UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Aplicación de la web de las cosas a una vivienda inteligente

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Ramón Esteve Ferrer

Director: Joan Fons i Cors

2014-2015

Resumen

El presente proyecto ilustra el desarrollo de una solución REST basada en el Internet de las Cosas. Su finalidad es dotar de una capa de accesibilidad e interoperabilidad a los dispositivos que forman una vivienda inteligente. A través de servicios web REST aplicados sobre estos recursos se logra que las funcionalidades de los mismos (encender y apagar luces, abrir o cerrar persianas o consultar datos de temperatura y humedad) sean accesibles de forma universal por aplicaciones externas al sistema. Estas aplicaciones pueden ser programas de control por voz, control por movimiento, por sonido, o aplicaciones web para dispositivos móviles. Se consigue de este modo que aplicaciones muy heterogéneas entre sí puedan interactuar con los dispositivos haciendo uso de la interfaz uniforme proporcionada por el protocolo HTTP y la filosofía REST. El proyecto finaliza con el desarrollo de un prototipo de aplicación cliente, y se detalla el modo en el que interactúa con el servidor REST de la vivienda inteligente.

Palabras clave: SmartHome, REST, API, vivienda, inteligente, recursos, interfaz, web

Abstract

The present project illustrates the development of a REST solution based on the Internet of Things (IoT). The purpose of this system is to provide a layer of accessibility and interoperability to the devices that make a smart home. Through applied REST web services on these resources we managed to make their functionalities (turn light son and off, open or close blinds or consult data concerning temperature and humidity) be a universally accessible system by external applications. These can be voice and motion control programs, movement control, sound, or web applications for mobile, making it very heterogeneous. These applications can interact with other devices using the uniform interface provided by the HTTP protocol and the REST philosophy. The project ends with the development of a prototype client application, and details about how it interacts with the REST server of a smart home.

Keywords: SmartHome, REST, API, home, smart, resources, interface, web

Tabla de contenidos

1. Introducción

El Internet de las Cosas (*Internet of Things*) es un concepto que ha ganado una gran relevancia en los últimos años. La idea básica que subyace a este concepto es la de dotar a objetos cotidianos con la capacidad de **compartir información**, comunicándose e interactuando unos con otros. La información procedente de estos objetos puede ser consumida, interpretada, consultada e incluso procesada por otros objetos. Es la facultad que estos objetos adquieren para interactuar por sí mismos con otros objetos lo que otorga a esta tecnología de grandes posibilidades de explotación.

Por supuesto, este tipo de tecnología precisa de ciertos elementos para que sea viable. En primer lugar, la información que los dispositivos transmiten debe ser **uniforme en su formato**, con el fin de facilitar la comunicación entre los diferentes elementos del sistema. Si además pretendemos que elementos externos al sistema interactúen con estos objetos el uso de una interfaz homogénea se convierte en algo indispensable. Estos elementos externos pueden ser aplicaciones cliente pensadas para interactuar con los objetos.

La **interfaz uniforme** es este elemento vertebrador que dota de la infraestructura necesaria para la comunicación entre los dispositivos internos del sistema, y que además proporciona acceso a esa información por parte de elementos externos al mismo. Es decir, para que sea posible esta comunicación entre aplicaciones clientes externas al sistema y los dispositivos que lo conforman es necesario que tanto la aplicación cliente como el servidor del sistema y los dispositivos internos compartan una misma interfaz para comunicarse. **REST** proporciona una interfaz simple y a la vez potente que permite esta comunicación entre dispositivos y aplicaciones heterogéneos a través de un **servicio web**, y utilizando un protocolo ampliamente extendido como es el **HTTP**.

1.1. Motivación

La implantación del Internet de las Cosas ofrece una nueva forma de interactuar con los objetos que nos rodean. Dentro del Internet de las Cosas, un objeto debe ser identificable inequívocamente, para que otros objetos puedan consultar su información de forma precisa. Si pensamos en el concepto tradicional de las redes de comunicación, enseguida pensaremos en una dirección IP que lo diferencie del resto de objetos del mismo tipo que ya existen en la red. En un entorno tan globalizado como el actual, para poder dotar a los millones de objetos existentes en el planeta de una dirección IP única será necesaria la implantación del protocolo ipV6.

Si observamos cómo funciona un servidor web tradicional, este esquema se puede adaptar al ámbito particular del Internet de las Cosas. Realmente el funcionamiento es muy parecido, y se puede establecer cierta analogía entre las páginas web ofrecidas por un servidor web y la información que ofrecen los dispositivos inteligentes. Pero nos falta un elemento adicional, que en el caso del servidor web tradicional ya damos por supuesto. La **interfaz de interoperabilidad** entre nuestro programa cliente y el servidor, que en el caso del internet tradicional nos lo proporciona un navegador web. En el caso del internet de las cosas, un programa cliente no podrá acceder a la información de los dispositivos si antes no le dotamos de una interfaz de comunicación capaz de interactuar con el servidor, realizando las consultas correctamente e interpretando las respuestas como se espera.

Es decir, para que la información de un objeto sea accesible, los consumidores (aplicaciones cliente) de esa información necesitarán conocer previamente cómo está representada y cómo acceder a ella. Es necesario dotar a estas aplicaciones cliente de un **estándar** que proporcione comunicación fiable con el servidor que ofrece la información de los dispositivos.

Es en el campo particular de la representación y acceso de los objetos donde la recientemente acuñada tecnología **REST** tiene un papel clave que desarrollar, ya que puede ser utilizada para definir qué son los objetos, qué información comparten en sus diferentes representaciones y establece para cada uno de ellos una identificación única, sencilla y representativa, muy parecida a las URL que hoy día usamos para navegar por la web.

Además REST proporciona una interfaz uniforme para la interoperabilidad de dispositivos lo que la dota de una versatilidad enorme. En el punto 2.3 REST se analizan los aspectos básicos de REST y en el punto 3 Diseño de aplicaciones RESTful se profundiza sobre el desarrollo de aplicaciones que cumplan con las reglas RESTful.

REST se refiere a los objetos con la denominación de **recursos**. Por lo tanto, a partir de este momento utilizaremos la denominación recurso para referirnos a cualquiera de los elementos que forman parte de la vivienda inteligente.

1.2. Ámbito

El presente trabajo se centra en la aplicación del concepto del Internet de las Cosas en el entorno concreto de una **vivienda inteligente.** Es sobre los recursos que conforman la vivienda (incluyendo a la vivienda en si misma como un recurso más) sobre los cuales se va a proceder a implantar esta tecnología.

El proyecto parte de un sistema de vivienda inteligente ya existente. Dicho sistema ha sido desarrollado en lenguaje Java y cuenta con una serie de dispositivos ya creados.

Todos los dispositivos cuentan a su vez con sus respectivas funcionalidades asociadas. La implementación del sistema de la vivienda inteligente, los dispositivos presentes en la misma y las funcionalidades de cada dispositivo quedan fuera del ámbito de este proyecto y no va a ser detallado en los puntos siguientes.

Se introducen los siguientes conceptos de notación, los cuales se desarrollan ampliamente en el punto 4. Diseño de la capa de interoperabilidad REST

- **SmartHome**: vivienda inteligente. Se identifica como el recurso principal donde se albergan el resto de recursos del sistema.
- **Device**: se trata de un dispositivo capaz de ofrecer información al sistema de la SmartHome. Un dispositivo puede ser una puerta, un medidor de temperatura, una nevera, etc. En el apartado 4.1. Identificación de los recursos se detallan los tipos de dispositivos presentes en el sistema.
- **DeviceFunctionality**: se trata de una funcionalidad ofrecida por un dispositivo (Device). Para la aplicación cliente es el recurso mediante el cual puede interactuar con el dispositivo. Las funcionalidades serán diferentes en función del tipo de dispositivo que la ofrece.

Los Devices presentes en la SmartHome de partida **carecen de la capacidad de ser accesibles desde el exterior del propio sistema**. Durante el desarrollo del presente proyecto se pretende dotar a este sistema cerrado de una interfaz **que permita a agentes externos al mismo poder interactuar con dichos dispositivos**, pero la creación del sistema que compone a la SmartHome no forma parte de los objetivos del trabajo.

Por lo tanto el ámbito real del trabajo está en un punto intermedio entre el sistema SmartHome, sus Devices y DeviceFunctionalities y las aplicaciones externas que tratan de acceder a la información ofrecida por el sistema SmartHome. En el siguiente diagrama se señala la capa de interoperabilidad en color verde.

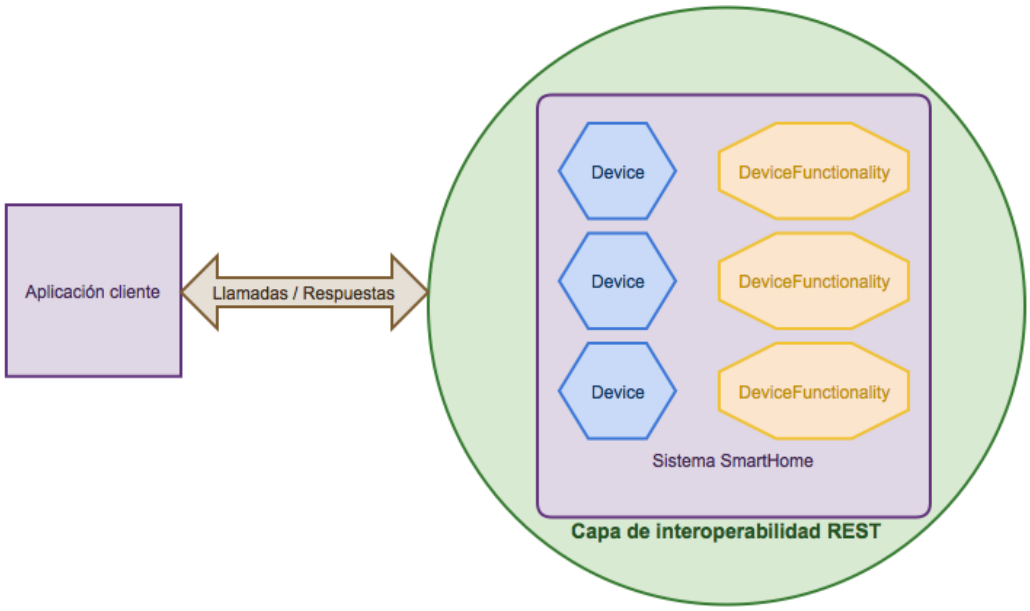

Figura 1 - Capa de interoperabilidad REST

1.3. Objetivos

Los recursos de la SmartHome cuentan con la capacidad de compartir información, a través del servidor central del sistema, pero esto no es suficiente para que la información que proporcionan sea útil. Es necesario que los recursos además sean capaces de comunicarse y de interactuar con otros recursos de la vivienda, y a su vez de ofrecer sus funcionalidades e información a dispositivos del exterior. Para que esta comunicación entre dispositivos y aplicaciones sea posible, necesitan de un mecanismo que permita **identificar a cada recurso de forma única e inequívoca** dentro del sistema de la SmartHome.

Otro de los objetivos consiste en **lograr que la información proporcionada por los recursos sea accesibles desde el exterior por dispositivos y aplicaciones externas** al sistema. Para lograrlo se hace evidente que es necesario establecer reglas sobre cómo se va a representar, ofrecer y consumir dichas funcionalidades desde el exterior, y que dichas reglas deben estar consensuadas tanto por los recursos internos como por los externos. En definitiva, es necesario **desarrollar una interfaz de comunicación común** para todos los elementos de la vivienda, y que pueda ser utilizada por un gran número de dispositivos y aplicaciones de control, independientemente de su plataforma.

El objetivo principal del proyecto es **crear una capa de servicios pensando en la integración con sistemas externo**s. Estos sistemas externos pueden ser de naturaleza tan variada como módulos de control activados por voz, interfaces de control desde navegadores web, o aplicaciones de control especialmente diseñadas para plataformas concretas, como móviles o tabletas.

Dada la misma variedad y heterogeneidad de los diferentes dispositivos y aplicaciones que pueden ser empleados para esta labor, es indudable que cobra importancia que la interfaz de comunicación con el exterior sea compatible con el mayor número de dispositivos posible. Además debe ser simple, para que dispositivos con conectividad inalámbrica lo puedan usar de forma fluida, y a la vez potente para que pueda ofrecer toda la información que el cliente precise y le permita realizar las operaciones necesarias.

Es por ello que, de entre el gran abanico de posibilidades existentes para el diseño de interfaces se ha elegido como plataforma para el desarrollo de dicha capa de interoperabilidad el modelo **REST**, y se emplea la **interfaz uniforme basada en el protocolo HTTP**, que es un protocolo ampliamente extendido y compatible con la gran mayoría de dispositivos disponibles en el mercado. A la vez REST es más **sencillo de implementar, más ligero y más compatible que otras alternativas**, como los sistemas basados en SOAP.

La meta final que se persigue es **proporcionar a aplicaciones cliente externas al sistema accesibilidad a las funcionalidades de la SmartHome** a través de la interfaz uniforme proporcionada por la capa de interoperabilidad REST, que simplifica el acceso, sea cual sea la plataforma en la que esté desarrollada la aplicación final.

Por lo tanto el objetivo principal del presente trabajo consiste en **diseñar, desarrollar e implementar una capa de interoperabilidad o API basada en REST** que pueda ser utilizada, en primer lugar por todos los recursos de la vivienda inteligente para interactuar unos con otros, y en segundo lugar por aplicaciones externas para interactuar con las funcionalidades del sistema.

El segundo objetivo es que **el desarrollo de la API REST se debe adaptar de la forma más fiel posible a las reglas y directrices que definen una aplicación REST o RESTful,** y que se detallan en el punto 3. Diseño de aplicaciones RESTful

El tercer objetivo consiste en **desarrollar un prototipo de una sencilla interfaz cliente web** con la que poder interactuar con los recursos desde cualquier navegador web, empleando la capa API REST.

1.4. Planificación

Con el fin de lograr los objetivos presentados, se planifica el desarrollo del proyecto de acuerdo con los siguientes puntos:

- **Diseño de la interfaz uniforme de la API REST**
  1. Identificación de los recursos existentes.
  2. Clasificación de los recursos según su tipo.
  3. Identificación de las operaciones que ofrece cada recurso.
  4. Establecer las reglas de construcción de las URIs para los recursos y para sus operaciones.
  5. Establecer la representación de los recursos, en base a la información que ofrecen.
  6. Establecer el formato en el que se proporcionará la información de los recursos.

- **Elección de la plataforma de desarrollo, lenguaje de programación y Frameworks necesarios.**
  1. Desarrollo de las clases Java necesarias para dotar de funcionalidad la interfaz.
  2. Despliegue de la aplicación.
  3. Pruebas y test.

- **Desarrollo de la interfaz cliente web**
  1. Implementación de las páginas HTML básicas.
  2. Implementación de las funciones JavaScript para dotar de funcionamiento dinámico a la web cliente.
  3. Implementación de una hoja de estilos para mejorar la presentación del cliente.
  4. Pruebas de ejecución del cliente contra el servidor de la SmartHome a través de la capa de interoperabilidad REST.

2. Tecnologías empleadas

Durante el desarrollo de este proyecto ha sido necesaria la utilización de diversas herramientas de programación, las cuales se resumen a continuación.

En primer lugar, el sistema SmartHome de partida al que se pretende dotar de una interfaz de acceso REST está basado en *OSGi Framework* para Java. Como ya se ha mencionado en el punto 1.2 Ámbito, el desarrollo de este sistema queda fuera del presente proyecto, y no se realiza un análisis en profundidad de su implementación interna.

El entorno de desarrollo de toda la aplicación ha sido el IDE de *Eclipse* en su versión Juno, y el lenguaje de programación escogido ha sido Java EE. El desarrollo de la propia interfaz, aplicando el modelo de desarrollo REST para Java se basa en el framework de programación orientado a Java *RESTlet*. Se ha tratado de adaptar de la forma más fiel posible la filosofía REST al desarrollo de la API de modo que pueda considerarse como una aplicación *RESTful compliance*.

En el desarrollo del prototipo de cliente web se ha empleado páginas web *HTML* en combinación con *JavaScript*. El desarrollo del prototipo de cliente se ha realizado utilizando el IDE *Eclipse Luna for Web Developers*. Como servidor web para alojar al prototipo de cliente se ha empleado la aplicación integrada *XAMPP* que incluye un servidor web *Apache*.

A continuación se detallan brevemente las tecnologías mencionadas.

2.1. OSGi Framework

Esta tecnología ha sido empleada para el desarrollo del sistema de la SmartHome sobre el cual se añade la capa de interoperabilidad REST. Dado que dicho desarrollo queda fuera de los objetivos del presente proyecto, no se realizará un análisis exhaustivo de este *framework*.

Como se menciona en [15], *OSGi Framework* es un conjunto de especificaciones que definen un sistema de componentes dinámicos para Java. Estas especificaciones permiten un modelo de desarrollo donde las aplicaciones están compuestas por diversos componentes diferentes y reutilizables. Este modelo permite a los componentes ocultar sus implementaciones a otros componentes durante la comunicación entre los mismos, que se realiza a través de servicios. Estos servicios no son otra cosa que objetos que se comparten entre los componentes de la aplicación.

El empleo de la tecnología *OSGi* permite simplificar el desarrollo de programas complejos, detectar errores de forma sencilla y aumentar la reutilización del código. Prueba de su éxito es que ha sido utilizada en el desarrollo de IDES tan conocidos como *Eclipse* o *Spring*.

2.2. Eclipse Java EE

El entorno de desarrollo *Eclipse* es una herramienta muy extendida en el desarrollo de aplicaciones basadas en Java. En el desarrollo de la API REST se ha empleado la versión Juno para Java EE. Se puede descargar de su web oficial: http://www.eclipse.org/ide

En [17] se detalla Eclipse como un entorno de desarrollo versátil y potente, el cual incorpora un gran número de funcionalidades que facilitan la tarea de programar. Entre estas funcionalidades cabe destacar la consola de entrada y salida estándar, el informe de errores y el entorno de *debug,* que permite parar la ejecución del código en las líneas que presentan problemas, teniendo acceso al valor de las variables del programa en el momento indicado.

Es compatible con muchos lenguajes de programación, y proporciona una interfaz personalizada al lenguaje utilizado. El editor de texto reconoce los comandos estándar del lenguaje empleado, y simplifica el acceso a las librerías de uso general.

Además permite añadir de forma sencilla nuevas librerías a nuestros proyectos, de forma que los métodos de estas librerías estén inmediatamente disponibles para nuestras aplicaciones.

Para finalizar, Eclipse simplifica el proceso de compilación y generación de archivos ejecutables, puesto que lo hace de forma transparente y automatizada. También permite exportar el resultado final del proyecto en diversos formatos, dependiendo del lenguaje de programación elegido.

En el caso del presente proyecto el lenguaje de programación elegido para el desarrollo de la API REST es Java por dos motivos esenciales:

- El sistema de SmartHome de partida se ha desarrollado en Java.
- El Framework de desarrollo RESTlet, que adopta las directrices de REST al desarrollo de APIS, está basado a su vez íntegramente en Java.

Como se pretende que el desarrollo de la API REST esté basado en el Framework RESTlet, y además que se adapte al sistema de la SmartHome, la elección de Java como lenguaje de desarrollo para la API era la opción más lógica.

La configuración del entorno de programación se detalla en los Anexos. Por otra parte, el proceso de incorporar el Framework RESTlet a Eclipse Java EE ha sido tan sencillo como incorporar la librería correspondiente al proyecto, y se detalla en el punto 9.1.1. Añadir las librerías externas necesarias

2.2.1. Eclipse Web Developer

Para el desarrollo del prototipo de cliente web se ha optado por una versión del IDE Eclipse orientada al desarrollo de aplicaciones web.

En el desarrollo de aplicaciones web no se suele emplear ni la consola del sistema ni el entorno de *debug*. Dado que este entorno de desarrollo se ha adaptado a la programación de aplicaciones web, el editor de código interpreta lenguajes específicos de éste ámbito, como JavaScript o HTML.

Además se ha configurado un servidor web Apache, necesario para alojar las páginas del prototipo y dotarlas de funcionalidad. La configuración de este entorno de programación y del servidor web empleado se detalla en 9.2. Anexo 2 - Configuración del cliente web

2.3. REST

REST (acrónimo de *REpresentational State Transfer, como se detalla en [9]*) es un estilo de arquitectura de sistemas en red simple que se basa en un protocolo de comunicaciones cliente-servidor cacheable y sin estado. Por lo general, ese protocolo es el **HTTP**, aunque REST no establece restricción alguna a la hora de elegir el protocolo de comunicación para nuestra aplicación.

Por lo tanto no podemos hablar de una versión específica de REST, puesto que , más que de una aplicación o de un *framework* particular, se trata de un **conjunto de reglas y directrices que indican al programador cómo desarrollar aplicaciones que se adapten a las reglas RESTful**. En el punto 3- Diseño de aplicaciones RESTful se describen ampliamente las reglas que debe seguir una aplicación para que pueda considerarse que cumple las directrices RESTful.

REST es más sencillo de usar que el conocido enfoque de **SOAP** (*Simple Object Access Protocol)*, que requiere escribir o utilizar un programa servidor (para servir datos) y un programa cliente (para solicitar datos).

En REST no es necesario el desarrollo de una compleja interfaz cliente que interprete los datos del servidor, puesto que proporciona un método universal a los

programas clientes para interactuar directamente con los métodos disponibles en los recursos. Es lo que se conoce como **interfaz uniforme**.

El estilo REST hace hincapié en que las interacciones entre los clientes y servicios se ve reforzada por tener un número limitado de operaciones (HEAD, GET, POST, PUT y DELETE). Como podemos ver, estas operaciones o métodos son los métodos estándar del protocolo HTTP.

La flexibilidad es proporcionada por la posibilidad de asignar a los recursos sus propios indicadores de recursos únicos universales (URI). Debido a que cada recurso tiene un indicador específico, REST evita la ambigüedad.

REST se utiliza a menudo en las aplicaciones móviles, sitios web de redes sociales, herramientas de *mashup* (aplicaciones web híbridas) y procesos de negocio automatizados.

2.4. RESTlet Framework

Este framework de desarrollo para Java, gratuito y descargable desde internet, pertenece al proyecto de código abierto www.restlet.org. Para el desarrollo de este proyecto se ha empleado la versión 2.1.7 para Java Standar Edition.

Se trata de una adaptación al lenguaje de programación Java de las reglas de desarrollo REST. Ha sido ideado con la intención de facilitar el desarrollo de este tipo de aplicaciones RESTful, utilizando como lenguaje de programación Java. Está compuesto por diversos componentes, siendo el más importante de ellos su API. Dicha API, ubicada en el paquete *org.restlet* del framework, proporciona todas las herramientas necesarias para desarrollar aplicaciones y APIS al estilo RESTful. En [10] se establecen algunas de las funcionalidades más importantes que aporta:

- **La interfaz Uniforme**: establece las reglas estándar para interactuar con los recursos a través de un sistema de peticiones y respuestas.
- ***Components***: son contenedores lógicos para las aplicaciones REST.
- ***Connectors***: son los que permiten las comunicaciones entre los diversos componentes REST a través del protocolo establecido, normalmente HTTP, aunque podemos utilizar otro según nuestras necesidades.
- ***Representations***: exponen el estado de un recurso REST.

Aparte de estas funcionalidades básicas, RESTlet aporta otras muchas, a las cuales denomina extensiones, y que tienen usos muy diversos, como la seguridad, la compresión de datos o el uso de la cache. Se ha empleado en el desarrollo del proyecto la extensión referente al manejo de JSON: org.restlet.ext.json.jar, tal y como se detalla en el punto 9.1.1 Añadir las librerías externas necesarias

2.5. XAMPP

XAMPP es un paquete de aplicaciones que integra diversos servidores orientados a ser utilizados como plataforma de aplicaciones web. Se puede descargar el instalador de su página web oficial: https://www.apachefriends.org/index.html. El paquete lo conforman estas aplicaciones:

- **MySQL Database**: Una base de datos orientada a aplicacines web. Su página oficial es https://www.mysql.com
- **ProFTPD**: Un sencillo servidor FTP. Su página oficial es http://www.proftpd.org
- **Apache Web Server**: El servidor web más extendido mundialmente. Incluye soporte para el lenguaje de programación PHP5. Su página web oficial es https://httpd.apache.org

El paquete XAMPP simplifica la instalación y configuración de estas tres aplicaciones, que por separado pueden resultar complejas de configurar. Instalando XAMPP las tres aplicaciones quedan instaladas y configuradas para trabajar de forma conjunta.

Además proporciona una interfaz unificada desde la que es posible iniciar y parar los tres servidores de forma individual o conjunta. En este proyecto tan solo se ha hecho uso del servidor web Apache.

2.6. JSON

Durante el proceso de diseño de nuestra aplicación llega un punto en el que es necesario establecer la representación de los recursos, y el formato que dichas representaciones adoptan. El tipo de formato utilizado tradicionalmente en servicios web de tipo *SOAP* para la transmisión de información es el XML.

XML es un lenguaje de marcado muy extendido, ya que presenta algunas características que en una primera instancia lo hacen adecuado para transmitir información homogénea entre aplicaciones que pueden ser muy diferentes entre sí. Las más importantes son que está basado en texto, que es relativamente fácil de interpretar y que la información contenida es independiente de la posición que ocupa dentro del texto.

Sin embargo, y como se detalla en [8], la realidad es que actualmente XML ya no es la mejor opción para su uso como intercambio de información entre aplicaciones, debido a su elevado tamaño en relación con la información real (o útil) que almacena. Esto se debe a que, aparte de la información en sí misma, XML necesita añadir a estos

datos envolturas propias de su formato (como las cabeceras y las etiquetas) que no son realmente útiles en la interpretación de la información transmitida.

El formato JSON (JavaScript Object Notation) es un formato alternativo a XML, que presenta todas las ventajas de interoperabilidad y compatibilidad del primero y que añade otras adicionales, como su reducido peso en relación con XML.

Además JSON es fácil de leer e interpretar y resulta más sencillo crear una estructura que clasifique la información en formato JSON que en su homólogo XML sin perder un ápice de interoperabilidad.

Estas cualidades lo hacen especialmente indicado para su uso para transmitir información entre aplicaciones, incluso cuando estas aplicaciones corren en entornos de redes inalámbricas, donde el tamaño de los datos transmitidos es primordial para la fluidez de la comunicación.

2.7. RESTClient

Con el fin de ayudar a la visualización y comprensión de JSON complejos se emplea la herramienta **RESTClient**, que no es más que una interfaz web de un cliente REST, y que se instala en el navegador web en forma de *plugin*.

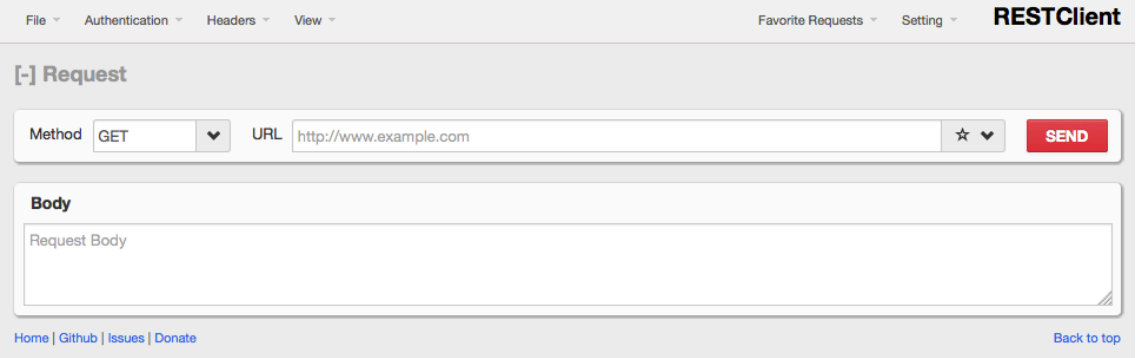

Figura 2 - Interfaz de RESTClient

La interfaz es muy sencilla. La parte principal consiste en un formulario desde el cual podemos cambiar el método empleado en la petición, introducir la URL de destino y añadir un cuerpo, si es necesario.

Un JSON generado en texto plano suele tener una estructura que dificulta su interpretación. Veamos un ejemplo, generado sobre un servidor local de test:

```
{"Name":"SmartHome Server","Status":"OK (200) - The request has
succeeded","Description":"Example SmartHome
application","URI":"http://localhost:8222/"}
```

Y el mismo JSON visualizado con RESTClient:


```
{
    "Name": "SmartHome Server",
    "Status": "OK (200) - The request has succeeded",
    "Description": "Example SmartHome application",
    "URI": "http://localhost:8222/"
}
```


Figura 3 - JSON de respuesta al GET

La representación en forma tabulada de los datos facilita su comprensión. A continuación se detallan las características más útiles que ofrece programa.

2.7.1. Seleccionar el método de la petición con RESTClient

RESTClient permite cambiar el método utilizado en nuestra petición al servidor. Normalmente desde un navegador se suelen mandar peticiones GET o POST (dependiendo del método de envió establecido en el formulario empleado para transmitir los datos), y no es habitual el envío de peticiones de tipo PUT, OPTIONS o DELETE. Pero con esta herramienta podemos fácilmente cambiar el método empleado en la petición, seleccionándolo de entre este listado:

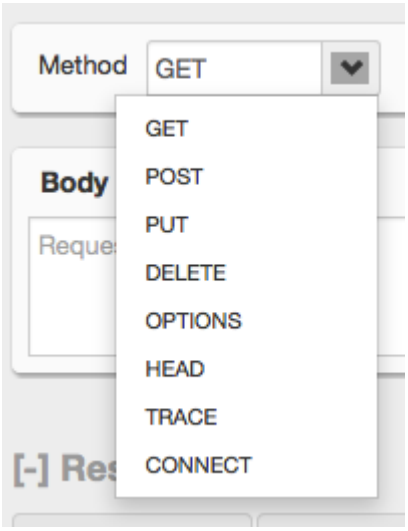

Figura 4 - Listado de métodos HTTP

Podemos ver en la imagen que el listado corresponde a la interfaz uniforme HTTP. Si queremos ver la respuesta a una petición OPTIONS a la URI http://localhost:8222

cambiaremos la petición al método OPTIONS e introduciremos la URI. El resultado será este:

```
1.  Status Code       : 200 OK
2.  Accept-Ranges     : bytes
3.  Allow             : GET, OPTIONS
4.  Content-Length    : 0
5.  Date              : Tue, 21 Oct 2014 18:25:12 GMT
6.  Server            : Restlet-Framework/2.1.7
```

Figura 5 - Resultado de OPTIONS

Lo más relevante del resultado de una operación OPTIONS, aparte de poder visualizar los métodos disponibles en el recurso consultado (GET y OPTION en este ejemplo) es que no se envía en la respuesta ningún cuerpo. No hay JSON de respuesta, ésta se envía únicamente utilizando las cabeceras HTTP, cumpliendo de esta forma con el estándar. Las operaciones disponibles se muestran en la cabecera *Allow*.

Los métodos POST, HEAD , TRACE, DELETE y CONNECT no se emplean a lo largo del proyecto, por lo que no se referencian en la interfaz uniforme HTTP empleada en la implementación de la API REST.

2.7.2. Añadir nuevas cabeceras HTTP a las peticiones

Otra de las funcionalidades interesantes de RESTClient es el poder modificar fácilmente las cabeceras de nuestras peticiones web para, por ejemplo, indicar que la información que se envía en el cuerpo de una petición PUT está en formato de JSON. El menú *Headers* de la interfaz de RESTClient permite añadir y modificar las cabeceras de nuestras peticiones de forma sencilla.

