Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Control de una vivienda inteligente mediante calendario

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Guillermo César Monzó Elvira

Tutor: Joan Josep Fons i Cors

2013-2014

Resumen

Este proyecto consiste en dotar de un sistema de calendario a una vivienda inteligente, de forma que el usuario potencial del servicio sea capaz de programar eventos de tipo domótico. Cuando ocurre un evento, se desencadena la acción planificada en un dispositivo del hogar. Se aborda pues, un problema de interoperabilidad entre una vivienda inteligente y un servicio externo de gestión de calendarios.

La solución que se ofrece es el desarrollo de dos herramientas: un servidor y un cliente. La aplicación servidor implementa una capa de servicios REST y es la que enlaza la vivienda con el sistema de calendarios exterior. La aplicación cliente hace uso de la funcionalidad ofrecida por esta capa de interacción para facilitar al usuario final la planificación de eventos que produzcan acciones en la vivienda.

Este proyecto está dirigido a un lector con conocimientos en protocolos web, programación de aplicaciones, tecnologías de la información y/o sea un apasionado de la domótica.

Palabras clave: API, REST, domótica, calendario, evento.

Abstract

This project consists of providing a calendar system for intelligent households, so that the service's user is capable of programming home automation events. When an event happens the planned action is triggered in a household device. This is, therefore, an interoperability problem between an intelligent household and an external calendar management service.

A proposed solution is the development of two tools: a server and a client. The server application implements a REST service layer and it is this that connects the household with the external calendar system. The client application makes use of the functionality offered by this application layer to make it easier for the user to plan events that produce actions in the household.

This project is directed at a reader with knowledge of web protocols, application programming, information technologies and/or anyone with a keen interest in home automation.

Keywords : API, REST, automation, calendar, event.

Índice de siglas

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
MVC	Modelo Vista Controlador
PHP	PHP Hypertext Pre-processor
REST	REpresentational State Transfer
XML	

Índice de ilustraciones

Ilustración 1: Estructura de carpetas de la infraestructura 16
Ilustración 2: Servidor de la SmartHome 16
Ilustración 3: Diagrama de interacción 20
Ilustración 4: Tablas de la base de datos 22
Ilustración 5: Diseño de la interfaz de inicio de sesión 23
Ilustración 6: Diseño de la interfaz móvil de inicio de sesión 23
Ilustración 7: Diseño de la interfaz de gestión de calendarios 24
Ilustración 8: Diseño de la interfaz móvil de gestión de calendarios 24
Ilustración 9: Diseño de la interfaz de añadir eventos 25
Ilustración 10: Diseño de la interfaz de añadir eventos 26
Ilustración 11: Creación de un nuevo proyecto en Google Developer Console......27
Ilustración 12: Habilitar el acceso al API de Google Calendar 28
Ilustración 13: Creación de las credenciales para la aplicación servidor 28
Ilustración 14: Resumen de las credenciales 29
Ilustración 15: Estructura de la aplicación servidor 30
Ilustración 16: Diagrama de autenticación, autorización e interacción con Google Calendar mediante OAuth2......31
Ilustración 17: Algoritmo de control de eventos 52
Ilustración 18: Estructura de la aplicación web......57
Ilustración 19: Cabecera usuario de Google 62
Ilustración 20: Cabecera usuario administrador 62
Ilustración 21: Panel de administración de dispositivos 63
Ilustración 22: Vista detalle de un dispositivo 63
Ilustración 23: Vista UPDATE de un dispositivo 63
Ilustración 24: Vista CREATE de un dispositivo...... 64
Ilustración 25: Inicio de sesión...... 64
Ilustración 26: Inicio de sesión (versión móvil) 65
Ilustración 27: Gestión de calendarios 65
Ilustración 28: Gestión calendarios (versión móvil) 66
Ilustración 29: Formulario de creación de eventos 66
Ilustración 30: Definición de a recurencia del evento......67
Ilustración 31: Formulario de creación de eventos (versión móvil)......67
Ilustración 32: Error en el inicio de sesión...... 68

Tabla de contenidos

1.Introducción 9

1.1. Motivación 9

1.2. Problema a resolver 9

1.3. Objetivos 10

1.4. Metodología y planificación 10

2. Especificación de requisitos 12

2.1. Servidor 12

2.2. Cliente 12

3. Análisis 13

3.1. Tecnologías 14

3.1.1. Google Calendar 14

☐ Autenticación y autorización con OAuth2 14

3.1.2. Restlet framework 15

3.1.3. Infraestructura de vivienda inteligente 15

3.1.4. Base de datos 18

3.1.5. Yii framework 19

3.1.6. AJAX, Bootstrap y jQuery 19

3.2. Interacción 20

4. Diseño 20

4.1. Servidor 20

4.1.1. Arquitectura en capas 20

4.1.2. Recursos REST disponibles en el API 21

4.2. Base de datos 21

4.3. Aplicación web cliente 22

5. Implementación 27

5.1. Preparación previa 27

5.1.1. Librerías cliente del API de Google Calendar 27

5.1.2. Registrar la aplicación en Google Developer Console. 27

5.1.3. Descargar Restlet framework 29

5.1.4. Herramientas de desarrollo 29

5.2. Servidor 30

5.2.1. Credencial de acceso mediante el protocolo OAuth2 30

5.2.2. Enrutamiento de peticiones REST 32
5.2.3. Recursos del API 34
- ☐ Uso de la cabecera HTTP Authorization 34
- ☐ CalendariosResource 35
- ☐ CalendarioResource 39
- ☐ EventosResource 41

5.2.4. Eventos domóticos 47
5.2.5. Gestión de eventos en tiempo real 51
5.2.6. Desencadenamiento de acciones en la vivienda inteligente 55
5.3. Cliente 56
5.3.1. Modelo-Vista-Controlador 56
5.3.2. Autenticación y autorización 57
5.3.3. Peticiones AJAX autenticadas 58
5.3.4. Interfaces de administración 62
5.3.5. Interfaces de usuario 64
6. Problemas encontrados 68
7. Conclusiones 69
Bibliografía 70

1. Introducción

Este proyecto final de grado consiste en la elaboración de una herramienta capaz de programar eventos en un calendario. Un evento consistirá en realizar una acción domótica en una vivienda inteligente. En este contexto, podemos incluir no solo hogares, si no también fábricas, supermercados, grandes almacenes, etc., siempre y cuando posean una infraestructura adecuada para interactuar con ellas, de forma que sea posible automatizar tareas o controlar dispositivos.

1.1. Motivación

Actualmente, la domótica está en auge gracias a las últimas novedades en avances tecnológicos, cuyo objetivo se centra en mejorar la calidad de vida del usuario en el hogar. Por ejemplo, los últimos televisores y frigoríficos inteligentes recomiendan películas o hacen la compra. El ahorro energético también es una realidad, ya que los sensores de temperatura o de luz de la instalación regularán la climatización e iluminación del hogar.

Implementar un calendario basado en eventos programados incrementa el potencial de una vivienda inteligente. Por ejemplo, un sistema de riego programado se activa un día y durante un tiempo definidos, realizando un riego óptimo que favorece el cuidado del jardín. O por ejemplo, planificar un evento que ocurra durante los meses de invierno que consista en encender la calefacción de las habitaciones a partir de una hora concreta y desactivarla cuando la temperatura de la estancia supere el umbral establecido.

1.2. Problema a resolver

Durante el desarrollo de este trabajo se ha utilizado una versión reducida de una implementación real de una vivienda inteligente, facilitada por el tutor de este proyecto. Esta herramienta modificada ofrece la posibilidad de desplegar un modelo de vivienda inteligente virtual con la que establecer una conexión simulada y consultar o consumir recursos, como dispositivos o sensores, sin la necesidad de estar conectado físicamente a ella.

La infraestructura proporcionada contiene una capa de interoperabilidad con servicios REST (Representational State Transfer) que permite interactuar mediante peticiones HTTP (Hypertext Transfer Protocol) con los dispositivos de la vivienda, llamados *Device Functionalities*. Gracias a esta API (Application Programming Interface) se puede desarrollar cualquier tipo de aplicación que consuma o consulte estos recursos.

Se ha decidido utilizar Google Calendar para la gestión de los calendarios de la vivienda ya que ofrece una API REST para desarrolladores muy completa y práctica. Permite crear, editar y eliminar calendarios y eventos, añadir recordatorios, participantes, etc. Además, existe la posibilidad de definir eventos periódicos, una funcionalidad muy interesante y a tener en cuenta en el ámbito de una vivienda inteligente.

Así pues, la problemática en este instante es cómo enlazar un sistema externo, como Google Calendar, con la vivienda inteligente. Para resolver este obstáculo, se propone la creación de dos aplicaciones: un servidor y un cliente. La aplicación servidor se encarga de consultar y consumir los recursos de la vivienda inteligente mediante su API de acceso y gestiona la conexión con una cuenta de Google Calendar para obtener los calendarios y eventos del usuario.

Además, debe implementar una capa de interoperabilidad REST propia para interactuar con la aplicación cliente: un panel de control web en el que definir los eventos que se deseen. De esta forma queda resuelta la interoperabilidad con la vivienda inteligente.

1.3. Objetivos

El objetivo principal de este proyecto es añadir una nueva funcionalidad a la vivienda inteligente: programar eventos domóticos. Utilizando un calendario, el usuario puede añadir un nuevo suceso y en el momento –fecha y hora– en el que ocurra, se desencadena una acción en la vivienda.

Para ello, es necesario:

- Desarrollar una aplicación servidor que conecte la vivienda inteligente con Google Calendar, utilizando sus respectivas APIs de acceso. y que además implemente una capa propia de servicios REST.

- Desarrollar una aplicación web cliente que interactúe únicamente con la anterior, permitiendo al usuario crear eventos (periódicos o no) en su cuenta de Google Calendar y especificar qué acciones se realizarán en la vivienda cuando ocurra el evento y durante cuánto tiempo. Además, desde un panel de gestión, un usuario administrador puede ser capaz de insertar nuevos dispositivos con sus correspondientes acciones, en caso de adquisición de un nuevo elemento.

- La aplicación servidor debe ser capaz de consultar estos eventos y provocar los cambios en los dispositivos del hogar.

1.4. Metodología y planificación

El desarrollo de este proyecto ha pasado por distintas fases hasta obtener un resultado final que cumple con el objetivo principal del proyecto. Los procedimientos realizados en cada fase han sido de vital importancia, ya que una etapa sin unos buenos cimientos y sin una investigación previa de las distintas alternativas, provoca que las etapas siguientes se vean afectadas de forma negativa.

Es por esto que el presente proyecto atravesará primero una fase de requisitos funcionales, para especificar y asegurar lo que se espera del producto final. Seguidamente viene la fase de análisis, en la que se realiza un estudio de las tecnologías disponibles relacionadas con el contexto de este trabajo y se decide cuál utilizar en función de sus características y de las funcionalidades que puede aportar al desarrollo del proyecto. Tras la etapa de análisis, comienza la fase de diseño. Esta

importante fase sirve para plasmar los requisitos del proyecto, tomar decisiones y empezar a darle forma. Se centra en cómo será la estructura de las aplicaciones a desarrollar y de sus componentes. Inmediatamente después de la fase de diseño, se aborda la fase de implementación de las aplicaciones servidor y cliente. Esta etapa responde al cómo se han realizado las aplicaciones y qué elementos hacen que la integración entre la vivienda inteligente y un servicio externo como Google Calendar quede resuelta. Finalmente, se exponen las conclusiones personales que se derivan del producto final.

