture: function (filename, extension, quality){

      extension = extension || 'jpg';

      quality = quality || '100';

      var defer = $q.defer();
```

```
        navigator.screenshot.save(function (error, res){
          if (error) {
             console.error(error);
             defer.reject(error);
          } else {
             console.log('screenshot saved in: ', res.filePath);
             defer.resolve(res.filePath);
             pathImagen= defer.resolve(res.filePath);
          }
        }, extension, quality, filename);
        return defer.promise;
    }  }; }]);
```

Como se puede ver el servicio cuenta con una función llamada capture que es la recoge los parámetros y hace la llamada al método de guardado. Aunque el método se llame “save” hace la captura de pantalla y el guardado, además de tener sus correspondientes callbacks. Se puede ver que recibe tres parámetros, el nombre que es obligatorio y la extensión y calidad que si no decimos nada los pone por defecto. Como el nombre se lo pasamos nosotros como parámetro no hay problema para luego poder recuperarlo.

Una vez instalado el plugin hubo que hacer que funcionara. Habiendo visto la documentación y estudiado cómo funcionaban sus métodos no fue difícil hacerlo funcionar. Lo que había que hacer era añadir el servicio “$cordovaScreenshot” al controlador correspondiente, y hacer la llamada al método teniendo el nombre ya generado:

```
$cordovaScreenshot.capture($scope.imagename);
```

Como no le pasamos ni extensión ni calidad el método lo pone por defecto a ‘jpg’ y ‘100’. Para la generación de nombre seguí un patrón para todas las imágenes y este era poner de nombre el título de pantalla en la que se encuentra, un espacio y la fecha actual, un ejemplo sería:

“Alarma-detallada Fri May 20 2016 04:11:40 GTM-400 (EDT).jpg”

Cuando ya comprobé que las capturas se hacían correctamente hubo que hacer las pruebas pertinentes para ver que todo funcionase bien. Las pruebas fueron bastante bien, las capturas se hacían y se guardaban sin problemas y como el nombre generado se hacía de manera que no hubiera 2 iguales no hubo problemas en el tema de duplicado de nombres. Este sería un ejemplo de una captura y de una captura con gráfico:

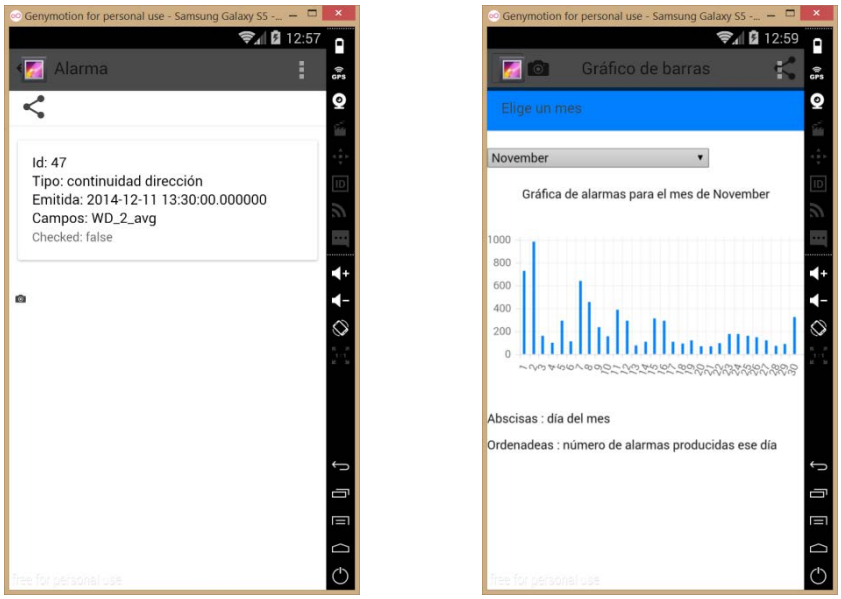


Imagen 6

Una vez que las pruebas fueron pasadas con éxito hubo que hacer que se pudieran compartir con las redes sociales o que se pudieran mandar por correo. El plugin utilizado para esto es el mismo que utilicé anteriormente ya que cuenta con la opción de pasarle un archivo adjunto para poder compartirlo.

Este archivo se le pasa como parámetro en la llamada al método de compartición. Decidí reemplazar el que tenía, el cual compartía los datos de la alarma detallada o de los gráficos en formato JSON, por uno que solo se enviaba la imagen que acabábamos de capturar ya que el cliente me dijo que enviar las 2 informaciones podía ser redundante.

Para hacer la llamada al método de envío hubo que leer la bibliografía para saber qué parámetro era el correcto que había que rellenar. En el repositorio de GitHub venía un ejemplo de cómo compartir solo un imagen que resultó ser de mucha utilidad.

```
<button onclick="window.plugins.socialsharing.share(null, null,
'https://www.google.nl/images/srpr/logo4w.png', null)">image only</button>
```

En el único parámetro que está con valor hay que darle un path válido, ya puede ser local o como en este caso de una imagen de internet, y es el propio plugin el que se encarga de compartir la imagen. En nuestro caso la llamada se hacía así:

<button class="button button-icon ion-android-share-alt "
onclick="window.plugins.socialsharing.share(messagesocialsharing,null,filepath)"/>

Hay algunas diferencias con respecto al anterior. Para empezar el último parámetro que encontramos es el llamado “filepath” que es el equivalente al único parámetro con valor del anterior método y es que si no introduces más parámetros pone valores nulos por defecto.

Y el segundo detalle que se puede observar es que hay un parámetro más llamado “messagesocialsharing” y es que me di cuenta de que si el usuario entraba en la pantalla de una alarma detallada y le daba a compartir con una red social, la aplicación no tendría la captura hecha ya que no le habría dado al botón por lo que no se mostraría nada. Así que la solución a esto fue que por defecto, si no se hacía nada y se le diese directamente a

“compartir” se enviaría los datos de la alarma o de los gráficos en formato JSON (el primer parámetro del método). Si se hiciera la captura de pantalla la variable “messagesocialsharing” pasaría a tener valor nulo, se haría la captura de pantalla y se le daría valor a la variable “filepath” para que pudiera ser compartida y la información no estuviera duplicada como pidió el cliente. Este sería el aspecto del código en cuestión:

```
messagesocialsharing = alarmaAMostrar;

$scope.screenshot = function(){

        $scope.check = true;

        $scope.title="Alarma-detallada";

        $scope.currentDate = new Date();

        $scope.imagename = $scope.title +" "+ $scope.currentDate;

        $cordovaScreenshot.capture($scope.imagename);

        messagesocialsharing = "";

        filepath = "file:///storage/emulated/0/Pictures/"+$scope.imagename+".jpg";