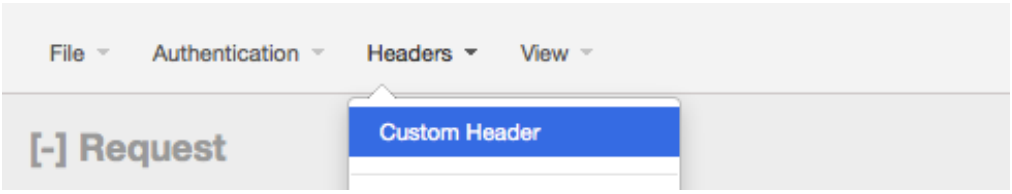

Figura 6 - Menú Headers de RESTClient

Al seleccionar la opción *Custom Header* de este menú se nos permite crear una cabecera personalizada. En el formulario que aparece podemos indicar el nombre de la cabecera y el valor que toma:

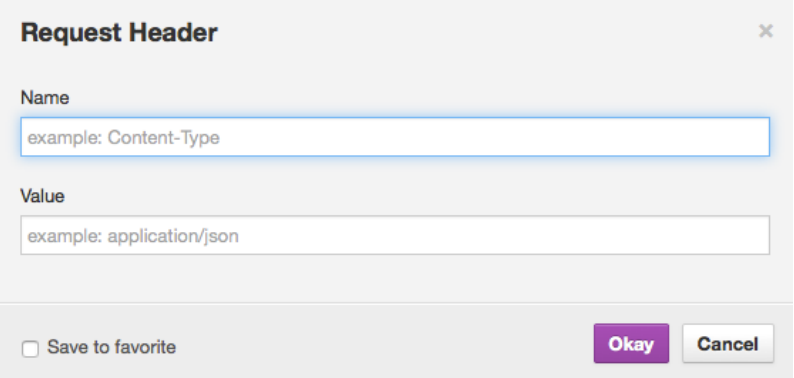

Figura 7 - Añadir una cabecera HTTP personalizada con RESTClient

En el caso concreto de una petición PUT que utilice JSON como formato para enviar la información necesitaremos añadir dos cabeceras HTTP a la petición. Son las siguientes:

- **Accept: application/json** indica los tipos de formato aceptables para la respuesta. En caso de no indicar nada, se podría asumir que el cliente acepta todos los tipos. Con este valor indicamos que en la respuesta nuestro cliente espera recibir un JSON.
- **Content-Type: application/json** indica en qué formato se envía el cuerpo de la petición al servidor. Con este valor le indicamos al servidor que el cuerpo del mensaje está en formato JSON.

Con RESTClient añadiríamos estas cabeceras de la forma descrita y con la siguiente información:

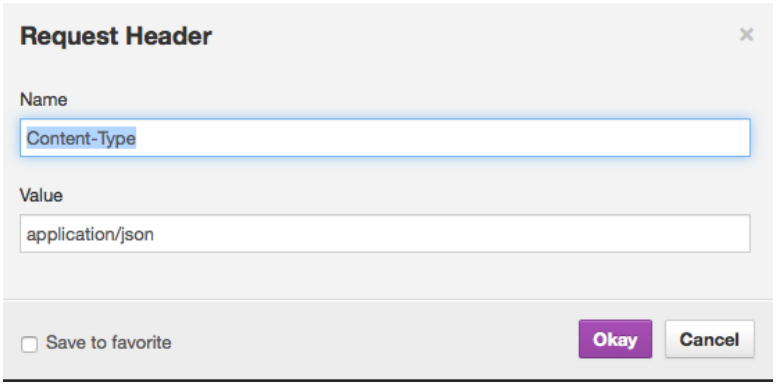

Figura 8 - Añadir la cabecera Accept

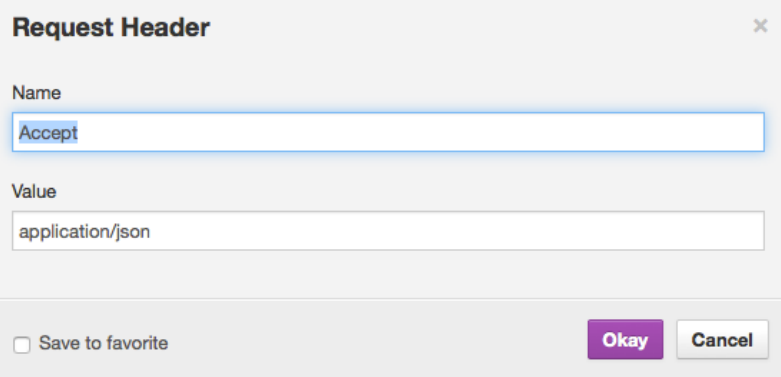

Figura 9 - Añadir la cabecera Content-Type

Otra de las ventajas de crear de este modo nuestras cabeceras personalizadas es que son almacenadas por la aplicación para su posterior posible uso en nuevas peticiones. Para añadir a nuestra petición una de las cabeceras personalizadas ya creadas solo tenemos que seleccionarla del menú *Headers*:

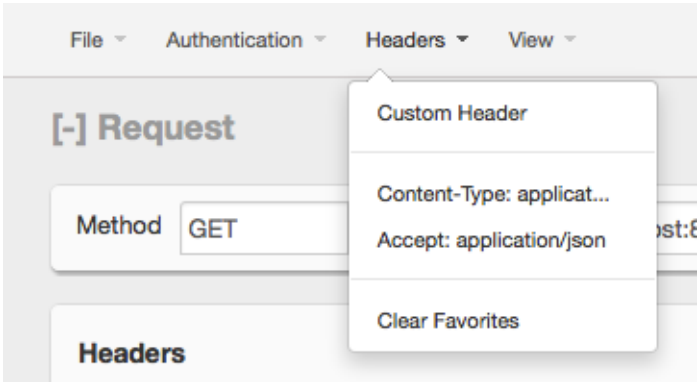

Figura 10 - Lista de cabeceras personalizadas

Cuando seleccionemos una cabecera de este listado, ésta se añadirá automáticamente a las cabeceras de las peticiones que realicemos.

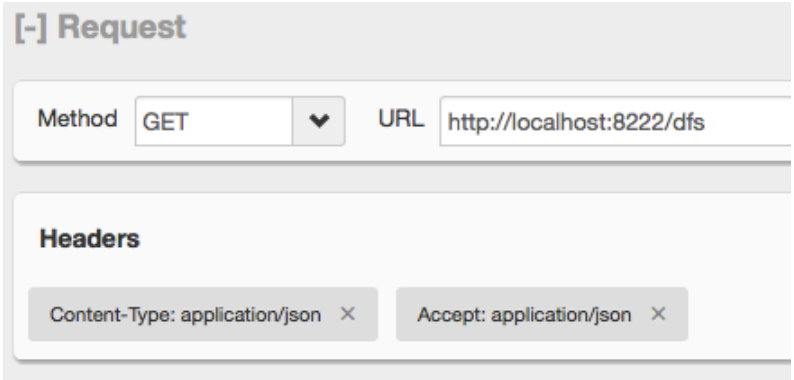

Figura 11 - Cabeceras personalizadas añadida

2.7.3. Modificar el cuerpo de la petición HTTP

Cuando se realiza una operación PUT con JSON como formato para enviar los datos, es necesario añadir al *body* de nuestra petición el JSON que contiene los datos a transmitir en la petición *(payload)*.

RESTClient permite modificar fácilmente el *body* de nuestra petición a través del campo *Body* de su interfaz. En la imagen se muestra un ejemplo en la que la petición PUT se envía acompañada de un pequeño JSON.

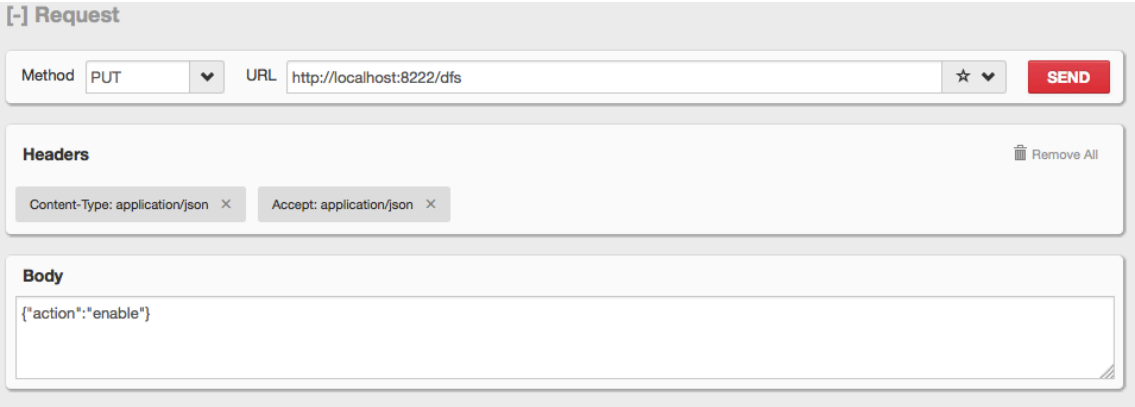

Figura 12 - Cuerpo de la petición PUT con RESTClient

2.7.4. Mostrar el resultado de la petición

Una vez realizada la petición RESTClient permite ver el resultado de la misma en el apartado Response de su interfaz. La información se divide en diferentes aparatados, según el tipo de datos mostrado. En concreto es posible visualizar:

- Las cabeceras HTTP de la respuesta
- El *body* de la respuesta en texto plano
- El *body* de la respuesta coloreado y tabulado
- El *body* de la respuesta en texto plano tabulado.

Como ejemplo se expone la información mostrada al realizar una operación GET sobre la URI de entrada a la SmartHome http://localhost:8222

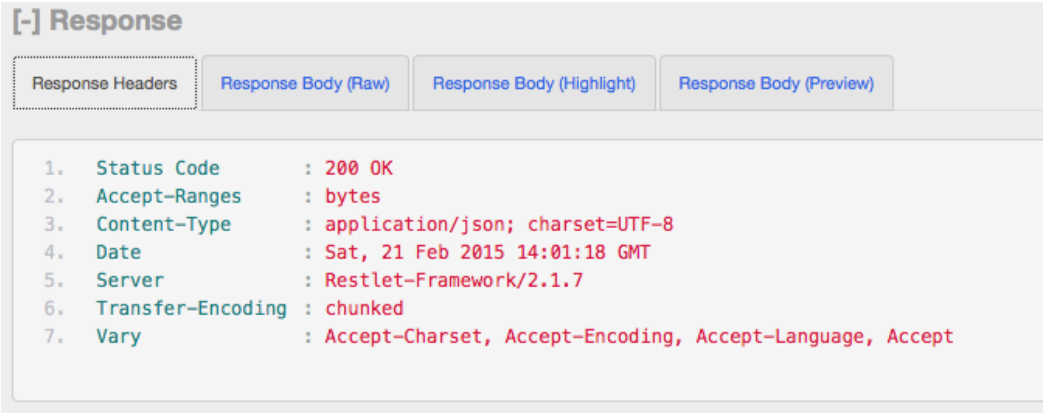

Figura 13 - Cabeceras de la respuesta GET en RESTClient

Figura 14 - JSON en formato de texto plano

Figura 15 - JSON en formato resaltado

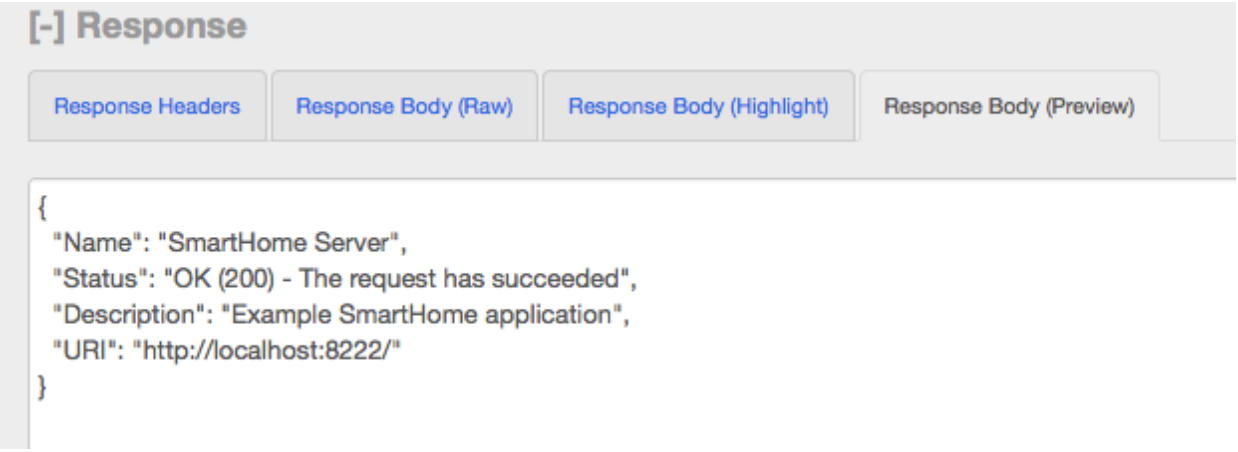

Figura 16 - JSON en formato texto tabulado sin resaltar

3. Diseño de aplicaciones RESTful

Como ya se ha comentado, REST es un conjunto de principios que definen cómo se supone que los estándares web, tales como HTTP y URI, deben ser utilizados. REST es una alternativa ligera a mecanismos como RPC (*Remote Procedure Calls*) y servicios web (SOAP, WSDL, etc.).

A pesar de ser simple, REST incorpora todas las funciones necesarias, y no hay básicamente nada que puedas hacer con Servicios Web que no se pueda hacer con una arquitectura REST.

En [9] se detalla que, al igual que *Web Services*, un servicio REST es:

- Independiente de la plataforma (no le importa si el servidor es Unix, el cliente es un Mac, o cualquier otra cosa).
- Independiente del lenguaje de programación (C# puede hablar con Java, etc).
- Basada en estándares (se ejecuta en la parte superior de HTTP).
- Se puede utilizar fácilmente en la presencia de firewalls.

REST no es un estándar. Aunque existen frameworks de programación REST (dentro de los cuales podemos enmarcar a Restlet), trabajar con REST es tan simple que a menudo se puede comparar con la forma de utilizar las funciones de la biblioteca estándar de lenguajes como Perl, Java o C#.

En [2] se detalla que, aunque REST no sea un estándar, sí hace uso de diversos estándares:

- HTTP
- URL
- XML / HTML / GIF / JSON / etc (Representaciones de recursos)
- text / xml, text / html, image / gif, image / jpeg, etc (tipos MIME)

Existen una serie de componentes clave en una arquitectura REST. El cumplimiento de los mismos determina si una determinada aplicación puede considerarse RESTful o no. En los siguientes puntos se detallan los más importantes.

3.1. Recursos

Para REST “un recurso es cualquier cosa lo suficientemente importante como para ser referenciada como un objeto en sí mismo” (Richardson, 2007, p. 80). Si ponemos como ejemplos objetos de la vida real, y nos centramos en el ámbito de una

SmartHome, podemos considerar como recursos los elementos que forman esta vivienda. Dentro de este supuesto, una puerta puede ser considerada como recurso, ya que tiene la suficiente relevancia dentro de la casa como para ser considerada recurso. Sin embargo el marco de la puerta no tiene la suficiente relevancia como para pasar a ser considerado como un recuso. Es un elemento que forma parte del recurso puerta, y para nosotros será indistinguible de la propia puerta como recurso.

Los recursos se identifican mediante direcciones URI lógicas. El estado y la funcionalidad se representan mediante los recursos. Las direcciones URI lógicas implican que los recursos son universalmente direccionables por otras partes del sistema.

Los recursos son el elemento clave de un verdadero diseño RESTful, en lugar de los "métodos" o "servicios" que se utiliza en la *RPC* y *SOAP Web Services* , respectivamente. Permiten ver los datos del producto como un recurso, y este recurso debe contener toda la información necesaria (o enlaces al mismo).

3.2. Construcción correcta de las URIs

Un recurso debe tener, al menos, una URI (*Universal Resource Identifier)*. La URI es el nombre y la dirección del recurso. Si una pieza de información no tiene una URI, entonces no es un recurso.

Existen algunas reglas básicas que deben seguirse para construir la URI de un recurso, y que se detallan en el capítulo 4 de [12]:

- Los nombres de URI no deben implicar una acción, por lo tanto debe evitarse usar verbos (métodos) en ellos.
- Deben ser únicas, una misma URI no puede identificar a más de un solo recurso.
    - ¿Puede una misma URI apuntar a varios recursos diferentes? Por definición la respuesta es que no, puesto que en ese caso los recursos apuntados por esa URI dejarían de ser inequívocamente identificables, lo que contradice totalmente las reglas REST. Una única URI no puede designar a dos recursos. Sólo puede designar a uno, ya que si designase a más de uno ya no sería un *Universal Resource Identifier*, contradiciendo su propia definición.
- Un mismo recurso puede tener varias URIS que lo identifiquen.
    - Por definición, un recurso debe tener al menos una URI pero, ¿puede un recurso tener más de una URI? O dicho de otra manera, ¿es posible que diferentes URIS designen al mismo recurso?. REST permite esto, de manera que no tenemos la rigidez de tener que acceder a un recurso siempre desde la misma URI.

    - Si un recurso tiene múltiples URIs es más sencillo para los clientes hacer referencias a ese recurso. En caso de que un recurso tenga más de una URI se recomienda tener una URI "canónica". Cuando un cliente accede al recurso a través de la URI canónica, el servidor responde con un estado HTTP 200, que significa "OK", Si en cambio se accede a través de una de sus URIs no canónicas, el servidor responde con un código 303 "See also" junto con la URI canónica.
- Deben ser independientes del formato.
- Deben mantener una jerarquía lógica.
    - Las URIs deben ser descriptivas y tener una estructura definida. Si un cliente conoce la estructura en la que están organizadas las URIs será capaz de crear puntos de entrada para el servicio de forma sencilla, siguiendo las reglas de jerarquía establecidas en el diseño.
- Los filtrados de información de un recurso no se hacen en la URI.

Estas reglas serán respetadas cuando se definan los recursos, y sus respectivas URIs, de la SmartHome.

Veamos un ejemplo, basado en uno expuesto en [9], con el que podemos comparar lo simple que resulta REST comparado con la arquitectura SOAP tradicional. Imaginemos que tenemos un conjunto de dispositivos inteligentes en una SmartHome, la cual cuenta con su propio servidor SOAP, y deseamos consultar información de un dispositivo en concreto. Los únicos datos de los que disponemos son ID del dispositivo y la URI del servidor de la SmartHome. Utilizando *Web Services* y una arquitectura SOAP la consulta devolvería algo parecido a esto:

```
<?xml version="1.0"?>

    <soap:Envelope
     xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
            <soap:body pb="http://www.mySmartHome/devices">
             <pb:GetDeviceDetails>
                    <pb:DeviceID>PUERTA.PRINCIPAL</pb:DeviceID>
               </pb:GetDeviceDetails>
            </soap:Body>
    </soap:Envelope>
```

Observemos que se devuelve un archivo XML que estará incluido en el *payload* de la respuesta con envoltura SOAP.

Sin embargo, la misma consulta con arquitectura REST devolvería una simplemente una URL como esta:

http://mySmartHome/devices/DeviceDetails/PUERTA.PRINCIPAL

Esta URL se envía al servidor mediante una petición GET simple, y la respuesta HTTP son los datos en bruto, sin incluirlos en una envoltura de nada y sin ningún tipo de filtro. Tan sólo los datos que se necesitan de una manera que se pueden utilizar directamente.

3.3. Enlaces y conectividad

En los servicios RESTful, las representaciones suelen ser *hipermedia*: documentos que contienen no solo datos, sino enlaces que proporcionan acceso a otros recursos. Un solo recurso no debe ser abrumadoramente grande. Cuando proceda, un recurso debe contener enlaces a información adicional (al igual que en las páginas web). Es decir, dentro de las representaciones que nos ofrecen los recursos se incluyen enlaces que nos conducirán a nuevas representaciones o a otros recursos relacionados.

El servidor dirige la trayectoria del cliente al servir *hipermedia*: vínculos y formularios dentro de las representaciones de hipertexto. El servidor envía al cliente directrices sobre qué recursos están cerca unos de otros. Muestra como obtener a partir del estado actual un estado relacionado.

Un servicio web está conectado en el grado en el que puedes poner el servicio en diferentes estados tan solo siguiendo enlaces y rellenando formularios. Los recursos deben enlazarse entre sí mediante sus representaciones.

Un recurso que no esté relacionado con otros recursos, y que no sea accesible mediante enlaces proporcionados por recursos adyacentes no tiene ninguna utilidad, puesto que se encontraría aislado del resto.

3.4. Sin estado

Que una aplicación REST carezca de estado significa que toda petición HTTP transcurre en absoluto aislamiento. Cuando un cliente realiza una petición HTTP añade toda la información que el servidor necesita para completarla.

El servidor nunca depende de la información de anteriores peticiones. Si esa información es importante, el usuario deberá enviarla de nuevo al servidor en la nueva petición, como se menciona en [12], capítulo 4.

La propiedad sin estado dice que los posibles estados del servidor también son recursos, y se les debe proporcionar sus propias URIs. En este sentido, un motor de

búsqueda es un servicio con un infinito número de posibles estados: uno por cada cadena de búsqueda que podamos introducir.

Es un servicio sin estado porque cada vez que un cliente hace una petición termina por volver donde ha empezado. Cada petición es independiente de las demás. Un cliente puede hacer peticiones de esos recursos cualquier número de veces, en cualquier orden, y siempre obtendrá el mismo resultado.

Retener el estado permitiría que las consultas HTTP individuales fuesen mucho más simples, pero provocaría que el protocolo HTTP fuese mucho más complicado. Eliminar el estado de las peticiones hace que se eliminen muchas condiciones de fallo. HTTP es un protocolo intrínsecamente sin estado, por lo que cuando se escriben servicios web adoptamos la propiedad sin estado por defecto. El modo más usual de romper la propiedad sin estado es utilizar las sesiones HTTP. Las cookies son *unRESTful*.

Existen dos tipos de estado: el estado de la aplicación y el estado del recurso. El estado de la aplicación existe en la parte cliente, mientras que el estado del recurso está en el servidor.

RESTful requiere que el estado permanezca en la parte del cliente, y sea transmitido al servidor en cada petición que se realice. El servidor puede dirigir al cliente hacia nuevos estados, enviando al cliente enlaces con estado para que el cliente los siga, pero no puede almacenar ningún tipo de estado.

Un mismo recurso puede tener varias representaciones (en varios idiomas, por ejemplo). En estos casos, ¿cómo averigua el servidor por cual de ellas está preguntando el cliente? La forma más simple es proporcionar una URI distinta para cada una de las representaciones del recurso.

La principal ventaja de aplicar este sistema es que la URI proporciona al servidor toda la información necesaria para atender la petición.
La desventaja consiste en que siempre que se proporcionan varias URIs para un mismo recurso hay dilución. Distintas representaciones de un mismo recurso puede hacer que parezcan recursos diferentes, cuando reamente se tratan de diferentes URIs de un mismo recurso con varias representaciones.

La alternativa es la negociación de contenido. Se proporciona una única URI canónica . Cuando un cliente hace una petición de esta URI, proporciona cabeceras de petición HTTP especiales que indican que tipo de representación está dispuesto a aceptar (preferencias de idioma del navegador web, por ejemplo).

Otros meta-datos de las peticiones pueden ser: formas de pago, credenciales de autenticación, tiempo de respuesta, directivas de *catching*, IP del cliente o formato de salida (JSON, XML, etc).

Es RESTful mantener esta información en las cabeceras HTTP, así como también es RESTful añadir estos meta-datos en las URI y, tan poco como sea posible, en los meta-

datos de la petición. Las URIs son más útiles que los meta-datos, ya que se pueden pasar de un cliente a otro cliente sin pérdida de información.

3.5. El protocolo HTTP

Para desarrollar APIs REST los aspectos claves que hay que dominar y tener claros son:

- Métodos HTTP estándar
- Códigos de estado
- Aceptación de tipos de contenido

Para manipular los recursos, HTTP nos dota de los siguientes métodos estándar con los cuales debemos operar, descritos en [5]:

- GET: Para consultar y leer recursos
- POST: Para crear recursos
- PUT: Para editar recursos
- DELETE: Para eliminar recursos.
- HEAD: Para obtener información de un recurso
- OPTIONS: Para obtener los métodos que soporta un recurso

Quizá debido al desconocimiento o el soporte de ciertos navegadores, los desarrolladores web han usado, durante los últimos años, únicamente los métodos GET Y POST para realizar todas estas acciones. Si trabajamos con REST, esto sería un error de base y puede darnos problemas incluso a la hora de nombrar nuestros recursos, obligándonos a poner verbos en las URLs, y por tanto incumpliendo la estructura de REST.

Por otra parte, uno de los errores más frecuentes a la hora de construir una API suele ser crear nuestras propias herramientas en lugar de utilizar las que ya han sido creadas, pensadas y testadas. El error más habitual en el desarrollo de APIs es crear interpretes para los códigos de error y códigos de estado fuera de los estándares. Este es un error común que tiene varios inconvenientes:

- No es REST ni es estándar.
- El cliente que acceda a este API debe conocer el funcionamiento especial y cómo tratar los errores de la misma, por lo que requiere un esfuerzo adicional importante para trabajar con nosotros.
- Tenemos que preocuparnos por mantener nuestros propios códigos o mensajes de error, con todo lo que eso supone.

Permite conectar mediante vínculos las aplicaciones clientes con las APIs, permitiendo a dichos clientes despreocuparse por conocer de antemano del cómo acceder a los recursos.

En cuanto a los tipos de contenido aceptados por la aplicación, con hipermedia básicamente añadimos información extra al recurso sobre su conexión a otros recursos. Sin embargo, necesitamos que el cliente que accede a nuestra API entienda que esa información no es propia del recurso, sino que es información añadida que puede utilizar para enlazar unos recursos con otros.

Para conseguir esto, debemos utilizar las cabeceras estándar HTTP *Accept* y *Content-Type*, para que tanto el cliente como la API, sepan que están hablando hipermedia.

Hipermedia es útil por ejemplo para que el cliente no tenga que conocer las URLs de los recursos, evitando tener que hacer mantenimientos en cada uno de los mismos si en un futuro dichas URLs cambian. También es útil para automatizar procesos entre APIs sin que haya interacción humana.

3.6. La interfaz uniforme

Utilizando el protocolo HTTP, hay solo unas pocas cosas básicas que se le pueden hacer a un recurso, como se detalla en [12], capítulo 4. HTTP proporciona cuatro métodos básicos para las cuatro operaciones más comunes:

- Obtener la representación de un recurso: GET
- Crear un nuevo recurso: PUT en una nueva URI o POST en una URI existente
- Modificar un recurso existente: PUT en una URI existente
- Borrar un recurso: DELETE

3.6.1. GET, PUT Y DELETE

Para buscar o borrar un recurso, el cliente manda una petición GET o DELETE a su URI. En el caso de GET el servidor devuelve una representación en el cuerpo (*body*) de la respuesta. Para una petición DELETE el cuerpo de la respuesta puede contener un mensaje de estatus o nada en absoluto.

Para crear o modificar un recurso, el cliente manda una petición PUT que normalmente incluye el cuerpo. El cuerpo contiene la nueva representación del recurso propuesta por el cliente. Qué datos la forman y en qué formato están esos datos depende del servicio.

3.6.2. HEAD y OPTIONS

Aparte de los métodos básicos descritos, el protocolo HTTP ofrece otros métodos adicionales que son conocidos como métodos de utilidades. HEAD y OPTIONS son los dos más representativos. Este es su significado:

- HTTP HEAD: Obtener sólo la representación en meta-datos del recurso.
- HTTP OPTIONS: Comprobar qué métodos soporta un recurso en particular.

HEAD obtiene los meta-datos de un recurso sin descargar el cuerpo del recurso (que puede ser enorme). Un cliente puede usar HEAD para comprobar si un recurso existe, o encontrar información acerca del recurso sin obtener su representación completa. HEAD proporciona exactamente lo que ofrece GET pero sin el cuerpo (*payload*).

El método OPTIONS permite al cliente descubrir qué esta permitido hacer con el recurso. La respuesta a una petición OPTIONS contiene la cabecera HTTP *Allow*, en la cual se aloja el subconjunto de la interfaz uniforme que soporta el recurso.

Allow: GET, HEAD

Las cabeceras que el cliente manda en sus peticiones pueden afectar a la cabecera *Allow* que el servidor manda en la respuesta. No existe un estándar aceptado sobre lo que un cliente debe mandar en una petición OPTIONS. Aparte de la cabecera *Allow*, no hay un estándar sobre lo que el servidor debe mandar en la respuesta.

3.6.3. POST

Este método tiene esencialmente dos propósitos: uno que encaja con las limitaciones de REST, y otro que queda fuera de REST.

POST es un método diseñado para permitir:

- Anotación de recursos existentes.
- Postear un mensaje.
- Proporcionar un bloque de datos, como resultado de un formulario de envío, a un programa manejador de datos.
- Extender una base de datos como una operación de añadido.

La función actual de POST es determinada por el servidor y normalmente depende de la URI de la petición.

En un diseño RESTful, POST normalmente se utiliza para crear recursos subordinados a otros recursos: recursos que existen en relación con algún otro recurso “padre”. Los datos que se postean y su formato depende del servicio, pero al igual que en PUT, en ese momento es cuando el estado de la aplicación se convierte en el estado del recurso. La mayoría de las veces se usa POST en el sentido de “añadir”.

¿Por qué no se puede usar PUT para crear recursos subordinados? La diferencia entre PUT y POST es esta: el cliente usa PUT cuando es el encargado de decidir que URI tendrá el recurso nuevo. El cliente usa POST cuando es el servidor el que decide qué URI tendrá el nuevo recurso.

El método POST es un método que permite al cliente crear nuevos recursos sin tener que conocer su URI exacta. En muchos casos el cliente tan solo necesita saber la URI de un recurso "padre" o "factoría". El servidor toma la información de la entidad-cuerpo y la usa para crear un nuevo recurso "debajo" del recurso "padre". La respuesta a este tipo de petición POST normalmente contiene un código de estado 201 "Creado".

La cabecera *Location* contiene la URI del nuevo recurso subordinado creado. Ahora que el recurso existe y el cliente conoce su URI, futuras peticiones pueden utilizar el método PUT para modificar ese recurso, GET para obtener una representación del mismo o DELETE para borrarlo.

A veces una acción POST no conlleva la creación de un nuevo recurso, sino que tan solo añade nueva información al estado de un recurso ya existente (en forma de logs, por ejemplo).

3.6.4. Sobrecarga de POST

El método POST se ha utilizado tradicionalmente cómo el método de envío en los formularios HTTP para mandar un bloque de datos al servidor. El programa que recibe los datos en el servidor manipula dichos datos para devolver a partir de los mismos otros datos al cliente.

Se le denomina sobrecarga (en analogía a la sobrecarga de operadores de un lenguaje de programación) porque se utiliza un método HTTP para dotar de significado a un número de métodos no-HTTP.

Cada petición HTTP debe contener la información sobre el método. Cuando esa información no se puede encontrar en el método HTTP, la interfaz deja de ser uniforme, como se menciona el [12], capítulo 4. En el caso de la sobrecarga POST el método real puede ser cualquier cosa. No sabemos qué acciones realizará sobre los datos la aplicación del lado cliente. Esta forma de operar se acerca más al estilo RPC, y se aleja de REST.

¿Como puede el servidor saber si se está creando un recurso nuevo o si se están añadiendo datos a la representación actual del recurso? Es necesario añadir información adicional en algún lugar de la petición HTTP, lo que contradice la filosofía RESTful. Recordemos que toda la información necesaria para satisfacer la petición debe estar presente en la misma URI.

3.6.5. Seguridad e idempotencia

Cuando se usan correctamente, GET y HEAD son seguros. GET, HEAD, PUT y DELETE son idempotente.

Se dice que son seguras ya que las peticiones GET y HEAD leen datos, no los modifican ni cambian su estado.

Pueden existir algunos efectos laterales: algunos recursos pueden tener contadores que se incrementan con cada petición GET o HEAD. El estado del servidor y el estado del recurso cambian con la petición GET. Un cliente no debe realizan nunca una petición GET o HEAD pensando únicamente en los efectos laterales, que no deben preocuparle.

Una operación idempotente es aquella que tiene el mismo efecto independiente del lugar en el cual se aplique, o si se aplica más de una vez (como multiplicar por cero). Por analogía, "una operación idempotente es aquella en la que hacer una petición es lo mismo que hacer una serie de peticiones idénticas" (Richardson, (2007), p. 102). La segunda y sucesivas peticiones dejan el recurso exactamente en el mismo estado en el que lo dejó la primera.

Las operaciones idempotentes son GET, HEAD, PUT y DELETE.

No se debe permitir a los clientes realizar PUT con representaciones que cambien el estado del recurso en términos relativos (incrementar en una unidad el valor).

La seguridad y la idempotencia permiten a un cliente realizar peticiones HTTP de confianza sobre una red insegura.

POST no es ni seguro ni idempotente. Realizar dos peticiones POST idénticas sobre el mismo recurso "factoría" posiblemente creará dos recursos diferentes con la misma información.

