Introducción:

Este informe describe la implementación de una aplicación de gestión de instancias de dispositivos IoT utilizando Django y SQLite. La aplicación permite crear, editar, borrar y visualizar modelos de dispositivos, reglas y eventos. Se establecen algunas limitaciones para cada modelo y se proporcionan capturas de pantalla de la interfaz de usuario. Además, se describe la capa lógica de la aplicación, que incluye el controlador, los dispositivos IoT y el motor de reglas, que utiliza la librería de Python MQTT para la comunicación entre los componentes.

Implementación:

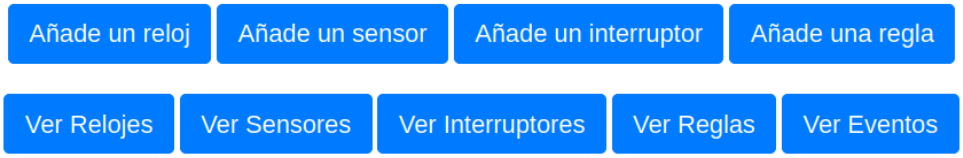
Interfaz de usuario y persistencia:

Para esta práctica se decidió utilizar Django junto con su gestión de persistencia por defecto: SQLite. Se ha construido una aplicación básica de gestión de instancias de dispositivos (Modelos), reglas y eventos. Se establecen las siguientes limitaciones para cada modelo:

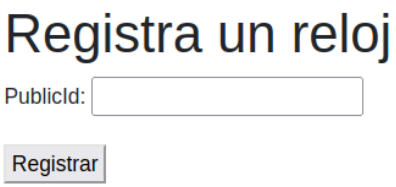
- Los interruptores:
  - Se puede crear/editar/borrar/visualizar.
  - Se puede indicar el estado y PublicId de un sensor en su creación y modificación.
- Los sensores:
  - Se puede crear/editar/borrar/visualizar.
  - Solo se podrá indicar el PublicID y en la creación y modificación y su estado por defecto al crearse será 0.
- Los relojes:
  - Se puede crear/editar/borrar/visualizar.
  - Los relojes no mantienen un estado, solo tienen un PublicId.
- Las reglas:
  - Se puede crear/editar/borrar/visualizar.
- Los eventos:
  - Solo se pueden visualizar, ya que se generan internamente a causa de el cambio de estado o mensaje de algún dispositivo.

Se adjuntan algunas capturas sobre la interfaz de usuario:

1. Página principal.

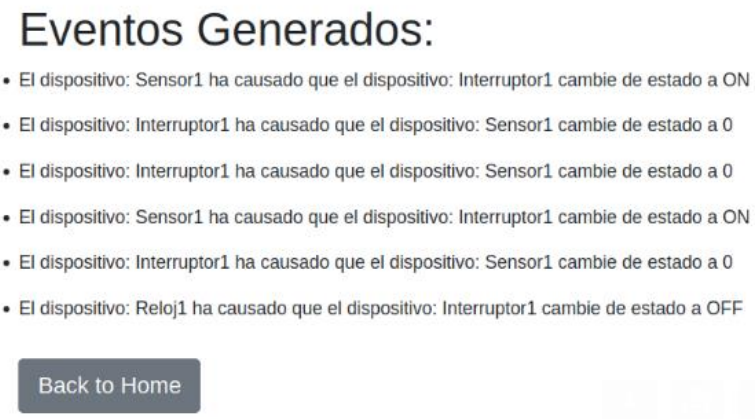


2. Registrar un reloj.



3. Visualizar un Sensor.

4. Visualizar eventos generados:



Capa Lógica:

Se ha utilizado la librería para Python de cliente de MQTT: *paho*. En la carpeta backend se pueden encontrar 5 ficheros:

- **Controller.py:** Simula un controlador que se suscribe a los topics de: RuleEngine y dispositivos que hayan sido lanzados. El controller es encargado de al recibir un evento de RuleEngine, generar la acción en el dispositivo correspondiente. De la misma manera, si recibe un mensaje de cambio de estado de algún dispositivo, llama a RuleEngine para comprobar si hay eventos y acciones que ejecutar.

Cuando se lanza Controller.py, para cada dispositivo IoT en la base de datos, se comprueba si su estado inicial dispara alguna acción sobre otro dispositivo.