    }
```

Como se puede observar el controlador da por defecto valor a la variable “messagesocialsharing” pero cuando se hace la captura de pantalla pasa a tener valor nulo y al contrario con la variable “filepath” recibe el valor del path de la imagen capturada.

El valor de la ruta absoluta hubo que consultarlo en internet para ver donde Android guardaba las imágenes por defecto y era en: = file:///storage/emulated/0/Pictures/ que es la carpeta “Pictures” o “Imágenes” de cualquier móvil.

Una vez que ya teníamos arreglado el problema de qué compartir en cada momento, llegó la hora de ver que de verdad funcionase. Para ello primero se comenzó con las pruebas en el emulador ya que es mucho más fácil hacerlas y es un entorno de prueba. Estas salieron muy bien. Si no se hacía captura compartía como cuerpo del texto la información en formato JSON y si se hacía la captura de pantalla se adjuntaba una imagen “.jpg”. Este sería el aspecto de las 2 opciones en la pantalla de una alarma detallada:

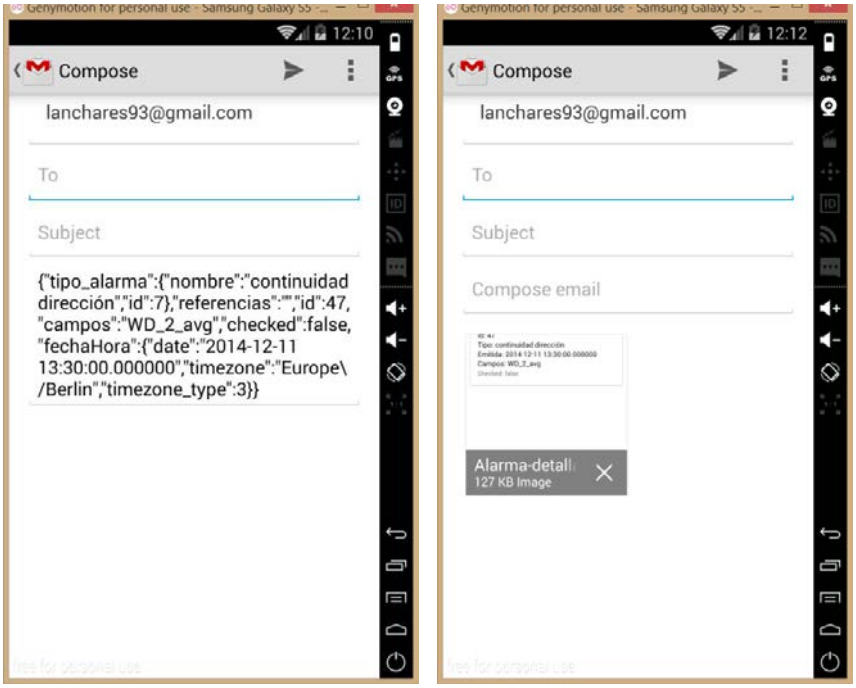


Imagen 7

E hicimos lo mismo para las pantallas de los gráficos:

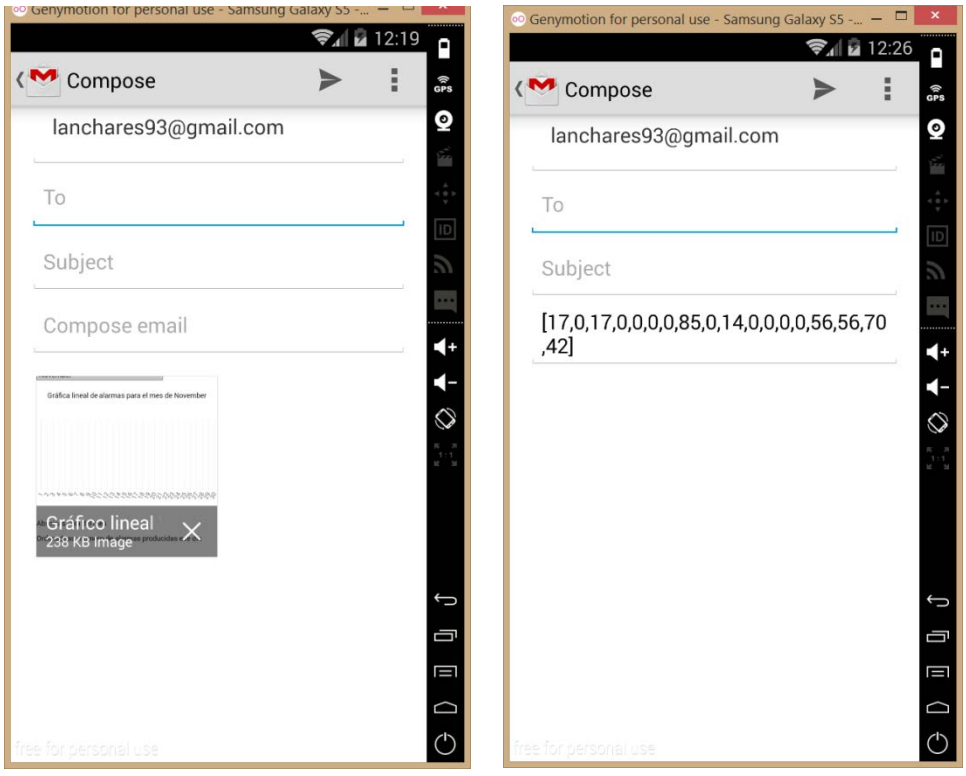


Imagen 8

Como la información de los gráficos en formato JSON no era muy representativa recomendamos hacer la captura de pantalla para obtener una información mucho más clara.

Así que con las pruebas ya hechas en un emulador pasamos a hacer las pruebas en un dispositivo real. Para hacerlo no hizo falta generar una APK ya que cada vez que se hacen

cambios en el código, se guardan y se ejecuta la aplicación se genera automáticamente una APK que se guarda en: <Tuproyecto>\platforms\android\build\outputs\apk

Con la APK instalada en el móvil empezamos las pruebas por compartir la información de una alarma detallada sin haber hecho la captura y efectivamente en el correo (por poner un ejemplo de aplicación) no había nada adjunto y tenía un texto como mensaje con la información en formato JSON. Luego probamos a hacer una captura de pantalla y a compartirla y en este caso también fue todo bien ya que se adjuntaba correctamente y no había nada en el cuerpo del mensaje.

Luego pasamos a hacer las pruebas en las pantallas de los gráficos. Empezamos compartiendo sin haber hecho una captura de pantalla para que en el cuerpo del mensaje apareciera el array con los datos como se puede observar en la foto anterior y así pasó, pero lo que nos interesaba era la captura de pantalla. Cuando hicimos la prueba, la primera vez la captura se hizo bien pero donde debería estar el gráfico no había nada.

Lo hablé con mi tutor de la empresa y decidimos probarlo en diferentes dispositivos de los compañeros y a ser posible de diferentes marcas pero en todos pasaba lo mismo. Era como si el elemento HTML “canvas” no lo detectase a la hora de hacer la captura. Busqué información en internet, en la propia documentación del plugin, en los foros de Ionic y Cordova pero no había nada relacionado con eso. Así es como lucían las capturas de pantalla:

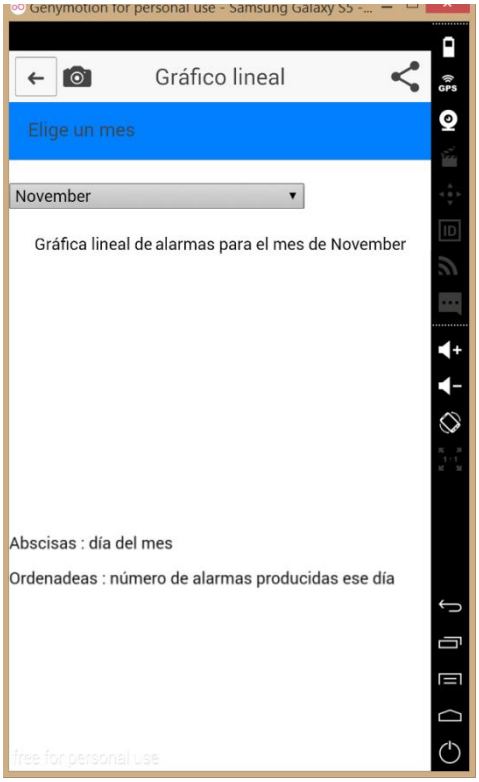


Imagen 9

Así que hubo que buscar una solución. Estuvimos barajando diferentes posibilidades como el buscar otro plugin que hiciera capturas de pantalla o intentar poner el gráfico en otro elemento HTML que no fuera un canvas, pero estas propuestas no eran del todo convincentes.

Entonces se me ocurrió la idea de intentar transformar el elemento HTML canvas en una imagen. Así que indagué por internet cómo poder hacerlo y encontré un plugin que nos solucionaba el problema, ya que a partir de un canvas generaba una imagen de extensión ".png". El plugin se llama Canvas2ImagePlugin que lo que hace es a partir de un canvas guardarte una imagen con su contenido en el almacenamiento por defecto del dispositivo, es el propio plugin el que se encarga de guardar la imagen.

Para instalarlo solo hizo falta una orden, gracias a la ayuda del gestor de paquetes de Cordova:

```
cordova plugin add
https://github.com/devgeeks/Canvas2ImagePlugin.git or phonegap local plugin
add https://github.com/devgeeks/Canvas2ImagePlugin.git
```

Una vez instalado hubo que leerse la documentación pero era muy sencilla ya que solo consta de un método de llamada en el cual solo hay que indicarle el "id" del elemento HTML canvas que se va a tranformar.

Esta sería la llamada al método en cuestión:

```
window.canvas2ImagePlugin.saveImageDataToLibrary(
                function(msg){
                    console.log(msg);
                },
                function(err){
                    console.log(err);
                },
                document.getElementById('myCanvas'),
);
```

Como vemos, aparte del canvas también tiene una función de successCallback y failureCallback en los que podemos indicar lo que queramos.

Una vez que tuvimos instalado y operativo el plugin llegó el momento de hacer las pruebas. Primero las hicimos en el emulador porque si en este no funcionaba era muy improbable que funcionase en un dispositivo real por lo que nos ahorraríamos tiempo. Para las pruebas solo las hicimos en las pantallas de gráficos ya que en la pantalla de Alarma detallada funcionaba todo bien por lo que ahí no se tocó nada. En las pantallas de los gráficos lo que se hizo fue simplemente reemplazar el método de captura de pantalla por el de transformación de canvas a imagen.

Las primeras pruebas fueron bien, la imagen del gráfico se veía bien y se podía entender la información que se representar. Aunque no era una captura de la pantalla lo hablamos mi

tutor de la empresa y yo y quedó contento con el resultado por lo que seguimos con las pruebas. Así quedaría la imagen después de la conversión:

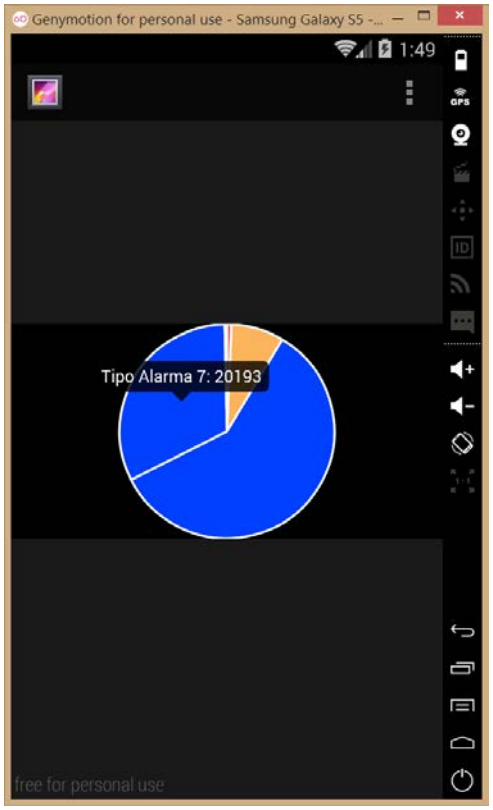


Imagen 10

Es la imagen vista con el visor de imágenes del emulador.

Cuando comprobamos que las imágenes se generaban bien en el emulador decidimos hacer las pruebas en un dispositivo real para lo que obtuvimos la APK generada automáticamente y las probamos primero en mi móvil. La prueba fue bien, la imagen era correcta, se almacenaba en la carpeta por defecto, que es la de las imágenes y se veía muy nítida. Por lo que decidimos probarlo en más dispositivos y a ser posible de diferente marca para comprobar que funcionase en todos y así fue.

Como todas las pruebas en dispositivos fueron bien, decidimos pasar a hacer que esa imagen generada se pudiera compartir con las redes sociales. En principio como solo habíamos reemplazado el método de la captura de pantalla no tendría por qué afectar al método de compartición, bastaría con pasarle el path y el nombre de la imagen como hacíamos antes.

Pero había un problema y era que como el plugin generaba el nombre de la imagen de manera automática no había forma de recuperarlo y además el nombre generado era bastante poco intuitivo, un ejemplo: c2i_184201673453.png. Para solucionar esto, la manera más eficiente y que haría el trabajo mucho más fácil que encontré fue modificar el propio plugin para que pudiera recibir un parámetro más y que este fuera el nombre.

Para hacerlo hubo que modificar los 2 ficheros del plugin, el archivo JavaScript y el archivo Java. En el archivo JavaScript es donde se recibe la llamada que se hace en el controlador y lo

que hace es comprobar que si el parámetro no es de succsessCallback o de failureCallback haga la llamada al método del archivo Java, aparte de algunas conversiones del elemento HTML canvas. Los cambios que se hicieron en este archivo fueron cambiar la definición del método del archivo JavaScript que es el que llamamos desde el controlador, en el que añadimos un parámetro más. Aquí vemos como era antes y como quedó después:

saveImageDataToLibrary:function(successCallback, failureCallback, canvasId) {…}

saveImageDataToLibrary:function(successCallback, failureCallback, canvasId,filename) {…}

Y el otro cambio en este fichero fue en la llamada al método del archivo Java donde también añadimos un elemento más al array que se le pasa y que contiene los parámetros. Aquí vemos como era antes y como quedó después:

return cordova.exec(successCallback, failureCallback,
"Canvas2ImagePlugin","saveImageDataToLibrary",[imageData]);

return cordova.exec(successCallback, failureCallback,
"Canvas2ImagePlugin","saveImageDataToLibrary",[imageData,filename]);

Ahora pasemos a mencionar los cambios en el archivo Java el cual se encargaba de hacer la captura y de guardarla. Los parámetros están almacenados en un array llamado “data” y como ambos son de tipo “String” para acceder a ello el plugin utiliza la orden “optString(<índice>)” así que hubo que añadir una nueva línea para que recogiera el nuevo parámetro y lo almacenase en una variable. Este sería el aspecto:

String base64 = data.optString(0);

String filename = data.optString(1);

Lo siguiente que hace el plugin es la transformación del elemento HTML canvas a una imagen pero aquí no hubo que hacer ningún cambio ya que no nos afectaba para nada. Una vez que hace la captura llama al método de guardado de la imagen (ya que va en un método aparte) en el que originariamente solo se le pasaba la imagen pero nosotros lo modificamos para que aceptase el nombre de esta como un parámetro más:

File imageFile = savePhoto(bmp,filename);

El método lo que hace es devolver una imagen y posteriormente comprueba que esta no sea nula y en de no serlo actualiza la galería y escribe por consola el String equivalente a la imagen:

```
if (imageFile == null)

        callbackContext.error("Error while saving image");

// Update image gallery

scanPhoto(imageFile);

callbackContext.success(imageFile.toString());
```

Pero veamos cuales fueron los cambios que hicimos en el método “savePhoto”. Hubo que modificar la cabecera del método para que pudiera recibir un parámetro más:

private File savePhoto(Bitmap bmp,String filename) {…}

Y por último en la orden de guardar la imagen solo hubo que reemplazar el nombre generado automáticamente por el plugin por el que habíamos introducido nosotros como un nuevo parámetro:

File imageFile = new File(folder, filename+ ".png");

Una vez que el plugin ya podía recibir el nombre que se le iba a dar a la imagen generada ya podíamos crear el path para pasárselo al método de compartición con las redes sociales. El nombre generado era el mismo que para la captura de pantalla, el título de la pantalla correspondiente seguido de la fecha actual.

Cuando ya tuvimos todo esto listo llegó la hora de hacer las pruebas. Como siempre empezamos primero en la pruebas en el emulador, que como ya he explicado es el entorno de pruebas y es mucho más fácil hacerlas y fueron bastante bien ya que como lo único que habíamos hecho era reemplazar el método de generación de la imagen y el path era el mismo (ya que tanto como el método de la captura de pantalla como el de conversión de canvas a imagen guardaban las imágenes en el mismo directorio), para el método de compartición este cambio había sido totalmente transparente. Este sería el aspecto del envío de un correo con la nueva imagen que habíamos generado adjuntada:

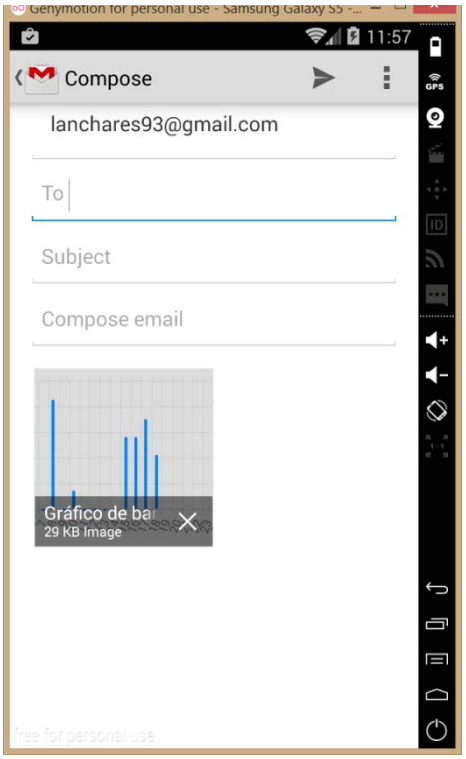


Imagen 11

Como las pruebas en emulador fueron bien comenzamos las pruebas en un dispositivo real. Creímos que estas iban a ser correctas ya que el método de conversión del elemento HTML

canvas a imagen funcionó la primera vez que se probó y el de compartir con las redes sociales ya sabíamos que funcionaba. Y efectivamente las pruebas realizadas fueron un éxito ya que las fotos se adjuntaban correctamente y si hacías el envío el correo recibido tenía la imagen.

Pero como ahora en las pantallas de gráficos enviábamos solo el gráfico en sí y no toda la información adicional que se podía ver en la captura de pantalla, el tutor de la empresa y yo estuvimos decidiendo si en el método de compartición debíamos añadir también los datos en formato JSON (estos datos se seguían añadiendo en el caso de que se compartiera y no se hubiera hecho previamente la captura de pantalla).

Al final decidimos añadirlo solo en el gráfico de sectores ya que sin la información adicional del resto de la pantalla era el que más constaba de entender. En el resto bastaba con poner un pequeño texto en el correo o incluso con el asunto creímos que ya sería necesario.

Así que este era el resultado de cómo quedaría un envío de correo del gráfico de sectores:

Imagen 12

Como se puede ver en la imagen mi tutor de la empresa y yo también decidimos qué datos compartir y al final llegamos a la conclusión de que sería mucho más representativo compartir los datos en formato JSON que recibimos directamente del servidor en vez de los que compartíamos antes que eran los datos ya tratados para que el plugin de representación de gráficos pudiera interpretarlo:

Imagen 13

Aunque en el resto de gráficos, si se compartía sin haber hecho la captura de pantalla, los datos que se compartían sí eran los datos modificados para que el plugin de representación de gráficos pudiera interpretarlo:

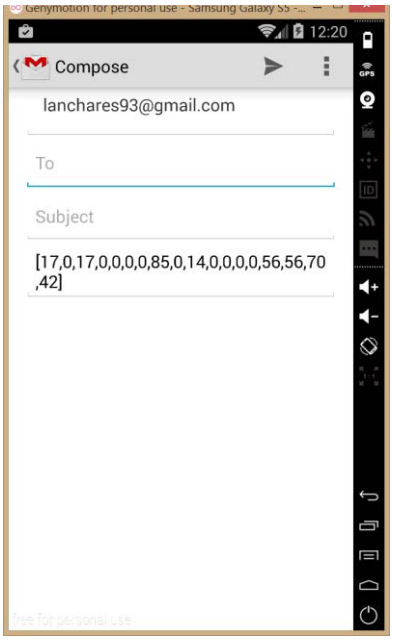


Imagen 14

Estos datos serían los correspondientes a este gráfico:

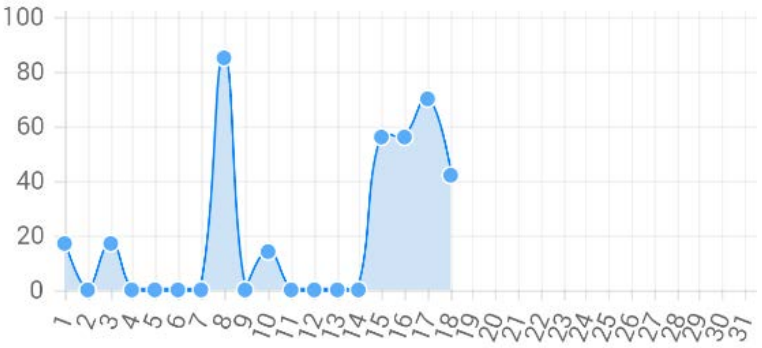


Imagen 15

Una vez que introducimos todos los cambios y mejoras como la automatización del id de registros de cada dispositivo y poder hacer capturas de pantalla y su posterior difusión en las redes sociales, la aplicación ya casi estaba lista para poder ser usada de manera comercial.

Pero aun había detalles que había que pulir y que mi tutor de la empresa me dijo que corrigiera.

Lo primero que me pidió fue que en el menú de ajustes hubiera un menú “select” en el que poder elegir el idioma. Para hacerlo utilice al elemento HTML <select> y para que encajara con

la estética de la aplicación, gracias a ionic y sus estilos pude hacerlo. Este sería el código HTML del menú “select”:

```
<label class="item item-input item-select">
        <div class="input-label">
           Idioma
        </div>
        <select ng-model="miidioma" ng-options="idiomas.indexOf(idioma) as idioma for idioma in idiomas" ng-change="showSelectIdioma(miidioma)">
           <option value="" selected="selected">{{idiomasel}}</option>
           <option ng-repeat="idioma in idiomas" style="width:100px;">
              {{idioma}}
           </option>
        </select>
  </label>
```

Con el elemento HTML “label” envolvemos el menú “select” y le damos el estilo gracias a la orden “class="item item-input item-select"” y le ponemos un título dentro del elemento HTML “div”. Ya una vez dentro del menú “select” asociamos el valor de la opción elegida con la variable “miidioma” gracias al tag HTML de Ionic “ng-model” y con el tag “ng-options” y la orden escrita dentro “(idiomas.indexOf(idioma) as idioma for idioma in idiomas)” obtenemos el índice del elemento elegido. Podríamos elegir el valor del idioma elegido (“castellano”, “inglés”,etc…) pero Ionic modifica los nombres cuando son introducidos en un menú y no es solo texto plano por lo que nos fue más fácil obtener su índice y en el controlador obtener el valor a través del índice obtenido.

Con el tag “ng-change” definimos el evento “on-change” y decimos a qué función tiene que llamar cuando haya un cambio. Por último dentro de las opciones del menú “select” para mostrarlos lo hacemos a través del tag “ng-repeat” al cual hay que pasarle una variable que pueda ser iterable (en este caso la variable “idiomas” que es de tipo array) y un iterador para ir recorriendo la primera variable. Y con la sintaxis “{{idioma}}” mostramos el valor de cada iteración. Este sería el aspecto que tendría el menú:

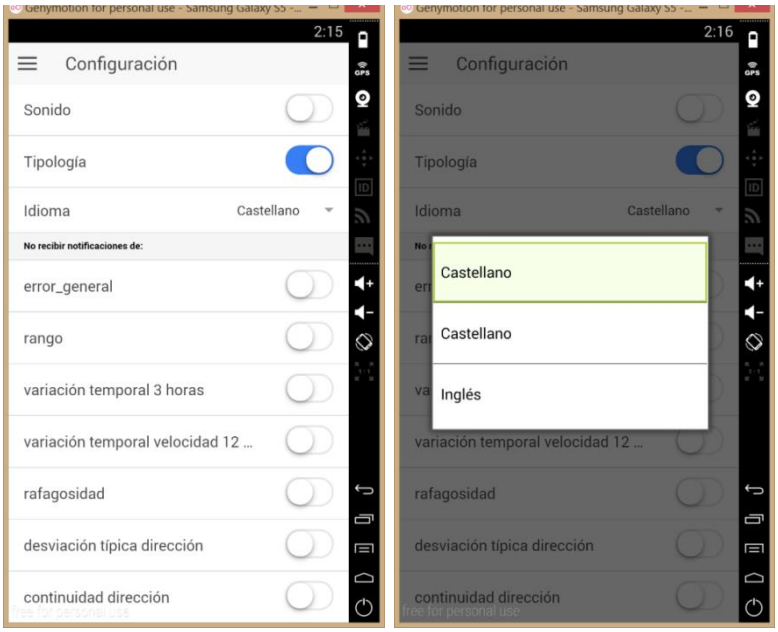


Imagen 16

Ahora pasemos a describir las funciones que realiza el controlador a la hora de elegir el idioma. Como este menú no tenía por qué tener funcionalidad no había que mantener el idioma elegido aunque la aplicación se cerrase, de haber sido así habría recurrido a guardarlo en una base de datos.

Pero al no tener funcionalidad recurrí a guardarlo en una variable global para que el idioma elegido estuviera seleccionado en el menú todo el tiempo que la aplicación estuviera ejecutándose sin cerrarse. Y para guardar el idioma seleccionado en el menú del controlador hay un parámetro que se recibe que es un índice correspondiente al array con todos los idiomas, con este índice obtenemos el idioma y se lo asignamos a la variable global. Este sería el código en cuestión:

```
$scope.showSelectIdioma = function(miIdioma){


    $scope.idiomasel = $scope.idiomas[miIdioma];

    idiomaselecionado = $scope.idiomasel;

}
```

Con este arreglo hecho sólo quedó hacer un par de mejoras para que la aplicación estuviera lista para poder entregársela al cliente.

La primera mejora fue realizada en la pantalla de los gráficos, esta mejora consistía en ir mostrando los menús y botones a medida que se iban introduciendo los datos. Esto se hizo así ya que el tutor de la empresa y yo creímos que iba a ser más cómodo, daría lugar a menos equivocaciones y así se daría tiempo a que el gráfico se fuera cargando.

Empezaré explicando la pantalla del gráfico de sectores ya que es la más sencilla. El aspecto inicial de esta pantalla luce así:

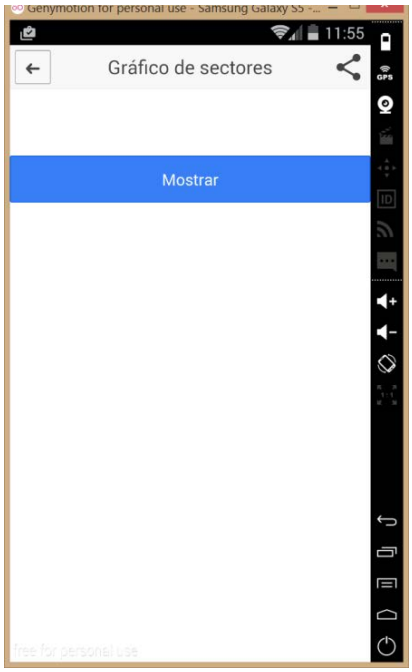


Imagen 17

Solo consta de un botón el cual se ocultará cuando se pulse para dejar paso al gráfico en sí:

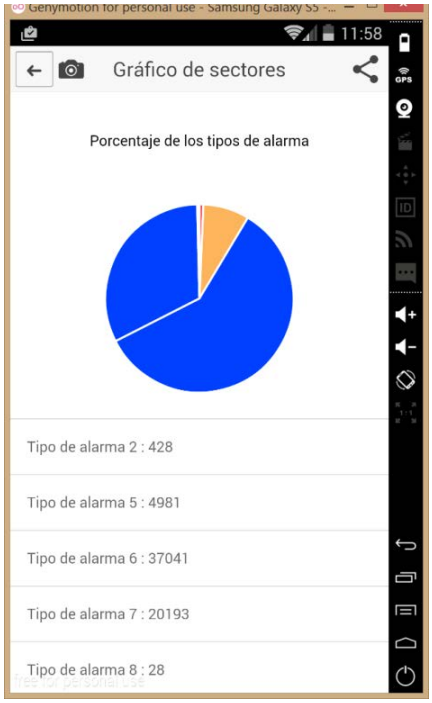


Imagen 18

Para hacer esto utilizamos el tag de HTML de Ionic “ng-if” al cual se le pasa un booleano, que también puede ser una variable, y en función del valor de esta se mostrará o no:

<button ng-if="muestraboton" class="button button-block button-positive" ng-click="cambia()" type="submit">

Mostrar</button>

Por defecto la variable “muestraboton” está a “true” cuando se entra en la pantalla y cuando se pulsa se llama a la función “cambia()” que hace lo siguiente:

```
$scope.cambia = function(){
   $scope.muestra=1;
   $scope.muestraboton = 0;
}
```

Iguala la variable “muestraboton” a cero, ya que en Ionic para definir booleanos la manera de hacerlo es mediante 0 o 1, e iguala la variable “muestra” a uno que es la que está asociada al elemento HTML canvas donde se pinta el gráfico, al título de este y a la lista con los valores de cada tipo de alarma:

```
<p ng-if="muestra" align="center">Porcentaje de los tipos de alarma</p>

<canvas id="myCanvas" ng-if="muestra" tc-chartjs-pie chart-options="options" chart-data="data" ></canvas>

<ion-item ng-if="muestra" ng-repeat="numtype in numtypes">
    <p>Tipo de alarma {{numtype["id"]}} : {{numtype["1"]}}</p>
</ion-item>
```

En las otras pantallas de los gráficos el proceso seguido fue bastante parecido pero como había que elegir el año y el mes había un paso más intermedio antes de mostrar el gráfico.

El aspecto de la pantalla por defecto sería el siguiente:

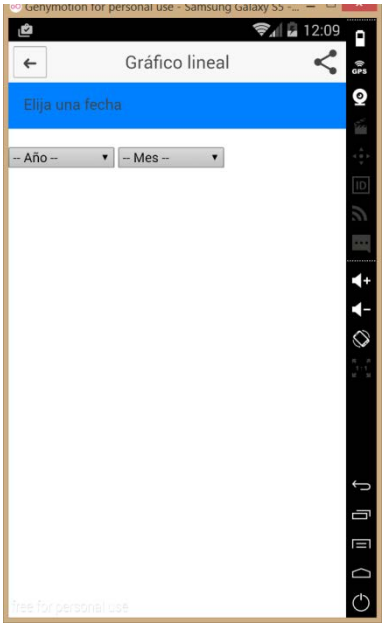


Imagen 19

Una vez que se han escogido los valores de los 2 menú “select” estos no desaparecen porque el tutor de la empresa y yo creímos que no sería muy práctico ya que se podría haber equivocado y que así pudiera reelegir pero sí que aparece el botón de mostrar como en la pantalla del gráfico de sectores. Este sería el aspecto de la pantalla:

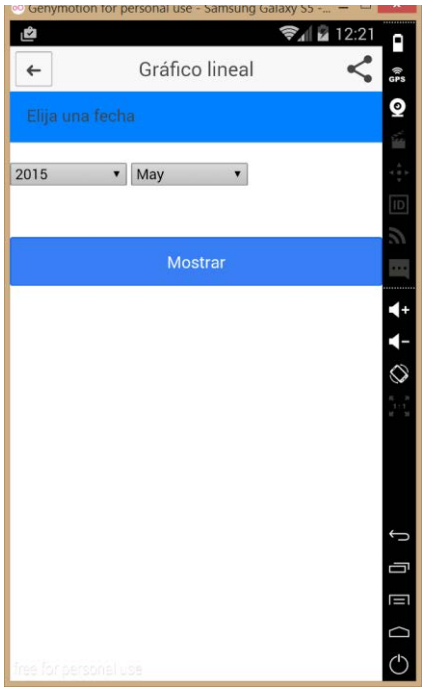


Imagen 20

Y cuando este se pulsa el resultado es el mismo que en la pantalla del gráfico de sectores y luciría así:

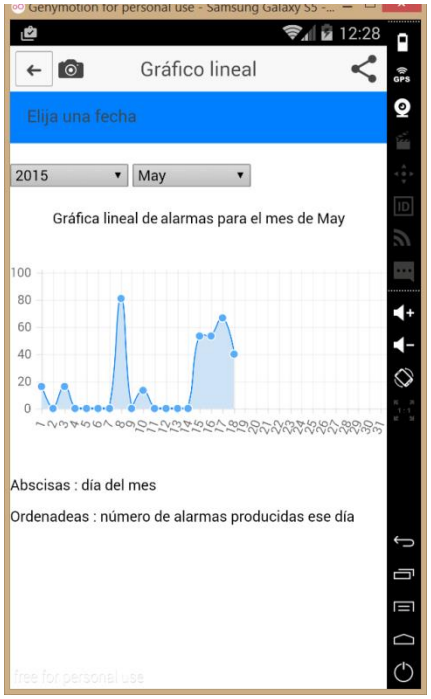


Imagen 21

La última mejora que fue introducida estaba relacionada con la captura de pantalla. Y esta consistía en primero añadir un icono representativo, y el elegido fue una cámara:

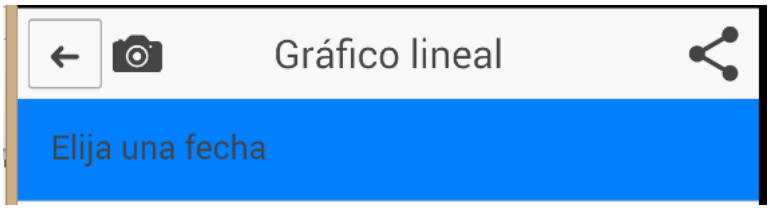


Imagen 22

Este tipo de iconos son fáciles de añadir ya que Ionic ofrece clases CSS que se pueden encontrar en http://ionicons.com/ donde hay una gran variedad de iconos disponibles.

Para añadir esta clase basta con ponerlo dentro del atributo “class” del elemento HTML que quieras modificar:

<button class=" button button-icon ion-ios-camera" ng-click="screenshot()" type="submit"/>

Este icono fue añadido en la pantalla de la alarma detallada y en las pantallas de los gráficos.

Pero en las pantallas de los gráficos se siguió el mismo patrón que con el gráfico en sí, solo se mostrará una vez que se haya pulsado el botón de “mostrar”. Lo hablamos mi tutor de la empresa y yo y creímos que no tendría mucho sentido que hubiera un botón de hacer la captura si no había nada interesante. Para hacer esto volvimos a utilizar el tag HTML de Ionic “ng-if”:

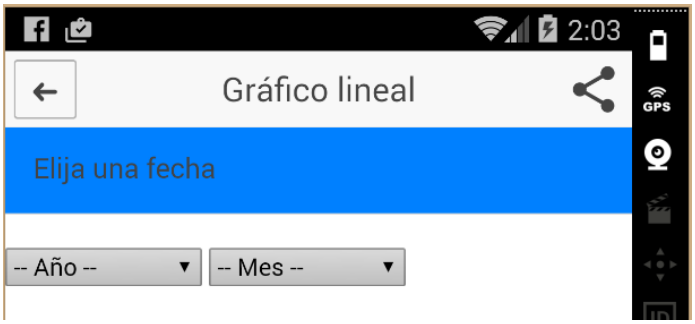


Imagen 23

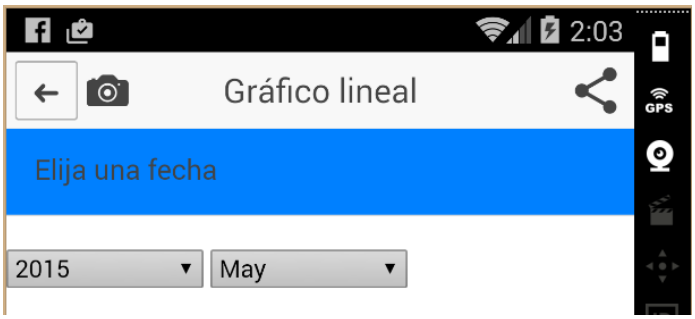


Imagen 24

Y la segunda parte de esta mejora consistía en que apareciera un símbolo de “check” una vez que se hubiera hecho la captura de pantalla. Esta mejora se introdujo en las mismas pantallas en las que se hiciera la captura, ya que el “check” aparece al lado de botón de captura de pantalla. Para hacer que apareciera este símbolo volvimos a utilizar el tag HTML de Ionic “ng-

if" y que en el método de captura de pantalla igualara la variable "check" a 1 una vez que la captura se hubiera hecho:

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

<captura de pantalla>

$scope.check = 1;

};
```