El mal uso más común de la interfaz uniforme es exponer operaciones inseguras a través de GET. Como realizar un DELETE a través de un GET a una URI del tipo http://example/resource/01/delete

En el ejemplo no se está obteniendo información del recurso, se está borrando el recurso. No es HTTP estándar, y por lo tanto, no es REST.

3.6.6. Porqué es importante una interfaz uniforme

Lo importante en REST no es que se utilice la interfaz estándar que define HTTP. REST especifica que se debe utilizar una interfaz estándar, pero no indica qué interfaz utilizar. Lo importante es la uniformidad. Que cada servicio utilice la interfaz HTTP de la misma forma.

Sin una interfaz uniforme, se debería aprender para cada servicio cómo este espera recibir información y cómo la va a devolver.

Se puede programar a un ordenador para que entienda lo que significa GET, y esa compresión se puede trasladar a cualquier servicio web REST. El código específico de la aplicación puede vivir en el manejo de la representación. Sin hacer uso de la interfaz uniforme puedes tener múltiples métodos para hacer lo mismo. Cada servicio habla en un lenguaje diferente.

Algunas aplicaciones extienden la interfaz uniforme de HTTP. WebDAV añade los métodos COPY, MOVE y SEARCH. Usar estos métodos no viola ninguna de las restricciones de REST, porque REST no especifica cómo debe ser la interfaz uniforme. Su uso puede violar la ROA (*Resource Oriented Arquitecture*) pero el servicio puede seguir siendo orientado a recurso en un sentido general.

La verdadera razón para no usar WebDAV es que puede hacer que tu servicio sea incompatible con otros servicios RESTful. Tu servicio usará una interfaz uniforme distinta a la del resto. Por ello crear tus propios métodos HTTP es una muy mala idea. Tu vocabulario personalizado te posiciona en una comunidad de un solo individuo.

Otra interfaz uniforme consiste únicamente en GET y POST sobrecargado. Para obtener la representación de un recursos usas GET. Para crear, modificar o borrar un recurso se usa POST. Esta interfaz es totalmente RESTful, pero no se adapta a la ROA.

4. Diseño de la capa de interoperabilidad REST

Como primer paso del diseño de la API de control REST debemos definir qué objetos de la SmartHome serán considerados recursos REST. Una parte de esta identificación consiste en establecer los tipos de recurso que existen y qué comandos de la interfaz uniforme HTTP ofrece cada tipo. Es por ello que se divide la definición de los recursos en tres fases diferenciadas, como se comentan en [12], capítulo 5:

1. **Identificación de recursos:** qué es un recurso y qué no lo es.
2. **Establecer las URIS de los recursos:** una vez identificado un recurso, debemos proporcionarle un identificador único.
3. **Determinar la interfaz uniforme y sus representaciones**: básicamente consiste en decidir qué operaciones de la interfaz HTTP ofrecerá cada recurso. Para cada operación hay que determinar una representación acorde con el formato elegido.

4.1 Identificación de los recursos

Dentro de la vivienda inteligente contamos con una serie de elementos que pueden considerarse como recursos REST. La mayoría de estos elementos son dispositivos domóticos que ofrecen una serie de funcionalidades.

El primer recurso que podemos definir es la propia **SmartHome** que nos ofrecerá acceso al resto de recursos. Podemos identificar este recurso principal como la URL de entrada a nuestra aplicación, a través de la cual accederemos al resto de los recursos.

Otro tipo de recurso son los **dispositivos domóticos** que forman la SmartHome, en adelante diremos que se trata de recursos del tipo **Device**. Este tipo de recurso se puede dividir en tipos, según las funcionalidades que ofrecen. Existen tres tipos básicos de recurso Device:

- **Sensores:** recogen información y se les puede consultar magnitudes físicas, como temperatura, humedad, etc.
- **Actuadores:** pueden realizar acciones sobre algo.
- Dispositivos que son una **combinación** de sensor y actuador.

Las funcionalidades de los dispositivos también se consideran como recursos, a los que en adelante denominaremos como recursos del tipo **DeviceFunctionality**, y se catalogan a su vez de la siguiente forma:

- ***Switch/bistate*:** solo pueden tomar dos valores posibles (1/0).
- ***Togglebistate:*** biestables que pueden conmutar su valor. Cuando se les pide que cambien de estado cambia a ON si están OFF y a OFF si están ON.
- ***Dimmer/bistate*:** son biestables también, pero su valor puede ser medido en porcentaje (50%).
- ***Pulse/signal*:** no mantienen el estado. Simplemente avisan de que ha ocurrido un evento mediante un pulso o señal.
- ***Numeric Value*:** gestionan valores numéricos. En el caso de los sensores representan las medidas tomadas. En el caso de los actuadores representan valores sobre los que es posible actuar.
- ***Moviment*:** representan a objetos móviles que se pueden desplazar o girar. Normalmente se trata de movimientos en un sentido, en el contrario y la acción de parar (izquierda/derecha, arriba/abajo, iniciar/parar, etc.).
- ***Positionable Moviment*:** objetos móviles a los que se les puede indicar una posición relativa, es decir, se les puede pedir que realicen un movimiento indicando porcentualmente cuánto queremos que se muevan.

Además de estos recursos, fácilmente identificables, nuestra aplicación considera dos tipos más de recursos, que no representan objetos concretos o funcionalidades definidas, sino que sirven como contenedores de otros recursos.

Por un lado se define el recurso del tipo **Devices** como un recurso que contiene recursos del tipo Device. Es decir, un recurso Devices no representa a un recurso Device en concreto, sino que contiene enlaces a otros recursos que son del tipo Device. A su vez, se define el recurso **DeviceFunctionalities** que contendrá recursos del tipo DeviceFunctionality. En nuestra aplicación solo existirá un único recurso del tipo Devices y un único recurso del tipo DeviceFunctionalities.

Un recurso del tipo Device estará relacionado con uno o más recursos del tipo DeviceFunctionality, aplicando el concepto de la conectividad entre recursos propia de REST. Esto no significa que para acceder a un recurso del tipo DeviceFunctionality debamos acceder forzosamente a través del recurso Device relacionado. La conectividad entre recursos simplemente ofrece una forma sencilla de localizar determinados recursos, pero seguimos contando con la posibilidad de acceder directamente a cualquier recurso a través de su propia URI.

Poniendo un ejemplo práctico, imaginemos que queremos encender la luz principal del comedor. Esta funcionalidad será un interruptor (*switch*) que controla el encendido y apagado de esta luz. Podemos (si la conocemos) acceder directamente a la URI de esta funcionalidad y encender o apagar la luz. Sin embargo en una vivienda podemos tener docenas de interruptores, y puede ser tedioso averiguar la URI de interruptor exacto que buscamos. Si en lugar de ello accedemos al panel de interruptores del comedor, en el cual se encuentra ubicado este interruptor, el resultado de nuestra búsqueda se limitará a los pocos interruptores presentes en ese panel. Es una forma de simplificar la localización de la funcionalidad necesaria para interactuar con cierto objeto.

Esta es una de las muchas alternativas posibles que REST ofrece para establecer una jerarquía (ficticia) entre los recursos. De este modo diremos que un recurso DeviceFunctionality está relacionado con un único recurso del tipo Device, y a su vez que un recurso del tipo Device puede tener relaciones con uno o más recursos del tipo DeviceFunctionality.

En el siguiente esquema se representa la estructura entre los recursos contenedores Devices y DeviceFunctionalities y los recursos enlazados Device y DeviceFunctionality, pero no se establecen relaciones entre recursos Device y DeviceFunctionality, ya que estas relaciones dependen de cada recurso Device.

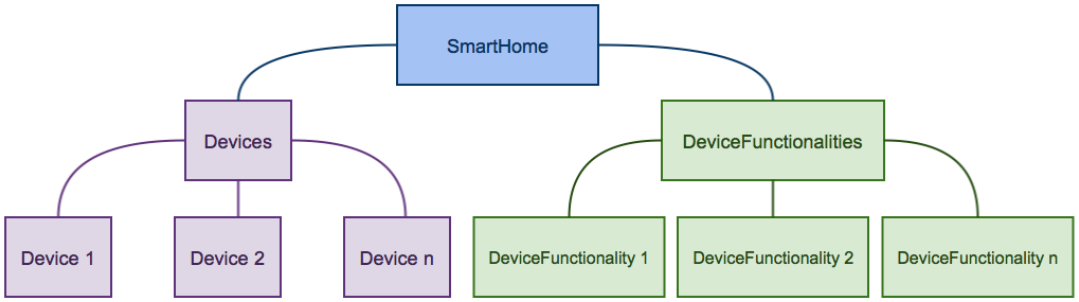

Figura 17 - Diagrama de recursos

Del mismo modo, la relación entre un recurso y las DeviceFunctionalities relacionadas con el mismo se representa de la siguiente forma:

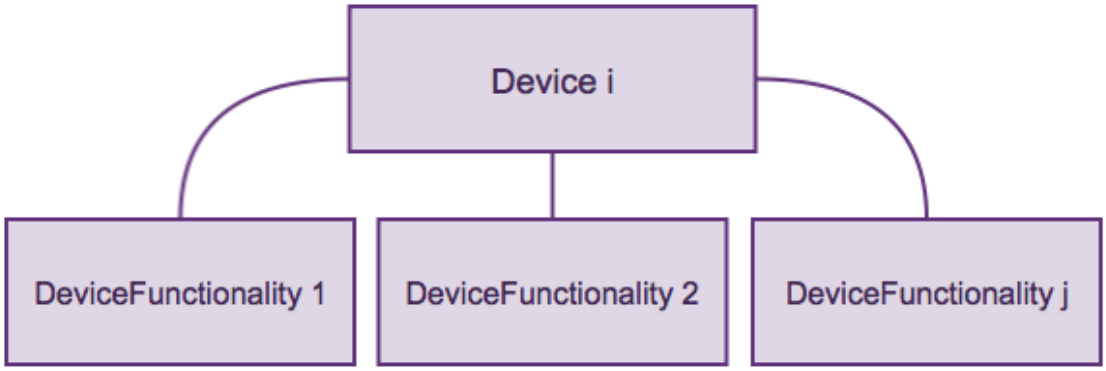

Figura 18 - Recurso Device y DeviceFunctionality relacionadas

Estas DeviceFunctionality no estarán relacionadas con ningún otro recurso Device del sistema.

4.2 Establecer las URIs canónicas de los recursos

Como ya se ha definido en el punto 3.1 Recursos, todo recurso debe tener al menos una URI, mediante la cual nombraremos al recurso de manera inequívoca. Esta URI debe contener toda la información relevante necesaria para realizar una petición al recurso.

El servidor web de nuestra aplicación también debe tener una URI. La URI de nuestro servidor estará compuesta por el protocolo utilizado, en este caso HTTP, el nombre del dominio del servidor web en el cual corre nuestra aplicación y por el puerto en el cual el servidor de la SmartHome atiende las peticiones, que se configura a 8222. Por tanto, si el dominio de nuestro servidor se llamase smarthomeserver, la URI de nuestro servidor sería http://smarthomeserver:8222

Todos los recursos tendrán una URI relativa a la URI del servidor. Por ejemplo, podemos definir un recurso principal del tipo Devices que tendrá una URI de este tipo: http://smarthomeserver:8222/devices. A través de esta URI se podrán listar todos los recursos del tipo Device contenidos en la SmartHome. Es decir, que se implementará el método GET de este recurso para que muestre dicho listado.

Podremos acceder a un Device en concreto a través de una URI construida de la siguiente manera: http://smarthomeserver:8222/devices/{dev-id}, siendo dev-id el identificador de dicho Device.

También contaremos con un recurso principal del tipo DeviceFunctionalities, al que asignaremos la URI http://smarthomeserver:8222/dfs desde el cual listaremos todas los recursos del tipo DeviceFunctionality de la SmartHome, independientemente del recurso Device al que pertenezcan.

Del mismo modo que antes, para acceder a una determinado recurso de este tipo lo haremos desde una URI como http://smarthomeserver:8222/dfs/{func-id} , siendo func-id el identificador de dicho recurso.

Como ya ha sido indicado en el punto 3.2 Construcción correcta de las URIs, un mismo recurso puede tener más de una URI, además de la principal o canónica. Las URIs aquí descritas serán consideradas como las canónicas de estos recursos.

Sin embargo, dado que todo el desarrollo documentado en la memoria se ha realizado en base a un **servidor local de test**, y que no se ha tenido la oportunidad de implantar la API en un entorno real, debemos limitarnos a utilizar en la memoria las direcciones URL relativas a nuestra máquina local.

El dominio del servidor de la máquina local de test se establece en localhost. De esta manera, las URL que se utilizan en el resto de la memoria están compuestas por direcciones que empiezan por http://localhost. Por ejemplo, la URI de la SmartHome local es http://localhost:8222, la URI de un Device es

http://localhost:8222/devices/{dev-id} y la de una DeviceFunctionality tiene la forma http://localhost:8222/dfs/{func-id}

4.3 La interfaz uniforme y sus representaciones

En un lenguaje de programación orientado a objetos, una clase puede tener cualquier número de métodos u operaciones, pero un recurso HTTP expone una interfaz uniforme de un máximo de seis métodos HTTP. Estos métodos permiten realizar únicamente las operaciones más básicas: crear (PUT o POST), modificar (PUT), leer (GET), y borrar (DELETE). Se puede expandir esta interfaz añadiendo métodos utilizando POST sobrecargado, pero obtendríamos un recurso del estilo RPC, y alejado de REST.

La interfaz uniforme significa que el diseño orientado a recursos debe tratar a los métodos de la forma en que un diseño orientado a objetos los podría considerar verbos o acciones. Cuando se necesite añadir un nuevo método a una de las clases recursos, que expandiría la interfaz uniforme HTTP, en lugar de ello se debe definir un nuevo tipo de recurso que implemente en su interfaz el nuevo método, que a su vez formará parte de la interfaz uniforme del nuevo tipo de recurso creado. Por lo tanto, **cada tipo de recurso implementará un subconjunto de la interfaz uniforme que ofrece HTTP**.

Los recursos de la SmartHome son accesibles a través de sus URIS. Podemos acceder al recurso http://localhost:8222 que representa a la SmartHome mediante una operación GET a través de un navegador web y el servidor nos devolverá determinada información, como el nombre que le hemos dado a nuestro servidor o el estado del mismo.

Pero debemos establecer también el formato en el que se transmitirá la información, tanto en las peticiones procedentes de los clientes como en las repuestas a dichas peticiones que procedan del servidor.

REST no establece un formato predeterminado para dotar de estructura a estos datos. Queda en manos del diseñador el elegir un formato u otro en función de las necesidades de las aplicaciones a las que está destinada la interfaz REST.

En el caso de este proyecto se considera que el formato debe ser compatible con un gran número de dispositivos y al mismo tiempo lo suficientemente ligero como para que las transmisiones se efectúen de forma rápida y fluida, pero sin que ello conlleve una pérdida de información. Hay que considerar que hoy en día es cada vez más común el uso de aplicaciones desde dispositivos móviles como *Smartphones* o *tablets*, los

cuales, además de sus limitaciones físicas en cuanto a potencia del procesador y escasa memoria, suelen conectar mediante redes inalámbricas a los servidores.

Por lo tanto necesitamos que la información sea transmitida en un formato sencillo, ligero y a la vez potente. Por estos motivos se ha elegido el formato **JSON** para representar la información transmitida, puesto que cumple con todos los requisitos expuestos.

De esta forma, por ejemplo, podemos establecer que la información que va a devolver una petición GET a la URI de nuestro servidor http://localhost:8222 va a ser:

- **Nombre**: el nombre que establezcamos para nuestra SmartHome.
- **Estado**: si la petición ha tenido éxito o no.
- **Descripción**: descripción detallada de la SmartHome.
- **URI**: dirección mediante la cual accederemos a la SmartHome.

En forma de JSON, si hacemos un GET a http://localhost:8222 lo que veremos en el navegador web será algo parecido a esto:

```
{"Name":"RESTful SmartHome Server","Status":"OK (200) - The request
has succeeded","Description":"This is a prototype of a SmartHome
Server","URI":"http://localhost:8222/"}
```

Este es un JSON muy sencillo que muestra muy poca información, pero en algunos casos obtendremos JSON mucho más complicados, que contendrán mucha información dividida en campos o dividida en listas o colecciones que comparten un denominador común. Es recomendable el empleo de alguna herramienta que facilite la lectura de los JSON generados. Se ha optado por utilizar la herramienta RESTClient, descrita en el punto 2.7 RESTClient

En el siguiente punto se describen todos los tipos de recursos existentes, junto las operaciones implementadas en cada uno, haciendo énfasis tanto en el formato de entrada por parte del cliente como en el de la salida producida por el procesamiento de las peticiones en la parte del servidor.

4.3.1. Recurso SmartHome

Este recurso es la puerta de entrada a nuestra aplicación y representa a la vivienda, por lo que tan solo implementará la parte de la interfaz uniforme relativa a obtener la representación del recurso (GET) y el comando que permite obtener la interfaz implementada (OPTIONS).

Esta interfaz uniforme simplificada se representa en la siguiente tabla:

RECURSO	SmartHome
URI	http://localhost:8222
OPERACIONES	
OPTIONS	Obtenemos el listado de operaciones disponibles sobre la SmartHome (GET y OPTIONS)
GET	Obtenemos una representación de la vivienda, compuesta por: • Nombre de la SmartHome. • Estado HTTP de la petición a la SmartHome • Descripción de la SmartHome. • URI del recurso SmartHome

Tabla 1 - Interfaz Uniforme del recurso SmartHome

El recurso SmartHome tiene definidas las operaciones GET y OPTIONS, las cuales componen el subconjunto de la interfaz uniforme establecido para este recurso. A continuación se muestran las entradas esperadas por el servidor para cada operación, y el formato de la salida correspondiente.

4.3.1.1. Operación OPTIONS

La operación OPTIONS se ejecuta enviado una petición HTTP OPTIONS a la URI http://localhost:8222. La respuesta que se obtiene en este caso no es un JSON, sino que las operaciones admitidas por el recurso se indican en la cabecera *Allow* del protocolo HTTP:

```
1.  Status Code       : 200 OK
2.  Accept-Ranges     : bytes
3.  Allow             : GET, OPTIONS
4.  Content-Length    : 0
5.  Date              : Fri, 20 Feb 2015 17:19:30 GMT
6.  Server            : Restlet-Framework/2.1.7
```

Figura 19 - Operación OPTIONS del recurso SmartHome

4.3.1.2. Operación GET

La operación GET se realiza siempre sobre la URI del recurso. En el caso del recurso SmartHome, la URI es http://localhost:8222. La respuesta que el servidor manda al cliente es un JSON con la siguiente estructura, que se ha tabulado para facilitar su lectura:

```
{
  "Name": "SmartHome Server",
  "Status": "OK (200) - The request has succeeded",
  "Description": "This a prototype of a SmartHome Server",
  "URI": http://localhost:8222/
}
```

El significado de los campos del JSON que se muestra a la salida son:

- **Name**: nombre que le hemos dado a nuestro servidor
- **Status**: indica si la consulta ha tenido o no éxito. Utiliza los códigos estándar del protocolo HTTP. Un código 200 indica éxito en la petición.
- **Description**: La descripción que le hemos dado a nuestra SmartHome.
- **URI**: La URI de acceso al recurso SmartHome.

4.3.2. Recurso Devices

Devices es un recurso contenedor que engloba a los distintos recursos Device que componen la SmartHome. La URI de acceso a este recurso Devices es http://localhost:8222/devices.

Este recurso proporciona un modo de poder listar las URIS de todos los dispositivos de la vivienda, por lo que las únicas acciones permitidas sobre Devices serán GET y OPTIONS. En forma de tabla quedaría de la siguiente forma:

RECURSO	DEVICES
URI	http://localhost:8222/devices
OPERACIONES	
OPTIONS	Obtenemos el listado de operaciones disponibles sobre el recurso contenedor (GET y OPTIONS)

GET	Obtenemos una representación en JSON del recurso Devices compuesta por:
	• URI completa o canónica del recurso Devices
	• Listado de recursos Device presentes en la SmartHome. Cada entrada de este listado está compuesta por:
	  ○ Nombre del Device
	  ○ URI relativa (Link) del Device
	  ○ Identificador (ID) del Device

Tabla 2 - Interfaz uniforme del recurso Devices

El listado generado mediante el método GET contiene información relevante para el acceso a cada uno de los recursos del tipo Device, mientras que el método OPTIONS tan solo nos muestra qué métodos de la interfaz implementa el recurso. A continuación se describen detalladamente ambos.

4.3.2.1. Operación OPTIONS

Realizar una consulta OPTIONS sobre la URI http://localhost:8222/devices nos mostrará un resultado similar a este, en el que se nos indican las operaciones disponibles sobre el recurso: GET y OPTIONS.

```
Status Code       : 200 OK
Accept-Ranges     : bytes
Allow             : GET, OPTIONS
Content-Length    : 0
Date              : Tue, 09 Dec 2014 18:28:54 GMT
Server            : Restlet-Framework/2.1.7
```

Figura 20 - Operación OPTIONS del recurso Devices

4.3.2.2. Operación GET

La operación GET sobre la URI http://localhost:8222/devices genera un JSON en el que se muestra información de cada uno de los recursos del tipo Device registrados en la SmartHome. Concretamente, por cada recurso Device se muestra su identificador (DevID) y su URI relativa (Link). El link es una URI al recurso Device que no contiene información sobre el servidor, el protocolo o el puerto. A continuación se muestra un extracto del JSON generado, del que se muestra el principio y el final:


```
{
  "Devices": [
    {
      "Name": "Il·luminació Casa",
      "Link": "/devices/DEV-IL",
      "DevID": "DEV-IL"
    },
    {
      "Name": "Polsadors Pis",
      "Link": "/devices/DEV-SP",
      "DevID": "DEV-SP"
    },
    {
      "Name": "IP CAM AXIS M1031W",
      "Link": "/devices/DEV-AXIS.M1031W",
      "DevID": "DEV-AXIS.M1031W"
    },

###  SE HAN BORRADO DEVICES DEL LISTADO PARA REDUCIR SU TAMAÑO  ###

{
      "Name": "Estació Meteo",
      "Link": "/devices/DEV-ESTACIO.METEO",
      "DevID": "DEV-ESTACIO.METEO"
    },
    {
      "Name": "Detector Climatologia Adversa",
      "Link": "/devices/DEV-DETECTOR.CLIMATOLOGIA.ADVERSA",
      "DevID": "DEV-DETECTOR.CLIMATOLOGIA.ADVERSA"
    },
    {
      "Name": "Porta Finca",
      "Link": "/devices/DEV-PORTA.FINCA",
      "DevID": "DEV-PORTA.FINCA"
    }
```

```
  ],
  "URI": "http://localhost:8222/devices"
}
```

4.3.3. Recurso Device

Los recursos **Device** son la abstracción de los dispositivos físicos. Cada uno de ellos contiene un conjunto de enlaces a los recursos del tipo DeviceFuntionality relacionados con el dispositivo, y que representan las funcionalidades que el dispositivo ofrece. La URI de cada recurso Device se determinará en base a un parámetro del dispositivo al que denominaremos **identificador** {id} del Device. La URI completa o canónica del Device con identificador dev_1 sería http://localhost:8222/devices/dev_1.

El identificador en sí mismo debe ser descriptivo, cumpliendo los preceptos de REST. En el sistema de la SmartHome utilizado como base de la interfaz REST los identificadores de los recursos ya se encuentra definidos. Se utilizan como identificadores de los recursos cadenas de texto que describen un poco al recurso que representan. De esta forma, por ejemplo, el identificador de la puerta principal de la SmartHome es DEV-PORTA.FINCA. En el mismo identificador se detalla que el recurso pertenece al tipo Device (DEV), que se trata de una puerta (PORTA) y que es la principal de la vivienda (FINCA).

En la siguiente tabla se muestra un resumen de la interfaz uniforme:

RECURSO	DEVICE
URI	http://localhost:8222/devices/{id}
OPERACIONES	
OPTIONS	Obtenemos el listado de operaciones disponibles para un Device (GET y OPTIONS)

GET	Obtenemos una representación en JSON del dispositivo compuesta por:
	• Nombre del Device
	• Identificador del Device (ID)
	• Link (URI reducida)
	• Estado del Device (si se ha iniciado o está detenido)
	• Disponibilidad del Device (si está activado o no)
	• Listado de los recursos DeviceFunctionality disponibles en este dispositivo. Por cada recurso DeviceFunctionality enlazado se muestra:
	  ○ Nombre del recurso
	  ○ Identificador (ID)
	  ○ Estado actual
	  ○ Estado anterior
	  ○ URI relativa (Link)
	• URI completa o canónica del Device

	• Otros datos menos relevantes, como la lista de localizaciones (*locations*) y su estado de quiescencia.
PUT	La operación PUT se emplea para que el recurso realice determinada acción. La acción se incluye en el cuerpo de la petición PUT en formato JSON. Las acciones disponibles para los recurso del tipo Device son: • *enable*: activa un Device desactivado • *disable*: desactiva un Device activo

Tabla 3 - Interfaz uniforme del recurso Device

Algunos de los datos mostrados en la operación GET, como son la lista de localizaciones (*locations*) o la quiescencia del recurso no son relevantes para el desarrollo del presente trabajo, por lo que aunque están presentes en los recursos Device del sistema SmartHome usado como base, no se va a detallar ni su utilidad ni su funcionamiento en el resto de la memoria.

4.3.3.1. Operación OPTIONS

La operación OPTIONS devuelve en las cabeceras HTTP de la respuesta a la petición los métodos soportados por este tipo de recurso, como se aprecia en la siguiente imagen.

```
1.  Status Code      : 200 OK
2.  Accept-Ranges    : bytes
3.  Allow            : GET, OPTIONS
4.  Content-Length   : 0
5.  Date             : Thu, 04 Dec 2014 18:05:28 GMT
6.  Server           : Restlet-Framework/2.1.7
```

Figura 21 - Operación OPTIONS

Es en la cabecera HTTP *Allow* donde se muestran los métodos soportados: GET y OPTIONS.

4.3.3.2. Operación GET

La operación GET realizada sobre la URI de un Device devuelve un JSON con información relativa al dispositivo. A continuación se muestra un ejemplo de una consulta GET realizada sobre un Device concreto. Se utiliza en el ejemplo de nuevo al Device que representa la puerta principal de la SmartHome, cuya URI es http://localhost:8222/devices/DEV-PORTA.FINCA

```
{
  "Name": "Porta Finca",
  "Link": "/devices/DEV-PORTA.FINCA",
  "Is_Enabled": true,
  "Is_Quiescent": false,
```

```
  "Locations": [
    {
      "Link": "/location/LOC-EXTERIOR",
      "Id": "LOC-EXTERIOR"
    }
  ],
  "Id": "DEV-PORTA.FINCA",
  "DeviceFunctionalities": [
    {
      "Name": "Forrellat",
      "Current_State": "UNKNOWN",
      "Link": "/dfs/DF-PORTA.FINCA.FORRELLAT",
      "Previous_State": "UNKNOWN",
      "FuncID": "DF-PORTA.FINCA.FORRELLAT"
    }
  ],
  "Is_Started": true,
  "URI": "http://localhost:8222/devices/DEV-PORTA.FINCA"
}
```


Lo más destacable de este JSON es el hecho de que, además de la información relativa al propio dispositivo, como puede ser su ID o su link, también **aparece dentro del JSON información sobre las funcionalidades que el recurso ofrece**.

En el ejemplo anterior el Device ofrece una única funcionalidad cuyo Id es **DEV-PORTA.FINCA.FORRELLAT**. Uno de los datos que se nos proporciona de las funcionalidades del Device es su Link. **A través de este Link podemos construir fácilmente la URI completa de la funcionalidad.** Para obtener la URI completa o canónica de cualquier recurso a partir de su URI relativa o Link basta con concatenar al principio del Link la URI canónica de la SmartHome.

Sabiendo que la URI canónica de la SmartHome es http://localhost:8222 es fácil obtener la URI canónica de cualquier recurso a partir de su Link con el método descrito. En este caso la URI generada para la DeviceFunctionality DEV-PORTA.FINCA.FORRELLAT sería http://localhost:8222/dfs/DEV-PORTA.FINCA.FORRELLAT

De esta forma los recursos de la SmartHome cumplen con otra de las directrices básicas de REST: la **interconexión** de unos recursos con otros, de modo que es posible alcanzar desde un recurso a los recursos relacionados con el mismo.

Este es sólo una de las muchas posibilidades que REST ofrece para enlazar unos recursos con otros, aplicando su principio de conectividad. Es el programador quien tiene libertad para determinar la forma en la que unos recursos se enlazan con otros, y cómo acceder desde un recurso dado a los enlaces de los recursos relacionados.

Realizando una operación GET sobre la URI de la DeviceFunctionality DF-PORTA.FINCA.FORRELLAT podemos comprobar a su vez que la funcionalidad pertenece al Device DEV-PORTA.FINCA:


```
{
  "Device": {
    "Name": "Porta Finca",
    "Link": "/device/DEV-PORTA.FINCA",
    "Id": "DEV-PORTA.FINCA"
  },
  "Is_Enabled": true,
  "Type": "bistate",
  "Is_Quiescent": false,
  "Locations": [
    {
      "id": "LOC-EXTERIOR",
      "link": "/location/LOC-EXTERIOR"
    }
  ],
  "Previous_State": "UNKNOWN",
  "Is_Started": true,
  "Name": "Forrellat",
  "Current_State": "UNKNOWN",
  "Subtype": "ON_OFF",
  "Link": "/dfs/DF-PORTA.FINCA.FORRELLAT",
  "Last_Action": "unknown",
  "Id": "DF-PORTA.FINCA.FORRELLAT",
  "URI": "http://localhost:8222/dfs/DF-PORTA.FINCA.FORRELLAT"
}
```


4.3.3.3. Operación PUT

Los recursos del tipo Device implementan este método mediante el envío de un JSON incluido en el cuerpo (*body*) de la petición que contiene la acción a realizar. La estructura del JSON empleado es simple y está compuesta por un único campo "*action*" cuyo valor determinará la acción a realizar sobre el Device:

```
{"action":"acción_a_realizar"}
```

Los posibles valores de este parámetro para este tipo de recursos son dos:

- **"enable":** Activa un Device desactivado. No tiene efecto sobre un Device ya activo, respetando la regla de idempotencia del método PUT.
- **"disable":** Descativa un Device activado. No tiene efecto sobre un Device ya desactivado, respetando la regla de idempotencia del método PUT.

También será necesario enviar, junto con el JSON apropiado en el cuerpo de la petición PUT, las cabeceras HTTP necesarias para su correcta interpretación por parte de nuestro servidor. Dado que no contamos con una verdadera aplicación cliente, vamos a utilizar la aplicación RESTClient para realizar las peticiones a nuestro

servidor. Esta aplicación permite, entre otras cosas, añadir a nuestra petición cabeceras HTTP personalizadas.

Las cabeceras a añadir serán las siguientes:

- **Accept: application/json** indica los tipos de formato aceptables para la respuesta. En caso de no indicar nada, se podría asumir que el cliente acepta todos los tipos. Con este valor indicamos que en la respuesta nuestro cliente espera recibir un JSON.
- **Content-Type: application/json** indica en qué formato se envía el cuerpo de la petición al servidor. Con este valor le indicamos al servidor que el cuerpo del mensaje está en formato JSON.

En el punto 2.7 RESTClient se describe cómo añadir las cabeceras HTTP necesarias y cómo incluir en el *body* el JSON de nuestra petición PUT. Además RESTClient permite almacenar las cabeceras HTTP que añadamos a la petición, de modo que puedan ser empleadas en futuras peticiones.

