UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA EN TELECOMUNICACIONES Y
LICENCIATURA EN ADMINISTRACIÓN Y DIRECCIÓN DE
EMPRESAS

Curso Académico 2016/2017

Trabajo Fin de Carrera

MY APP ANALYZER: fomentando el desarrollo móvil entre los jóvenes

Autor : Alexandra Ortega Martín

Tutor : Dr. Gregorio Robles

Proyecto Fin de Carrera

My App Analyzer: fomentando el desarrollo móvil entre los jóvenes

Autor : Alexandra Ortega Martín

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 2017, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2017

Dedicado a

mi familia / mi abuelo / mi abuela

II

Agradecimientos

Aquí vienen los agradecimientos... Aunque está bien acordarse de la pareja, no hay que olvidarse de dar las gracias a tu madre, que aunque a veces no lo parezca disfrutará tanto de tus logros como tú... Además, la pareja quizás no sea para siempre, pero tu madre sí.

Resumen

Aquí viene un resumen del proyecto. Ha de constar de tres o cuatro párrafos, donde se presente de manera clara y concisa de qué va el proyecto. Han de quedar respondidas las siguientes preguntas:

- ¿De qué va este proyecto? ¿Cuál es su objetivo principal?
- ¿Cómo se ha realizado? ¿Qué tecnologías están involucradas?
- ¿En qué contexto se ha realizado el proyecto? ¿Es un proyecto dentro de un marco general?

Lo mejor es escribir el resumen al final.

Summary

Here comes a translation of the “Resumen” into English. Please, double check it for correct grammar and spelling. As it is the translation of the “Resumen”, which is supposed to be written at the end, this as well should be filled out just before submitting.

Índice general

Índice de figuras

Capítulo 1

Introducción

COMENTAR TODA LA DOCUMENTACION QUE SAQUE AL PPIO SOBRE APPS En este capítulo se introduce el proyecto. Debería tener información general sobre el mismo, dando la información sobre el contexto en el que se ha desarrollado.

No te olvides de echarle un ojo a la página con los cinco errores de escritura más frecuentes[1].

1.1.

1.1.1. Estilo

Sobre el uso de las comas[2]

```
From gaurav at gold-solutions.co.uk  Fri Jan 14 14:51:11 2005
From: gaurav at gold-solutions.co.uk (gaurav_gold)
Date: Fri Jan 14 19:25:51 2005
Subject: [Mailman-Users] mailman issues
Message-ID: <003c01c4fa40$1d99b4c0$94592252@gaurav7klgnyif>

Dear Sir/Madam,
How can people reply to the mailing list?  How do i turn off
this feature? How can i also enable a feature where if someone
replies the newsletter the email gets deleted?
```

[1] http://www.tallerdeescritores.com/errores-de-escritura-frecuentes

[2] http://narrativabreve.com/2015/02/opiniones-de-un-corrector-de-estilo-11-recetas-par
html

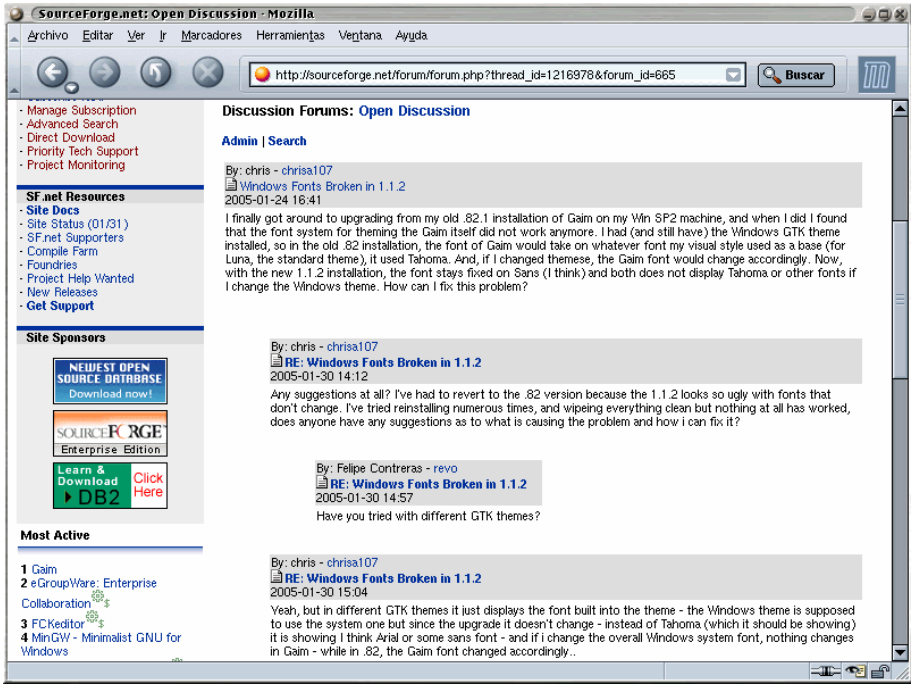

Figura 1.1: Página con enlaces a hilos

```
Thanks

From msapiro at value.net  Fri Jan 14 19:48:51 2005
From: msapiro at value.net (Mark Sapiro)
Date: Fri Jan 14 19:49:04 2005
Subject: [Mailman-Users] mailman issues
In-Reply-To: <003c01c4fa40$1d99b4c0$94592252@gaurav7klgnyif>
Message-ID: <PC17302005011410485105780lb04d55@msapiro>

gaurav_gold wrote:
>How can people reply to the mailing list?  How do i turn off
this feature? How can i also enable a feature where if someone
replies the newsletter the email gets deleted?

See the FAQ
>Mailman FAQ: http://www.python.org/cgi-bin/faqw-mm.py
article 3.11
```

1.2. Estructura de la memoria

En esta sección se debería introducir la esctura de la memoria. Así:

- En el primer capítulo se hace una intro al proyecto.
- En el capítulo 2 se muestran los objetivos del proyecto.
- A continuación se presenta el estado del arte.
- ...

Capítulo 2

Objetivos

2.1. Objetivo general

El objetivo general de este proyecto es crear una plataforma donde los nuevos programadores puedan evaluar su código de forma que no sólo puedan conocer sus puntos fuertes y débiles sino que además puedan obtener nuevos retos para mejorar sus habilidades computacionales.

Niños y jóvenes con interés por la programación conforman el público objetivo al que va destinada la aplicación. Por ello se han adaptado el diseño, funcionalidad y lenguaje para que resulten la experiencia de usuario sea lo más atractiva y lúdica posible.

En el sistema de clasificación hemos huido de las puntuaciones númericas, convirtiéndolas en tres niveles representados por los colores del semáforo: alto (verde), medio (amarillo) y bajo (rojo). De esta manera convertimos la experiencia en un juego, enmarcándola en un contexto más positivo para el usuario y alejándonos de las puntuaciones númericas utilizadas de los exámenes.

Toda la aplicación se relaciona con el usuario a través de un lenguaje coloquial, sencillo y en inglés. Se eligió este idioma porque es importante familizarizar a los niños con el mismo y reforzar sus conocimientos a través del juego.

2.2. Objetivos específicos

- Analizar las posibilidades de personalización y bloques computacionales que ofrece App Inventor como herramienta para crear programas.

- Definir las habilidades y capacidades del usuario a analizar. De los múltiples enfoques considerados y de cara a una simplicación de la información final se optó por crear tres grandes bloques de análisis: componentes, programación y usabilidad.

- Crear una aplicación Django donde se recoja la lógica anterior y poder ofrecer al usuario la información de manera clara y resumida. El usuario podrá además tener un registro de los proyectos guardados con anterioridad para poder revisar su clasificación.

- Unir las tecnologías Django, Bootstrap y Ajax para mejorar la capa de *Front End* y hacer la experiencia de usuario mucho más dinámica.

2.3. Planificación temporal

La planificación temporal del proyecto ha sido definida en función de los objetivos específicos marcados siguiendo un orden que permitiera avanzar en todos los aspectos de la aplicación:

- **Fase I**: búsqueda de documentación sobre App Inventor y creación de una cuenta en su web [1]donde poder analizar los bloques disponibles para el usuario, el contenido del archivo comprimido (*.aia*) en que se guardan los programas y cómo se representa la información dentro del mismo.

- **Fase II**:

  - Creación de una aplicación Django con una web simple donde subir el fichero con el proyecto, descomprimirlo y guardar su información en base de datos.
  - Añadir la posibilidad de tener una cuenta de usuario en la aplicación donde se almacenen los diferentes proyectos subidos de cara a futuras consultas.

- **Fase III**: definición de los puntos a analizar en cada proyecto subido e implementar la lógica de evaluación y clasificación.

- **Fase IV**: incorporar a la web las tecnologías Bootstrap y Ajax para añadir dinamismo y mejorar su aspecto.

[1] `http://appinventor.mit.edu/explore/`

Capítulo 3

Estado del arte

En este capítulo se explicarán brevemente las principales tecnologías utilizadas en el proyecto.

3.1. App Inventor

MIT App Inventor es un entorno de desarrollo web para la creación de aplicaciones móviles basadas en el sistema Android. Su principal objetivo es democratizar la programación y acercarla al público más joven transformándolo de consumidor de tecnología, a creador de la misma.

'Anyone Can Build Apps That Impact the World'

Figura 3.1: Web App Inventor

Gracias a su entorno gráfico sencillo e intuitivo, permite la construcción gratuita de programas reduciendo la curva de aprendizaje y eliminando barreras de entrada ya que para utilizarlo

sólo es necesario un ordenador con internet y un móvil con sistema operativo Android. Pero además de las ventajas desde el punto de vista tecnológico también hay que añadir el componente social. En su web nos permiten no sólo subir el código sino que además se pueden ver y descargar gratuitamente los programas de otros usuarios para usarlos o modificarlos. Desde su creación en 2015 es utilizado en colegios para mejorar la visión lógica de los estudiantes y crear comunidades educativas que conectan a jóvenes de cualquier parte del mundo.

