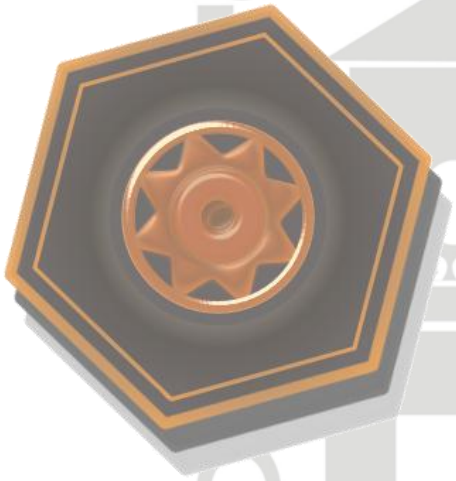
Proyecto Turawet:

Integración de herramientas de modelado, recopilación y explotación de datos. Aplicaciones prácticas en casos reales.



Autores: Francisco José Cabrera Hernández
Nicolás Pernas Maradei
Romén Rodríguez Gil

Director: José Luis Roda García

Fecha: 14 de Julio de 2011

D. José Luis Roda García, Profesor de la Escuela Técnica Superior de Ingeniería Informática, y adscrito al Departamento de Estadística, Investigación Operativa y Computación de la Universidad de La Laguna.

CERTIFICA: Que la presente memoria titulada ***Proyecto Turawet: Integración de herramientas de modelado, recopilación y explotación de datos. Aplicaciones prácticas en casos reales.***, ha sido realizada bajo mi dirección por los estudiantes **Francisco José Cabrera Hernández, Nicolás Pernas Maradei y Romén Rodríguez Gil**, y constituye su Proyecto Fin de Carrera de Ingeniería Informática por la Universidad de La Laguna.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos que haya lugar, firmo el presente certificado en La Laguna, a **14 de Julio de 2011**.

Fdo.: D. José Luis Roda García

Resumen

Esta memoria pretende describir el proyecto Turawet, un proyecto que comenzó su andadura en Julio de 2010 cuando los tres integrantes de este proyecto nos reunimos con el que posteriormente sería nuestro director, José Luis Roda García para comenzar a definir lo que terminaría siendo plasmado en este documento como nuestro proyecto de fin de carrera.

A finales de Julio de 2010 se celebró la primera reunión formal en la sede de Gerencia de Urbanismo de La Laguna, siendo este hito, la fecha que estableció el inicio formal del proyecto.

Inicialmente Turawet se planteó más como una evolución de un proyecto anterior desarrollado por otros alumnos en colaboración con Gerencia de Urbanismo de La Laguna, llamado GeoBloc [1]; que como un proyecto propio. Rápidamente se observó que el proyecto que se estaba dibujando tenía unas características marcadamente diferentes a su predecesor, con lo que se decidió comenzar un nuevo proyecto desde la base, pudiendo así tomar nuestras propias decisiones de diseño y no depender de las restricciones que nos imponía GeoBloc. Nuestro antecesor quedó como una experiencia previa de la que extraer información y aprender, tanto de sus aciertos como de sus errores.

Turawet rápidamente se definió como un proyecto ambicioso y que pretendía ser una solución integral a la recolección de datos. Uno de los primeros puntos de acuerdo a los que se llegó en la etapa de análisis fue optar por un desarrollo genérico, en lugar de desarrollar una aplicación *ad-hoc* para las inspecciones de urbanismo como en la implementación que nos precedía. Éste desarrollo genérico propuesto, luego sería fácilmente exportable a las inspecciones de urbanismo, pero también a muchos otros campos, como por ejemplo a uno de los casos que finalmente se aplicó, el inventario de poblaciones de plantas amenazadas.

Con el transcurso del tiempo, la definición de requisitos y comienzo del diseño de la aplicación, se concluyó que, como solución integral que pretendía ser, este proyecto debería incluir, en primer lugar, un **sistema de modelado de formularios**, que permitiera crear formularios de forma cómoda, ágil y que fuera sencillo de utilizar para un usuario no técnico. En segundo lugar se esperaba desarrollar una **aplicación móvil que permitiera la recolección** de datos, creando instancias a partir de los formularios disponibles y enviándolas, una vez finalizadas a un repositorio o almacén. Por último y siempre en el marco de dar una solución integral para la recolección de datos, se deseaba encontrar una forma de explotar los datos recolectados. A tal fin se ideó una **herramienta de administración y cuadro de mandos**, desde la que se pudieran hacer consultas a los datos recolectados, gestionarlos y obtener estadísticas o mapas georreferenciados con las diferentes instancias de los mismos.

El proyecto poco a poco fue definiendo las líneas de actuación y fue tomando cuerpo y materializándose hasta llegar a transformarse en un prototipo final que integra las tres herramientas principales anteriormente descritas:

1. Se diseñó y creó un modelador web fácilmente utilizable y basado en las tecnologías web más recientes (HTML5 [2], jQuery [3] y CSS3 [4],…).

2. Se desarrolló una herramienta de recolección de datos para dispositivos basados en Android [5].
3. Por último, se desarrolló un administrador web que integra un sistema de base de datos para el almacenamiento de instancias y formularios, así como un cuadro de mandos que incluye generación de estadísticas de formularios y geolocalización de instancias.

Cabe destacar que todas estas herramientas fueron diseñadas para ser capaces de interoperar entre sí a través de servicios web y que finalmente, y para ilustrar el potencial del proyecto, se han aplicado en tres casos reales que son:

1. El inventario de actividades (en colaboración con Gerencia de Urbanismo de La Laguna [6]).
2. La notificación de desperfectos en las infraestructuras municipales (como iniciativa propia orientada al Ayuntamiento de La Laguna).
3. Los inventarios de poblaciones de plantas amenazadas (en colaboración con el departamento de Biología Vegetal de la Universidad de La Laguna [7]).

Agradecimientos

En primer lugar queremos agradecer a nuestro director de proyecto, José Luis Roda García, por darnos la oportunidad de desarrollar una herramienta con tanto potencial y por arriesgarse siempre a innovar y confiar en el alumnado, dándonos gran libertad en la toma de decisiones. A su vez queremos agradecerle haber estado colaborando estrechamente con nosotros en todo lo que le hemos solicitado y haber aportado sugerencias, ideas de mejora, recursos y contactando con profesionales de las tecnologías de la información y otros ámbitos que nos pudieran asesorar en el desarrollo, interesarse por el proyecto o colaborar en el desarrollo de casos prácticos.

Además queremos agradecer a la que fuera, en cursos pasados, nuestra profesora de base de datos, Virginia González Rodríguez, por responder a nuestras dudas sobre el diseño de la base de datos en las etapas iniciales del proyecto.

También queremos agradecer a Luis López Cabrera y a Roberto Martín Gorrín, junto al resto del departamento de informática de Gerencia de Urbanismo de La Laguna, su colaboración, aportación de ideas y el tiempo invertido en las reuniones; así como su ayuda para poder lanzar uno de los ejemplos finales: el inventario de actividades.

Continuando con los casos de ejemplo desarrollados, queremos agradecer al Departamento de Biología Vegetal de la Universidad de La Laguna, y en concreto a los profesores Octavio Rodríguez Delgado, Antonio García Gallo y Pedro Luis Pérez de Paz su colaboración en el desarrollo del formulario de poblaciones de plantas amenazadas y el tiempo invertido en modelar el formulario ajustándose a las posibilidades de nuestra herramienta. Además, agradecer a Octavio el tiempo invertido en recolectar instancias de dicho formulario de plantas amenazadas a fin de que el caso de ejemplo tuviera el valor añadido de poseer datos reales.

Queremos agradecer a Pedro González Yanes, administrador del Centro de Cálculo de la Escuela Técnica Superior de Ingeniería Informática, al igual que a Carlos Peña Dorta y Antonio Manuel López González, ambos de Arte Consultores Tecnológicos S.L, por reunirse con nosotros y darnos su opinión del prototipo, así como consejos sobre usabilidad del mismo y facilitarnos nombres de productos similares de los que extraer ideas o con los que compararnos.

También queremos agradecer a Carlos Moisés Castillo Velázquez de Emprende ULL [8] y a Lorenzo García García de la OTRI [9] por hacernos ver el potencial de este proyecto e intentarnos asesorar sobre aspectos comerciales y legales del proyecto que en un futuro podría explorarse como producto comercial.

Merece a su vez agradecimiento la FEULL (Fundación Empresa Universidad de La Laguna), como organización, por concedernos un servidor con IP pública desde el que ejecutar nuestra herramienta de Administración y donde almacenar nuestra base de datos.

Por otra parte, y como no puede ser menos, los tres queremos agradecer a nuestras familias y parejas su apoyo incondicional y ánimos cada vez que los hemos necesitado.

A mis padres, Francisco y Margarita, sin su lucha y esfuerzo no hubiese podido llegar hasta aquí. A mis hermanas, por la ayuda prestada y el ánimo en los momentos más difíciles. A Leticia, su cariño, comprensión y apoyo han sido el impulso necesario para afrontar este último tramo.

Francisco J. Cabrera Hernández.

Quiero dedicar este trabajo, y el de estos últimos 5 años, a toda mi familia que siempre me ha apoyado desde cerca y a los otros que lo hacen al otro lado del charco. En especial a mi madre, por meterme en la cabeza la idea que me hizo llegar hasta acá. También a mi segunda familia, por la ayuda incondicional que me han dado durante estos años, que sin ellos no hubiese sido posible. Y por supuesto a mis compañeros de equipo, que esto recién empieza ;)

Nicolás Pernas Maradei

A Mónica, gracias a quien he vivido muchos de los mejores y más felices momentos de mi vida. Gracias, mi amor, por tu apoyo incondicional estos últimos cinco años; gracias por tu contagiosa alegría y por tu infinito cariño.

A mis padres y a mi hermano, por ayudarme y darme ánimos cuando lo he necesitado.

Romén Rodríguez Gil.

Tabla de contenidos

Lista de figuras y tablas

Parte I. Introducción y estado del arte

Capítulo 1. Introducción

Resumen:

- Se describirá el proyecto Turawet de forma general.
- Haremos un repaso y explicación del vocabulario que se utilizará en este documento.
- Se plantearán los objetivos principales que nos marcamos antes y durante el desarrollo del proyecto.
- A modo de curiosidad, explicaremos el porqué de Turawet como nombre para nuestro proyecto.

1.1 Descripción del proyecto

El proyecto Turawet nace con la finalidad de dar una solución electrónica, genérica e integral a la recolección de datos vía web y dispositivos móviles.

Como solución electrónica que es, pretende evitar los engorros que ocasionan los documentos en papel, facilitar su organización, mejorar las búsquedas y permitir explotar los datos recolectados. Conocido el hecho de que, por lo general, la recolección de datos manual redunda en pérdidas de tiempo, imprecisiones y en posibles pérdidas de información. Todo ello infiere en una recolección ineficiente, a veces inconsistente y sobre todo costosa, tanto en tiempo como en dinero.

En cuanto a la generalidad del proyecto, la gran y principal ventaja de Turawet es que no está diseñado para ser una solución a un problema concreto de recolección de datos (como puedan ser las encuestas electorales o informes para inspectores de urbanismo). Nuestro proyecto pretende ser una plataforma de recolección genérica, que sirva de base a desarrollos "ad-hoc" muy simplificados y basados en una arquitectura sólida, común, testeada y desarrollada meticulosamente para ser válida en todos los ámbitos en los que requiera aplicarse.

El proyecto Turawet no sólo es una plataforma electrónica y genérica de recolección de datos. También es una solución integral. En primer lugar integra una herramienta web de modelado de formularios (**Seed o Modelador**) que permite a un usuario inexperto desarrollar formularios con un gran conjunto de campos posibles (desde sencillos campos de texto o numéricos hasta coordenadas GPS; pasando por Imágenes, vídeos, *comboboxes*, *radiobuttons* o grabaciones de audio).

El segundo módulo integrado en el proyecto es el recolector, para el que hemos desarrollado una implementación concreta (**BeeDroid o Recolector**). Este módulo es una aplicación para móviles Android, que recibe formularios desde el repositorio y permite al usuario rellenarlos (como hemos dicho anteriormente, permitiendo capturar imágenes, vídeos, audio o coordenadas GPS) para posteriormente enviar las instancias completas al repositorio. Como línea futura de desarrollo se considerará interesante crear un recolector web que nos permita rellenar instancias nuevas o completar una instancia comenzada desde el recolector móvil, desde nuestro equipo sobremesa o portátil.

El tercer módulo integrado en el proyecto Turawet es el gestor del repositorio (**BeeKeeper o Administrador**). Este módulo gestiona la información de formularios e instancias de los mismos. Es el encargado de almacenar los formularios que recibe del modelador, vía servicio web, en la base de datos y, los envía al recolector cuando éste se los solicita. Posteriormente recibe las instancias rellenas y las almacena (desplegadas también en una base de datos a fin de poder explotar la información con posterioridad). Este módulo permite, a un administrador gestor o ejecutivo, visualizar los formularios e instancias almacenadas en el repositorio. Además, una vez que la información de las instancias es guardada, podremos realizar estadísticas, extrayendo información sobre los campos de un formulario en base a las instancias rellenas. También podremos representar en un mapa dónde han sido creadas las instancias (en el caso de "Cuidemos La Laguna", esta utilidad se traducirá como un mapa de los desperfectos registrados).

1.2 Vocabulario

En este apartado se describen algunos conceptos que se utilizarán frecuentemente en este documento y que merecen especial atención, por lo que es necesario establecer unas definiciones para los mismos.

Formulario

Agrupación de campos que como conjunto poseen un significado común y que se rigen bajo una estructura determinada. Serán rellenados durante la recolección de datos correspondiente. Incorpora ciertos metadatos como el autor, la fecha de creación, la versión o el nombre.

Instancia

Conjunto de datos asociados a campos de la definición de un formulario. Existirán instancias incompletas (pendientes de finalizar) o completas.

1.3 Objetivos

Los objetivos principales del proyecto Turawet son:

- Dar una solución integral a la recolección de datos electrónica.
- Ser genérico y capaz de almacenar y explotar cualquier formulario que se desee modelar.
- Permitir a un usuario crear formularios de forma clara y sencilla.
- Permitir a los usuarios rellenar formularios (crear instancias) de forma sencilla desde sus dispositivos móviles y desde un navegador.
- Incluir todos los campos habituales de los formularios en papel.
- Incluir ventajas sobre los formularios en papel, como la captura de imágenes, video, sonido.
- Almacenar en una base de datos todos los formularios e instancias.

- Permitir hacer consultas a dicha base de datos para conocer valores de determinados campos de un formulario en todas las instancias rellenas del mismo.
- Permitir extraer información geolocalizada de las instancias o dibujar dichas instancias en un mapa.
- Permitir elaborar estadísticas de un formulario en función de las instancias rellenas del mismo.
- Ser una plataforma segura.
- Permitir rellenar instancias de formularios de forma *off-line*, es decir, sin necesidad de tener conexión a internet. Posteriormente, estas instancias guardadas en el teléfono se podrán enviar al repositorio al disponer de conexión.

1.4 ¿Por qué Turawet?

Turawet es una palabra de origen Bereber (Amazigh [11]) utilizada por los antiguos habitantes de las Islas Canarias para dar nombre a la miel de abejas.

El modelo que hemos planteado de recolección de datos guarda un gran símil con la vida de las Abejas.

En este símil, el objetivo del proyecto Turawet es producir Miel (datos o información valiosa). Esta información se almacena en la colmena (el repositorio). Hay una abeja reina (Modelador) que diseña la estrategia a seguir para producir la miel. La reina les indica a las abejas obreras (recolectores) que deben de rellenar sus canastas (formularios) con el polen (datos) que se encuentra en las flores. Cuando las abejas obreras (recolectores) rellenan sus canastas (formularios) con polen (datos) vuelven a la colmena (repositorio).

Por último, el polen (datos) almacenado en la colmena se transforma en miel (información) que el apicultor (administrador) recoge, aprovecha y consume (en nuestro caso, realizando consultas, haciendo *data minin*g [12] y tomando decisiones estratégicas en función de la información obtenida).

Además, el logotipo del proyecto ha sido diseñado a tenor de lo antes descrito. Se trata de un hexágono que pretende representar una celda de un panal de abejas y que en su interior posee una *pintadera* canaria, un sello elaborado por los aborígenes de las islas. Se trata, además, de un gráfico vectorial generado con la herramienta libre Inkscape [13].

Capítulo 2. Estado del arte

Resumen:

- Realizaremos un análisis de los diferentes proyectos y aplicaciones que pretenden también dar una solución a la recogida de datos móvil.
- Justificaremos nuestra decisión de desarrollar nuestro propio proyecto desde el inicio.

2.1 Introducción

Antes de comenzar con las tareas propias del proyecto Turawet, nos vimos en la necesidad de investigar cómo se encontraba el estado del arte, en lo que a proyectos o productos similares al que pretendíamos desarrollar se refiere. Esta tarea es realmente necesaria, ya que brinda una visión amplia de lo que se está gestando en otros grupos y comunidades de desarrollo.

Para nuestra sorpresa, nos encontramos con varios equipos de trabajo que habían desarrollado aplicaciones muy similares a la que teníamos en mente; o, al menos, encontramos varios proyectos que iban en la misma línea de trabajo. Además, pudimos conocer en mayor profundidad algunas de ellas, que por momentos fueron candidatas a formar parte de nuestro proyecto.

A lo largo de este capítulo comentaremos los pros y los contras de cada uno de estos proyectos y las razones que nos llevaron a desarrollar el proyecto Turawet, enfatizando más unos aspectos que otros de manera que tengamos una ventaja competitiva frente a otros proyectos del mismo ámbito.

A continuación, hablaremos de los proyectos más cercanos al nuestro en cuanto a funcionalidades se refiere. Algunos de ellos se podrían catalogar como competidores directos de Turawet. Iremos comentando las diferentes experiencias vividas en esta primera etapa de investigación intentando definir lo que es característico de cada uno de los proyectos consultados, analizando aspectos interesantes de cada uno de ellos y viendo qué podemos hacer para que nuestro proyecto ofrezca un valor añadido a los ya existentes.

2.2 Proyectos similares

2.2.1 OpenRosa

Es un consorcio creado en 2007 para fomentar y facilitar la interoperabilidad (creando estándares) entre herramientas móviles de recolección de datos.

Con OpenRosa [17], se han definido estándares para la recolección de datos o la representación de los formularios (basado en la tecnología XForm del W3C [18]). Este consorcio, en el que se hallan integrados varios de los proyectos o aplicaciones que comentaremos en este capítulo, se encarga de velar por estos estándares de interconexión y proponer mejoras sobre ellos.

Como indican en su sitio web, los estándares que promueven buscan permitir a desarrolladores y sus herramientas:

4. Diseñar un formulario para recolectar datos.
5. *Renderizar* dicho formulario en un dispositivo móvil.
6. Almacenar los datos recolectados en un servidor para analizarlos.

Entre los integrantes de OpenRosa se encuentran: Cell-life, Dimagi, EpiSurveyor, GATHERdata, ODK y openXdata [19].

Además de distintas especificaciones de interfaces a seguir a la hora de transmitir datos, gestionarlos o extraer estadísticas de ellos; OpenRosa ha creado el cliente móvil de código libre JavaRosa (comentado en el apartado 2.3.1).

2.2.2 ODK

Open Data Kit es un proyecto realizado en el seno de la Universidad de Washington que pretende realizar las mismas funciones que nuestro proyecto. ODK comenzó como un proyecto patrocinado por *google.org* en abril de 2008 en las oficinas de Google en Seattle.

Es el proyecto más genérico y prometedor de los que hemos encontrado en el ámbito de la recolección de datos por ser, quizás, el más enfocado a *smartphones*. No en vano, está financiado por un premio otorgado por Google y posee detrás una gran comunidad de desarrolladores.

Este proyecto norteamericano, de código abierto, integra una herramienta modeladora bastante intuitiva para modelar formularios (Build) y generar formularios siguiendo el estándar XForm definido por OpenRosa. Además, integra una herramienta de recolección de datos desarrollada para la plataforma Android (Collect); y un sencillo administrador del repositorio que nos permite ver los formularios e instancias disponibles, así como eliminarlos o modificarlos (Aggregate).

También dispone de otras herramientas para, por ejemplo, mover instancias de un repositorio a otro o un módulo que permite transformar el formulario en voz para poder realizar con ella una llamada telefónica y recoger la información según las teclas del teléfono introducidas por el usuario; entre otras.

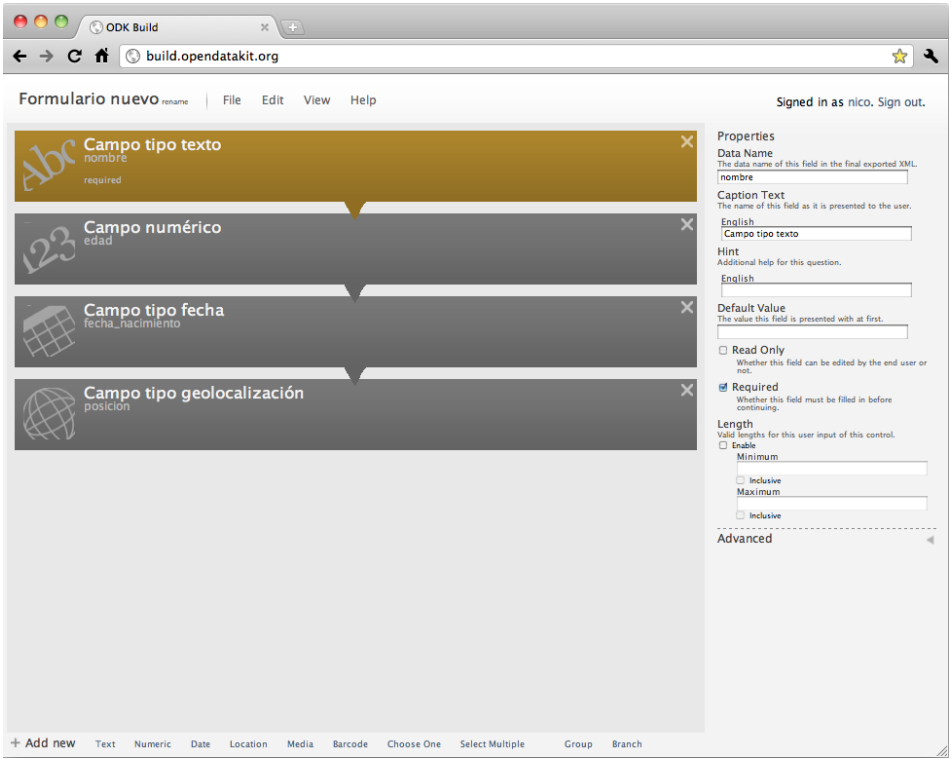


Figura 2-1 Pantalla de la aplicación web de modelado de formularios de ODK.

La Figura 2-1 nos da una idea general de la aplicación de modelado incluida en el proyecto ODK. Es una herramienta bastante completa que permite generar XForms (según el estándar definido por OpenRosa) donde podemos incluir todo tipo de campos (texto, fecha, numérico, geolocalización…). Por supuesto nos permite alterar propiedades de todos estos campos e incluye tratamiento para grupos y listas.

Como características más destacables de lo comentado anteriormente tenemos que:

- Es código libre y posee una gran comunidad de desarrolladores.
- Trabaja sobre dispositivos móviles basados en Android
- Sigue los estándares definidos por el consorcio OpenRosa.
- Se centra en la recogida de datos principalmente, dejando a un lado su explotación.

Cuando comenzamos con el proyecto, ODK no se encontraba, ni mucho menos, en el estado actual y en su momento nos fue más fácil descartarlo.

El hecho de que se tratase de un desarrollo en un estado bastante avanzado y con un numeroso equipo de desarrolladores (de grandes empresas de nuestro sector) suponía un gran problema a la hora de incorporarnos al desarrollo, en el caso de que quisiésemos llevar a cabo nuestra idea tomando como base ODK. A ello hay que sumarle la complejidad de adaptarse a un equipo en el que muchos de sus miembros se dedican a tiempo parcial y que posee un alto

número de contribuyentes y desarrolladores de todas partes del mundo, con unas ideas de diseño ya fijadas y un flujo de trabajo creado.

Por otro lado, nos interesaba seguir otras vías distintas a las marcadas por ODK, queríamos darle mayor importancia a la explotación de los datos y la geolocalización de los mismos así como llevar a cabo otra serie de ideas interesantes que teníamos en mente, y tampoco queríamos atarnos a los estándares fijados por el consorcio OpenRosa aún en desarrollo.

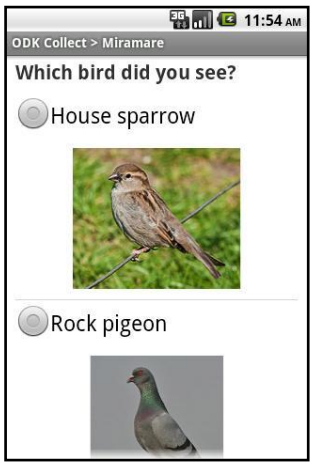


Figura 2-2 Detalle del recolector Android de ODK.

2.2.3 GeoBloc

GeoBloc es un proyecto desarrollado en la Universidad de La Laguna entre los años 2009 y 2010 por David Dinesh A. Harjani y Jorge Carballo.

Surgió a partir de una necesidad específica que desde Gerencia de Urbanismo de La Laguna (Tenerife, España) hicieron llegar a la Universidad. Querían tratar de plasmar en un formato digital y móvil los formularios en papel que tenían que cumplimentar los inspectores de urbanismo, de forma que pudiesen explotar esa información.

GeoBloc, finalmente, acabó estando integrado por un equipo de cuatro desarrolladores, estudiantes de la Escuela Técnica Superior de Ingeniería Informática, que generaron una aplicación recolectora para la plataforma Android así como una aplicación de escritorio basada en Java como asistente para la generación de formularios y un almacén web de instancias y formularios. Como representación de los formularios, utilizaron un XML definido por el propio equipo.

Decidimos no continuar con su desarrollo y prescindir de sus herramientas pues compartíamos ideas distintas en cuanto a cuestiones básicas del diseño, además de que GeoBloc no contaba con ningún módulo de explotación de los datos recolectados ni alguna interfaz que facilitase la misma.

2.3 Productos similares

2.3.1 JavaRosa

En el campo de la recolección de datos en teléfonos móviles, el proyecto pionero fue JavaRosa [14].

JavaRosa es un cliente de XForms [15] escrito en Java Mobile Edition (J2ME [16]). Su desarrollo está orientado a dispositivos móviles de todo tipo, desde teléfonos inteligentes y PDA's con recursos hardware elevados, hasta aparatos de gama más baja como los Nokia 6085 y 2630. Poder utilizar JavaRosa en dispositivos de bajos recursos es una de las más altas prioridades del proyecto, debido a su aplicación en países en vías de desarrollo.

La versión 1.0 Alpha fue lanzado en septiembre de 2010. Dada la amplia gama de dispositivos móviles a los que estaba orientado el proyecto, así como la gran variedad de tecnologías con la que estos diferentes dispositivos cuentan, era de esperar que nos encontráramos con un proyecto de gran complejidad y serias limitaciones de desarrollo. Este proyecto simplifica al máximo la interfaz para poder ser funcional en casi cualquier dispositivo, en detrimento de la usabilidad. Un coste a pagar por un desarrollo para plataformas tan heterogéneas.

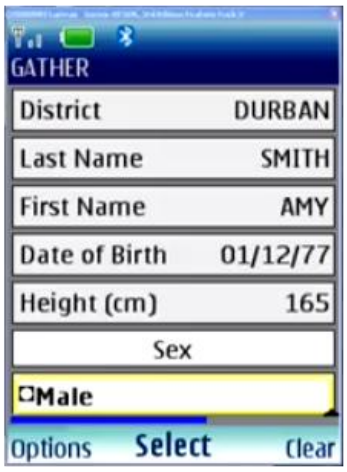


Figura 2-3 Pantalla del cliente JavaRosa

Como principales ventajas, el proyecto JavaRosa tiene la posibilidad de ser utilizado en múltiples dispositivos (aunque enfocado hacia los de gama baja) y se trata de un proyecto libre. El principal inconveniente viene, precisamente, por la intención de potenciar su uso desde el mayor número de dispositivos posibles. Estas restricciones limitan la usabilidad de la aplicación y no permiten utilizar las novedosas funcionalidades que brindan en la actualidad los *smartphones*.

Otro inconveniente del proyecto es el hecho de estar aún en desarrollo y ser desarrollado por una comunidad. Estos dos factores hacen que sea difícil para nosotros aplicarlo a nuestro proyecto de fin de carrera. Tendríamos que utilizar una versión aún inmadura de JavaRosa y cualquier cambio que quisiéramos realizar nos sería muy complejo de llevar a cabo por el desconocimiento de la implementación realizada hasta la fecha y por el

hecho de ser una aplicación casi finalizada donde las decisiones de diseño e implementación ya han sido tomadas, en unas direcciones muy claras, y que no encajan con las que nosotros habíamos pensado tomar.

2.3.2 Cell-Life Capture

Cell-Life [20] es una organización sin ánimo de lucro cuyo objetivo es mejorar las vidas de los afectados por el VIH en Sudáfrica mediante el uso apropiado de las tecnologías móviles.

Además de para el VIH, esta organización provee tecnologías para la gestión de otras enfermedades infecciosas como la tuberculosis.

Cell-Life surgió como un proyecto de investigación de la universidad de Ciudad del Cabo en 2001, convirtiéndose en una compañía independiente en 2005. Inicialmente el principal producto de esta organización fue Aftercare, que podemos traducir como “cuidados posteriores”. Aftercare era un sistema para teléfonos móviles que mantenía en contacto a enfermeras y cuidadores a domicilio para el tratamiento y cuidado de enfermos del VIH.

También han desarrollado otro proyecto, llamado iDART, basado en la idea de realizar una inteligente dispensación de antirretrovirales. Esta herramienta, en 2009 era utilizada ya en 20 clínicas que dispensaban estos medicamentos a más de 45 000 pacientes.

Cell-Life ha continuado trabajando en la recolección de datos vía móvil con Emit [22] y su nueva versión Cell-Life Capture [21]. Cell-Life Capture es una plataforma de pago de recolección de datos que incluye modelador, recolector y un módulo de estadísticas e informes. El recolector está desarrollado en Java y pensado para dispositivos que utilicen esta tecnología (dispositivos de gama baja).

En 2007 se convirtió en miembro fundador del consorcio OpenRosa para la recolección de datos móviles. En ese mismo año, lanzaron el proyecto “teléfonos móviles para el VIH”, explotando aplicaciones de información, comunicación y servicios interactivos para dar soporte al sector del VIH.

Cell-Life cuenta con varios premios, entre ellos dos veces ha estado en el Top 100 de tecnologías (2005 y 2006). También cabe destacar el premio SANGONeT Web [21] por el mejor uso de tecnologías móviles en 2009.

2.3.3 CommCare

CommCare [25] es un producto de Dimagi [24]. Dimagi es una reconocida empresa estadounidense de tecnología con conciencia social que pretende ayudar a las organizaciones a ofrecer atención médica de calidad a las comunidades urbanas y rurales de todo el mundo. Cuenta con un equipo de médicos, ingenieros de software y arquitectos de sistemas de salud que aplican su experiencia y conocimientos, utilizando la tecnología para resolver algunos de los problemas más difíciles del mundo.

CommCare es una solución de código abierto y colaborativo. Es un producto orientado a las visitas médicas. Basa en JavaRosa su recolector orientado a teléfonos móviles básicos y utiliza ODK para sus dispositivos Android. Pretenden desplegar su tecnología a gran escala y trabajan en la construcción de soluciones que pueden aprovechar el rápido crecimiento de las

tecnologías móviles en todo el mundo. Por otra parte, integra una solución basada en Django en la parte de servidor, donde integra una herramienta de modelado y un sistema que envía por e-mail los resultados recolectados. Es importante tener en cuenta el ámbito concreto para el que está desarrollado este sistema, que no es otro que la atención médica.

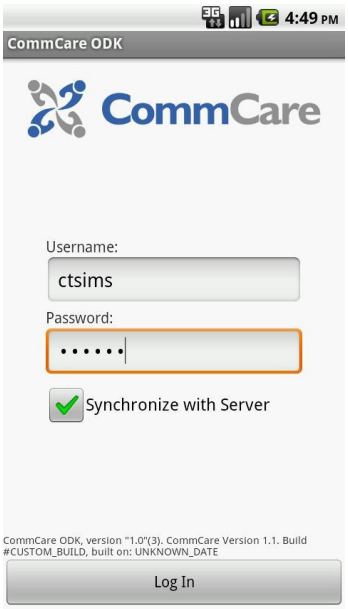


Figura 2-4 CommCare en Android, aplicación basada en ODK.

Dimage también es miembro de OpenRosa y defiende esta idea de colaboración abierta para crear herramientas, basadas en estándares, de código abierto, que incluyan recolección de datos o almacenamiento, así como análisis de los datos recolectados y generación de informes. A través de una de sus iniciativas han fomentado localmente el entrenamiento de programadores en países en desarrollo y les han puesto a trabajar en proyectos que impactan directamente en sus países de origen.

Esta compañía ha desarrollado soluciones personalizadas para gobiernos, organizaciones no lucrativas y multinacionales. Además, ha trabajado en más de 13 países diferentes (desde las aldeas de la África subsahariana hasta Estados Unidos).

Al igual que Cell-Life, pretenden encontrar el eslabón perdido en la prestación de asistencia sanitaria (con especial énfasis en herramientas de bajo costo para comunidades marginadas). El espectro de aplicación de sus herramientas va desde la educación pública hasta la atención crónica. Siempre con un objetivo: mejorar la salud de todos.

2.3.4 EpiSurveyor

Esta herramienta, se define a sí misma como “la forma más rápida, fácil y menos cara de recolectar datos en teléfonos móviles”.

EpiSurveyor [25] permite a cualquiera crearse una cuenta, diseñar formularios y descargarlos en el teléfono para comenzar a recolectar datos, en minutos y de manera gratuita.

Ganadora de varios premios como el “Wall Street Journal Award for Technology Innovation in Healthcare" [27].

Es utilizada, según ellos mismos cuentan, por miles de grupos alrededor del mundo para recolectar datos en salud, economía, agricultura y muchos otros temas. Entre sus usuarios destacados se encuentra el banco mundial, que lo utilizó en 2011 para recolectar información sobre la actividad anti-corrupción en Guatemala. Según comentan, la utilización de esta herramienta permitió reducir los costes mientras facilitaba el control de calidad y mejoraba la velocidad de implementación. También lo utiliza, entre otros, una organización sin ánimo de lucro llamada Aquaya [28]. Ésta organización vela por mejorar la salud de las personas dándoles acceso a fuentes de agua limpia. Utilizan EpiSurveyor para muchos de sus proyectos, como por ejemplo, en Vietnam, donde más de 50 operarios de sistemas de control de aguas envían datos diarios a los gestores.

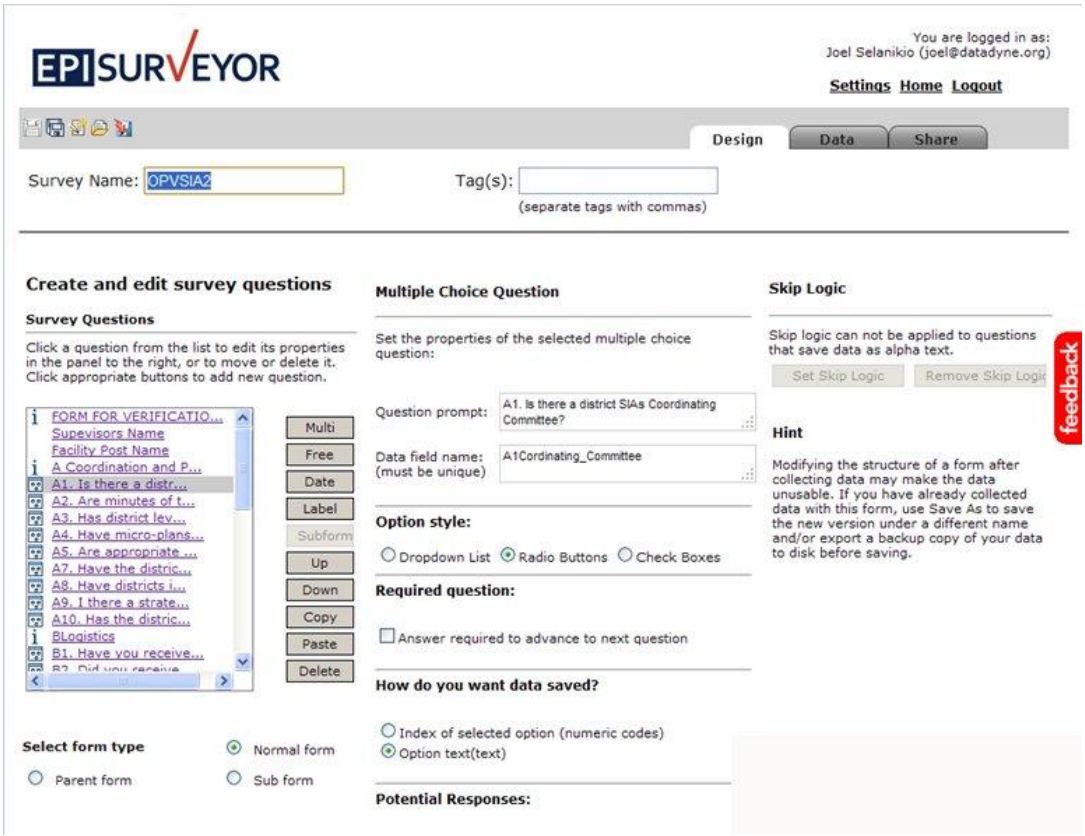


Figura 2-5 Aplicación de generación de formularios EpiSurveyor.

Como vemos a tenor de la captura de la anterior, se centra más en realizar encuestas sencillas sin posibilidad de añadir información multimedia y con una interfaz compleja. Para los dispositivos móviles con J2ME cuenta con un cliente basado en JavaRosa. También cuenta con un cliente para Android, en fase beta, basado en ODK [29] y que no hemos podido probar puesto que el enlace de descarga no estaba disponible.

Figura 2-6 Aplicación móvil de EpiSurveyor.

Dispone de versiones de pago sin muchas de las restricciones de la versión gratuita y con estadísticas adaptadas a ejemplos concretos. El código fuente no es abierto si bien cuenta con unas APIs escasamente documentadas. Por esto último, descartamos utilizarlo como base de nuestro proyecto, dado que no podremos desarrollar sobre su código fuente para añadir las posibilidades multimedia y de geolocalización con las que contará Turawet, así como un modelador más intuitivo y usable.

2.3.5 GATHERdata

GATHERdata [30] es la plataforma propuesta por AED [31] (Academy for Educational Development) para la recolección de datos. AED, con sede en Washington, es una organización sin fines de lucro que trabaja para mejorar la educación, la salud, la sociedad civil y el desarrollo económico a nivel mundial. Implementa programas que atienden apersonasen los50 estados de EE.UU. y en más de 150 países.

La plataforma GATHERdata pretende, al igual que nuestro proyecto, abordar la recolección de datos desde dispositivos móviles en tiempo real y, con ellos, realizar análisis e informes. Incluye herramientas para modelar formularios, para mostrarlos en teléfonos móviles, para rellenarlos desde el propio teléfono así como herramientas para realizar estadísticas con la información almacenada en la base de datos central.

El código fuente fue liberado al público en su versión 1.0 en diciembre de 2009. Desde entonces, ha habido muy poca actividad en su desarrollo a tenor de los cambios que se pueden visualizar en el repositorio alojado en GitHub [32].

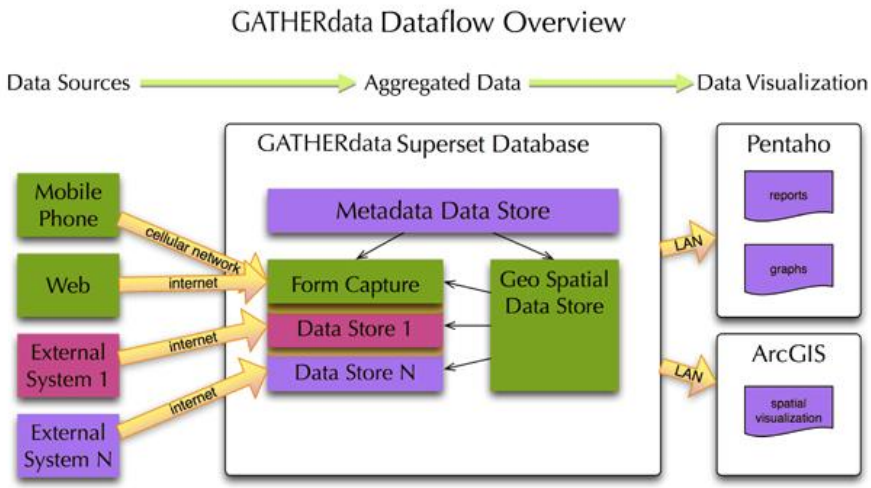
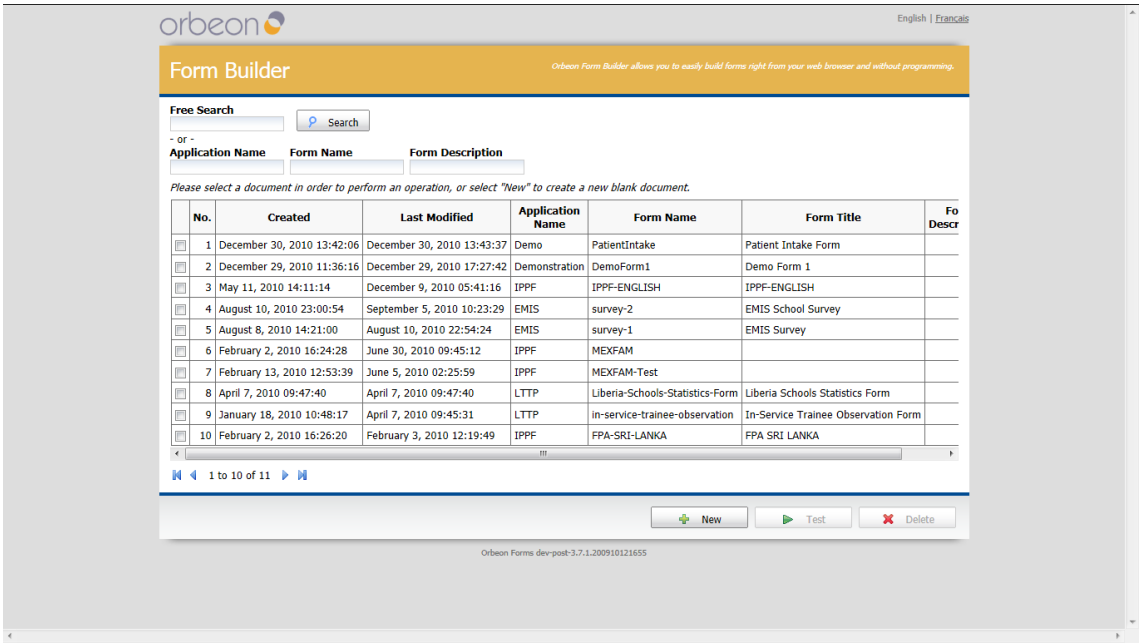


Figura 2-7 Flujo de trabajo de GATHERdata.

Se facilita poca información sobre la plataforma y no nos fue posible acceder al servidor donde se debería alojar la página principal del proyecto pues permaneció inaccesible durante los días que lo intentamos. A través de buscadores, conseguimos localizar un enlace a una demostración online (aunque algunos módulos no eran funcionales) de la cual mostramos capturas a continuación:



	No.	Created	Last Modified	Application Name	Form Name	Form Title	Fo Descr
	1	December 30, 2010 13:42:06	December 30, 2010 13:43:37	Demo	PatientIntake	Patient Intake Form	
	2	December 29, 2010 11:36:16	December 29, 2010 17:27:42	Demonstration	DemoForm1	Demo Form 1	
	3	May 11, 2010 14:11:14	December 9, 2010 05:41:16	IPPF	IPPF-ENGLISH	IPPF-ENGLISH	
	4	August 10, 2010 23:00:54	September 5, 2010 10:23:29	EMIS	survey-2	EMIS School Survey	
	5	August 8, 2010 14:21:00	August 10, 2010 22:54:24	EMIS	survey-1	EMIS Survey	
	6	February 2, 2010 16:24:28	June 30, 2010 09:45:12	IPPF	MEXFAM		
	7	February 13, 2010 12:53:39	June 5, 2010 02:25:59	IPPF	MEXFAM-Test		
	8	April 7, 2010 09:47:40	April 7, 2010 09:47:40	LTTP	Liberia-Schools-Statistics-Form	Liberia Schools Statistics Form	
	9	January 18, 2010 10:48:17	April 7, 2010 09:45:31	LTTP	in-service-trainee-observation	In-Service Trainee Observation Form	
	10	February 2, 2010 16:26:20	February 3, 2010 12:19:49	IPPF	FPA-SRI-LANKA	FPA SRI LANKA	

Figura 2-8 Pantalla con listado de formularios de “orbeon”, parte del proyecto GATHERdata.

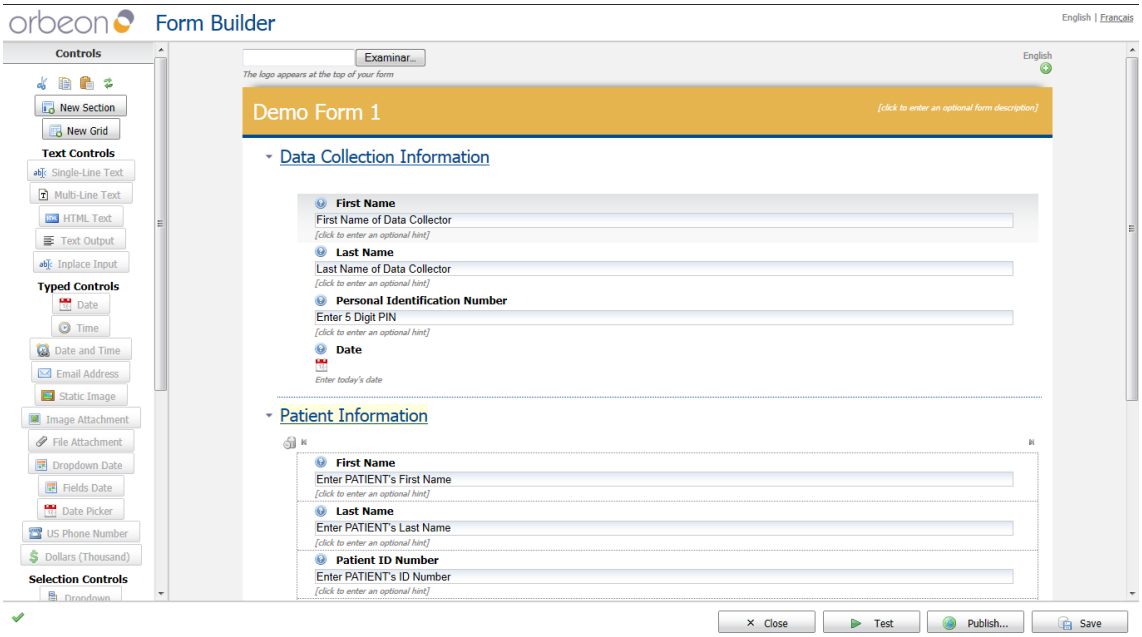


Figura 2-9 Modelado de formularios “orbeon”, del proyecto GATHERdata.

Si bien el planteamiento de esta aplicación podría parecer similar al nuestro, AED ha centrado su esfuerzo en hacerla compatible con dispositivos móviles con escasos recursos de forma que pueda recolectar datos en los países del tercer mundo en donde desarrolla su labor.

En nuestro caso, Turawet sigue otra vía y pretende ser una alternativa que busca la facilidad y comodidad de los *smartphones* que actualmente inundan el mercado, frente a la necesidad de AED de dar respuesta en lugares remotos del planeta donde estas tecnologías no son asequibles.

Figura 2-10 Cliente JavaRosa con formulario de GATHERdata.

Durante el desarrollo de esta plataforma, AED ha trabajado junto a Dimagi y el Open Rosa Consortium, siguiendo el estándar XForms y utilizando JavaRosa como cliente J2ME para dispositivos móviles.

2.3.6 OpenXdata

Varias organizaciones de diferentes partes del mundo se reunieron en Junio de 2010 desde Noruega, para formar una asociación independiente encargada del desarrollo, mejora, mantenimiento y soporte de la plataforma OpenXdata [33].

Como la mayoría de las otras aplicaciones comentadas, OpenXdata permite recolectar información en dispositivos con Java, haciendo uso en aquellos dispositivos que lo permiten de contenido multimedia y GPS.

Por medio del "openXdata server", dispondremos de una aplicación modeladora de formularios, administración de usuarios y permisos así como podremos exportar la información recolectada a distintos formatos (como, por ejemplo, para fijarla en una base de datos relacional).

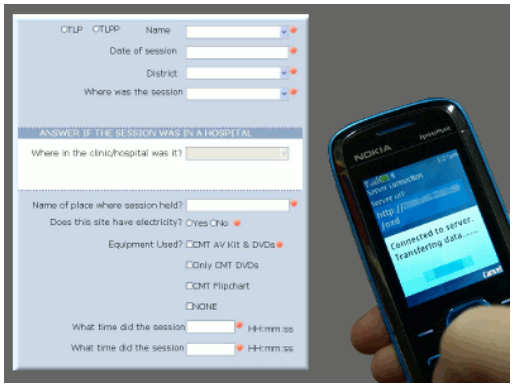


Figura 2-11 Recolectores web y móvil de OpenXdata.

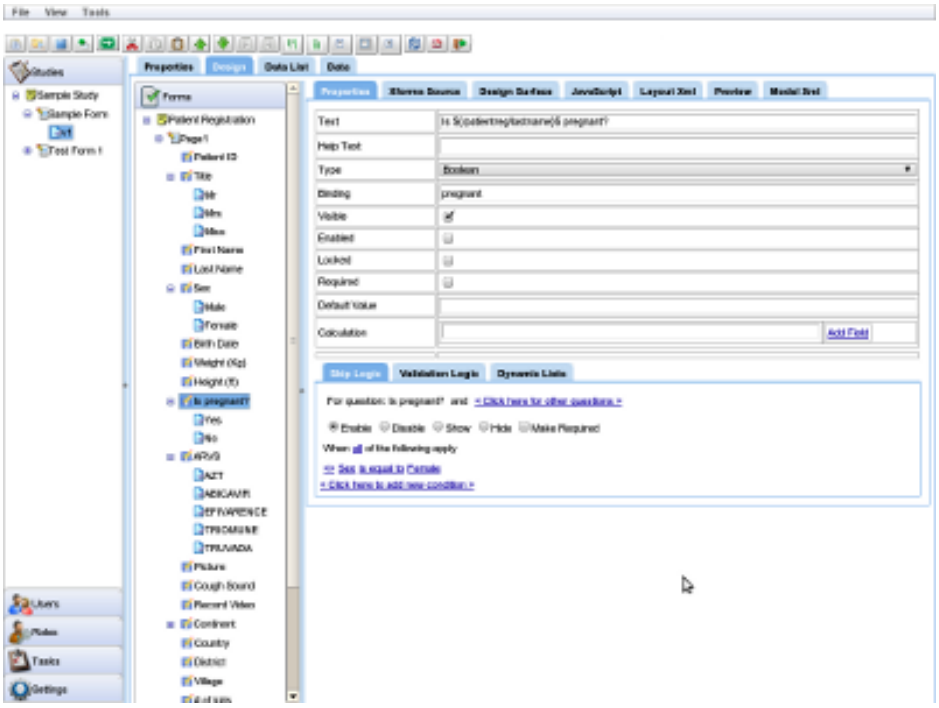


Figura 2-12 Pantalla del modelador de formularios de OpenXdata.

Dispone, además del recolector móvil, de una aplicación web basada en Java para poder rellenar formularios.

2.3.7 Comparativa

A continuación vemos una tabla donde se comparan las funcionalidades de los diferentes proyectos y/o productos que se han estudiado anteriormente.

	Recolector						Modelador			Administrador			
	Plataforma	Contenido multimedia	GPS	¿Es usable?	Continuar instancias	Recolector web	Plataforma	HTML5	¿Es usable?	Plataforma	Estadísticas	Informes	Geolocalización
JavaRosa		✗	?	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Capture		?	?	✓	✓	✓		✗	✓		✓	✓	?
CommCare		✓	✓	✗	?	✗	?	?	?	?	?	?	?
EpiSurveyor		✓	✓	✓	?	✗		✗	✓		✓	✓	✗
GATHERdata	?	?	✓	?	?	✓	?	?	?	?	✓	✓	✓
OpenXdata		✓	✓	✓	?	?		✗	✓		✓	✓	?
ODK		✓	✓	✓	✓	✗		✓	✓		✗	✗	✗
GeoBloc		✓	✓	✓	✓	✗		✗	?	✗	✗	✗	✗
Turawet		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓

Leyendas

- ✓ - Funcionalidad implementada
- ✗ - Funcionalidad no implementada
- ? - No encontramos suficiente información
- - Funciona sobre J2ME
- - Funciona sobre Android
- - Funciona sobre Windows Mobile
- - Plataforma Web
- - Plataforma Desktop

Figura 2-13 Tabla comparativa del estado del arte.

2.4 El espacio de Turawet

En Julio de 2010 nos llega la idea de un recolector de datos móvil como proyecto de fin de carrera por parte de nuestro director de proyecto, José Luis Roda García. Se proponen como posibles casos de uso reales, una aplicación para los inspectores de urbanismo, en la cual puedan completar diversos formularios referentes a viviendas u obras, y otra para el cuerpo de policía, con la que se facilite la cumplimentación de multas.

Tras hacer un estudio del estado del arte, haciendo especial hincapié en ODK, quizás por ser el proyecto más completo, concluimos que realizar nuestro propio proyecto de recolección de datos era una idea viable por muchos motivos. En aquel entonces (y aún a día de hoy, julio de 2011), el proyecto ODK estaba en desarrollo. Involucrarse en este proyecto se antojaba complicado debido a su poca madurez, la lejanía con el núcleo de desarrollo y por tener un grupo de trabajo ya formado; máxime cuando nuestro objetivo era hacer un proyecto de fin de carrera en un plazo determinado y con resultados tangibles y que fueran inequívocamente propios. Por otra parte, intentar adaptarlo a nuestras necesidades no era una opción viable, por la magnitud del proyecto, el hecho de estar aún en desarrollo, el desconocimiento de toda la implementación realizada hasta la fecha y de algunas de las tecnologías utilizadas. También se observaron dificultades para la adaptación de ODK a nuestros fines debido a que las decisiones de diseño ya habían sido tomadas en gran medida y tendríamos que adaptarnos a ellas. Un ejemplo de esta rigidez se encontraba en los tipos de campos disponibles para los formularios o en las tecnologías utilizadas. A todo ello se sumó la falta de documentación sobre el proyecto y el hecho de que no cuentan con un recolector web (que nosotros considerábamos oportuno desarrollar).

Turawet, al ser desarrollado en el marco de un proyecto de fin de carrera tenía por objetivos ser un proyecto interesante, novedoso y motivador. Un proyecto que nos permitiera formarnos en nuevas tecnologías y asentar nuestros conocimientos en materia de bases de datos o desarrollo web; añadiendo también la experiencia de diseñar un proyecto relativamente grande y complejo; estructurando, integrando y tomando decisiones consensuadas de diseño, además de tener especial cuidado con la documentación.

Por todo ello, Turawet se presenta como una idea interesante con el potencial de explotar la recolección de datos en nuestro entorno, sin competidores a nivel local y compitiendo a nivel global con herramientas desarrolladas en otras partes del mundo.

Parte II. Turawet: El proyecto

Capítulo 3. Descripción general

Resumen:

- Presentamos Turawet como sistema integral de recolección de datos.
- Detallamos el análisis y la captura de requisitos llevadas a cabo.
- También se describen las decisiones de diseño tomadas a nivel general.

3.1 Introducción

En este capítulo hablaremos de forma global de aquellos puntos más importantes en el desarrollo común del proyecto, hablando de Turawet como solución global a la recolección de datos de cualquier tipo.

Con Turawet como aplicación, queremos ser capaces de aportar al usuario una vía rápida y simple con la que, en pocos minutos, sea capaz de recolectar la información que necesite siguiendo la estructura que precise, con los campos que requiera y sin necesidad de conocimientos técnicos, como conocer un lenguaje específico o dominar una plataforma concreta.

Queremos aunar simplicidad con facilidad de uso sin dejar a un lado la funcionalidad. Nuestro proyecto será una solución integral para el modelado de formularios, la recolección de datos, el almacenamiento de información y el tratamiento y explotación de esos datos.

3.2 Etapa de análisis

Comenzando con el ciclo habitual de desarrollo de software, el primer paso consistirá en extraer los requisitos o requerimientos que necesita cumplir el proyecto para transformarse en una aplicación real. Para ello, fomentaremos reuniones con clientes potenciales y nos pondremos en su piel para poder extraer todos los requisitos que creemos que debe cumplir nuestra herramienta.

Ponerse en la piel de un usuario de nuestra aplicación no fue una tarea sencilla, a pesar de que, cuando aún se gestaba la idea del proyecto, mantuvimos algunas reuniones con personas ajenas al mismo, quienes tenían en su momento el rol de clientes potenciales, y que de hecho, terminarían siendo uno de los casos de implantación de Turawet (Gerencia de Urbanismo de La Laguna). De la entrevista apenas pudimos capturar, a grandes rasgos, algunos de los requerimientos más básicos, pues el resto de lo que aquí se expondrá fue fruto de una tarea de análisis compleja y que nos depararía semanas hasta fijar unos requisitos mínimos. Es necesario mencionar que en todo el ciclo de desarrollo del proyecto fuimos descubriendo nuevos requisitos que se incorporaron en la medida de lo posible al proyecto.

La idea inicial era, en pocas y simples palabras: “lograr, de alguna forma, generar un formulario con una serie de campos que aparezcan en un dispositivo móvil y que dicho móvil pudiese rellenar y enviar esos datos hacia un lugar donde almacenarlos”. De esa frase se deducen dos cosas. La primera, que nuestro objetivo es tratar de almacenar electrónicamente

datos que han sido recogidos desde un dispositivo móvil, y la segunda, que deberemos facilitar al usuario una interfaz amigable donde sea capaz de modelar un formulario.

Sobra decir que, para explotar los datos almacenados, no hemos detectado aún ninguna observación con los requisitos establecidos en el párrafo anterior, pero naturalmente aparecerá la necesidad de incluir un gestor que muestre la información recolectada así como permita su gestión y añada ciertas funcionalidades de explotación de dichos datos almacenados.

3.2.1 Requisitos generales

A tenor de lo comentado anteriormente y como requisito inicial para dar solución al problema de recolección de datos que se nos planteaba, se hizo necesario dividir el proyecto en tres módulos claramente diferenciados.

El primero de ellos será la aplicación para la creación de formularios. Facilitará la labor de modelar un formulario acorde a las múltiples necesidades que se pueden encontrar en ámbitos muy diversos. Por ello debe ser muy genérico pero a la par intuitivo. No se especifica, en esta etapa una forma clara y definida de plantear una herramienta de modelado, pues las restricciones son bastante generales.

Un segundo módulo se encargará de la recolección de datos. Hará uso de los formularios obtenidos de la primera de las herramientas y permitirá rellenarlos y enviar los datos recogidos hacia el módulo de almacenamiento.

Por último, existirá un tercer módulo que será el encargado de almacenar los formularios generados, así como permitir su gestión, facilitar servicios para su descarga, almacenar instancias rellenas y dar la posibilidad de visualizar los datos. Asimismo, se requiere poder explotar los datos almacenados. También, dado que la recolección, como ya se ha comentado, estará principalmente enfocada a dispositivos móviles, será muy importante explotar la localización georreferenciada de la misma.

Dado que la tarea de modelar un formulario no es necesario que se realice en dispositivos móviles y debido a que se nos pide una interfaz genérica, accesible desde cualquier lugar, usable y unificada, se ha decidido de forma unánime realizar una aplicación web. Con ello lograremos que una misma herramienta cumpla la función para la que ha sido diseñada sin tener que realizar una “portabilidad” entre las distintas plataformas existentes.

Ocurre más de lo mismo con el módulo de almacenamiento y administración. Para que los datos puedan ser consultados desde cualquier lugar y desde cualquier sistema tan sólo haciendo uso de conexión a Internet, es requisito fundamental que se trate de una aplicación web.

