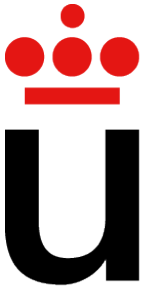


Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA AEROESPACIAL EN AERONAVEGACIÓN E INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Curso Académico 2020/2021

Trabajo Fin de Grado

DR. SNAP! UN ANALIZADOR DE LA EXPRESIVIDAD DE PROYECTOS SNAP!

Autor : Paula Rodríguez Martínez

Tutor : Dr. Gregorio Robles Martínez

Trabajo Fin de Grado

Dr. Snap!

Autor : Paula Rodríguez Martínez

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día de julio de 2021, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de julio de 2021

Dedicado a

mi familia

II

Agradecimientos

Agradezco a mi familia el apoyo que me ha dado durante los años de carrera y a mis amigos que he conocido en el doble grado ya que sin ellos no hubiese podido sacarme la carrera. En especial, a mi mejor amigo Miguel que ha sido el que me ha aguantado en los viajes a la universidad, a mi gran amiga Ángela que la he aguantado en sus nervios de los exámenes y a mi hermana Patricia con la que he pasado muchas noches programando mientras ella dibujaba. Por último, agradecer a los que han estado a lo largo de estos años y ya no están.

Resumen

En la actualidad, aprender programación aporta al ser humano habilidades como la resolución de problemas, la abstracción de ellos o la depuración de errores. En este mundo tecnológico se ha hecho imprescindible la docencia de asignaturas que introduzcan a los estudiantes en el mundo de los programadores. Snap! Berkeley es una plataforma que ofrece la posibilidad de crear programas de forma visual con bloques que realizan funciones o acciones sin necesidad de aprender un lenguaje de programación. De esta forma se puede introducir la programación y sus conceptos básicos desde edades tempranas haciendo que sea de una forma divertida, interactiva y didáctica.

Este trabajo de fin de grado aspira a crear una herramienta web que analiza los proyectos o programas de Snap! creados por los estudiantes basando el análisis en ocho conceptos de pensamiento computacional. Con ello se pretende que el alumno pueda ver sus puntos fuertes y sus carencias para mejorar y conseguir un nivel mayor de programación. Además, la herramienta se postula a ser una ayuda para el profesorado en la corrección de los proyectos de un aula entera.

La aplicación web se llama Dr. Snap! y surge de la idea del famoso analizador Dr. Scratch que analiza los proyectos de la plataforma Scratch. Para este trabajo se utilizarán tecnologías como el entorno Django, el lenguaje Python, las plantillas de HTML o el servidor que ofrece `https://www.pythonanywhere.com/`.

Summary

Nowdays, learning programming provides the human being with skills such as solving problems, abstracting them or debugging errors. Within the technological world, the teaching of subjects that introduce students to the world of programmers has become essential. Snap! Berkeley is a platform that offers the possibility of visually creating programs with blocks that can perform functions or actions without the need to learn a programming language. This way, programming and its basic concepts can be introduced from an early age, making it fun, interactive and didactic.

This end-of-degree project aims to create a web tool that analyzes Snap! projects or programs created by students, by basing the analysis on eight computational thinking concepts. This is intended for the student to notice their own strengths and weaknesses, the goal being the improvement and achievement of a higher level of programming. In addition, the tool is postulated to be an additional way of helping the teachers when correcting the projects from an entire classroom.

The web application is called Dr. Snap! It comes from an idea that was inspired by the famous Dr. Scratch analyzer, a web application that analyzes the projects of the Scratch platform. For this work, technologies such as the Django environment, the Python language, HTML templates or the server offered by `https://www.pythonanywhere.com/` will be used.

Índice general

Índice de figuras

Capítulo 1

Introducción

Con el avance de las tecnologías, la programación y el pensamiento computacional se han extendido de tal forma que ya se incluyen en el temario de la etapa académica educación secundaria obligatoria. Un programa no es más que un conjunto de sentencias que dan instrucciones a un ordenador y es necesario aprender un lenguaje de programación con el que escribirlas.

Este trabajo de fin de grado gira entorno a una plataforma de programación interactiva denominada Snap! Berkeley. En ella los usuarios crean programas a partir de bloques ya definidos o creando tus propios bloques. Además, ofrece la opción de subir todos tus proyectos y cuenta con un foro, con tutoriales y con un manual de referencia. Se creará una aplicación web en la que los usuarios puedan subir el proyecto creado en Snap! para ser analizado y poder obtener resultados basados en su nivel de programación.

La aplicación web se llamará Dr. Snap! y surge de la ya implementada herramienta web Dr. Scratch que analiza proyectos de la plataforma Scratch [14]. A raíz de Scratch se desarrolló Snap! Berkeley como extensión y mejora, por lo que Dr. Snap y Dr. Scratch giran ambos entorno a la misma idea y funcionalidad. Sin embargo, este proyecto pretende ser completamente independiente y con ideas nuevas y originales.

Con la intención de ser una herramienta con carácter académico, se desarrollará tanto para estudiantes como para profesores con el objetivo de que los estudiantes aprendan a programar de forma óptica, clara y concisa y a los profesores se les facilite el trabajo de corrección.

1.1. Conceptos básicos de programación

1.1.1. Programación en bloques

A raíz del desarrollo que se ha producido en la docencia de los fundamentos de programación y en las TIC, surge el concepto de ludificación. “La ludificación es un método con la finalidad de crear una experiencia significativa y motivadora a través de la integración de mecánicas de juego en entornos y aplicaciones no lúdicas” [12]. Es decir, debe crear un entorno atractivo que promueva la diversión combinada con el aprendizaje y la actividad cognitiva buscando siempre la motivación interna de los participantes [17]. En nuestro contexto, el sistema de ludificación es la plataforma de lenguaje visual previamente mencionada Snap! Berkeley en la que los participantes programan a través de bloques funcionales.

La programación en bloques consiste en crear programas encajando unas piezas predefinidas que permiten hacer funciones o procedimientos sin necesidad de un lenguaje de programación. Tiene similitudes al pseudocódigo y al diagrama de control. Es un tipo de enseñanza activa que promueve el pensamiento computacional, las diferentes formas de analizar y abordar los problemas y el diseño constructivo visual. En este ámbito, Scratch es una de las plataformas virtuales más conocidas y permite la implementación de juegos, animaciones e historias. En ella se pueden crear programas como se indica en la figura 1.1 donde se observan diferentes bloques con funciones distintas.

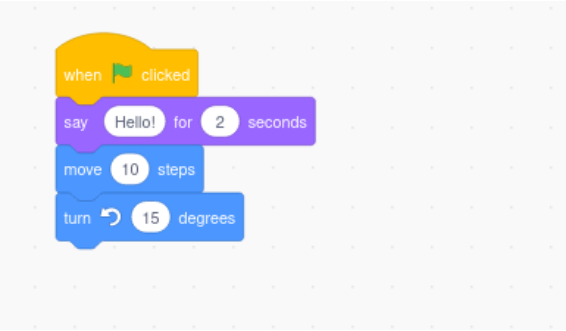


Figura 1.1: Ejemplo de programa con bloques (en este caso es un programa de la plataforma Scratch).

1.1.2. Pensamiento computacional

El pensamiento computacional es "el proceso de pensamiento que interviene en la formulación de los problemas y sus soluciones, de manera que las soluciones se representen de forma que pueda ser realizada por un procesador de información" [19]. Para poder evaluar sus competencias se han definido los siguientes puntos basados en la herramienta Dr. Scratch:

1) Paralelismo: sucesión simultánea de varias instrucciones o acciones. Por ejemplo, se necesitaría paralelismo si quisiésemos que dos personajes giren 15º a la vez.

2) Condicionales: cuando las instrucciones se pueden ejecutar o no en función del resultado de una condición. Entre los condicionales más típicos se incluyen los operadores lógicos (and, or, not).

3) Control de flujo: hace referencia al orden en el que se ejecutan las instrucciones. El orden natural en un programa sería que la instrucciones fuesen ejecutándose una a una. Sin embargo, también se debe incluir la opción de ejecutar un conjunto de instrucciones hasta que suceda otro evento o que las instrucciones se ejecuten las veces que queramos (hablamos de bucles).

4) Abstracción: se consigue dividiendo un problema grande en pequeños conjuntos más sencillos y simples separando en *scripts* o creando bloques definidos.

5) Sincronización: supone el ajuste temporal de diferentes eventos. Es decir, cuando se desea ejecutar una instrucción una vez ha terminado otra se requiere que ambos eventos estén sincronizados.

6) Interactividad con el usuario. Así, conseguimos hacer intervenir al usuario de alguna forma como puede ser introducir un nombre o pulsar una tecla.

7) Datos: esto incluye las operaciones con las variables, listas y atributos.

8) Diversidad: supone el uso de distintos tipos de bloques como de movimiento y de evento.

1.2. Snap! Berkeley

Snap! es una plataforma virtual que se programó en JavaScript como extensión de la ya mencionada plataforma Scratch [9]. Snap! define un lenguaje de programación visual que permite programar por bloques. Como novedades implementa listas avanzadas y permite crear bloques definidos más potentes. La página principal de Snap! se observa en la figura 1.2 y en ella podemos ver que nos ofrece la opción de correr el programa, de ver ejemplos y de obtener el manual de referencia. En Snap! Berkeley, además de crear programas, se pueden subir estos proyectos a la comunidad de Snap! para que todo el mundo pueda probarlos y ver cómo se han implementado.

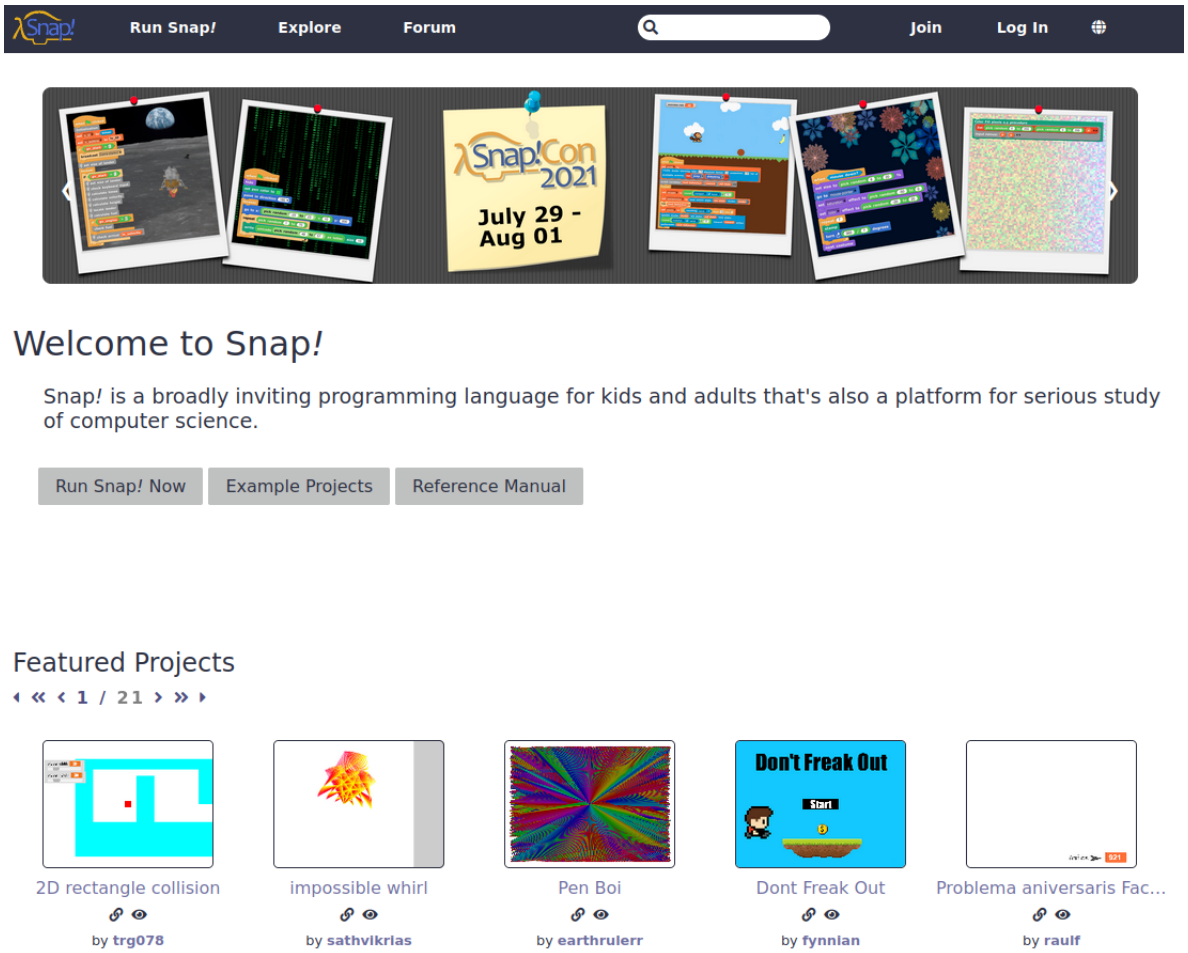


Figura 1.2: Página principal de Snap! Berkeley

El manual de referencia consta de once capítulos y de dos anexos y en ellos se explica en detalle todo lo relacionado con el manejo de sus bloques para programar [10]. Por ejemplo, el

capítulo I sección A trata sobre cómo crear paralelismo en Snap! gracias a la configuración de un tipo de bloques o el capítulo III te explica cómo crear tus propios bloques definidos.

El entorno de programación de Snap! donde se crean los programas se indica en la figura 1.3. En la sección indicada como A) se encuentran los bloques definidos de 8 categorías para crear los programas, la sección B) es donde se crean los conjuntos de instrucciones arrastrando los bloques y uniéndolos entre ellos. Por último, la sección C) es donde se pueden ver dichas instrucciones ejecutándose y siendo aplicadas a los personajes que definimos.

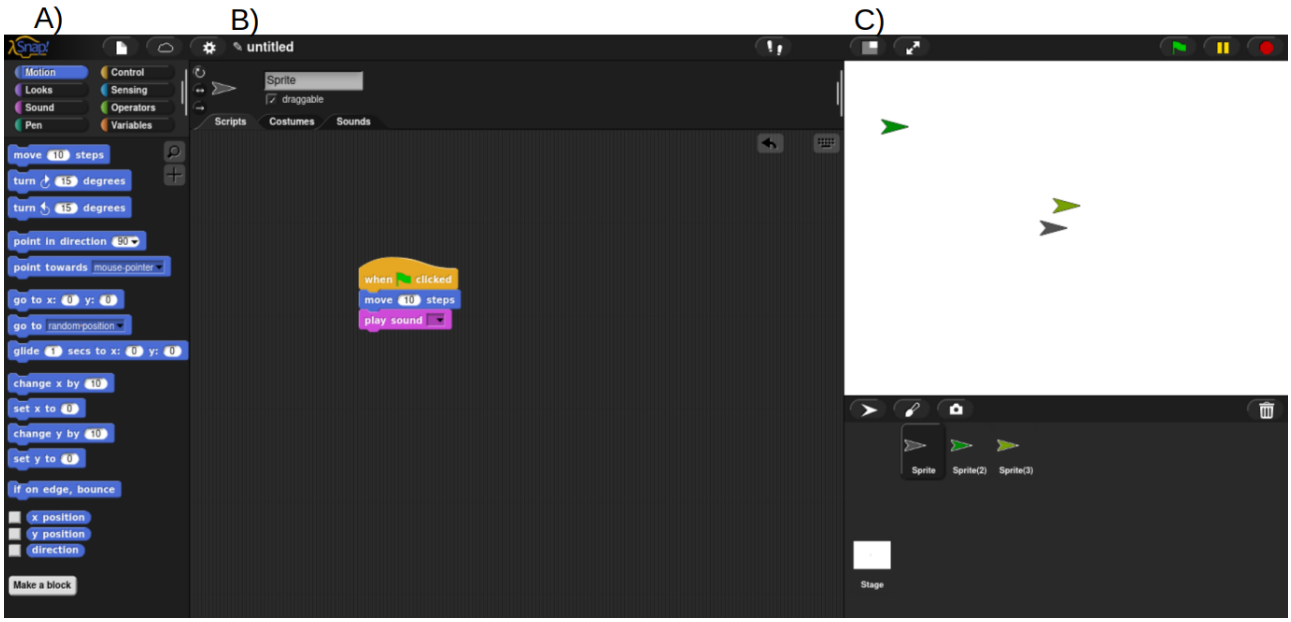


Figura 1.3: Entorno para programar en Snap!

Capítulo 2

Objetivos

2.1. Objetivo general

Mi trabajo fin de grado consiste en crear una herramienta docente que analiza los proyectos creados en Snap! Berkeley, una implementación que permite programar de forma interactiva a partir de bloques. Para ello se ha creado una aplicación web a través del *framework* Django y con el servidor PythonAnywhere[1].

2.2. Objetivos específicos

Para este proyecto hemos definido dos modos de interactuar:

- El modo estudiante consiste en analizar un fichero XML obtenido a través de la URL del proyecto de Snap! El análisis está basado según una rúbrica que lo divide en 8 secciones, admitiendo una calificación del 0 al 3 cada una.
    - Paralelismo
    - Condicionales
    - Control de flujo
    - Abstracción
    - Sincronización

[1] `https://www.pythonanywhere.com`

- Interactividad
- Representación de datos
- Uso de categorías o diversidad

El nivel final, que es el que da la calificación total del proyecto obtenida a partir de la media de todas sus partes, resulta ser:

- Nulo $\rightarrow$ [0, 0.5)
- Básico $\rightarrow$ [0.5, 1.5)
- Intermedio $\rightarrow$ [1.5, 2.5)
- Avanzado $\rightarrow$ [2.5, 3]

- El modo profesor amplía el modo estudiante con la posibilidad de analizar un archivo ZIP en el que se encuentran los XML de los proyectos de sus alumnos. Una vez analizados todos los archivos XML, se da la opción de descargar las puntuaciones de las 8 secciones y el nivel final en formato CSV.

Por último, para ambos modos se han definido varios *dashboards* en el que se muestran gráficos de los datos de los proyectos analizados.

2.3. Planificación temporal

Este trabajo de fin de grado ha requerido un tiempo estimado de siete meses dedicándole de media cuatro tardes/noches a la semana. Al tener que hacer las prácticas de la carrera, el trabajo de fin de grado de aeroespacial y continuar con mi trabajo de camarera en un restaurante se me hizo imposible ir más rápido. En la figura 2.1 se visualiza un diagrama del tiempo que aproximadamente he dedicado a los puntos más importantes del trabajo.

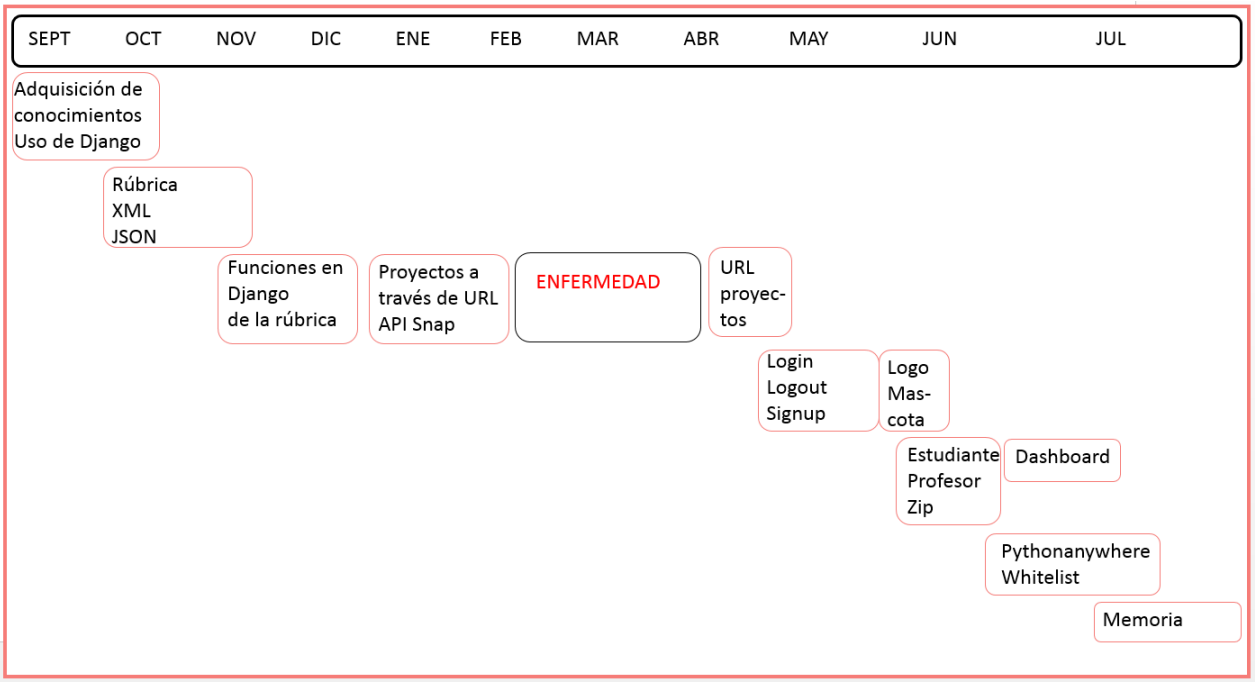


Figura 2.1: Planificación temporal del trabajo de fin de grado.