2. Especificación de requisitos

En la fase de especificación de requisitos se detallan los requerimientos funcionales y las limitaciones que se esperan de las dos aplicaciones a desarrollar:

2.1. Servidor

- Ser el nexo de unión entre Google Calendar y la vivienda inteligente.
- Disponer de una capa de servicios REST para poder interactuar con ella mediante métodos HTTP (GET, POST, PUT, DELETE, OPTIONS).
- Recoger la información que reciba de la aplicación cliente sobre el evento, procesarla e insertarla en Google Calendar.
- Debe realizar comprobaciones periódicas para conocer si existe algún evento que vaya a ocurrir en ese instante de tiempo.
- En el momento en el que se inicie un evento, debe provocar que se desencadene la acción sobre el dispositivo de la vivienda inteligente.
- Los recursos de la capa de servicios REST no tienen que responder a todos los métodos HTTP

2.2. Cliente

- Facilita la inserción de eventos en un calendario, permitiendo especificar todos los campos que definen un evento.
- Un evento está compuesto por un título, una descripción, una fecha y hora de inicio y finalización, su periodicidad –si procede–, el dispositivo de la vivienda con el que interactuar y la acción a realizar sobre el mismo.
- La interacción con la capa de servicios REST se realiza mediante formularios, indicando la dirección del recurso.
- Debe permitir la inserción de nuevos dispositivos y acciones así como administrarlos.
- Debe permitir que el usuario inicie sesión con su cuenta de Google para poder visualizar sus calendarios y eventos personales.
- Debe tener un diseño adaptativo para que pueda ser utilizada desde un ordenador, teléfono móvil o tablet.

3. Análisis

Tras conocer los objetivos y los requisitos funcionales de las aplicaciones, se procede a realizar un estudio de la tecnología disponible dentro del contexto en el que se encuentra este proyecto y a decidir qué se va a utilizar para el desarrollo de las mismas y por qué.

Se va desarrollar dos tipos de aplicaciones bien distintas. Por un lado, se propone crear una aplicación servidor que permita la comunicación entre una vivienda inteligente y un sistema de calendarios externo a ella, como por ejemplo Google Calendar. Este servidor debe escuchar peticiones de creación de nuevos eventos, por lo que se busca una solución orientada a la web: implementar una capa de servicios REST.

La tecnología REST es un tipo de servicio web que permite el intercambio de mensajes sin estado, es decir, trata cada petición como una transacción independiente y que no tiene relación con cualquier solicitud anterior (Sandoval 2009, 7). Para acceder a un recurso se utiliza una URL que lo identifica y mediante métodos HTTP es posible consultarlo (GET), crearlo (POST), modificarlo (PUT) o eliminarlo (DELETE). Un sistema se considera RESTful si cumple con las siguientes características (Sandoval 2009, 7-8):

- Debe ser un sistema cliente-servidor
- Debe ser sin estado, es decir, el servidor no tiene que mantener la sesión del usuario
- Tiene soporte para cachear peticiones
- Cada recurso debe tener una única dirección y un punto de acceso válidos
- Debe implementarse en una capa para facilitar la escalabilidad

En la tabla siguiente, se pone un ejemplo de dos recursos. Se entiende por recurso cualquier elemento que pueda ser accedido y transferido entre clientes y servidores desde la Web. Se puede interactuar con el recurso usuarios mediante los métodos GET, POST, PUT y DELETE y con el recurso usuario mediante GET, PUT y DELETE.

Recurso	GET	POST	PUT	DELETE
URL_BASE = http://www.dominio.com	X	X	X	X
URL_BASE /usuarios	Lista una colección de usuarios y sus detalles	Crea un nuevo recurso usuario	Modifica toda la colección con una nueva	Elimina toda la colección
URL_BASE /usuarios/{id_usuario}	Obtiene la representación del usuario	No se utiliza, mejor PUT	Modifica el usuario y si no existe lo crea	Elimina el recurso de la colección

Tabla 1: Ejemplo de recurso

Generalmente se obtiene una representación del recurso que puede ser en dos formatos: XML o JSON, aunque también puede ser una imagen o cualquier otro tipo de contenido MIME válido.

3.1. Tecnologías

3.1.1. Google Calendar

Google Calendar es un servicio de agenda y calendario electrónico que ofrece la empresa Google a todos los usuarios que dispongan de una cuenta de correo GMail. Se trata de una interfaz web en la que podemos crear calendarios y añadir eventos en ellos, exportar el calendario en formato XML, iCal o HTML. Posee un sistema de notificaciones muy amplio ya que el usuario puede recibir las alertas con un mensaje de texto, por correo electrónico o también puede sincronizar su teléfono para recibir las notificaciones directamente en el dispositivo y conocer en todo momento los eventos que están por ocurrir.

Como se ha indicado en la introducción, Google Calendar dispone de un API REST de acceso para desarrolladores y facilita una serie de librerías en distintos lenguajes de programación para crear aplicaciones. En el momento de redactar el presente documento, el API está disponible en su versión 3 y con soporte para:

- **Java**
- JavaScript (beta)
- .NET
- Objective-C
- PHP (beta)
- Python

Por estar más familiarizado con el lenguaje de programación Java, se decide utilizar esta librería cliente que se incluirá en el desarrollo de la aplicación servidor.

Uno de los aspectos destacables de Google Calendar es la posibilidad de definir eventos periódicos mediante reglas de recurrencia. De esta forma, el usuario puede planificar eventos que ocurran con una frecuencia diaria, semanal, mensual o anual. Esta funcionalidad aumentaría la funcionalidad del calendario domótico, pudiendo generar eventos periódicos para los fines de semana o los meses de verano, por ejemplo.

➢ ***Autenticación y autorización con OAuth2***

Todas las APIs de Google utilizan el protocolo OAuth2 para la autenticación y autorización y se resume en 4 pasos (Google s.f.):

1. *Obtener unas credenciales de desarrollador*: Desde el panel de desarrolladores de Google (https://console.developers.google.com/) debemos registrar nuestra aplicación para obtener un identificador de cliente (*client ID*) y una contraseña (*client secret*) que incluiremos en nuestras aplicaciones.

2. *Obtener un código de acceso*: Esta petición es necesaria antes de acceder a la información privada del usuario. En ella se especifica un parámetro alcance (*scope*) que controla qué recursos y operaciones se le permiten al token de acceso al API. El usuario se autentica en nuestra aplicación con su cuenta de Google y a continuación se le comunica qué información

personal se va a consumir, para que pueda dar su consentimiento o no. Si acepta, el servidor de autorizaciones de Google envía a la aplicación un código de acceso para intercambiarlo por un token.

3. *Intercambiar el código de acceso por un token de acceso*: Una vez el usuario ha dado su consentimiento a la aplicación, se intercambia el código por un token de acceso al API especificada anteriormente en el alcance, de forma que solamente se puede acceder a esa información privada del usuario.

4. *Renovar el token de acceso*: Los token de acceso tienen un tiempo de vida limitado, por lo que es necesario renovarlos con un token de refresco (*refresh token*). Este token también se utiliza para poder acceder a la información privada del usuario en cualquier momento y sin la necesidad de que el usuario haya iniciado sesión en el servicio.

3.1.2. Restlet framework

Restlet es un *framework* de desarrollo de servicios REST en Java que permite crear aplicaciones cliente/servidor al mismo tiempo, es decir, puede consumir fácilmente recursos remotos usando un conector HTTP y también escuchar peticiones de clientes en un puerto y responderles con la representación del recurso solicitado.

Esta funcionalidad es de gran utilidad y facilitará no solo la implementación de la capa de servicios REST de la aplicación servidor, sino que también contribuirá a que la interacción con los recursos de la vivienda inteligente se realice de manera sencilla y transparente al usuario.

3.1.3. Infraestructura de vivienda inteligente

El modelo de vivienda inteligente proporcionado (Cors 2014) es una aplicación OSGi. Este *framework* para crear aplicaciones en Java se caracteriza por separar las aplicaciones en módulos independientes (*bundles*), de forma que se puedan iniciar, detener, instalar, desinstalar, etc.,

La siguiente imagen muestra la estructura de carpetas de la vivienda:

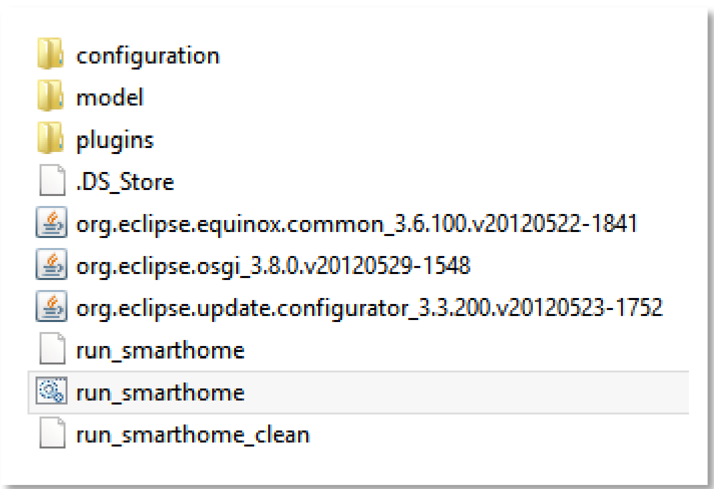

Ilustración 1: Estructura de carpetas de la infraestructura

Para ejecutar el programa basta con crear un archivo ejecutable .bat con el siguiente contenido (se requiere Java instalado para ejecutarlo):

```
#!/bin/bash
java -jar org.eclipse.osgi_3.8.0.v20120529-1548.jar -console
-consoleLog
```

Al lanzar la aplicación se ejecuta el siguiente comando para ver el estado de los componentes:

osgi> ss

```
C:\WINDOWS\system32\cmd.exe
ss
"Framework is launched."

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.8.0.v20120529-1548
1       ACTIVE      org.eclipse.equinox.common_3.6.100.v20120522-1841
2       ACTIVE      org.eclipse.update.configurator_3.3.200.v20120523-1752
3       ACTIVE      org.apache.felix.gogo.command_0.8.0.v201108120515
4       ACTIVE      org.apache.felix.gogo.runtime_0.8.0.v201108120515
5       ACTIVE      org.apache.felix.gogo.shell_0.8.0.v201110170705
6       ACTIVE      org.eclipse.core.contenttype_3.4.200.v20120523-2004
7       ACTIVE      org.eclipse.core.filesystem_1.3.200.v20120522-2012
8       ACTIVE      org.eclipse.core.jobs_3.5.200.v20120521-2346
9       ACTIVE      org.eclipse.core.runtime_3.8.0.v20120521-2346
10      ACTIVE      org.eclipse.emf.common_2.8.0.v20120606-0717
11      ACTIVE      org.eclipse.emf.ecore.xmi_2.8.0.v20120606-0717
12      ACTIVE      org.eclipse.emf.ecore_2.8.0.v20120606-0717
13      ACTIVE      org.eclipse.equinox.app_1.3.100.v20120522-1841
14      ACTIVE      org.eclipse.equinox.console_1.0.0.v20120522-1841
15      ACTIVE      org.eclipse.equinox.preferences_3.5.0.v20120522-1841
16      ACTIVE      org.eclipse.equinox.registry_3.5.200.v20120522-1841
17      ACTIVE      org.eclipse.equinox.util_1.0.400.v20120522-2049
18      ACTIVE      org.eclipse.equinox.wireadmin_1.0.0.v20080407
19      ACTIVE      org.eclipse.osgi.services_3.3.100.v20120522-1822
20      ACTIVE      org.eclipse.osgi.util_3.2.300.v20120522-1822
21      ACTIVE      SmartHomes.Model_1.0.0.201405252201
22      ACTIVE      SmartHomes.Server.Configuration_1.0.0.201405252201
23      ACTIVE      SmartHomes.Server_1.0.0.201405252206
osgi>
```

