Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Desarrollo de un sistema para la alimentación automática de aves rapaces

Autor:
D. David de la Varga del Caño

Tutores:
Dr. Diego R. Llanos Ferraris
D. Guillermo Vicente Oliva

Resumen

El siguiente Trabajo Fin de Grado consiste en el desarrollo de un prototipo para la alimentación automática de aves rapaces.
El propósito que se plantea: el dueño de un ave rapaz será capaz de alimentar a su animal desde cualquier lugar, para lo que es necesario un sistema que sirva de control y el prototipo que será controlado, el prototipo será fabricado. De este modo el cetrero que así es como se denomina al dueño de un ave rapaz, pueda utilizar tanto el sistema de control, a través de una página web, como el comedero físico para mantener atendidas las necesidades alimenticias de su ave.
Se cubrirán las necesidades de alimentar al animal en un instante previamente establecido o en un momento decidido por el cetrero. Además, se mantendrá la información más relevante de cada comedero.

Abstract

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Contexto y motivación

Este proyecto surge debido a la necesidad de atención diaria de todas las aves de cetrería y el control exhaustivo de su alimentación. El proyecto consiste en la realización de un prototipo de comedero automático para aves rapaces. Dicho proyecto constará de dos partes fundamentales, la aplicación web y el comedero, como objeto físico. Mediante el sistema que se propone construir, se podría gestionar de una manera cómoda la alimentación de las aves de cetrería, permitiendo así, de forma remota, racionar adecuadamente la comida necesaria para el ave. Al proyecto se le ha decidido llamar *HungryFalconry*, cetrería hambrienta, refiriéndose a la necesidad que pretende cubrir.

1.2. Objetivos y alcance

El alcance del proyecto sería conseguir un prototipo completamente funcional con, al menos, las siguientes funcionalidades cubiertas:

1. Mantener una información completa del estado del comedero.
2. Servicio de una ración de comida en el momento deseado.
3. Programación del servicio de comida para una fecha y una hora concretas y previamente establecidas.

El segundo punto se verá limitado por la cantidad de raciones que podrá almacenar el comedero. Con ración se refiere a la cantidad de comida que se le ofrecerá al animal de una vez, es decir, si se ejecuta la orden de servir comida, se servirá lo correspondiente a una ración. La cantidad de comida contenida en una ración será responsabilidad del usuario.
El prototipo que se llevará a cabo constará de una ración. La idea es que sea modular, es decir, que se puedan poner varios prototipos de una ración adyacentes, cubriendo así varias raciones. No obstante, esto quedará mejor ilustrado en los planos del diseño del comedero físico, pues será diseñado e imprimido en 3D.

1.3. Explicación del escenario

Como el mundo de la cetrería no es algo mayormente conocido, se procede a explicar brevemente en que consiste el cuidado de un ave de presa y así se pretende justificar este proyecto.
Estos animales actúan por el hambre, es decir, ellos en la naturaleza salen a buscar alimento si se sienten hambrientos, si no es así permanecerán posados. Por este hecho y para que las aves obedezcan a su dueño, es necesario saber su sensación de hambre, el cual se controla a través de su peso. Para ello es necesario gestionar de forma precisa lo que comen diariamente.
Un ejemplo, si el dueño del pájaro quisiera salir fuera un fin de semana debería dejar a alguien al cargo de tus animales. Aquí es donde entra la solución que propone este proyecto, podrá controlar a las aves sin necesidad de encontrarse dónde ellas estén.
Ahora que se ha puesto en situación de lo que será el proyecto se procede a detallar los requisitos del mismo.

Capítulo 2

Análisis de requisitos

2.1. Definición de los actores

2.1.1. Administrador

El Administrador es el encargado de dar de alta nuevos Usuarios y los Comederos asociados a estos Usuarios.

2.1.2. Usuario

El Usuario puede utilizar la aplicación web para consultar la información de sus comederos y ejecutar diversas funciones: servir una ración, programar...

2.1.3. Usuario Externo

El Usuario Externo es aquel que no está registrado en el sistema. Él podrá consultar la información del proyecto y las instrucciones de uso.

2.2. Requisitos funcionales

1. La aplicación permitirá al Usuario identificarse mediante usuario y contraseña.
2. La aplicación mostrará todos los comederos que tengas activos, en cada uno podrán verse los datos relativos al mismo: nombre, raciones actuales, raciones máximas y descripción de la programación.
3. La aplicación permitirá al Usuario introducir una nueva cantidad de raciones.
4. La aplicación permitirá al Usuario modificar las raciones previamente introducidas.
5. La aplicación permitirá dar la orden de servir una ración.
6. La aplicación permitirá programar el servicio de una/s ración/es para un día o varios días y para uno o varios comederos, permitiendo elegir la hora para cada uno de los días y para cada comedero.
7. La aplicación permitirá borrar la programación establecida para un comedero.
8. La aplicación permitirá al Usuario editar la programación de cada uno de los comederos.
9. La aplicación permitirá editar el nombre del comedero.

10. La aplicación permitirá al Administrador registrar un nuevo Usuario mediante un nombre de usuario, una contraseña y la MAC asociada a la Raspberry en uso.
11. La aplicación permitirá al Administrador dar de alta nuevos Comederos.
12. La aplicación permitirá al Administrador asignar un número máximo de raciones a un modelo de Comedero.
13. La aplicación permitirá al Administrador visualizar todos los Usuarios y los Comederos que poseen.
14. La aplicación permitirá ver toda la información relativa al proyecto.
15. La aplicación permitirá ver las instrucciones de uso de toda la aplicación

2.3. Requisitos no funcionales

1. La aplicación web se desarrollará en el framework Django basado en Python.
2. La aplicación web será al menos compatible para los navegadores Google Chrome y Mozilla Firefox.
3. La aplicación web se albergará en un servidor (PythonAnywhere).
4. La Raspberry funcionará sobre Raspbian.
5. La Raspberry actuará sobre un servo para servir la comida.
6. La Raspberry estará embebida en el comedero con el que interactuará.
7. El comedero estará diseñado en freeCAD 0.16. Se adjuntarán los planos.
8. El comedero se imprimirá en 3D.
9. El prototipo del comedero se imprimirá en un plástico denominado PLA (Poliácido láctico).

2.4. Reglas de negocio

1. Las raciones máximas que un comedero puede servir están limitadas por el número de huecos para almacenarlas.
2. El número máximo de programaciones que se pueden establecer para un comedero estará limitado por la cantidad de raciones actuales albergadas por el comedero y nunca será superior al número máximo de raciones posibles.
3. Si existen raciones previas registradas en el comedero se advertirá antes de añadir más raciones.
4. La programación más temprana que se puede establecer para un comedero es, como máximo, al día siguiente del que se encuentra.

2.5. Requisitos de información

1. La cantidad de comida por ración depende del usuario que utilice la aplicación. Típicamente el cetrero sabrá la cantidad de comida adecuada que contendrá cada ración.
2. Es obligación del usuario de la aplicación retirar las raciones que perezcan o no utilizará.

Una vez conocidos los requisitos del sistema a elaborar se procede a planificar el proyecto.

Capítulo 3

Plan de proyecto

3.1. Propósito, alcance y objetivos

Como se ha introducido anteriormente el propósito de este proyecto es cubrir la necesidad de alimentar remotamente a las aves de cetrería, para así mantener sus cuidados aunque el dueño no se encuentre en el lugar donde las aves habitan. Se podrá servir a decisión el alimento, o bien, programar un servicio. También se mantendrá informado del estado de cada comedero. Todo esto accesible desde la interfaz web de *Hungry Falconry*.

3.2. Artefactos de proyecto

El conjunto de artefactos RUP está compuesto del siguiente modo:

- Modelo de negocio
- Requisitos
- Análisis
- Diseño
- Implementación
- Pruebas
- Despliegue

3.3. Plan de trabajo

A continuación se realizará una estimación del esfuerzo y duración de las diferentes etapas que lo compondrán, siguiendo un modelo en cascada. Para llevar a cabo este proyecto es necesario tener conocimientos de los siguientes ámbitos:

- Conocimientos de Python
- Conocimientos de HTML, CSS junto con Bootstrap
- Conocimientos sobre el Framework Django
- Conocimientos en SQL, específicamente SQLite
- Conocimientos de control de servomotores mediante Python y pines GPIO en Raspberry Pi
- Conocimientos de diseño 3D con FreeCAD 0.16
- Conocimientos en impresión 3D

Fase de Inicio

1. Inicio de la fase de inicio
   - Predecesoras:
   - Duración:
2. Elicitación de requisitos
   - Predecesoras: 1
   - Duración: 5 días
   - Se obtiene los requisitos para alcanzar los objetivos que se proponen. Además de definir con que herramientas se llevarán a cabo.
3. Calendario de tareas
   - Predecesoras: 2
   - Duración: 1 días
   - Se distribuyen las tareas a lo largo del periodo de desarrollo del proyecto.
4. Adquisición de conocimientos en Django
   - Predecesoras: 3
   - Duración: 12 días
   - En este periodo se espera adquirir los conocimientos suficientes para trabajar con dicho framework.
5. Control de versiones
   - Predecesoras: 3
   - Duración: 1 días
   - Se establecerá un sistema de control de versiones git, en GitHub.
6. Fin de la fase de inicio
   - Predecesoras: 4, 5
   - Duración:

Fase de elaboración

7. Inicio de la fase de elaboración
   - Predecesoras: 6
   - Duración:
8. Casos de Uso
   - Predecesoras: 7
   - Duración: 5 días
   - Se identificarán y describirán los casos de uso del sistema a partir de los requisitos.
9. Modelo de dominio
   - Predecesoras: 8

- Duración: 1 días
- Realización del modelo del dominio. Identificar las clases implicadas.

10. Modelo de la base de datos

    - Predecesoras: 9
    - Duración: 1 días
    - Realización del modelo de datos. Una versión simplificada de lo que será la base de datos.

11. Diseño en 3D del prototipo

    - Predecesoras: 7
    - Duración: 9 días
    - Se diseñará el prototipo con FreeCAD. Software de diseño en 3D.

12. Boceto de la interfaz de usuario

    - Predecesoras: 9
    - Duración: 1 días
    - Se realizarán unos bocetos de la interfaz de usuario que servirán como referencia.

13. Diseño de la arquitectura del sistema

    - Predecesoras: 10, 12
    - Duración: 2 días
    - La mayoría de la arquitectura del sistema viene determinada por el framework.

14. Diseño de la base de datos

    - Predecesoras: 13
    - Duración: 2 días
    - Se construirá el esquema final de la base de datos. Su creación va ligada a los denominados modelos del framework.

15. Diagramas de secuencia en diseño

    - Predecesoras: 13, 14
    - Duración: 5 días
    - Realización de los diagramas de secuencia de los casos de uso de mayor relevancia.

16. Fin de la fase de elaboración

    - Predecesoras: 11, 15
    - Duración:

Fase de construcción

17. Inicio de la fase de construcción

    - Predecesoras: 16
    - Duración:

18. Configuración del entorno de desarrollo
    - Predecesoras: 17
    - Duración: 1 día
    - Se configurará una base de datos de prueba y se instalará lo correspondiente al framework.
19. Desarrollo del backend
    - Predecesoras: 18
    - Duración: 20 día
    - Se desarrollará la gestión de usuarios y funcionalidades. Así como el API REST con ¡. El script cliente con Python.
20. Desarrollo del frontend
    - Predecesoras: 18
    - Duración: 10 día
    - Se realizarán todas las vistas del sistema, la capa de presentación, con Bootstrap, HTML y CSS.
21. Impresión 3D del prototipo
    - Predecesoras: 18
    - Duración: 5 día
    - Se imprimirá el prototipo en 3D. La duración viene determinada por el tiempo invertido por la impresora.
22. Fin de la fase de construcción
    - Predecesoras: 19, 20, 21
    - Duración:

Fase de Transición

23. Inicio de la fase de transición
    - Predecesoras: 22
    - Duración:
24. Pruebas
    - Predecesoras: 23
    - Duración: 8 días
25. Elaborar manual del programador
    - Predecesoras: 24
    - Duración: 3 días
26. Elaborar manual del usuario
    - Predecesoras: 24
    - Duración: 2 días
27. Fin de la fase de transición
    - Predecesoras: 25, 26
    - Duración:

3.3.1. Diagramas de Gantt

A continuación se muestra los diagramas de Gantt de la planificación del proyecto, es una aproximación de la calendarización. En el **Anexo I - Diagrama de Gantt Completo** se muestra la planificación completa.

Fase de Inicio

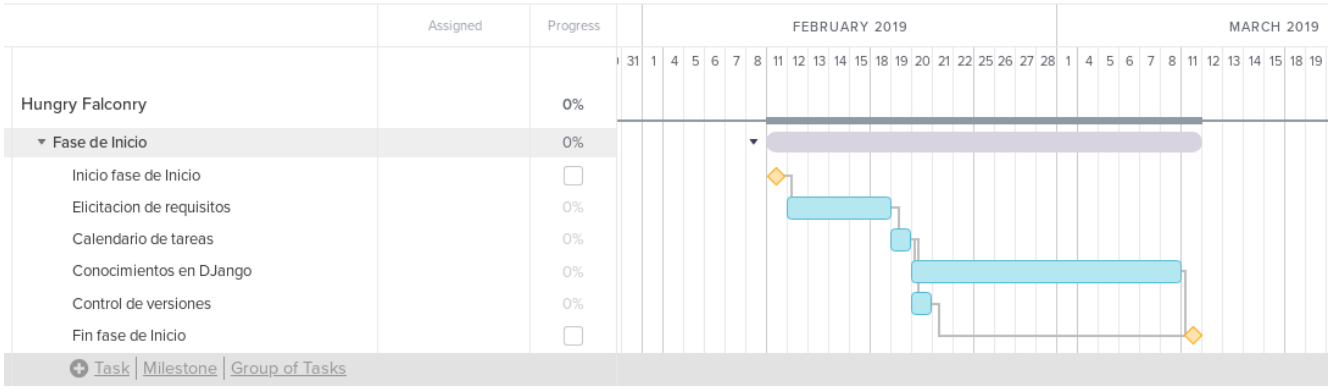


Figura 3.1: Diagrama de Gantt para la Fase de Inicio

Fase de Elaboración

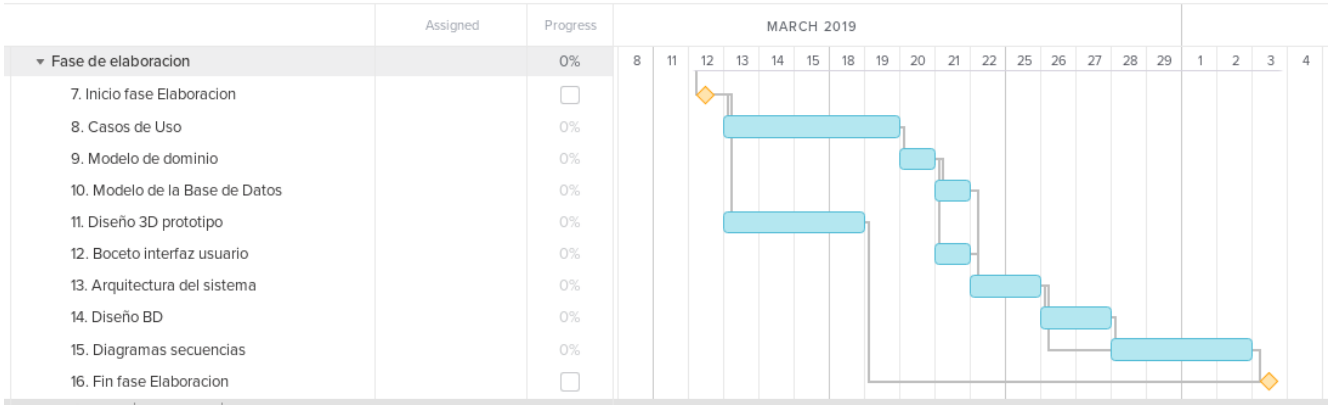


Figura 3.2: Diagrama de Gantt para la Fase de Elaboración

Fase de Construcción

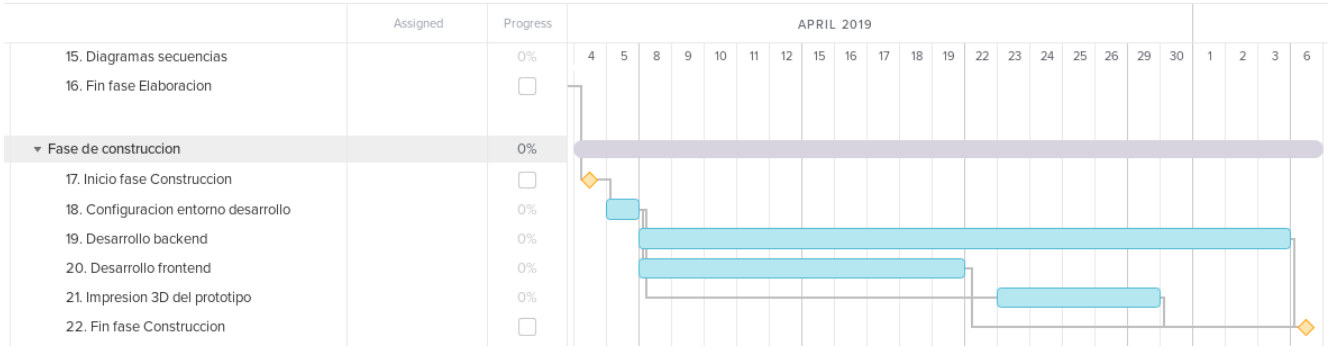


Figura 3.3: Diagrama de Gantt para la Fase de Construcción

Fase de Transición

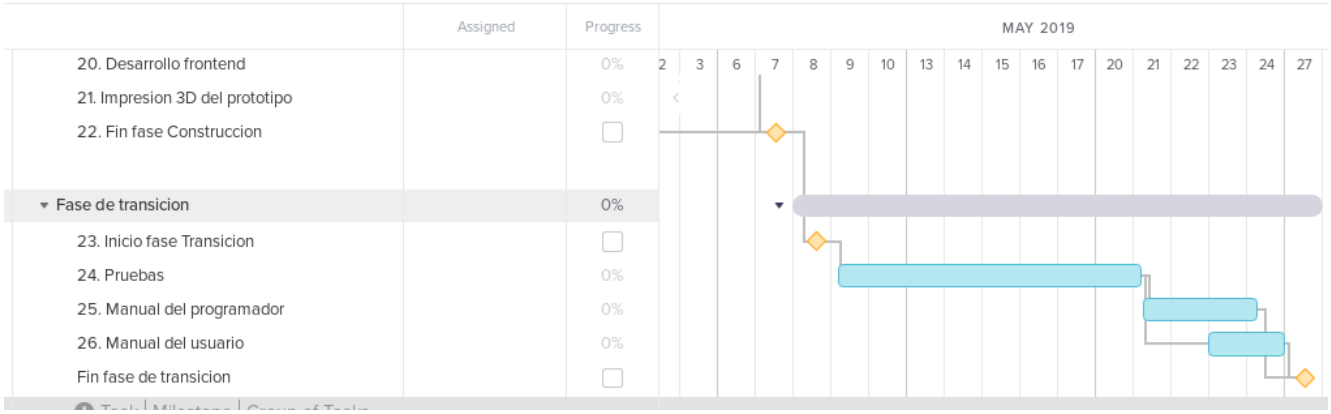
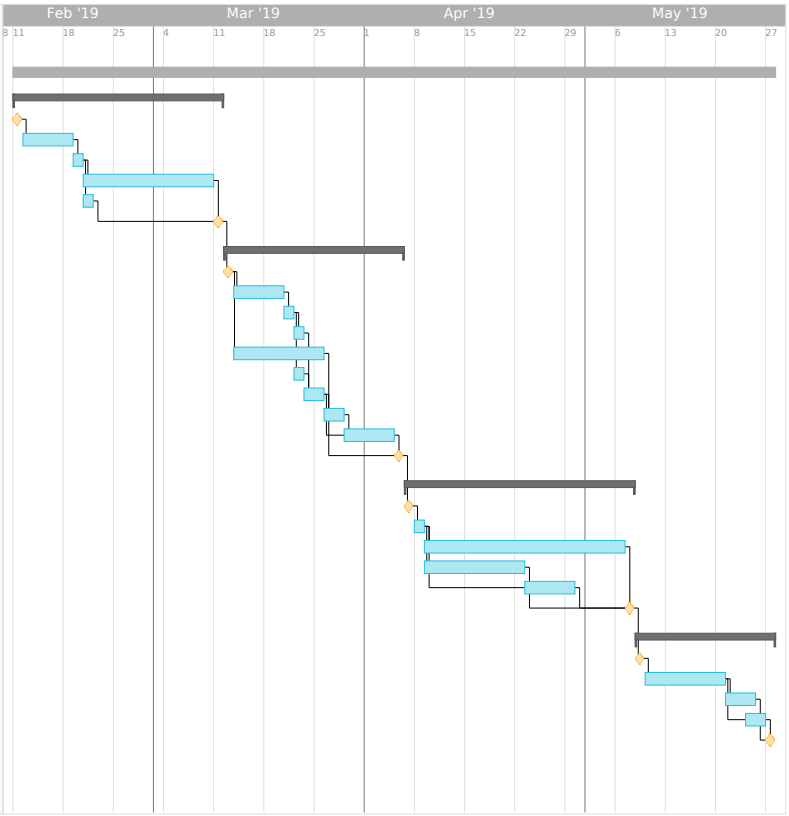


Figura 3.4: Diagrama de Gantt para la Fase de Transición

Diagrama de Gantt completo

teamgantt
Created with Free Edition



Hungry Falconry	**0h**	**0%**
Fase de Inicio	**0h**	**0%**
Inicio fase de Inicio	0	0%
Elicitacion de requisitos	0	0%
Calendario de tareas	0	0%
Conocimientos en DJango	0	0%
Control de versiones	0	0%
Fin fase de Inicio	0	0%
Fase de elaboracion	**0h**	**0%**
7. Inicio fase Elaboracion	0	0%
8. Casos de Uso	0	0%
9. Modelo de dominio	0	0%
10. Modelo de la Base de Datos	0	0%
11. Diseño 3D prototipo	0	0%
12. Boceto interfaz usuario	0	0%
13. Arquitectura del sistema	0	0%
14. Diseño BD	0	0%
15. Diagramas secuencias	0	0%
16. Fin fase Elaboracion	0	0%
Fase de construccion	**0h**	**0%**
17. Inicio fase Construccion	0	0%
18. Configuracion entorno desarrollo	0	0%
19. Desarrollo backend	0	0%
20. Desarrollo frontend	0	0%
21. Impresion 3D del prototipo	0	0%
22. Fin fase Construccion	0	0%
Fase de transicion	**0h**	**0%**
23. Inicio fase Transicion	0	0%
24. Pruebas	0	0%
25. Manual del programador	0	0%
26. Manual del usuario	0	0%
Fin fase de transicion	0	0%

3.4. Presupuesto

En este proyecto hay que tener en cuenta dos factores a la hora del presupuesto, el proyecto software y el proyecto IoT.

3.4.1. Proyecto software

El proyecto se desarrollará en 4 meses. Dado que el sueldo medio de un Ingeniero Informático en España es de 25.600 euros, los 4 meses son 8.534 euros. Más 2.816 euros de gastos sociales para la empresa.

3.4.2. Proyecto IoT

El salario medio de un diseñador Industrial en España es de 25.600 euros. Esto resulta que para los 4 meses hace un total de 8.534 euros. Más 2.816 euros de gastos sociales para la empresa.

3.4.3. Total

A las cantidades anteriores hay que sumar el gasto en la compra de hardware (120 euros), el coste del material de impresión (25 euros) y la amortización del desgaste de la impresora (10 euros). El total resultante es: 8.534 + 2.816 + 8.534 + 2.816 + 120 + 25 + 10 = **22.855 euros**.

Después de haber visto la planificación del proyecto se procederá a la fase de análisis.

Capítulo 4

Modelo de Análisis

Esta sección se dedicará al análisis de la aplicación, utilizando los requisitos anteriores como punto de partida.
Se tratarán:

- Casos de Uso. Diagrama UML y descripción.
- Diagramas UML de secuencia de los Casos de Uso.
- Diagrama UML del modelo del dominio.

4.1. Casos de uso del sistema

Ahora se estudiarán los casos de uso del sistema. Están descritos como una secuencia de interacciones actor-sistema. Con los casos de uso se define el sistema y su entorno. Además se profundiza en los requisitos funcionales y servirán como base para las posteriores pruebas del sistema.

4.1.1. Diagrama de Casos de Uso

Figura 4.1: Diagrama de Casos de uso

4.1.2. Casos de Uso del actor Administrador

Registrar Usuario

CU-1	**Registrar Usuario**
Actor	**Administrador**
Descripción	El sistema deberá permitir al administrador registrar nuevos Usuarios en el sistema.
Precondición	El actor ha iniciado sesión en el sistema.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción registrar usuario.
2	El sistema muestra un formulario con al menos las entradas: nombre, contraseña y dirección MAC de la Raspberry.
3	El actor completa los campos adecuadamente y acepta registrar un nuevo usuario.
4	El sistema muestra un mensaje de registro correcto.
Postcondición	El nuevo usuario queda registrado en la base de datos.
Excepciones	
Variación	**Acción**
3a	El actor selecciona la opción cancelar y el caso de uso queda sin efecto.
4a	Ocurre algún error en el proceso, se muestra un mensaje de error y el caso de uso continúa en el paso 3.

Tabla 4.1: Caso de Uso 1 - Registrar Usuario

4.1.3. Casos de Uso del actor Usuario

Servir una ración

CU-2	**Servir una ración**
Actor	**Usuario**
Descripción	El sistema deberá permitir servir una ración de un determinado comedero en el momento en el que se desee.
Precondición	El actor ha iniciado sesión en el sistema y hay raciones en el comedero.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción servir una ración, del menú de cualquiera de los comederos.
2	El sistema muestra un mensaje de confirmación
3	El actor selecciona la opción Sí.
4	El sistema manda un mensaje al servidor para realizar dicha acción y se muestra una mensaje de que todo ha funcionado correctamente.
Postcondición	Se ha servido una nueva ración del comedero correctamente.
Excepciones	
Variación	**Acción**
3a	El actor selecciona la opción No y el caso de uso queda sin efecto.
4a	Ocurre algún error en el proceso, se muestra un mensaje de error y el caso de uso continúa en el paso 3.

Tabla 4.2: Caso de Uso 2 - Servir una ración

Programar un comedero

CU-3	**Programar un comedero**
Actor	**Usuario**
Descripción	El sistema deberá permitir programar el servicio de una ración o varias, para un día o varios, de uno o múltiples comederos.
Precondición	El actor ha iniciado sesión en el sistema, el/los comedero/s tiene/n raciones y no tiene/n una programación previa.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción programar comederos, en el menú de un comedero.
2	El sistema muestra el menú de programación de comederos.
3	El actor introduce el día y la hora y pulsa guardar y finalizar programación.
4	El sistema muestra un mensaje de éxito y el caso de uso finaliza.
Postcondición	Se ha añadido una nueva programación correctamente para el comedero seleccionado.
Excepciones	
Variación	**Acción**
3a	El actor selecciona la opción Cancelar y el caso de uso queda sin efecto.
3b	El actor selecciona la opción Aceptar, la programación se guarda y el caso de uso continúa en el paso 2.
4a	Ocurre algún error en el proceso, se muestra un mensaje de error y el caso de uso continúa en el paso 3.