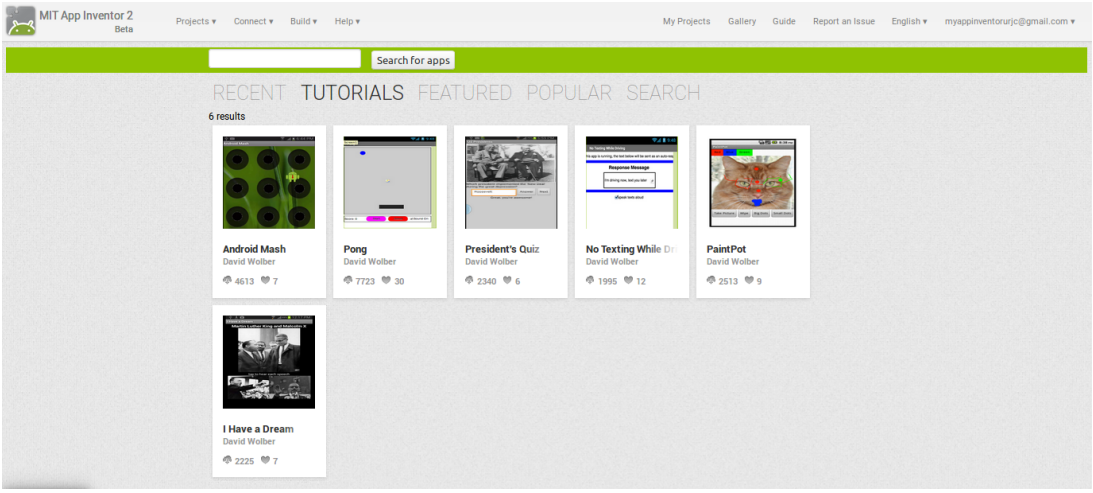

Figura 3.2: Web App Inventor

Para poder utilizarlo hay que tener una cuenta de Google y entrar en su web[1]. Tras iniciar sesión y seleccionar la aplicación, disponemos de dos herramientas: Designer y Blocks. Ambas funcionan con de la funcionalidad *drag & drop*, una vez decidido qué componente o bloque utilizar, lo arrastraremos hasta la previsualización de pantalla (*Viewer*) para situarlo en una zona determinada o conectarlo con otros elementos.

3.1.1. App Inventor Designer

En este apartado se construye la interfaz de usuario del programa, añadiendo y personalizando los componentes.

[1] `http://appinventor.mit.edu/explore/`

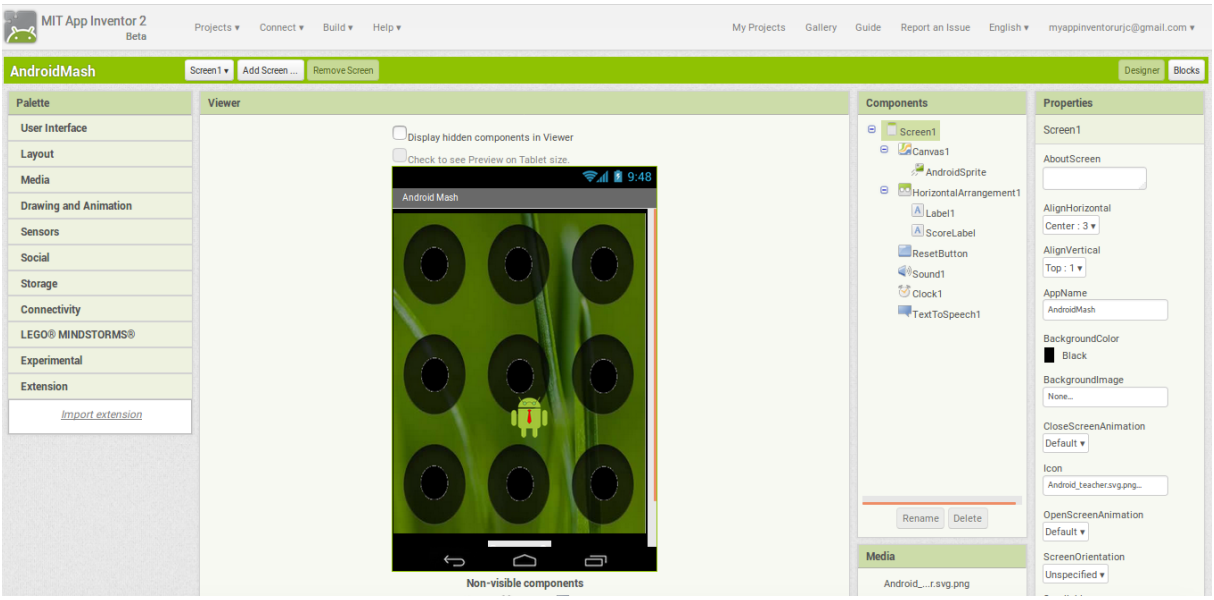

Figura 3.3: Web App Inventor

Incluye una amplia librería que nos permite escoger entre componentes de distinta complejidad y que serán objeto de análisis en nuestra aplicación como veremos en el siguiente capítulo:

- *User Interface*: botones, imágenes, casillas y todo tipo de elementos que forman parte de la interfaz de usuario y forman la estructura básica del programa.
- *Layout*: crea secciones horizontales o verticales en la pantalla.
- *Media*: cámara de fotos o video, reproductor de audio, grabadora...
- *Drawing and animation*: paneles de dibujo o delimitador de zonas para interactuar con el usuario a través de la pantalla táctil.
- *Sensors*: reloj, geolocalizador, podómetro, sensor de proximidad...
- *Social*: permite acceder a la lista de contactos, compartir ficheros o texto e incluso conectarse a Twitter.
- *Storage*: con gestores de ficheros y bases de datos.
- *Connectivity*: creación de servidor/cliente de Bluetooth y posibilidad de realizar peticiones HTTP.

- *Lego Mindstorms*[2]: interfaz de alto nivel para controlar los componentes de los robots Lego.

- *Experimental*: como alternativa al almacenamiento de datos incluido en *Storage*, se puede incluir Firebase[3], una base de datos creada por Google optimizada para dispositivos móviles.

- *Extension*: importador de extensiones de terceros.

3.1.2. App Inventor Blocks

Todo en App Inventor está causado por un evento: el click en un botón, la finalización de un reloj, el inicio de una conexión... Tras crear la máscara de la aplicación con los componentes, la dotaremos de logíca en la vista Blocks.

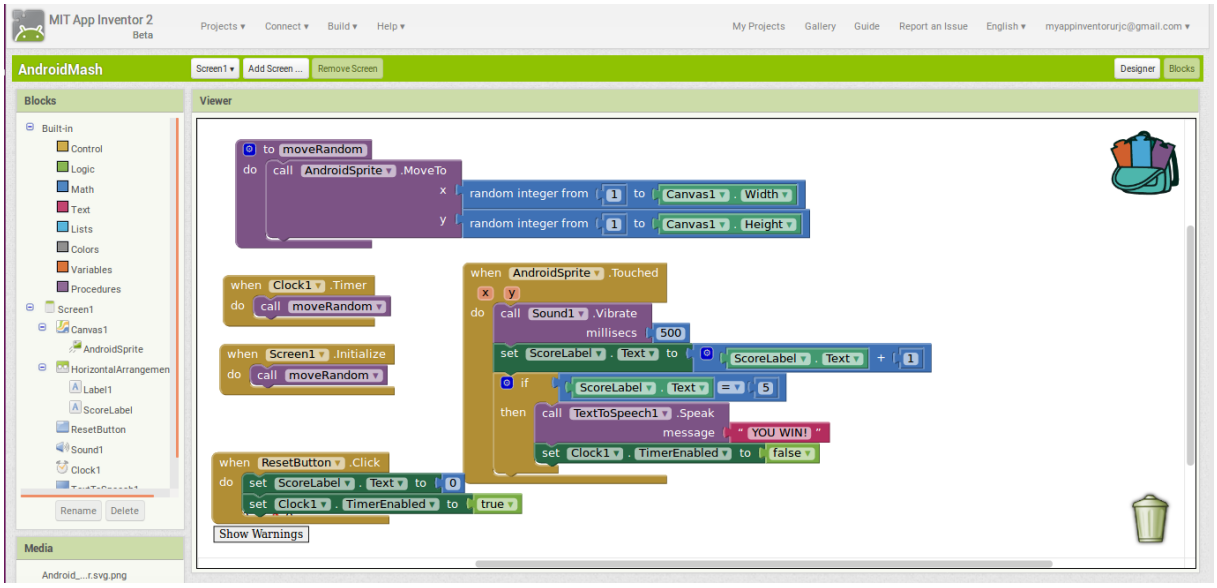

Figura 3.4: Web App Inventor

Por pantalla podremos insertar:

- *Built-in*: bloques lógicos predefinidos que podemos combinar entre sí:

  - *Control*: condicionales y bucles.

[2] https://www.lego.com/en-gb/mindstorms/
[3] https://firebase.google.com/

- *Logic*: comparadores, booleanos (*Strue/false*) y puertas lógicas (*and/or*)
- *Math*: operaciones matemáticas algebraicas y trigonométricas.
- *Text*: funciones para variables tipo texto: contiene, logitud, concatenación...
- *Lists*: creación, consulta y modificación de listas de variables.
- *Colors*: selector de color.
- *Variables*: creación, inicialización y modificación de variables locales y globales.
- *Procedures*: definición de procedimientos

- *Screen*: gestión de eventos sobre los componentes añadidos en cada pantalla.