Ilustración 2: Servidor de la SmartHome

Como se puede apreciar, todos los módulos están activos y ya es posible interactuar con la vivienda. Esta consola debe permanecer abierta siempre, ya que de lo contrario terminaría el proceso servidor. Al terminar, basta con ejecutar:

osgi> exit
Really want to stop Equinox? (y/n; default=y) y

Para consultar el estado de los dispositivos se debe realizar peticiones HTTP desde un cliente al API REST del servidor, que escucha las peticiones en el puerto 8182 de la máquina dónde se ejecuta (en este caso localhost), utilizando una URL de la forma:

http://servidor:8182/devFunc/{id_devfunc}

Los únicos métodos HTTP permitidos por la aplicación son GET y PUT y siempre devuelven o utilizan una representación JSON. Por ejemplo, para consultar el estado de la *Device Functionality* que tiene por identificador "DF-CUINA.IL.AUXILIAR", se ejecuta una petición GET sobre la URL:

http://localhost:8182/devFunc/DF-CUINA.IL.AUXILIAR

Para cambiar su estado de encendida a apagada o viceversa, se realiza una petición PUT a la misma URL pero indicando en el *payload* la acción que se desea ejecutar y el valor de la misma siguiendo la sintaxis JSON:

{action: id_acción, value: valor}

Las acciones que se pueden realizar sobre un dispositivo dependen del tipo de recurso y el elemento value será opcional en función de la acción indicada. La siguiente tabla expone los distintos tipos de *Device Functionalities* de la vivienda, las acciones más relevantes y algunos ejemplos de dispositivo (Cors 2014):

Tipo de recurso	Acciones	Ejemplos
Todos los recursos	start: no requiere value. Arranca el recurso stop: no requiere value. Detiene el recurso remove: no requiere value. Elimina el recurso enable: no requiere value. Activa el recurso disable: no requiere value. Desactiva el recurso togEnable: cambia el estado de enable/disable (permuta)	
DeviceFunctionality: **Todas**	read: fuerza la lectura del estado/valor actual	
DeviceFunctionality: **Bistate** Representa dispositivos que pueden estar en dos estados	biaON: activa / enchufa / conecta biaOFF: desactiva / desenchufa / desconecta	DF-CUINA.NEVERA.EN (Enchufe de la nevera) DF-CUINA.CAFETERA.EN (Enchufe de la cafetera)

(activo/inactivo, conectado/desconectado, etc.)		
DeviceFunctionality: **ToggleBistate** Dispositivos de tipo bistate que permiten la conmutación del estado actual, funcionando como un interruptor	Permite las acciones de tipo bistate y además: biatoggle: permuta el estado (si estaba activo, lo desactiva, y viceversa)	Se aplica a la iluminación, climatización... DF-MENJ.CL.AC (Aire acondicionado del comedor) DF-CUINA.IL.BANCADA (Luz cocina encimera) DF-BC.CL.CALEFACTOR (Calefactor baño común)
DeviceFunctionality: **Dimmer** Es un bistate regulable, se puede especificar un valor.	diaSET%: establece valor (value) en porcentaje diaSETº: establece valor (value) en angular diaSETox: establece valor (value) en hexadecimal	Se aplica a luces graduables
DeviceFunctionality: **Movement** Dispositivo que se puede mover en dos sentidos y que se puede parar. El desplazamiento puede ser vertical (arriba / abajo / para), horizontal (izquierda / derecha / para) o angular (abre / cierra)	movaOPEN: abre/sube movaCLOSE: cierra/baja movaSTOP: para	Se designa a persianas, estores, puertas automáticas...
DeviceFunctionality: **Numeric Value** Representa un dispositivo que en el que se puede establecer su valor numérico	numaSET: establece un valor (value) a la funcionalidad	Normalmente se asocia a sensores (de viento, de lluvia, de luminosidad, de temperatura, etc.) DF-ESTACIO.METEO.TEMPERATURA (sensor de temperatura exterior) DF-ESTACIO.METEO.LUMINOSITAT (sensor de luminosidad exterior)

Tabla 2: Resumen de los tipos de dispositivos y acciones más relevantes

3.1.4. Base de datos

Google Calendar no permite especificar campos personalizados a un evento, por lo que sería difícil asignarle un dispositivo de la vivienda inteligente y la acción asociada. Una solución podría ser guardar esta información en el campo descripción del evento,

pero sería muy engorroso ya que habría que tratar las cadenas de texto y separarlas en dispositivo y acción.

Además, como se ha especificado en los objetivos y en los requisitos de este proyecto, la aplicación cliente debe ser capaz de introducir nuevos dispositivos y administrarlos, en caso de que se instalen nuevas unidades en la vivienda inteligente.

Por tanto, se considera necesaria la inclusión de una base de datos MySQL. La aplicación servidor insertará el evento tanto en el calendario de Google Calendar como en la base de datos y la aplicación cliente será capaz de realizar operaciones CRUD (create, read, update, delete) sobre dispositivos y acciones.

3.1.5. Yii framework

Como acabamos de comentar, la aplicación web cliente debe facilitar un panel de administración que permita crear y modificar nuevos dispositivos y acciones. Dentro del amplio abanico de herramientas de creación de webs (Wordpress, Joomla, Drupal, CakePHP, Django, Symfony…) se decide utilizar Yii.

Yii es un *framework* PHP para el desarrollo de aplicaciones web siguiendo el patrón MVC (modelo, vista, controlador). Dispone de una herramienta de generación de código muy sencilla que lee la tabla indicada de la base de datos para crear un modelo y a partir de éste, genera el controlador y las vistas, que permiten operaciones CRUD sobre el modelo. También dota a las aplicaciones de un sistema de autenticación completo, que puede estar basado en roles. Posee una gran comunidad de desarrolladores y varios *plugins* para incrementar la funcionalidad de las aplicaciones web.

3.1.6. AJAX, Bootstrap y jQuery

Uno de los requisitos de la aplicación cliente es que tenga un diseño adaptativo y funcione en cualquier dispositivo. El usuario debe poder navegar por la web tanto desde un ordenador como desde un *smartphone* o *tablet*.

Por ello, se decide que la maquetación de la web se realizará con Bootstrap, un *framework* HTML, CSS y JavaScript para desarrollar de forma rápida interfaces web. Se caracteriza por estar enfocado en el diseño adaptativo, siendo los dispositivos móviles su prioridad.

También se incluirá en la aplicación cliente la librería jQuery. Esta librería JavaScript es muy utilizada en el desarrollo de páginas web ya que permite modificar elementos del documento HTML de forma sencilla mediante el uso de selectores. Facilita también el uso de llamadas AJAX (Asynchronous Javascript And XML) que realizaremos para interactuar con el API REST del servidor. Estas llamadas asíncronas solicitan la información al servidor y se puede seguir trabajando en la web sin preocuparse por la respuesta.

3.2. Interacción

Así pues, tras realizar este análisis de las tecnologías disponibles y después de seleccionar aquellas que facilitan y se aproximan más a los objetivos que persigue este proyecto; se presenta a continuación un diagrama resumen de todo el proceso de interacción entre los distintos servicios y elementos involucrados.

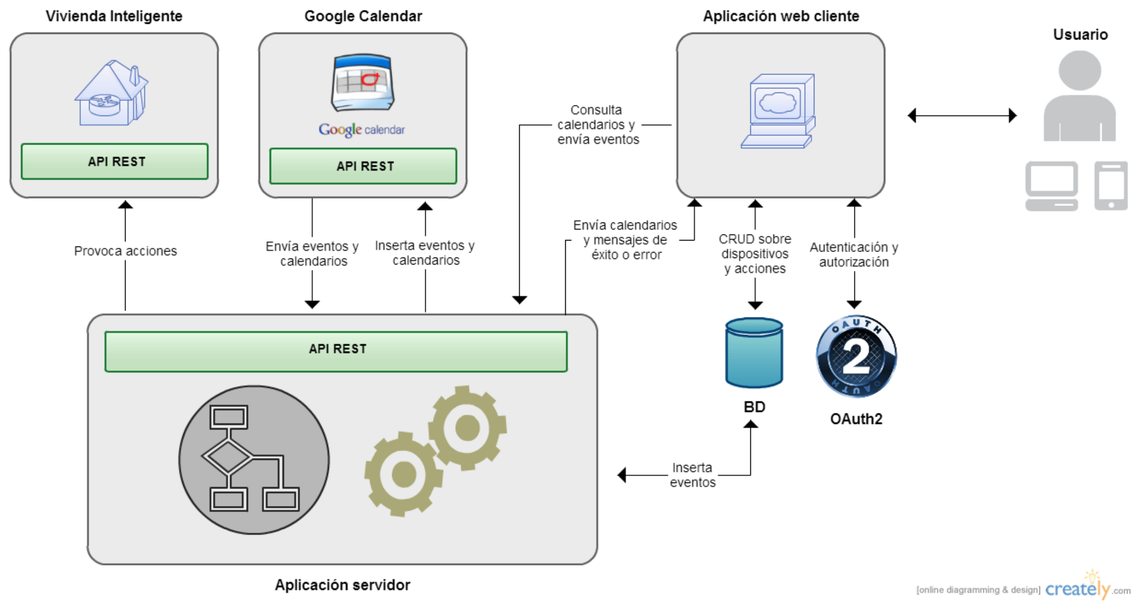

Ilustración 3: Diagrama de interacción

4. Diseño

Una vez finalizado el análisis de los elementos involucrados en el proyecto, se procede a la fase de diseño. En este apartado, se detalla el diseño de las aplicaciones servidor y cliente y de la base de datos.

4.1. Servidor

4.1.1. Arquitectura en capas

Una de las recomendaciones en el desarrollo de *software* es dividir el código en distintas capas o paquetes para mejorar la abstracción entre clases. Se opta por esta filosofía de diseño y se diferencian cinco capas:

- **application**: contendrá la clase con el main principal y otra que se encargue de dirigir las peticiones REST a los recursos correspondientes.
- **dal**: es la capa de acceso a la información almacenada en la base de datos.
- **domain**: alberga los objetos que se utilizarán en la aplicación.

- **oauth**: contendrá las clases encargadas del acceso al API de Google Calendar.

- **resources**: la implementación de todos los recursos que ofrezca el API REST compondrán esta capa.

4.1.2. Recursos REST disponibles en el API

Como se ha explicado en el apartado 3.1.2. Restlet framework, los recursos ofrecidos por un servicio REST se consumen y consultan usando una URL.

Un de las buenas prácticas a la hora de diseñar un API REST es utilizar en la URL el nombre del recurso a obtener en plural (Sahni 2013). De esta forma, podemos interactuar con el recurso mediante métodos HTTP en un solo *endpoint*.

El API REST que implementará la aplicación servidor ofrecerá los siguientes recursos:

- **Calendarios** de la cuenta de Google Calendar del usuario. Utilizará representación en formato JSON y aceptará métodos GET, POST, OPTIONS y DELETE para poder recuperar, crear y borrar calendarios, respectivamente.

- **Eventos** de la cuenta de Google Calendar del usuario pero con la información del dispositivo involucrado y la acción a realizar. Utilizará representación en formato JSON y aceptará métodos GET, POST y OPTIONS para poder recuperar y crear eventos. Cuando se invoca mediante POST, insertará el evento tanto en Google Calendar como en la base de datos.

En un primer momento, se desarrolló el proceso de autorización y autenticación con OAuth2 en el API REST pero por motivos de funcionalidad y usabilidad relacionados con la filosofía REST que se explicarán en el apartado 5.1.2. “Registrar la aplicación en Google Developer Console”, se decide trasladar este proceso a la aplicación cliente.

4.2. Base de datos

Debido a la necesidad de disponer de un panel de control web que sea capaz de trabajar con nuevos dispositivos que en un supuesto caso fueran adquiridos, se decide crear una base de datos que albergue todos los dispositivos. La vivienda inteligente cuenta con cerca de 400 funcionalidades, por lo que solo se insertarán alrededor de 15 dispositivos relevantes en el ámbito de funcionalidades programables, por ejemplo, relacionadas con la iluminación, las persianas o la climatización, descartando sensores o botoneras.

