



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:
Manual de Programador**

Autor:
D. David de la Varga del Caño

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

Capítulo 1

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.

1.1. Comprensión del framework

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

1.2. 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 [?]. 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.

1.3. Configuración de la Raspberry Pi

1.3.1. Hardware y software

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

1.3.2. 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 Stretch (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*

1.3.3. 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.

1.3.4. 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 &
```