- *Any Component*: gestión de eventos globales por componente. Por ejemplo, podemos definir una acción a ejecutar cuando se pulse un botón en cualquiera de las pantallas que componen el programa.

3.2. Python

Python es un potente lenguaje de progamación de alto nivel entre cuyas características principales destaca una: la sencillez. Gracias a su facilidad de uso y de lectura, cualquier programador puede entender su sintaxis y adaptarla a sus necesidades. Además es de código abierto y multiplataforma, es decir, puede redistribuirse gratuitamente en cualquier sistema operativo: Linux, Unix, Windows o MAC OS.

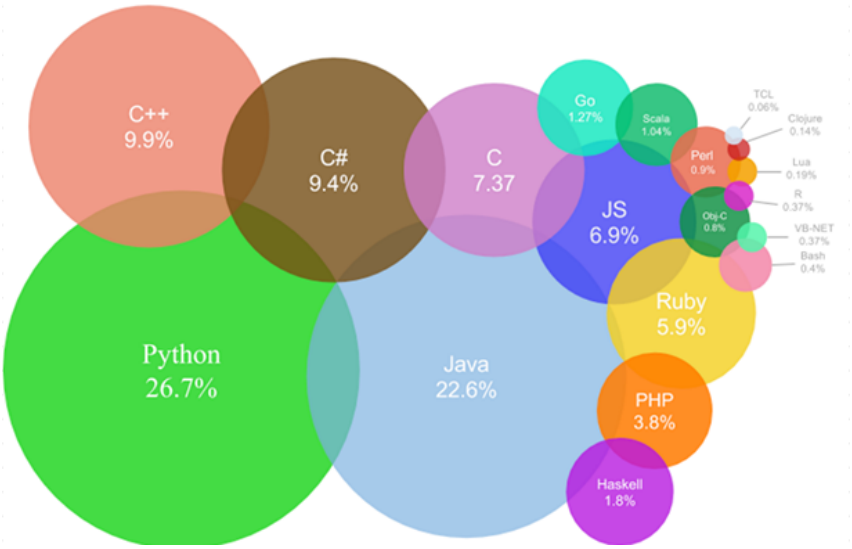

Figura 3.5: Lenguajes más importantes en 2016. Fuente: www.codeeval.com

Algunas características a destacar:

- Sintaxis sencilla: como hemos comentado, se busca una escritura simple, reduciendo caracteres redundantes y favoreciendo su interpretación rápida. Como ejemplo de ésto, la definición de los bloques por el nivel del sangrado (eliminando llaves o delimitadores de fin de instrucción) o la sustitución de los caracteres lógicos &&, || y ! por palabras como *and, or* y *not*.

- Incluye multitud de librerías optimizadas para reducir el código y el tiempo de ejecución.

- Dinámicamente tipado: no es necesario definir los tipos de datos previamente sino que podemos ir añadiendo variables y automáticamente Python les asignará un tipo en función de su contenido.

- Fuertemente tipado: no convierte automáticamente unos tipos de datos en otros.

- Incluye un modo interactivo donde poder ver el resultado de las instrucciones inmediatamente sin necesidad de compilar previamente el programa.

- Soporta programación orientada a objetos.

- Puede extenderse añadiendo módulos implementados en otros lenguajes como C o C++.

3.3. Django

Django es un framework de alto nivel escrito en Python para construir aplicaciones web de forma gratuita, sencilla y rápida. Incluye todas las librerías necesarias que pueden utilizarse en el desarrollo web, de manera que el programador sólo tiene que preocuparse en diseñar su aplicación sin necesidad de 'reinventar la rueda'.

Principales características:

- Rápido: las funciones están optimizadas para que su ejecución sea lo más rápida posible.
- Seguro: ayuda a los desarrolladores a evitar los errores más comunes en seguridad.
- Escalable: es flexible y fácilmente escalable, adaptándose a las necesidades de la aplicación.
- Código abierto: el uso de funciones predefinidas no excluye que el programador tenga acceso a su contenido para reutilizar o modificarlas. Todo el ćodigo es accesible para los usuarios.
- Versátil: gracias a Django podemos crear todo tipo de aplicaciones como servidores de contenidos (`www.openstack.org`) o redes sociales(`www.instagram.com`).

3.4. SQLite

SQLite es un sistema de gestión de bases de datos basado en el estándar SQL. Sus principales características son:

- Trasacciones atómicas, consistencia, aislamiento, y durabilidad ante fallos de sistema.
- No necesita configuración previa para poder utilizarlo.
- Multiplataforma: Android, BSD, iOS, Linux, Mac, Solaris, VxWorks y Windows
- No es una parte independiente del programa con el que se comunica sino que se integra con él, reduciendo la latencia de acceso. La base de datos se guarda como un fichero estándar dentro del sistema.
- De dominio público.

3.5. HHTP

HHTP son las siglas de Hypertext Transfer Protocol, un protocolo de comunicación web a nivel aplicación. Se utiliza en aplicaciones distribuidas y define los mensajes a intercambiar entre cliente y servidor dentro del protocolo pregunta/respuesta.

- No mantiene estado: no guarda información de conexiones anteriores. En caso de necesitar hacerlo se utilizan cookies, ficheros de información almacenados en el cliente por el servidor.
- Es colaborativo.
- Los mensajes enviados están escritos en texto plano y siguen la estructura:
  - Método de petición + Url del recurso + versión HTTP del cliente. Concretamente en nuestra aplicación usaremos los métodos GET para petición simple de recursos (solicitar una página web) y POST para envío de datos de cara a su procesamiento (envío de datos de formulario).
  - Cabecera con metadatos: idioma, tipo de servidor, fecha, cookie...
  - Cuerpo con datos a intercambiar.

3.6. HTML5

HyperText Markup Language o HTML es el lenguage estándar de la World Wide Web (WWW) en el que se define el formato para elaborar las páginas web. Está basado en ficheros de texto plano donde la interpretación del código recaerá sobre el navegador del cliente, que será el encargado de solicitar los recursos especificados y mostrar el contenido correctamente.

Los ficheros HTML utilizan etiquetas y atributos para identificar cada elemento del documento:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>App Inventor Analyzer</title>
  </head>
```

```
  <body>
    <p>Cuerpo de la web</p>
  </body>
</html>
```

Actualmente se encuentra en la versión 5 cuyas principales novedades permiten mayor dinamismo en las páginas web: etiquetas para 2D y 3D, generación de tablas dinámicas, nuevos tipos de datos en formularios o funcionalidad Drag & Drop.

3.7. CSS

Cascading Style Sheets o CSS es un lenguaje de diseño gráfico utilizado para dar estilo a ficheros HTML y XML. A través de la separación entre contenido y formato permiten eliminar líneas redundantes en los documentos, realizar modificaciones globales más fácilmente y adaptar la visualización para diferentes métodos de renderizado. Para utilizar CSS se necesita:

- Fichero .css: en el se definen las reglas mediante pares selector - bloque de declaración.

```
header .intro-text .intro-heading {
  font-family: "Montserrat", "Helvetica Neue", Helvetica, Arial, sans-
    serif;
  text-transform: uppercase;
  font-weight: 700;
  font-size: 50px;
  margin-bottom: 25px;}
```

- Fichero .html: donde se identifican los elementos a formatear con el selector especificado en el fichero .css.

```
<div class="intro-text">
        <div class="intro-lead-in">Welcome to MIT App Inventor
    Analyzer</div>
        <div class="intro-heading">It's Nice To Meet You</div>
        <a href="#services" class="page-scroll btn btn-xl">Tell Me
    More</a>
</div>
```

3.8. JavaScript

JavaScript es un lenguaje de programación interpretado utilizado principalmente en el lado del cliente para añadir dinamismo a las páginas web mostradas por el navegador. Al ser interpretado, no necesita ser compilado previamente ya que se interpreta a tiempo real, mejorando la eficiencia de las aplicaciones.

3.9. Bootstrap

Bootstrap es un framework de código abierto para el desarrollo de páginas y aplicaciones web creado por desarrolladores de Twitter. Este conjunto de herramientas une las tecnologías HTML, CSS, y Javascript para adaptar dinámicamente el formato de las webs al dispositivo utilizado (ordenador, tableta o móvil). La estructura básica de Bootstrap sigue el sistema de ficheros de la figura 3.6. En ella podemos ver cómo los ficheros CSS y JavaScript base vienen incluidos en su versión normal (*.css) y minificada (*.min.css). Esta última es una compresión en la que se eliminan caracteres innecesarios en el código como espacios o saltos de línea, y su objetivo es reducir el tiempo de procesado de los ficheros por el navegador.