Tabla 4.3: Caso de Uso 3 - Programar un comedero

Editar Programación

CU-4	**Editar programación**
Actor	**Usuario**
Descripción	El sistema deberá permitir editar la programación de un comedero.
Precondición	El actor ha iniciado sesión en el sistema, el comedero tiene raciones y una programación previa.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción editar programación.
2	El sistema muestra el menú de editar programación de un comedero.
3	Se ejecuta el caso de uso «Programar un comedero»
4	El sistema muestra un mensaje de éxito y el caso de uso finaliza.
Postcondición	
Excepciones	
Variación	**Acción**

Tabla 4.4: Caso de Uso 4 - Editar programación

Borrar programación

CU-5	**Borrar programación**
Actor	**Usuario**
Descripción	El sistema deberá permitir borrar la programación de un comedero.
Precondición	El actor ha iniciado sesión en el sistema, el comedero tiene una programación previa.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción borrar programación.
2	El sistema muestra un mensaje de confirmación.
3	Se actor selecciona Si.
4	El sistema muestra un mensaje de éxito y el caso de uso finaliza.
Postcondición	Se ha borrado la programación del comedero.
Excepciones	
Variación	**Acción**
3a	El actor selecciona No. El caso de uso finaliza y queda sin efecto.

Tabla 4.5: Caso de Uso 5 - Borrar programación

Registrar el número de raciones

CU-6	**Registrar el número de raciones**
Actor	**Usuario**
Descripción	El sistema deberá permitir registrar el número de raciones que previamente se han introducido en el comedero.
Precondición	El actor ha introducido las raciones en forma de alimento en el comedero.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción nuevas raciones.
2	El sistema muestra el menú de añadir raciones.
3	El actor registra el número de raciones y pulsa aceptar.
4	El sistema muestra un mensaje de éxito y el caso de uso finaliza.
Postcondición	Se ha registrado la cantidad de raciones que ahora contiene el comedero.
Excepciones	
Variación	**Acción**
1a	El actor selecciona añadir más raciones.
2a	El comedero tenia raciones ya introducidas, entonces el sistema advierte de que se eliminaran las anteriores raciones.
3a	El actor introduce un número superior a la cantidad de raciones máximas permitidas por el comedero, el sistema muestra un error y el caso de uso continúa en el paso 2.
3b	El actor introduce 0 o menos raciones, el sistema muestra un error y el caso de uso continúa en el paso 2.
3c	El actor pulsa Atrás y el caso de uso queda sin efecto.
4a	Ocurre algún error en el proceso, se muestra un mensaje de error y el caso de uso continúa en el paso 3.

Tabla 4.6: Caso de Uso 6 - Registrar el número de raciones

Editar número de raciones

CU-7	**Editar número de raciones**
Actor	**Usuario**
Descripción	El sistema deberá permitir editar el número de raciones que previamente se han introducido en el comedero.
Precondición	Se ha ejecutado el Caso de Uso «Registrar el número de raciones de un comedero».
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción editar raciones.
2	El sistema muestra el menú de editar raciones y la cantidad anterior de raciones.
3	El actor pulsa la opción añadir ración.
4	El sistema aumenta en 1 el número de raciones por cada pulsación.
5	El actor pulsa Aceptar.
6	El sistema muestra un mensaje de éxito y el Caso de Uso finaliza.
Postcondición	Se ha editado la cantidad de raciones que ahora contiene el comedero.
Excepciones	
Variación	**Acción**
2a	El comedero no tenía raciones previamente introducidas, se muestra un mensaje de error y el Caso de Uso finaliza.
3a	El actor pulsa la opción restar ración.
3b	El actor pulsa cancelar y el caso de uso queda sin efecto.
4a	El sistema disminuye en 1 el número de raciones por cada pulsación.
5a	El actor pulsa Cancelar y el Caso de Uso queda sin efecto.
6a	Ocurre algún error en el proceso, se muestra un mensaje de error y el caso de uso continúa en el paso 5.

Tabla 4.7: Caso de Uso 7 - Editar número de raciones

Identificarse

CU-8	**Identificarse**
Actor	**Usuario y Administrador**
Descripción	El sistema permitirá iniciar sesión en él.
Precondición	El usuario debe estar registrado en el sistema.
Secuencia Normal	
Paso	**Acción**
1	El actor entra en el sistema.
2	El sistema muestra la vista de iniciar sesión.
3	El actor introduce el nombre de usuario, contraseña y pulsa Iniciar Sesión.
4	El sistema comprueba los datos introducidos y se ejecuta el caso de uso «Consultar información de todos mis comederos» para el rol de Usuario y la vista correspondiente al Administrador en el caso de que el rol sea Administrador.
Postcondición	El actor ha iniciado sesión en el sistema.
Excepciones	
Variación	**Acción**
3a	El actor pulsa Cancelar y el caso de uso queda sin efecto.
4a	El sistema comprueba que los datos introducidos son erróneos, muestra un mensaje de error y el caso de uso continúa en el paso 2.

Tabla 4.8: Caso de Uso 8 - Identificarse

Consultar información de todos mis comederos

CU-9	**Consultar información de todos mis comederos**
Actor	**Usuario**
Descripción	El sistema mostrará la información de todos los comederos que tenga el Usuario en funcionamiento.
Precondición	Se ha ejecutado el Caso de Uso «Identificarse».
Secuencia Normal	
Paso	**Acción**
1	El sistema muestra una lista con toda la información correspondiente a los comederos que el usuario tiene activos.
Postcondición	
Excepciones	
Variación	**Acción**

Tabla 4.9: Caso de Uso 9 - Consultar información de todos mis comederos

Editar información

CU-10	**Editar información**
Actor	**Usuario**
Descripción	El sistema permitirá editar la información de un comedero tales como el nombre.
Precondición	El actor se ha identificado en el sistema y tiene comederos activos.
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción de editar información de un comedero.
2	El sistema muestra el menú de editar propiedades de un comedero.
3	El actor modifica el nombre. Pulsa Aceptar.
4	El sistema muestra un mensaje de éxito y el caso de uso finaliza.
Postcondición	La información editada queda guardada.
Excepciones	
Variación	**Acción**
3a	El actor pulsa Atrás y el caso de uso queda sin efecto.
4a	Ocurre algún error en el proceso, se muestra un mensaje de error y el caso de uso continúa en el paso 3.

Tabla 4.10: Caso de Uso 10 - Editar información

4.1.4. Casos de uso del actor Usuario Externo

Consultar información del proyecto

CU-11	**Consultar información del proyecto**
Actor	**Usuario Externo**
Descripción	El sistema permitirá ver toda la información relevante para el usuario del proyecto
Precondición	
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción de información del proyecto.
2	El sistema muestra toda la información referente al proyecto.
Postcondición	
Excepciones	
Variación	**Acción**

Tabla 4.11: Caso de Uso 11 - Editar Consultar información del proyecto

Consultar instrucciones de uso de la aplicación

CU-12	**Consultar instrucciones de uso de la aplicación**
Actor	**Usuario Externo**
Descripción	El sistema permitirá ver las instrucciones de uso de la aplicación.
Precondición	
Secuencia Normal	
Paso	**Acción**
1	El actor selecciona la opción de instrucciones de uso.
2	El sistema muestra toda la información referente a las instrucciones de uso. Que pasos hay que seguir para hacer cada funcionalidad.
Postcondición	
Excepciones	
Variación	**Acción**

Tabla 4.12: Caso de Uso 12 - Consultar instrucciones de uso de la aplicación

4.2. Modelo de dominio

El siguiente diagrama muestra las clases detectadas junto con sus atributos.

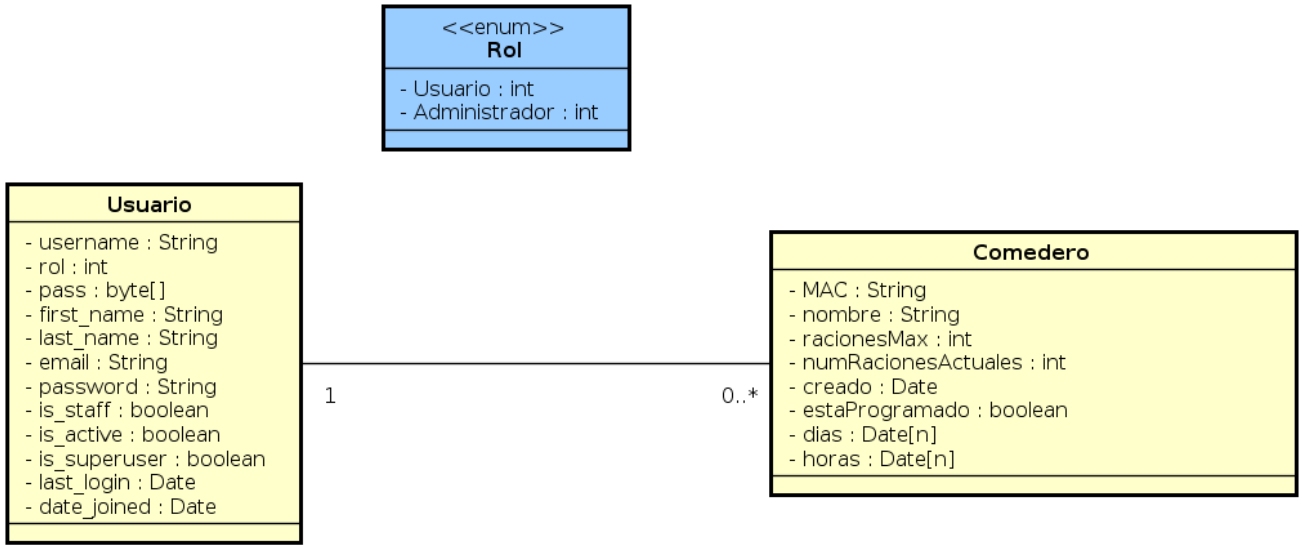


Figura 4.2: Diagrama modelo de dominio en análisis

4.3. Modelo de datos

El siguiente apartado muestra el diagrama entidad relación de la base de datos que se implementará. Los campos *algorithm*, *salt* y *hash* serán explicados en la fase de diseño.

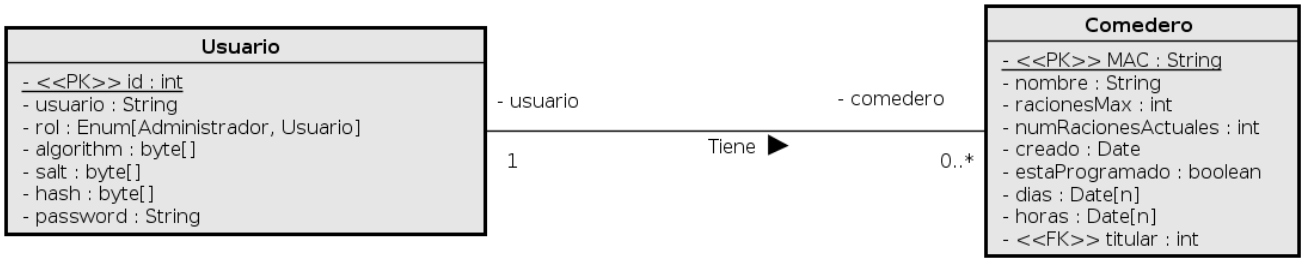


Figura 4.3: Diagrama de la base de datos en análisis

Realizada la fase de análisis se procede a la fase de diseño.

Capítulo 5

Diseño

5.1. Arquitectura del sistema

Como el proyecto se desarrollará con el framework Django, la arquitectura viene definida por él mismo. En esencia Django sigue una arquitectura Modelo-Vista-Controlador (MVC) pero con una salvedad, a las vistas las denominan plantillas (Templates) y a los controladores, vistas (Views). Así que se podría establecer el acrónimo MTV (Model-Template-View) [4].

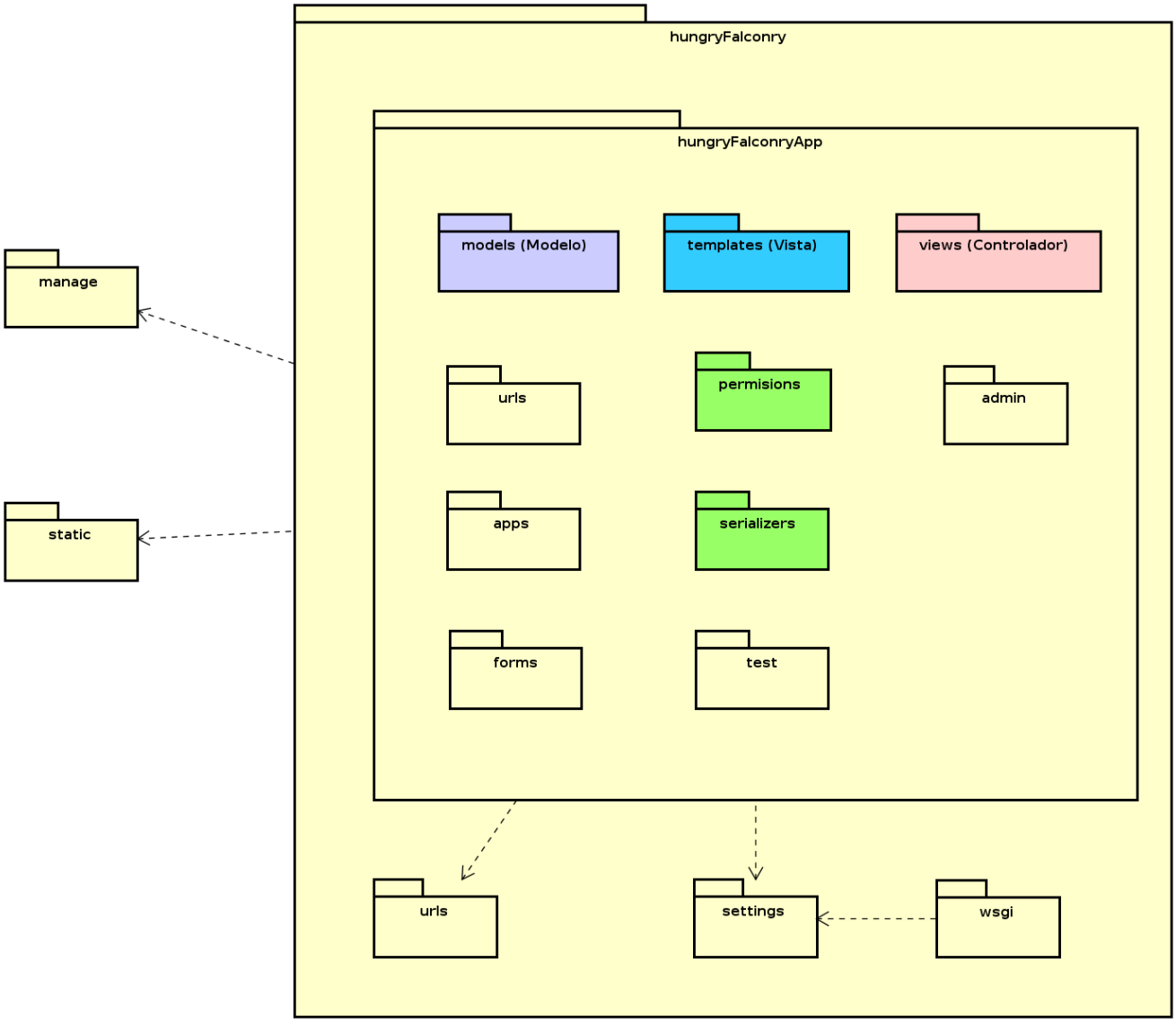


Figura 5.1: Arquitectura Django

En realidad los paquetes no son directorios como tal, se trata de ficheros Python (.py) que alojan funciones y clases pero que están diferenciados por funcionalidad. Es decir, en el fichero **views.py** están todos los controladores, actúa como paquete de clases, pero no es un directorio, es un fichero Python. Lo primero que hay que diferenciar es la existencia de tres niveles. Se procede a explicar del nivel más externo al más interno. En el nivel más externo de todos se encuentra el **manage**, encargado, entre una multitud de funcionalidades, de crear los proyectos y las aplicaciones, crear superusuarios, arrancar el servidor... En el paquete **static** se encuentran todos los archivos gráficos, ficheros css y javascript.
El siguiente nivel, el nivel de proyecto, llamado *hungryFalconry*, alberga las opciones (**settings**), las **urls** y el paquete **wsgi**, utilizado para el despliegue. El paquete urls se encarga del direccionamiento interno del proyecto: para una URL dada se ejecutará una acción. Además incluye las URL de las aplicaciones del proyecto.
El paquete **settings** se encarga de toda la configuración del proyecto, librerías instaladas, opciones de estas librerías, conexión a la base de datos, idioma, zona horaria...
Nivel de aplicaciones. Aquí se encuentran las aplicaciones desarrolladas dentro del mismo proyecto. En este caso solo existe una, denominada **hungryFalconryApp**. Dentro de cada aplicación existen tres paquetes más relevantes que el resto. El paquete **models**, encargado de crear los objetos y además plasmarlos en forma de tabla a la base de datos. Sigue actuando como el Modelo en una arquitectura MVC. Las plantillas (**templates**), corresponden a la vista en arquitectura MVC. Albergan el HTML y javascript correspondientes para visualización de la página. Por último las **views**, actúan como controladores de vista, por cada acción ocurrida en las templates se ejecutará una view.
El resto de paquetes:

- **urls**: Asocia cada URL a una acción, que es ejecutada en las views.
- **admin**: Registra los modelos para que sean accesibles desde la página de administración. Para así poderlos crear, editar, borrar...
- **apps**: Registra todas las aplicaciones instaladas en el proyecto.
- **forms**: Crea formularios. Es capaz de crear un formulario y ubicarlo en las templates para luego recoger sus valores y enviarlos a las views.
- **test**: Para crear pruebas de las aplicaciones.
- **permisions**: Pertenece a Django Rest Framework. Establece los permisos para los recursos de la API Rest.
- **serializers**: Pertenece a Django Rest Framework. Se encarga de enviar los campos establecidos de la base de datos a través de la API Rest.

5.2. Interfaz de usuario

En esta sección se incluirán los bocetos de las vista que compondrán la página web [1]. Sólo se muestra la vista del Usuario, pues, gracias al framework, las funciones de gestión que llevaría a cabo el Administrador vienen ya implementadas.