Esta base de datos estará compuesta por 3 tablas:

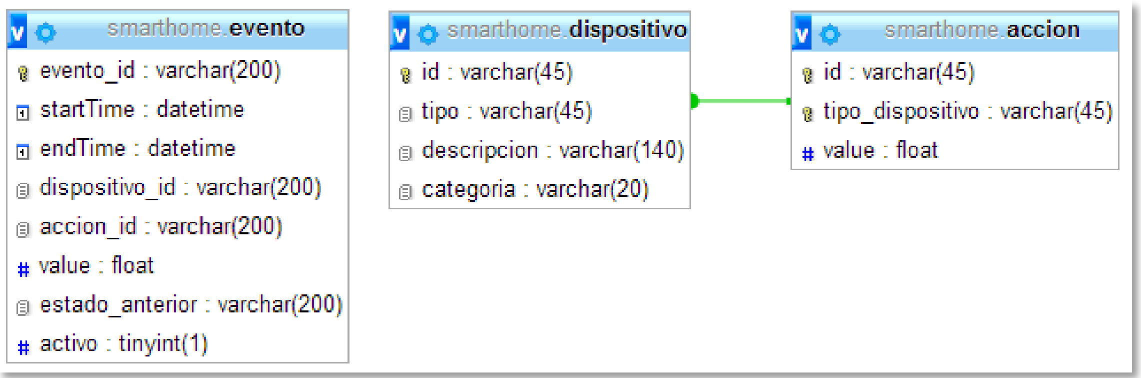

Ilustración 4: Tablas de la base de datos

La tabla evento contiene información relacionada con el evento creado desde el panel de control. Se conoce la fecha y hora de inicio y fin del evento, el dispositivo involucrado y la acción asociada, el valor de la misma (opcional según la acción), el estado anterior en el que se encontraba el dispositivo (por defecto “UNKNOWN”) y si ese evento está ocurriendo o no (por defecto, un evento siempre estará no activo). El API REST recibe en el *endpoint* “/eventos”, se crea el evento en Google Calendar y después se inserta en la base de datos utilizando el mismo identificador, que será la clave primaria de la tabla.

La tabla dispositivo hace referencia a un dispositivo de la vivienda inteligente. Como clave primaria se utiliza el nombre de la funcionalidad y como clave ajena tipo (*bistate*, *togglebistate*, *dimmer*, *movement* o *numeric value*), que enlazará con el campo “tipo_dispositivo” de la tabla acciones. De esta forma, cada vez que se inserte un dispositivo nuevo se le asignarán las acciones correspondientes en función de su tipo. También se conoce una pequeña descripción y la categoría a la que pertenece. Este último campo es una decisión personal que permite clasificar el dispositivo según su funcionalidad dentro de la vivienda inteligente, por ejemplo, una luz entraría dentro de la categoría “iluminación”, la calefacción y el aire acondicionado pertenecerían a la categoría “climatización”; y las persianas y estores a la categoría “ventanas”.

Por último, la tabla acción representa todas las acciones que se pueden realizar sobre un tipo de dispositivo determinado. La clave primaria es compuesta, de forma que dispositivos de distinto tipo pueden tener acciones comunes y diferentes. Por ejemplo, un dispositivo *bistate* puede realizar las acciones “biaON” y “biaOFF”, mientras que un dispositivo *togglebistate* puede realizar, además de estas dos, la acción biaTOGGLE. El campo *value* sigue siendo opcional.

4.3. Aplicación web cliente

Acto seguido, se procede a explicar el diseño de las interfaces gráficas que compondrán la aplicación web cliente. Se distinguen los siguientes escenarios:

- **Identificación**

  En esta página, el usuario puede iniciar sesión en el sistema con su cuenta de Google para crear nuevos eventos o puede acceder al panel de administración de dispositivos y acciones identificándose como un usuario administrador.

Ilustración 5: Diseño de la interfaz de inicio de sesión

Ilustración 6: Diseño de la interfaz móvil de inicio de sesión

- **Gestión de calendarios**

Desde esta pantalla y una vez identificado, el usuario puede añadir un nuevo calendario en su cuenta de Google Calendar si lo desea. El campo descripción no es obligatorio. Mediante una lista desplegable con todos los calendarios disponibles, el usuario puede seleccionar uno y eliminarlo.

Estas operaciones de creación y eliminación se realizarán a través de la capa de servicios REST de la aplicación servidor, quien creará o eliminará el calendario de la cuenta de Google Calendar del usuario.

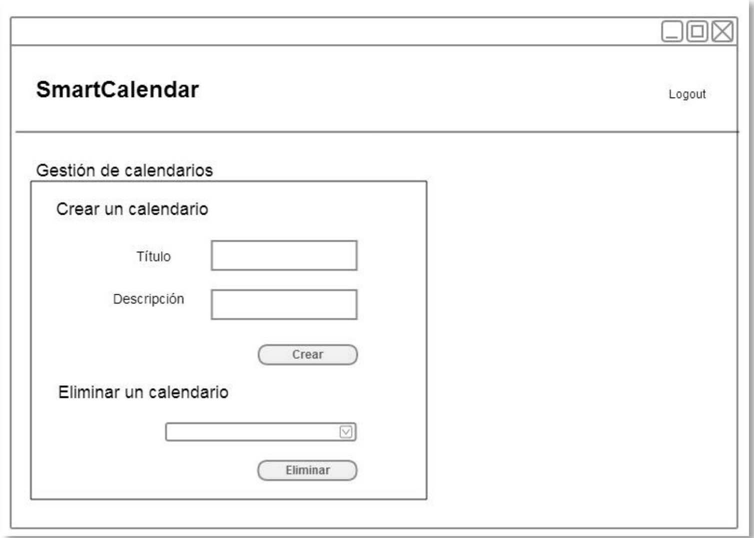

Ilustración 7: Diseño de la interfaz de gestión de calendarios

Ilustración 8: Diseño de la interfaz móvil de gestión de calendarios

- **Añadir un evento**

En este escenario el usuario se dispone a insertar un nuevo evento en el calendario seleccionado para la vivienda inteligente. La interfaz está dividida en dos paneles en la versión escritorio y una única columna en la versión móvil. El formulario de creación de eventos se sitúa en el panel izquierdo mientras que el calendario seleccionado ocupa el panel derecho.

Después de elegir un calendario con el que trabajar, el usuario debe indicar un título para el evento, siendo la descripción opcional. A continuación, especifica la duración del evento y su periodicidad –si procede– diaria o semanal, indicando para esta qué días de la semana se activará el evento.

Por último, es posible filtrar los dispositivos por el campo categoría, comentado anteriormente en el diseño de la base de datos y elegir la acción deseada mediante un menú desplegable. Para finalizar, se indica el valor asociado a la acción y se hace clic en el botón “crear”.

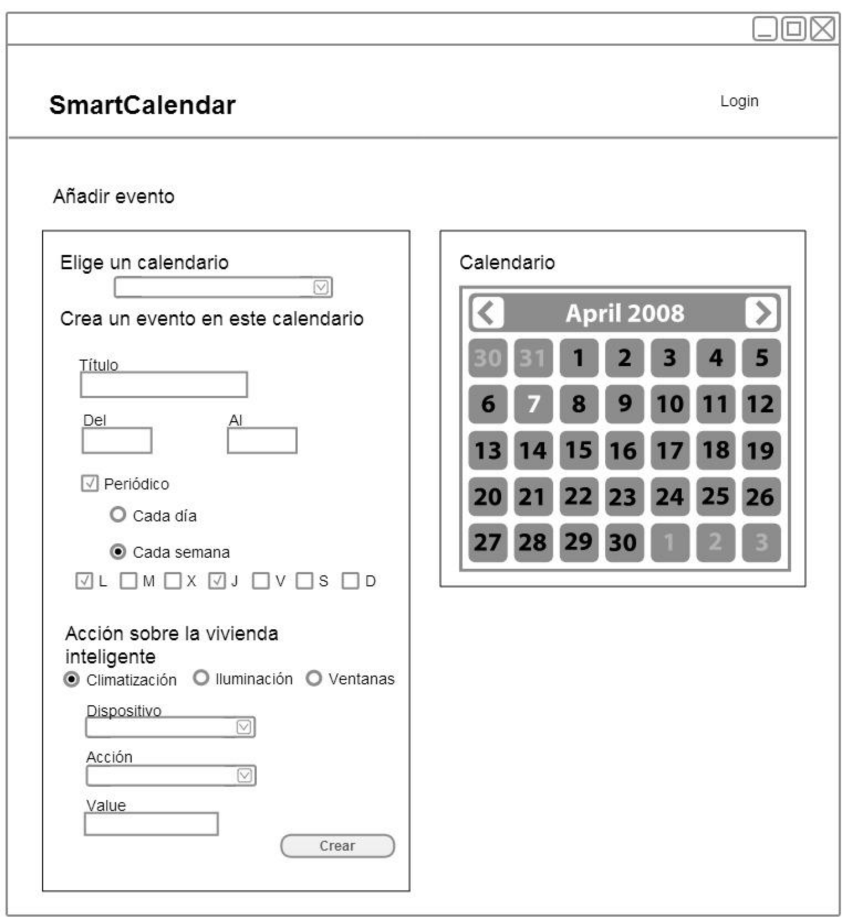

Ilustración 9: Diseño de la interfaz de añadir eventos

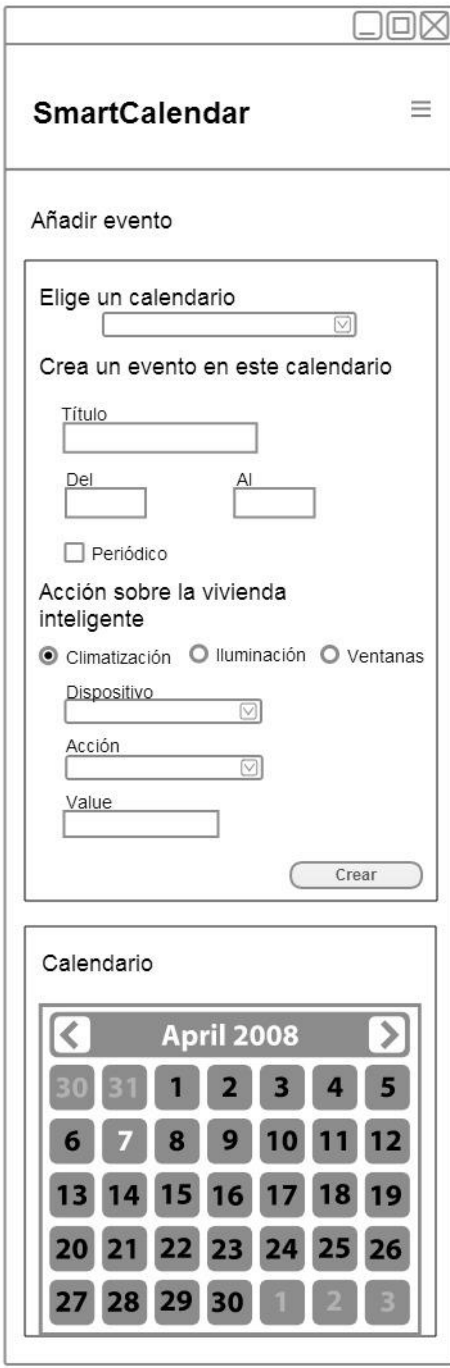

Ilustración 10: Diseño de la interfaz de añadir eventos

5. Implementación

En este apartado se aborda la implementación de las dos aplicaciones que componen este proyecto. Primero, se detalla el proceso de desarrollo de la aplicación servidor y después, una vez establecido como interactuar con ella, se explica la creación de la aplicación cliente.

5.1. Preparación previa

En este apartado se introducen unas pautas previas antes de explicar el desarrollo de cada una de las aplicaciones que componen este proyecto. Son aspectos relativos a la descarga de las librerías necesarias y a la configuración de las herramientas a utilizar.

