Proyecto Fin de Máster /Postgrado MTIC – UIB

LocalCommuter

Gestor Móvil de Transporte Público de Mallorca

Autor: Raúl Muñoz Díaz
Director del Proyecto: Miquel Mascaró Portells
Palma 2012/2013

Índice de contenido

Intención

Se pretende desarrollar una aplicación para la plataforma móvil Android que facilite a los usuarios de transporte público de Mallorca el acceso e interacción con todo tipo de información referente a los distintos servicios públicos de transporte, ya sea [1]

- ~~Bus interurbano (TIB)~~
- Bus urbano (EMT)
- ~~Metro~~
- ~~Tren~~
- ~~Otros:bicipalma~~

Pudiendo consultar información referente a :

- Ubicación de paradas
- Listado de paradas por línea
- Próximo autobús de la línea x en la parada actual (geolocalizada)
- Parada más próxima y su distancia.
- Búsqueda de paradas por nombre

Funcionalidades desestimadas para esta versión de la aplicación por exceder los requerimientos de tiempo definidos para esta titulación:

- Itinerario más óptimo (camino mínimo) para ir de una estación/parada a otra interactuando entre distintos sistemas de transporte.
    - Se trata de una funcionalidad muy interesante y deseable pero su implementación excedería con creces los límites de tiempo del proyecto de postgrado.

- Sugerencia de Rutas entre dos puntos.
- Integración con Puntos de Interés (POIs)

Sólo EMT

Como se puede comprobar, en la versión final del proyecto sólo se ha implementado el sistema de transporte de bus urbano EMT ya que como se explica más tarde en el Estado del Arte, los otros sistemas públicos ya están integrados en la plataforma de *Google Transit* con lo cual , cualquier usuario con disponibilidad de *gmaps* (disponible en todas las plataformas móviles) puede consultar itinerarios de estos sistemas de transportes.

1 Inicialmente sólo se contempla EMT Palma, ver Estado del Arte

Motivación

Los ciudadanos de Mallorca en general y de Palma en particular se caracterizan por hacer un uso muy leve de los sistemas de transporte públicos; en el caso que nos ocupa, los buses de la EMT.

Puede haber motivos socioculturales implícitos en este hecho, que no se van a valorar, pero sí que se pueden evaluar los motivos de bajas frecuencias y sobretodo de infraestructura tecnológica puesta al servicio del ciudadano, limitada y deficiente.

En líneas generales: hay muy pocos paneles electrónicos de marquesina que informen sobre los siguientes buses de cada línea y a los usuarios les es imposible memorizar la frecuencia de paso y el horario de salida de cada bus para cada línea en cada parada que sea susceptible de utilizar.

Consultar esa información desde un dispositivo móvil tampoco ha sido algo fácil (hasta la reciente publicación en el mes de junio 2013 de una app móvil desarrollada para la EMT por una empresa externa); el usuario debía o bien descargarse el pdf[2] con los horarios de la línea desde la web *http://emtpalma.es* o bien enviar un SMS a un servicio automático con el consiguiente coste.

Si a todo esto, sumamos que, dado el tamaño y fisonomía de Palma, donde se puede estimar que la mayoría de desplazamientos no supera los 3 km (½ hora a pie) y que la frecuencia de paso de los buses va de los 7 minutos de la línea 8 en días laborables a los 120minutos de la 31 en días festivos; con un promedio global de 28 minutos entre todas las líneas (teniendo en cuenta los distintos horarios por línea y sin evaluar el peso específico de cada una de ellas) , se puede estimar los motivos de un uso tan bajo en desplazamientos con este medio.

Por todo ello se identifica la necesidad de disponer de una aplicación móvil que proporcione al usuario la información en tiempo real de cada parada en cuanto a horarios de pasada de cada línea.

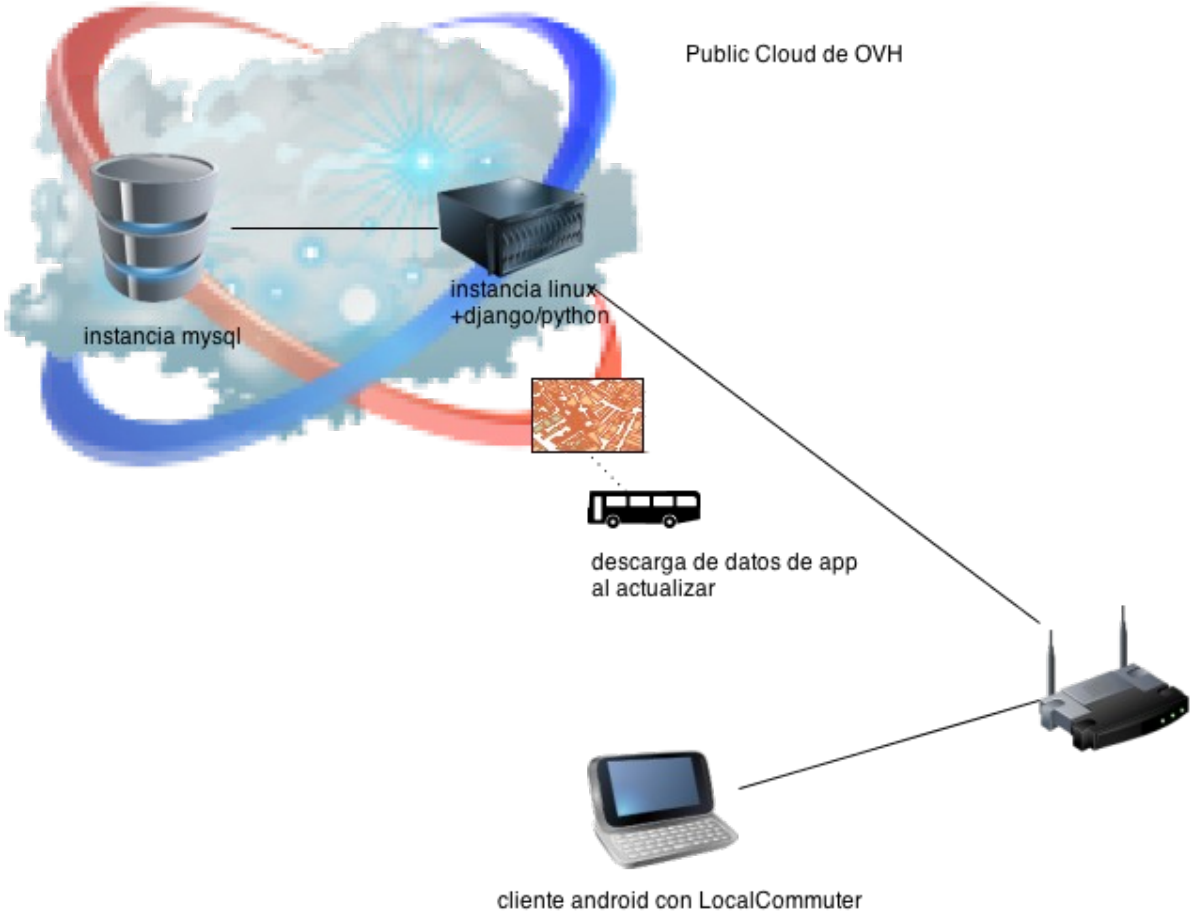
En cuanto a los otros sistemas de transporte público masivos de la isla referenciados en el punto anterior, existe un consorcio dependiente del Consell de Mallorca que los aúna y que, a pesar de no haber desarrollado ninguna aplicación móvil ni una web accesible desde smartphones para consultar las frecuencias de los distintos sistemas sí los ha integrado en el sistema nativo de itinerarios de desplazamiento de *google maps* (mediante el estándar GTFS de Google Transit) .

2 Horario laboral Línea 3: http://www.emtpalma.es/EMTPalma/Front/imgdb/264025.pdf

Glosario de Términos

FOSS	Free Open Sourced Software - software de código abierto
GTFS	General Transit Feed Specification
feed	objeto de sindicalización de contenidos
Apache Maven	software abierto para construcción y despliegue de proyectos java.
Apache Tomcat	servidor de aplicaciones de código abierto
OTP (Open Trip Planner)	Plataforma FOSS de desarrollo de rutas de viaje
OSM (Open Street Map)	Repositorio de mapas liberados con licencias libres.
RESTFul	protocolo de comunicación que permite publicar una API para definir un servicio web.
JSON	formato de intercambio de información
OpenMaemo	sistema operativo móvil basado en fuentes abiertas y desarrollado por Nokia.
GPL (General Public License)	grupo de licencias de código abierto.
app	software que corre en un smartphone.
python	lenguaje de programación multiplataforma
django	framework de desarrollo web basado en python

Plataforma Tecnológica



Entorno Servidor

Servidor Ubuntu-Server 12.04 con

- webapp django 1.4 corriendo sobre apache2 y mod_wsgi,
- scripts python para rastrear la web de EMT (web crawler),
- mysql 5.5, apache2, wsgi (conector de python para apache2) y python2.7

Instancia de cloud pública en OVH con un servidor django gestionando el modelo de datos e integrando los datos anónimos de los usuarios, a sincronizar en la siguiente actualización.

Entorno Cliente

Dispositivo Android 4.0.3 con conexión de datos y GPS habilitado

Estado del Arte

Contexto Actual y futuro:OpenData, GTFS

Actualmente se contemplan varios contextos en el ámbito de la gestión de Sistemas de Transporte Público por parte de las distintas agencias gubernamentales.

Por un lado tenemos una situación local con una serie de fuentes de información heterogéneas y no integradas (EMT Palma, TIB, Ferrocarrils de Mallorca o Metro Palma) que no comparten sus datos ni los publican mediante estándares abiertos, en contra de quienes tratan de fomentar iniciativas independientes como es Open Data, promoviendo la apertura de los sistemas de información de las administraciones para la investigación y el desarrollo del conocimiento, dado que al fin y al cabo es información pública.

Afortunadamente esta situación está mejorando gracias a que algunas agencias locales se hayan sumado a este movimiento; en nuestro caso se puede citar a la caib [3]

OpenData

Se entiende *OpenData* como un nuevo paradigma de publicación de datos con la premisa de que éstos sean de libre acceso para todos los ciudadanos, sin limitaciones técnicas o legales y que a su vez estén disponibles a través de servicios no remunerados y , sobre todo, en formatos basados en estándares abiertos y estructurados, libres de licencias privativas.

Es un concepto que, en el seno de las administraciones públicas, debería ser ya un hecho ya que proporciona un marco de transparencia y eficiencia que puede impulsar el emprendimiento y la mejora de servicios de terceros, como empresas o emprendedores privados que pretendan explotar esa información permitiéndoles contribuir al crecimiento económico. En cuanto al ciudadano, es un factor de transparencia y guía para la participación ciudadana.

Por otra parte hay un conjunto de países y de regiones que han iniciado proyectos de OpenData dentro de sus zonas de actuación como por ejemplo, encontramos el proyecto **Aporta** del Gobierno de España, el proyecto **Open Data Euskadi** del País Vasco, el proyecto **Datos de Asturias** del Principado de Asturias, el proyecto data.gov.uk del gobierno británico, el proyecto data.gov del Gobierno de los Estados Unidos o el proyecto London Datastore de la región de Londres entre muchos otros que son buenos ejemplos de reutilización de información del sector público. Se pueden consultar todos ellos en el catálogo de Datasets públicos.

Como recurso informativo adicional se puede mencionar el sitio web de la fundaciontic.org donde se informa de novedades acerca de implementación de entornos open data en toda España.