Y así luciría antes y después de haber hecho la captura de pantalla:

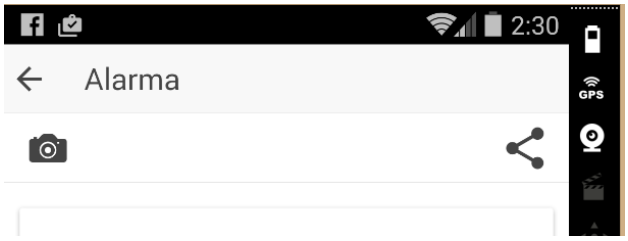


Imagen 25

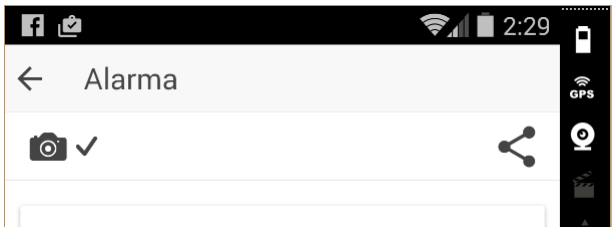


Imagen 26

Para estos cambios y mejoras realizadas las pruebas que se hicieron fueron hechas solo en el emulador ya que los cambios a nivel de la aplicación no eran muy grandes por lo que mi tutor de la empresa y yo supusimos que si funcionaban en el emulador también funcionarían en un dispositivo real.

Las pruebas fueron muy bien y el cliente quedó satisfecho con el resultado así que lo siguiente que se hizo fue hacer el último commit del proyecto. Ya que a lo largo del proyecto, cada día o cada vez que se añadía un gran cambio y las pruebas para ese cambio funcionaban, se hacía un commit y hacía un push a mi cuenta de GitHub para tener un control de versiones y tener el proyecto guardado en la nube por si lo perdía en mi ordenador.

Una vez guardado el proyecto se generó la APK definitiva y fue entregada al cliente para que pudiera probar a fondo todas las funcionalidades. El cliente quedó muy satisfecho con el resultado, por lo que lo único que quedaba por hacer era borrar el repositorio de GitHub ya que era particular mío y no de la empresa. Una vez borrado se creó un nuevo repositorio en mi cuenta de BitBucket en los que estaban todos los empleados de la empresa para que subiera ahí el proyecto y que fuera más seguro ya que este no era público.

Con todas estas tareas hechas ya se pudo dar por finalizado el proceso de despliegue en producción de la aplicación móvil.

Segunda fase

Fase de estudio

Esta segunda fase consistía en primer lugar en hacer un estudio de cómo traducir la aplicación puesta en producción en la primera fase y desarrollada para Android a una aplicación compatible con iOS. Y en caso afirmativo de que los estudios dijeran que era viable hacer la traducción se procedería a ella.

Para hacer el estudio lo primero que hicimos fue dividirlo en tres partes. La primera sería un estudio de viabilidad, es decir si realmente se podía hacer un traducción de una aplicación desarrollada para Android a iOS. La segunda sería un estudio de las maneras en las que se podría hacer esta traducción, en el caso de que se pudiera. Y la tercera sería una vez elegida la metodología a seguir para hacer la traducción cual y en qué orden serían los pasos a seguir para hacerlo.

Estudio de viabilidad. Este estudio había que hacerlo a priori porque teníamos que averiguar si se podía traducir o no. Aunque las aplicaciones desarrolladas con Ionic son multiplataforma y el lenguaje utilizado es JavaScript.

Una vez resuelto el tema de la viabilidad de la traducción comenzamos la segunda parte, la cual consistía en qué maneras había de traducir una aplicación que ya estaba desarrollada para Android y encontramos gran variedad de resultados. Los principales métodos para traducir una aplicación eran los siguientes: uno, crear una aplicación desde cero y empezar a desarrollarla para los 2 sistemas operativos, dos, en la aplicación ya creada añadir algo parecido a un nuevo proyecto para el nuevo sistema operativo y desarrollarlo desde cero, y la última que encontramos fue crear una aplicación desde cero para el nuevo sistema operativo e ir reutilizando código.

La primera opción no nos pareció viable ya que ya teníamos desarrollada la aplicación para un lenguaje.

La segunda opción podría haber sido válida pero al no tener muchos conocimientos sobre la traducción de aplicaciones desarrolladas con Ionic y al tener un tiempo de entrega algo ajustado decidimos decantarnos por la tercera opción ya que según las comparaciones hechas en diferentes sitios web decían que a lo mejor nuestra segunda opción podría ser más rápida, pero había que tener un conocimiento amplio del tema y que la tercera opción era más fácil de llevar a cabo y además la diferencia de tiempo con respecto a la otra tampoco es que fuera muy significativa.

Por lo que nos decantamos por hacer una nueva aplicación desde cero, reutilizando el código de la anterior. Supusimos que sería bastante útil ya que al estar desarrollada en el mismo lenguaje compartirían muchas similitudes.

Así que comenzamos con el estudio de cómo hacer este tipo de traducción y nos dimos cuenta de que esta nueva aplicación iba a compartir prácticamente todo el código, ya que en un sitio web decían que una manera de traducir la aplicación era crearla desde cero un en la carpeta

“js” (que en la aplicación desarrollada para Android también estaba) copiáramos todo el código y con eso bastaba.

Los resultados de las demás investigaciones que hicimos fueron muy parecidos al anterior (reutilizar el código en la carpeta “js”) por lo que nos decidimos a seguir ese patrón. Al final lo que buscábamos era crear una metodología para la traducción de aplicaciones multiplataforma.

Puesta en práctica

Como ya decidimos en la fase de estudio, la opción escogida fue la de crear una aplicación desde cero y reutilizar el código de la anterior aplicación.

Pero para poder desarrollar una aplicación para iOS hace falta un Mac ya que solo es ahí donde se pueden emular dispositivos iPhone. Y, gracias a mis tutores del TFM pude hacer uso de un equipo de la UR.

Una vez decidida la metodología a seguir y con un Mac disponible, lo primero que tuvimos que hacer fue instalar Cordova e Ionic para poder crear nuestra aplicación, ya que son los que nos proporcionarán todas las herramientas de instalación de plugins, configuración del proyecto para un sistema operativo determinado, etc…

Para instalarlos utilizamos la orden:

```
npm install -g cordova ionic
```

Simplemente con esa orden se descarga e instala Cordova e Ionic.

Una vez instalados pasamos a crear nuestro proyecto, para ello Ionic nos proporciona una serie de comandos muy sencillos para hacerlo:

```
ionic start {appname} {template}
```

El primer parámetro es el nombre de nuestro proyecto y el segundo es el tipo de aplicación que queremos tener. Hay varias opciones pero la que utilizamos en la aplicación original fue “side-menu” la cual hace que la aplicación se estructure sobre un menú lateral. Esta sería la sintaxis final:

```
ionic start quanApp side-menu
```

Una vez que tenemos el proyecto creado hay que configurarlo para hacerlo compatible con un sistema operativo. Cuando se ejecuta la orden lo que hace es crear el esqueleto de ficheros de una determinada manera en función del sistema operativo elegido y poner los ficheros que tienen código nativo en su lenguaje correspondiente. Esta es la orden:

```
ionic platform add ios
```

Con esto ya tendríamos creado nuestro proyecto y si lo quisiéramos emular y ejecutar solo tendríamos que ejecutar las siguientes órdenes:

```
$ ionic build ios
$ ionic emulate ios
```

Pero en nuestro caso utilizaremos otra orden un poco diferente.