5.1.1. Librerías cliente del API de Google Calendar

El primer paso es descargar desde el sitio web de la API de Google Calendar la librería cliente para desarrollar aplicaciones. Se incluirá en la aplicación servidor junto a las demás librerías necesarias y permitirá crear los objetos que conectarán con los servicios que ofrece Google Calendar. En la fase de análisis se decidió implementar la aplicación en el lenguaje de programación Java, de modo que descargaremos esta librería.

5.1.2. Registrar la aplicación en Google Developer Console.

Para acceder a cualquiera de las APIs de Google desde una aplicación, es necesario solicitar unas credenciales de desarrollador. El sitio web Google Developer Console es un panel de administración que permite dar de alta un proyecto y ofrece una serie de servicios entre los que destacan un repositorio de código con control de versiones, la obtención de credenciales para las aplicaciones y la posibilidad de definir a qué API o APIs se tendrá acceso, que en este caso será el API de Google Calendar con una limitación de 200.000 peticiones al día.

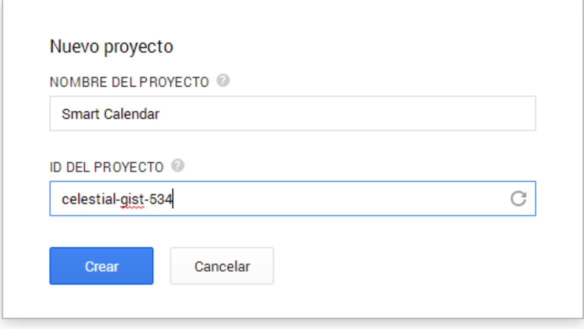

Ilustración 11: Creación de un nuevo proyecto en Google Developer Console

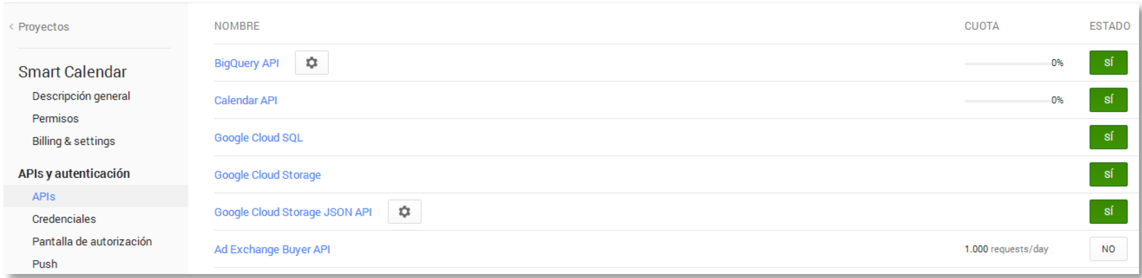

Ilustración 12: Habilitar el acceso al API de Google Calendar

Para generar las credenciales de la aplicación hay que acceder al apartado "Credenciales" e indicar el tipo de aplicación que se va a desarrollar y la URI de redireccionamiento. La aplicación servidor va a comportarse como una aplicación web, el API será accesible mediante una URL. La URI de redireccionamiento es la URL a la que el servidor de autorizaciones de Google envía el código de acceso una vez el usuario da el consentimiento para que la aplicación pueda acceder a su información privada.

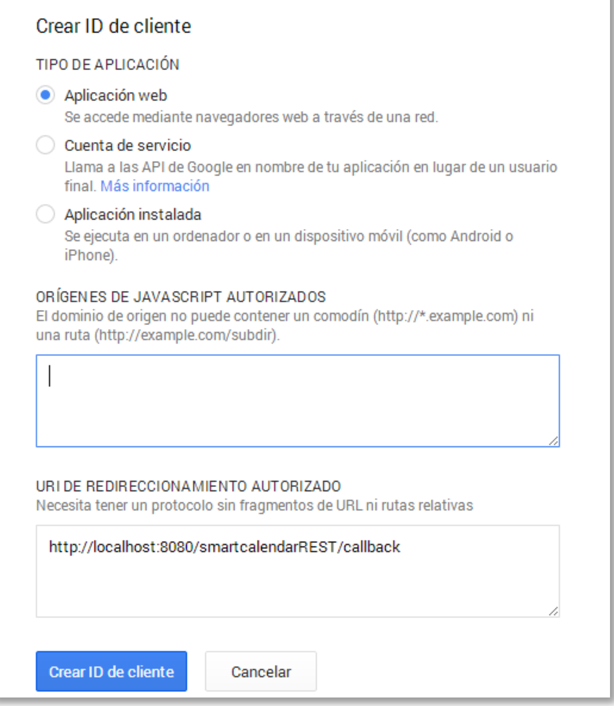

Ilustración 13: Creación de las credenciales para la aplicación servidor

Una vez creadas las credenciales, hacemos clic en el botón "Descargar JSON" para incluirlas en nuestra aplicación mediante un fichero JSON (por motivos de seguridad se ha ocultado parte del secreto de la credencial).

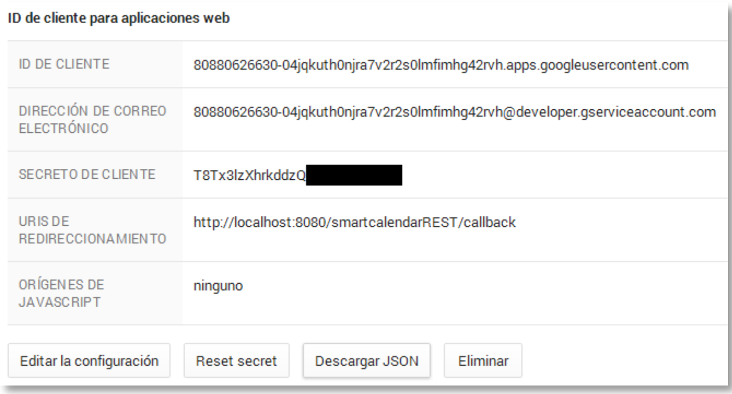

Ilustración 14: Resumen de las credenciales

Como se indicó en el apartado de diseño, en un primer momento se decidió que se implementara en el API REST la autenticación y autorización del usuario mediante el protocolo OAuth2. Se crearon dos *endpoints* en el API, "/login" y "/callback", para experimentar con el acceso a Google Calendar y estudiar su comportamiento. En el recurso "/login" se redirigía al usuario a una página de Google para que iniciara sesión y autorizara a la aplicación y, si aceptaba, se recibía el código de acceso en el recurso "/callback".

Sin embargo, como se ha explicado en la fase de análisis, un servicio REST se caracteriza por utilizar peticiones sin estado, por tanto la autenticación no debería depender de cookies o sesiones (Sahni 2013). Se optó entonces por realizar la autenticación y autorización en la aplicación cliente y enviar el token de acceso en cada petición REST al API servidor utilizando la cabecera Authorization de HTTP. Este proceso se detalla en el apartado 5.2.3. "Recursos del API", en el punto "Uso de la cabecera HTTP Authorization".

5.1.3. Descargar Restlet framework

Como se vio en la fase de análisis, la capa de servicios REST se implementará utilizando el *framework* Restlet. Desde su sitio web podemos descargar las librerías Java necesarias para comenzar a desarrollar la capa de servicios REST de la aplicación servidor.

5.1.4. Herramientas de desarrollo

Se ha utilizado el programa Eclipse Kepler como entorno de desarrollo tanto para la aplicación servidor como para la aplicación cliente. Creamos un proyecto web dinámico nuevo, de forma que la aplicación esté preparada para desplegarse en un contenedor de aplicaciones si fuera necesario, como por ejemplo Tomcat. En la carpeta WebContent\WEB-INF del proyecto, se han creado dos carpetas más:

- **classes**: contiene un fichero JSON llamado "client_secrets.json", que es el archivo anteriormente descargado con las credenciales de la aplicación.
- **lib**: contiene todas las librerías que necesita la aplicación para funcionar

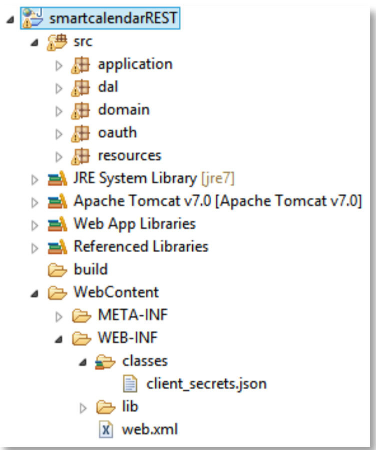

Ilustración 15: Estructura de la aplicación servidor

La aplicación web cliente requiere un servidor web y una base de datos para funcionar, por lo que se utiliza la herramienta XAMPP para instalar Apache, un servidor HTTP con soporte para páginas web hechas en el leguaje PHP; y MySQL para la base de datos.

5.2. Servidor

Tras esta configuración previa necesaria, se procede a explicar el desarrollo de la aplicación servidor.

5.2.1. Credencial de acceso mediante el protocolo OAuth2

En la fase de diseño se optó por una arquitectura en capas separadas según la funcionalidad de las clases. La clase del paquete “oauth” es la que gestionará las acciones relacionadas con la obtención de una credencial de acceso al API de Google Calendar utilizando el protocolo OAuth2.

En primer lugar, el usuario se autentica mediante la aplicación cliente y se le redirige a la pantalla de consentimiento en la que decide si autoriza a la aplicación para que acceda a su información privada. Si el usuario permite el acceso, se devuelve al usuario a la aplicación con un código de acceso. Este código de acceso se vuelve a enviar a los servidores de autenticación de Google para intercambiarlo por un token de acceso al API de Google Calendar. Este token de acceso se enviará en cada petición REST que se realice a la aplicación servidor, siendo esta la que se encargue de solicitar la información requerida a Google Calendar usando el token de acceso.

En el siguiente diagrama se muestra este proceso:

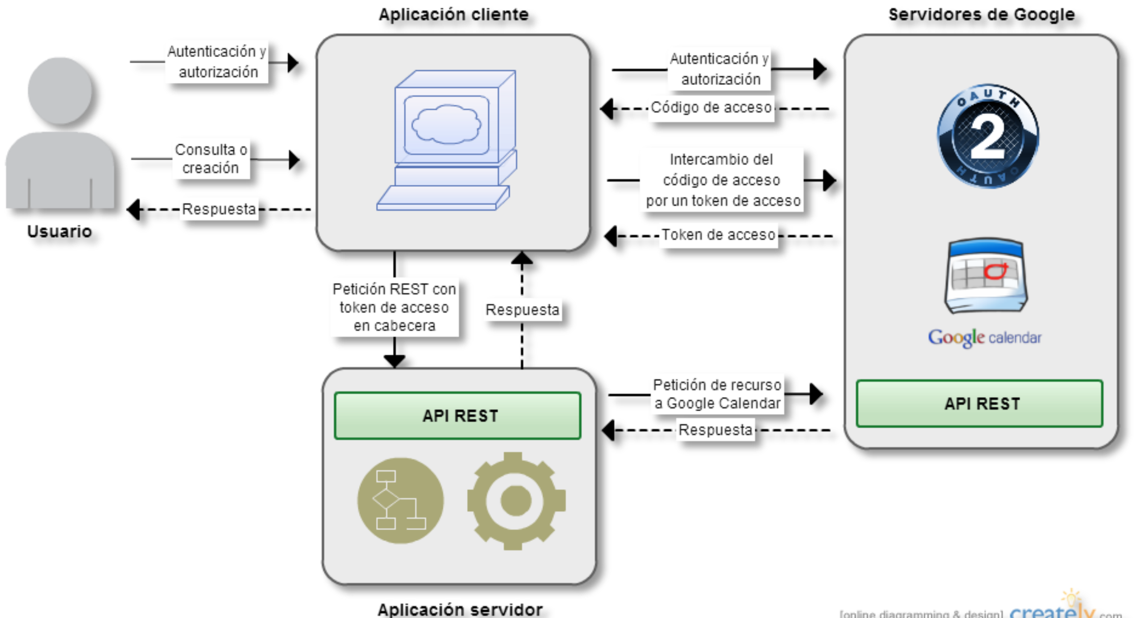

Ilustración 16: Diagrama de autenticación, autorización e interacción con Google Calendar mediante OAuth2

Una vez decidido que la autenticación y la autorización se realizarán en la aplicación cliente y que será esta la que envíe el token de acceso en cada petición REST mediante una cabecera HTTP Authorization, se acomete la tarea de interactuar desde la aplicación servidor con los servidores de Google utilizando el token de acceso recibido.