3 http://www.caib.es/caibdatafront/definicio.jsp?lang=es

GTFS (API de Google Transit)[4]:

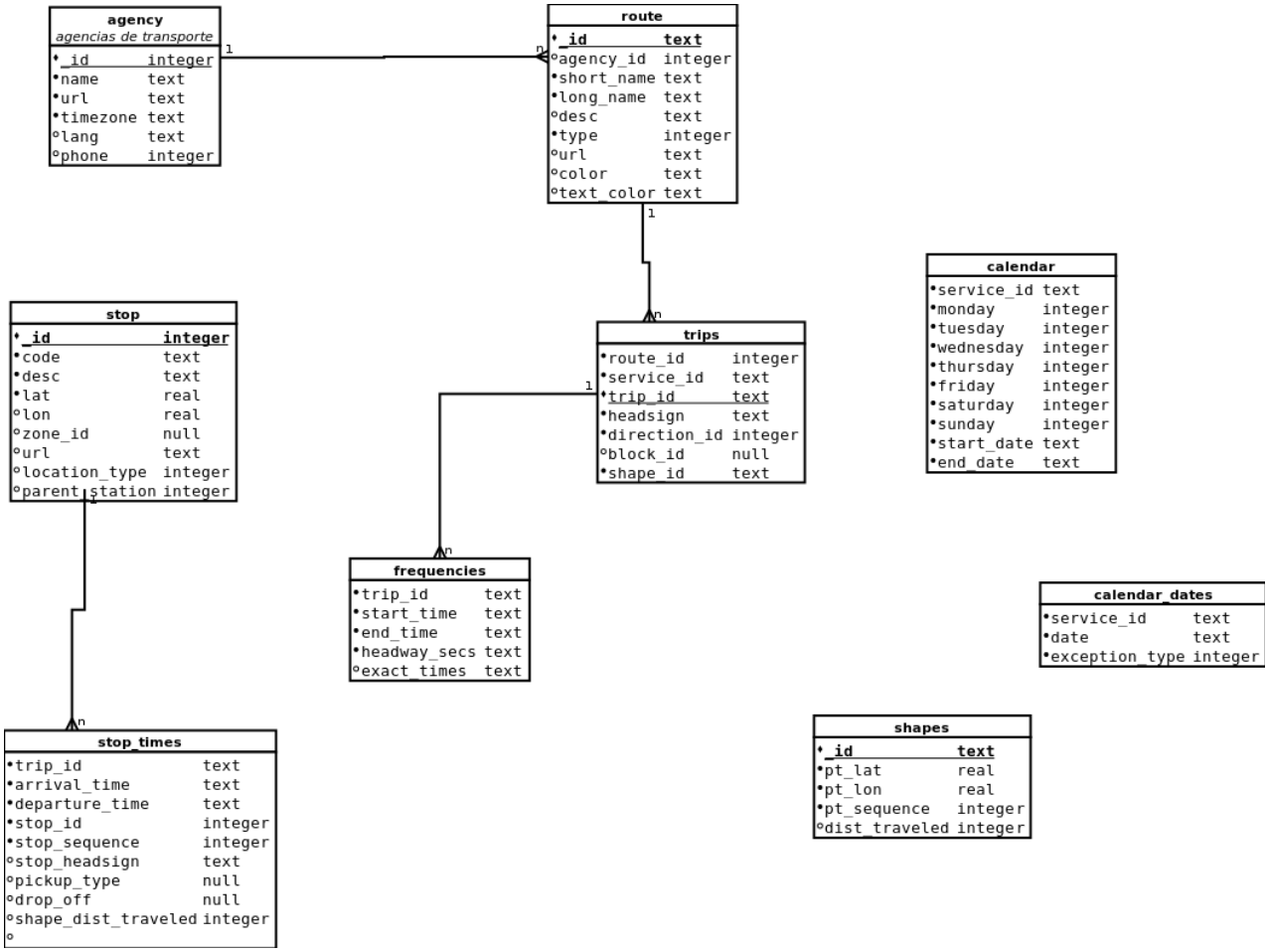
A través del proyecto GTFS (General Transit Feed Specification) se define una API pública para planificar sistemas de transporte público y su información geográfica asociada.

Está orientado por otro lado a que las agencias de transporte público compartan en formato *feed* (sindicalización) los datos de tránsitos de sus sistemas correspondientes.

Desde un primer momento se ha analizado minuciosamente el desarrollo de LocalCommuter en base a los principios establecidos por este estándar por el cual se define un modelo de datos que permite guardar la relación entre las agencias de transporte, sus líneas, sus rutas, las paradas o las frecuencias. Desafortunadamente se trata de un servicio de uso limitado a agencias oficiales de transporte y por tanto no es posible su uso en un proyecto académico ni *freelance*.

Cada agencia de transporte debe formatear el volcado de cada una de estas entidades en CSV y publicarlo en el servicio de *Google Transit*. De este modo cualquier usuario de un dispositivo móvil con la aplicación *google maps* instalada puede consultar estos datos así como realizar itinerarios de transporte público.

4 https://developers.google.com/transit/



Modelo de datos GTFS de Google Transit[5]

La API de *Google Transit* no puede ser usada para propósitos académicos y en general fuera del ámbito de las propias agencias de transporte a las que reconoce como único actor a la hora de publicar rutas, horarios e información del sistema de transportes en general.

Open Trip Planner

Plataforma *open source* que proporciona itinerarios multi-modo a los viajeros.

Está orientado a ofrecer los caminos más eficientes entre distintas rutas de transportes heterogéneos proporcionando un rango de aplicaciones que van de la información al pasajero hasta áreas de análisis espacial, planificación urbana, investigación sobre accesibilidad e ingeniería de transportes.

Todos los componentes de *OpenTripPlanner* se basan en estándares abiertos, desde el software de código abierto, su arquitectura, su metodología de desarrollo hasta la plataforma *OpenStreetMap*.

5 Descripción del modelo en el apéndice A

Las principales características son:

- Permite generar mapas a pie, en bici y trayectos de sistemas de transporte.
- Gestiona tiempos de viaje, tipo de pavimento, elevación y proporciona una IU para ajustar estos tres factores.
- Muestras perfiles gráficos de elevación para trayectos en bici.
- Interactúa con formatos estándares: GTFS, USGS , OpenStreetMap, Navteq, shapefiles, etc.
- Planifica viajes en ~100ms dentro de una ciudad de tamaño moderado como es Palma.
- Expone una API RESTful (XML and JSON), con la que otras apps puede interactuar.

A su vez proporciona una completa API así como toda la documentación necesaria para crear un entorno de desarrollo basado en *Maven* y el servidor de aplicaciones *Tomcat* de cara a diseñar las aplicaciones web con *OpenStreetMap* permitiéndoles la descarga automática de los mapas del entorno para crear nuestros grafos y trayectos de transportes.

Open Trip Planner proporciona un cliente para la plataforma Android para interactuar con los servidores OTP mediante la API OTP RESTFul del proyecto *opentripplanner-api-webapp*.

Marco Legal de la Aplicación LocalCommuter

Hoy en día no se puede desarrollar una aplicación sin prestar atención a las licencias de los distintos sistemas, librerías de programación, aplicaciones, frecuencias de transmisión, etc. Especialmente en aplicaciones que utilizan gran cantidad de dispositivos como puede ser el caso de desarrollos para *tablets* y *smartphones*.

En nuestro caso la orientación hacia un entorno *bastante* libre como es *Android* (no tanto como lo fuere *OpenMaemo* convertido recientemente en *Tizen* o como pueda ser *FirefoxOS*) y tan extendido nos proporciona cierta protección a nivel de licencias propietarias, gracias al uso del kernel *Linux* y de licencias *Apache* o *GPL* en la gran mayoría de sistemas que lo conforman.

A pesar de ello no dejan de haber restricciones que indudablemente alteran el curso del desarrollo.

A este respecto se puede considerar la disyuntiva inicial de si utilizar la API y entorno de *gmaps* para el desarrollo e integración de nuestra *app* dentro del dispositivo, con los beneficios evidentes que supone.

Sin embargo la alternativa de *OpenStreetMaps* nos da la tranquilidad que nos puede quitar el uso de algunas partes de *gmaps*:

- Uso de rutas de transporte y todo el sistema de *GTFS* está reservado a agencias de transporte con precios por ruta estimados en unos 130$/ruta.
- La exportación de rutas en formato shape puede verse sujeta a restricciones de uso por parte de los proveedores de mapas de *google* (*navtech* entre otros)

Así, pese a seguir orientados en el uso de *gmaps* y de toda su API de cara a cumplir los plazos de desarrollo e implementación del proyecto, cualquier uso en entorno de producción se haría ya una vez migrado a la plataforma de *OpenStreetMaps* o de manera similar *OpenTripPlanner*.

Requerimientos

Dada la complejidad de obtener los datos de líneas, paradas y horarios se han reducido los requerimientos de usuario en la aplicación final con respecto de los establecidos inicialmente.

Por ello se define la necesidad de una aplicación que permita conocer en tiempo real la información de líneas en base a una parada. Se entiende que el usuario de esta aplicación es un usuario de la red de transporte público con el propósito de :

- Desplazarse a su puesto de trabajo
- Acudir a su centro de estudios
- Regresar a casa

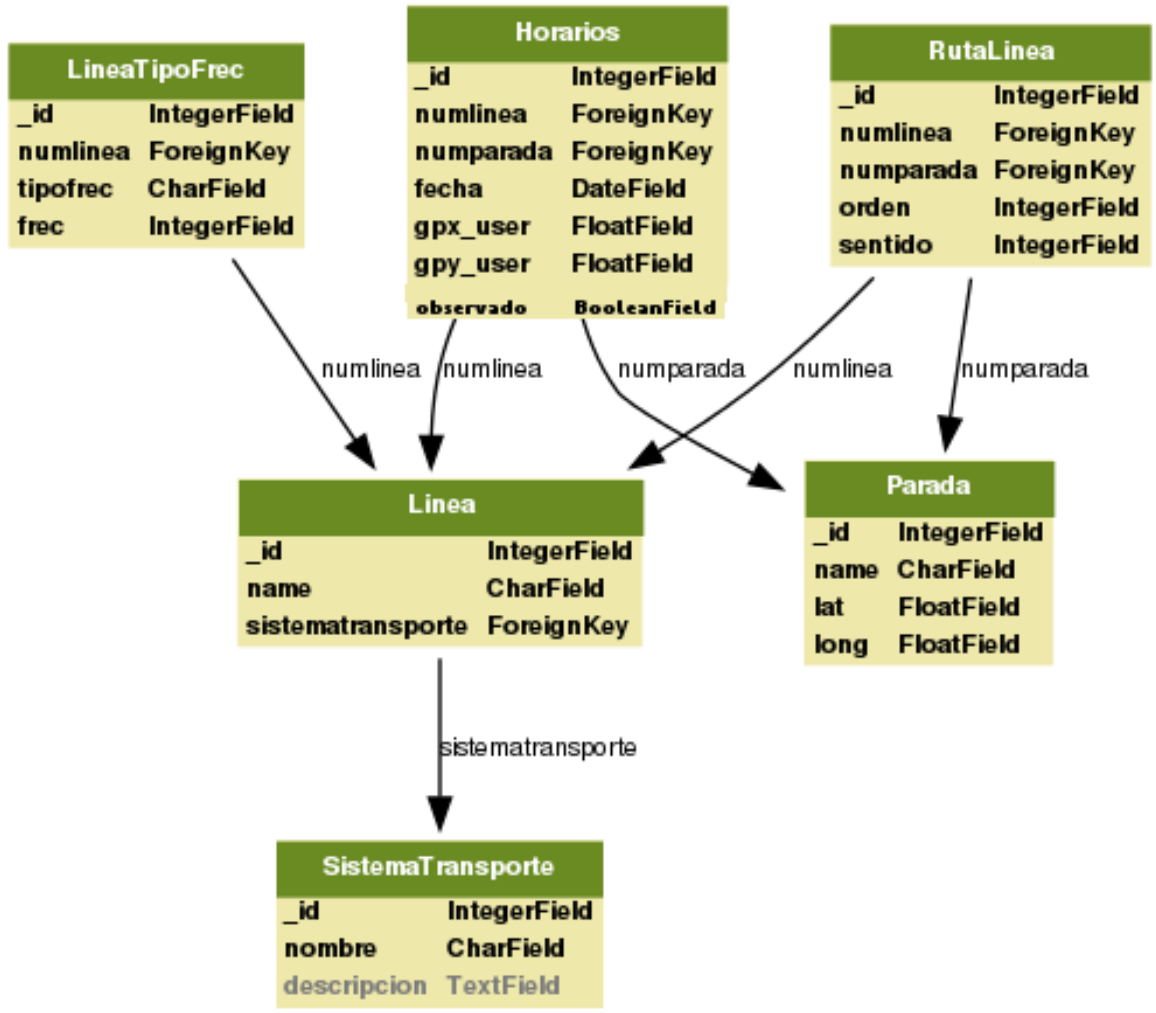
En estos tres escenarios es habitual tener la necesidad de saber a qué hora va a pasar el autobús por la parada más próxima

Requerimientos de Usuario a satisfacer por la aplicación:

- Conocer las paradas más próximas al usuario
- Obtener el tiempo que resta para que pase el bus de la línea L por la parada P.
- Geoposicionar al usuario en el mapa de la ciudad
- Conocer la distancia de las paradas a la posición del usuario o localización.
- Visualizar todas las paradas de una línea L, diferenciando el sentido (ida / vuelta)
- Sincronizar horarios de paradas y líneas con otros usuarios

Cabe hacer incapié en este último requerimiento; dada la dificultad para obtener datos reales de horarios de paso de buses, al estar accesible únicamente en formato binario (pdf) que a pesar de ser susceptible de ser analizado sintácticamente (parser) por nuestra aplicación python gracias a las librerías *pdfminer*, no sigue un formato unificado y estructurado entre un pdf de una línea y el de otra, hecho que obviamente imposibilita la automatización de la captura de esos datos. Es por ello que, se decide por un modelo de datos compartidos entre distintos usuarios, centralizado en una webapp *django*.

Arquitectura



Modelo de datos DER generado con el módulo django-extensions de django y la librería graphviz

El Modelo de la aplicación requiere de la definición de las entidades:

Sistema de Transporte : entidad que guarda la información básica del sistema de transporte, ya sea Bus EMT, Tren Inca, Bus Interurbano TIB o Metro.
A nivel de atributos se reduce su definición al identificador, nombre y descripción.

Ésta entidad se relaciona directamente con sus **Líneas** de transporte, ya sea L1, L2, …, LN; donde línea tampoco requiere mayor información que su identificador y su nombre.

Por otro lado se define la entidad **Parada** que almacena el código (con correspondencia directa al de la parada física), el nombre, que suele ser el de la calle y número donde se encuentra y sus coordenadas geográficas.