Con el proyecto creado primero hubo que comprobar que la aplicación se emulaba correctamente para comprobar que el proyecto se había creado correctamente. Para ello utilizamos la siguiente orden:

Ionic run ios –liveroad --consolelogs

Y pudimos comprobar que se emulaba sin ningún problema. Así que comenzamos el desarrollo de la aplicación.

Como habíamos visto en Internet, en algunos foros decían que copiando el código de la carpeta “js” de Android en la carpeta “js” de iOS ya bastaba, así que lo primero que probamos fue eso pero sin esperanzas de que fuera a funcionar.

Y efectivamente así pasó, la aplicación se ejecutaba pero lo único que aparecía era una pantalla en blanco por lo que decidimos volver a la aplicación original e ir paso a paso.

Para explicar los pasos que seguimos primero hay que explicar cómo se estructura la aplicación. Esta consta principalmente del fichero index.html donde se incluirán las librerías y plugins que instalemos, la carpeta “js” que es la que contiene el código JavaScript y dentro de esta encontramos los siguientes ficheros: app.js es donde se estructura la aplicación, se hacen los enrutamientos y es el primer fichero que se ejecuta cuando se arranca la aplicación; el siguiente es controllers.js que es donde para cada vista se define su controlador; luego encontramos services.js que es donde definimos servicios como por ejemplo conexiones con diferentes url´s o cualquier método que queremos guardar como servicio; y por último dependiendo del plugin hay algunos que tienen que añadir un archivo “.js” dentro de esta carpeta. Y la otra carpeta de importancia es la llamada “templates” que es donde se guardan los archivos “.html” correspondientes a cada vista.

Lo primero que hicimos fue maquetar la aplicación, es decir hacer que todas las vistas de la original se vieran aunque no tuvieran funcionalidad. Para ello había que modificar el archivo app.js para añadir las vistas, en controllers.js añadir el controlador para cada vista (bastaba con dejarlo vacío, ya no tendrían funcionalidad) y añadir los archivos “.html” en la carpeta “templates”. Este sería un ejemplo de cómo añadir una vista y el de un controlador:

```
.state('app.uncheckedAlarms', {
  cache: false,
  url: "/uncheckedAlarms",
  views: {
```

```
      'menuContent': {
        templateUrl: "templates/AlarmsList.html",
        controller: 'UncheckedAlarmsCtrl'
      }
    }
  })
```

.controller('UncheckedAlarmsCtrl', function($scope, $state,UncheckedAlarms, CheckAlarm, Alarm,$http, $ionicScrollDelegate,Types,UncheckedTipo,$ionicPlatform) {… });

Una vez hechos los cambios pertinentes se hicieron las pruebas en el emulador y los resultados fueron buenos ya que todas las pantallas aparecían sin problemas. Por lo que decidimos avanzar al siguiente paso que era hacer que aparecieran las alarmas en el listado de la pantalla "Alarmas no vistas".

Para conseguir esto había que añadir un nuevo servicio, el que hacía la petición al servidor para obtener las alarmas y modificar el controlador para obtener los datos requeridos por el nuevo servicio (la vista no hacía falta modificarla porque se copió tal cual de la original). Este sería el servicio añadido:

```
.factory('UncheckedAlarms', function ($resource) {
  return{
        queryParams : function(rowstart){ return $resource('http://quantic.quansw.com/app/alarms/all?type=&from_date=&to_date=&checked=0&pags=si&firstrow='+rowstart);}
  }
})
```

Esta petición devuelve las 10 últimas alarmas producidas a partir de la fila que se le pasa por el parámetro "rowstart".

Y la manera de llamar a este servicio desde el controlador se hace de la siguiente manera:

```
$scope.alarms= UncheckedAlarms.queryParams(0).query();
```

Lo datos recibidos por el servicio son almacenados en la variable "$scope.alarms".

Con estos cambios hechos se comenzaron las pruebas pero los resultados no fueron los esperados ya que volvía a aparecer la pantalla blanca en el emulador y esto era debido a diversos fallos.

El primero y más importante se debía a que faltaba una librería de añadir, esta librería ya venía descargada e instalada con Ionic pero había que añadirla explícitamente en el index.html:

<script type="text/javascript" src="lib/ionic/js/angular/angular-resource.min.js"></script>

Esta librería permite hacer peticiones a url´s por lo que para nuestra aplicación era esencial.

El siguiente fallo era producido porque para que un controlador pueda utilizar un servicio este debe ser añadido en la cabecera del controlador, así que solo hubo que añadirlo y listo:

.controller('UncheckedAlarmsCtrl', function(UncheckedAlarms){…});

UnchekedAlarms sería el servicio, es una versión reducida del anterior para resaltar cómo se incluye el servicio.

Y el último fallo fue más difícil de detectar ya que al nunca haber programado con un Mac y ser mis primeros días no contaba con una consola de logs en la que poder ver los fallos y lo único que me guiaba era lo que se veía en el emulador. En este caso lo que ocurría era que la pantalla se veía bien pero donde deberían estar las alarmas aparecía vacío, eso quería decir que algo fallaba en la petición. Tras investigar descubrí que faltaba una última cosa por añadir y eran las credenciales ya que al ser una API que requiere de usuario y contraseña para utilizarla había que mandar junto a la petición del servicio unas cabeceras con el usuario, la contraseña y el encoding correspondiente. En un primer momento los datos de la cabecera los rellené de manera estática pero en un futuro el usuario y la contraseña se cogerían cuando el cliente se loguease. Así sería como se añadirían las caberas a la petición:

$http.defaults.headers.common['Authorization']= 'Basic '+ Base64.encode(user+':'+pwd);

Con estos fallos resueltos pudimos realizar las pruebas y pudimos comprobar cómo las alarmas se mostraban de manera correcta:

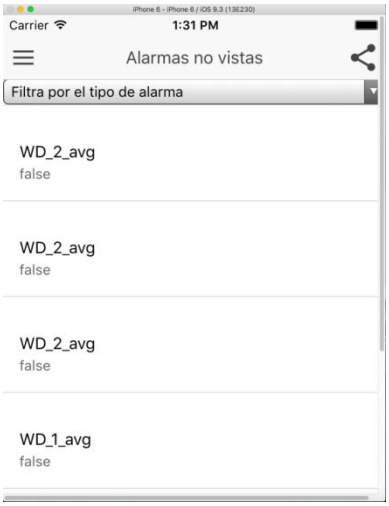


Imagen 27

La siguiente funcionalidad que decidimos implementar fue la representación de los gráficos para lo cual hacía falta un plugin. El plugin utilizado fue el mismo que en la aplicación original

ya que estaba disponible para Android e iOS, además la manera de utilizarlo no cambiaba en lo más mínimo. La instalación se hizo de manera manual ya que en su página web (https://github.com/carlcraig/tc-angular-chartjs) había un sencillo manual y en la aplicación original ya tuve algún problema al instalarlo de manera automática.

Para instalar el plugin lo único que había que hacer era incluir los ficheros "Chart.js" y "tc-angular-chartsjs.js" en la carpeta "js", incluir los plugins en el index.html:

```
<script type="text/javascript" src="js/Chart.js"></script>
<script type="text/javascript" src="js/angular.js"></script>
<script type="text/javascript" src="js/tc-angular-chartjs.js"></script>
```

Y añadir el plugin al controlador: `angular.module( 'app', ['tc.chartjs']);`

Una vez que teníamos instalado el plugin empezamos probando solo en una de las vistas de los gráficos, en la del gráfico de barras en concreto. Y lo que hicimos fue copiar el código del controlador correspondiente de la aplicación original ya que no debería haber ningún cambio y añadir el servicio que pediría los datos para representar el gráfico, así que se hicieron los cambios y se realizó la primera prueba pero el gráfico no aparecía.

Mediante mensajes por consola y "alerts" (mensajes en la pantalla del emulador) pudimos comprobar que la petición sí que se hacía pero debía de haber un fallo en el formateo de los datos, ya que se recibían en formato JSON y había que transformarlos a un array de valores. Así que estuve investigando y el problema estaba en cómo trata iOS las fechas, ya que con el código que teníamos no valía. No podíamos hacer un simple "new Date" de la fecha que recibíamos ya que no lo reconocía como fecha y el bucle que formateaba los datos se paraba ahí y no se mostraba nada. Para que se pudiera obtener la fecha hubo que hacer esto:

```
$scope.t = $scope.datos[1].fechaHora.date.split(/[- :]/);

$scope.d = new Date($scope.t[0], $scope.t[1]-1, $scope.t[2], $scope.t[3], $scope.t[4],
$scope.t[5]);
```

Separábamos la fecha en sus distintas partes y creábamos una nueva a partir de éstas.

Con el tema de las fechas solucionado, la petición se hacía y los datos se formateaban de forma correcta por lo que pudimos hacer la prueba en el emulador y este fue el resultado:

Imagen 28

La siguiente funcionalidad que quisimos implementar fue la de poder compartir datos con las redes sociales. Para ello lo primero que tuvimos que hacer fue instalar el plugin necesario que afortunadamente era el mismo que en la aplicación original ya que era compatible con ambas plataformas.

Para instalarlo fue bastante sencillo gracias a la serie de comandos que ofrece Cordova para la instalación de plugins.

cordova plugin add cordova-plugin-x-socialsharing

Cuando ya lo tuvimos instalado no hubo ni siquiera que cambiar el código ya que el único que había estaba en el HTML de las vistas por lo que ya lo habíamos copiado (en las fotos anteriores se puede ver que el botón ya estaba aunque no tenía funcionalidad). Solo hubo que darle valor a la variable que iba a contener el mensaje a enviar. Este sería el código afectado:

messagesocialsharing = $scope.alarms;

<button class="button button-icon button-clear ion-android-share-alt "
onclick="window.plugins.socialsharing.share(messagesocialsharing)">

Una vez hechos estos cambios se pudieron hacer las pruebas pertinentes y comprobamos que el plugin funcionaba correctamente:

Imagen 29

En la captura no aparecen aplicaciones para compartir porque en el emulador no había ninguna instalada, pero la variable con el mensaje se le pasaba correctamente y en los log pudimos comprobar que no había ningún error.