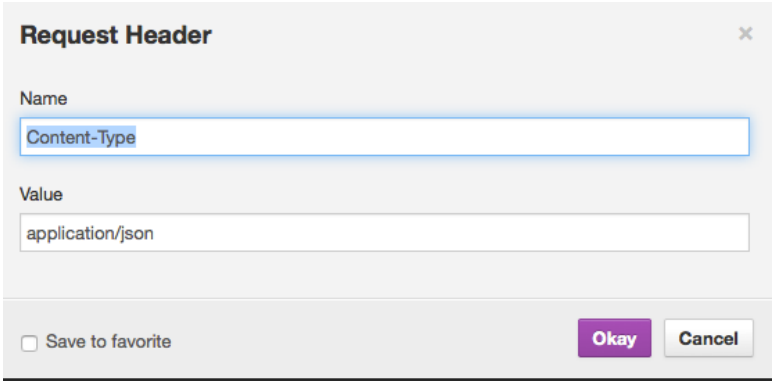

Figura 22 - Cabecera Content-Type

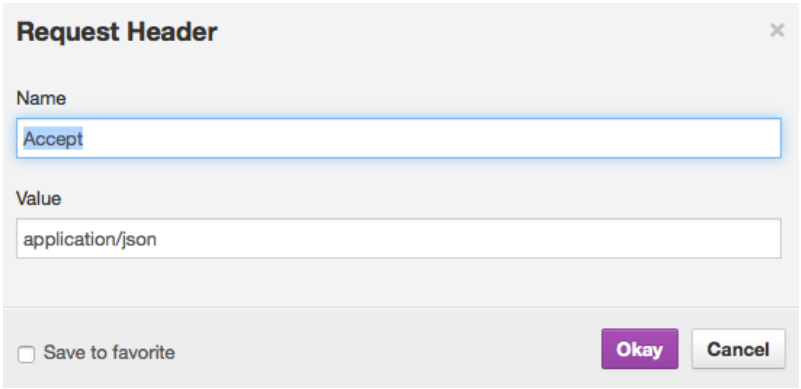

Figura 23 - Cabecera Accept

Como ejemplo ilustrativo se detalla la petición para desactivar el Device http://localhost:8222/devices/DEV-PORTA.FINCA

En primer lugar desde RESTClient seleccionaremos la operación PUT y en el *body* añadiremos el siguiente JSON:

```
{"action":"disable"}
```

Añadimos las cabeceras HTTP *Accept* y *Content-Type* que ya hemos creado y realizamos la petición a la URI del recurso:

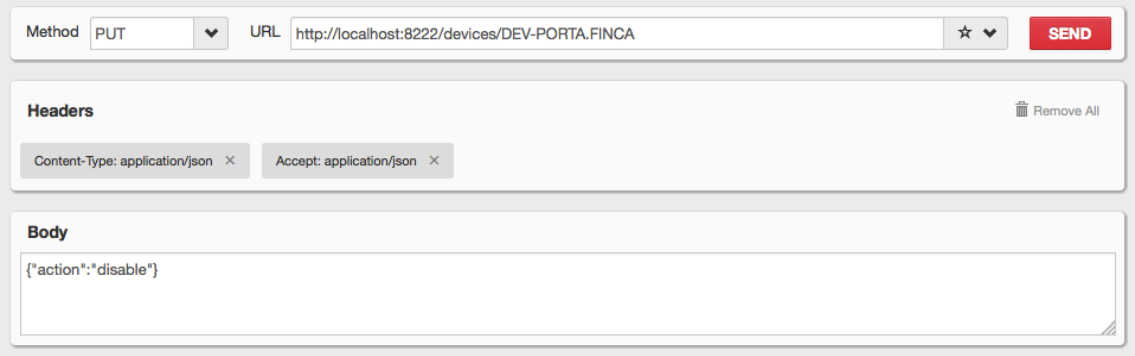

Figura 24 - Petición PUT a un Device

Como resultado obtendremos, si la petición tiene éxito, una respuesta OK por parte del servidor:

```
Status Code       : 200 OK
Accept-Ranges     : bytes
Content-Type      : application/json; charset=UTF-8
Date              : Sat, 17 Jan 2015 14:44:04 GMT
Server            : Restlet-Framework/2.1.7
Transfer-Encoding : chunked
Vary              : Accept-Charset, Accept-Encoding, Accept-Language, Accept
```

Figura 25 - Respuesta de petición PUT

La operación PUT no devuelve ningún cuerpo (payload) en su respuesta, por lo que a continuación realizamos una operación GET sobre la URI del recurso, obteniendo el siguiente JSON:

```
{
  "Name": "Porta Finca",
  "Link": "/devices/DEV-PORTA.FINCA",
  "Is_Enabled": false,
  "Is_Quiescent": false,
  "Locations": [
    {
      "Link": "/location/LOC-EXTERIOR",
      "Id": "LOC-EXTERIOR"
    }
  ],
  "DeviceFunctionalities": [
```

```
    {
      "Name": "Forrellat",
      "Current_State": "UNKNOWN",
      "Link": "/dfs/DF-PORTA.FINCA.FORRELLAT",
      "Previous_State": "UNKNOWN",
      "FuncID": "DF-PORTA.FINCA.FORRELLAT"
    }
  ],
  "Id": "DEV-PORTA.FINCA",
  "Is_Started": true,
  "URI": "http://localhost:8222/devices/DEV-PORTA.FINCA"
}
```


Se ha señalado en rojo que el Device efectivamente pasa a estar desactivado.

4.3.4. DeviceFunctionalities

El objetivo de este recurso es proporcionar un modo de poder listar todas las funcionalidades presentes en la vivienda, dotando de cierta jerarquía lógica a la estructura de recursos de la SmartHome, y estableciendo una división por tipo de recurso que diferencie fácilmente entre recursos del tipo Device (que se listan con Devices) y recursos del tipo DeviceFunctionality, listados con el recurso DeviceFunctionalities. Las únicas acciones permitidas sobre DeviceFunctionalities serán GET y OPTIONS. En la siguiente tabla se expone de forma resumida el subconjunto de la interfaz uniforme implementada por este recurso:

RECURSO	DeviceFunctionalities
URI	http://smarthomeserver:8222/dfs
OPERACIONES	
OPTIONS	Obtenemos el conjunto de operaciones de la interfaz uniforme disponible para este recurso (GET y OPTIONS)

GET	Obtenemos una representación en JSON del recurso DeviceFunctionalities compuesta por:
	• Un listado de todos los recursos del tipo DeviceFunctionality que están implementados en la SmartHome. Por cada DeviceFunctionality presente en el listado se muestran estos datos:
	  ○ Nombre de la DeviceFunctionality
	  ○ URI relativa (Link)
	  ○ Identificador (FuncID)
	• URI del recurso DeviceFunctionalities

Tabla 4 - Interfaz uniforme del recurso DeviceFunctionalities

A continuación se detallan las operaciones de la interfaz uniforme implementadas por este tipo de recurso.

4.3.4.1. Operación OPTIONS

La operación OPTIONS devuelve en la cabecera HTTP los métodos soportados por este tipo de recurso, como se aprecia en la siguiente imagen.

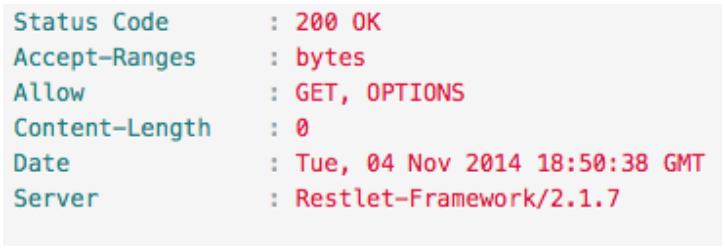

Figura 26 - Operación OPTIONS

Los métodos de la interfaz uniforme HTTP implementados por este recurso aparecen en la cabecera *Allow*.

4.3.4.2. Operación GET

La operación GET se ejecuta mandando la petición a la URI http://localhost:8222/dfs El resultado que se obtiene de esta consulta es un JSON que muestra un listado con información resumida de todos los recursos DeviceFunctionality registrados. Para cada DeviceFunctionality registrada se muestra su nombre (Name), su URI relativa (Link) y su identificador (FuncId). A continuación se muestra un fragmento del mismo.

```
{
  "DeviceFunctionalities": [
    {
      "Name": "Llums Pis",
      "Link": "/dfs/DF-IL.TOT",
      "FuncID": "DF-IL.TOT"
    },
    {
      "Name": "Polsadors - Display",
      "Link": "/dfs/DF-SP.DISPLAY",
      "FuncID": "DF-SP.DISPLAY"
    },
    {
      "Name": "Polsadors - Leds",
      "Link": "/dfs/DF-SP.LEDS",
      "FuncID": "DF-SP.LEDS"
    },
    {
      "Name": "Polsadors - Rellotge",
      "Link": "/dfs/DF-SP.RELLOTGE",
      "FuncID": "DF-SP.RELLOTGE"
    },

###  SE HAN BORRADO DEVICES DEL LISTADO PARA REDUCIR SU TAMAÑO  ###
    {
      "Name": "Sensor Alba/Crepuscle",
      "Link": "/dfs/DF-ESTACIO.METEO.ALBA-CREPUSCLE",
```

```
      "FuncID": "DF-ESTACIO.METEO.ALBA-CREPUSCLE"
    },
    {
      "Name": "Forrellat",
      "Link": "/dfs/DF-PORTA.FINCA.FORRELLAT",
      "FuncID": "DF-PORTA.FINCA.FORRELLAT"
    }
  ],
  "URI": "http://localhost:8222/dfs"
}
```


Tal y como se puede apreciar en el extracto del JSON, la última línea indica la URI desde la cual es accesible el recurso DeviceFunctionalities.

4.3.5. DeviceFunctionality

Este tipo de recursos representan las funcionalidades que los recursos Device ofrecen. El objetivo principal de la interfaz REST es proporcionar acceso a estas funcionalidades a aplicaciones externas al sistema de la SmartHome. Las operaciones disponibles para este tipo de recurso son OPTIONS, GET y PUT.

En la siguiente tabla se muestra un resumen de su subconjunto de la interfaz uniforme.

RECURSO	DeviceFunctionality
URI	http://localhost:8222/dfs/{id}
OPERACIONES	
OPTIONS	Obtenemos el listado de operaciones disponible para este contenedor (GET, HEAD, OPTIONS y PUT)
GET	Obtenemos una representación de la funcionalidad compuesta por: • Nombre de la DeviceFunctionality • Identificador del recurso (ID) • URI relativa (Link) • Estado actual del recurso • Estado anterior del recurso • Disponibilidad del recurso (si está activo o no) • Tipo de DeviceFunctionality • Subtipo de DeviceFunctionality • Si el recurso está o no iniciado • Última acción realizada • Device al cual se encuentra enlazada la DeviceFunctionality. Del recurso Device enlazado se muestra: ○ Nombre del recurso

	○ Identificador (ID) ○ URI relativa (Link) • URI completa de la DeviceFunctionality • Otros datos menos relevantes, como la lista de localizaciones (*locations*) y su estado de quiescencia.
PUT	La operación PUT se emplea para que el recurso realice determinada acción. La acción se incluye en el cuerpo de la petición PUT en formato JSON. Las acciones disponibles son distintas según el tipo de DeviceFunctionality. Por otra parte, las operaciones comunes a todas las DeviceFunctionalities son: • *enable*: activa una DeviceFunctionality desactivada • *disable*: desactiva una DeviceFunctionality activa • *togEnable*: conmuta el estado actual a su opuesto • *read*: captura el valor actual del recurso.

Tabla 5 - Interfaz uniforme del recurso DeviceFunctionality

4.3.5.1. Operación OPTIONS

Cuando realizamos una petición OPTIONS sobre un recurso DeviceFunctionality obtenemos unas cabeceras como estas, en las que se indica que las operaciones disponibles para este tipo de recurso son GET, OPTIONS y PUT:

```
Status Code       : 200 OK
Accept-Ranges     : bytes
Allow             : GET, OPTIONS, PUT
Content-Length    : 0
Date              : Sun, 26 Oct 2014 12:32:32 GMT
Server            : Restlet-Framework/2.1.7
```

Figura 27 - Cabecera de OPTIONS

4.3.5.2. Operación GET

La operación GET se realiza directamente sobre la URI del recurso. Como respuesta a esta petición se obtiene un JSON con información estructurada sobre la funcionalidad consultada. A continuación se expone el JSON generado por una consulta GET sobre el recurso del tipo DeviceFunctionality con URI http://localhost:8222/dfs/DF-MENJ.IL.AUXILIAR.


```
{
  "Device": {
    "Name": "Llum Menjador",
    "Link": "/device/DEV-MENJ.IL",
    "Id": "DEV-MENJ.IL"
  },
  "Is_Enabled": true,
  "Type": "togglebistate",
  "Is_Quiescent": false,
```

```
  "Locations": [
    {
      "id": "LOC-MENJ",
      "link": "/location/LOC-MENJ"
    }
  ],
  "Previous_State": "UNKNOWN",
  "Is_Started": true,
  "Name": "Llum Menjador Auxiliar",
  "Tags": [
    {
      "Link": "/tag/BaixarPersianesAlEncendreLlum",
      "Id": "BaixarPersianesAlEncendreLlum"
    }
  ],
  "Current_State": "UNKNOWN",
  "Subtype": "ON_OFF",
  "Classifiers": [
    {
      "Link": "/classifier/ED:IL",
      "Id": "ED:IL"
    }
  ],
  "Link": "/dfs/DF-MENJ.IL.AUXILIAR",
  "Last_Action": "unknown",
  "Id": "DF-MENJ.IL.AUXILIAR",
  "URI": "http://localhost:8222/dfs/DF-MENJ.IL.AUXILIAR"
}
```


A continuación se detallan algunas de las secciones del JSON generado:

- **Tags**: Etiquetas que sirven para ayudar a identificar el propósito de la DeviceFunctionality. En este caso el valor indica que esta DeviceFunctionality baja las persianas de una ventana al encender determinada luz.
- **Current-state**: Indica el estado actual de la DeviceFunctionality. El estado UNKNOW que aparece significa que no sabemos el estado actual, puesto que no hemos interactuado con él y por tanto el estado es desconocido.
- **Last-action**: Indica la última acción que se ha efectuado sobre la DeviceFunctionality. Como aún no hemos efectuado acción alguna, su estado es también UNKNOW.
- **Is-enabled**: Indica si la DeviceFunctionality se encuentra en estado de atender o no peticiones. Se trata de un estado booleano con dos posibles valores: true o false.
- **Type**: Indica el tipo de la DeviceFunctionality. En este ejemplo se trata del tipo *togglebistate* (un biestable que permite conmutar sus estados).
- **Id**: Identifica la DeviceFunctionality y se utiliza para construir la URI.
- **Locations**: Es una matriz que indica en qué localizaciones se puede encontrar al Device asociado. En este caso solo hay una, LOC-MENJ, que identifica al comedor de la vivienda.
- **Is-started**: Indica si el Device asociado ha sido arrancado.

- **Subtype**; Indica el subtipo de la DeviceFunctionality. En este caso es un biestable, indicado por el valor ON_OFF.
- **Name**: Este es el nombre de la DeviceFunctionality, en este caso *Llum Menjador Auxiliar*.
- **Device**: Muestra los datos referentes al Device enlazado con la DeviceFunctionality.
    - **ID**: Identificador del Device, en este caso DEV-MENJ.IL.
    - **Name**: Nombre del Device.
    - **Link**: Se emplea para construir la URI del Device. Si añadimos la URI de la SmartHome a la información de este link obtendremos la URI del Device: http://localhost:8222/device/DEV-MENJ.IL. Este es uno de los posibles métodos que REST ofrece para obtener las URIS canónicas de los Devices a través de los JSON de sus DeviceFunctionalities.
- **Previus-state**: Estado anterior a la operación efectuada. Como no hemos efectuado ninguna operación, su valor es UNKNOW.
- **URI**: La URI de la DeviceFunctionality.

Las cabeceras HTTP que se obtienen al resolver una petición GET con éxito sobre un recurso DeviceFunctionality son las siguientes:

```
Status Code       : 200 OK
Accept-Ranges     : bytes
Content-Type      : application/json; charset=UTF-8
Date              : Sun, 26 Oct 2014 12:19:23 GMT
Server            : Restlet-Framework/2.1.7
Transfer-Encoding : chunked
Vary              : Accept-Charset, Accept-Encoding, Accept-Language, Accept
```

Figura 28 - Cabecera de respuesta GET

Se puede apreciar en la cabecera que se devuelve un JSON, lo cual viene indicado en el valor de *Content-Type*.

Como ya se ha comentado en el punto 4.3.3.2. Operación GET, relativo a los recursos Device, los recursos del tipo DeviceFunctionality están enlazados con un único recurso del tipo Device, al contrario de lo que sucedía con los recursos Device que podían tener enlazados varios recursos DeviceFunctionalities. Es lógico, ya que normalmente una funcionalidad estará presente en un dispositivo concreto y solo en uno, pero ese dispositivo puede disponer de más de una funcionalidad.

En el JSON se pueden visualizar algunos datos del Device al que pertenece la DeviceFunctionality. Particularmente interesantes de cara a la interactividad entre recursos son su identificador y su URI relativa (Link), mediante la cual podemos

construir la URI completa del Device, cumpliendo una vez más con el **principio de interconexión de recursos de REST.**

4.3.5.3. Operación PUT

La ejecución de una petición PUT es más compleja, puesto que este tipo de petición debe ir acompañada de un cuerpo que contenga la orden que se debe ejecutar. El envío de ordenes a los recursos DeviceFunctionality será de nuevo en formato JSON. De esta forma, en el JSON se utiliza un parámetro llamado "*action*" que indica que queremos efectuar una acción sobre la DeviceFunctionality, la cual vendrá determinada por el valor de este parámetro "*action*".

El valor que puede tomar este parámetro depende del tipo de DeviceFunctionality, puesto que cada tipo implementa acciones diferentes, aparte de las acciones que son comunes a todas ellas. Los diferentes tipos de DeviceFunctionality existentes y las operaciones implementadas en cada uno de ellos son:

- **bistate**
  - **biaON**: cambia el estado a ON.
  - **biaOFF**: cambia el estado a OFF.
- **togglebistate**
  - **biaToggle**: conmuta el estado actual a su opuesto.
- **dimmer**
  - **pulseON**: envía una señal de activación.
  - **pulseOFF**: envía una señal de desactivación.
  - **Toggle**: conmuta el estado actual a su opuesto.
- **movement**
  - **diaSET%**: establece el valor en porcentaje
  - **diaSETº**: establece el valor en angular
  - **diaSETox**: establece el valor en hexadecimal
- **numeric value**
  - **numaSET**: establece el valor numérico

Un ejemplo de JSON que envía a un DeviceFunctionality del tipo *bistate* la orden de que el biestable pase a modo ON sería el siguiente:

```
{"action":"biaON"}
```

Además, para que nuestro cliente REST sea capaz de enviar JSON en el cuerpo de la petición, y que estos datos sean interpretados y tratados adecuadamente, debemos modificar la cabecera de envío de la petición para indicar, en primer lugar, que el contenido es del tipo JSON, y en segundo lugar, que el servidor web debe aceptar JSON como formato de entrada de la petición.

Como ya se ha detallado en el punto 4.3.3.3. Operación PUT, relativo a la operación PUT de los recursos Device, esto se realiza fácilmente desde el menú *Headers* de RESTClient, que permite añadir y modificar las cabeceras de nuestras peticiones.

Crearemos dos cabeceras personalizadas con el siguiente contenido (si no las hemos creado ya en puntos anteriores):

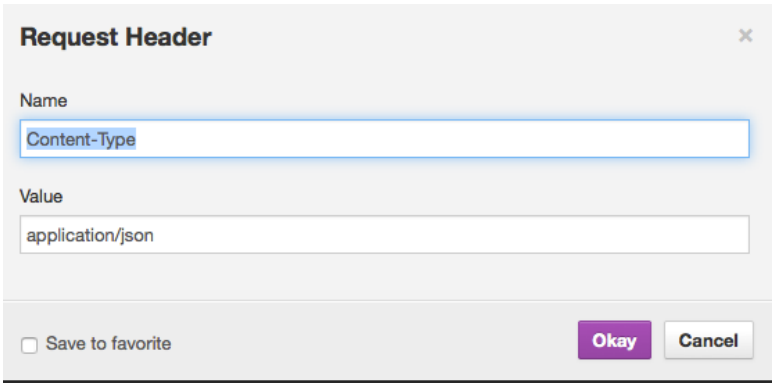

Figura 29 - Cabecera Content-Type

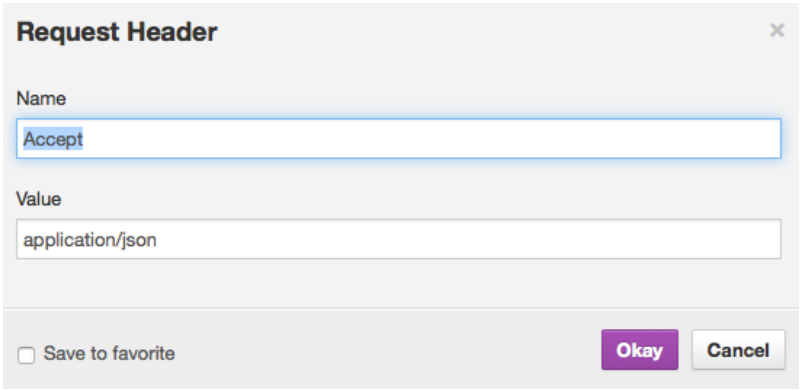

Figura 30 - Cabecera Accept

Finalmente, la ejecución de nuestra petición PUT desde RESTClient quedaría como en la siguiente imagen:

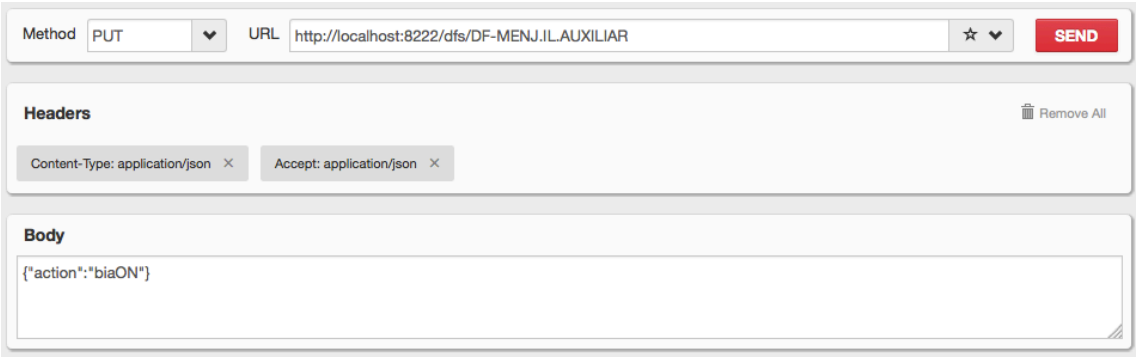

Figura 31 - Operación PUT

Según el propio estándar de HTTP, una petición PUT no tiene el porqué devolver nada. La petición se ejecuta en el servidor, realiza las operaciones solicitadas en el

recurso indicado en la URI y devuelve solamente el estado indicando si la petición ha tenido éxito o no.

```
1.  Status Code       : 200 OK
2.  Accept-Ranges     : bytes
3.  Content-Length    : 0
4.  Date              : Sat, 21 Feb 2015 17:55:02 GMT
5.  Server            : Restlet-Framework/2.1.7
```

Figura 32 - Cabeceras de PUT sobre una DeviceFunctionality

No vemos el nuevo estado del recurso, si lo hemos modificado, ni ningún otro resultado de la operación efectuada. Para poder ver los cambios a continuación debemos realizar una operación GET sobre el recurso. Veremos que en este caso se han cambiado determinados valores.

El JSON que obtenemos tras procesar la operación PUT en el recurso anterior es este:


```
{
  "Device": {
    "Name": "Llum Menjador",
    "Link": "/device/DEV-MENJ.IL",
    "Id": "DEV-MENJ.IL"
  },
  "Is_Enabled": true,
  "Type": "togglebistate",
  "Is_Quiescent": false,
  "Locations": [
    {
      "id": "LOC-MENJ",
      "link": "/location/LOC-MENJ"
    }
  ],
  "Previous_State": "UNKNOWN",
  "Is_Started": true,
  "Name": "Llum Menjador Auxiliar",
  "Tags": [
    {
      "Link": "/tag/BaixarPersianesAlEncendreLlum",
      "Id": "BaixarPersianesAlEncendreLlum"
    }
  ],
  "Current_State": "bisON",
  "Subtype": "ON_OFF",
  "Classifiers": [
    {
      "Link": "/classifier/ED:IL",
      "Id": "ED:IL"
    }
  ],
  "Link": "/dfs/DF-MENJ.IL.AUXILIAR",
  "Last_Action": "biaON",
```

```
  "Id": "DF-MENJ.IL.AUXILIAR",
  "URI": "http://localhost:8222/dfs/DF-MENJ.IL.AUXILIAR"
}
```


Se han marcado en rojo las diferencias respecto al JSON obtenido en la operación GET anterior. Tras efectuar la acción *biaON* podemos ver que el estado actual *(current-state)* pasa a ser *bisON*, indicando que el Device está en estado ON. Además se indica que la última acción *(last-action)* realizada sobre el Device ha sido *biaON*. El resto de datos permanecen inalterados, como era de esperar. También podemos destacar que el estado anterior (*previous-state*) sigue siendo desconocido. Esto es cierto, puesto que el estado anterior, en el que realizábamos una operación GET sobre un dispositivo recién inicializado era desconocido.

4.3.5.4. La idempotencia de la operación PUT

¿Qué sucederá si efectuamos una operación para cambiar el estado del Device del ejemplo anterior a OFF? Deberemos enviar una petición PUT cuyo JSON sea este:

```
{"action":"biaOFF"}
```

Si a continuación se efectúa un GET sobre esta DeviceFunctionality el JSON obtenido será este otro:


```
{
  "Device": {
    "Name": "Llum Menjador",
    "Link": "/device/DEV-MENJ.IL",
    "Id": "DEV-MENJ.IL"
  },
  "Is_Enabled": true,
  "Type": "togglebistate",
  "Is_Quiescent": false,
  "Locations": [
    {
      "id": "LOC-MENJ",
      "link": "/location/LOC-MENJ"
    }
  ],
  "Previous_State": "bisON",
  "Is_Started": true,
  "Name": "Llum Menjador Auxiliar",
  "Tags": [
    {
      "Link": "/tag/BaixarPersianesAlEncendreLlum",
      "Id": "BaixarPersianesAlEncendreLlum"
    }
  ],
  "Current_State": "bisOFF",
  "Subtype": "ON_OFF",
  "Classifiers": [
    {
      "Link": "/classifier/ED:IL",
      "Id": "ED:IL"
```

```
    }
  ],
  "Link": "/dfs/DF-MENJ.IL.AUXILIAR",
  "Last_Action": "biaOFF",
  "Id": "DF-MENJ.IL.AUXILIAR",
  "URI": "http://localhost:8222/dfs/DF-MENJ.IL.AUXILIAR"
}
```


Vemos que efectivamente, el estado actual ha cambiado a *bisOFF* indicando que el Device se encuentra en estado OFF. Además se indica que la última acción efectuada ha sido *biaOFF*. Pero en este caso si se ha cambiado el valor del estado previo, indicando que antes de efectuar la operación *biaOFF* el estado del Device era *bisON*.

¿Y si volvemos a efectuar la misma operación PUT *biaOFF* al mismo recurso DeviceFunctionality de nuevo? Es decir, realizamos una nueva petición PUT enviando este mismo JSON.

```
{"action":"biaOFF"}
```

a un Device que ya se encuentra en estado *bisOFF*. Lo que sucederá se puede observar en el JSON obtenido tras efectuar a continuación una nueva operación GET.


```
{
  "Device": {
    "Name": "Llum Menjador",
    "Link": "/device/DEV-MENJ.IL",
    "Id": "DEV-MENJ.IL"
  },
  "Is_Enabled": true,
  "Type": "togglebistate",
  "Is_Quiescent": false,
  "Locations": [
    {
      "id": "LOC-MENJ",
      "link": "/location/LOC-MENJ"
    }
  ],
  "Previous_State": "bisON",
  "Is_Started": true,
  "Name": "Llum Menjador Auxiliar",
  "Tags": [
    {
      "Link": "/tag/BaixarPersianesAlEncendreLlum",
      "Id": "BaixarPersianesAlEncendreLlum"
    }
  ],
  "Current_State": "bisOFF",
  "Subtype": "ON_OFF",
  "Classifiers": [
    {
      "Link": "/classifier/ED:IL",
      "Id": "ED:IL"
    }
  ],
```


```
  "Link": "/dfs/DF-MENJ.IL.AUXILIAR",
  "Last_Action": "biaOFF",
  "Id": "DF-MENJ.IL.AUXILIAR",
  "URI": "http://localhost:8222/dfs/DF-MENJ.IL.AUXILIAR"
}
```

Podemos observar que **no existen diferencias entre este JSON y el obtenido en la petición *biaOFF* anterior.** Esto se debe a que nuestra operación PUT cumple con uno de los principios del estándar HTTP: la **idempotencia**. Este principio se detalla en 3.6.5 Seguridad e idempotencia y significa que no importa el número de veces que realicemos una misma operación consecutivamente sobre el mismo recurso, ya que el resultado obtenido será siempre el mismo. Las operaciones GET y OPTIONS también cumplen con el principio de idempotencia.

De esta forma también se explica el porqué no varía el campo *previous-state* entre una petición *biaOFF* y otra consecutiva, ya que realmente no se ha llegado a cambiar el estado del Device. Por lo tanto el valor del campo previous-state debe permanecer inalterado hasta que el estado del Device cambie con una nueva orden biaON.

5. Implementación de la aplicación.

Una vez concluida la etapa de diseño y análisis de la aplicación, se pasa a la fase de implementación de la misma.

El lenguaje de desarrollo elegido es Java, por los motivos expuestos en el punto 2. Tecnologías empleadas. La aplicación se basa concretamente en RESTLet, que es un Framework de desarrollo de APIS basadas en REST para Java, como se detalla en el punto 2.4 RESTlet Framework de la presente memoria.

En los puntos 1.2 Ámbito y 1.3 Objetivos se menciona que el propósito del proyecto no es desarrollar el sistema completo de la SmartHome y los recursos asociados a la misma, sino desarrollar una interfaz REST que permita a aplicaciones clientes externas al sistema interactuar con dichos recursos. Por lo tanto **el desarrollo de la interfaz REST parte de la base del sistema SmartHome ya implementado**, y que queda fuera del ámbito de este trabajo. El desarrollo de la API se basará en el expuesto en [14], adaptando la aplicación de los principios REST a una interfaz de interoperabilidad para los recursos presentes en el sistema de partida.

5.1. Estructura de una aplicación RESTLet

Una aplicación RESTlet típica se diseña utilizando el concepto de capas. Cada capa interna es englobada por otra externa de modo que las llamadas procedentes del exterior del sistema son atendidas en primer lugar por las capas externas. Si una capa no puede atender una llamada, pasa la llamada a la capa siguiente, de modo que cuando la llamada llega a la capa interna el recurso ya tiene toda la información necesaria para atenderla de forma adecuada.

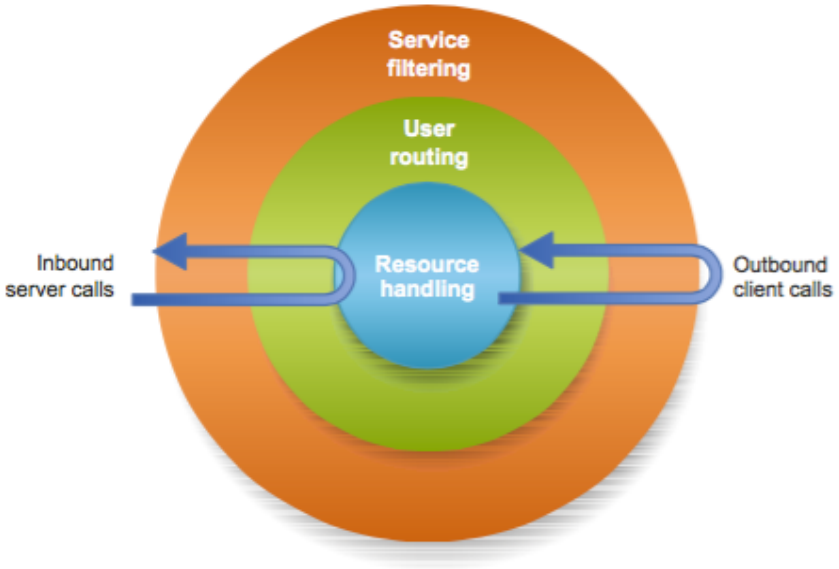

Figura 33 - Capas de una aplicación RESTlet. Extraído de [14]

Como se puede apreciar en la figura anterior, una llamada externa, o llamada entrante con destino al servidor (*inbound server calls* en el diagrama) pasa primero por la capa exterior que filtra el servicio que se solicita. Esta capa es común a todos los recursos de la aplicación.