- **Dispositivos IoT:** Si queremos una correcta funcionalidad tenemos que lanzar un script de Python por cada dispositivo en la base de datos, es decir, si tenemos 3 sensores en la base de datos y 1 interruptor, habrá que lanzar tres veces dummy-sensor y una vez dummy-switch junto con sus PublicIds correspondientes. Esto dado que si el cambio de estado en un dispositivo genera un cambio en otro y el dispositivo afectado no está ejecutando, no se persistirá el nuevo estado debido al cambio en la base de datos.
    - **Dummy-switch.py:** Es el dispositivo más simple, cuando se lanza asume el estado almacenado en la base de datos. Se suscribe al topic: redes2/2302/G11/switchid/get y redes2/2302/G11/switchid/get , el controlador publica en estos topics para consultar o modificar el estado de un switch. Cuando llega una petición en el topic "set" se modificará el estado con una probabilidad de éxito definida al lanzar el interruptor. Si tiene éxito se persiste el cambio en la base de datos. En caso de éxito o fallo, se avisa al controlador a través del topic: redes2/2302/G11/switchid. Los interruptores **soportan acciones** sobre su estado.
    - **Dummy-sensor.py:** A diferencia del interruptor, este dispositivo envía al controlador cada cierto tiempo (--interval) valores entre –min y –max con un incremento de –increment, definidos al lanzar el dispositivo. El sensor también tiene topics para set y get y por ende, **soportan acciones** sobre su estado.
    - **Dummy-clock.py:** Este dispositivo es stateless y por ende **no soporta acciones** de cambio de estado. Lo único que hace es enviar cada segundo una cantidad de mensajes (--rate), parámetro definido por el usuario al lanzar el dispositivo. Estos mensajes serán la hora actual del sistema por defecto con un incremento de 1 segundo. Se pueden cambiar estos parámetros al lanzar el reloj de igual manera.
- **Rule-engine.py**: Carga y parsea las reglas de la base de datos y se encarga de recibir mensajes del controlador informando de cambios de estado en sensores e interruptores, o mensajes de hora de relojes. Al recibir estos mensajes comprueba si hay alguna regla existente sobre el dispositivo, si es así, comprueba si su nuevo estado genera alguna acción sobre otro dispositivo y de ser correcto, genera un evento temporal que envía al controlador en forma de string para que este genere la acción y almacene dicho evento.
- **Reglas**: Las **reglas deben seguir necesariamente el siguiente formato** para posteriormente poder ser parseadas por el **Rule-engine.py:**

  [Dispositivo1] [comparación] [estadoX] --> [Dispositivo2] [estadoY]

  Donde:

- Dispositivo1: Es el dispositivo que genera la acción por actual estado o cambio de estado.
- Comparación: Es la comparativa que forma parte de la condición. Para cada tipo de dispositivo se establecen los siguientes que **deben seguirse necesariamente**:
  - Interruptores: *esta/está.*
  - Sensores*: igual/mayor/ menor.*
  - Relojes: *marca.*
- estadoX: estado que tiene que tener Dispositivo1 para generar una acción sobre Dispositivo2.
- Dispositivo2: El dispositivo que cambia de estado a causa de Dispositivo1.
- EstadoY: El nuevo estado al que va a cambiar Dispositivo2.

De manera general, una regla se puede interpretar de la siguiente forma: "*Si el Dispositivo1 se encuentra o cambia a un estado mayor/menor/igual/marca/está a estadoX, el Dispositivo2 cambiará a un estadoY".*

Algunos ejemplos de reglas correctamente formadas serían las siguientes:

- Reloj1 marca 08:00:00 --> Interuptor1 OFF
- Sensor1 igual 20 --> Interruptor1 ON
- Interruptor1 ON --> Sensor1 30

Es **importante** que haya espacios entre cada valor o nombre de la regla y la presencia de la flecha "-->".

- **Eventos**: Cada vez que se genere una acción sobre un dispositivo, el controlador automáticamente generará un evento que guardará en la base de datos. Estos podrán ser visibles desde la API de django.