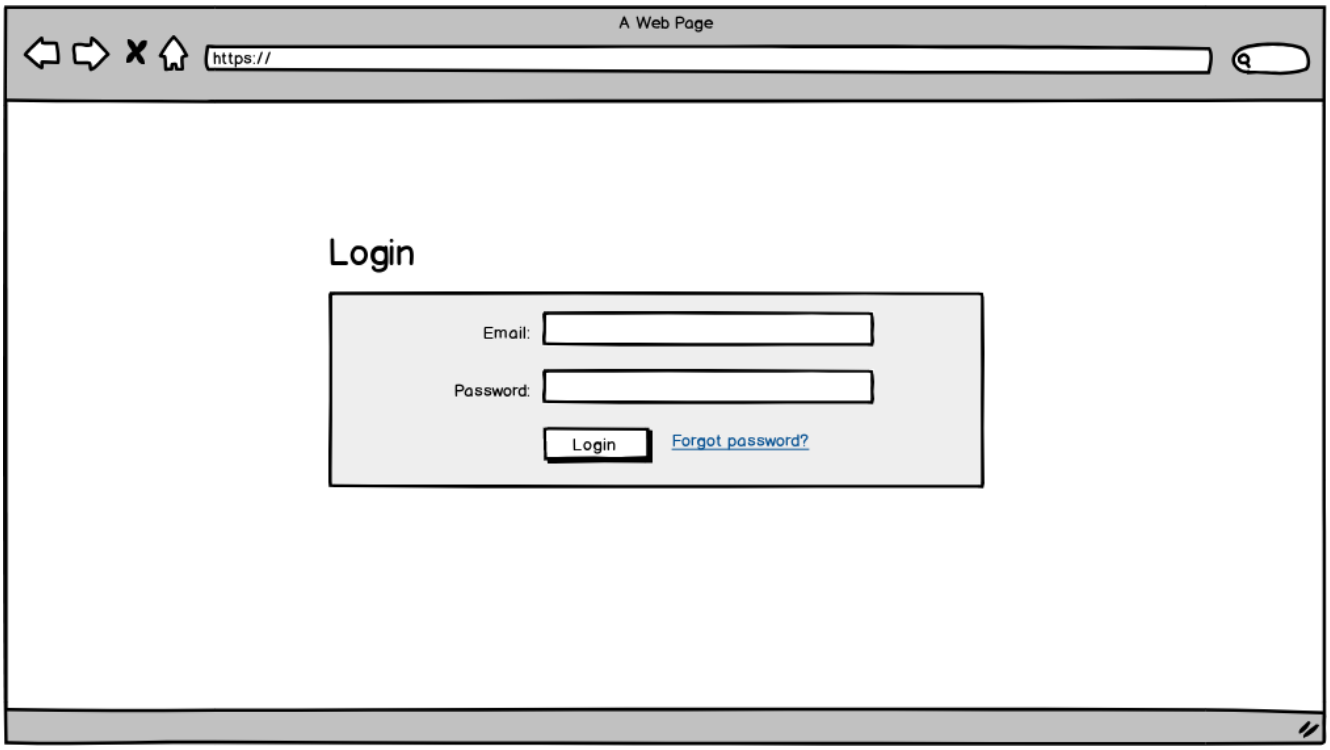


Figura 5.2: Vista inicio de sesión

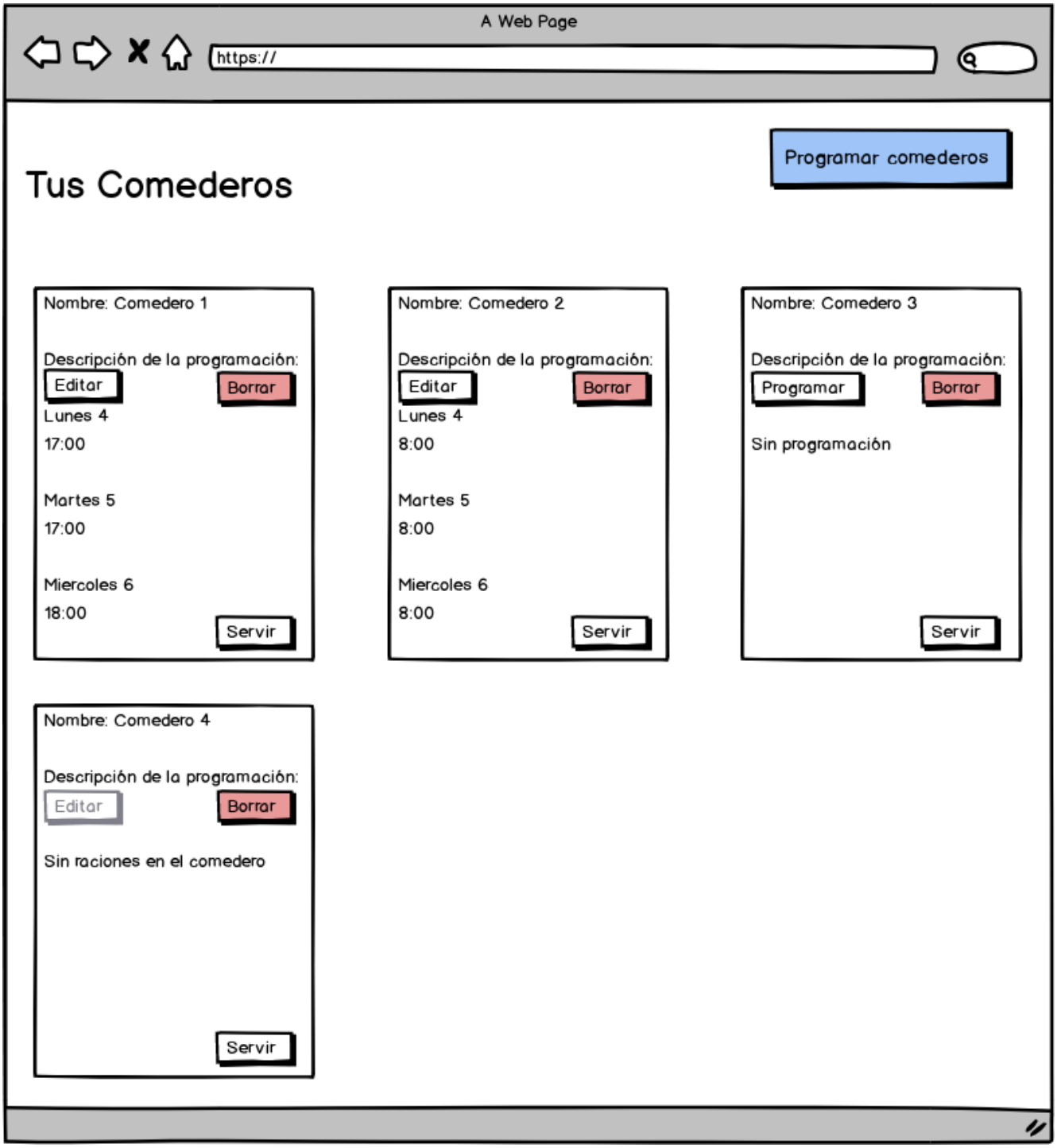


Figura 5.3: Vista todos los comederos

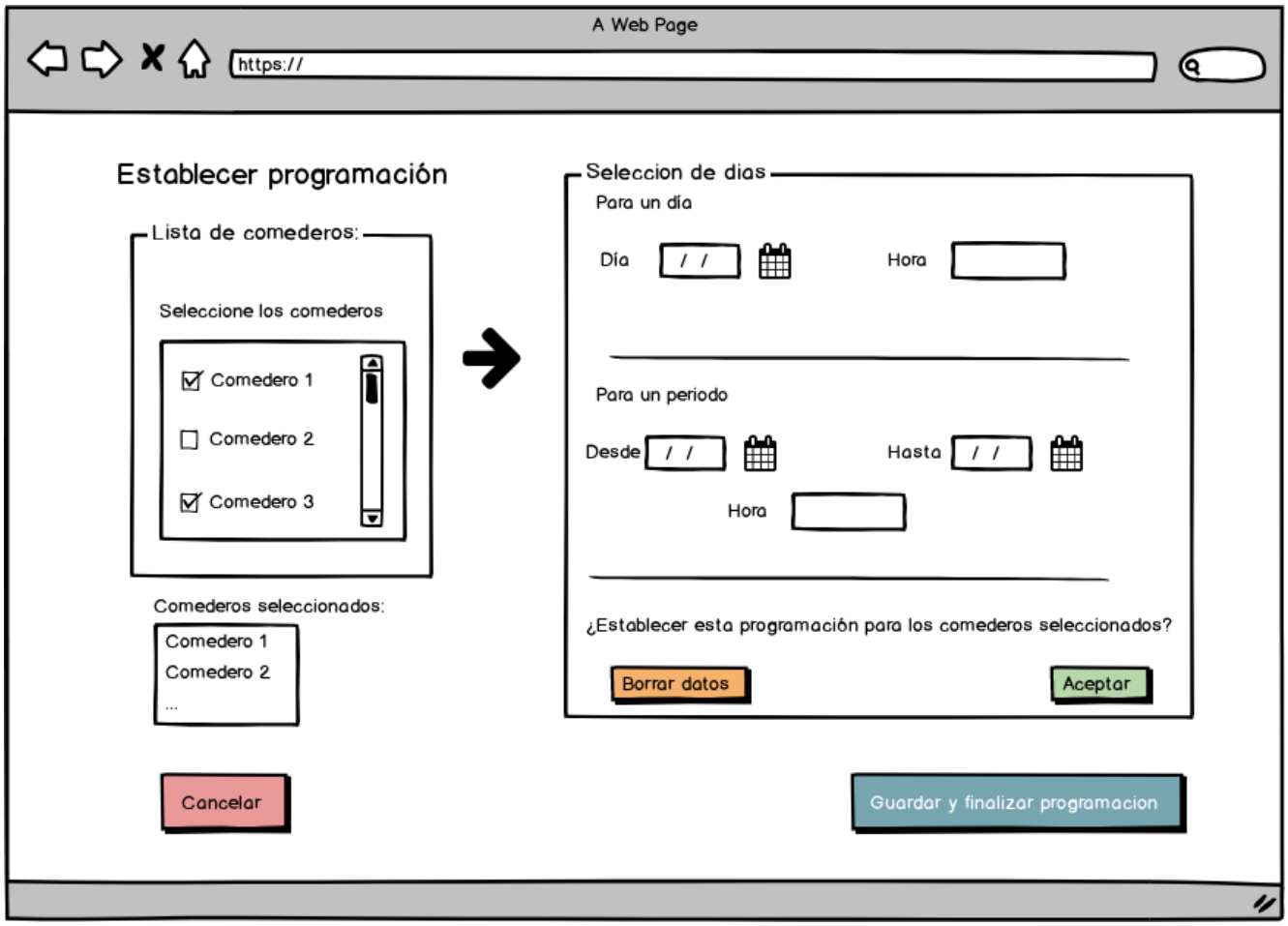


Figura 5.4: Vista programación de los comederos

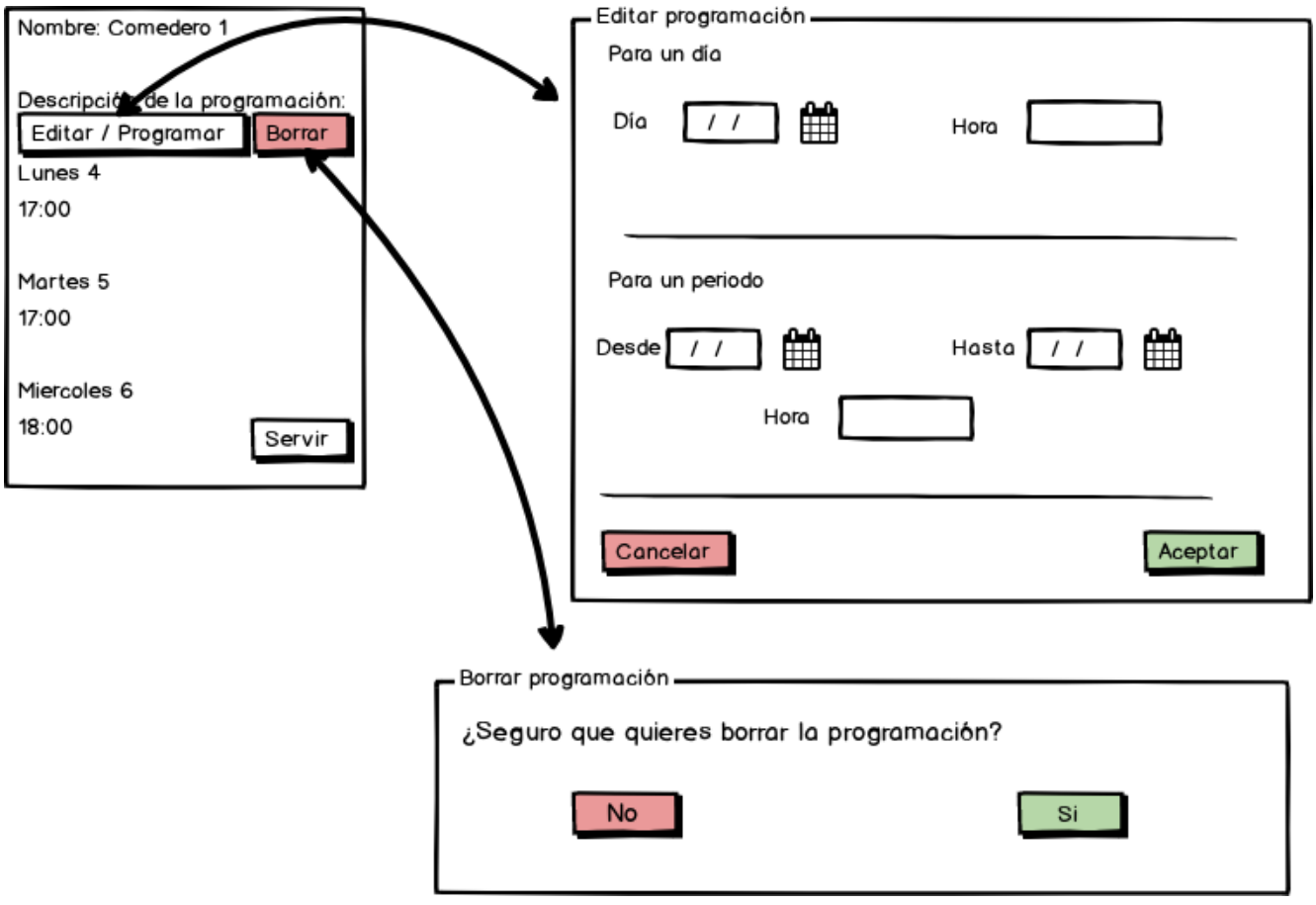


Figura 5.5: Vista edición y borrado de la programación

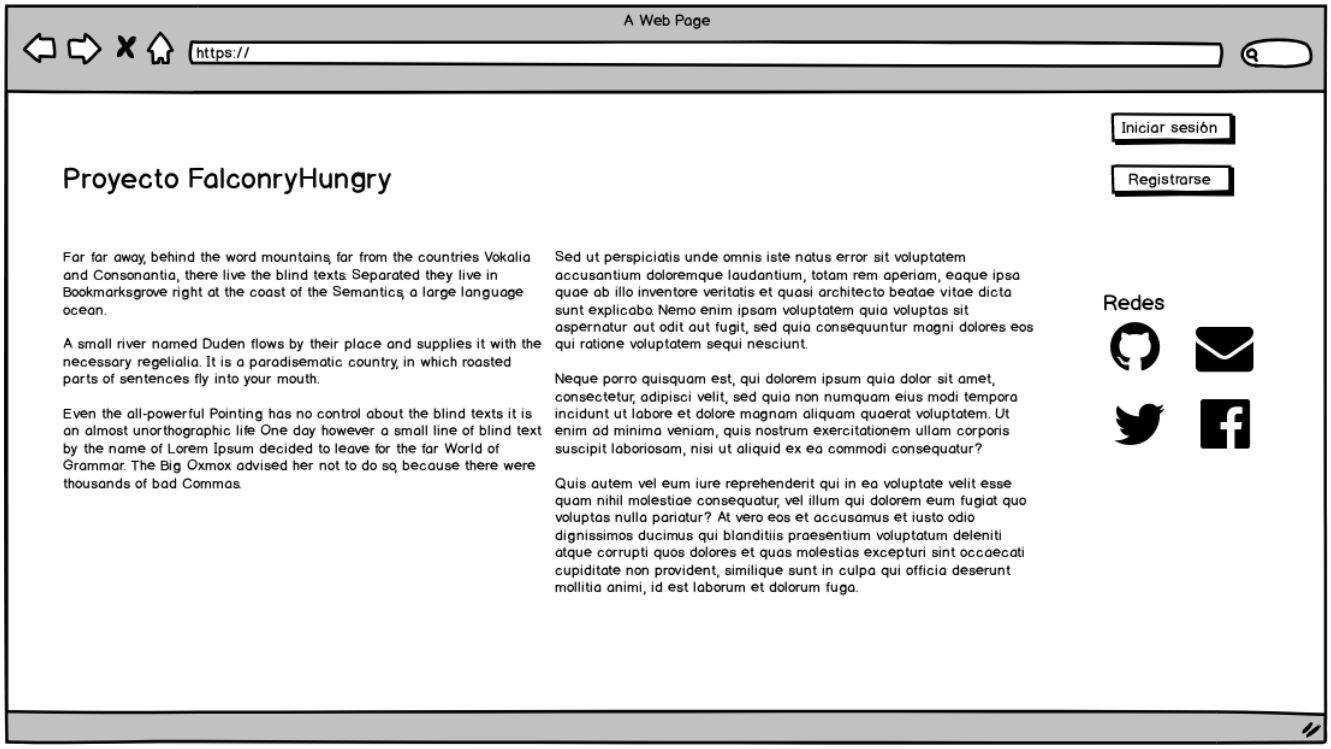


Figura 5.6: Vista información del proyecto

5.3. Modelo de dominio en diseño

A continuación se muestra el modelo de dominio en fase de diseño. Se muestran las clases junto con sus operaciones.

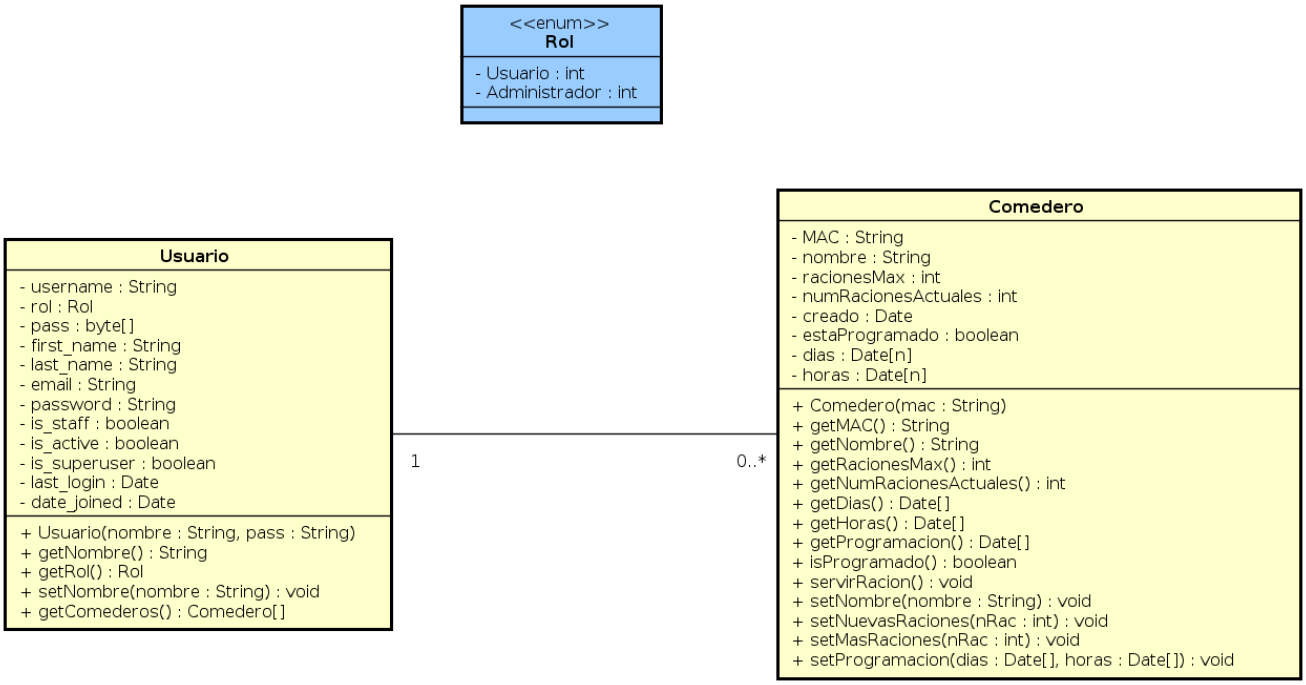


Figura 5.7: Modelo de dominio en la fase de diseño

5.4. Modelo de datos

A continuación se muestra diagrama entidad relación de la base de datos. Será una base de datos SQLite. Se detallan las relaciones y claves, tanto primarias como foráneas. Se ha optado por sacar una nueva entidad en la base de datos, *Programación*. La razón de esta decisión es debido a que la base de datos utilizada, SQLite, no permite desde Django almacenar listas de objetos, en este caso dos listas de objetos fecha y hora. Una de las formas sería a través de una estructura de datos JSON, se descartó esta opción por la forma en la que trata el framework el almacenamiento de JSONs. Además, en la entidad Programación se ha establecido un atributo más, denominado *servida*, indica si una programación concreta se ha ejecutado, es decir ha llegado la hora que ella tenía almacenada y se ha actuado en consecuencia sirviendo una ración.
Por otro lado la forma de almacenar las contraseñas por el framework es la siguiente: <algorithm>$<iterations>$<salt>$<hash>. Esta encriptación se denomina **PBKDF2**. Separados por el carácter de signo de dólar consisten en: el algoritmo de hash, el número de iteraciones de algoritmos (factor de trabajo), el salt aleatorio y el hash es la contraseña resultante [3].

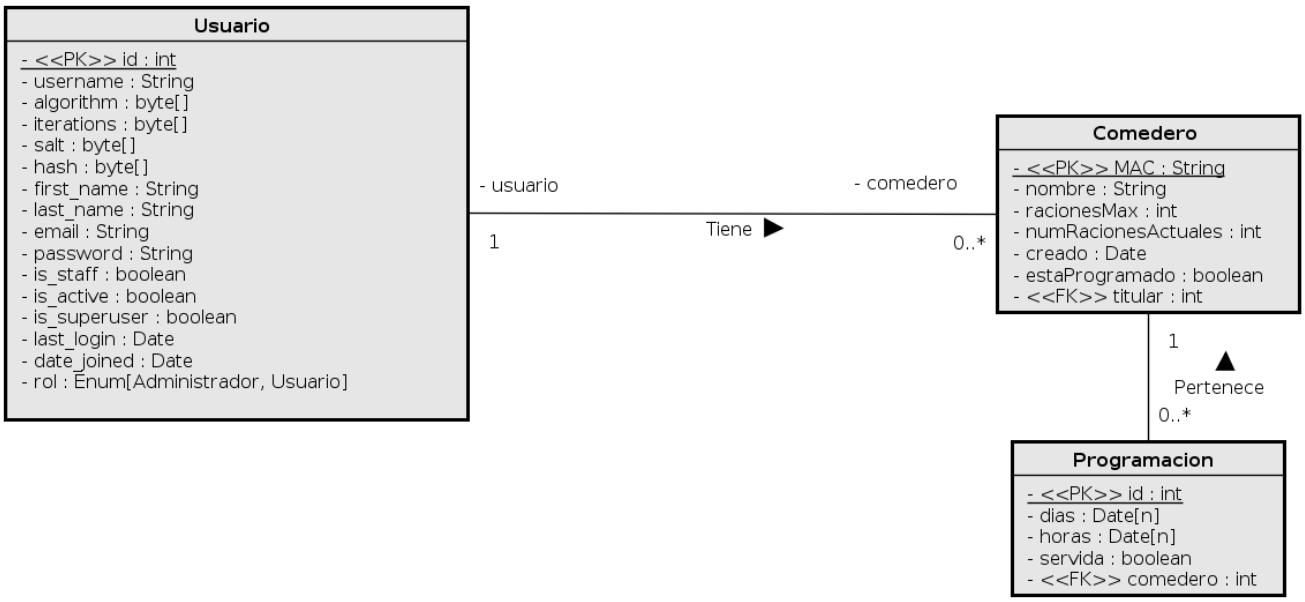


Figura 5.8: Modelo de datos en diseño

5.5. Diagramas de secuencia

Los diagramas de secuencia son un tipo de diagramas de decisión, en los que se muestra la interacción entre los objetos mediante mensajes que se transmiten entre ellos. Los objetos están representados mediante líneas de vida. Las líneas de vida representan la existencia del objeto que lleva su nombre a lo largo del tiempo. Se detallarán los considerados más relevantes para el proyecto. No toda la funcionalidad queda reflejada exactamente cómo funciona, pues el framework encapsula muchas de ellas.