Una vez filtrada, la petición pasa por un segundo filtro que la dirige hacia el recurso solicitado. Este filtro lo podemos configurar según las necesidades de nuestro sistema, y sirve a su vez para especificar la forma de construir las URIs canónicas de cada tipo de recurso presente en nuestra aplicación.

Por último, la petición llega al recurso concreto solicitado, el cual la atenderá de forma apropiada y enviará, si procede, una respuesta a la petición (*outbound client calls* en el diagrama). En el siguiente diagrama se clarifica cómo una petición externa va pasando por los sucesivos filtros hasta que finalmente es atendida por el recurso apropiado.

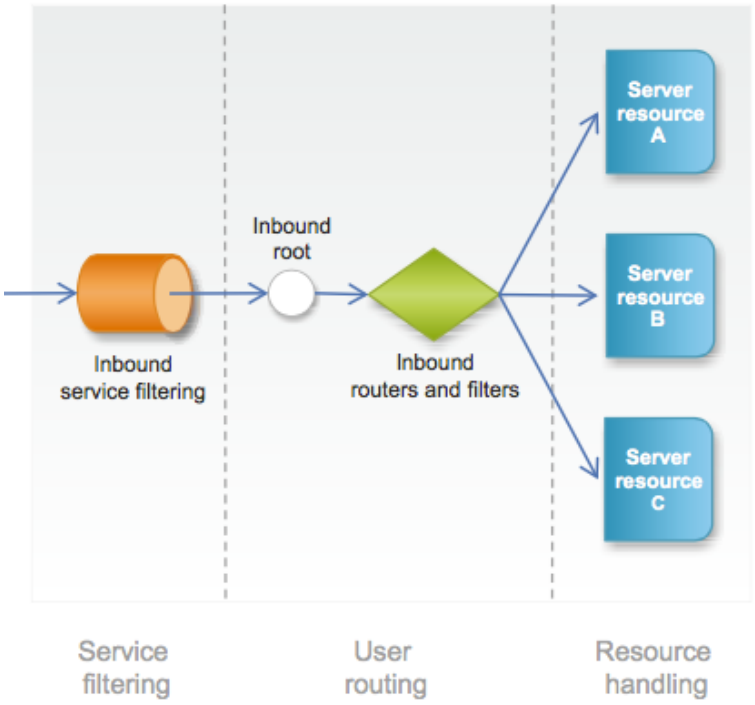

Figura 34 - Filtros de entrada. Extraído de [14]

Por lo tanto, aplicando este modelo, la aplicación RESTlet queda compuesta por los siguientes elementos:

- **Un filtro previo** que se utiliza para filtrar peticiones al servicio, por motivos de seguridad, o para redirigir ciertas peticiones a otros servicios (*Service filtering*).
- **Una clase principal** que se emplea para levantar la aplicación y proporcionar el servicio que atiende las llamadas (*Inbound root*).
- **Un filtro posterior** que redirige las peticiones a los recursos (*User routing*)
- **Las clases de los recursos** propiamente dichos que son las que atienden las peticiones y mandan las respuestas apropiadas (*Resource handling*).

5.2. Esquema general de la aplicación.

Siguiendo el esquema RESTlet expuesto en el punto anterior, nuestra aplicación está compuesta por tres componentes fundamentales:

- **La clase SmartHomeComponent** que levanta el servicio, instancia la aplicación e implementa el filtro *Inbound Service*.
- **La clase SmartHomeApplication** que establece las reglas del router interno para redirigir las peticiones entrantes a los recursos adecuados (*Inbound root*).

- **Las clases Resource** que atienden las peticiones a cada tipo de recursos y devuelven la respuesta adecuada (*Resource handling*). Existirá una clase de tipo Resource por cada tipo de recurso existente.

Gráficamente se puede representar de la siguiente manera:

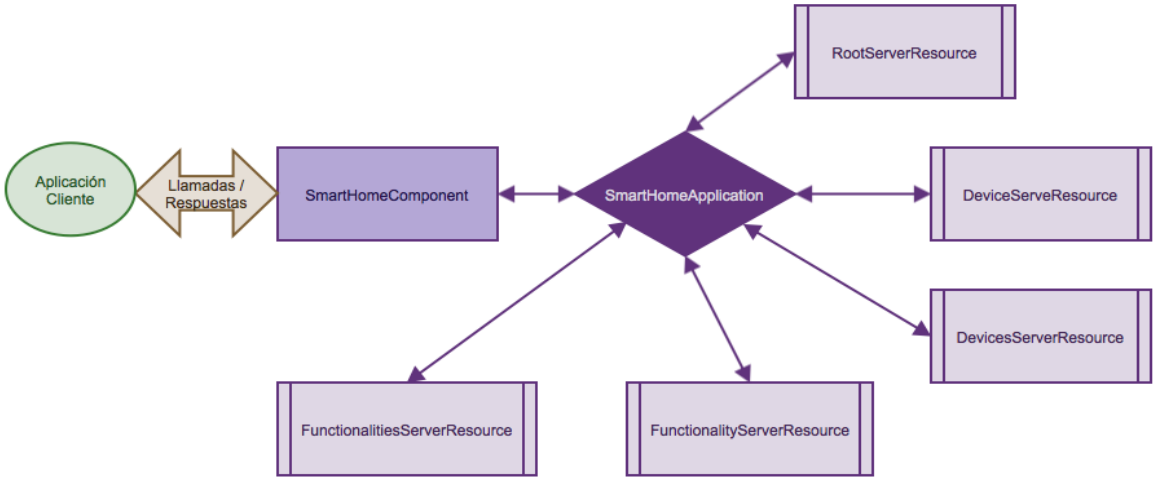

Figura 35 - Esquema de la aplicación

En el gráfico se detalla como las peticiones procedentes de las aplicaciones clientes enviadas a nuestro sistema SmartHome son en primera instancia filtradas y enviadas a la clase correcta para atender la petición. El filtro se realiza en función de la URI a la cual se envía la petición.

El primer filtro (*Inbound Service*) se realiza en la clase *SmartHomeComponent* y determina a qué servicio se está mandando la petición. En nuestro caso tan solo contamos con el servidor *localhost* escuchando en el puerto 8222, pero un sistema real podría contar con múltiples servidores escuchando cada uno peticiones en un puerto diferente.

El segundo filtro (*Inbound root*) se implementa en la clase *SmartHomeApplication*, la cual redirige la petición a la clase del tipo *Resource* apropiada.

Las clases *Resource* que atienden a las peticiones a los recursos se componen, además de por:

- **La clase RootServerResource,** que implementa la interfaz *RootResource* y da respuesta a las peticiones enviadas a la URI raíz del servidor http://localhost:8222
- **La clase DeviceServerResource,** que implementa la interfaz *DeviceResource* y atiende las peticiones dirigidas a recursos de tipo Device.

- **La clase DevicesServerResource**, que implementa la interfaz *DevicesResource* y atiende las peticiones dirigidas al recurso Devices
- **La clase FunctionaliyServerResource**, que implementa la interfaz *FunctionalityResource* y atiende las peticiones dirigidas a recursos del tipo DeviceFunctionality.
- **La clase FunctionalitiesServerResource**, que implementa la interfaz *FunctionalitiesResource* y atiende las peticiones dirigidas al recurso DeviceFunctionalities.

Siguiendo el esquema de la figura anterior, si se recibe una petición GET en la URI http://smarthomeserver:8222 la API filtrará esta petición, determinando que el destino es el servidor *smarthomeserver* que escucha en el puerto 8222, y que el destino de la petición es el recurso *RootResource*, por lo que la petición será atendida por el método de la clase *RootServerResource* que atienda las peticiones GET, el cual devolverá un JSON como este:

```
{
  "Name": "RESTful SmartHome Server",
  "Status": "OK (200) - The request has succeeded",
  "Description": "This is a prototype of a SmartHome Server",
  "URI": "http://localhost:8222/"
}
```

A continuación se detallan algunas de las clases Java más importantes de la aplicación. En los Anexos se establece la jerarquía de clases y su clasificación en los diferentes paquetes o *packages,* a los cuales se hace referencia en los puntos siguientes.

5.3. La clase Configuration

Esta clase, ubicada en el *package* smarthomes.rest.api.configuration, se utiliza para proporcionar los parámetros configurables del resto de la aplicación, los cuales se emplean reiteradamente en la implementación del resto de clases. La implementación de esta clase es la siguiente:

```
package smarthomes.rest.api.configuration;

public interface Configuration {
	public final static int SERVER_PORT = 8222;
	public final static String ROUTER_FUNCID = "funcId";
	public final static String ROUTER_DEVID = "devId";
	public final static String FUNCTIONALITIES_URI = "dfs";
	public final static String DEVICES_URI = "devices";
	public final static String COMPONENT_NAME = "RESTful SmartHome Server
                                                Component";
	public final static String COMPONENT_DESCRIPTION = "This is a prototype of a
                                                    restlet component";
```


```
        public final static String COMPONENT_OWNER = "RestFul SmartHome server API
                                                     project";
        public final static String COMPONENT_AUTHOR = "The Restlet SmartHome PFG
                                                      Team";
        public final static String APPLICATION_NAME = "RESTful SmartHome Server";
        public final static String APPLICATION_DESCRIPTION = "This is a prototype of a
                                                             SmartHome Server";
        public final static String APPLICATION_OWNER = "RestFul SmartHome server API
                                                       project";
        public final static String APPLICATION_AUTHOR = "The Restlet SmartHome PFG
                                                        Team";
    }
```

A continuación se detallan cada uno de los parámetros de configuración más importantes.

- **SERVER_PORT**: configura el puerto en el que el servidor de la aplicación atenderá las peticiones. Se establece en el 8222.
- **ROUTER_FUNCID**: proporciona parte de la cadena utilizada en la construcción de las URIs de los recursos del tipo DeviceFunctionality. En concreto representa al identificador de la DeviceFunctionality a la que se envía la petición. Se emplea en la construcción del filtro de llamadas entrantes (*Inbound router*).
- **ROUTER_DEVID**: proporciona parte de la cadena para de construcción de las URIs de los recursos del tipo Device. En concreto representa al identificador del Device al que se envía la petición. Se emplea en la construcción del filtro de llamadas entrantes (*Inbound router*).
- **FUNCTIONALITIES_URI**: establece parte de la cadena que se utiliza para la construcción de la URIs del recurso del tipo DeviceFunctionalies. Combinando este parámetro con ROUTER_FUNCID se obtiene la URI relativa (Link) del recurso DeviceFunctionality al que va dirigida la petición. Forma parte del filtro de llamadas entrantes (*Inbound router*).
- **DEVICES_URI**: establece parte de la cadena que se utiliza para la construcción de la URIs del recurso del tipo Devices. Combinando este parámetro con ROUTER_DEVID se obtiene la URI relativa (Link) del recurso Device al que va dirigida la petición. Se utiliza en el filtro de llamadas entrantes (*Inbound router*).

Los parámetros cuyo nombre empieza por COMPONENT_ son referentes a la configuración de la clase SmartHomeComponent, encargada de lanzar la aplicación.

Los parámetros cuyo nombre empieza por APPLICATION_ son propios de la configuración del servidor de la API , y como su nombre indica sirven para establecer el nombre, la descripción y el propietario, tanto de la aplicación como del propio servidor de la API. Algunos de estos parámetros son empleados en el JSON generado al realizar una consulta GET sobre la URI http://localhost:8222

5.4. La clase SmartHomeApplication

Esta clase es la clase principal de la aplicación, ya que implementa el filtro *Inbound root* encargado de redirigir las peticiones a las clases que las atienden. Sin esta clase no se podrían enviar peticiones a los recursos. Esta clase se ubica en el *package* smarthomes.rest.api.server y debe extender de la clase padre *Application* de RESTlet. La declaración de *packages* necesarios para su compilación es la siguiente:

```
package smarthomes.rest.api.server;

import org.osgi.framework.BundleContext;
import org.pros.upv.es.smarthomes.Tools.components.SearchTools;
import org.restlet.Application;
import org.restlet.Restlet;
import org.restlet.routing.Router;
```

Los parámetros declarados proporcionan el contexto necesario para acceder a los recursos presentes en el servidor de la SmartHome, así como las herramientas necesarias para localizar dichos recursos.

```
public class SmartHomeApplication extends Application{
	protected BundleContext context = null;
	protected SearchTools st = null;
```

El método constructor de la clase proporciona el contexto necesario y las herramientas de búsqueda de recursos.

```
	public SmartHomeApplication(BundleContext context) {
		this.context = context;
		this.st = new SearchTools(this.context);
	}
```

Se implementa a su vez un método que permita acceder a las herramientas de búsqueda.

```
	public SearchTools getSearchTools() {
		return this.st;
	}
```

A continuación se detallan los métodos más importantes de esta clase.

5.4.1. CreateInboundRoot()

RESTlet proporciona un método dentro de la clase *Application* especialmente ideado para dar soporte al filtro de llamadas entrantes (*Inbound root*). Este método debe sobre escribirse en nuestra clase extendida con tal de aplicar nuestros filtros personalizados. En este caso aplicaremos estos filtros:

```
@Override
	public Restlet createInboundRoot() {
		Router router = new Router(getContext());
		router.attach("/", RootServerResource.class);
		router.attach("/" + Configuration.FUNCTIONALITIES_URI,
				FunctionalitiesServerResource.class);
		router.attach("/" + Configuration.FUNCTIONALITIES_URI + "/{"+
					Configuration.ROUTER_FUNCID + "}",
					FunctionalityServerResource.class);
		router.attach("/" + Configuration.DEVICES_URI,
					DevicesServerResource.class);
		router.attach("/" + Configuration.DEVICES_URI + "/{" +
					Configuration.ROUTER_DEVID + "}",
					DeviceServerResource.class);
		return router;
	}
```

Los parámetros de configuración están detallados en la clase *SmartHomeConfiguration* del *package* smarthomes.rest.api.configuration y se detallan en el punto 5.2 La clase SmartHomeConfiguration.

Una vez configurado nuestro filtro, la aplicación atenderá estos tipos de peticiones:

- **Las dirigidas al servidor de la aplicación** a la URI / y que son atendidas por la clase *RootServerResource*.
- **Las dirigidas a recursos Device**, enviadas a URIs de la forma /devices/devId y atendidas por la clase *DeviceServerResource*.
- **Las dirigidas al recurso Devices** enviadas a la URI /devices y atendidas por la clase *DevicesServerResource*.
- **Las dirigidas a recursos DeviceFunctionality**, enviadas a URIs de la forma /dfs/funcId y atendidas por la clase *FunctionalityServerResource*.
- **Las dirigidas al recurso DeviceFunctionalities**, enviadas a la URI /dfs y atendidas por la clase *FunctionalitiesServerResource*.

Todas las clases que atienden las peticiones sobre recursos se alojan en el *package* smarthomes.rest.api.server e implementan su respectiva interfaz del *package* smarthomes.rest.api.common. Como ejemplo se describe la petición a un recurso del tipo Device. Observemos que la instrucción

```
router.attach("/" + Configuration.DEVICES_URI + "/{" +
		Configuration.ROUTER_DEVID + "}",
					DeviceServerResource.class);
```

según el valor de los parámetros DEVICES_URI y ROUTER_DEVID, redirige las peticiones enviadas a URIS construidas de la forma /devices/devId a la clase *DeviceServerResource*. Esta gestiona la petición y devuelve una respuesta. **La redirección de las peticiones entrantes a las correspondientes clases que las atienden se realiza en base a la URI del recurso a la que va dirigida dicha petición.**

5.4.2. SmartHomeApplication()

El constructor SmartHomeApplication establece el resto de parámetros de la aplicación: nombre, descripción, autor y propietario.

```
        public SmartHomeApplication(){
                setName(Configuration.APPLICATION_NAME);
                setDescription(Configuration.APPLICATION_DESCRIPTION);
                setOwner(Configuration.APPLICATION_OWNER);
                setAuthor(Configuration.APPLICATION_AUTHOR);
                setStatusService(new SmartHomeStatus());
        }
}
```

5.5. La clase SmartHomeComponent

Esta clase se ubica en el *package* smarthomes.rest.api.server. Es la encargada de iniciar al servidor web y de instanciar a la clase *SmartHomeApplication* y por lo tanto arranca y detiene al servidor que atiende las llamadas entrantes dirigidas a la aplicación e inicia la API que procesa las peticiones y envía las respuestas. Los paquetes necesarios para su implementación son estos:

```
package smarthomes.rest.api.server;

import org.osgi.framework.BundleContext;
import org.restlet.Component;
import org.restlet.data.Protocol;
import smarthomes.rest.api.configuration.Configuration;
```

En su implementación, como primer paso establecemos el contexto. Dicho contexto pertenece a la SmartHome sobre la que se aplica la interfaz REST, y es utilizado para localizar los recursos incializados.

```
        public class SmartHomeComponent extends Component{
                protected  BundleContext context = null;
```

5.5.1. SmartHomeComponent()

El método constructor crea un componente RESTlet que está compuesto por una instancia de la clase *SmartHomeApplication* a la que se le añaden otros parámetros, como el puerto del servidor. Estos parámetros se encuentran definidos en la clase *SmartHomeConfiguration* y se detallan en el punto 6.4 La clase SmartHomeConfiguration. El código fuente del método es el siguiente:

```
        public SmartHomeComponent(BundleContext context) {
                this.context=context;
                setName(Configuration.COMPONENT_NAME);
                setDescription(Configuration.COMPONENT_DESCRIPTION);
                setOwner(Configuration.COMPONENT_OWNER);
                setAuthor(Configuration.COMPONENT_AUTHOR);
                getServers().add(Protocol.HTTP,  Configuration.SERVER_PORT);
                getDefaultHost().attachDefault(new SmartHomeApplication(context));
        }
}
```

Observemos que se establece el protocolo y el puerto que va a ser empleado en la aplicación. En este caso se establece que el protocolo será HTTP, lo que nos permite utilizar la interfaz uniforme de este protocolo, y los comandos que implementa. Por otro lado el puerto se establece en el parámetro SERVER_PORT de la clase *SmartHomeConfiguration* y su valor es 8222. Por lo tanto la dirección URI del servidor según el protocolo establecido y el puerto de escucha, se establece en http://smarthomeserver:8222

Esta dirección raíz se concatenará con la dirección relativa de los recursos a consultar, con el fin de obtener la URI completa o canónica del recurso en cuestión.

5.6. La clase interfaz RootResource

Las clases del tipo interfaz se ubican todas en el *package* smarthomes.rest.api.common. Estas clases contienen las cabeceras de los métodos públicos que obligatoriamente deben implementar las clases que implementen la interfaz en su declaración.

Todas las interfaces del *package* smarthomes.rest.api.common son implementadas por una clase del *package* smarthomes.rest.api.server. Esta interfaz en concreto es implementada por la clase *RootServerResource*.

La clase *RootServerResource* implementa la representación del recurso principal de la aplicación, del cual colgarán el resto de recursos. A su vez, este recurso implementa los métodos GET y OPTIONS de la interfaz uniforme del protocolo HTTP, tal y como se señala en el punto 4.3. La interfaz uniforme y sus representaciones

Los métodos de la interfaz están pensados para sobre escribir los métodos HTTP y darles la funcionalidad adaptada a nuestros recursos. Estos son los *packages* necesarios para su compilación:

```
package smarthomes.rest.api.common;

import org.json.JSONException;
import org.restlet.representation.Representation;
import org.restlet.resource.Get;
import org.restlet.resource.Options;
```

Y la implementación de la clase es esta:

```
        public interface RootResource {
         @Get ("json")
          public Representation represent() throws JSONException;

        @Options ("wadl|html")
        public void commands();
}
```

El método *represent()* sobre escribe la operación GET del estándar HTTP para el recurso del tipo ROOTRESOURCE, el cual representa al servidor de la API. Devuelve una representación en JSON del recurso consultado.

Este método *commands()* sobre escribe la operación OPTIONS del estándar HTTP para los recursos del tipo ROOTRESOURCE (SmartHome). La respuesta de esta consulta devuelve el conjunto de operaciones HTTP soportadas por el recurso en la cabecera HTTP de la respuesta. Estas operaciones son GET y OPTIONS.

Estos dos métodos públicos deberán implementarse y sobre escribirse en la clase *RootServerResource* que extiende de esta clase.

5.7. La clase RootServerResource

Como ya se menciona en el punto anterior, esta clase se ubica en el *package* smarthomes.rest.api.server e implementa la interfaz *RootResource*.

La función de esta clase es la de atender las peticiones dirigidas al recurso principal de la aplicación a través de la URI http://localhost:8222. Los *packages* necesarios para su compilación son estos:

```
package smarthomes.rest.api.server;

import java.util.HashSet;
import java.util.Set;
import org.json.JSONException;
import org.json.JSONObject;
import org.restlet.data.Method;
import org.restlet.data.Reference;
import org.restlet.engine.header.Header;
import org.restlet.ext.json.JsonRepresentation;
```


```
import org.restlet.representation.Representation;
import org.restlet.resource.ResourceException;
import org.restlet.resource.ServerResource;
import org.restlet.util.Series;
import smarthomes.rest.api.common.RootResource;
```

Y esta es la implementación de la clase y la declaración de parámetros:

```
public class RootServerResource extends ServerResource implements RootResource{

        String metodo;
```

A continuación se detallan el resto de sus métodos más importantes:

5.7.1. setCustomHttpResponseHeader

El método *setCustomHttpResponseHeader* se utiliza para añadir nuevas cabeceras a la cabeceras HTTP de respuesta de las peticiones que se encuentran definidas por defecto en RESTlet. Se utiliza para sortear la limitación CORS de JavaScript del cliente web detallada en el punto 6.7. El problema de Origen Cruzado de JavaScript (CORS).

```
@SuppressWarnings({ "unchecked" })
public void setCustomHttpResponseHeader(String header, String value) {
        Series<Header> series =(Series<Header>)
        this.getResponseAttributes().get("org.restlet.http.headers");
        Series<Header> responseHeaders = series;
        if (responseHeaders == null) {
            responseHeaders = new Series<Header>(Header.class);
            this.getResponseAttributes().put("org.restlet.http.headers",
                                             responseHeaders);
        }
        responseHeaders.add(new Header(header, value));
    }
```

5.7.2. doInit()

Se sobre escribe el método *doInit()* de RESTlet que se utiliza para realizar acciones al inicializar el recurso. En este caso obtiene el método HTTP empleado para realizar la petición al recurso y establece el valor de la cabecera de respuesta *Access-Control-Allow-Origin* a "*". El uso de esta cabecera está relacionado con la restricción *Cross Origin* de JavaScript, la cual se detalla en el punto 6.7. El problema de Origen Cruzado de JavaScript (CORS)

```
        @Override
    protected void doInit() throws ResourceException {
              metodo = this.getMethod().toString();
              setCustomHttpResponseHeader("Access-Control-Allow-Origin", "*");
        }
```

5.7.3. represent()

El método público *represent()* es uno de los métodos públicos presentes en la interfaz *RootResource*, sobre escribe la operación GET del estándar HTTP para los recursos del tipo *RootResource*, es decir, la aplicación principal del API de la SmartHome. Devuelve una representación en JSON del recurso consultado.

```
@Override
	public Representation represent() throws JSONException{
		JSONObject SH = new JSONObject();
		//SH.put("Name", "SmartHome Server");
		SH.put("Name", Configuration.APPLICATION_NAME);
		//SH.put("Description", "Example SmartHome application");
		SH.put("Description", Configuration.APPLICATION_DESCRIPTION);
		SH.put("URI", new Reference(getReference(), "..")
				.getTargetRef().toString());
		SH.put("Status", this.getStatus());
		//SH.put("Method", metodo);
		return new JsonRepresentation(SH);
	}
```

5.7.4. commands()

El método público *commands*, presente en la interfaz *RootResource*, sobre escribe la operación OPTIONS del estándar HTTP para los recursos del tipo ROOTRESOURCE. La respuesta de esta consulta devuelve el conjunto de operaciones HTTP soportadas por el recurso. Estas operaciones son GET y OPTIONS.

```
	@Override
	public void commands(){
		Set<Method> meths = new HashSet<Method>();
		meths.add(Method.GET);
		meths.add(Method.OPTIONS);
		this.getResponse().setAllowedMethods(meths);
	}
}
```

5.8. La clase Interfaz FunctionalityResource

Como el resto de clases del tipo interfaz, la clase FunctionalityResource se ubica en el *package* smarthomes.rest.api.common. Esta interfaz es implementada por la clase *FunctionalityServerResource*.

La clase *FunctionalityServerResource* implementa la representación de las funcionalidades presentes en los dispositivos domóticos. Este tipo de recurso implementa los métodos GET, PUT y OPTIONS de la interfaz uniforme del protocolo HTTP, tal y como se señala en el punto 4.3. La interfaz uniforme y sus representaciones

Los métodos de la interfaz están pensados para sobre escribir los métodos HTTP y darles la funcionalidad adaptada a nuestros recursos. Estos son los *packages* necesarios para su compilación:

```
package smarthomes.rest.api.common;

import org.json.JSONException;
import org.restlet.ext.json.JsonRepresentation;
import org.restlet.representation.Representation;
import org.restlet.resource.Get;
import org.restlet.resource.Options;
   import org.restlet.resource.Put;
```

La implementación de la clase es la siguiente:

```
public interface FunctionalityResource {

       @Get ("json")
       public Representation represent() throws JSONException;

       @Options ("wadl|html")
       public void commands();

       @Put ("json")
        public void operation(JsonRepresentation action) throws
                                                        JSONException;
   }
```

5.9. La clase FunctionalityServerResource

Esta clase se ubica en el *package* *smarthomes.rest.api.server* e implementa la interfaz *FunctionalityResource*. La función de esta clase es la de atender las peticiones dirigidas a los recursos del tipo DeviceFunctionality. Estos son los *packages* necesarios para la compilación de los métodos implementados en esta clase:

```
package smarthomes.rest.api.server;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import
org.pros.upv.es.smarthomes.Devices.Functionality.interfaces.IDeviceFunctionality;
import org.pros.upv.es.smarthomes.ECA.constants.ActionConstants;
import org.pros.upv.es.smarthomes.Identifier.components.Identifier;
import org.pros.upv.es.smarthomes.Identifier.interfaces.IIdentifier;
import org.pros.upv.es.smarthomes.Resources.components.SmartHomeResource;
import org.restlet.data.Method;
import org.restlet.data.Reference;
```

```
import org.restlet.data.Status;
import org.restlet.engine.header.Header;
import org.restlet.ext.json.JsonRepresentation;
import org.restlet.representation.Representation;
import org.restlet.resource.ResourceException;
import org.restlet.resource.ServerResource;
import org.restlet.util.Series;
import smarthomes.rest.api.common.FunctionalityResource;
```

La declaración de la clase y sus parámetros es esta:

```
public class FunctionalityServerResource extends ServerResource implements
FunctionalityResource{

	private String funcId;
	private IDeviceFunctionality df = null;
```

A continuación se detallan sus clases más importantes.

5.9.1. setCustomHttpResponseHeader

El siguiente método es utilizado para resolver el problema del *Cross-Origin* que se detalla en el punto 6.7. El problema de Origen Cruzado de JavaScript (CORS)

```
@SuppressWarnings({ "unchecked" })
	public void setCustomHttpResponseHeader(String header, String value) {
	Series<Header> series =(Series<Header>)
this.getResponseAttributes().get("org.restlet.http.headers");
		Series<Header> responseHeaders = series;
	if (responseHeaders == null) {
	    responseHeaders = new Series<Header>(Header.class);
	    this.getResponseAttributes().put("org.restlet.http.headers",
responseHeaders);
	}
	    responseHeaders.add(new Header(header, value));
    }
```

5.9.2. doInit()

El método *doInit()* de la clase *ServerResource* de RESTlet de la cual extiende la clase *FunctionalityServerResource* se emplea para ejecutar otros métodos al iniciar un recurso de la clase *DeviceFunctionality*. Se sobre escribe para capturar el identificador de la funcionalidad y para añadir cabeceras HTTP a la respuesta a la petición, con el fin de solucionar el problema *CrossOrigin de JavaScript*, el cual se detalla en el punto 6.7. El problema de Origen Cruzado de JavaScript (CORS)

```
@Override
protected void doInit() throws ResourceException {
        this.funcId = getAttribute("funcId").toString();
        setCustomHttpResponseHeader("Access-Control-Allow-Origin", "*");
        setCustomHttpResponseHeader("Access-Control-Allow-Methods",
                                    "GET, PUT, OPTIONS");
        setCustomHttpResponseHeader("Access-Control-Allow-Headers", "origin,
                                    content-type, accept");
}
```

5.9.3. searchFunc()

Este método realiza una búsqueda del recurso identificado por el parámetro global *funcId* en el contexto de la SmartHome, utilizando las utilidades de búsqueda de la aplicación que representa al sistema de la vivienda inteligente, y lo almacena en el parámetro global *df*. Una vez almacenado ya podremos realizar operaciones sobre el recurso *DeviceFunctionality*. Su código fuente es este:

```
protected void searchFunc() {
        df = (IDeviceFunctionality) ((SmartHomeApplication)
             this.getApplication()).getSearchTools().
                     search(IDeviceFunctionality.class,
                     new Identifier(funcId));
}
```

5.9.4. TagsJSON()

El método TagsJSON devuelve el listado de etiquetas (*tags*) de la DeviceFunctionality en formato de JSONArray. Por cada *tag* su entrada en el *array* incluye:

- **Id:** Identificador de la etiqueta (*tag*)
- **Link:** URI relativa

Este JSON será integrado en la representación de la DeviceFunctionality obtenido por el método represents, encargado de procesar las peticiones HTTP GET.

```
private JSONArray TagsJSON() throws JSONException{
        List<IIdentifier> tags_list = df.getTags(); //Lista de tags
        JSONArray tags = new JSONArray();
        JSONObject FunB_tags = new JSONObject();
        FunB_tags.put("Id" , tags_list.get(0));
        FunB_tags.put("Link", "/tag/"+tags_list.get(0));
        //falta el name
        tags.put(FunB_tags);
        return tags;
}
```

Las etiquetas o *tags* son un tipo de recurso que no se han llegado a implementar en la API, y a pesar de que cuentan con un Id y una URI no implementan ningún subconjunto de la interfaz uniforme HTTP, por lo que carecen de funcionalidad en este proyecto.

5.9.5. ClasifiersJSON()

El método ClasifiersJSON devuelve el listado de *classifiers* de la DeviceFunctionality en formato de JSONArray. Por cada *classifier* su entrada en el *array* incluye:

- **Id**: Identificador del *classifier*
- **Link**: URI relativa del *classifier*

Este JSON será integrado en la representación del Device obtenida por el método *represent*, encargado de procesar las peticiones HTTP GET.

```
	private JSONArray ClasifiersJSON() throws JSONException{

		JSONArray classifiers = new JSONArray();
		JSONObject FunB_classifiers = new JSONObject();
		df.getClassifiers().get(0);
		FunB_classifiers.put("Id",df.getClassifiers().get(0));
		FunB_classifiers.put("Link", "/classifier/"+df.getClassifiers().get(0));
		classifiers.put(FunB_classifiers);
		return classifiers;
	}
```

Al igual que con las etiquetas (*tags*), los *classifiers* son un tipo de recurso que no se ha llegado a implementar en la API y que por lo tanto carece de funcionalidad.