Limitaciones:

- Si se añade un dispositivo a la base de datos mientras controller.py está ejecutando, este no se cargará automáticamente al sistema. Es necesario relanzar controller.py para poder comprobar si se dispara alguna acción debido al estado inicial del dispositivo añadido.
- Si se añade una regla a la base de datos mientras rule-engine.py está ejecutando, esta no se cargará automáticamente al sistema. Es necesario relanzar rule-engine.py para que el rule-engine tenga en cuenta la nueva regla.
- Los identificadores (PublicId) de los dispositivos deben ser únicos, no se puede tener dos dispositivos con un mismo identificador, incluso si son tipos distintos de dispositivos.
- No se puede generar acciones sobre relojes dado que no tienen un estado, sin embargo, estos si pueden desencadenar un evento.

- Si se añade una Regla que debería generar algún evento sobre un dispositivo, no se generará a menos que se vuelva a lanzar el Controller.py. Pues Controller.py es el único script de Python que comprueba para cada dispositivo, si su estado actual genera un evento llamando a RuleEngine.
- No se es posible establecer reglas concatenadas, es decir, no es posible indicar que un dispositivo genera acciones distintas y en distintos dispositivos en una misma regla, habrá que ponerlas en reglas separadas como se muestra a continuación:
    - Sensor1 igual 20 --> Interruptor2 ON
    - Sensor1 igual 20 --> Interruptor1 OFF

Requisitos:

1. Controller deberá:
    - recibir y realizar cambios de estado sobre los dispositivos a través de Broker MQTT.
    - Generar eventos.
    - Persistir cambios en los dispositivos.
    - Recibir acciones a realizar sobre dispositivos IoT.

- Rule Engine deberá:
    - Ante un evento, comprueba reglas y lanza acciones.
    - Gestionar reglas del sistema, comprueba reglas para realizar acciones y desencadenar eventos.
- Se deberá definir un formato textual que el sistema tratará.
- Se deberá poder añadir/editar/borrar reglas.
- Los eventos que se comprueban contra las reglas son efímeros, si el Rule Engine cae y se recupera, no procesa eventos antiguos.
- Se deberán poder generar eventos internos.
- Todos los dispositivos deberán comunicar su cambio de estado.
- Se deberá almacenar información sobre los dispositivos registrados, reglas y eventos generados.

Casos de uso:

Caso de uso 1: Sensor1 cambia de estado a ON y Sensor2 se apaga.

Actores involucrados: Interruptor1, Interruptor2, Controller, RuleEngine

Resumen: El Interruptor1 cambia de estado de OFF a ON debido a una modificación mediante la Api de Django o un evento previo y ocasiona que se genere una acción sobre Interruptor2, de apagarlo.

Pre-condiciones: Interruptor1 y Interruptor2 deberán estar registrados y lanzados mediante dummy-Sensor.py. Deberá haber una regla similar a la siguiente:

Sensor1 esta ON → Sensor2 OFF

Adicionalmente, Controller.py y RuleEngine.py deberán estar ejecutándose.

Post-condiciones: El Interruptor2 se apagará y se almacenará un evento en la base de datos: "*el dispositivo: Interruptor1 ha causado que el dispositivo: Interruptor2 cambie de estado a OFF".*

Curso básico de eventos:

1- El Controller detecta un cambio de estado del Interruptor1 a ON y llama a RuleEngine para comprobar si existe alguna regla que genere un evento.
2. Rule Engine revisa entre la lista de reglas parseadas, si alguna corresponde a Interruptor1. Si es así comprueba si se cumple la condición y notifica a Controller la acción a realizar sobre Interruptor2.
3. Controller deja un mensaje en el topic correspondiente de Interruptor2 para cambiar su estado a OFF y almacena el evento.
4. Si existe una instancia lanzada de Interruptor2 (dummy-switch.py), este cambiará su estado y lo persistirá en la base de datos, finalmente, avisará a Controller sobre su cambio.

Ejecución de la práctica y acceso a API:

Primero, es importante instalar los paquetes necesarios. Para esto, se adjunta un script: ***install_environment.sh*** que instala django y la librería paho, en caso de no tenerlos en el entorno de Python.

Acceso a la API de Django:

Se brinda un fichero Makefile que simplifica los comandos a realizar, para acceder a la API solo basta con ejecutar el comando:

Make runserver

Django proporcionará un enlace al cual se tiene que acceder.