Pero a la hora de recolectar surge una gran pregunta: ¿qué hacer si en el lugar en el que estamos no se dispone de conexión a Internet? Cuando hablábamos de las otras dos herramientas que componen Turawet, no requerimos una herramienta disponible de manera *offline*. En este caso, sí que necesitamos que el usuario pueda cumplimentar un formulario en cualquier momento y sin depender del estado de las redes o de su tarifa de datos. Lo que le interesa al usuario es la recolección de datos, almacenar esos datos en el teléfono y, en el momento que disponga de conexión, enviarlos hasta un lugar seguro, un almacén.

Aunque a día de hoy cada vez son más los dispositivos conectados a la red de redes, existen muchas causas por las cuales se puede carecer de conexión y que se han de tener en cuenta en el ámbito de nuestro proyecto, desde encontrarse en un edificio sin cobertura hasta hallarse en un lugar apartado de núcleos urbanos en donde dicha cobertura no exista. Además un usuario puede no tener contratada una tarifa de datos. Por todo ello, no podemos limitar las zonas donde se puede realizar la recogida y recolección de datos.

Para poder contar con las capacidades hardware del dispositivo móvil, así como explotar las funcionalidades propias de su sistema operativo y lograr almacenar en él, de forma persistente una instancia completa o incluso una definición de un formulario, necesitaremos integrarnos con la plataforma y dejar a un lado las limitaciones del navegador web. Por ello hemos optado por generar una aplicación adaptada a uno de los principales sistemas operativos móviles del mercado actual y que detallaremos más adelante.

Pensando además en que, a veces puede no resultar cómodo introducir campos que requieran más de unas pocas palabras desde un dispositivo móvil (y por tanto con limitaciones físicas de tamaño), hemos decidido añadir la posibilidad de completar en un momento posterior, instancias a través de una aplicación web, aun habiendo sido iniciadas desde el dispositivo.

Por último, y como requisito fundamental, todos los módulos comentados del proyecto deben estar comunicados y ser totalmente interoperables.

3.2.2 Modelo de dominio

Como parte de la etapa de análisis, se presenta un diagrama que sigue la notación especificada en UML [35] y que representará el modelo de dominio de nuestro proyecto. Con ello pretendemos representar mejor los requisitos reconocidos durante la etapa de análisis.

Un diagrama de modelo de dominio es una forma de representar tanto conceptos físicos como no físicos, que estén relacionados con el ámbito global y real del proyecto. Nos ayudará a comprender mejor los conceptos con los que trabajaremos tanto nosotros como nuestra aplicación y que, además, utilizaremos durante la redacción de los siguientes capítulos de este documento.

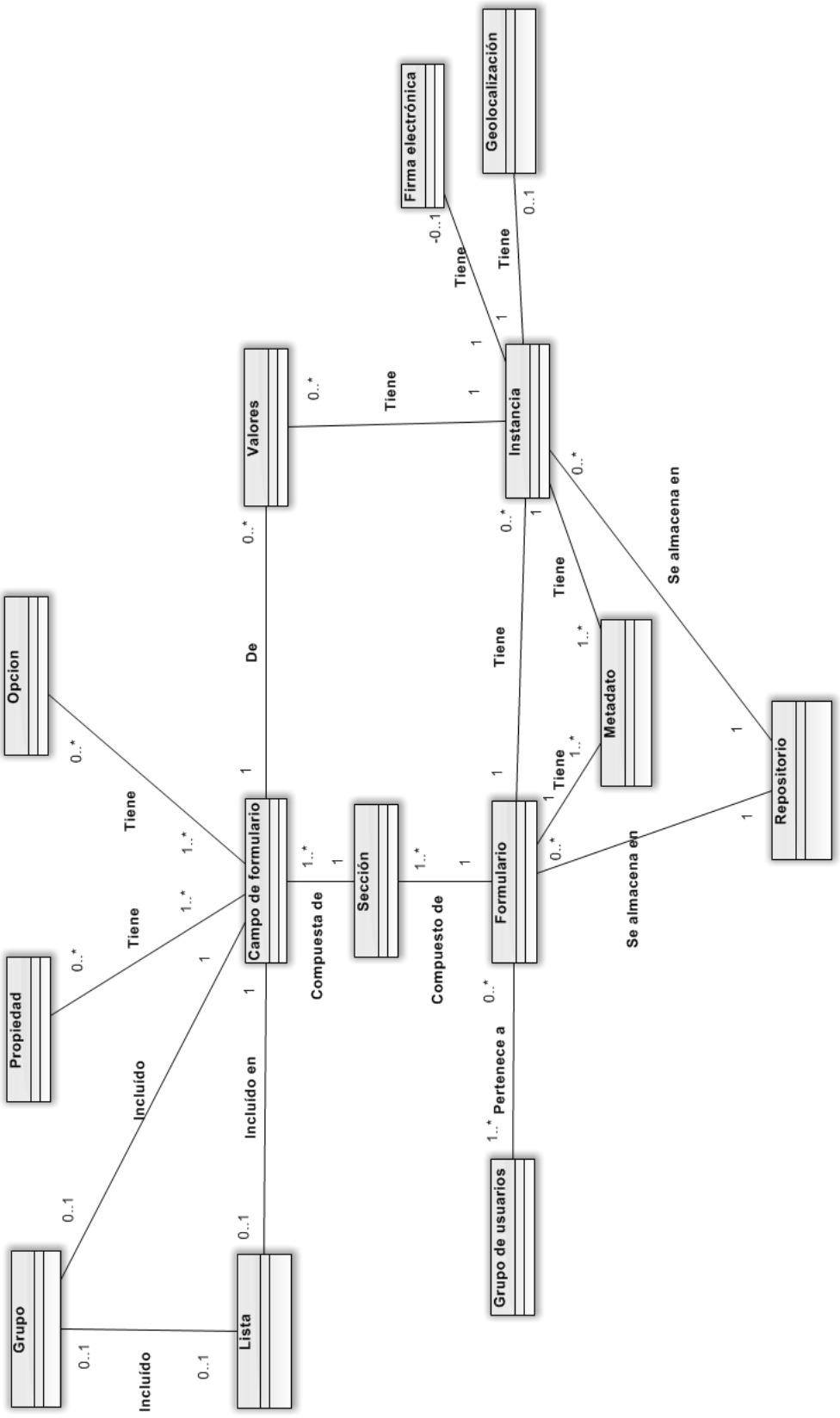


Figura 3-1 Modelo de dominio

La Figura 3-1 representa el modelo de dominio de nuestro proyecto. Es fácil ver que, a primera vista, los conceptos más relacionados con el resto son: Formulario, Campo de formulario e Instancia. Y es que son estos los conceptos principales entorno a los que girará el resto.

Dado que estos tres conceptos, al igual que otros, han sido tratados y explicados en otros apartados como vocabulario general del proyecto y dado que representan conceptos muy sencillos y, la mayoría, auto explicativos, pasaremos a comentar las diferentes relaciones que entre ellos existen.

Partiendo del concepto más básico, Formulario, podemos decir que un formulario, como concepto, está compuesto de al menos una, y por lo general más, secciones. Pertenece a uno o más grupo de usuarios y de un formulario se generan instancias, que representarán formularios con datos ya recogidos. Los formularios se almacenan en un único repositorio y poseen una serie de metadatos que representarán propiedades específicas del mismo.

A su vez, la sección estará compuesta de uno o más campos de formularios, generando con ello una definición completa de formulario. Cada campo podrá tener propiedades y, de ser el caso, una serie de valores posibles a tomar u opciones. A esto añadimos un concepto algo más abstracto como puede ser un “grupo” que no es más que un conjunto de campos de formulario con una semántica común y que pueden permitir múltiples valores, es decir, a la hora de rellenar la instancia puede haber un número variable de datos rellenos para un mismo grupo.

En una instancia de un formulario aparecen, fundamentalmente, valores, o lo que es lo mismo, los datos recogidos durante la recolección. Dichos valores estarán asociados a un campo de una definición de un formulario.

Una instancia se almacenará en un único repositorio, el mismo en el que aparece la definición del formulario y, adicionalmente, una instancia puede poseer una serie de metadatos, comunes al formulario en cuanto a tipos de metadatos pero no así en cuanto a valor. Otras propiedades añadidas y que no aparecen relacionadas con el formulario son: la geolocalización o posición geográfica donde fue recolectada la información, así como una firma digital que rija la validez de los datos recolectados.

3.2.3 Diagrama de casos de uso

A continuación se muestra el diagrama de casos de uso del escenario principal. Cada uno de los actores tiene un cometido en el sistema. Así, un modelador tiene como objetivo modelar formularios nuevos (lo que implicará enviarlos al repositorio una vez creados). Por su parte, un recolector se descargará los formularios disponibles del repositorio y generará instancias (que enviará al repositorio para almacenarlas). Por último, el usuario Administrador (que también puede ser gestor o ejecutivo) será el encargado de gestionar los formularios e instancias del repositorio, así como de acceder al cuadro de mandos y obtener información valiosa de los datos almacenados (estadísticas de un formulario o representación geolocalizada de instancias).

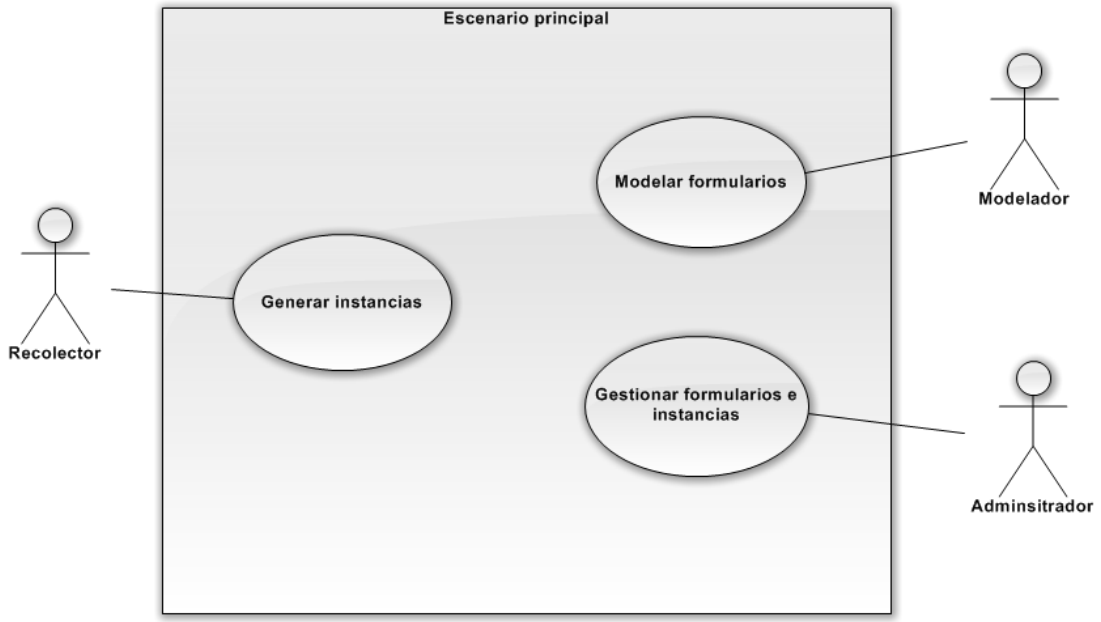


Figura 3-2 Diagrama de casos de uso del escenario principal.

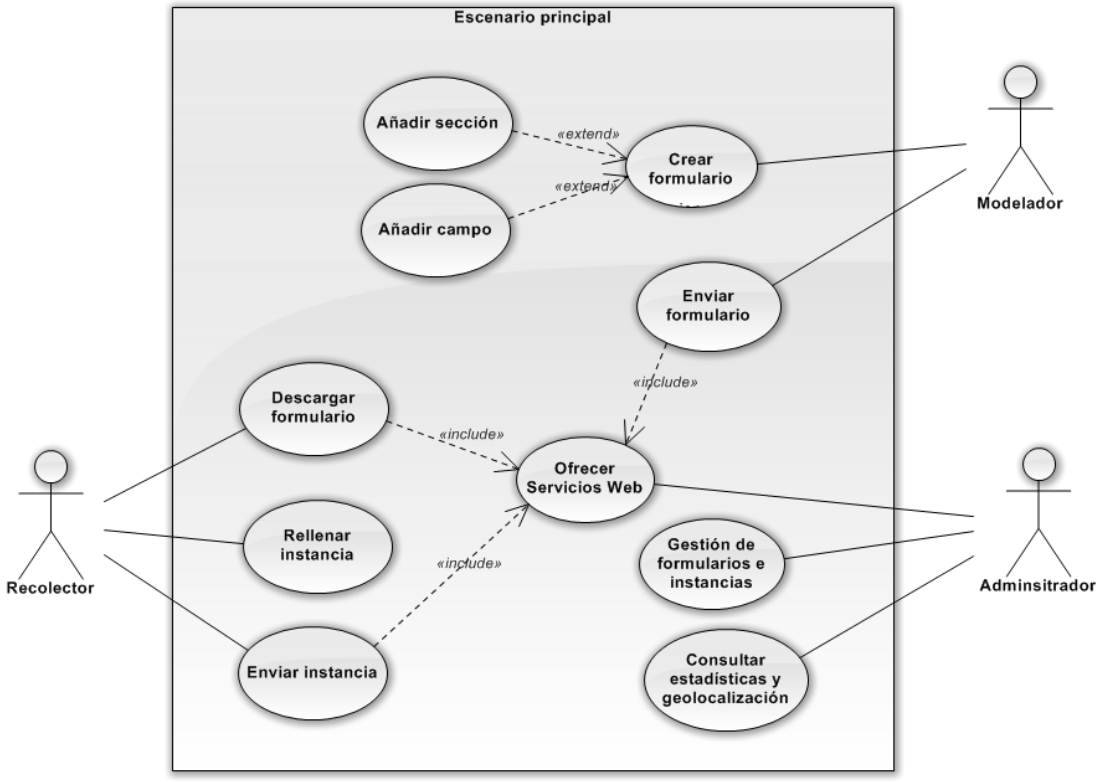


Figura 3-3 Diagrama de casos de uso del escenario principal detallado.

En la Figura 3-3 se representa, de nuevo, el escenario principal con unos casos de uso algo más detallados. En él se refleja lo que comentábamos someramente en el párrafo anterior, detallando un poco más las relaciones entre los casos de uso de los diferentes actores. El

modelador podrá crear un formulario (añadiendo secciones y campos al mismo) y una vez creado lo podrá enviar al administrador, por medio de los servicios web que éste último le ofrece. También, como ya hemos comentado, el recolector podrá descargar los formularios gracias a las interfaces de acceso que nos proporciona el administrador, rellenarlas y enviarlas de vuelta a éste. Por último, el actor administrador, además de ofrecer servicios web, puede gestionar los formularios e instancias almacenados en el repositorio, así como realizar consultas sobre los datos recibidos de las instancias.

En los sucesivos capítulos se detallarán cada uno de los módulos del proyecto Turawet y se realizará un diagrama de casos de uso específico para cada caso, entrando en mayor detalle que en el mostrado anteriormente a nivel general.

3.2.4 Diagrama de secuencia

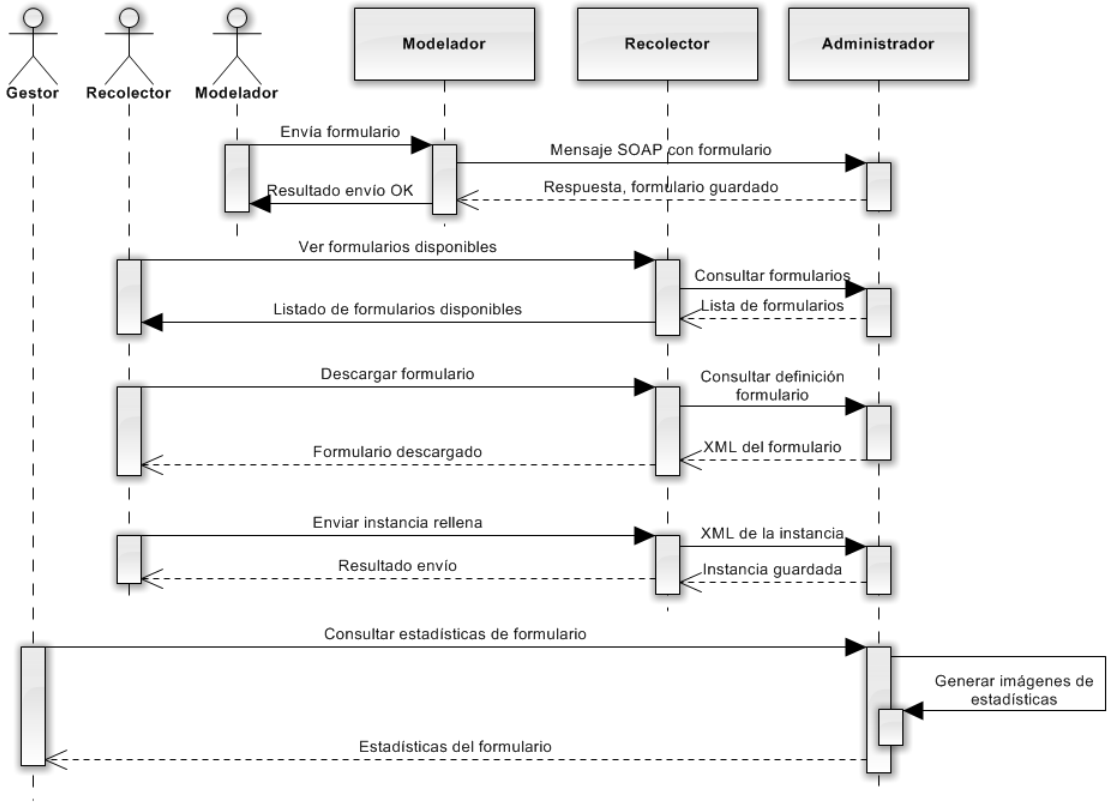


Figura 3-4 Diagrama de secuencia general.

En la Figura 3-4 se representa un flujo de trabajo completo, haciendo partícipes a los tres módulos del proyecto. Como primer paso, el actor modelador enviará un formulario hasta el módulo Modelador y éste lo transmitirá al Administrador de forma que persista. Una vez realizados estos pasos, se le mostrará al usuario un mensaje confirmándole que el formulario se ha guardado correctamente. Poco después, el usuario recolector tratará de ver los formularios disponibles desde el Recolector, haciendo que éste contacte con el Administrador y finalmente despliegue la lista de formularios al usuario (entre los que se encontrará el que

acabamos de modelar).El usuario recolector decide descargarse el formulario y rellenar, con él, una instancia que envía hasta el repositorio del Administrador. Por último, el gestor accede a la administración para visualizar las estadísticas de los datos recogidos por el recolector, estadísticas que se generarán en el módulo Administrador.

3.3 Etapa de diseño

Tras la definición de los requisitos de la aplicación, surgen las primeras cuestiones relacionadas con el diseño de la misma. A continuación se describen las decisiones de diseño que afectan a todo el ámbito del proyecto.

3.3.1 Decisiones de diseño

Las principales decisiones de diseño relacionadas con el proyecto de forma global son:

- Utilización del *framework* Django de Python como tecnología para el servidor web.
- El módulo de almacenamiento y gestión, así como el de modelado de formularios, serán aplicaciones web para mejorar su accesibilidad desde cualquier sistema operativo o dispositivo. Además harán uso de Django como *framework* de desarrollo.
- Para el módulo de recolección dispondremos principalmente de una aplicación Android. Adicionalmente se presentará una alternativa web.
- Se utilizarán servicios web para el paso de mensajes, de forma que se logre la interoperabilidad entre los diferentes módulos del proyecto. El servidor de los servicios web se integrará en la aplicación de Administración (módulo Administrador).
- Utilizaremos XML para representar instancias y formularios.

Tecnología para el servidor web

A día de hoy, para desarrollar una aplicación web, existen múltiples opciones tecnológicas a emplear en el lado del servidor. Además, para cada una de las opciones disponibles existen *frameworks* que facilitan el trabajo en gran medida y nos permiten desarrollar aplicaciones web orientadas a seguir el paradigma modelo-vista-controlador.

Antes de decidir por qué tecnología decantarnos para el servidor web, decidimos informarnos, consultando numerosos artículos, de los cuales surgió uno [36] especialmente explicativo en defensa de Django, el *framework* de Python [37].

En el citado artículo comparativo se tratan Ruby on Rails [38] y Django [34]. Como puntos a favor de Django/Python que nos llevaron a decantarnos por ellos se encuentran:

- El mayor uso de Python como lenguaje de programación frente a Ruby [39].
- El aumento de popularidad que está teniendo Django frente a Ruby on Rails.
- A pesar de que ambos *frameworks* siguen el paradigma modelo-vista-controlador, Django guarda una mayor claridad en su forma de implementarlo.

- La instalación de Django es más flexible.
- Django además permite un mejor tratamiento de las URLs que Ruby on Rails.
- La administración y gestión de usuarios y grupos de Django es más completa que la de Ruby on Rails, necesitando este último *framework* algunos *plugins* para llegar al nivel de Django en este aspecto.
- Es cierto que para trabajar con AJAX [40] es más cómodo utilizar Ruby on Rails, pero nosotros, en principio, no teníamos pensado utilizar esta tecnología. De hecho, en la versión final de nuestro prototipo no utilizamos AJAX.
- La documentación de Ruby también parece ser más abundante que la de Django, aunque cada vez esta comparativa está más igualada debido al auge de Django.
- Por último Ruby también gana en la facilidad para la instalación de *plugins*, aunque los de Django tampoco son especialmente difíciles de instalar.
- Por último comentar que la integración de los test en Ruby es más sencilla, así como el despliegue de la aplicación. En cualquier caso, teníamos experiencia previa con Django en este punto y no lo considerábamos complicado.

En definitiva, una vez vista la comparativa, observando que la tendencia general se inclina por el uso de Django (en detrimento de Ruby on Rails) y sabiendo que contábamos con un mayor conocimiento de partida de Python/Django (aunque muy bajo), nos decantamos por esta última opción.

Django

Como características principales de Django hay que destacar que basa sus aplicaciones web, como ya hemos indicado anteriormente, en el paradigma modelo-vista-controlador [41].

En Django, los modelos del paradigma se llaman igualmente modelos (*models*). Por el contrario, los controladores se llaman vistas (*views*), lo cual es, en nuestra opinión, una mala elección de nombre. Por último, las vistas se llaman plantillas (*templates*).

Los modelos de Django definen una capa de abstracción del sistema gestor de base de datos que tengamos, siendo una suerte de clase intermedia para el almacenamiento. Por su parte, los controladores describen las funciones que se llamarán al acceder a una URL u otra, siendo estas a su vez las que llaman a las diversas vistas (*templates*) que son las encargadas de generar el HTML que se visualizará en el navegador.

La representación intermedia: Los ficheros XML

Dado que nuestro proyecto integra tres grandes herramientas y que es necesaria la interoperabilidad entre ellas, necesitábamos una representación intermedia de formularios e instancias. Para ello de discutió si utilizar XForms que es el estándar de la W3C para la recolección de datos o utilizar un esquema propio basado en XML o JSON [42].

Finalmente, al sopesar las ventajas e inconvenientes de las diferentes alternativas, decidimos modelar, en la etapa de diseño, un DSL [43] (Domain Specific Language) propio basado en XML para que ocupara este rol de representación intermedia.

El motivo de descartar XForms fue principalmente debido a su complejidad, pues conllevaba un aprendizaje exhaustivo del estándar, además de complicar bastante el sencillo modelo que proponíamos. Por otra parte, nuestra herramienta no pretendía explotar datos recolectados de terceros ni pretendíamos que otras aplicaciones utilizaran nuestros datos recolectados con lo que no consideramos perjudicial utilizar una estructura de intercambio propia.

Al elegir entre utilizar una representación en XML o en JSON, finalmente decidimos optar por XML debido a que nos era más familiar y que existen gran cantidad de herramientas para el *parsing* de este tipo de documentos (ElementTree [44] para Python, SAX [45] o DOM [46] para Java,…).

A continuación se muestra el **XML Schema** [47] que describe los ficheros XML de definición de **formularios**.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
            version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="form">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="id" type="xsd:string"/>
        <xsd:element name="meta">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="version" type="xsd:int"/>
              <xsd:element name="name" type="xsd:string"/>
              <xsd:element name="author" minOccurs="0">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="user" type="xsd:string"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="geolocalized" type="xsd:string"/>
              <xsd:element name="description" type="xsd:string" minOccurs="0"/>
              <xsd:element name="creationdate" type="xsd:string" minOccurs="0"/>
              <xsd:element name="active" type="xsd:string" minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="sections">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="unbounded" name="section">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="id" type="xsd:string"/>
                    <xsd:element name="name" type="xsd:string"/>
                    <xsd:element name="fields">
                      <xsd:complexType>
                        <xsd:sequence>
                          <xsd:element maxOccurs="unbounded" name="field">
                            <xsd:complexType>
                              <xsd:sequence>
                                <xsd:element name="id" type="xsd:string"/>
                                <xsd:element name="label" type="xsd:string"/>
                                <xsd:element name="type" type="xsd:string"/>
                                <xsd:element name="required" type="xsd:string"
                                             minOccurs="0"/>
```

```
                        <xsd:element name="options" minOccurs="0">
                          <xsd:complexType>
                            <xsd:sequence>
                              <xsd:element maxOccurs="unbounded" name="option">
                                <xsd:complexType>
                                  <xsd:sequence>
                                    <xsd:element name="label"
                                                 type="xsd:string"/>
                                    <xsd:element name="value"
                                                 type="xsd:string"/>
                                    <xsd:element name="default"
                                                 type="xsd:string"
                                                 minOccurs="0"/>
                                  </xsd:sequence>
                                </xsd:complexType>
                              </xsd:element>
                            </xsd:sequence>
                          </xsd:complexType>
                        </xsd:element>
                        <xsd:element name="properties" minOccurs="0">
                          <xsd:complexType>
                            <xsd:sequence>
                              <xsd:element maxOccurs="unbounded"
                                           name="property">
                                <xsd:complexType>
                                  <xsd:sequence>
                                    <xsd:element name="name"
                                                 type="xsd:string"/>
                                    <xsd:element name="value"
                                                 type="xsd:string"/>
                                  </xsd:sequence>
                                </xsd:complexType>
                              </xsd:element>
                            </xsd:sequence>
                          </xsd:complexType>
                        </xsd:element>
                      </xsd:sequence>
                    </xsd:complexType>
                  </xsd:element>
                  <xsd:element maxOccurs="unbounded" name="group" minOccurs="0">
                    <xsd:complexType>
                      <xsd:sequence>
                        <xsd:element name="id" type="xsd:string"/>
                        <xsd:element name="list" type="xsd:string"/>
                        <xsd:element name="label" type="xsd:string"/>
                        <xsd:element name="required" type="xsd:string"
                                     minOccurs="0"/>
                        <xsd:element maxOccurs="unbounded" name="field">
                          <xsd:complexType>
                            <xsd:sequence>
                              <xsd:element name="id"type="xsd:string"/>
                              <xsd:element name="label"type="xsd:string"/>
                              <xsd:element name="type"type="xsd:string"/>
                              <xsd:element name="required"type="xsd:string"
                                           minOccurs="0"/>
                              <xsd:element name="options"minOccurs="0">
                                <xsd:complexType>
                                  <xsd:sequence>
                                    <xsd:element maxOccurs="unbounded"
                                                 name="option">
                                      <xsd:complexType>
                                        <xsd:sequence>
                                          <xsd:element name="label"
                                                       type="xsd:string"/>
                                          <xsd:element name="value"
                                                       type="xsd:string"/>
                                          <xsd:element name="default"
                                                       type="xsd:string"
                                                       minOccurs="0"/>
                                        </xsd:sequence>
                                      </xsd:complexType>
                                    </xsd:element>
                                  </xsd:sequence>
                                </xsd:complexType>
                              </xsd:element>
                              <xsd:element name="properties" minOccurs="0">
```

```
                                    <xsd:complexType>
                                      <xsd:sequence>
                                        <xsd:element maxOccurs="unbounded"
                                                     name="property">
                                          <xsd:complexType>
                                            <xsd:sequence>
                                              <xsd:element name="name"
                                                           type="xsd:string"/>
                                              <xsd:element name="value"
                                                           type="xsd:string"/>
                                            </xsd:sequence>
                                          </xsd:complexType>
                                        </xsd:element>
                                      </xsd:sequence>
                                    </xsd:complexType>
                                  </xsd:element>
                                </xsd:sequence>
                              </xsd:complexType>
                            </xsd:element>
                            <xsd:element name="properties" minOccurs="0">
                              <xsd:complexType>
                                <xsd:sequence>
                                  <xsd:element maxOccurs="unbounded"
                                               name="property">
                                    <xsd:complexType>
                                      <xsd:sequence>
                                        <xsd:element name="name"
                                                     type="xsd:string"/>
                                        <xsd:element name="value" type="xsd:int"/>
                                      </xsd:sequence>
                                    </xsd:complexType>
                                  </xsd:element>
                                </xsd:sequence>
                              </xsd:complexType>
                            </xsd:element>
                          </xsd:sequence>
                        </xsd:complexType>
                      </xsd:element>
                    </xsd:sequence>
                  </xsd:complexType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Código 3-1 Propuesta de XML Schema (XSD) de un formulario.

El esquema anterior muestra cómo debe ser un documento XML que represente a un formulario.

El primer elemento que se debe encontrar es la etiqueta `form`. A partir de ella observamos que ha de aparecer un identificador y tras éste, comenzarán los metadatos (versión, nombre, autor,…). Como observación cabe destacar que el autor no tiene porqué estar necesariamente presente. Esto se debe a que en ésta primera aproximación de diseño decidimos no implementar la gestión de usuarios y al diseñar el esquema de XML permitimos que no se tuviera que introducir necesariamente un usuario creador.

Tras terminar toda la sección de metadatos comenzarán las secciones del formulario, denominadas `sections`. Como indica que atributo `maxOcurs="unbounded"` tenemos que

puede haber muchas secciones dentro de un formulario. Cada sección debe tener un identificador y un nombre, además de una lista de campos a continuación.

Cada campo tendrá una serie de elementos que le corresponden (identificador, etiqueta, tipo,…). Teniendo especial tratamiento el campo “requerido” que sólo aparecerá en el fichero XML en caso de que el correspondiente campo sea requerido. Por ello, esta etiqueta puede aparecer o no.

Además cada campo puede tener una lista de propiedades y otra de opciones.

Por su parte, dentro de la sección, además de campos puede haber grupos de campos. Estos grupos tienen internamente etiquetas que indican su ID, su etiqueta identificadora, si es una lista, si es requerido, etc. Dentro de cada grupo habría una lista de campos que pertenecen a dicho grupo (y que pueden tener sus opciones y propiedades como cualquier campo). Por su parte, y para finalizar, el grupo también puede tener sus propiedades.

A continuación se incluye el fichero **XML Schema** que describe los ficheros XML de las **instancias**.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
            version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="instance">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="id" type="xsd:string"/>
        <xsd:element name="meta">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="formid" type="xsd:int"/>
              <xsd:element name="author">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="user" type="xsd:string"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="creationdate" type="xsd:string"/>
              <xsd:element name="modificationdate" type="xsd:string"/>
              <xsd:element name="completed" type="xsd:boolean"/>
              <xsd:element name="imei" type="xsd:string" minOccurs="0"/>
              <xsd:element name="signature" type="xsd:string" minOccurs="0"/>
              <xsd:element name="geolocalization" minOccurs="0">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="latitude" type="xsd:decimal"/>
                    <xsd:element name="longitude" type="xsd:decimal"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="sections">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="unbounded" name="section">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="id" type="xsd:int"/>
                    <xsd:element name="fields">
                      <xsd:complexType>
                        <xsd:sequence>
                          <xsd:element maxOccurs="unbounded" name="field">
                            <xsd:complexType>
```

```
                              <xsd:sequence>
                                <xsd:element name="id" type="xsd:string"/>
                                <xsd:element name="value" type="xsd:string"/>
                                <xsd:element name="order" type="xsd:int"/>
                                <xsd:element name="formfieldid" type="xsd:int"/>
                              </xsd:sequence>
                            </xsd:complexType>
                          </xsd:element>
                          <xsd:element minOccurs="0" maxOccurs="unbounded" name="group">
                            <xsd:complexType>
                              <xsd:sequence>
                                <xsd:element name="groupid" type="xsd:int"/>
                                <xsd:element maxOccurs="unbounded" name="element">
                                  <xsd:complexType>
                                    <xsd:sequence>
                                      <xsd:element maxOccurs="unbounded" name="field">
                                        <xsd:complexType>
                                          <xsd:sequence>
                                            <xsd:element name="id"type="xsd:string"/>
                                            <xsd:element name="ordengrupo"
                                                         type="xsd:int"/>
                                            <xsd:element name="value"
                                                         type="xsd:string"/>
                                            <xsd:element name="order"type="xsd:int"/>
                                            <xsd:element name="formfieldid"
                                                         type="xsd:int"/>
                                          </xsd:sequence>
                                        </xsd:complexType>
                                      </xsd:element>
                                    </xsd:sequence>
                                  </xsd:complexType>
                                </xsd:element>
                              </xsd:sequence>
                            </xsd:complexType>
                          </xsd:element>
                        </xsd:sequence>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Código 3-2 Propuesta de XML Schema (XSD) de una instancia.

En esta ocasión describiremos el esquema (fichero XSD) del fichero XML de una instancia. Al igual que el fichero de formularios, comienza con un identificador y una sección de metadatos (autor, fechas de creación y modificación, geolocalización, etc…).

Tras el apartado e metadatos nos encontramos con el de secciones. En este caso el esquema es mucho más sencillo debido a que en el caso de las instancias, para cada campo sólo se almacena su identificador como campo-instancia, su valor, su orden dentro de la instancia y finalmente el identificador del campo-formulario al que hace referencia.

Cabe destacar que el campo valor de cada uno de los campos-instancia puede ser muy diferente (desde una imagen a un texto, pasando por números o fechas).

Para el caso de los grupos, también hay una gran simplificación respecto al XML del formulario. En una instancia, cuando aparezca la etiqueta grupo, sólo esperamos que aparezca el id del grupo y una lista de elementos (el resto de la información del grupo estará en el XML

del formulario). Cada elemento puede estar formado por uno o muchos grupos. Si estamos ante una lista, habrá muchos elementos. Por el contrario, si nos encontramos ante un grupo que no forma parte de una lista, sólo habrá un elemento. Ilustrando esto con un ejemplo, si el grupo pretende representar a un familiar de quién rellena la instancia, cada elemento estará formado, por ejemplo, por un campo “nombre” y otro “parentesco”. De tal manera que en la instancia, dentro del grupo, tendremos tantos elementos como familiares; cada uno con su nombre y parentesco con la persona que rellenó la instancia.

Se ilustra ahora al lector con un ejemplo de fichero **XML** con la definición de un **formulario**, basado en el XSD anterior.

```
<?xml version="1.0" encoding="UTF-8"?>
<form>
  <id/>
  <meta>
    <version>1</version>
    <name>Ejemplo prototipo 1</name>
    <author>
      <user>turawet</user>
    </author>
    <geolocalized/>
  </meta>
  <sections>
    <section>
      <id>1</id>
      <name>Información personal</name>
      <fields>
        <field>
          <id>1</id>
          <label>Nombre</label>
          <type>TEXT</type>
          <required/>
          <creationdate><creationdate/>
          <properties>
            <property>
              <name>MAX_LENGTH</name>
              <value>100</value>
            </property>
          </properties>
        </field>
        <field>
          <id>2</id>
          <label>Fotografía</label>
          <type>IMAGE</type>
          <properties/>
        </field>
      </fields>
    </section>
    <section>
      <id>2</id>
      <name>Preferencias</name>
      <fields>
        <field>
          <id>3</id>
          <label>Color favorito</label>
          <type>RADIO</type>
          <required/>
          <properties/>
          <options>
            <option>
              <label>Rojo</label>
              <value>R</value>
            </option>
            <option>
              <label>Verde</label>
              <value>V</value>
            </option>
            <option>
```

```
              <label>Azul</label>
              <value>A</value>
            </option>
          </options>
        </field>
        <field>
          <id>4</id>
          <label>Fecha favorita</label>
          <type>DATE</type>
          <properties/>
        </field>
      </fields>
    </section>
    <section>
      <id>3</id>
      <name>Información familiar</name>
      <fields>
        <field>
          <id>5</id>
          <label>Número de hijos</label>
          <type>NUMERIC</type>
          <properties/>
        </field>
      </fields>
    </section>
  </sections>
</form>
```

Código 3-3 Ejemplo de fichero XML de un formulario.

A continuación, y por último, se muestra un ejemplo de fichero **XML** con la representación de una **instancia**, basado en el XSD anteriormente descrito.

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
  <id/>
  <meta>
    <formid>1</formid>
    <author>
      <user>turawet</user>
    </author>
    <creationdate>2011-06-20</creationdate>
    <modificationdate>2011-06-20</modificationdate>
    <completed>true</completed>
    <geolocalization>
      <latitude>28.353685</latitude>
      <longitude>-16.371717</longitude>
    </geolocalization>
  </meta>
  <sections>
    <section>
      <id>1</id>
      <fields>
        <field>
          <id/>
          <value>Alberto</value>
          <order>1</order>
          <formfieldid>1</formfieldid>
        </field>
        <field>
          <id/>
          <value>
            <filename>mifotografia.png</filename>
<binary>iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAYAAAAf8/9hAAALVWlUWHRYTUw6Y29tLmFkb2JlLnht
cAAAAAAAPD94cGFja2V0IGJlZ2luPSLvu78iIGlkPSJXNU0wTXBDZWhpSHpyZVN6TlRjemtjOWQi
…... ==</binary>
          </value>
          <order>2</order>
```