Capítulo 3

Estado del arte

En este apartado se explicarán de forma breve y concisa los lenguajes de programación, bases de datos, modelos, arquitecturas y demás tecnologías que se hayan necesitado para la implementación de nuestra aplicación web. Haremos una clara distinción entre los conceptos básicos que se han necesitado adquirir y las tecnologías usadas para que nuestra aplicación ofrezca las prestaciones prometidas y tenga un buen diseño web.

3.1. Snap! Berkeley

Como hemos mencionado previamente, la aplicación web será un analizador de proyectos creados desde Snap! Berkeley. Cabe destacar que el funcionamiento de Snap! como lenguaje resulta ser muy potente pudiendo crear programas complejos y elaborados. Para entender el funcionamiento de la programación en Snap! es necesario definir algunos conceptos básicos de la programación por bloques [10]:

- Sprite: Hace referencia a los diferentes gráficos o personajes de nuestro programa. Por ejemplo, en la figura 3.1 se puede observar que tenemos dos sprites (cuadrado y círculo) y a cada uno de ellos se les pueden asignar funciones de movimiento, sonidos, es decir, cada sprite tiene sus propios *scripts*. Esto es muy útil sobre todo para crear animaciones, historias o juegos interactivos con personajes. Los sprites tienen una apariencia que puede cambiarse gracias a los "customes" que se pueden pintar como los dos del ejemplo anterior o se pueden importar.

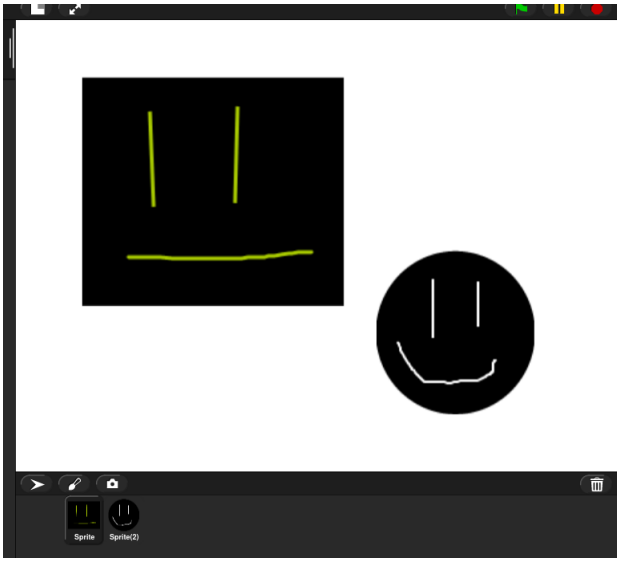


Figura 3.1: Dos sprites en SNAP!

- Script: es una pila o agrupación de bloques que se inicializan con los llamados *hat blocks* y realizan un conjunto de acciones como en la figura 3.2.

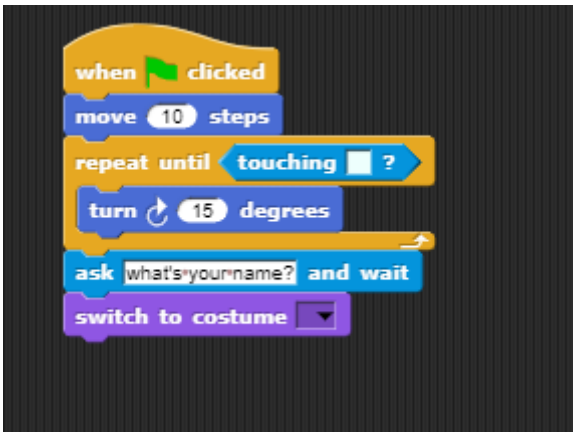


Figura 3.2: Script en Snap!

- Block: son los elementos básicos del lenguaje de Snap! Se deben arrastrar a la zona de *scripting* para encajarlos y crear scripts que hagan funciones. Existen diferentes clasificaciones:

  1) Según las categorías:

     a) Motion: bloques que permiten dar movimiento al sprite o personaje.

     b) Looks: permite cambiar las propiedades de apariencia del personaje.

c) Sound: permite añadir sonidos, cambiar el volumen, entre otros.

d) Pen: permite pintar líneas o crear estampas del sprite.

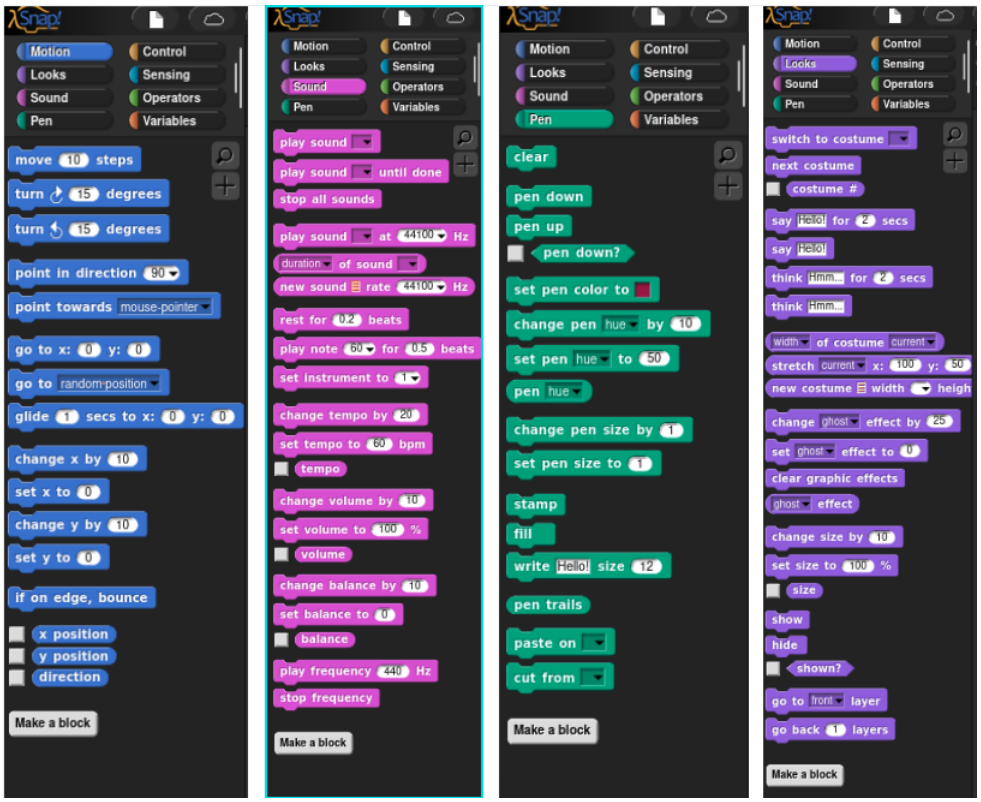


Figura 3.3: Bloques de tipo motion, look & sound y pen, respectivamente.

e) Control: bloques para inicializar los scripts o para hacer bucles.

f) Sensing: permite distinguir interacciones como cuando pulsas una tecla o cuando tocas un borde con el ratón.

g) Operators: permite hacer operaciones como sumas, multiplicaciones, operaciones lógicas, entre otras.

h) Variables: incluye listas, operaciones con listas, inicialización de variables, entre otras.

2) Según su funcionalidad:

a) Command block: se corresponden con bloques que tienen acciones ya definidas.

b) Reporter block: proporciona un valor de entrada que será usada en una acción.

Figura 3.4: Bloques de tipo motion, look & sound y pen, respectivamente.

c) Predicate: es otro tipo de “reporter block” ya que devuelve una entrada de tipo True o False. Tiene sentido usarlos en bucles o condicionales.

3) Según su forma:

a) Hat block: son los que se usan para inicializar un script y solo se pueden poner al principio.

b) Brick shape: de forma simple y de tipo command block.

c) C shape: es de tipo command pero admite dentro de él otro script o un único brick shape. Se corresponde con los bucles y los condicionales.

d) E shape: compuesto con dos bloques C shape.

e) Oval shape: se corresponde con los reporter block. Por ejemplo, las variables son óvalos de color naranja.

f) Hex shape: se corresponde con los predicate block.

En la figura 3.5 tenemos varios ejemplos.

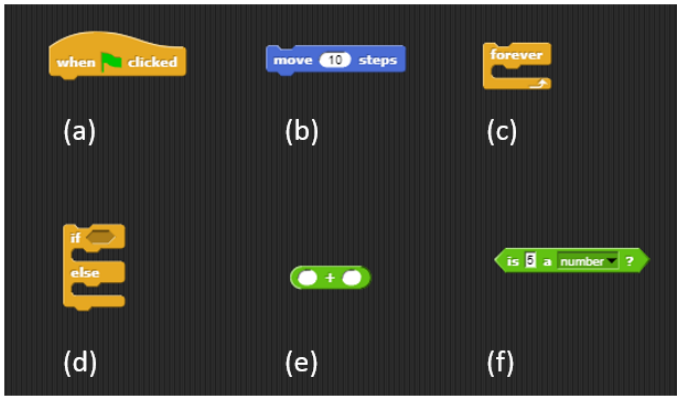


Figura 3.5: Bloques según la forma en Snap!

3.2. Modelo cliente-servidor

A día de hoy, la mayoría de las aplicaciones utilizan un esquema basado en el llamado modelo cliente-servidor. Es una arquitectura orientada a servicios donde los clientes son los usuarios de los servicios y de sus servidores asociados [6].

- Cliente: es la parte encargada de mandar peticiones al servidor y recibir las respuestas de este. Los usuarios interaccionan con la parte del cliente.
- Servidor: es la parte encargada de ofrecer un servicio, es decir, de recibir peticiones por parte de los clientes y de proveerles las respuestas adecuadas a dichas peticiones. [11]

Las ventajas de este modelo:

- Centralizado
- Mantenimiento simple
- Da servicios a más de un cliente
- Interoperabilidad con el uso de protocolos estándar

Las desventajas de este modelo:

- Los problemas de congestión son directamente proporcionales al número de clientes

- El mantenimiento simple
- Es escalable [11]

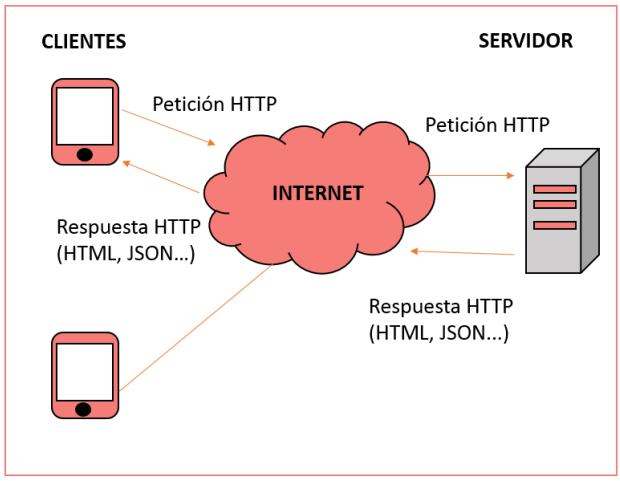


Figura 3.6: Modelo cliente servidor

Cabe mencionar que es un modelo de aplicación distribuida ya que las tareas que se realizan (solicitar recursos o enviarlos) se reparten entre cliente y servidor. Por tanto, distinguimos 3 partes fundamentales de una aplicación distribuida:

- Capa de presentación: también llamada interfaz de usuario, es responsable del control de la interacción entre el usuario y la aplicación. Se engloban todas las tareas que deben ser realizadas por la parte cliente del modelo general [13]. Por ejemplo, para el aspecto de la aplicación web se puede usar CSS, un lenguaje de estilos. Además,esta capa se corresponde en la actualidad con el concepto de *frontend* de una aplicación web.
- Capa de negocio: es la capa intermedia ya que se comunica con la capa de presentación y con la capa de datos [4]. Asegura las directrices o reglas del negocio, controla las secuencias de acciones y aísla los cambios de las interfaces de los usuarios y de los datos [13].
- Capa de datos: es la que tiene los datos y la que se encarga de acceder a ellos y operar con los mismos. Esta capa se corresponde con el concepto de *backend*. Las tecnologías que se incluyen en esta capa son tales como los gestores de bases de datos o los frameworks.

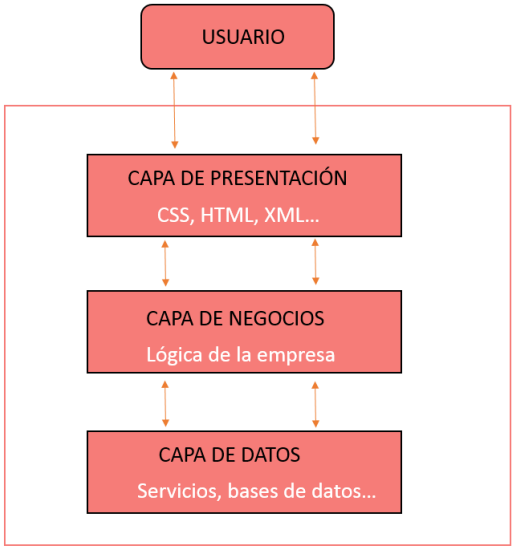


Figura 3.7: Modelo de 3 capas

3.3. Arquitectura MVC y Django

La arquitectura MVC divide las partes de una aplicación en modelo, vista y controlador de forma que se pueda implementar cada parte por separado. Al ser entidades independientes, el cambio de una de ellas no supondrá la necesidad de cambiar ninguna de las otras [7]. Django es el framework que usamos para esta aplicación y se basa en este patrón MVC:

- Modelo: se encarga de la representación de los datos. En Django se corresponde con el módulo models.py.

- Vista: se corresponde a la lógica de aplicación y se encarga de la presentación al usuario de los datos del modelo. En Django, las vistas se corresponden al módulo views.py y delegan la presentación de información en las plantillas o templates. Por eso, se puede decir que su arquitectura es MTV (modelo-template-vista) en vez de MVC como podemos observar en la figura 3.8.

- Controlador: recibe los eventos de entrada y contiene reglas de gestión de eventos como indicar cambios al modelo o elegir la vista adecuada para mostrar resultados.

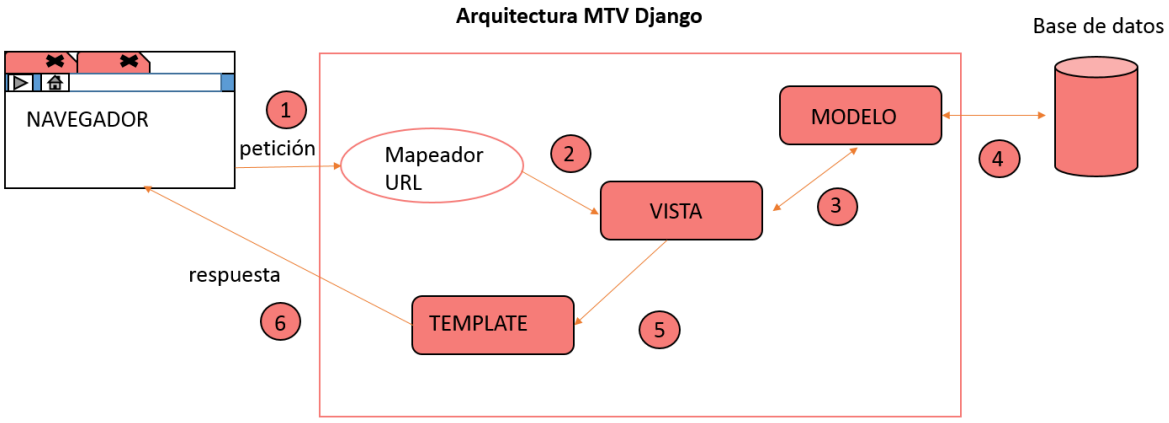


Figura 3.8: Arquitectura MVC en Django

3.4. Representational state transfer

REST es un modelo de diseño de servicio web que define un conjunto de principios de una arquitectura basada en recursos apto únicamente para aplicaciones distribuidas. Supone una mejora en la facilitación en el intercambio de datos entre el cliente y el servidor que, por ejemplo, el protocolo simple de acceso a objetos (SOAP). Sus principios son:

a) Modelo cliente (capa de presentación) - servidor (capa de negocio y datos) sin estado. No es necesario recordar ningún dato ya que las peticiones HTTP que se generan llevan incluidas todas las cabeceras y datos necesarios para dar una respuesta sin depender del estado anterior.

b) Estructura URI para los recursos.

c) Transferencia de hipermedios como XML o JSON para la representación de datos o de estado de los recursos.

d) Uso de métodos HTTP y principio CRUD (create, read, update and delete). Por tanto, las operaciones que afectan a un recurso pueden ser:

- GET: petición para obtener un recurso.
- POST: petición para crear un recurso en el servidor.
- PUT: petición para actualizar un recurso en el servidor.
- DELETE: petición para eliminar un recurso [16].

3.5. FRONTEND

Para la interfaz de usuario o la capa de presentación hemos utilizado en nuestro diseño web diferentes tecnologías como son HTML, CSS y JavaScript.

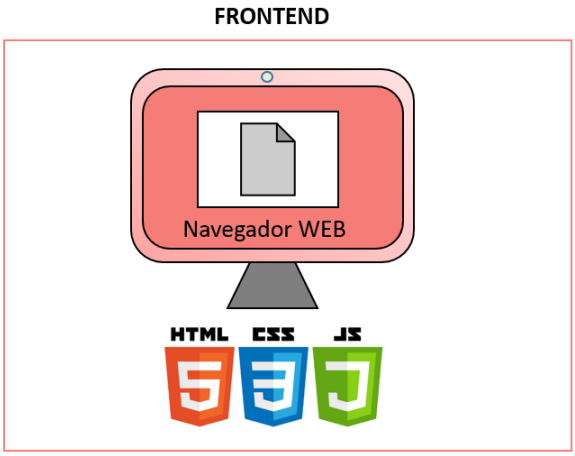


Figura 3.9: Elementos FRONTEND

3.5.1. HTML

HyperText Markup Language, en español lenguaje marcado de hipervínculos, es el lenguaje básico de construcción de páginas web. Se basa en texto que contiene referencias a enlaces, imágenes, tablas, textos u otros elementos. Un elemento HTML debe ir entre etiquetas que definen la orden o instrucción de lo que se quiere insertar [18]. Por ejemplo, para insertar un título se podría usar la etiqueta *h1* o para insertar una imagen *img*. HTML se encarga de crear el contenido de la página web pero no del estilo o apariencia de dicha página, para ello es necesario el CSS. Es decir, si nosotros queremos insertar un formulario para rellenar datos en nuestra página web va a ser necesario definir en HTML los elementos del formulario como son sus atributos (action y method), los campos que queremos que el usuario rellene con sus datos y un botón para darle a enviar. Ahora, para que el formulario tenga una apariencia en concreto tendremos que usar CSS para dar estilo a los elementos HTML del formulario.