```
bootstrap
├── css
│   ├── bootstrap.css
│   └── bootstrap.min.css
├── js
│   ├── bootstrap.js
│   └── bootstrap.min.js
└── img
    ├── glyphicons-halflings.png
    └── glyphicons-halflings-white.png
```

Figura 3.6: Directorio de archivos Bootstrap

HTML simple

```
<!DOCTYPE html>
<html>
<head>
<title>Bootstrap 101 Template</title>
</head>
<body>
<h1>Hello, world!</h1>
<script src="http://code.jquery.com/
    jquery.js"></script>
</body>
</html>
```

HTML con Bootstrap

```
<!DOCTYPE html>
<html>
<head>
<title>Bootstrap 101 Template</title>
<!-- Bootstrap -->
<link href="css/bootstrap.min.css"
    rel="stylesheet" media="screen">
</head>
<body>
<h1>Hello, world!</h1>
<script src="http://code.jquery.com/
    jquery.js"></script>
<script src="js/bootstrap.min.js"></
    script>
</body>
</html>
```

Figura 3.7: HTML simple VS. con Bootstrap

3.10. Ajax

Asynchronous JavaScript And XML o AJAX es una técnica de desarrollo asíncrona para actualización de páginas web bajo petición sin necesidad de recargarlas completamente, lo que mejora la eficiencia de las aplicaciones. Utilizando AJAX podemos utilizar varias tecnologías: HTML (datos), CSS (estilo), Document Object Model o DOM (actualización dinámica de contenido), HTMLHttpRequest (petición al servidor) y XML (formato de envío de información).

En nuestra aplicación My App Inventor podemos ver un ejemplo de esta tecnología en la plantilla *userprofile.html* donde actualizaremos el contenido de la página en función de la opción escogida por el usuario: ver lista de proyectos, guardar un nuevo proyecto o actualizar perfil.

```
<a class="page-scroll" href="#usercontent" onclick="loadUserProjects()">
        My proyects</a>
<a class="page-scroll" href="#usercontent" onclick="loadNewProject()"">
        Add New</a>
<a class="page-scroll" href="#usercontent" onclick="loadUpdateProfile()">
        Update profile</a>

...

<!-- Load UserProjects JavaScript -->
<script>
function loadUserProjects() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("usercontent").innerHTML =
      this.responseText;
    }
  };
  xhttp.open("GET", "/userprojects/", true);
  xhttp.send();
}
</script>
```

Capítulo 4

Diseño e implementación

En este capítulo se detallarán la arquitectura y el diseño implementados en My App Inventor, relacionando las diferentes tecnologías utilizadas y explicando el flujo interno del programa.

4.1. Arquitectura general

La arquitectura de la aplicación se basa en el modelo cliente-servidor descrito en la figura 4.1. Una vez que el usuario ha entrado en su cuenta dentro de la web, tendrá dos opciones: volver a analizar un proyecto existente o subir el fichero comprimido *.aia* que previamente ha descargado de la web de App Inventor[1] y analizarlo. Ambas peticiones serán recogidas por el manejador Django que internamente realizará las acciones necesarias para obtener una evaluación del programa. Los resultados finales serán renderizados por BootStrap y mostrados al usuario en su navegador web.

Como se puede observar en la siguiente figura tenemos dos componentes principales:

- Servidor: compuesto por una aplicación Django y una base de datos SQLite. Mientras que Django realizará la interacción con la capa de Front End y realizará el análisis de los datos, la base de datos nos ayudará a almacenar la información relativa a los usuarios y sus proyectos.

- Cliente: el usuario accede a la aplicación mediante un navegador web desde el que se realizan las peticiones GET o POST (Http Request) y mostrará los datos recibidos desde el

[1] `http://appinventor.mit.edu/explore/`

Servidor (HTTP Response). El desarrollo de esta parte del Front End se realiza a conjuntamente con Django y Bootstrap de manera que la web se adapta al dispositivo utilizado por el usuario (ordenador o móvil)

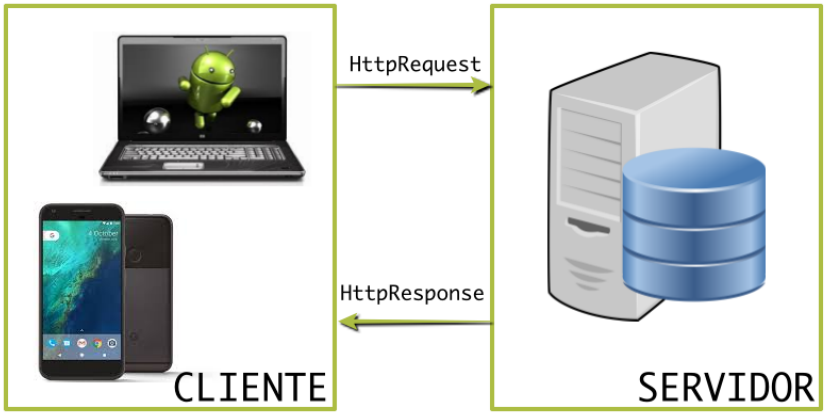

Figura 4.1: Arquitectura Cliente-Servidor

4.2. Diseño e implementación del servidor

Como hemos visto, la arquitectura del Servidor consta de dos grandes bloques: la lógica de Django y el almacenamiento de SQLite.

4.2.1. Lógica servidor: Django

La lógica del proyecto My App Inventor está organizada en los siguientes directorios:

- Projecto Django - **analyzeMyApp**: dentro nos encontraremos con los ficheros de configuración. En *settings.py* indicaremos los parámetros principales del programa (directorios, idioma, base de datos . . .), mientras que en *wsgi.py* especificaremos las reglas de comunicación entre un servidor web y nuestra aplicación cuando despleguemos en producción.

- Aplicación - **myAnalyzer**: en este directorio guardaremos la lógica de la aplicación, es decir, qué hacer cuando recibimos una petición, cómo será nuestro modelo de datos, qué pasos seguir al analizar un fichero recibido. . . . En las siguientes secciones veremos más en detalle este apartado.

- Ficheros estáticos - **static**: donde están las imágenes, estilos CSS y pequeños JavaScript para utilizar con Bootstrap.
- Plantillas - **templates**: incluye todas las plantillas HTML que formarán parte de la interacción del usuario con la aplicación.
- Proyectos guardados - **saved_projects**: además de guardar la información más relevante en base de datos, guardaremos una copia de seguridad de cada proyecto subido por el usuario en disco, así en caso de posibles ampliaciones de la lógica de evaluación, se podrá actualizar el resultado de forma transparente al usuario sin necesidad de que vuelva a cargar su programa de nuevo.

```
/
├── manage.py
├── analyzeMyApp
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── myAnalyzer
│   ├── __init__.py
│   ├── admin.py
│   ├── scoreMyAppMessages.py
│   ├── admin.py
│   ├── forms.py
│   ├── models.py
│   ├── urls.py
│   ├── apps.py
│   ├── scoreMyApp.py
│   ├── parser.py
│   ├── views.py
│   ├── errors.py
│   └── tests.py
├── static
├── templates
└── saved_projects
```

Figura 4.2: Directorio de archivos My App Inventor

En líneas generales, el flujo de la aplicación sigue los pasos de la Figura 4.3 y se define dentro del directorio **myAnalyzer**. Cuando un usuario registrado realiza una petición a la aplicación, se diferencia entre guardar un nuevo proyecto o cargar uno preexistente. Tras este paso,

se procede al análisis del programa y como paso final, se devuelve una respuesta con la clasificación obtenida. Durante todo el proceso, si se recibe una petición de un usuario no registrado, se redirige al mismo a la página de inicio.

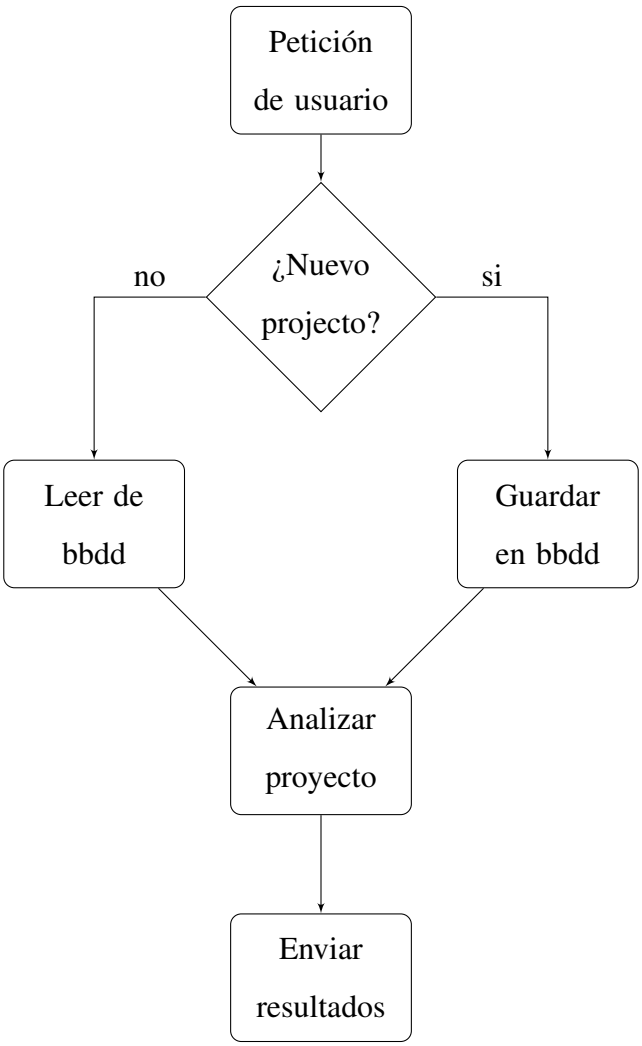

Figura 4.3: Flujo principal My App Inventor

En el momento en que la aplicación recibe una petición HTTP del usuario con una URL concreta, ésta se mapea en el fichero *urls.py*. Éste a su vez redirige la información a un procedimiento concreto de *wsgi.py* en función de la operación a realizar: iniciar o cerrar sesión, actualizar datos de perfil, cargar un nuevo programa, ver los anteriormente guardados . . . No todas las URLs son accesibles al usuario mediante la petición GET; por ejemplo, sólo podremos acceder a la vista que analiza los proyectos del usuario,*views.showUserAnalyzeProjectsPage*, mediante una petición de tipo POST insertada en un formulario de la aplicación para mante-

ner la integridad de los datos recibidos y controlar su contenido. Lo mismo ocurre con la vista *views.showDownloadPage* en la que se accede a la petición POST a través de un formulario.