```
          <formfieldid>2</formfieldid>
        </field>
        <field>
          <id/>
          <value>R</value>
          <order>3</order>
          <formfieldid>3</formfieldid>
        </field>
      </fields>
    </section>
    <section>
      <id>2</id>
      <fields>
        <field>
          <id/>
          <value>
            <day>19</day>
            <month>6</month>
            <year>2011</year>
          </value>
          <order>4</order>
          <formfieldid>4</formfieldid>
        </field>
      </fields>
    </section>
    <section>
      <id>3</id>
      <fields>
        <field>
          <id/>
          <value>10</value>
          <order>5</order>
          <formfieldid>5</formfieldid>
        </field>
      </fields>
    </section>
  </sections>
</instance>
```

Los servicios web: La interconexión entre los diferentes sistemas.

Una de las primeras dudas que surgieron cuando estábamos diseñando la plataforma, fue cómo íbamos a comunicar las distintas aplicaciones que la conformaban. La interconexión debía ser lo suficientemente flexible y eficiente para que éstas pudieran operar de forma cómoda y relativamente eficiente. Además, había que tener en cuenta un factor importantísimo: la heterogeneidad de tecnologías con las que trabajábamos. Por un lado tenemos una aplicación para dispositivos móviles Android, hecha en Java; pero con una posible exportación futura a otras plataformas como iOS [48] e incluso una aplicación web. Luego, el modelador y el repositorio estarían desarrollados utilizando el *framework* Django de Python. Además, como el modelador sería una aplicación web, cabía la posibilidad de hacer los envíos de forma asíncrona, haciendo uso de AJAX. En fin, la diversidad de tecnologías era importante y el factor tiempo jugaba en nuestra contra.

Visto el panorama, y tras hacer una pequeña investigación sobre las tecnologías que se estaban utilizando en otros proyectos, hicimos una lista de posibles candidatos para la parte de comunicación entre sistemas.

Una de las primeras tecnologías que analizamos fue CORBA [49], un estándar que establece una plataforma de desarrollo de sistemas distribuidos, facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. Parecía interesante, más aún porque era algo que tendríamos que aprender a usar, ya que ninguno de nosotros tenía experiencia en

ese campo. Desgraciadamente, fue descartado debido a que no encontramos una manera sencilla de utilizarla en la aplicación móvil. En los foros de desarrollo de Google no la recomendaban como una solución a tomar a la hora de desarrollar aplicaciones Android.

Por otro lado, estudiamos la posibilidad de utilizar Java RMI [50]. Pensábamos que siendo una tecnología completamente basada en Java, podíamos implementarlo en la aplicación Android. Incluso vimos la posibilidad de comunicar Java con Python basándonos en este modelo. El problema llegó cuando descubrimos que Java RMI no estaba disponible para Android., este fue el motivo de descarte de esta segunda tecnología.

Finalmente, estudiamos la idea de utilizar servicios web. Para comenzar, necesitábamos contar con un servidor que publicara los servicios ofrecidos. Este servidor de servicios web, estaría escrito en Python, mediante el framework Django. Además, era necesario implementar dos clientes, uno en el modelador web y otro en la aplicación móvil. El modelador sería desarrollado en Python, al igual que el administrador del repositorio. Para el caso de las aplicaciones Python vimos que existían dos principales módulos ya desarrollados (soaplib [51], pensado principalmente para ser servidor de servicios web, y Suds [52], diseñado especialmente como cliente de servicios web). Para la aplicación Android, usaríamos una librería diseñada para ser usada en plataformas Java restringidas, como Applets [53] o J2ME y que era totalmente compatible.

Siguiendo la línea de desarrollo basada en servicios web, contribuíamos a utilizar estándares abiertos extendidos. El envío de mensajes es ligero, eficiente y sin demasiado *overhead* por parte de los protocolos. Todos los aspectos anteriormente descritos son considerados muy valiosos, ya que la aplicación con mayor tráfico de datos dentro de la plataforma iba a ser la móvil, que contaría con una conexión de datos con un ancho de banda reducido.

Por otro lado, teníamos la posibilidad de elegir entre dos formas de implementar servicios web. Una era hacerlos basados en SOAP [54] y la otra, utilizando REST [55]. Por la experiencia que teníamos, SOAP nos parecía una buena opción, ya que era conocida y muy cómoda para trabajar y depurar. Sin embargo, REST era una tecnología nueva y parecía ser bastante eficiente. Finalmente, nos inclinamos a desarrollar una solución basada en SOAP, principalmente porque era conocida por nosotros y porque las librerías que habíamos consultado para los diferentes lenguajes utilizados se adaptaban mejor a ella.

Una vez que habíamos determinado la metodología de interconexión que íbamos a utilizar y que contábamos con las herramientas necesarias para hacerlo, teníamos que definir minuciosamente que servicios nos harían falta para cumplir con los requisitos básicos de la plataforma. Los servicios básicos con los que debíamos contar, se enumeran a continuación:

a) Obtener todos los formularios disponibles para descargar. Este servicio sería consumido por la aplicación móvil cada vez que el usuario desease descargarse un formulario nuevo, mostrándole todos los formularios que hay disponibles.
b) Descargar la definición formulario en concreto. Una vez que el usuario recolector ha seleccionado un formulario, hará falta que se lo descargue a su terminal móvil.
c) Subir una nueva instancia. Este servicio permite a la aplicación móvil guardar una instancia en el repositorio.

d) Subir una nueva definición de formulario. Este servicio será consumido por el modelador web cada vez que se cree un nuevo formulario. El usuario modelador enviará el formulario al repositorio a fin de que esté disponible para descargar por los recolectores.

Para finalizar esta explicación, en la siguiente imagen se observa un esquema gráfico de cómo estarían desplegados los distintos componentes encargados de la comunicación entre las diferentes aplicaciones.

Capítulo 4. El modelador

Resumen:

- Se presenta uno de los tres módulos fundamentales del proyecto Turawet: el modelador, también conocido como Seed.
- Se recogen las etapas de análisis, toma de requisitos y diseño, previas a la implementación.
- Se detallan las tecnologías utilizadas, los principales problemas que aparecen durante la etapa de desarrollo o implementación, así como la solución a los mismos.

4.1 Introducción

Seed (de aquí en adelante Modelador), será el módulo encargado de modelar el formato, el tipo y el orden de los campos cuya información se recolectará posteriormente a través, principalmente, de dispositivos móviles.

Con él, lograremos de una forma muy simple componer un formulario que recoja cualquier dato que se requiera, pudiendo especificar el tipo de cada uno así como añadir validaciones adicionales, restricciones a la hora de rellenar el formulario, fijar la estructura de la información mediante secciones o añadir otra información relevante del mismo como, por ejemplo, si se trata o no de un formulario con instancias geolocalizadas. Con todo, conseguimos una aplicación en la que cualquier persona con escasos conocimientos técnicos y en pocos minutos logra confeccionar un formulario acorde a los datos que precisa recolectar.

Por tanto, una vez definido el mensaje a enviar entre el módulo modelador y el módulo administrador (la representación que tendría un formulario como texto) comenzamos a diseñar la aplicación que finalmente se encargará de generar los formularios.

4.2 Etapa de análisis

Durante la etapa de análisis definiremos a grandes rasgos los requisitos principales que la aplicación requiere para realizar su función. Función que no es otra que: diseñar y estructurar los datos de un formulario de forma que finalmente éste pueda almacenarse por otro de los módulos de la aplicación, el administrador.

Sabiendo que, a priori, la tarea de generación de formularios de recogida de datos haciendo uso de métodos tradicionales llega a tornarse una tarea compleja, haremos especial hincapié en que la interfaz de la aplicación sea amigable al usuario, usable y lo más intuitiva posible. Tratando de conseguir una aplicación cuya característica principal sea su gran facilidad de uso y sin caer en la complejidad de otras herramientas de modelado. Para ello, realizaremos un detallado análisis de usabilidad de la misma durante la etapa de diseño.

Otra labores de análisis llevadas a cabo consistieron en la identificación de los tipos de campos que iba a poseer el modelador, dejando la posibilidad abierta a modificaciones futuras. Dichos campos deben poder agruparse o gestionarse en grupos. Adicionalmente, dichos grupos de campos pueden ser tratados como listas de campos dinámicas para contemplar

aquellos casos en los que se desconoce a priori el número de datos a recoger, opción que también debe aparecer en el modelador.

Por último, el modelador debe generar un resultado tal que el contenido del mensaje representando al formulario generado sea manejable por el administrador del repositorio.

4.2.1 Requisitos

De lo comentado anteriormente podemos extraer, como requisitos generales, que la aplicación debe:

- Generar formularios en donde desplegar campos de una serie de tipos prefijados, como son: campos de texto, imágenes, audio, vídeo, campos numéricos, fechas y selectores de opciones (*checkbox*, *combo* y *radio*). Así como dar facilidades para la ampliación del conjunto de campos disponibles.
- Gestionar las propiedades independientes de cada campo, tamaño máximo o validaciones adicionales.
- Estructurar los campos dividiéndolos en grupos de campos (agrupados con una misma semántica) así como secciones.
- Permitir modelar formularios con campos o grupos de campos que sean dinámicos. Por ejemplo, listas de imágenes o listas de personas (en donde persona sería un grupo con tres campos, por ejemplo, uno para el número de identificación, otro para el nombre y un tercero para sus apellidos).
- Aunar facilidad de uso y rapidez en el desarrollo de formularios sin dejar de ofrecer al usuario la posibilidad de gestionar todas sus características. Como pueden ser: generar instancias geolocalizadas o establecer campos requeridos.
- Interfaz simple y uniforme en todos los sistemas.
- Ser capaz de comunicarse y enviar un formulario al módulo administrador de Turawet manejando, además, la respuesta del mismo.

4.2.2 Diagramas UML

Durante esta etapa de análisis, además, representaremos mediante algunas de las notaciones especificadas en UML, el comportamiento general del módulo Modelador de Turawet.

Para ello hemos hecho uso de una interesante y completa aplicación de modelado UML llamada Software Ideas Modeler [56].

Diagrama de casos de uso

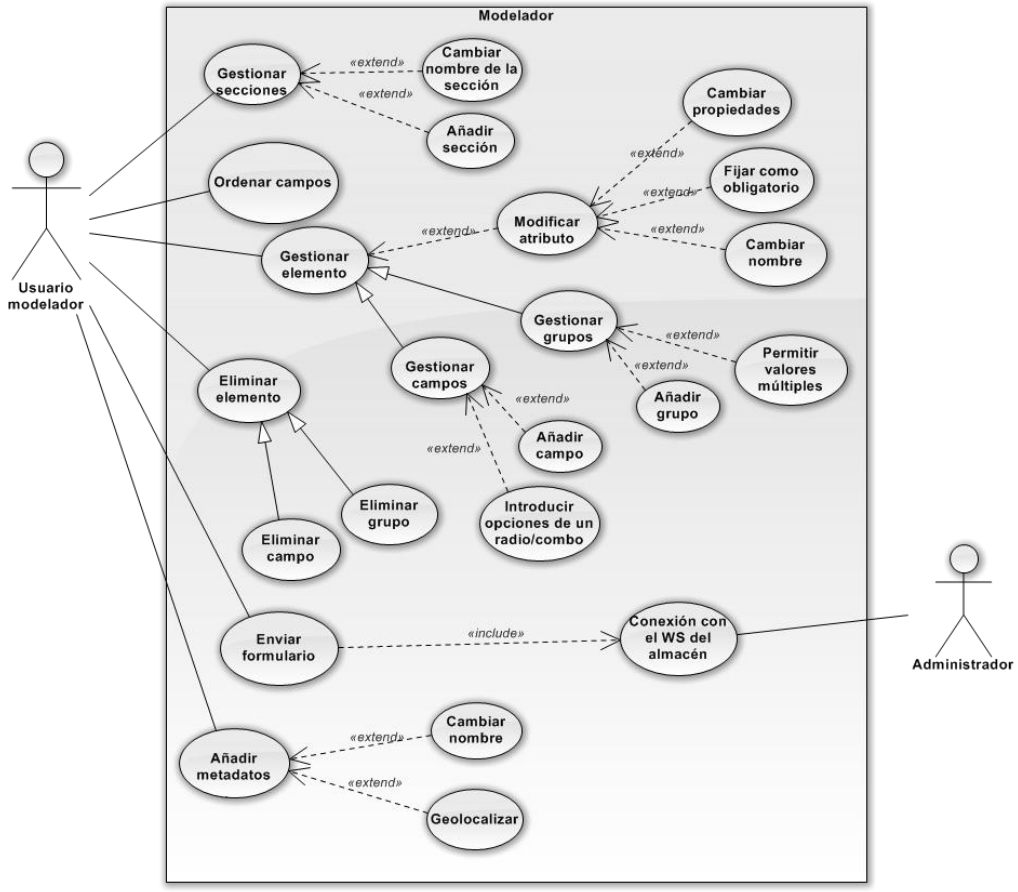


Figura 4-1 Diagrama de casos de uso del modelador, escenario principal.

En la Figura 4-1, se muestra un diagrama de casos de uso del módulo Modelador, siguiendo la notación gráfica propuesta por UML.

Como actores principales de los casos de uso propuestos aparecen:

1. **El usuario modelador**: Haciendo referencia a la persona que tomará el rol de creador de formularios, utilizando para ello nuestra herramienta, tomando todas las decisiones propias del modelado.
2. **El módulo repositorio**: Como entidad no física, este actor nos facilitará aquellas funciones de las que otros casos de uso de la aplicación necesitarán nutrirse finalmente para que los formularios moldeados persistan.

Descripción de casos de uso

A continuación, se detallan los casos de más importantes representados en el diagrama, el resto se pueden encontrar en el apéndiceCasos de uso del Modelador:

Identificador: *CU-M-4*

Nombre: “Añadir campo”.

Descripción

El modelador añade un nuevo campo a una sección al formulario.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Se ha creado al menos una sección.

Post-condiciones

1. Se crea una instancia de campo.
2. Se asocia el campo con la sección correspondiente.

Frecuencia

Frecuente.

Flujo de escenario principal

1. El usuario accede a la pantalla de creación de un formulario.
2. El usuario arrastra el campo sobre una sección.
3. Aparece el nuevo campo de la sección en el documento.

Identificador: *CU-M-3*

Nombre: “Modificar atributo”.

Descripción

El usuario modelador desea cambiar el valor que toma un atributo de un campo o grupo determinado.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Ha añadido una sección al formulario
3. La sección dispone de al menos un campo o grupo de campos.

Post-condiciones

1. Se modifica el valor del atributo de la instancia del campo o grupo del formulario correspondiente.

Frecuencia

Muy frecuente.

Flujo de escenario principal

1. El usuario pulsa sobre el valor del atributo a modificar.
2. Modifica el valor.
3. El valor se actualiza en el elemento visual que representa el atributo.

Identificador: *CU-M-6*

Nombre: “Eliminar campo”.

Descripción

El modelador elimina un campo de una sección del formulario.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Se ha creado al menos una sección.
3. Existe al menos un campo en una sección.

Post-condiciones

1. Se localiza la instancia del campo dentro de la sección.
2. Se desasocia el campo de la sección correspondiente.
3. Se elimina el campo.

Frecuencia

Frecuencia normal.

Flujo de escenario principal

1. El usuario accede a la pantalla de creación de un formulario.
2. El usuario pulsa sobre el botón de eliminar de un campo.
3. Desaparece el campo de la sección en el documento.

Diagrama de secuencia

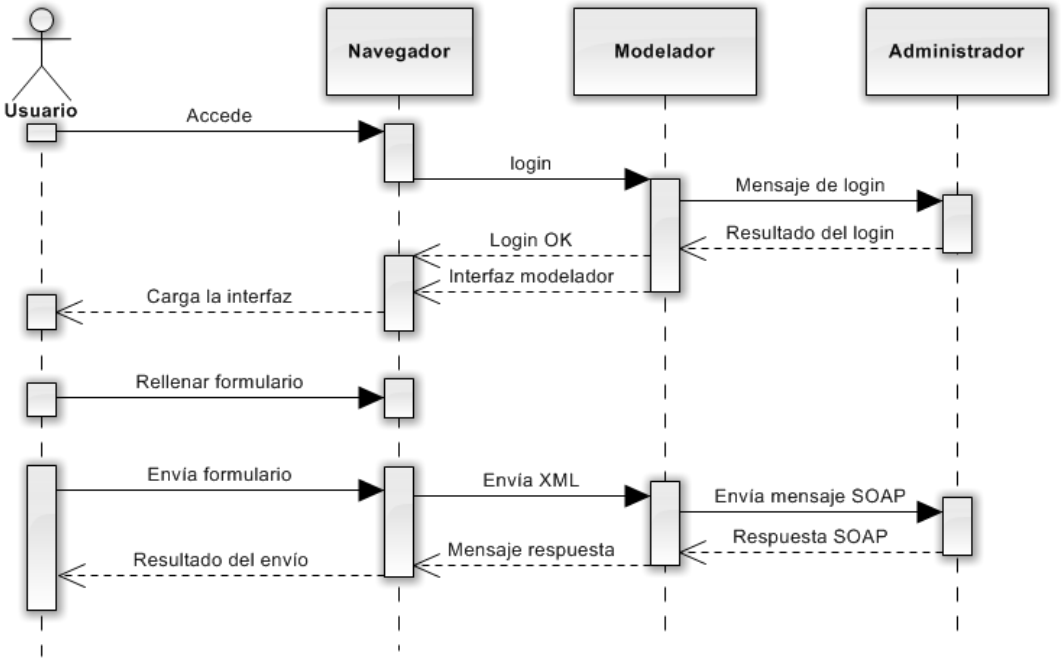


Figura 4-2 Diagrama de secuencia del modelador.

En el diagrama de secuencia de la Figura 4-2 se representa la interacción entre el usuario modelador y los diferentes módulos del proyecto, a lo largo del tiempo, para realizar un envío correcto de un formulario.

Dentro de todas los flujos o caminos posibles que se pueden tomar, hemos elegido representar en el diagrama el más significativo de todos y, por lo general, el que más valor aportará al lector de manera que tenga una visión global del cometido principal del modelador.

Pasando a detallar cada acción, antes de acceder a la interfaz del modelador será necesario iniciar la sesión del usuario haciendo *login* desde el módulo administrador, o Beekeeper. Una vez realizado este paso se mostrará en el navegador la interfaz de la aplicación.

Tras modelar un formulario completo, añadiendo elementos al mismo y modificándolos, el usuario modelador enviará el formulario modelado, encargándose el navegador de transformar lo que el usuario ha creado de forma gráfica a un XML que enviará al servidor de nuestro módulo. Éste, a su vez, se comunicará mediante un servicio web con el módulo de administración, o Beekeeper, el cuál le responderá que el formulario se ha guardado correctamente.

Finalmente se le mostrará al usuario un mensaje que refleje que la petición de envío se ha realizado correctamente y sin ningún problema.

4.3 Etapa de diseño

Una vez fijados los requisitos debemos dar respuesta a ellos. Durante la etapa de diseño daremos resolución a las preguntas que irán surgiendo a raíz del análisis realizado. Justificando, en los siguientes apartados, nuestras decisiones y detallando las elecciones de diseño tomadas dentro de las distintas opciones que podrían existir.

4.3.1 Decisiones de diseño

Dado que requerimos una interfaz unificada y sencilla, se nos plantea la primera pregunta, cuya respuesta repercutirá en una de las decisiones más importantes y que marcarán el resto del diseño así como de la implementación: ¿Qué entorno utilizar para la aplicación? ¿Una aplicación de escritorio con diferentes variantes para los distintos sistemas o una interfaz web genérica para todos con las posibles limitaciones de un navegador web?

Evidentemente, hemos optado por tomar el segundo camino. Generar una única aplicación web que se abstraiga de la plataforma sobre la que se ejecuta, que cumpla con los estándares marcados y que se aproveche de los recientes avances en las tecnologías en dicho ámbito tiene, sin duda, más ventajas que inconvenientes.

¿Por qué una aplicación web?

La respuesta a esa pregunta es sencilla. A día de hoy, el mundo se mueve en la nube, cada vez son más las aplicaciones de las que hacemos un uso, más o menos intenso, ejecutándose desde un servidor remoto y utilizando como interfaz final con el usuario un navegador web. El correo electrónico, los gestores de fotografías o los gestores de documentos colaborativos, son ejemplos de aplicaciones que en pocos años han evolucionado de la mano de Internet, dejando atrás el entorno de escritorio y logrando así unificar su interfaz, eliminando la sensibilidad a la plataforma que los ejecuta, la necesidad de instalación, siendo actualizadas continuamente y accesibles desde cualquier equipo conectado a la red de redes.

Mediante una aplicación web, logramos disponer de una única aplicación que ejecuta en un gran y heterogéneo grupo de equipos y sistemas. Desde dispositivos móviles, pasando por televisores conectados a la red, así como los tradicionales equipos portátiles o de sobremesa. No haciendo distinción entre su sistema operativo o los requisitos previos para su instalación y uso.

Es simple, sólo nos hace falta un navegador y conexión a Internet.

Si bien es cierto que de ninguna forma se trata de la panacea. Es bien sabido que no todos los navegadores cumplen los estándares definidos, y de cumplirlos no los interpretan de la misma forma. Pero el número y complejidad de las modificaciones y los parches a aplicar son sensiblemente menores.

¿Por qué "Drag & Drop"?

Al comenzar el análisis del proyecto, partimos con la idea consensuada de crear una aplicación web basada en el "*drag & drop*" (arrastrar y soltar). Porque en ella, el usuario tan sólo debe ir arrastrando los distintos tipos de campos hacia un área delimitada, uno detrás de otro o en el orden que requiera, pudiendo editar su información (nombre, propiedades, obligatoriedad o no...), con lo que evitamos un alto número de clics que pudiesen hacer engorroso el proceso. Cualquier usuario, sin muchos conocimientos y con una muy breve introducción a la herramienta, será capaz de generar su propio formulario en un intervalo muy corto de tiempo.

Para un usuario final, resulta mucho más intuitivo arrastrar un campo hacia la zona donde quiere que aparezca que, por ejemplo, hacer clic en un botón y que, sin seguir la trayectoria del usuario, aparezca repentinamente el campo en la zona editable del formulario. En la vida real, para colocar un elemento en un lugar determinado, partiendo de otro lugar y sin importar el tipo de elemento, el ser humano está habituado a coger dicho objeto y visualizar todo el trayecto desde el punto de partida hasta el lugar de reposo del mismo. No parece tan natural presionar un botón y que el objeto se transporte hacia su destino.

Otras decisiones de diseño

Al hilo del apartado anterior, ocurre lo mismo, por ejemplo, con la edición de propiedades de los campos, tales como longitud máxima o si se trata o no de un correo electrónico (entre otras validaciones), donde inicialmente barajamos incluirlas sobre el mismo espacio visual que ocupaba el propio campo en el formulario o, como hacen algunas herramientas, mostrar una pequeña ventana externa donde dichas propiedades fuesen editables tras hacer clic en una zona específica, pudiendo desorientar al usuario con el cambio de contexto. Finalmente, siguiendo con nuestros principios de elegir aquello que fuese, a priori, lo más intuitivo, optamos por la edición "en línea"; es decir, el texto será editable al pulsar sobre él.

Otra decisión de diseño fue la de fijar, de entrada, el ancho de la zona "usable" de la aplicación a 960 píxeles. ¿El por qué? La ventana de un navegador varía constantemente su tamaño y, por tanto, la zona dibujable del mismo no siempre es la misma, cambia con las diferentes resoluciones de pantalla o al escalar el tamaño de la propia ventana. Y dado que nos interesaba generar una aplicación que simule, en parte, un formulario en papel; no nos interesaba variar el ancho del mismo, puesto que en resoluciones grandes las zonas abiertas y sin información serían mayores y empobrecerían la interfaz final. Además, dentro de las aplicaciones web, un ancho de 960 píxeles es un estándar de facto entre diseñadores. Existen *frameworks* de estilos que fijan a ese valor el ancho de la página.

Dentro de la zona usable de la aplicación, un pequeño porcentaje vertical del lateral derecho de la misma se dedicará al panel donde aparecerán los diferentes tipos de campos que podremos arrastrar. Hemos tomado la decisión de que dicho panel se encuentre en un lateral (y en concreto, en el derecho) en donde los campos del formulario aparecerán uno debajo del otro. Si lo comparamos, por ejemplo, con el desplazamiento del cursor para arrastrar un campo desde una barra horizontal situada en la parte superior, este último es relativamente mayor.

Finalmente, para favorecer el desplazamiento durante el modelado del formulario en casos de un número extenso de campos y secciones, será necesario facilitar el desplazamiento entre las secciones del formulario por medio de algún tipo de acceso rápido.

4.3.2 Mockups

Para poder realizar un análisis del diseño de la aplicación y antes de pasar a la implementación, hemos decido hacer uso de *mockups* [57]para diseñar el aspecto de la interfaz del módulo modelador del proyecto.

Los *mockups*, en el mundo del software, son una forma de representar en papel la interfaz de un prototipo de aplicación, sin necesidad de implementar todo lo que un prototipo conllevaría.

Para generar las siguientes figuras hemos utilizado la aplicación Balsamiq Mockups [58]. Con esta herramienta somos capaces de diseñar interfaces con aspecto “de papel” en muy pocos minutos y con una gran cantidad de recursos personalizables.

Creación de un formulario

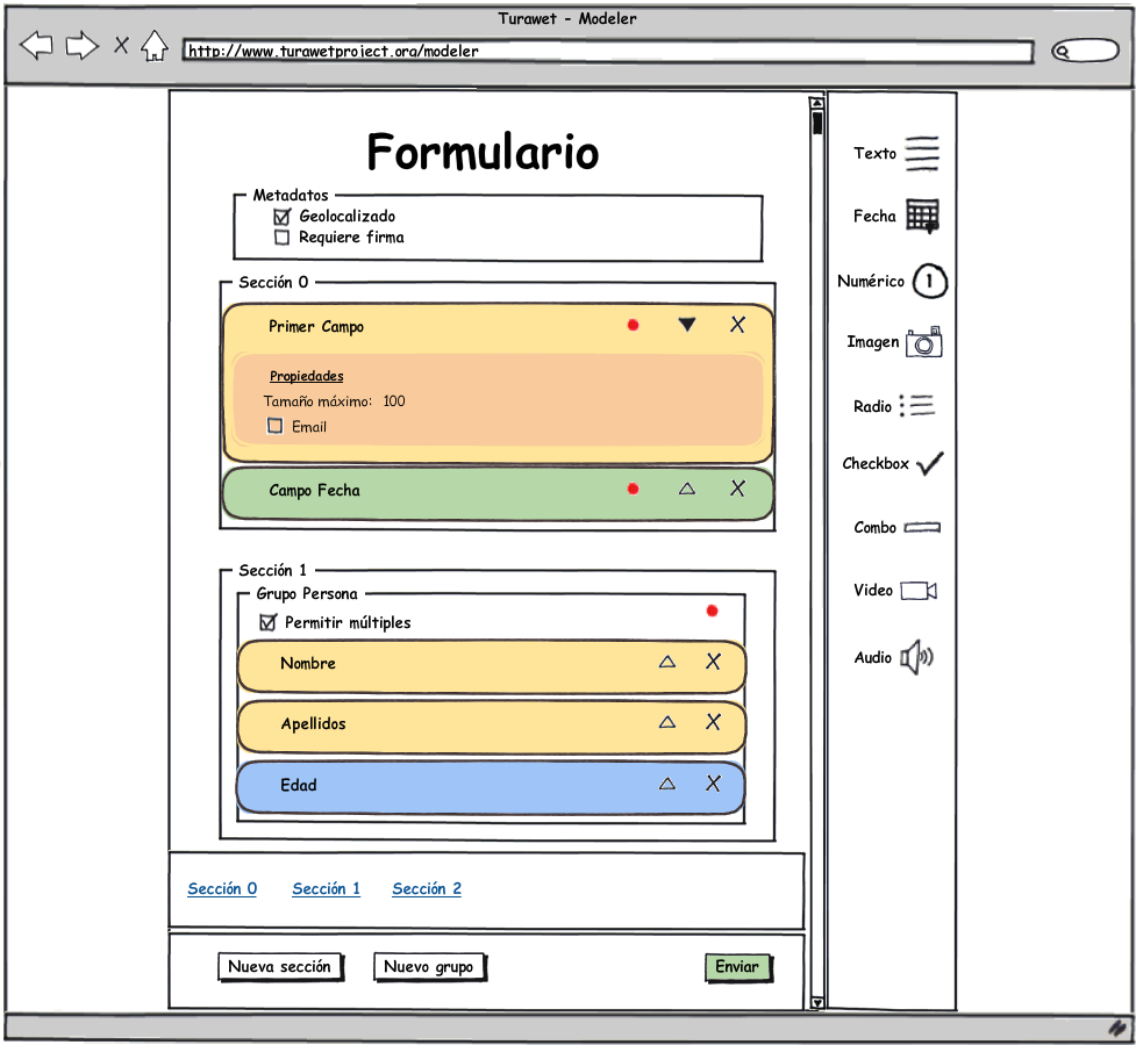


Figura 4-3 Mockup del modelador, diseño de formulario con campos de texto.

En la Figura 4-3 se representa el aspecto que tendrá la aplicación modeladora de formularios en un navegador web y con un formulario en proceso de generación. En ella aparecen dos zonas verticales claramente diferenciadas: la zona central donde aparece la sección correspondiente al formulario y su contenido y la zona lateral derecha donde aparecen los tipos de campos disponibles para añadir al formulario.

Además, en la parte inferior aparecen una serie de enlaces para un desplazamiento rápido entre secciones así como una barra inferior con diversos botones que permitirán añadir una nueva sección al formulario, añadir un nuevo grupo al formulario o enviar el formulario modelado.

Los tipos de campos que hemos decidido incluir son:

- *Tipo texto*: representará cualquier campo de un formulario donde se introduzcan una serie de caracteres alfanuméricos. Adicionalmente podrán realizarse sobre él validaciones extra como que no supere una longitud máxima o que el texto que contiene forme un correo electrónico.

- *Tipo fecha*: servirá para introducir una fecha a partir de un *widget* que nos facilite la tarea, por ejemplo un calendario. Entre otras propiedades, nos permitirá, al modelar, modificar el formato de la fecha.
- *Tipo numérico*: representa una cadena formada estrictamente por caracteres numéricos. Una validación adicional, establecida en las propiedades, puede ser el número más grande a introducir o el número máximo y mínimo de dígitos para, por ejemplo, códigos postales.
- *Imagen*: Campo que servirá para tomar una captura desde la cámara del dispositivo móvil y almacenarla.
- *Radio*: Nos permitirá introducir una serie de opciones (con una etiqueta que se mostrará en el texto de la opción y un valor que será el que finalmente se almacene) desde el modelador, de los cuales, finalmente, el usuario recolector elegirá uno y solamente uno.
- *Checkbox*: Campo que guardará un valor verdadero en caso de que se marque o falso en caso de que no se marque.
- *Combo*: Similar a Radio pero en esta ocasión las opciones se muestran las en una lista desplegable.
- *Vídeo*: Permite, con la cámara del dispositivo, grabar un vídeo y almacenarlo.
- *Audio*: Nos permitirá grabar el sonido captado por el micrófono de nuestro dispositivo recolector.

Dado que se encuentra en proceso de generación, ya aparecen en el formulario metadatos rellenos así como una serie de secciones con campos en su interior.

Según los metadatos del formulario de la Figura 4-3 y a tenor de los *checkbox* seleccionados, las instancias del formulario poseerán geolocalización pero no requerirán de firma. Habrá que rellenar obligatoriamente (puntos rojos) los campos Primer Campo, Campo Fecha y el grupo Persona y por lo tanto todos sus campos (que además es una lista y permite múltiples Personas en una misma instancia de formulario).

Cada campo, representado por un rectángulo de color (variando el color según su tipo) con las esquinas redondeadas, posee un nombre y una serie de botones que desempeñan diferentes acciones en su parte derecha. De derecha izquierda aparece: una “X” cuya función es eliminar el campo del formulario, una flecha que nos servirá para desplegar las propiedades (y los valores de las opciones para aquellos tipos de campos que las posean) y por último un punto gris que tras pulsarlo cambia a color rojo y vuelve obligatorio o requerido el campo a la hora de rellenar el formulario.

En el campo con nombre “Primer Campo” dentro de la sección con nombre “Sección 0”, aparecen desplegadas las propiedades del mismo, teniendo un tamaño máximo de 100 caracteres.

Tanto el nombre del propio formulario como los nombres de las secciones así como los nombres de los grupos, los nombres de los campos y los valores de las propiedades, son textos editables al pulsar sobre ellos.

Dada la simplicidad del *mockup*, el resto de elementos de la interfaz se entienden tan simples que no requieren de explicación.

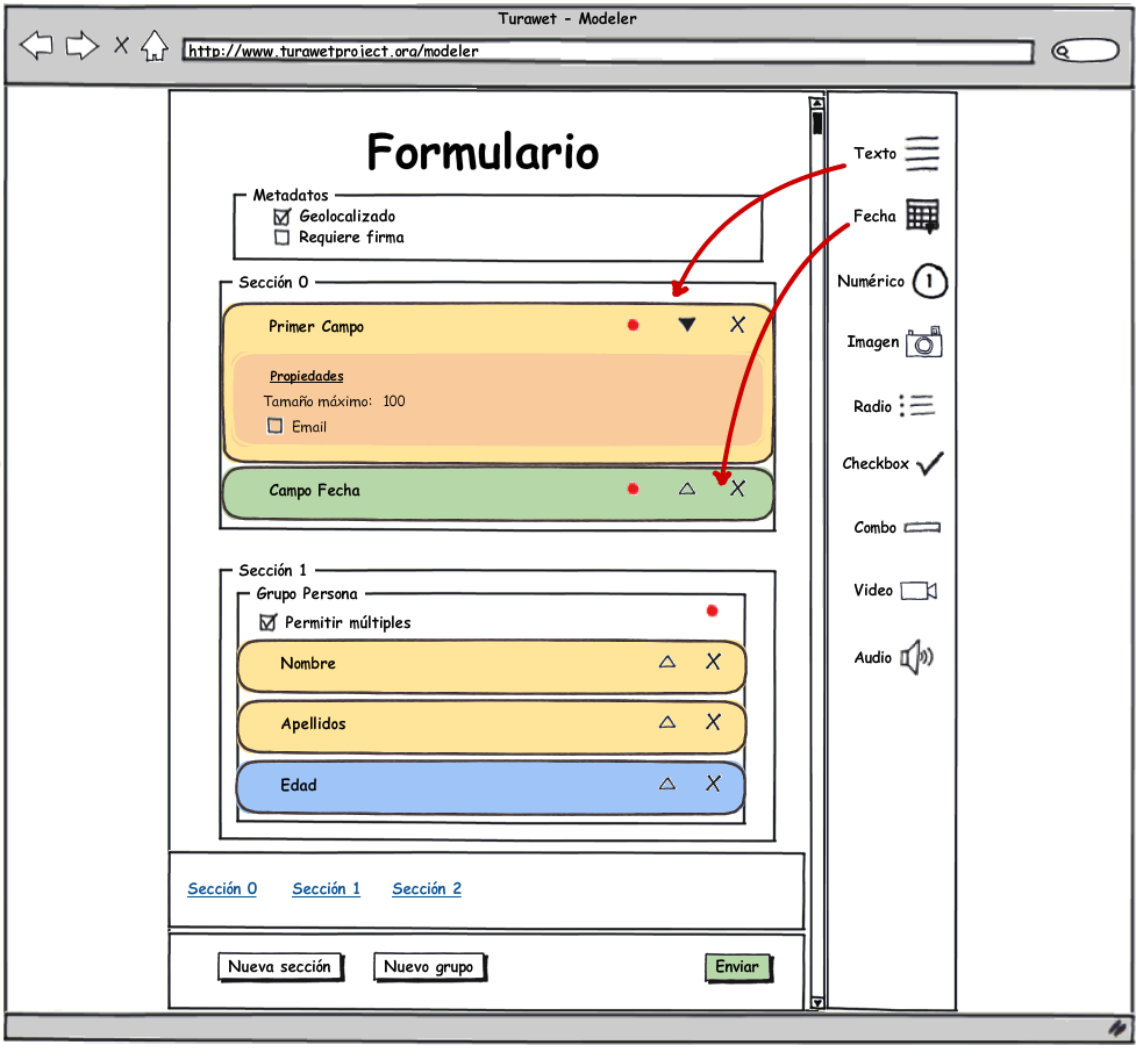


Figura 4-4 Mockup del modelador, diseño de formulario detalle del “arrastrar y soltar”.

En la Figura 4-4 vuelve a aparecer la pantalla de modelado de un formulario si bien se destaca el recorrido que deberá realizar el modelador arrastrando los tipos de campos seleccionados desde el lateral derecho hacia la Sección 0, uno primero y otro después, pudiendo reordenar los campos dentro de la sección.

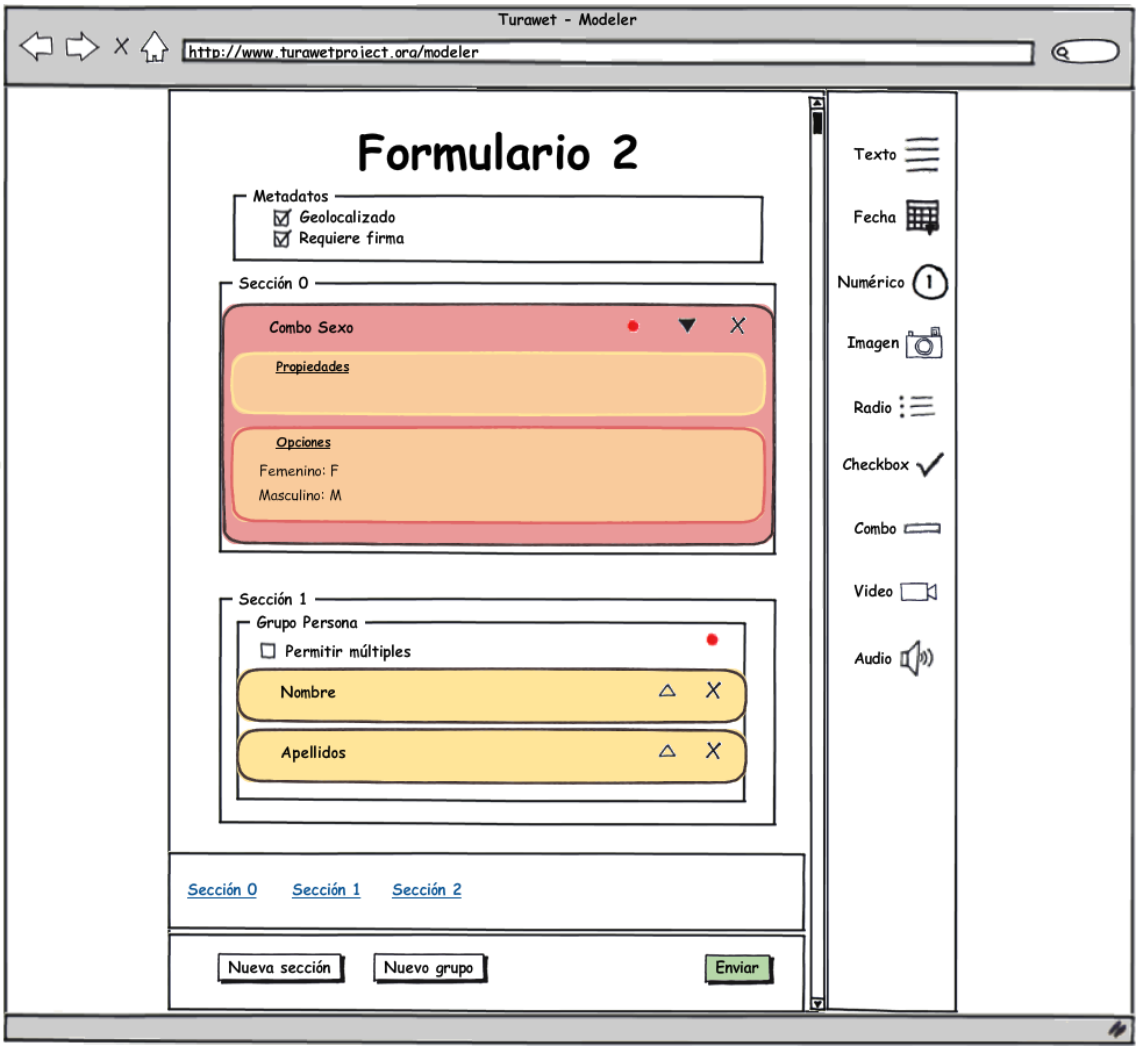


Figura 4-5 Mockup del modelador, diseño de formulario con campos de selección (checkbox, radios y combos).

En la Figura 4-5 aparece el modelado de otro formulario distinto, con las propiedades desplegadas para un campo de tipo Combo y mostrando la forma en la que aparecerían las diferentes opciones del mismo con sus correspondientes valores.

Envío de un formulario

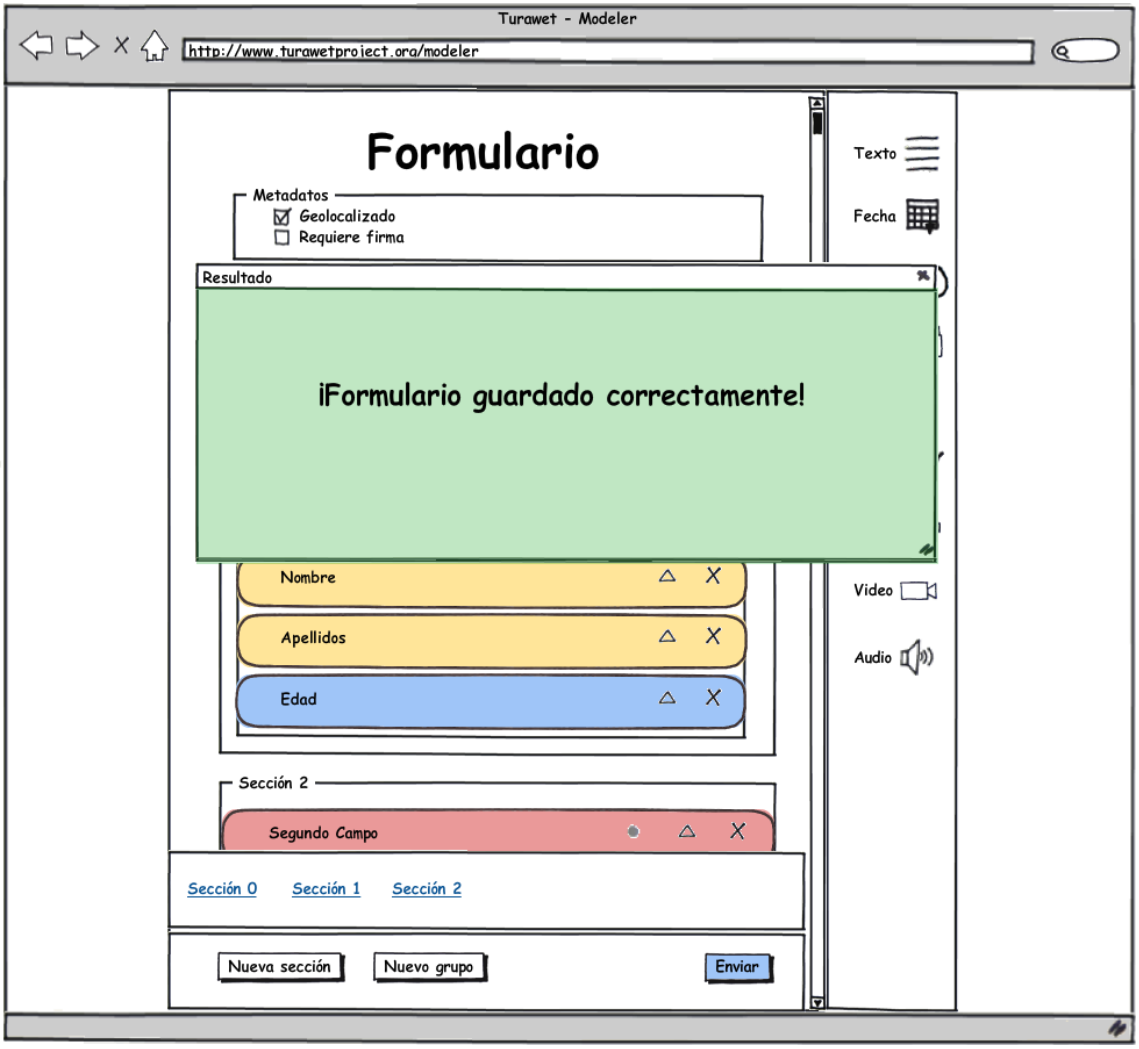


Figura 4-6 Mockup del modelador, envío de formulario.

En este caso, en la Figura 4-6 se ha representado la ventana de éxito que obtendría el usuario tras pulsar el botón “Enviar” (destacado en color azul) y guardar correctamente el formulario en el repositorio. A partir de ese momento el formulario dejaría de ser editable y pasaría a formar parte del listado de formularios disponibles en el repositorio.

4.4 Etapa de implementación

Una vez capturados los requisitos que deberá cumplir la aplicación modeladora de formularios, es momento de detallar la implementación. En esta etapa, nos basaremos en los planos generados y las descripciones que los acompañan para llevar a cabo la ejecución de las ideas de diseño. Obteniendo, finalmente, un prototipo con un acabado simple para el uso pero usable y completo en funcionalidades.

4.4.1 Tecnologías utilizadas

Antes de pasar a los detalles de implementación, hemos realizando un breve resumen sobre las tecnologías que en este módulo se han integrado.

- Para el entorno de ejecución del servidor que se encargará de procesar las peticiones realizadas desde el navegador al módulo Modelador así como la conexión con el módulo Beekeeper, hemos confiado en **Django,** el *framework* de desarrollo de aplicaciones web de **Python**.
- Además, para implementar el "*drag & drop*" hemos utilizado **HTML5** manejando los eventos del navegador con **JavaScript** [59], con el que también desarrollaremos el modelo de clases.
- Hemos enriquecido la interfaz con hojas de estilos en cascada (**CSS**).
- Para las animaciones y el tratamiento de los elementos del documento hemos decidido utilizar la librería JavaScript **jQuery**, la más arropada por la comunidad de desarrolladores y poseedora de una excelente documentación y gran compatibilidad con los diferentes navegadores que copan el mercado.
- Para consumir los **servicios web** proporcionados por el módulo Beekeeper, haremos uso del cliente **SOAP** para Python **SUDS**.

4.4.2 Decisiones de implementación

A continuación, y al igual que se ha realizado en las etapas anteriores, comentaremos las principales decisiones tomadas durante la etapa de implementación, además de justificar los razonamientos que nos han llevado hasta ellas.

Antes de comenzar, se ha de tener en cuenta que, como decisión de diseño general, la representación final del formulario será un XML, definido en el Parte I.Capítulo 1 de este documento. Generar este XML será el cometido final de nuestro módulo.

¿Por qué una aplicación web con HTML5 y jQuery?

Con la aplicación modeladora web del módulo Seed de Turawet, hemos generado una interfaz limpia, que logra centrar al usuario en lo importante de un formulario: "campos y secciones"; sin que éste se pierda durante su uso. Para ello, hemos decidido utilizar las posibilidades que nos proporciona la nueva definición del lenguaje de marcas predominante en el ámbito web, *HyperTextMarkup Language*, en su versión más reciente: HTML5.

Con la nueva versión, la especificación HTML da un salto similar al que supuso la inclusión del *tag* `<img/>` en la versión 2.0 publicada en 1995, una auténtica revolución. Entre otras cosas, HTML5 incluye en su definición los *tags* <audio/> y <video/> que nos permiten reproducir audio e imágenes en movimiento sin herramientas ni *plugins* de terceros, directamente desde el navegador, y sin depender de tecnologías propietarias.

Si bien podríamos lograr un efecto similar haciendo uso exclusivo de JavaScript y HTML4, en nuestro caso hemos utilizado el nuevo atributo definido en HTML5: *draggable*; para lograr ese "arrastrar y soltar" del que hablamos.

Por ejemplo, una etiqueta que identifique un bloque de contenido que haya sido definido con la capacidad de arrastrarlo quedaría así:

```
<div id="miContenido" draggable=true>
       Contenido "Arrastrable"
</div>
```

Código 4-1 Ejemplo de contenedor HTML con atributo *draggable*.

Tan sólo fijando el valor de este atributo a verdadero (*true*), un navegador compatible con HTML5 lo interpretará y modificará el cursor de forma que al situarlo sobre el elemento se perciba que se puede interactuar con él, podremos, incluso, pulsar sobre el elemento y arrastrarlo.

También, hemos querido hacer uso del conocimiento de una gran comunidad de desarrolladores JavaScript, utilizando jQuery durante el desarrollo, un *framework* muy bien documentado, software libre y de código abierto. Hecho que, como ya veremos, logrará que se enriquezca notablemente la interfaz final, a sabiendas de que aquello que utilizamos ha sido testeado por cientos de desarrolladores, siendo compatible con casi la totalidad de navegadores web disponibles en el mercado.

jQuery además, nos permitirá manipular el árbol DOM del documento y, en un futuro, añadir fácilmente la técnica AJAX a las comunicaciones entre el navegador y el servidor de nuestra aplicación.

Introducción a DOM

Para generar, eliminar o modificar nuevos elementos representables en el navegador a medida que el usuario interactúa con la aplicación, se hace necesario el manejo del documento HTML de forma dinámica en cliente.

Gestionaremos los diferentes elementos representables en un documento HTML rigiéndonos a la API definida por el estándar DOM (*Document Object Model*) del W3C. Por medio de DOM, lograremos que nuestra aplicación pueda acceder y modificar la estructura y los estilos de nuestro documento HTML, todo ello desde un lenguaje como JavaScript.

A modo de ejemplo, supongamos que nuestro documento contiene un párrafo con identificador “prueba” como el ejemplificado en la siguiente etiqueta HTML:

```
<p id="prueba">...</p>
```

Para hacer referencia a dicho párrafo podríamos hacerlo de la siguiente forma:

```
Document.div["prueba"]
```

O mediante el método `getElementByID(id)` del objeto `Document` pasándole como parámetro el identificador de la etiqueta.

¿Dónde almacenamos la información representada por los elementos del documento?

Si bien podríamos haber usado el propio documento HTML como contenedor de información adicional, relativa al XML final del formulario que tendrá que generar nuestra aplicación, ocultándosela al usuario, lo más razonable será envolver la información en objetos y separarla de los elementos visuales de la interfaz.

Para ello, durante el proceso de modelado dispondremos, además de los elementos visuales que verá el usuario y que mostrará el navegador web, de una jerarquía de instancias de objetos JavaScript, quienes serán realmente los contenedores de la información relevante de los distintos componentes de un formulario y con los que, finalmente, generaremos el XML resultante de la etapa de modelado.

A la vez que se realiza un cambio en la interfaz (añadir un nuevo campo, eliminar un campo existente, modificar una etiqueta o añadir una propiedad) se reflejará también, de forma totalmente transparente al usuario, en el objeto correspondiente en memoria que representa al elemento.

Añadiendo para cada objeto un método que lo traduzca a su representación en XML logramos simplificar enormemente el proceso de envío del formulario. Siguiendo esta filosofía sólo deberíamos llamar al método `toXML()` del formulario y éste se encargaría de recorrer sus elementos hijos y llamar a su vez a los métodos de éstos.

Si no lo hiciéramos así, nos veríamos obligados a volver a leer el documento generado por la aplicación para poder generar, a partir de sus elementos, el XML resultante. Con la complejidad añadida que desarrollar un nuevo *parser* podría suponer, así como el incremento en el tiempo de envío y dejando a un lado los beneficios propios de la programación orientada a objetos y de la encapsulación.

Diagrama de clases de la aplicación

Para gestionar la estructura de objetos de la que hemos hablado en los párrafos anteriores y que almacenará la información del formulario modelado, hemos desarrollado una estructura de clases que cumple con los requisitos marcados.

A continuación, se muestra el diagrama de clases generado, siguiendo la notación especificada en UML:

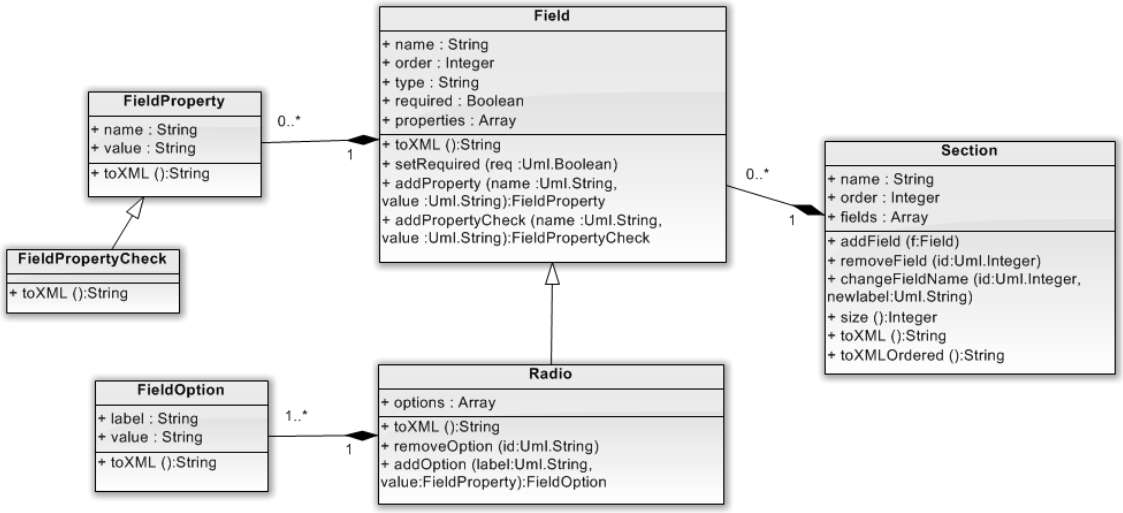


Figura 4-7 Diagrama de clases del modelador.

Como muestra el diagrama, no aparece el objeto Formulario pues no habrá más de un formulario a la vez, sólo el que se está modelando.

En su lugar, existe un vector de objetos **`Section`** representando las secciones del formulario actual. Los objetos `Section` disponen de un atributo para su nombre (*`name`*), otro atributo para su número de orden (*`order`*) dentro del formulario y un vector de campos (*`fields`*). Además, dispone de los métodos:

- *`addField`*: Añadir un nuevo campo a la sección.
- *`removeField`*: Borrar un campo de la sección.
- *`changeFieldName`*: Modificar el nombre de un campo de la sección.
- *`size`*: Devuelve el número de campos de una sección.
- *`toXML`*: Devuelve el XML de la sección.
- *`toXMLOrdered`*: Devuelve el XML de la sección con los campos ordenados.

El atributo *`fields`* contiene un vector de objetos **`Field`**. Los objetos `Field` representarán los campos de un formulario. Poseen un atributo *`name`* que almacenará el nombre del campo, *`order`* que identificará el número de orden del campo dentro de la sección, *`type`* identificando el tipo del campo, *`required`* para especificar si se trata de un campo

requerido o no (a la hora de rellenarlo) y un array *properties* de propiedades. También se facilitan los métodos:

- `toXML`: Que retornará el XML del campo.
- `setRequired`: Permite fijar si se trata o no de un campo requerido.
- `addProperty`: Añadir una nueva propiedad al campo.
- `addPropertyCheck`: Añadir una nueva propiedad de tipo checkbox a un campo.

Las propiedades de un campo vienen especificadas por la clase **`FieldProperty`**, que contendrá un nombre o etiqueta de la propiedad (*name*), así como el valor de la misma (*value*). El único método del que dispone es *toXML* que nos devolverá su traducción a XML.

Como caso especial de propiedad, tenemos las propiedades de tipo *check*. Los valores posibles para estas propiedades son: verdadero o falso. Para representarlas disponemos de la clase **`FieldPropertyCheck`** que hereda de la clase `FieldProperty` sobrescribiendo el método `toXML` para generar el XML correspondiente a una propiedad de este tipo.

Como caso especial de campo, tenemos la clase **`Radio`** que extiende de la clase `Field`, añadiendo un vector *options* de opciones. Representa un campo cuyos valores posibles a tomar, a la hora de rellenar el formulario, son fijos. También dispone de los métodos:

- *toXML*: Sobrescribe el método de la clase padre para permitir añadir al XML la traducción de sus opciones.
- *removeOption*: Nos permite eliminar una opción del campo.
- *addOption*: Añadir una opción al campo.

Los objetos Radio dispondrán de un vector `options` que contendrá objetos del tipo **`FieldOptions`** y que representarán los valores posibles a tomar por un campo de tipo `Radio`. En ellos almacenaremos su etiqueta o el valor a mostrar (*label*) y el valor real de la opción (*value*). Dispone de un método `toXML` que servirá para devolvernos la representación XML de la opción.

OOP en JavaScript

A pesar de que JavaScript sea un lenguaje de *scripting*, se puede emular el paradigma OOP [60] (Object-Oriented Programming) mediante mecanismos como los prototipos. Mientras que, por ejemplo, JAVA es un lenguaje basado en clases, JavaScript es un lenguaje basado en prototipos. En JavaScript no existe la diferencia entre la clase y la instancia que habitualmente aparece en los lenguajes de programación orientados a objetos, el prototipo no es más que un objeto abstracto, aunque se suele decir que una instancia es un objeto creado a partir de un constructor determinado de un prototipo. Además, un objeto puede heredar métodos y propiedades de otro objeto por medio de un prototipo constructor que genere nuevos objetos.

Para definir una clase, existe más de una forma de declararla sintácticamente hablando. Nosotros hemos elegido aquella que más similitud posee con los lenguajes de OOP como JAVA. Una definición de una clase podría quedar como sigue:

```
/**
 * Clase Ejemplo
 */
functionEjemplo(varConstructor){
        this.atributo = varConstructor;
        this.metodo = function(){
                /* Código */
                return "Hola " + this.atributo;
        }
}
```

Código 4-2 Ejemplo de definición de clase en JavaScript.

Como vemos, la clase se declara como una función que recibe como parámetros los argumentos del constructor, además de tener atributos y métodos propios. Nuestra clase de ejemplo se instanciaría de la siguiente forma:

```
var instancia = new Ejemplo("Mundo!");
/* Llamada a un método */
instancia.metodo();
```

Código 4-3 Ejemplo de instanciación de una clase en JavaScript.

La herencia se consigue mediante *prototypal inheritance*, es decir, se asigna como prototipo del nuevo objeto un objeto de la clase padre y a éste último se le modifican los atributos o métodos. Con este mecanismo conseguiremos herencia simple, para obtener herencia múltiple deberemos simularla con mecanismos más complejos.

Durante el desarrollo de los modelos, y para evitar añadir siempre las mismas líneas en aquellos modelos que hereden de otros, hemos decidido incluir una función que, tras la definición de la clase, indica que esa clase hereda de otra.

```
/**
 * Clase Padre
 */
function Padre(){
        this.atributo = "Mundo!";
        this.metodo = function(){
                /* Código */
                Return "Hola " + this.atributo;
        }
}
function Hija(varConstructor){
        this.atributo2 = varConstructor;
}
// Herencia
extend(Padre,Hija);
```

Código 4-4 Ejemplo de mecanismo de herencia.

Mediante la llamada a la función `extend`, que hemos declarado previa a la definición de nuestros modelos, le indicamos que la clase `Hija` extenderá a la clase `Padre`. El cuerpo de esta función es el siguiente:

```
// Inheritance
function surrogateCtor(){}
function extend(base, sub){
        surrogateCtor.prototype = base.prototype;
        sub.prototype = new surrogateCtor();
        sub.prototype.constructor = sub;
}
```

Código 4-5 Mecanismo de herencia en objetos JavaScript.

En primer lugar deberemos crear un objeto de la clase padre, porque en JavaScript sólo podemos igualar el prototipo de una clase a un objeto de otra clase y no a un prototipo, para que la primera herede de la segunda. Realizamos un nuevo objeto con la sentencia `new surrogateCtor()` y se lo igualamos al prototipo de nuestra clase hija. Por último, y para poder llamar al constructor del padre, igualamos el constructor del prototipo de la clase hija al de la clase padre.

Maquetación, estructuración y plantillas

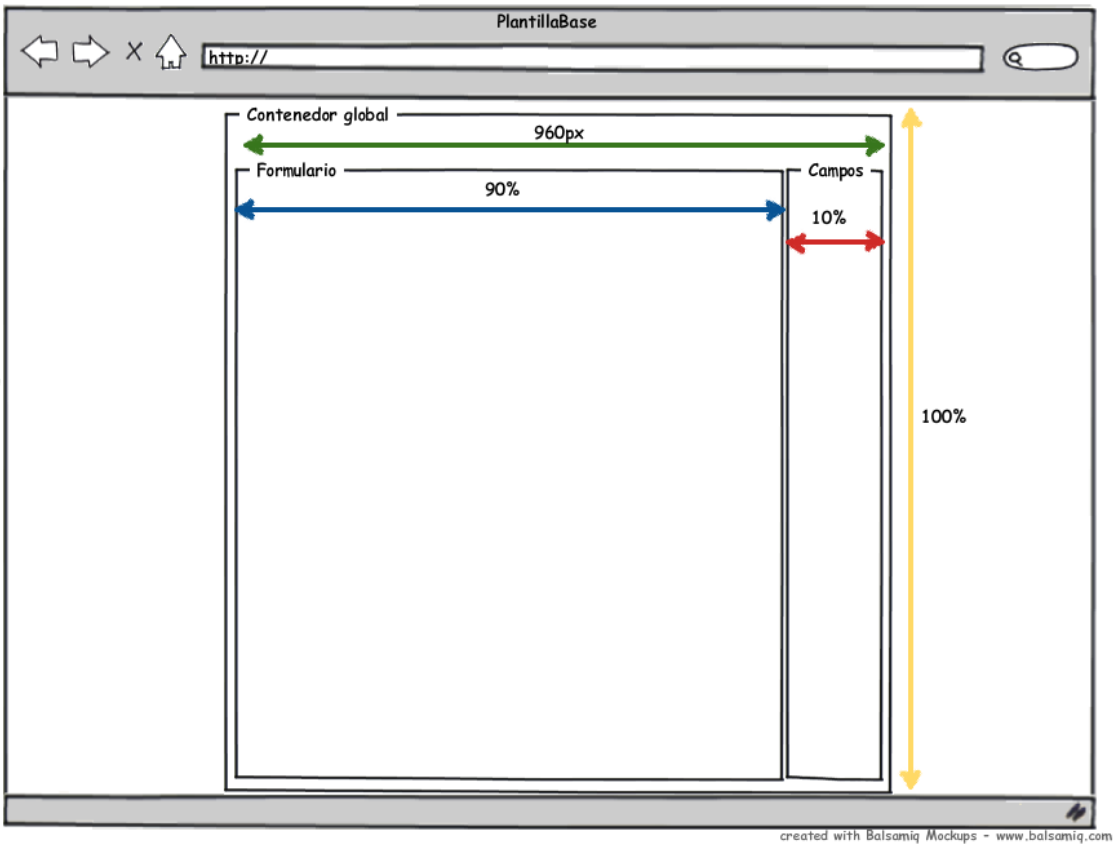


Figura 4-8 Plantilla base del módulo modelador.

Para dotar a nuestra página web de una interfaz que la distinga de una página web corriente, le dé aspecto de aplicación y siga los requisitos marcados durante la etapa de diseño, realizamos una plantilla básica en HTML con las secciones representadas en la Figura 4-8.

En primer lugar tendremos un contenedor centrado en la ventana del navegador y de un tamaño de 960 píxeles de ancho y el cien por ciento de la altura. Dentro hemos dispuesto un contenedor a la izquierda que ocupa el 90 por ciento del primero (864 píxeles) y el resto, el 10 por ciento (96 píxeles), lo ocupará el panel lateral derecho.

Para conseguir esta disposición así como para decorar la información y estructurarla, hemos hecho uso de hojas de estilo en cascada o CSS (*Cascading Style Sheets*), lenguaje diseñado y desarrollado por el *World Wide Web Consortium*. Separando la estructura del documento de la presentación de su contenido. Generando una compleja hoja de estilos en un fichero independiente.

Tras definir la plantilla base con los bloques comentados anteriormente, así como otros bloques que no detallaremos, hemos utilizado el mecanismo de plantillas provisto por Django[61] para fijar dicha plantilla como plantilla base sobre la que extender el resto de documentos HTML de la aplicación.

Por ejemplo, sobre la plantilla base definimos el bloque de contenido de la sección referente al formulario, siguiendo el lenguaje de plantillas de Django, de la siguiente forma:

```
<div id="containt">
    <div id="form">
        {% block mainframe %}
        {% endblock %}
    </div>
    ...
</div>
```

Código 4-6 Detalle de la plantilla “base.html” del modelador.

Y en el documento que extenderá a esta plantilla, para incorporar los elementos de la pantalla de creación de un nuevo formulario insertaríamos el contenido de ese bloque de la siguiente forma:

```
{% extends "base.html" %}
{% block mainframe %}
        <h1 class="formname">Nombre del formulario</h1>
        <!--Resto del contenido-->
        ...
{% endblock %}
```

Código 4-7 Detalle de la plantilla “create.html” del modelador.

Django, al presentar la página final al usuario, generará un documento HTML mezclando los códigos de ambas plantillas. Sobre el bloque `mainframe` desplegaremos dinámicamente, con el uso de JavaScript, los campos del formulario que estemos modelando.

¿Cómo implementamos el drag & drop?

Como hemos comentado, haremos uso del nuevo atributo definido en la especificación 5 de HTML: draggable. Para ello necesitamos identificar los tres elementos fundamentales que intervienen en el proceso: algo que arrastrar, un área sobre la que soltar y una serie de manejadores de dichos eventos escritos en JavaScript.

Con HTML5, tan sólo estableciendo el atributo `draggable` de un elemento al valor "true" estaremos indicando al navegador que renderiza la web, que se trata de un elemento que podremos arrastrar. Es el propio navegador quien nos facilitará dicha tarea. Por ejemplo, si fijásemos el atributo `draggable` de una etiqueta de una imagen con el valor "true", podríamos, sin más, arrastrar la imagen por el documento. Este comportamiento, que era el comportamiento por defecto para el *tag* `<img/>` en algunos navegadores, ahora es fácilmente manipulable y se puede incluir a cualquier elemento de nuestro documento.

Como se ve en los bocetos desarrollados durante la etapa de diseño (ver 4.3Etapa de diseño), nuestra aplicación dispondrá de dos áreas principales, la primera donde se encuentran los tipos de campos (que arrastrar o con los que hacer "*drag*") y la segunda el área sobre la que se formará el formulario (donde soltar o hacer "*drop*"). Dichos elementos se encuentran definidos como elementos `div` de nuestro documento HTML principal y son identificados mediante dos identificadores únicos, de manera que puedan ser seleccionados de entre todos los elementos que componen el documento, como ya hemos comentado en la sección de introducción a DOM.

Figura 4-9 Elementos arrastrables y zonas "drop" de la interfaz del modelador.

Tal y como se representa en la Figura 4-9, existen varios elementos, que pueden ser arrastrados, situados en el lateral derecho de la interfaz de la aplicación que se corresponderán con los tipos de campos permitidos en el proceso de modelado de un formulario. Estos elementos podrán "soltarse" en dos tipos de áreas diferenciadas: la primera será un elemento sección, caso en el que el campo generado se añadirá como el último campo de la sección; la

segunda será otro elemento campo. De soltarlo sobre este último, el nuevo campo se insertará en la posición inmediatamente superior a ese campo.

Para ello se han desarrollado dos manejadores o *handlers* del evento `drop` para los elementos de tipo sección y tipo campo. Su intención es la misma y su cuerpo es similar, salvando las diferencias de que, en el caso de hacer "*drop*" sobre un campo, los elementos se hallan en niveles de jerarquía distintos.

Para los elementos *arrastrables* será necesario implementar una función que se encargue de manejar el evento `dragstart`. Adicionalmente, manejaremos el evento `dragend` para eliminar los cambios realizados en los estilos del elemento realizados por el manejador del evento `dragstart`.

Dado que hemos decidido implementar nuestro propia versión *drag & drop* con HTML5, en lugar de utilizar alguna librería externa de entre las múltiples disponibles en Internet, explicaremos a continuación el código desarrollado para cumplir con este requisito de la aplicación modeladora de formularios.

```
var myfields = document.querySelectorAll('#fieldsBar > ul > li > .item');
$('#fieldsBar > div.buttons').disableSelection();
for(var i = 0; i < myfields.length; i++){
  var actualField = myfields[i];
  // DRAGSTART
  addEvent(actualField,'dragstart',function(e){
     e.dataTransfer.setData('text',this.id);
     this.style.backgroundColor = '#C9676C';
     e.dataTransfer.setDragImage(this.getElementsByTagName("img")[0],10,10);
  });
  // DRAGEND
  addEvent(actualField,'dragend',function(e){
     this.style.removeProperty("background-color");
  });
  // HOVER
  $(actualField).hover(
     function(){ $('div.label',this).fadeIn();},
     function(){ $('div.label',this).fadeOut();}
  );
}
```

Código 4-8 Manejadores de los eventos `dragstart` y `dragend`.

```
// DROP
addEvent(section,'drop',function(evt){
    if(evt.stopPropagation) evt.stopPropagation();
    // Obtenemos el ID transferido en el DRAG
    var idDrag = evt.dataTransfer.getData('text');
    var item = $('#'+ idDrag);
    if((typeof item !="undefined")&&(item.length >0)){
        // Listado de campos del formulario
        var lis = $('li',this);
        // Variables para el nuevo campo
        var fieldName =  $('p:first', item).text();
        var type =  $('div.type', item).text();
        var fieldType =  $('div.fieldtype', item).text();
        var sectionId = parseInt((this.id).match(/\d+/));
```

```
        var id = parseInt(formSections[sectionId].size());
        // Creamos el nuevo campo
        var newField = createNewField(id, fieldName,sectionId, idDrag,
                                      type, fieldType);
        newField.appendTo(this);
    }
    return false;
});
```

Código 4-9 Manejador del evento `drop` para una sección.

```
// DROP
addEvent(field,'drop',function(evt){
    ...
    if((typeof item != "undefined")&&(item.length >0)){
        var lis = $('li',this.parentNode);
        ...
        var sectionId = parseInt((this.parentNode.id).match(/\d+/));
        var id = parseInt(formSections[sectionId].size());
        ...
        // Agregamos el campo al formulario
        newField.insertBefore(this);
    }
    return false;
});
```

Código 4-10 Extracto del manejador del evento `drop` para un campo.

Tratando de lograr la mayor compatibilidad entre navegadores, se ha añadido la función `addEvent`, que se encargará de añadir las funciones a la escucha de los eventos, adecuando el proceso según el navegador sobre el que se esté ejecutando la aplicación

En la tabla Código 4-8 vemos el fragmento que se encargará de detectar todos los elementos del panel lateral derecho que podrán ser arrastrados a las distintas zonas de las que hemos hablado y les asignará las funciones que manejan los eventos en el momento en que estos ocurran. Dichos manejadores de eventos se han declarado como funciones anónimas en la propia llamada a la función `addEvent` del evento correspondiente.

En primer lugar obtenemos todos los elementos en un listado y procedemos a iterar sobre ellos, añadiendo a cada campo el manejador del evento `dragstart`, que se lanza en el momento en que se pulsa sobre el elemento y se empieza a arrastrar. Dentro de la función, se fija el identificador del tipo de campo que estamos arrastrando como el dato a transferir en el evento hasta la zona donde soltar. Además, se establece un color de fondo para el elemento de forma que éste quede destacado en el panel derecho durante el desplazamiento. Finalmente se fija la imagen que permanecerá bajo el cursor mientras éste se desplaza. Por defecto en algunos navegadores esta imagen se generará a partir de todo el elemento, mientras que en otros navegadores será una imagen por defecto; en nuestro caso la fijaremos a la imagen del tipo de campo.

Una vez se suelta el elemento, se lanza el evento `dragend` con cuyo manejador eliminaremos el fondo de color que previamente habíamos establecido.

Por último, aparecen unas líneas llamando a métodos de las librerías jQuery que mostrarán una animación en el momento en que el cursor pase sobre alguno de los elementos del panel lateral derecho. En este caso se mostrará la etiqueta con el nombre del tipo de campo en un rectángulo negro semitransparente a un lado de la imagen, con un efecto de aparecer y desaparecer.

Las tablas Código 4-9 y Código 4-10 muestran los manejadores para el evento `drop` de la sección y el campo respectivamente, en este último se muestran únicamente los cambios con respecto al primero. Las principales diferencias entre ambos códigos radican primero en el nivel dentro de la estructura del documento en la que se encuentran (uno es una sección y el otro un campo dentro de una sección) y segundo en el comportamiento que tiene lugar cuando se suelta sobre ellos un nuevo campo. Mientras que al soltar en la sección el campo se añadirá al final, al soltar sobre un campo se añadirá justo encima de éste.

En ambos códigos hacemos uso de los selectores que nos proporciona jQuery (función `$("")`) para obtener los diferentes elementos, así como los datos necesarios para crear el objeto que representa al nuevo campo. Estos últimos datos están ocultos dentro del código HTML de los elementos que arrastramos desde el lateral derecho:

```
<li>
<a class="item" href="#" id="campo1" draggable="true">
    <div class="type">text</div>
    <div class="fieldtype">TEXT</div>
    <img src="images/fields/tw_icon_text.png"/>
    <div class="label">
       <p><strong>Campo de texto</strong></p>
    </div>
</a>
</li>
```

Código 4-11 Ejemplo de tipo de campo a arrastrar desde el lateral derecho del modelador.

Una vez recopilada toda la información necesaria, llamaremos a otra función que hemos definido con el nombre de `createNewField`, que recibe como parámetros el identificador del nuevo campo dentro de la sección, su nombre, la sección a la que se añade, el identificador del elemento arrastrado, el tipo de campo con el que será traducido y el tipo de campo con el que será visualizado (estos dos últimos generalmente coinciden).

```
/*****************************************************************/
/* Crea un nuevo campo para el formulario                        */
/*****************************************************************/
function createNewField(id, name, section, idDrag, type, fieldType){
   ...
}
```

Código 4-12 Detalle de la declaración de la función createNewField.

La función `createNewField` es probablemente una de las más complejas dentro del código JavaScript programado, generará toda la jerarquía de etiquetas necesaria para representar en el navegador un campo, así como preparará las acciones disponibles dentro de

dicho elemento: etiqueta editable al hacer *click*, botón de borrado, botón de campo requerido, despliegue de propiedades, despliegue de opciones… También generará un identificador del tipo `sXfY` en donde "X" corresponderá con el identificador de la sección e "Y" con el identificador del campo, de forma que sólo con el identificador del campo podamos saber en qué sección se encuentra y dentro de la misma cuál es su posición. Además, generará el objeto campo pertinente para añadirlo al objeto sección sobre el que se crea.

Hemos decidido añadir a los diferentes elementos que iremos generando durante el modelado identificadores compuestos de forma que, realizando un pequeño análisis sobre ellos, podamos saber donde se encuentra exactamente el elemento. Por ejemplo, el campo `s1f2` será el segundo campo de la primera sección, mientras que el identificador `s1f2p1` se correspondería a su primera propiedad y `s1f2o1` a su primera opción.

Scrolling automático

Tratando de facilitar el modelado de formularios, hemos implementado el *scrolling* automático del contenedor del formulario de forma que al arrastrar un campo sobre una sección el nuevo campo generado se sitúe en el primer cuarto de la pantalla de edición del formulario de forma automática y con una animación de desplazamiento suave.

Para lograrlo hemos calculado la posición del nuevo elemento sobre el contenedor del formulario y hemos animado el cambio de la propiedad `scrollTop` del elemento, mediante la función `animate()`, aplicada al contenedor del formulario.

Ordenación y borrado de campos

Para permitir reordenar los campos de una sección de un formulario, hemos utilizado la función `sortable` presente en jQuery UI, la librería para interfaces de usuario de jQuery.

El usuario podrá arrastrar un campo dentro de una sección y llevarlo sobre cualquier otro campo de la sección, fuese cual fuese la posición sobre la que se hubiese insertado. Esto no modificará el identificador de dicho campo que continuará siendo el mismo, independientemente de la posición.

```
$(section).sortable({items:"li"});
$(section).disableSelection();
```

Código 4-13 Extracto de código marcando los campos dentro de secciones como ordenables.

Tan sólo con las líneas del Código 4-13 logramos que los elementos de nuestro listado de campos sean editables. Le indicamos que, sobre la sección, haga ordenables los elementos "
" de la misma y, además, marcamos los textos de la sección como no seleccionables de forma que durante el proceso de reordenación el cursor del usuario no pueda, por error, seleccionar el texto de la misma.

Para eliminar un campo, de forma similar a eliminar una opción de un campo, realizaremos un borrado del vector de campos de la sección, quedando este elemento del vector vacío hasta el final del proceso de modelado. Esto se tendrá en cuenta a la hora de recorrer los campos, comprobando primero que el valor esté definido.

Etiquetas editables

Para permitir que al pulsar sobre un texto éste sea editable, ya sea una etiqueta de un campo, de una sección, el nombre de un formulario, las etiquetas y valores de las opciones o los valores de las propiedades; hemos utilizado el *plugin* de edición en línea jQuery *Inline/In-Place Edit Plugin* [62]. Nos permitirá añadir dicha funcionalidad de forma que podamos tener un prototipo funcional y usable, si bien en un futuro requerirá de algunas modificaciones que mejoren la relación con el usuario.

Para marcar un párrafo (`<p/>`) como editable, sólo necesitaremos llamar desde el elemento al método `inlineEdit()`. Adicionalmente, y dada nuestra implementación, necesitaremos pasarle como parámetro un diccionario en donde indiquemos que tras la acción de guardar (*save*) necesitaremos obtener el nuevo valor del texto para actualizarlo correctamente en el objeto de nuestro formulario correspondiente. Con ese cometido añadiremos una función manejadora que se lanzará tras la edición (*callback)*.

Acceso rápido a secciones

Al añadir una nueva sección al formulario, se genera un enlace en la barra inferior para acceder más rápidamente a ella. Al pulsar sobre el enlace se centra el contenedor del formulario sobre dicha sección y el borde de la misma permanece en rojo durante un breve instante de tiempo.

Con esta funcionalidad añadida logramos facilitar y agilizar el modelado de formularios con múltiples secciones.

Generación del XML y conexión con el repositorio (Beekeeper)

Como ya hemos comentado, el hecho de haber realizado una programación orientada a objetos simplificará mucho la generación del XML correspondiente al formulario modelado. Al pulsar el botón enviar del formulario, tan sólo deberemos recorrer el vector de secciones y llamar, para cada una, a su método `toXML()`. Éste, a su vez, llamará al método `toXML()` de sus campos y éstos a los de sus propiedades y/u opciones. Con lo cual, nosotros sólo nos deberemos encargar de escribir correctamente el método `toXML()` para cada uno de los modelos de forma que traduzca cada elemento al XML que definamos en el entorno global del proyecto.

Surge un problema a raíz de la posible reordenación de campos dentro de una sección. Para evitar reordenar los elementos del vector de campos de una sección cada vez que se reordene un campo de la misma, no realizamos modificación alguna cuando un elemento cambia de posición sino que, durante la traducción, generamos el XML siguiendo la jerarquía

final de los elementos del documento, con la ayuda de DOM. Es decir, en lugar de recorrer el vector desde el elemento cero hasta su tamaño total, recorremos los elementos del contenedor de la sección del documento HTML y con su identificador iremos extrayendo los elementos del vector de campos de la sección.

Una vez generado el XML, a la hora de comunicarnos con los servicios webs de Beekeeper, crearemos un objeto cliente del módulo de Python Suds. Codificaremos el XML del formulario a "*url safe base64 encode"* y realizaremos la llamada correspondiente al servicio `upload_new_form` del servidor del repositorio.