Ya teníamos cubierta la funcionalidad de compartir con redes sociales, pero no el 100% ya que faltaba poder compartir las capturas de pantalla. Así que decidimos que esa iba a ser nuestra siguiente funcionalidad a cubrir.

Empezamos con la captura de pantalla convencional, no la de convertir un elemento HTML canvas a una imagen. Para ello hicimos como con el plugin anterior y lo instalamos con la ayuda de los comandos de Cordova:

Cordova plugin add https://githun.com/gitawego/cordova-screenshot.git

Cuando ya lo teníamos instalado hubo que añadir el servicio de captura de pantalla en el archivo "service.js" y el método de captura de pantalla en el controlador de la vista "Alarma detallada", ya que era el único sitio donde lo íbamos a utilizar. Se añadió el método pero había que cambiar una cosa, el path en el que se guardaba la foto. En la aplicación de Android el path lo añadíamos nosotros de manera manual, pero para este caso ese path no nos valía así que investigué dentro del propio plugin y encontré que dentro del servicio que habíamos añadido, dentro del callback de éxito devolvía una variable que contenía el path donde se guardaba la captura. Así que lo que hice fue obtener ese path y darle ese valor a la variable global que se le pasará al método de compartición con las redes sociales:

```
console.log('screenshot saved in: ', res.filePath);

        filepath = res.filePath; <variable global>

        defer.resolve(res.filePath);

        pathImagen= defer.resolve(res.filePath);
```

Una vez que hicimos estos cambios pudimos hacer las pruebas de hacer una captura de pantalla y ver si ésta se compartía con las redes sociales. Los resultados fueron buenos, el path que obtenía del callback era correcto y la imagen se pasaba correctamente al método de compartición:

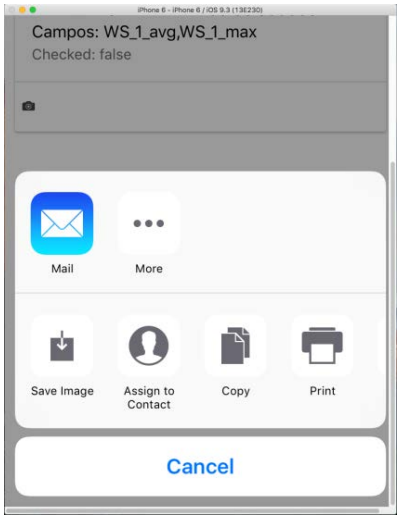


Imagen 30

Como se ve aparecen en la imagen anterior nuevos botones en los que se pueden hacer diferentes acciones con la imagen recién capturada.

Ya teníamos una parte hecha pero aún quedaba la parte de crear una imagen a partir de un elemento HTML canvas, que se hacía en las pantallas de los gráficos.

Para hacerlo lo primero que hubo que hacer fue instalar el plugin. Afortunadamente era el mismo que en la aplicación de Android por lo que la instalación ya sabía cómo se hacía y fue muy sencilla gracias a los comandos que ofrece Cordova:

`cordova plugin add https://github.com/devgeeks/Canvas2ImagePlugin.git` or `phonegap local plugin add https://github.com/devgeeks/Canvas2ImagePlugin.git`

Con el plugin instalado, hubo que empezar a añadir el código necesario para que funcionase. En la parte de las vistas no hubo que hacer nada ya que el código estaba ya añadido, pero en los controladores había que añadir el método de captura y conversión del canvas.

```
window.canvas2ImagePlugin.saveImageDataToLibrary(
    function(msg){
        console.log(msg);
    },
    function(err){
        console.log(err);
    },
    document.getElementById('myCanvas'),
    $scope.imagename
)
```

Sólo con estos cambios el método ya funcionaba, ya que se hacía la conversión del canvas a una imagen y ésta se guardaba, pero aún quedaba un detalle por terminar. Este detalle se trataba de que la imagen que guardábamos no podíamos asignarle un nombre por lo que luego no teníamos forma de recuperarla (como sucedió con el plufgin en la aplicación de Android). Así que en esta me propuse hacer lo mismo. El problema era que el plugin estaba escrito en Xcode y al no haber tenido nada de experiencia con este lenguaje me costó mucho entender su funcionamiento. Una vez que se entendió y se hicieron algunos cambios hicimos alguna prueba pero el plugin modificado daba error de compliación todo el rato por lo que lo dejamos sin modificar. Esto suponía que no teníamos manera de recuperar las imágenes. Pero afortunadamente descubrí que al hacer la conversión y el guardado de la imagen esta además se quedaba en una especie de caché y a la hora de compartirla con las redes sociales se adjuntaba correctamente:

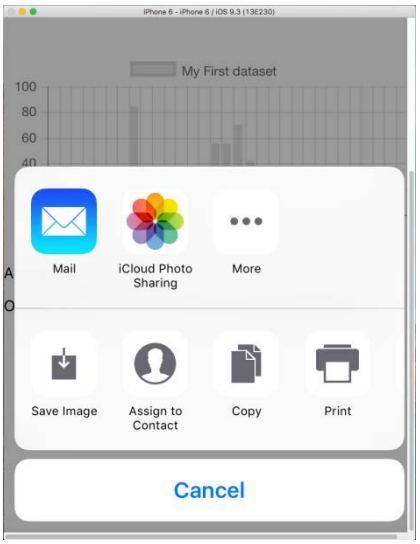


Imagen 31

Las opciones que aparecen son distintas que en la imagen anterior porque se guardaban en directorios diferentes.

La siguiente funcionalidad que se introdujo fue el login. Ya que hasta este momento entrabas directamente a la pantalla de alarmas no vistas y en la cabecera de las peticiones había un usuario y una contraseña por defecto.

Esta funcionalidad fue la más sencilla de introducir, ya que sólo hubo que copiar el código de la aplicación original y funcionó. Se modificaron los archivos “controllers.js” y “app.js”, en el primero se modificó el controlador de la pantalla de login, y el código introducido lo que hacía era recoger los datos del formulario, los ponía en la cabera de la petición y si ésta era correcta guardaba estos datos en la base de datos y redirigía a la pantalla de alarmas no vistas. Y el código introducido en “app.js” servía para comprobar si había algún usuario logueado en la aplicación para ir a la pantalla de alarmas no vistas o a la de login. Lo que hacía el código era consultar en la base de datos si había algún usuario guardado y de ser así pondría los datos en las cabeceras de las peticiones y haría la redirección a la pantalla de alarmas no vistas, y en el caso de que no hubiera ningún usuario guardado le redirigiría a la pantalla de login.

Una vez terminado el login decidimos añadir la funcionalidad de poder filtrar por un tipo de alarma en el listado de alarmas no vistas y en el listado de todas las alarmas y el que las alarmas se fueran cargando a medida que se deslizaba hacia abajo la pantalla. Esto fue bastante parecido de hacer al login ya que casi no hubo problemas, bastó con modificar los controladores de las vistas “alarmas no vistas” y “todas las alarmas” copiando el código. El filtrado de las alarmas fue bastante bien y pudimos hacer las pruebas con buenos resultados:

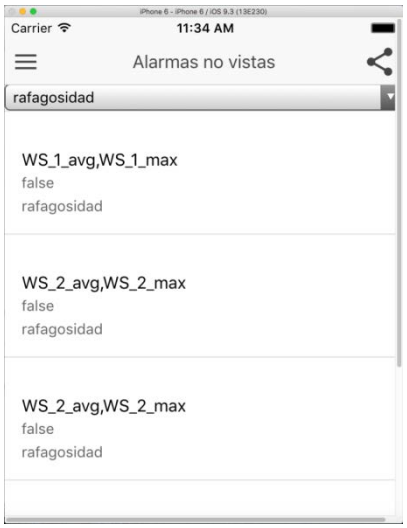


Imagen 32

Y para hacer que las alarmas se fueran descargando a medida que se iba deslizando hacia abajo hubo un problema y era que las alarmas se descargaban bien pero la detección de cuándo se llegaba al final de la pantalla no funcionaba del todo bien y además las alarmas tardaban bastante en cargarse. Estuvimos investigando y esto era producido por el emulador. Algún problema parecido ya nos pasó en el emulador de Android. Pero una vez resueltos estos problemas pudimos dar por finalizada esta funcionalidad.

Y por último antes de intentar añadir la funcionalidad de las notificaciones push, añadimos la funcionalidad de poder mostrar o no la tipología de las alarmas y algunos cambios en las vistas, ya que en el emulador de iOS algunos elementos HTML se descuadraron.

Para la tipología hubo que modificar el controlador de la pantalla de ajustes copiando el código de la aplicación original y comenzar a hacer las pruebas. Estas fueron un éxito ya que la opción de mostrar la tipología o no se guardaba incluso cerrando la aplicación ya que se guardaba en una base de datos y también pudimos comprobar que si no estaba seleccionada no se mostraba la tipología y viceversa.

La última funcionalidad que se intentó añadir fue la de recibir notificaciones push, estuve investigando el funcionamiento de la notificaciones push en iOS y afortunadamente era bastante parecido a cómo funcionan en Android y además el plugin utilizado en la aplicación original también era compatible con iOS. Pero aún quedaba bastante por estudiar del funcionamiento de estas y nos encontramos con un problema difícil de superar. Este que para poder recibir notificaciones push hacía falta tener instalado iTunes y esta aplicación no está disponible en el emulador de iOS y además había que registrar el dispositivo en el servidor de notificaciones push de Apple (se explicará a continuación) y para este registro hacía falta un número de teléfono del que el emulador carecía. Por tanto en vista de que no se iba a poder implementar esta funcionalidad en el emulador y que carecía de un iPhone en el que poder hacer la pruebas mi tutor y yo decidimos que lo mejor era hacer un ejemplo teórico del funcionamiento de estas notificaciones y de cómo se implementaría en un dispoditivo.

Así que pasemos a explicar el funcionamiento y su implementación.

El esquema de las notificaciones push es como se indica en la figura, hay un proveedor que envía la notificación la cual llega a nuestro dispositivo y si pulsamos en ella hace que se abra nuestra aplicación.