5.9.6. represent()

El método *represent()* sobre escribe la implementación del método GET del estándar HTTP para dotar de una nueva representación a los recursos del tipo DeviceFunctionality. Devuelve una representación en JSON del recurso consultado a través de una operación GET enviada a su URI.

```
	@Override
	public Representation represent() throws JSONException{
		searchFunc();
		//Lista de locations
		List <IIdentifier> locations_list = df.getLocations();
		JSONObject FunB = new JSONObject();

		if (df.getTags() != null) {
			JSONArray tags = TagsJSON();
			FunB.put("Tags", tags);
```


```
                }

                //OBTENGO LA LISTA DE CLASSIFIERS SI ES DISTINTO DE NULL
                if (df.getClassifiers() != null ) {
                        JSONArray classifiers = ClasifiersJSON();
                        FunB.put("Classifiers", classifiers);
                }

                //Construyo array de locations
                JSONArray locations = new JSONArray();
                JSONObject FunB_locations = new JSONObject();
                FunB_locations.put("id" , locations_list.get(0));
                FunB_locations.put("link", "/location/"+ locations_list.get(0));
                locations.put(FunB_locations);

                Integer lastAction = (Integer) df.getLastAction();
                if (lastAction != null) {

        FunB.put("Last_Action",SmartHomeResource.getActionNameWithCode(df.getLastAction()
));
                }
                FunB.put("Current_State",df.getCurrentState());
                FunB.put("Is_Enabled",df.isEnabled());
                FunB.put("Id",df.getIdentifier().getID());
                FunB.put("Is_Quiescent",df.isQuiescent());
                FunB.put("Type",df.getDeviceFunctionalityType());
                FunB.put("Subtype",df.getDeviceFunctionalitySubType());
                FunB.put("Name",df.getName());
                FunB.put("Locations", locations);

                //CREAMOS UN NUEVO OBJETO JSON PARA EL DEVICE AL QUE PERTENENCE LA
                FUNCTIONALITY
                JSONObject DEV = new JSONObject();
                DEV.put("Id",df.getDeviceIdentifier().getID());
                String dev_uri = "/device/" + df.getDeviceIdentifier().getID();
                DEV.put("Link",dev_uri);
                DEV.put("Name",df.getDevice().getName());
                //Añadimos el JSON DEV al JSON de la Functionality
                FunB.put("Device", DEV);

                FunB.put("Previous_State",df.getPreviousState());
                FunB.put("Is_Started",df.isStarted());

                //MOSTRAR CURRENTVALUE SI DISTINTO DE NULL
                if (df.getCurrentValue() != null) {
                        FunB.put("Current_Value", df.getCurrentValue());
                }

                //Construimos la URI completa de la funcionalidad
                String uri = new Reference(getReference(), "./" +
this.funcId).getTargetRef().toString();
                String[] link = uri.split("/");
                FunB.put("Link", "/"+link[link.length-2]+"/"+link[link.length-1]);

                //Muestra la URI del recurso
                FunB.put("URI", new Reference(getReference(), "./" +
this.funcId).getTargetRef().toString());
```

```
            return new JsonRepresentation(FunB);
        }
```

Los recursos *location* que aparecen en el JSON son, al igual que los tags y los classifiers, un tipo de recurso que no se ha llegado a implementar y que por lo tanto carece de funcionalidad. Representarían a las localizaciones de la SmartHome (comedor, pasillo, baño, etc.).

El JSON producto de la ejecución de este método es similar al expuesto en el punto 4.3.5.2. Operación GET, referente a la interfaz uniforme de los recursos del tipo DeviceFunctionality.

5.9.7. operation

El método *operation* sobre escribe la operación PUT del estándar HTTP para los recursos del tipo DeviceFunctionality. Se utiliza para mandar una orden mediante el parámetro de entrada "*operation*", que debe enviarse en formato de JSON y con esta estructura:

```
{"action":"acción_a_realizar"}
```

Las acciones que pueden realizar los recursos del tipo DeviceFunctionality son muy diversas y dependen del tipo de funcionalidad. Se describen todas ellas en el punto 4.3.5.3. Operación PUT

La operación PUT es idempotente, tal y como se señala en el punto 4.3.5.4. La idempotencia de la operación PUT.

```
@Override
public void operation(JsonRepresentation action) throws JSONException{
        searchFunc();
        JSONObject jsonPut = null;
        int result = -10;

        Status s = new Status(200);

        try{
                jsonPut = action.getJsonObject();
        }catch (JSONException E) {
                setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
        }

        try
        {
                String toDo = null;
                if (jsonPut.has("action"))
                {
                        toDo = jsonPut.getString("action");
                        System.out.println("Action: " + toDo);
```


```
                        //EJECUTA LA ACCIÓN SEÑALADA EN EL CAMPO ACTION
                        result = df.executeAction(toDo,null);

                        switch(result)
                        {
                                case ActionConstants.ExecutionResult_OK:
                                        s = new Status(200);
                                        System.out.println("OK");
                                        break;
                                case ActionConstants.ExecutionResult_NothingToDo:
                                        System.out.println("Nothing to do");
                                        s = new Status(202); //Accepted
                                        break;
                                case ActionConstants.ExecutionResult_Disabled:
                                        System.out.println("Disabled");
                                        s = new Status(503);
                                        break;
                                case ActionConstants.ExecutionResult_Failed:
                                        System.out.println("Failed");
                                        s = new Status(500);
                                        break;
                                case ActionConstants.ExecutionResult_Cancelled:
                                        System.out.println("Cancelled");
                                        s = new Status(408); //Request Time out
                                        break;
                                case
                                ActionConstants.ExecutionResult_ConditionNotSatisfied:
                                        System.out.println("Solicitud incorrecta");
                                        s = new Status(400);
                                        break;
                                case ActionConstants.ExecutionResult_BlockedResource:
                                        System.out.println("Bloqueado");
                                        s = new Status(423);
                                        break;
                                case ActionConstants.ExecutionResult_OperationNotFound:
                                        System.out.println("Metodo no implementado");
                                        s = new Status(501);
                                        break;
                                default: break;
                        }
                        setStatus(s);
                }
                else //El json no contiene parámetro action => Bad Request
                {
                        System.out.println("el json no tiene action");
                        s = new Status(400);
                        setStatus(s);
                }
        }
        catch (JSONException E)
        {
                System.out.println("Excepción grave al ejecutar la acción ");
        }
}
```

Una vez realizada la acción solicitada, se evalúa la respuesta del servidor y en función de dicha respuesta se modifica el estado de la petición, de acuerdo al estándar del protocolo HTTP, los cuales se detallan en el punto 5.8.10. Los posibles estados de la operación PUT.

5.9.8. commands()

El método *commands* sobre escribe la operación OPTIONS del estándar HTTP para los recursos del tipo DeviceFunctionality. La respuesta de esta consulta devuelve el conjunto de operaciones HTTP soportadas por el recurso.

Estas operaciones son GET, OPTIONS y PUT.

```
        @Override
        public void commands(){
                Set<Method> meths = new HashSet<Method>();
                meths.add(Method.GET);
                meths.add(Method.PUT);
                meths.add(Method.OPTIONS);
                this.getResponse().setAllowedMethods(meths);
        }
}
```

5.9.9. Traza de una operación PUT sobre un recurso DeviceFunctionality

Con el fin de ilustrar el funcionamiento de las operaciones PUT, en este apartado se documenta todo el proceso mediante el cual una petición PUT enviada a un recurso del tipo DeviceFunctionality es procesada.

El recurso elegido para realizar la petición está identificado por la URI relativa /dfs/DF-PORTA.FINCA.FORRELLAT. Se trata de una funcionalidad del tipo *bistate*, cuyas posibles operaciones se detallan en el punto 4.3.5.3. Operación PUT.

Esta funcionalidad está enlazada con el Device con URI relativa device/DEV-PORTA.FINCA, como se puede comprobar en el apartado correspondiente del JSON generado por una operación GET sobre la URI http://localhost:8222/dfs/DF-PORTA.FINCA.FORRELLAT

```
    "Device": {
       "Name": "Porta Finca",
       "Link": "/device/DEV-PORTA.FINCA",
       "Id": "DEV-PORTA.FINCA"
       }
```

En primer lugar se envía la petición PUT al servidor de la API, utilizando para ello la aplicación cliente RESTClient. En la siguiente imagen se ilustra la configuración de la petición utilizando esta aplicación, de forma similar a como se detalla en el punto 5.3.3 Operación PUT

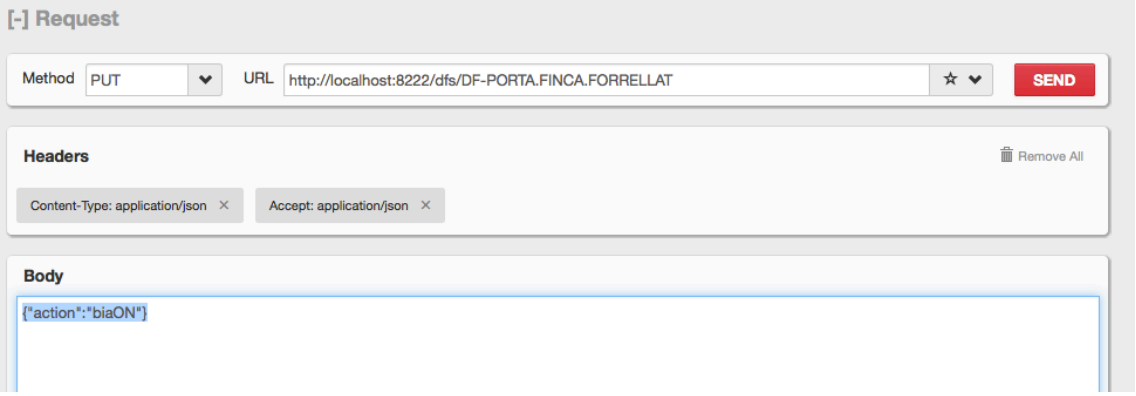

Figura 36 - Petición PUT

Al realizar la petición PUT, ésta es recibida en el servidor de la API a través de la clase *SmartHomeComponent*, que es la que ha iniciado el servicio. La petición pasa a la clase *SmartHomeApplication*, que implementa el filtro *Inbound root*. Este filtro determina que la URI a la que va dirigida la petición pertenece a un recurso del tipo *DeviceFunctionality*, y que por tanto debe ser procesada por la clase correspondiente *FunctionalityServerResource* perteneciente al *package* smarthomes.rest.api.server.

El método que implementa la operación PUT de la interfaz uniforme HTTP para la clase *FunctionalityServerResource* es *operation*. Una vez invocado este método el estado del recurso cambia a ON. Podemos consultar el nuevo estado con el JSON resultante de realizar una operación GET sobre el recurso, y será similar al expuesto en el punto 4.3.5.2. Operación GET referente a los recursos del tipo DeviceFunctionality.

5.9.10. Los posibles estados de la operación PUT

Como ya se ha señalado, el método *operation* es el encargado de servir las peticiones del tipo PUT. El resultado de la operación PUT puede depender de factores muy diversos. Una petición no siempre tendrá éxito, y su fracaso deberá ser puesta en contexto con el fin, por ejemplo, de que el cliente pueda tomar las medias oportunas en función del tipo de error devuelto. Se establecen una serie de códigos de respuesta a la operación PUT, siempre respetando la interpretación realizada del estándar HTTP que se puede consultar en [6].

De este modo, los posibles resultados de una operación PUT y sus correspondientes códigos de estado son:

Resultado de la operación	Código HTTP del estado	Mensaje del estado HTTP
La operación termina con éxito	200	OK
La operación no hace nada	202	Accepted
Recurso desactivado	503	Service Unavailable
La operación fracasa	500	Failed
Se cancela la operación	408	Canceled
Solicitud incorrecta	400	Bad Request
Recurso bloqueado	423	Locked
Método no implementado	501	Not Implemented

Tabla 6 - Códigos de estado HTTP

Tomando de nuevo como ejemplo el recurso identificado por la URI http://localhost:8222/dfs/DF-PORTA.FINCA.FORRELLAT podemos enviar una petición incorrecta al servidor, enviando en el cuerpo de la petición PUT un JSON como este:

```
{"action":"operación_que_no_existe"}
```

Como es obvio, al intentar procesar el JSON se determina que no se encuentra implementada la *operación_que_no_existe* y se devuelve un código de estado 501. Utilizando RESTClient como cliente con el que lanzar la petición PUT lo que veríamos en la respuesta es algo parecido a esto:

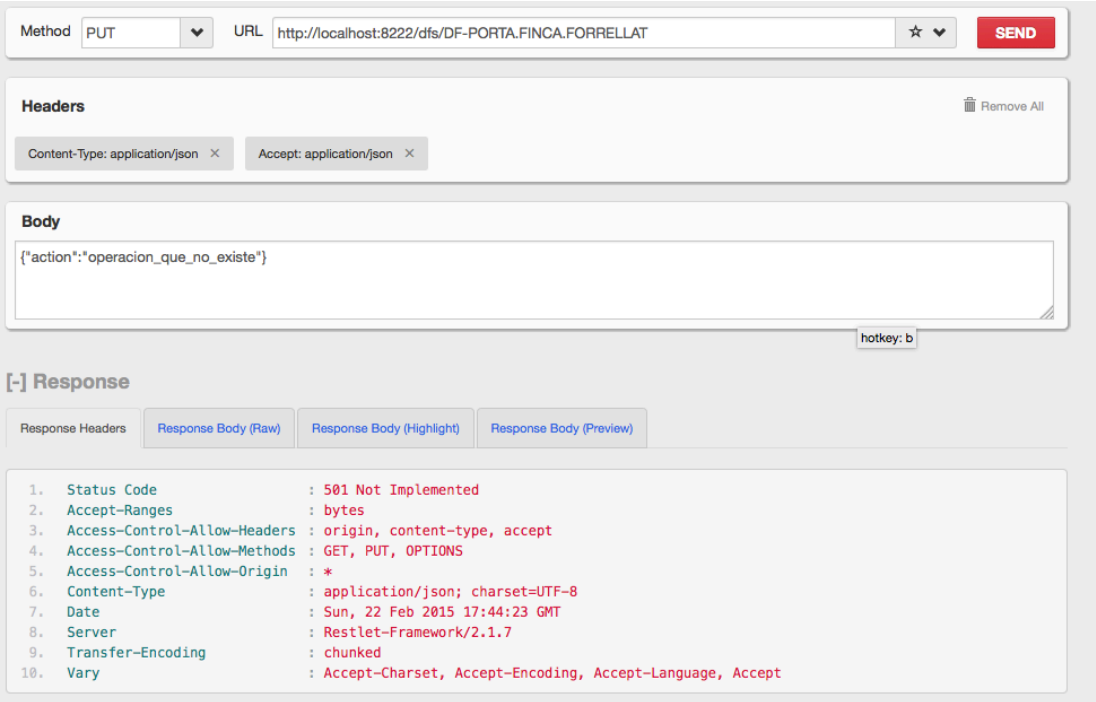

Figura 37 - Estado devuelto por un petición PUT incorrecta

Como se puede apreciar en la imagen, el estado devuelto por el servidor efectivamente corresponde a un *501 Not Implemented*.

6. Prototipo de cliente web

Como punto final del presente trabajo se desarrolla un sencillo prototipo de cliente web con el fin de ejemplificar cómo se realizaría la utilización de la API RESTlet por parte de aplicaciones externas.

Se ha elegido la tecnología web por ser la más próxima al protocolo HTTP utilizado en la interfaz uniforme empleada en el desarrollo de la API RESTlet, y porque su desarrollo no entraña excesiva dificultad a la hora de crear aplicaciones cliente simples, pero se podría haber desarrollado en cualquier lenguaje de programación que permitiese emplear métodos de la interfaz HTTP, como un *servlet* de Java.

El cliente web está basado en páginas en formato HTML combinadas con funciones en JavaScript, las cuales dotan a la aplicación de una interfaz dinámica simple y de una hoja de estilo *css* que controla su aspecto.

No se pretende conseguir el desarrollo de una interfaz completa, ya que se aleja del objetivo principal de este trabajo. La interfaz web desarrollada tan solo sirve como un ejemplo sencillo del funcionamiento de la API, y de cómo puede ser utilizada por aplicaciones clientes externas para la consulta y control de los recursos presentes en la SmartHome. Al ser un prototipo no se implementan todas las funcionalidades que ofrece la interfaz de la API.

6.1. Tecnologías empleadas

Para el desarrollo de la aplicación ha sido necesario preparar un entorno de programación separado de entorno de desarrollo de la API. Esto además ayuda a diferenciar que son dos aplicaciones totalmente independientes.

Como Framework de programación se ha optado por *Eclipse Luna for Web Developers Edition* descrito en el punto 2.2.1. Eclipse Web Developer.

Además, es necesario contar con un servidor web que aloje las páginas que conforman al cliente, y además este servidor debe contar con la posibilidad de servir páginas dinámicas basadas en JavaScript.

Se ha optado por emplear la solución integral XAMPP que permite instalar de forma sencilla un servidor web basado en Apache, el lenguaje de programación PHP5, un servidor de bases de datos MySQL y un servidor FTP. Para el desarrollo del cliente web solo ha sido empleado el servidor web Apache.

Por último, es necesario contar con ciertas herramientas en el navegador web con las que poder visualizar mensajes de error y realizar tareas de debug, con el fin de facilitar el desarrollo del código necesario. Es por ello que se ha empleado como navegador web *Firefox Developer Edition*, el cual cuenta con la posibilidad de visualizar los mensajes de consola, así como otras utilidades como visualización de mensajes de error, visualizar el código de las páginas HTML y de JavaScript etc.

La configuración del entorno de programación y puesta en marcha del servidor web Apache se detalla en 9.2. Anexo 2 - Configuración del entorno de desarrollo del cliente web

6.2. La página index.html

Como toda aplicación web, el acceso al prototipo de cliente web se realiza desde su página de inicio index.html, que se realiza de forma automática al acceder a la URI http://localhost/cliente_web de la máquina donde se encuentra el servidor web Apache que lo aloja. Se trata de una interfaz simple y sencilla, propia de un prototipo de cliente web, no de una aplicación completa. La interfaz se puede dividir en varias secciones:

1. Información sobre la SmartHome
2. Formulario de selección de Devices / DeviceFunctionalities / Actions
3. Información sobre el Device seleccionado
4. Información sobre la DeviceFunctionality seleccionada
5. Información sobre la *action* ejecutada

En esta imagen se marcan cada una de estas secciones diferenciadas por colores:

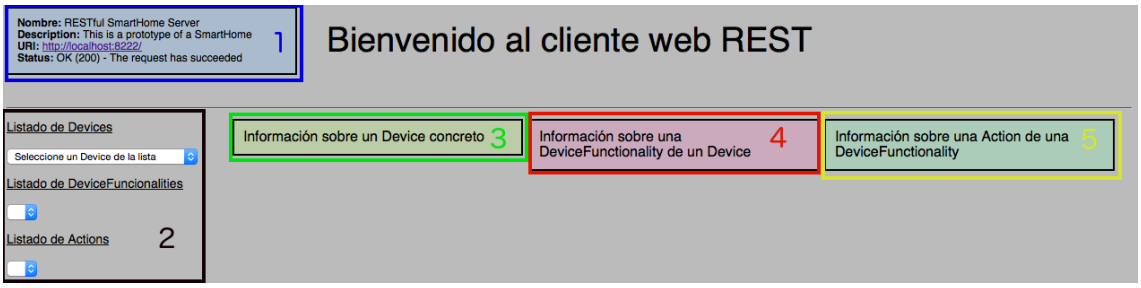

Figura 38 - Interfaz del cliente web

A continuación se describe el código de las diferentes secciones en las que se divide la página de inicio.

6.2.1. <Head>

En la cabecera de la página web se establecen algunas características, como el título de la página o el fichero de estilos que controla el aspecto de la misma. A continuación se detallan las líneas de código más importantes.

```
<link rel="stylesheet" type="text/css" href="estilo.css">
```

Esta línea contenida en el head de la página establece cual va a ser el documento de estilo *css* que va a controlar el aspecto de la web. Este documento se detalla en el punto 6.6. El archivo estilo.css

```
<script lang="javascript" type="text/javascript"
        src="ajax.js">
</script>
```

Estas otras líneas hacen referencia al documento en formato JavaScript que contiene los métodos y funciones necesarios para dotar de interactividad dinámica a la interfaz. Este documento *ajax.js* se detalla en el punto 6.5. El archivo ajax.js

6.2.2. <Body>

El resto de código fuente perteneciente al *body* (cuerpo) del documento que se emplea para mostrar la información en el navegador y se puede dividir en varias secciones, dependiendo del tipo de recurso del cual muestran información. Se detallan a continuación cada una de las secciones en las que se divide la página:

6.2.2.1 infoSmarthome

En esta etiqueta div se muestra información relativa a la SmartHome. Los datos mostrados son extraídos del JSON generado por la operación GET sobre la URI http://localhost:8222 que se detalla en el punto 4.3.1.2. Operación GET. Las funciones JavaScript que obtienen esta información se inician al cargar la página y se detallan en el punto 6.5. El archivo ajax.js. Su código fuente es este:

```
<div id="infoSmarthome">
Informaci&oacute; relativa al servidor de la SmartHome.
</div>
```

Y los datos se visualizan como se muestran en la siguiente imagen:

Figura 39 - Información mostrada relativa al recurso SmartHome

6.2.2.2 DeviceList

Se trata de un listado desplegable que contiene los nombre de todos los recursos de tipo Device registrados. El listado se obtiene mediante la función JavaScript *getDevices()* que se lanza al cargar la página de inicio y se detalla en el punto 6.5. El archivo ajax.js. El código fuente donde se declara este listado es el siguiente:

```
<p class="subrayado">Listado de Devices</p>
<form id="form1" name="form1">
<select name="DevicesList" id ="ListaDevices" onchange="ObtenerValorLista();">
</select>
</form>
<div id="device">
Informaci&oacute;n sobre un Device concreto
</div>
```

Se puede apreciar que este listado desplegable además cuenta con una función JavaScript asociada al evento *onchage*: *ObtenerValorLista(),* que se desencadena cuando cambia el valor seleccionado en la lista. Este método se detalla también en el 6.5. El archivo ajax.js. En la siguiente imagen se muestra un fragmento del listado generado:

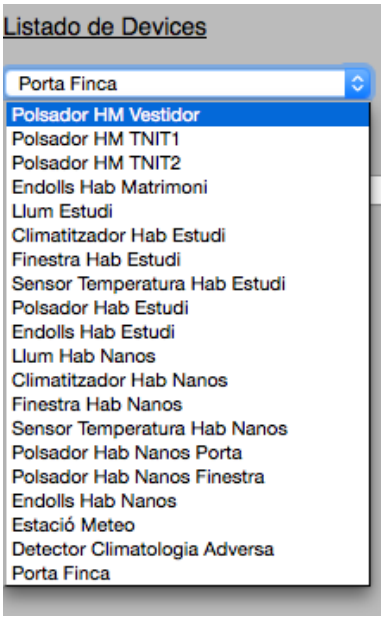

Figura 40 - Listado de recursos Device

6.2.2.3 FuncList

Es un listado desplegable que contiene los nombres de cada uno de los recursos del tipo DeviceFunctionality contenidos en el recurso Device que se ha seleccionado en el listado anterior y se genera automáticamente al seleccionar un valor en el listado *DeviceList*. El código fuente es este:

```
<p class="subrayado">Listado de Funcionalities</p>
<form id="funcs" name="form2">
<select name="FuncList" id ="ListaFuncs" onchange="ObtenerValorListaFunc();">
</select>
</form>
<div id="functionality">
Informaci&oacute;n sobre una DeviceFunctionality de un Device
</div>
```

Se puede apreciar que este formulario además cuenta con una función JavaScript asociada al evento onchage, *ObtenerValorListaFunc()*, que se desencadena al seleccionar un valor en el listado. Este método se detalla en el punto 6.5. El archivo ajax.js. En la siguiente imagen se muestra un ejemplo de este listado:

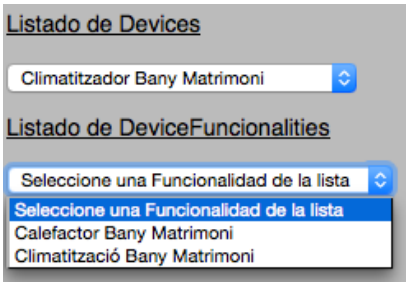

Figura 41 - Listado de recursos DeviceFunctionality para un Device concreto

6.2.2.4 ActList

Es un listado desplegable que contiene las posibles acciones (*actions*) a realizar sobre el recurso DeviceFunctionality seleccionado en el listado *FuncList*. El código fuente es el siguiente:

```
<p class="subrayado"></p>
<form id="actions" name="form3">
<select name="ActList" id ="ListaAct" onchange="ObtenerValorListaAct();">
</select>
</form>
<div id="action">
Informaci&oacute;n sobre una Action de una DeviceFunctionality
</div>
```

Como en los casos anteriores, este listado cuenta con una función JavaScript asociada al evento onchage: *ObtenerValorListaAct()*, que se desencadena cuando cambia el valor seleccionado en el listado. Este método se detalla en el punto 6.5. El archivo ajax.js. En la imagen siguiente se muestra un ejemplo de este listado.

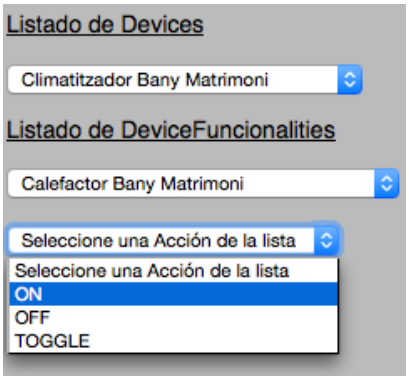

Figura 42 - Acciones disponibles para la DeviceFunctionality seleccionada

6.2.3. Código fuente completo

A continuación se expone el código fuente completo de la página de inicio:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Cliente Web SmartHome REST</title>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<link rel="stylesheet" type="text/css" href="estilo.css">
<script lang="javascript" type="text/javascript"
        src="ajax.js">
</script>
</head>
<body style="background-color:#bbbbbb;" onload="getSmartHomeInfo();getDevices();">
<h1>Bienvenido al cliente web REST</h1>
<div id="infoSmarthome">
Informaci&oacute; relativa al servidor de la SmartHome.
</div>
<br />
<hr />

<p class="subrayado">Listado de Devices</p>
<form id="form1" name="form1">
<select name="DevicesList" id ="ListaDevices" onchange="ObtenerValorLista();">
</select>
</form>
<div id="device">
Informaci&oacute;n sobre un Device concreto
</div>

<p class="subrayado">Listado de DeviceFuncionalities</p>
<form id="funcs" name="form2">
<select name="FuncList" id ="ListaFuncs" onchange="ObtenerValorListaFunc();">
</select>
</form>
<div id="functionality">
Informaci&oacute;n sobre una DeviceFunctionality de un Device
</div>
```

```
<p class="subrayado"></p>
<form id="actions" name="form3">
<select name="ActList" id ="ListaAct" onchange="ObtenerValorListaAct();">
</select>
</form>
<div id="action">
Informaci&oacute;n sobre una Action de una DeviceFunctionality
</div>
</body>
</html>
```

6.3. El archivo ajax.js

En este fichero se implementan las funciones JavaScript que otorgan de funcionalidad dinámica a la interfaz del cliente web. El código JavaScript se basa en ejemplos recopilados de [1] y de [11]. La generación de los ficheros JSON desde JavaScript se ha basado en ejemplos expuestos en [7]. A continuación se detallan las más importantes:

6.3.1. getXMLHTTPRequest()

Este método crea una nueva petición (*request*) HTTP que será empleada para enviar comandos al servidor. Se emplea para inicializar variables con instancias de esta función para posteriormente ser empleadas en las funciones que lanzan las peticiones al servidor. El código fuente es el siguiente:

```
function getXMLHTTPRequest() {
	var request = false;
	try
	{
		request = new XMLHttpRequest();
	}
	catch (error1)
	{
		request = false;
	}
	return request;
}
```

6.3.2. getSmartHomeInfo()

Esta función realiza una petición GET al servidor de la SmartHome y se invoca al cargar la página *index.html* del cliente web. Cuando el servidor manda la respuesta, esta es tratada por la función *useHttpResponse()*, la cual obtiene el JSON generado, en

el caso de que la consulta finalice con éxito. A continuación el JSON es tratado para ser mostrado adecuadamente en la interfaz de la siguiente manera:

Figura 43 - Información mostrada de la SmartHome

La función utiliza dos variables globales: *http* que se inicializa con una instancia de la función *getXMLHTTPRequest()* y *ServerURI* que se inicializa con la URI del servidor de la SmartHome. Este es el código fuente:

```
var http = getXMLHTTPRequest();
var ServerURI =  "http://localhost:8222";

function getSmartHomeInfo() {
	var myurl = ServerURI;
	http.open("GET", myurl, true);
	http.onreadystatechange = useHttpResponse;
	http.send(null);
}

function useHttpResponse() {
	if (http.readyState == 4) {
		if (http.status == 200) {
			var SmartHomeInfo = http.responseText;
			var JSONcomponents = JSON.parse(SmartHomeInfo);
			document.getElementById('infoSmarthome').innerHTML =
			"<strong>Nombre: </strong>" + JSONcomponents.Name
			+ "<br />" + "<strong>Description: </strong>"
			+ JSONcomponents.Description + "<br />"
			+ "<strong>URI: </strong> <a href=" + JSONcomponents.URI + ">"
			+ JSONcomponents.URI + "</a><br />"
			+ "<strong>Status: </strong>" + JSONcomponents.Status + "<br />";
		}
	} else {
		document.getElementById('infoSmarthome').innerHTML = "Error al obtener
                                                        los datos de la SmartHome";
	} }
```

6.3.3. getDevices()

Esta función realiza una petición GET a la URI http://localhost:8222/devices, y se invoca al cargar la página *index.html* del cliente web. La respuesta a la petición GET es tratada por el método *useHttpDevicesResponse(),* que obtiene, si la petición tiene éxito, el JSON que contiene el listado de recursos del tipo Device de la SmartHome. Este JSON es posteriormente tratado para crear a partir del mismo una lista desplegable en un formulario de la interfaz del cliente web.

Por cada entrada de la lista se muestra el nombre del Device, y el valor de cada entrada corresponde a la URI relativa (Link) del Device. De esta forma el combo es utilizado para lanzar consultas sobre un Device concreto a través de su URI.

En la siguiente imagen se muestra un fragmento de la lista generada:

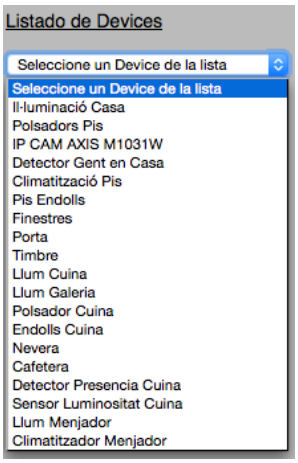

Figura 44 - Combo desplegable de Devices

El método requiere inicializar previamente una variable global *httpDevices* a una instancia del método *getXMLHTTPRequest()* para realizar la petición GET. El código fuente es el siguiente:

```
var httpDevices = getXMLHTTPRequest();

function getDevices() {
	var myurl = ServerURI + "/devices";
	httpDevices.open("GET", myurl, true);
	httpDevices.onreadystatechange = useHttpDevicesResponse;
	httpDevices.send(null);
}

function useHttpDevicesResponse() {
	if (httpDevices.readyState == 4) {
		if (httpDevices.status == 200) {
			var DevicesList = httpDevices.responseText;
			var JSONcomponents = JSON.parse(DevicesList);
			var out = '<option value="">Seleccione un Device de la
					lista</option>';
		    var i;
		    for(i = 0; i < JSONcomponents.Devices.length; i++) {
		        out += '<option value= "' + JSONcomponents.Devices[i].Link + '">'
		            + JSONcomponents.Devices[i].Name + '</option>';
		    }
		    document.getElementById('ListaDevices').innerHTML = out;
		}
	} else {
		document.getElementById('ListaDevices').innerHTML = "Error al obtener el
                                                            listado de Devices";
	}
```