```
from suds.client importClient
from base64 import urlsafe_b64encode as sb64encode
...
formXMLb64 = sb64encode(formXML)
myclient = Client(url, cache=None)
returnvalue = myclient.service.upload_new_form(formXMLb64)
...
```

Código 4-14 Ejemplo de creación de cliente y llamada al servicio web.

En caso de que la respuesta sea correcta, se le mostrará al usuario una ventana flotante con un mensaje de éxito; de no ser así, aparecerá el XML del formulario generado y un mensaje de error.

Organización del código

Para terminar con la etapa de implementación, comentaremos la estructura en la que hemos dividido el desarrollo. Hemos estructurado el código JavaScript en cuatro ficheros bajo la URL "/modeler/js":

- *dragdrop.js*: contiene el código encargado de implementar el "arrastrar y soltar" ya comentado.
- *models.js*: en él se definen nuestras clases JavaScript.
- *fillform.js*: es el encargado de generar y gestionar tanto los elementos del documento HTML como los objetos de nuestra jerarquía de clases.
- *parseform.js*: aquí se encuentran las funciones encargadas de realizar la traducción final de los objetos instanciados al XML correspondiente.

Las librerías externas están incluidas en la dirección "/modeler/js/lib" de la aplicación, de manera local, para evitar depender de la disponibilidad de un servidor ajeno a nuestro desarrollo.

El código correspondiente a la gestión de los servicios webs se encuentra en una aplicación Django con nombre “ws_client”, separada de nuestra aplicación “modeler” pero perteneciendo al mismo proyecto Seed de Turawet.

Por otro lado, las imágenes de las plantillas, generadas casi al completo por nosotros, se encuentran divididas en los siguientes enlaces relativos a “/modeler/”:

- “images/buttons”: aquí se encuentran las imágenes de los botones de las acciones a realizar sobre los campos, presentes en el lateral superior derecho de cada campo.
- “images/fields”: bajo esta dirección aparecen las imágenes que representan los diferentes tipos de campos disponibles desde el modelador.
- “images”: aquí se encuentran el resto de imágenes comunes a toda la aplicación (el logo del proyecto, etc.).

4.4.3 Prototipo

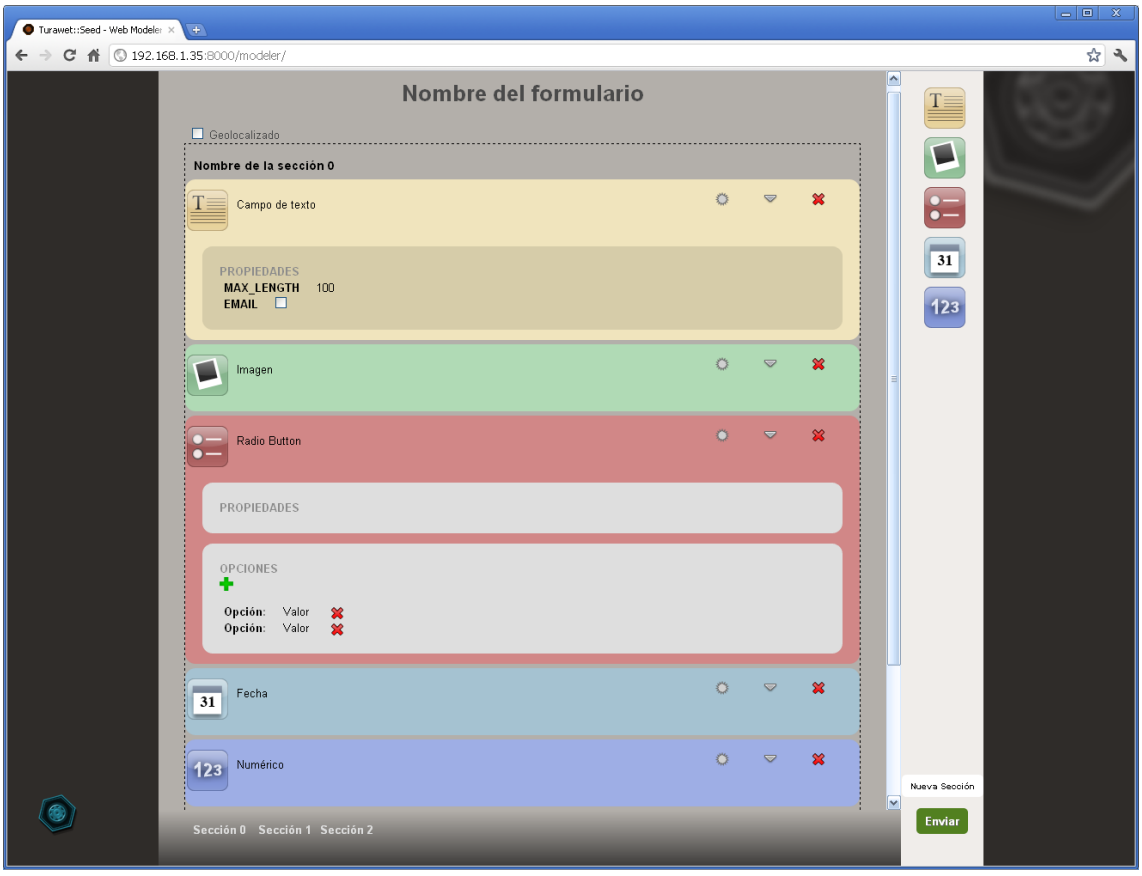


Figura 4-10 Imagen del modelador web, primer prototipo.

En la Figura 4-10 se muestra la pantalla del prototipo generado tras la fase de implementación. En ella aparecen varios campos insertados en la primera sección de un formulario de ejemplo, durante el proceso de modelado.

Ampliaremos la información sobre el prototipo que hemos desarrollado en el Capítulo 7.

Capítulo 5. El recolector

Resumen:

- En este capítulo se describe la aplicación móvil de recolección de datos.
- Se repasan las diferentes fases de desarrollo: análisis, diseño e implementación.
- En la etapa de diseño se ilustra al lector con diversos diagramas UML (diagrama de casos de uso y de secuencias) que preparamos con el fin de plasmar nuestras ideas de la aplicación. Asimismo se describen los *mockups* que preparamos para diseñar la interfaz.

5.1 Introducción

En este capítulo, hablaremos del segundo componente del proyecto Turawet, el módulo BeeDroid (de aquí en adelante Recolector). Nos centraremos en la aplicación que hemos desarrollado para la entrega del proyecto de fin de carrera, que consta de una aplicación móvil para dispositivos Android. Es importante tener en cuenta que desde un principio, nuestra idea fue contar con otros recolectores, que abran el abanico de posibilidades a los usuarios finales. Por enumerarlos de alguna forma, tenemos planeado la implementación de un recolector web, que permita rellenar formularios desde un navegador, con la comodidad que eso conlleva, otro desarrollo para dispositivos iOS y posiblemente Windows Phone 7 [63].

El recolector será el encargado de almacenar los formularios que el usuario vaya a utilizar y le va a permitir rellenar instancias de estos. Si buscamos un símil con una metodología de recolección de datos en la actualidad, podríamos ver a esta aplicación como una especie de carpeta o portafolio, en donde habría muchas copias de varios tipos de formularios, y con los que el usuario podría recolectar datos de cualquier índole. Las ventajas de nuestra aplicación son muchas: para empezar estos formularios son electrónicos, no hay papel de por medio; se pueden actualizar en cualquier momento y casi desde cualquier sitio; las instancias se pueden enviar a un repositorio central de forma casi inmediata; existe la posibilidad de recolectar datos multimedia como imágenes o vídeos, cosa imposible con formularios tradicionales; entre otras muchas más.

En los siguientes apartados, comentaremos los aspectos más importantes de esta aplicación. Empezaremos por una primera etapa, en donde realizamos un análisis de los requisitos funcionales que queríamos cumplir. Luego hablaremos de todas las decisiones de diseño que tomamos para conseguir un modelo robusto, que fuese fiel al análisis previo y que estuviese muy ligado a las tecnologías que íbamos a utilizar. Por último, comentaremos como hemos llevado a cabo el desarrollo de este módulo, los inconvenientes que se nos antepusieron y la forma de superarlos, entre otras muchas cosas.

5.2 Etapa de análisis

Esta es la primera etapa del desarrollo de la aplicación móvil. En ella, analizaremos de qué forma comenzamos a construir la aplicación desde cero, teniendo en cuenta todos los requisitos y funcionalidades que nos habíamos planteado, así como los requisitos intrínsecos de la plataforma Turawet, que nos vienen heredados debido a la interacción con los otros módulos.

La primera idea que tenemos en mente, es que con esta aplicación cualquier usuario pueda rellenar un formulario en cualquier momento y lugar. Que no haga falta que cuenta con un dispositivo físico específico o de elevado coste, sino todo lo contrario; y que además le fuese lo muy familiar. El claro ejemplo es un teléfono móvil.

La idea estaba clara, teníamos que crear una aplicación móvil, que el usuario se pueda descargar en su dispositivo y comenzar a utilizar desde el primer momento. Como se comentaba en la introducción, este módulo funciona como un portafolio de formularios que permite al usuario recolectar datos de cualquier índole y poder almacenarlos de forma sencilla, rápida y remota.

Para centrarnos en el contexto del proyecto global, esta aplicación será la utilizada por todos aquellos usuarios que tengan la necesidad de recolectar datos de algún tipo. Todos ellos podrán obtener los formularios que estén disponibles en el repositorio central, almacenarlos en su aparato móvil y comenzar a rellenar instancias. Recordamos que los formularios son definidos con la aplicación modeladora, de la cual hablamos en el capitulo anterior.

5.2.1 Requisitos

Algunos de los requisitos funcionales, surgieron de las reuniones que tuvimos con el personal de la Gerencia de Urbanismo de La Laguna o bien heredados de GeoBloc, aplicación la cual nombramos en el apartado referente al estado del arte. Aún así, la gran mayoría de estos, fueron definidos por nosotros, como si fuésemos nuestros propios clientes. Esto también nos ayudó a contar con una lista muy bien definida desde un comienzo y en donde sabíamos que no habría lugar a cambios posteriores. Si los enumeramos en forma de lista, tenemos los siguientes requisitos:

- Conectarse con el repositorio central.
- Obtener formularios que estén almacenados en él.
- Almacenar los formularios en cada dispositivo móvil para su posterior utilización.
- Mostrar de forma gráfica los campos de cada formulario para poder rellenar las instancias de la forma más cómoda posible.
- Permitir el almacenamiento local de las instancias, tanto de las que estén completas como de las que no.
- Permitir el envío de las instancias, tanto de las que estén completas como de las que no.
- Permitir que cada instancia sea capaz de contener información de la localización geográfica en donde fue completada.

- Almacenar metadatos, como la fecha de creación, modificación, el IMEI del dispositivo, entre otros.

A partir de la lista anterior, nos hicimos un esquema global de lo que necesitábamos incluir en la aplicación. El paso siguiente, fue diseñar un prototipo que siguiera las pautas que teníamos definidas.

5.2.2 Diagramas UML

En esta sección, comentaremos los distintos diagramas UML que hemos generado en la etapa de análisis.

Diagrama de casos de uso

El primer diagrama que presentamos, es el de casos de uso. El mismo, está orientado únicamente al recolector móvil. Se pueden contemplar todas las acciones significativas que son realizables con la aplicación.

Como es de esperar en este tipo de diagramas, primero se presentan todos los casos de uso en el escenario principal y luego se pasa a una explicación más detallada de cada uno de ellos. Para complementar esta información, queremos introducir brevemente a los actores.

Los actores, generalmente son las personas que interactúan con la aplicación y realizan acciones sobre ellas, utilizando las funcionalidades que esta les brinda. Para el caso del escenario principal del recolector, vamos a tener únicamente dos actores. Por un lado está el Recolector. Este puede ser cualquier persona que cumpla el papel de usuario de la aplicación. Será el encargado de recolectar los datos, rellenando instancias de formularios. Deberá estar dado de alta en la plataforma Turawet, para contar con un usuario y una contraseña que lo identifique unívocamente. Como ya comentábamos previamente en otros apartados, el usuario pertenecerá a un grupo de usuarios y tendrá acceso únicamente a los formularios de su grupo.

En el otro lado, tendremos al segundo actor participante de este escenario principal: el repositorio. Este es un actor especial, ya que no es una persona, sino más bien un sistema o aplicación. Cabe mencionar que se profundizará sobre él en el siguiente capítulo de esta memoria, por tanto no hablaremos demasiado de sus funcionalidades. Únicamente haremos mención de una de ellas, que son los servicios que ofrece.

El repositorio, entre otras cosas, pone a disposición de los recolectores móviles una gama de servicios, para que estos interactúen con el almacén de formularios e instancias que se han ido generando. Podríamos decir que este es un actor pasivo, pero aún así, lo consideramos fundamental en cuanto a la descripción de la aplicación móvil se refiere.

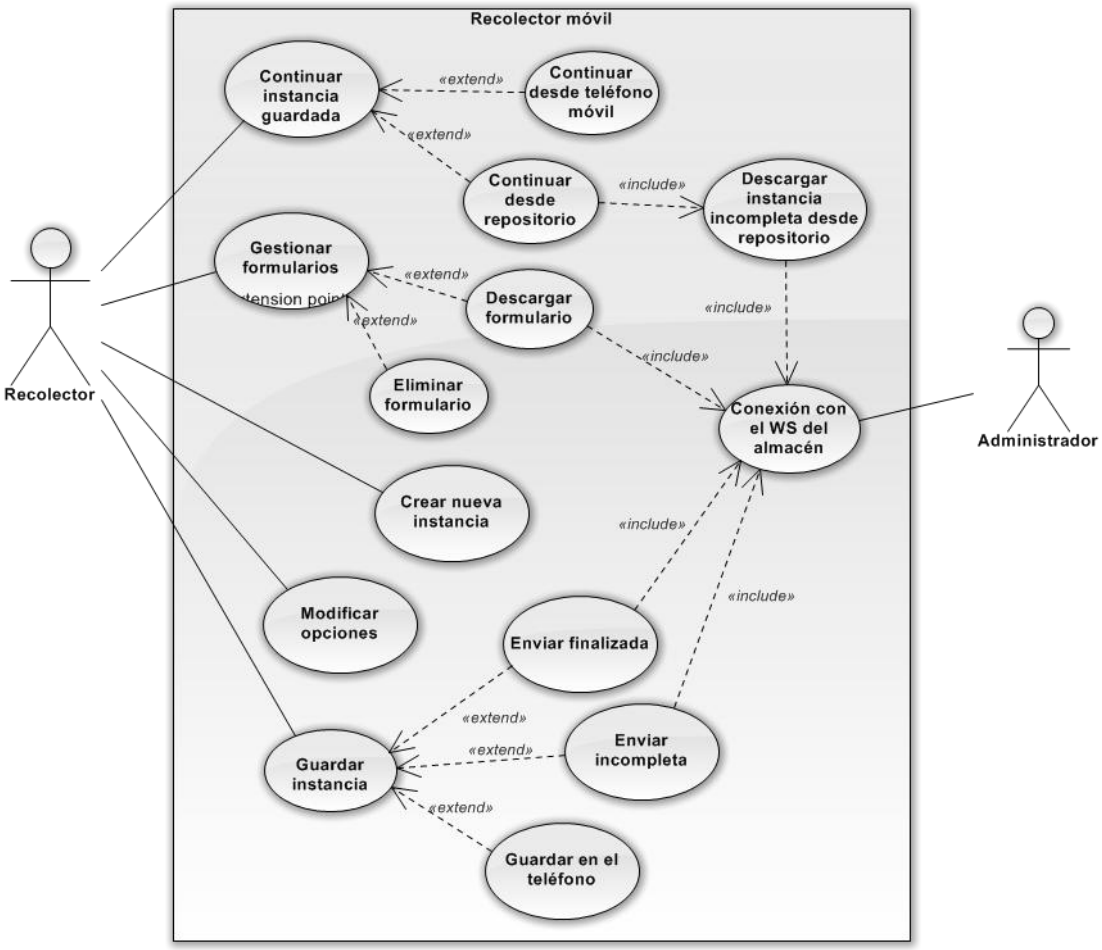


Figura 5-1 Diagrama de casos de uso del recolector.

Descripción de los casos de uso

A continuación explicaremos en profundidad los casos de uso más relevantes. El resto de ellos, se explicarán en el anexo final.

Identificador: CU-R-1

Nombre: Crear nueva instancia.

Descripción

El usuario recolector crea una nueva instancia de un formulario

Actores

Recolector.

Precondiciones

1. El usuario ha hecho *login* en la aplicación.
2. El usuario ha descargado la definición de un formulario.
3. El usuario ha seleccionado un formulario para completar.

Post-condiciones

1. Se crea una nueva instancia de un formulario.
2. Se han recogido distintos valores en la instancia.

Frecuencia

Muy frecuente

Flujo de escenario principal

1. El usuario accede a la aplicación, haciendo login, si es que no lo había hecho antes.
2. Comprueba que posee una copia del formulario que desea rellenar. De no ser así, la descarga.
3. Crea una nueva instancia del citado formulario.

Identificador: CU-R-2

Nombre: Enviar finalizada.

Descripción

El usuario envía una instancia finalizada al repositorio

Actores

Recolector.

Repositorio.

Precondiciones

1. El usuario ha creado una nueva instancia de un formulario.

Post-condiciones

1. La instancia es enviada y almacenada en el repositorio.

Frecuencia

Muy frecuente

Flujo de escenario principal

1. El usuario ha creado una nueva instancia de un formulario y la ha rellenado en su totalidad.
2. Selecciona la opción "Guardar y enviar" y la instancia se envía y guarda en el repositorio.

Identificador: CU-R-7

Nombre: Descargar formulario.

Descripción

El usuario se descarga una nueva definición de un formulario que está disponible en el repositorio.

Actores

Recolector.

Repositorio.

Precondiciones

1. El usuario se ha autenticado en la aplicación.
2. Se conecta al repositorio y recibe una lista de formularios que aún no posee.

Post-condiciones

1. Un nuevo formulario se almacena en el teléfono.

Frecuencia

Muy frecuente.

Flujo de escenario principal

1. El usuario se ha autenticado en la aplicación.
2. Se conecta con el repositorio y éste le envía una lista de formularios que no posee.
3. El usuario selecciona todos los formularios que desea descargar y elige la opción "Descargar formularios".
4. Los formularios son almacenados en el dispositivo móvil.

Diagrama de secuencia

Hasta ahora hemos venido explicando los casos de uso más importantes de la plataforma móvil. Hicimos una descripción detallada de los más importantes, teniendo en cuenta cuales eran las circunstancias que debían darse para que el usuario pudiese realizar las tareas que los conforman. Además, comentamos las consecuencias que producían cada una de esas tareas, de forma que teníamos un panorama general de la aplicación antes y después de ejecutar una acción concreta.

Para completar la visión del funcionamiento general esperado de la aplicación móvil, vamos a presentar el segundo diagrama UML que obtuvimos de la fase de diseño. Este es el ***diagrama de secuencia***. Este diagrama nos sirve para contemplar el flujo de información, peticiones y respuestas que existe entre todos los participantes y actores que vimos en el escenario principal. Por un lado tendremos al actor principal, que será el usuario de la aplicación recolectora. Este va a interactuar con el dispositivo móvil, haciendo peticiones y recibiendo respuestas en forma de pantallas, nuevas vistas, mensajes de confirmación, acceso a diferentes secciones, etc.

Los otros actores que forman parte del escenario no son usuarios, sino más bien sistemas o aplicaciones. Interactuando directamente con el usuario recolector, tenemos a la aplicación recolectora. Como se comentaba anteriormente, esta recibe las peticiones del usuario y reacciona frente a ellas. Además, es la encargada de comunicarse con el repositorio en los momentos que se requiera. El tercer actor implicado es este, el repositorio. Básicamente se encargará de atender las peticiones de la aplicación móvil y devolverle los resultados que espera. Recordemos que el repositorio cuenta con una cartera de servicios que las aplicaciones recolectoras consumen para obtener definiciones de formularios o enviar instancias.

El diagrama de secuencia que estamos presentando muestra el flujo de un escenario, donde el usuario recolector cumple todo el ciclo de recogida de datos para un formulario. Pueden existir muchos diagramas como este, uno para cada caso de uso, y ser todo lo detallado y complejo que se requiera. Nosotros hemos desarrollado únicamente este ya que cubre los principales aspectos de la aplicación y muestra el comportamiento esperado de todos los actores en una situación normal, donde no se producen errores.

Por otro lado, se contemplan los casos donde existan problemas de cualquier índole y el flujo varíe debido a ellos. No los hemos reflejado en forma de diagrama, simplemente porque son demasiados casos y no aportarían mucha más información. En el caso de que ocurra algún error inesperado, ya sea cuando se está descargando un formulario o enviando una instancia, la aplicación se comportará de la forma esperada, mostrando el correspondiente error de forma amigable al usuario y continuará con su funcionamiento normal.

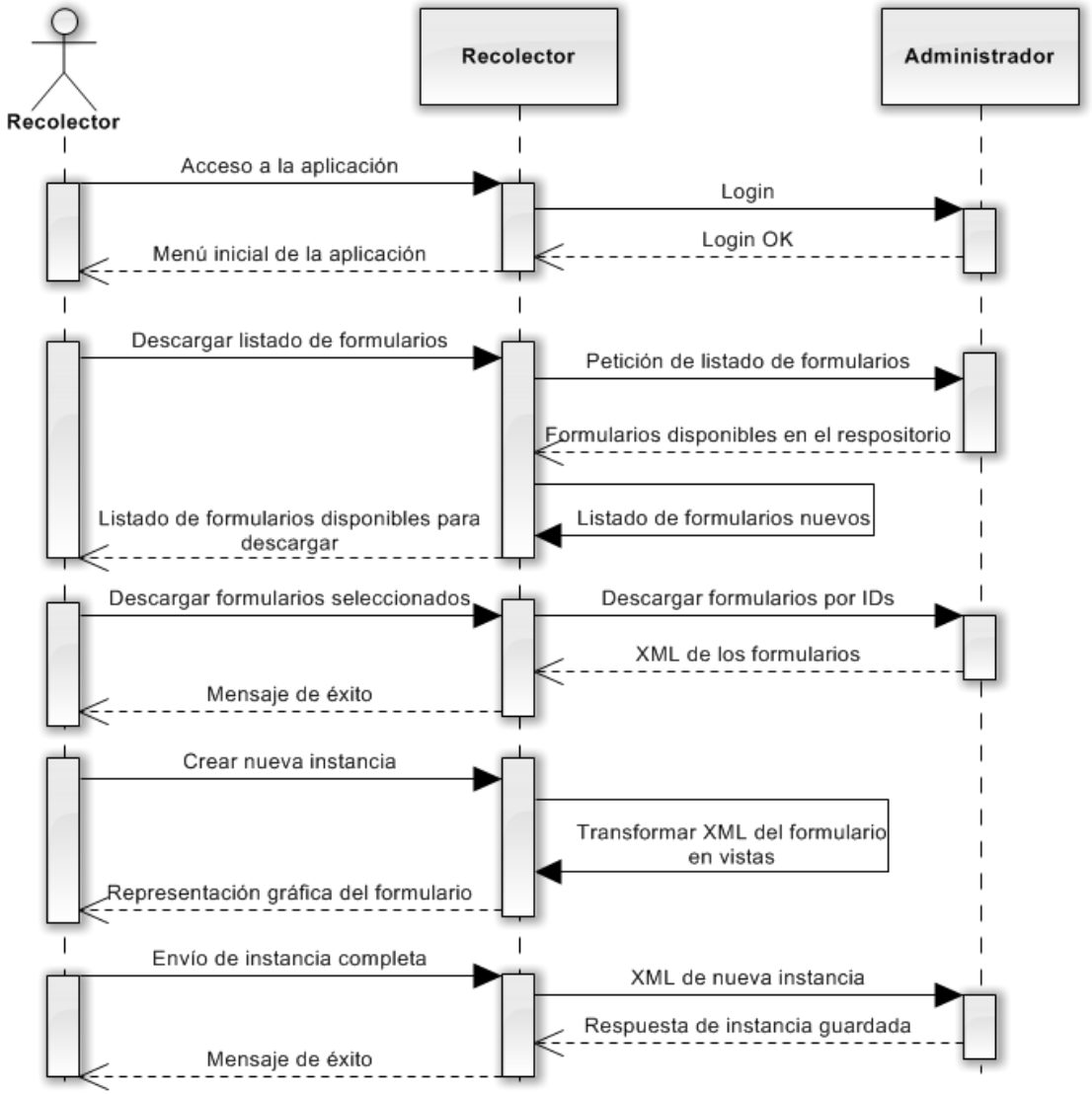


Figura 5-2 Diagrama de secuencia de la aplicación recolectora.

Centrándonos en el diagrama de secuencia, vemos el ciclo completo de uso de la aplicación, en un caso donde no se producen errores. Primeramente el usuario debe autenticarse en el sistema, por tanto, debe estar dado de alta en él. El encargado de rectificarlo es el repositorio, quien lleva la cuenta de los usuarios y sus permisos de acceso. Una vez hecho el *login*, el usuario desea obtener una lista de formularios que aún no posee. El recolector móvil solicita al repositorio todos los formularios disponibles, este responde con el listado correspondiente, y los formularios son mostrados al usuario para que seleccione cuales desea descargar. Es importante tener en cuenta que el repositorio responderá únicamente con una identificación de los formularios, es decir, con su nombre y versión. Cuando el usuario realiza su selección, solicita al recolector que descargue ese nuevo listado de formularios. Nuevamente, el recolector hace una solicitud al servidor, pero esta vez con el listado de formularios a descargar. El repositorio responderá con el listado de formularios completos, identificación más la definición de los mismos. Una vez que los obtiene, los almacena correspondientemente y delega el control al usuario. Este selecciona aquel que quiere

completar y comienza a crear la nueva instancia. Desde que la tiene completa, clama al recolector que la envíe al repositorio. Este hace lo propio y si todo funciona correctamente, muestra un mensaje de confirmación de que la nueva instancia ha sido guardada con éxito.

5.3 Etapa de diseño

En la etapa de diseño, nos pusimos manos a la obra para diseñar una aplicación que fuera capaz de cumplir con todas las funcionalidades que nos habíamos planteado previamente.

5.3.1 Decisiones de diseño

En este apartado, queremos comentar las principales decisiones de diseño que hemos tomado al enfrentarnos con la aplicación recolectora. Principalmente estábamos buscando hacer un diseño simple y elegante, y que a la hora de la implementación se intentó que fuera lo más similar al diseño posible, no llegando a tener todas las funcionalidades previstas. Por supuesto que tuvimos muy en cuenta experiencia del usuario, intentando en todo momento que sea la mejor posible.

A continuación, se describen las decisiones de diseño de forma detallada.

Recordar la sesión de usuario

La primera decisión que tomamos, fue referente a las sesiones de usuarios. Recapitulando en la idea de funcionamiento de la aplicación, cada usuario cuenta con un nombre y una contraseña que lo identifica unívocamente. Además, todos los usuarios pertenecerán a uno o varios grupos. Con esta clasificación, cada usuario podrá acceder únicamente a los formularios que estén asociados a los grupos a los que él pertenezca. Por tanto, es muy importante que los usuarios se autentiquen antes de comenzar a usar la aplicación.

Ahondando en ese concepto, nos dimos cuenta que podría ser molesto o incómodo, obligar al usuario a iniciar sesión con su nombre y contraseña cada vez que utilice la aplicación. Por lo tanto, decidimos que se le solicitarán las credenciales de acceso la primera vez que el usuario inicie la aplicación. Esos datos se validarán contra el repositorio, para corroborar que efectivamente esta dado de alta en la plataforma y una vez hecho el login, se podrá utilizar la aplicación normalmente.

La próxima vez que el usuario acceda, la aplicación tendrá almacenados sus datos de autenticación, con lo cual no se los volverá a solicitar. Con esto conseguimos que el usuario comience rápidamente a recolectar datos. Claro está que en cualquier momento que lo desee, el éste se podrá desconectar del sistema, yendo al apartado de *Opciones de Configuración* y seleccionando la opción *Desconectarse*.

Mostrar un campo por pantalla

El principal problema que le encontramos a los teléfonos móviles, para ser utilizados como interfaces para recolectar datos, es el tamaño de sus pantallas y lo complicado que puede llegar a ser escribir mucho texto, usando los pequeños teclados que proporcionan. Debido a que los usuarios del Recolector pasarán el grueso del tiempo completando instancias de formularios, la forma en que lo hacen, es un aspecto a tener muy en cuenta; por tanto la interacción con la aplicación a la hora de recoger datos, ha de ser diseñada muy cuidadosamente.

Principalmente tenemos el problema de cómo vamos a mostrar los campos del formulario en la pantalla del teléfono. Si mostramos demasiados campos juntos, la interfaz puede tornarse incómoda, cargante y difícil de llevar. No podemos olvidar que los campos, además de mostrar el correspondiente *widget* que lo representa, deben mostrar el nombre de la sección a la que pertenecen y una etiqueta informativa a modo de indicación del dato que se pretende recoger.

Por estas razones, creemos que la mejor opción para disponer todos los campos del formulario, es hacerlo de manera que un campo ocupe únicamente una pantalla, y que todos ellos conformen una lista por la que el usuario pueda navegar de atrás hacia adelante.

Moverse entre campos horizontalmente

Definida la forma en la que íbamos a mostrar los campos de un formulario, nos centramos en encontrar la mejor forma de poder navegar entre ellos. Recapitulando, íbamos a tener una lista de pantallas, en donde cada una de ellas mostraría la representación de un único campo, con los textos indicativos correspondientes. Varias ideas se nos vinieron a la cabeza: moverse entre campos de arriba a abajo, utilizar botones para ir al campo siguiente o anterior, moverse entre secciones de izquierda a derecha y luego de arriba a abajo entre campos de la misma sección, etc. Concluimos que todas ellas presentaban problemas de usabilidad, eran poco intuitivas o simplemente, incómodas.

Afortunadamente, llegamos a la idea que creemos es la más simple y elegante de todas, movernos entre campos de forma horizontal y utilizando gestos con los dedos. Utilizando este método, todos los campos conformarían una lista de pantallas que se irían colocando una al lado de la siguiente, donde únicamente se mostraría una a la vez y el usuario podría ir moviéndose entre ellas utilizando un dedo, haciendo un gesto de "arrastrar y soltar" o *swipe,* si nos guiamos por la literatura inglesa. Creemos que es la opción más cómoda, tanto para los usuarios como para el desarrollo.

Navegar entre secciones

El diseño de la disposición de los campos en la pantalla que comentamos en el punto anterior, nos pareció el más cómodo en todos los aspectos. Utilizando un dedo, el usuario iría arrastrando pantallas de izquierda a derecha, moviéndose entre los distintos campos del formulario.

El único inconveniente que hemos detectado, viene dado cuando un formulario es extenso o cuando el usuario quiere desplazarse rápidamente de un campo a otro y estos están "lejos" entre sí. Para solventar este problema, y contribuir asía la usabilidad de nuestra aplicación, pensamos en la posibilidad de movernos entre secciones de forma rápida y directa. Para ello, tendríamos una opción en el menú, que nos muestre un listado de todas las secciones del formulario. Seleccionando una de ellas, podríamos movernos directamente a esa sección.

Esta es una funcionalidad que se obtiene a partir de una idea de que barajamos cuando diseñamos la disposición de campos y secciones. La idea la comentábamos en el apartado anterior y consistía en moverse entre secciones horizontalmente y verticalmente entre campos de una misma sección.

5.3.2 Mockups

Con la lógica de negocio ya definida, el siguiente paso que dimos, fue diseñar las interfaces gráficas. Este es un aspecto sumamente importante, ya que es la parte de la aplicación con la que el usuario tendrá que interactuar. Por tanto debe cumplir unos índices de usabilidad básicos para que la experiencia de uso no sea negativa.

Para conseguir una presentación lo más lograda posible y que la experiencia del usuario sea la mejor, decidimos utilizar la herramienta Balsamiq para el desarrollo de *mockups*, tal y como lo hicimos con la aplicación modeladora. Los *mockups* nos brindan una primera visión de lo que sería la interfaz de la aplicación. En base a esto, podemos realizar un mejor diseño de las interfaces; discutir el nivel de usabilidad que tienen; plasmar en papel las ideas que tiene cada uno de nosotros y así, poder compartirla mejor con el resto del grupo; y por supuesto, afinar el trabajo de cara a la implementación final, ya que teniendo una imagen clara de cómo se vería la aplicación, el desarrollo es más cómodo y rápido.

A continuación, veremos los distintos *mockups* que fuimos generando. Cabe destacar, que estos diseños son simplemente orientativos y que posteriormente el prototipo no terminó siendo exactamente igual. Además, hubieron casos en los que al utilizar el prototipo, nos dimos cuenta que las interfaces necesitaban algunos reajustes.

Pantalla de login

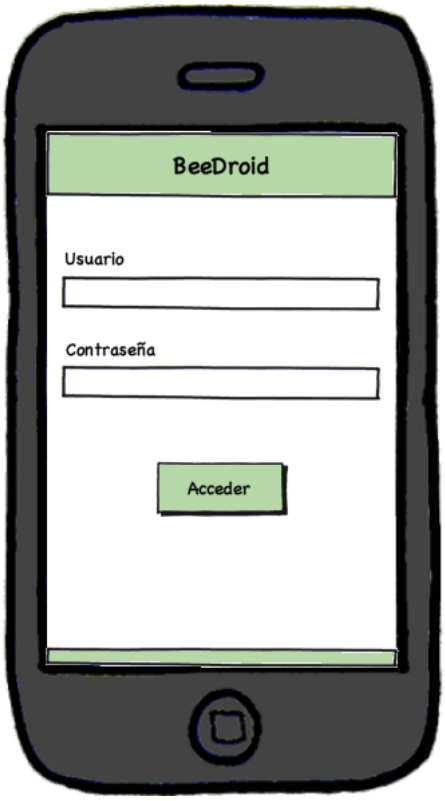


Figura 5-3 *Mockup* de la pantalla de *login*.

La primera vez que se inicie la aplicación, el usuario verá una pantalla como esta. Se le solicitará un nombre de usuario y una contraseña para validarse en la aplicación. Las próximas veces que el usuario inicie la aplicación, no se le volverán a solicitar esta autenticación, simplemente por temas de comodidad. Quedará a criterio del usuario si querrá dejar la sesión iniciada en todo momento, o si lo cree oportuno, cerrarla cada vez que termina de utilizar la aplicación.

Portada inicial

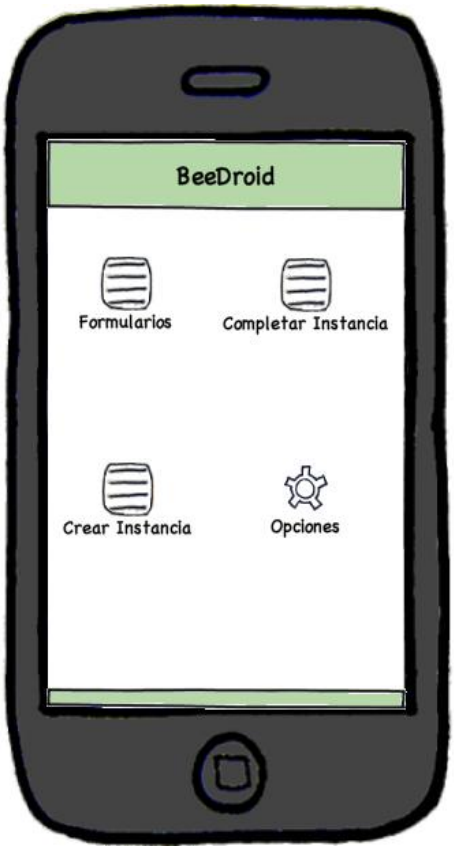


Figura 5-4 *Mockup* de la portada inicial.

Esta será la pantalla que el usuario vea cada vez que comience a utilizar la aplicación siempre y cuando esté *logueado* en el sistema. Como podemos apreciar, cuenta con cuatro accesos directos. Uno de ellos para acceder a los formularios que tiene almacenados en el teléfono, otro para continuar completando una instancia que haya almacenado previamente, el tercero para comenzar una nueva instancia de un formulario, y por último, una pantalla con opciones de configuración. Con las siguientes imágenes, profundizaremos en cada una de estas pantallas.

Formularios

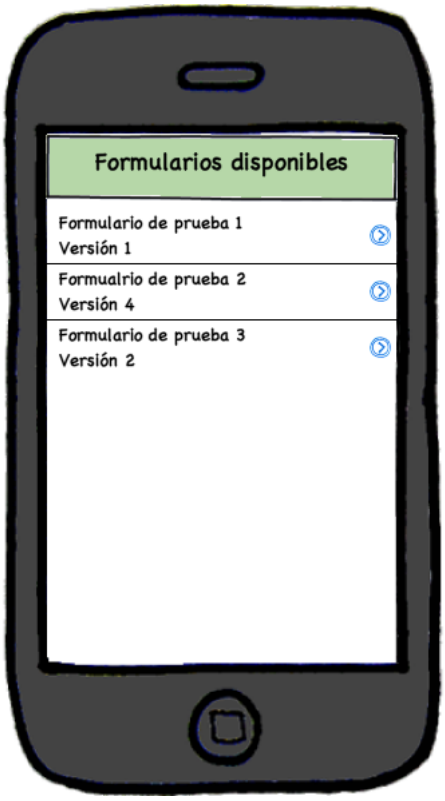


Figura 5-5 *Mockup* de la pantalla de formularios.

Lo siguiente que vemos, es la pantalla de gestión de los formularios. Estos se presentan en forma de lista en vertical. Cada ítem de esta lista, representa un formulario que está almacenado en el dispositivo móvil. Por comodidad y simpleza, se muestran únicamente el nombre y la versión de cada uno. Todos los ítems de la lista serán seleccionables y mostrarán información extendida del formulario correspondiente.

Información de un formulario

Figura 5-6 *Mockup* de la información detallada de un formulario.

Seleccionando un formulario de la lista anterior, podemos acceder a una vista con información extendida sobre él. En la imagen vemos como se muestran los principales datos del formulario: el nombre del autor, la fecha de creación y una pequeña explicación de para que debería ser utilizado. También tendremos la posibilidad de eliminarlo, seleccionando la opción correspondiente en el Menú Android.

Descargar nuevos formularios

Figura 5-7 *Mockup* de la opción del menú para descargar formularios.

Volviendo a la pantalla de los formularios, habrá una opción del menú que permita consultar si existen formularios nuevos disponibles para descargar. Ya lo comentábamos, pero el recolector se pondría en contacto con el repositorio, obtendría una lista de identificadores de formularios, la cual mostraría al usuario.

Formularios disponibles para descargar

Figura 5-8 *Mockup* de la lista de formularios nuevos disponibles para descargar.

Cuando el usuario seleccione la opción del menú para descargar formularios, la aplicación mostrará una lista con los formularios que ha obtenido del repositorio y que no estén almacenados en el dispositivo. Este matiz es interesante, ya que no si ya tenemos almacenado un formulario en el dispositivo, no lo muestre como disponible. Esta lista mostrará el nombre y versión de los formularios, así como un selector que permitirá marcarlos para poder descargarlos. Además, en las opciones del menú, tendremos las acciones básicas de una lista de ítems seleccionables: podremos seleccionarlos todos, seleccionar ninguno y descargar los formularios seleccionados.

Completar nueva instancia

Volviendo a la portada inicial, pasaremos a comentar como sería la interfaz de una nueva instancia. Al seleccionar el botón *Crear Instancia*, el usuario tendrá la posibilidad de elegir de qué formulario quiere crear la nueva instancia. La pantalla será muy similar a la de formularios, con una lista vertical, con sus elementos seleccionables.

Cada vez que el usuario se disponga a crear una nueva instancia de un formulario, el sistema mostrará todos los campos de este, uno por pantalla, tal y como explicábamos anteriormente.

Básicamente las interfaces de todos los campos tendrán la misma estructura. Comenzando desde arriba, tenemos el título de la sección, con letra negrita. Más abajo, una etiqueta de texto, que brinde información sobre los datos que se pretenden recoger. Por último, tenemos un *widget* correspondiente al campo definido en el formulario.

A continuación, veremos solo algunos de los campos que tendremos disponibles.

Figura 5-9 *Mockup* de una pantalla de un campo tipo imagen.

Figura 5-10 *Mockup* de una pantalla de un campo tipo geolocalización.

Figura 5-11 *Mockup* de un campo tipo *radio-button* con las opciones del menú.

En el *mockup* anterior, además de ver una vista de un campo tipo *radio-button*, también podemos apreciar las opciones disponibles en el menú. El usuario puede acceder a ellas tan solo presionando la tecla de Menú Android. Las opciones permiten finalizar la instancia y enviarla al repositorio, enviarla al repositorio sin completar, guardarla en local o navegar entre secciones.

5.4 Etapa de implementación

Hasta el momento, hemos estado hablando del trabajo que realizamos diseñando la aplicación móvil. Vimos como comenzamos desde cero, únicamente con ideas de lo que pretendíamos construir; hasta llegar a materializarlas en esquemas, diagramas y flujos que representan lo que pretendemos que sea el Recolector. Sabemos cómo se tiene que ver la capa de presentación, cual es la lógica de negocio que debe de existir por debajo y cuáles son los datos que queremos persistir.

En este apartado, llevaremos a la práctica estos diagramas y explicaremos todo el proceso de implementación. Debemos decir, que nuestra experiencia previa con todas las tecnologías utilizadas en el proyecto, en general no era mucha, pero en el caso de Android, era nula. Por tanto, ese aspecto tiene mucho mérito por parte de nosotros tres, ya que logramos construir una aplicación desde cero, siendo ésta, la primera que implementábamos para una plataforma móvil.

A continuación, pasaremos a comentar los principales aspectos de la implementación de la aplicación móvil. Por razones de espacio, no podremos cubrir todos y cada uno de los detalles del código, por lo tanto haremos una selección de los módulos más relevantes, explicaremos las tecnologías que intervinieron y como orientamos la arquitectura de la aplicación.

5.4.1 Tecnologías utilizadas

En este apartado enumeraremos las diferentes tecnologías que hemos utilizado para la implementación del recolector. Recordamos que está desarrollado para correr sobre el sistema operativo móvil Android, por tanto esta puramente escrito en Java y XML. Damos por sentado que todas las funcionalidades que ellos ofrecen son de conocimiento popular.

- La aplicación recolectora ésta desarrollada en **Java**, más concretamente en la versión 6. Prácticamente no existen diferencias entre lo que se puede hacer con este lenguaje en una aplicación de escritorio y una específica para Android. Solamente aparecen algunas limitaciones en cuanto a generación de código en tiempo de ejecución, pero no nos hemos topado con esa necesidad. Por tanto, nuestro conocimiento en el lenguaje, fue algo que pudimos reutilizar.
- Una de las grandes ventajas que ofrece el *SDK* de Android, es que permite crear código a partir de ficheros **XML**. El desarrollador, utilizando una notación puramente **XML**, define recursos que posteriormente puede utilizar en su aplicación. Estos recursos pueden ser interfaces, cadenas de texto, imágenes, etc. Android, de forma eficiente, transforma esos **XML** en código ejecutable para que el desarrollador pueda referenciarlo desde su código.

Básicamente se utiliza una clase autogenerada, llamada la clase *R*, en donde existen índices que apuntan estos objetos en memoria.

- Como comentábamos anteriormente en la descripción general del proyecto, las definiciones de formularios vienen especificadas en ficheros **XML**. Cuando el recolector muestra un formulario para que el usuario rellene una instancia de él, primero realiza una traducción de ese **XML** a vistas que pueda mostrar. Estas traducciones se hacen a través de un módulo traductor o *parser* definido enteramente por nosotros. Para hacerlo posible, nos apoyamos en el paradigma que define **SAX-Parser** para el tratamiento de XML. El motivo de esta elección es simplemente por eficiencia. Las otras alternativas que aparecían como candidatas para implementar el traductor eran DOM y Pull-Parser [64]. Ambas dos, parecen ser más amigables con el desarrollador, pero consumen mucho más recursos. Por lo tanto, preferimos complicar la lógica de traducción a cambio de hacerla más eficiente. Con **SAX-Parser**, tuvimos escribir una clase que implemente una interfaz y que defina cinco métodos, que serán llamados cuando se abra el documento XML, cuando se abra un *tag*, cuando se cierre un *tag*, cuando se lea texto dentro de un *tag* y cuando se cierre el documento, respectivamente. Profundizaremos más en este tema en el apartado siguiente.
- Para la implementación del cliente de los servicios Web, utilizamos una librería externa a Android, llamada **KSOAP2** [65]. Nos vimos en la necesidad de recurrir a ella debido a que el sistema operativo no brinda estas funcionalidades de forma nativa. **KSOAP2** surge de un proyecto libre que pretendía brindar la posibilidad de consumir servicios **SOAP** a aplicaciones basadas en J2ME. En su versión actual, es totalmente compatible con Android. Como comentábamos anteriormente, sabíamos que **SOAP** no era la opción más eficiente, pero si con la que estábamos más familiarizados. Aún así, en el primer prototipo nos vimos sorprendidos con su desempeño y rendimiento, por tanto esto dejó de ser un problema significativo.
- Android ofrece varias posibilidades de almacenamiento en el dispositivo móvil. Estas pueden ser en ficheros alojados en la memoria interna del teléfono, alojados en la memoria externa (tarjeta *SD*) o datos privados de la aplicación almacenados en una base de datos, utilizando **SQLITE** [66]. Como no nos interesa que el usuario tenga acceso a las definiciones de los formularios ni a las instancias almacenadas, decidimos que la mejor opción era contar con una base de datos. Android ofrece una API bastante intuitiva y simple para gestionar la base de datos. Ahondaremos en este tema en apartados posteriores.

5.4.2 Decisiones de implementación

Nombradas las tecnologías que utilizamos en el desarrollo, pasaremos a comentar como llevamos a cabo la implementación.

Nunca nos habíamos enfrentado al desarrollo de aplicaciones para una plataforma como esta, por tanto tuvimos que superar una pequeña curva de aprendizaje. Para nuestra suerte, la comunidad desarrolladora de aplicaciones Android es muy fuerte, la documentación que ofrecen en la página oficial es muy buena y las herramientas de trabajo están bien diseñadas y son de fácil utilización. Aún así, nos vimos en la necesidad de hacer pequeñas pruebas con los ejemplos que vienen incluidos en el *SDK* para comenzar a practicar. También

cabe mencionar, que en la etapa inicial de la implementación, el desarrollo se hizo utilizando el emulador (incluido en el *SDK*).

Una vez que ya entendimos los conceptos básicos del desarrollo para Android, comenzamos con un pequeño prototipo, extremadamente básico: únicamente contaba con una portada inicial con opciones de menú que permitieran al usuario acceder a las distintas pantallas. Luego fuimos desarrollando distintas pantallas en donde se ofrecían las funcionalidades más básicas que teníamos planteadas: mostrar la lista de formularios que estuviesen almacenados en el teléfono, mostrar los formularios disponibles para descargar y mostrar una pantalla con opciones de configuración. Todo estaba muy verde y no había ningún tipo de lógica por detrás, únicamente interfaces.

Lo siguiente que implementamos, fue el cliente de los servicios web, consumiendo solamente el servicio de obtener los formularios disponibles para descargar. En este punto, ya teníamos funcionando el servidor en local, con los servicios desplegados y testeados, por lo que el desarrollo fue directo. Nos basamos en muchos ejemplos que encontramos por la red, ya que KSOAP2 no cuenta con una buena documentación. Finalmente, conseguimos un cliente que sea capaz de conectarse con el servidor y descargarse una lista de formularios disponibles. Aclaremos que esta lista contaba únicamente con el nombre y versión de cada formulario, datos necesarios para identificarlos.

Luego implementamos la lógica que permitiese seleccionar uno o varios formularios para poder descargárselos al dispositivo. Esta tarea fue relativamente costosa, ya que nos vimos en la necesidad de investigar profundamente la política que sigue Android para la gestión de elementos seleccionables en la pantalla. Cada uno de estos elementos sería un formulario que el cliente nos ha devuelto como disponible para descargar. El usuario seleccionará aquellos que desea obtener y finalmente se descargarían. Tras mucha investigación y varias pruebas, vimos que la mejor opción era contar con un componente que conociese la lista de elementos de la que se puede elegir (la lista de formularios disponibles), así como aquellos que están seleccionados o no (formularios que el usuario pretende descargar). A este se le conoce como adaptador. Es la forma más adecuada de trabajar, ya se logra separar la capa de vistas, en donde el usuario interactúa seleccionando elementos, de la capa de lógica, en donde se conoce los elementos seleccionados y se actúa en función a ellos. Además, conseguimos que este adaptador fuese eficiente debido a que mantendría en memoria únicamente aquellos elementos que el sistema estuviese mostrando en cada momento. Por último, el adaptador recibiría la orden de devolver los elementos seleccionados para que el cliente pudiese descargarlos del repositorio. Fue aquí en donde implementamos el método que consumiese el segundo servicio desplegado en el servidor, descargarse una definición completa de un formulario (XML).

Con el cliente operativo, debíamos almacenar los formularios que se descargaban. Nos vimos en la necesidad de programar otro de los grandes componentes de la aplicación, el módulo de persistencia. Este gestionaría la base de datos que almacena los formularios descargados, de forma independiente al resto de la aplicación.

Uniendo las piezas, teníamos un artefacto que se encarga de la comunicación con el repositorio, otro que gestiona la persistencia de la aplicación mediante una base de datos SQLITE. Además, contábamos con las correspondientes interfaces que permitiesen al usuario interactuar con el sistema. Llegados a este punto, necesitábamos comenzara rellenar

instancias de los formularios. Esta tarea fue, sin lugar a dudas, la más compleja de todo el proceso; que constaba de dos partes claramente diferenciadas: la primera era hacer la traducción de una definición en XML de un formulario a clases intermedias para finalmente generar vistas que se puedan mostrar la en pantalla y por otro lado gestionar estas vistas para que el usuario sea capaz de navegar entre todos los campos del formulario de forma elegante.

Como ya adelantamos en el apartado de tecnologías utilizadas, para implementar el traductor utilizamos SAX-Parser. Este traductor recorrería el XML del formulario y generaría una serie de objetos que representan a cada campo del formulario. La metodología que sigue SAX, se basa en utilizar una implementación de una interfaz manejadora o *handler*, definida por el desarrollador, que cuente con una serie de métodos que este espera encontrar. Estos métodos son los siguientes:

- `startDocument`: invocada cuando se abre el documento XML. Se utiliza para tareas de inicialización de parámetros y atributos.
- `endDocument`: se invoca cuando se cierra el documento. En nuestro caso, no nos hizo falta.
- `startElement`: se llama cada vez que SAX abre un nuevo *tag* XML. Recibe como argumentos el nombre del *tag*, el *namespace* y los atributos que este tiene. Generalmente, con una serie de *flags*, se lleva la cuenta de los *tags* que han sido abiertos, y se instancian objetos que se usarán al cerrar los correspondientes elementos homónimos.
- `characters`: es llamada cuando se detecta texto perteneciente a un elemento XML. Este se va metiendo en un *buffer*, que es consumido por la clase cuando el *tag* se cierra.
- `endElement`: se llama cada vez que SAX cierra un *tag* XML. Se actúa en función al texto que se haya recogido dentro del elemento y se establecen atributos a los objetos anteriormente creados.

A partir de los elementos que se fuesen encontrando en el XML, se construyen estructuras intermedias, para almacenar de forma manejable toda la información recaudada. Estos objetos se encargan de contener datos sobre el tipo de campo con el que se ésta trabajando, así como una serie de parámetros adicionales. Se planteó una estructura jerárquica en forma de árbol en donde el nodo padre sería la instancia del formulario, tendría como hijos todas las secciones del formulario y estas a su vez, los campos pertenecientes a cada una de ellas. El objeto instancia (`InstanceBean`) recogería toda esa estructura y es la salida que se obtiene del proceso de *parsing*.

Con esta estructura creada, se pasaría a generar vistas que correspondan con cada uno de los campos. Para ello, contamos con una clase especializada en dicha tarea: `InstanceBeanManager`. Esta recogería la estructura de objetos intermedios procedentes de la traducción y cuando se le solicite, haría un segundo paso de conversión a objetos tipo vistas de Android (`View`). Más adelante detallaremos este proceso.

El siguiente paso era gestionar la lista de campos que obtuvimos de la traducción, para poder presentarlas al usuario y que este se pueda mover entre ellas. Ya habíamos comentado

en la etapa de diseño, que queríamos mostrar los campos del formulario en una galería horizontal, donde el usuario pueda moverse entre ellos de izquierda a derecha, utilizando gestos con los dedos. Muchas fueron las pruebas que realizamos y la investigación se hizo tediosa, pero finalmente apoyándonos en ejemplos que encontramos por la red, conseguimos nuestro cometido. Explicaremos más sobre este componente en apartados posteriores.

Concluyendo con la implementación, teníamos que ser capaces de generar un XML de salida, que defina la instancia y que recoja los valores introducidos por el usuario; para luego poder enviarlo al repositorio y allí almacenarlo. Para ello, añadimos lógica a cada una de las clases intermedias que se corresponden con cada campo del formulario, para que contasen con un método que devuelva, en formato XML, el tipo de campo que albergan y los valores que el usuario haya introducido. El objeto instancia, nodo padre de la estructura jerárquica antes comentada, comenzaría una serie de llamadas en cadena a las secciones hijas y estas a su vez, los campos que las conforman, para que se fuese generando dicho XML. Por último, tuvimos la necesidad de implementar la tercera llamada a los servicios web, esta vez para enviar la traducción final de una instancia al repositorio.

En los siguientes puntos, queremos profundizar en los aspectos y componentes más importantes de la aplicación, para dejar claro todo el proceso.

Plataforma Android

Ya hemos explicado las principales características de la plataforma Android. Ahora nos centraremos en los aspectos de importancia en la implementación del recolector.

Una de las primeras decisiones que hay que tomar cuando se comienza el desarrollo de una aplicación Android, es el grado de compatibilidad que se quiere obtener. Este todavía sigue siendo uno de los puntos débiles del sistema operativo, ya que la política a seguir en términos de compatibilidad entre versiones y fabricantes no está bien definida. Las distintas decisiones que tuvimos que tomar, se enumeran a continuación.

- **Versión del sistema operativo**. Cuando comenzamos el desarrollo de la aplicación, la versión actual de Android era la 2.3 (*Gingerbread* [67]), lanzada en febrero de 2011. Decidimos descartarla como base de nuestra implementación, por la poca presencia que tenía en el mercado, básicamente debido al poco tiempo que había transcurrido desde su lanzamiento. Por otro lado, debíamos tener en cuenta la compatibilidad con versiones más antiguas, lo cual generaría varios inconvenientes, ya que una aplicación desarrollada para una versión relativamente "vieja", puede que no funcionasen una actual y viceversa; con lo que habría que implementar varias versiones de la aplicación, que estén preparadas para soportar las distintas versiones de Android. Para evitar todos los problemas de compatibilidad, optamos por basar la implementación en la versión estable más extendida y actual de Android, la 2.2 (*Froyo* [68]). Con esta decisión, cortamos todo tipo de compatibilidad con versiones anteriores.
- **API Level**. En el desarrollo de aplicaciones Android, además de pensar en la versión del sistema operativo sobre la que se quiere desarrollar, hay que tener en cuenta el *API Level*, que no es más que la versión de la API del *framework*. Este identificador le permite al sistema determinar correctamente si una aplicación que se está instalando

es compatible con él. Generalmente, con cada nuevo lanzamiento de una versión, se actualiza la API, pero no siempre es el caso. En estos lanzamientos, se especifican los principales cambios que se hicieron en las APIs, nombrando las funcionalidades nuevas que se añaden y aquellas que pasan a ser obsoletas. Por tanto, se deben tener claras las funcionalidades del sistema que se van a requerir en la aplicación y contrastarlas con las diferentes APIs, para conocer cuáles de estas son válidas. Dado que no íbamos a contar con compatibilidad con versiones anteriores, llegamos a la conclusión utilizaríamos la versión 8 de las APIs del *framework*, correspondiente con la versión que habíamos elegido.

- **Paquetes de terceros**. Cuando se determina el *API Level* o la versión del SDK con la que se va a trabajar, también se puede optar por la inclusión de paquetes de terceros, para funcionalidades específicas de algunos fabricantes, por ejemplo Samsung [69] o HTC [70]. Como no queríamos implementar ninguna funcionalidad demasiado compleja que pueda necesitar el uso de alguna de estas librerías, decidimos descartarlas por completo y apoyarnos únicamente en el *framework* estándar de Android. Esto nos ayudaría a hacer más portable la aplicación, para que pueda correr en todo tipo de dispositivos.
- **Temas gráficos de terceros**. Al igual que comentábamos en el punto anterior, algunos fabricantes también ponen a disposición de los desarrolladores paquetes gráficos para adaptar sus aplicaciones a los temas modificados que se utilizan en sus dispositivos. Por las mismas razones que se mencionaron antes, descartamos esta idea.

Arquitectura de la aplicación

Este apartado está reservado para explicar la organización que hemos seguido en la estructura del código de la aplicación. Repasaremos los distintos paquetes y se explicarán sus razones de ser, así como las responsabilidades que tienen cada uno de ellos.

En la siguiente imagen, vemos la estructura de directorios que conforman los mencionados paquetes.

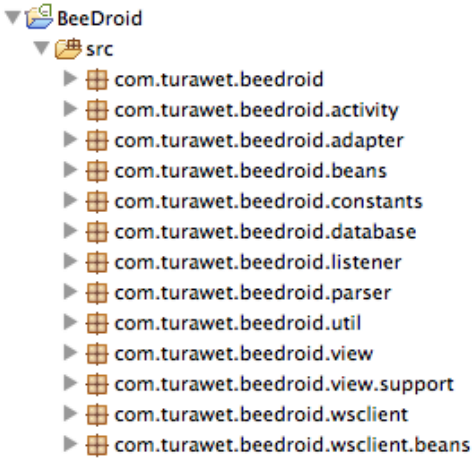


Figura 5-12 Paquetes de la aplicación.

- **com.turawet.beedroid:** contiene la *activity* que se carga como portada inicial y está pensado para que aloje a la *activity* de *login*
- **com.turawet.beedroid.activity:** aloja a todas las *activities* de la aplicación.
- **com.turawet.beedroid.adapter:** contiene adaptadores definidos por nosotros, para gestionar las diferentes listas de formularios o instancias que haya en el dispositivo de forma eficiente, ya que permite tener cargados en memoria únicamente aquellos elementos que se estén mostrando en la pantalla, y se consigue mantener separadas la capa de presentación de la capa de lógica.
- **com.turawet.beedroid.beans:** clases intermedias utilizadas en la traducción de los XML de formularios e instancias a vistas. Más adelante comentaremos más sobre ellas.
- **com.turawet.beedroid.constants:** recoge una clase constante que se utiliza para albergar todas las constantes de la aplicación.
- **com.turawet.beedroid.database:** paquete donde se encuentran las clases encargadas de gestionar y manipular la base de datos. Se comentarán más sobre ellas más adelante.
- **com.turawet.beedroid.listener:** contiene una clase que cumple la función de *listener*, que se mantiene a la escucha del sistema de posicionamiento esperando nuevas posiciones cuando el usuario solicita geolocalizar la instancia que esta rellenando.
- **com.turawet.beedroid.parser:** aquí se encuentran las clases encargadas de la traducción de los XML de los formularios a clases intermedias para luego poder generar las vistas de los campos.
- **com.turawet.beedroid.util:** alberga clases auxiliares, por ejemplo para generar mensajes de avisos al usuario.
- **com.turawet.beedroid.view:** aquí se encuentran las distintas vistas desarrolladas por nosotros. Recordemos que Android provee un gran conjunto de tipos de interfaces que se pueden utilizar, pero en casos excepcionales, se pueden definir unas propias. Nosotros implementamos una vista base para cada campo y una para la galería de campos de una instancia.
- **com.turawet.beedroid.view.support:** alberga la clase encargada de realizar la traducción de clases intermedias que se obtuvieron a partir de los XML de formularios, a objetos tipo vistas que se mostrarán luego en el dispositivo.
- **com.turawet.beedroid.wsclient:** cliente SOAP que consume los servicios Web desplegados en el repositorio.
- **com.turawet.beedroid.wsclient.beans:** clases auxiliares que se usan como parámetros en las llamadas a los servicios Web.

Cliente Web Services

Para comunicar la aplicación móvil con el repositorio, se debía implementar un cliente que consuma los servicios web publicados en el repositorio. Como ya adelantamos en apartados anteriores, íbamos a basar la comunicación en SOAP y para el desarrollo del cliente en Android, usaríamos la librería KSOAP2.

La clase que implementa el cliente, es una clase *sigleton*. Aplicamos este patrón de diseño debido a que el cliente sería único e iba a ser utilizado muy a menudo. Con esto conseguimos realizar solamente una instancia de la clase y mantenerla siempre cargada en memoria, acelerando el acceso a ella.

```
WSClient ws = WSClient.getInstance();
```

Código 5-1 Obteniendo una instancia del cliente

```
privateWSClient()
{
      transportSE = newHttpTransportSE(Cte.WSClient.URL_TO_WSDL,
                                       Cte.WSClient.TIMEOUT);
}

private synchronized static void createInstance()
{
      if(wsClient == null)
            wsClient =new WSClient();
}

public static WSClient getInstance()
{
    createInstance();
    return wsClient;
}
```

Código 5-2 Clase definida como *singleton*

Para conectar con el repositorio, utilizamos la clase `HttpTransportSE`, definida en KSOAP2. Esta recibe la *URL* al *WSDL* [71]donde se establecen los servicios web disponibles para consumir y el tiempo en milisegundos que va a esperar por una respuesta antes de dar un error de conexión. Esta tiene definido un método por cada servicio consumido. Estos son:

- `getFormByNameVersion`: devuelve la definición de un formulario a partir de su nombre y versión.
- `getAllFormPreview`: devuelve todos los formularios disponibles para descargar (únicamente nombre y versión).
- `uploadNewInstance`: envía al repositorio una instancia completa.

En el siguiente cuadro, vemos método que implementa el cliente para consumir el servicio que obtiene la definición de un formulario a partir de su nombre y versión. Del código

podemos notar que el objeto `request`, definido en la línea 71, es el encargado de adjuntar los parámetros que recibe el servicio (añadidos en las líneas 78 y 79). En la línea 83 se hace la llamada a este, pasándole el sobre SOAP (`envelope`) que contiene los parámetros que espera; y por último, en la línea 85, obtenemos la respuesta, que es devuelta encapsulada en otro objeto.