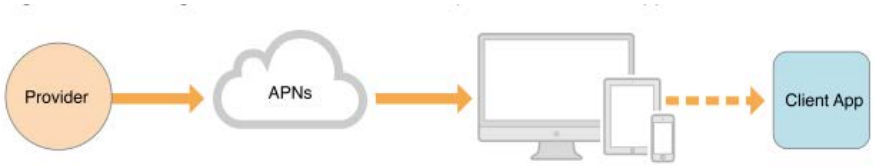


Imagen 33

Al proveedor hay que pasarle un token que está asociado a un número de teléfono y que tiene información que hace que el servicio APNs localice el dispositivo en el que está instalada la aplicación. Además también se utiliza para autenticar el routing de la notificación.

Este token lo envía la propia app, la cual lo recibe cuando se registra en el servicio de notificaciones.

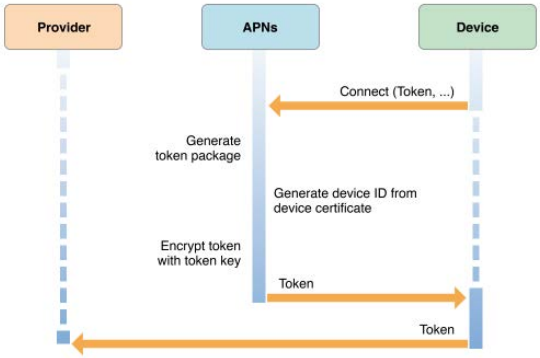


Imagen 34

La notificación que se envía va en formato JSON que contiene los datos que quieras envíar al dispositivo pero también contiene otros como por ejemplo como quieres que sea la notificación, sonido, vibración, silenciosa.

Pero para poder recibir estas notificaciones había que hacer varios cambios, en la parte del servidor y en la parte de la aplicación. Empecemos por la parte de la aplicación.

Esta me resultó algo más fácil ya que utilicé el mismo plugin que en la aplicación original pero aun así cambiaban algunos métodos. Lo primero fue instalarlo para ello utilizamos los comandos de Cordova:

Cordova plugin add https://github.com/phonegap-build/PushPlugin-git

Con el plugin instalado hubo que hacer cambios, sobre todo en el archivo “app.js” ya que el método de recepción de notificaciones no iba a funcionar no lo incluimos en el controlador de la pantalla por defecto aunque es muy parecido al de la aplicación original. Este sería el método de registro del dispositivo aunque no tenga funcionalidad:

```
pushNotification.register(
```

```
        tokenHandler,
        errorHandler,
        {
        "badge":"true",
        "sound":"true",
        "alert":"true",
                "ecb":"onNotificationAPN"
        }
);
```

El “tokenHandler” es el token del que hemos hablado antes que contiene información del dispositivo.

Esto sería para la parte del cliente, pero en el lado del servidor también había que hacer cambios. Estos consistían en el código para enviar las notificaciones y hacer que la conexión fuera segura. Para que la conexión fuera segura lo primero que necesitábamos era una autorización de Apple.

Esta autorización asegura que APNs está conectado a un servidor autorizado por Apple a que puede mandar notificaciones push. Para asegurar esta conexión hay que seguir una serie de pasos.

Lo primero es que cada proveedor (o servidor) debe tener un certificado único y una clave criptográfica privada que se usará para validar la conexión con APNs. El certificado que es proporcionado por Apple lo que hace es identificar la información proporcionada por el proveedor, esta información es principalmente una ID asociada con la aplicación móvil.

El proveedor inicia una conexión segura con APNs mediante una autenticación TLS punto a punto. Después de iniciarse esta conexión recibes el certificado de APNs y lo validas, luego se lo envías a APNs el cual lo valida también. Una vez termina este proceso la conexión segura TLS se establece; y ahora APNs puede asegurar que la conexión se hace con un proveedor legítimo.

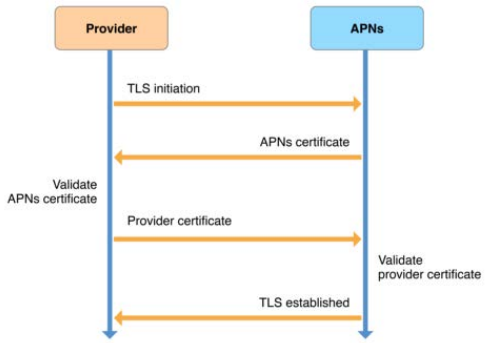


Imagen 35

El dispositivo móvil tiene que seguir un proceso parecido al del proveedor pero se hace de manera automática y una vez que termina el dispositivo recibe un certificado y una clave que se guardan en el kaychain.

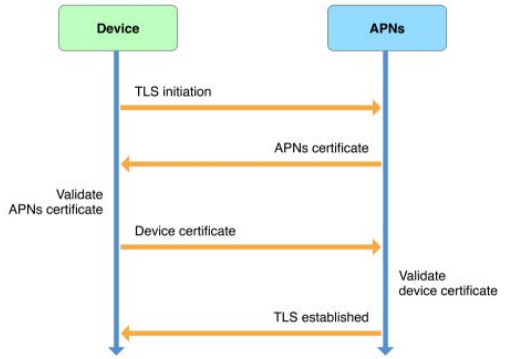


Imagen 36

Y la última parte para completar el envío de notificaciones se trata de la creación de los tokens necesarios para el envío de las notificaciones. El dispositivo se tiene que registrar para poder recibir notificaciones para ello lo que hace es mandar una petición a APNs el cual genera un token único para esa aplicación usando información contenida en su certificado, luego lo encripta con la clave del token y se lo envía al dispositivo como un objeto NSData. Cuando este lo recibe lo tiene que enviar al proveedor en formato binario o hexadecimal. Y cuando el proveedor lo reciba ya estará completo el proceso, pero el proveedor no podrá enviar ninguna notificación a esa aplicación sin el token.

Con el estudio sobre las notificaciones push realizado y las demás funcionalidades incluidas en la aplicación, mi tutor y yo decidimos que ya se podía dar por terminada la fase de traducción de la aplicación desarrollada en Android a iOS.

Revisión de la gestión

Después de haber terminado este trabajo solo queda hacer una revisión de las tareas previstas y las realizadas. Como las tareas no estaban divididas en iteraciones y era mucho más estudio que trabajo práctico fue muy difícil hacer una buena predicción de las horas que iban a tomar cada una de las fases.

Iteraciones	Duración	Horas planificadas	Horas reales
Tarea 1	3 semanas	110 horas	150 horas
Tarea 2	5 semanas	150 horas	100 horas
Reuniones	20 horas	20 horas	20 horas
Memoria	30 horas	30 horas	35 horas

Tabla 3

Desviaciones

La desviación más significante es la correspondiente a la tarea1, ya que creímos que nos iba a costar mucho menos hacer su despliegue, pero el cliente durante el proceso de despliegue quiso añadir nuevas funcionalidades lo que retrasó bastante su finalización.

La siguiente desviación importante es la que corresponde a la segunda tarea, en este caso costó menos de lo esperado ya que el estudio y la traducción a iOS se hizo de manera muy fluida ya que casi todo el código se pudo reutilizar prácticamente sin tener que modificarlo lo que agilizó muchísimo el proceso.

Conclusiones y trabajo futuro

Lo primero como conclusión profesional mencionar que el cliente quedó muy satisfecho con el resultado ya que se iba a hacer un uso real de la aplicación y porque la metodología de traducción de Android a iOS quedó validada con el caso de uso particular de la aplicación "QuanApp" que al final resultó ser muy eficiente y llevó menos tiempo del planificado. Es necesario notar que la aplicación móvil no hace uso de componentes "dependientes del SO subyacente" (como persistencia o servicios) y es principalmente gráfica porque se centra en la actualización de las vistas lo cual facilitó la tearea ya que si hubiéramos utilizado código nativo la traducción a iOS habría sido mucho más complicado por la falta de conocimientos de XCode.

Para terminar, este trabajo ha sido muy interesante de llevar a cabo ya que realizado tareas que hasta ahora no me había tocado hacer como desplegar una aplicación en producción, tratar con clientes reales y profundizar las tecnologías móviles. Y además he tenido que utilizar nuevos lenguajes que, afortunadamente algunos se estudiaban en el Master como Php y otros que no como xCode. También utilicé nuevas tecnologías al hacer la traducción a iOS. En conclusión este trabajo me ha parecido muy bonito, gratificante y divertido de hacer.

Y como trabajo futuro lo más inmediato será terminar de implementar la funcionalidad en la aplicación de iOS de recepción de notificaciones push ya que no se pudo llevar a cabo porque no teníamos los medios suficientes.

Bibliografía

Recursos Web

http://carlcraig.github.io/tc-angular-chartjs/

Plugin: https://github.com/carlcraig/tc-angular-chartjs

Plugin: https://github.com/EddyVerbruggen/SocialSharing-PhoneGap-Plugin

Plugin: http://ngcordova.com/docs/plugins/pushNotifications/

https://github.com/phonegap/phonegap-plugin-push

Plugin: https://github.com/gitawego/cordova-screenshot

Plugin: https://github.com/devgeeks/Canvas2ImagePlugin

https://thinkster.io/ionic-push-notifications-tutorial

https://www.airpair.com/ionic-framework/posts/push-notifications-using-ionic-framework

Ionic: http://stackoverflow.com/questions/32428856/ionic-controller-and-service-structure

http://ionicframework.com/html5-input-types/

http://stackoverflow.com/questions/32846184/create-ios-datepicker-like-select-in-ionic

http://stackoverflow.com/questions/5324178/javascript-date-parsing-on-iphone

Notificaciones push en iOS: http://stackoverflow.com/questions/30602340/how-to-generate-valid-apns-certificate-p12-for-use-in-gcm-for-ios

Notificaciones push en iOS: https://identity.apple.com/pushcert/

Notificaciones push en iOS: http://quickblox.com/developers/How_to_create_APNS_certificates

Notificaciones push en iOS: https://developer.apple.com/download/

Notificaciones push en iOS: https://help.como.com/hc/en-us/articles/201581452-Generate-a-New-Apple-Push-Notification-Service-APNs-Certificate

http://symfony.com/doc/current/components/http_foundation/sessions.html

PHP Symfonyhttp://stackoverflow.com/questions/7192916/how-do-you-access-a-users-session-from-a-service-in-symfony2

XCode: http://stackoverflow.com/questions/8756683/best-way-to-parse-url-string-to-get-values-for-keys

Ionic: http://ionicframework.com/docs/guide/testing.html

XCode: https://forum.ionicframework.com/t/how-to-submit-ios-app-to-app-store-using-xcode/1795

http://ionicframework.com/docs/guide/publishing.html