```
urlpatterns = [
                url(r'^login/', views.showLoginPage),
                url(r'^logout/', views.showLogoutPage),
                url(r'^createprofile/', views.showCreateProfilePage),
                url(r'^userprofile/', views.showUserProfilePage),
                url(r'^download/', views.showDownloadPage),
                url(r'^updateprofile/', views.showUpdateProfilePage),
                url(r'^userprojects/', views.showUserProjectsPage),
                url(r'^analyze/', views.showUserAnalyzeProjectsPage),
                url(r'^$', views.showLoginPage),
                ]
```

Veamos en más detalle los diferentes procedimientos implicados en el proceso global que forman parte de la aplicación.

Comunicación

Como hemos comentado anteriormente, si un usuario no inicia sesión en la aplicación, se le devolverá siempre a la página inicial. Entre todas las librerías (o *views*) que incorpora Django, utilizaremos el Sistema de Autenticación [2] para complementar las vistas que nos ayudarán a realizar las operaciones más comunes como iniciar o cerrar sesión (*views.showLoginPage, views.showLogoutPage*), mantener la misma entre peticiones al servidor o comprobar si un usuario está conectado.

Todas las peticiones se realizarán a través de los objetos HttpResponse [3] de Django que manejan solicitudes GET y POST según corresponda. Por ejemplo, en la vista de inicio de sesión, se diferencia entre la solicitud de *login* (GET), donde respondemos al usuario con un formulario en el que introducir sus datos de registro, y la recepción de los mismos en el método POST para autenticarle.

Tras el inicio de sesión, el usuario será redirigido a la página principal donde podrá evaluar su código en función de si ya está subido a la aplicación o no, como vimos en la Figura 4.3.

[2] https://docs.djangoproject.com/en/1.11/topics/auth/default/

[3] https://docs.djangoproject.com/en/1.11/ref/request-response/

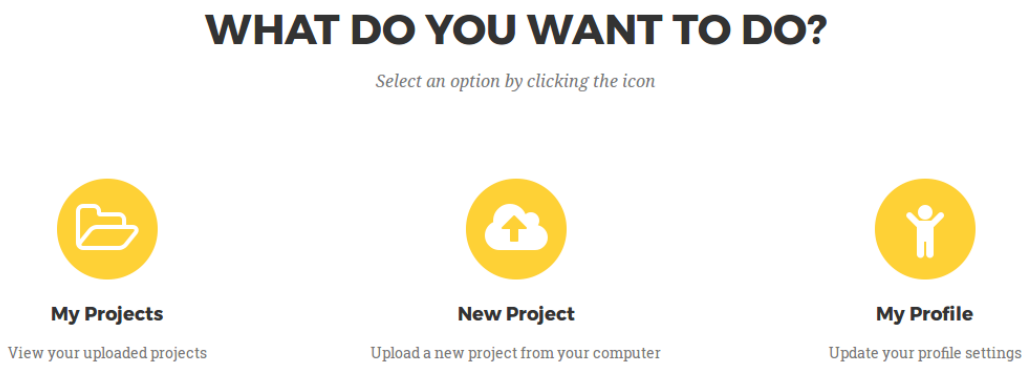

Figura 4.4: Opciones disponibles

Primero veamos el caso donde un programador quiere analizar su código por primera vez. La función *views.showDownloadPage* devuelve en la primera petición (GET) un formulario en el que el usuario podrá buscar en su disco el fichero comprimido *.aia* que previamente se ha descargado de su cuenta en App Inventor. Tras pulsar el botón Enviar, el navegador enviará una petición POST con el fichero al Servidor, donde se comprobará que no existe previamente para este usuario y descomprimirá, para posteriormente almacenar la información más relevante en base de datos.

La estructura en la que App Inventor guarda la información sigue el esquema descrito en la figura 4.5. En la carpeta *assets* se almacenan los ficheros estáticos de la aplicación, como las imágenes y sonidos. En *project.properties* se especifica la configuración del proyecto: versión del código, primera pantalla, tamaño.... Y finalmente en el último nivel del directorio *src* se encuentran la información relativa a los bloques utilizados (*.scm) y las lógicas que los relacionan (*.bky) por pantallas. Los datos más importantes para My App Inventor se encuentran en este último nivel y por ello serán los que guardemos en base de datos para su posterior análisis.

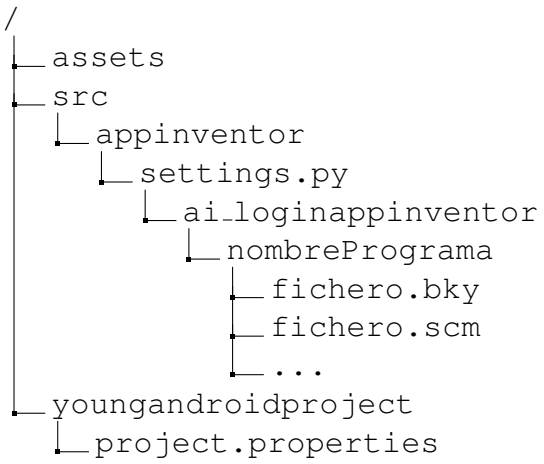

Figura 4.5: Directorio App Inventor

Si en un primer caso partíamos de un nuevo fichero de código, nuestra aplicación también permite al usuario guardar los proyectos guardados para poder volver a analizarlos en el futuro. Al llamar a la función *views.showUserProjectsPage* el Servidor listará todos los programas almacenados para el usuario y éste podrá volver a obtener su clasificación.

Tanto si partimos de un código nuevo como de uno ya existente, la última operación que hace My App Inventor es analizar el código. ¿Y por qué analizar algo que ya está puntuado? Pensando en futuras mejoras e implementaciones del algoritmo de análisis, de esta forma hacemos posible que el usuario siempre tenga su disposición la última versión del mismo con la clasificación más actualizada. Así no es necesario preprocesar todos los programas en base de datos en caso de actualización ya que ésta se realiza bajo demanda.

Análisis de código

El análisis de los programas del usuario se realiza a tres niveles: componentes, programación y usabilidad, cada uno con sus correspondientes subniveles. Como habíamos comentado en la sección anterior, en base de datos tendremos la información principal de las pantallas que forman cada proyecto. Para cada pantalla tenemos:

- Fichero **.scm**: contiene los bloques añadidos al proyecto y sus propiedades en formato JSON (JavaScript Object Notation)[4], un formato de representación de objetos con una estructura sencilla basada en relaciones nombre-valor. En función del bloque utilizado

[4] `http://www.json.org/`

tendremos unas propiedades u otras: posición (*AlignHorizontal*), fondo (*BackgroundColor*), tamaño (*Width*) ...


```
#|
$JSON
{"authURL":["ai2.appinventor.mit.edu"],"YaVersion":"159","Source":"Form","Properties":
{"$Name":"LogInScreen","$Type":"Form","$Version":"20","AlignHorizontal":"3","AlignVertical":"2","AppName":"Multifunctionalaplication"
[{"$Name":"HorizontalArrangement1","$Type":"HorizontalArrangement","$Version":"3","BackgroundColor":"&H00FFFFFF","Width":"-2","Uuid":
[{"$Name":"VerticalScrollArrangement1","$Type":"VerticalScrollArrangement","$Version":"1","AlignHorizontal":"3","BackgroundColor":"&H
[{"$Name":"LogInBox","$Type":"TextBox","$Version":"5","Width":"-2","Hint":"Username, e-mail or phone","Uuid":"1160039573"},
{"$Name":"LogInPasswordtBox","$Type":"PasswordTextBox","$Version":"3","Width":"-2","Hint":"Password","Uuid":"827897089"}]},
{"$Name":"VerticalArrangement1","$Type":"VerticalArrangement","$Version":"3","AlignVertical":"2","BackgroundColor":"&H00FFFFFF","Heigl
[{"$Name":"SignInBtn","$Type":"Button","$Version":"6","BackgroundColor":"&H00FFFFFF","Text":"Sign In","Uuid":"-756982208"}]}]},
{"$Name":"HorizontalArrangement2","$Type":"HorizontalArrangement","$Version":"3","BackgroundColor":"&H00FFFFFF","Width":"-2","Uuid":"
[{"$Name":"VerticalScrollArrangement2","$Type":"VerticalScrollArrangement","$Version":"1","BackgroundColor":"&H00FFFFFF","Width":"-2"
[{"$Name":"FullNameSU","$Type":"TextBox","$Version":"5","Width":"-2","Hint":"Full Name","Uuid":"1890435708"},
{"$Name":"EmailSU","$Type":"TextBox","$Version":"5","Width":"-2","Hint":"E-mail","Uuid":"-1325954197"},
{"$Name":"PhoneSU","$Type":"TextBox","$Version":"5","Width":"-2","Hint":"Phone number","NumbersOnly":"True","Uuid":"592322586"},
{"$Name":"UsernameSU","$Type":"TextBox","$Version":"5","Width":"-2","Hint":"Username","Uuid":"-604227488"},
{"$Name":"PasswordSU","$Type":"PasswordTextBox","$Version":"3","Width":"-2","Hint":"Password","Uuid":"-1827733330"},
{"$Name":"CPasswordSU","$Type":"PasswordTextBox","$Version":"3","Width":"-2","Hint":"Confirm password","Uuid":"-710498798"},
{"$Name":"AgeSU","$Type":"TextBox","$Version":"5","Width":"-2","Hint":"Age","NumbersOnly":"True","Uuid":"-595745446"},
{"$Name":"HorizontalArrangement4","$Type":"HorizontalArrangement","$Version":"3","AlignHorizontal":"3","AlignVertical":"2","Backgroun
[{"$Name":"HorizontalArrangement5","$Type":"HorizontalArrangement","$Version":"3","AlignHorizontal":"3","AlignVertical":"2","Backgrou
[{"$Name":"Label1","$Type":"Label","$Version":"4","Text":"Gender:","Uuid":"1447773381"}]},
{"$Name":"GenderSpinner","$Type":"Spinner","$Version":"1","ElementsFromString":"Male, Female","Uuid":"-592189594"}]}]},
{"$Name":"VerticalArrangement2","$Type":"VerticalArrangement","$Version":"3","AlignHorizontal":"3","AlignVertical":"2","BackgroundCol
[{"$Name":"SignUpBtn","$Type":"Button","$Version":"6","BackgroundColor":"&H00FFFFFF","Text":"Sign Up","Uuid":"-2024365814"}]}]},
{"$Name":"HorizontalArrangement3","$Type":"HorizontalArrangement","$Version":"3","AlignHorizontal":"2","BackgroundColor":"&H00FFFFFF"
[{"$Name":"SignBtn","$Type":"Button","$Version":"6","BackgroundColor":"&H00FFFFFF","Text":"Sign Up","Uuid":"1042598961"}]},
{"$Name":"TinyWebDB1","$Type":"TinyWebDB","$Version":"2","ServiceURL":"l","Uuid":"-378396360"},
{"$Name":"TinyDB1","$Type":"TinyDB","$Version":"1","Uuid":"708141961"},{"$Name":"Clock1","$Type":"Clock","$Version":"3","Uuid":"18109
{"$Name":"Notifier1","$Type":"Notifier","$Version":"4","Uuid":"-1240548683"}]}}
|#
```


Figura 4.6: Ejemplo fichero .scm

- Fichero **.bky**: en él se almacena en formato XML (Extensible Markup Language)[5] todas las relaciones entre bloques activos y su funcionamiento. Será por tanto el principal fichero del que extraeremos la información estructural y funcional a analizar.