Las entidades restantes dependen de **Línea y Parada**; por un lado se define el tipo de frecuencia de cada línea en la entidad **LineaTipoFrec**, donde se almacena la relación de número de línea, tipo de frecuencia, definida por la EMT y obtenida a partir del webcrawler que por lo general se reduce a dos tipos: Laborables y Festivos. Finalmente se define la frecuencia media en minutos.

```
sqlite> select * from "LineaTipoFrec";
id| linea| tipo | frecuencia(t)
1|10|'Laborables'|15
2|10|'Festivos'|30
```

Por otro lado en **Horarios** se define la información de los horarios en cada Parada: número de línea, número de parada, fecha en formato datetime (dd/mm/yyyy hh:mm:ss) de manera que no solo se pueda interactuar con la hora de pasada del bus, sino que si es un horario muy antiguo , descartarlo directamente.

A este respecto, tener en cuenta el hecho de que en promedio, los horarios se actualizan cada 3 meses, aunque hay épocas del año con horarios especiales como puede ser navidad, o semana santa por tanto el hecho de poder descartar horarios por antigüedad sin , necesariamente eliminarlos del sistema, puede ser interesante a nivel de modelo.

Por otro lado la entidad Horario incluye información de geolocalización, en este caso no de la parada en cuestión, sino del propio usuario, de forma que en versiones posteriores de la aplicación, se pueda explotar la información estadística referente a por ejemplo: *Distancia media del usuario a la parada cuando hizo la consulta* o como registro de ubicaciones del usuario en caso de una implementación posterior de la aplicación que contemple el registro de usuarios, actualmente los usuarios son anónimos.

Finalmente se incluye un campo booleano **observado** que hace referencia a si el horario es oficial (extraído a partir de la agencia de transporte) o si por el contrario es un horario observado e incluido por algún usuario de la aplicación.

La última entidad por describir es **RutaLinea** que es básicamente la relación N-M entre Líneas y Paradas, con sus correspondientes identificadores como claves foráneas, además de la información del órden que guarda la parada p en la línea l y del sentido de la ruta ya sea ida o vuelta.

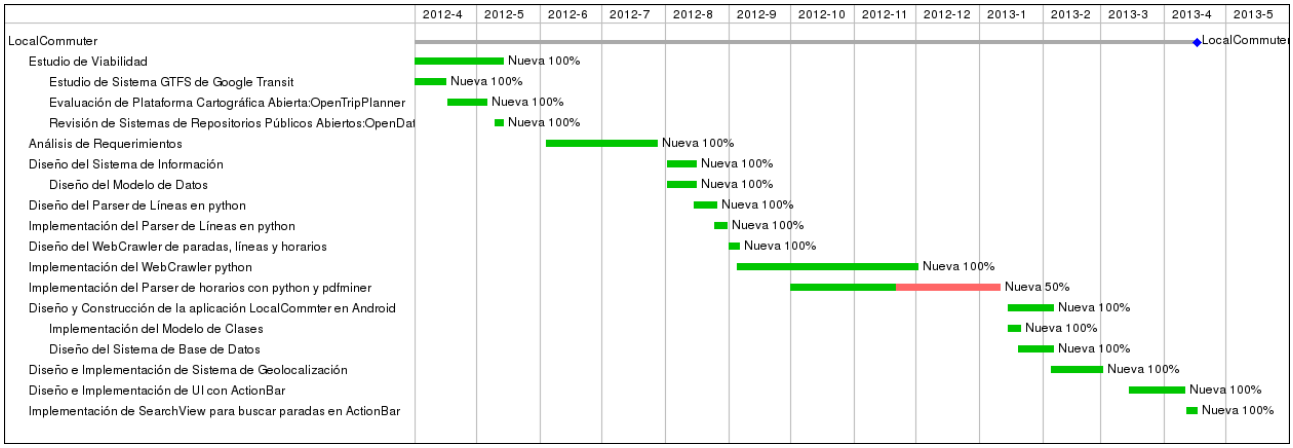
Descripción del Modelo de Datos en la versión sqlite3 de la aplicación Android:

```
sqlite> .schema

CREATE TABLE "LINEAS" ( "_id" INTEGER PRIMARY KEY, "name" TEXT);
CREATE TABLE "LineaTipoFrec" ( "_id" INTEGER PRIMARY KEY AUTOINCREMENT, "numlinea" INT
NOT NULL, "tipofrec" TEXT, "frec" INTEGER);
CREATE TABLE "PARADA" ( "_id" INTEGER PRIMARY KEY, "name" TEXT, "lat" REAL, "long" REAL);
CREATE TABLE "RealTimeTable" ( "_id" INTEGER PRIMARY KEY AUTOINCREMENT, numlinea
INTEGER NOT NULL, numparada INTEGER NOT NULL, fecha DATETIME, "gpx_user" REAL, "gpy_user"
REAL , observado boolean default true not null);
CREATE TABLE "RedTransporte" (
   "_id" INTEGER PRIMARY KEY AUTOINCREMENT,
   "name" TEXT,
   "description" TEXT
);
```

```
CREATE TABLE "RutaLinea" ( "_id" INTEGER PRIMARY KEY AUTOINCREMENT, "numlinea" INTEGER NOT NULL, "numparada" INTEGER NOT NULL, "orden" INT NOT NULL, "sentido" INT NOT NULL);

CREATE TABLE "SistemaTransporte" (
    "_id" INTEGER PRIMARY KEY,
    "name" TEXT );
CREATE TABLE android_metadata ("locale" TEXT);
CREATE UNIQUE INDEX "name" on sistematransporte (name ASC);
```

Planificación de Tareas

Se define la planificación del proyecto visible a través del diagrama de Gantt con las actividades y tareas y sus relaciones y tiempos de desarrollo.



Siguiendo un modelo de desarrollo en base a metodologías ágiles se han definido tres Actividades principales descompuestas en sus correspondientes tareas:

Análisis: Estudio de Viabilidad

- Estudio del Sistema de gestión de Transportes GTFS de Google Transit
- Evaluación de la plataforma de desarrollo cartográfico open source OpenTripPlanner
- Revisión de los sistemas de repositorios públicos abiertos: OpenData
- Análisis de Requerimientos

Diseño del Sistema de Información:Diseño de los Sistemas de Bases de datos

Diseño e Implementación del *Parser* de Líneas

Diseño e Implementación del *WebCrawler* de Paradas

Diseño e Implementación del Parser de Horarios con python y PdfMiner

Diseño y Construcción de la aplicación LocalCommuter en Android

- Implementación del modelo de clases
- Diseño del Sistema de Base de datos
- Diseño e Implementación del sistema de Geolocalización
- Diseño de la Interfaz Gráfica de Usuario con *ActionBar*
- Implementación de la vista de búsqueda de paradas con SearchView en ActionBar
- Interacción Cliente-Servidor mediante API RestFul y JSON del lado Android

Obtención de Datos

Desde un primer momento se ha tenido claro que el tipo de almacenamiento ideal para la aplicación móvil era una base de datos sqlite3. En cualquier caso se da paso a explicar las alternativas y el motivo de su descarte.

- **Preferencias**: mediante el uso de pares clave/valor se puede almacenar tuplas de información, tal como se hace con los parámetros de la aplicación o cuando se guarda el estado de una actividad antes de pausarla o recrearla (cambio de orientación) ; por tanto no se puede considerar un buen método para almacenar grandes cantidades de información.

- **Fichero de texto:** almacenar información en la memoria interna o en la tarjeta SD es una opción factible habiéndose llegado a considerar la idea de almacenar toda la información de las distintas entidades del modelo de datos en ficheros CSV, tal como se hace para interactura con la API de *Google Transit* en formato *GTFS*. De todas maneras, una vez descartada la posibilidad de poder interactuar con esos servicios, se ha descartado por el sacrificio que supone prescindir de la potencia de SQL.

- **XML**: este formato estándar es apropiado para integración de aplicaciones o acceso a servicios web online (RestFul, JSON) disponiendo en Android de librerías tan potentes como SAX o DOM. De todas maneras, para el acceso y gestión de datos de nuestra aplicación, dependiente de un modelo de datos relacional y estructurado, es demasiado laboriosa la interacción sin obtener ninguna ventaja significativa.

- **ContentProviders**: en realidad no es una alternativa a la base de datos sino un complemento, definiendo unas operaciones que permiten acceder a los datos del modelo de la aplicación desde otras aplicaciones. Como esto escapa de los requerimientos funcionales se ha declinado su implementación.

- **Internet:** el acceso a los datos de la aplicación almacenados en un servidor web es una opción que requiere de unas condiciones de velocidad de conexión elevada que ninguna aplicación menor se puede permitir el lujo de exigir a sus usuarios.

Una vez está claro el uso de una base de datos relacional para la gestión de los datos de aplicación y con el diseño de GTFS como estándar se analiza las fuentes de información que van a proveer de datos al modelo, identificando una serie de **servlets** de emtpalma.es de los que nutrirse.
Se definen las tareas de:

- Recorrido del servlet de la EMT para capturar el listado de líneas, paradas, sus interacciones, los tipos de horario de una línea y su frecuencia.
- Reconocimiento de registros de información mediante **Expresiones Regulares.**
- Sanitización o limpieza de formato de los datos capturados y creación de un fichero sqlite3.

De este proceso sin duda el uso de las expresiones regulares apropiadas a cada caso es capital a la hora de extraer la información de manera ágil y limpia, sin necesidad de postproceso.

Fichero sqlite3 con el esquema y carga inicial de datos

Una vez se dispone del fichero de volcado para generar el esquema y cargar el volcado de datos recogidos por el crawler, se crea una base de datos sqlite3 que se cuelga en la carpeta assets del proyecto Eclipse, de manera que tanto al compilar como al empaquetar la aplicación se disponga de la información mínima para funcionar la aplicación.

Internamente dentro de la clase SQLiteOpenHelper se hace la copia de la base de datos original en assets a la ruta final que sigue un formato estándar en android:

/data/data/com.dominio.aplicacion/databases

De este modo ya se dispone de la base de datos popularizada con los datos del crawler listos para explotarlos con la aplicación.

Datos Compartidos dinámicos

Uso del módulo python Django-Piston para publicar una API que interactúe con el modelo de datos de la aplicación del lado servidor.

Gracias al diseño modular de *django* es fácil integrar módulos *python* para ampliar su funcionalidad.

Concretamente interesa la publicación del modelo de datos de la aplicación del lado servidor mediante una API RESTFul para que los distintos clientes Android corriendo la aplicación puedan sincronizar, inicialmente los horarios, ampliable en cualquier caso a la sincronización de cambios en paradas o líneas de cara a no tener que forzar al usuario a actualizar la aplicación con tanta frecuencia.

Para ello se define por tanto una API RestFul capaz de publicar o recuperar datos en formato JSON.

JSON es un estándar abierto para la representación de datos basado en Javascript que utiliza el patrón llave/valor para representar la información así como arrays asociativos en caso de tipos de datos más complejos.

Es bastante ligero y muy apropiado para el uso en dispositivos móviles; la alternativa XML es demasiado pesada por el uso de etiquetas que sobrecargan el payload innecesariamente;

El uso del arbol DOM o de SAX como parser de información es conveniente en aplicaciones de escritorio pero implica demasiada carga computacional para un móvil.

Piston necesita definir una carpeta api/ de la que cuelgue el manejador principal o handler.py, que se vincula con el modelo en el models.py. El handler define el manejo de las operaciones CRUD.

Models.py de la aplicación django

```
class SistemaTransporte(models.Model):
    _id = models.IntegerField(primary_key=True, db_column="_id")
    nombre = models.CharField(max_length=25)
    descripcion = models.TextField(max_length=250,blank=True)

    def __unicode__(self):
        return u'%s' % self.nombre


class Linea(models.Model):
    _id = models.IntegerField(primary_key=True, db_column="_id")
    name = models.CharField(max_length=25)
    sistematransporte = models.ForeignKey(SistemaTransporte)
    def __unicode__(self):
        return u'%s' % self.name

class  Parada(models.Model):
    _id = models.IntegerField(primary_key=True, db_column="_id")
    name = models.CharField(max_length=25)
    lat = models.FloatField()
    long = models.FloatField()
    def __unicode__(self):
        return u'%s' % self.name

class LineaTipoFrec(models.Model):
    _id = models.IntegerField(primary_key=True)
    numlinea = models.ForeignKey(Linea)
    tipofrec = models.CharField(max_length=25)
    frec = models.IntegerField()
    def __unicode__(self):
        return u'%s' % self.tipofrec

class RutaLinea(models.Model):
    _id = models.IntegerField(primary_key=True)
    numlinea = models.ForeignKey(Linea)
    numparada = models.ForeignKey(Parada)
    orden = models.IntegerField()
    sentido = models.IntegerField()

class Horarios(models.Model):
    _id = models.IntegerField(primary_key=True)
    numlinea = models.ForeignKey(Linea)
    numparada = models.ForeignKey(Parada)
    fecha = models.DateField()
    gpx_user = models.FloatField()
    gpy_user = models.FloatField()
```

A continuación se definen los distintos *handlers* que conectan con el modelo por un lado y definen la interacción del *webservice* por otro, ya sea en modo GET o PUT, discriminando entre varios tipos de peticiones y por tanto sirviendo distintos subconjuntos de datos de acuerdo con la selección.