5.5.1. Diagrama de secuencia del caso de uso: Registrar usuario

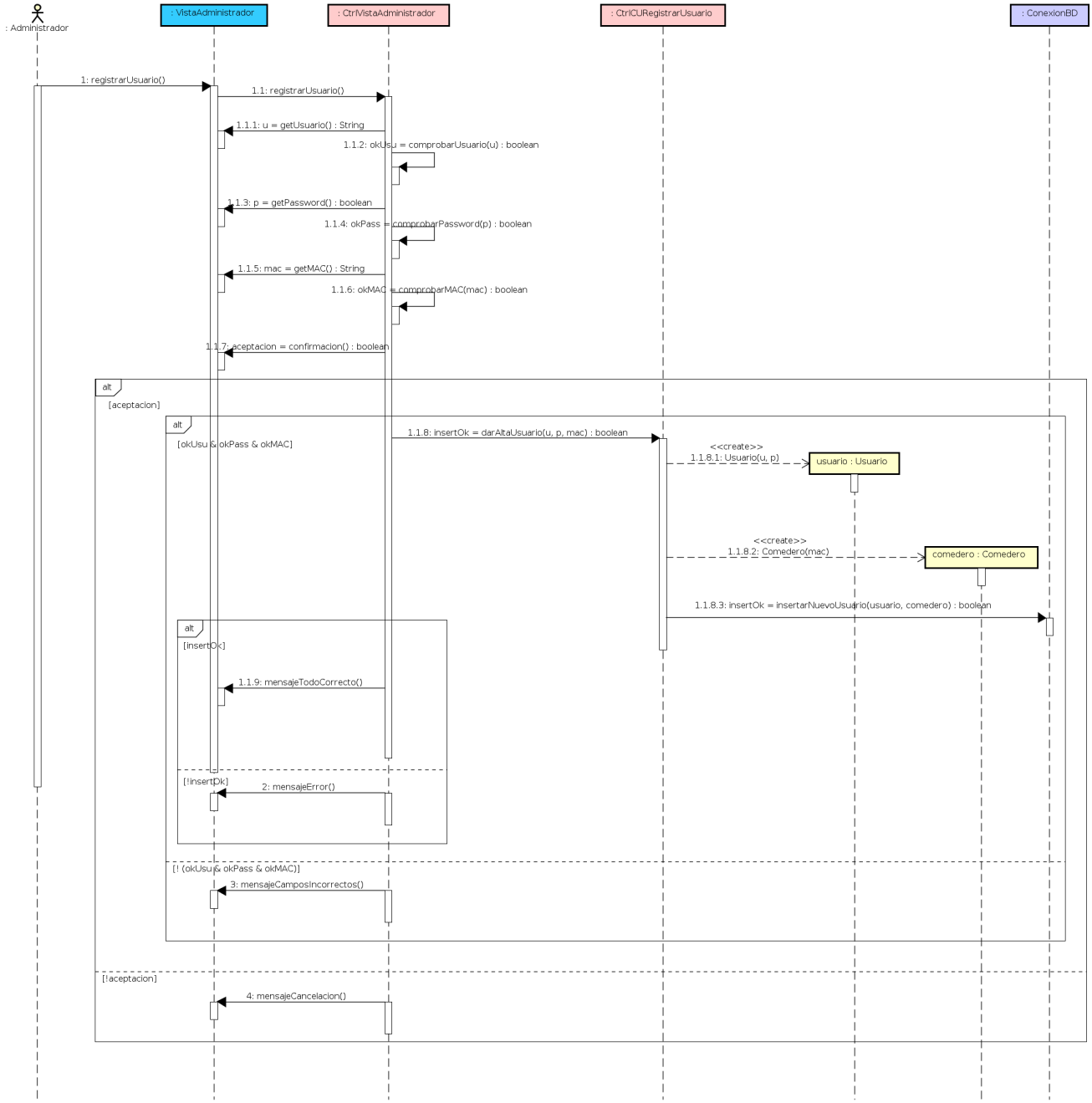


Figura 5.9: Diagrama de secuencia del caso de uso: Registrar usuario

5.5.2. Diagrama de secuencia del caso de uso: Servir una ración

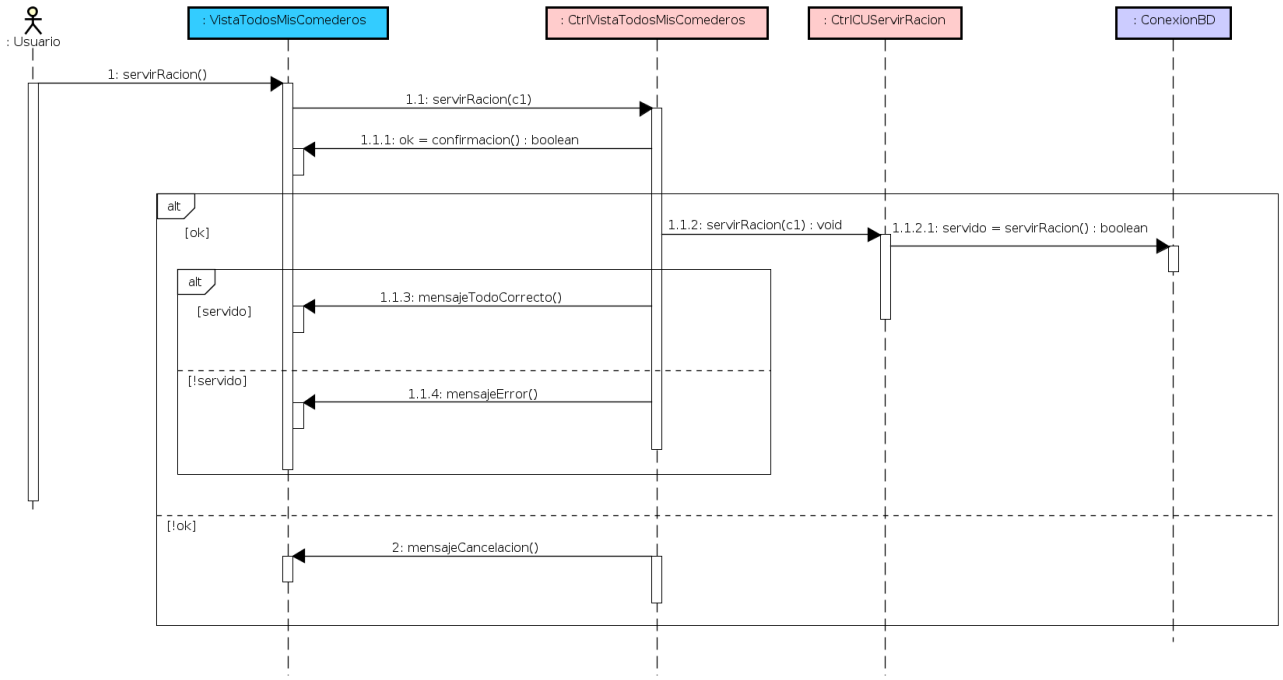


Figura 5.10: Diagrama de secuencia del caso de uso: Servir una ración

5.5.3. Diagrama de secuencia del caso de uso: Programar un comedero

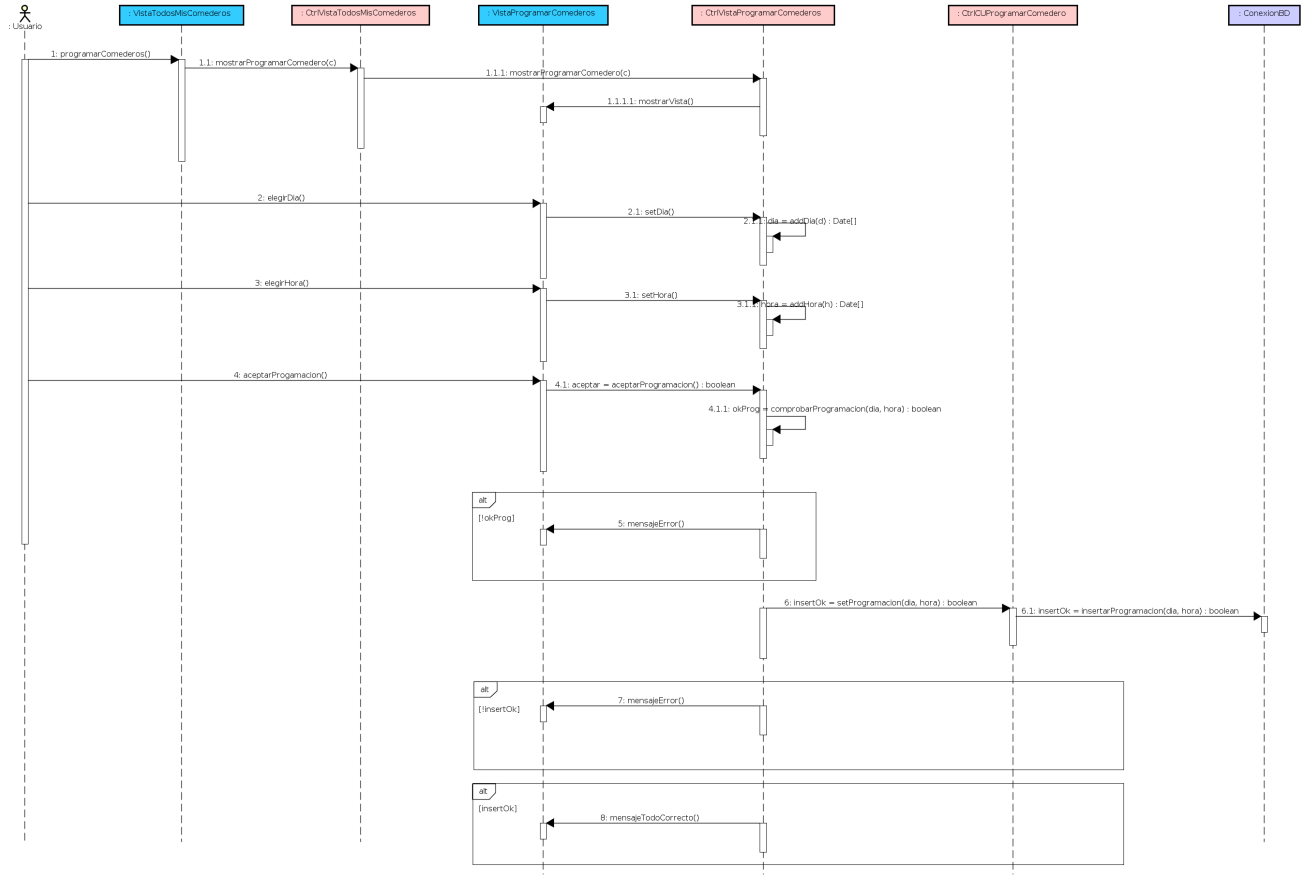


Figura 5.11: Diagrama de secuencia del caso de uso: Programar un comedero

5.5.4. Diagrama de secuencia del caso de uso: Borrar Programación

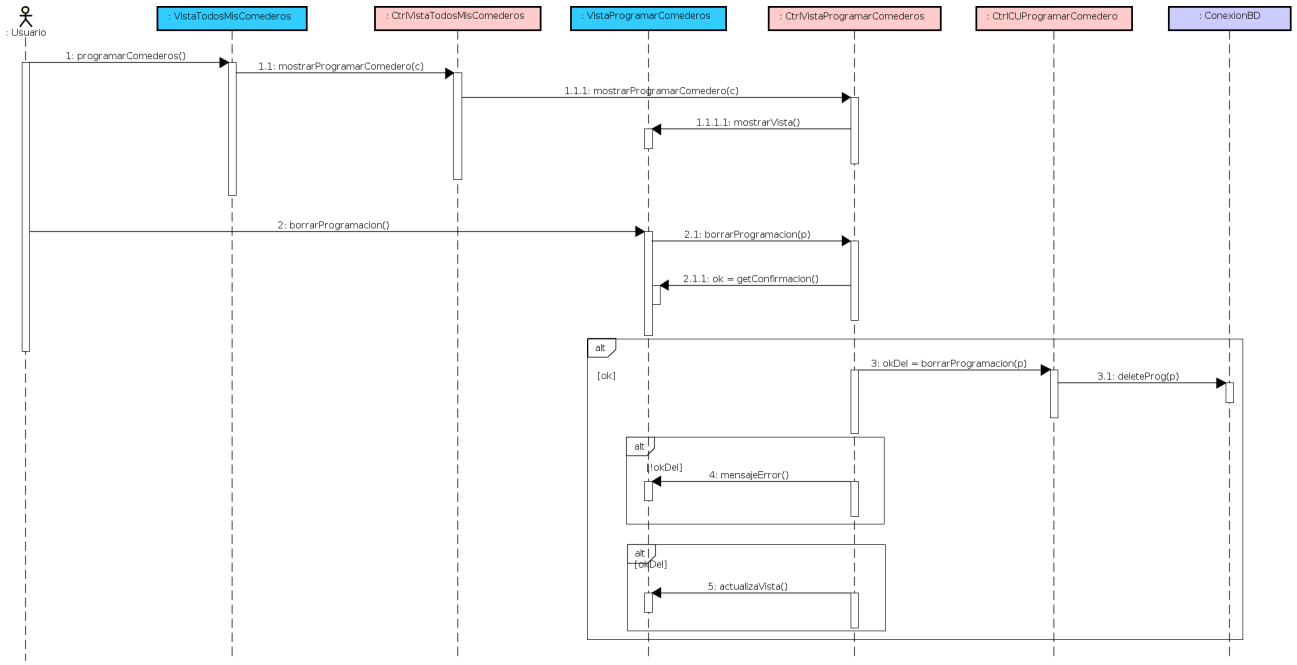


Figura 5.12: Diagrama de secuencia del caso de uso: Borrar Programación

5.5.5. Diagrama de secuencia del caso de uso: Registrar número de raciones

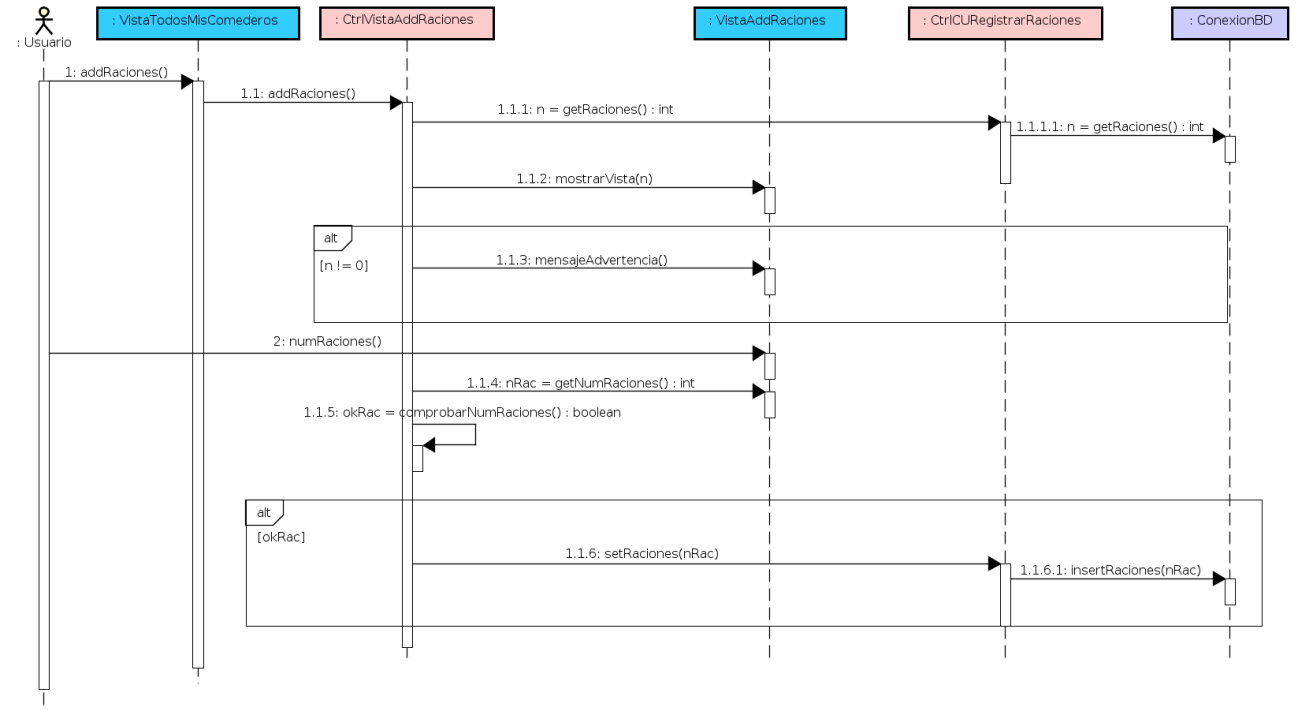


Figura 5.13: Diagrama de secuencia del caso de uso: Registrar número de raciones

5.5.6. Diagramas de flujo para la Raspberry Pi

Autenticación contra el API Rest

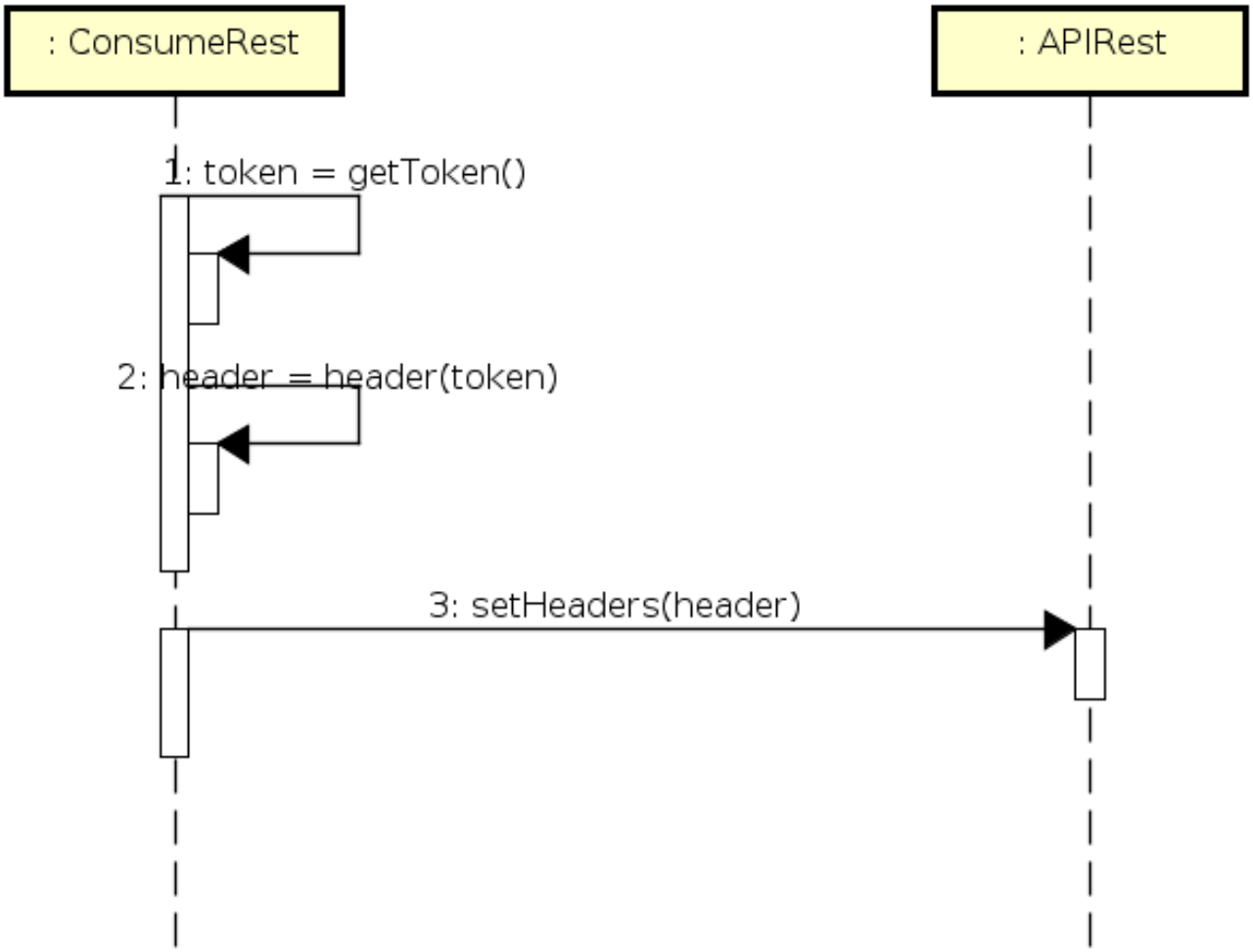


Figura 5.14: Diagrama de secuencia: Autenticación contra la API Rest

Escucha activa para la espera de un servicio y guardar programación

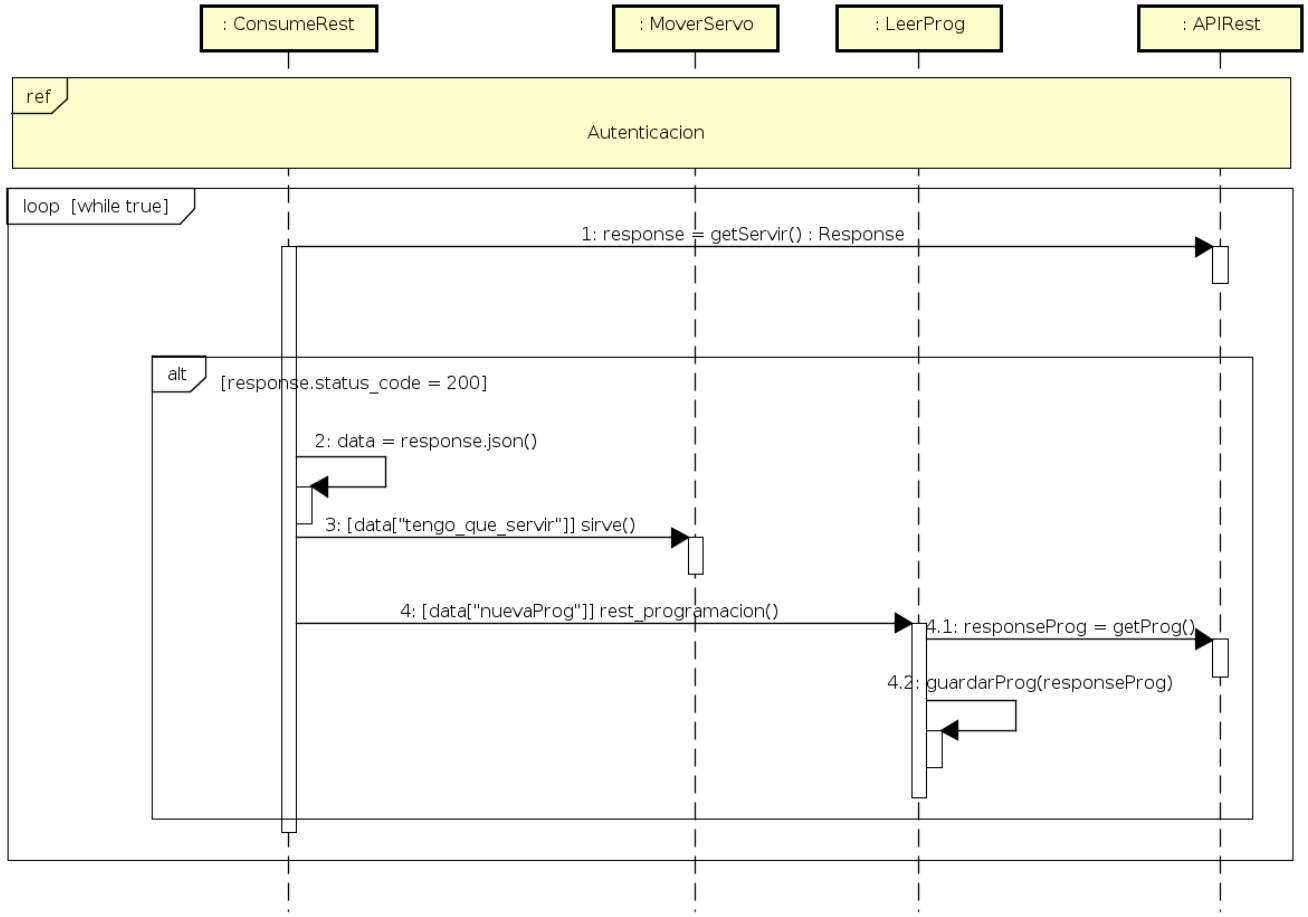


Figura 5.15: Diagrama de secuencia: Servir y guardar programación

Cumplir programación establecida y guardada

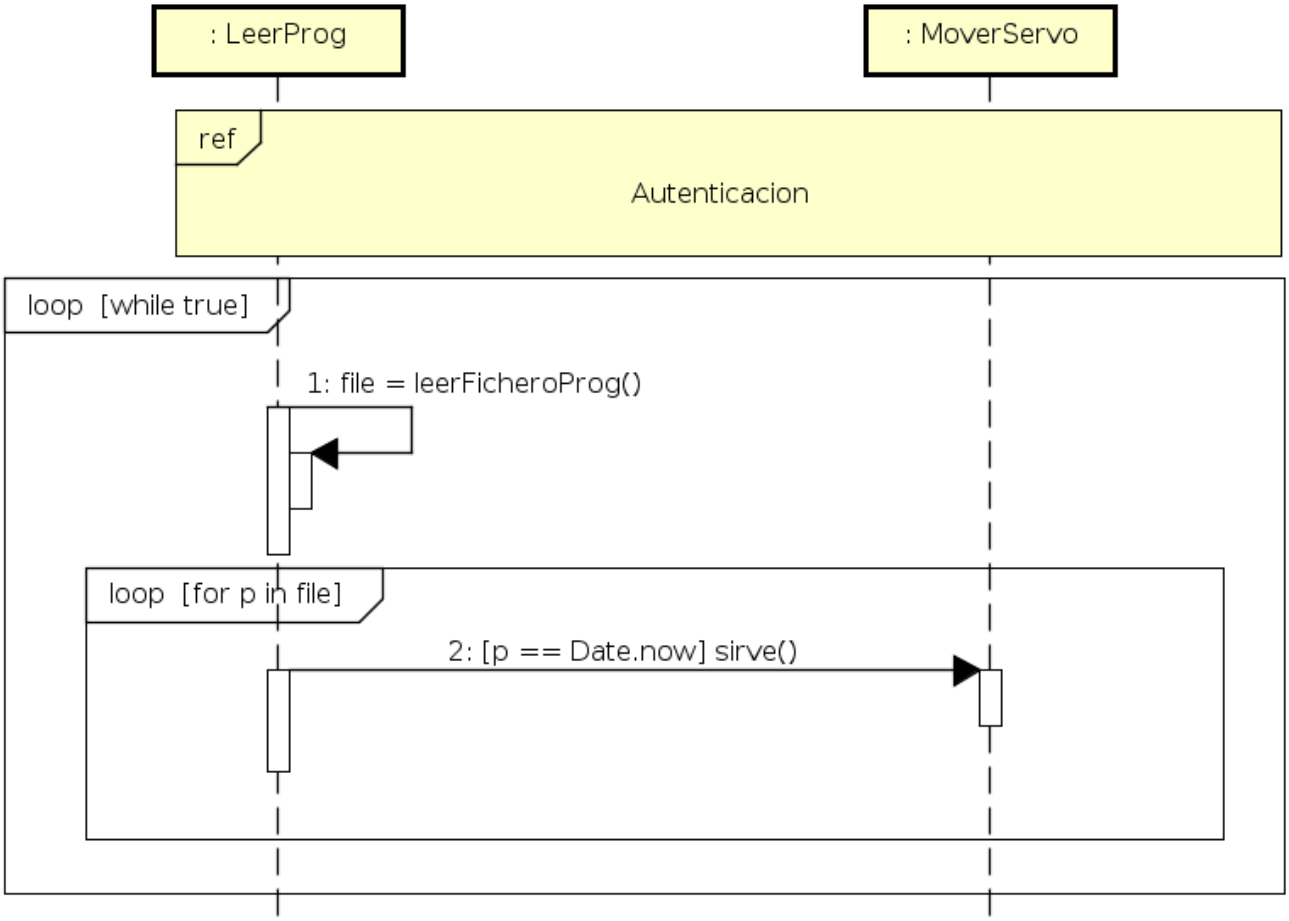


Figura 5.16: Diagrama de secuencia: Consumir programación establecida

5.6. Planos del comedero

5.6.1. Parte superior del Comedero

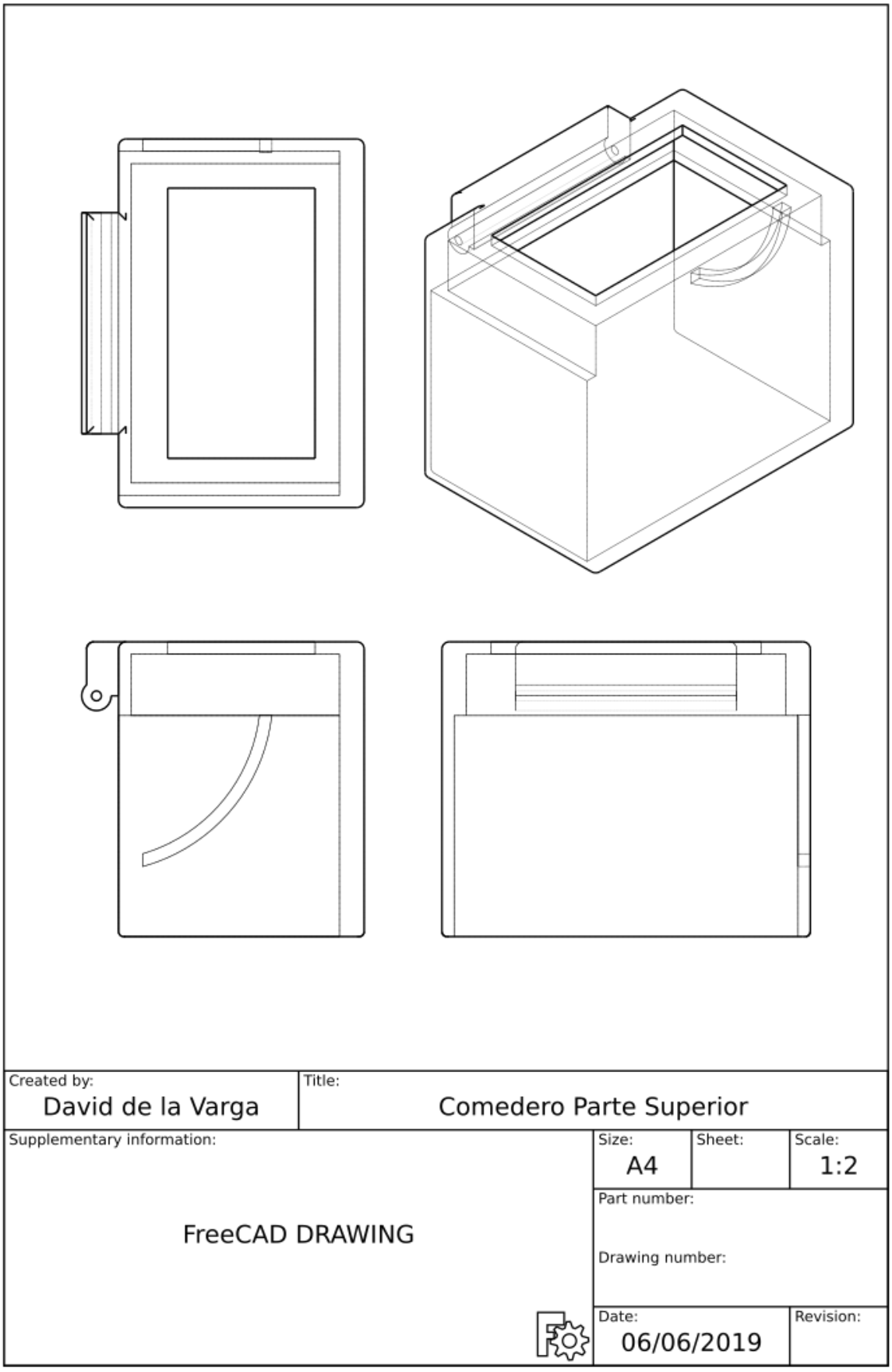


Figura 5.17: Plano parte superior del comedero

5.6.2. Parte inferior del Comedero

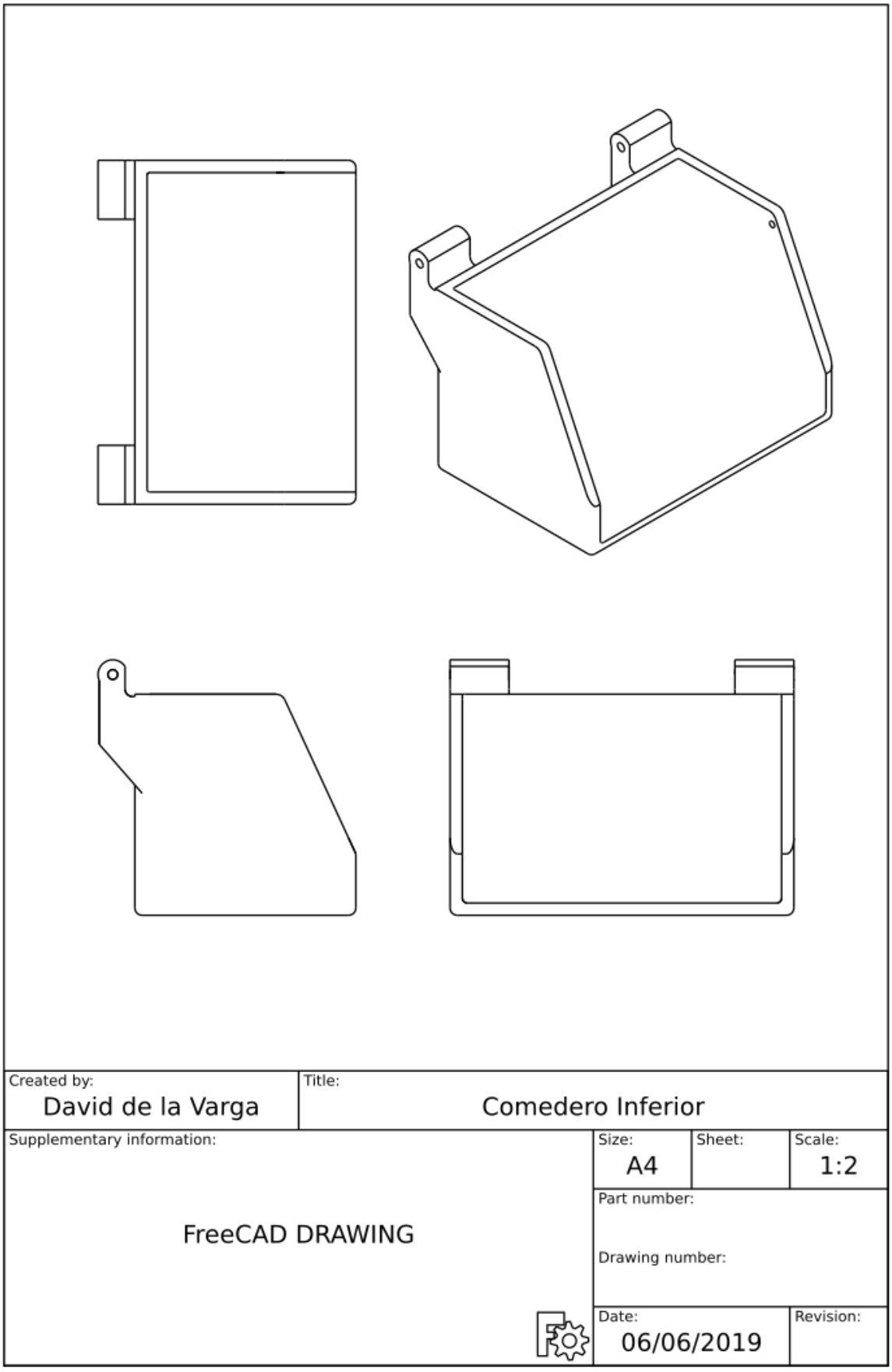


Figura 5.18: Plano parte inferior del comedero

Establecidas ambas partes del diseño del sistema, aplicación web y prototipo, se procede a su realización.

Capítulo 6

Implementación

6.1. Entorno de desarrollo

Para el desarrollo de este proyecto en Django se ha utilizado el IDE Pycharm, es un entorno de desarrollo para Python. Pycharm está desarrollado por JetBrains [10]. Para el frontend se ha utilizado Bootstrap 4, integrándolo en el Pycharm. Aunque en algunas funcionalidades ha sido necesario utilizar versiones anteriores debido a novedad de la versión cuatro [5].

6.2. Versiones necesarias de software

Para el uso de la aplicación es necesario tener un ordenador con conexión a internet desde un navegador:

- Google Chrome
- Mozilla Firefox

Por otra parte es necesario el comedero, compuesto por la estructura impresa en 3D y la Raspberry Pi que lo controlará.

6.3. Herramientas utilizadas

Todas las herramientas necesarias han sido instaladas y utilizadas en un entorno virtual para mayor comodidad con la gestión de versiones. A continuación se describirán las herramientas utilizadas para el desarrollo de la aplicación.

- **Django**, se profundizará en una sección posterior
- **django-bootstrap4** para el desarrollo de HTML, CSS y JS en la versión 0.0.8 [7]
- **django-bootstrap-datepicker-plus** para el uso de calendarios interactivos, en su versión 3.0.5 [6]
- **djangorestframework** para todo el desarrollo y gestión de la API Rest, en su versión 3.9.2 [9]
- **django-jquery** para la animación de diferentes funcionalidades, en su versión 3.1.0 [8]

Archivo *requeriments.txt* en el **Anexo II - Archivo requirements.txt**

6.4. Control de versiones

Para el control de versiones se ha utilizado Git, con un repositorio privado en GitHub.

6.5. Servidor de la aplicación

Para desplegar la aplicación se utilizado un servidor gratuito llamado PythonAnywhere [11]. Es una plataforma que sostiene un entorno Python ya instalado y listo para utilizarlo.

6.6. Implementación de la base de datos

Por defecto Django utiliza una base de datos SQLite, se trata de una base de datos relacional, de código abierto [12].

6.7. Sobre el framework Django

Django es un proyecto de código abierto desarrollado por *Django Software Foundation* [2]. Se trata de un framework para desarrollo web basado en Python. La versión utilizada para el desarrollo de esta aplicación es la versión Django 2.2, junto con Python 3.5. El framework cubre backend principalmente, y frontend en la justa medida:

- Frontend: En el código HTML, mediante anotaciones inserta código Python en cualquier lugar del documento. Existen dos tipos de anotaciones:

  1. {% ... %} Para insertar bloques, en los cuales se realizarán diferentes funcionalidades. Un ejemplo lo ilustrará mejor:

     {% for item in list %}
     Código que se repetirá
     {% endfor %}

     El *Código que se repetirá*, se mostrará en la página web tantas veces como se itere el bucle. Existen etiquetas para las sentencias básicas de programación, como pueden ser: ïf-else". Además para cargar diferentes librerías o archivos de formato se utiliza la anotación "load".
  2. {{ ... }} Encerrará una variable, de la cual se mostrará su valor en el momento de ejecución.