```
<xml xmlns="http://www.w3.org/1999/xhtml">
  <block type="component_event" id="1" x="867" y="-418">
    <mutation component_type="Button" instance_name="SignUpBtn" event_name="Click"></mutation>
    <field name="COMPONENT_SELECTOR">SignUpBtn</field>
    <statement name="DO">
      <block type="controls_if" id="2" inline="false">
        <mutation else="1"></mutation>
        <value name="IF0">
          <block type="logic_negate" id="3" inline="false">
            <value name="BOOL">
              <block type="lists_is_in" id="4" inline="false">
                <value name="ITEM">
                  <block type="text_trim" id="5" inline="false">
                    <value name="TEXT">
                      <block type="component_set_get" id="6">
                        <mutation component_type="TextBox" set_or_get="get" property_name="Text" is_generic="false" instance_name="UsernameSU"></
mutation>
                        <field name="COMPONENT_SELECTOR">UsernameSU</field>
                        <field name="PROP">Text</field>
                      </block>
                    </value>
                  </block>
                </value>
                <value name="LIST">
                  <block type="lexical_variable_get" id="7">
                    <field name="VAR">global userByUsername</field>
                  </block>
                </value>
              </block>
            </value>
          </block>
        </value>
        <statement name="DO0">
          <block type="lists_add_items" id="8" inline="false">
```


Figura 4.7: Ejemplo fichero .bky

[5] https://www.w3.org/XML/

La función *views.showUserAnalyzeProjectsPage* será la encargada de recibir la identificación del proyecto a analizar y responder al usuario con su evaluación. Tras consultar el contenido de las pantallas en base de datos, llamará a la función *scoreMyApp.getScore* que analizará cada nivel individualmente. En este proceso, las funciones llamadas se han organizado en diferentes clases según su propósito:

- **scoreMyApp**: contiene todas las funciones relativas a analizar el XML y obtener las estadísticas. Su función principal es *scoreMyApp.getScore* y en esta sección nos centraremos en su análisis.

- **scoreMyAppMessages**: procesa las clasificaciones y genera mensajes personalizados para el usuario.

Dentro de *scoreMyApp.getScore* analizaremos cada nivel pantalla por pantalla, comparando los resultados en cada iteración de forma que la estructura final contenga toda la información del programa.

```
if componentLevels['Score'] > generalScore['ComponentLevels']['Score']:
# Update
generalScore['ComponentLevels']['Score'] = componentLevels['Score']

G1 = generalScore['ComponentLevels']['L1_components']
S1 = componentLevels['L1_components']
generalScore['ComponentLevels']['L1_components'] = returnUnion(G1,S1,0)
```

La clasificación se almacenará en un diccionario en el que guardaremos las estadísticas de cada nivel. Para obtener las puntuaciones individuales se obtiene una media con los niveles de las clasificaciones, mientras que la puntuación final es una media ponderada de los tres niveles a analizar. Los componentes utilizados y las habilidades de programación tienen un peso del 45 % cada uno en la media mientras que la usabilidad de la aplicación se lleva el 10 % restante al considerarse una característica a evaluar pero que no debe tener la misma importancia en la nota final pues es un concepto más de diseño que de habilidad en la programación.

Nivel	**Subniveles**	**Ponderación**
ComponentLevels	Score,L1_components,L2_components,L3_components	45 %
ProgrammingLevels	Score,Flow,Data,Variable,Generalization	45 %
ScreensLevels	Score,Screens	10 %

Figura 4.8: Estructura de la clasificación

Para la evaluación de los **componentes** se han clasificado en tres niveles todos los bloques disponibles en App Inventor en función de su nivel de complejidad, independientemente de su naturaleza. Por ejemplo, en el nivel Bajo nos encontramos con bloques tipo cuadro de texto y relojes, mientras que en el nivel Alto tenemos componentes de Lego Mindstorms o bases de datos experimentales como FirebaseDB.

Subnivel	Bloques
Bajo	**InterfazUsuario** [Button CheckBox DatePicker Image Label ListPicker ListView Notifier Slider Spinner TextBox TimePicker] **Diseño** [HorizontalArrangement HorizontalScrollArrangement TableArrangement VerticalArrangement VerticalScrollArrangement] **Media** [ImagePicker] **Dibujo** [Ball Canvas ImageSprite] **Sensores** [Clock]
Medio	**InterfazUsuario** [PasswordTextBox WebViewer] **Media** [Camcorder Camera Player Sound SoundRecorder SpeechRecognizer TextToSpeech VideoPlayer YandexTranslate] **Sensores** [AccelerometerSensor BarcodeScanner OrientationSensor Pedometer] **Social** [ContactPicker EmailPicker PhoneCall PhoneNumberPicker Sharing Texting Twitter] **Almacenamiento** [File TinyDB] **Conectividad** [ActivityStarter BluetoothClient]
Alto	**Sensores** [GyroscopeSensor LocationSensor NearField ProximitySensor] **Almacenamiento** [FusionTablesControl TinyWebDB] **Conectividad** [BluethoothServer Web] **Lego** [NctDrive NctColorSensor NxtLightSensor NxtSoundSensor NxtTouchSensor NxtUltrasonicSensor NxtDirectCommands Ev3Motors Ev3ColorSensor Ev3GyroSensor Ev3TouchSensor Ev3UltrasonicSensor Ev3Sound Ev3UI Ev3Commands] **Experimental** [FirebaseDB]

Figura 4.9: Componentes

Una vez definidos los tres niveles, con la función *scoreMyApp.componentLevels_Score* buscaremos todas las ocurrencias de cada tipo de bloque dentro de la pantalla, guardando los resultados en tres diccionarios:

```
L1_found = {'UserInterface': UserInterface_L1_found,
                'Layout': Layout_L1_found,
                'Media': Media_L1_found,
                'Drawing': Drawing_L1_found,
                'Sensors': Sensors_L1_found}

L2_found = {'UserInterface': UserInterface_L2_found,
        'Media': Media_L2_found,
        'Sensors': Sensors_L2_found,
```

```
        'Social': Social_L2_found,
        'Storage': Storage_L2_found,
        'Connectivity':Connectivity_L2_found}

L3_found = {'Sensors': Sensors_L3_found,
        'Storage': Storage_L3_found,
        'Connectivity':Connectivity_L3_found,
        'Lego':Lego_L3_found,
        'Experimental':Experimental_L3_found}
```

Tras contabilizar el número de componentes total en cada subnivel, se asignará la puntuación de este módulo en función de la nota máxima alcanzada y guardaremos tanto la puntuación como el análisis para poder utilizarlos más adelante.