Para ello, se crea dentro del paquete "oauth" la clase OAuth.java. Esta clase está compuesta por:

- **Declaración de variables y constructor:**

  HTTP_TRANSPORT es un objeto de tipo HttpTransport que permite conectar las aplicaciones con la capa de transporte de HTTP.

  La variable *JSON_FACTORY* define cómo se construyen las instancias de un elemento en formato JSON.

  clientSecrets será el atributo que contenga las credenciales de la aplicación y *token* representa el token de acceso al API de Google Calendar.

```
private static final HttpTransport HTTP_TRANSPORT = new NetHttpTransport();
private static final JsonFactory JSON_FACTORY = JacksonFactory.getDefaultInstance();

private static GoogleClientSecrets clientSecrets;
private static GoogleTokenResponse token;

public OAuth(){ token=null;}
```

- **Método *getCredential(TokenResponse token)***

  Este método es el encargado de realizar el intercambio del token de acceso por una credencial que permita el acceso a Google Calendar. Recibe como parámetro el token, llama al método que crea la credencial y la devuelve.

```
public static GoogleCredential getCredential(TokenResponse token)
        throws IOException, GeneralSecurityException{

        //Intercambio token por credencial
        GoogleCredential credential = createCredentialWithToken(token);
```

```
        return credential;
}
```

- **Método *createCredentialWithToken (TokenResponse tokenResponse)***

Este método, recibe el token de acceso, carga las credenciales de desarrollo de la aplicación y las utiliza para crear un objeto GoogleCredential, que se devuelve a la función anterior. Para construirlo se necesita el método de transporte, cómo se definen las instancias JSON, las credenciales de la aplicación y el token.

```
public static GoogleCredential createCredentialWithToken(TokenResponse tokenResponse) {
        System.out.println("Creando credencial...");

        //Cargamos las credenciales de desarrollador
        if (getClientSecrets()==null){
                setClientSecrets(loadClientSecrets());
        }

        return new GoogleCredential.Builder().setTransport(HTTP_TRANSPORT)
                .setJsonFactory(JSON_FACTORY)
                .setClientSecrets(getClientSecrets())
                .build().setFromTokenResponse(tokenResponse);
}
```

- **Método *loadClientSecrets()***

Se encarga de leer el fichero *client_secrets.json* descargado en la fase de preparación previa y crea un objeto GoogleClientSecrets con esta información.

```
public static GoogleClientSecrets loadClientSecrets()throws IOException {
        System.out.println("Cargando secretos del desarrollador...");
        return GoogleClientSecrets.load(JSON_FACTORY, new InputStreamReader(
                OAuth.class.getResourceAsStream("/client_secrets.json")));
}
```

- **Métodos *set(...)* y *get()***

Permiten el acceso a los atributos de la clase y consultar o cambiar su valor

```
public static GoogleTokenResponse getToken() {
        return token;
}
public static void setToken(GoogleTokenResponse token) {
        OAuth.token = token;
}
public static GoogleClientSecrets getClientSecrets() {
        return clientSecrets;
}
public static void setClientSecrets(GoogleClientSecrets clientSecrets) {
        OAuth.clientSecrets = clientSecrets;
}
```

5.2.2. Enrutamiento de peticiones REST

En este apartado se aborda cómo utilizar el framework Restlet para definir la asociación entre las peticiones REST y los recursos solicitados. Restlet permite especificar una clase que realice esta vinculación, que extenderá de la clase Application. Escuchará las peticiones y las dirigirá a las clases encargadas de la implementación de los recursos, que extenderán de la clase ServerResource.

Cada recurso puede disponer de varios métodos que responden en función del tipo de llamada HTTP. Gracias a las anotaciones encima de la cabecera de las funciones que introduce Restlet, se pueden ejecutar métodos distintos si la petición se realiza por GET

o por POST, por ejemplo. Cuando una petición se realiza por POST, se está enviando información al servidor y esta información se recibe en un objeto de tipo *Representation*. Además, también es posible indicar diferentes formatos para la representación de un recurso en función de las cabeceras Accept y Content-Type del protocolo HTTP que lleve la petición, pero en esta API solo se utilizará formato JSON en todas las consultas.

Como hemos comentado en la fase de diseño, la capa de servicios REST de esta aplicación se compone de dos recursos bien diferenciados: calendarios y eventos. Para realizar acciones sobre estos recursos se utilizan URLs de acceso, la aplicación mapea cada petición y la redirige a la clase que implementa el recurso solicitado. Por tanto, lo primero será definir la URL base de acceso al API y qué recursos estarán vinculados a las URLs. En el paquete "application" de la aplicación, se encuentran las clases PrincipalServer.java y SmartCalendarApplication.java.

- **PrincipalServer.java**

Esta clase contiene el método main principal de la aplicación. Consiste en crear un servidor HTTP que atienda peticiones en el puerto 8080. En este punto, definimos que la clase SmartCalendarApplication va a gestionar todas las llamadas a la URL base de acceso al API, que será:

http://localhost:8080/smartcalendarREST

La clase SmartCalendarApplication realizará el mapeo de las peticiones.

```
package application;

import org.restlet.Component;
import org.restlet.data.Protocol;

public class PrincipalServer {
    public static void main(String[] args) throws Exception {
        // Crea un nuevo componente
        Component component = new Component();

        // Añade un servidor HTTP escuchando en el puerto 8080
        component.getServers().add(Protocol.HTTP, 8080);

        // Vincula el punto de entrada al API con la clase SmartCalendarApplication
        component.getDefaultHost().attach("/smartcalendarREST",
                    new SmartCalendarApplication());

        // Inicializa el componente
        component.start();
    }
}
```

- **SmartCalendarApplication.java**

El API REST ofrece calendarios y eventos que serán accesibles desde las distintas URLs:

- Calendarios:
  http://localhost:8080/smartcalendarREST/calendarios
- Calendario específico:
  http://localhost:8080/smartcalendarREST/calendarios/{idCalendario}
- Eventos de un calendario:
  http://localhost:8080/smartcalendarREST/calendarios/{idCalendario}/eventos

Esta clase es un componente Restlet que actúa como un enrutador. Es la encargada de redirigir la petición REST que siga alguno de los patrones anteriores a la

clase que gestiona el recurso solicitado. Por tanto, se crean tres nuevas clases en el paquete “resources” que contendrán la implementación de cada recurso:

- CalendariosResource.java
- CalendarioResource.java
- EventosResource.java

```
public class SmartCalendarApplication extends Application {
	@Override
	public Restlet createInboundRoot() {

		Router router = new Router(getContext());
		//CALENDARIOS
		router.attach("/calendarios", CalendariosResource.class);

		//CALENDARIO POR ID
		router.attach("/calendarios/{calendarId}",
		CalendarioResource.class);

		//EVENTOS DEL CALENDARIO
		router.attach("/calendarios/{calendarId}/eventos",
		EventosResource.class);

		/*LOGIN y CALLBACK - NO UTILIZADOS
		router.attach("/login", OAuthResource.class);
		router.attach("/callback", CallbackResource.class);*/

		return router;
	}
}
```

5.2.3. Recursos del API

Se procede a continuación con la implementación de los recursos que ofrecerá la capa de servicios REST de la aplicación. Todos estos recursos están implementados en las clases del paquete “resources”.

➢ ***Uso de la cabecera HTTP Authorization***

El protocolo HTTP proporciona un método de autenticación utilizando la cabecera Authorization. Se utiliza para dotar de seguridad a un recurso web y restringir el acceso solo a las peticiones autenticadas. La forma de autenticarse puede ser el envío de un usuario y contraseña cifrados o en este caso la aplicación cliente enviará el token de acceso. De esta forma, se consigue que cada petición REST a la capa de servicios del servidor sea una petición sin estado, sin necesidad de utilizar sesiones.

Para extraer la información de las cabeceras HTTP se utilizará el método *getMessageHeaders(Message message)* facilitado en el foro de Restlet (Peierls 2012) y que se incluye en el fichero Utils.java creado para albergar funciones de ayuda.

```
package resources;

import java.util.concurrent.ConcurrentMap;

import org.restlet.Message;
import org.restlet.engine.header.Header;
import org.restlet.util.Series;

public class Utils {

	private static final String HEADERS_KEY = "org.restlet.http.headers";
```

```
        @SuppressWarnings("unchecked")
        static Series<Header> getMessageHeaders(Message message) {
                ConcurrentMap<String, Object> attrs = message.getAttributes();
                Series<Header> headers = (Series<Header>) attrs.get(HEADERS_KEY);
                if (headers == null) {
                        headers = new Series<Header>(Header.class);
                        Series<Header> prev = (Series<Header>)
                        attrs.putIfAbsent(HEADERS_KEY, headers);
                        if (prev != null) { headers = prev; }
                }
        return headers;
        }
}
```

Será necesario que en cada recurso que ofrezca la capa de servicios REST se realice el siguiente proceso para obtener el token de acceso de la cabecera de la petición y con él generar una credencial para interactuar con el API de Google Calendar.

El primer paso es recuperar el token de acceso de la cabecera de la petición haciendo uso de la función *getMessageHeaders(Message message)*, pasándole como argumento la petición que llega al recurso. El método *getRequest()* facilitado por la librería Restlet permite recoger la petición. Con el método *getValuesMap()* se crea un objeto de tipo *Map* en el que cada clave es una cabecera y su valor es el valor que tenga la cabecera.

Una vez conseguido el token, el siguiente paso es intercambiarlo por una credencial de acceso al API de Google Calendar. Para ello, se crea un objeto *TokenResponse*, se le asigna el token de acceso recibido en la cabecera y se pasa al método de la clase OAuth *getCredential(TokenResponse token)*, visto anteriormente en el apartado 5.2.1. De esta forma, con la credencial recibida se puede tener acceso al servicio de Google Calendar.

```
//Obtención de las cabeceras de la petición
Map<String, String> headers = Utils.getMessageHeaders(getRequest()).getValuesMap();

//Guardamos el token de acceso que viene en la cabecera
String token = headers.get("Authorization");

//Creación de un objeto TokenResponse y le asignamos el token recibido
TokenResponse GoogleToken = new TokenResponse();
GoogleToken.setAccessToken(token);

//Intercambio del token por una credencial
GoogleCredential credential = OAuth.getCredential(GoogleToken);
```

Este proceso de obtención del *token* de acceso y de la credencial se repite en cada recurso, referenciándose en los campos de código como `/**Proceso de obtención del token de acceso y su posterior intercambio por una credencial de acceso de tipo GoogleCredential**/`

- ***CalendariosResource***

Este recurso permite obtener una representación simplificada de los calendarios en Google Calendar del usuario. De toda la información que ofrece Google Calendar sobre un calendario (tipo, etag, id, título, descripción, localización y zona horaria) (Google s.f.), para el propósito del proyecto es suficiente con conocer el identificador, el título y la descripción. Acceder a este recurso es necesario para que el usuario sea capaz de crear o consultar los calendarios de su cuenta para poder elegir al que pertenecerán los eventos programados para la vivienda inteligente. Se procede a explicar cómo reacciona la clase CalendariosResource.java en función del método HTTP utilizado en la petición.

```
public class CalendariosResource extends ServerResource{

	private static final HttpTransport HTTP_TRANSPORT = new NetHttpTransport();
	private static final JsonFactory JSON_FACTORY =
JacksonFactory.getDefaultInstance();
	private static final String APPLICATION_NAME = "SmartCalendar";

	@Get("json")
	public StringRepresentation getJSON() throws IOException,
			GeneralSecurityException, JSONException{
	//IMPLEMENTACIÓN
	return new StringRepresentation(obj.toString(), MediaType.APPLICATION_JSON);
	}

	@Post("json")
	public Representation creaCalendario(Representation entity) throws IOException,
			GeneralSecurityException{
	//IMPLEMENTACIÓN
	return new StringRepresentation(createdCalendar.getId(), MediaType.TEXT_HTML);
	}

	@Options
	public void options(Representation entity) {
	//IMPLEMENTACIÓN
	}
}
```

- GET

Si la petición de este recurso se realiza mediante el método HTTP GET, se devuelve un array JSON con todos los calendarios del usuario, donde cada elemento es un calendario representado por un objeto JSON. En cada representación se conoce el identificador, el título y la descripción del calendario. Gracias a la anotación @Get("json") de Restlet se especifica que este método se activará cuando la petición HTTP sea mediante GET y el cliente espere recibir la respuesta en formato JSON.