<html>	Define un documento html
<head>	Define el encabezado
<body>	Define el cuerpo de la página html
<canvas>	Define contenedor para gráficos
<p>	Define un párrafo
<img>	Inserta una imagen

Figura 3.10: Etiquetas en HTML

3.5.2. CSS

Cascading Style Sheets, en español hojas de estilo en cascada, es el lenguaje de estilos que permite modificar la apariencia de páginas webs construidas en HTML. La sintaxis de una instrucción CSS es una declaración que tiene una propiedad y un valor [1]. La propiedad es la característica de estilo que se quiere cambiar, por ejemplo si queremos cambiar el fondo la propiedad sería “background”, y el valor sería el color o imagen del fondo que se querría cambiar. Se agrupan las declaraciones referidas a un elemento concreto de HTML, denominado selector, en un bloque. En la figura 3.11 se muestra un ejemplo de CSS dando estilo a un código HTML.

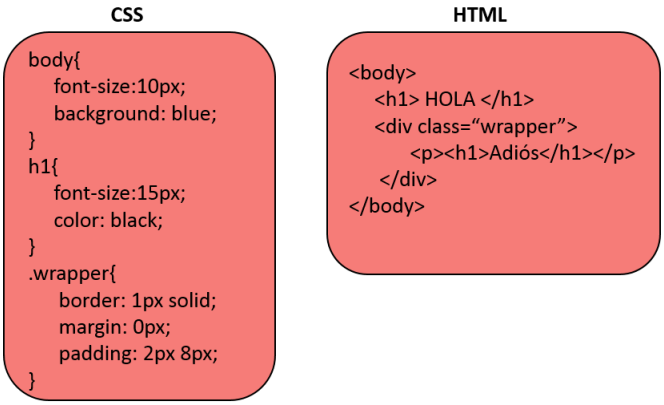


Figura 3.11: CSS asociado al HTML

3.5.3. JavaScript

Lenguaje de programación interpretado y orientado a objetos usado para crear contenido dinámico en las páginas webs. Para insertar código JavaScript en HTML se debe indicar con la etiqueta script. Ofrece la oportunidad de convertir una página web que en un principio era estática, en una página interactiva donde se puedan incluir menús emergentes, botones para hacer click o incluso animaciones [3]. Esto es posible gracias al modelo de programación basada en eventos utilizado por JavaScript en el que el script espera a que el usuario interactúe de alguna forma (evento), procesa la información y responde con una acción a través de una función. Por ejemplo, cuando el usuario pincha la pantalla el evento se denomina onClick. Cada evento tiene asociada una función que ejecuta JavaScript denominada "event handler" [15].

3.6. BACKEND

Se corresponde con la parte del servidor, la base de datos y la aplicación. Para ello utilizamos un lenguaje de programación (en nuestro caso Python), un framework para desarrollar el sitio web (Django), una base de datos (sqlite3) y un proveedor de servidor online donde lanzar la aplicación (pythonanywhere). En la figura 3.12 vemos distintas tecnologías que forman parte del *backend*.

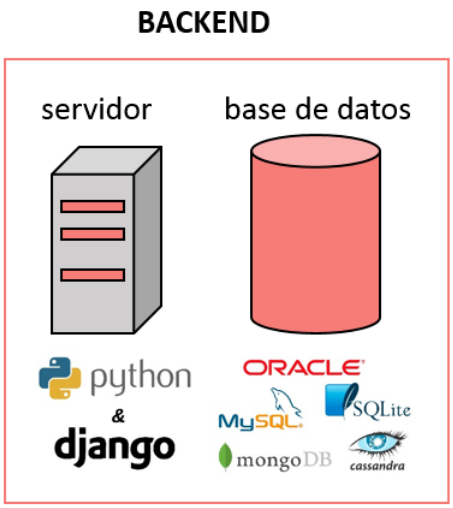


Figura 3.12: BACKEND

3.6.1. Python

Lenguaje de programación creado por Guido van Rossum en los 90, administrado por Python Software Foundation y con licencia de código abierto. En nuestra aplicación web usaremos la versión 3.8.5. Las características principales de este lenguaje son las siguientes [8]:

- Lenguaje de alto nivel e interpretado
- Tipado dinámico
- Fuertemente tipado
- Multiplataforma
- Programación orientada a objetos aunque permite también la programación imperativa, funcional y programación orientada a aspectos.

3.6.2. Django

Es un framework de alto nivel que permite crear una aplicación web usando Python de forma sencilla cuyas características más destacadas son [5]:

- Tiene un mapeador objeto relacional (ORM) que es una biblioteca que reconoce y relaciona cómo es la base de datos o el código que tú escribas.
- Conjunto de bibliotecas HTTP que gestionan las peticiones y respuestas.
- Biblioteca de ruta de URLs para localizar el código que corresponde a la URL introducida.
- Sistema de plantillas que conforman la interfaz del usuario.
- Basado en el patrón MVC con la variación de que las vistas delegan la presentación al sistema de plantillas.
- Por defecto la base de datos es sqlite3 pero se puede usar otras como MySQL, PostgreSQL u Oracle configurándolo en el módulo settings.py.

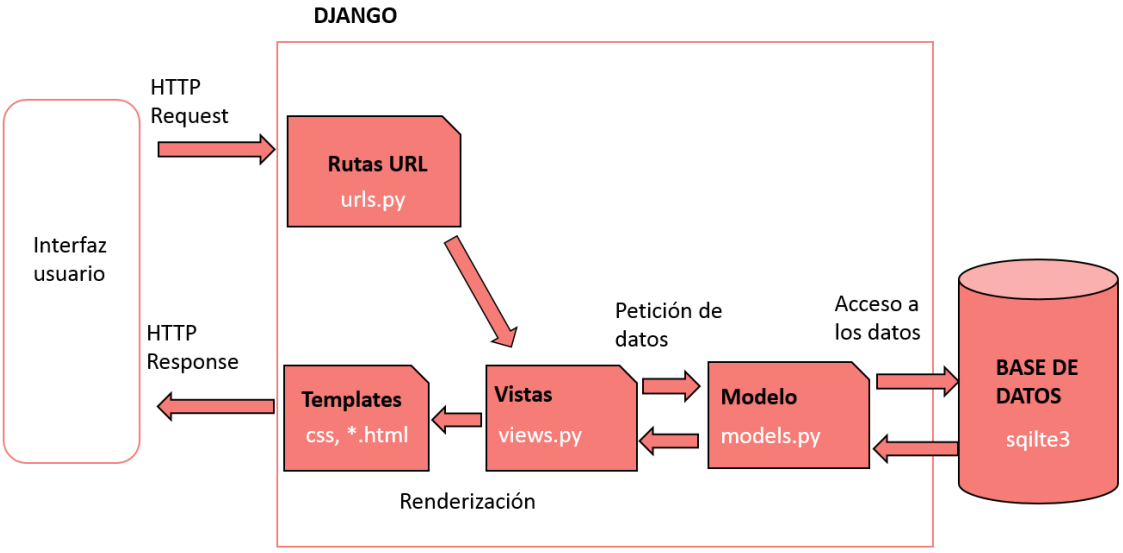


Figura 3.13: Aplicación en Django

3.6.3. SQLite3

SQLite es un motor de datos SQL mundialmente utilizado, este motor está desarrollado en una biblioteca programada en C la cual tiene compatibilidad con la mayoría de lenguajes de programación utilizados a día de hoy. SQLite es un motor SQL confiable dado que la mayoría del código fuente presente en su biblioteca está destinado a la realización de test unitarios y procesos de validación, cabe destacar en SQLite las siguientes características [2]:

- Autónomo y casi independiente: En el proceso de ejecución no existe un uso de más recursos que los proporcionados en el código fuente de SQLite, no realizando así un uso de bibliotecas u programas externos del sistema.

- Carencia de servidor: Para este motor el sistema cliente-servidor clásico es cosa del pasado, pudiendo permitir que el proceso que requiera de SQLite lea y escriba directamente desde los archivos de la base de datos en disco.

- Configuración Cero: A diferencia de aplicaciones u otras tipologías de bases de datos, SQLite opta por la configuración cero, en la cual no es necesario configurar nada ni definir ningún parámetro para indicar a nuestro sistema que SQLite está presente. En la biblioteca no podremos encontrar ficheros de configuración.

- Base de Datos transaccional: Esta dinámica aporta seguridad y estabilidad al proceso de la base de datos, en caso de una avería del sistema o de un corte de energía durante la transacción con un registro de la base de datos, SQLite detiene o confirma la transacción de forma automática, evitando de esta forma transacciones incompletas o fallos en el proceso.

3.6.4. PythonAnywhere

Proporciona un servidor para poder subir tu aplicación web a la nube. Es compatible con proyectos de Django y existen dos modalidades: de pago y gratuita. La de pago tiene más ventajas y opciones como es la posibilidad de elegir el dominio, al contrario que la versión gratuita que te impone el dominio `http://yourusername.pythonanywhere.com/`. En nuestro caso, la aplicación estará alojada en `http://prodriguezmartinez.pythonanywhere.com/`.

Capítulo 4

Diseño e implementación

En este capítulo detallaremos cómo hemos trabajado con las tecnologías explicadas en el capítulo 3 para poder crear la aplicación Dr. Snap!

4.1. Arquitectura general

La arquitectura de la aplicación creada la podemos dividir como se muestra en la figura 4.1.

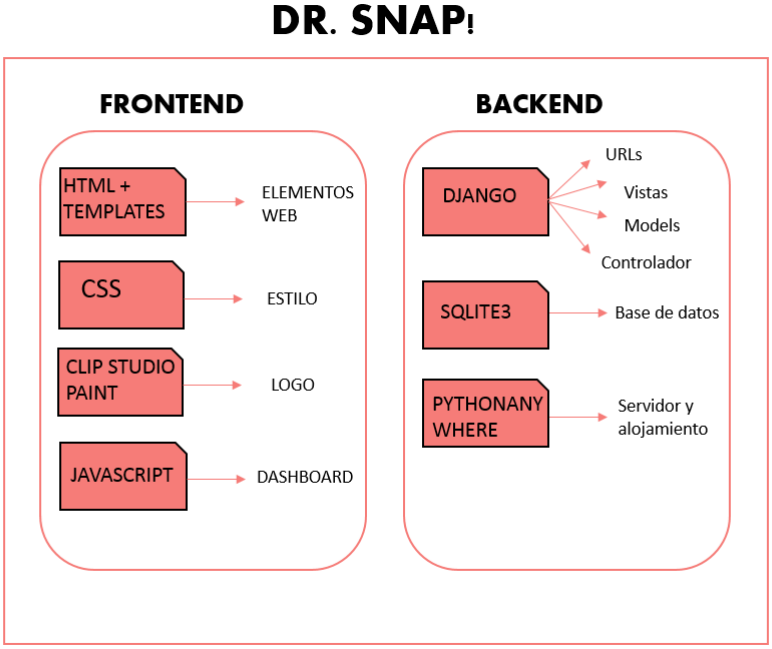


Figura 4.1: Esquema global de la aplicación.

4.1.1. FRONTEND

En este apartado se desarrolla la capa de presentación o interfaz del usuario. Para el aspecto de la aplicación web hemos añadido un estilo CSS asociado a unas plantillas HTML (templates) y hemos diseñado un logo con el programa Clip Studio Paint.

a) **HTML Y TEMPLATES**:

Las plantillas definen los elementos HTML de la página web en conjunto con el CSS que se encarga de dar el estilo y apariencia de los elementos de las plantillas. En este proyecto se ha usado una plantilla de referencia gratuita que se puede encontrar en el enlace siguiente. Se han hecho los cambios pertinentes para quedar de la forma que aparece en la figura 4.2 donde se define la cabecera, el menú horizontal, el logo y el pie de página.

Figura 4.2: Página índice de Dr. Snap!

A raíz de esta template “índice” se han extendido las demás gracias a las funcionalidades de Django“built-in template” y “template inheritance” que permiten dejar bloques sin definir los cuales las plantillas “hijo” que se creen serán las encargadas de rellenarlos. Estas heredan los elementos de la de referencia como es la cabecera o el menú principal. De esta forma, se evita repetir código. En la figura 4.3 se muestra el funcionamiento de la herencia de plantillas de Django en Dr. Snap! donde index.html es la plantilla de referencia y *.html, las plantillas hijo que dan contenido al bloque cuerpo heredando todos los demás elementos de index.html.

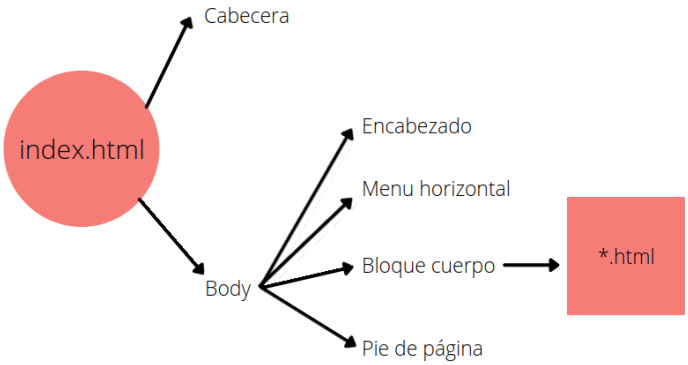


Figura 4.3: Templates inherit en Django

Así resulta un directorio de plantillas para nuestra aplicación web como en la figura 4.4 donde cada una de ellas se cargará con el contenido adecuado según las peticiones que reciba el servidor de Django o las respuestas que proporcione. Esto se consigue gracias a la función de renderizado que nos provee el framework Django y que se explicará más adelante en la parte de *backend*.

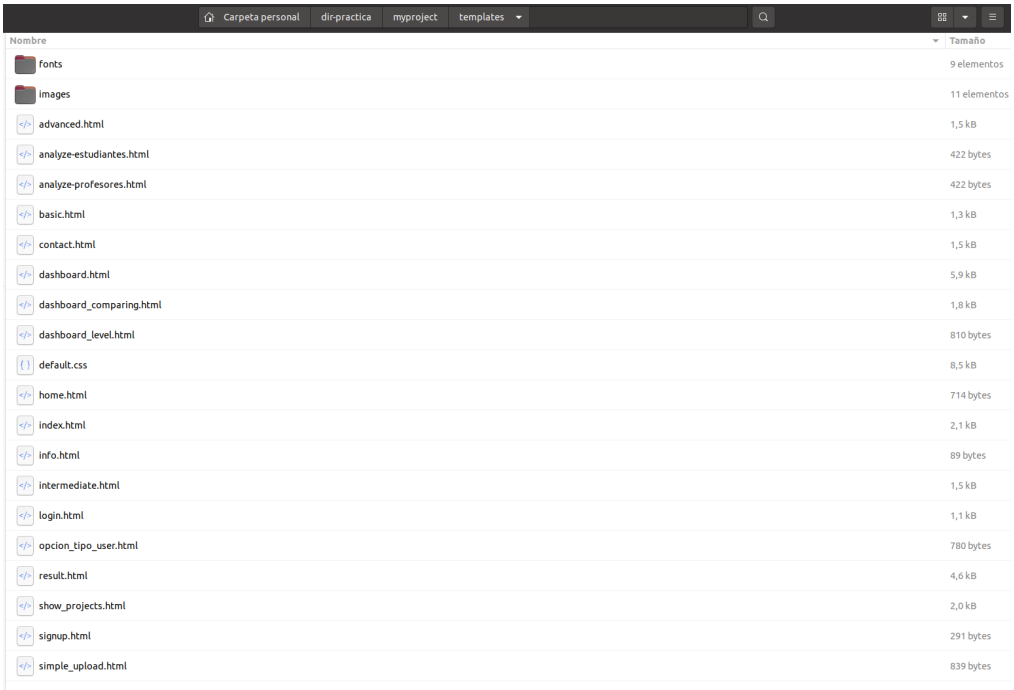


Figura 4.4: Directorio de templates de Dr. Snap!

b) **CSS**:

Como ya explicamos, el CSS es un lenguaje de estilo que proporciona las propiedades de apariencia de los elementos definidos en las plantillas HTML. Por ejemplo, en la figura 4.5 observamos como se vería la página sin CSS y con CSS.

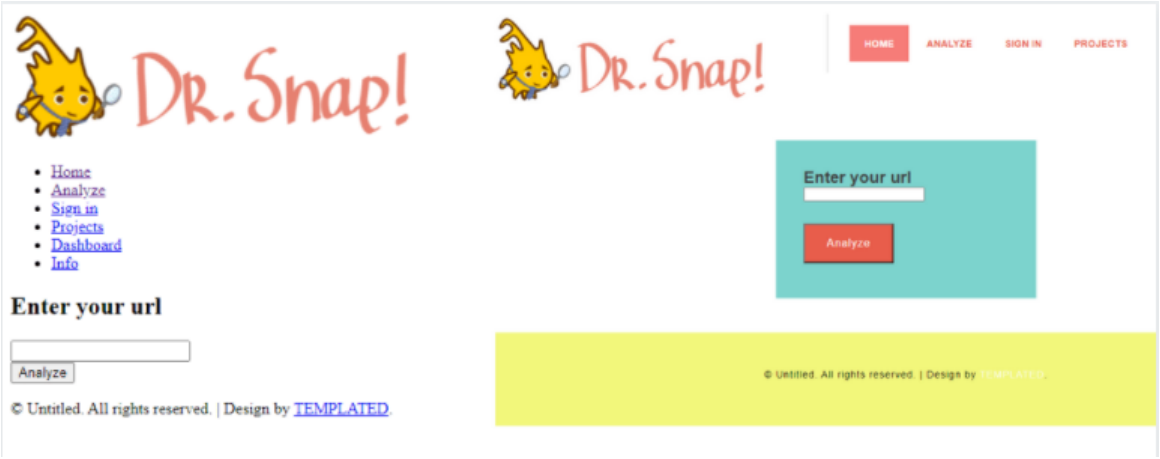


Figura 4.5: Página web sin CSS y con CSS

Así, para dar estilo al botón de analyze del formulario lo definiríamos en CSS como aparece en la figura 4.6, donde se hace referencia al elemento button (etiqueta en HTML) y se escriben las propiedades que queremos que tenga como es el color, el tamaño o la letra.

```
.button {
    display: inline-block;
    margin-top: 2em;
    padding: 0.8em 2em;
    background: #E85D48;
    line-height: 1.8em;
    letter-spacing: 1px;
    text-decoration: none;
    font-size: 1em;
    color: #FFF;
}
```

Figura 4.6: Código CSS de un botón

c) **CLIP STUDIO PAINT PRO**:

Programa de ilustración digital que se ha utilizado para diseñar el logo y algunas imágenes. Para este programa se ha requerido el conocimiento básico de diseño de viñetas, aplicación de tonos y texturas y del manejo de diferentes capas. Para el logo me he basado en el muñeco original de Snap! y le he añadido unas gafas, cejas, una corbata, un reloj y una lupa para darle un toque de profesor. En la figura 4.7 se puede ver la comparación de ambos.

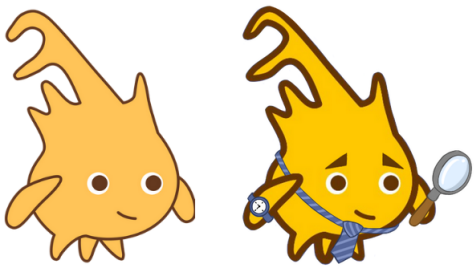


Figura 4.7: Muñeco de Snap! y de Dr. Snap!

Además del logo y el muñeco, se ha cambiado la imagen principal de Snap! Berkeley añadiendo el lema de *analyze your own project* en vez de *build your own project* y cambiando al muñeco de Dr. Snap!. Se hizo gracias al sistema de capas que permite borrar zonas y añadir nuevas y se puede ver en la imagen 4.8.

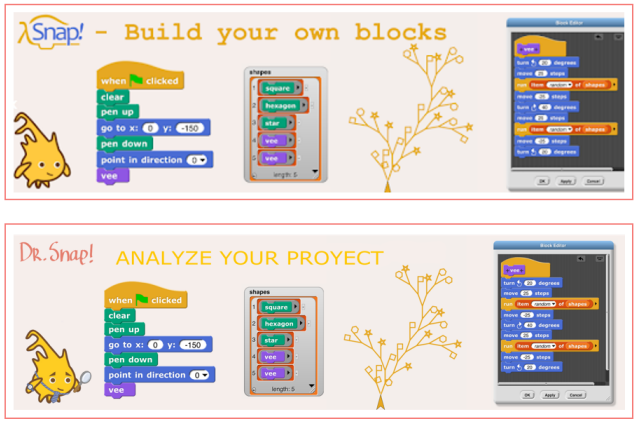


Figura 4.8: Imagen principal aplicación web Dr. Snap!