En el módulo busapiHandler.py se define la acción por la cual se van a servir al cliente de la petición HTTP todos los registros de horarios de los últimos 3 meses.

```
class BusApiHandler(BaseHandler):
    allowed_methods = ('GET')
    model = Horarios
    fields=("_id","numlinea", "numparada","fecha",”hora”,"gpx_user","gpy_user")

    @throttle(10, 1*60) # 10 call per minute
    def read(self, request, horarios_id=None,*args, **kwargs):
        try:
            periodo = datetime.now() - timedelta(90)
            return Horarios.objects.filter(fecha__gt=periodo)
```

Con esta consulta via HTTP se recupera un subconjunto de datos en formato JSON del tipo:


```
[

    {
        "hora": "22:03:04",
        "numlinea": {
            "_state": "<django.db.models.base.ModelState object at
0x7fd560306250>",
            "_id": 5,
            "sistematransporte_id": 1,
            "name": "'Es Rafal Nou-Son Dureta'"
        },
        "fecha": "2013-09-14",
        "gpx_user": null,
        "gpy_user": null,
        "_id": 1,
        "numparada": {
            "lat": " '39.56886291503906'",
            "_id": 9,
            "_state": "<django.db.models.base.ModelState object at
0x7fd560306550>",
            "name": "' Passeig Maritim 2'",
            "long": " '2.6339919567108154'"
        }
    },
    {
        "hora": "18:40:24",
        "numlinea": {
            "_state": "<django.db.models.base.ModelState object at
0x7fd560306810>",
            "_id": 9,
            "sistematransporte_id": 1,
            "name": "'Son Espanyol'"
        },
        "fecha": "2013-09-16",
        "gpx_user": 0.0,
        "gpy_user": 0.0,
        "_id": 2,
        "numparada": {
            "lat": " '39.55937957763672'",
            "_id": 12,
            "_state": "<django.db.models.base.ModelState object at
0x7fd560306c90>",
            "name": "' Can Barbera'",
            "long": " '2.626497983932495'"
        }
    }
]
```


Donde los corchetes [] delimitan un Array JSON mientras que las llaves {} definen un JSON Object.

En él se puede observar los valores interesantes de la consulta: hora de pasada del bus, número de línea y número de parada.

O bien en formato XML añadiendo el *caret* xml a la petición HTTP desde el cliente:
`http://192.168.0.103:8000/busapi/alltimetables/xml/`

```
<response><resource><numlinea><_state><django.db.models.base.ModelState object
at
0x7f807c1bd410></_state><_id>1</_id><sistematransporte_id>1</sistematransporte_i
d><name>L1</name></numlinea><fecha>2013-09-
11</fecha><gpx_user>0.0</gpx_user><gpy_user>0.0</gpy_user><_id>2</_id><numparada
><lat>38.233</lat><_id>10</_id><_state><django.db.models.base.ModelState object
at 0x7f807c1bd550></_state><name>Cr Valldemossa
221</name><long>2.6466</long></numparada></resource></response>
```

En el lado cliente Android se definen los métodos para recuperar el listado de Horarios y cargarlo en la base de datos.

Dado el ciclo de vida de las aplicaciones Android donde se dispone de una única actividad por aplicación activa al usuario, el uso de un thread activo para acceder a datos de un *webservice* no es muy conveniente. En estos casos se recomienda el uso de las AsyncTasks para definir tareas asíncronas que no interfieran en la ejecución del resto de la aplicación.

Por un lado se requiere la definición de la clase JSONParser que convierte la petición HTTP en un objeto JSON

Definición de Variables

```
    static InputStream is = null;
    static JSONObject jObj = null;
    static String json = "";
```

Captura del contenido HTTP en la variable InputStream *is*

```
        DefaultHttpClient httpClient = new DefaultHttpClient();
        HttpPost httpPost = new HttpPost(url);
        HttpResponse httpResponse = httpClient.execute(httpPost);
        HttpEntity httpEntity = httpResponse.getEntity();
        is = httpEntity.getContent();

        BufferedReader reader = new BufferedReader(new InputStreamReader(is, "utf-8"), 8);
        StringBuilder sb = new StringBuilder();
        String line = null;
        while ((line = reader.readLine()) != null) {
            sb.append(line + "\n");
        }
            is.close();
        json = sb.toString();
```

Parsing del objeto String a objeto JSON

```
        try {
            jObj = new JSONObject(json);
        } catch (JSONException e) {
            Log.e("JSON Parser", "Error de parsing " + e.toString());
        }
        return jObj;
```

Acceso a servicios de red en la actividad principal

El uso de servicios de conectividad en la actividad principal está totalmente desaconsejado por parte de la comunidad de desarrolladores de Android, principalmente por motivos de rendimiento, ya que el tiempo de ejecución de una aplicación puede alterarse demasiado en función de las condiciones de conectividad del terminal en un momento dado.

Es más, desde la versión de API de Android nivel 9 (correspondiente con la versión 2.3 del sistema) este tipo de implementación devuelve un error de ejecución en la aplicación, para lo cual hay que añadir en el método onCreate el siguiente trozo de código con el cual se desactiva la política *StrictMode* en el hilo principal o *UI Thread.*

```
        if (android.os.Build.VERSION.SDK_INT > 9) {
          StrictMode.ThreadPolicy policy = new  StrictMode.ThreadPolicy.Builder().
                                                permitAll().build();
          StrictMode.setThreadPolicy(policy);
        }
```

En cualquier caso hay mecanismos más apropiados como el uso de tareas asíncronas *AsyncTasks*

En este caso se implementa una *AsyncTask* que recupera los datos del servidor mediante *serialización* de objetos *JSON* a partir de la consulta a la API *RestFul* del servidor *django*.

Las tareas asíncronas o *AsyncTasks* tienen tres tipos genéricos:

- Params: el tipo de los parámetros enviados a la tarea durante la ejecución
- Progress: el tipo de unidades de progreso publicadas durante el cálculo en *background* de la tarea.
- Result: el tipo de resultado del cálculo en background

Se definen 4 pasos para el proceso de una tarea:

- onPreExecute() que se invoca en el UI thread o hilo principal de ejecución, inmediatamente después de que la tarea sea ejecutada
- doInBackground(Paaram se invoca en segundo plano inmediatamente después del anterior.
- onProgressUpdate(Progress se llama en el hilo principal después de una llamada a publishProgress(Progress...).
- onPostExecute(Result), se ejecuta después del calculo en segundo plano

En LocalCommuter basta definir la acción en doInBackground() ya que no hay un requerimiento específico de cuándo se tiene que lanzar la tarea de sincronizar los datos de la aplicación con nuevos registros horarios.

```
        private class LeeHorariosJSONTask extends AsyncTask
           <String, Void, String> {
               protected void doInBackground(String  urls) {
                   actualizaJSON(url);
               }

      private void actualizaJSON(String miurl) {
```

Se crea la instanciaJSON Parser

```
              JSONParser jParser = new JSONParser();
```

se carga el String JSON de la URL

```
              JSONObject json = jParser.getJSONFromUrl(miurl);
              Log.i("actualizaJSON: ",json.toString());

              try {
```

Captura del array de horarios

```
                      json_horarios = json.getJSONArray(TAG_HORA);
                      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
                      // recorrido de horarios
                      for (int i = 0; i < json_horarios.length(); i++) {
                             System.out.println("registros json horarios"+String.valueOf(i));
                             JSONObject c = json_horarios.getJSONObject(i);
```

se guardan los nodos json en variables previa su carga en base de datos

```
                             String id = c.getString(TAG_ID);
                             String numlinea = c.getString(TAG_NUMLINEA);
                             String numparada = c.getString(TAG_NUMPARADA);
                             String gpx = c.getString(TAG_GPX);
                             String gpy = c.getString(TAG_GPY);
                             String fecha = c.getString(TAG_FECHA+""+TAG_HORA);

                      ContentValues values = new ContentValues();
                      values.put("numlinea", numlinea);
                      values.put("numparada", numparada);

                      values.put("fecha", dateFormat.format(fecha));
                      values.put("numlinea", numlinea);
                      db.insert(TABLE_TIMETABLES, values);
                      }
              } catch (JSONException e) {
                      e.printStackTrace();
              }

      }
```

Diseño e implementación

Desarrollo del Web Crawler en Python

Se desarrolla una aplicación en lenguaje *python 2.7* que rastrea una serie de *servlets* de la web de emtpalma.es para obtener la información de :

- Identificador, nombre y descripción de parada, además de su ubicación geográfica
- Relación de líneas por parada
- Descripción de línea y tipo de horario (laboral, festivo) así como la frecuencia media.

Implementación del Parser de PDF de horarios

Se da paso a explicar la implementación del parser de pdf de cada línea, encargado de descargar los pdf de cada uno de los horarios de cada línea.

A su vez se diseña un script en python en entorno servidor (accesible desde la *app* Android que parsee los documentos en formato pdf de origen desde la web de la EMT y los extraiga en formato texto para posteriormente extraer la información formateada en base a expresiones regulares usando el módulo **re** de python.:

Se define la url a la que se pasará una secuencia numérica correspondiente a cada línea de autobús de manera que se pueda descargar el contenido con urllib2.urlopen(url) para su manipulación posterior, un diccionario para almacenar la información de cada línea de cara a manipular las url de cada una de ellas.

```
urlOrigen ='http://www.emtpalma.es/EMTPalma/Front/lineas.es.svr?accion=entrada&cod_linea='
urlAbs = 'http://www.emtpalma.es/EMTPalma/Front/'

dicLineas = {}
```

Acto seguido, por medio del módulo *BeautifulSoup* se recupera el contenido de la página en busca de una etiqueta de tipo link que contenga el patrón 'pdf' al final de la cadena.

```
def parseLinea(numLinea):

    url = urlOrigen+str(numLinea)

    try:
        handler = urllib2.urlopen(url)
        text = handler.read()
        handler.close()
    except URLError as e:
        print e.reason
    soup = BeautifulSoup(text)
    current_link = ''
    for link in soup.findAll('a', href=True):
        current_link = link.get('href')
```

```
        if current_link.endswith('pdf'):
            url=urlAbs+current_link
            ruta = descargaPdf(url)
            if (ruta != ""):
                print convert_pdf(ruta)
            else:
                print "nada que convertir"
            print "gestDic %d %s" % (numLinea, current_link)
            dicLineas[current_link] = numLinea
```

Finalmente se descarga el archivo del enlace recuperado y se guarda el contenido del diccionario de líneas con la relación del nombre de fichero y la línea en cuestión

```
def saveDic():
    with open("diccionario_pdfs.txt","wb") as f:
        csv.writer(f).writerows((k,v)  for k, v in dicLineas.iteritems())
```

El método convert-pdf convierte el contenido del pdf que se pasa como parámetro mediante el uso de la librería PdfMiner:

```
def convert_pdf(path):

    rsrcmgr = PDFResourceManager()
    retstr = StringIO()
    codec = 'utf-8'
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, codec=codec, laparams=laparams)

    fp = file(path, 'rb')
    process_pdf(rsrcmgr, device, fp)
    fp.close()
    device.close()

    str = retstr.getvalue()
    retstr.close()
    return str
```

Desgraciadamente los datos que se obtienen a partir de este proceso están desestructurados (consultar anexo) y no son homogéneos dando como resultado una salida en texto con una organización distinta si se trata de los horarios de una línea o de otra: algunos pdf son a 1 columna, otros a 2, algunos tienen cuadros de notificación con información adicional y en general no hay una manera automatizada de manipular esa información de manera que al cargarla en la *app* android, esta información sea fiable.

Por otro lado, el hecho de que sólo se disponga de los horarios de salida de cada bus en cada una de sus rutas diarias hace más complicado aún si cabe el calculo teórico de la *hora de llegada* por una parada en concreto.

La aproximación más fiable que se ha calculado se basa en, o bien calcular el *offset* o desfase horario entre la hora de salida desde el origen de la ruta con respecto a la parada X, dividiendo la frecuencia de la ruta (que no se conoce y se debería estimar) por el número de paradas y finalmente sumando este valor al de origen; algo así como :

Ts = Tiempo de Salida del bus en la ruta actual desde la parada de origen
Tp= Tiempo de pasada por la parada actual.
Tt = Tiempo total que se estima tarda el bus en recorrer la ruta en el sentido actual.
Op= Orden de la parada actual en la ruta.

$$Tp=Ts+(Tt/Op)$$

Considerando que no se puede estimar el estado del tráfico, las condiciones climáticas o las posibles averías del bus para obtener un horario de pasada más ajustado al real y que ni siquiera se dispone del valor Tt de cada ruta, tampoco se puede por tanto dar una estimación teórica fiable.

WebCrawler: generando la fuente de datos

La obtención de toda la información de nuestro modelo está basada en la captura de datos mediante rastreo de servicios web accesibles para cualquier usuario a través de la web de la empresa municipal de transportes de Palma.

Se han identificado una serie de *servlets* que se pueden consultar desde python gracias a las librerías urllib2 o HtmlParser.