- Backend: Se desarrolla totalmente en Python junto con las librerías propias del framework. Sostiene todo lo relativo a los formularios (validación y creado), consultas a la base de datos, encaminamiento interno de las aplicaciones, configuraciones, despliegue... Quedará mejor detallado en el apartado sobre la estructura de ficheros en el manual para el programador. Además gracias a la incorporación de otras herramientas cómo Django Rest Framework se pueden ampliar las aplicaciones de una forma sencilla. Documentación de la API Rest en el **Anexo I - Documentación API Rest**.

- Librerías instaladas en el proyecto a modo de archivo **requirements.txt**

```
amqp==2.4.2
billiard==3.6.0.0
certifi==2019.3.9
chardet==3.0.4
coreapi==2.3.3
coreschema==0.0.4
Django==2.2
django-bootstrap-datepicker-plus==3.0.5
django-bootstrap4==0.0.8
django-bootstrap4-datetimepicker==4.2
django-compat==1.0.15
django-jquery==3.1.0
djangorestframework==3.9.2
idna==2.8
itypes==1.1.0
Jinja2==2.10.1
kombu==4.5.0
Markdown==3.1.1
MarkupSafe==1.1.1
Pygments==2.4.2
pytz==2018.9
requests==2.22.0
six==1.12.0
sqlparse==0.3.0
uritemplate==3.0.0
urllib3==1.25.3
vine==1.3.0
```

Figura 6.1: Arquivo requirements.txt

Desarrollado el sistema se procede a realizar las pruebas pertinentes.

Capítulo 7

Pruebas

En el presente capítulo se llevarán a cabo las pruebas sobre el sistema desarrollado, comprobando así que cumple los requisitos establecidos. Se realizarán pruebas de caja blanca sobre el frontend de la aplicación web.

7.1. Pruebas del frontend

Las pruebas se realizarán con la aplicación desplegada. Se comprobará que se cumplen los requisitos y las restricciones impuestas.

7.1.1. Pruebas de caja negra para los usuarios generales

Prueba 1.1 para requisito funcional 1.

Prueba 1.1	
Descripción	Esta prueba está destinada a probar el funcionamiento del inicio de sesión
Acción	Iniciar sesión en la aplicación web con rol Usuario
Resultado Esperado	El usuario tiene acceso a la vista previa de todos sus comederos

Prueba 1.2 Requisito funcional 5.

Prueba 1.2	
Descripción	Esta prueba está destinada a probar el funcionamiento de servir una ración
Acción	Pulsar botón servir
Resultado Esperado	El comedero asociado al botón actúa moviéndose y sirviendo, se resta una ración de las raciones actuales

Prueba 1.3 Requisito funcional 3.

Prueba 1.3	
Descripción	Esta prueba está destinada a probar el funcionamiento de añadir raciones a un comedero.
Acción	Se añade nuevas raciones
Resultado Esperado	Las raciones correspondientes al comedero se modifican por las nuevas introducidas

Prueba 1.4 Requisito funcional 4.

Prueba 1.4	
Descripción	Esta prueba está destinada a probar el funcionamiento de modificar las raciones
Acción	Se modifican las raciones de un comedero
Resultado Esperado	Se sobrescribe el valor de raciones anterior por el que actual introducido

Prueba 1.5 Requisito funcional 6.

Prueba 1.5	
Descripción	Esta prueba está destinada a probar el funcionamiento de programar un comedero
Acción	Se programa el servicio de una ración
Resultado Esperado	Se añade un nuevo día y hora a la lista de programaciones, su estado es pendiente, se refleja con un icono de reloj

Prueba 1.6 Requisito funcional 6.

Prueba 1.6	
Descripción	Esta prueba está destinada a probar el funcionamiento de programar un comedero
Acción	Llega el instante programado
Resultado Esperado	Se sirve una ración del comedero, se resta una ración y la programación asociada se marca como servida con un tick verde en la lista de programaciones

Prueba 1.7 Requisito funcional 7.

Prueba 1.7	
Descripción	Esta prueba está destinada a probar el funcionamiento de borrar una programación
Acción	Se borra una programación
Resultado Esperado	Se elimina la programación correspondiente. Si es una programación servida no se borrará

Prueba 1.8 Requisito funcional 9.

Prueba 1.8	
Descripción	Esta prueba está destinada a probar el funcionamiento de editar el nombre del comedero
Acción	Se edita el nombre del comedero
Resultado Esperado	Se edita el nombre del comedero y se refleja en la vista de todos los comederos

7.1.2. Pruebas de caja blanca para los usuarios generales

Prueba 1.9 Requisito funcional 3 y 4.

Prueba 1.9	
Descripción	Esta prueba está destinada a probar el funcionamiento de añadir nuevas raciones
Acción	Se accede al menú de añadir raciones.
Resultado Esperado	Como existen raciones anteriores se muestra un mensaje de advertencia.

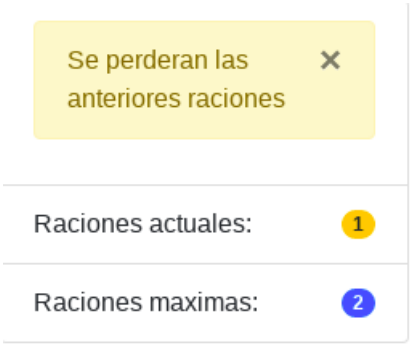


Figura 7.1: Mensaje de advertencia añadiendo raciones

Prueba 1.10 Requisito funcional 3 y 4.

Prueba 1.10	
Descripción	Esta prueba está destinada a probar el funcionamiento de añadir más raciones de las permitidas.
Acción	Se intenta añadir más raciones que el número máximo de raciones.
Resultado Esperado	Se muestra un mensaje de error, advirtiendo de que es un número de raciones erróneo.

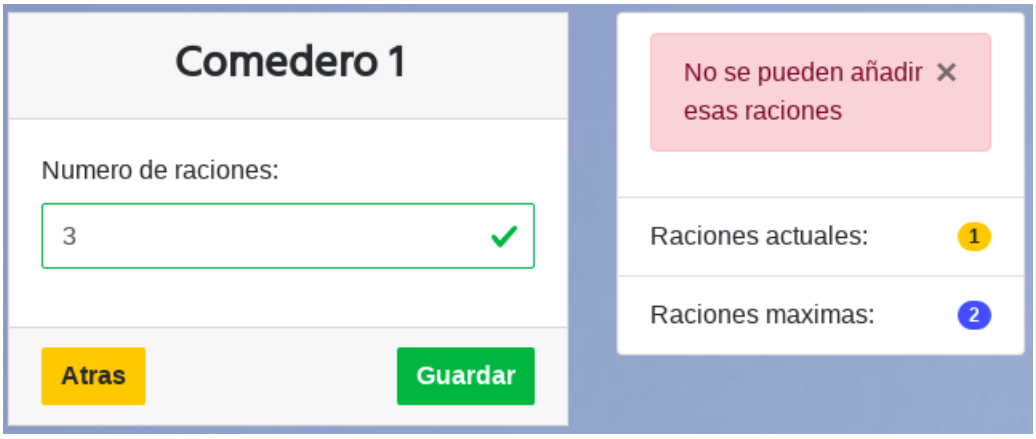


Figura 7.2: Mensaje de error añadiendo más raciones de las permitidas

Prueba 1.11 Requisito funcional 3 y 4.

Prueba 1.11	
Descripción	Esta prueba está destinada a probar el funcionamiento de añadir un número negativo de raciones.
Acción	Se intenta añadir un número negativo de raciones.
Resultado Esperado	Se muestra un mensaje de error, advirtiendo de que es un número de raciones erróneo.

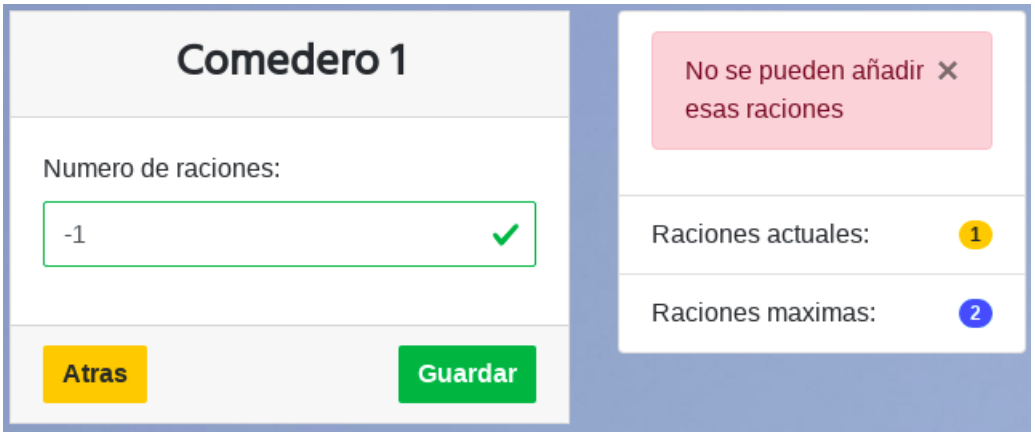


Figura 7.3: Mensaje de error añadiendo un número negativo de raciones

Prueba 1.12 Requisito funcional 3 y 4.

Prueba 1.12	
Descripción	Esta prueba está destinada a probar el funcionamiento de añadir cero raciones.
Acción	Se intenta añadir cero raciones.
Resultado Esperado	Se muestra un mensaje de error, advirtiendo de que es un número de raciones erróneo.

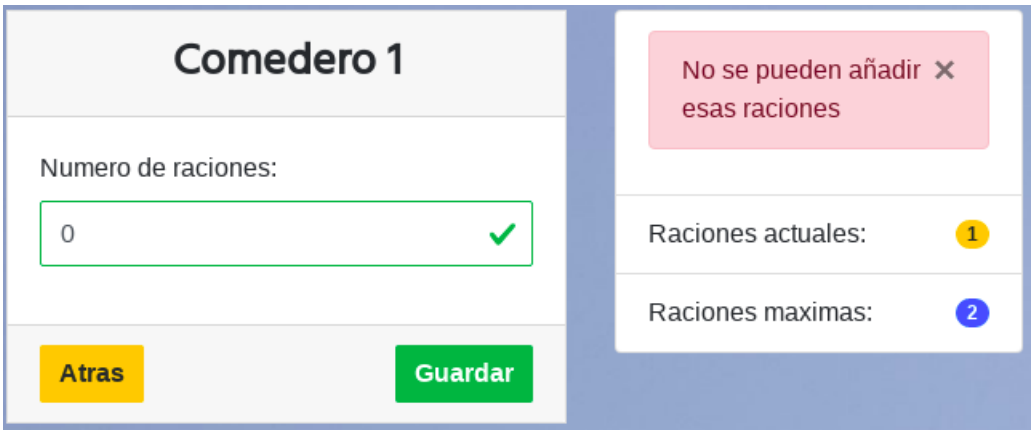


Figura 7.4: Mensaje de error añadiendo cero raciones

En el caso de las figuras proporcionadas para las pruebas 1.10, 1.11 y 1.12 se observa que el comedero posee una ración, este parámetro no influye en el comportamiento ni las pruebas ni del programa, porque si se introduce un número de raciones correcto este valor se sobrescribirá.

Prueba 1.13 Requisito funcional 6.

Prueba 1.13	
Descripción	Esta prueba está destinada a probar el funcionamiento de añadir más programaciones que raciones actuales.
Acción	Se intenta añadir una segunda programación teniendo solo una ración en el comedero.
Resultado Esperado	Se muestra un mensaje de error, advirtiendo de que es no se puede programar más servicios que raciones.

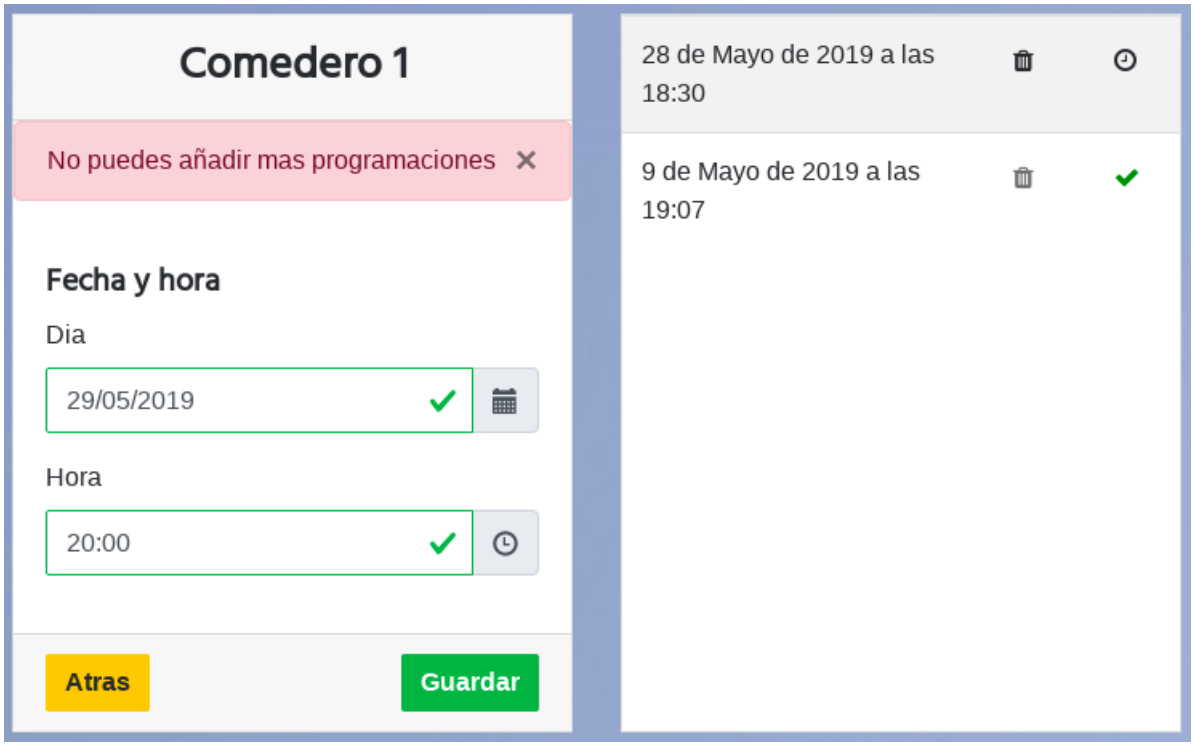


Figura 7.5: Mensaje de error añadiendo más programaciones que raciones actuales

7.1.3. Pruebas de caja negra para los usuarios administradores

Prueba 2.1 Requisito funcional 1.

Prueba 2.1	
Descripción	Esta prueba está destinada a probar el funcionamiento del inicio de sesión
Acción	Iniciar sesión en la aplicación web con rol Administrador
Resultado Esperado	El usuario tiene acceso a todas las funcionalidades de administrador

Prueba 2.2 Requisito funcional 10.

Prueba 2.2	
Descripción	Esta prueba está destinada a probar el funcionamiento de registrar un nuevo usuario
Acción	Se registra un usuario nuevo
Resultado Esperado	El nuevo usuario sale en la lista de usuarios, con todos sus datos

Prueba 2.3 Requisito funcional 11.

Prueba 2.3	
Descripción	Esta prueba está destinada a probar el funcionamiento de registrar un nuevo comedero
Acción	Se da de alta un nuevo comedero
Resultado Esperado	El nuevo comedero aparece en la lista de todos los comederos, con su nombre. Además aparece asociado a un usuario existente

7.1.4. Pruebas de caja blanca para los usuarios administradores

No se realizarán pruebas de caja blanca para el sitio del Administrador, pues es una funcionalidad proporcionada por el propio framework, así que se supondrá su correcto funcionamiento.

7.1.5. Pruebas sobre la API Rest

Para las siguientes pruebas se usará la interfaz gráfica proporcionada por Django REST framework. Se adjunta la URL sin la dirección del servidor.

Previa autenticación en todas las pruebas. En este caso la autenticación es mediante usuario y contraseña.

Prueba 3.1

Prueba 3.1	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación GET para todos los comederos.
Resultado Esperado	Se consigue un JSON con todos los comederos del sistema.

```
GET /comederos/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "mac": "XX:XX:XX:XX:XX:XX",
        "nombre": "Comedero 1",
        "tengo_que_servir": true,
        "numRacionesActuales": 1,
        "nuevaProgramacion": true
    },
    {
        "mac": "X1:XX:XX:XX:XX:XX",
        "nombre": "Comedero 2",
        "tengo_que_servir": true,
        "numRacionesActuales": 1,
        "nuevaProgramacion": true
    },
    {
        "mac": "X2:XX:XX:XX:XX:XX",
        "nombre": "Comedero 3",
        "tengo_que_servir": true,
        "numRacionesActuales": 1,
        "nuevaProgramacion": false
    }
]
```

Figura 7.6: GET comederos API REST

Prueba 3.2

Prueba 3.2	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación GET para un comedero.
Resultado Esperado	Se consigue un JSON con la información del comedero deseado.

```
GET /comederos/XX:XX:XX:XX:XX:XX/
```

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "mac": "XX:XX:XX:XX:XX:XX",
    "nombre": "Comedero 1",
    "tengo_que_servir": true,
    "numRacionesActuales": 1,
    "nuevaProgramacion": true
}
```

Figura 7.7: GET un sólo comedero API REST

Prueba 3.3

Prueba 3.3	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación GET para la programación de un comedero.
Resultado Esperado	Se consigue un JSON con todas las programaciones del comedero deseado.

```
GET /programacion/XX:XX:XX:XX:XX:XX/get
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 88,
        "dia": "2019-05-09",
        "hora": "19:07:00",
        "comedero": "XX:XX:XX:XX:XX:XX",
        "servida": true
    },
    {
        "id": 113,
        "dia": "2019-05-28",
        "hora": "18:30:00",
        "comedero": "XX:XX:XX:XX:XX:XX",
        "servida": false
    }
]
```

Figura 7.8: GET programación para un comedero API REST

Prueba 3.4

Prueba 3.4	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación GET para los usuarios del sistema y sus comederos.
Resultado Esperado	Se consigue un JSON con todos los usuarios y sus comederos.

```
GET /users/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "username": "david",
        "comederos": []
    },
    {
        "id": 4,
        "username": "davdela",
        "comederos": [
            "XX:XX:XX:XX:XX:XX",
            "X1:XX:XX:XX:XX:XX",
            "X2:XX:XX:XX:XX:XX"
        ]
    }
]
```

Figura 7.9: GET usuarios y sus comederos API REST

Prueba 3.5

Prueba 3.5	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación GET para un sólo usuario del sistema y sus comederos.
Resultado Esperado	Se consigue un JSON con datos del usuario y sus comederos.