Por último, para expresar los niveles de cada proyecto se me ocurrió poner al muñeco con un bocadillo de texto diciendo el nivel del proyecto. Los 4 niveles se observan en la figura 4.9.

Figura 4.9: Imagen que dice el nivel de un proyecto.

d) **JAVASCRIPT**:

Se ha usado JavaScript para implementar un conjunto de dashboards. Un dashboard es un gráfico que muestra datos en forma de gráficos (diagrama de barras, de línea o donuts, por ejemplo) y con ello ver un resumen de los datos para poder estudiarlos y analizar posibles tendencias . En esta aplicación se han desarrollado tres de ellos gracias a JavaScript y a su biblioteca de código abierto Chart.js que admite ocho tipos de gráficos (de línea, de barra, de donut, circular, burbuja, radar, polar y de dispersión) y es responsive, adapta su diseño para la mejor visualización posible según el dispositivo desde el que se vea. De los tipos que proporciona hemos seleccionado el gráfico de barras y el gráfico de donut. Esta biblioteca, además de implementar dichos gráficos, permite la configuración y la interacción con ellos. Por ejemplo, se puede cambiar los colores de los gráficos, poner nombres a cada eje, añadir animaciones o gestionar todo tipo de eventos. En la figura 4.10 se observa un gráfico circular y un gráfico de líneas.

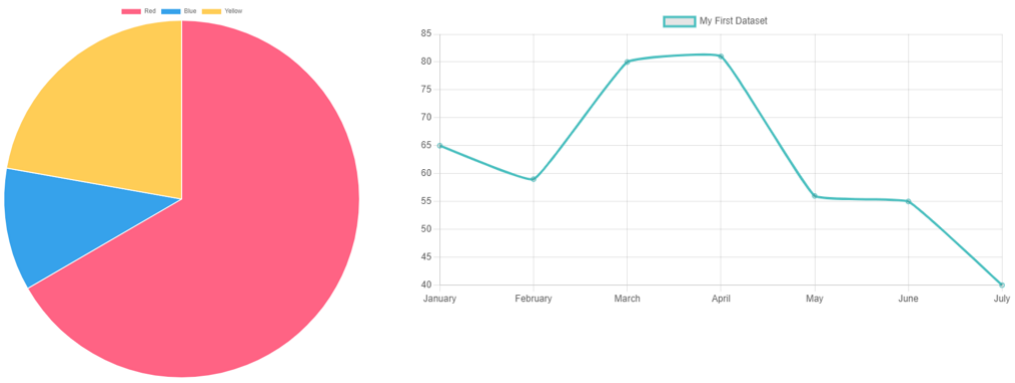


Figura 4.10: Gráfico circular y de líneas respectivamente de Chart.js

4.1.2. BACKEND

a) **DJANGO:**

Hemos descargado Django versión 3.1.1 y Python 3.8.5 para el desarrollo de la aplicación web. Como hemos mencionado Django será el entorno que nos permitirá crear la aplicación web de una forma muy sencilla.

Cuando nos creamos una aplicación en Django, lo primero que tenemos que hacer es configurar los diferentes módulos que hay. En la figura 4.11 podemos ver una representación de las carpetas y módulos más importantes que hay en un proyecto de Django: la carpeta llamada “myproject” contiene todos los módulos y archivos que gestionan el proyecto, la carpeta de nombre “myfirstapp” contiene todos los módulos y archivos que gestionan la aplicación web, el archivo “db.sqlite3” es la base de datos, la carpeta “media” es la que contiene todos los ficheros zip que los profesores suben, la carpeta “template” es en la que se guardan todas las plantillas HTML y el archivo “manage.py” es el manejador que se ejecuta, por ejemplo, para lanzar la aplicación o para crear nuevas migraciones.

Veamos los módulos más importantes con mayor profundidad:

- settings.py: se debe configurar el nombre de la aplicación que has creado, los servidores que están permitidos, como es en nuestro caso *pythonanywhere* o *mylocalhost*; todos los paths o rutas donde encontrar los archivos estáticos, las plantillas u otros; la base de datos que queremos utilizar (por defecto, utiliza sqlite3) e incluso hay que

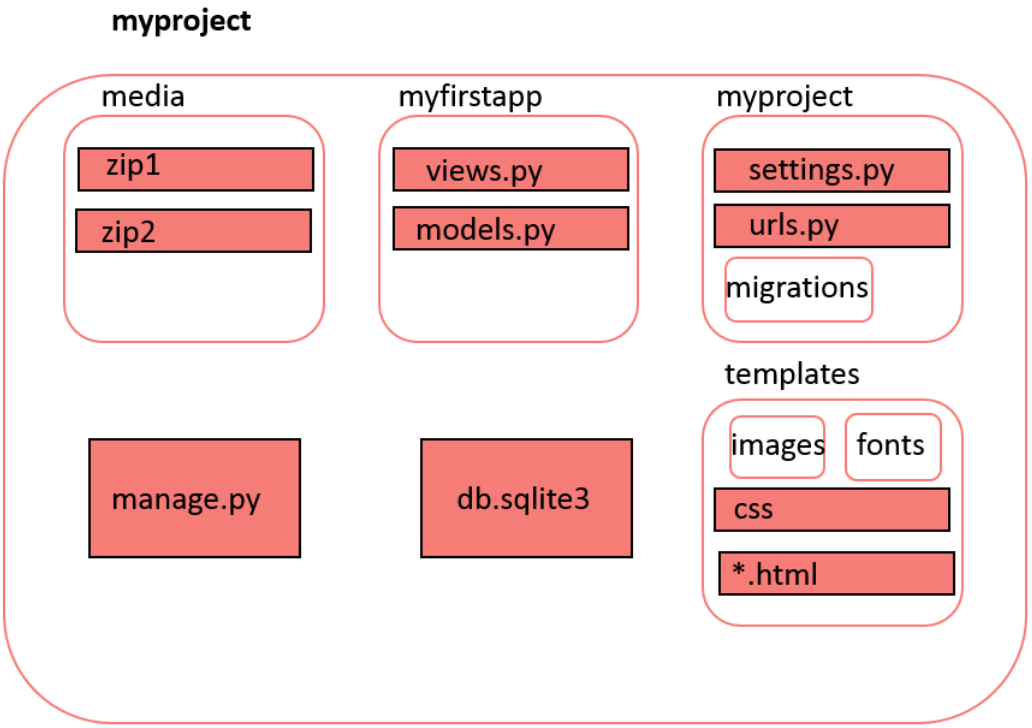


Figura 4.11: Principal directorio del proyecto en Django.

configurar datos como la zona horaria o el idioma.

- urls.py: en este módulo definimos las posibles urls que vamos a encontrar en nuestra aplicación web y especificamos la vista asociada a dicho path url. En la tabla 4.1 podemos ver un resumen de las distintas urls que hemos definido con su vista asociada y la funcionalidad que tienen.

- views.py: son las funciones que reciben una petición HTTP, deben devolver una respuesta HTTP y están asociadas a un path de url como hemos explicado anteriormente. El ORM de Django recibe un path con una petición, mapea en el listado de urls a qué vista corresponde y se ejecuta dicha función para devolver la respuesta correcta renderizada con una de las plantillas HTML. Es decir, por ejemplo, si hacemos desde el navegador una petición GET para el recurso /analyze, la función vista analyze nos devolverá la respuesta HTTP definida en la función renderizada con la plantilla que hicimos para dicho recurso. La figura 3.8 resume lo explicado. También se pueden hacer redirecciones a otras páginas, en caso de error mostrar página

PATH URL	VISTA	FUNCIÓN
/	principal	Proporciona la página principal de la aplicación
/contact	info	Página que da información sobre la aplicación
/analyze	analyze	Página que permite analizar los proyectos o ficheros comprimidos en un zip según el modo en el que estés.
/basic	basic	Informa sobre los criterios para nivel básico.
/intermediate	intermediate	Informa sobre los criterios para nivel intermedio.
/advanced	advanced	Informa sobre los criterios para nivel avanzado.
/login	login_user	Permite iniciar sesión.
/type-signup	choose	Permite registarse eligiendo si eres estudiante o profesor.
/signup	signup	Permite registrarse.
/logout	logout_user	Permite cerrar sesión.
/projects	show_projects	Permite ver los proyectos analizados o introducir un zip para descargarse el proyecto en csv.
/project/<name>	show_project	Permite ver las puntuaciones de un proyecto y su nivel.
/dashboard	dashboard	Enseña los gráficos con los proyectos por nivel y da la opción de ver los de un zip o la comparación de dos zips.
/dashboard/<tag>	dashboard_level	Permite ver los proyectos al hacer un click en uno de los niveles de los dashboard.
/admin/	admin.site.urls	Gestiona los usuarios y contraseñas, entre otras.

Cuadro 4.1: Paths admitidos con sus vistas asociadas de Django.

404 o, incluso, pasar datos a las plantillas.

- models.py: se definen las clases y sus respectivos campos para la base de datos. En nuestra aplicación web tendremos tres clases: usuario, proyectos y el tipo de usuario (profesor o estudiante). En la figura 4.12 quedan reflejadas las tres clases y sus campos correspondientes. Cabe destacar que la clase user ya viene implementada por el sistema de autenticación de usuarios de Django (en "django.contrib.auth"), el cual proporciona funciones como login, logout o signup o la clase *models.user*.

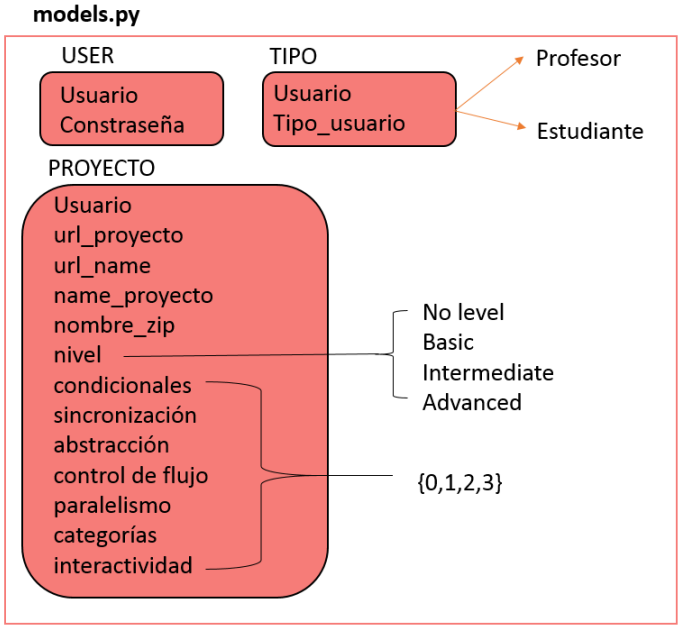


Figura 4.12: Clases y campos definidos en models.py

b) **SQLITE3:**

Como hemos dicho previamente, nuestra base de datos será la que viene por defecto en Django, es decir, sqlite3. Es una base de datos sencilla y para crearla lo primero que debemos hacer es indicarla en settings.py. Una vez definidas las clases en models.py se debe hacer una migración con los cambios hechos para luego migrarla. Cada vez que se cree una clase, se añada un campo nuevo o se elimine se debe hacer una migración que refleje estos cambios y luego migrarla a la base de datos. En la figura 4.13 se observan todas las tablas que hay creadas, sin embargo, las únicas que hemos creado nosotros, gracias al módulo de models.py, son las que empiezan por el nombre de la aplicación, es decir,

myfirstapp_proyectos y *myfirstapp_tipo*. Todas las demás son implementadas por Django para diferentes funcionalidades, por ejemplo, *auth user* es la tabla que representa la ya comentada clase user que viene ya predefinida con los campos de contraseña, Gmail, primer apellido...

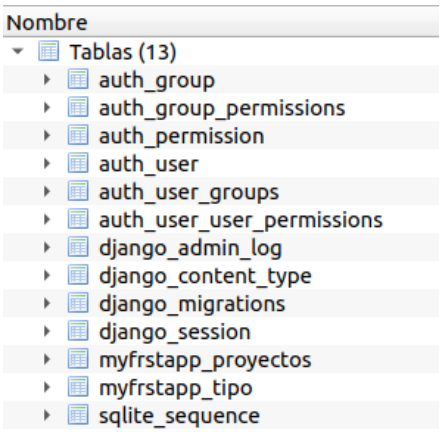


Figura 4.13: Tablas de la base de datos de la aplicación

c) **PYTHONANYWHERE:**

Para lanzar la aplicación de Django en pythonanywhere lo primero que debes hacer es crearte una cuenta de forma gratuita y clonar tu proyecto que debe estar previamente subido a GitHub. El siguiente paso es crearte un entorno virtual y descargar Django. Hay que configurar los paths del directorio donde está el código, las templates y los recursos estáticos. Una vez hecho todo esto ya tenemos disponible nuestra aplicación web durante 3 meses, en nuestro caso, en `http://prodriguezmartin.pythonanywhere.com`.

Una vez que lancé la aplicación web al probarla tenía un error que impedía usarla: no dejaba hacer peticiones ni redirecciones a otros sitios webs a no ser que estos estuvieran en la *whitelist* de pythonanywhere o que mi cuenta fuese de pago. Para solucionar el problema, me puse en contacto con los técnicos y conseguí, tras mucha insistencia, añadir el dominio de `https://cloud.snap.berkeley.edu/` a la *whitelist* ya que me denegaron el dominio de https://snap.berkeley.edu/ lo que supuso algunos cambios mínimos en el código de la aplicación web.

4.2. Funcionalidades de Dr. Snap!

En este apartado vamos a explicar todas las funciones que he implementado para que la aplicación web cumpla su cometido como herramienta docente. Estas siguen un orden cronológico ya que algunas han surgido con el objetivo de mejorar o complementar otras.

4.2.1. Analizar proyectos

Para analizar un proyecto se necesita la URL del proyecto de Snap! y así conseguir el XML donde viene guardada la información de bloques, scripts y sprites del proyecto. La URL se consigue a través de un formulario como el de la figura 4.14. Una vez que tenemos la URL, se pensó que habría que interactuar con la API de Snap!, sin embargo, tras ver el código fuente descubrimos que no es necesario ya que existe una función implementada en JavaScript que te devuelve el XML simplemente haciendo un GET con el recurso /projects/user_name/project_name a Snap! Berkeley.

En la figura 4.14 se ven los pasos necesarios para conseguir los datos de un proyecto de Snap! Berkeley: el usuario envía la URL de su proyecto a través de un POST, se parsea la URL, se forma la nueva URL y se hace una petición GET con esa URL a Snap! Berkeley; el cual, nos devuelve el XML correspondiente.

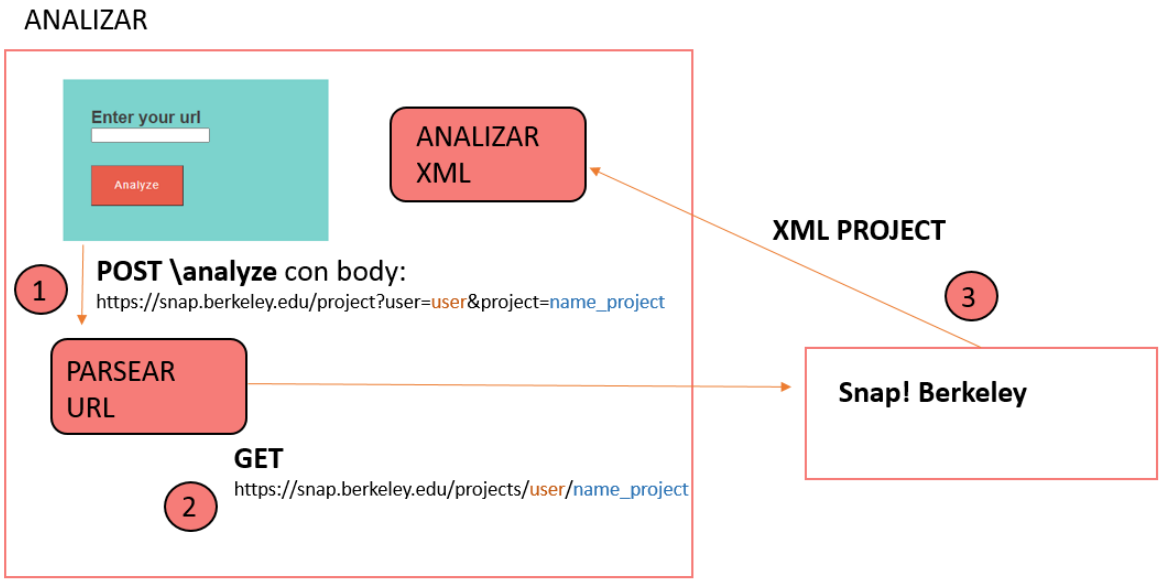


Figura 4.14: Petición del XML de un proyecto

Una vez que tenemos el XML tenemos que parsearlo para poder obtener todos los datos necesarios y, para ello, hemos hecho uso de la API xml.sax. Un archivo XML es un texto compuesto de etiquetas creado para formar una estructura de datos e información. Gracias al *ContentHandler* y al *parser* de xml.sax podemos acceder a cada etiqueta y a su contenido. Para ello, la clase *ContentHandler* define 3 funciones muy importantes:

- *def __init__ (self)* en la que se deben inicializar todas las variables.
- *def startElement (self, name, attrs)* es la función que encuentra inicios de etiqueta XML, donde name se convierte en el nombre de la etiqueta y attrs en los atributos que hay en la etiqueta. Por ejemplo, block s="move" el nombre de la etiqueta es block, el atributo es s y su valor es move. En este punto guardaremos en un archivo JSON los atributos y valores correspondientes según necesitemos. Se ha tomado la decisión de trabajar con JSON porque es mucho más sencillo y rápido.
- *def endElement(self, name)* es la función que se encuentra con el cierre de etiquetas XML y sirve para saber que ya has avanzado a otra etiqueta.

Para analizar un documento XML tenemos que:

- Crear un lector de XML con *make_parser()*.
- Crear un manejador de contenido del XML con *myContentHandler()*.
- Definir las funciones del *ContentHandler* y añadírselas al *parser* con *setContentHandler(theHandler)*.
- Parsear la url donde se encuentra el archivo XML con el parser ya definido.

Este paso puede que haya sido el más complicado de todos pues los proyectos de Snap! tienen una estructura XML más o menos similar pero no llega a ser única por lo que al intentar analizar las etiquetas resultó difícil porque no todos los proyectos encajaban con lo que había definido. En la figura 4.15 se refleja el esquema de etiquetas que se ha supuesto y cómo lo hemos guardado. Con esta estructura pretendemos poder guardar en un JSON solamente los sprites que hay, los scripts y los bloques que se utilizan. Como extra se han guardado las variables y los bloques que puede crear un usuario.

El JSON se compone de 4 listas (sprites, variables, block-definition y name block-definition) que contienen diccionarios con sus datos. Por ejemplo, en sprites se guardan diccionarios por cada bloque parseado con los campos nº sprite, nº script, block y nº block.