```
67.
68. publicFormInfoBean getFormByNameVersion(FormIdentificationBeanform)
69. throwsIOException,XmlPullParserException
70. {
71.         SoapObject request =
                newSoapObject(Cte.WSClient.NAMESPACE,
                              Cte.WSClient.GET_XMLFORM_BY_NAME_VERSION);
72.
73.         SoapSerializationEnvelope envelope =
                            newSoapSerializationEnvelope(SoapEnvelope.VER11);
74.
75.         String encodedName =
76.         Base64.encodeToString(form.getName().getBytes(),
                                  Base64.URL_SAFE);
77.
78.         request.addProperty(FormWsBean.name, encodedName);
79.         request.addProperty(FormWsBean.version, form.getVersion());
80.
81.         envelope.setOutputSoapObject(request);
82.
83.         transportSE.call(Cte.WSClient.GET_XMLFORM_BY_NAME_VERSION,
                             envelope);
84.
85.         String xmlForm = envelope.getResponse().toString();
86.
87.         returnnewFormInfoBean(form, xmlForm);
88. }
89.
```

Código 5-3 Ejemplo de llamada a un servicio web.

Gestor de la base de datos

El otro gran componente de módulo recolector, es el encargado de manejar la persistencia. Como ya adelantábamos, esta se hace a través de una base de datos privada a la aplicación, basada en SQLITE, que almacena los formularios que el usuario se descarga. En un futuro, además implementaremos el almacenamiento de instancias, para que estas se puedan guardar en el dispositivo y ser enviadas con posterioridad.

Al igual que el cliente de los servicios web, el gestor de la base de datos es una clase *singleton*, por tanto la forma en la que se instancia es exactamente igual.

Aplicando un buen diseño de clases, hemos otorgado todas las responsabilidades de creación, acceso y gestión de la base de datos a la clase `DataBaseManager`. De esta forma, logramos encapsular todas las tareas relativas a la base de datos en esta clase. A su vez, esta hace uso de la clase `DataBaseOpenHelper`, que extiende a `SQLiteOpenHelper`, definida en la API Android para gestionar bases de datos SQLITE.

La primera vez que se instancia un objeto del tipo `DataBaseOpenHelper`, si la base de datos que se le especifica no existe, se crea; luego simplemente se referencia. Como decíamos, nuestra clase gestora tiene un atributo de este tipo:

```
private DataBaseOpenHelper dbAccessor;
```

Nuestra clase implementa un método para cada tarea de alto nivel que se requiere en el resto de la aplicación:

- `saveForms`: recibe una lista de definiciones de formularios y los almacena.
- `getSavedFormsIdentification`: devuelve los identificadores de los formularios que están almacenados (nombre y versión).
- `getFormInfo`: devuelve la definición de un formulario concreto a partir de su nombre y versión.

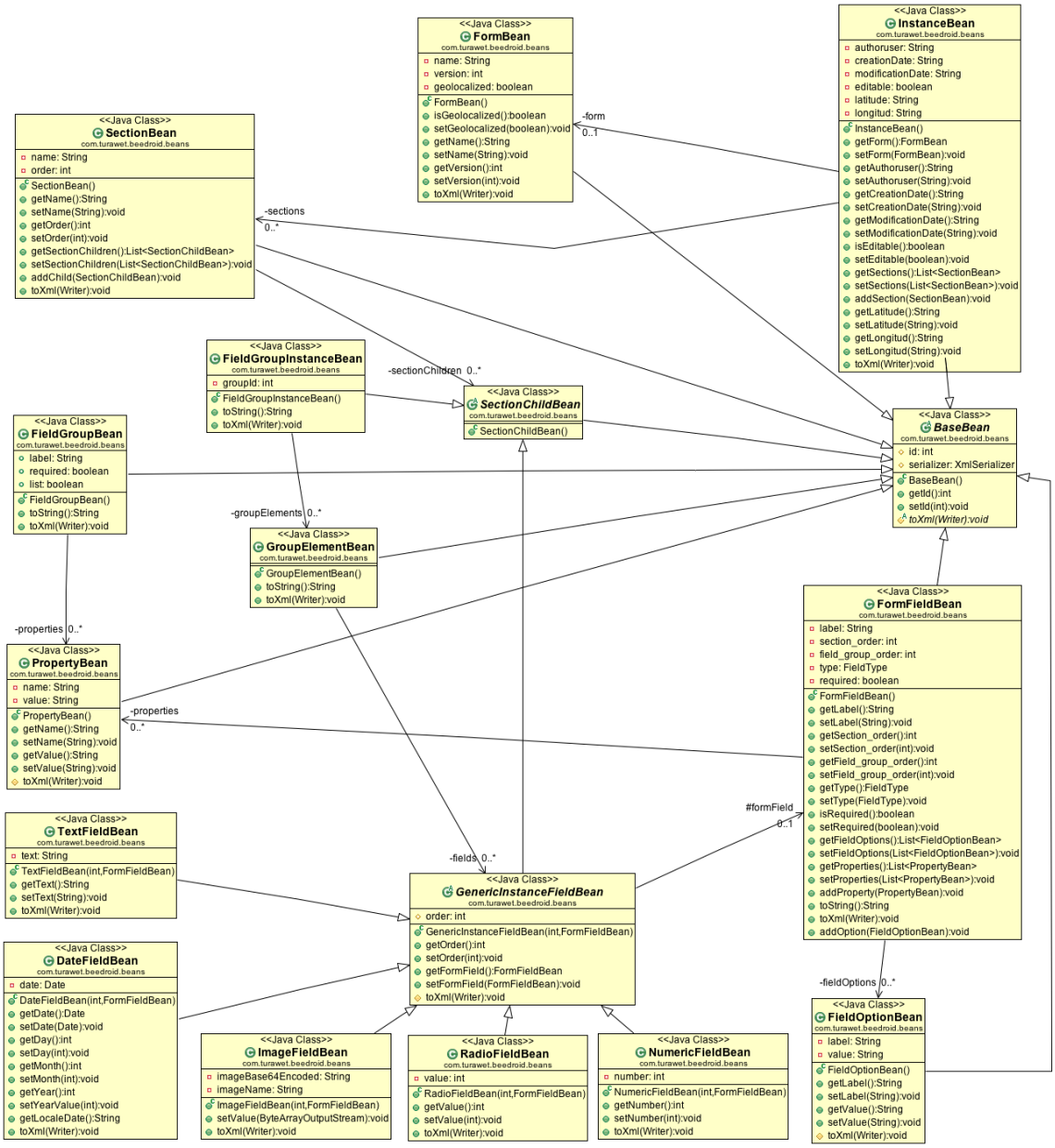
En el siguiente cuadro de código podemos ver uno de los métodos citados anteriormente. Como podemos comprobar, el atributo `dbAccesor` sirve de interfaz para trabajar con la base de datos directamente. En este caso, estamos haciendo inserciones en ella, pero funciona de la misma forma cuando se pretenden realizar consultas. Podríamos decir que nuestra clase define la lógica para montar las consultas o inserciones y este se encarga de interactuar directamente con la base de datos.

```
public boolean saveForms(List<FormInfoBean> formsToSave)throws SQLException
{
	SQLiteDatabase db = dbAccessor.getWritableDatabase();
	ContentValues values = new ContentValues(formsToSave.size())
	for (FormInfoBean formInfo : formsToSave)
	{
		values.put(Cte.DataBase.NAME, formInfo.getFormId().getName());
		values.put(Cte.DataBase.VERSION,
				formInfo.getFormId().getVersion());
		values.put(Cte.DataBase.XML, formInfo.getXml());
		db.insertOrThrow(Cte.DataBase.FORMS_INFO_TABLE,null, values)
	}
	return true;
}
```

Código 5-4 Método que almacena definiciones de formularios en la base de datos.

Clases intermedias

En el diagrama de clases que se describe a continuación se muestran las clases intermedias que se han desarrollado a fin de tener una representación temporal de formularios e instancias en los procesos de traducción de los ficheros XML de formularios e instancias en el teléfono móvil.



Código 5-5 Diagrama de clases intermedias de representación en el Recolector móvil.

Traducción de formularios a vistas

Como último punto de la etapa de implementación, nos gustaría profundizar en el desarrollo del artefacto *software* encargado de traducir de las definiciones en XML de los formularios, a pantallas que se puedan mostrar en el dispositivo móvil; así como la gestión de

estas vistas junto a la interacción con el usuario .Para entender el proceso, vamos a ir paso a paso, comentando cada uno de los puntos.

El proceso comienza cuando un usuario selecciona un formulario para completar una instancia. Recordamos que la definición de ese formulario debe estar almacenada en el dispositivo. El usuario accede a la sección "Crear Instancia" y elige dicho formulario.

La *activity* encargada de gestionar todo el proceso es `FillNewInstanceActivity`. Esta recibe el nombre y la versión del formulario que el usuario ha seleccionado. Con esa información, se extrae de la base de datos el XML con la definición propia del formulario, tal y como muestra el siguiente cuadro.

```
Bundle parameters = getIntent().getExtras();
String name = parameters.getString(FormWsBean.name);
String version = parameters.getString(FormWsBean.version);
FormIdentificationBeanform =
       newFormIdentificationBean(name, version);
DataBaseManager db = DataBaseManager.getInstance(this);
String xml = db.getFormInfo(fo).getXml();
```

Código 5-6 Obtención del XML del formulario seleccionado.

Con el XML del formulario, se llama al *parser* o traductor basado en SAX-Parser, que ya habíamos comentado. Este devolverá una estructura jerárquica intermedia, basándose en las clases que vimos en el apartado anterior, que nos permita acceder a todos los campos y secciones del formulario, tanto para leer información, como para escribir valores. Cabe mencionar que esta representación del formulario, es fiel a la organización definida en el modelo entidad relación.

```
XmlToBeansParser parser = newXmlToBeansParser(xml);
InstanceBean instance = parser.getInstance();
```

Código 5-7 Traducción de XML a clases intermedias.

En el objeto `instance`, tendremos entonces la estructura que comentábamos. Este hará el trabajo de nodo raíz de un árbol, colgando de él, tendremos las secciones que conforman el formulario y por último, a los campos que las conforman.

En este punto, debemos introducir un gran actor en este proceso, la clase `InstanceBeanManager`. Esta clase, va a generar una lista de vistas correspondientes a cada uno de los campos que conformen la instancia. Para ello, recorrerá convenientemente el árbol y determinando el tipo de campo que se trate en cada caso, generará una vista para él. Estas vistas comparten una estructura común, definida en la clase `FieldView`. El formato de estas vistas es el que se muestra a continuación:

Figura 5-13 Estructura de vista de los campos.

Por tanto, tras la segunda traducción (de clases intermedias a vistas), se obtendrá una lista de "pantallas" que están preparadas para ser mostradas. Esta lista será gestionada por el último actor que aparece en escena, el `BeanViewFlipper`. Esta clase será la encargada de manejar la interfaz de la *activity* `FillNewInstanceActivity`. Como ya se comentaba anteriormente, todas las vistas de los campos se disponen de forma de lista horizontal, y el usuario podrá navegar por ella de izquierda a derecha utilizando gestos con los dedos. La gran ventaja que añade el uso de una clase como esta, es que se separa, una vez más, la gestión de la capa de presentación con la lógica de negocio, lo cual es una buena práctica.

Recapitulando, ya conseguimos traducir la definición de un formulario a vistas, las cuales son mostradas para que el usuario complete la instancia con los valores que crea adecuados. Finalmente, cuando este termine con la recolección, se procederá a enviar la instancia completa. Cabe destacar que en esta versión del prototipo, no se ha implementado la opción de poder guardar la instancia en el dispositivo.

Para cumplimentar esta tarea, es necesario realizar un proceso inverso al ya comentado, debemos obtener los valores que ha introducido el usuario y generar un XML que luego se envíe al repositorio. Una vez, más el encargado de comenzar con este trabajo, es el `InstanceBeanManager`, con el método `readFieldValues`.

```
instanceManager.readFieldValues(flipper.getChildContainer());
String instanceXml = instanceManager.instanceToXml();
```

Código 5-8 Proceso de generación del XML de instancias.

Como su nombre indica, se leen todos los valores almacenados en los campos e internamente, se almacenan en la instancia intermedia. Una vez hecho esto, se pasa a la traducción a XML. El gestor de instancias (`instanceManager`) se encargará de solicitar la traducción a la instancia, para que ésta a su vez lo haga con las secciones que la conforman y

finalmente los campos que pertenecen a ellas. Podemos decir que el proceso de traducción se propaga por todo el árbol, donde cada nodo se traduce a sí mismo y se lo ordena a sus hijos.

```
public void toXml(Writer writer)
throws IllegalArgumentException, IllegalStateException, IOException
{
	serializer =Xml.newSerializer();
	serializer.setOutput(writer);
	serializer.startDocument("UTF-8",null);

	/* ... */

	for(SectionBean section : sections)
	section.toXml(writer);

	serializer.endTag(XmlTags.namespace,
			XmlEnumTags.sections.toString());
	serializer.endTag(XmlTags.namespace,
			XmlEnumTags.instance.toString());
	serializer.endDocument();
}
```

Código 5-9 Traducción de una instancia.

El código del cuadro anterior, muestra el método de traducción de la instancia (`InstanceBean`). Este es llamado por el `InstanceBeanManager`, quién inicia todo el proceso. La instancia se encarga de traducir sus valores, básicamente los metadatos e invoca las traducciones de las secciones. Como podemos apreciar, todos los nodos comparten un parámetro: `writer`. Sobre este, se irá escribiendo el XML que generen. Finalmente el `InstanceBeanManager`, pasará de ese objeto a un `String` que se pueda tratar (objeto `instanceXml` del cuadro 5-8).

Por último, cuando el XML de salida es generado, hacemos la llamada al servicio web para enviar la instancia al repositorio.

```
	WSClient ws = WSClient.getInstance();
	boolean result = ws.uploadNewInstance(instanceXml);
```

Código 5-10 Envío del XML de instancias al repositorio.

5.4.3 Prototipo

Figura 5-14 Portada inicial del prototipo final.

En la Figura 5-14 se muestra la pantalla de inicio de la aplicación recolectora para dispositivos Android que hemos detallado en este capítulo (BeeDroid). Como vemos, existen 4 acciones principales. Podemos acceder al apartado de formularios para visualizar los formularios disponibles o descargar nuevos formularios del servidor. Asimismo, podemos crear nuevas instancias de un formulario ya descargado. La opción de completar instancias aún no está disponible, pues no hemos desarrollado aún la herramienta que permita almacenar instancias en el teléfono de forma temporal. Finalmente, las opciones nos permiten modificar la configuración de la aplicación (cambiando, por ejemplo, la IP del servidor repositorio).

En el Capítulo 7 se describirá el prototipo en general, incluyendo al Recolector. Se ilustrará el sistema con un mayor número de capturas de pantalla y una explicación más detallada de cada una de ellas.

Capítulo 6. El administrador

Resumen:

- Se presenta uno de los tres módulos fundamentales del proyecto Turawet: El Administrador del repositorio, también conocido como BeeKeeper (apicultor).
- Detallamos las etapas de análisis, toma de requisitos y diseño, previas a la implementación.
- En la etapa de análisis presentaremos la necesidad de la integración de un cuadro de mandos.
- Se presentan diagramas UML (de casos de uso y de secuencia) y se detalla exhaustivamente la base de datos, así como la representación XML intermedia utilizada.
- En la etapa de implementación, se detallan las tecnologías utilizadas y las observaciones más relevantes.

6.1 Introducción

BeeKeeper, (de aquí en adelante Administrador), es un módulo del proyecto Turawet que a su vez se compone de dos sub-módulos, que son el repositorio y la aplicación de administración propiamente dicha.

El repositorio agrupa a la base de datos, los servicios web y los módulos de *parsing* que se encargan de pasar desde ficheros XML (bien sean de formularios o de instancias) a elementos en la base de datos.

Por otra parte, el administrador propiamente dicho es una aplicación web que permite a los administradores, gestores o ejecutivos acceder al listado de formularios que les compete y poder ver las instancias rellenas, unas estadísticas elaboradas a partir de dichas instancias o geolocalizar las instancias en un mapa. Además permite borrar instancias de un formulario o incluso el formulario completo.

6.2 Etapa de análisis

Para la administración del repositorio, en esta etapa se dio especial importancia a la toma de requisitos, centrada principalmente en la definición de los formularios e instancias, así como en la interconexión que tendría que existir entre éste módulo de almacenamiento y gestión y los otros dos módulos del proyecto (modelador y recolector) que deberían enviar o solicitar información al administrador del repositorio.

6.2.1 Requisitos

Como requisitos generales de este módulo tenemos:

- Permitir el almacenamiento estructurado y genérico de formularios e instancias en un mismo modelo de datos.
- Brindar una interfaz de comunicación que permita la interoperabilidad entre los distintos módulos del proyecto.

- Crear una aplicación con una interfaz sencilla, usable y accesible permitiendo a un usuario poder acceder cómodamente a los datos almacenados.
    - A través de la citada aplicación, permitir ver los formularios e instancias almacenadas.
    - Asimismo, permitir ver estadísticas de un formulario (en función de todas las instancias almacenadas de él).
    - Permitir también la visualización de todas las instancias geolocalizadas sobre un mapa.

La necesidad de un sistema de base de datos

Realizando la fase de análisis del proyecto y la toma de requisitos se dio especial importancia a la forma de almacenamiento de los datos recolectados. El gran valor añadido de este proyecto es la explotación de los datos que se hayan obtenido al rellenar las diferentes instancias de un formulario. Además de almacenar toda esa información, queríamos que su almacenamiento fuera estructurado y consistente; pudiendo recuperar la información, por campos, en cualquier momento.

Se consideró necesario que en el futuro se pudieran hacer consultas sobre los datos almacenados, por ejemplo obteniendo los resultados de todas las instancias de un formulario, para un determinado campo; o para mostrar estadísticas que permitieran tomar decisiones a un gestor, administrador o ejecutivo en función de la información obtenida de todas las instancias del formulario, que fueron rellenas a través de la herramienta recolección.

¿Por qué un módulo de estadísticas?

Como hemos dicho anteriormente, una de las características con mayor potencial del proyecto Turawet es la explotación de los datos recolectados en las instancias de formularios. Para permitir explotar los datos (consultándolos o haciendo *Data Mining* a través de otra herramienta) hemos se consideró necesario diseñar cuidadosamente y desplegar un meta-sistema de almacenamiento de datos. El sistema de base de datos que planteamos nos permite tener dividida y relacionada toda la información de las instancias y formularios, de manera que podemos extraer toda la información que deseemos haciendo consultas a este meta-sistema, bien mediante consultas directas a los datos (SQL[72]) o bien mediante una aplicación que haga las consultas y las plasme sobre una interfaz de usuario sencilla. La información relativa al diseño y despliegue de la base de datos las veremos en las posteriores etapas de diseño e implementación de este módulo del proyecto.

Teniendo en cuenta este desarrollo de base de datos con tan alto potencial para extraer información valiosa de los datos almacenados en ella, hemos decidido integrar en el proyecto una aplicación que sirva de cuadro de mando para gestores, administradores o ejecutivos de una empresa u organización que deseen obtener y explotar información de los datos recolectados y almacenados.

Dentro de la herramienta de administración del repositorio (Beekeeper) hemos integrado un sistema de estadísticas. Este sistema nos permite obtener estadísticas de un formulario concreto. Para ello se examinan los datos recolectados de todas las instancias de

dicho formulario y se elaboran diagramas circulares con estadísticas de aquellos campos con valores de respuesta limitados por el modelador (es el caso de los *radio buttons*, *combo boxes*, *check boxes,*...).

En definitiva, se valoró como un requisito importante de la aplicación de administración del repositorio el permitir obtener información estadística de los datos recolectados (en función del tipo de campo que se trate), para fomentar y facilitar así la toma de decisiones de gestores, administradores o ejecutivos en función de los resultados de las instancias de un formulario.

El valor añadido de la geolocalización

La geolocalización es un parámetro relevante que se encuentra en boga en multitud de disciplinas (desde la fotografía, hasta las estadísticas localizadas). Durante el análisis de las funcionalidades que serían interesantes para el proyecto, decidimos continuar con esta tendencia y ofrecer una herramienta que permitiera explotar los datos almacenados utilizando la geolocalización.

Permitir almacenar información geolocalizada de las instancias nos permitirá que, desde el cuadro de mando de la aplicación de administración, que trataremos en este capítulo, podamos representar gráficamente, en un mapa, dónde han sido rellenadas las diferentes instancias de un formulario, teniendo esto aplicación en gran multitud de campos.

Ilustrando la geolocalización de instancias procedemos a exponer varios ejemplos en los que sería de gran utilidad práctica. En la inspección técnica de edificios, un inspector rellenaría una instancia que, al estar geolocalizada tendría por si misma información de la ubicación del edificio que está valorando. Además, posteriormente a la inspección de muchos edificios, se podrían ver en un mapa todos los edificios que han sido inspeccionados, y potencialmente se podrían aplicar filtros por zonas, temporales (viviendas inspeccionadas un determinado mes) o por estados de las viviendas.

Otra aplicación práctica de la geolocalización vendría dada por la utilidad de localizar ejemplares de plantas pertenecientes a especies en peligro de extinción. Existen biólogos especializados en botánica que hacen, periódicamente, un inventario de los ejemplares existentes de dichas especies amenazadas. La utilidad de realizarlo con nuestra herramienta y, además, teniendo la oportunidad de ver en un mapa la situación geográfica de todos y cada uno de los ejemplares que ha localizado es muy interesante en este campo, para ver la evolución de las poblaciones a lo largo de los años (viendo como aparecen nuevos ejemplares o desaparecen algunos ya existentes) todo esto de forma fácil, cómoda y muy visual.

6.2.2 Diagramas UML

En primer lugar realizamos un diagrama UML general del administrador, donde se pueden observar las diferentes funcionalidades que han de estar disponibles en este módulo, que incluye el repositorio y la administración del mismo.

Diagrama de casos de uso

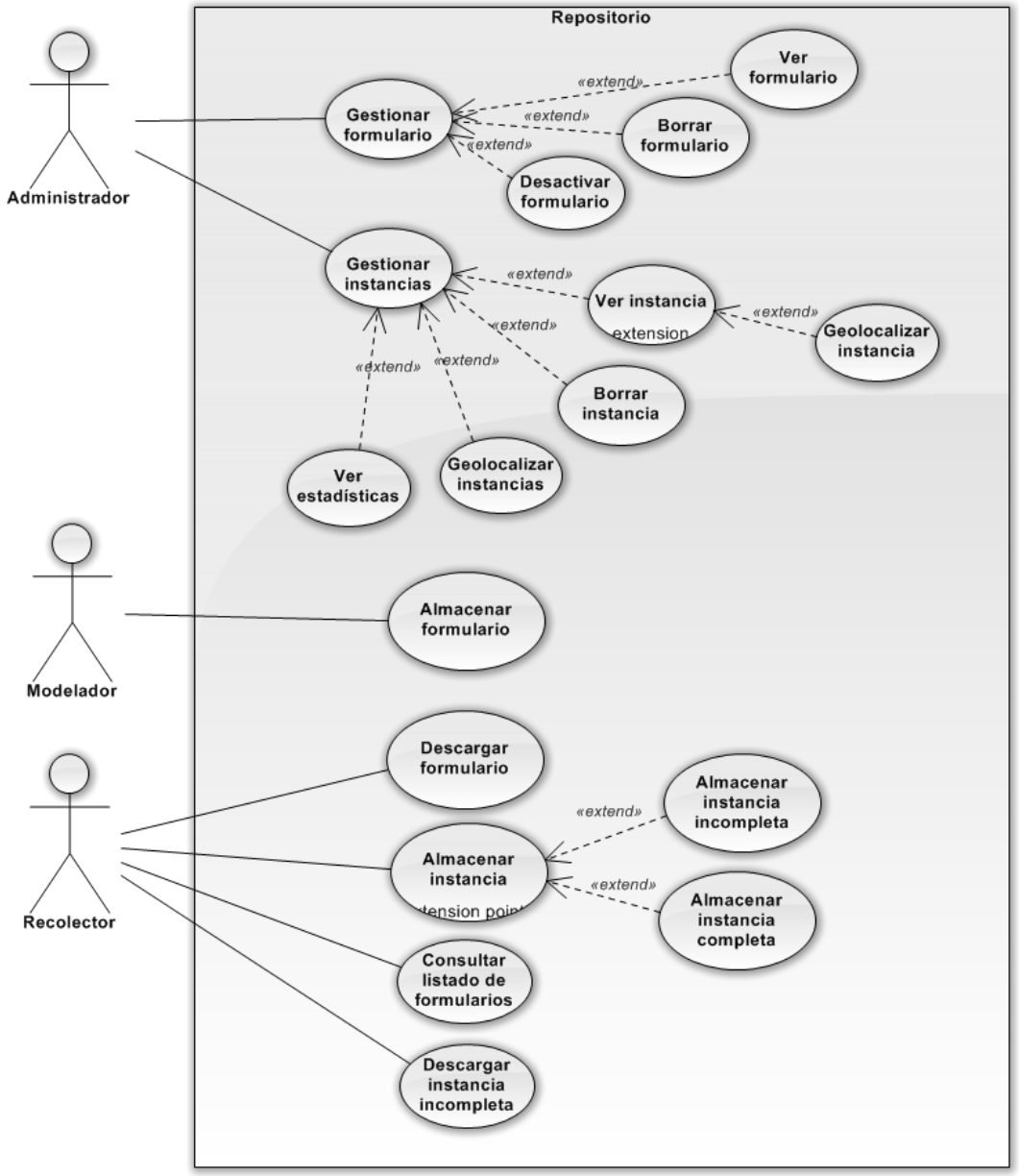


Figura 6-1 Diagrama de casos de uso del administrador del repositorio

En el diagrama anterior se muestran los actores del subsistema de administración del repositorio así como los principales casos de uso que tienen lugar en este módulo.

El actor **administrador**, que también engloba a gestores o ejecutivos, es el único que accede directamente a la interfaz de usuario web del administrador de repositorio para realizar las acciones de eliminación o desactivación de formularios, así como visualización de estadísticas o de instancias geolocalizadas.

Por otra parte, el actor **Modelador** hace referencia a la aplicación de modelado que se conecta, vía servicio web, con el administrador de repositorio para enviar los formularios.

Finalmente, el actor **recolector** no es otro que la aplicación de recolección (móvil o web) que se conecta, también por servicio web, al repositorio para descargar formularios o instancias incompletas. También se puede conectar para enviar nuevas instancias que se han rellenado o instancias que han sido completadas recientemente y que en el repositorio se encontraban incompletas.

Descripción de los casos de uso

Para clarificar el diagrama de casos de uso anterior, llevaremos a cabo una pequeña explicación de cada uno de los principales casos de uso involucrados.

Identificador: *CU-A-3.*

Nombre: “Borrar formulario”.

Descripción

El administrador decide eliminar completamente un formulario y todas sus instancias de la base de datos.

Actores

Administrador (Gestor o ejecutivo).

Precondiciones

El usuario se ha identificado como administrador, gestor o ejecutivo en la aplicación de administración del repositorio.

Post-condiciones

1. Se elimina el formulario de la base de datos.
2. Se eliminan todas las instancias del formulario eliminado.

Frecuencia

Muy poco frecuente**.**

Flujo de escenario principal

1. El usuario se ha identifica como administrador, gestor o ejecutivo.
2. El usuario consulta los formularios disponibles.
3. El usuario selecciona el formulario deseado y hace *click* en “eliminar formulario”.
4. El usuario confirma la eliminación.

Identificador: *CU-A-4.*

Nombre: “Desactivar formulario”.

Descripción

El administrador decide desactivar un formulario, imposibilitando así la creación de nuevas instancias del mismo.

Actores

Administrador (Gestor o ejecutivo).

Precondiciones

El usuario se ha identificado como administrador, gestor o ejecutivo en la aplicación de administración del repositorio.

Post-condiciones

1. Se modifica el atributo “activo” del formulario, cambiándose a falso.

Frecuencia

Poco frecuente**.**

Flujo de escenario principal

1. El usuario se ha identifica como administrador, gestor o ejecutivo.
2. El usuario consulta los formularios disponibles.
3. El usuario selecciona el formulario deseado y hace *click* en “desactivar formulario”.

Identificador: *CU-A-8.*

Nombre: “Borrar instancia”.

Descripción

El administrador decide eliminar una instancia concreta de un formulario.

Actores

Administrador (Gestor o ejecutivo).

Precondiciones

El usuario se ha identificado como administrador, gestor o ejecutivo en la aplicación de administración del repositorio.

Post-condiciones

1. Se elimina la instancia correspondiente.

Frecuencia

Poco frecuente**.**

Flujo de escenario principal

1. El usuario se ha identificado como administrador, gestor o ejecutivo.
2. El usuario consulta los formularios disponibles.
3. El usuario selecciona uno de los formularios y consulta las instancias del mismo.
4. El usuario selecciona la instancia deseada y hace *click* en “eliminar instancia”.

Identificador: *CU-A-11.*

Nombre: “Almacenar formulario”.

Descripción

El modelador envía un formulario, vía servicio web, al repositorio para su almacenamiento. Se procederá al análisis sintáctico del mismo y posterior despliegue en la base de datos.

Actores

Modelador.

Precondiciones

Se recibe un formulario desde la aplicación de modelado.

Post-condiciones

1. Se almacena el formulario (desplegado) en la base de datos.

Frecuencia

Frecuente**.**

Flujo de escenario principal

1. Se recibe un formulario desde el modelador vía servicio web.
2. Se procede a un análisis sintáctico del formulario recibido.
3. Si el formulario es correcto se despliega en la base de datos.
4. Se envía un mensaje de confirmación o de error al modelador.

Identificador: *CU-A-15.*

Nombre: “Almacenar instancia completa”.

Descripción

El recolector envía una instancia completa, vía servicio web, al repositorio para su almacenamiento. Se procederá al análisis sintáctico de la misma y posterior despliegue en la base de datos, teniendo en cuenta que está completa.

Actores

Recolector.

Precondiciones

Se recibe una instancia desde la aplicación de recolección de datos.

Post-condiciones

1. Se almacena la instancia (desplegada) en la base de datos.
2. Se cambia el atributo “está completa” de la instancia a verdadero.

Frecuencia

Muy frecuente**.**

Flujo de escenario principal

1. Se recibe una instancia desde el recolector vía servicio web.
2. Se procede a un análisis sintáctico de la instancia recibida.
3. Si la instancia es correcta se despliega en la base de datos.
4. Se envía un mensaje de confirmación o de error al recolector.

Diagramas de secuencia

A continuación observaremos varios diagramas de secuencia del administrador del repositorio que ilustran algunos de los casos de uso mencionados anteriormente.

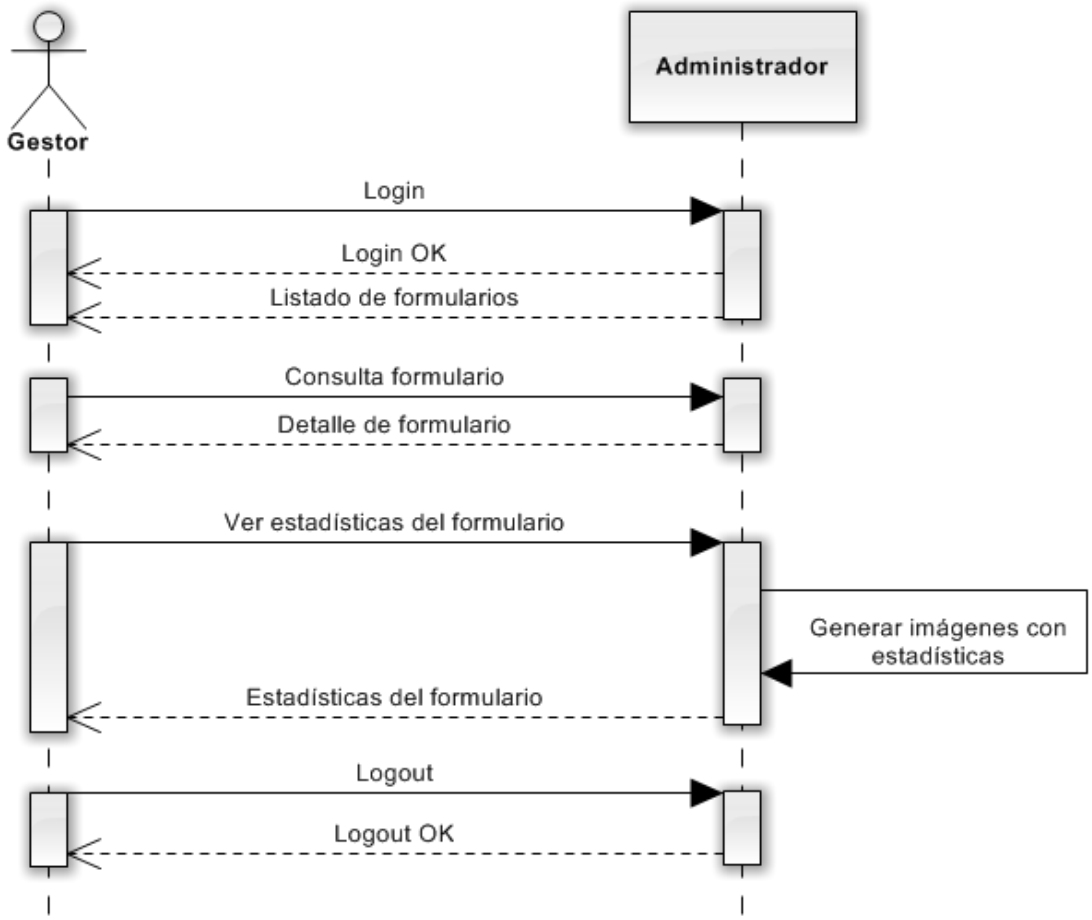


Figura 6-2 Diagrama de secuencia de ver estadísticas

En el diagrama anterior se ilustra cómo un usuario administrador, gestor o ejecutivo puede visualizar las estadísticas de un formulario. Al ser una secuencia de eventos muy sencilla sólo cabe resaltar el evento “Generar imágenes con estadísticas”. Éste evento es el encargado de llamar a la aplicación correspondiente para que genere dinámicamente las estadísticas del formulario consultado, recurriendo a la información de todas las instancias de dicho formulario.

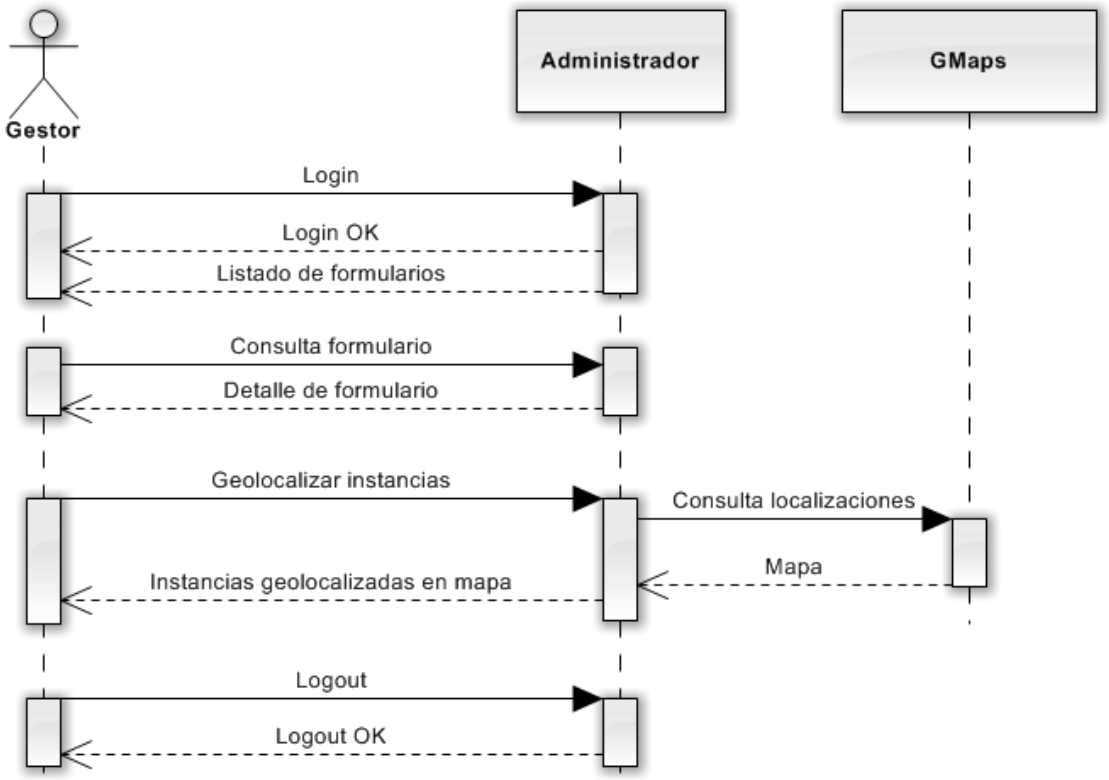


Figura 6-3 Diagrama de secuencia de consultar geolocalización

En el diagrama anterior se ilustra cómo un usuario administrador, gestor o ejecutivo puede visualizar las instancias geolocalizadas de un formulario en un mapa. Una vez que se selecciona “geolocalizar instancias” se llama a un módulo o aplicación encargada de dibujar en el mapa dichas instancias. Para ello es necesario conectarnos con las API de *Google Maps*, como se refleja en el evento “Consulta localizaciones”. A estas API le enviaremos una serie de puntos que ellas nos dibujarán en un mapa. Dicho mapa nos será devuelto para poderlo mostrar al usuario.

6.3 Etapa de diseño

La parte fundamental para el diseño del repositorio y su administración es el esquema de base de datos, así como la forma de insertar y consultar los datos en la misma. En este punto, también tiene gran importancia el esquema XML que se ha de definir para formularios e instancias y que será utilizado como formato intermedio de almacenamiento de los datos.

Sin embargo, también existen aspectos a tener en cuenta en la etapa de diseño de cara a la herramienta de administración del repositorio, como son la interfaz y las diferentes funcionalidades que debe ofrecer al usuario.

6.3.1 Decisiones de diseño

La base de datos

En la etapa de diseño realizamos varios esquemas de la Base de Datos, depurando constantemente las versiones anteriores. En todo momento primó el esfuerzo de diseñar una base de datos que nos permitiera extraer la información sin grandes costes, manteniendo en todo lo posible la normalización.

A continuación mostramos la versión final del diagrama ER de la Base de datos, describiendo primero las entidades, posteriormente las relaciones y por último los comportamientos ante acciones de modificación y eliminación sobre elementos de una entidad.

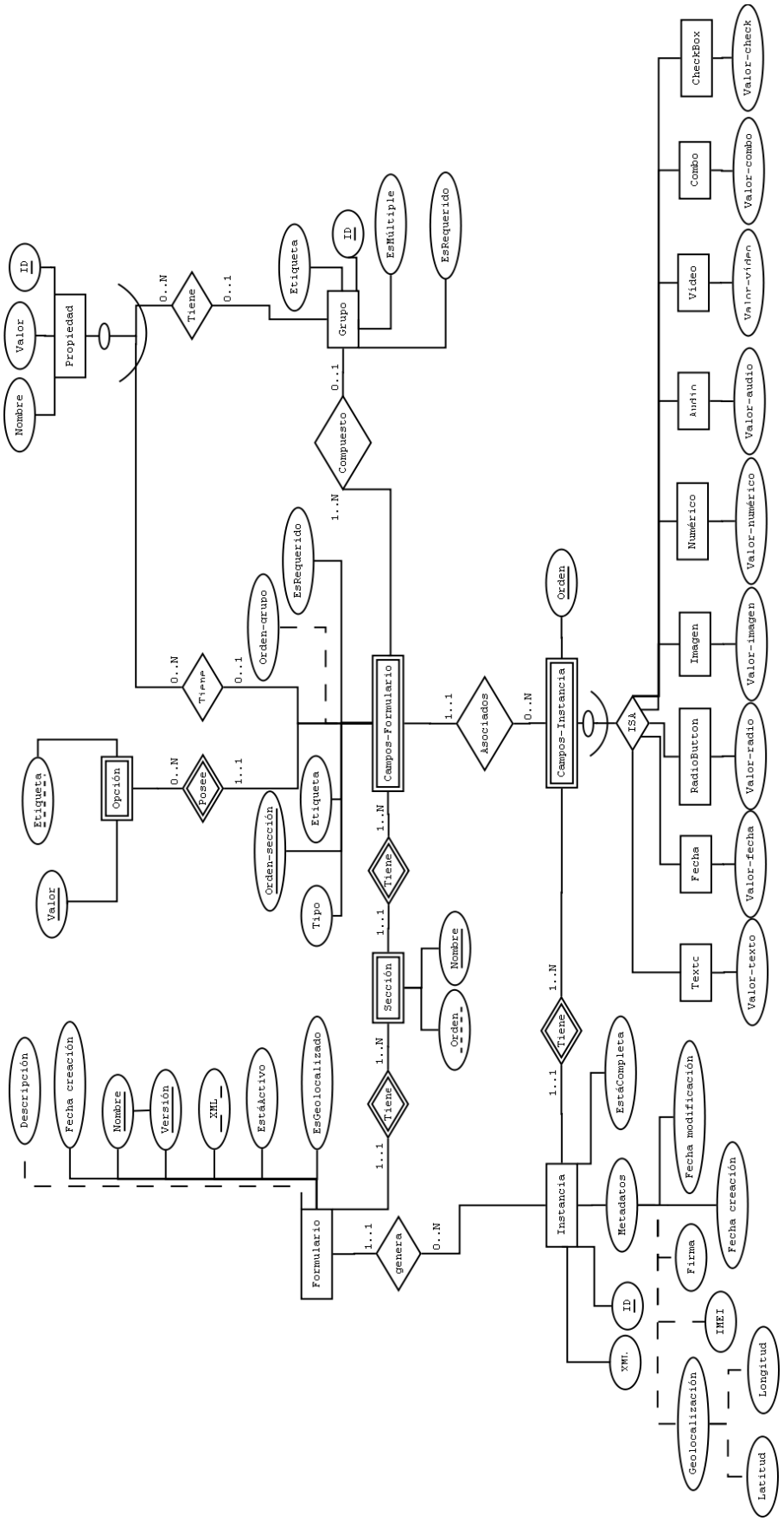


Figura 6-4 Modelo ER de la Base de Datos del repositorio.

Una de las principales entidades es **Formulario**. Esta entidad, como su propio nombre indica, describe un formulario (principalmente sus metadatos). Entre sus atributos se encuentran su nombre, versión y descripción. Además tendremos un campo *booleano* que nos indica si el formulario está activo o no; y un campo de tipo cadena de caracteres que almacenará el XML del formulario recibido desde el modelador. Cabe destacar que en nuestro modelo la gran mayoría de los campos no admiten valores nulos. Aquellos que sí los admiten serán comentados cuando les corresponda.

El objetivo de almacenar el XML como un campo de tipo cadena de caracteres no es otro que el de agilizar las descargas de los formularios desde la aplicación móvil (no teniendo que reconstruir el formulario cada vez que se quiera realizar una descarga). Es preciso tener en cuenta que un formulario no se modifica nunca. Si se quiere modificar un formulario ya existente, tendremos que crear uno nuevo basado en el anterior. Esta nueva creación es considerada como una nueva versión del formulario (aunque a todos los efectos es como un formulario nuevo con la salvedad de tener el mismo nombre que su predecesor y un número de versión superior en una unidad).

Un formulario se identifica por la pareja nombre-versión, si bien cabe destacar que, a la hora de implementar la base de datos, esta entidad, al igual que todas las demás, tendrán un atributo ID que añadirá el *framework* de desarrollo utilizado y que agilizará las consultas.

Cada formulario tiene varias **Secciones**. Éstas representan otra entidad en nuestro modelo, cuyos únicos campos son nombre y orden (dentro de un formulario).

Como peculiaridad de ésta entidad tenemos que es una entidad débil de Formulario con una relación por identificación (relación que explicaremos en detalle más adelante). Una sección se identifica dentro de un formulario con su nombre (no hay más de una sección con un determinado nombre en un formulario concreto), pero también se puede identificar con el orden (pues cada sección tiene su propio orden dentro de un formulario).

A su vez, una sección tiene campos, que hemos llamado en el diagrama **Campos-Formulario**, para diferenciarlos de los Campos-Instancia, que son los que contienen los valores y que se verán más adelante. Ésta nueva entidad (Campo-Formulario) es a su vez una entidad débil de la sección; y al igual que le pasa a la sección con el formulario, un campo-formulario tiene una relación de identificación con la sección a la que pertenece.

Cada campo-formulario posee una descripción de sí mismo. Tiene como atributos el tipo, la etiqueta (texto descriptivo del campo) y si es requerido o no. Además, el atributo “orden-sección” es un campo que indica el orden que tiene un campo-formulario dentro de una sección, y que identifica unívocamente al campo dentro de dicha sección.

El atributo “orden-grupo” especifica el orden de un atributo dentro de un grupo, en caso de que pertenezca a uno, con lo que es de los pocos campos que permiten valores nulos en nuestro modelo.

Cada campo-formulario puede tener varias propiedades u opciones. Las opciones las tendrán aquellos campos de tipo similar a *“Radio-button”* o *“Combo”*. Por otra parte, todos los campos son susceptibles de tener propiedades (como longitud máxima, en caracteres, para un campo de texto).

La entidad **opciones** se compone de dos atributos (etiqueta-valor), que se ilustran sencillamente utilizando el ejemplo anterior: “longitud-máxima: 100 caracteres”.

Las **propiedades**, por su parte, también poseen una pareja de atributos nombre-valor; pero además poseen un identificador numérico en el modelo, debido a que la pareja nombre-valor por sí misma no es un identificador, al poder existir dos propiedades con el mismo nombre-valor siendo una de un grupo y otra de un campo.

En cuanto a la entidad **grupo**, sus atributos son la etiqueta del grupo (un texto descriptivo que defina al grupo), y dos atributos *booleanos* que indiquen si es requerido (todos sus campos han de ser rellenados) o si es múltiple (indica que estamos ante una lista de elementos de este tipo).

La identificación de un grupo viene dada por un campo identificador (ID), debido a que es una entidad independiente y puede haber grupos con la misma etiqueta en diferentes formularios (cosa que no podemos controlar desde la propia entidad grupo).

La parte inferior del diagrama corresponde con las instancias de formularios. En primer lugar tenemos la entidad **instancia**, que tiene como atributos todos los metadatos que corresponden a una instancia, como son: geolocalización (que incluye latitud y longitud), el IMEI del teléfono que ha rellenado la instancia (en caso de que sea un teléfono), la firma electrónica (en caso de que se pueda o requiera) y las fechas de creación y modificación. Además, se almacena un atributo booleano que indica si la instancia está completa o no y un atributo de tipo cadena de caracteres que almacenará el XML de la instancia, para hacer más eficiente el proceso de envío al teléfono. Este envío tendrá lugar si se desea hacer una modificación de alguna de las instancias almacenadas en la base de datos.

La identificación de una instancia viene dada por un atributo identificador que hemos definido (ID). Esto sucede al no tener definida, esta entidad, ninguna forma clara de identificarse a través de sus atributos.

A su vez, cada instancia está relacionada con muchos **campos-instancia**. Un campo-instancia es otra entidad que representa el valor concreto de un campo-formulario en la instancia actual. Se identifica, dentro de la instancia, por un atributo orden. Este campo difiere del orden de un campo-formulario debido a que pueden aparecer campos dinámicos. Por ejemplo, si estamos rellenando una instancia de un formulario que tiene un grupo múltiple llamado “Miembro de la unidad familiar”, el cual posee un campo nombre y otro DNI, al rellenar la instancia introduciremos tantos campos instancia como miembros tenga la unidad familiar correspondiente, estando relacionados todos estos campos-instancia con el mismo campo-formulario. Sin embargo cada campo-instancia del mismo campo-formulario tendrá su propio orden dentro de la instancia.

Hay **muchas entidades que heredan** de la entidad campo-instancia. Concretamente, hay tantas como tipos de datos permitimos en nuestro modelo (texto, radio, imagen, audio, vídeo,…). Cada una de estas entidades se diferencia de las otras en el tipo de su único atributo, que no es otro que el valor. El tipo de dicho atributo será una cadena de caracteres para el texto, una fecha para el de tipo fecha o una imagen para el correspondiente tipo imagen.

Además de las entidades comentadas anteriormente, es preciso explicar las relaciones que las unen en el diagrama. Así, en primer lugar tenemos la interrelación “**tiene**”, que asocia

secciones a un formulario. Como ya comentamos anteriormente, es una relación por identificación, donde se indica que una sección se identifica por sí misma dentro de un formulario. Esto quiere decir que la identificación de una sección vendrá dada por el atributo identificador principal de la entidad sección propiamente dicha (orden-sección), concatenada con el correspondiente identificador del formulario (la pareja nombre-versión).

Prosiguiendo con la descripción de la interrelación que asocia las secciones con el formulario, observamos que una sección pertenecerá obligatoriamente a un formulario y además sólo a un formulario. Por otra parte, un formulario para tener sentido ha de tener una sección como mínimo y puede tener todas las que se quieran.

La siguiente interrelación a describir es la también apodada "**tiene**" que asocia las secciones con los campos-formulario. También es una relación por identificación y en este caso nos indica que un campo-formulario está asociado obligatoriamente a una sección y sólo a una y, por otra parte, una sección ha de tener al menos un campo-formulario para ser admitida y puede tener todos los deseados.

La interrelación "**posee**" es la que enlaza a un campo sus opciones permitidas (para los casos de selección de una opción entre varias, como los *radio* o *combo*). Al ser también una relación por identificación, tenemos que una opción se asocia a uno y solo uno de los campo-formulario, mientras que los campos-formularios pueden tener múltiples opciones; no estando obligados a tener una (por ejemplo, los campos de tipo texto no tienen opciones).

Prosiguiendo con la descripción de las interrelaciones, llegamos a la siguiente, llamada nuevamente "**tiene**" y que relaciona un campo-formulario con las propiedades que pueda tener asociadas. Como vemos, un campo-formulario puede no tener ninguna propiedad asociada o tener muchas; mientras que una propiedad puede ser, o bien de un campo-formulario, o bien de un grupo. Esta restricción se representa mediante la exclusividad y totalidad que se encuentra bajo la entidad propiedad entre las dos interrelaciones que llegan a ella (la que hemos descrito previamente con los campos-formulario y otra equivalente, también llamada "**tiene**" hacia los grupos). De esta forma, una propiedad siempre será necesariamente o bien de un campo o bien de un grupo.

A su vez, existe una interrelación encargada de establecer una conexión entre los campos-formularios y los grupos llamada "**Compuesto**". Un campo-formulario puede estar o no en un grupo, pero como mucho podrá estar en uno (no se permite anidamiento de grupos). Cabe destacar en este apartado que si deseamos hacer una lista, aunque sea de un único campo (por ejemplo una galería de imágenes) lo que deberemos hacer es crear un grupo cuyo atributo "esMúltiple" sea verdadero (para el ejemplo de la galería de imágenes crearemos un grupo múltiple que tendrá solamente un campo-formulario asociado de tipo imagen).

Continuando con la parte inferior del diagrama, tenemos otra relación "**genera**" que enlaza a los formularios con las instancias. Observando esta interrelación en detalle se extrae que cada instancia tiene que pertenecer y pertenece a un único formulario (como es lógico). Por su parte, un formulario puede tener muchas instancias o no tener ninguna.

A su vez, las instancias deben estar relacionadas con los campos-instancia. De esta necesidad surge otra interrelación "**tiene**". Cada instancia tendrá al menos un campo-instancia, pudiendo tener muchos. Al contrario, un campo-instancia siempre estará asociado a una instancia, lo que se observa además por depender de la instancia en identificación, siendo el

identificador principal de un campo-instancia la pareja formada por la ID de la instancia concatenada con el orden del campo-instancia.

La siguiente relación que podemos observar es "**Asociados**", que indica a qué campo-formulario está asociado un campo-instancia. En este punto vemos que un campo-instancia también depende en existencia de un campo-formulario (además de de la instancia) y por ello un campo-instancia está asociado exclusivamente a un campo-formulario. A su vez, un campo-formulario puede tener múltiples campos-instancia asociados (en caso de listas) pudiendo no tener ninguno asociado en caso de que no haya instancias del formulario.

Por último nos queda la relación **ISA**. Ésta es la relación propia de la herencia de un campo-instancia hacia cada uno de los posibles tipos concretos de dicho campo. Es una relación con totalidad (un campo instancia tiene que ser de uno de los tipos hijo) y con exclusividad (no puede ser de dos tipos hijo a la vez).

En cuanto a las **modificaciones y borrados** de las diferentes entidades, hemos concluido que el comportamiento debe ser el de restringir en las modificaciones y propagar en cascada en las eliminaciones. Por ejemplo, si un formulario es eliminado, entonces ni sus secciones, ni sus campos ni sus instancias tienen sentido, con lo que han de ser eliminadas también. Otro ejemplo de este caso es: si se elimina un campo-formulario, ni sus opciones ni sus propiedades tienen sentido, con lo que se ha de propagar la eliminación. Por otra parte, en ningún caso tiene sentido una modificación de los elementos de una relación. Ilustrando esta afirmación, no tiene sentido cambiar la sección a la que pertenece un campo-formulario, debido a que hemos dicho que un formulario finalizado es inmutable. Lo mismo sucede con los campos instancia: pueden cambiar sus valores, pero no la instancia a la que pertenecen ni el campo-formulario que les define.

La gestión de usuarios

Durante el proceso de diseño, también se planteó la posibilidad de incluir una gestión de usuarios completa para el sistema. Desde un primer momento se la consideró una idea interesante, pero que estaría en un segundo nivel de prioridad, pues el interés principal del proyecto era conseguir hacer funcionar un primer prototipo de la aplicación. Un primer prototipo que fuera sencillo, ilustrara los aspectos más importantes y novedosos del proyecto y que necesariamente no requería tener implementada una gestión de usuarios, al considerarse una funcionalidad en segunda línea de importancia.

Sin embargo, realizamos un diseño de lo que debería de ser esta gestión de usuarios. A continuación se plasma una extensión el esquema Entidad-Relación anteriormente descrito, incluyendo en este caso solamente la parte de gestión de usuarios.

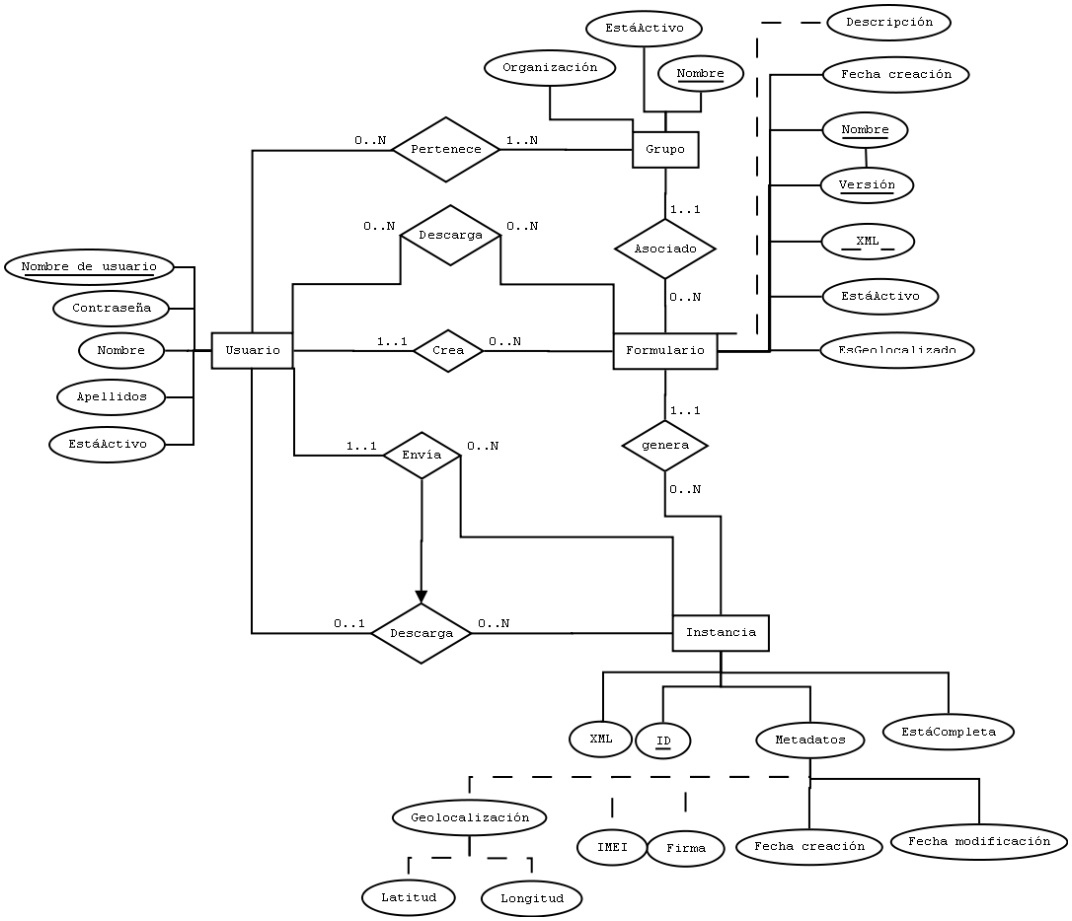


Figura 6-5 Diagrama ER para la gestión de usuarios

En la Figura 6-5 vemos que incluir la gestión de usuarios añadiría dos nuevas entidades al modelo: Usuario y Grupo. La entidad **Usuario** tiene como atributos principales propuestos: el nombre de usuario (que lo identifica unívocamente), la contraseña, el nombre real de la persona, sus apellidos y un atributo que indica si está activo o, si por el contrario, la cuenta ha sido deshabilitada. Por otra parte, la entidad **Grupo** sólo posee dos atributos: el nombre de dicho grupo (que es el identificador principal de la entidad) y la organización a la que pertenece (por ejemplo, un nombre de grupo puede ser “profesores de la ULL” y su organización “Universidad de La Laguna”).

Lo que sí se aprecia en este nuevo diagrama es un incremento sustancial del número de interrelaciones. La primera interrelación que describiremos es “**Pertenece**”, que relaciona un usuario con un grupo. Un usuario está obligatoriamente en un grupo como mínimo, pudiendo estar en muchos de ellos. Asimismo, a un grupo pueden pertenecer muchos usuarios, pero también puede estar vacío (de usuarios).

La siguiente interrelación que describiremos es “**Asociado**”. Un grupo puede tener asociados desde cero a muchos formularios, mientras que un formulario siempre se encuentra asociado necesariamente a un único grupo en nuestro modelo, aunque esto puede verse

modificado en el futuro (por ejemplo: el formulario “Inspección técnica de edificios” se encontrará asociado, únicamente al grupo “Inspectores de urbanismo”).

Por su parte, la interrelación “**Crea**” describe las creaciones de formularios. Un usuario puede no crear ningún formulario o crear muchos, pero un formulario siempre ha de tener un creador y sólo uno, que es la persona que modeló el formulario.

Además de crear formularios, los usuarios pueden descargarlos. El historial de descargas puede ser interesante para futuros análisis de los usuarios. Por ello, en nuestro diagrama aparece la interrelación “**Descarga**” entre usuario y formulario. Un formulario puede ser descargado por muchos usuarios o no ser descargado. Por su parte, un usuario puede descargar todos los formularios que quiera, sin estar obligado a realizar ninguna descarga.

En cuanto a las instancias se refiere, la primera interrelación es “**Envía**”. Un usuario envía desde cero a muchas instancias, siendo una instancia concreta únicamente enviada por un usuario, que es el que ha recolectado dichos datos.

La interrelación “**Descarga**” de las instancias depende en inclusión de la interrelación “Envía”. Esto quiere decir que para poder descargarnos una instancia (para editarla) tenemos que haber sido nosotros los que la hayamos creado (enviado). Para las descargas de instancias tenemos que un usuario puede descargar muchas instancias, siempre que haya sido él quién las haya enviado. Asimismo, una instancia, una vez enviada podrá ser o no descargada para modificación por el usuario que la envió, pero en caso de ser descargada, podrá serlo sólo por dicho usuario.

En esta parte del modelo existen algunas **restricciones semánticas adicionales** (RSA) que se tendrán que controlar y que no pudimos representar en el diagrama:

1. Al crear un formulario, el grupo que se le asigne al formulario ha de ser uno de los que sea miembro el usuario que está creándolo.
2. Un usuario sólo puede descargar formularios que estén asociados a grupos a los que él pertenece.

En cuanto a las **modificaciones y borrados** de las diferentes entidades, en el caso de la gestión de usuarios, necesitaremos realizar un análisis un poco más cuidadoso en el diagrama anterior de la base de datos.

Para las interrelaciones “crea” y “envía” tanto el borrado como la modificación quedan restringidos. No permitimos cambiar quién fue el creador de un formulario concreto o el que envió una instancia concreta si ya se ha almacenado. De igual forma, no permitiremos el borrado de usuarios si ya han creado formularios o han rellenado instancias. Lo que haremos, si queremos desactivar una cuenta de este tipo de usuarios, es modificar el atributo “EstáActivo” y ponerlo a falso. Lo mismo sucede para la interrelación “Descarga” de la instancia.

Para la interrelación “asociado” seguimos la misma política asociada a los grupos: restringir las modificaciones y las eliminaciones. Si un grupo desaparece se marcará como inactivo, pero no se borrará a menos que no tenga formularios asociados.

Para la interrelación “descarga” (de los formularios) tenemos que tener en cuenta dos casos: qué ocurre si se modifica/elimina el usuario y qué ocurre si se modifica/elimina el formulario. En este caso, tanto para el caso del usuario como para el caso del formulario las

modificaciones están restringidas y los borrados se propagan en cascada. No tiene sentido que queramos cambiar el usuario que se descargó el formulario ni el formulario que se descargó. En el caso del borrado, permitimos que se realice el borrado, pues no tiene sentido almacenar las descargas de un formulario que ya no existe; de la misma manera que no tiene sentido almacenar las descargas de un usuario que ya no existe.

Para el caso de la interrelación “pertenece” sucede lo mismo con usuarios y grupos. Para el caso de las modificaciones, las restringimos y para el caso de los borrados, los propagamos en cascada. Explicando el caso de los borrados, que puede crear más confusión, si un usuario es eliminado, para nada queremos conservar la información de los grupos a los que pertenece. Asimismo, si un grupo es eliminado, no nos interesa conocer los usuarios que pertenecían a dicho grupo.

6.3.2 Mockups

También en esta etapa de diseño dimos importancia a la interfaz, desarrollando varias versiones de *mockups* hasta llegar a una aproximación que nos pareciera lo suficientemente intuitiva y sencilla para el usuario.

Descripción de un formulario almacenado

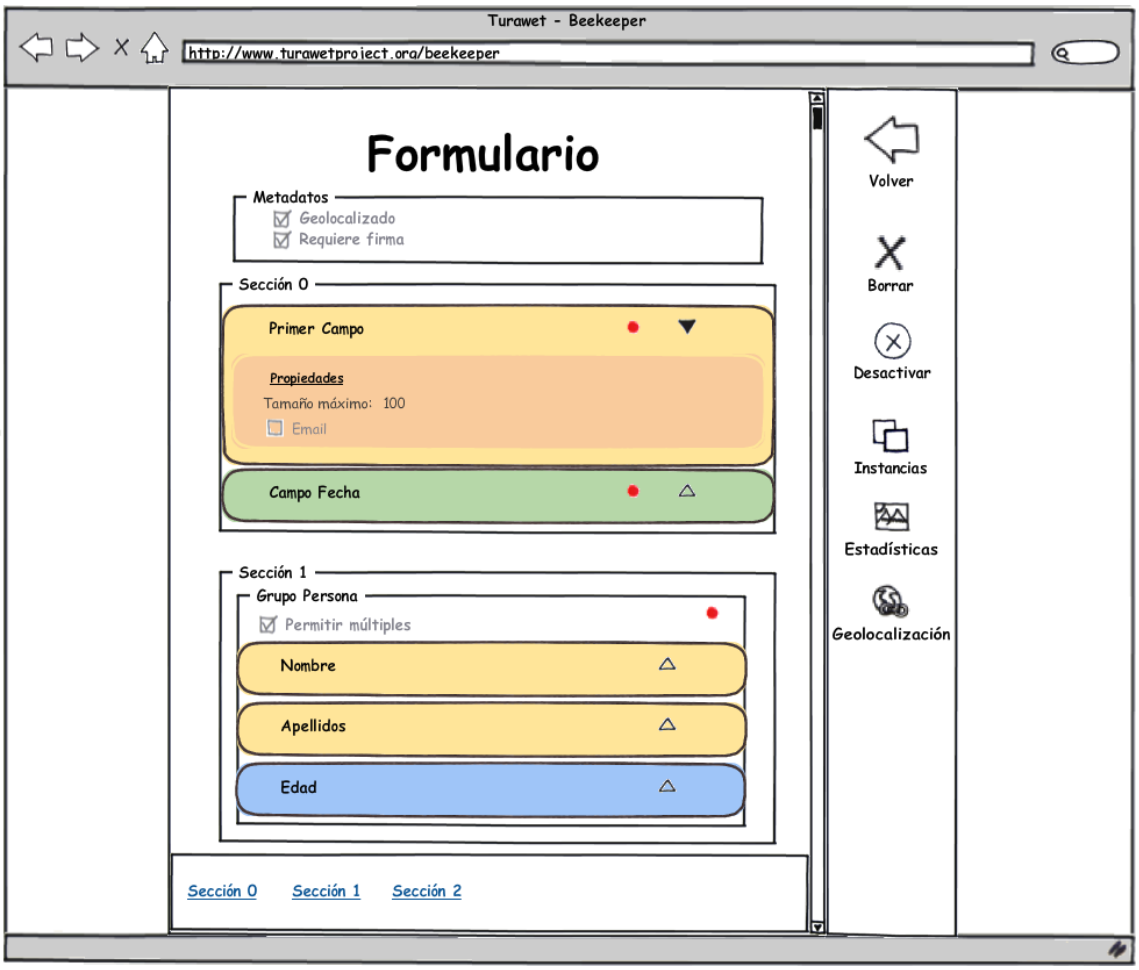


Figura 6-6 Mockup de un formulario en el Administrador

En el *mockup* anterior se ilustra un ejemplo de visualización de un formulario almacenado en el repositorio. Como podemos observar, la pantalla se halla dividida en 3 secciones principales: la central (donde se muestra el formulario con sus campos), la barra inferior (donde se muestran enlaces a las diferentes secciones del formulario) y por último la barra de opciones, que nos muestra las opciones disponibles en la pantalla actual.

En la parte central, lo primero que observamos es el nombre del formulario. A continuación nos aparece la primera sección, que no es una sección propiamente dicha, sino una meta-sección que hemos incluido, donde se muestran los metadatos del formulario (si es geolocalizado, si requiere firma, quién fue el autor,…). El resto de secciones son las secciones creadas al modelar. De ellas podemos ver los campos con sus opciones y propiedades.

La parte inferior, como hemos indicado previamente no es más que una lista de enlaces a las secciones del formulario. Al pulsar en cualquiera de los enlaces a una sección se llevará a cabo un desplazamiento en la parte central de la pantalla, colocándonos en el comienzo de la sección deseada.

Por último, la barra de la parte derecha agrupa todas las acciones que puede realizar el usuario al visualizar una instancia.

Estadísticas de un formulario

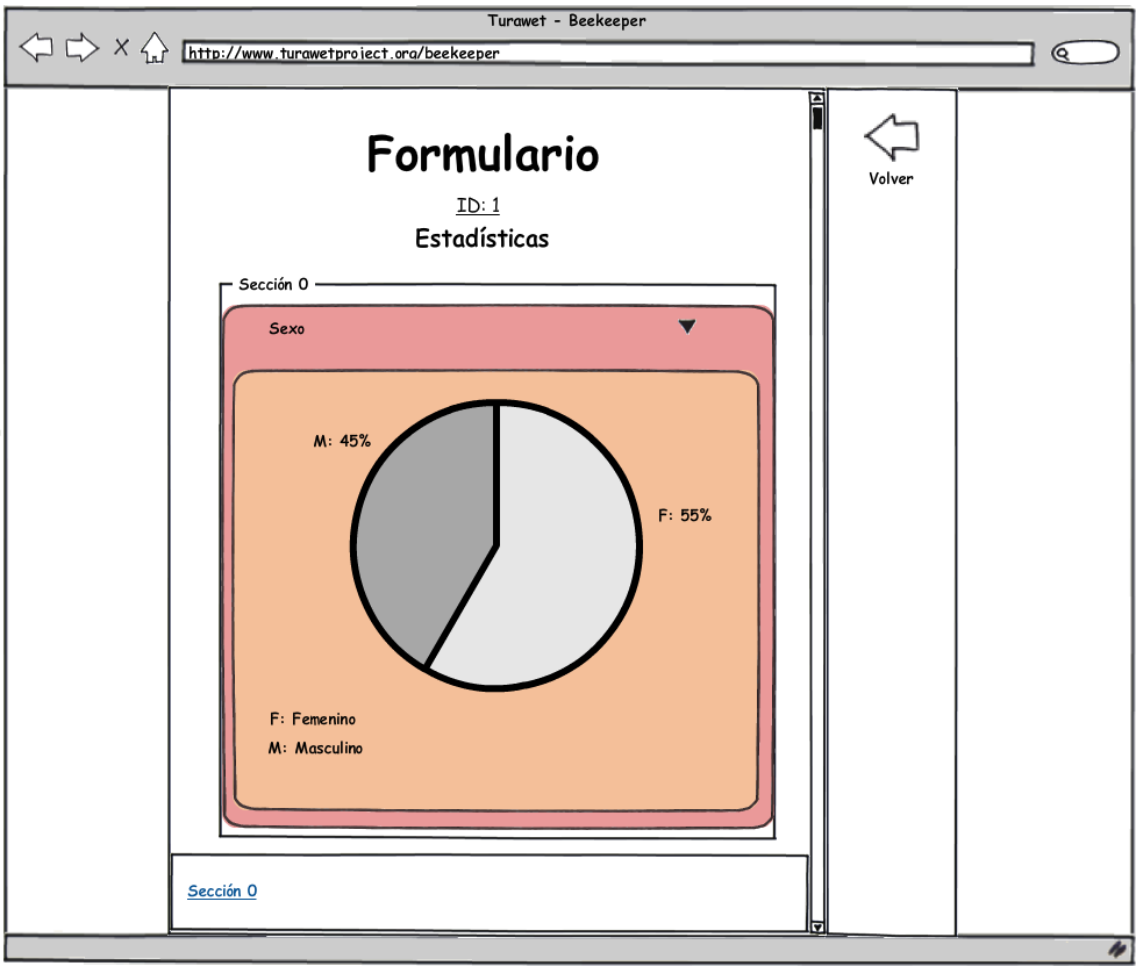


Figura 6-7 Mockup de estadísticas de un formulario en el BeeKeeper

En este caso nos encontramos visualizando las estadísticas de un formulario concreto. Las divisiones de la interfaz coinciden con las explicadas anteriormente. En este caso sólo se visualizarán los campos con valores múltiples (*combo, radio, check*) por ser los que mejor nos permiten realizar un análisis estadístico sobre ellos. Para cada uno de los campos que se muestren se podrá consultar un diagrama circular donde podamos apreciar el porcentaje correspondiente a cada una de las opciones elegidas. También contemplamos la realización de diagramas de barras para poder mostrar, en términos absolutos, la aceptación de las diferentes opciones.

Geolocalización de las instancias de un formulario

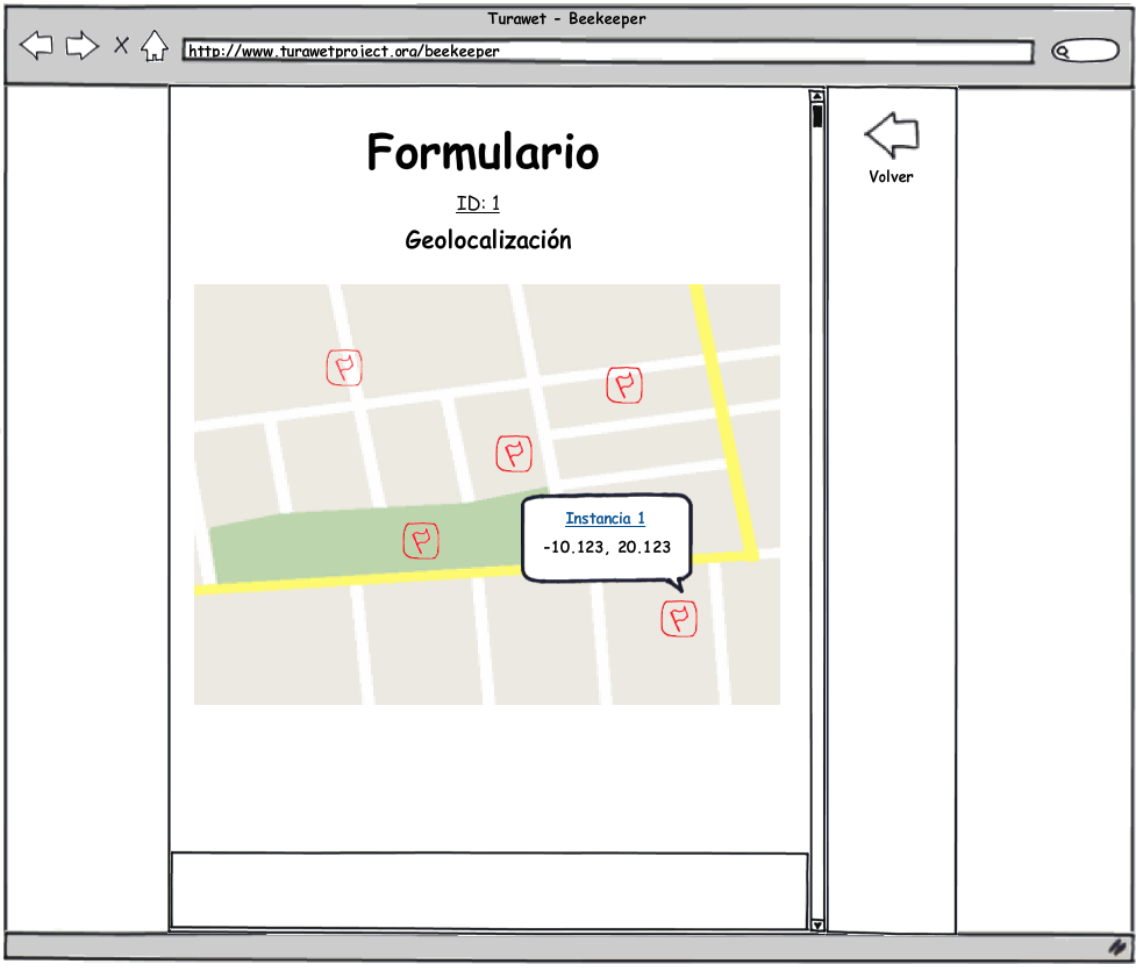


Figura 6-8 Mockup de geolocalización de las instancias de un formulario

Este *mockup* ilustra la geolocalización de todas las instancias de un formulario. Al presionar sobre cualquiera de los iconos que representan una instancia obtendremos información de la misma. Ésta información será un enlace a la instancia (donde podamos ver los valores rellenos de cada uno de los campos del formulario) y una imagen (en caso de que la instancia tenga una). La imagen tendrá utilidad en algunas de nuestras implementaciones de ejemplo, como la aplicación para la notificación de desperfectos o en la de fichas de plantas amenazadas.

Descripción de una instancia almacenada

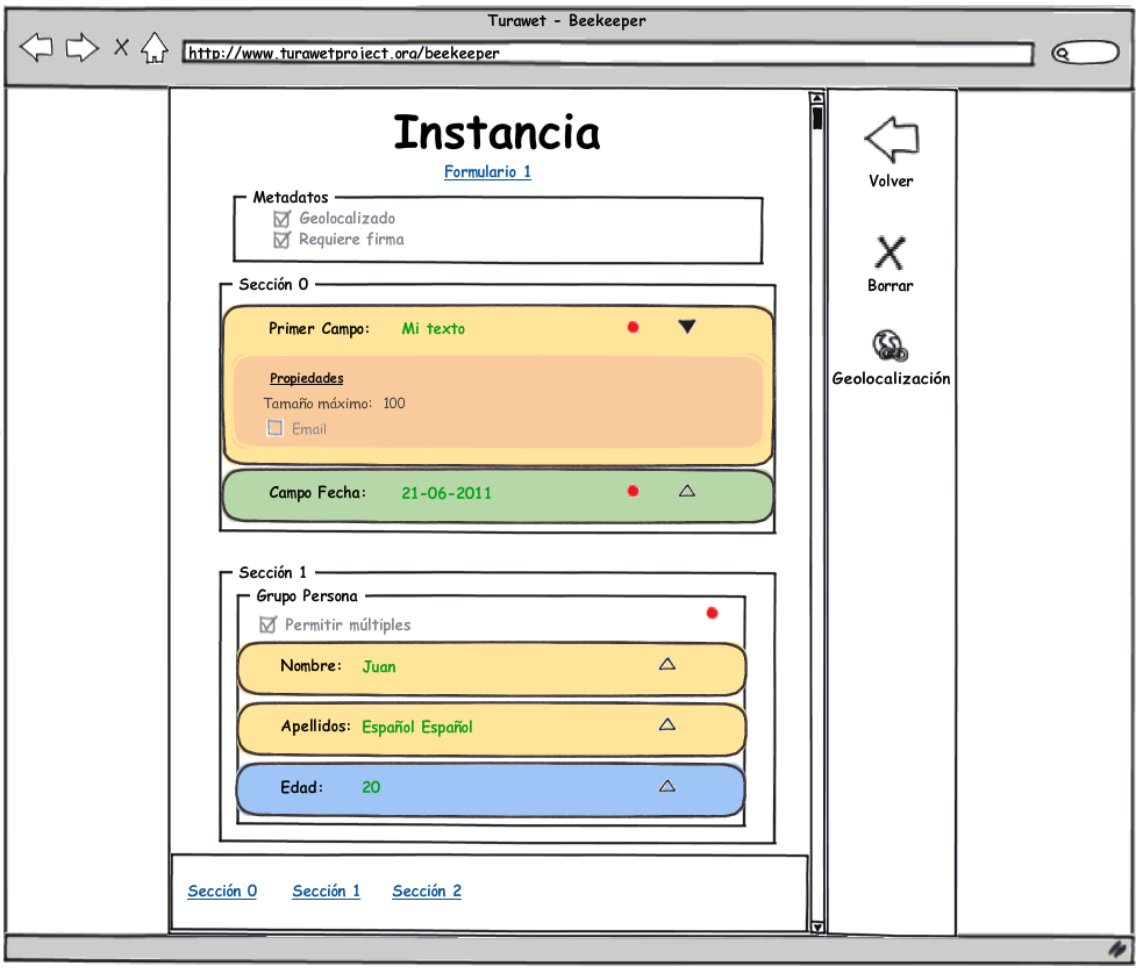


Figura 6-9 Mockup de una instancia en el Administrador.

En el *mockup* anterior se ilustra un ejemplo de visualización de una instancia almacenada en el repositorio. Como podemos observar, la interfaz es muy similar a la de visualizar formularios. La pantalla se halla dividida en 3 secciones principales: la central (donde se muestra la instancia con sus campos rellenos), la barra inferior (donde se muestran enlaces a las diferentes secciones) y por último la barra de opciones, que nos muestra las opciones disponibles en la pantalla actual.

Geolocalización de una instancia concreta

Figura 6-10 Mockup de geolocalización de una instancia concreta.

Este *mockup* ilustra la geolocalización de una instancia concreta. Al igual que en la geolocalización de todas las instancias de un formulario, en este caso, al presionar sobre el icono que representa a la instancia obtendremos información de la misma. Ésta información será, como indicamos anteriormente, un enlace a la instancia (donde podamos ver los valores rellenos de cada uno de los campos del formulario) y una imagen (en caso de que la instancia tenga una). La imagen tendrá utilidad en algunas de nuestras implementaciones de ejemplo, como la aplicación para la notificación de desperfectos o en la de fichas de plantas amenazadas.

6.4 Etapa de implementación

En este apartado se describe la implementación realizada del módulo Administrador. Se ha intentado seguir, en todo momento, el diseño elaborado previamente, aunque, como es lógico, se produjeron ciertos cambios durante esta etapa de implementación que modificaron el diseño propuesto inicialmente, enriqueciéndolo y solventando algunos errores.

6.4.1 Tecnologías utilizadas

A continuación se describen, someramente, las tecnologías utilizadas para este módulo del proyecto:

- Como base fundamental del administrador del repositorio nos encontramos con el *framework* de desarrollo para Python **Django**, que ya ha sido comentado con anterioridad y que nos simplifica en gran medida el desarrollo de aplicaciones web siguiendo el paradigma modelo-vista-controlador.
- Como sistema gestor de base de datos, en principio decidimos utilizar **SQLite** durante las fases iniciales del desarrollo del primer prototipo. Con posterioridad hicimos algunas pruebas sobre **Oracle XE**[73]. Éste sistema gestor lo instalamos en el servidor que tuvimos en el centro de cálculo de la ETSII en etapas centrales del desarrollo del primer prototipo. Finalmente, y motivados por la migración al servidor de la FEULL volvimos a SQLite, sobre todo para evitar perder tiempo instalando un nuevo sistema gestor en el nuevo servidor, invirtiendo el tiempo ahorrado en mejorar el desarrollo de la aplicación.
- Para publicar los servicios web desde nuestro servidor/repositorio, decidimos utilizar el módulo **Soaplib** de **Python**.
- Para desarrollar los analizadores sintácticos de XML del administrador del repositorio (los encargados de realizar las inserciones en base de datos de un formulario o instancia a partir de un XML recibido) utilizamos el lenguaje de programación **Python**, con su módulo DOM para XML llamado **ElementTree**.
- Para la aplicación web de administración del repositorio utilizamos múltiples tecnologías, que coincidirán con las utilizadas en el modelador. Cabe destacar las librerías **JavaScript** de **JQuery** y, por supuesto **HTML** y **CSS**.
- Además, merecen especial atención los módulos **Pycha** [74] de Python para la generación de estadísticas, así como las librerías JavaScript para el servicio de mapas de **Google API** [75] para la parte de geolocalización.

6.4.2 Decisiones de implementación

Modelos

Una vez estuvieron bien definidos los mínimos del diagrama ER de la base de datos, así como los de los ficheros XSD de los XML para formularios e instancias, se transcribió esta información a los modelos de Django.

Los modelos de Django, son una herramienta que proporciona este *framework* y que vienen a ser la implementación Django del patrón de implementación *DAO* [76] *(Data access object)*. Estos modelos definen las entidades de nuestra base de datos, con sus atributos correspondientes y sus relaciones. Además nos permiten, a posteriori, realizar consultas de

forma muy sencilla a los datos almacenados, abstrayéndonos del sistema gestor que estemos utilizando.

A continuación se describe un ejemplo de consulta que nos permite obtener el formulario con identificador “1” de nuestra base de datos:

```
form = Form.objects.get(id=1)
```

Para ilustrar gráficamente los modelos de Django que implementamos, hemos generado automáticamente un diagrama de clases UML de dichos modelos mediante el módulo de Python *django command extensions* [77].

Para obtener dicho diagrama UML mediante el módulo citado, nos colocamos en el directorio del subproyecto en cuestión (en nuestro caso en el sub-proyecto “db_models” dentro del proyecto BeeKeeper). A continuación ejecutamos:

```
./manage.py graph_models db_models –o UML.png
```

Y obtenemos, en el fichero “UML.png” el diagrama de clases UML que representa los modelos Django que hemos implementado para construir nuestra base de datos, siguiendo el diagrama ER expuesto en el apartado de diseño de este mismo capítulo.

En la Figura Figura 6-11 se puede observar el diagrama de clases UML de los modelos generado con la herramienta anteriormente citada.

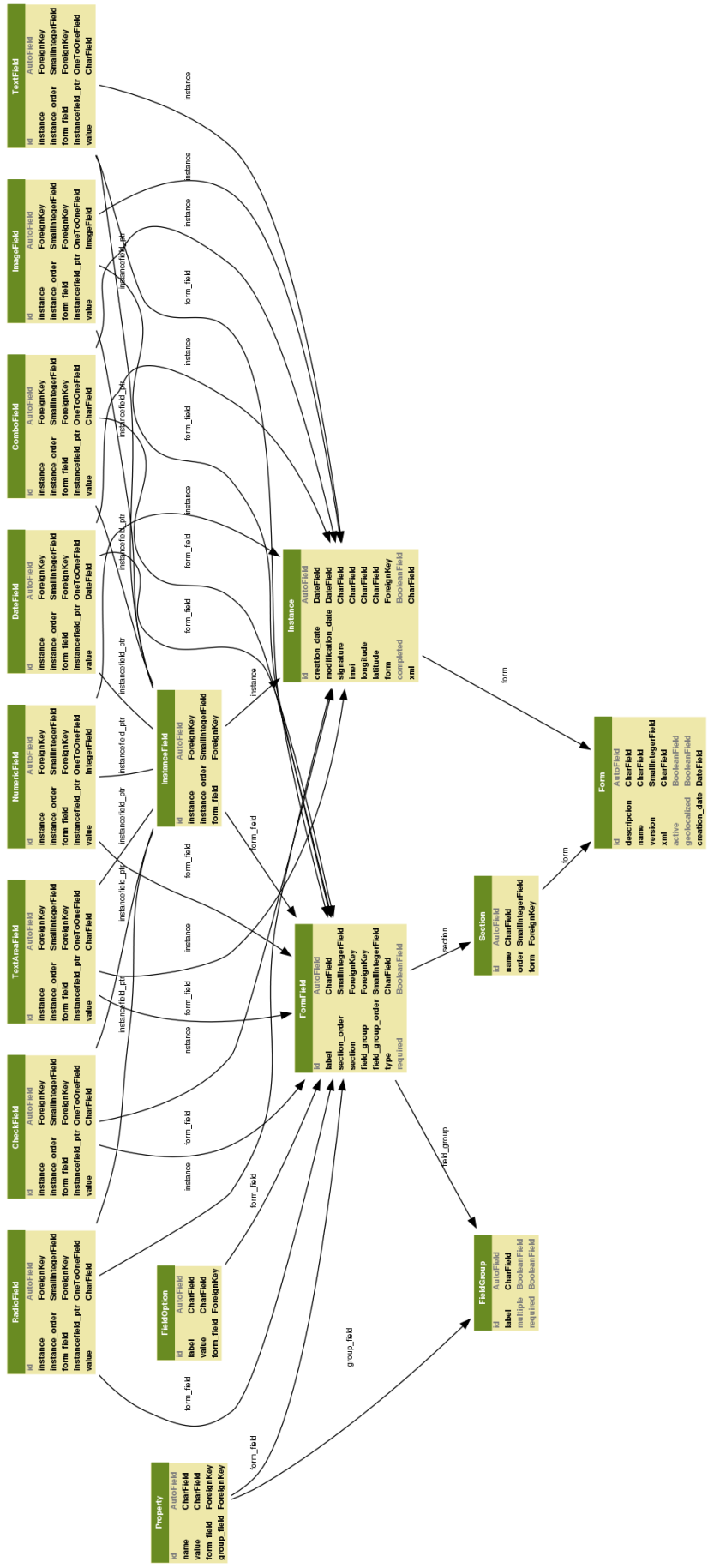


Figura 6-11 Diagrama de clases UML que representa a los modelos de Django para la Base de Datos del repositorio.

Describiendo el fichero de modelos, tenemos que en primer lugar se encuentra la clase formulario, cuyos atributos no son otros que los planteados en el diagrama ER.

Como dato de interés se puede observar la abstracción que nos brinda Django de los tipos de datos del sistema gestor. Cada uno de los campos de nuestros modelos está definido con tipos de datos de Django (`models.CharField`, `models.BooleanField`, etc...). Asimismo, también es preciso destacar que, por defecto, los atributos de Django no permiten valores nulos y que, en caso de que deseemos que los permitan (como es el caso de la descripción) hemos de indicárselo explícitamente.

Por otra parte, es preciso indicar que cada clase que representa un modelo tendrá a su vez otra clase opcional internamente, que será la clase `Meta`, donde podemos definir claves alternativas formadas por la concatenación de varios atributos, definiendo la *tupla* `unique_together` con los atributos que forman la clave. En el caso de un formulario, existirá una clave que será nombre-versión y que se ilustra en el Código 6-1 que aparece a continuación y que ilustra la clase `Form` de los modelos.

```
class Form(models.Model):
    descripcion = models.CharField(max_length=2048, null=True)
    name = models.CharField(max_length=256)
    version = models.SmallIntegerField()
    xml = models.CharField(max_length=16192, unique=True)
    active = models.BooleanField()
    geolocalized = models.BooleanField()
    creation_date = models.DateField()
    class Meta:
        unique_together = ('name','version')
        ordering = ['name']
    def __unicode__(self):
        return unicode(self.name)
```

Código 6-1 Modelo Django para la entidad formulario.

Como hemos comentado anteriormente, la clase `Form` tendrá todos sus atributos con los tipos de datos de Django. Además se puede observar cómo se indican las características especiales de un atributo, como son el permitir valores nulos o ser clave alternativa. Además, como indicamos en los párrafos anteriores, existe una clase `Meta` que nos permite definir claves alternativas complejas, formadas por la concatenación de varios atributos.

También existen otros elementos interesantes que consideramos preciso ilustrar, como son el tratamiento de las claves ajenas (*foreign keys*) o la inclusión de múltiples claves alternativas. Para ello se muestra a continuación un segundo ejemplo de código de los modelos. Esta vez, el ejemplo de la clase Sección.

```
class Section(models.Model):
    name = models.CharField(max_length=128)
    order = models.SmallIntegerField()
    form = models.ForeignKey(Form)
    class Meta:
        unique_together = (('name','form'),('order','form'))
        ordering = ['form']
    def __unicode__(self):
        return unicode(self.name)
```

Código 6-2Modelo Django para la entidad sección.

En este caso podemos observar, en la clase `Meta`, que existen dos claves alternativas. Una vendrá dada por la concatenación del nombre de la sección con el identificador del formulario en el que se halla incluida (recordemos que en el esquema ER la sección tenía una dependencia en identificación de la entidad formulario). La otra clave alternativa será la concatenación del identificador del formulario con el orden de la sección (que es único en el formulario).

Por otra parte, cabe destacar el tratamiento que hace Django de las interrelaciones. En el ejemplo anterior de la sección, vemos que existe una clave ajena `form` que referencia al formulario correspondiente en el que se halla la sección. Esta clave ajena se corresponde con la interrelación “tiene” que unía al formulario con la sección en el esquema ER (era una interrelación de tipo 1:N).

Para interrelaciones de otro tipo (1:1 ó N:M) existe un tratamiento diferente. En el caso de las interrelaciones 1:1, Django nos ofrece la función `OneToOneField` que es similar a una clave ajena con el atributo `unique` a verdadero y con la salvedad de que el lado de la relación inverso al del objeto donde se defina, retornará solamente un objeto y no varios como se hace en las claves ajenas.

Por último, para el caso de las interrelaciones N:M, Django nos ofrece otra alternativa de implementación. Esta alternativa se representa a continuación con un ejemplo que escenifica una situación en la que existen usuarios y grupos como entidades y una interrelación “miembro” que relaciona un usuario con los grupos a los que pertenece.

```
class User(models.Model):
    name = models.CharField(max_length=128)
    surname = models.CharField(max_length=128)
    def __unicode__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(User, through='Membership')
    def __unicode__(self):
        return self.name

class Membership(models.Model):
    user = models.ForeignKey(User)
    group = models.ForeignKey(Group)
```

Código 6-3 Ejemplo de relaciones N:M representadas en modelos de Django .

Servicios web

En este punto del proyecto, ya teníamos definida la forma de comunicación entre aplicaciones que desarrollaríamos y las tecnologías que íbamos a utilizar para implementarla. Recordemos que pretendíamos *interoperar* con 3 herramientas que estaban desarrolladas utilizando tecnologías muy dispares. Para ello, apostamos por una solución basada en servicios web SOAP, como ya se ha descrito con anterioridad.

La primera etapa de la implementación, nos exigía contar con una máquina que publicara estos servicios. Además, este servidor debía ser el mismo que se encargase de gestionar el repositorio, donde los formularios y las instancias iban a residir almacenados en una base de datos. El servicio web, al recibir un elemento invocará a una herramienta de análisis sintáctico, que recibirá los XML de formularios e instancias y los almacenará en nuestro almacén de datos. Por tanto, la idea estaba clara, teníamos que implementar servicios web (WS) que fueran accesibles desde el exterior a través del servidor Django del módulo Administrador.

Para la implementación de los servicios web utilizamos el módulo soaplib de Python. Soaplib es una librería *soap* fácilmente extensible y que proporciona varias herramientas útiles para la creación y publicación de servicios web SOAP en Python. Entre otras funcionalidades, este paquete incluye la generación de WSDL bajo demanda para los servicios publicados, así como permite adjuntar datos binarios.

En nuestra implementación, para publicar el servicio web insertamos en el fichero “urls.py” del sub-módulo de servidor de servicios web en el Administrador, una línea donde se indique la acción que se ha de realizar cuando se desee consultar el WSDL.

```
    urlpatterns = patterns('ws_server.views',
           (r'service.wsdl','service'),
           (r'service','service'),
    )
```

Código 6-4 Patrones de URL que enlazan al WSDL.

Cuando se encuentre la cadena `service.wsdl` al final de la URL solicitada, se ejecutará `service`, que no es más que una variable que instancia un objeto de la clase `SoapService`. Esta clase la hemos desarrollado nosotros, extendiendo de la clase `DjangoSoapService`, y es la clase que tiene como métodos cada uno de los servicios web que hemos desarrollado y que comentaremos a continuación.

```
service = csrf_exempt(SoapService())
```

Código 6-5 Instanciación de un objeto de la clase `SoapService`

Finalmente, implementamos cinco servicios web:

1. **Get_all_forms_preview**: El cual devuelve una lista formada por parejas nombre-versión de todos los formularios disponibles.
2. **Get_xmlform_by_name_version(name, versión)**: Éste servicio web ha de ser consultado enviando el nombre y versión del formulario que queremos descargar. La respuesta del WS será el envío del XML del formulario solicitado.
3. **Get_forms_ids(forms_id)**: Este servicio web recibe una lista con identificadores de formularios y devuelve los objetos formulario solicitados.
4. **Upload_new_form(xml)**: Este servicio es consumido por el modelador. El modelador envía un XML a este servicio y se realiza una llamada al analizador sintáctico que se encarga de desplegar el formulario recibido como XML en la base de datos. Se devolverá un booleano que indicará éxito o fracaso en la operación.
5. **Upload_new_instance(xml)**: Este servicio es consumido por el recolector. El recolector envía un XML a este servicio y se realiza una llamada al analizador sintáctico que se encarga de desplegar la instancia recibida como XML en la base de datos. Al igual que en el caso del servicio web anterior, se devolverá un booleano que indicará éxito o fracaso en la operación.

Traductores (*Parsers*)

Hemos implementado dos *parsers* en el administrador del repositorio. El primero `form_xml2db_parser` traduce los formularios XML que se reciben desde el modelador vía el servicio web `upload_new_form` a modelos de Django y los inserta en la base de datos. Por su parte, el segundo analizador sintáctico `instance_xml2db_parser` traduce las instancias en XML recibidas desde el recolector Android vía el *web service* `upload_new_instance` y realiza un proceso similar al anterior. Primero transforma los datos en XML a objetos intermedios de Django para su posterior inserción en la base de datos.

Para la implementación de estos traductores hemos utilizado un módulo DOM de Python llamado ElementTree que nos facilita en gran medida el proceso de traducción.

Traductor de formularios

El traductor de formularios está estructurado y dividido en varias funciones. Existe una función general `generateModels` que se encarga de crear el objeto formulario de los modelos a partir del XML (básicamente insertará al objeto `Form` sus correspondientes metadatos). En primer lugar, se crea el *parser* llamando a la función XML de ElemenTree pasándole como parámetro el fichero XML de definición del formulario, como se ilustra en el siguiente ejemplo:

```
    def generateModels(self, xml):
        if xml isNone:
            return False
        else:
            parser = XML(xml)
           …
           …
        return True
```

Código 6-6Creación del *parser* de ElementTree a partir del XML.

Para obtener el valor de cada etiqueta XML utilizaremos la función del *parser* de ElementTree `fintext` como se ilustra en el siguiente ejemplo, donde se intenta obtener la fecha de creación del formulario y, en caso de no existir, le asigna la fecha actual:

```
      creation_date = parser.findtext('meta/creationdate')
            if creation_date == None:
               creation_date = datetime.date.today()
```

Código 6-7 Inserción de la fecha de creación de un formulario.

Tras obtener los metadatos, obtendremos todas las secciones gracias a la herramienta ElementTree y su función `findall`, que nos permite buscar por etiquetas en todo el fichero XML y nos devuelve una lista con las ocurrencias. Una vez tenernos las secciones del formulario, iteramos sobre ellas y dentro del bucle y sobre una sección concreta, extraemos sus campos, para posteriormente llamar a la función `parse_generic_field` con la lista de campos y el modelo ya construido de la sección. En el siguiente ejemplo se ilustra cómo se extraen las secciones y cómo se recorren posteriormente, extrayendo los campos de cada sección y llamando a la función de *parsing* de campos:

```
sections = parser.findall("sections/section")
for section in sections:
    …
    fields = section.findall("fields/field")
    …
    self.parse_generic_field(fields, section_model, i, j)
    …
```

Código 6-8 Traducción de los formularios en XML a la base de datos. Recorriendo secciones.

La función `parse_generic_field` se encarga de construir los diferentes objetos de campos y enlazarlos a la sección correspondiente. A su vez, comprueba, para cada campo, si posee opciones o propiedades. En tales casos llama a las funciones `parse_field_properties` o `parse_field_options` para crear las instancias de objetos `option` o `property`.

Dentro de cada sección también se itera sobre los grupos. Al igual que se extraen todos los campos de una sección y se llama al método que inserta los campos, también se extraen los grupos que hay en cada sección. Una vez extraídos los grupos, se llama a la función `parse_generic_group` que obtiene la información de los grupos, instancia los objetos intermedios con la información del XML y realiza el proceso de almacenamiento en la base de datos.

En el caso de los grupos, es de interés resaltar que internamente poseen campos, con lo que se llamará a la función encargada del tratamiento de los campos indicándole que los campos provienen de un grupo al que han de referenciar. Por último, nos resta mencionar que los grupos también pueden tener propiedades, con lo que, en caso de hallarlas, se llamará a la función de tratamiento de las propiedades para grupos.

Traductor de instancias

El traductor de instancias está estructurado de forma muy similar al de formularios.

En la función principal se recogen todos los metadatos y se crea la instancia (aunque esta herramienta también está pensada para permitir la actualización de instancias). Al igual que en el otro traductor, se itera sobre las secciones y se llama a las funciones de tratamiento de campos o grupos.

La función de tratamiento de campos difiere bastante de la que teníamos en el traductor de formularios. Esta diferencia viene dada, sobre todo, por la necesidad de instanciar diferentes modelos en función del tipo de campo con que nos encontremos. Para ello definimos un diccionario que asociará una función del tipo de datos, como se ilustra a continuación:

```
actionSwitch ={
    'TEXT': self.parse_text_field,
    'TEXTAREA': self.parse_textarea_field,
    'NUMERIC': self.parse_numeric_field,
    'DATE': self.parse_date_field,
    'RADIO': self.parse_radio_field,
    'COMBO': self.parse_combo_field,
    'CHECKBOX': self.parse_check_field,
    'IMAGE': self.parse_image_field
    }
...
...
for field in instance_fields:
    ...
    ...
    instance_field_model = actionSwitch[field_model.type](field)
    ...
```

Código 6-9 Diccionario con los tipos de datos y las funciones traductoras correspondientes.

Las funciones que trabajan sobre los diferentes tipos de campo son bastante similares. Una peculiaridad de cada una es el tratamiento que hacen de la etiqueta `value` (pues hay algunas etiquetas `value` complejas, que a su vez están compuestas de otras etiquetas, como es el caso de las imágenes). Otra peculiaridad importante es que, como es lógico, cada función instanciará un modelo diferente (pues hay un modelo por cada tipo de datos).

```
def parse_image_field(self, parser):
        filename = parser.findtext('value/filename')
        binary_image = parser.findtext('value/binary')
        file_content = ContentFile(a2b_base64(binary_image))
        instance_field_model = ImageField()

        return(instance_field_model, filename, file_content)
```

Código 6-10 Función encargada de la traducción de los campos tipo imagen.

En el fragmento de código anterior podemos observar cómo para un campo de tipo imagen se ha de obtener el nombre del fichero que se le dará a la imagen al almacenarla en el servidor. También se observa cómo se obtiene la imagen codificada en base 64. Además de instanciar un objeto de tipo imagen y devolver los valores de los campos, debemos decodificar la imagen, para lo que hemos utilizado el módulo de Python *binascii* y su función *a2b_base64*.

El administrador de Django

Django nos proporciona por defecto un módulo sencillo para la administración de la base de datos, que en nuestro caso pretendemos que sea sólo para el equipo de desarrollo de Turawet, no siendo utilizable, en principio, por el usuario final de nuestra herramienta, aunque este sea un gestor, ejecutivo o administrador, debido a que nosotros hemos desarrollado un módulo de administración más acorde a las necesidades de estos usuarios.

Accediendo al módulo administrador de Django con el rol de súper-usuario, que tuvimos que crear durante la instalación de la base de datos, podremos ver las diferentes tablas que hay almacenadas en nuestra base de datos, así como las *T-uplas* insertadas en cada tabla (siempre de manera engorrosa y pensado para personas con conocimientos técnicos). Con este administrador podremos realizar nuevas inserciones, modificaciones o borrados manuales de cada una de las tablas que poseamos y sus elementos.

Como hemos indicado en los párrafos anteriores, en el marco de este proyecto, la utilidad de este administrador queda reducida al equipo de desarrollo durante la implementación o corrección de errores puntuales durante el mantenimiento.

El administrador y cuadro de mandos

Además de toda la lógica de base de datos, incluyendo modelos, herramientas de traducción y servidor de servicios web; hemos desarrollado un administrador del repositorio que

integra un cuadro de mandos y está pensado para el uso de usuarios gestores, administradores o ejecutivos, externos al equipo de desarrollo.

Este módulo ha sido desarrollado en base a las funcionalidades que se describieron en la etapa correspondiente al diseño. Es una aplicación web, que ha sido desarrollada intentado ser lo más similar al modelador posible, a fin de que el usuario reconozca en todo momento que se encuentra en una de las herramientas del proyecto Turawet.

Gestión de usuarios y control de acceso

Para la herramienta web de administrador del repositorio hemos implementado un sistema de autenticación. El sistema de autenticación que hemos implementado para esta primera versión es bastante sencillo, y se ha incluido, solamente, para dotar de un mínimo de seguridad al administrador del repositorio y, por ende, a los datos almacenados.

Hemos empleado una utilidad de Django que permite la creación de lo que denominan `MIDDLEWARE_CLASSES`. Son clases intermedias, como su propio nombre indica, que se ejecutan siempre que se solicita la carga de un controlador.

A continuación se muestra la clase intermedia que hemos utilizado para controlar el acceso de los usuarios.

```
class SiteLogin:
    "This middleware requires a login for every view"
    def process_request(self, request):
        if re.search('beekeeper', request.path)and \
           (not re.search('images', request.path))and \
           (not re.search('css', request.path))and \
           (not re.search('js', request.path))and \
           request.path != '/beekeeper/accounts/login/'and\
           request.user.is_anonymous():
           if request.POST:
               return login(request)
            else:
               return HttpResponseRedirect \
               ('/beekeeper/accounts/login/?next=%s'% request.path)
```

Código 6-11 Clase intermedia para el control de acceso.

El fragmento de código anterior realiza una serie de comprobaciones, algunas de ellas con expresiones regulares. Se comprueba que la URL actual pertenece al módulo `beekeeper` dentro de la aplicación `BeeKeeper`. Éste módulo es el administrador del repositorio. De esta forma, para otros módulos el servicio web no se nos solicita autenticación y pueden utilizarse sin problemas desde aplicaciones externas. También se comprueba que, si solicitamos acceso al CSS, ficheros JavaScript, o a las imágenes, éste acceso se nos permite sin necesidad de autenticarnos, para poder visualizar correctamente la aplicación.

Por último, se verifica que, si además de estar dentro del módulo `beekeeper` y estar solicitando carga de un controlador, no estamos ya en la página de *login* y además somos un usuario anónimo; entonces es cuando se nos reenvía a la pantalla de acceso de usuarios.

Gestión de formularios e instancias: visualización y eliminación

Una vez que un usuario se haya autenticado en el administrador del repositorio, podrá visualizar todos los formularios disponibles para los que tenga permisos. Es necesario mencionar que actualmente, en la primera versión del prototipo, quien acceda a esta herramienta y se autentique, siempre podrá ver todos los formularios disponibles en el repositorio, debido a que no se ha implementado la gestión de usuarios y grupos propuesta en la etapa de diseño, sino se han definido algunos usuarios con todos los permisos.

Para cada uno de los formularios disponibles se nos permite ver la descripción del mismo, con una interfaz similar a la que se presenta cuando diseñamos un nuevo formulario desde el modelador (pero no siendo editable). También se nos permite, para cada formulario, ver las instancias rellenas que hay de cada uno de ellos. Para estas interfaces que estamos describiendo, y que son similares a la del modelador, se continúan utilizando diferentes funciones de JQuery, como la que nos permite desplegar las propiedades u opciones de un campo, o la funcionalidad de *scrolling* de la barra inferior de secciones.

Además, entre las funcionalidades de gestión del administrador del repositorio, también se nos permite eliminar tanto instancias como formularios, teniendo en cuenta que la eliminación de un formulario significará la eliminación de todas las instancias pertenecientes a él.

Ahondando en la eliminación de formularios e instancias, se puede observar que en la visualización de cada formulario o instancia, así como al lado del nombre de cada uno en las respectivas listas, existe un aspa roja. Al pulsar sobre ella, hemos decidido que se muestre un mensaje de confirmación para las eliminaciones (como es habitual en todo tipo de aplicaciones). La secuencia de eventos que tiene lugar al pulsar el aspa se describe a continuación. En primer lugar, el aspa es el *submit* de un formulario cuya acción es volver a la misma URL en la que se encontraba (lo que significará volver al controlador de la página correspondiente). Al haber enviado el formulario, también se ha enviado un campo oculto `elementToDelete` cuyo valor será el ID del formulario o instancia a eliminar.

```
def showFormList (request):
    ...
    ...
    if request.method == 'POST':
        ...
        context["formToDelete"] = postform.cleaned_data['elementToDelete']
        context["deleting"] = True
    else:
        context["deleting"] = False

    return render_to_response('formularios.html', context);
```

Código 6-12 Controlador para la lista de formularios

En el controlador (*view* en Django) se distinguen dos casos. El caso en el que se haya enviado un formulario y el caso en el que no, como se ilustra en el fragmento Código 6-12 incluido en este documento. Si se detecta que se ha enviado un formulario por el método POST, se envía al *template* una variable con el ID del formulario/instancia que se borrará si se confirma el borrado y, además, se cambia el valor de la variable booleana `deleting` a verdadero.

Tras la recarga de la página, se mostrará el mensaje de confirmación, para el que se utiliza una función JavaScript que hace que la aparición y desaparición del mensaje de confirmación sea paulatina.

```
<scriptlanguage="JavaScript">
            $("#resultado").toggle('slow');
</script>
```

Código 6-13 Efecto de aparición/desaparición en JavaScript

Cuadro de mandos

En los párrafos anteriores se describe la parte de gestión de formularios e instancias del administrador del repositorio. Sin embargo, la parte que mayor interés puede tener para un usuario gestor o ejecutivo es el cuadro de mandos. Esta herramienta integra un módulo de estadísticas y otro de geolocalización. El primero nos muestra estadísticas de un formulario en función de todas las instancias rellenas del mismo y el segundo nos muestra un mapa con las instancias geolocalizadas de un formulario. Estos módulos se detallarán exhaustivamente en los siguientes apartados.

Implementación del sistema de estadísticas de formularios

Para desarrollar la herramienta que nos permita consultar estadísticas sobre un formulario hemos utilizado un módulo de Python, llamado Pycha. Este módulo desarrollado por el español Lorenzo Gil Sánchez y nos ofrece una serie de funcionalidades basadas en Cairo [78] para elaborar diagramas de barras o circulares sobre los datos que deseemos.

En primer lugar debemos extraer de nuestra base de datos, los datos que deseamos representar en un diagrama estadístico. Para la extracción utilizaremos las funciones de consulta de los modelos de Django y almacenaremos los elementos en una lista temporal. Las funciones de Pycha nos exigen que les pasemos los datos con una determinada estructura, por lo que, como segundo paso debemos formar la estructura necesaria con nuestros datos. A continuación se muestra un ejemplo de la estructura de datos que debemos pasar como parámetro a la función de Pycha.

```
dataSet =(
    ('Coca Cola', ((0, 10), )),
    ('Fonteide', ((0, 15), )),
    ('Seven Up', ((0, 20), )),
    ('Dorada', ((0, 25), )),
    )
```

Código 6-14 Ejemplo de estructura de datos con valores para representar un diagrama circular en Pycha.

En el fragmento de código anterior, se representa en azul la etiqueta del dato correspondiente y el segundo número en verde representa el valor asociado a la etiqueta para un diagrama circular (nótese que para los diagramas circulares, el primer elemento en verde de cada *t-upla* siempre ha de ser 0). En el ejemplo del Código 6-14 podría considerarse que son el número de unidades de cada una de las bebidas.

Este módulo de estadísticas, además, nos permite incluir opciones como puede ser, por ejemplo, el estilo del fondo. A continuación se muestran las opciones utilizadas en nuestro caso, donde se esconden (ocultan) el fondo y la leyenda.

```
options ={
    'legend':{
        'hide':True,
    },
    'background':{
        'hide':True,
    },
}
```

Código 6-15 Ejemplo de opciones que se pueden indicar para elaborar el diagrama.

Finalmente, tras asociar el `dataset` a un objeto de tipo `PieChart` de Pycha, llamar a su método de *rendering* e instanciar otro objeto de tipo *byte stream*; llamaremos a otra función del módulo Pycha de nombre `write_to_png` pasándole el *stream* anterior, en el que nos será devuelta la imagen PNG.

En nuestro caso hemos decidido utilizar ***streams* en lugar de ficheros**, como parece lógico debido a la breve validez temporal que tendrán los diagramas estadísticos en nuestro sistema, pues debemos tener en cuenta que cada vez que una nueva instancia se rellena, los diagramas estadísticos elaborados del formulario al que pertenece la nueva instancia son susceptibles de haber cambiado. Comprendemos que la utilidad del almacenamiento de los gráficos en ficheros es para aplicaciones estáticas en las que los datos, o no cambian, o cambian muy raras veces.

Una vez que hemos obtenido la imagen generada por Pycha, la devolveremos como resultado de nuestra función en la vista (como un objeto MIME de tipo `image/png`) al *template* que nos la solicitara. Posteriormente será el navegador, encargado de *renderizar* la imagen, el que se encontrará como un objeto MIME en el HTML en lugar de un fichero. Se ha de tener en

cuenta que el HTML habrá sido generado a través de los *templates* que hemos definido en nuestra aplicación basada en Django. En el *template* de estadísticas es donde se realizan las llamadas al controlador encargado de devolver el diagrama estadístico, cuya respuesta se ilustra en el fragmento de código siguiente.

```
output = io.BytesIO()
surface.write_to_png(output)

returnHttpResponse(output.getvalue(), mimetype="image/png")
```

Código 6-16 Se devuelve el objeto MIME de la imagen con estadísticas desde el controlador.

Implementación de la utilidad de geolocalización de instancias

Como se ha comentado ya en varias ocasiones, la utilidad de la geolocalización en la recogida de datos es indiscutible. También es algo generalmente aceptado, que, a día de hoy, cuando se habla de representar datos geolocalizados, las APIs de Google, y en concreto las de GoogleMaps, son el referente por antonomasia.

Cuando un usuario solicita a la aplicación que geolocalice una instancia o todas las instancias de un formulario, esto se traduce en una llamada al controlador correspondiente. El controlador que se encarga de la geolocalización de todas las instancias de un formulario, lo que hace es, en primer lugar obtener todas las instancias de la base de datos que correspondan al formulario en cuestión. Una vez que se tienen todas las instancias de un formulario, se recorren una a una y se insertan en una lista, junto con la primera imagen que tenga dicha instancia, si es que la tiene. Dicha imagen será la que se muestre al pulsar sobre el icono de la instancia sobre el mapa.

Detallando el *template* que hemos desarrollado para esta funcionalidad, y que es el llamado por la vista comentada anteriormente, cabe destacar que las funciones JavaScript de GoogleMaps introducen toda la información en un `DIV` cuyo identificador sea `map`. Con esto podemos decir que, referido a HTML sólo tenemos este elemento en el *template*, el cual aparecerá en la parte central de la aplicación web de administración.

Por otro lado, en el bloque preparado para añadir JavaScript se generará, de manera dinámica un vector como el que se muestra en el siguiente fragmento de código:

```
varmarkers = [
  {
    'name': '1',
    'location': [28.353685, -16.371717],
    'message': '<p class="message" align="center">
                  <a href="/beekeeper/show_instance/1">
                     Ver intancia nº 1 completa
                  </a><br/><br/>
                  <img src="/beekeeper/media/images/mifotografia_6.png"/>
                </p>'
  },
```

```
];
```

Código 6-17 Ejemplo de marcador para GoogleMaps.

Para elaborar dicho vector se hace uso de las etiquetas propias del lenguaje de plantillas. Se utilizan para recorrer las instancias y obtener los valores de las coordenadas así como la imagen.