```
GET /users/4/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 4,
    "username": "davdela",
    "comederos": [
        "XX:XX:XX:XX:XX:XX",
        "X1:XX:XX:XX:XX:XX",
        "X2:XX:XX:XX:XX:XX"
    ]
}
```

Figura 7.10: GET un solo usuario y sus comederos API REST

Prueba 3.6

Prueba 3.6	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación PUT para modificar la programación de un comedero.
Resultado Esperado	Se modifica los datos de la programación.

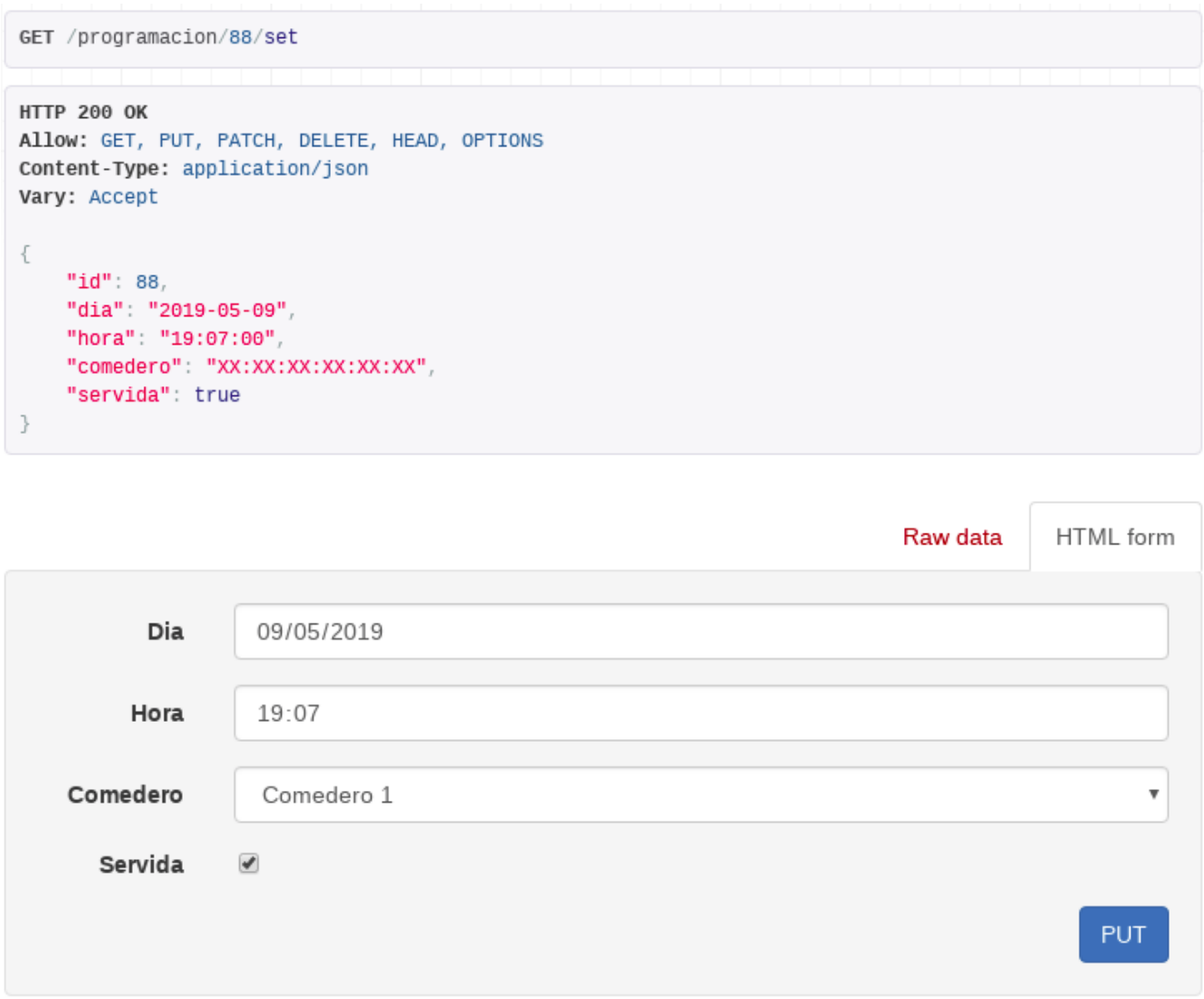


Figura 7.11: PUT programación de un comedero API REST

Prueba 3.7

Prueba 3.7	
Descripción	Esta prueba está destinada a probar el funcionamiento de la operación PUT para modificar la información de un comedero.
Resultado Esperado	Se modifica los datos del comedero.

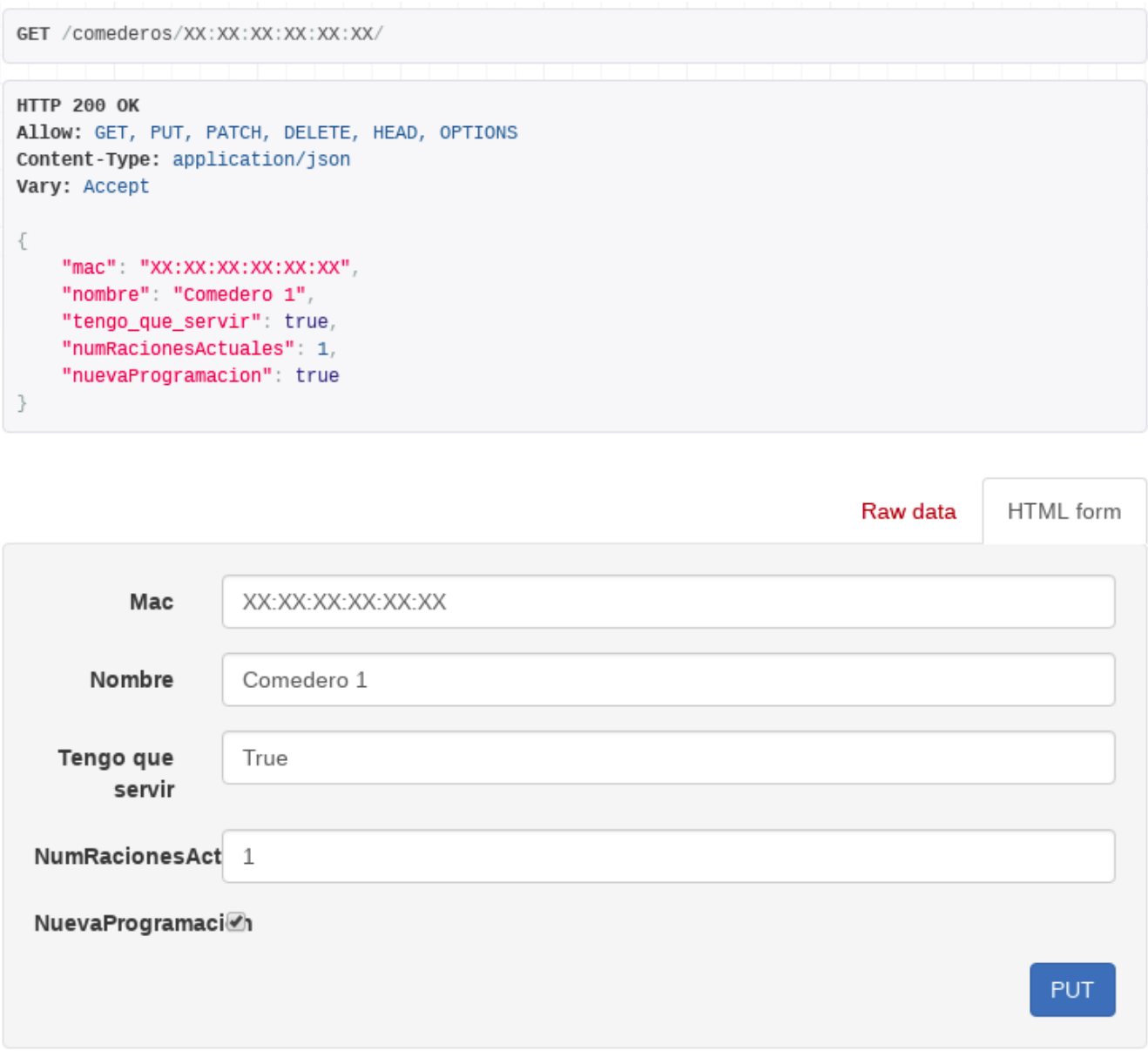


Figura 7.12: PUT programación de un comedero API REST

Después de haber realizado las pruebas se explicará el uso de la aplicación para los posibles usuarios.

Capítulo 8

Manual del usuario

En este capítulo se mostrará cómo hacer uso de la aplicación. Lo primero que hay que realizar es ponerse en contacto con el dueño del sistema vía correo electrónico, a la dirección:

`hungryfalconry@gmail.com`

Se proporcionarán los datos de usuario y contraseña deseados, Además, si existiese más de un modelo de comederos, sería necesario pedir el modelo elegido. Con estos datos y una vez que el administrador nos ha dado de alta, tendremos acceso a la aplicación web, accesible mediante:

`http://davidelavarga.pythonanywhere.com/login/`

8.1. Usuario normal

Como se ha mencionado antes, la aplicación será accesible mediante: `http://davidelavarga.pythonanywhere.com/login/`.

8.1.1. Inicio de sesión

Se inicia sesión desde la página de *Inicio de sesión* (ver Fig. 8.1), una vez dentro se mostrará el menú de todos los comederos que posea el usuario (ver Fig. 8.2), a partir de ahora se denominará también pantalla principal. Al menos será posible ver un comedero con las características que se decidieron en la petición realizada por correo electrónico.
El botón de servir se encuentra deshabilitado, pues el comedero actualmente no contiene comida en su interior, por lo que no se puede servir ni programar.

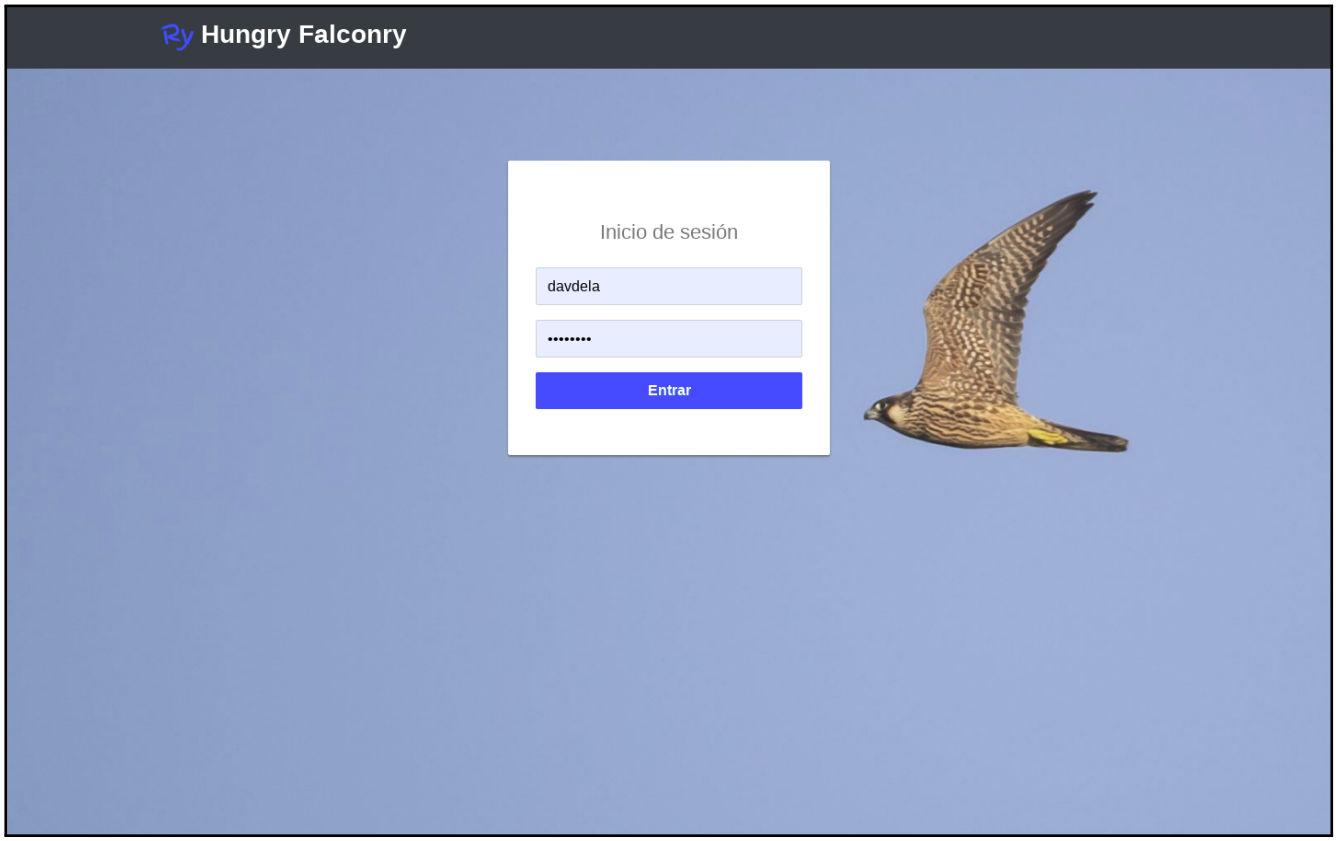


Figura 8.1: Vista de inicio de sesión

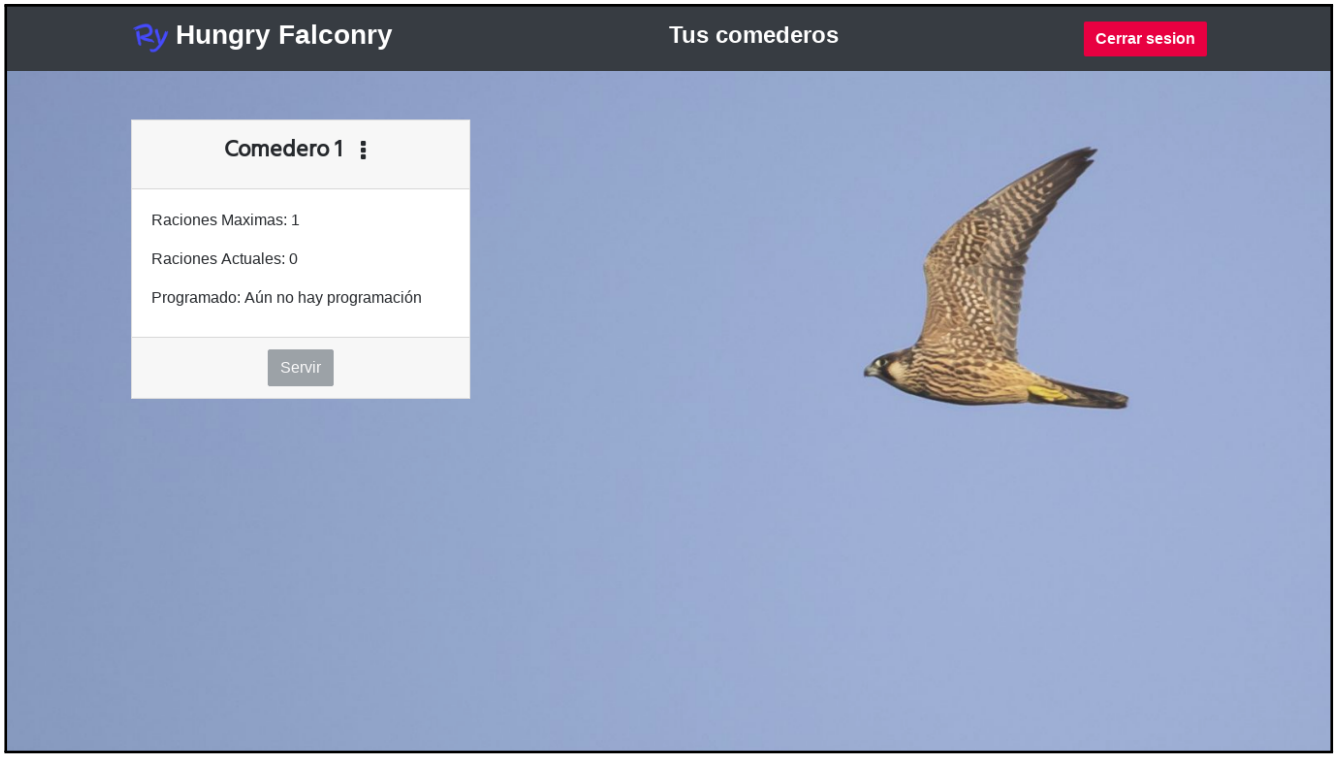


Figura 8.2: Vista de todos los comederos del usuario por primera vez

8.1.2. Editar nombre

Lo primero que se recomienda es cambiar el nombre del comedero (ver Fig. 8.3). Para ello nos dirigimos a los 3 puntos situados a la derecha del nombre del comedero al cual queremos cambiar el nombre y seleccionamos la opción de **Editar nombre**. Añadimos un nuevo nombre y damos a Guardar. Automáticamente se cambiará el nombre del comedero. El cambio también se verá reflejado en la pantalla principal.

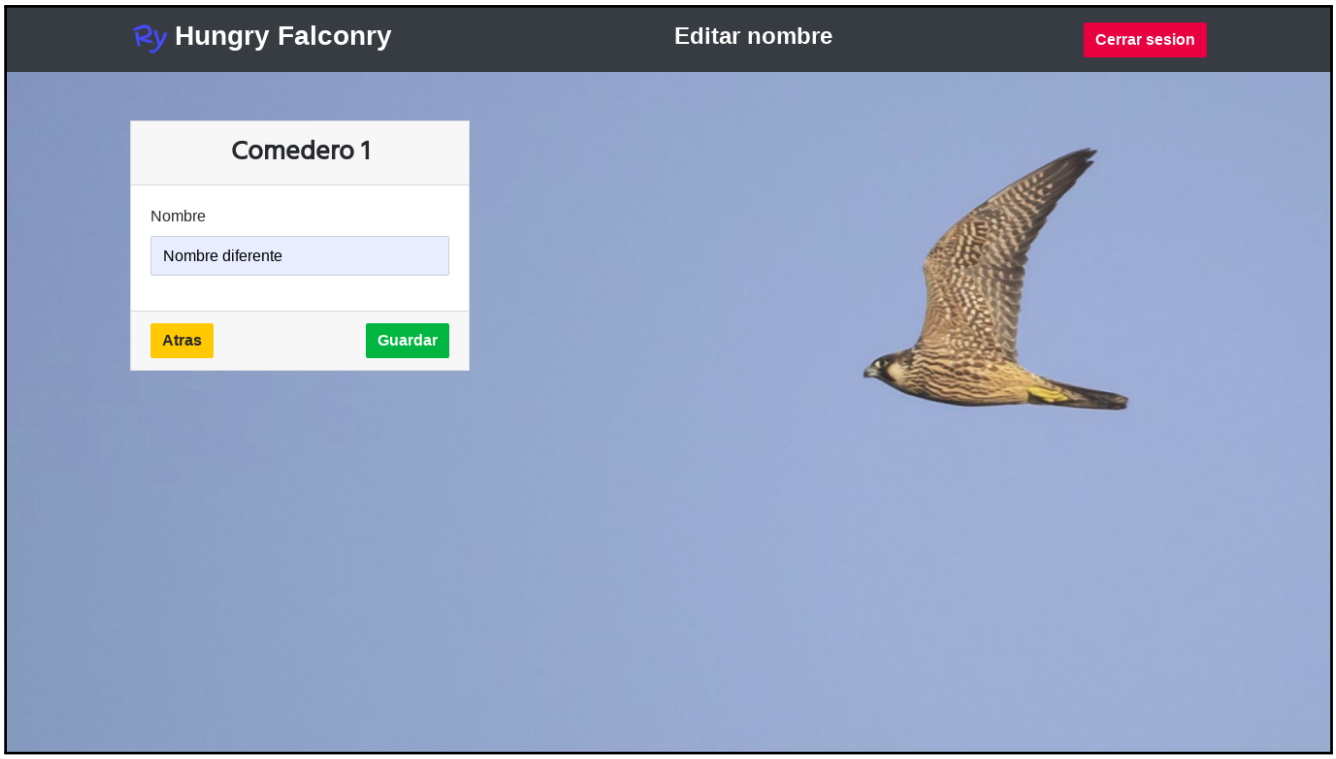


Figura 8.3: Vista edición del nombre del comedero

8.1.3. Añadir raciones

El siguiente paso a realizar sería introducir las raciones correspondientes en el comedero. Una vez colocada la comida en el comedero procedemos a registrar las raciones que hemos introducido. Para ello, nos dirigimos a los 3 puntos situados a la derecha del nombre del comedero y seleccionamos la opción de **Añadir raciones**. En el siguiente menú (ver Fig. 8.4), se completa automáticamente el campo con el número máximo de raciones que permite el comedero en cuestión. Si no hemos introducido esa cantidad de raciones en el comedero cambiaríamos esta cifra. En el caso de que el comedero tuviera raciones anteriores, se mostraría un mensaje de advertencia diciendo que se eliminarán las anteriores. Esto puede suceder por varios motivos, por ejemplo, que una de las raciones no se ha gastado y se haya tenido que retirar, por lo que el sistema no registra su consumo. Seleccionamos guardar. Ahora tenemos raciones registradas en nuestro comedero, lo que habilita las opciones de **Servir** y **Programar comedero** en nuestra pantalla principal (ver Fig. 8.5).

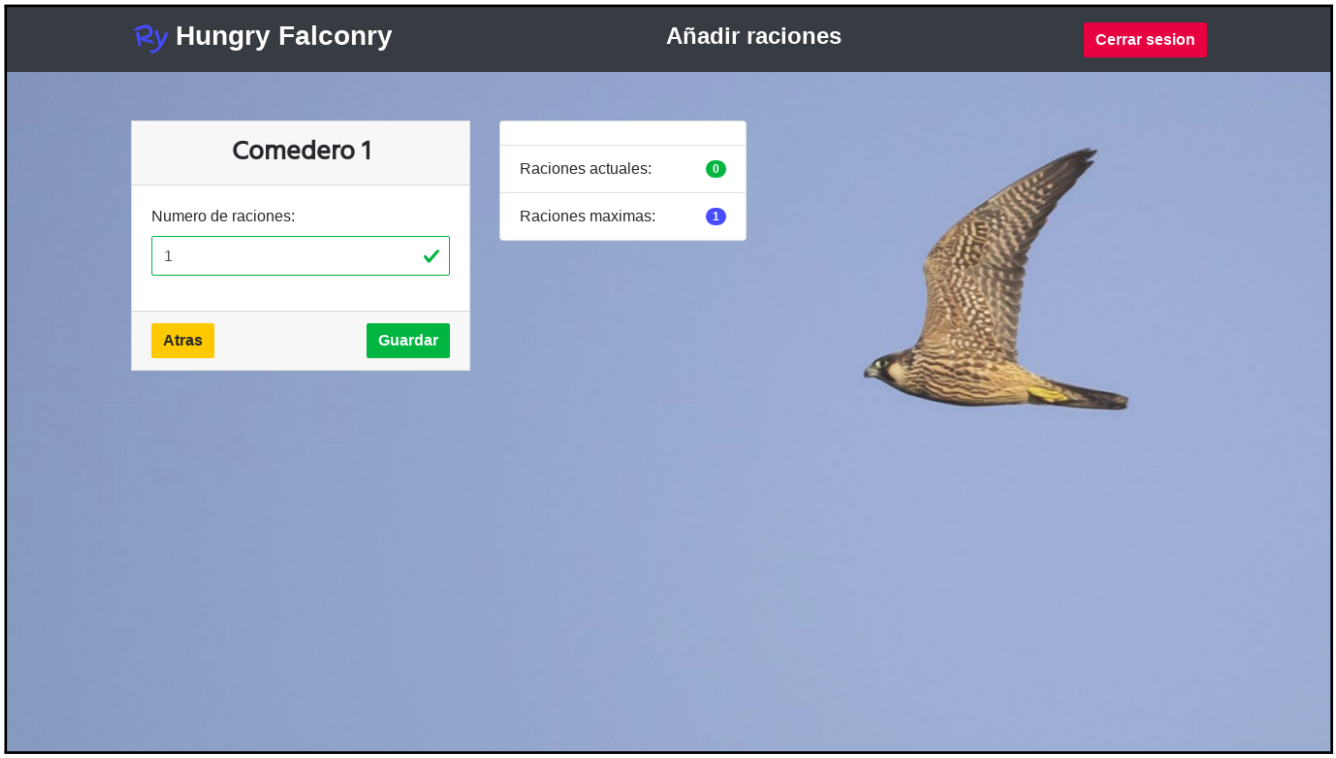


Figura 8.4: Vista añadir raciones

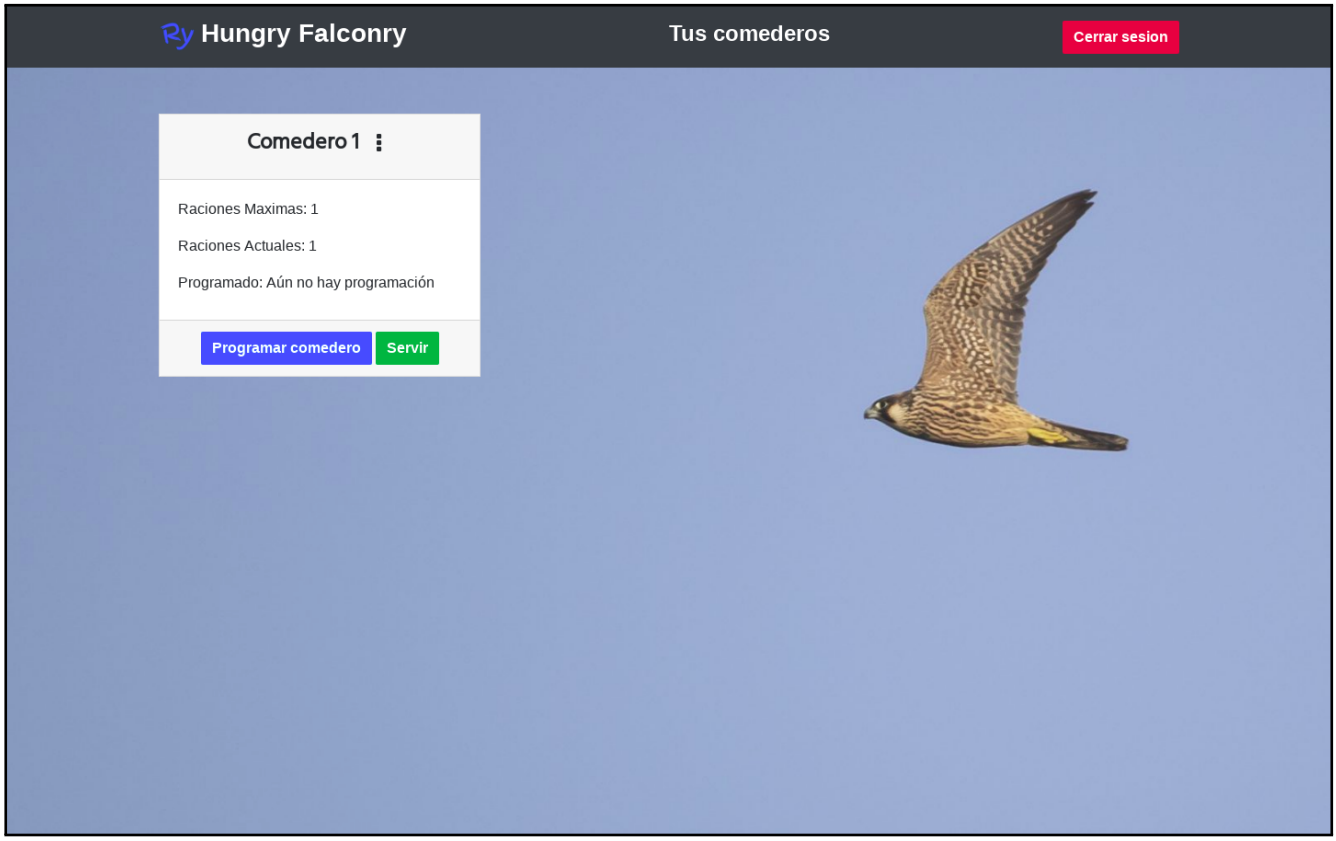


Figura 8.5: Vista todos los comederos con raciones

8.1.4. Programar comedero

Para programar el servicio de una ración nos dirigimos al menú de **Programar comedero**, desde la pantalla principal. El menú de programar comedero (ver Fig. 8.6). En la parte izquierda de este menú se observa los campos a rellenar: día y hora. En la parte derecha se observa una lista del historial de todas las programaciones, marcadas con un tick verde en caso de que se haya ejecutado y en el caso de que esté pendiente de ejecutarse con un reloj. Las programaciones que ya se cumplieron no se podrán borrar, las pendientes de ejecución sí.
Para añadir una nueva programación elegimos el día gracias a un calendario y la hora gracias a un reloj digital, ambas ayudas aparecen al pulsar dentro del recuadro. Una vez añadida una nueva programación aparecerá en la lista de la derecha. Esta nueva programación se podrá borrar pulsando en el icono de la papelera (Fig. 8.7). Además todas las programaciones pendientes de ejecutarse se muestran en la pantalla principal.

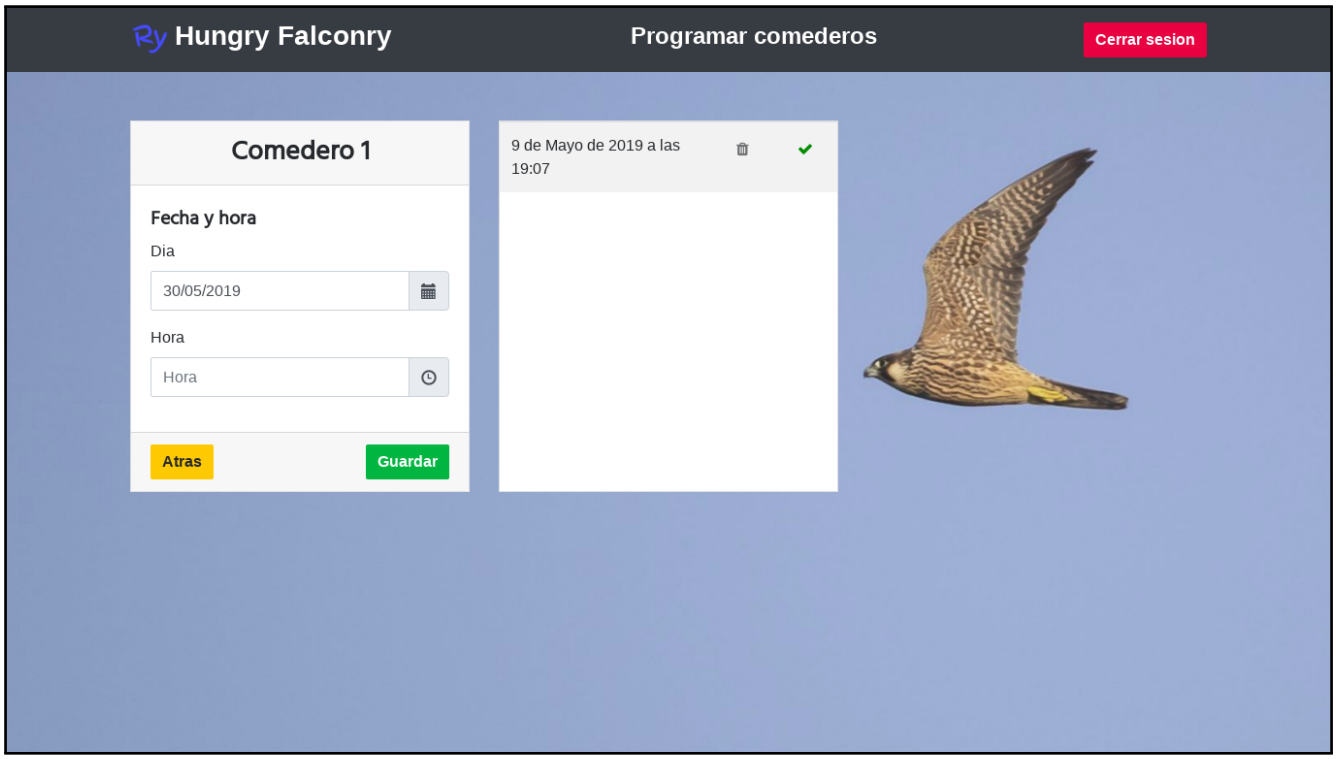


Figura 8.6: Vista programar comederos

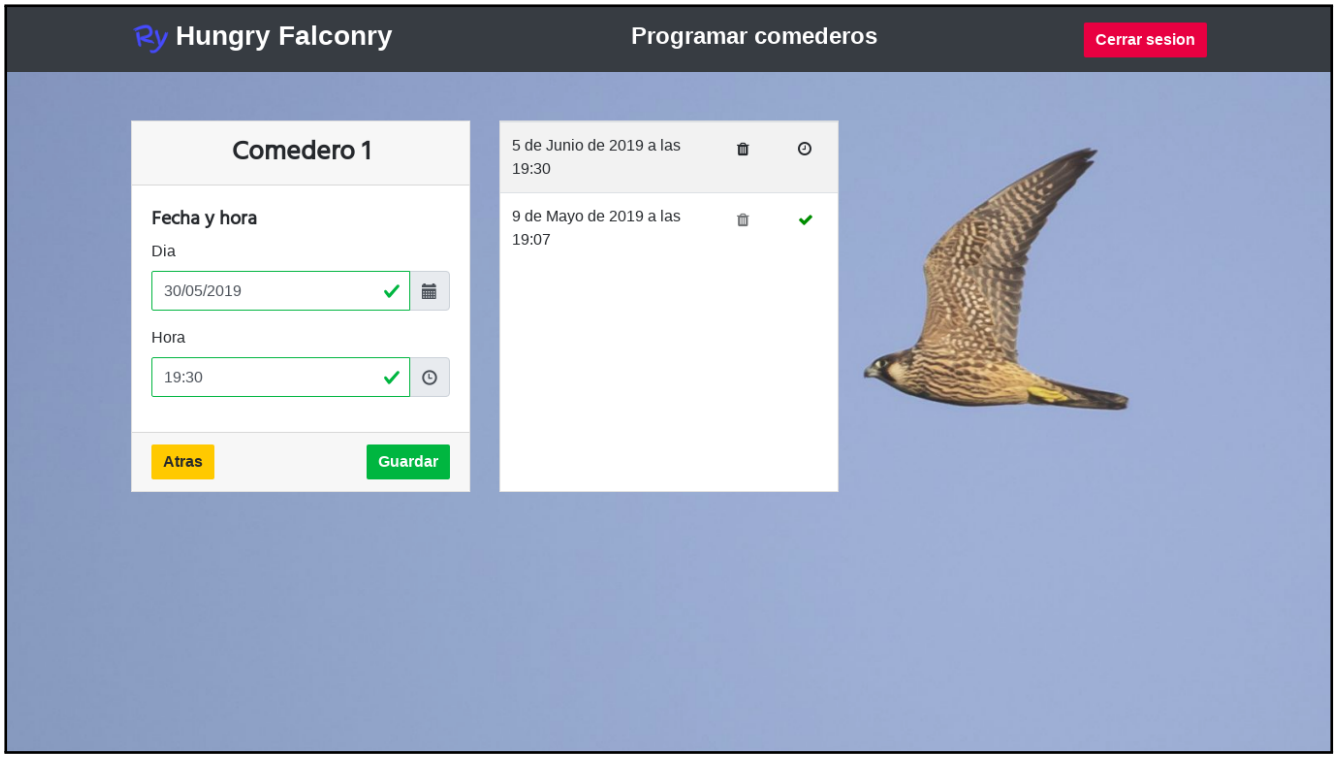


Figura 8.7: Vista programar comederos con nueva programación

8.1.5. Borrar programación

Para borrar una programación nos dirigimos al menú de **Programar comedero** (ver Fig. 8.8) y allí pulsamos la papelera asociada a la programación que queremos eliminar. Nos mostrará un mensaje de confirmación y en el caso de aceptar (Si), se borrará la programación.

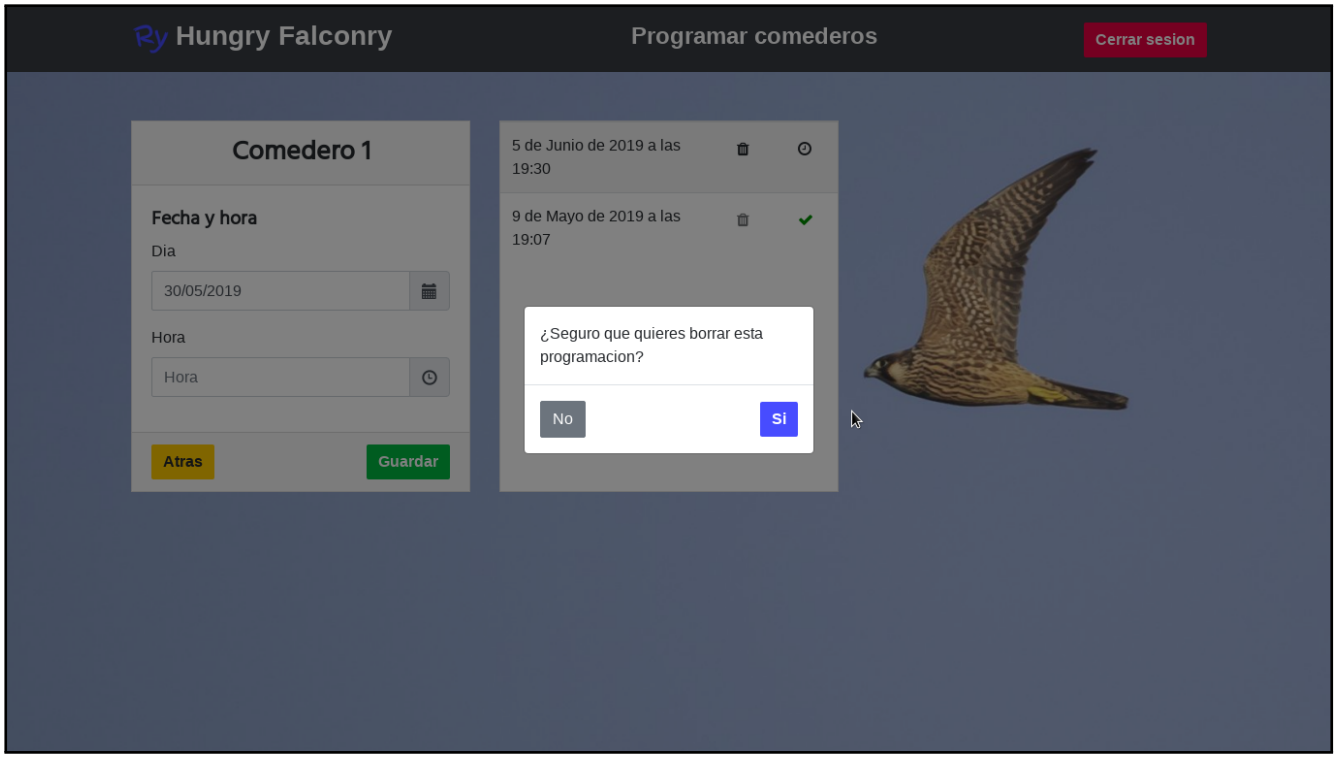


Figura 8.8: Vista programar comederos con nueva programación

8.1.6. Servir

Para servir inmediatamente una ración del comedero podemos hacerlo pulsando la opción de **Servir** de la pantalla principal. Sólo se podrá servir si hay raciones introducidas en el comedero y en el caso de que el número de programaciones no sea igual al número de raciones que guarde el comedero, es decir, si hay una programación por ración no se podrá servir.

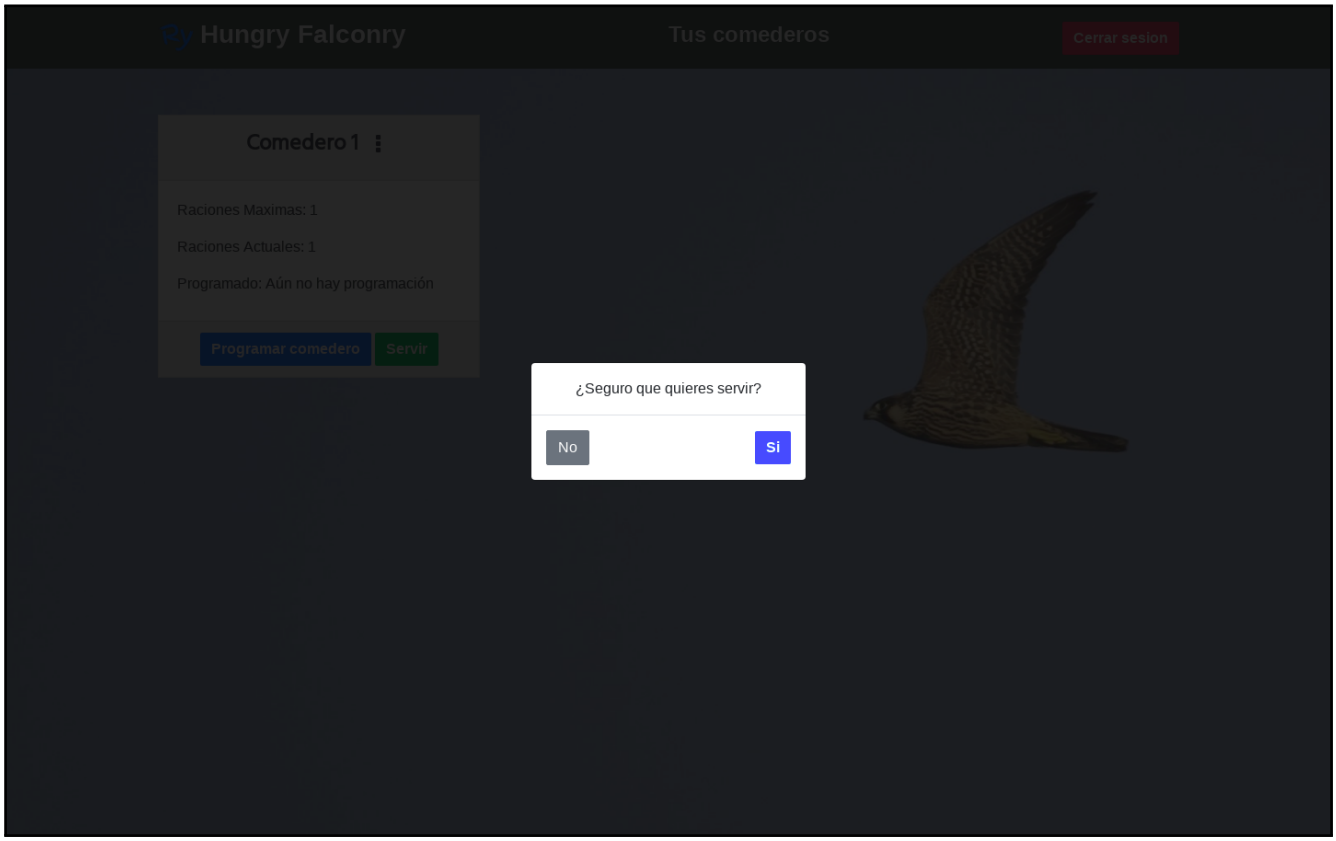


Figura 8.9: Vista confirmación cuando se sirve

8.2. Usuario administrador

El menú de administración es accesible desde `http://davidelavarga.pythonanywhere.com/admin`.

8.2.1. Iniciar sesión

El administrador iniciará sesión (ver Fig. 8.9) y se le mostrará el panel de control (ver Fig. 8.10). En el panel de control se observa toda la funcionalidad que el administrador puede modificar: Grupos, Usuarios, Comederos y Programación. Los Tokens son necesarios para la autenticación. Y a la derecha del panel de control se guarda un historial de acciones que va realizando el administrador.

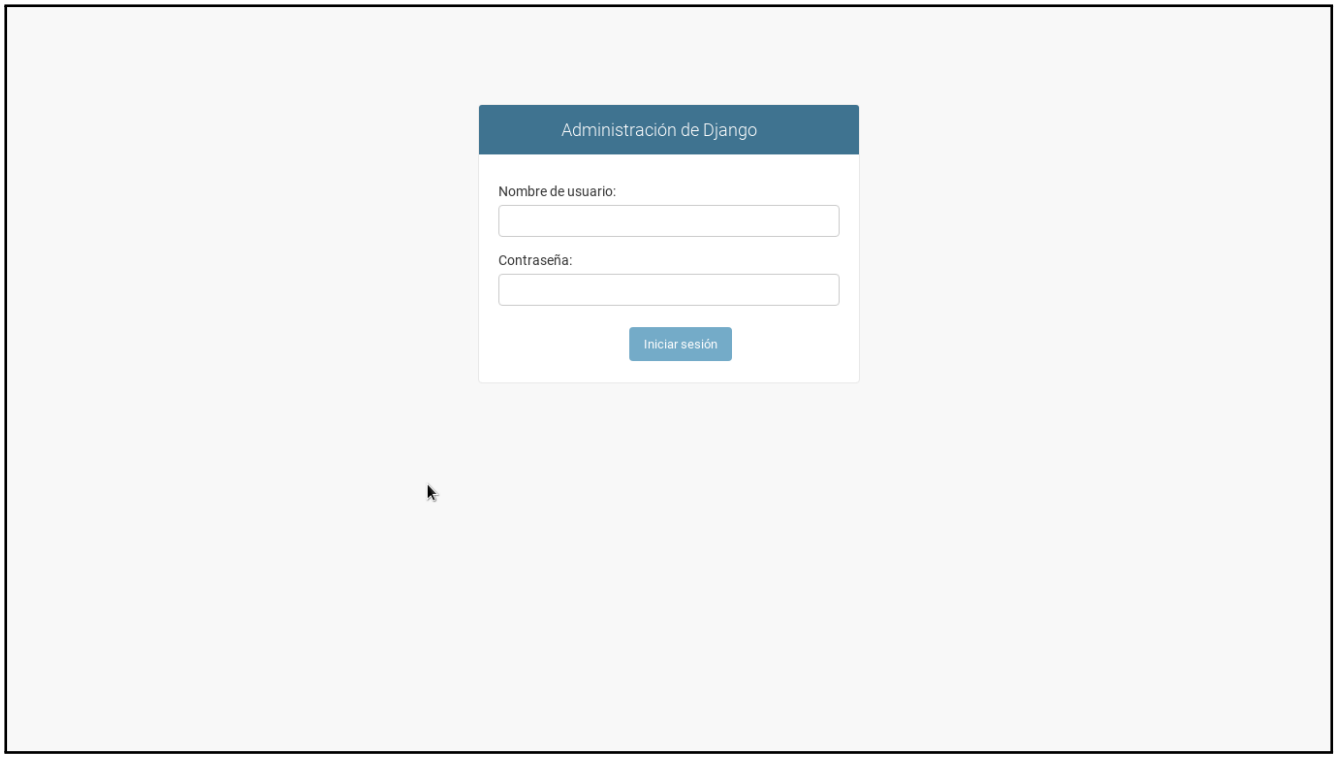


Figura 8.10: Vista inicio de sesión para Administrador

8.2.2. Crear Usuario

Nos dirigiremos a la creación de Usuarios presionando el texto de **Añadir** en la fila de Usuarios. Y se mostrará la vista de creación de Usuarios (ver Fig. 8.11). Se completan los campos con la petición correspondiente del Usuario. Se pulsa GRABAR.

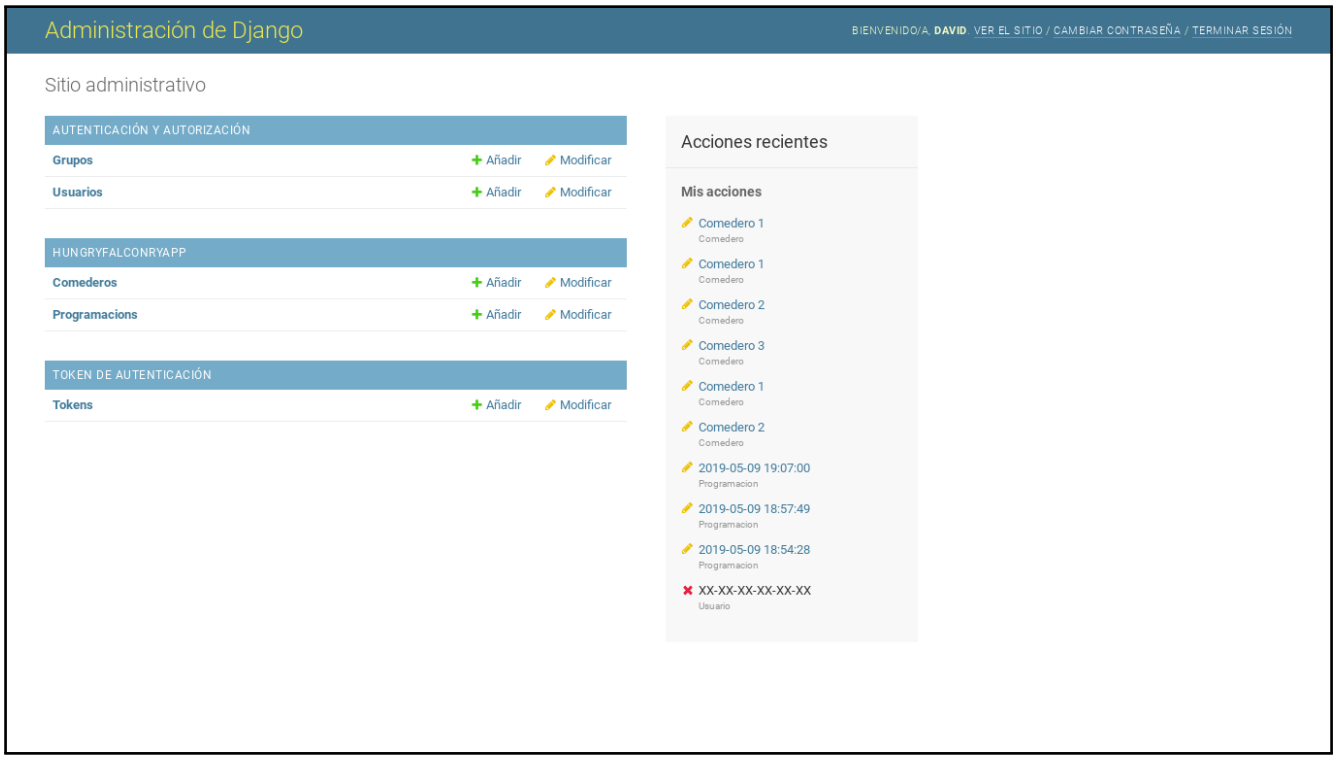


Figura 8.11: Vista panel de control para Administrador

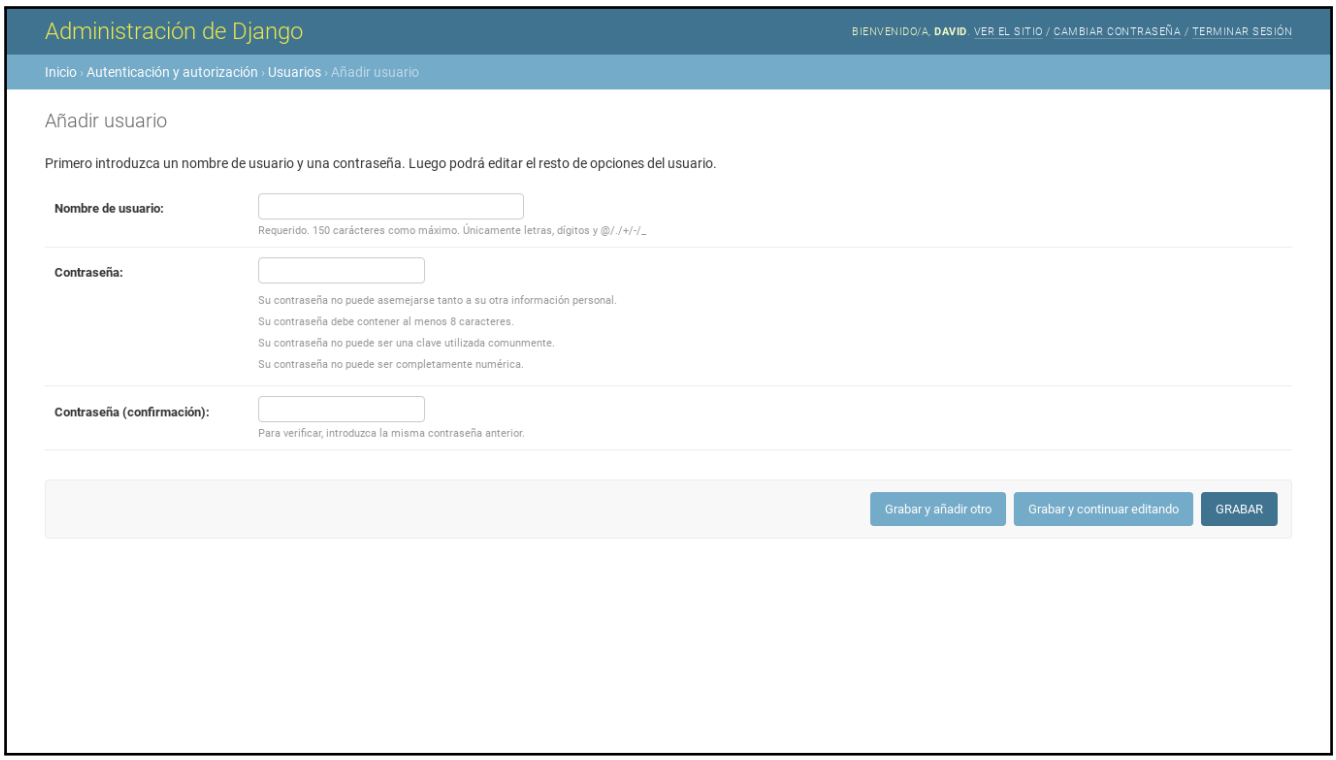


Figura 8.12: Vista añadir Usuario para Administrador

8.2.3. Crear Comedero

Ahora hay que crear el Comedero que se le proporcionará al Usuario. Nos dirigimos a **Añadir** en la fila de Comederos. Se completan los campos adecuadamente. En el campo MAC es importante poner correctamente la dirección MAC de la Raspberry Pi embebida en el comedero. Después para asociar al Usuario adecuado lo seleccionamos en el desplegable del campo titular.

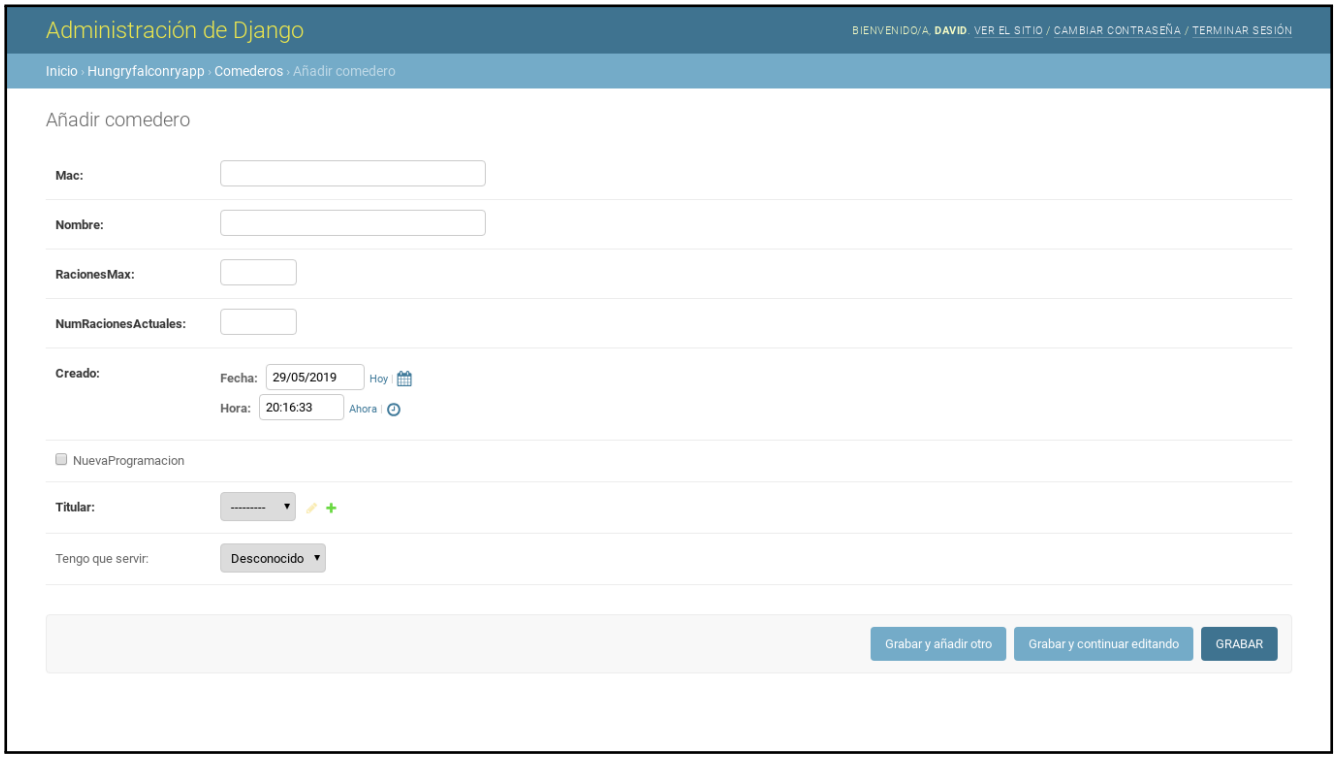


Figura 8.13: Vista añadir Comedero para Administrador

8.2.4. Generar Token

Es necesario generar un token para el usuario y añadirlo al programa que se ejecuta en la Raspberry Pi. Se realizará dirigiéndonos al apartado de Tokens y seleccionando el usuario deseado, una vez adquirido el token lo copiaremos en la línea pertinente del programa que se encuentra en la Raspberry: /home/pi/Desktop/pruebaAPIRest/simulacion_rest.py (ver Fig. 8.14 y 8.15).