```
}
```

6.3.4. getDevice(link)

Esta función realiza una petición GET a la URI del recurso Device seleccionado en el combo desplegable generado previamente por la función *getDevices()*. El parámetro de entrada *link* es la URI relativa del recurso Device, obtenida por la función *ObtenerValorLista()* que se invoca al seleccionar un elemento de la lista desplegable de Devices, y que es la que realiza la llamada a *getDevice(link)*.

La respuesta a la petición GET es tratada por el método *useHttpDeviceResponse(),* que obtiene, si la petición tiene éxito, el JSON que contiene la información perteneciente al recurso Device consultado. Este JSON es tratado para mostrar en la interfaz del cliente web la información más relevante. Concretamente se muestra los siguiente:

1. **Nombre del Device**
2. **Iniciado:** si está iniciado o no (true/false)
3. **Activado:** muestra si el estado del Device es activo o inactivo (true/false)
4. **ID:** Identificador del Device

En la siguiente imagen se muestra la información mostrada para el Device con URI http://localhost:8222/devices/DEV-CL

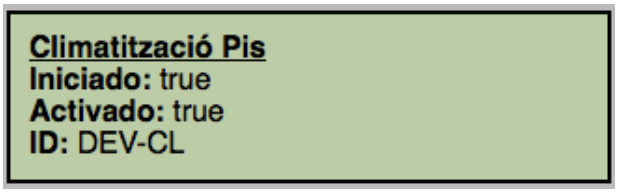

Figura 45 - Datos mostrados de un Device

Al tratarse de un prototipo, y no de un cliente web completo, se ha optado por mostrar la mínima información necesaria y fiel a la aportada al JSON recibido. Una interfaz más completa podría mostrar mucha más información relativa al Device a partir del JSON generado.

El parámetro *link* con el que se realiza la llamada contiene únicamente la URI relativa del Device. En este caso este *link* contiene el valor */devices/DEV-CL*. **Para poder realizar la consulta al Device es necesario obtener su URI canónica o completa**, la cual se obtiene concatenando la URI relativa del Device con la URI canónica del servidor de la SmartHome, que es http://localhost:8222 y que está almacenada en la variable global *ServerURI*. De este modo se obtiene la URI canónica http://localhost:8222/*devices/DEV-CL*. Este método para obtener las URIS canónicas

de un recurso a través de sus URIS relativas o *links* y se utiliza en otras funciones JavaScript del cliente web.

Como se comenta en el punto 4.3.3.2. Operación GET, **en el JSON obtenido se muestra también información perteneciente a todos los recursos del tipo DeviceFunctionality relacionados con el Device.** A partir del listado de recursos DeviceFunctionality relacionados se crea una lista desplegable en un formulario de la interfaz web de la aplicación.

Por cada entrada de la lista se muestra el nombre de la DeviceFunctionality, y el valor de cada entrada corresponde a su URI relativa o *link*. De esta forma el listado es utilizado para lanzar consultas sobre un recurso concreto del tipo DeviceFunctionality, relacionado a su vez con el Device actualmente seleccionado, a través de su URI relativa. De esta forma se aplica el principio de interconexión de recursos que ofrece REST.

La función *getDevice()* requiere inicializar previamente una variable global *httpDevice* a una instancia de la función *getXMLHTTPRequest()* para realizar la petición GET. El código fuente de las tres funciones descritas es el siguiente:

```
function ObtenerValorLista() {
	var lista = document.getElementById("ListaDevices");
	var valorSeleccionado = lista.options[lista.selectedIndex].value;
	getDevice(valorSeleccionado);
}

var httpDevice = getXMLHTTPRequest();

function getDevice(link) {
	var myurl = ServerURI + link;
	httpDevice.open("GET", myurl, true);
	httpDevice.onreadystatechange = useHttpDeviceResponse;
	httpDevice.send(null);
}

function useHttpDeviceResponse() {
	if (httpDevice.readyState == 4) {
		if (httpDevice.status == 200) {
		    var DeviceData = httpDevice.responseText;
		    var JSONcomponents = JSON.parse(DeviceData);
		    out = "";
		    out += "<strong><u>" + JSONcomponents.Name + "</u></strong><br />";
		    out += "<strong>Iniciado: </strong>" + JSONcomponents.Is_Started
		          + "<br />";
		    out += "<strong>Activado: </strong>" + JSONcomponents.Is_Enabled
		          + "<br />";
		    out += "<strong>ID: </strong>" + JSONcomponents.Id + "<br />";
		    document.getElementById('device').innerHTML = out;

		    var i;
		    var outForm = '<option value="">Seleccione una Funcionalidad de la
		                  lista</option>';
```

```
                    for(i = 0; i < JSONcomponents.DeviceFunctionalities.length; i++) {
                        var funcURI = JSONcomponents.DeviceFunctionalities[i].Link;
                        outForm += '<option value = "'
                                + JSONcomponents.DeviceFunctionalities[i].Link + '">'
                                + JSONcomponents.DeviceFunctionalities[i].Name
                                + '</option>';
                    }
                    document.getElementById('ListaFuncs').innerHTML = outForm
                }
        }
        else
        {
                document.getElementById('device').innerHTML = "Error al obtener los datos
                                                              del Device";
        }
}
```

6.3.5. getFunc(link)

Esta función realiza una petición GET a la URI del recurso DeviceFunctionality seleccionado en el combo desplegable generado previamente por la función *getDevice()*. El parámetro de entrada link necesario corresponde a la URI relativa (*Link*) del recurso DeviceFunctionality consultado, y que es obtenido automáticamente por la función *ObtenerValorListaFunc()* que realiza la llamada.

La respuesta a la petición GET es tratada por el método *useHttpFuncResponse(),* que obtiene, si la petición tiene éxito, el JSON que contiene la información relativa al recurso DeviceFunctionality consultado. Este JSON es tratado para mostrar en la interfaz del cliente web la información más relevante. Concretamente se muestra los siguiente:

1. **Nombre del recurso DeviceFunctionality**
2. **Tipo:** bistate, togglebistate, numeric value, etc
3. **Iniciado:** si está iniciado o no (true/false)
4. **Activado:** muestra si el estado es activo o inactivo (true/false)
5. **ID:** Identificador del recurso DeviceFunctionality
6. **Estado Actual:** estado al realizar la última acción
7. **Estado Anterior:** estado antes de realizar la última acción.
8. **Última acción:** última acción realizada sobre el recurso

En la siguiente imagen se puede observar la información mostrada para el recurso con URI http://localhost:8222/dfs/DF-CL.ACs

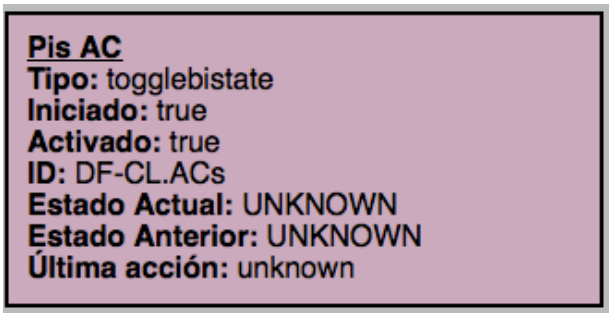

Figura 46 - Información mostrada de un recurso DeviceFunctionality

De nuevo se recalca que al tratarse de un prototipo, y no de un cliente web completo, se ha optado por mostrar la mínima información necesaria. Una interfaz más completa podría mostrar mucha más información relativa al recurso DeviceFunctionality a partir del JSON generado.

Una vez determinando el tipo de DeviceFunctionality concreto, se puede determinar el tipo de operaciones que podemos realizar con él, tal y como se describen en el punto 4.3.5.3. Operación PUT. Este listado es obtenido por la función *FuncActionList(type)*. A partir del listado de acciones (*Actions*) disponibles para el recurso DeviceFunctionality se crea una lista desplegable en un formulario de la interfaz web de la aplicación cliente.

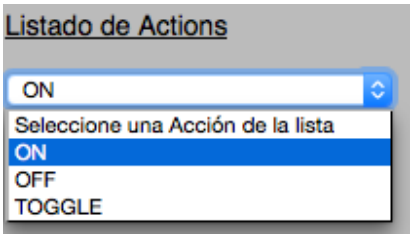

Figura 47 - Listado de acciones

Como se trata de un prototipo, la función solamente contempla los casos en los que el tipo de DeviceFunctionality sea *bistate*, *togglebistate* o *dimmer*. De forma análoga se podrían desarrollar el resto de casos para que se incluyan las acciones de todos los tipos posibles de recursos DeviceFunctionality.

Por cada entrada de la lista se muestra el nombre de la acción a realizar, y el valor de cada entrada corresponde a su identificador. De esta forma el listado es utilizado para lanzar una acción en concreto al recurso DeviceFunctionality seleccionado.

La función *getFunc(link)* requiere inicializar previamente una variable global *httpFunc* a una instancia de la función *getXMLHTTPRequest()* para realizar la petición GET. La URI relativa del recurso DeviceFunctionality es almacenada en la variable global FuncLink, la cual será empleada posteriormente para ejecutar la acción seleccionada. El código fuente de las funciones descritas es el siguiente:

```
var FuncLink="";

function ObtenerValorListaFunc() {
	var lista = document.getElementById("ListaFuncs");
	var valorSeleccionado = lista.options[lista.selectedIndex].value;
	FuncLink = valorSeleccionado;
	getFunc(valorSeleccionado);
}

var httpFunc = getXMLHTTPRequest();

function getFunc(link) {
	var myurl = ServerURI + link;
	httpFunc.open("GET", myurl, true);
	httpFunc.onreadystatechange = useHttpFuncResponse;
	httpFunc.send(null);
}

function useHttpFuncResponse() {
	if (httpFunc.readyState == 4) {
		if (httpFunc.status == 200) {
		    var FuncData = httpFunc.responseText;
		    var JSONcomponents = JSON.parse(FuncData);
		    var out = "";
		    out += "<strong><u>" + JSONcomponents.Name + "</u></strong><br />";
		    out += "<strong>Tipo: </strong>" + JSONcomponents.Type + "<br />";
		    out += "<strong>Iniciado: </strong>" + JSONcomponents.Is_Started
		        + "<br />";
		    out += "<strong>Activado: </strong>" + JSONcomponents.Is_Enabled
		        + "<br />";
		    out += "<strong>ID: </strong>" + JSONcomponents.Id + "<br />";
		    out += "<strong>Estado Actual: </strong>"
		        + JSONcomponents.Current_State + "<br />";
		    out += "<strong>Estado Anterior: </strong>"
		        + JSONcomponents.Previous_State + "<br />";
		    out += "<strong>&Uacute;ltima acci&oacute;n: </strong>"
		        + JSONcomponents.Last_Action + "<br />";
		    document.getElementById('functionality').innerHTML = out;
		    FuncActionList(JSONcomponents.Type);
		}
	}
	else
	{
	      document.getElementById('functionality').innerHTML = "Error al obtener los
	                                                           datos de la DeviceFunctionality";
	}
}

function FuncActionList(type) {
	var outForm = '<option value="">Seleccione una Acci&oacute;n de la
lista</option>';
	if (type == "togglebistate")
	{
		outForm += '<option value = "biaON">ON</option>';
		outForm += '<option value = "biaOFF">OFF</option>';
		outForm += '<option value = "biaTOGGLE">TOGGLE</option>';
	}
	if (type == "bistate")
	{
```


```
                outForm += '<option value = "biaON">ON</option>';
                outForm += '<option value = "biaOFF">OFF</option>';
        }
        if (type == "dimmer")
        {
                outForm += '<option value = "biaON">ON</option>';
                outForm += '<option value = "biaOFF">OFF</option>';
                outForm += '<option value = "biaPULSEON">PULSE ON</option>';
                outForm += '<option value = "biaPULSEOFF">PULSE OFF</option>';
        }

        document.getElementById('ListaAct').innerHTML = outForm;
}
```

6.3.6. putFuncAction(action)

Esta función realiza una petición PUT al recurso DeviceFunctionality actualmente seleccionado. Como se ha detallado en el punto 4.3.5.3. Operación PUT, el método PUT se emplea para ordenar acciones a realizar a los recursos DeviceFunctionality. El parámetro *action,* que representa la acción a realizar, corresponde con el valor de la opción seleccionada en el listado de acciones disponibles, y que ha generado previamente la función *getFunc(link)*. Este valor es obtenido por la función *ObtenerValorListaAct()*, que se invoca al seleccionar un elemento del listado de acciones y que es la que realiza la llamada a *putFuncAction(action)*.

Al ejecutar una acción se mostrará en la interfaz web el JSON generado para ser enviado en el *body* de la petición PUT, y si ha tenido o no éxito su ejecución.

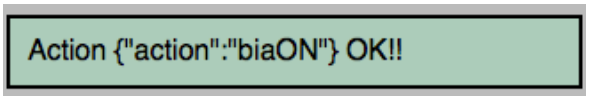

Figura 48 - Información mostrada de una acción ejecutada correctamente

El código fuente de estas funciones es el siguiente:

```
function ObtenerValorListaAct() {
        var lista = document.getElementById("ListaAct");
        var valorSeleccionado = lista.options[lista.selectedIndex].value;
        var JSONaction='{"action":"' + valorSeleccionado + '"}';
        putFuncAction(JSONaction);
}

var ActRequest = getXMLHTTPRequest();

function putFuncAction(action) {
        var myurl = ServerURI + FuncLink;
        ActRequest.open("PUT", myurl, true);
        ActRequest.setRequestHeader('Accept','application/json');
        ActRequest.setRequestHeader('Content-Type','application/json');
        ActRequest.onreadystatechange = useHttpActResponse(action);
        ActRequest.send(action);
```

```
}

function useHttpActResponse(action){
	if (ActRequest.readyState == 1) {
		if (ActRequest.status == 0) {
			var out = "Action " + action + " OK!!";
			document.getElementById('action').innerHTML = out;
			getFunc(FuncLink);
		}
		else
			document.getElementById('action').innerHTML = "¡¡FALLO AL REALIZAR
			                                               LA ACTION!!";
	}
}
```

En el código fuente de la función *putFuncAction(action)* se añaden dos instrucciones necesarias para agregar a la petición PUT las cabeceras HTTP necesarias para tratar el contenido del *body* en formato JSON. Estas cabeceras son las ya comentadas en el punto 4.3.3.3. Operación PUT referente a los recursos Device y en el 4.3.5.3. Operación PUT referente a los recursos del tipo DeviceFunctionality.

6.3.7. Visualizar el cambio de estado después de un PUT

La petición PUT no devuelve un *payload* en su respuesta, para poder visualizar el efecto ocasionado por la ejecución de la acción sobre el recurso DeviceFunctionality seleccionado, será necesario efectuar una nueva operación GET sobre este recurso. De esto se encarga la llamada a *getFunc(link)*, la cual realiza una operación GET sobre la URI de la DeviceFunctionality seleccionada. Se realiza una llamada a getFunc(*FuncLink*) en el último paso de la función *useHttpActResponse(action)* que maneja la respuesta a la petición PUT lanzada desde *putFuncAction(action)*. La URI relativa de esta DeviceFunctionality se encuentra almacenada en la variable global *FuncLink*.

De este modo, al finalizar correctamente la operación PUT se realiza el GET sobre la DeviceFunctionality, actualizando en la interfaz web el estado de la misma y permitiendo apreciar el cambio realizado por la acción.

6.4. El archivo estilo.css

Este fichero controla el aspecto de la interfaz web. Se utiliza para cambiar el color del fondo de la página web, la fuente del texto, el color del fondo y el tamaño de las etiquetas div en las cuales se muestra la información que se consulta. Al tratarse de un fichero de configuración de estilo bastante simple no se profundiza en el

funcionamiento del mismo, el cual tampoco es uno de los objetivos del presente trabajo. El código fuente del fichero es el siguiente:

```
@CHARSET "US-ASCII";

h1{
	font:38px normal verdana, helvetica, arial, sans-seriff;
	text-align: center;
}
#infoSmarthome{
	width:300px;
	background-color:#aabbcc;
	border:2px solid #000000;
	padding:10px;
	font:12px normal verdana, helvetica, arial, sans-seriff;
	position: absolute;
	top: 5px;
	text-align: left;
}
#device{
	width:300px;
	background-color:#bbccaa;
	border:2px solid #000000;
	padding:10px;
	font:16px normal verdana, helvetica, arial, sans-seriff;
	position: absolute;
	top: 130px;
	left: 260px;
	text-align: left;
}
#functionality{
	width:300px;
	background-color:#ccaabb;
	border:2px solid #000000;
	padding:10px;
	font:16px normal verdana, helvetica, arial, sans-seriff;
	position: absolute;
	top: 130px;
	left: 590px;
	text-align: left;
}
#action{
	width:300px;
	background-color:#aaccbb;
	border:2px solid #000000;
	padding:10px;
	font:16px normal verdana, helvetica, arial, sans-seriff;
	position: absolute;
	top: 130px;
	left: 920px;
	text-align: left;
}
p{
	font:14px normal verdana, helvetica, arial, sans-seriff;
	text-decoration: underline;
}
.subrayado{
	text-decoration: underline
```

```
}
```

6.5. El problema de Origen Cruzado de JavaScript (CORS)

Como se detalla en [16], los programas JavaScript presentan una seria limitación a la hora de lanzar consultas sobre servicios alojados en un dominio diferente al del propio servidor que lanza la consulta. En el caso de las consultas lanzadas desde las funciones JavaScript del cliente web la consulta parte del servidor http://localhost, mientras que el destino de la consulta se encuentra en http://localhost:8222. A pesar de que en el entorno de test ambas aplicaciones se encuentran alojadas en la misma máquina, y en el mismo servidor, el hecho de que el puerto de acceso de la API REST de la SmartHome escuche por un puerto distinto es suficiente para que se active la restricción de Origen Cruzado.

Evidentemente, sería un agujero de seguridad importante que un servidor permitiese a cualquier función JavaScript que lo solicitase ejecutar código sin comprobar el origen de la petición, y es por ello que se restringe por defecto la ejecución de JavaScript únicamente a páginas alojadas **en el mismo dominio** que la página destino de la petición.

Si lanzamos una petición con JavaScript desde http://localhost hacia http://localhost:8222 en lugar de obtener la respuesta a nuestra petición el servidor responderá con un mensaje de error similar al siguiente:

"Cross-Origin Request Blocked: The Same Origin Policy disallows Reading the remote resource at http://localhost:8222. This can be fixed by moving the resource to the same domain or enabling CORS."

En el mismo mensaje de error se nos proponen dos posibles soluciones a nuestro problema: podemos mover el recurso solicitado al mismo dominio donde está alojada la página que lanza la consulta o podemos activar CORS.

La primera opción queda descartada por el hecho de que a pesar de que nos encontramos en un entorno de pruebas debemos considerar que la situación normal en un entorno real será que nuestra aplicación cliente rara vez se encontrará alojada en el mismo dominio (ni siquiera en el mismo servidor físico) que nuestra aplicación API de la SmartHome.

Por tanto solo nos queda como alternativa la activación de CORS, que consiste en delimitar el alance del *Cross-Origin* añadiendo cabeceras HTTP adicionales en la inicialización de los recursos que atienden las peticiones, de modo que se anule esta restricción y las peticiones sean atendidas.

Es por ello que en primer lugar se modifica el código fuente de las clases java de la API REST encargadas de atender las peticiones HTTP dirigidas a los recursos, de modo que incluyan al inicializar dichos recursos las cabeceras necesarias para activar CORS. Las clases que necesitan ser modificadas son estas:

1. RootServerResource
2. DevicesServerResource
3. DeviceServerResource
4. FunctionalitiesServerResource
5. FunctionalityServerResource

A todas estas clases se les añade un nuevo método, basado en el código expuesto en [3], de ámbito privado denominado *setCustomHttpResponseHeader* cuyo cometido es añadir al listado de cabeceras HTTP por defecto de RESTlet una cabecera adicional a la que se le puede establecer su nombre y su valor. Este método será invocado en el método de inicialización del recurso *doInit()* de cada clase, añadiendo las cabeceras necesarias al recurso para la activación de CORS. Las cabeceras necesarias varían en función de si el recurso inicializado implementa o no el método PUT en su interfaz uniforme . El código fuente del método es este:

```
@SuppressWarnings({ "unchecked" })
public void setCustomHttpResponseHeader(String header, String value) {
        Series<Header> series =(Series<Header>)
        this.getResponseAttributes().get("org.restlet.http.headers");
        Series<Header> responseHeaders = series;

        if (responseHeaders == null) {
            responseHeaders = new Series<Header>(Header.class);
            this.getResponseAttributes().put("org.restlet.http.headers",
                                             responseHeaders);
        }
            responseHeaders.add(new Header(header, value));
}
```

Los únicos recursos que implementan el método PUT en su interfaz uniforme son Device y Functionality. Las clases que atienden las peticiones enviadas a este tipo de recurso son, respectivamente, DeviceServerResource y FunctionalityServerResource. En estos dos casos será necesario añadir estas tres caberas, basadas en las expuestas en [4]:

- **Access-Control-Allow-Origin:** Activa el uso de *CORS*. Su valor se puede establecer específicamente al dominio en el cual se quiere activar el uso de *CORS*. Se establece a valor **"*"** (todos los dominios).
- **Access-Control-Allow-Methods**: Activa el uso de *CORS* únicamente para los métodos HTTP establecidos en su valor. Se establece al valor **"GET, PUT, OPTIONS"** que son los métodos de la interfaz uniforme HTTP implementados.

- **Access-Control-Allow-Headers**: Indica qué cabeceras es necesario que se incluyan en la petición para que *CORS* pueda enviar la respuesta. Se establece en el valor **"origin, content-type, accept"**. Las cabeceras *Content-Type* y *Accept* ya se encontraban añadidas a la petición mediante la aplicación cliente.

Por lo tanto el método *doInit()* que inicializa estos recursos se modifica con la inicialización de estas tres cabeceras, quedando de la siguiente forma:

```
@Override
protected void doInit() throws ResourceException {
        this.funcId = getAttribute("funcId").toString();
        setCustomHttpResponseHeader("Access-Control-Allow-Origin", "*");
        setCustomHttpResponseHeader("Access-Control-Allow-Methods", "GET, PUT, OPTIONS");
        setCustomHttpResponseHeader("Access-Control-Allow-Headers",
                                    "origin, content-type, accept");
}
```

Por otra parte, las clases que sólo implementan los métodos GET y OPTIONS de la interfaz uniforme HTTP tan solo necesitan añadir la cabecera Acces-Control-Allow-Origin, quedando por tanto sus métodos *doInit()* de la siguiente forma:

```
@Override
protected void doInit() throws ResourceException {
        metodo = this.getMethod().toString();
        setCustomHttpResponseHeader("Access-Control-Allow-Origin", "*");
}
```

Una vez agregadas estas cabeceras, se puede comprobar que efectivamente son añadidas a las cabeceras de las respuestas a las peticiones. Por ejemplo, al realizar una operación GET, PUT u OPTIONS sobre el recurso del tipo DeviceFunctionality con URI http://localhost:8222/dfs/DF-MENJ.IL.AUXILIAR podemos observar a través de RESTClient que las cabeceras de respuesta son estas:

```
1.  Status Code                 : 200 OK
2.  Accept-Ranges               : bytes
3.  Access-Control-Allow-Headers : origin, content-type, accept
4.  Access-Control-Allow-Methods : GET, PUT, OPTIONS
5.  Access-Control-Allow-Origin  : *
6.  Content-Type                : application/json; charset=UTF-8
7.  Date                        : Fri, 27 Feb 2015 15:52:24 GMT
8.  Server                      : Restlet-Framework/2.1.7
9.  Transfer-Encoding           : chunked
10. Vary                        : Accept-Charset, Accept-Encoding, Accept-Language, Accept
```

Figura 49 - Cabeceras CORS añadidas a las respuestas HTTP

Al incluir estas cabeceras en las respuestas, *CORS* queda activado y la restricción de Origen Cruzado de JavaScript es solventada, permitiendo por lo tanto la comunicación entre aplicaciones web que corren en diferentes dominios.

6.6. Prueba de ejecución del cliente web

Como punto final de la descripción del cliente web se incluye un pequeño ejemplo de ejecución en el cual se pretende realizar una acción sobre un DeviceFunctionality.

En primer lugar, y como es obvio, debemos tener en funcionamiento los siguientes elementos:

- Servidor de la SmartHome con URI http://localhost:8222
- Aplicación cliente con URI http://localhost/cliente_web

Iniciamos un navegador web y accedemos a la URI del cliente:

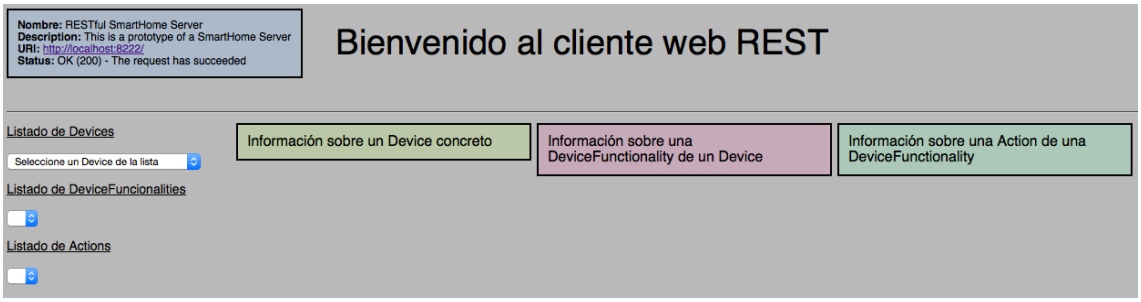

Figura 50 - Página de inicio del cliente web

Al acceder a la página de inicio se ejecutan automáticamente dos consultas:

- **Consulta GET sobre el recurso principal del servidor de la Smarthome:** esta consulta se realiza sobre la URI http://localhost:8222 y se usa para mostrar la información de la SmartHome.
- **Consulta GET sobre el recurso Devices:** esta consulta se realiza sobre la URI http://localhost:8222/devices y se usa para generar el listado de Devices.

Figura 51 - Listado de Devices

En el ejemplo seleccionaremos de entre el listado el Device con el nombre *Porta, el cual representa a una de las puertas de la SmartHome.*

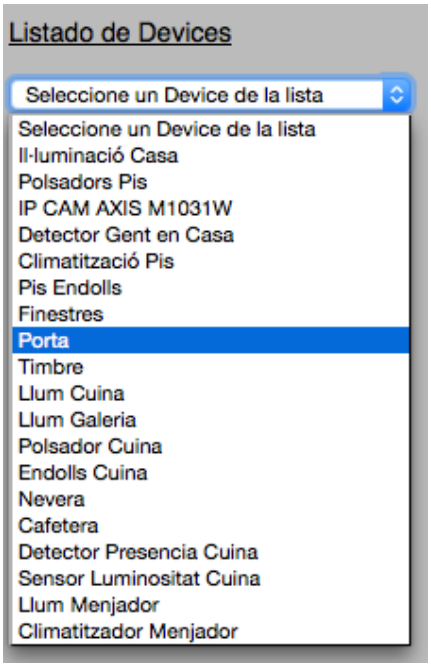

Figura 52 - Selección de un Device del listado.

Al seleccionar el Device realizamos una nueva consulta GET, esta vez sobre la URI de este Device. En este caso esta URI es

http://localhost:8222/devices/DEV-PORTA.PIS

Con el JSON obtenido por la consulta GET sobre el Device se realizan dos acciones:

- Actualizar la información del Device sobre la etiqueta destinada a esta función.
- Generar el listado de recursos del tipo DeviceFunctionality relacionadas con este Device.

En la siguiente imagen se aprecia el resultado de ambas acciones:

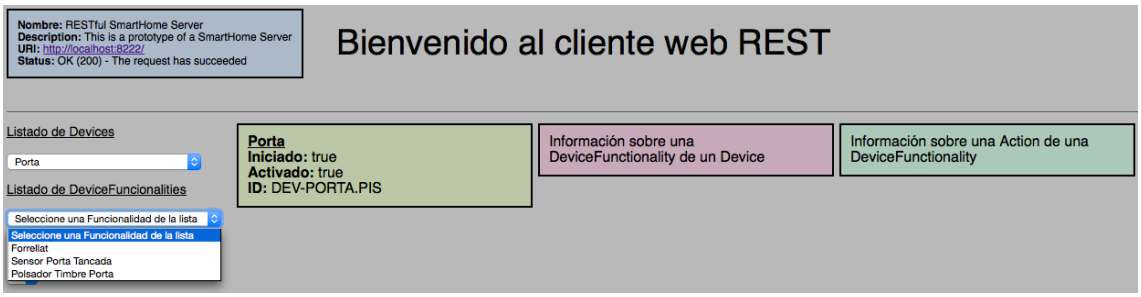

Figura 53 - Información del Device y listado de recursos DeviceFunctionality

De la lista de recursos DeviceFunctionality seleccionamos el que se llama *Forrellat*, que representa la cerradura de la puerta.

De nuevo al realizar la selección se lleva a cabo una nueva consulta GET, esta vez sobre la URI del recurso DeviceFunctionality, que es esta: http://localhost:8222/dfs/DF-PORTA.FORRELLAT

Con el JSON generado con esta nueva consulta GET se realizan dos acciones:

- Actualizar la información del recurso DeviceFunctionality en la etiqueta div destinada a esta función.
- Generar el listado de acciones (*actions*) que se pueden realizar sobre este tipo de DeviceFunctionality. Concretamente se trata de un *bistable*, por lo que sus acciones son solamente dos: cambiar su estado a ON y cambiar su estado a OFF.

En la siguiente imagen se aprecia el resultado de ambas acciones:

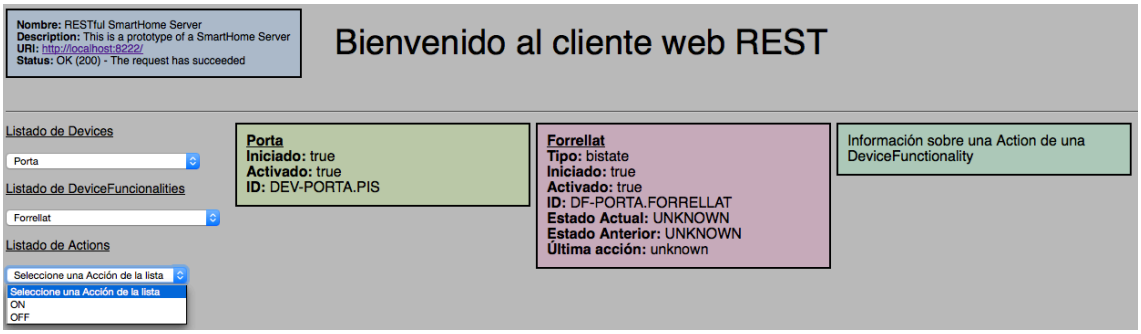

Figura 54 - Información de la DeviceFunctionality y listado de acciones

Es importante resaltar el estado UNKNOWN (desconocido) en el cual se encuentra la DeviceFunctionality, dado que no hemos interactuado aún con ella, y que tiene relación con lo ya comentado en el punto 4.3.5.2. Operación GET.

Seleccionamos una acción cualquiera, y dado que el estado actual es desconocido, el estado mostrado será modificado sea cual sea la acción elegida. Elegimos la acción ON para mandar a la cerradura la orden de abrirse, por lo que se realizará una petición PUT sobre su URI con el siguiente JSON en su *body*:

```
{"action":"biaON"}
```

Si la petición finaliza con éxito se mostrará el JSON generado en la etiqueta div destinada a tal efecto. La cerradura se abrirá y veremos la actualización del estado de la DeviceFunctionality en el navegador:

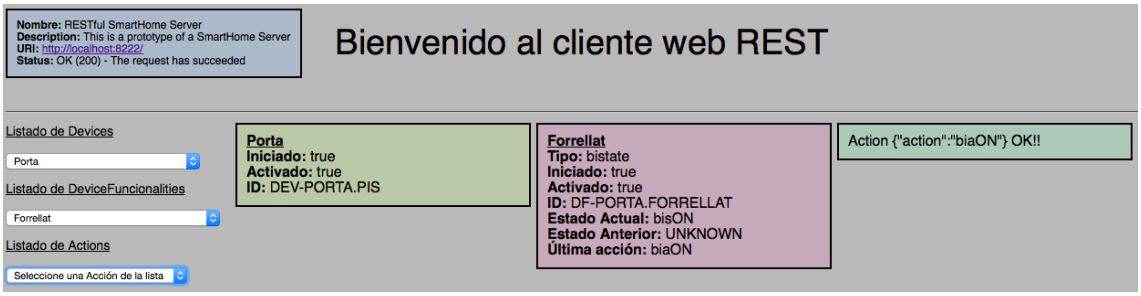

Figura 55 - Ejecución de la acción ON sobre la DeviceFunctionality

Aplicando una nueva operación, esta vez OFF a la misma DeviceFunctionality, veremos de nuevo su cambio de estado:

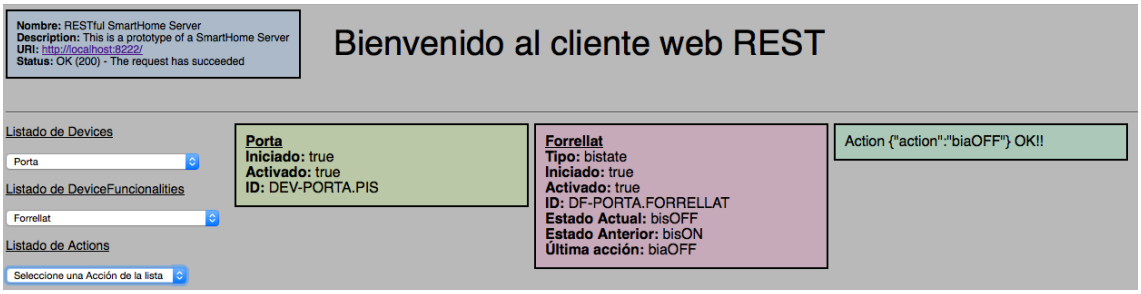

Ilustración 56 - Cambio de estado después de una acción OFF

7. Conclusiones

A la hora de diseñar interfaces de comunicación en las cuales participan actores muy heterogéneos es muy importante definir el conjunto de reglas que van a determinar el funcionamiento de esas comunicaciones.

Compartir una **interfaz universal** entre los elementos propios al sistema que ofrece la información y los elementos externos que la consumen y la utilizan es básico para facilitar esta comunicación e interacción.

En el presente proyecto se ha logrado desarrollar una interfaz de comunicaciones basada en **REST** y haciendo uso de **la interfaz uniforme** que proporciona el protocolo **HTTP**. Esta interfaz se ha integrado como enlace de comunicaciones entre el sistema modelo, en el que se definen los recursos de una vivienda inteligente, y una sencilla aplicación cliente. En el presente proyecto se ha tratado de respetar de la forma más fiel posible la filosofía REST y se puede considerar a la aplicación resultante como **RESTful**.

REST ofrece una serie de pautas que facilitan el diseño de interfaces de comunicación basadas en un protocolo estándar y extendido, de forma simple y sencilla. Además es altamente configurable y flexible, ya que permite elegir el protocolo de comunicación a utilizar, si bien lo más usual es hacer uso de **HTTP y de su interfaz uniforme.** Sobre dicha interfaz uniforme cada recurso del sistema implementará el subconjunto de la interfaz que más se adapte a su propósito.

El hecho de que todos los elementos implicados en la comunicación compartan una misma interfaz uniforme simplifica el diseño de aplicaciones cliente que accedan a estos recursos, puesto que basta con implementar las peticiones HTTP necesarias para obtener la información solicitada, o realizar la acción requerida.

REST especifica que se debe utilizar una interfaz estándar, pero no indica qué interfaz utilizar. **Lo importante es la uniformidad y que cada recurso utilice la interfaz HTTP de la misma forma**. Sin una interfaz uniforme, se debería aprender para cada recurso cómo éste espera recibir la información y cómo la va a devolver. Crear tus propios métodos HTTP es una muy mala idea. Tu vocabulario personalizado te posiciona en una comunidad de un solo individuo.

Como todas las peticiones se basan en los métodos de la interfaz uniforme HTTP no es necesario desarrollar nuevos métodos propios que realicen estas acciones. De esta forma podemos leer cualquier dispositivo realizando un GET sobre su URI, realizar determinada acción con un PUT y conocer su interfaz uniforme con un OPTIONS.

El API desarrollada en el presente proyecto no implementa todos los métodos de la interfaz estándar HTTP. Se podría extender el desarrollo de la interfaz uniforme que

implementan los recursos de este proyecto para añadir otros métodos que no están presentes, como POST, HEAD o DELETE. La funcionalidad de estos métodos sería parecida a la siguiente:

- **POST**: añade nuevos recursos
- **DELETE**: elimina un recurso.
- **HEAD**: proporciona la misma información que GET pero sin *payload* en la respuesta.

El método HEAD se suele emplear para realizar consultas rápidas y simples, normalmente como paso previo a realizar una consulta más compleja y pesada. Por ejemplo se utilizaría para comprobar si determinado recurso existe o está operativo antes de realizar una determinada acción con conste elevado sobre el mismo.

Por otra parte, en el proyecto no se contempla la posibilidad de añadir nuevos recursos al sistema SmartHome, pero se podría implementar un método que realizase esta acción. El método HTTP empleado sería POST y se añadiría al subconjunto de la interfaz uniforme del recuso Devices. La operación POST necesitaría un *body* con los siguientes elementos:

- Nombre del Device
- Identificador del Device

Esta información se transmitiría en formato JSON, como en este ejemplo:

```
{“Name”:”new_device”, ”ID”:”Dev_ID”}
```

El nuevo Device se crearía en estado inactivo y no accesible, sin funcionalidades relacionadas. El resto de información importante del Device se puede generar fácilmente a través de estos datos.

De forma similar se podría desarrollar un método que añadiese recursos DeviceFunctionality a un Device añadiendo la operación POST a la interfaz uniforme del recurso DeviceFunctionalities.

El método DELETE también se podría añadir a las interfaces uniformes de los recursos Devices y DeviceFunctionalities para que se permitiese eliminar recursos del tipo Device y DeviceFunctionality.

Otra de las posibilidades que no se llegan a explotar en el trabajo es la consulta y envío de peticiones a grupos de recursos del mismo tipo. Imaginemos que en una determinada habitación tenemos cinco luces, cada una representada por su correspondiente recurso DeviceFunctionality y de tipo *bistate*, y todas ellas enlazadas al Device con identificador *Cuadro-de-luces-salon*.

Si quisiéramos encender todas estas luces al mismo tiempo deberíamos lanzar la correspondiente orden PUT sobre cada uno de los recursos DeviceFunctionality relacionadas con el Device.

Otra forma más eficiente de hacerlo sería crear una colección de DeviceFunctionality que contuviese los Link de esas funcionalidades y mandar una única petición PUT a la URI de la colección (que sería un nuevo tipo de recurso, con su correspondiente interfaz uniforme) que encendiese todas las luces a la vez.

Al no tener que implementar métodos ajenos a la interfaz uniforme para realizar todas las acciones de comunicación con los recursos se consigue que la compatibilidad de la interfaz con aplicaciones cliente sea muy elevada, puesto que estas aplicaciones solo deben adaptar sus peticiones a la interfaz uniforme, intercalando con los recursos mediante los métodos propios del protocolo HTTP.

El manejo de errores también es estándar, no tenemos porqué desarrollar mensajes de error adaptados a nuestras aplicaciones. HTTP nos proporciona los posibles estados y los mensajes de error asociados a los mismos, lo que simplifica este tratamiento.

En cuanto al formato de las representaciones de los recursos transmitidas, es muy conveniente que dicho formato sea simple, sencillo, con poco peso y basado en texto, de forma que se transmita la información estrictamente necesaria, que la comunicación sea rápida y fluida. . Además este formato no debe restar funcionalidad a las representaciones ni presentar pérdida de información. JSON cumple con todos estos requisitos y es por ello que ha sido elegido como formato para el presente proyecto.

Siendo HTTP el protocolo de comunicación empleado, lo más natural es aprovecharlo para desarrollar aplicaciones cliente en formato web que aprovechen de forma nativa toda su funcionalidad. Estas aplicaciones cuentan con otra ventaja añadida, y es que se pueden ejecutar desde cualquier dispositivo hardware capaz de ejecutar un simple navegador web. Por lo tanto se pueden diseñar aplicaciones cliente que pueden ejecutarse en muy diversos dispositivos, desde *smartphones* a *tablets* o portátiles, independientemente de su sistema operativo o fabricante.

Sin embargo la elección de este protocolo no limita el alcance de su usabilidad solo a los dispositivos de este tipo. Se pueden desarrollar aplicaciones cliente basadas en sensores de movimiento o control por voz que hagan uso a su vez de la interfaz uniforme HTTP.

Ante tal grado de accesibilidad la seguridad y el control de acceso al sistema cobra una importancia destacada. Una vez más se pueden aprovechar los mecanismos estándar que HTTP ofrece para implementar un sistema de seguridad basado en las cabeceras HTTP. En [13] se detalla lo necesario para realizar esta implementación. Sería sencillo hacerlo utilizando las cabeceras que implementan el control de acceso, utilizando para dicho control el filtro que el diseñador de la interfaz considere: usuarios autentificados, control de acceso por determinadas IPs, etc.

RESTlet también nos proporciona un filtro previo que se puede utilizar para filtrar peticiones al servicio, por motivos de seguridad, o para redirigir ciertas peticiones a otros servicios (Service filtering).

8. Enlaces y bibliografía

[1] Ballard, P. y Moncur, M. (2009). Ajax, JavaScript y PHP. Madrid: Anaya Multimedia.

[2] Roger, L. C. () Building Web Services the REST Way. Recuperado el 28 de Septiembre de 2014, de http://www.xfront.com/REST-Web-Services.html

[3] Flax, A. (2008). Custom HTTP Response Headers with Restlet. Recuperado el 10 de Enero de 2015, de http://blog.arc90.com/2008/09/15/custom-http-response-headers-with-restlet/

[4] Sharp, R. (2011). Getting CORS working. Recuperado el 12 de Diciembre de 2014, de https://remysharp.com/2011/04/21/getting-cors-working

[5] Gourley, D. y Totti, B. (2002). HTTP: The Definitive Guide. Sebastopol: O'Reilly.

[6] HTTP Status Codes. Recuperado el 17 de Noviembre de 2014, de http://www.restapitutorial.com/httpstatuscodes.html

[7] JavaScript JSON. Recuperado el 20 de Enero de 2015, de http://www.w3schools.com/js/js_json.asp

[8] JSON: The Fat-Free Alternative to XML. Recuperado el 1 de Febrero de 2015, de http://www.json.org/xml.html

[9] Elkstein, M. () Learn REST: a Tutorial. Recuperado el 5 de Octubre de 2014, de http://rest.elkstein.org/

[10] Restlet 2.1 – Tutorial. Recuperado el 20 de Abril de 2014, de http://restlet.com/technical-resources/restlet-framework/tutorials/2.1

[11] Sandoval, J. (2009). RESTful Java Web Services. Birmingham: Packt Publishing.

[12] Richardson, L. y Ruby, S. (2007). RESTful Web Services. Sebastopol: O'Reilly.

[13] Subbu, A. (2010). RESTful Web Services Cookbook. Sebastopol: O'Reilly.

[14] Louvel, J., Templier, T. y Boileau, T. (2013). RESTlet in action. Shelter Island, NY: Manning

[15] The OSGi Architecture. Recuperado el 7 de Marzo de 2014, de http://www.osgi.org/Technology/WhatIsOSGi

[16] Monsur, H. (2013). Using CORS. Recuperado el 14 de Enero de 2015, de http://www.html5rocks.com/en/tutorials/cors/?redirect_from_locale=es

[17] Workbench User Guide. Recuperado el 12 de Marzo de 2014, de http://help.eclipse.org/juno/index.jsp?nav=%2F0

9. Anexos

9.1. Anexo 1 - Configuración del entorno de desarrollo de la API

Para poder utilizar las funcionalidades que aporta la aplicación de SmartHome, así como para poder emplear el framework de RESTlet deberemos antes de nada configurar el entorno de trabajo de nuestra aplicación.

Al seleccionar Eclipse como entorno de desarrollo, en primer lugar crearemos un espacio de trabajo (*workspace*) en el que colocaremos los directorios que contienen los proyectos java necesarios para correr la aplicación de la SmartHome. La aplicación de la SmartHome se compone de tres proyectos java, denominados SmartHomes.Model, SmartHomes.Server y SmartHomes.Server.Configuration. Con tan solo importar dichos proyectos a nuestro workspace los tendremos accesibles.

No se detalla la finalidad de estos proyectos, ya que dicho desarrollo queda al margen del objetivo del presente trabajo. Como comentario se añade una breve descripción de cada uno de ellos:

- **SmartHomes.Model** – Crea las representaciones de los recursos y sus funcionalidades.
- **SmartHomes.Server** – Crea el servidor de la aplicación. Se puede considerar que el servidor es la propia SmartHome, de la cual colgarán el resto de recursos.
- **SmartHomes.Server.Configuration** – Incluye las configuraciones necesarias para el correcto funcionamiento del servidor. Es una forma sencilla de cambiar la configuración de dicho servidor sin cambiar excesivas líneas de código.

Una vez ya contamos con los tres proyectos de la Smarthome disponibles en el wokspace podemos crear un nuevo proyecto java que dará soporte a nuestra nueva aplicación. El nombre de nuestro nuevo proyecto es SmartHomes.REST.API, siguiendo con las reglas de nomenclatura utilizadas en los proyectos de la SmartHome. En la siguiente imagen se muestra el workspace con los cuatro proyectos.

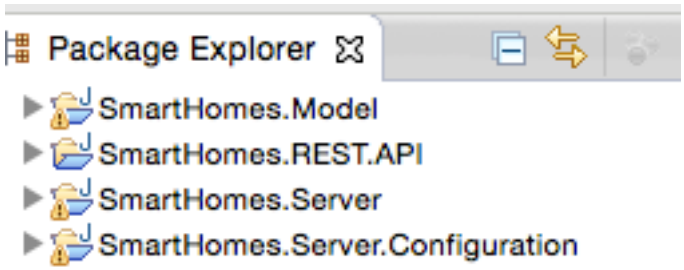

Figura 57 - Workspace de Eclipse

9.1.1. Añadir las librerías externas necesarias

Para que la aplicación pueda funcionar es necesario que añadamos al proyecto las referencias a ciertas librerías externas que serán necesarias para la compilación de ciertas partes del código fuente. Se detallan de forma básica a continuación.

RESTlet: El Framework de desarrollo de APIs REST basado en Java cuenta con sus paquete propio de librerías que podemos descargar directamente de su página web http://www.restlet.org

En este proyecto se emplea la versión 2.1.7 de RESTlet. Esta versión actualmente se encuentra descatalogada por el equipo de desarrollo de RESTlet. En el momento de redactar la presente memoria la versión actual de RESTlet es la 2.3 y es la única que podemos descargar. Sin embargo esto no afecta al presente proyecto, que se sigue basando en la versión 2.1.7

De entre las diferentes versiones que se ofrecen en la web seleccionamos la versión para Java Estándar Edition. El fichero en cuestión se denomina restlet-jse-2.1.7.zip y ocupa unos 80 MB de disco.

Una vez hemos descargado el fichero zip, descomprimimos el contenido en una carpeta. En el interior de la carpeta descomprimida encontraremos un directorio denominado lib y en su interior al fichero org.restlet.jar que contiene las librerías esenciales para poder utilizar el Framework en nuestra aplicación.

JSON: Para poder emplear los ficheros en formato JSON necesarios para el funcionamiento de nuestra aplicación, así como las funciones necesarias para crearlos e interpretarlos, debemos añadir a nuestro proyecto las librerías especificas para el tratamiento de este tipo de ficheros.

Para ello descargaremos de la web http://www.json.org el fichero org.json.jar que contiene dichas librerías.

Librerías RESTlet para el manejo de JSON: junto con el paquete principal de RESTlet se incluyen un gran número de otros paquetes que añaden funcionalidades

opcionales a las aplicaciones desarrolladas con este Framework. Estas librerías se denominan externas y necesitaremos añadir a nuestro proyecto las referentes al manejo de ficheros JSON que se engloban en el paquete org.restlet.ext.json.jar y lo encontraremos en el directorio lib de la carpeta donde hemos descomprimido el resto del Famework.

Librería SmartHomes.Server.Common: esta librería forma parte del entorno de desarrollo de la aplicación de la SmartHome y se le proporciona al alumno junto con el resto de proyectos Java que conforman dicha aplicación. El fichero contiene las funciones necesarias para conectar nuestra API con los recursos procedentes de la SmartHome. A diferencia del resto de la aplicación, este elemento se le entrega al alumno directamente como fichero jar, de modo que puede ser añadido a las librerías externas del proyecto de la API REST. Con las funciones de estas librerías podremos interactuar con instancias de los recursos de la SmartHome y utilizarlos en nuestra API.

Tras configurarlo, el Build Path de nuestro proyecto debería quedar más o menos como en la siguiente imagen:

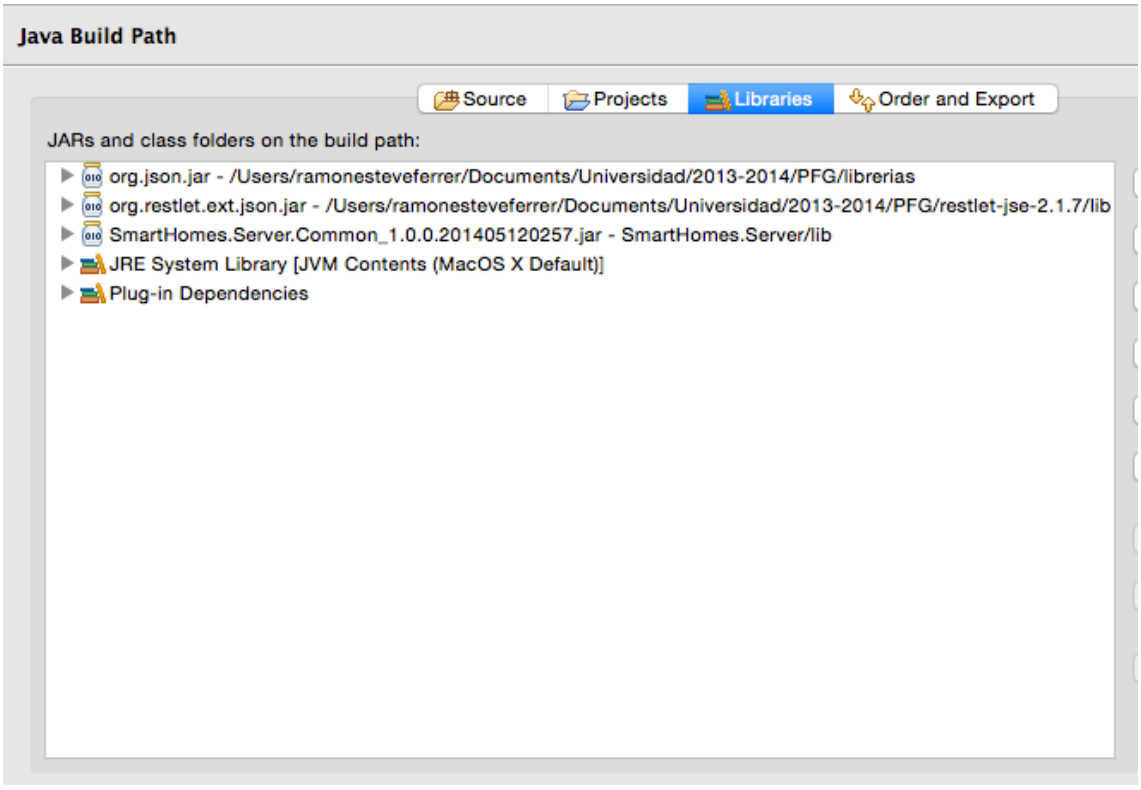

Figura 58 - Buid Path del proyecto

9.1.2. Configurar las dependencias

Incluir las librerías no es suficiente. Para poder emplear la mayoría de los métodos debemos además configurar las dependencias entre nuestro proyecto y las librerías empleadas. Para ello accederemos a la configuración del fichero META-INF/MANIFEST.MF de nuestro proyecto. En la pestaña *Dependencies* de este fichero añadimos las dependencias relativas a los paquetes de las librerías que queremos emplear. De otra manera Eclipse no será capaz automáticamente de encontrar en dichas librerías los métodos que necesitamos emplear.

Las dependencias que debemos configurar son en su mayor parte referentes al paquete RESTlet y al manejo de JSON. En concreto debemos añadir las siguientes:

- org.json
- org.osgi.framework
- org.restlet
- org.restlet.data
- org.restlet.ext.json.
- org.restlet.representation
- org.restet.routing
- org.restlet.service
- org.restlet.util

Una vez configuradas, las dependencias deben quedar como en la imagen inferior:

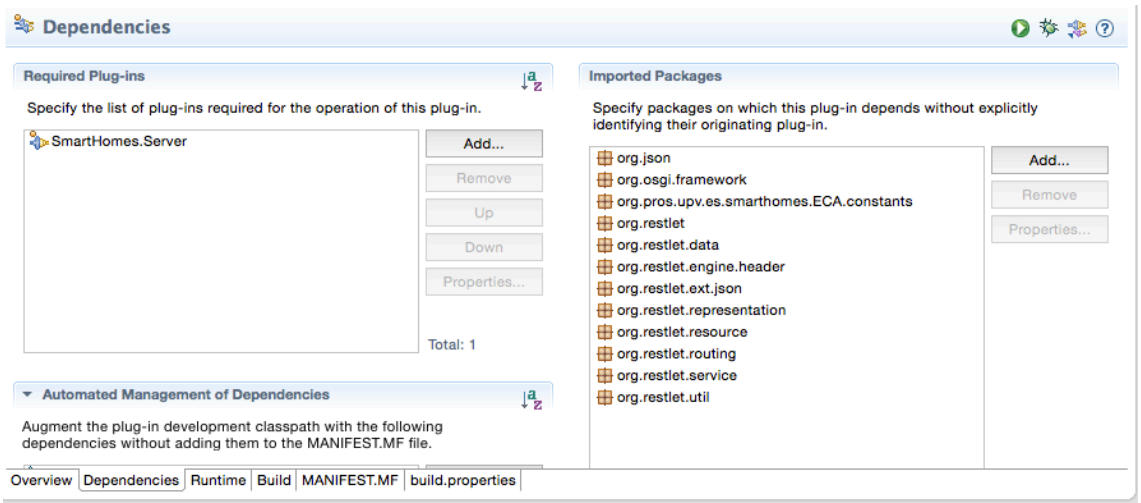

Figura 59 - Configurar dependencias

En la imagen aparecen algunas dependencias que se requieren en partes particulares de la aplicación y que se documentan en su apartado correspondiente.

Finalmente, en la pestaña MANIFEST.MF deben quedar reflejadas todas las dependencias añadidas. Debe presentar un aspecto similar a este:

SmartHomes.REST.API

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SmartHomes.REST.API
Bundle-SymbolicName: SmartHomes.REST.API
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: smarthomes.rest.api.Activator
Import-Package: org.json,
 org.osgi.framework,
 org.pros.upv.es.smarthomes.ECA.constants,
 org.restlet,
 org.restlet.data,
 org.restlet.engine.header,
 org.restlet.ext.json,
 org.restlet.representation,
 org.restlet.resource,
 org.restlet.routing,
 org.restlet.service,
 org.restlet.util
Require-Bundle: SmartHomes.Server
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Figura 60 - MANIFEST.MF

9.1.3. Creación de los packages necesarios.

Dividiremos nuestra API en diferentes paquetes de clases (*packages*, como se denominan en Eclipse), con el fin de clasificar las clases nuevas que se van creando según su propósito. Los cuatro *packages* necesarios son estos:

- **smarthomes.rest.api** – En este *package* se incluyen las clases necesarias para la configuración del contexto de nuestra aplicación. Este contexto es necesario para consultar los recursos registrados en la SmartHome.
- **smarthomes.rest.api.common** – En este *package* se ubican las clases del tipo interfaz, que describen los métodos de ámbito público que se deben desarrollar en el resto de clases.
- **smarthomes.rest.api.configuration** – Ese *package* contiene una única clase Configuration que sirve para establecer los valores de los parámetros configurables de la aplicación, tales como URI y puerto del servidor. De este modo configurar estos parámetros es rápido, sencillo y no implica modificar excesivas líneas de código.
- **smarthomes.rest.api.exceptions** – Estas clases se utilizan para el tratamiento de los errores y excepciones que pueden llegar a ocurrir durante la ejecución de los métodos de las clases del resto de la aplicación. Es una forma de gestionar de forma eficiente los mensajes de error y los métodos que desencadenan estos errores.
- **smarthomes.rest.api.server** – Este *package* contiene las clases principales de la aplicación, que a su vez implementan las interfaces de las clases del *package* smarthomes.rest.api.common. Además de estos métodos las clases de este *package* implementan otros métodos de ámbito interno.

La estructura final de los packages de la aplicación se muestra en la siguiente imagen:

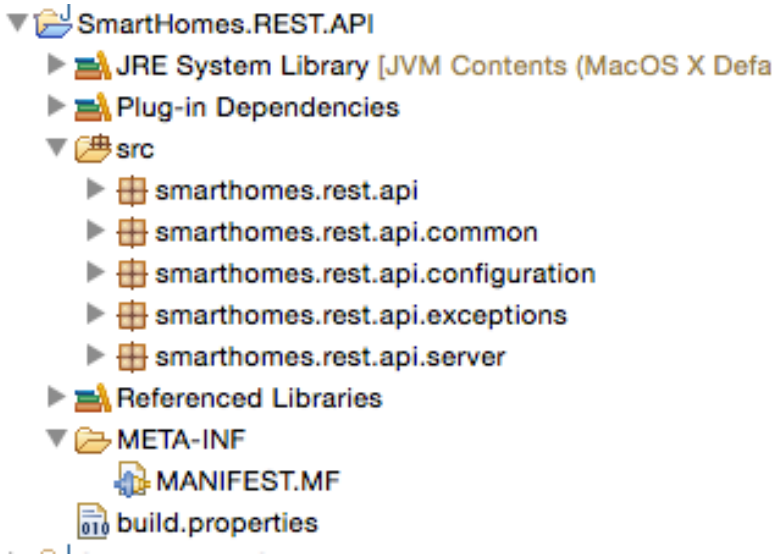

Figura 61 - Estructura de packages

9.2. Anexo 2 - Configuración del cliente web

En primer lugar realizamos la descarga del entorno de desarrollo Eclipse Luna en su versión para desarrollo web desde su página web oficial https://eclipse.org/

Al iniciar Eclipse podremos seleccionar la ubicación de nuestro workspace, o crear uno nuevo. A continuación creamos el proyecto web que albergará el código de nuestro cliente. Para crear un nuevo proyecto web, desde el menú de Archivo / Nuevo seleccionamos la opción “Proyecto web”

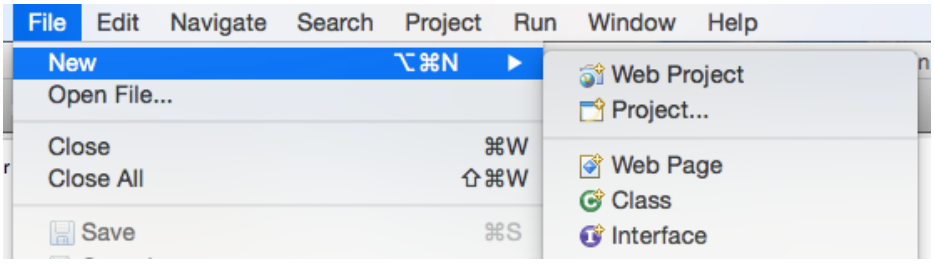

Figura 62 - Crear un nuevo proyecto web

El nombre de nuestro proyecto es cliente_web. Eclipse automáticamente creará el proyecto en el wokspace y su estructura:

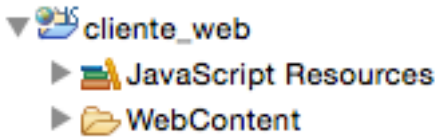

Figura 63 - Estructura del proyecto web

En el interior de la carpeta WebContent es donde se almacenan los ficheros html y JavaScript que forman parte de la aplicación.

9.2.1. Puesta en marcha del servidor web

Como ya se ha comentado, se instalará la solución integrada XAMPP. En primer lugar descargamos el instalador de su página oficial https://www.apachefriends.org/es/index.html

Una vez instalada la aplicación podemos controlar los servicios que ofrece desde el programa de gestión que incluye.

Figura 64 - Icono del manager de XAMPP

Iniciando el manager veremos una pantalla como esta:

Figura 65 - pantalla de inicio del manager de XAMPP

Para empezar a controlar nuestros servidores seleccionaremos la pestaña “Manage Servers”, la cual nos muestra este aspecto:

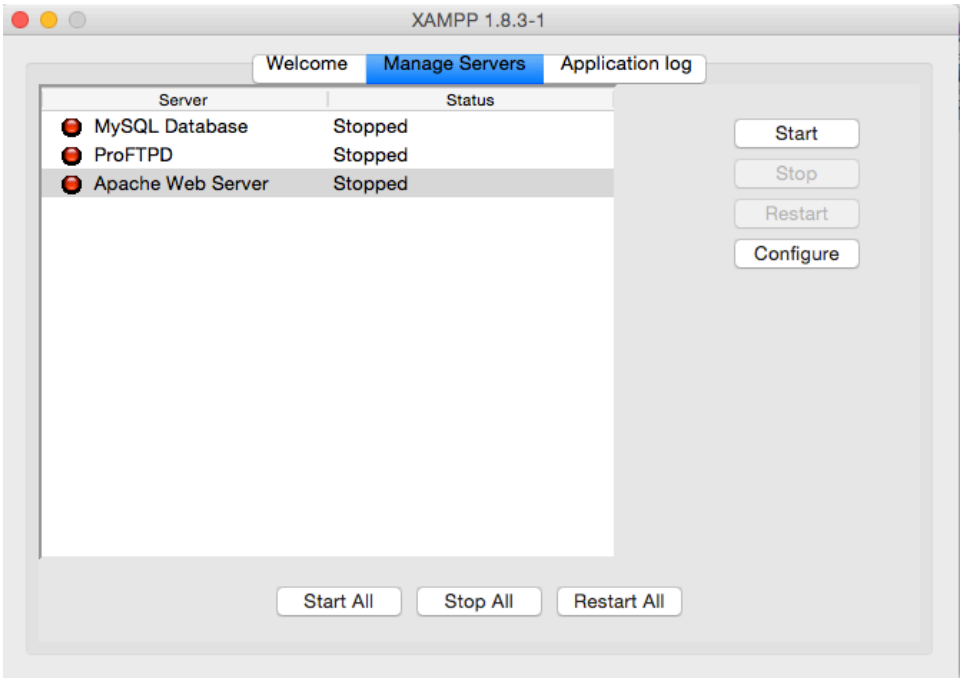

Figura 66 - Servidores XAMPP

Como se puede apreciar en la anterior imagen , XAMPP ofrece muchas más servicios de los necesarios para el desarrollo del cliente web. Solo es necesario iniciar el servidor web Apache, seleccionando este servidor y pulsando el botón Start.

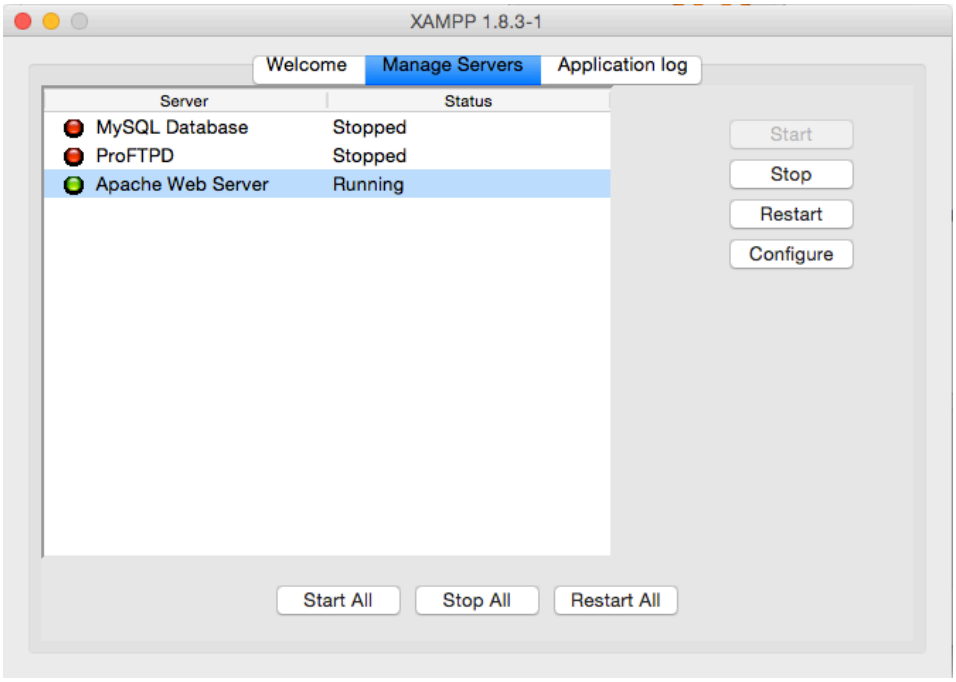

Figura 67 - Servidor Apache iniciado

Una vez iniciado el servidor, podemos abrir un navegador web y acceder a su página por defecto en la URL http://localhost

Figura 68 - Página de inicio de XAMPP

La página web de nuestro proyecto colgará de la URL principal del servidor, concretamente se enlazará a la URI http://localhost/cliente_web

Tabla de figuras

Índice de tablas