Una vez obtenida la credencial, se usa para crear un servicio *Calendar* que permite utilizar todas las funcionalidades del API de Google Calendar. Mediante el método *calendarList()* se obtiene el recurso *CalendarList* que consiste en una lista de todos los calendarios del usuario. Se recorre dicha lista creando el objeto JSON que será el *array* de calendarios que finalmente devuelve el método como representación del recurso.

```
@Get("json")
public StringRepresentation getJSON() throws IOException,
	GeneralSecurityException, JSONException{

	/**Proceso de obtención del token de acceso y su posterior intercambio por una
	credencial de acceso de tipo GoogleCredential**/

	//Con la credencial ya podemos crear un servicio Calendar
	Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,JSON_FACTORY,
		credential).setApplicationName(APPLICATION_NAME).build();

	//Recuperamos una lista con todos los calendarios del usuario
	CalendarList feed = calendario.calendarList().list().execute();

	//Creación de la respuesta JSON
	JSONObject obj = new JSONObject();
	JSONArray lista = new JSONArray();

	for (CalendarListEntry entry : feed.getItems()) {

		JSONObject cal = new JSONObject();
		cal.put("id", entry.getId());
		cal.put("summary", entry.getSummary());
		cal.put("description", entry.getDescription());
		lista.put(cal);
	}
```

```
        obj.put("calendarios", lista);
        return new StringRepresentation(obj.toString(), MediaType.APPLICATION_JSON);
}
```

- POST

Si se realiza una petición POST a un recurso /calendarios se creará un nuevo calendario en la cuenta de Google Calendar del usuario. El *payload* o cuerpo de la petición debe ser en formato JSON y el único campo necesario que contendrá será el título que se le quiera dar al nuevo calendario, ya que el identificador se genera al insertar el calendario y el resto de campos son opcionales.

Normalmente los métodos POST son consecuencia del envío de un formulario a un servidor. En la aplicación cliente deberá haber un formulario para la creación de nuevos calendarios que realice el envío de los datos mediante una petición POST a este *endpoint*.

Se etiqueta al método *creaCalendario(Representation entity)* con una anotación @Post("json") para especificar que este método se activará cuando se realice una petición REST a este recurso mediante POST. Al ser una llamada de este tipo, el método recibe la información enviada en el *payload* o cuerpo de la solicitud mediante el objeto "*entity*" de tipo *Representation*.

Para gestionar los campos del formulario se decide utilizar una tabla *hash*, donde la clave será el nombre del campo del formulario (atributo HTML *name* de cualquier etiqueta *input*, *select*, *checkbox*...) y el valor será el contenido del elemento. Se itera sobre el formulario y se inserta cada par clave-valor en la tabla hash.

Con los métodos *setSummary(String titulo)* y *setDescription(String descripción)* añadimos el título y la descripción –si hubiera– al objeto *Calendar*, se inserta en la cuenta de Google Calendar y se devuelve como respuesta el identificador del calendario creado.

```
@Post("json")
public Representation creaCalendario(Representation entity) throws IOException,
        GeneralSecurityException{

        /**Proceso de obtención del token de acceso y su posterior intercambio por una
        credencial de acceso de tipo GoogleCredential**/

        //Con la credencial ya podemos crear un servicio Calendar
        Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,JSON_FACTORY,
                credential).setApplicationName(APPLICATION_NAME).build();
        //Los datos llegan desde un formulario en el entity
        Hashtable<String, String> datos_form = new Hashtable<String, String>();
        Form form = new Form(entity);
        logging.info("#######WEB-FORM#######");
        for (Parameter parameter : form) {
                //Mapeo de los campos del formulario recibido
                datos_form.put(parameter.getName(), parameter.getValue());
                logging.info(parameter);
        }
        //Creación de un Calendar (model)
        com.google.api.services.calendar.model.Calendar calendar = new
        com.google.api.services.calendar.model.Calendar();
        //Añadimos título y descripción si hubiera
        calendar.setSummary(datos_form.get("tituloCalendario"));
        if (datos_form.get("descripcionCalendario").compareTo("")!=0){
                calendar.setDescription(datos_form.get("descripcionCalendario"));
        }
```

```
        //Insertamos en Google Calendar el nuevo calendario y devolvemos su ID
        com.google.api.services.calendar.model.Calendar createdCalendar =
        calendario.calendars().insert(calendar).execute();

        return new StringRepresentation(createdCalendar.getId(), MediaType.TEXT_HTML);
}
```

- OPTIONS

Este método HTTP representa una petición de información sobre las opciones de comunicación disponibles en el recurso solicitado (World Wide Web Consortium 1999). Es necesario implementarlo también, ya que las peticiones AJAX que se realicen entre dominios con un tipo de contenido distinto de “application/x-www-form-urlencoded”, “multipart/form-data” o “text/plain” harán que el navegador ejecute primero una petición OPTIONS y después la llamada correspondiente (The jQuery Foundation s.f.).

Además, las peticiones AJAX solo son posibles si el puerto, el protocolo y el dominio del servidor y del cliente son los mismos (Müller 2012). Si la llamada se realiza entre dominios, fallará. Esta problemática –conocida como *cross-domain*– se utiliza como medida de seguridad en los navegadores. La solución se implementa en el lado del servidor y consiste en añadir a la respuesta HTTP la cabecera *Acces-Control-Allow-Origin* con el origen de la petición, que se puede obtener de la cabecera *Origin* de la petición (World Wide Web Consortium 2014).

En este método también se puede informar a los clientes de qué métodos HTTP están permitidos en el recurso añadiendo la cabecera *Access-Control-Allow-Methods* a la respuesta. Por último se indica que se aceptan las cabeceras *Content-Type* y *Authorization*.

```
@Options
public void options() {

        Map<String, String> headers = Utils.getMessageHeaders(getRequest())
        .getValuesMap();
        String origin = headers.get("Origin");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Methods", "GET,POST,OPTIONS");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Headers", "Content-Type");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Headers", "Authorization");
}
```

Es necesario también que todos los recursos incluyan la cabecera *Acces-Control-Allow-Origin* no solo en el método OPTIONS si no también en cada función asociada a los distintos métodos HTTP permitidos por el recurso. Por lo que incluiremos estas líneas de código en los métodos *getJSON()* y *creaCalendario(Representation entity)* del recurso calendarios y también en todos los demás recursos en sus respectivas funciones.

```
Map<String, String> headers = Utils.getMessageHeaders(getRequest()).getValuesMap();

String origin = headers.get("Origin");
```

➤ ***CalendarioResource***

Este recurso permite realizar operaciones de consulta o eliminación sobre un calendario específico que se indica en la URL de la petición REST mediante su identificador. De esta forma, el usuario puede elegir el calendario con el que desea trabajar. Este recurso responde a peticiones GET, DELETE y OPTIONS sobre un calendario determinado y la implementación se realiza en la clase CalendarioResource.java, cuya estructura se presenta a continuación:

```
public class CalendarioResource extends ServerResource{

	private static final HttpTransport HTTP_TRANSPORT = new NetHttpTransport();
	private static final JsonFactory JSON_FACTORY =
JacksonFactory.getDefaultInstance();
	private static final String APPLICATION_NAME = "SmartCalendar";

	@Get("json")
	public Representation getCalendarById() throws IOException,
	GeneralSecurityException, JSONException{

	//IMPLEMENTACIÓN

	return new StringRepresentation(obj.toString(), MediaType.APPLICATION_JSON);
	}

	@Delete
	public void deleteEventById() throws IOException, GeneralSecurityException{

	//IMPLEMENTACIÓN

	}

	@Options
	public void options(Representation entity) {
	//IMPLEMENTACIÓN
	}
}
```

- GET

Una petición mediante este método sobre un recurso /caledarios/{calendarId} devuelve solamente la representación en formato JSON del calendario indicado en la URL. Al igual que en la representación de todos los calendarios, solo se mostrará el identificador del calendario, el título y la descripción.

Se añaden la cabecera necesaria *Acces-Control-Allow-Origin* anteriormente comentada con el origen de la petición para permitir las llamadas entre dominios o *cross-domain*. Con la credencial obtenida se crea el servicio con Google Calendar y se solicitan todos los calendarios de la cuenta del usuario.

Para recuperar el identificador del calendario solicitado en la URL se utiliza el método *getRequestAttributes().get("calendarId")* proporcionado por el framework Restlet. El atributo debe ser el mismo que el indicado entre llaves {} en el patrón de la URL realizado en la clase SmartCalendarApplication.java. Una vez obtenido el calendario solicitado en la petición, se compara con todos los calendarios de la cuenta recuperados previamente y cuando se encuentra se construye el objeto JSON que será la representación del recurso.

```
@Get("json")
public Representation getCalendarById() throws IOException, GeneralSecurityException,
JSONException{

	String origin = headers.get("Origin");
```

```
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);

        /**Proceso de obtención del token de acceso y su posterior intercambio por una
        credencial de acceso de tipo GoogleCredential**/

        //Con la credencial ya podemos crear un servicio Calendar
        GoogleCredential credential = OAuth.getCredential(GoogleToken);
        Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,
                JSON_FACTORY, credential).setApplicationName(
                APPLICATION_NAME).build();

        CalendarList feed = calendario.calendarList().list().execute();
        JSONObject obj = new JSONObject();
        JSONArray lista = new JSONArray();

        //Obtenemos el id del calendario de la URL
        String id = (String) getRequestAttributes().get("calendarId");

        for (CalendarListEntry entry : feed.getItems()) {
                //De toda la lista de calendarios cojo el calendario con ese ID
                if(entry.getId().compareTo(id)==0){
                        JSONObject cal = new JSONObject();
                        cal.put("id", entry.getId());
                        cal.put("summary", entry.getSummary());
                        lista.put(cal);
                }
        }
        obj.put("calendario", lista);
        return new StringRepresentation(obj.toString(), MediaType.APPLICATION_JSON);
}
```

- DELETE

Una petición utilizando este método HTTP sobre un recurso /calendarios/{calendarId} realizará una operación de borrado del calendario de GoogleCalendar cuyo identificador sea el especificado en la URL. Tras añadir la cabecera que facilita el *cross-domain* y obtener la credencial de interacción con el API, se recupera el identificador del calendario solicitado en la URL. Una vez conseguido solamente queda eliminarlo de todos los calendarios del usuario con la operación *calendars().delete(String identificador)*. El método DELETE no tiene una respuesta en forma de representación definida, por lo que se devuelve el código de estado HTTP 200 para indicar que la eliminación ha sido correcta.

```
@Delete
public void deleteEventById() throws IOException, GeneralSecurityException{

        String origin = headers.get("Origin");
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);

        /**Proceso de obtención del token de acceso y su posterior intercambio por una
        credencial de acceso de tipo GoogleCredential**/


        //Con la credencial ya podemos crear un servicio Calendar
        GoogleCredential credential = OAuth.getCredential(GoogleToken);
        Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,
                JSON_FACTORY, credential).setApplicationName(
                        APPLICATION_NAME).build();

        //Obtenemos el id del calendario de la URL
        String id = (String) getRequestAttributes().get("calendarId");

        //Eliminamos el calendario con el ID recibido
        calendario.calendars().delete(id).execute();
        getResponse().setStatus(new Status(200));
}
```

- OPTIONS

Al igual que el recurso anterior, este debe indicar qué operaciones están permitidas sobre una petición /calendarios/{calendarId}. Lo único que cambia con respecto al anterior recurso es que en este las operaciones sobre un calendario no admiten el método POST y sí el método DELETE. Por tanto, solo habría que cambiar los métodos soportados por este recurso.

```
@Options
public void options() {

        Map<String, String> headers = Utils.getMessageHeaders(getRequest())
        .getValuesMap();
        String origin = headers.get("Origin");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Methods", "GET,DELETE,OPTIONS");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Headers", "Content-Type");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Headers", "Authorization");
}
```

- ***EventosResource***

Se aborda ahora el otro tipo de recurso que ofrece la capa de servicios REST: eventos. Para realizar operaciones sobre los eventos de un calendario, las peticiones tendrán que dirigirse a /calendarios/{calendarId}/eventos. Este *endpoint* gestiona solicitudes de consulta de todos los eventos de un calendario mediante peticiones realizadas usando el método GET y la creación de nuevos eventos en el calendario utilizando el método POST.