```
#TOKEN de autenticacion
TOKEN = "Introducir token AQUI"
headers = {'Authorization': 'Token '+TOKEN}
```

Figura 8.14: Vista Token en el código

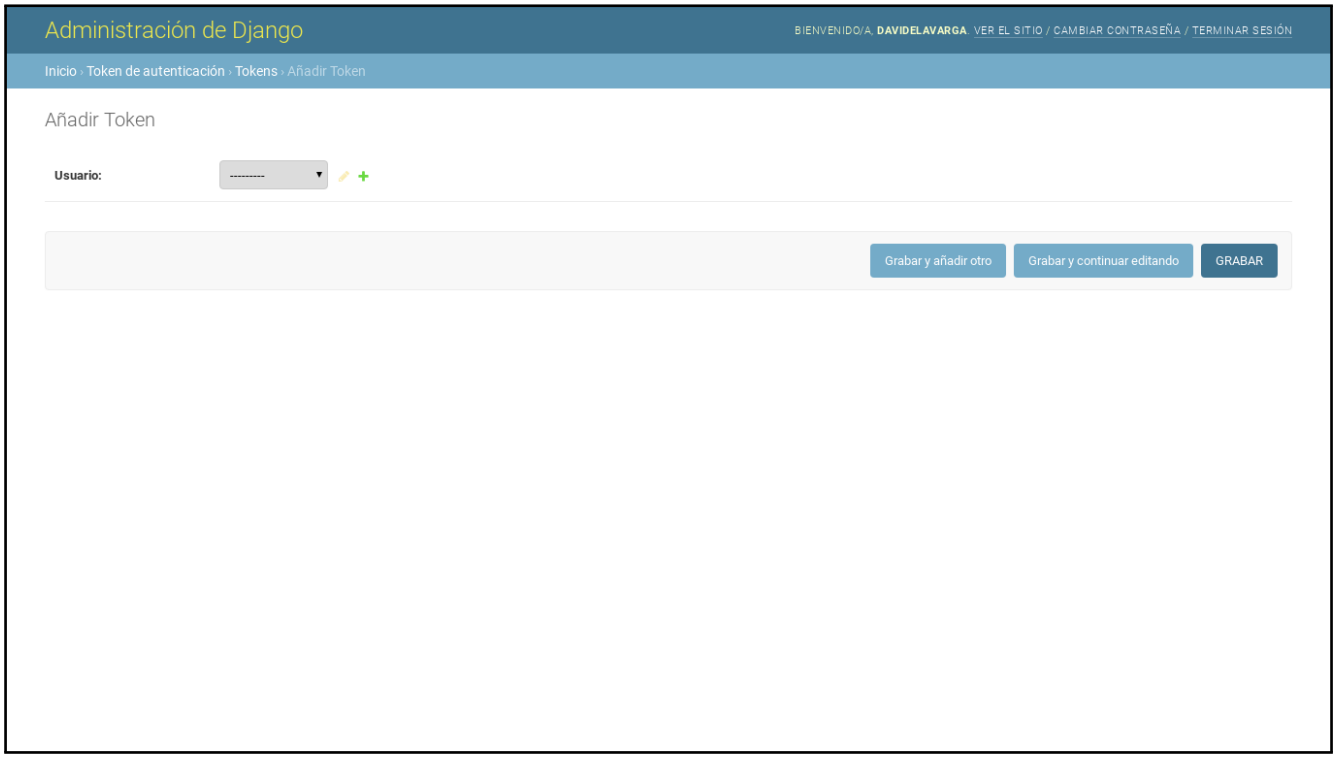


Figura 8.15: Vista añadir Token para un Usuario

Explicadas las instrucciones de uso de la aplicación, ahora se llevará a cabo la explicación de la aplicación para un desarrollador.

Capítulo 9

Manual del programador

En este capítulo se detallarán las cuestiones necesarias para que un posible futuro programador pudiera extender el sistema, bien añadiendo más funcionalidades o modificando y mejorando las actuales.

9.1. Comprensión del framework

Para poder entender y posteriormente trabajar sobre esta aplicación es necesario y primordial conocer el framework Django 2.2 [2]. Además, para la comprensión de la API Rest es necesario conocimiento sobre Django Rest Framework [9].

9.2. Estructura de ficheros de Django

La siguiente estructura está explicada, aunque a menor nivel de detalle, en el apartado 5.1 Arquitectura. La estructura de directorios que sigue es la siguiente:

hungryFalconry/
– hungryFalconry/
—— settings.py
—— urls.py
—— wsgi.py
—— __init__.py
– hungryFalconryApp/
—— migration/
—— **templates**/
———— hungryFalconryApp/
———————— add_raciones.html
———————— base.html
———————— editar_info.html
———————— programar_info.html
———————— tus_comederos.html
———— registration/
———————— login.html
—— __init__.py
—— admin.py
—— apps.py
—— forms.py
—— **models.py**
—— permissions.py

—— serializers.py
—— **urls.py**
—— **views.py**
—— tests.py
– static/
—— admin/
—— bootstrap4_datetime/
—— bootstrap_datepicker_plus/
—— css/
—— images/
—— rest_framework/
– manage.py
– requirements.txt
– db.sqlite3

Una vez conocida la estructura de directorios y ficheros que sigue el sistema, se procede a describirlos. El directorio principal, **hungryFalconry**, contiene el proyecto, las aplicaciones, la base de datos y los ficheros necesarios para arrancar y configurar el servidor.
El directorio hungryFalconry, que cuelga del anterior, contiene todo lo relativo a configuraciones (settings.py), todas las URLs del proyecto y las aplicaciones (urls.py) y todo lo necesario para el despliegue (wsgi.py).
El directorio hungryFalconryApp corresponde a la aplicación con su nombre. Dentro de él se encuentra:

- **templates/hungryFalconryApp**: contiene los HTML para la vista en el navegador. El fichero **base.html** contiene la información común a todas las plantillas, a excepción de la plantilla login.html, situada en el directorio registration/.
- **templates/registration**: HTML para la plantilla de inicio de sesión.
- **models.py**: Es una representación de las tablas de la base de datos, además cuando se aplica una migración (necesaria cuando se crea o modifica un modelo) se crea una tabla con el nombre del modelo en cuestión y una columna por etiqueta (CharField, IntegerField ...) existente en el modelo. Aquí existen los modelos: **Comedero**, **Programación** y **Usuario**. También existen algunos métodos utilizados para la validación de determinados campos de los modelos.
- **views.py**: Son los tradicionalmente denominados controladores [4]. Se han desarrollado métodos para las funcionalidades de la aplicación y clases para las de la API Rest.
  - **tus_comederos**: Recupera los Comederos del Usuario y muestra la vista de sus comederos.
  - **servir**: Para el comedero seleccionado cambia el valor de la variable tengo_que_servir a True. Gracias a este cambio y a través del consumo de la API Rest, el comedero asociado actuará en consecuencia, sirviendo una ración y devolviendo esta variable a False, también se restará una ración.
  - **add_raciones**: Para el nombre del comedero pasado como parámetro muestra la vista de añadir raciones, una vez completado el formulario guarda el número de raciones nuevas.
  - **editar_info**: Para el nombre del comedero pasado como parámetro muestra la vista de editar información y una vez completado el formulario modifica el nombre del comedero.
  - **programar_comederos**: Para el nombre del comedero pasado como parámetro muestra la vista de programar comederos, una vez seleccionado día y hora guarda una nueva

programación para el comedero. Cuando llegue el momento, el comedero, mediante el consumo de la API Rest, actuará sirviendo una ración y marcando la programación correspondiente como servida, también se restará una ración.

- **borrar_programacion**: Una vez en la vista programar_comederos y seleccionando el icono de la papelera se borrará la programación.

  Toda la documentación de la API en `http://davidelavarga.pythonanywhere.com/docs` **y en el Anexo 1**

- (API Rest) **ComederoList**, para la URL comederos/ devuelve la lista de todos los comederos.
- (API Rest) **ComederoDetail**, para las acciones de POST, PUT y PATCH sobre los comederos.
- (API Rest) **ProgramacionList**, para la URL programacion/pk/get devuelve las programaciones relativas al comedero con dicha pk (Clave primaria).
- (API Rest) **ProgramacionDetail** para las acciones de POST, PUT y PATCH sobre las programaciones.
- (API Rest) **UserList**, para la URL users/ devuelve todos los usuarios.
- (API Rest) **UserDetail** para la URL users/pk se pueden ejecutar las acciones de POST, PUT y PATCH del comedero con dicha pk (Clave primaria).

- **forms.py**: Aquí se crean los denominados formularios, por lo general en esta aplicación un formulario se crea a partir de un conjunto de etiquetas de una tabla de la base de datos. Cada formulario se creará a la hora de construir la template, esto sucede cuando se invoca a algún método o clase de views.py.

- **serializers.py**, se encarga de determinar que subconjunto de etiquetas de los modelos se envían a través de la API Rest. Existe una clase serializer por cada modelo: **ComederoSerializer**, **UserSerializer**, **ProgramacionSerializer**.

- **permissions.py**, define los permisos para los recursos de la API Rest. **IsOwnerOrReadOnly** establece permisos de lectura para los recursos y de lectura y escritura para el propietario de ese recurso. Entendiendo como propietario el usuario que se ha registrado.

- **static/**, almacena todos los archivos para el estilo y formato de los diferentes elementos de la aplicación. Para las siguientes librerías instaladas en el proyecto contiene los ficheros **.css**, **JavaScript (.js)** e **imágenes** necesarias. Las librerías instaladas son admin, bootstrap4_datetime, bootstrap_datepicker_plus y rest_framework. Los directorios **images/** y **css/** están creados por el desarrollador para la aplicación, contienen las imágenes y el estilo de la aplicación, respectivamente.

9.3. Configuración de la Raspberry Pi

9.3.1. Hardware y software

- Raspberry Pi 3 Model B+
- Micro SD 16 GB
- 2018-11-13-raspbian-stretch-full.img

9.3.2. Instalación

Para la elección del SO que se instalará la Raspberry Pi 3, desde ahora *RPi*, se ha decidido escoger el sistema operativo diseñado para este tipo de dispositivos, Raspbian. Raspbian es una distribución del sistema operativo GNU/Linux basado en Debian Strech (Debian 9.4). La versión utilizada de Raspbian para el proyecto es la más actualizada en el momento en el que se escribe este documento, en este caso consta la versión: *2018-11-13-raspbian-stretch-full*.
Para la instalación

Pasos de instalación

- Obtención de la imagen del sistema operativo.
  La imagen del sistema operativo ha sido descargada de la página oficial de Raspberry Pi:
  `https://www.raspberrypi.org/downloads/`

- Copia del archivo *.img* en la tarjeta Micro SD. Se ha insertado la tarjeta Micro SD en un ordenador con un SO GNU/Linux. Mediante una consola de comandos se ha ejecutado la siguiente instrucción:

  dd bs=4M if=2018-11-13-raspbian-stretch.img of=/dev/sdX conv=fsync

  Esta instrucción quemará la imagen del sistema operativo en la tarjeta Micro SD. El siguiente paso sería introducir la SD en la RPi y conectar un Ethernet y la alimentación.

Aclaraciones

Debido a que no se disponía de teclado y ratón para conectarlos a la RPi, se procedió a configurar una conexión SSH. Para ello:

1. Se creó un archivo SSH.txt en la partición boot de la Micro SD. El archivo se encuentra vacío.

2. Se conectó la RPi a la red local, mediante cable Ethernet.

3. Mediante la herramienta *Angry IP Scanner* se averiguó la IP que el router había asignado a la RPi.

4. Bien desde una consola de comandos, o bien, desde un programa, como puede ser PuTTY, se configuró la conexión SSH.
   IP: 192.168.1.45
   Puerto: 22

5. Se inició sesión en la RPi mediante el usuario *pi* y la contraseña *raspberry*

9.3.3. Configuración de la conexión WiFi

Una vez establecida la conexión vía red Ethernet es necesario debido a la naturaleza del proyecto configurar la conexión WiFi, para ello:

1. Nos dirigimos al archivo wpa_supplicant.conf de nuestra Raspberry:

   cd /etc/wpa_supplicant/

2. Lo editamos: sudo vi wpa_supplicant.conf

3. Añadimos lo siguiente al final del archivo:

   network={
   ssid='nombre de la red WiFi'
   psk= 'contraseña de la red WiFi'
   key_mgmt=WPA-PSK
   }

4. Reiniciamos y buscamos la IP asignada a nuestra RPi vía WiFi.

5. Configuramos una conexión SSH con PuTTY, como se ha mencionado antes. Ya podemos acceder a la RPi desde una conexión WiFi.

9.3.4. Ejecución automática del script

El script desarrollado se encuentra en /home/pi/Desktop/pruebaAPIRest/simulacion_rest.py y hay que ejecutarlo cada vez que se inicie la Raspberry Pi, esto es, poniendo la siguiente línea en el fichero /.bashrc :

/usr/bin/nohup /usr/bin/python /home/pi/Desktop/pruebaAPIRest/simulacion_rest.py &

Una vez visto todo lo necesario para que un desarrollador pudiera continuar el proyecto se mostrarán los documentos gráficos del prototipo y su funcionamiento.

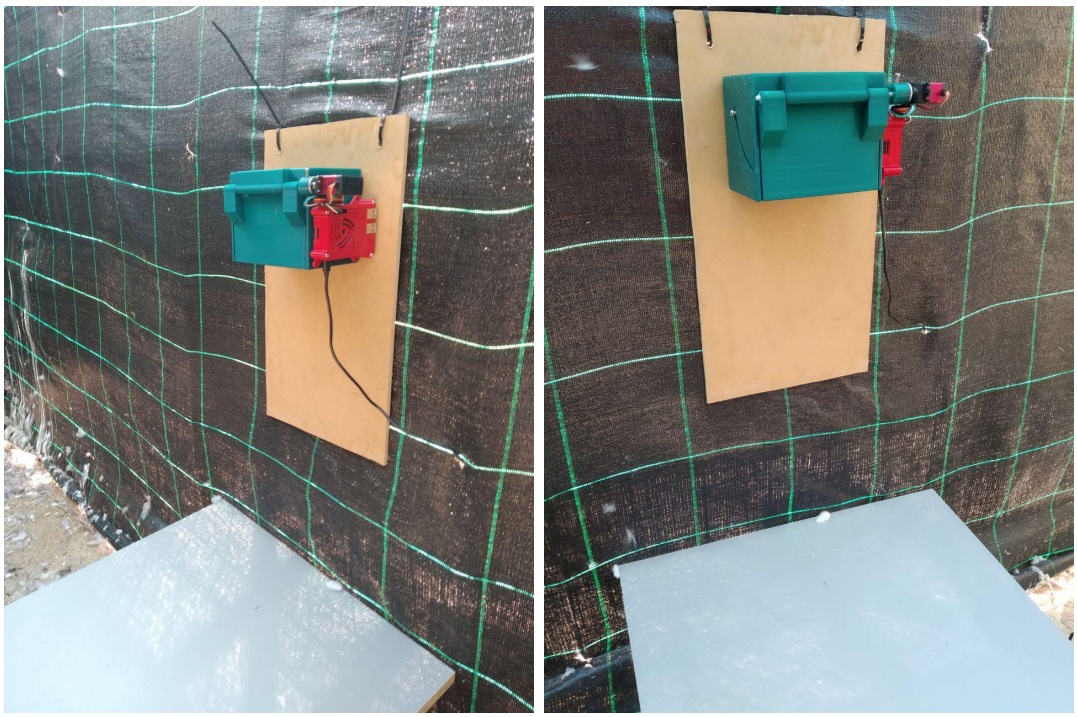
Capítulo 10

Resultados

En este capítulo se mostrarán los resultados de la fabricación del prototipo y su funcionamiento en un entorno real.

10.1. Imágenes

A continuación se muestran las imágenes del prototipo terminado e instalado en la ubicación donde se utilizó.



(a) Vista lateral izquierdo (b) Vista lateral derecho

Figura 10.1: Imágenes del prototipo

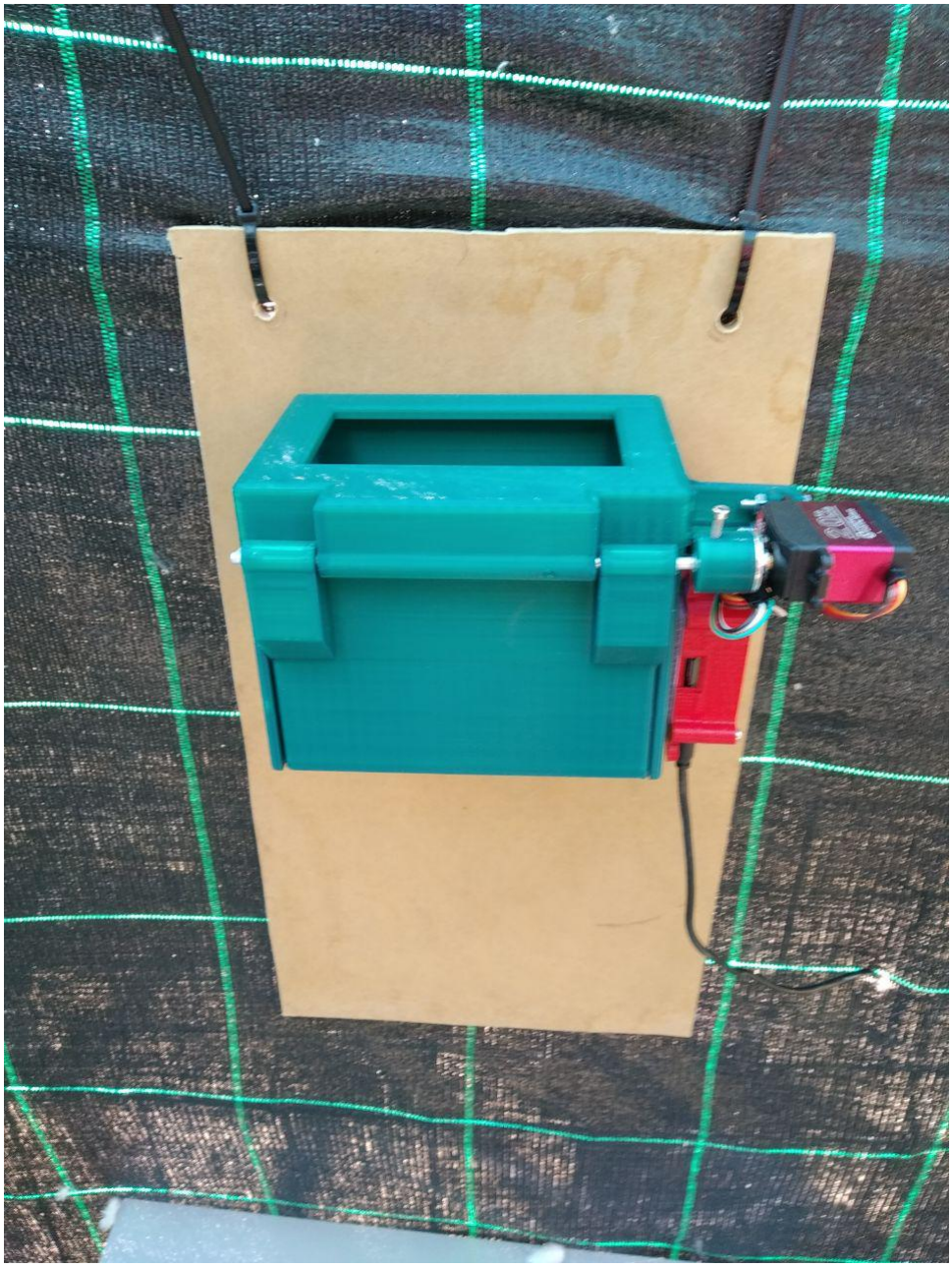


Figura 10.2: Vista frontal

Figura 10.3: Imágenes del prototipo

10.2. Vídeo

En la siguiente URL se muestra un vídeo del funcionamiento real del proyecto Hungry Falconry, probado con halcones reales. Está subido a la plataforma YouTube:

`https://youtu.be/jWsAhrH3_Xw`

Mostrados los resultados obtenidos se procede a concluir el proyecto y se detallará el trabajo futuro del mismo.

Capítulo 11

Conclusiones y trabajo futuro

En este capítulo se abordarán las cuestiones realizadas, comenzando su descripción por "Se ha hecho...", "Se ha diseñado...", etcétera. Además se añadirá el trabajo futuro para, primero mejorar y luego ampliar el sistema actual.

11.1. Conclusiones

En este TFG se ha diseñado e implementado un sistema para la gestión alimenticia de aves rapaces, además se ha fabricado un prototipo que sirve de comedero para tal función. Ambos proyectos completan una necesidad, permitiendo gestionar la laboriosa tarea de control del peso y el alimento que requiere el cuidado de estas aves. En los siguientes puntos se resume brevemente el trabajo realizado:

- Se ha realizado el análisis, el diseño, el plan de pruebas, el plan de proyecto y los manuales para usuarios y desarrolladores.
- Se ha implementado un control sobre el estado de los comederos.
- Se ha desarrollado la gestión del servicio de comida.
- Se ha desarrollado la programación de servicios de comida, manteniendo un historial.
- Se ha desarrollado la gestión de usuarios y comederos.
- Se ha desarrollado una API REST para el control remoto de los comederos.
- Se ha desarrollado un sistema de escucha activa consumiendo la API REST.
- Se ha desarrollado un sistema para el control de un servomotor a través de pines GPIO de Raspberry Pi.
- Se ha diseñado un modelo en tres dimensiones del comedero.
- Se ha imprimido en 3D el modelo antes mencionado.
- Se ha construido y ensamblado el modelo impreso.

11.2. Trabajo futuro

En esta sección se intentará mostrar las posibles mejoras y extensiones del proyecto, en consonancia con el trabajo desarrollado en este TFG. Serán ordenadas de mayor proximidad de realización a menor. Se detallan a continuación:

- Aplicación de algún tipo de envoltorio para convertir esta aplicación en una aplicación móvil.
- Desarrollo de una interfaz de gestión de usuarios más completa.
- Diseño de un modelo de comedero capaz de ser construido por módulos. Es decir, que un comedero pueda ser de una ración y luego pueda ampliarse a tres raciones fácilmente.
- Refrigeración del comedero para evitar que el alimento perezca. A través de células peltier y sistema aislante tipo PIRALU (panel de espuma rígida de poliisocianurato revestido de aluminio).
- Implementar un sistema de comunicación alternativo a la conexión WiFi, permitiendo así la conexión desde lugares remotos (3G, 4G).
- Implantación de un sensor de presión en el comedero para cerciorar que el alimento ha sido expulsado del interior cuando se ha realizado la acción de servir.

Bibliografía

[1] https://balsamiq.com/. Balsamiq mockups. Último acceso: 10 de junio de 2019.

[2] https://djangoproject.com/. Django. Último acceso: 12 de junio de 2019.

[3] https://docs.djangoproject.com/en/2.2/topics/auth/passwords/. Password management in django. Último acceso: 12 de junio de 2019.

[4] https://docs.djangoproject.com/es/2.2/faq/general/. Django parece ser un framework mvc, pero ustedes llaman al controlador «vista», y a la vista «plantilla». ¿cómo es que no usan los nombres estándares?, Marzo 2019. Último acceso: 12 de junio de 2019.

[5] https://getbootstrap.com/. Bootstrap. Último acceso: 12 de junio de 2019.

[6] https://pypi.org/. django-bootstrap-datepicker-plus. Último acceso: 12 de junio de 2019.

[7] https://pypi.org/. django-bootstrap4. Último acceso: 12 de junio de 2019.

[8] https://pypi.org. Django jquery. Último acceso: 12 de junio de 2019.

[9] https://www.django-rest framework.org/. Django rest framework. Último acceso: 12 de junio de 2019.

[10] https://www.jetbrains.com/. The python ide for professional developers. Último acceso: 12 de junio de 2019.

[11] https://www.pythonanywhere.com/. Host, run, and code python in the cloud! Último acceso: 12 de junio de 2019.

[12] https://www.sqlite.org/index.html. Sqlite. Último acceso: 12 de junio de 2019.

Anexos

Anexo I

Apéndice A

Documentación API Rest

Hungry Falconry API Rest