No es ningún secreto que *python* es un lenguaje muy potente, gracias a la gran cantidad de módulos disponibles de la mano del desarrollador a golpe de comando pip/easy_install

Mediante la extensión de funcionalidad que proporcionan módulos como urllib2 o BeautifulSoup para el análisis de contenido web, *pdfminer* para la manipulación de ficheros en formato pdf o **RE** en el caso siguiente para el reconocimiento avanzado de patrones de caracteres es posible implementar a bajo nivel y con una sintaxis clara y fácilmente comprensible cualquier requerimiento que sea preciso.

Módulo RE para análisis de expresiones regulares

En el parser de líneas se define una serie de instancias re para el reconocimiento de los distintos items del fichero html que conforman la información útil de cada línea de transporte.

```
    #patrones de la información que queremos extraer en formato regex
    nombrelinea = re.compile( r"^\W*<span\sstyle=\"color:.*\">\r$ ")
    tipodia = re.compile( r"(Laborables|Festivos)" )
    freq = re.compile( r"Frecuencia media:.(\d+)" )
    ida = re.compile( r"div class=\"ida" )
    vuelta = re.compile( r"div class=\"vuelta" )
    parada = re.compile( r"<a class=\"nombre\".*id=\"parada(\d*)_(\d*)" )
```

Mediante el comando re.compile se define el patrón de la expresión regular *regex*

De especial interés és el uso del patrón de *regex* del tipo **lookbehind** que permite extraer como variables una serie de patrones precedidos por otra cadena concreta.

En el *servlet* de paradas la coordenada viene definida en esta etiqueta html

<a style="position.*inline;\".*maps\?ll=(\d{:2}\.\d{:5}),(\d{:2}\.\d{:5})&

El contenido entre paréntesis se almacena en dos variables: \1 y \2, que corresponden con las coordenadas gps de la parada numParada.

Por otro lado cabe destacar la posibilidad en todo momento de extraer cada uno de los patrones en modo de variables *python* para su posterior almacenamiento como dato útil del modelo de datos.

En este *scriptlet* se realiza una búsqueda intensiva de todas las coordenadas de las paradas.

```
    reg = re.compile( r"(?<=maps\?ll=)(\d+\.\d+),(\d+\.\d+)" )
    xy = reg.findall( text )
    print xy
    print "parada %d\n" % numStop
```

Parsing de Líneas y Paradas

El parser de líneas recupera a partir de una llamada HTTP al *servlet* de líneas toda la información disponible sobre número y nombre de la línea así como información acerca de los tipos de horarios y sus frecuencias, generando la salida en formato *csv* de cada registro de las tablas línea y tipo.

A su vez recupera la asociación de líneas y paradas, ya sea para el tipo de trayecto de **ida** o de **vuelta.**

```
url = "http://www.emtpalma.es/EMTPalma/Front/lineas.es.svr?accion=entrada&cod_linea="
totallineas = 50
arrayQueries = []
def parseLinea(content, lineactual):

    #instancia HTMLParser() para escapar los caracteres unicode como apostrofes
    h = HTMLParser()
    i=0
    counttiempo = 0
    listaparadas = []

    tipofrec = ""
    ultimosentido = 0
    nlinea = "" #nombre de linea para insert
    for linea in content:
        i+=1
        match = re.search( r"^\W*<span\sstyle=\"color:.*\">\r$" , linea )
        if match:
          nlinea = h.unescape(content[i])
          nlinea = nlinea.strip()
          print "INSERT INTO \"LINEAS\" VALUES(" + str(lineactual) + ", \"" + str(nlinea) + "\");"
          query = "INSERT INTO \"LINEAS\" VALUES(" + str(lineactual) + ", \"" + str(nlinea)+ "\");"
          arrayQueries.append(query)
        esnombrelinea = re.search( nombrelinea, linea )
        estipo = re.search( tipodia, linea )
        esfreq = re.search( freq, linea )
        esida = re.search( ida, linea )
        esvuelta = re.search( vuelta, linea )
        esparada = re.search( parada, linea )
        if esnombrelinea:
            print content[i]
        elif estipo:
            tipofrec = str(estipo.group(1))
            print tipofrec
#           print str(counttiempo)
        elif esfreq:
            counttiempo+=1
            print tipofrec
            print esfreq.group(1)
            #simulamos insert de linea con: numlinea, nombre temporal, freq laboral, freq festivo
            print "insert into \"LINEA-TIPOFREC\" VALUES(" + str(lineactual) + ", \"" + tipofrec +
"\", "+ str(esfreq.group(1))  + ");"
            query = "insert into \"LINEA-TIPOFREC\" VALUES(" + str(lineactual) + ", \"" + tipofrec +
```

```
"\", "+ str(esfreq.group(1))  + ");\n"
            arrayQueries.append(query)

        #ahora buscamos el orden de paradas
        elif esida:
            ultimosentido = 0
        elif esvuelta:
            ultimosentido = 1
        elif esparada:
            print "parada " + esparada.group(1) + " " + esparada.group(2)
            print "insert into \"RUTA-LINEA\" VALUES(" + str(lineactual) + "," + esparada.group(2) +
", " + esparada.group(1) + ", " + str(ultimosentido) + ");"
            query = "insert into \"RUTA-LINEA\" VALUES(" + str(lineactual) + "," + esparada.group(2)
+ ", " + esparada.group(1) + ", " + str(ultimosentido) + ");\n"
            arrayQueries.append(query)
        else:
            pass

def initLinea(numlinea):
    url2 = url + str(numlinea)
    try:
        handler = urllib2.urlopen(url2)
        content = handler.readlines()
        handler.close()
    except URLError as e:
        print e.reason
    return content

def saveList():
    ftemp = open("queries-linea-completa.txt", "wb")
    for item in arrayQueries:
        ftemp.write(str(item))
    ftemp.close()


def main():

    lineactual = 0
    while (lineactual < totallineas):
        lineactual += 1
        try:
            content = initLinea(lineactual)
        except:
            continue
        parseLinea(content, lineactual)
        saveList()


if __name__ == "__main__":
    main()
```

De esta manera y en <100 líneas de código se ha obtenido la mayor parte de la información para el modelo de datos de la aplicación móvil, generando la fuente de datos para rellenar la base sqlite3 que después se cargará en memoria de la aplicación *android*.

Desarrollo de la Aplicación Android

Clases del Modelo

A pesar de que el Modelo de Datos de la aplicación se compone por una serie de entidades relacionadas, la interacción con la mayoría de ellas es de sólo lectura o extracción de información generándose carga por parte del usuario únicamente en la tabla de *Horarios Observados*.

Ello junto con la manipulación necesaria, a nivel de instancias de clase, de los objetos *Parada* de cara al cálculo de distancias hace que no definamos más Clases del Modelo que éstas; accediendo al resto mediante *queries sql* de la clase AdminSQLiteOpenHelper.

En el primer caso, para la Clase Stop (parada) se definen las variables y los correspondientes *setters/getters*.

```
        protected int id;
        protected String name;
        protected double lat;
        protected double lng;
```

Por otro lado se implementa mediante herencia la clase StopDist, que extiende de Stop para añadir la variable `protected double dist;` sus correspondientes *getters/setters* y los métodos para el cálculo de distancias.

En el caso de la clase de Horarios Observados (RealTimeTable) de igual modo, se implementan los métodos para añadir y eliminar valores de cada atributo:

```
        private int id;
        private int numlinea;
        private int numparada;
        private int hora;
        //fecha se añade directamente a la db como datetime('now'), en principio no definimos una
variable, si lo hicieramos usar int que basta para precisión dia-hora-segundo
        private double gpx_user;
        private double gpy_user;
```

Base de Datos SQLite3

Preparación de la base de datos a partir de los volcados python

El principal resultado de los volcados que producen los *scripts python* del proyecto es una base de datos *sqlite3* que se copia en la ruta assets del proyecto *android* y se copia a la base de datos interna de la *app* con el método copyDataBase del AdminSQLiteHelper

El único requerimiento de cara a pre-crear una base de datos *sqlite* compatible con *Android* és la creación de una tabla android_metadata con el valor de idioma de la aplicación.

```
CREATE TABLE "android_metadata" ("locale" TEXT DEFAULT 'en_US')
INSERT INTO "android_metadata" VALUES ('es_ES')
```

El resto del fichero de volcado crea y carga cada tabla del modelo con los datos del *parser*.

Uso del Patrón de diseño Singleton

Al trabajar con la base de datos local de la aplicación *android* se ha pensado en el patrón singleton para garantizar que sólo se implementa una instancia de nuestra base de datos durante el ciclo de vida de la *app*.

De este modo, desde el onCreate de la actividad principal se inicializa la instancia de base de datos.

```
context = getApplicationContext();
private AdminSQLiteOpenHelper db = AdminSQLiteOpenHelper.instance(context);
```

Que a su vez comprueba si ya existe una instancia o sino la crea.

```
            public static AdminSQLiteOpenHelper instance(Context context) {
                if (sInstance == null) {
                    sInstance = new AdminSQLiteOpenHelper(context);
                }
                return sInstance;
            }
```

Normalmente al trabajar en entornos *Java* basta con definir una variable estática como instancia de clase pero haciéndolo de esta manera queda conceptualmente más claro el objetivo que se busca.

Por otro lado cabe denotar el uso de una definición de método estático con lo que se busca prevenir fugas de memoria *(*memory leaks); es decir, tener que volver a cargar en memoria nuestro objeto base de datos por haber reiniciado la actividad.[6]

En base a recomendaciones de la comunidad *Android* no se debe abusar del uso de este procedimiento ya que la memoria de *heap* reservada por aplicación es de 16Mb de la cual se dispone del mecanismo de Garbage Collector de la máquina virtual Dalvik para limpiar de objetos con frecuencia.

6 http://android-developers.blogspot.com.es/2009/01/avoiding-memory-leaks.html

Implementación de la clase AdminSQLiteOpenHelper

Cuando se trabaja con bases de datos *sqlite* en *android* hay una tendencia generalizada a heredar de SQLiteOpenHelper de cara a interactuar con la fuente de datos. De esta manera es obligatorio implementar los métodos onCreate y onUpgrade.

El método onCreate es llamado la primera vez que se crea la base de datos y por tanto es donde se define estructura de tablas y datos de carga iniciales mientras que el onUpgrade se llama cuando se debe actualizar la base de datos.

Desde un primer momento se identificó la dificultad de trabajar con este paradigma dado que ya se disponía de una carga inicial bastante amplia de datos (992 paradas, 30 líneas, sus interacciones y los horarios observados iniciales); por ello se orientó el desarrollo a una carga inicial completa basada en una base de datos precreada a partir de los datos del *web crawler.*

Motivo por el cual ambos métodos obligatorios, están vacíos:

```
@Override
public void onCreate(SQLiteDatabase db) {}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {}
```

Por otro lado en el constructor de AdminSQLiteOpenHelper se llama a la superclase para crear y mantener una referencia al contexto pasado por parámetro para poder acceder a los recursos de la aplicación

```
private AdminSQLiteOpenHelper(Context context) {

        super(context, DATABASE_NAME, null, DATABASE_VERSION);
```

Creamos una base de datos vacía en la ruta por defecto para popularla con nuestros datos

```
this.getReadableDatabase();
```

Y a continuación se copian los datos de la base de datos origen a la de sistema, vacía

```
public void copyDataBase() throws IOException {

    // Se abre la db local como input stream
    InputStream miInput = context.getAssets().open(DATABASE_NAME);

    // Ruta de la nueva db vacía
    String outFileName = DB_PATH + DATABASE_NAME;

    // Se abre la db vacía como output stream
    OutputStream miOutput = new FileOutputStream(outFileName);

    //Copia de datos:
    // transferimos los bytes del input al output
    byte[] buffer = new byte[1024];
    int length;
    while ((length = miInput.read(buffer)) > 0) {
        miOutput.write(buffer, 0, length);
    }

    // Se cierran los streams
    miOutput.flush();
    miOutput.close();
    miInput.close();
}
```

Gestión de Errores

Al no utilizar el mecanismo estándar de interacción con la base de datos (se copia pregenerada desde los assets de la aplicación cuando se instala) conviene validar que ésta existe en la ruta habitual (/data/data/aplicacion/databases/base.db)

Para ello se define un objeto File que carga la información a partir de la ruta del fichero *sqlite3*, información que luego podemos visualizar en los logs de depuración del LogCat.

```
File dbFile = context.getDatabasePath(AdminSQLiteOpenHelper.DATABASE_NAME);
Log.d("DB PATH", dbFile.getAbsolutePath());
Log.d("DB SIZE", "tamaño"+ dbFile.length() + "");
```

Conectividad

Es un factor funcional clave en la aplicación ya que necesitamos conocer la ubicación actual del usuario para poder compararla con la de las paradas de nuestra base de datos y así proporcionar la información de distancia a las mismas así como centrar el mapa en el usuario y visualizar las paradas cercanas.

Conviene usar *GPS* combinado con el *Network Location Provider* de Android para capturar la info de localización del usuario ya que aunque *GPS* es más preciso, sólo funciona en exteriores, consume batería y no devuelve la localización con un frecuencia razonable, al contrario del *network location provider* que a través de la Estación base y la señal WiFi que funciona in/outdoors, és rápido y consume menos batería.