```
if c3 > 0 : # High score for complex components
score = 3
elif c2 > 0: # Medium score for complicated components
score = 2
else: # Low score for simple components
score = 1

results = {'Score':score,
       'L1_components':L1_found,
       'L2_components': L2_found,
       'L3_components': L3_found}
```

En la **programación** se revisarán competencias del usuario en distintas áreas de caracter abstracto como el control de flujo o el uso de funciones. Se han intentado agrupar a alto nivel las posibles etiquetas para poder organizarlas por funcionalidad y dentro de la misma, en los mínimos grupos, de manera que para cada subnivel tendremos siempre dos grupos globales que nos ayudarán a determinar la puntuación.

Subnivel	**Identificadores**
Flow	component_event controls_
Data	lists component_set_get
Variable	global_declaration lexical_variable_get lexical_variable_set math_ text_ local_declaration_statement local_declaration_expression
Generalization	procedures_ is_generic

Figura 4.10: Programación

- **Control de flujo** *scoreMyApp.flow_control*: identifica las expresiones lógicas para ejecutar ciclos, estructuras condicionales o eventos que dependen del comportamiento de otro componente o variable.
  - Nivel alto: se incluyen expresiones lógicas y eventos dependientes.
  - Nivel medio: se incluyen expresiones lógicas o eventos dependientes.
  - Nivel bajo: no incluye ningún control de flujo.
- **Gestión de datos** *scoreMyApp.data_control*: detecta el uso de listas para organización datos y asignación o modificación de valores a variables y componentes.
  - Nivel alto: se incluyen listas y modificadores de variables.
  - Nivel medio: se incluyen listas o modificadores de variables.
  - Nivel bajo: no se utilizan etiquetas de gestión de datos.
- **Representación de variables** *scoreMyApp.variable_control*: se tendrá en cuenta si el usuario utiliza declaraciones de variables tanto globales como locales y, en función del tipo de programa, expresiones matemáticas (sumas, potencias, operaciones de trigonometría...) o expresiones relacionadas con textos (unión, convertir a mayúsculas, contiene...).
  - Nivel alto: se incluyen declaración de variables y funcionalidades de matemáticas o texto.
  - Nivel medio: se incluyen declaración de variables o funcionalidades de matemáticas o texto.

- Nivel bajo: no se utilizan métodos referenciados a variables.

- **Generalización** *scoreMyApp.generalization_control*: la optimización de código a través de funciones y la generalización de procesos para un mismo tipo de variable son características que todo buen programa debe tener ya que facilitan su lectura y permiten la reutilización procedimientos.

  - Nivel alto: se incluyen procedimientos y eventos genéricos para un mismo tipo de componente.
  - Nivel medio: se incluyen procedimientos (en esta ocasión se ha considerado que el uso de procedimientos es una funcionalidad básica e imprescindible)
  - Nivel bajo: no se utiliza ningún tipo de mecanismo de generalización.

Al igual que en el nivel anterior, se guardarán todas las estadísticas e identificadores obtenidos en un diccionario además de la nota media de todas las subcategorías:

```
results = {'Score':score,
        'Flow': flowCtrl,
        'Data': dataCtrl,
        'Variable': variableCtrl,
        'Generalization': generalizationCtrl}
```

Por último analizaremos la **usabilidad** extrapolándola al número de pantallas que forman la aplicación. Una de la características de las aplicaciones actuales es su simplicidad para el usuario. Incluir gran cantidad de pantallería y opciones reduce su usabilidad al convertirla en poco intuitiva. Si queremos mejorar la experiencia de usuario conviene que nuestro programa tenga un número suficientemente bajo de pantallas para que resulte fácil de utilizar pero no sea excesivamente sencillo para que no pierda funcionalidad. Tras estudiar las aplicaciones existentes en el mercado, en este apartado se han definido los siguientes niveles dentro de la función *scoreMyApp.nScreens_control*:

- Nivel alto: entre 3 y 10 pantallas distintas. Ejemplos: Twitter, Facebook, Instagram...
- Nivel medio: entre 1 y 3 pantallas. Ejemplos: Notas, Reloj, Calculadora
- Nivel bajo: una única pantalla.

Tras el análisis de los componentes, técnicas de programación y usabilidad para todas las pantallas que componen el programa a analizar, obtendremos el siguiente esquema de resultados recopilando la máxima información posible:

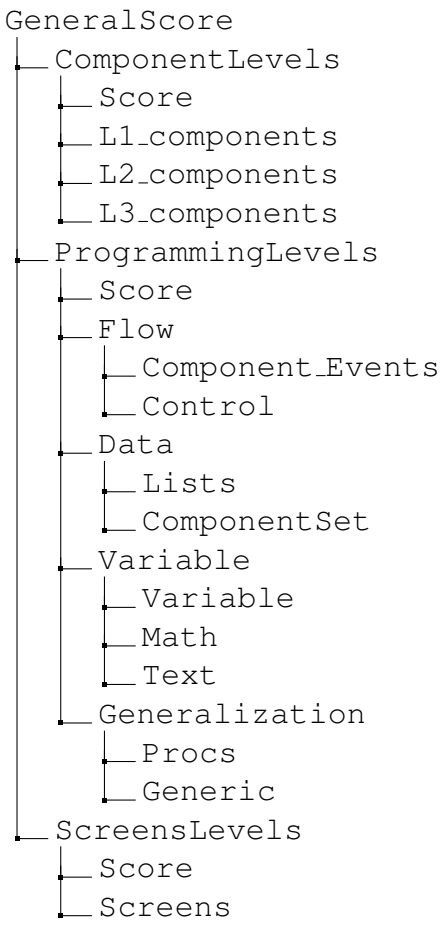

Figura 4.11: Resultados del análisis

Envío de resultados

Volviendo a la vista encargada de obtener los resultados finales, *views.showUserAnalyzeProjectsPage*, tras el análisis de los ficheros que componen el programa, llamaremos a las funciones de la clase *scoreMyAppMessages*. En ella procesaremos toda la información y la mostraremos al usuario de forma intuitiva y simplificada a través de imágenes y mensajes de texto cortos.

Gracias a la función *scoreMyAppMessages.getScoreMsgs* obtendremos todos los datos necesarios que queremos mostrar al usuario a partir de los resultados vistos en el apartado anterior:

- Nivel global - *general_score_msg*: a partir de la puntuación general tendremos un mensaje que resumirá el nivel promedio de la aplicación.

```
# General Score message
if score['Score'] == 3:
        general_score_msg = 'Great job! Your app has a HIGH score!
    Check out the different skills to improve it even more'
elif score['Score'] == 2:
        general_score_msg = 'Great job! Your app has a MEDIUM score!
    Check out the different skills to reach the next level'
else:
        general_score_msg = 'Ups! Your app has a LOW score. Check out
    the different skills to improve it'
```

- Nivel por categoría - *comp_score_msg, progr_score_msg, sched_score_msg*: componentes, programación y usabilidad mostrarán un mensaje personalizado en función de la puntuación media de sus subniveles.

- Desglose de resultados por categoría - *comp_score_info, progr_score_info, sched_score_info*: cada apartado incluirá la información más importante del mismo además de un consejo para mejorar el nivel actual. Dicho mensaje se mostrará siempre, pues aunque el usuario haya obtenido la máxima puntuación trataremos de motivarle para que incluya más complejidad a su programa o mayor diversidad de componentes.

    - Componentes: listará los bloques utilizados para conseguir este nivel.

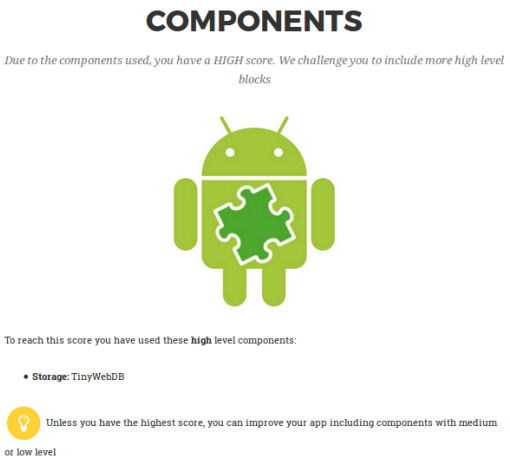

Figura 4.12: Resultados de la categoría Componentes

- Programación: resumirá por categorías todos los identificadores añadidos al código junto con el número de veces que se utilizan de forma que el usuario pueda tener una visión global la estructura de su programa.

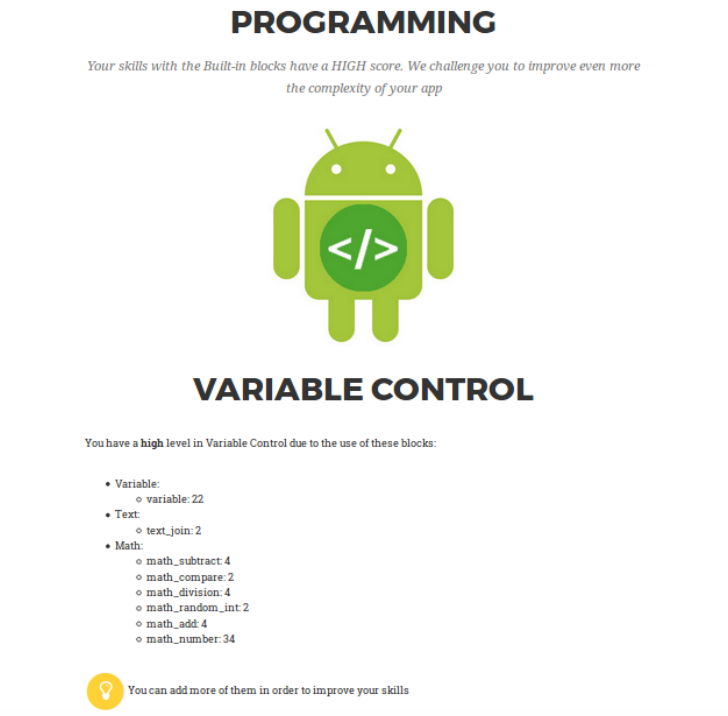

Figura 4.13: Resultados de la categoría Programación

- Usabilidad: muestra el número de pantallas que forman el programa.

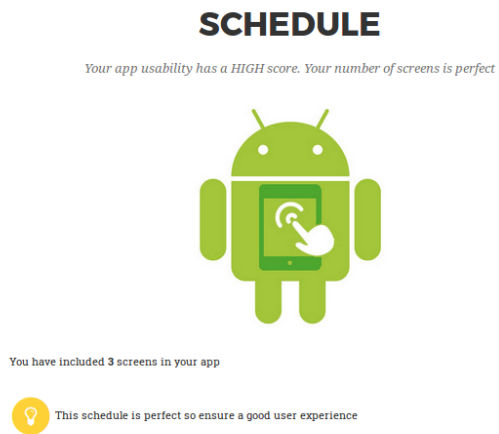

Figura 4.14: Resultados de la categoría Usabilidad

Una vez obtenidos todos los resultados, el siguiente pasó será enviarlos al navegador del

usuario a través de la página *analyzeAppCode.html*.En ésta se unen la mayoría de las tecnologías vistas en el capítulo Estado del Arte:

- HTML: como estándar para definir la estructura básica del archivo, legible e interpretable por el navegador. Identificado en la siguiente etiqueta dentro del propio fichero:

```
<!DOCTYPE html>
```

- CSS: antes de enviar el archivo HMTL, parsearemos tanto los resultados del análisis como el estilo para integrarlos con el resto de valores de la página. Todos los formatos estarán definidos en diferentes ficheros .css almacenados estáticamente en el Servidor.