```
# Install the Python client library
$ pip install coreapi
```

api-token-auth

create

POST `/api-token-auth/`

Interact

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`username` **required**	Valid username for authentication
`password` **required**	Valid password for authentication

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["api-token-auth", "create"]
params = {
    "username": ...,
    "password": ...,
}
result = client.action(schema, action, params=params)
```

comederos

list

GET `/comederos/`

Interact

Devuelve una lista de todos los Comederos.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "list"]
result = client.action(schema, action)
```

create

POST `/comederos/`

Interact

Devuelve una lista de todos los Comederos.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`mac` **required**	
`nombre` **required**	
`tengo_que_servir`	
`numRacionesActuales` **required**	
`nuevaProgramacion`	

Hungry Falconry API Rest

- api-token-auth
- comederos
- programacion
- users

- Authentication: none
- Source Code: python
- shell
- javascript
- python

```python
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "create"]
params = {
    "mac": ...,
    "nombre": ...,
    "tengo_que_servir": ...,
    "numRacionesActuales": ...,
    "nuevaProgramacion": ...,
}
result = client.action(schema, action, params=params)
```

read

GET `/comederos/{mac}/`

⇄ Interact

Devuelve una instancia de Comederos.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`mac` **required**	A unique value identifying this comedero.

```python
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "read"]
params = {
    "mac": ...,
}
result = client.action(schema, action, params=params)
```

update

PUT `/comederos/{mac}/`

⇄ Interact

Edita una instancia de Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`mac` **required**	A unique value identifying this comedero.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`mac` **required**	
`nombre` **required**	
`tengo_que_servir`	
`numRacionesActuales` **required**	
`nuevaProgramacion`	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "update"]
params = {
    "mac": ...,
    "mac": ...,
    "nombre": ...,
    "tengo_que_servir": ...,
    "numRacionesActuales": ...,
    "nuevaProgramacion": ...,
}
result = client.action(schema, action, params=params)
```

partial_update

Interact

PATCH `/comederos/{mac}/`

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`mac` **required**	A unique value identifying this comedero.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`mac`	
`nombre`	
`tengo_que_servir`	
`numRacionesActuales`	
`nuevaProgramacion`	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "partial_update"]
params = {
    "mac": ...,
    "mac": ...,
    "nombre": ...,
    "tengo_que_servir": ...,
    "numRacionesActuales": ...,
    "nuevaProgramacion": ...,
}
result = client.action(schema, action, params=params)
```

delete

Interact

DELETE `/comederos/{mac}/`

Elimina una instancia de Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`mac` **required**	A unique value identifying this comedero.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "delete"]
params = {
    "mac": ...,
}
result = client.action(schema, action, params=params)
```

read_0

Interact

GET `/comederos/{mac}{format}`

Devuelve una instancia de Comederos.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`format` **required**	
`mac` **required**	A unique value identifying this comedero.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "read_0"]
params = {
    "format": ...,
    "mac": ...,
}
result = client.action(schema, action, params=params)
```

update_0

Interact

PUT `/comederos/{mac}{format}`

Edita una instancia de Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`format` **required**	
`mac` **required**	A unique value identifying this comedero.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`mac` **required**	
`nombre` **required**	
`tengo_que_servir`	
`numRacionesActuales` **required**	
`nuevaProgramacion`	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "update_0"]
params = {
    "format": ...,
    "mac": ...,
    "mac": ...,
    "nombre": ...,
    "tengo_que_servir": ...,
    "numRacionesActuales": ...,
    "nuevaProgramacion": ...,
}
result = client.action(schema, action, params=params)
```

partial_update_0

Interact

PATCH `/comederos/{mac}{format}`

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`format` **required**	
`mac` **required**	A unique value identifying this comedero.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`mac`	
`nombre`	
`tengo_que_servir`	
`numRacionesActuales`	
`nuevaProgramacion`	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "partial_update_0"]
params = {
    "format": ...,
    "mac": ...,
    "mac": ...,
    "nombre": ...,
    "tengo_que_servir": ...,
    "numRacionesActuales": ...,
    "nuevaProgramacion": ...,
}
result = client.action(schema, action, params=params)
```

delete_0

Interact

DELETE `/comederos/{mac}{format}`

Elimina una instancia de Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`format` **required**	

Parameter	Description
mac required	A unique value identifying this comedero.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["comederos", "delete_0"]
params = {
    "format": ...,
    "mac": ...,
}
result = client.action(schema, action, params=params)
```

programacion

read

Interact

GET /programacion/{id}/get{format}

Devuelve una lista de todas las Programaciones de un Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
id required	A unique integer value identifying this programacion.
format required	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "read"]
params = {
    "id": ...,
    "format": ...,
}
result = client.action(schema, action, params=params)
```

create

Interact

POST /programacion/{id}/get{format}

Devuelve una lista de todas las Programaciones de un Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
id required	A unique integer value identifying this programacion.
format required	

Request Body

The request body should be a "application/json" encoded object, containing the following items.

Parameter	Description
dia	
hora	
comedero required	
servida	

Hungry Falconry API Rest

- api-token-auth
- comederos
- programacion
- users

- Authentication: none
- Source Code: python
- shell
- javascript
- python

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "create"]
params = {
    "id": ...,
    "format": ...,
    "dia": ...,
    "hora": ...,
    "comedero": ...,
    "servida": ...,
}
result = client.action(schema, action, params=params)
```

read_0

GET `/programacion/{id}/set{format}`

Interact

Devuelve una instancia de Programaciones.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.
`format` **required**	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "read_0"]
params = {
    "id": ...,
    "format": ...,
}
result = client.action(schema, action, params=params)
```

update

PUT `/programacion/{id}/set{format}`

Interact

Edita una instancia de Programacion.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.
`format` **required**	

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`dia`	
`hora`	
`comedero` **required**	
`servida`	

```python
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "update"]
params = {
    "id": ...,
    "format": ...,
    "dia": ...,
    "hora": ...,
    "comedero": ...,
    "servida": ...,
}
result = client.action(schema, action, params=params)
```

partial_update

Interact

PATCH `/programacion/{id}/set{format}`

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.
`format` **required**	

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`dia`	
`hora`	
`comedero`	
`servida`	

```python
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "partial_update"]
params = {
    "id": ...,
    "format": ...,
    "dia": ...,
    "hora": ...,
    "comedero": ...,
    "servida": ...,
}
result = client.action(schema, action, params=params)
```

delete

Interact

DELETE `/programacion/{id}/set{format}`

Elimina una instancia de Programacion.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.
`format` **required**	

Hungry Falconry API Rest

- api-token-auth
- comederos
- programacion
- users

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "delete"]
params = {
    "id": ...,
    "format": ...,
}
result = client.action(schema, action, params=params)
```

get > list

Interact

GET `/programacion/{id}/get`

Devuelve una lista de todas las Programaciones de un Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "get &gt; list"]
params = {
    "id": ...,
}
result = client.action(schema, action, params=params)
```

get > create

Interact

POST `/programacion/{id}/get`

Devuelve una lista de todas las Programaciones de un Comedero.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`dia`	
`hora`	
`comedero` **required**	
`servida`	

- Authentication none
- Source Code python
- shell
- javascript
- python

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "get &gt; create"]
params = {
    "id": ...,
    "dia": ...,
    "hora": ...,
    "comedero": ...,
    "servida": ...,
}
result = client.action(schema, action, params=params)
```

set > read

GET `/programacion/{id}/set`

⇄ Interact

Devuelve una instancia de Programaciones.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "set &gt; read"]
params = {
    "id": ...,
}
result = client.action(schema, action, params=params)
```

set > update

PUT `/programacion/{id}/set`

⇄ Interact

Edita una instancia de Programacion.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this programacion.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`dia`	
`hora`	
`comedero` **required**	
`servida`	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "set &gt; update"]
params = {
    "id": ...,
    "dia": ...,
    "hora": ...,
    "comedero": ...,
    "servida": ...,
}
result = client.action(schema, action, params=params)
```

set > partial_update

Interact

PATCH `/programacion/{id}/set`

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` required	A unique integer value identifying this programacion.

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
`dia`	
`hora`	
`comedero`	
`servida`	

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "set &gt; partial_update"]
params = {
    "id": ...,
    "dia": ...,
    "hora": ...,
    "comedero": ...,
    "servida": ...,
}
result = client.action(schema, action, params=params)
```

set > delete

Interact

DELETE `/programacion/{id}/set`

Elimina una instancia de Programacion.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` required	A unique integer value identifying this programacion.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["programacion", "set &gt; delete"]
params = {
    "id": ...,
}
result = client.action(schema, action, params=params)
```

users

list

Interact

GET `/users/`

Devuelve una lista de todos los Usuarios.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["users", "list"]
result = client.action(schema, action)
```

read

Interact

GET `/users/{id}/`

Devuelve una instancia de Usuario.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this usuario.

```
import coreapi

# Initialize a client & load the schema document
client = coreapi.Client()
schema = client.get("http://localhost:8000/docs/")

# Interact with the API endpoint
action = ["users", "read"]
params = {
    "id": ...,
}
result = client.action(schema, action, params=params)
```

read_0

Interact

GET `/users/{id}{format}`

Devuelve una instancia de Usuario.

Path Parameters

The following parameters should be included in the URL path.

Parameter	Description
`id` **required**	A unique integer value identifying this usuario.
`format` **required**	