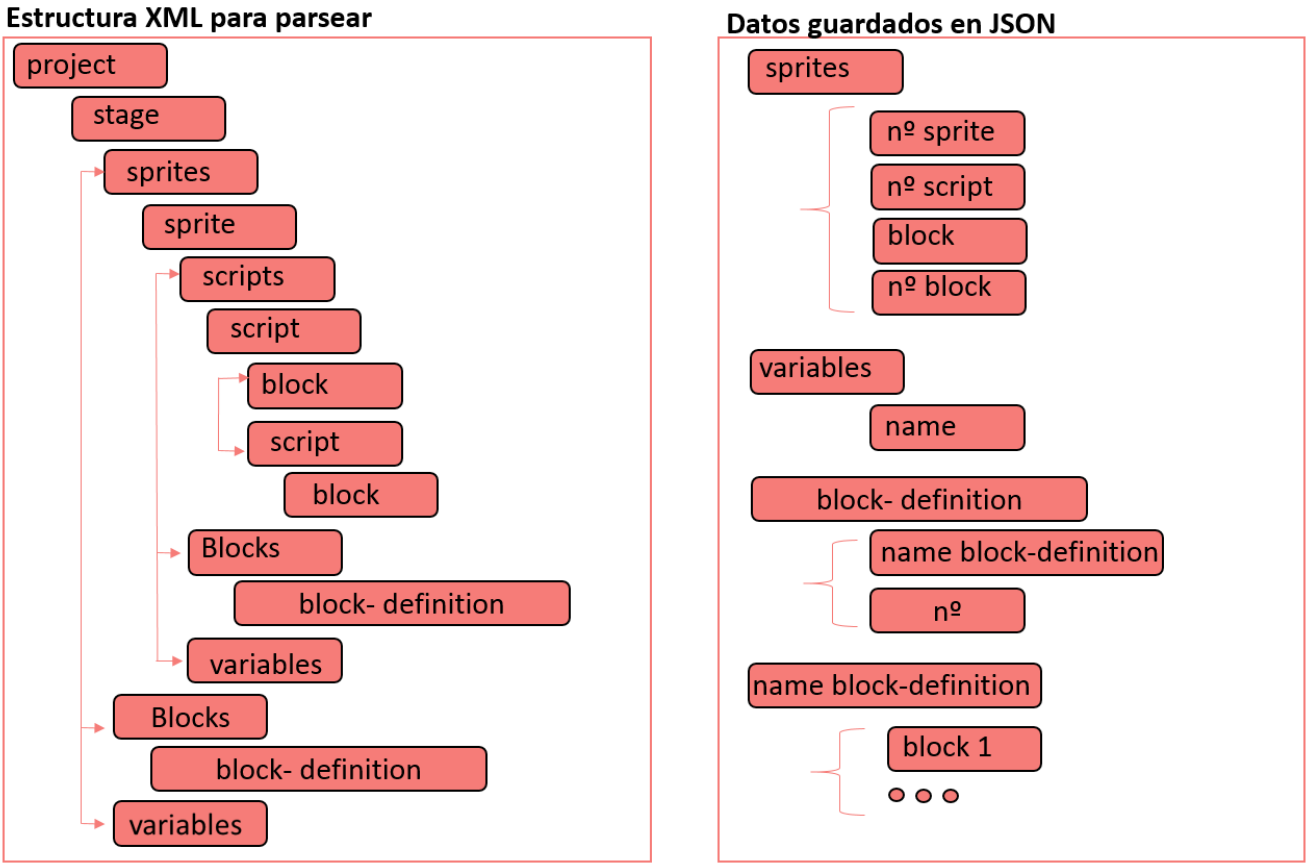


Figura 4.15: Estructura seguida para parsear el XML y estructura seguida para guardar los datos parseados

Una vez que los datos se han guardado en el JSON, se procedió a implementar la rúbrica en Django con ocho categorías: paralelismo, abstracción, datos, flujo de control, diversidad o categorías, condicionales, sincronización e interactividad. Los criterios de evaluación quedan reflejados en la tabla 4.2 y tiene de base la de Dr. Scratch [14]. Cada categoría tendrá, por tanto, una puntuación Nula (0), Básica (1), Intermedia (2) o Avanzada (3), y para la puntuación global, se hará la media de las ocho categorías resultando una puntuación de: nivel nulo de 0 a 0,5, básico de 0,5 a 1,5, intermedio de 1,5 a 2,5 y avanzado de 2,5 a 3. Si el usuario ha iniciado sesión toda esta información se guarda en la base de datos en la tabla *proyecto* ya mencionada.

	Básico	Intermedio	Avanzado
Paralelismo	1 sprite & >= 2 scripts receiveGo receiveKey	>= 2 sprites & >= 1 script receiveGo receiveKey	>=1 sprite & 2 >= scripts receiveCondition receiveMessage receiveOnClone
Condicionales	Do if	DoIfElse ReportIfElse	ReportAnd ReportOr ReportNot
Control flujo	>= 2 blocks en un script	DoForever doRepeat	DoUntil for
Abstracción	>=2 scripts	>=2 scripts, sprites	Definition-blocks
Sincronización	doWait	doBroadcast receiveMessage doStopThis doPauseAll	doWaitUntil doBroadcastAndWait receiveOnClone receiveCondition
Interactividad	receiveGo receiveKey	receiveInteraction reportTouchingObject reportKeyPressed	doAsk video audio
Datos	Atributos de los personajes	Operaciones con variables	Operaciones con listas
Diversidad	<= 2 categorías	2<categorías<=6	>=7 categorías

Cuadro 4.2: Rúbrica implementada para analizar proyectos de Snap!

4.2.2. Usuarios

Se propuso la idea de que los usuarios tuviesen la opción de hacerse una cuenta y así poder guardar todos los proyectos analizados en la base de datos. Para ello, se implementó en primer lugar la opción de iniciar y cerrar sesión gracias a las funciones de *login* y *logout* que ofrece Django en su sistema de autenticación (*django.contrib.auth.models*). Este sistema permite distinguir cuando un usuario está autenticado y cuando no, pudiendo hacer funciones exclusivamente para usuarios como es el caso del recurso */projects* que muestra los proyectos que el usuario ha analizado previamente. Sin embargo, para crear usuarios se tenía que gestionar desde el recurso /admin y solo lo podía hacer el superusuario, por lo que se añadió la opción de hacerse una cuenta. Se puede importar un formulario para crear un usuario que ya viene predefinido (pero de aspecto modificable) en *django.contrib.auth.models* y a partir de él, un usuario se registra y solo faltaría que en la función de *signup* autenticásemos ese usuario y lo iniciásemos sesión directamente. En la figura 4.16 se observa la página de iniciar sesión y la de hacerse una cuenta. Cabe destacar que se implementó que si el usuario estaba logueado apareciese un recuadro arriba de todas las páginas con la opción de cerrar sesión. Los usuarios con sus contraseñas quedan guardados en la tabla de la base de datos denominada *auth_user*.

Figura 4.16: Inicio de sesión y registro en Dr. Snap!

4.2.3. Modo profesor o estudiante

Con la anterior opción se conseguía tener una base de datos con los proyectos analizados de los usuarios pero Dr. Snap! pretende ir más allá que analizar solo proyectos, como herramienta

docente debe aportar distintas funcionalidades según el tipo de usuario que seas. Surge así, la distinción entre si eres estudiante o eres un profesor. Para ello se crea el modelo "tipo" que ya hemos mencionado donde se asocia el usuario con el tipo de usuario. Se implementa de forma que al darle al enlace de "hacerse una cuenta", mostrado en la figura 4.16, te lleve al recurso /type-signup donde tienes dos formularios para elegir si eres estudiante o profesor. Una vez que seleccionas el tipo, la nueva entrada se guarda en la base de datos y se te redirige a la página de registro de la figura 4.16. Esta funcionalidad queda reflejada en la figura 4.17. A continuación, se definen las prestaciones de casa modo:

- Estudiante: tiene la opción de analizar sus proyectos y de ver gráficos con los proyectos analizados según los niveles obtenidos.

- Profesor: puede analizar proyectos propios o un conjunto de proyectos que deben estar comprimidos en formato ZIP. Se pensó que un profesor tendría varias clases y sería más sencillo analizar todos los proyectos de una clase subiéndolo como ZIP. Además, se da la opción de poder descargar en formato CSV la clase entera introduciendo el nombre del archivo ZIP que haya sido analizado previamente.

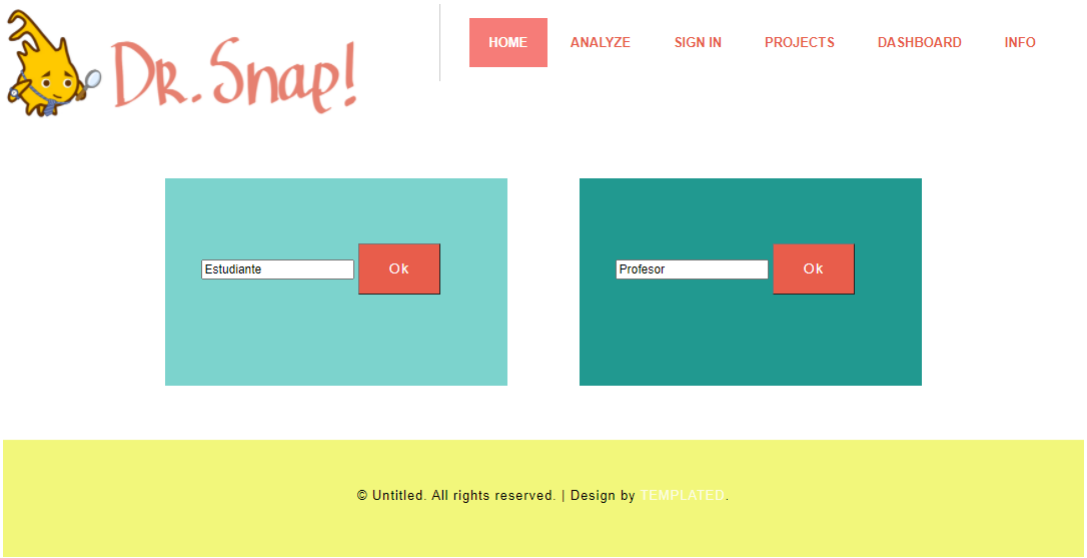


Figura 4.17: Página para elegir el tipo de usuario en el proceso de registro en Dr. Snap!

4.2.4. Analizar ZIPs

Para poder analizar un archivo ZIP debe ser subido primero a la aplicación por el usuario de tipo profesor. En la función de analizar proyectos, si estás logueado como profesor aparece un nuevo formulario en el que se ha implementado la posibilidad de importar un ZIP desde el ordenador. Para trabajar con el archivo comprimido hemos utilizado la biblioteca "FileSystemStorage" importada de "django.core.files.storage" y con la biblioteca "zipFile". De primeras, se dudó si trabajar con el archivo en local o en disco. Al final, me decanté por guardarlo en disco y trabajar con él ya que era menos complejo. En la figura 4.18 se muestran las opciones que ofrece la página analizar en caso de ser profesor. Los pasos para esta funcionalidad son los siguientes:

- Guardar el archivo ZIP en la carpeta llamada media, vista en la figura 4.11. Esto se consigue con una función de "FileSystemStorage" llamada *fs.save()*.
- Quedarnos con la url donde se encuentra el ZIP con la función *fs.url()*.
- Descomprimir el ZIP con "zipFile" en una carpeta que se hace llamar: *nombredeusuario&nombredelarchivo* con la función *fs.urlextractall (password, path)*.
- Analizar cada proyecto calculando las puntuaciones y guardando los datos en la base de datos. En este punto es cuando se decide añadir un campo del modelo *proyecto* que se llamó *nombre_zip* ya que de esta forma los proyectos que no pertenecen a ningún ZIP simplemente este campo se pondría a NULL.

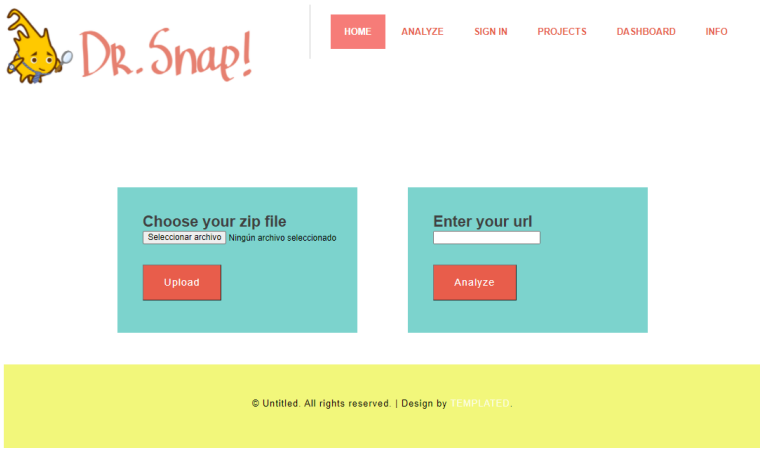


Figura 4.18: Formularios para subir un ZIP o analizar un proyecto en Dr. Snap!

4.2.5. Dashboard

Los 3 dashboard que tenemos son los que enseñamos en la figura 4.19 y otro más donde se comparan los resultados de dos ZIPs que se hayan subido a la aplicación (figura 4.20). Los datos que hemos cogido son los resultados de los proyectos que hay guardados en las cuentas de un estudiante o un profesor, es decir, han debido ser analizados previamente y guardados en la base de datos. Según el modo:

- Estudiante: sólo están disponibles los gráficos de la figura 4.19 y en ella se muestran los resultados de los proyectos que el alumno ha analizado en Dr. Snap! en un diagrama de barras y en donut.

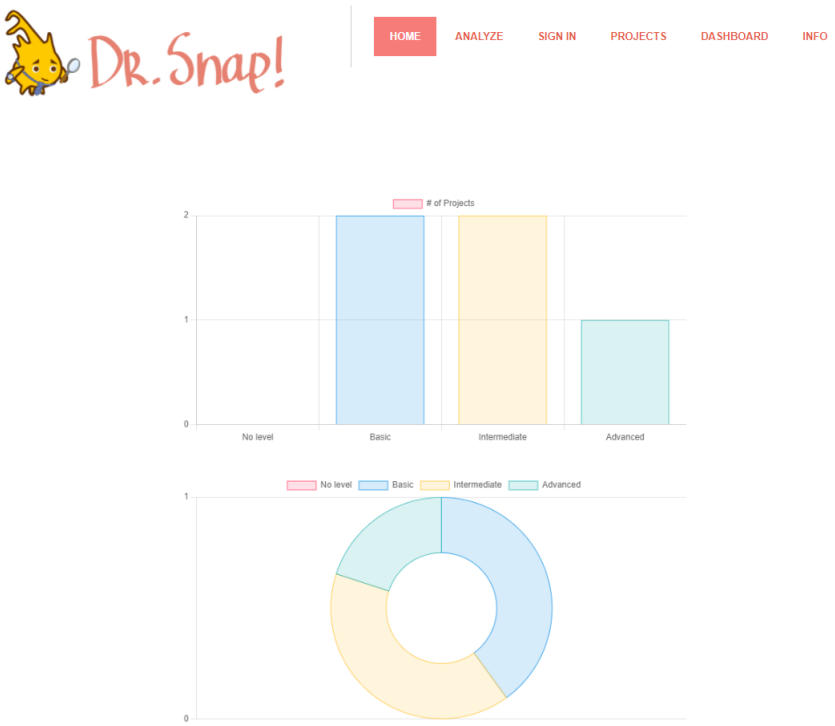


Figura 4.19: Dashboard de barra y de donut.

- Profesor: además de los mencionados, se da la opción de ver los gráficos con los datos de una clase o incluso comparar un gráfico con dos clases diferentes (figura 4.20). Cuando hablamos de clase se debe entender como un ZIP con los proyectos de los alumnos de una clase y que esta se ha analizado previamente gracias a la opción que damos de analizar ZIP.

Figura 4.20: Comparación de dos Zips.

Para implementarlos hemos tenido que añadir un elemento “canvas” de HTML que básicamente es un contenedor donde se dibujan los gráficos gracias a la biblioteca de JavaScript ya mencionada “Chart.js”. Una vez que dichos gráficos estaban implementados, se ha añadido la funcionalidad de que si haces click en una de las barras, por ejemplo en la azul de basic, te redirija a una tabla donde aparecen todos los proyectos de nivel básico. Esto ha requerido el aprendizaje de la programación basada en eventos ya que hacer click en la pantalla es un evento llamado onClick y hay que manejarlo para que distinga si estás pinchando dentro del elemento *canvas*, si es fuera de las barras o porciones del donut y en caso de que sea dentro de ellas, distinguir qué nivel está pinchando.

Capítulo 5

Experimentos y validación

Para validar y testear la aplicación web hemos distinguido diferentes fases o puntos a lo largo que ha ido avanzando el proyecto. En orden cronológico tenemos las siguientes validaciones:

5.1. Parseo del XML

Para ello se tuvieron que analizar más de 50 proyectos, algunos creados por mí y otros cogidos de proyectos subidos a Snap! Berkeley. Con esto se pretendía deducir la estructura más común y global que presentaban la mayoría de los proyectos en formato XML. Por ejemplo, en la figura 5.1 se muestra el fichero XML que se obtiene al añadir simplemente el bloque *move* en un script. En primera instancia se supuso que los XML tendrían las etiquetas project, stage, sprites, sprite, scripts, script y block ya que en stage están todos los sprites, en cada sprite están los scripts y en cada script están los bloques utilizados. Sin embargo, esta aproximación era errónea ya que al agregar nuevos bloques el XML añadía nuevas etiquetas que provocaban continuos errores de lectura. Algunos puntos a tener en cuenta son:

- Se debe distinguir la etiqueta *block* con dos atributos muy importantes: *s* que hace referencia a todos los bloques ya definidos por Snap! Berkeley y *custom* que hace referencia a los creados por los usuarios.

- Se debe tener en cuenta que los bloques definidos pueden aparecer tanto a la altura de block (con el atributo *custom*) como a la altura de los sprites (con la etiqueta *block-definition*).

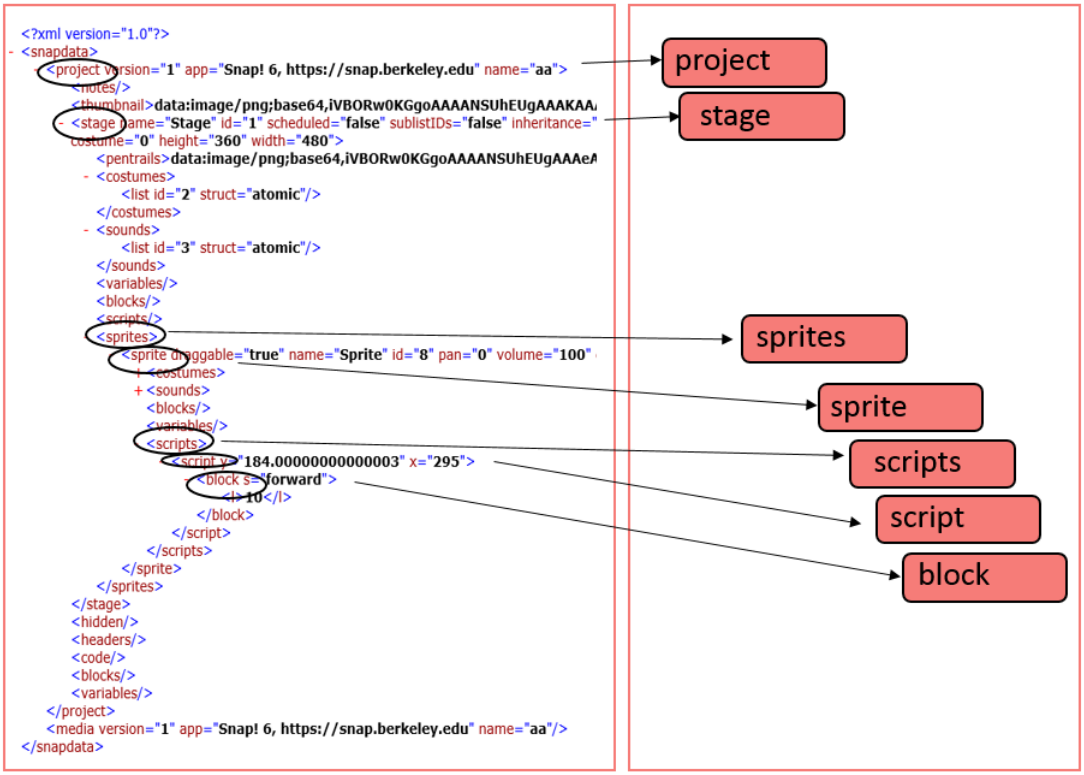


Figura 5.1: Primera aproximación estructura XML

- Las variables se pueden tratar de igual manera que el punto anterior.
- Un bloque ya predefinido por Snap! tiene la etiqueta block y el atributo *s*. Sin embargo, hay algunos bloques que pueden tener scripts dentro como observamos en la figura 5.2. Esto debe tenerse en cuenta a la hora de abrir o cerrar las etiquetas scripts ya que puede hacerse de forma errónea.