Dentro de la gestión del servicio de *GPS* de cara a optimizar los recursos y no estar escuchando datos todo el tiempo, hay toda una serie de optimizaciones sobre las que se puede profundizar.

Por otro lado la API de localización (*Location*) nos permite utilizar otros servicios de ubicación como los proporcionados por la red, gps ya mencionado o modo pasivo:

·LocationProviders:

network	usa la red movil o WI-Fi para determinar la mejor localización; suele implicar una mejor precisión en habitaciones o espacios cerrados que GPS.
gps	uso del receptor GPS para obtener la mejor localización mediante satélitess GPS. Suele tener mejor precisión que network.
passive	permite participar en localizaciones de otros componentes y aplicaciones del dispositivo de cara a ahorrar energía en el uso de sensores.

Por ello vamos a definir el requerimiento de aplicación de que el servicio de GPS debe estar activado.
Como medida de seguridad se añade una llamada Intent para que el usuario active el servicio GPS si no se encuentra activo:

```
LocationManager service = (LocationManager) getSystemService(LOCATION_SERVICE);
boolean enabled = service
  .isProviderEnabled(LocationManager.GPS_PROVIDER);
if (!enabled) {
  Intent intent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
  startActivity(intent);
}
```

Dado que de esta manera estamos *obligando* al usuario a ir a la pantalla de configuración, se entiende necesario añadir un elemento condicional por parte del usuario en forma de caja de diálogo: *AlertDialog*.

La versión final de nuestra gestión del sensor GPS define dos clases *GPSManager* y *GPSListener* donde derivamos todo el código para la gestión del sensor, implementando por otra parte los métodos de *LocationListener* que nos permite gestionar las actualizaciones de ubicación del usuario; imprescindible para cuando se pretende gestionar, a modo de servicio en background, el desplazamiento dentro de un autobús y el reconocimiento de una parada en cada uno de estos casos.

Por tanto, en onCreate de la actividad principal nos queda una llamada al método GPSManager.start(), definición:

En onCreate de CommuterActivity.java:

```
        //instanciamos un objeto gps del GPSManager al que le pasamos el contexto de la actividad
                GPSManager gps = new GPSManager(CommuterActivity.this);
                gps.start();
```

Implementación del metodo GPSManager.start():

```
public void start() {
```

Obtenemos el servicio de localización a partir del contexto de aplicación mediante el método *getSystemService*. Método que por otro lado también nos da acceso a servicios de aplicación tan interesantes como :

- POWER_SERVICE para la gestión de energía de nuestra aplicación
- NOTIFICATION_SERVICE para informar al usuario de eventos en *background*
- ALARM_SERVICE para recibir *intents* en el momento que defina el usuario
- SEARCH_SERVICE para gestionar búsquedas en nuestra app

```
mlocManager = (LocationManager) activity
        .getSystemService(Context.LOCATION_SERVICE);

   if (mlocManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
   //Lanzamos la instancia de GPSListener que implementa LocationListener
   //para gestionar actualizaciones del sensor
                gpsListener = new GPSListener(activity, mlocManager);
```

Buscamos actualizaciones de ubicación

```
                mlocManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1, 1,
                 gpsListener);

                gpsListener.onLocationChanged(mlocManager
                                .getLastKnownLocation(LocationManager.GPS_PROVIDER));
```

Sistema de Notificación

En android hay principalmente 2 sistemas de notificación principales:

- el uso de objetos Toast para lanzar pequeños mensajes de información o debug:

```
            Toast.makeText(context,
                           "cursor rows " + Integer.toString(cursor.getColumnCount()),
                           Toast.LENGTH_SHORT).show();
```

Recibe por parámetros el **contexto** de la aplicación, definido por Activity.this o bien creando un objeto context y cargando el valor de getApplicationContext(); el mensaje a mostrar y la duración, definida por Toast.LENGTH_SHORT o LENGTH_LONG.

- Mediante objetos AlertDialog que proporcionan opcionalidad por parte del usuario.

En este caso aprovechamos para terminar con el código del método GPSManager.start() iniciado en la sección anterior de Conectividad.

Básicamente definimos una instancia de la clase (android.app.)AlertDialog con el contexto de aplicación como parámetro y llamamos al método Builder para diseñar nuestra caja de diálogo.

A continuación se definen las acciones en base a que el usuario pulse Sí o No, llamando al intent implícito que abre la ventana de configuración de GPS del dispositivo en caso afirmativo.

```
  } else {
       AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(activity);
       alertDialogBuilder
                       .setMessage("el GPS está deshabilitado, desea activarlo?")
                       .setCancelable(false)
                       .setPositiveButton("Habilitar GPS",
                               new DialogInterface.OnClickListener() {
                                   public void onClick(DialogInterface dialog,
                                           int id) {
                                       Intent callGPSSettingIntent = new Intent(

android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS);
                                       activity.startActivity(callGPSSettingIntent);
                                   }
                               });
               alertDialogBuilder.setNegativeButton("Cancel",
                       new DialogInterface.OnClickListener() {
                           public void onClick(DialogInterface dialog, int id) {
                               dialog.cancel();
                           }
                       });
               AlertDialog alert = alertDialogBuilder.create();
               alert.show();
```

También vamos a necesitar las capacidades de LocationServices para alertar al usuario de las paradas más cercanas:

Necesitamos las capacidades que ofrece esta clase para Registrar/desRegistrar la ejecución del Intent que ejecute el alta de una nueva entrada de horario en la tabla RealTimeTable al pararse el autobús en una ubicación cuyas coordenadas coincidan con las de una parada de la línea; es decir, cuando el dispositivo entre en el radio de proximidad de su par latitud/longitud.

Gestión del Mapa

Con la nueva API v2 de *gmaps* definimos un objeto *GoogleMaps* que nos permite usar:
no sólo latitud-longitud (target) y zoom (como hasta ahora), sino también de orientación (*bearing*) y ángulo de visión (*tilt*).

Definición del objeto CameraUpdate

El movimiento de la cámara (desde el punto de vista de usuario) se construye con un objeto *CameraUpdate*

Para movimientos más básicos como actualización de latitud y longitud o el nivel de zoom podemos usar la clase *CameraUpdateFactory* y sus métodos estáticos como puede ser CameraUpdateFactory.[zoomIn|zoomOut|zoomTo]()

Para actualizar la latitud/longitud de la cámara ejecutamos
CameraUpdateFactory.newLatLng(lat, long)

Por otro lado si se desea actualizar latitud, longitud y zoom al mismo tiempo se utiliza LatLngZoom(lat,long,zoom)

El movimiento lateral del mapa (*panning*) se consigue con los métodos de scroll:

- CameraUpdateFactory.scrollBy(scrollHoriz,scrollVert)

donde scroll es un identificador de pixel.

Tras crear el objeto con los parámetros de posición y nivel de zoom hay que llamar a los métodos moveCamera() /animateCamera() de nuestro objeto GoogleMap

También se puede capturar o tratar eventos; creación y gestión de marcadores, dibujo de líneas/polígonos, etc...

En la versión v1 de la API había que crear una nueva capa (*overlay*) para capturar los eventos principales de pulsación.
Eso ya no es necesario con la versión v2 donde el nuevo componente de mapas soporta directamente los eventos de *click*, interpretado como movimiento de cámara, entre muchos otros.

Al dispararse el método *onClick* se llama a setOnMapClickListener() con un nuevo *listener* sobreescribiendo onMapClick().
Este método, por otra parte, recibe el parámetro *LatLng* con las coordenadas sobre las que ha pulsado el usuario.

Finalmente es importante destacar que para traducir las coordenadas físicas a coordenadas de pantalla hay que definir un objeto de tipo **Projection**, que se obtiene llamando a getProjection();

Posteriormente se llama a toScreenLocation() para obtener las coordenadas (x,y) de pantalla donde el usuario pulsó.

También se definen los *listeners* para el cambio de cámara, es decir, movimiento por el mapa y para añadir marcadores.

Finalmente mencionar que en LocalCommuter se ha definido, como uno de los requerimientos de usuario, que el mapa se centré inicialmente en Palma con un zoom de 12 puntos, suficiente para que abarque el núcleo de población en base al tamaño del visor de *gmaps* en la vista.

Latitud: 39.578
Longitud:2.6446
Zoom: 12.5
Orientación: 355.1
Ángulo: 39.35

```
private static final LatLng PALMA = new LatLng(39.581,2.645);
private GoogleMap map;
map.moveCamera(CameraUpdateFactory.newLatLngZoom(PALMA, 12));
```

Finalmente comentar que para actualizar la posición del usuario se registra un LocationListener mediante una variable de clase LocationManager para obtener actualizaciones de posición.

GUI

Orientación

Se establece el requerimiento de que la aplicación funcione con una única orientación en *portrait* para no tener que diseñar un nuevo *layout* apaisado ni que contemplar la lógica para guardar estados necesaria en Android, donde, al cambiar la orientación, todos los elementos (o vistas) de una actividad, inicializan sus valores por defecto.

Secuencia de Eventos del Ciclo de Vida al **Cambiar de Orientación** una Activity:

```
   onSaveInstanceState → onPause →onStop →
→ onCreate→onStart→onRestoreInstanceState→onResume
```

Para ello basta con añadir la propiedad android:screenOrientation=”portrait” a nuestro Manifest:

android:label="@string/app_name" android:screenOrientation="portrait">

A pesar de no implementarse, la lógica para contemplar el cambio de orientación se limita a guardar el estado de nuestras vistas principalmente con 2 métodos:

```
	@Override
	public void onRestoreInstanceState(Bundle savedInstanceState) {
		// Se restaura la posición del menú de navegación previamente serializado.
		if (savedInstanceState.containsKey(STATE_SELECTED_NAVIGATION_ITEM)) {
			getActionBar().setSelectedNavigationItem(
					savedInstanceState.getInt(STATE_SELECTED_NAVIGATION_ITEM));
		}
	}

	@Override
	public void onSaveInstanceState(Bundle outState) {
		// Se serializa la posición actual del menú drowpdown de navegación
		outState.putInt(STATE_SELECTED_NAVIGATION_ITEM, getActionBar()
				.getSelectedNavigationIndex());
	}
```

Con ellos se definen los listeners que escuchan el evento de cambio de estado en el Ciclo de Vida de la Actividad, de Activa a en Pausa y de en Pausa a onResume -> Activa.

Por tanto es en estos métodos donde tenemos que guardar las tuplas de estado en el objeto Bundle que define la instancia de nuestra actividad.

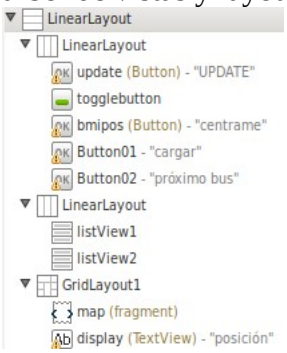
Uso de ActionBar

Durante el diseño de la aplicación se ha evolucionado de un *layout* inicial sobrecargado con un LinearLayout principal. dos LinearLayouts anidados además de un *GridLayout* que dificultaba sobremanera cualquier rediseño o modificación, a un diseño más minimalista con un único *LinearLayout* vertical.

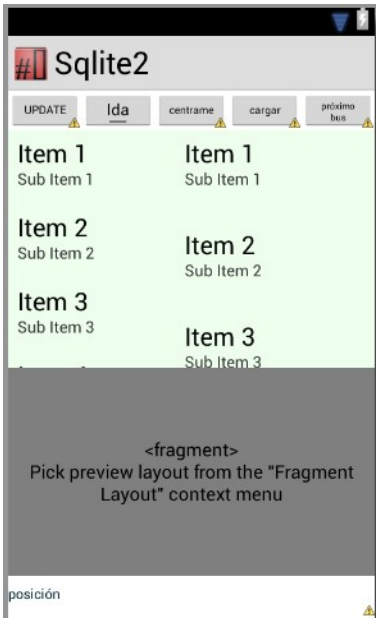
La eliminación de la botonera superior, substituyéndola por una elegante *ActionBar* y recuperando por tanto espacio en pantalla para agrandar el mapa y el *TextView* de notificaciones inferior además de la incorporación del *SearchView* son otras mejoras destacables en la *UI* gracias a *ActionBar*.