Ejecución de la práctica

Existen tres maneras de ejecutar/probar la práctica:

1. simulator.sh: Si ejecutamos el comando ./simulator.sh se abrirá una nueva terminal por cada uno de los siguientes actores: rule engine, controller, Reloj1, Interruptor1, Sensor1. De esta forma se puede ver de manera más ordenada cómo funciona cada actor y los mensajes que se envían entre

sí. **Es necesario registrar previamente** los siguientes dispositivos desde la API antes de ejecutar este script, pues de lo contrario se nos avisará que los dispositivos no están registrados:

- Interruptor1 con estado OFF.
- Sensor1
- Reloj1

Adicionalmente, se recomienda añadir las siguientes reglas para poder visualizar los eventos que se desencadenan a partir de un cambio de estado en Sensor1 y Reloj1:

- Sensor1 igual 10 → Interruptor1 ON
- Reloj1 marca 08:00:20 → Interruptor1 OFF

1. test_controller.py: Si ejecutamos el comando *Python3 test_controller.py* se realizarán las siguientes acciones:

Limpia la base de datos, crea un Interruptor y sensor y una regla que involucra estos dos dispositivos: Sensor1 igual 10 → Interruptor1 ON

Posteriormente, lanza un Controller, RuleEngune, DummySwitch y DummySensor correspondientes a los añadidos a la base de datos. Si se observa la terminal, se verá como se desencadena un evento cuando Sensor1 llega al valor de 10 y le envía su estado al controlador, en consecuencia, el Interruptor1 se apagará, todo se imprimirá en la terminal.

2. test_device.py: El script es muy similar a test_controller, sin embargo también se pone a prueba un Reloj que parte desde las 08:00:00 horas y se añade una regla que involucra este reloj y el interruptor. Las reglas que se añaden son las siguientes:

Sensor1 igual 10 → Interruptor1 ON

Reloj1 marca 08:00:20 → Interruptor1 OFF

Se podrá observar en la terminal que el Interruptor1 cambia a ON cuando el Sensor1 llega a 20 y vuelve a cambiar a OFF cuando el Reloj1 envía al controlador la hora de 08:00:20

Importante: todos los cambios generados en los dispositivos en los tests, incluyendo eventos generados, pueden verse desde la API de django.

Decisiones:

- **¿Controller y Rule engine han de ser aplicaciones separadas?, ¿por qué?, ¿qué ventajas tiene una y otra opción?**

  Nosotros consideramos que si era necesario que fueran aplicaciones separadas porque implementan funcionalidades distintas y el código es más modular y fácilmente escalable si separamos ambos actores. Una ventaja que tendría que estén juntos en un mismo script

Python, es que no habría que usar topics para su comunicación, entonces posiblemente la comunicación sería mucho más rápida.

- **¿Cómo se comunican Controller y Rule engine en la opción escogida?**

  Mediante un topic de MQTT.

- **¿Tiene sentido que alguno de estos componentes compartan funcionalidad?, ¿qué relación hay entre ellos?**

  Ambos actúan en conjunto para una misma finalidad: recibir y generar acciones, eventos y comprobar reglas.

- **¿Cuántas instancias hay de cada componente?**

  Como mencionado anteriormente, el número de instancias deberá ser la misma que haya en la base de datos.

- **¿Controller y Bridge han de ser aplicaciones separadas?, ¿por qué?, ¿qué ventajas tiene una y otra opción?**

  No se ha implementado una aplicación bridge, pues se optó por la opción de django.

- **¿Cómo se comunican Controller y Bridge en la opción escogida?**

  No se ha implementado una aplicación bridge, pues se optó por la opción de django.

Conclusiones:

En conclusión, a través de este proyecto se ha logrado implementar una aplicación básica de gestión de instancias de dispositivos IoT y reglas para generar eventos y acciones entre ellos. Se utilizó el framework Django y la gestión de persistencia por defecto SQLite. Además, se empleó la librería Paho para Python como cliente de MQTT para la capa lógica de la aplicación. Durante el desarrollo del proyecto se adquirió conocimiento sobre el uso de Django, SQLite y MQTT, así como la implementación de reglas y acciones en un sistema de dispositivos IoT. En resumen, se logró una implementación básica pero funcional para la gestión de dispositivos IoT y reglas en una aplicación web.