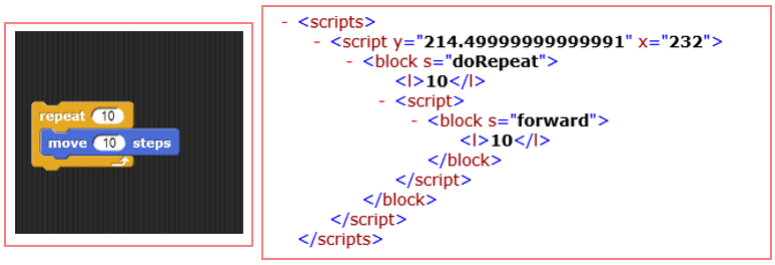


Figura 5.2: Script dentro de un bloque

La estructura final se mostró anteriormente en la figura 4.15 junto con el JSON donde se guardaron los datos de interés a la vez que se parseaba el XML. En la figura 5.3 se muestra el archivo

JSON que se guarda tras analizar el proyecto de la figura 5.2 y podemos observar que en la lista sprites tenemos dos diccionarios: uno para el bloque “repeat” y otro para el bloque “move”. Destacamos que ambos bloques pertenecen al sprite 1 y al script 1 como se indica en los valores de cada diccionario y como extra guardamos el número de cada bloque por saber el máximo de bloques que se usan en el proyecto. Al no tener variables ni bloques definidos, ambas listas están vacías.

/home/prodriguezmartin/snap/data.json

```
{
    "sprites": [
        {
            "sprite": 1,
            "script": 1,
            "block": "doRepeat",
            "num_block": 1
        },
        {
            "sprite": 1,
            "script": 1,
            "block": "forward",
            "num_block": 2
        }
    ],
    "variables": [],
    "block-definition": []
}
```

Figura 5.3: JSON que guarda los datos de un XML.

5.2. Implementación rúbrica en Django

Para la implementación de la rúbrica se han definido 3 funciones auxiliares que dan información sobre los datos del JSON guardado y 8 funciones principales que calculan la puntuación de cada sección. Las 3 funciones auxiliares son: calcular el número total de scripts, calcular el número total de sprites y ver si hay más de dos bloques en un script. Para validar esto usaremos un ejemplo fácil pero con mayor contenido como el de la figura 5.4 que consta de 1 sprite, 2 scripts y 7 bloques.

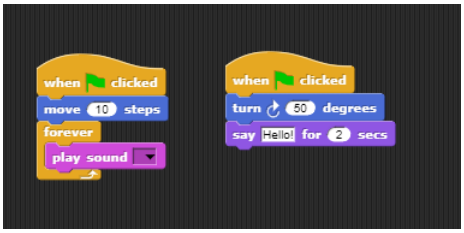


Figura 5.4: Ejemplo de Snap!

Las puntuaciones que deberían aparecer basándonos en la tabla 4.2 son:

- Paralelismo: 1 sprite, 2 scripts y uso de bandera verde supone estar en la puntuación básica (1).
- Condicionales: no se usa ningún bloque condicional ni operadores lógicos por lo que la puntuación es nula (0).
- Control de flujo: el primer script tiene 3 bloques y el segundo 2 lo que supone una puntuación básica. Sin embargo, el uso del bloque "forever" asciende la puntuación a intermedia (2).
- Abstracción: el tener 1 sprite, 2 scripts y ningún bloque definido supone una puntuación básica (1).
- Sincronización: en este ejemplo no hay ningún tipo de sincronización por lo que la puntuación debería resultar nula (0).
- Interactividad: la única interacción que observamos es la bandera verde por lo que tenemos una puntuación básica (1).
- Categorías o diversidad: tenemos en total 4 categorías diferentes por lo que la puntuación sería intermedia (2).
- Representación de datos: no se han definido ni variables ni listas pero usamos los atributos de posición "move" y "turn" por lo que la puntuación será básica (1).

En la figura 5.5 confirmamos que las puntuaciones esperadas son las correctas y validamos por tanto la implementación de la rúbrica.

Secciones	Puntuación
Condicionales	0
Sincronización	0
Control de flujo	2
Abstracción	1
Paralelismo	1
Categorías	2
Interactividad	1
Representación de datos	1

Figura 5.5: Ejemplos análisis de un proyecto.

5.3. Iniciar sesión y registrarse

Como ya hemos mencionado, para registrarse es necesario elegir el tipo de usuario (profesor o estudiante) y rellenar un formulario con los campos: nombre de usuario y contraseña. Crearemos un usuario que sea de tipo profesor, se llame *maria21* y su contraseña sea *telecomunicaciones2021*. En la figura 5.6 se puede observar como el usuario ha quedado bien registrado en la base de datos.

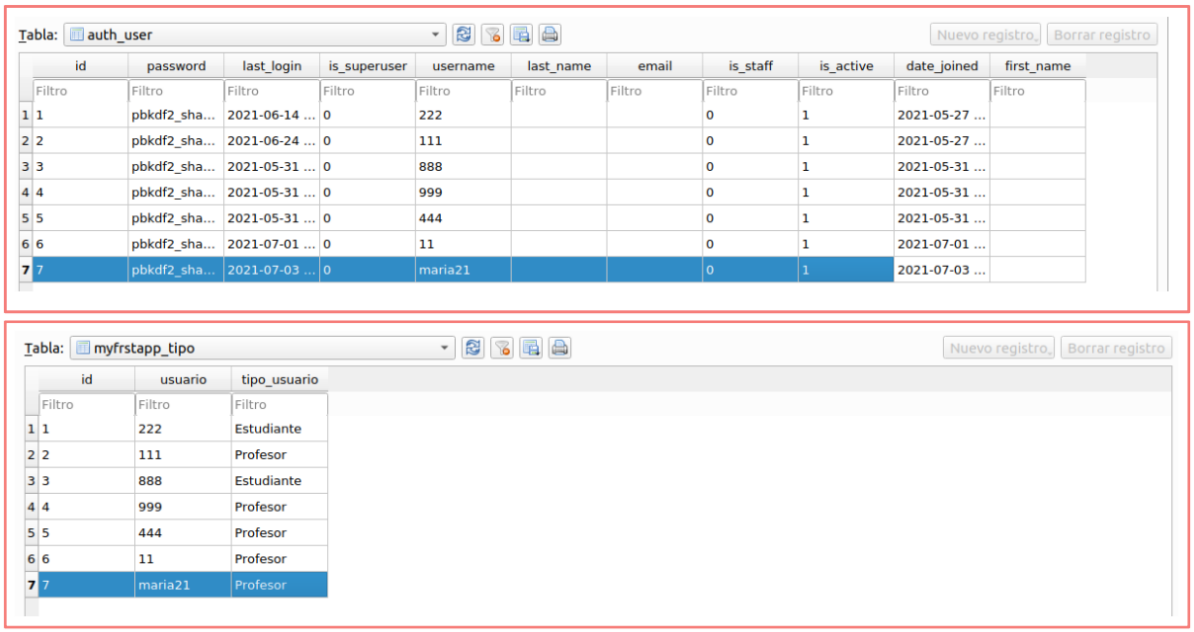


Tabla: auth_user

	id	password	last_login	is_superuser	username	last_name	email	is_staff	is_active	date_joined	first_name
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	pbkdf2_sha...	2021-06-14 ...	0	222			0	1	2021-05-27 ...	
2	2	pbkdf2_sha...	2021-06-24 ...	0	111			0	1	2021-05-27 ...	
3	3	pbkdf2_sha...	2021-05-31 ...	0	888			0	1	2021-05-31 ...	
4	4	pbkdf2_sha...	2021-05-31 ...	0	999			0	1	2021-05-31 ...	
5	5	pbkdf2_sha...	2021-05-31 ...	0	444			0	1	2021-05-31 ...	
6	6	pbkdf2_sha...	2021-07-01 ...	0	11			0	1	2021-07-01 ...	
7	7	pbkdf2_sha...	2021-07-03 ...	0	maria21			0	1	2021-07-03 ...	

Tabla: myfrstapp_tipo

	id	usuario	tipo_usuario
	Filtro	Filtro	Filtro
1	1	222	Estudiante
2	2	111	Profesor
3	3	888	Estudiante
4	4	999	Profesor
5	5	444	Profesor
6	6	11	Profesor
7	7	maria21	Profesor

Figura 5.6: Base de datos con las tablas *user* y *tipo*.

5.4. Proyectos en la base de datos

Al analizar un proyecto de un usuario registrado se deben guardar el usuario, las puntuaciones del proyecto y su nivel asociado, el nombre del proyecto, la URL de nuestra aplicación web, la URL de Snap! y si es zip o no. Desde el usuario de *maria21* vamos a analizar el proyecto de la figura 5.4 y podemos ver en la figura 5.7 que se han guardado todos los datos de forma correcta en la base de datos. Con esta tabla seremos capaces de acceder cuando queramos a las puntuaciones del proyecto gracias al recurso */project/name_project¿* que accede a esta base de datos filtrando por usuario y nombre del proyecto.

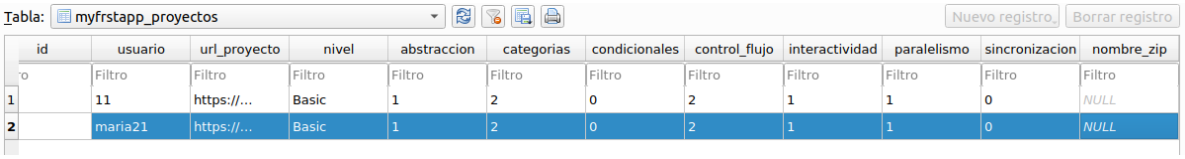


Tabla: myfrstapp_proyectos

Nuevo registro | Borrar registro

	id	usuario	url_proyecto	nivel	abstraccion	categorias	condicionales	control_flujo	interactividad	paralelismo	sincronizacion	nombre_zip
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1		11	https://...	Basic	1	2	0	2	1	1	0	NULL
2		maria21	https://...	Basic	1	2	0	2	1	1	0	NULL

Figura 5.7: Proyecto guardado en la base de datos.

5.5. ZIPs en la base de datos

De igual forma que en la sección anterior, vamos a comprobar que al analizar un ZIP los datos quedan correctamente guardados en la base de datos. El ZIP denominado *validacion.zip* usado de ejemplo consta de 3 proyectos sacados de Snap! Berkeley llamados: *Dogde.xml*, *S2E1 - Human body scanner.xml* y *Sierpinski carpet.xml*. En la figura 5.8 podemos ver como se han añadido los tres proyectos nuevos pertenecientes a *validacion.zip* a la base de datos que teníamos en la figura 5.7.


Tabla: myfrstapp_proyectos

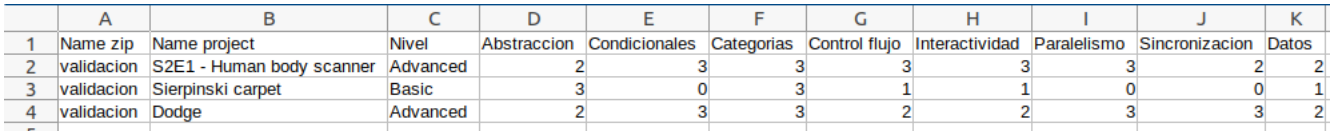
Nuevo registro | Borrar registro

	usuario	url_proyecto	nivel	abstraccion	categorias	condicionales	control_flujo	interactividad	paralelismo	sincronizacion	nombre_zip	name_proy
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	11	https://...	Basic	1	2	0	2	1	1	0	NULL	aa
2	maria21	https://...	Basic	1	2	0	2	1	1	0	NULL	aa
3	maria21	/home/...	Advanced	2	3	3	3	3	3	2	validacion	S2E1 - Hur
4	maria21	/home/...	Basic	3	3	0	1	1	0	0	validacion	Sierpinski
5	maria21	/home/...	Advanced	2	3	3	2	2	3	3	validacion	Dodge

Figura 5.8: ZIP guardado en la base de datos.

5.6. ZIP exportado como CSV

Usando el ZIP anterior si le damos a descargar como CSV en nuestra aplicación web observamos en la figura 5.9 que se exportan los datos de forma correcta. Simplemente se muestra el nombre del ZIP, el nombre del proyecto, el nivel del proyecto global y las puntuaciones de cada categoría. Esta funcionalidad se añadió como una ayuda a los profesores.



	A	B	C	D	E	F	G	H	I	J	K
1	Name zip	Name project	Nivel	Abstraccion	Condicionales	Categorias	Control flujo	Interactividad	Paralelismo	Sincronizacion	Datos
2	validacion	S2E1 - Human body scanner	Advanced	2	3	3	3	3	3	2	2
3	validacion	Sierpinski carpet	Basic	3	0	3	1	1	0	0	1
4	validacion	Dodge	Advanced	2	3	3	2	2	3	3	2

Figura 5.9: CSV correspondiente al ejemplo *validacion.zip*.

Capítulo 6

Resultados

Para ver los resultados de nuestra aplicación web haremos una distinción entre el modo profesor y el modo estudiante ya que cada uno tiene diferentes posibles respuestas.

6.1. Modo estudiante

Crearemos un usuario llamado *pablo21* de tipo estudiante y vamos a analizar nuestro primer proyecto llamado *Benham* sacado de la URL `https://cloud.snap.berkeley.edu/project?user=kinestheticlearning&project=Benham`. En la figura 6.1 se muestra el resultado de analizar dicho proyecto.

Figura 6.1: Página resultante de analizar el proyecto *Benham*.

Ahora si analizamos varios proyectos más con el mismo usuario tenemos la opción de ver una tabla donde se muestran todos los proyectos analizados con sus respectivas puntuaciones gracias al recurso */projects*. Esto se refleja en la figura 6.2 y cabe mencionar que la fila de *proyecto* son enlaces que te llevan a los resultados individuales del análisis de cada uno.

Figura 6.2: Página que muestra todos los proyectos analizados por el usuario.

Otra funcionalidad es que el estudiante tiene la opción de ver dos dashboard con los proyectos analizados como en la figura 6.3. En ellos se puede ver que hay 3 proyectos de nivel intermedio y uno de nivel avanzado.

Figura 6.3: Página que muestra los dashboard de los proyectos.

Por último, si hacemos *click* en la barra amarilla de valor intermedia, s nos va a redirigir a una página que implementa la tabla de proyectos vista en la figura 6.2 pero solamente mostrando

los proyectos de nivel intermedio. Esta funcionalidad es útil para cuando un estudiante quiera ver solamente los proyectos de un único nivel.

Figura 6.4: Página que muestra los proyectos del nivel cuando haces *click* en el dashboard.

6.2. Modo profesor

Aparte de tener las mismas funcionalidades que se ofrecen en el modo estudiante veremos los resultados de analizar un ZIP denominado *modo-profesor.zip* que guarda los mismos proyectos que ha analizado el estudiante *pablo21* para así ver que los resultados son los mismos. En la figura 6.5 observamos los resultados de analizar el ZIP y vemos que son los mismos que cuando los proyectos se analizaron por separado.

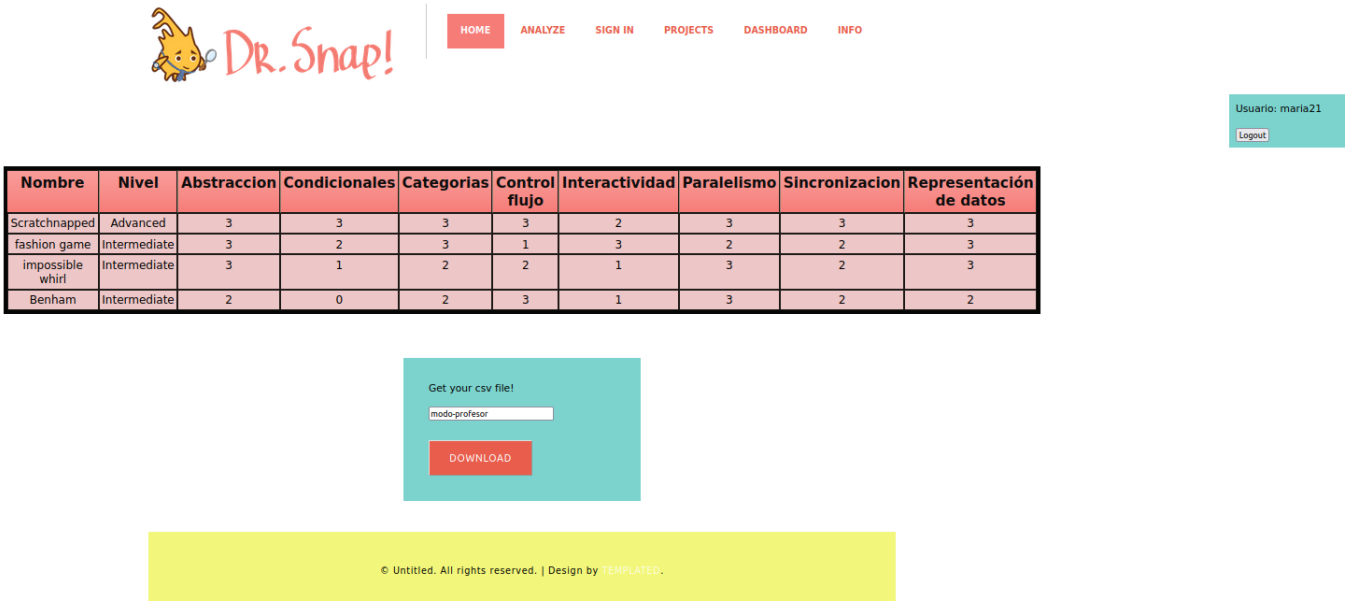


Figura 6.5: Página que muestra los proyectos analizados de un ZIP.

Por último, podemos ver los dashboard de los proyectos analizados como los estudiantes pero si queremos ver el de algunos de los ZIPs analizados o comparar dos de ellos tendremos que rellenar los formularios de la figura 6.6.

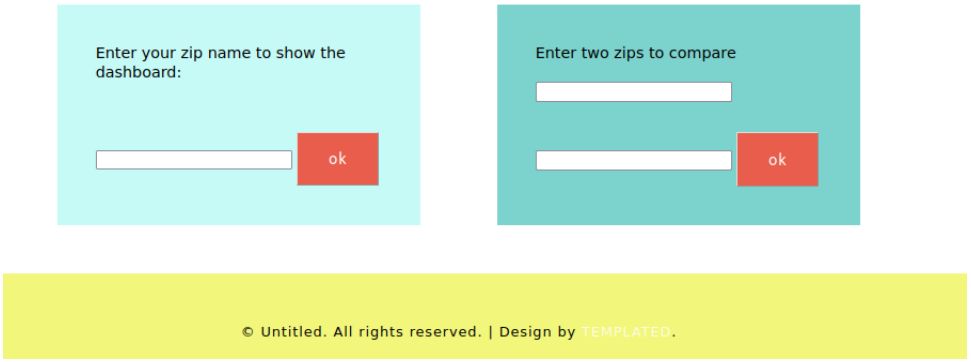


Figura 6.6: Formularios para obtener los dashboard correspondientes.

El resultado de obtener el dashboard de *modo-profesor.zip* resulta el mismo que en la figura 6.3 y el resultado de comparar el archivo *modo-profesor.zip* y el archivo *validaciones.zip* (previamente utilizado) se observa en la figura 6.7

Figura 6.7: Comparación de dos ZIPs

Capítulo 7

Conclusiones

7.1. Consecución de objetivos

La aplicación web ha conseguido los objetivos planteados ya que permite ser una herramienta docente tanto para un alumno como para un profesor. Ofrece diversas funcionalidades que pueden llegar a ser muy útiles para el ámbito académico y para la mejora y desarrollo de los conocimientos de programación. Cabe destacar que funciona solamente si las URLs de los proyectos pertenecen al dominio de `https://cloud.snap.berkeley.edu/` y no a `https://snap.berkeley.edu/` por lo que puede llevar a que te salga un error en la aplicación web. Otro error a mencionar que puede producirse es que el proyecto no cumpla con la estructura de parseo XML que hemos definido y ello haría imposible el análisis de dicho proyecto pero es un caso muy poco común. Por lo demás, la aplicación cumple lo cometido y predefinido en los objetivos.

7.2. Aplicación de lo aprendido

A lo largo de mi carrera he aprendido muchos conocimientos gracias a diversas asignaturas y ello ha hecho posible este trabajo de fin de grado. Estas son:

1. Fundamentos de la Programación me permitió aprender a comprender la programación y comenzar a pensar como un "buen" programador.