Diseño inicial

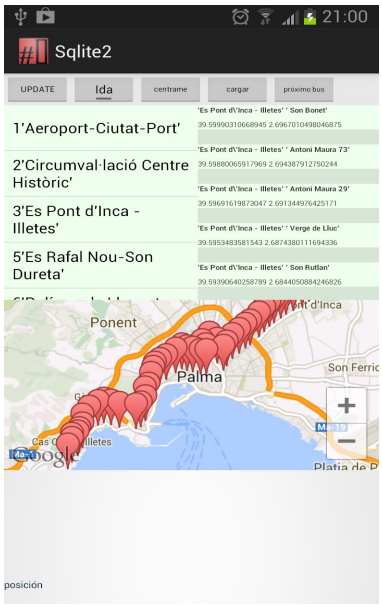
arbol de vistas y layouts



diseño visual

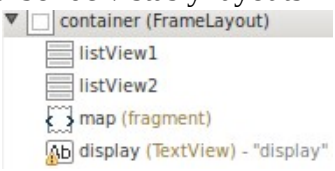


en funcionamiento

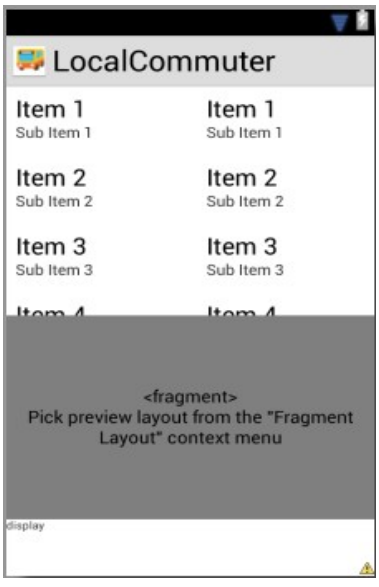


Diseño final

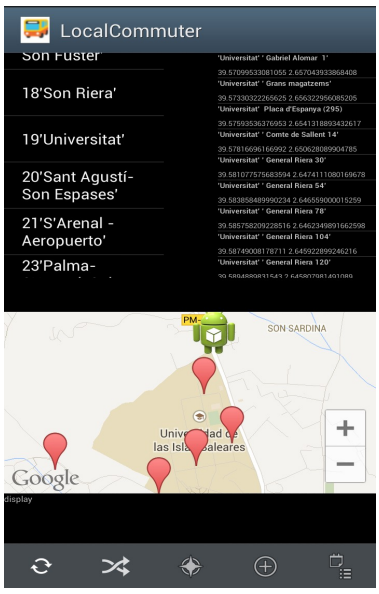
arbol de vistas y layouts



diseño visual



en funcionamiento



ActionBar es un destacado elemento de diseño para las apps Android que reemplaza la funcionalidad del menú de opciones, extendiendo sus capacidades a nivel de *framework.*

Puede mostrar el título, icono o las acciones que se pueden disparar en la *activity*. También permite la navegación por la aplicación de manera fluida entre distintos contextos de la aplicación ya sea mediante el botón *atrás*, el icono de la aplicación en la barra superior de título, pestañas, etcetera.

Entre las opciones que proporciona la barra de acciones para mostrar nuestras distintas opciones de navegación se encuentran:

- Pestañas,
- Dropdown list o listas desplegables
- Menu options, el menú de opciones típico de cualquier *app Android.*

Es común, como en el caso de LocalCommuter, completar el menú de opciones con acciones (*actions)* ejecutables dentro de la ActionBar mediante la definición de onCreateOptionsMenu().

Las *actions* se definen en un fichero xml mientras que se usa la clase MenuInflator para inflar las *actions* y añadirlas al *ActionBar.*.

A su vez se define el atributo *showAsAction* que permite definir como se muestra la acción, por ejemplo con ifRoom se activa la *action* y se muestra si hay sitio en el menú de opciones.

En nuestro caso implementamos la selección de acciones en ActionBar mediante la llamada al método onOptionsItemSelected() que recibe la *action* correspondiente como parámetro, permitiendo a partir de aquí implementar su funcionalidad.

Finalmente se tiene que onCreateOptionsMenu sólo se llama una ve así que si se pretende cambiar el menú más tarde hay que lanzar invalidateOptionsMenu() y luego ya sí se llama a onCreateOptionsMenu()

Es importante definir por otro lado un elemento *android:title* para cada ítem de menú en menu.xml

Interacción con ListView y Cursor de Base de datos

En este método de la clase LocalCommuter.java se define la acción de actualización del *ListView* de líneas.

En la actualización de un *ListView* a partir de un cursor de base de datos entran en juego 3 elementos principales: el propio *listview*, el cursor y el método *adapter* que hace de *datasource* para rellenar el contenido del *listview.*

De hecho , como se verá en el subapartado siguiente, personalizando y asignando un fichero de estilos (en xml) se puede cambiar la apariencia y el tipo de información que se muestre, ya sea en varias columnas o añadiendo una imagen en miniatura (*thumbnail)* de la línea.

```
public void updateListView() throws IOException {
		Cursor cursor = null;
		final ArrayList<String> list = new ArrayList<String>();
		final ArrayAdapter adapter = new ArrayAdapter(this,
				android.R.layout.simple_list_item_1, list);
		listView.setAdapter(adapter);
		try {
			cursor = db.rawSelect("SELECT * FROM " + TABLE_LINES);

			while (cursor.moveToNext()) {
				int id = cursor.getInt(cursor.getColumnIndex(COLUMN_ID));
				String s = cursor.getString(cursor.getColumnIndex(COLUMN_NAME))
						.replace("\\", "");
				lineas.put(Integer.valueOf(id), s);
				list.add(id + s);
			}
			cursor.close();
```

Personalización de ListView

Personalizar Adapter para mostrar información con fuentes pequeñas en ListViews.

En *android* hay toda una serie de plantillas que se pueden vincular con los objetos visuales *ListView* para mostrar un listado personalizado de información, normalmente recuperada por un cursor de base de datos. De todas formas, esas plantillas o *layouts* predefinidos no siempre se adaptan a las necesidades del desarrollador; es en ese caso en el que hay que definir tanto la clase que gestione el objeto *Adapter* como la plantilla de estilo *layout* con la apariencia deseada para cada registro o vista del *ListView*.

Normalmente se usa la clase ArrayAdapter para rellenar el contenido de un *ListView* pero si queremos modificar la apariencia ya hay que definir un *Adapter* nuevo derivado de BaseAdapter.

En el caso que nos ocupa se ha aplicado este método para adaptar el *ListView* de **Paradas**, en el que se pretendía mostrar más información por registro, concretamente dos filas con las coordenadas geográficas, el nombre de la línea y de la parada.

Para ello se crea la clase ShowStopsAdapter.java que hereda de BaseAdapter e implementa, en su método principal el proceso de definir un *arraylist* de paradas

```
public class ShowStopsAdapter extends BaseAdapter {
	Activity activity;
	ArrayList<Stop> items;
	private Context mContext;

	private LayoutInflater mLayoutInflater;
```

Al que en el método más relevante de la clase: getView, se vincula la vista actual con el layoutinflater que se encarga de rellenar la vista con la plantilla *layout* asignada, en este caso la plantilla list_view_item_row.xml definida en la ruta /layout del proyecto.

Contenido del layout del ListView de Paradas:

```
    <TextView
        android:id="@+id/listTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@+id/listImage"
        android:layout_toRightOf="@+id/listImage"
        android:text="Línea"
        android:textSize="10sp"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/listDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/listTitle"
        android:layout_marginTop="5dp"
        android:layout_toRightOf="@+id/listImage"
        android:maxLines="4"
        android:text="Dirección"
        android:textSize="12sp" />
```

Definición del método getView:

```
	public View getView(int position, View convertView, ViewGroup parent) {


		if (convertView == null) {
		    LayoutInflater inflater = (LayoutInflater) this.mContext
			    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
		    convertView = (View) inflater.inflate(
			    R.layout.listview_item_row, null);
		}
```

Definición de dos Vistas que corresponden con las del layout y a las que se asigna los valores del cursor actual pasados por parámetro.

```
	TextView tvtitle = (TextView) convertView.findViewById(R.id.listTitle);
	TextView tvdesc = (TextView) convertView.findViewById(R.id.listDescription);
	tvtitle.setText("Parada " + items.get(position).getId() + "  "
+items.get(position).getName());
	String str = Double.toString(items.get(position).getLat()) + " " +
Double.toString(items.get(position).getLng());

	tvdesc.setText(str);
	return convertView;
	}
```

Búsqueda de Paradas mediante Search View

Antes de Android 3.0 el desarrollador debía utilizar dos Activities distintas para poder implementar dos acciones distintas mediante *Intents*.

Esto sigue siendo igual en las últimas versiones (4.3) ya que en el ciclo de vida de una aplicación *android*, sólo una actividad puede ser presentada al usuario al mismo tiempo, con la salvedad de que se puede implementar búsquedas en la misma actividad mediante *Search Views* y los métodos *Searchable*.

Para cualquier otro supuesto, es necesario utilizar Fragments para interactuar con *varias actividades al mismo tiempo*.

En el caso de LocalCommuter se ha instanciado un elemento *SearchView* que se integra de manera elegante y muy intuitiva dentro del conjunto de *ActionBar*.

Su implementación se reduce por un lado a los siguientes procesos:

Se añade un ítem de menú en menu.xml donde se definen los parámetros típicos de un ítem de menú: identificador, ruta del icono, título, las características de visualización con el tag showAsAction y finalmente la parte más importante, la vinculación con su *actionViewClass*

```
  <item android:id="@+id/action_search"
          android:icon="@android:drawable/ic_menu_search"
          android:title="Busqueda de paradas"
          android:showAsAction="collapseActionView|ifRoom"
          android:actionViewClass="android.widget.SearchView" />
```

Por otro lado se tiene que definir un xml en res/xml/searchable.xml con la definición de la caja de búsqueda

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
        android:label="@string/app_name"
        android:hint="@string/search_hint" />
```

aquí se puede personalizar sobremanera la apariencia, en el caso de la aplicación actual se deja la configuración mínima para que funcione la interacción del *search view.*

A su vez en el Manifest hay que añadir la declaración del Intent así como vincularlo con el xml anterior, dentro de la etiqueta <activity></activity>.

```
<intent-filter>
                <action android:name="android.intent.action.SEARCH" />
            </intent-filter>

            <meta-data
                android:name="android.app.searchable"
                android:resource="@xml/searchable" />
```

Finalmente la implementación dentro de la actividad.
En onCreate se carga la consulta introducida por el usuario en el campo de búsqueda, se comprueba la acción y obtiene el string de input de usuario

```
Intent intent = getIntent();
if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String query = intent.getStringExtra(SearchManager.QUERY);
        buscaParadas(query);
}
```

Por último se llama al método buscaParadas con el string de búsqueda por parámetro.

La implementación es básicamente definir la consulta *sql* dentro de RutaLínea, para poder reutilizar el código de consulta y pintado de paradas del método onListView1 donde se selecciona la línea y se obtienen todas sus paradas. Por ello, al no llamar directamente a la tabla Paradas, aplicamos el modificador *distinct* para optimizar y asegurarse de que no se recuperan dos veces la misma parada.

```
String sqlquery = "select distinct l._id, l.name, p.name , p.lat, p.long, rl.orden,  rl.numparada
from lineas as l, parada as p, RutaLinea as rl where ( l._id = rl.numlinea ) and (rl.numparada =
p._id ) and lower(p.name) like \"%" + query + "%\"";
```

Localización: calculo de distancia entre el usuario y una parada

La implementación de la funcionalidad para el calculo de distancias usuario-parada hace uso de la API de Localización que en sus últimas versiones ha simplificado su uso pudiendo reducir a este pequeño scriptlet, donde en base a la posición del usuario y a la de la parada seleccionada en el ListView1, se convierten a objetos *Location* y su posterior comparación mediante el método distanceTo devuelve la distancia en metros.

Recuperación de la información del objeto Stop vinculado con la posición pulsada en el *listview*

```
paradactual = paradas.get(position);
LatLng pos = new LatLng(paradactual.getLat(), paradactual.getLng());
CameraUpdate centro = CameraUpdateFactory.newLatLngZoom(pos, 19);
mapa.moveCamera(centro);
// Calculamos la distancia del usuario con respecto a la parada
Location locationA = new Location("usuario");
TextView display = (TextView) findViewById(R.id.display);
if ((mipos.latitude > 0) && (mipos.longitude > 0)) {
        locationA.setLatitude(mipos.latitude);
        locationA.setLongitude(mipos.longitude);
        Location locationB = new Location("parada");

        locationB.setLatitude(paradactual.lat);
        locationB.setLongitude(paradactual.lng);

        float distance = locationA.distanceTo(locationB);

        display.setText("Esta parada está a"
                        + Float.toString(distance) + " metros del usuario");

        } else {
                display.setText("Posición de usuario no inicializada:"
                                + mipos.latitude + " " + mipos.longitude);