```
{% for instance, image in instances %}
    {% if instance.latitude %}
        {
            'name':'{{ instance.id }}',
            'location':[{{ instance.latitude }},{{ instance.longitude }}],
            {% if image.value %}
                'message':'...'
            {% else %}
                'message':'...'
            {% endif %}
        },
    {% endif %}
{% endfor %}
```

Código 6-18 Generación dinámica de marcadores.

En el fichero “gmaps_functions.js” tenemos la función `initialize` la cual crea el mapa en el elemento `map` (que es el `DIV` comentado anteriormente). Además, en el caso de que no existan marcadores, centra la imagen en un punto por defecto, que será la ETSII de la ULL en nuestro caso. También en esta función se establece el tipo de mapa por defecto (en nuestro caso `SATELITE`) así como el icono a mostrar en cada punto (en nuestro caso el logo del proyecto).

Es necesario indicar que la llamada a la función `initialize` se llevará a cabo en el evento `load` de la ventana, con el fin de cargar el mapa tras haber cargado la ventana.

Tras todas las acciones llevadas a cabo en la etapa de inicialización, para cada uno de los marcadores intermedios utilizados por nosotros, se instancian nuevos objetos `google.maps.Marker`, a los cuales se les incluirá la información correspondiente (nombre, imagen y coordenadas). Finalmente les añadiremos el evento `click` para que al pulsar sobre ellos se genere un bocadillo con la información de la instancia.

Por último se centrará el mapa en función de las instancias representadas.

6.4.3 Prototipo

En esta captura se observa la primera pantalla que nos aparece, una vez hecho el *login*, en la aplicación de administración (BeeKeeper) que se ha descrito en este capítulo. En la imagen podemos observar cómo aparecen todos los formularios disponibles y cómo, de cada uno de ellos, podemos ver su descripción (pulsando sobre su nombre), o bien elegir cualquiera de las acciones situadas a la derecha (desde ver la lista de sus instancias hasta la geolocalización de las mimas, también pudiendo elegir ver las estadísticas del formulario).

En el Capítulo 7 se describirá el prototipo en general, incluyendo al Administrador. Se ilustrará el sistema con un mayor número de capturas de pantalla y una explicación más detallada de cada una de ellas.

Parte III. Interoperabilidad, prototipo y casos prácticos

Capítulo 7. Interoperabilidad y prototipo

Resumen:

- En este capítulo se pretende describir cómo se comunican los tres módulos entre sí para poder actuar como una solución integral para la recolección de datos.
- Se describe la arquitectura general del sistema a través del Diagrama de despliegue y el de componentes.
- Finalmente se ilustra el funcionamiento del sistema con capturas de pantalla explicadas de un flujo completo de ejemplo (desde el modelado a la administración, pasando por la recolección de datos).

7.1 Introducción

En este apartado se detalla cómo se ha conseguido la interoperabilidad de los diferentes subsistemas de nuestro proyecto. Asimismo, se pretende dar una visión general del sistema, para lo que se incluye un Diagrama de Despliegue y un Diagrama de Componentes. Finalmente se describirá todo el proceso de uso del sistema, desde que se modela un formulario hasta que se analizan los resultados de todas las instancias enviadas, detallando el prototipo final del proyecto.

7.2 Diagrama de despliegue

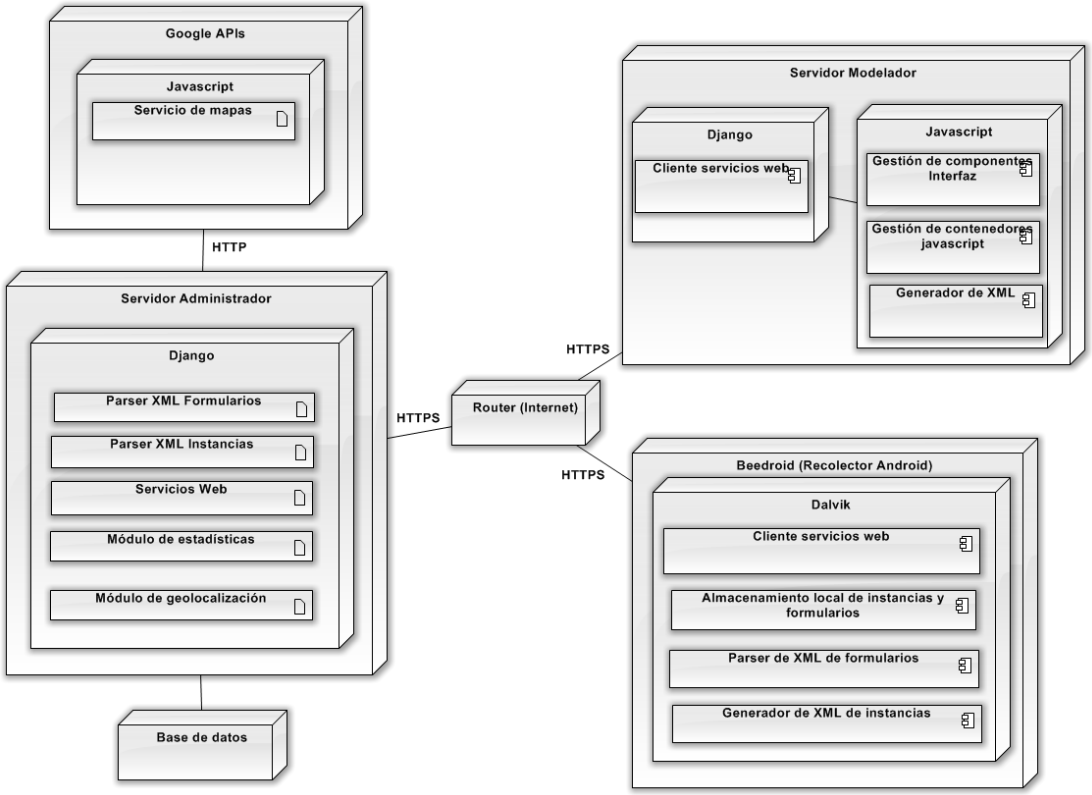


Figura 7-1 Diagrama de despliegue del proyecto.

En el diagrama de despliegue de la Figura 7-1, se muestra la arquitectura del sistema que conforma el proyecto Turawet. En él aparece el despliegue físico de nuestros artefactos o componentes *software* principales, que corren sobre componentes *hardware* tan dispares como un dispositivo móvil con Android o dos servidores web con Django.

En primer lugar, el servidor modelador dividirá su entorno de ejecución en dos partes, un entorno de servidor Python-Django, sobre el que se ejecutarán los componentes clientes para el consumo de los servicios web del módulo administrador, y un segundo entorno JavaScript que se ejecutará en el navegador cliente y sobre el que correrá el módulo que permiten al usuario modelar el formulario de forma gráfica a la vez que otro componente se encarga de representar esa información en una serie de objetos para, posteriormente, traducirlo al XML correspondiente.

En segundo lugar, el recolector móvil será un dispositivo Android ejecutando sus componentes sobre la máquina virtual de Java Dalvik [79]. Dichos componentes facilitarán la traducción del XML del formulario a la representación gráfica en el dispositivo de forma que éste sea rellenable, permitirán el consumo de los servicios web del administrador, almacenarán de forma local en el dispositivo formularios e instancias y generarán el XML correspondiente a las instancias que se rellenen para su envío al módulo administrador.

Por otro lado, el servidor administrador dispondrá de un entorno Python-Django sobre el que se ejecutarán los componentes dedicados a realizar el *parsing* de los XML de los formularios y las instancias, la generación de las estadísticas así como la localización de las instancias en un mapa, disponiendo además de un módulo especial encargado de ofrecer una serie de servicios web al resto de nodos. El nodo administrador, dispondrá de acceso a la base de datos de forma que pueda persistir en ella los datos de formularios e instancias que le llegan. Para completar el funcionamiento del componente encargado de la geolocalización, este nodo requerirá de conexión *http* con un nodo externo al proyecto donde se localizan las APIs de Google Maps para JavaScript.

Cabe destacar la unión de los distintos sistemas interoperables mediante Internet y conexión *HTTP Secure*, que permitirá llevar a cabo la comunicación entre los sistemas presentados de forma segura.

7.3 Diagrama de componentes

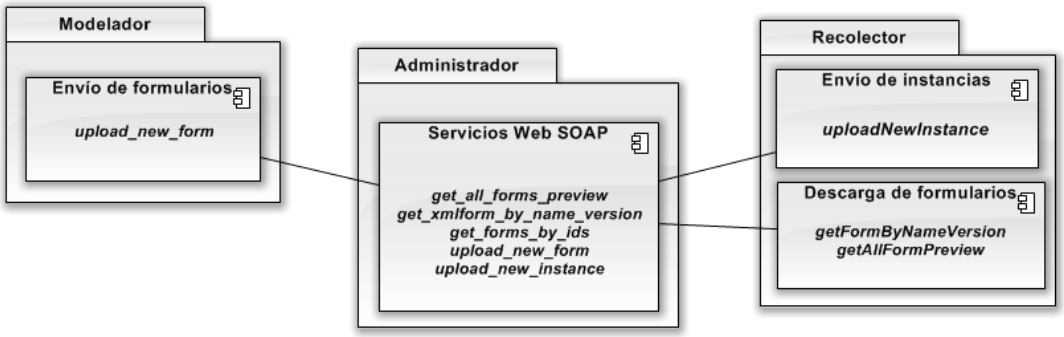


Figura 7-2 Diagrama de componentes del proyecto.

En este sencillo diagrama de componentes se pretende dar una visión muy básica de la forma en la que interactúan los diferentes módulos del proyecto a través de servicios web SOAP, como hemos explicado con anterioridad en el Capítulo 1.

Desde el módulo administrador se ofertarán una serie de servicios que consumirán los módulos modelador (para el envío de formularios) y recolector (para el envío de instancias rellenas así como para la descarga de formularios), como se puede observar en la figura anterior.

7.4 Prototipo final

A continuación se muestra un flujo de ejecución de todo el proyecto, ilustrado con diferentes capturas de cada una de las pantallas de los módulos desarrollados. Las imágenes son auto-explicativas y, además, poseen una breve reseña inferior que las describe, con lo que solamente incluiremos las capturas de pantalla con sus reseñas correspondientes, sin entrar en

descripciones más exhaustivas, que consideramos innecesarias y engorrosas en este punto, pues cada módulo ya ha sido descrito profusamente en capítulos anteriores.

Modelador

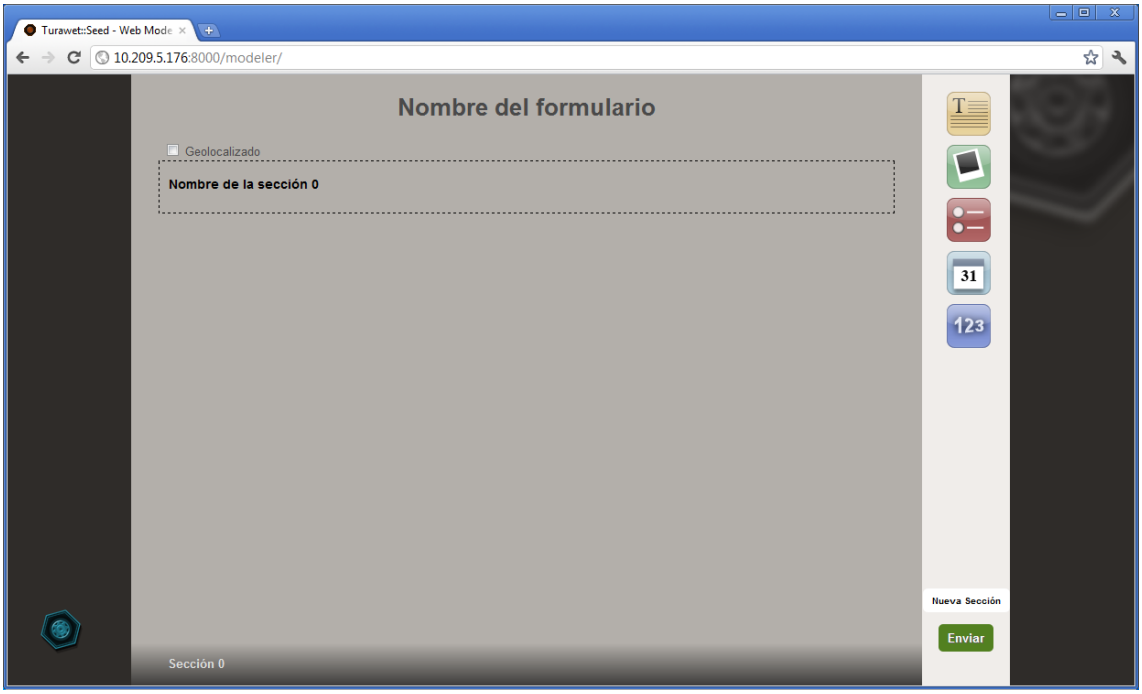


Figura 7-3 Pantalla inicial del modelador.

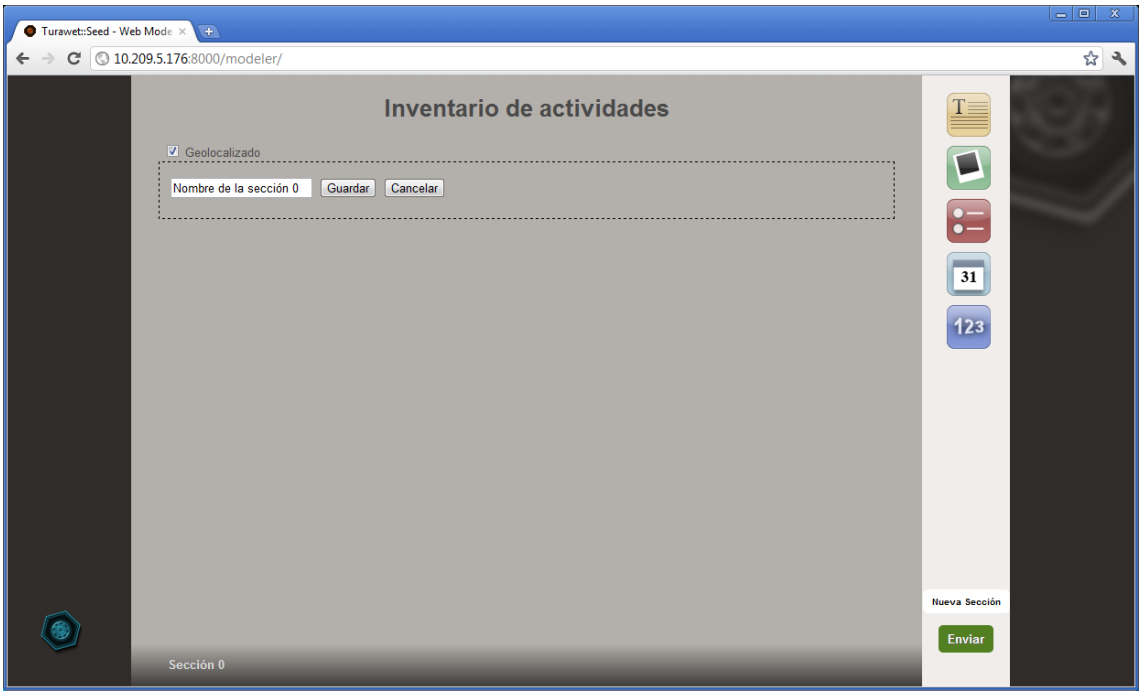


Figura 7-4 Se cambia el nombre del formulario y de la sección y se marca como geolocalizado.

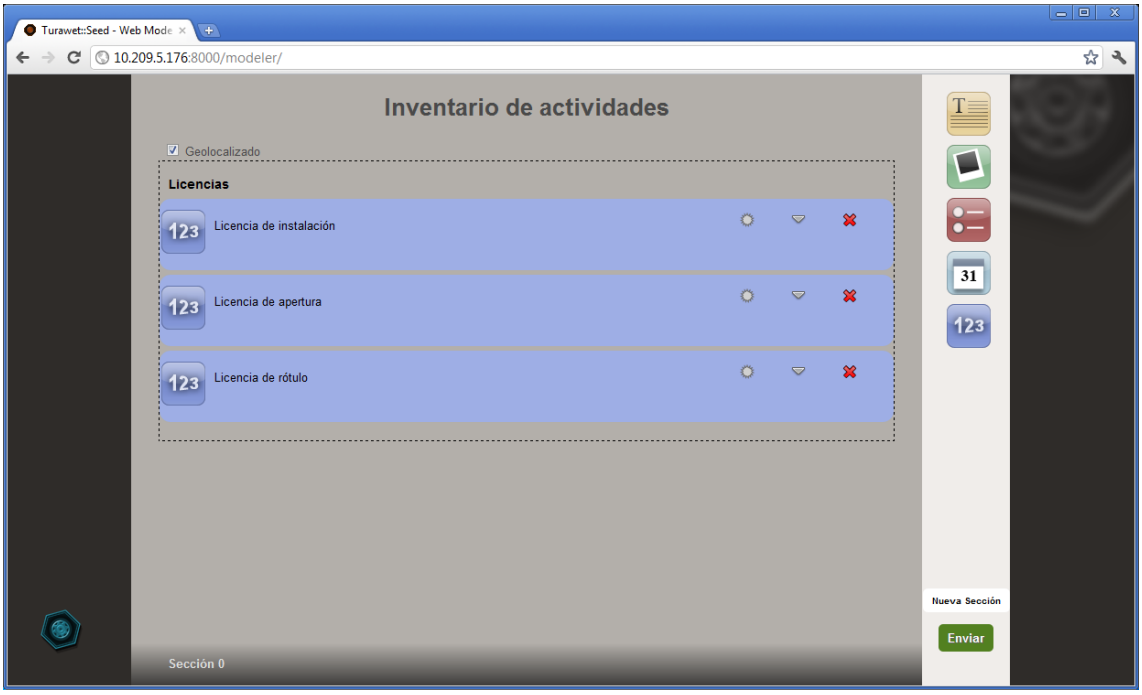


Figura 7-5 Añadimos los tres campos numéricos de licencias.

Figura 7-6 Añadimos un campo tipo *Radio* y editamos su nombre.

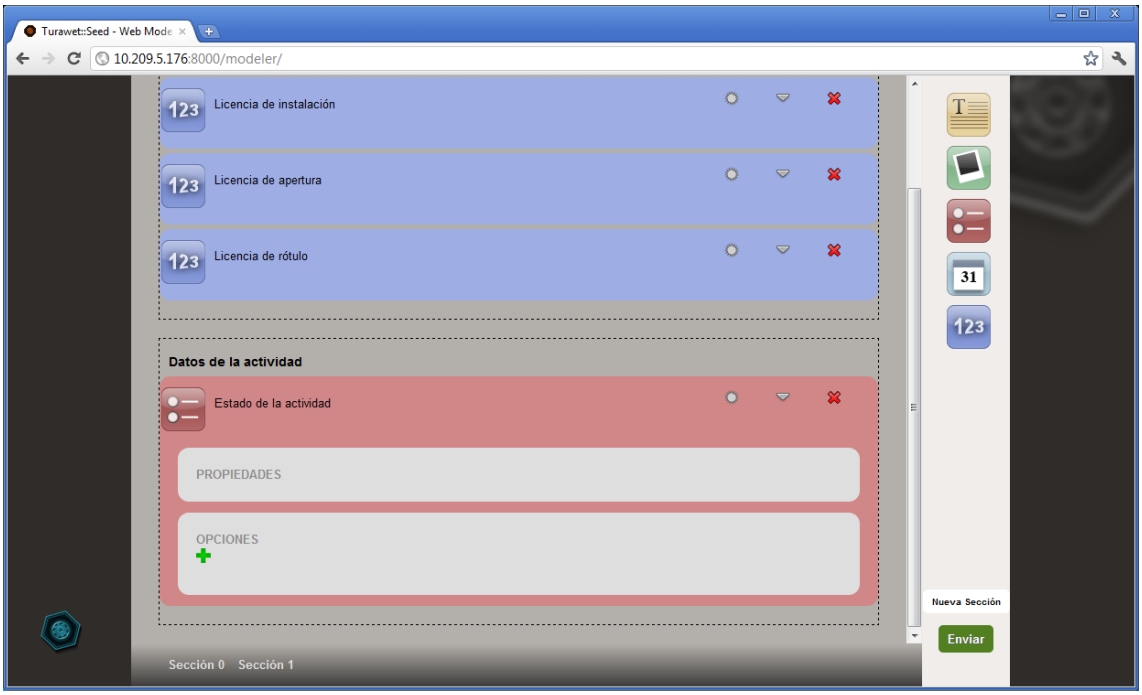


Figura 7-7 Desplegamos las propiedades y opciones del campo.

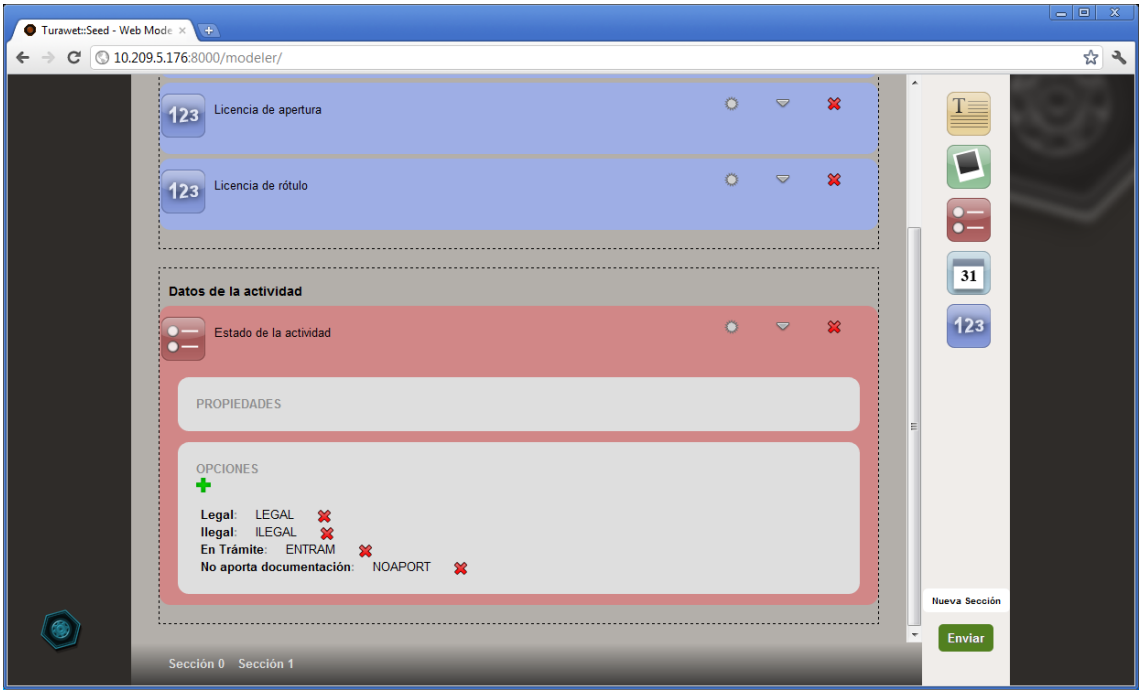


Figura 7-8 Se han añadido las opciones relativas al campo “estado de la solicitud”.

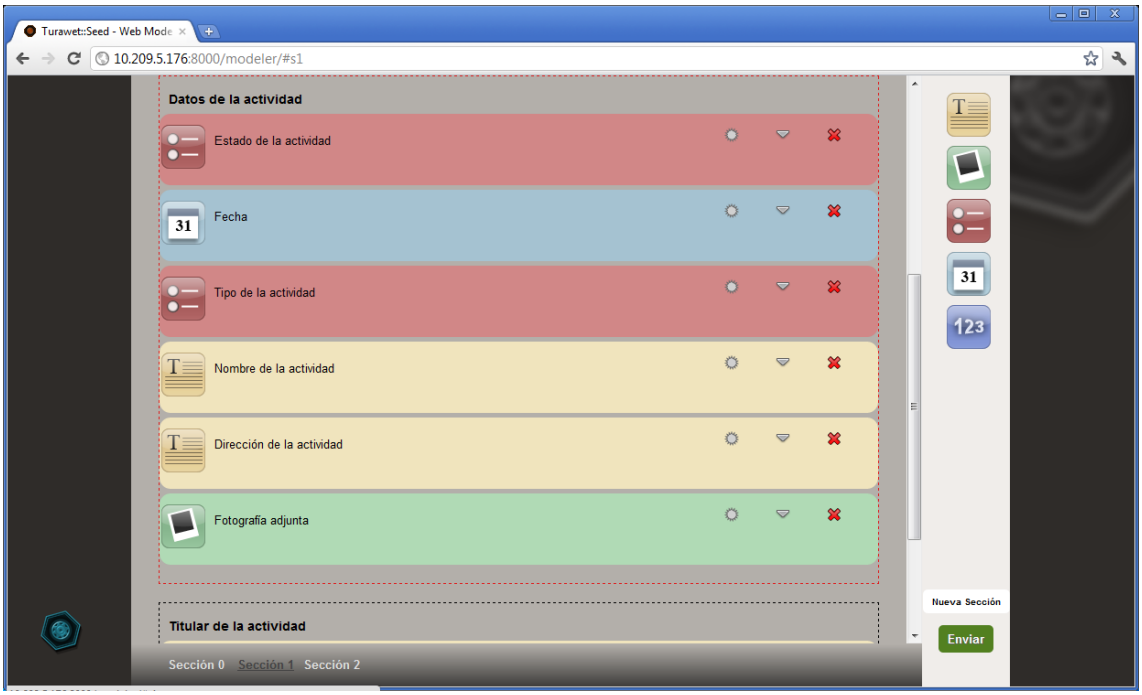


Figura 7-9 Utilizamos la barra de secciones movernos a la sección 1.

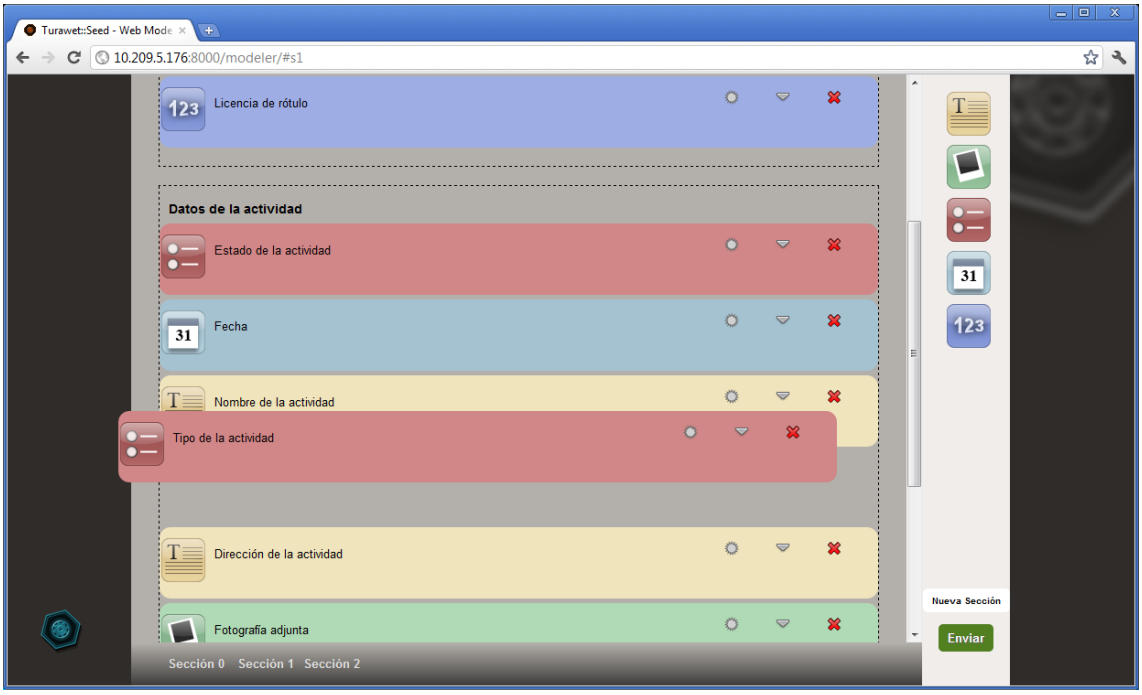


Figura 7-10 Se observa la facilidad para reordenar campos.

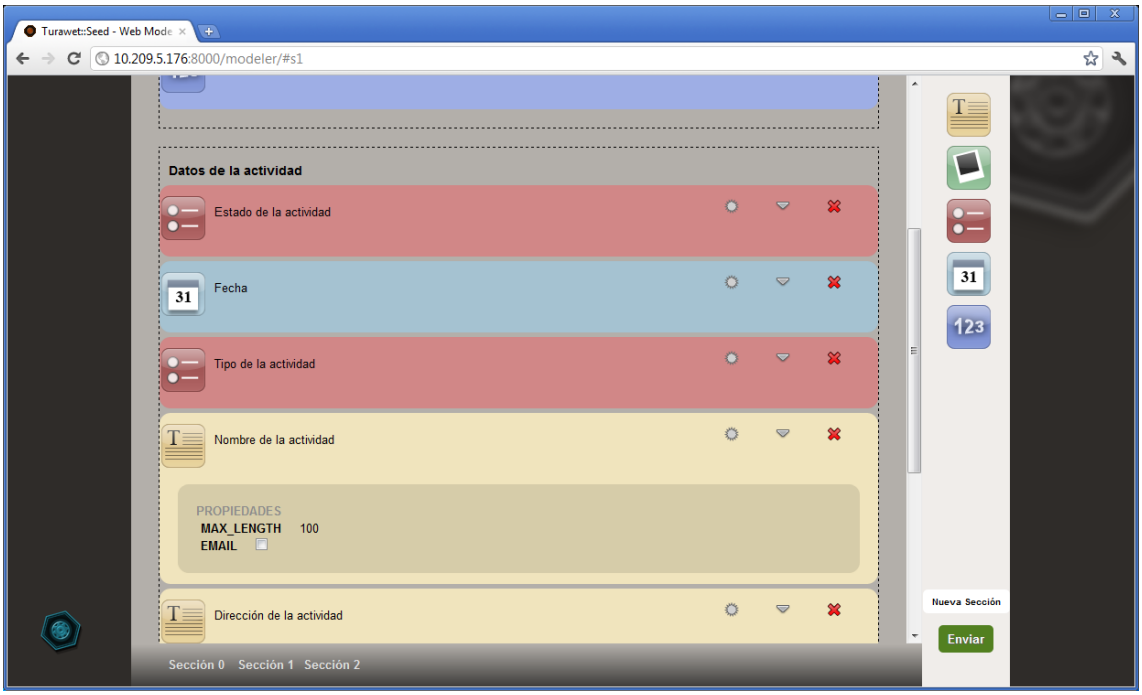


Figura 7-11 Se observan las propiedades disponibles para un campo tipo texto.

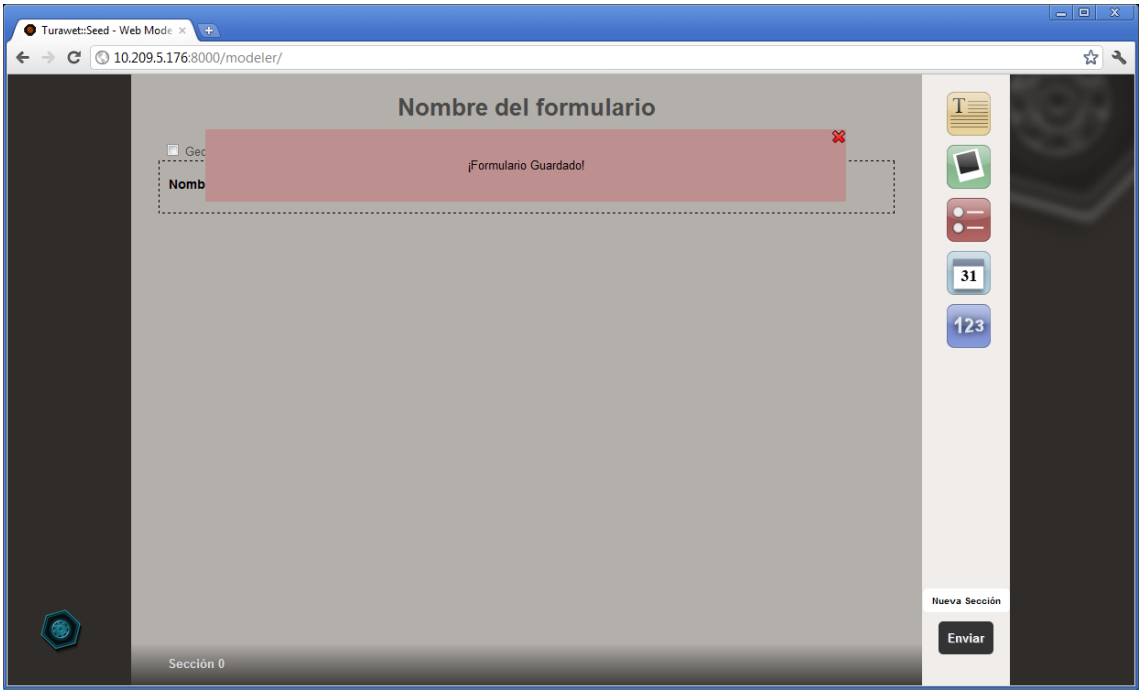


Figura 7-12 Se envía el formulario y se obtiene un mensaje de confirmación.

Recolector

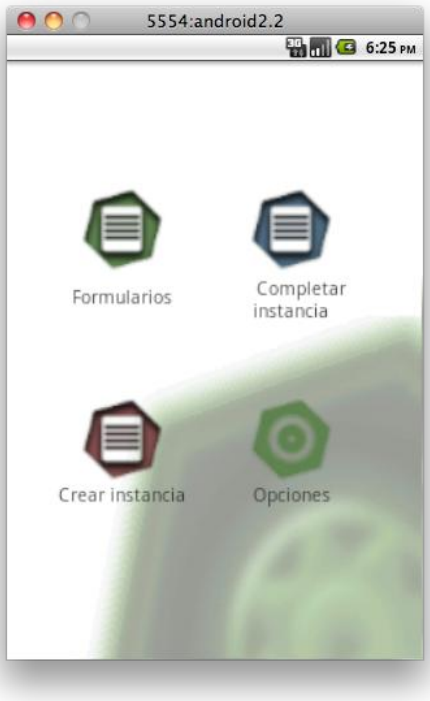


Figura 7-13 Pantalla de inicio del recolector.

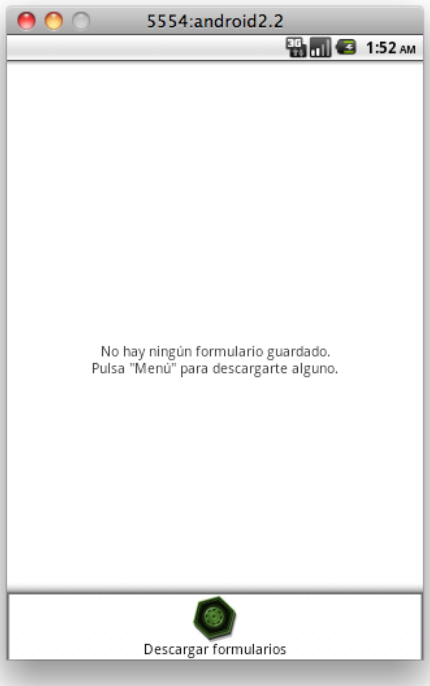


Figura 7-14 Accedemos a la sección “formularios” y visualizamos su menú.

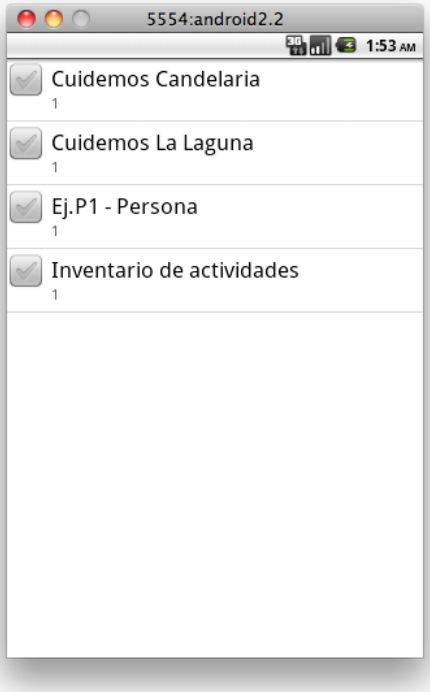


Figura 7-15 Obtenemos los formularios disponibles en el servidor.

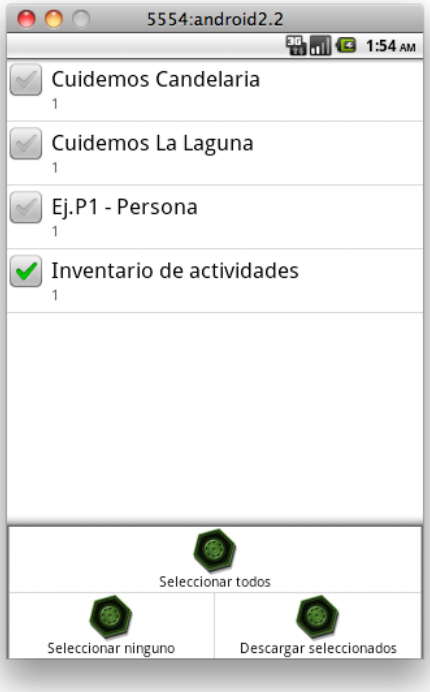


Figura 7-16 Seleccionamos un formulario para descargar (también se observa el menú).

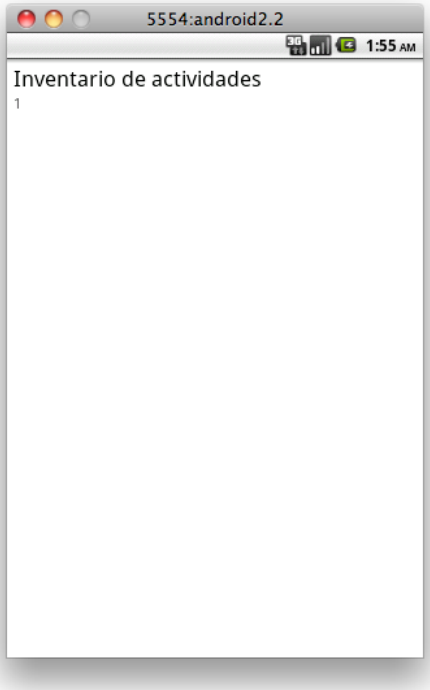


Figura 7-17 Accedemos a "crear instancia", mostrándosenos los formularios disponibles.

Figura 7-18 Comenzamos a rellenar la instancia, siendo el primer campo de tipo numérico.

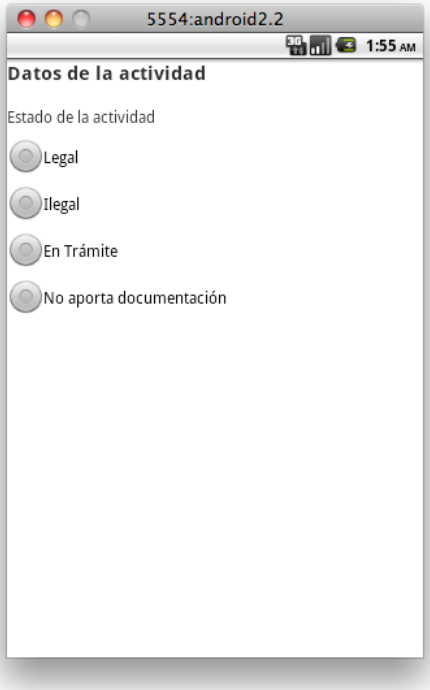


Figura 7-19 Ejemplo de campo con múltiples opciones.

Figura 7-20 Ejemplo de campo tipo fecha.

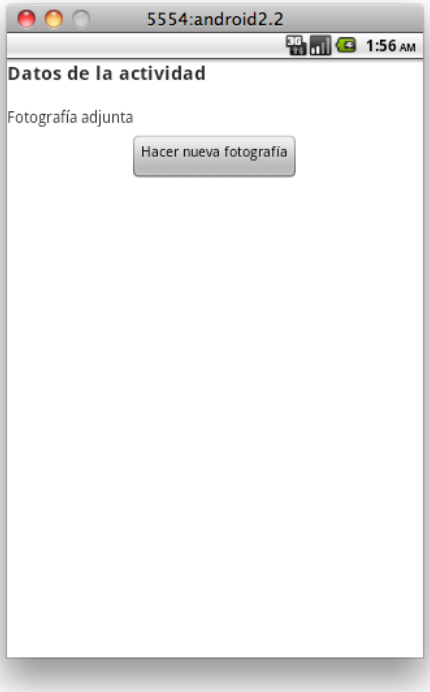


Figura 7-21 Ejemplo de campo imagen con un botón para lanzar la herramienta de fotografía.

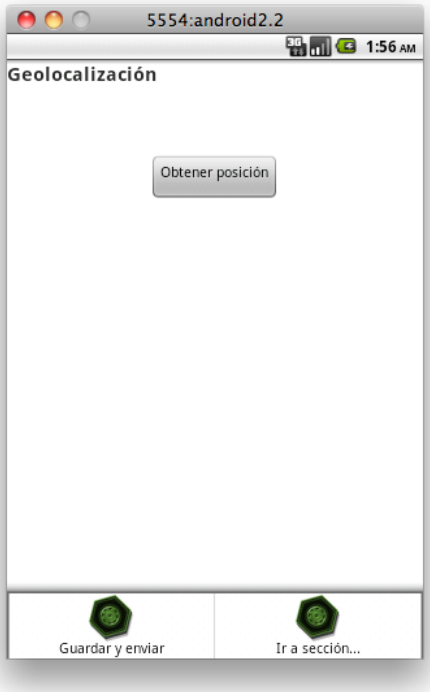


Figura 7-22 Ejemplo de sección de localización y menú (permite enviar o movernos entre secciones).

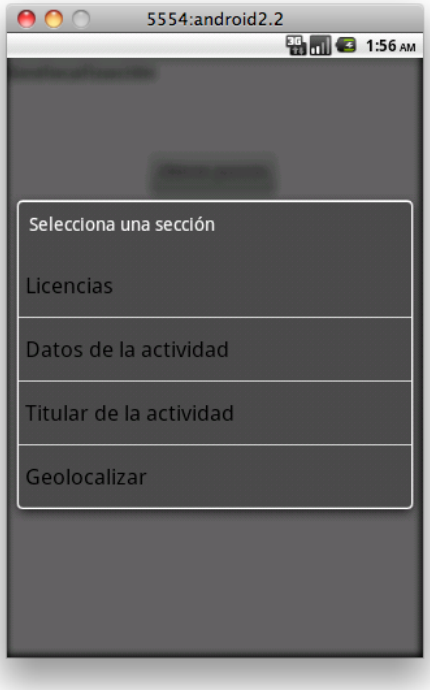


Figura 7-23 Menú de movimiento entre secciones (nótese que geolocalización se incluye como sección).

Administrador

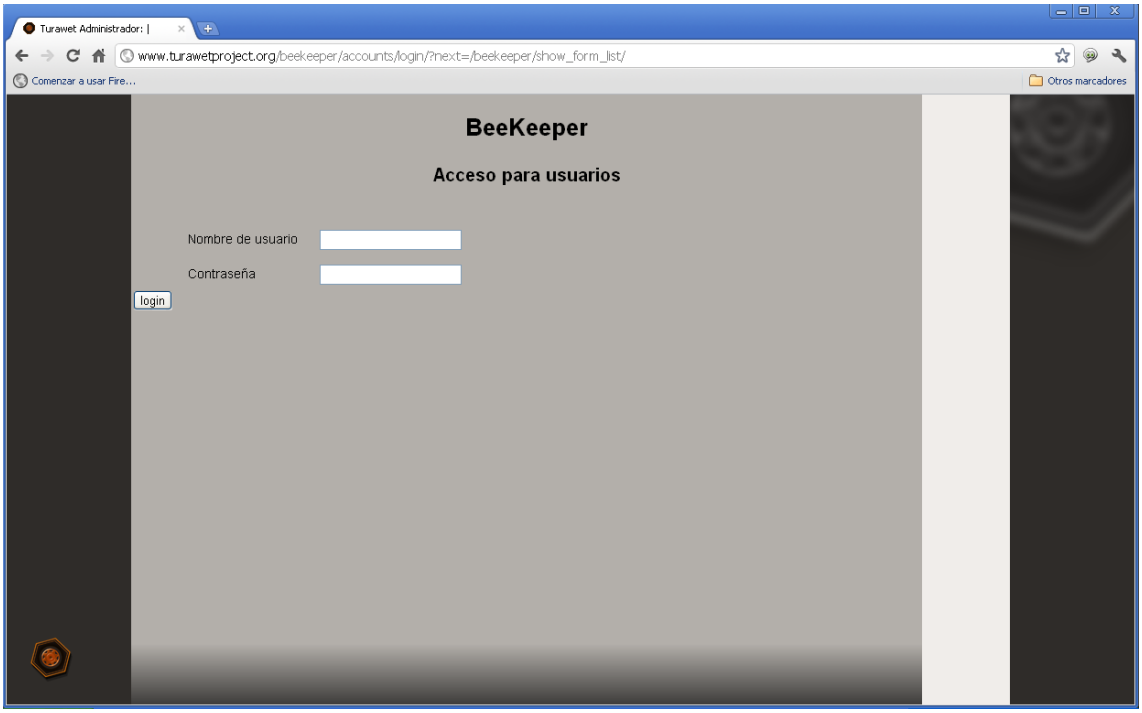


Figura 7-24 Pantalla de acceso al administrador.

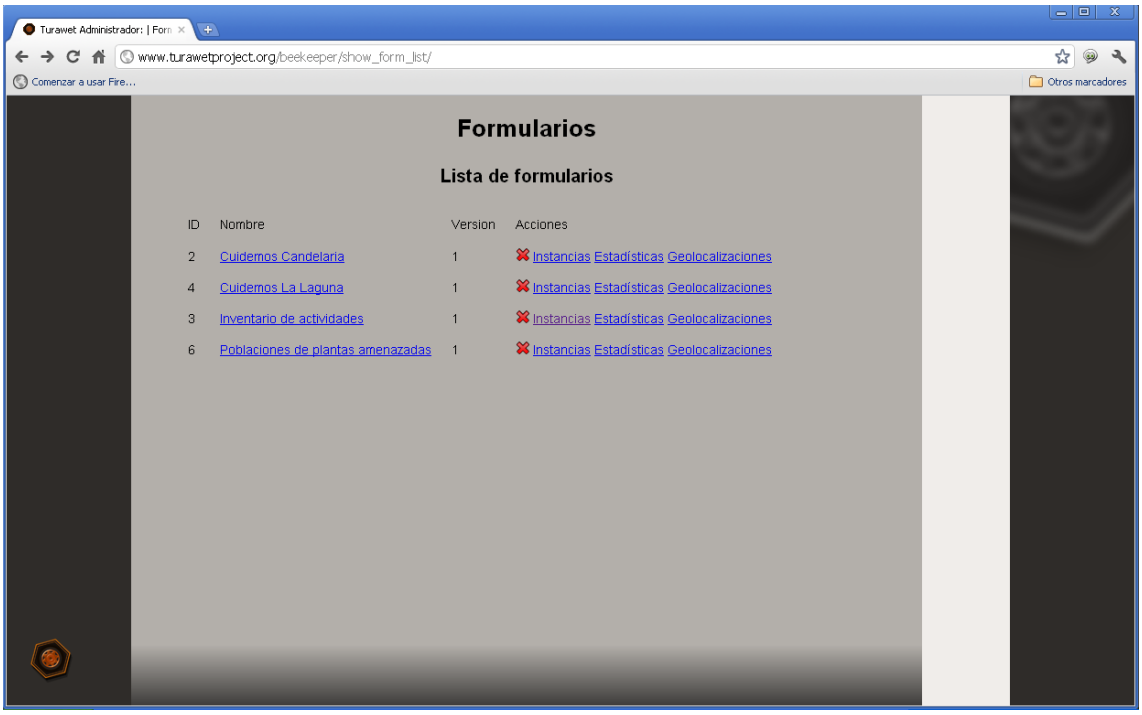


Figura 7-25 Lista de formularios disponibles.

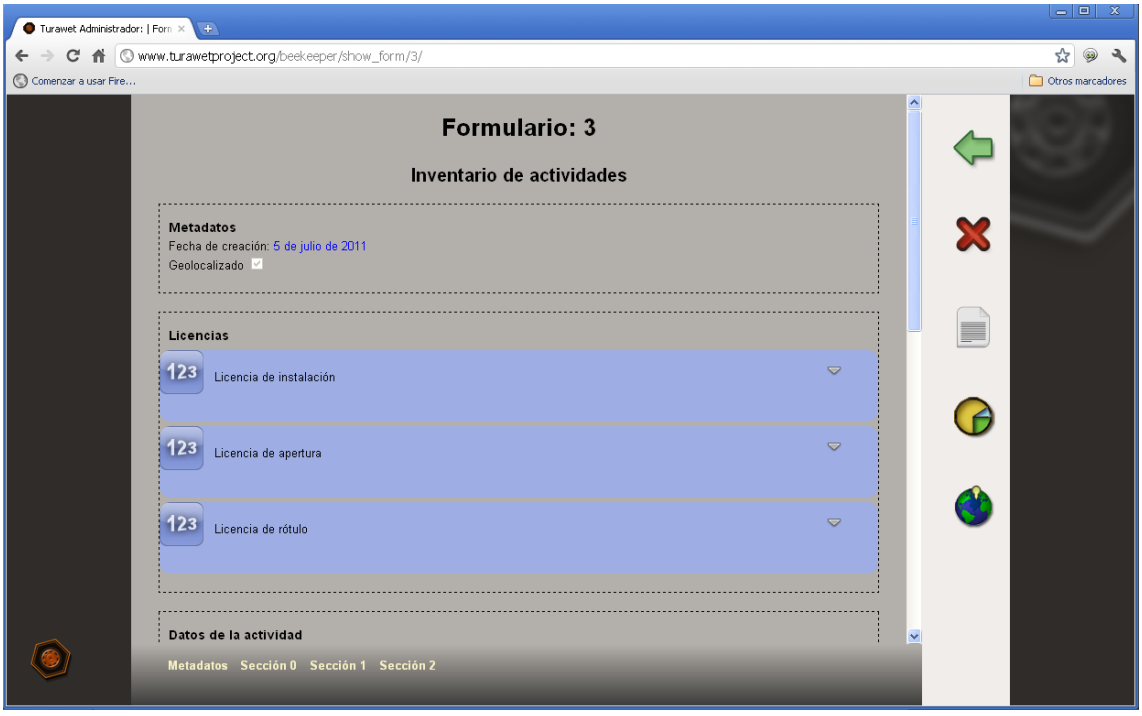


Figura 7-26 Visualización del formulario de inventario de actividades desde el administrador.

Figura 7-27 Listado de instancias disponibles del inventario de actividades.

Figura 7-28 Ejemplo de visualización de una instancia del inventario de actividades.

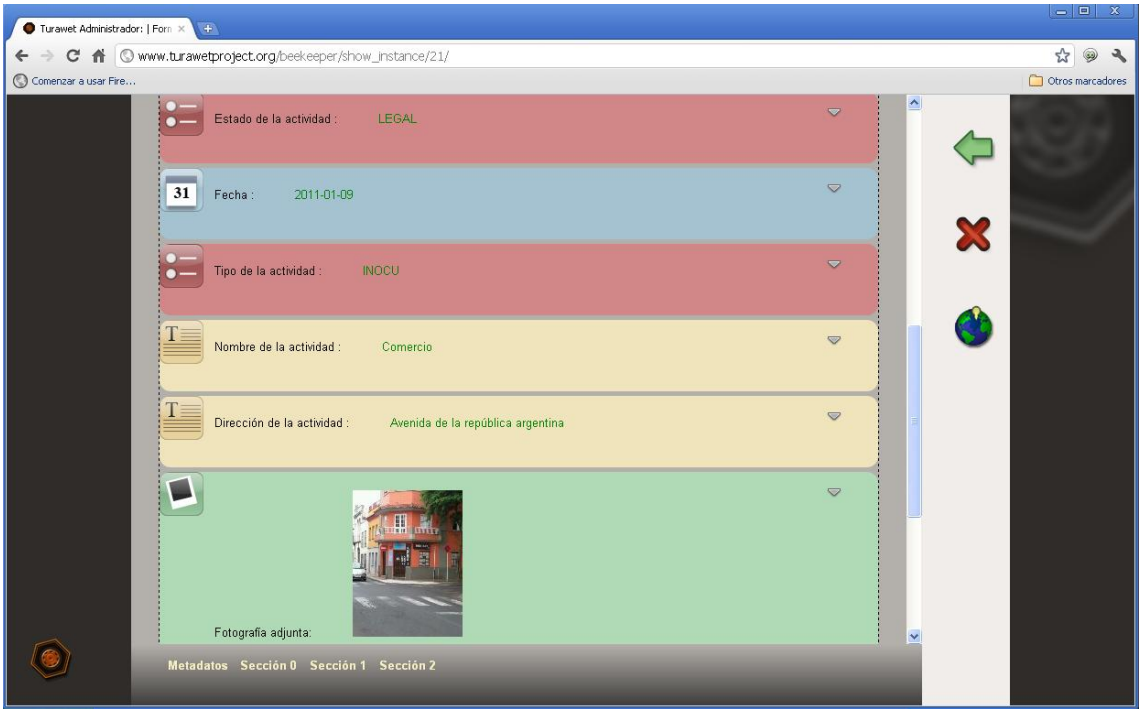


Figura 7-29 Parte inferior del ejemplo de instancia del inventario de actividades.

Figura 7-30 Geolocalización de la instancia anterior del inventario de actividades.

Figura 7-31 Geocalizando todas las instancias del inventario de actividades.

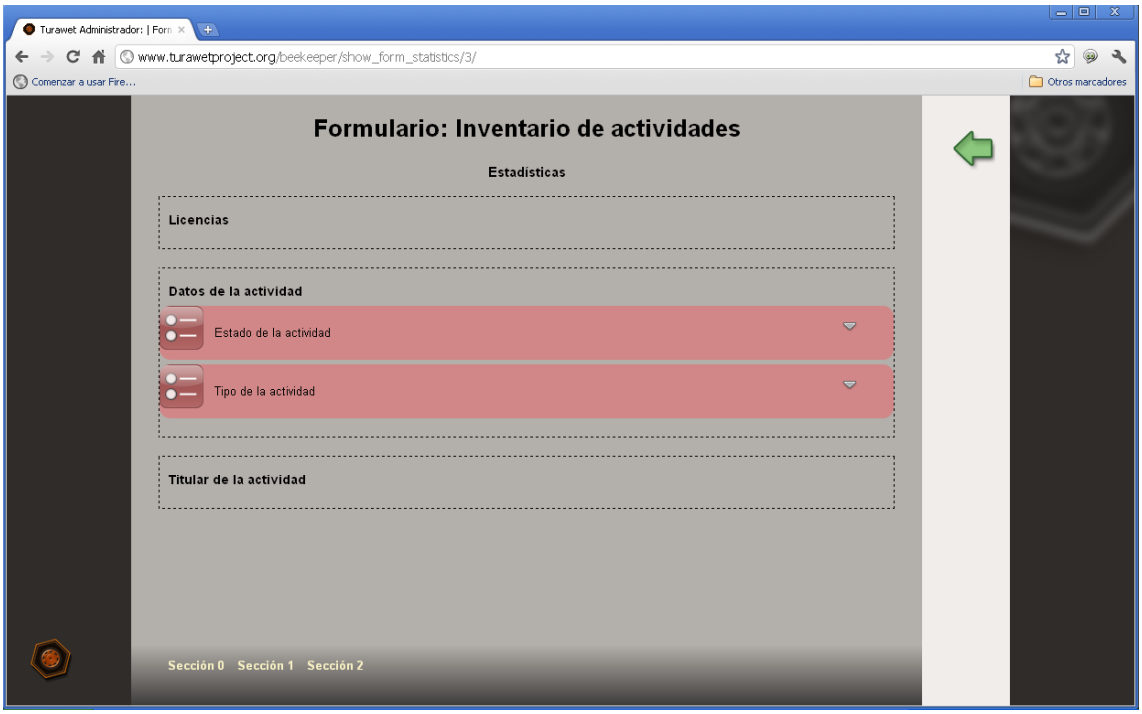


Figura 7-32 Campos disponibles para la generación de estadísticas. Inventario de actividades.

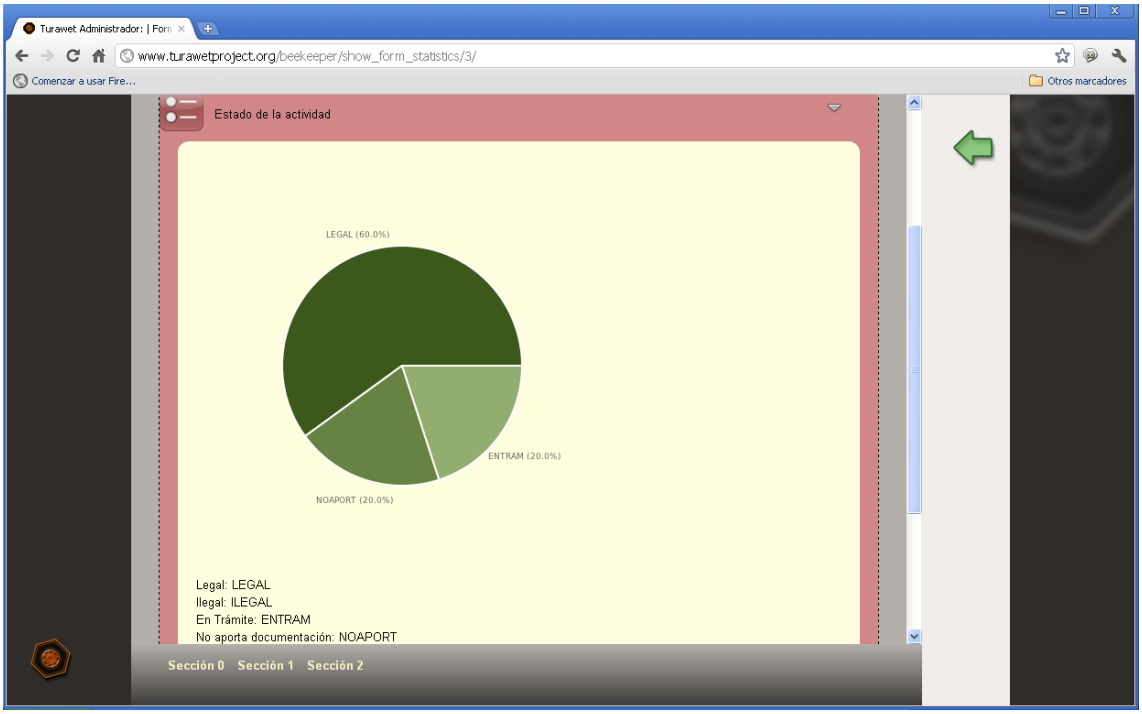


Figura 7-33 Estadísticas de los estados de las actividades.

Figura 7-34 Mensaje de confirmación para la eliminación de un formulario.

Capítulo 8. Casos prácticos

Resumen:

- Se pretende describir e ilustrar los tres casos prácticos que hemos desarrollado.
- De cada caso práctico se ha incluido un diagrama de casos de uso que lo esquematiza.
- Para cada prototipo, se muestran capturas de pantalla explicativas donde se observa cómo se ha aplicado el Proyecto Turawet a estos casos concretos.

8.1 Introducción

Desde el primer momento se pensó en Turawet como una herramienta aplicable a gran multitud de situaciones diferentes que requirieran la recolección de datos. Por ello, una vez finalizado el prototipo general del sistema, se contactó con la Gerencia de Urbanismo de La Laguna y el Departamento de Biología Vegetal de la Universidad de La Laguna para estudiar la implantación de nuestro proyecto en alguna de sus actividades de recolección de datos.

En este capítulo se describe cómo aplicamos el proyecto a tres casos prácticos reales, describiendo en primer lugar cada uno de los casos, y posteriormente, mostrando el resultado final de cada uno de los proyectos concretos.

8.2 Gerencia de Urbanismo de La Laguna

Introducción

Como ya se comentó en capítulos previos, hemos elegido a la Gerencia de Urbanismo de La Laguna, para mostrar un caso de uso real de Turawet. En la siguiente figura vemos una representación de un escenario, donde aparecen todos los partícipes del proceso de recolección y explotación de datos. Para la realización del diagrama, nos basamos en un formulario que modelamos junto a los compañeros de la Gerencia, que será utilizado para realizar un **inventario de las actividades en materia urbanística** que se llevan a cabo en la ciudad.

Diagrama de casos de uso

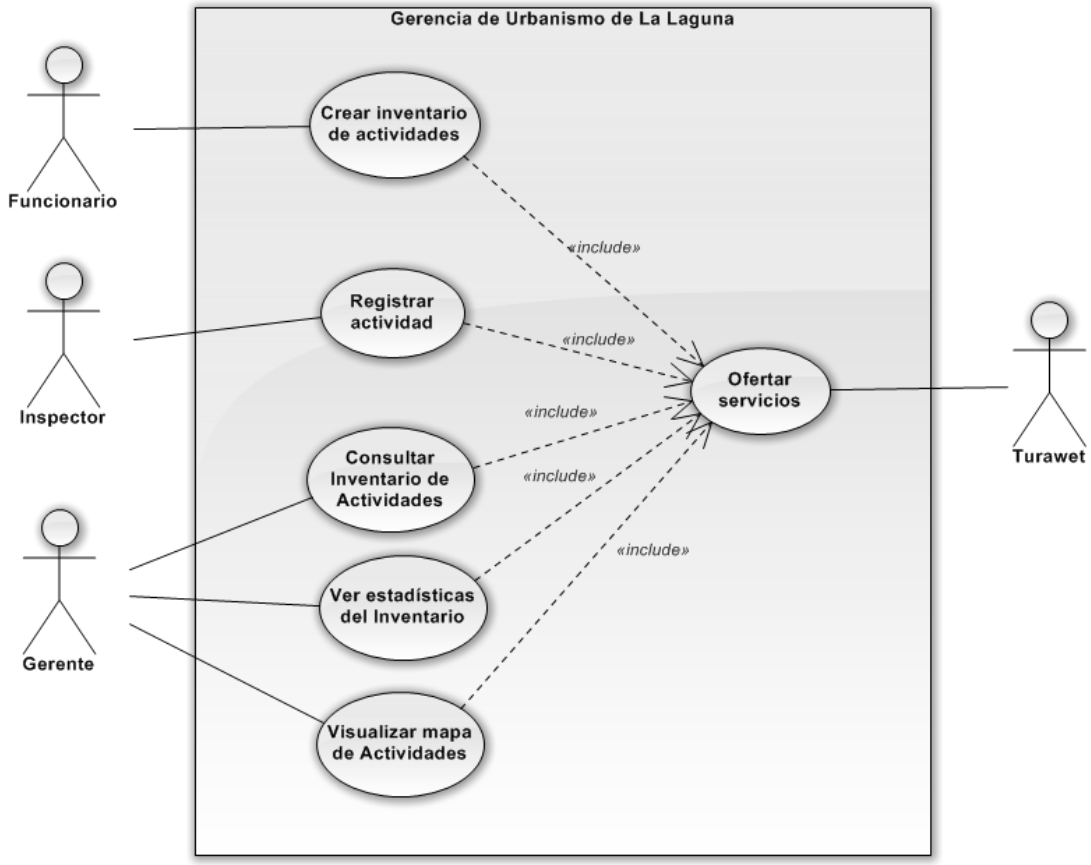


Figura 8-1 Diagrama de casos de uso en una posible implantación de Turawet en la Gerencia de Urbanismo de La Laguna.

Comenzando con la descripción del diagrama, podemos ver que por parte de la Gerencia tenemos tres actores. Cada uno de ellos cumplirá un rol diferente y hará uso de una aplicación concreta de la plataforma. Siguiendo el ciclo de vida de un formulario, primero aparece el actor que lo modela. Este será un funcionario administrativo del departamento, que se encargará de pasar a electrónico el modelo de formulario que ya se utiliza en papel, o bien diseñará uno nuevo, directamente en nuestra aplicación. Sea como fuere el caso, el nuevo formulario será enviado al repositorio para que sea utilizado por nuestro segundo actor, el inspector de urbanismo. Este cumplirá la función de usuario recolector. Dará de alta las nuevas actividades, rellenando instancias del formulario ya generado. Cuando la ronda de recolección acabe y todas las instancias estén almacenadas en el repositorio, el gerente, último actor en escena, se encargará de darle sentido a los datos. Hemos detectado tres casos de uso claros que este podría llevar a cabo:

1. **Consulta de instancias**: en este caso, de las diferentes actividades urbanísticas que se lleven a cabo.
2. **Ver estadísticas**: será interesante, para los gerentes, ver de forma gráfica por ejemplo, el estado de las actividades o los porcentajes de cada tipo de actividad.

3. **Visualizar mapa**: el gerente podría tener una visualización muy rápida de donde se ha hecho la recolección de datos, así como conocer las zonas donde hay más carga de actividades.

Prototipo

Figura 8-2 Modelado del formulario de Inventario de actividades

Figura 8-3 Recolección del Inventario de actividades.

Figura 8-4 Geolocalización de todas las actividades del inventario.

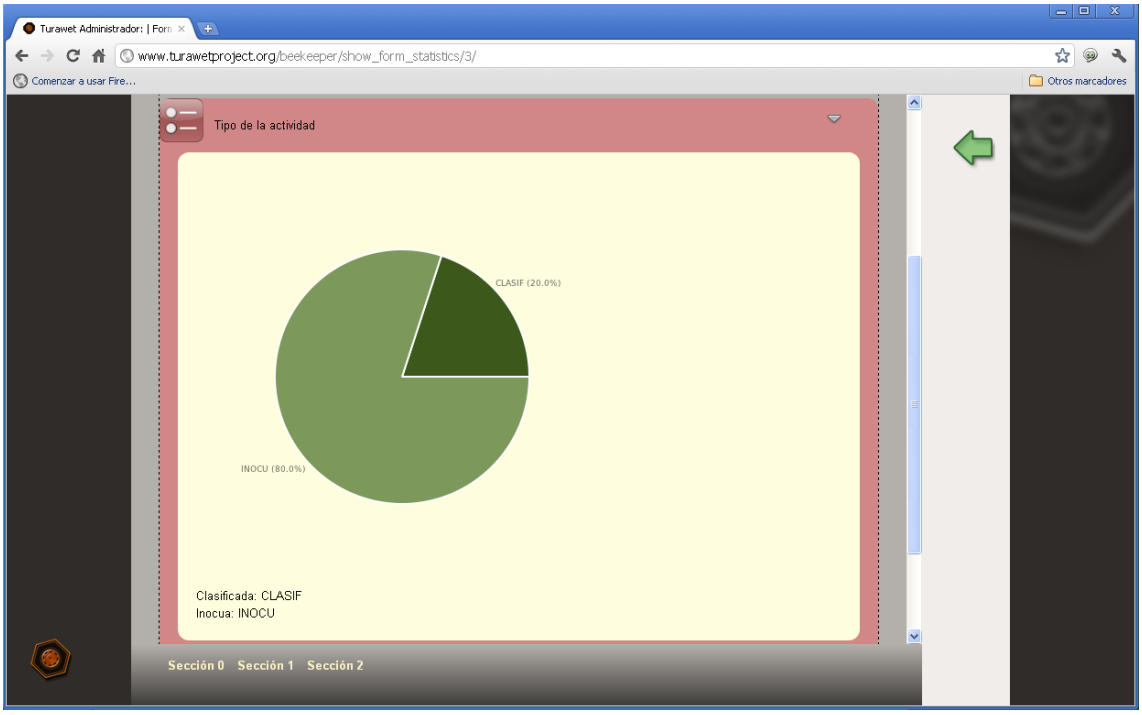


Figura 8-5 Estadísticas de clasificación de las actividades por tipo.

8.3 Departamento de Biología Vegetal de la ULL

Introducción

Para el segundo caso real de uso de la plataforma, contamos con la colaboración del Departamento de Biología Vegetal de La Universidad de La Laguna. Los investigadores de este departamento, realizan muchas tareas de recolección de datos en las afueras de la ciudad, por tanto, han visto una gran utilidad en nuestra aplicación.

El caso real que presentaremos a continuación, también surge del formulario que hemos desarrollado conjuntamente con ellos. Se trata de una de las tantas tareas de investigación e inventario que realizan más a menudo, en concreto, el **inventario de poblaciones de plantas amenazadas**.

Diagrama de casos de uso

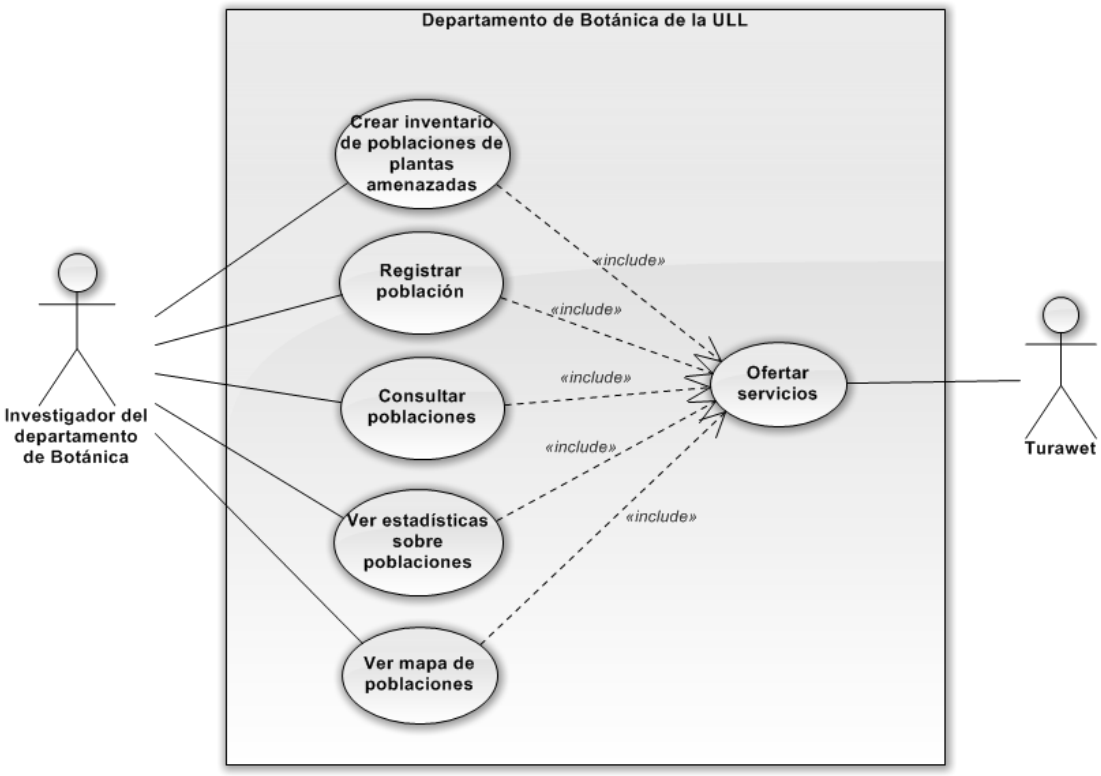


Figura 8-6 Diagrama de casos de uso de una implantación real de Turawet para el Departamento de Biología Vegetal de la ULL.

A diferencia del primer caso, presentado en el apartado anterior, en este solo contaremos con un actor, que hará uso de todas las herramientas que conforman el proyecto Turawet. Este actor es el propio investigador.

Inicialmente, el investigador modelará convenientemente el formulario que describe una población de plantas amenazadas. Una vez enviado al almacén de Turawet, se lo podrá descargar en su móvil Android. Con este formulario ya descargado, se dispondrá a recorrer sus rutas habituales tomando datos de todas aquellas especies de plantas que se encuentren amenazadas. De esta forma, conseguirá registrar poblaciones de plantas (conjuntos cercanos de varios ejemplares) de forma sencilla, cómoda y ahorrando gran cantidad de tiempo en la redacción posterior del informe. Además, las funcionalidades de estadísticas (por ejemplo, la visualización del porcentaje de las principales amenazas detectadas) así como la geolocalización en mapa de las poblaciones; darán un valor añadido de gran interés al investigador.

Sacando partido de todas las funcionalidades que le brinda la aplicación recolectora móvil, el investigador podrá obtener una imagen de la población estudiada, hacer un recuento del número de ejemplares que encuentre en cada población, tomar la posición exacta de donde se encuentran y tomar notas sobre cualquier aspecto que se considere oportuno (como la fenología, orientación o las principales amenazas de la población).

Además con posterioridad a la recolección de datos, haciendo uso del cuadro de mandos de la aplicación de administración, el investigador puede:

1. **Consultar población**: hacer una consulta detallada de todas y cada una las poblaciones de plantas que se hayan recolectado.
2. **Ver estadísticas sobre poblaciones**: el investigador podrá realizar un seguimiento de la evolución de las poblaciones a lo largo del tiempo, como por ejemplo, las variaciones del número de ejemplares. Además, en este punto se puede, por ejemplo, ver cuál es la principal amenaza de las poblaciones en términos globales, obteniendo así un análisis muy útil para la elaboración de informes.
3. **Ver mapa de poblaciones**: posiblemente una de las funcionalidades más valorada por los investigadores del Departamento de Biología Vegetal, sea tener la posibilidad de visualizar, geolocalizadamente, cada una de las poblaciones estudiadas en un mapa con vista satélite.

Prototipo

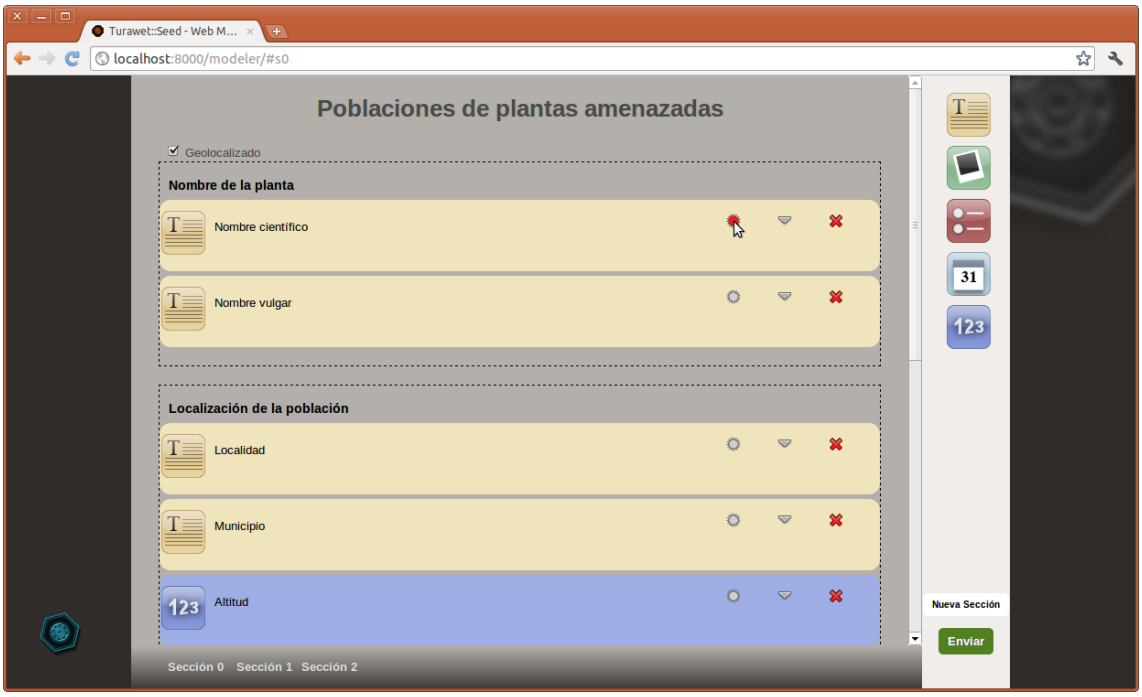


Figura 8-7 Modelado del formulario de poblaciones de plantas amenazadas.

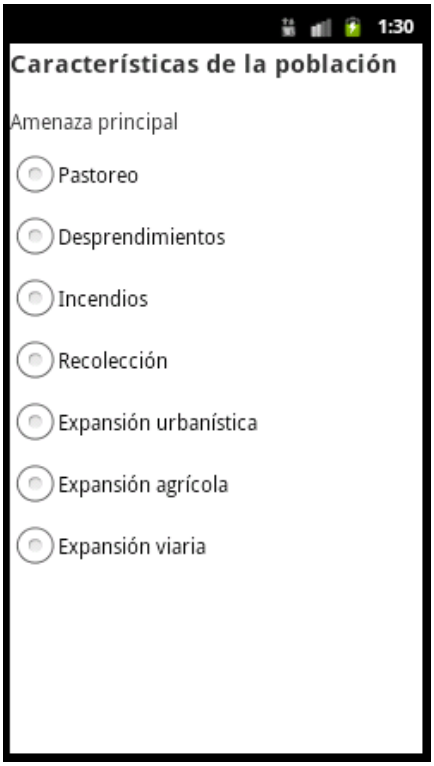


Figura 8-8 Recolección de poblaciones de plantas amenazadas.

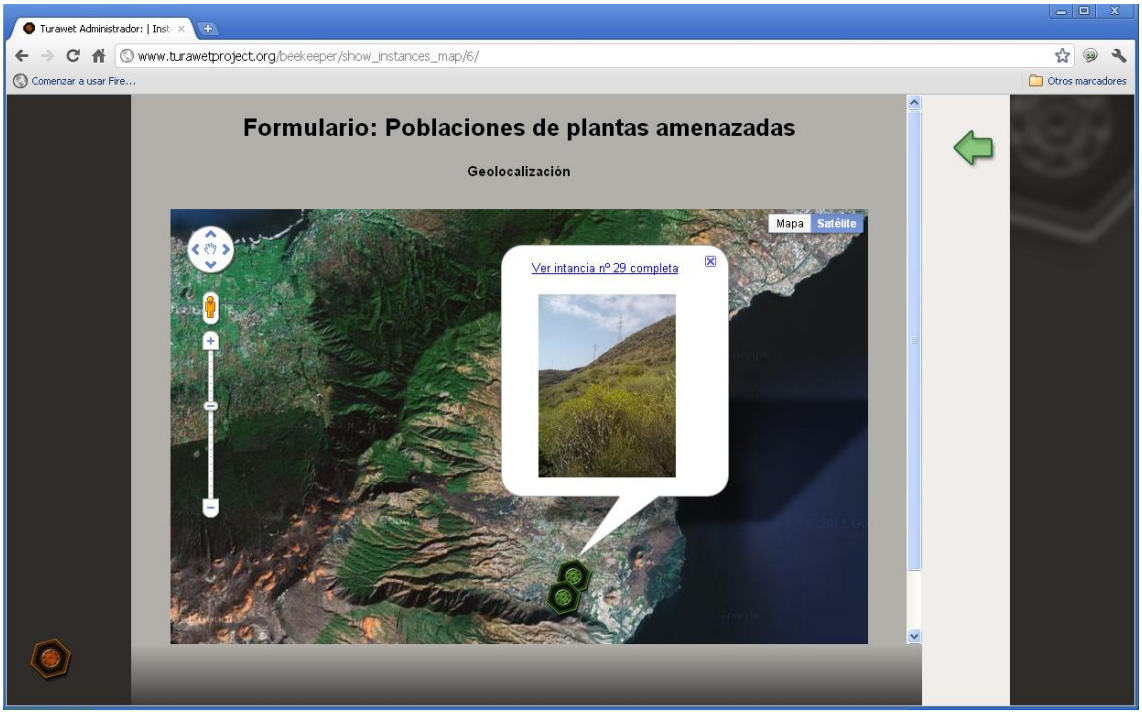


Figura 8-9 Geolocalización de dos de las poblaciones de plantas amenazadas, ilustrando una de ellas.

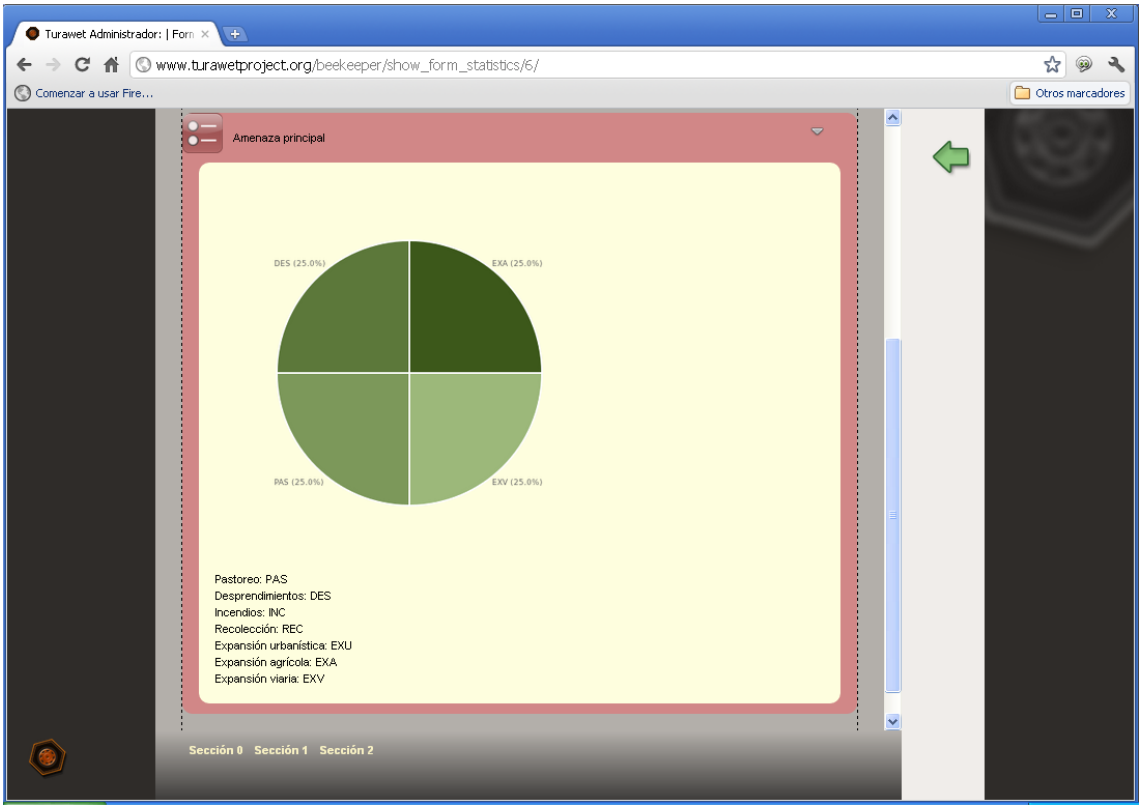


Figura 8-10 Estadísticas con las principales amenazas de las poblaciones.

8.4 Ayuntamiento de La Laguna

Introducción

Finalmente, un tercer caso real que encontramos interesante, surgió a partir de diferentes conversaciones con nuestro director de proyecto, en las cuales intentábamos encontrar un buen argumento para demostrar la potencialidad que tiene Turawet. Vimos que existen varias aplicaciones y páginas web, en donde cualquier persona puede reportar un desperfecto en la vía pública, enviando una imagen de la misma, una descripción y claro está, su localización. Algunas de las más conocidas son FixMyStreet [80] y StreetReport [81].

Nuestra propuesta se titula **Cuidemos La Laguna**, y está pensada para brindar un servicio a los ciudadanos, como los que antes comentábamos. La propuesta se puede llevar a cabo utilizando la aplicación tal y como se presenta en esta memoria, con la peculiaridad de hacer uso de un único formulario para recolectar los datos necesarios. Aprovechándonos de este hecho peculiar, podríamos adaptar la interfaz de la aplicación móvil para emular una aplicación hecha a medida.

Diagrama de casos de uso

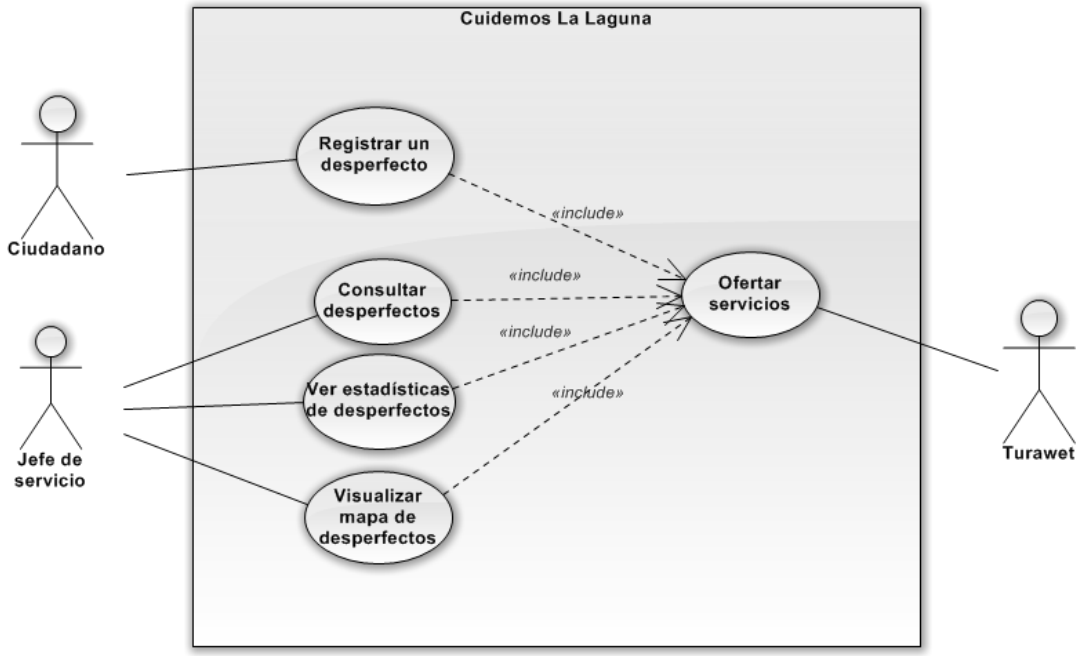


Figura 8-11 Diagrama de casos de uso para el Ayuntamiento de La Laguna.

Los partícipes del escenario serán: un ciudadano actuando como recolector y un representante del Ayuntamiento de La Laguna, que se encargue de revisar los desperfectos que sean reportados. El ciudadano se encargará de registrar desperfectos en la vía pública, tomará fotografías y geolocalizará la incidencia para dejar constancia en el Ayto.

Por otro lado, el Jefe de servicio, a través del cuadro de mandos realizará las siguientes acciones:

1. **Consultar desperfecto**: podrá acceder a los datos que hayan almacenado todos los ciudadanos, para consultar cualquiera en particular.
2. **Ver estadísticas de los desperfectos**: tendrá la posibilidad de ver estadísticas relativas a la gravedad de los desperfectos así como encuestas de satisfacción de los ciudadanos.
3. **Visualizar mapa de los desperfectos**: sobre un mapa de toda la ciudad de La Laguna, podrá ver una panorámica de todos los desperfectos reportados, para sacar conclusiones de cuáles son las zonas más afectadas.

Prototipo

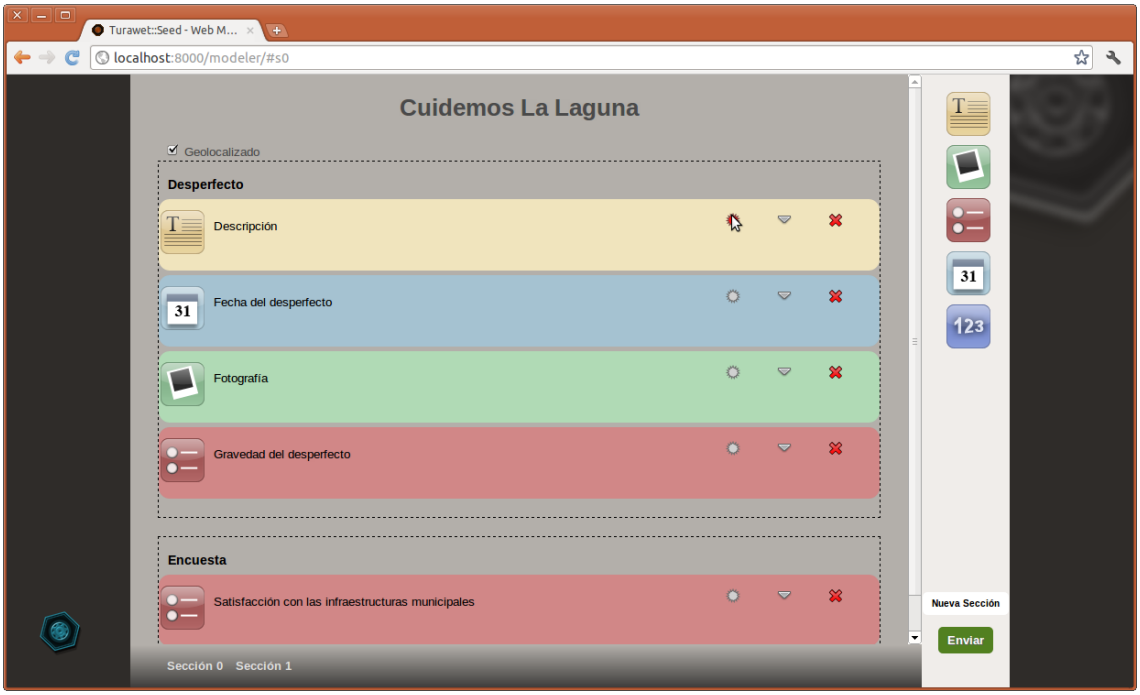


Figura 8-12 Modelado del formulario de "Cuidemos La Laguna".

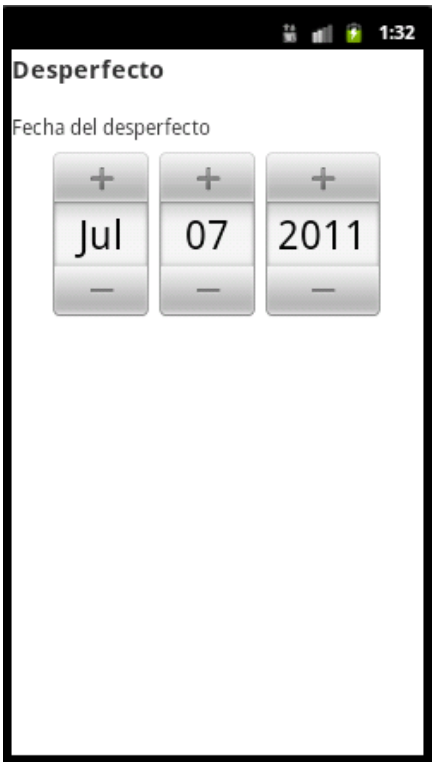


Figura 8-13 Recolección de desperfectos.

Figura 8-14 Geolocalización de los desperfectos ilustrando uno de ellos.

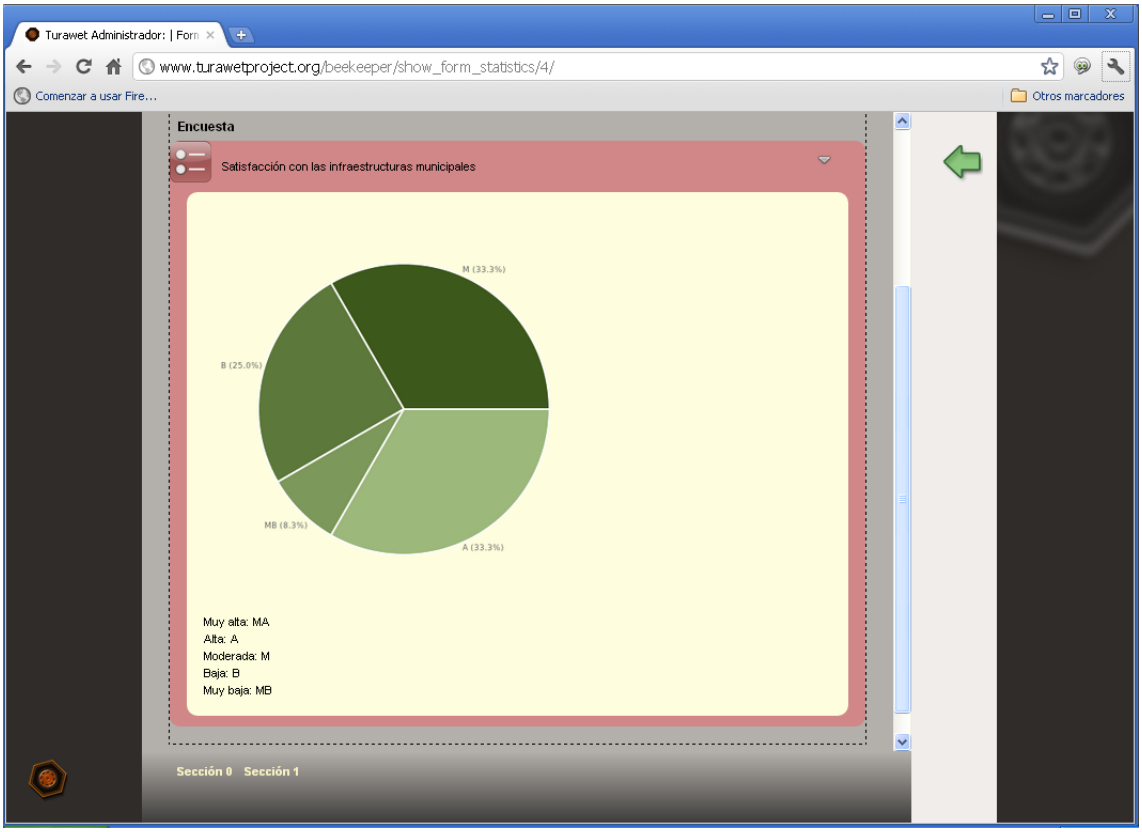


Figura 8-15 Resultados del campo de satisfacción con las infraestructuras municipales.

Parte IV. Conclusiones y futuro

Capítulo 9. Experiencia y conclusiones

Son muchas las conclusiones que podemos extraer de la experiencia de haber desarrollado este gran proyecto, que ha sido, a todas luces enriquecedor en muchos aspectos para nosotros. Nos hemos enriquecido en aspectos tan diversos como el gran número de nuevas tecnologías aprendidas (Android, Django, Servicios Web, jQuery, HTML5, etc…), el aprendizaje o mejoría de nuestras habilidades de trabajo en equipo, la capacidad de toma de decisiones y la adquisición de experiencia en el análisis y diseño de software y sistemas de información. Todo esto pasando por la mejoría de nuestra capacidad de documentación y habilidades de comunicación y defensa de nuestros proyectos ante profesionales o clientes potenciales, coronado finalmente con varias aplicaciones prácticas de nuestro proyecto.

En primer lugar comenzaremos detallando aquellas que probablemente serán las conclusiones más evidentes del proyecto. Estas conclusiones son aquellas relativas al **aprendizaje y uso de nuevas herramientas y tecnologías**.

En este punto cabe destacar nuestra incursión en la plataforma Android, de la que no teníamos, inicialmente, conocimiento alguno a nivel de desarrollo ninguno de los autores del proyecto. Nos vimos obligados a investigar y aprender por nosotros mismos desde el nivel más básico. Durante el desarrollo, pudimos comprobar la fantástica guía oficial para el desarrollador que posee Android. Además, cada vez que fue preciso, nos vimos obligados a consultar en diversas webs por dudas o ejemplos concretos para la implementación de funcionalidades concretas en esta plataforma, mejorando así sustancialmente nuestra capacidad de investigación y de comunicarnos con la comunidad de desarrolladores. Sobre todo, nos vimos muchas veces obligados a sumergirnos en foros especializados en desarrollo de aplicaciones móviles, intentando buscar respuesta a situaciones que no alcanzábamos a comprender y para las que no encontrábamos documentación explicativa o ejemplos específicos.

También nos sirvió de experiencia, al desarrollar para una plataforma relativamente joven como es Android, darnos cuenta de los pocos manuales y artículos sencillos y útiles que se pueden encontrar cuando una tecnología no ha madurado aún lo suficiente. Así, si bien para Java existen multitud de referencias, artículos en la web, tutoriales y documentación en general; muchos de los entresijos del desarrollo para móviles tuvimos que descubrirlos mediante el contacto, por medio de foros especializados, con desarrolladores con una mayor experiencia que nosotros en esta plataforma como se ha comentado anteriormente.

Por supuesto, desarrollar para Android, supuso para nosotros fortalecer nuestros conocimientos de Java y sus librerías y fue precisamente el conocimiento de este lenguaje, lo que nos facilitó en gran medida la adaptación a esta nueva plataforma de desarrollo para dispositivos móviles y tabletas.

Continuando con las conclusiones de aprendizaje de nuevas tecnologías, profundizamos en nuestros conocimientos sobre desarrollo web. Reafirmamos nuestra percepción de que las herramientas web van ganando terreno a las herramientas de escritorio a pasos agigantados, posiblemente por ser aplicaciones que al ejecutarse sobre un navegador, son implícitamente multiplataforma y no requieren de la instalación de software específico, y por tanto, pueden ser utilizadas desde cualquier dispositivo con conexión a internet.

Tuvimos la suerte de desarrollar nuestras aplicaciones web apoyándonos, para la parte del servidor en un *framework* como Django que facilita y agiliza enormemente el desarrollo. A pesar de que nuestros conocimientos sobre el *framework* y sobre el lenguaje Python en general eran muy limitados, no tardamos en descubrir el gran potencial que poseen. Sin duda creemos que el desarrollo de este mismo proyecto utilizando lenguajes de programación tradicionales hubiera sido mucho más largo y engorroso. Evidentemente, no todo son elogios para Django, pues su curva de aprendizaje es bastante pronunciada y, al igual que sucede con la plataforma Android, la documentación es bastante escasa, por lo que muchas veces, cuando surgía un error o duda, la resolución de la misma podía ser muy larga, requiriendo un gran esfuerzo de búsqueda en engorrosos foros y en manuelas muchas veces incompletos y poco clarificadores. En general echamos en falta un mayor número de ejemplos de código, tanto para Python/Django como para Android. Esta necesidad que tuvimos puede deberse a estar acostumbrados a lenguajes ya más asentados, como Java o C, para los que es muy sencillo encontrar ejemplos.

Continuando con el desarrollo en la nube, esta vez desde el marco del diseño de interfaces web, hemos de comentar que a pesar de que podía ser un campo en el que teníamos más de experiencia que en los otros dos, lo cierto es que la experiencia que teníamos en el diseño de interfaces de aplicaciones web era con las tecnologías HTML y CSS más tradicionales. Por ello, el desarrollo de las interfaces web de nuestro proyecto, utilizando la novedosa versión del lenguaje de marcas HTML (HTML5) y explotando el gran potencial de JavaScript, sobre todo con el empleo de sus potentes librerías de jQuery, hasta entonces desconocidas por nosotros; hizo que el desarrollo de las interfaces web no fuera un trabajo tan sencillo como cabía esperar en un principio. Bien es cierto que finalmente la experiencia de haber utilizado todas estas tecnologías nos ha aportado mucho conocimiento y estamos muy satisfechos con su aprendizaje.

Otro punto de experiencia tecnológica interesante vino de la mano de la integración y la comunicación entre los módulos, conseguido a través de los servicios web que desarrollamos. Una vez más, en este punto estamos muy agradecidos a Python y en concreto a la librería Soaplib de dicho lenguaje, que nos facilitó enormemente el desarrollo. El desarrollo de *web services* nos pareció una experiencia muy enriquecedora, debido, en gran parte a que son una forma de comunicar herramientas web sencillamente y a que se pueden aplicar a un gran abanico de proyectos que requieran comunicación con otras herramientas a través de internet.

Ahondando en el tema de la integración y la comunicación entre sistemas comenzado en el párrafo anterior, es importante indicar que una de las mayores experiencias que podemos extraer del desarrollo de este proyecto ha sido el darnos cuenta de la importancia de la interoperabilidad entre sistemas y el valor que tiene dar una solución integral. Para nosotros fue fundamental que las tres herramientas desarrolladas fueran capaces de comunicarse e interoperar correctamente. Además, después de haber realizado un extenso estudio del estado del arte y tras tener todo el sistema funcionando nos dimos cuenta de que para que una herramienta como esta tenga éxito, es realmente importante ofrecer una solución integral, formada por diferentes módulos que cubran todas las necesidades del dominio al que está dirigida.

Una conclusión importante que extraemos de este proyecto es la importancia de la Ingeniería del Software, entendiendo la ingeniería del software como el conjunto de métodos y

técnicas que nos permitan desarrollar software atendiendo, en nuestro caso, a las fases de análisis, diseño e implementación. Para nosotros fue fundamental hacer un análisis del proyecto para recabar todos los requisitos y tener una idea del dominio en el que nos encontrábamos. Asimismo, fue de vital importancia tener una etapa de diseño donde se definieran diagramas que nos permitieran tener una visión sencilla y global del sistema y de cada uno de sus módulos. Sobre todo, el diseño nos fue útil de cara a anticipar problemas e implementar nuestra idea teniendo claras las líneas de trabajo a seguir desde el primer momento. Además, es indiscutible que haber realizado una etapa de análisis y otra de diseño previas (o en ocasiones con iteraciones paralelas) a la etapa de implementación redunda en una mejora sustancial de la documentación del proyecto, siendo de gran utilidad poseer estos diagramas para comprender y hacer comprender a otras personas el potencial y funcionamiento de nuestro sistema.