2. Programación de Sistemas de Telecomunicación me enseñó la implementación del mo-

delo cliente-servidor.

3. Servicios y Aplicaciones Telemáticas me ha enseñado el uso de Django y el diseño del *FRONTEND* en aplicaciones webs.

4. Desarrollo de aplicaciones telemáticas, aunque nunca llegué a cursarla, con los apuntes de mis compañeros he aprendido el uso de CSS y la mejora en apariencia del diseño web.

5. Ingeniería de Sistemas de Información me ha permitido aplicar los conocimientos de bases de datos que imparte la asignatura.

7.3. Lecciones aprendidas

La realización de este proyecto ha hecho que aumenten mis conocimientos en distintas áreas de telecomunicaciones:

1. Apariencia de páginas webs con CSS y las templates de HTML. He aprendido sobre todo a diseñar botones, menús o formularios.

2. Mejora en el lenguaje Python y el entorno Django. He desarrollado conocimientos en el manejo de estructuras XML, de diccionarios y listas, de archivos JSON y en la abstracción de procedimientos o funciones demasiado grandes para dividirlas en apartados más pequeños. Además, he ampliado mis conocimientos en los métodos de GET y POST para poder hacer peticiones.

3. Probablemente lo que más haya aprendido haya sido el uso de JavaScript puesto que era la primera vez que lo usaba y me ha enseñado sobre todo la programación basada en eventos.

4. Pythonanywhere es un servidor gratuito que no sabía que existía y que es muy útil porque admite proyectos de Django ya implementados y es sencillo de utilizar.

7.4. Trabajos futuros

Un posible proyecto futuro sería la mejora de esta herramienta para que admita todo tipo de proyectos de Snap! independientemente de su estructura XML y que aporte nuevas funcionali-

dades como alguna de *machine learning* que pueda ver patrones de error repetidos y ayude a los estudiantes a mejorar en su programación evitando dichos patrones.

Apéndice A

Vistas creadas y templates HTML asociadas

Como hemos explicado anteriormente, en Django se puede renderizar una vista con una plantilla HTML mostrando en ella el contenido de dicha vista. En este apéndice veremos las vistas creadas y sus correspondientes plantillas para luego poder entender en el siguiente capítulo los diagramas de flujo.

A.1. Analyze

Vista que permite analizar los proyectos o ZIPs.

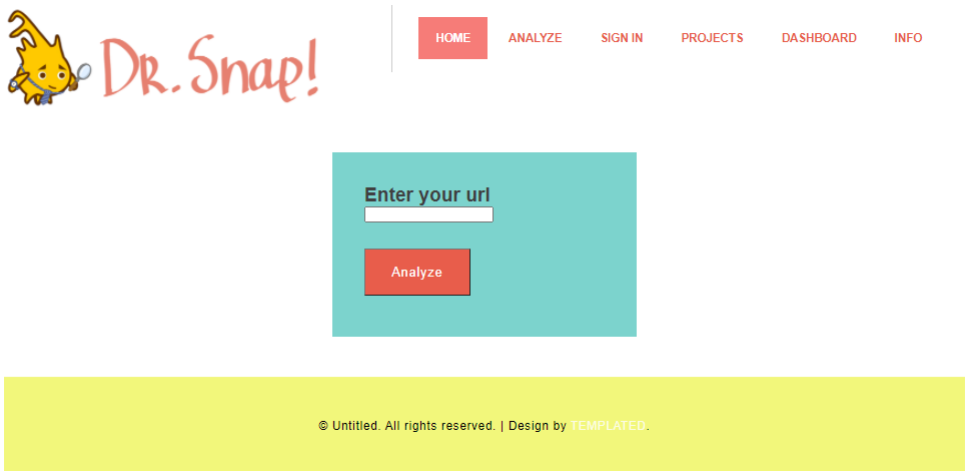


Figura A.1: analyze-estudiantes.html

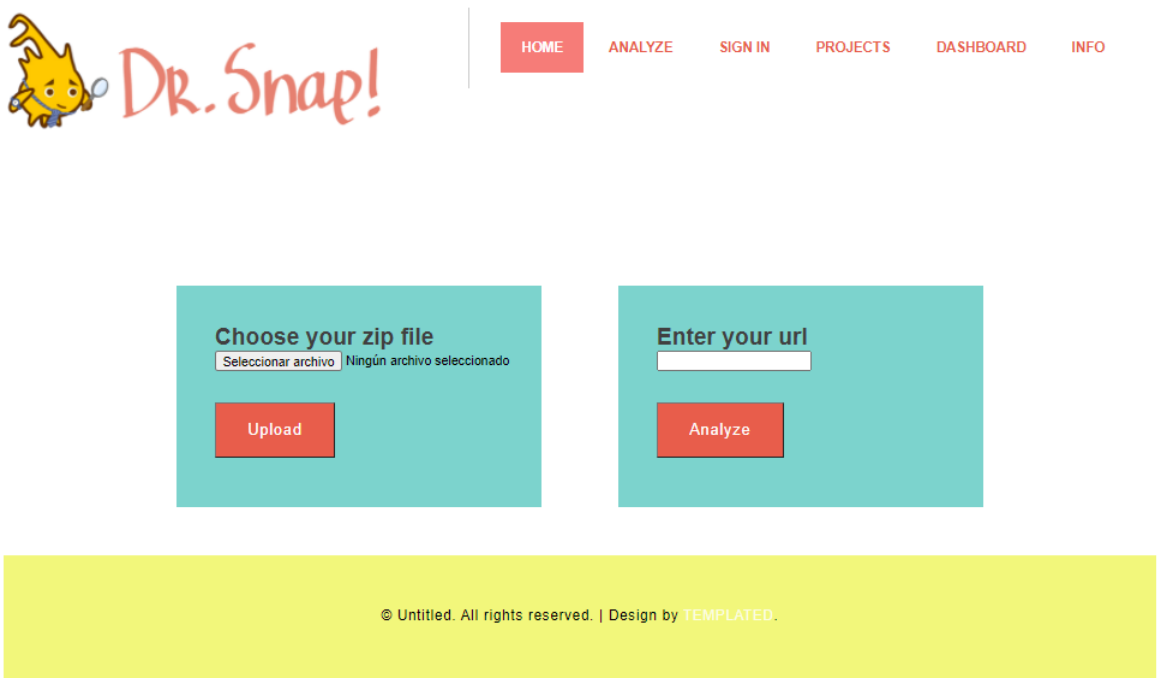


Figura A.2: simple-upload.html

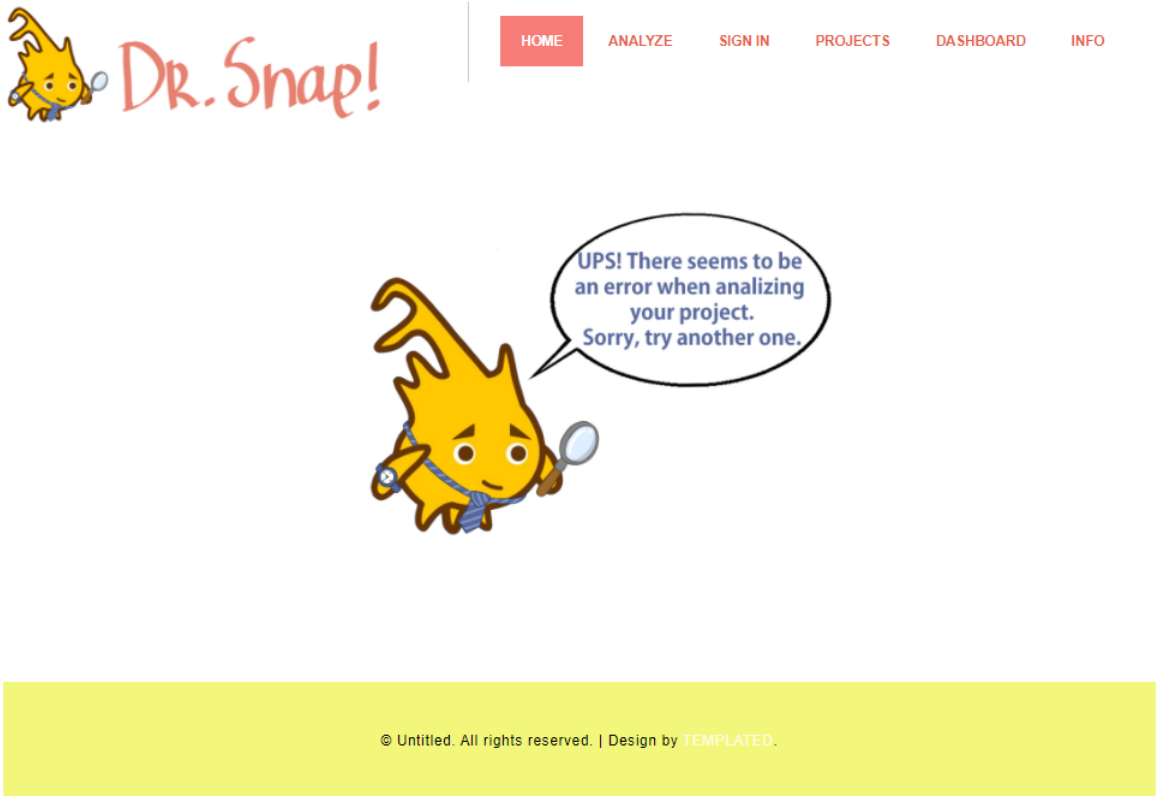


Figura A.3: error_analizar.html

Figura A.4: error_analizar_zip.html

Figura A.5: result.html para un proyecto

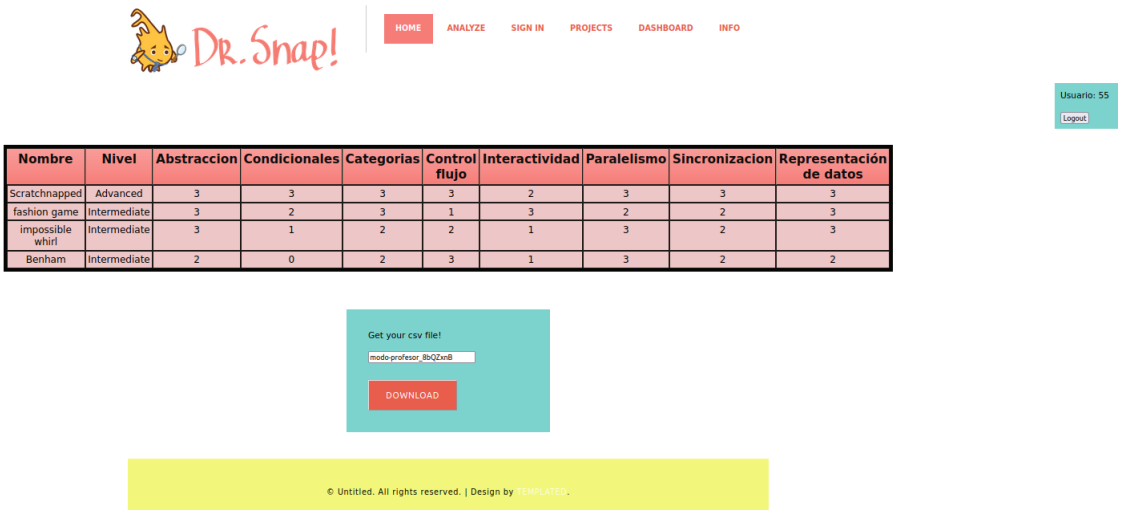


Figura A.6: result.html para un ZIP

A.2. Principal

Vista que proporciona la página principal de la aplicación web.

Figura A.7: home.html

A.3. Login_user

Vista que proporciona un formulario para iniciar sesión.

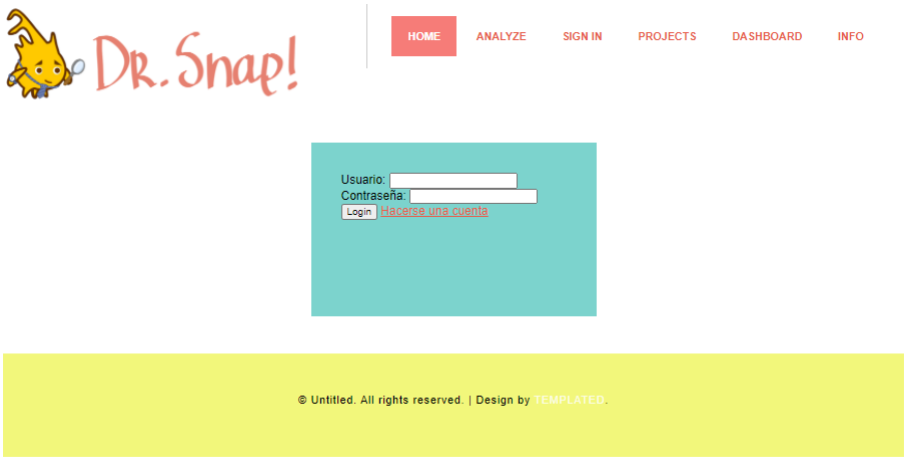


Figura A.8: login.html

A.4. Choose

Vista que permite elegir si eres alumno o profesor.

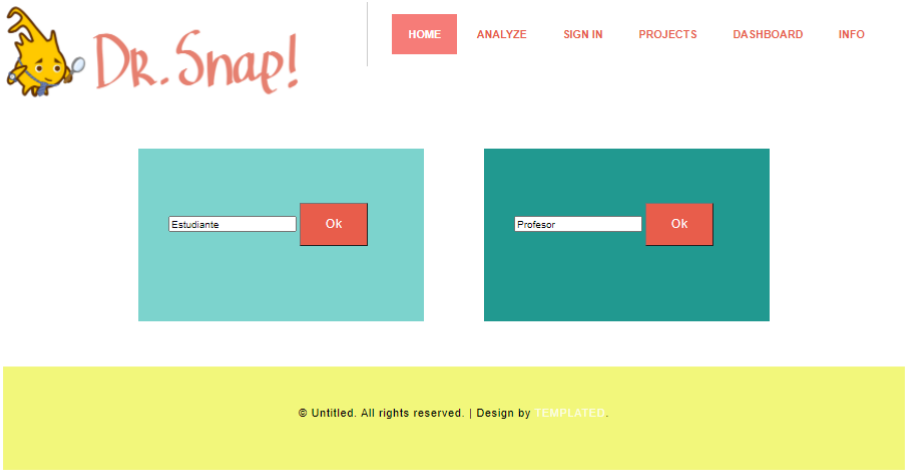


Figura A.9: opcion_tipo_user.html

A.5. Signup

Vista que proporciona un formulario para crearse una cuenta.

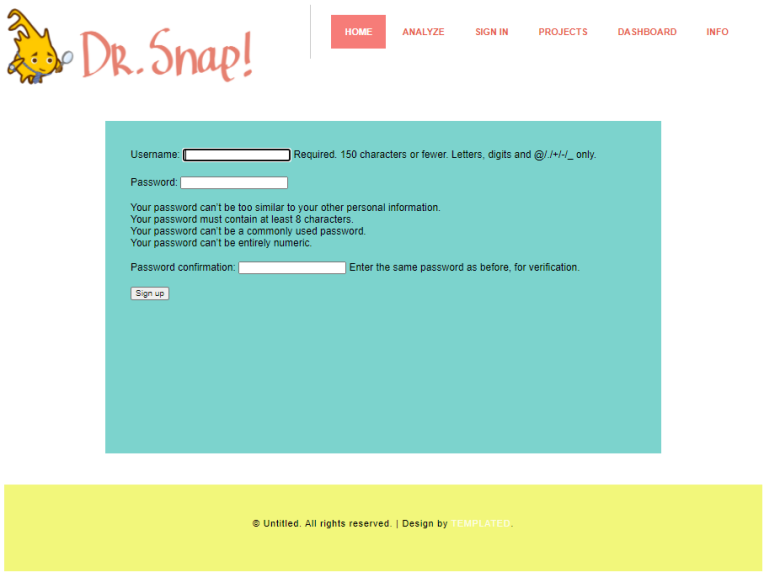


Figura A.10: signup.html

A.6. Show_projects

Vista que permite ver los proyectos o ZIPs que se hayan analizado y guardado en la base de datos. En la figura A.12 se muestra la plantilla para profesores, sin embargo para estudiantes es la misma pero sin el formulario de obtener el CSV.

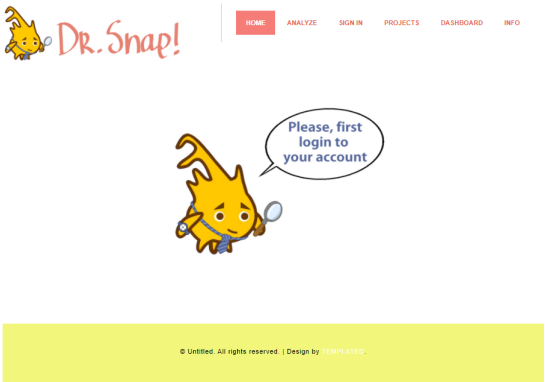


Figura A.11: please-login.html

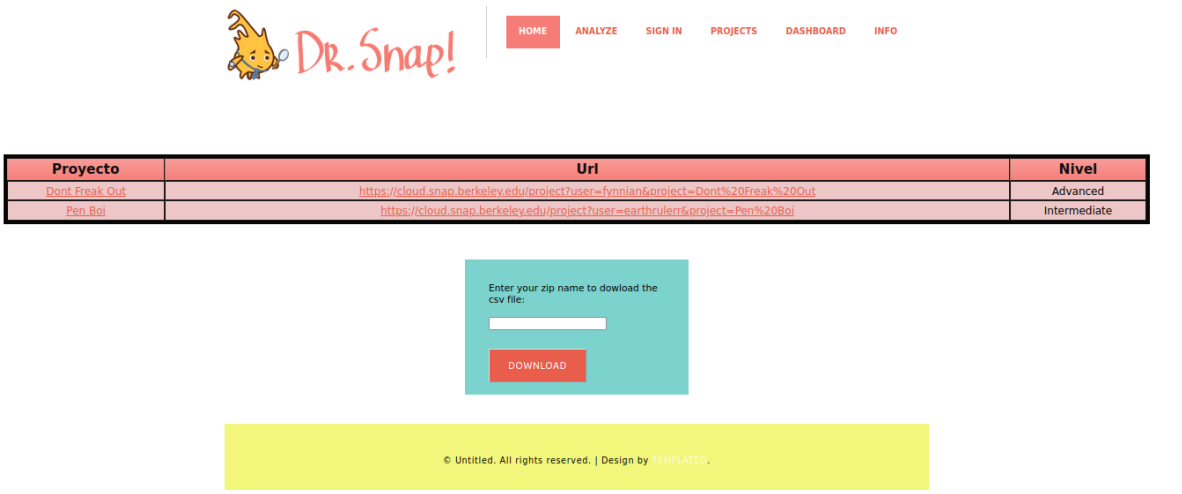


Figura A.12: show_projects.html para profesores

A.7. Show_project

Vista que proporciona las puntuaciones del proyecto ya analizado y guardado. Utiliza la plantilla ya vista en la figura A.5.

A.8. Dashboard

Vista que crea dos dashboard a partir de los datos de los proyectos analizados previamente por el usuario.