```
{% load staticfiles %}

...

<link href="{% static 'css/agency.min.css' %}" rel="stylesheet">
```

- JavaScript: gracias a este lenguaje de programación podemos implementar funciones dentro del HTML para que nuestra página sea más dinámica. Se utiliza por ejemplo, a la hora de ocultar o mostrar el menú:

```
...

// Highlight the top nav as scrolling occurs
$('body').scrollspy({
        target: '.navbar-fixed-top',
        offset: 51 });

// Closes the Responsive Menu on Menu Item Click
$('.navbar-collapse ul li a').click(function(){
    $('.navbar-toggle:visible').click(); });
```

- Bootstrap: *framework* que aúna el resto de tencnologías (HTML, CSS, JavaScript) y nos permite que la página se adapte a dispositivos móviles como se muestra en la figura 4.5.

```
<!-- Bootstrap Core CSS -->
<link href="{% static 'vendor/bootstrap/css/bootstrap.min.css' %}"
 rel="stylesheet">
```

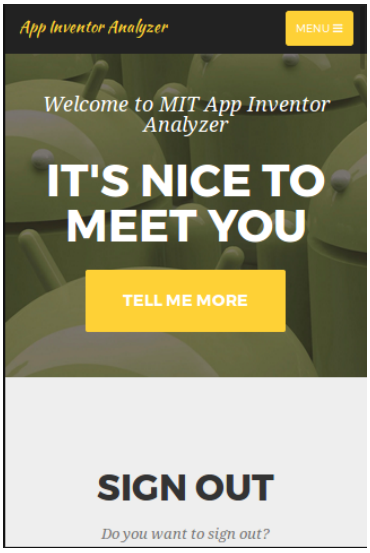

Figura 4.15: Versión móvil con Bootstrap

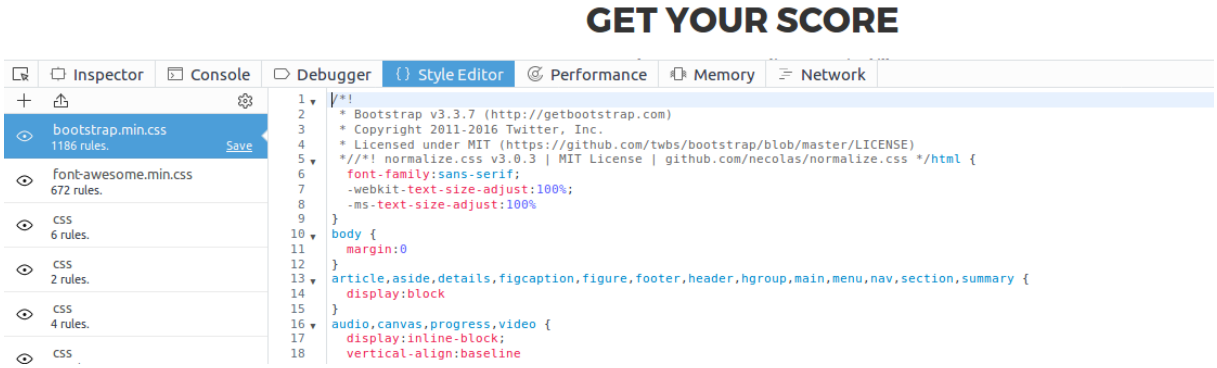

Figura 4.16: Petición de Bootstrap desde el navegador

4.2.2. Modelo de datos

En la organización y uso de la información se han utilizado dos modelos en función de las características de los datos a guardar:

- Contenido no persistente: variables locales creadas durante la ejecución de los procesos en el servidor.

- Contenido persistente: base de datos SQLite[6] y almacenamiento en memoria.

Como vimos en la sección anterior, los datos dinámicos donde se almacenan los resultados tras el análisis de código se guardan en una variable de tipo diccionario, donde se organizarán en pares clave-valor. En ella tendremos acceso tanto a las puntuaciones como a las prácticas de programación que las originan, ambas respresentadas en la figura 4.11.

Por otro lado tendremos el almacenamiento en memoria estática de los programas subidos por cada usuario. Para ello se ha creado un sistema de directorios donde ir descargando el código y así poder tener acceso a él en el futuro en caso de actualizaciones del sistema de análisis.

Los datos relativos a la información de usuario o los proyectos guardados por el mismo se guardarán en una base de datos, definida dentro de la aplicación en las clases de *models.py*. Para todo lo relativo a la creación y autenticación de usuarios, Django dispone de objetos predefinidos [7] que se han reutilizado añadiendo nuevas características. Para el caso del objeto User, que por defecto incluye como atributos primarios *username, password, email, first_name* y *last_name*, se ha usado de modelo base para la clase *UserProfile* extendiéndolo de forma que también incluya el campo *appinventorLogin*, necesario en la extracción de los ficheros comprimidos subidos por el usuario.

```
class UserProfile(models.Model):
    user = models.OneToOneField(User)

    # The additional attributes we wish to include.
    appinventorLogin = models.CharField(max_length=254)

    # Override the __unicode__() method to return out something meaningful!
    def __unicode__(self):
        return self.user.username
```

Para almacenar los usuarios existentes en la aplicación, sus proyectos y el contenido de los mismos se han creado las siguientes clases:

[6] https://www.sqlite.org

[7] https://docs.djangoproject.com/en/1.11/topics/auth

```
class Users(models.Model):
    userID = models.IntegerField(primary_key=True) # user_id
    projects = models.ManyToManyField('Projects') # The user can load and
    analyze multiple projects

class Projects(models.Model):
    projectName = models.TextField(primary_key=True) # user.id_filename.
    name
    screens = models.ManyToManyField('Screens') # The project can include
    multiple screens
    projectProperties = models.TextField() # Project properties

class Screens(models.Model):
    scrID = models.TextField(primary_key=True) # user.id_filename.
    name_Screen number
    bky = models.TextField() # Blockly info
    scm = models.TextField() # Screen Description
```

De esta forma un usuario puede tener varios proyectos que a su vez estarán compuestos por una o varias pantallas asociadas:

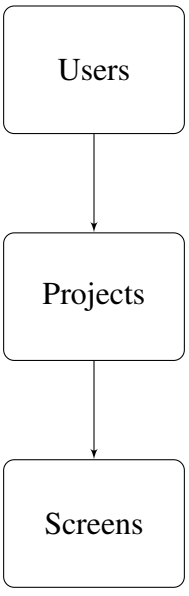

Figura 4.17: Modelo de datos

4.3. Manual de Usuario

Capítulo 5

Resultados

Capítulo 6

Conclusiones

6.1. Consecución de objetivos

Esta sección es la sección espejo de las dos primeras del capítulo de objetivos, donde se planteaba el objetivo general y se elaboraban los específicos.

Es aquí donde hay que debatir qué se ha conseguido y qué no. Cuando algo no se ha conseguido, se ha de justificar, en términos de qué problemas se han encontrado y qué medidas se han tomado para mitigar esos problemas.

6.2. Aplicación de lo aprendido

Aquí viene lo que has aprendido durante el Grado/Máster y que has aplicado en el TFG/TFM. Una buena idea es poner las asignaturas más relacionadas y comentar en un párrafo los conocimientos y habilidades puestos en práctica.

1. a
2. b

6.3. Lecciones aprendidas

Aquí viene lo que has aprendido en el Trabajo Fin de Grado/Máster.

1. a

2. b

6.4. Trabajos futuros

Ningún software se termina, así que aquí vienen ideas y funcionalidades que estaría bien tener implementadas en el futuro.

Es un apartado que sirve para dar ideas de cara a futuros TFGs/TFMs.

6.5. Valoración personal

Finalmente (y de manera opcional), hay gente que se anima a dar su punto de vista sobre el proyecto, lo que ha aprendido, lo que le gustaría haber aprendido, las tecnologías utilizadas y demás.

Apéndice A

Manual de usuario

Bibliografía

[1] App Inventor.
http://appinventor.mit.edu/explore/.

[2] Bootstrap.
http://getbootstrap.com/.

[3] Documentación Django.
https://www.djangoproject.com/.

[4] Documentación Python.
https://www.python.org/.

[5] Plantilla Bootstrap.
https://startbootstrap.com/.

[6] Sqlite.
http://www.sqlite.org/.

[7] W3schools.
https://www.w3schools.com/sql/.

[8] World Wide Web Consortium (W3C).
https://www.w3.org.

[9] M. S. Derek Walter. *Learning MIT App Inventor: A Hands-On Guide to Building Your Own Android Apps*. Addison-Wesley Professional, 2014.