Los diagramas de casos de uso nos permitían tener una visión general de las funcionalidades de cada uno de los módulos del proyecto y de cómo los usuarios debían interactuar con el sistema. Por otra parte, los diagramas de secuencia nos permitían analizar el flujo o la secuencia de eventos que tendría lugar en cada caso de uso. Los diagramas de componentes y de despliegue también nos permitieron tener una visión en conjunto de la aplicación y, sobre todo, los consideramos muy útiles de cada a la explicación del proyecto y la documentación. Por último tenemos que destacar los *mockups,* que no son más que unos diagramas o bocetos que nos permiten hacer un diseño preliminar de las interfaces. Con ellos llegamos a acuerdos de diseño y pudimos hacer pruebas de usabilidad con usuarios potenciales antes de comenzar el desarrollo real de las interfaces. En este punto tenemos que destacar la magnífica experiencia utilizando la herramienta Balsamiq para la generación de estos bocetos.

El caso del diseño de la base de datos tiene mención aparte. A pesar de que habíamos cursado estudios específicos de Bases de Datos, diseñar nuestra propia base de datos no fue tarea sencilla en absoluto. Estábamos ante un sistema bastante complejo que requirió un diseño meticuloso y que se fue modificando conforme avanzaba el proyecto, depurándose y ajustándose cada vez más a nuestras necesidades. Nos dimos cuenta una vez más de lo importante que es el almacén de información en un sistema como el nuestro, máxime cuando la explotación de los datos almacenados es una pieza fundamental del sistema.

Como **formación transversal del proyecto** también cabe destacar que mejoramos nuestros básicos conocimientos de diseño gráfico utilizando herramientas como Photoshop [82] o GIMP [83]para la creación de los iconos de los diversos módulos del proyecto. Por otra parte y como aprendizaje más novedoso utilizamos una herramienta de generación de gráficos vectoriales (Inkscape) para generar el logotipo del proyecto.

Un aspecto que mejoramos bastante fueron nuestros conocimientos de uso de herramientas de depuración (en especial del depurador de Eclipse [84], que utilizamos tanto en local como en remoto). A pesar de que habíamos utilizado ya herramientas de depuración bastantes veces e incluso el propio depurador del Eclipse, durante este proyecto tuvimos que hacer un uso extensivo de él, terminando los tres muy satisfechos con el citado depurador. Tanto es así que lo utilizamos tanto para depurar código Python (servidor Django) como Java (Android) y a su vez lo utilizamos en local (modelador o servidor local)a la par que en remoto (servidor y teléfono) en diferentes ocasiones.

Tuvimos varias reuniones muy enriquecedoras con potenciales clientes, asesores de patentes y expendeduría, así como con otros profesionales del sector de las tecnologías de la información. Esto nos permitió ver el proyecto como un producto potencial y obtener asesoramiento especializado y adquirir conocimientos de emprendeduría y del funcionamiento de las patentes, además de habituarnos a reunirnos con diferentes profesionales y comentar abiertamente nuestras ideas e intentar defender nuestro desarrollo.

Obtuvimos una grata realimentación de la utilización, en casos reales, de nuestras herramientas (caso de Gerencia de urbanismo de La Laguna o el departamento de Biología Vegetal de la Universidad de La Laguna). Muchas veces, en nuestra opinión, cuando un proyecto queda encajado en un marco teórico y no se ve una aplicación práctica directa no es tan enriquecedor ni satisfactorio como cuando se tiene la oportunidad de ver el proyecto que se ha desarrollado en funcionamiento, aplicado a varios casos reales donde puede ser útil y facilitar trabajo a otros profesionales.

Sin duda, en todo este tiempo invertido en el proyecto Turawet, mejoramos nuestra experiencia como grupo de trabajo, desarrollando por primera vez un proyecto de grandes dimensiones en el que nosotros mismos tuvimos los roles correspondientes a cada una de las etapas del desarrollo de software, como puedan ser los roles de analista, diseñador y desarrollador. Por otra parte, tuvimos un gran número de reuniones internas al equipo de desarrollo en las que nos vimos obligados a llegar a múltiples acuerdos y discutir los diferentes aspectos de diseño e implementación de las diferentes herramientas. Estas reuniones muchas veces fueron largas e intensas, pero siempre terminamos llegando a la mejor solución consensuada entre los tres.

Como ya hemos comentado anteriormente, mejoramos nuestra capacidad de investigación y auto-aprendizaje, especialmente en tecnologías recientes donde la documentación y los ejemplos no abundaban.

Obtuvimos la experiencia de realizar análisis de usabilidad con usuarios potenciales de las herramientas en las diversas reuniones, con clientes potenciales o con los colaboradores de los casos prácticos que implementamos. La experiencia de uso de estos colaboradores nos permitió mejorar varios aspectos de la interfaz de cara a aumentar su usabilidad.

Continuamos habituándonos a utilizar módulos y librerías de terceros ya desarrolladas para evitar una carga de trabajo añadida para desarrollar funcionalidades ya existentes. Asimismo, continuamos habituándonos a utilizar APIs de terceros (como es el caso de GoogleMaps) y sacar el máximo provecho a las herramientas ya existentes que nos pueden aportar un valor añadido con alguna funcionalidad concreta.

En general mejoramos nuestras capacidades de documentación y explicación de nuestro desarrollo, documentando todas y cada una de las fases de desarrollo de software en esta gran y descriptiva memoria que ha pretendido sintetizar este ambicioso proyecto.

Para terminar, y haciendo balance del proyecto, cabe destacar el gran tiempo que hemos dedicado desde que empezara a gestarse, hace aproximadamente un año (Julio de 2010) hasta la fecha actual (Julio de 2011). Si bien la dedicación al proyecto se fue intensificando conforme se acercaban los últimos meses de desarrollo, hemos de decir que siempre ha habido un flujo continuo de trabajo, siendo cierto que en los últimos meses el avance ha sido mucho mayor en las funcionalidades y el desarrollo general del proyecto. Este

avance final se ha debido sobre todo a un mayor conocimiento de las tecnologías, tras superar la etapa de aprendizaje, así como una mayor definición y concreción del proyecto, con unos objetivos más concretos y un mayor conocimiento general del dominio y las funcionalidades a desarrollar. Sin embargo, creemos que la experiencia que nos ha dado este proyecto, tanto en el aspecto tecnológico, como en el del trabajo en equipo, así como en el de desarrollar una idea interesante (pasando por todas las etapas de desarrollo de software) para, finalmente, ver la aplicación del proyecto a casos reales, ha sido tan interesante que compensa con creces el esfuerzo que hemos dedicado.

Capítulo 10. Líneas futuras de trabajo

En este capítulo se recogen todas las ideas futuras de trabajo, que por distintas razones, no hemos podido incluir en el prototipo entregado.

En definitiva, el proyecto ha sido muy ambicioso, ha habido y sigue habiendo muchos aspectos en los que trabajar, pues teníamos un gran número de ideas para desarrollar desde el principio. Desgraciadamente, no contamos con el tiempo suficiente para desarrollarlas todas. Tuvimos que centrarnos en las principales, para lograr un prototipo que fuera capaz de funcionar correctamente, de forma integral, a pesar de ser un sistema complejo y heterogéneo formado por diferentes módulos que basados en tecnologías muy dispares.

Principalmente existen dos vertientes claramente diferenciadas, en las que podemos clasificar las líneas futuras de trabajo. La primera de ellas, surge a partir de lo comentado al comienzo del capítulo y tiene relación con el grado de ambición que teníamos a la hora de enfrentarnos a un proyecto tan grande como este. Son bastantes las cosas que se nos quedan sin implementar de las planteadas en las etapas de análisis y diseño, y otras tantas las que quedan en nuestras cabezas como futuribles, pero como es de entender, en el marco de un proyecto de fin de carrera, el tiempo es limitado y no todo lo propuesto en un proyecto tan grande como este puede ser desarrollado finalmente.

Por otro lado, el segundo conjunto de ideas que se nos quedan fuera de este desarrollo, aparecen a raíz de correcciones y mejoras que pretendemos realizar sobre el producto presentado. Como se dice comúnmente, el papel todo lo aguanta y nosotros lo pudimos comprobar por última vez en la carrera. Puntualmente diseñamos la aplicación de la forma que nos parecía más correcta, intentamos cubrir aspectos de usabilidad, elegancia, eficiencia y por supuesto, brindar las funcionalidades que se habían acordado. Pero una vez que contábamos con el prototipo ya funcional, pudimos comprobar que aún existían muchos aspectos en donde seguir mejorando.

Haciendo un recuento, enumeraremos todas las ideas que se han quedado sin participación en el primer prototipo del proyecto Turawet:

- **Completar el listado de campos para el modelador y el recolector**. En el diseño original planteábamos la posibilidad de contar con campos tipo *checkbox*, *combo*, *widget para checkbox* de tres estados (on / off / deshabilitado), así como campos para vídeos y audio.
- **Materializar las propiedades de los campos.** En el modelador tenemos la posibilidad de añadir propiedades a los campos. Por ejemplo establecer el máximo número de caracteres de un campo de tipo texto. Estas restricciones no se están teniendo en cuenta en el recolector, y tampoco están completadas en le modelador.
- **Comprobar la obligatoriedad de los campos.** Cada campo puede ser obligatorio o no. Esto se permite en el modelador, pero en el recolector no se está teniendo en cuenta esta restricción. Cuando un usuario complete una instancia, el recolector debe comprobar que todos los campos obligatorios hayan sido cumplimentados.

- **Campos tipo grupo y listas**. Otra posibilidad de estructurar los campos, es a través de grupos y listas, como se planteó en los diagramas de la fase de diseño. Los grupos permitirían formar campos complejos a partir de tipos simples, sencillamente agrupándolos. Se podría definir un grupo *persona*, que conste de su *nombre* y *apellido*, siendo cada uno de estos un campo de texto simple. Por otro lado, tendríamos listas, que darían la posibilidad de tener campos con múltiples valores en las instancias. Una aplicación de este caso se encuentra, por ejemplo, para recoger todas las personas que viven en una vivienda. El usuario podría añadir dinámicamente nuevos elementos de tipo persona mientras rellena la instancia.
- **Gestión de usuarios y grupos.** Como ya comentamos, la gestión de usuarios no se pudo llevar a cabo completamente, creándose simplemente varios usuarios con todos los privilegios para ver todos los formularios e instancias. La idea es que los usuarios tendrán sus propias cuentas de usuario para utilizar las aplicaciones. Estos formarían parte de grupos, y podrían acceder únicamente a formularios que se asocien a sus grupos. De esta manera conseguimos restringir el acceso de los formularios solamente a aquellos formularios que les correspondan.
- **Conexión segura entre los módulos**. Un punto a favor en términos de seguridad, es permitir que los módulos se comuniquen entre sí a través de una conexión segura y cifrada, para evitar posibles escuchas no deseadas. Además, se pretende que el acceso al modelador y al administrador, también sea seguro, sobre *https*.
- **Posibilidad de almacenar instancias en el dispositivo móvil**. Otra idea que también hemos nombrado, es permitir al usuario guardar en su teléfono una instancia de forma temporal, para que la pueda enviar al repositorio en cualquier otro momento. Esta funcionalidad es fundamental, ya que los recolectores no siempre tienen que contar con una conexión de datos activa.
- **Enviar instancias incompletas al repositorio.** Otra funcionalidad que vemos muy útil, es poder enviar al repositorio instancias que no hayan sido completadas. Con esto, los usuarios recolectores, serían capaces de comenzar una instancia en un teléfono móvil, almacenarla temporalmente en el repositorio, y finalmente, completarla desde un recolector web.
- **Recolector web.** De la mano con la idea anterior, queremos brindar la oportunidad de que un usuario recolector pueda crear o continuar instancias desde un navegador en un ordenador y poder aprovecharse de las ventajas que tienen sobre los dispositivos móviles en cuanto a comodidad para redactar o el tamaño de la pantalla.
- **Recolector iOS**. Con la idea de ampliar el mercado lo más que se pueda, además de desarrollar recolectores para la plataforma Android, queremos contar con colectores que funcionen sobre dispositivos iOS, así como los *iPhone*, *iPad* o *iPod*.
- **Recolector sobre *tablets*.** Otra posibilidad de apertura del mercado, es implementar una versión del colector que se aproveche de las ventajas que ofrecen los *tablets*. Ya sean Android o iOS. Creemos que la comodidad de trabajo que se alcanza con este tipo de aparatos es muy buena y podríamos sacarle provecho. El prototipo actual puede funcionar sobre *tablets* Android, pero no está ajustada al tamaño ampliado de sus pantallas y la visualización no es todo lo buena que debería.

- **Personalizar los tipos campos disponibles según el grupo de usuarios (poder asociarles acciones).** En casos concretos puede haber grupos de usuarios que requieran tipos de datos complejos específicos que tengan acciones asociadas (como pueda ser un campo de tipo matrícula para un policía que esté asociado con un programa de reconocimiento de dígitos para la matrícula a través de una fotografía).
- **Búsquedas avanzadas con filtros en el administrador.** En el lado del administrador, y con la idea de explotar los datos de todas las formas posibles, creemos que la inclusión de un componente que permita búsquedas avanzadas mediante filtros sobre los campos, sería una gran idea. Para citar un ejemplo de uso, el usuario administrador podría seleccionar de todos las instancias de un formulario, aquellas que se hayan realizado en el último año, y poder hacer comparativas de algún tipo, como visualizar geolocalizadamente, solamente las instancias del año filtrado.
- **Módulo de generación de informes.** A partir de los datos recogidos con todas las instancias, el administrador contará con la capacidad de generar informes en PDF, en donde se muestren los datos más relevantes de su campaña de recogida de datos.
- **Firmar digitalmente las instancias.** Una de las primeras ideas que nos plantearon, fue la posibilidad de enviar instancias firmadas digitalmente por el usuario que las realiza. Este deberá contar con un certificado digital que lo identifique en su dispositivo, y antes de enviar la instancia al repositorio, firmaría los datos que haya recogido.
- **Estadísticas con diagramas de barras.** El componente de estadísticas en el administrador, únicamente permite generar gráficas circulares. Otra posibilidad sería que se generen gráficas en forma de barras o cualquier otro estilo que sea útil.

Además de las ideas enumeradas anteriormente, en versiones posteriores del prototipo se deberían mejorar las interfaces de los tres módulos, basándonos siempre en la experiencia de los usuarios y realizando cambios que fomenten la usabilidad de las aplicaciones.

Referencias

[1] Proyecto GeoBloc
Escuela Técnica Superior de Ingeniería Informática, Universidad de La Laguna
http://code.google.com/p/ull-etsii-geobloc

[2] HTML5 (HyperText Markup Language, versión 5)
World Wide Web Consortium (W3C)
http://dev.w3.org/html5/spec/Overview.html

[3] jQuery
Framework JavaScript
http://www.jquery.org

[4] CSS3 (Cascade Style Sheet, versión 3)
World Wide Web Consortium (W3C)
http://www.w3.org/TR/CSS/

[5] Android
Plataforma móvil de Google
http://www.android.com

[6] Gerencia de Urbanismo de La Laguna
San Cristóbal de La Laguna, Tenerife, España
http://www.gerenciaurbanismo.com/

[7] Departamento de Biología Vegetal de la Universidad de La Laguna
Universidad de La Laguna, Tenerife, España
http://www.ull.es/view/institucional/ull/Biologia_Vegetal/es

[8] Emprende ULL
Universidad de La Laguna, Tenerife, España
http://emprendeull.ning.com/

[9] OTRI (Oficina de Transferencia de Resultados de Investigación)
Universidad de La Laguna, Tenerife, España
http://www.otri.ull.es/otriweb/index.php

[10] XML (Extensible Markup Language)
World Wide Web Consortium (W3C)
http://www.w3.org/XML/

[11] Amazigh
Lenguaje Bereber
http://es.wikipedia.org/wiki/Bereberes

[12] Data minning
http://en.wikipedia.org/wiki/Data_mining

[13] Inkscape
Editor de gráficos vectoriales de código abierto.
http://inkscape.org/

[14] JavaRosa
Open Rosa Consortium
http://www.open-mobile.org/technologies/javarosa-open-rosa-consortium

[15] XForms
World Wide Web Consortium (W3C)
http://www.w3.org/TR/xforms11/

[16] J2ME (Java Mobile Edition)
ORACLE
http://www.oracle.com/technetwork/java/javame/index.html

[17] OpenRosa
http://www.openrosa.org

[18] World Wide Web Consortium (W3C)
http://www.w3.org

[19] Mobile technology: Small Wonder
Nizar Diamond Ali, *InpaperMagzine*
http://www.dawn.com/2011/02/27/mobile-technology-small-wonder.html

[20] Cell-Life
http://www.cell-life.org/

[21] Cell-Life Capture
http://www.cell-life.org/capture

[22] Emit
http://www.emitmobile.com.za

[23] SANGONeT Web
http://www.ngopulse.org/

[24] Dimagi
http://www.dimagi.com/

[25] CommCare
http://www.dimagi.com/commcare

[26] EpiSurveyor
http://www.episurveyor.org/user/index

[27] Wall Street Journal Award for Technology Innovation in Healthcare
http://www.dowjones.com/innovation/ei_winners_2009.html#Healthcare

[28] Aquaya
http://www.aquaya.org

[29] ODK (Open Data Kit)
http://opendatakit.org/

[30] GATHERdata
http://www.healthnet.org/gather

[31] AED (Academy for Educational Development)
http://www.aed.org/

[32] GitHub
https://github.com/

[33] OpenXdata
http://www.openxdata.org/

[34] Django
Framework para el desarrollo de aplicaciones web en Python
https://www.djangoproject.com/

[35] UML (Unified Modeling Language)
http://www.uml.org

[36] Ruby on Rails VS Django
Raúl González Duque, mundogeek.net
http://mundogeek.net/archivos/2007/08/20/ruby-on-rails-vs-django/

[37] Python
http://www.python.org/

[38] Ruby on Rails
http://www.rubyonrails.org.es/

[39] Ruby
http://ruby-lang.org/es/

[40] AJAX (Asynchronous JavaScript And XML)
http://en.wikipedia.org/wiki/AJAX

[41] MVC (Model-View-Controller)
http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[42] JSON (JavaScript Object Notation)
http://www.json.org/

[43] DSL (Domain Specific Language)
http://en.wikipedia.org/wiki/Domain-specific_language

[44] ElementTree
http://docs.python.org/library/xml.etree.elementtree.html

[45] SAX
http://www.saxproject.org/

[46] DOM (Document Object Model)
http://www.w3.org/DOM/

[47] XML Schema
http://www.w3.org/XML/Schema

[48] iOS
http://developer.apple.com/devcenter/ios/index.action

[49] CORBA (Common Object Request Broker Architecture)
http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

[50] Java RMI (Remote Method Invocation)
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

[51] Soaplib
http://soaplib.github.com/soaplib/2_0/

[52] Suds
https://fedorahosted.org/suds/

[53] Applets
http://es.wikipedia.org/wiki/Applet

[54] SOAP (Simple Object Acces Protocol)
http://www.w3.org/TR/soap/

[55] REST (Representational State Transfer)
http://en.wikipedia.org/wiki/Representational_State_Transfer

[56] SoftwareIdeasModeler
Aplicación ligera para el desarrollo de diagramas UML.
http://www.softwareideas.net/

[57] Mockup
http://en.wikipedia.org/wiki/Mockup

[58] Balsamiq Mockups
Herramienta de diseño de interfaces a base de mockups
http://balsamiq.com/products/mockups

[59] JavaScript
http://en.wikipedia.org/wiki/JavaScript

[60] OOP
http://en.wikipedia.org/wiki/Object-oriented_programming

[61] Django templates
https://docs.djangoproject.com/en/dev/ref/templates/

[62] Inline/In-Place Edit Plugin
https://github.com/caphun/jquery.inlineedit

[63] Windows Phone 7
Sistema operativo móvil de Microsoft.
http://www.microsoft.com/windowsphone/es-es/default.aspx?cmpid=MSCOM_ES_HP_Nav_Todos

[64] Pull-Parser
http://www.xmlpull.org/

[65] KSOAP2
Librería SOAP ligera para plataformas Android
http://code.google.com/p/ksoap2-android/

[66] SQLITE
http://www.sqlite.org/

[67] Android 2.3 Gingerbread
http://developer.android.com/sdk/android-2.3.3.html

[68] Android 2.2 Froyo
http://developer.android.com/sdk/android-2.2.html

[69] Samsung
http://www.samsung.com/es/

[70] HTC
http://www.htc.com/es/

[71] WSDL
http://www.w3.org/TR/wsdl

[72] SQL
http://en.wikipedia.org/wiki/SQL

[73] Oracle XE
Versión gratuita de la base de datos Oracle.
http://www.oracle.com/technetwork/database/express-edition/overview/index.html

[74] Pycha
Librería basada en Cairo para dibujar gráficas, desarrollada por Lorenzo Gil.
http://www.lorenzogil.com/projects/pycha/

[75] Google API
http://code.google.com/intl/es/apis/maps/

[76] DAO (Data Access Object)
http://en.wikipedia.org/wiki/Data_access_object

[77] django command extensions
https://github.com/django-extensions/django-extensions

[78] Cairo
Librería para creación de gráficos 2D.
http://cairographics.org/pycairo/

[79] Dalvik
http://www.dalvikvm.com/

[80] FixMyStreet
http://www.fixmystreet.com/

[81] StreetReport
http://streetreport.co.uk/

[82] Photoshop
http://www.adobe.com/es/products/photoshop.html

[83] GIMP
http://www.gimp.org/

[84] Eclipse
http://www.eclipse.org/

Apéndice: Casos de uso.

10.1 Casos de uso del Modelador

Identificador: *CU-M-1*

Nombre: “Añadir sección”.

Descripción

El modelador añade una nueva sección al formulario.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.

Post-condiciones

1. Se crea una instancia de sección.
2. Se asocia la sección con el formulario.

Frecuencia

Frecuente.

Flujo de escenario principal

1. El usuario accede a la pantalla de creación de un formulario.
2. El usuario pulsa el botón de añadir una nueva sección.
3. Aparece la nueva sección en el documento lista para arrastrar campos en su interior.

Identificador: *CU-M-2*

Nombre: “Cambiar nombre de la sección”.

Descripción

El usuario modelador modifica el nombre de una sección del formulario.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. El usuario ha añadido una sección al formulario

Post-condiciones

1. Se modifica el nombre de la instancia de la sección correspondiente asociada al formulario.

Frecuencia

Muy frecuente.

Flujo de escenario principal

1. El usuario pulsa sobre el nombre de la sección.
2. Introduce el nuevo nombre de la sección y pulsa en guardar.
3. El nombre se actualiza en el elemento visual que representa la sección.

Identificador: *CU-M-5*

Nombre: “Añadir grupo”.

Descripción

El modelador añade un nuevo grupo de campos a una sección al formulario.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Se ha creado al menos una sección.

Post-condiciones

1. Se crea una instancia de grupo.
2. Se asocia el grupo con la sección correspondiente.

Frecuencia

Frecuente.

Flujo de escenario principal

1. El usuario accede a la pantalla de creación de un formulario.
2. El usuario arrastra el grupo sobre una sección.
3. Aparece el nuevo grupo de la sección en el documento, disponible para arrastrar campos en su interior.

Identificador: *CU-M-7*

Nombre: “Eliminar grupo”.

Descripción

El modelador elimina un grupo de una sección del formulario.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Se ha creado al menos una sección.
3. Existe al menos un grupo en una sección.

Post-condiciones

1. Se localiza la instancia del grupo dentro de la sección.
2. Se eliminan todos los campos del grupo.
3. Se desasocia el grupo de la sección correspondiente.
4. Se elimina el grupo.

Frecuencia

Frecuencia normal.

Flujo de escenario principal

1. El usuario accede a la pantalla de creación de un formulario.
2. El usuario pulsa sobre el botón de eliminar de un grupo.
3. Desaparece el grupo de la sección en el documento.

Identificador: *CU-M-8*

Nombre: “Permitir valores múltiples”.

Descripción

El modelador establece un grupo como lista marcando la opción correspondiente en la interfaz.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Se ha creado al menos una sección.
3. Existe al menos un grupo en una sección.

Post-condiciones

1. Se localiza la instancia del grupo dentro de la sección.
2. Se modifica el atributo lista y se establece como verdadero en la instancia.

Frecuencia

Frecuencia normal.

Flujo de escenario principal

1. El usuario accede a la pantalla de creación de un formulario.
2. El usuario pulsa sobre el checkbox de hacer lista un grupo.
3. El checkbox queda marcado fijando el grupo como una lista.

Identificador: *CU-M-9*

Nombre: “Introducir opciones de un radio/combo”.

Descripción

El modelador añade una nueva opción a un campo de tipo radio o combo.

Actores

Usuario modelador

Precondiciones

1. El usuario ha accedido a la pantalla de creación de un nuevo formulario.
2. Se ha creado al menos una sección.
3. Existe al menos un campo de tipo radio o combo.

Post-condiciones

1. Se crea una nueva instancia de opción.
2. Se asocia la opción con el campo correspondiente.

Frecuencia

Muy frecuente.

Flujo de escenario principal

1. El usuario pulsa sobre el botón de añadir nueva propiedad a campo.
2. Aparece una nueva propiedad bajo el campo con su etiqueta y valor editables.

Identificador: *CU-M-10*

Nombre: “Añadir metadatos”.

Descripción

El modelador modifica el valor de un atributo del formulario.

Actores

Usuario modelador

Precondiciones

1. Se ha generado un formulario.

Post-condiciones

1. Se modifica el valor del atributo de la instancia del formulario.

Frecuencia

Muy frecuente.

Flujo de escenario principal

1. El usuario modifica el nombre del formulario o establece que se trata de un formulario geolocalizado.
2. Se modifica el valor del atributo en la interfaz.

10.2 Casos de uso del Recolector

Identificador: CU-R-3

Nombre: Enviar incompleta.

Descripción

El usuario envía una instancia incompleta al repositorio

Actores

Recolector.

Repositorio.

Precondiciones

1. El usuario ha creado una nueva instancia de un formulario.

Post-condiciones

1. La instancia es enviada y almacenada en el repositorio como incompleta.

Frecuencia

Muy frecuente

Flujo de escenario principal

1. El usuario ha creado una nueva instancia de un formulario y la ha rellenado parcialmente.
2. Selecciona la opción "Enviar incompleta" y la instancia se envía y almacena en el repositorio, como una instancia incompleta.

Identificador: CU-R-4

Nombre: Guardar en teléfono.

Descripción

El usuario comienza una instancia de un formulario, pero decide guardarla en el teléfono en lugar de enviarla. La instancia puede estar completa o no.

Actores

Recolector.

Precondiciones

1. El usuario ha creado una nueva instancia de un formulario.

Post-condiciones

1. La instancia se almacena en el teléfono.

Frecuencia

Frecuente

Flujo de escenario principal

1. El usuario ha creado una nueva instancia de un formulario. Es indiferente si la instancia esta completa o no.
2. Selecciona la opción "Guardar en local" y la instancia se almacena en el teléfono, para ser enviada posteriormente.

Identificador: CU-R-5

Nombre: Continuar desde teléfono móvil

Descripción

El usuario selecciona una instancia que tenga almacenada en el teléfono. La instancia ha sido guardada previamente.

Actores

Recolector.

Repositorio.

Precondiciones

1. El usuario ha creado una nueva instancia de un formulario.
2. El usuario a decido no enviar la instancia y la ha almacenado en el dispositivo móvil.

Post-condiciones

1. La instancia se envía al repositorio.

Frecuencia

Frecuente

Flujo de escenario principal

1. El usuario ha creado una instancia y la ha guardado en su teléfono para continuarla posteriormente.
2. El usuario selecciona la instancia guardada para completarla.
3. El usuario completa la instancia y la envía al repositorio.

Identificador: CU-R-6

Nombre: Descargar instancia incompleta del repositorio.

Descripción

El usuario obtiene una instancia que ha guardado previamente en el repositorio, para continuarla.

Actores

Recolector.

Repositorio.

Precondiciones

1. El usuario ha enviado una instancia incompleta al repositorio.

Post-condiciones

1. El usuario obtiene la instancia incompleta desde el repositorio.

Frecuencia

Normal.

Flujo de escenario principal

1. El usuario ha enviado una instancia incompleta al repositorio.
2. Consulta al repositorio las instancias que ha almacenado.
3. Selecciona la que desea y la descarga. Ya puede continuar rellenándola.

10.3 Casos de uso del Administrador

Identificador: *CU-A-14.*

Nombre: “Almacenar instancia incompleta”.

Descripción

El recolector envía una instancia incompleta, vía servicio web, al repositorio para su almacenamiento. Se procederá al análisis sintáctico de la misma y posterior despliegue en la base de datos, teniendo en cuenta que está incompleta.

Actores

Recolector.

Precondiciones

Se recibe una instancia desde la aplicación de recolección de datos.

Post-condiciones

1. Se almacena la instancia (desplegada) en la base de datos.
2. Se cambia el atributo “está completa” de la instancia a falso.

Frecuencia

Muy frecuente**.**

Flujo de escenario principal

1. Se recibe una instancia desde el recolector vía servicio web.
2. Se procede a un análisis sintáctico de la instancia recibida.
3. Si la instancia es correcta se despliega en la base de datos.
4. Se envía un mensaje de confirmación o de error al recolector.