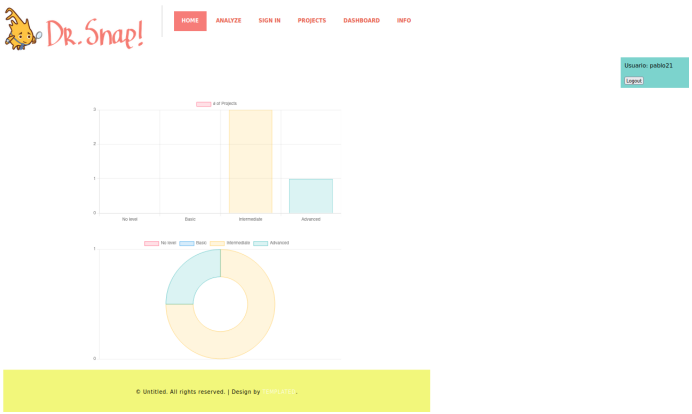


Figura A.13: dashboard.html para estudiantes

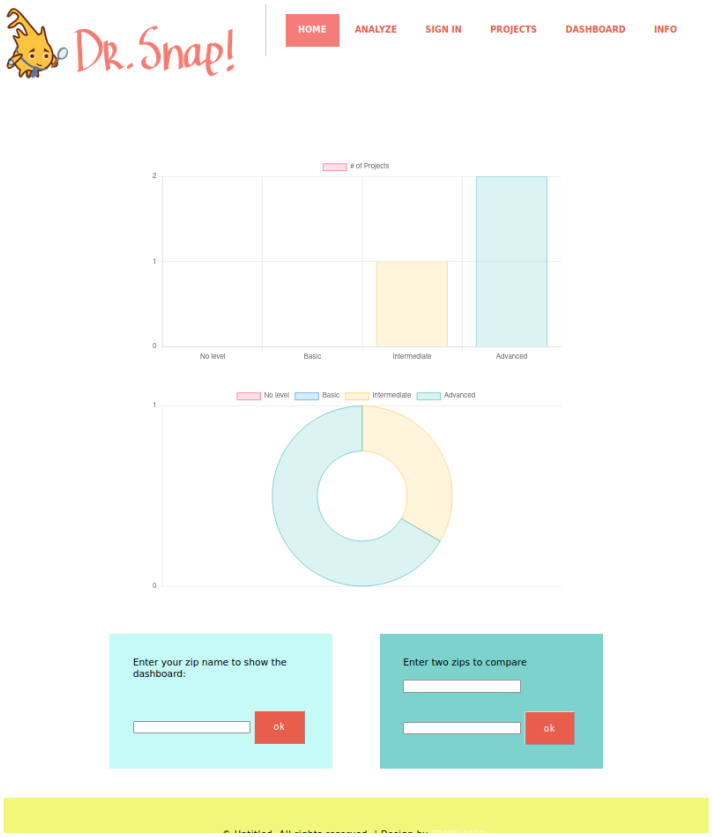


Figura A.14: dashboard.html para profesores

Figura A.15: dashboard_comparing.html

A.9. Dashboard_level

Vista que permite mostrar los proyectos del nivel del que has hecho click.

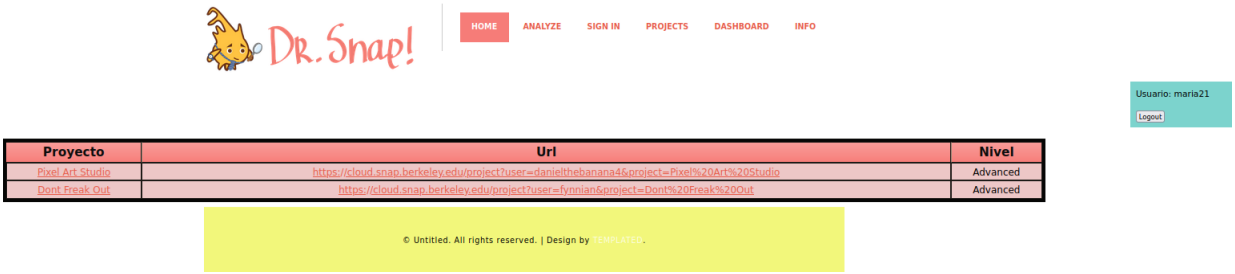


Figura A.16: dashboard_level.html

Apéndice B

Diagramas de flujo

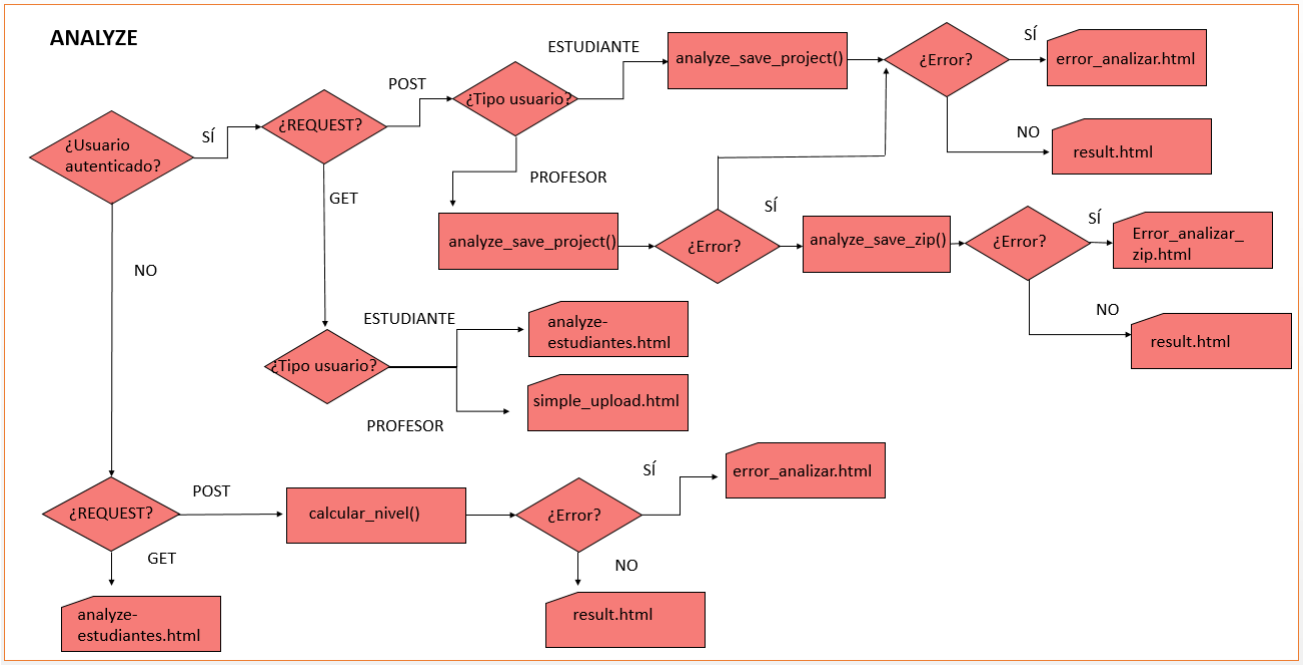


Figura B.1: Diagrama de flujo de la vista *analyze*.

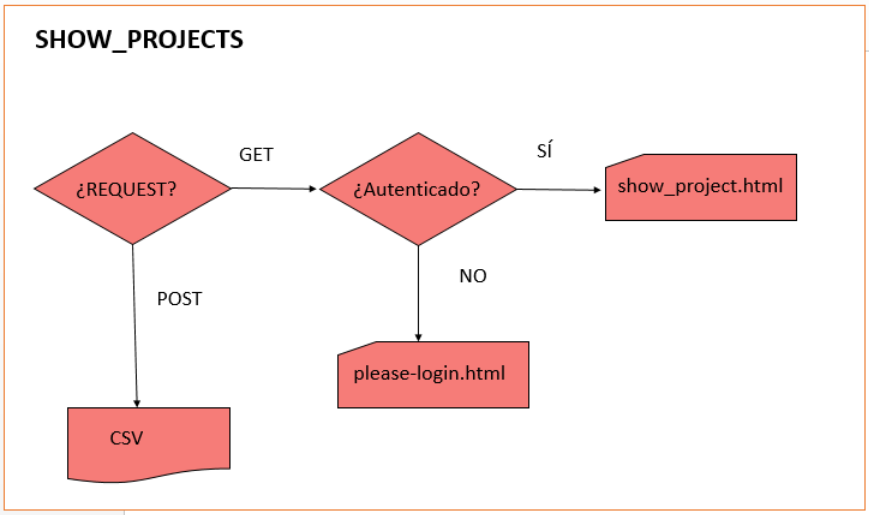


Figura B.2: Diagrama de flujo de la vista *show_projects*.

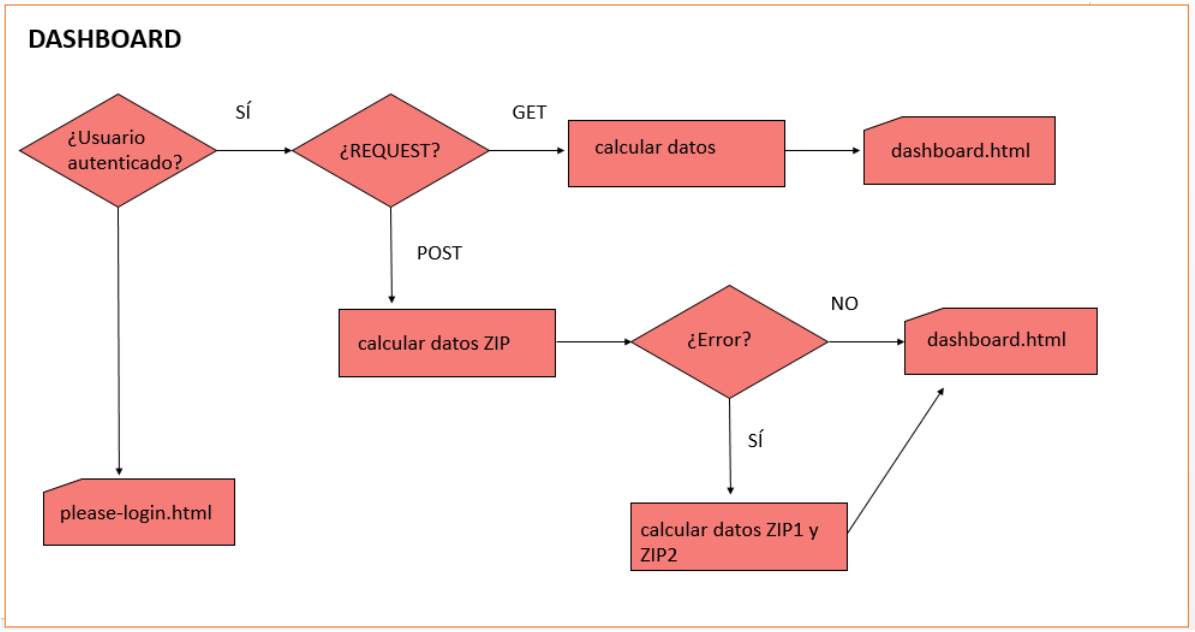


Figura B.3: Diagrama de flujo de la vista *dashboard*.

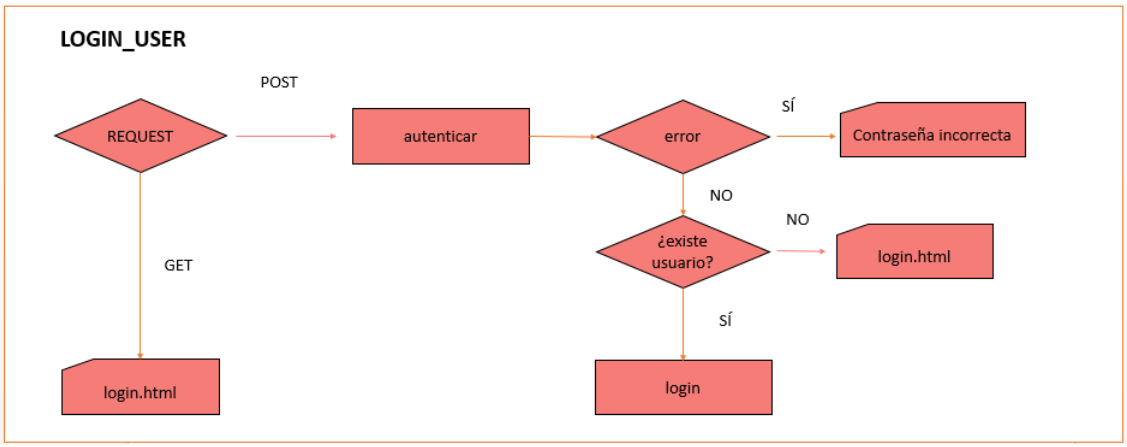


Figura B.4: Diagrama de flujo de la vista *login_user*.

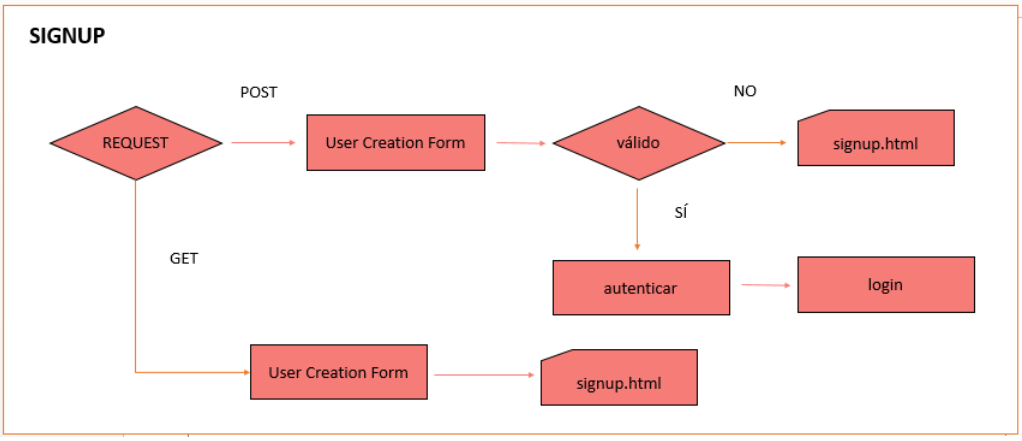


Figura B.5: Diagrama de flujo de la vista *signup*.

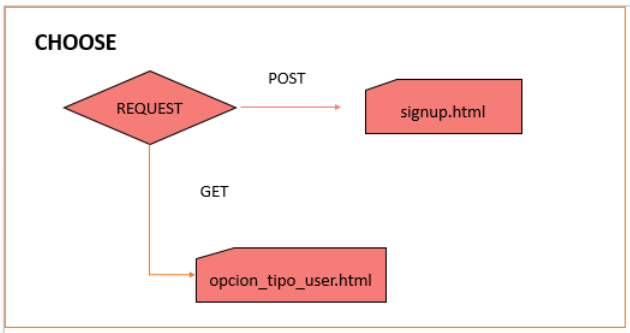


Figura B.6: Diagrama de flujo de la vista *choose*.

Apéndice C

PythonAnywhere

Al estar la aplicación alojada en el servidor que nos provee *pythonAnywhere* podemos ver el tráfico de la página web y su configuración como se observa en la figura C.1.

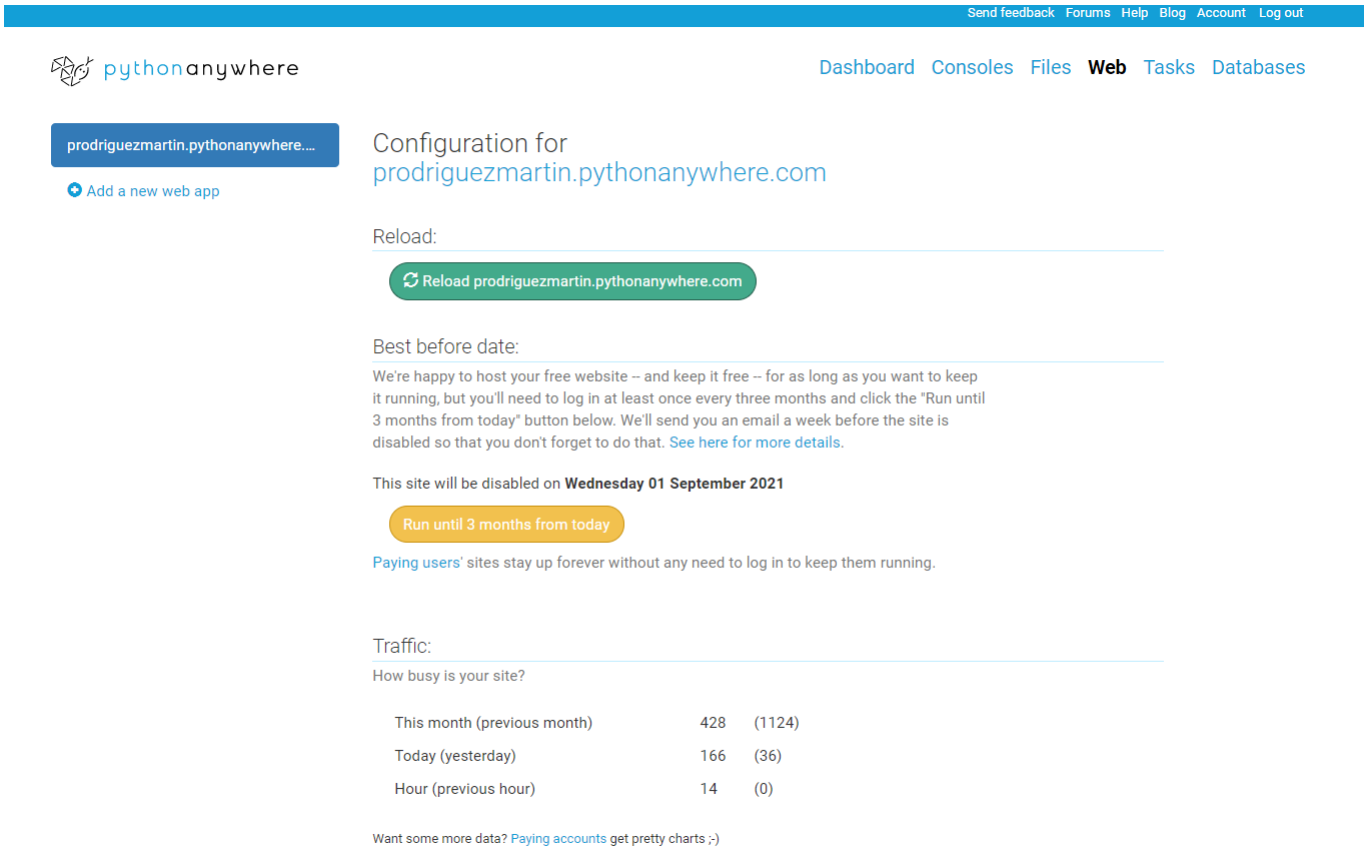


Figura C.1: Página de configuración de la aplicación de la web en *pythonAnywhere*

Bibliografía

[1] Css.

[2] Home about documentation download license support purchase search common links features when to use sqlite getting started prior releases sql syntax pragmas sql functions date & time functions aggregate functions window functions math functions json functions c/c interface spec introduction list of c-language apis the tcl interface spec quirks and gotchas frequently asked questions commit history bugs news what is sqlite?

[3] Javascript.

[4] J. C. Bazurto, A. Florencia, A. Rojas, J. Solórzano, et al. Arquitectura de aplicaciones distribuidas. *Ciencia Huasteca Boletín Científico de la Escuela Superior de Huejutla*, 6(11), 2018.

[5] J. Bennett. *Practical Django Projects*. Apress, 2009.

[6] L. Castro. *Arquitectura del Software*. Cengage Learning Editores, 2015.

[7] Y. D. González and Y. F. Romero. Patrón modelo-vista-controlador. *Telemática*, 11(1):47–57, 2012.

[8] R. González Duque. Python para todos, 2014.

[9] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley. Snap!(build your own blocks). In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 759–759, 2013.

[10] B. Harvey and J. Mönig. Snap! reference manual. *URL http://snap. berkeley. edu/SnapManual. pdf*, 2017.

[11] O. Lizama, G. Kindley, J. I. J. Morales, and A. Gonzales. Redes de computadores: Arquitectura cliente-servidor. *Universidad Tecnica Federico Santa Maria*, pages 1–8, 2016.

[12] J. M. S. López and R. C. Gutiérrez. Pensamiento computacional y programación visual por bloques en el aula de primaria. *Educar*, 53(1):129–146, 2017.

[13] E. Marini. El modelo cliente/servidor. *Recuperado el*, 5, 2012.

[14] J. Moreno-León and G. Robles. Dr. scratch: A web tool to automatically evaluate scratch projects. In *Proceedings of the workshop in primary and secondary computing education*, pages 132–133, 2015.

[15] J. E. Pérez. introduccion a javascript, 2019.

[16] A. Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33:18, 2008.

[17] O. Šimko. Ludificación: crear comunidades de aprendizaje atractivas. *EAD Educación de adultos y desarrollo*, 2014.

[18] E. Vértice. *Diseño básico de páginas web en HTML*. Editorial Vértice, 2009.

[19] J. Wing. Research notebook: Computational thinking—what and why. *The link magazine*, 6, 2011.