                }
```

Accesibilidad

A pesar de que en la demo el tamaño de texto de los listview y edittext es de una fuente pequeña arial 10, está definida usando la unidad sp (independiente de escala) en lugar de dp ya que combina el tamaño original de letra (en puntos por pixel) junto con las preferencias del dispositivo android donde se esté usando la app, por ello si el usuario tiene problemas de visión y ha definido una fuente de sistema de gran tamaño, esto se verá reflejado en nuestra app.

Conclusiones y visión de futuro

Durante el desarrollo de este proyecto se ha pasado por una gran parte de las APIs de programación del entorno *Android*, lo que ha permitido conocer en profundidad su desarrollo y evolución.

No cabe duda de que es una plataforma de desarrollo con un potencial increíble, donde todo está a disposición del programador, sin limitaciones geográficas o de poder adquisitivo, cualquiera puede llevar a cabo sus proyectos para entornos móviles independientemente del país de origen gracias a la apertura de conocimiento que supone un entorno de código abierto.

En cuanto a este aspecto, no cabe duda de que Google optó por el mejor de los caminos al abrir su código y hacerlo disponible a cualquiera, propiciando un ecosistema de fabricantes, desarrolladores o incluso implementadores de ROMs (versiones modificadas del *firmware Android*) capaces de adaptar cualquier versión del sistema operativo a sus propias necesidades o a las de los demás; o bien modificar todo el sistema eliminando u optimizando lo que le interese.

En cuanto a la evolución del sistema *Android*, ha supuesto en algunos casos un inconveniente, como en el caso del paso de la versión 1 de la API de *google maps* a la v2, al tener que aprender de nuevo sus métodos y metodologías aunque también ha sido un factor de mejora, ya que en ese mismo ejemplo la eficiencia del renderizado de mapas, marcadores o la propia

interacción usuario – máquina

ha supuesto una mejora considerable; así como la facilidad y simplificación que supone, desde el punto de vista del desarrollo, el paso de una versión de la API a otra más moderna.

Algo similar se ha observado con la aparición del entorno de *ActionBar*, donde se ha podido homogeneizar las acciones de menú con la interfaz de la *app*, algo que hubiera sido imposible de otra manera y sin disponer de grandes habilidades para el diseño gráfico.

Estos factores ponen en relieve la constante evolución que están viviendo los sistemas operativos para móviles, donde es más fácil que nunca quedarse obsoleto con una plataforma de desarrollo que cambia de versión todas y cada una de sus librerías de programación como mínimo cada 2 años sin ser este un factor temporal fijo.

En cuanto a la parte que más trabajo ha llevado en este proyecto, es decir, la fase de recolección de datos y su automatización a través del *crawler python* así como la extracción de información de los pdf de horarios; parte más laboriosa y menos fructífera ya que ni tan siquiera se ha podido utilizar

esos datos en la aplicación móvil por no ser más que una aproximación temporal de los datos reales.

Justo cuando la propia EMT acaba de publicar una *app* hecha por terceros donde precisamente el contenido que se ofrece al usuario es esa información que no se ha podido conseguir de ninguna manera, es decir la información sobre **el paso del próximo bus por la parada X**, gracias al sistema de *GPS* incluido en cada autobús y que de alguna manera se sincroniza con la sede central.

Lamentablemente no se ha podido identificar la manera de acceder a dicha información mediante fuentes abiertas ni tampoco por ingeniería inversa (algo que sí se ha podido hacer con el resto de información del sistema de transportes).

Por ello se podría considerar un acuerdo UIB–EMT para el intercambio de información, o bien el acceso a información en tiempo real de sus buses de linea por cada parada así como la pertinente recomendación a EMT de que integre sus líneas en *Google Transit*, para que cualquier usuario de gmaps pueda interactuar con los buses EMT en sus itinerarios de transporte por Mallorca.

En cualquier caso y a título personal cabe mencionar el aprendizaje que ha supuesto esta inmersión en el desarrollo para la plataforma *Android* en un proyecto donde se ha pasado por la gran mayoría de APIs de programación de que dispone el entorno.

No cabe duda de que la baja curva de aprendizaje y lo bien estructurado que és el lenguaje java en uso, así como la gran cantidad de librerías disponibles, pasando por la constante evolución y aparición de nuevos IDEs como Eclipse, Netbeans o Intellij Idea en su versión comunitaria del que incluso *Google* ha derivado recientemente su AndroidStudio.

Bibliografía

Referencias django/python

https://docs.djangoproject.com/)
django-piston (https://bitbucket.org/jespern/django-piston/
http://www.djangobook.com/en/2.0/index.html

Referencias android/java
Blog de S. G. Oliver sobre desarrollo en Android
http://www.sgoliver.net/

Web sobre desarrollo Android de Lars Vogel
http://www.vogella.com/articles/Android

Web oficial de Android developers
http://developer.android.com

Sitio web de interacción con otros desarrolladores y resolver dudas de programación
http://stackoverflow.com

Implementación del Parser JSON en Android: http://www.androidhive.info

Anexo

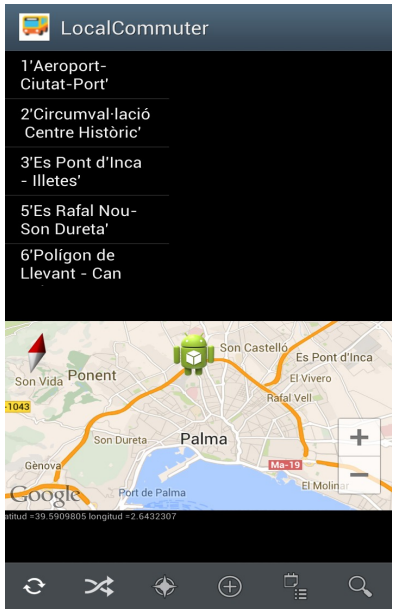
Juego de Pruebas

Uso genérico de LocalCommuter

1.El usuario accede a la aplicación sin GPS habilitado y recibe el dialogo para habilitar GPS

2.El usuario carga la lista de líneas con el botón de refrescar 

El mapa se centra en palma y posiciona al usuario en él

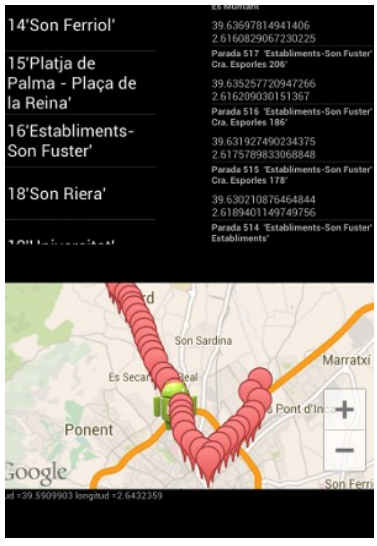


3.El usuario pulsa sobre la línea 16 en el ListView de líneas

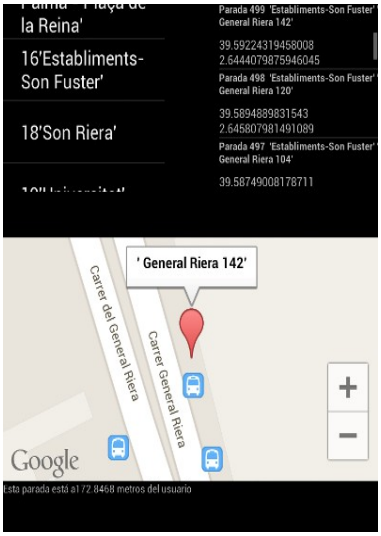
Inmediatamente se rellena el listado derecho con las paradas de la línea en sentido ida además de pintarse los marcadores de cada parada en el mapa.

Si se pulsa el botón para cambio de sentido 

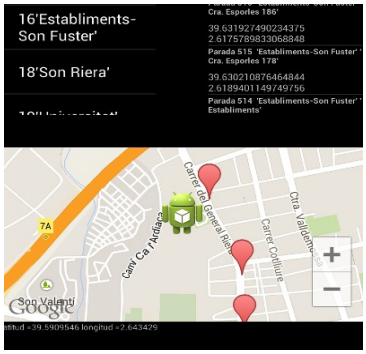
se recargan las paradas en la lista y en el mapa.



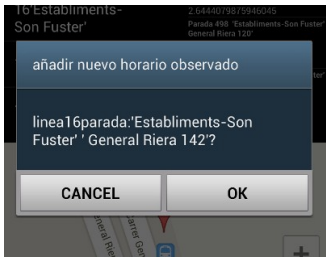
4.Al pulsar sobre una parada del listado se centra el mapa en su marcador.
A su vez se muestra la distancia del usuario a la parada en la vista informativa de abajo



5.Al pulsar el botón de centrar usuario 
se centra el mapa en el mismo, dibujando un androide verde para diferenciarlo de las paradas.

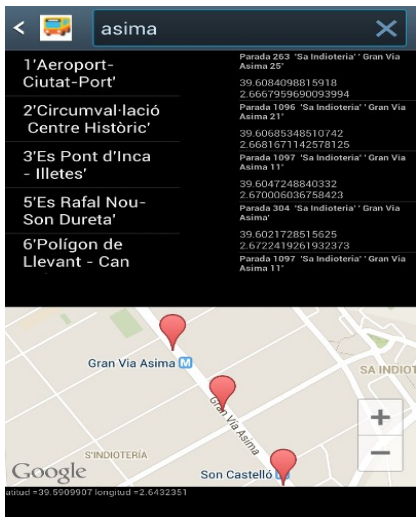


6.Con la línea y la parada seleccionada el usuario puede añadir un nuevo horario de pasada del bus en cuanto pulse sobre 



7.Si se quiere consultar la hora a la que pasa el siguiente bus por la combinación línea-parada actual (seleccionada) se lanza la acción con 

8.Finalmente, para buscar paradas por nombre se acciona el botón correspondiente que activa el SearchView 
En este caso se busca cualquier coincidencia con el término “asima”, mostrándose tanto la lista de paradas con el patrón así como pintándolas en el mapa.



Salida del Parser Pdf

donde se puede comprobar el formato desestructurado del contenido que impide automatizar la tarea de captura de horarios para todas las líneas.

10:18
14:03
21:03

10:33
14:18
21:32

10:48
14:33

Itinerari: Avinguda Gabriel Alomar, Corte Inglés, Avinguda Alexandre Rosselló, Comte Sallent, General Riera, Germanetes, Andreu Torrens, Balanguera, Rafael Rodríguez Méndez, Passeig Mallorca, Jaume III, Plaça de la Reina, Conquistador, Cort, Colom, Bosseria, nous Jutjats, Escola Graduada, Avinguda Alexandre Rosselló, Corte Inglés, Avinguda Gabriel Alomar

gestDic 2 imgdb/264127.pdf
pdf de la linea 2: http://www.emtpalma.es/EMTPalma/Front/imgdb/264125.pdf
descarga de pdf http://www.emtpalma.es/EMTPalma/Front/imgdb/264125.pdf
264125.pdf
./pdfs/264125.pdf
2 CIRCUMVAL□LACIÓ CENTRE HISTÒRIC

Feiners

02/09/2013

SORTIDES DE GABRIEL ALOMAR
08:06
07:55
07:44
07:33
07:22
07:11
07:00
10:51
10:40
10:29
10:18
10:07
09:56

Definición de Entidades del modelo GTFS

Hay una serie de entidades que hay que integrar en nuestro modelo de datos ER para compatibilizar nuestra Base de Datos con GTFS de cara a poder publicar e integrar las rutas de nuestro sistema de transporte con el entorno de Google Maps.

Agencia
Operador de una red de transportes públicos, normalmente una autoridad pública. Las Agencias se definen en agency.txt y pueden tener URL, telefonos o indicadores de idioma. Si estás proporcionando un feed que incluye vehículos operados por distintas agencias, puedes definir múltiples agencias en este fichero y asociarlas con cada Viaje(trip)

agency.txt
agency_id, agency_name,agency_url,agency_timezone,agency_phone,agency_lang
FunBus,The Fun Bus,http://www.thefunbus.org,America/Los_Angeles,(310) 555-0222,en

Ruta
Una ruta en el stand GTFS equivale a una Línea de un sistema de transporte público y se compone de uno o varios viajes (Trips)

routes.txt
route_id,route_short_name,route_long_name,route_desc,route_type
A,17,Mission,"The ""A"" route travels from lower Mission to Downtown.",3

Viaje (Trip)
Representación de un trayecto realizado por un vehículo mediante paradas (Stops). Los Viajes son específicos del tiempo – se definen como una secuencia de Paradas

La entidad Viaje usa Calendario para definir los días en que un Viaje está disponible.

trips.txt
route_id,service_id,trip_id,trip_headsign,block_id
A,WE,AWE1,Downtown,1
A,WE,AWE2,Downtown,2

Service
Los servicios definen un rango de fechas entre las cuales un Viaje está disponible en un rango de días de la semana, definidos en calendar.txt.

Un Servicio único se puede aplicar a diferentes Viajes y si un vehículo dado tiene distinta planificación (una para días de la semana y otra para el fin de semana) debería haber 2 entradas Viaje con las mismas Paradas asociadas a Servicios distintos así como distintos tiempos de parada.

Parada: stops.txt
```
stop_id,stop_name,stop_desc,stop_lat,stop_lon,stop_url,location_type,parent_stat
ion
S1,Mission St. & Silver Ave.,The stop is located at the southwest corner of the
intersection.,37.728631,-122.431282,,
S2,Mission St. & Cortland Ave.,The stop is located 20 feet south of Mission
St.,37.74103,-122.422482,,
```

Horario Paradas:stop_times.txt
```
trip_id,arrival_time,departure_time,stop_id,stop_sequence,pickup_type,dropoff_ty
pe
AWE1,0:06:10,0:06:10,S1,1,0,0,0
AWD1,0:06:20,0:06:20,S3,3,0,0,0
```

calendar.txt

```
service_id,monday,tuesday,wednesday,thursday,friday,saturday,sunday,start_date,e
nd_date
WE,0,0,0,0,0,1,1,20060701,20060731
```

WD,1,1,1,1,1,0,0,20060701,20060731

calendar_dates.txt

Se reflejan las excepciones en el servicio, por ejemplo WD (servicio Week Day) el día 3/7 de 2006 tiene la excepción 2, servicio no disponible. Por otro lado WE (week end) tiene la 1, disponible..

```
service_id,date,exception_type
WD,20060703,2
WE,20060703,1
WD,20060704,2
```