Un evento en GoogleCalendar dispone de numerosos campos (identificador, etag, estado, título, descripción, lugar, creador del evento, organizador, fecha de inicio, fecha fin, periodicidad, personas invitadas al evento…), de los cuales, dentro del ámbito de este proyecto, solo se utilizarán el identificador, el título, fecha de inicio y fin, y el identificador del evento periódico si es un evento recurrente.

Además, como se indicó en la fase de análisis, Google Calendar no ofrece la posibilidad de añadir campos personalizados, por lo que los nuevos eventos se crean por duplicado: en GoogleCalendar para conocer cuando ocurrirá y en la base de datos para guardar qué acción se debe realizar en la vivienda inteligente cuando suceda. A continuación, se detalla la implementación de los métodos HTTP qué este recurso permitirá en la clase EventosResource.java:

```
public class EventosResource extends ServerResource {

        private static final HttpTransport HTTP_TRANSPORT = new NetHttpTransport();
        private static final JsonFactory JSON_FACTORY =
JacksonFactory.getDefaultInstance();
        private static final String APPLICATION_NAME = "SmartCalendar";
        private static final String HEADERS_KEY = "org.restlet.http.headers";

        @Get("json")
        public Representation getJSON() throws JSONException, IOException,
                GeneralSecurityException{
```

```
        //IMPLEMENTACIÓN

return new StringRepresentation(obj.toString(), MediaType.APPLICATION_JSON);
}

@Post("application/json")
public Representation insertEvent(Representation entity) throws IOException,
        GeneralSecurityException, ParseException{

        //IMPLEMENTACIÓN

return new StringRepresentation(createdEvent.getId() , MediaType.TEXT_HTML);
}

@Options
public void options(Representation entity) {

        //IMPLEMENTACIÓN
}
```

- GET

Una petición de tipo GET sobre este recurso representa una solicitud de todos los eventos que existen en un calendario determinado. Los eventos de un calendario se obtienen mediante el método *events().list(String identificadorCalendario).* A esta petición se le pueden añadir parámetros para afinar más las búsquedas de eventos, como por ejemplo el parámetro *singleEvents*. Este parámetro booleano indica si se desea tratar los eventos recurrentes como un único evento (*true*) o como eventos diferentes (*false*). Se opta por esta última opción de forma que se puedan visualizar todos eventos. Como se ha indicado anteriormente, solo son interesantes en el ámbito de una vivienda inteligente los campos identificador, título, fecha de inicio y fin, y el identificador del evento periódico si es un evento recurrente; de modo que solo estos campos compondrán la representación final en formato JSON del recurso.

```
@Get("json")
public Representation getJSON() throws JSONException, IOException,
        GeneralSecurityException{

        String origin = headers.get("Origin");
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);

        /**Proceso de obtención del token de acceso y su posterior intercambio por una
        credencial de acceso de tipo GoogleCredential**/

        //Con la credencial ya podemos crear un servicio Calendar
        GoogleCredential credential = OAuth.getCredential(GoogleToken);

        Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,
                JSON_FACTORY, credential).setApplicationName(
                        APPLICATION_NAME).build();

        //Cogemos el id del calendario
        String cal_id = (String) getRequestAttributes().get("calendarId");

        JSONObject obj = new JSONObject();
        JSONArray lista = new JSONArray();

        //Pedimos todos los eventos del calendario especificado
        Events eventos = calendario.events().list(cal_id)
                .setSingleEvents(false).execute();
        for (Event evento : eventos.getItems()){
                JSONObject jso = new JSONObject();
                jso.put("id",evento.getId());
                jso.put("summary",evento.getSummary());
                jso.put("start",evento.getStart());
                jso.put("end",evento.getEnd());
```

```
            jso.put("recurrence",evento.getRecurrence());
            jso.put("recurringEventId",evento.getRecurringEventId());
            lista.put(jso);
        }
        obj.put("eventos", lista);
        return new StringRepresentation(obj.toString(), MediaType.APPLICATION_JSON);
}
```

- POST

Este es el método más importante de toda la capa de servicios REST de la aplicación servidor ya que es el encargado de gestionar las peticiones de creación de nuevos eventos en un calendario determinado. Al igual que en la creación de calendarios en el recurso /calendarios, la información relativa al evento será enviada mediante el uso de un formulario desde la aplicación cliente a este *endpoint*. Dado que el evento tendrá varios campos, se hace uso de un *logger* para comprobar cómo llega la información al recurso y si esta se procesa de manera correcta. Se realiza un mapeo de todos los campos del formulario recibido en una tabla hash tal y como se realizó en el método *creaCalendario(Representation entity)* del recurso "/calendarios".

Se instancia el objeto *event* del tipo *Event* proporcionado por las librerías del API de Google Calendar. Este objeto será el que se vaya construyendo según la información que llegue al recurso sobre el evento que se desea crear.

La aplicación es capaz de diferenciar dos tipos de eventos: los que tienen una duración de todo un día y los que tienen una duración específica, indicando su fecha y hora de inicio y fin. Además, estos eventos pueden ser periódicos, para los cuales se debe definir una regla de recurrencia siguiendo el estándar de intercambio de información de calendarios iCalendar (Dawson y Stenerson 1998). Una regla de recurrencia dentro de este estándar puede tener lo siguientes campos:

- **FREQ:** representa la frecuencia con la que ocurre el evento (DAILY, WEEKLY, MONTHLY, YEARLY).
- **UNTIL:** representa hasta qué día y fecha está en vigor la recurrencia del evento. Se define con formato *date-time* (fecha-fin T hora-fin zona-horaria) para eventos con duración especificada o solo como *date* (fecha-fin) para eventos que duran todo el día.
- **COUNT:** representa el número límite de veces que se puede repetir el evento. No puede aparecer en una regla en la que ya exista el parámetro UNTIL.
- **INTERVAL:** representa cada cuánto tiempo se repite el evento (si la frecuencia del evento es diaria cada cuántos días, si es semanal cada cuántas semanas, etc.).
- **BYSECOND, BYMINUTE, BYHOUR, BYDAY, BYMONTH, BYMONTHDAY:** representa cuándo ocurre el evento. Pueden aparecer en la misma regla pero sólo una vez, pudiendo indicar más de un valor separados por coma (en la hora A de los días B,C del mes D).

A continuación se facilita un ejemplo para terminar de comprender el estándar iCalendar:

- “Cada tres semanas, todos los Martes y Jueves hasta el 30 de Septiembre a las 13:00:00”:
  RRULE:FREQ=WEEKLY;UNTIL=20140930T130000Z;BYDAY=TU,TH;
  ITERVAL=3;

Por tanto, la aplicación servidor debe facilitar la llegada de todos los campos anteriores así como procesar todos sus valores. Primero se comprueba de qué tipo de evento se trata, si es un evento que dura todo el día o un evento con fecha y hora de inicio y fin. Para ello se requiere que en el formulario de creación de eventos se facilite al usuario la posibilidad de decidir el tipo de evento que va a dar de alta, siendo lo más cómodo un *checkbox* o un *radio button*, siempre y cuando el atributo nombre (*name*) del elemento HTML sea “allDay”.

Si el evento dura un día, solo se tiene en cuenta el día de inicio del evento y no el día ni la hora de finalización, de forma que el evento comienza a las 00:00:00 horas y termina a las 23:59:59 horas del mismo día. Si es recurrente, se utiliza solo el campo fecha de finalización para indicar en el parámetro UNTIL de la regla cuándo termina la recurrencia del evento.

Cuando el evento tiene una duración específica, el formato de la fecha y hora de inicio y finalización del evento sigue el siguiente patrón, ya que se deben almacenar las horas de inicio y fin: yyyymmddThhmmss+01:00; cuatro dígitos para el año y dos para el mes, el día, las horas, los minutos y los segundos. El último parámetro es la diferencia horaria con respecto al tiempo universal coordinado (UTC), siendo UTC+01:00 el huso horario de España.

De la misma forma que para controlar si un evento duraba un día o si su duración era limitada, la aplicación cliente debe facilitar la labor de indicar que se trata de un evento recurrente. El campo del formulario que se comprobará será el que tenga *“recurring”* como valor del atributo nombre. La regla de recurrencia se va construyendo en función de los parámetros recibidos. Primero se indica el comienzo de la regla y a continuación los parámetros FREQ, UNTIL –con un formato distinto dependiendo de si es un evento diario o con duración establecida–, INTERVAL y si la frecuencia es semanal, se detalla que días de la semana ser repite el evento en el parámetro BYDAY.

Cabe destacar que la fecha y hora en el parámetro UNTIL no llevan guiones ni dos puntos, respectivamente. Estos caracteres se eliminan haciendo uso de la función *replace(carácterAntiguo, carácterNuevo)*, remplazándolos por cadenas vacías. También hay que matizar que se utiliza un contador para conocer si hay más de un día que se repita el evento para insertar la coma que los separa.

Una vez se ha creado la regla de recurrencia, se inserta junto al título, las fechas de inicio y fin en el objeto *event* creado anteriormente; antes de enviarlo a la cuenta de Google Calendar del usuario.

```
@Post("application/json")
public Representation insertEvent(Representation entity) throws IOException,
GeneralSecurityException, ParseException{
        Log logging = LogFactory.getLog(EventosResource.class);
        String origin = headers.get("Origin");
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);
```

```java
/**Proceso de obtención del token de acceso y su posterior intercambio por una
credencial de acceso de tipo GoogleCredential**/

//Con la credencial ya podemos crear un servicio Calendar
GoogleCredential credential = OAuth.getCredential(GoogleToken);
Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,
                      JSON_FACTORY, credential).setApplicationName(
                              APPLICATION_NAME).build();
Event event = new Event();
Hashtable<String, String> datos_form = new Hashtable<String, String>();
Form form = new Form(entity);
logging.info("#######WEB-FORM#######");
for (Parameter parameter : form) {
        //Mapeo del evento
        datos_form.put(parameter.getName(), parameter.getValue());
        logging.info(parameter);
}
String calendar = datos_form.get("calendar");
String nombre = datos_form.get("nombre");
String iniForm  = datos_form.get("inicio");
String iniTimeForm  = datos_form.get("iniTime");
String endForm  = datos_form.get("fin");
String endTimeForm  = datos_form.get("finTime");
String allDay = datos_form.get("allDay");
String recurring = datos_form.get("recurring");
String freq = datos_form.get("freq");
String repetir = datos_form.get("repetir");

EventDateTime startEventDateTime = new EventDateTime();
EventDateTime endEventDateTime = new EventDateTime();

// All Day Event
if(allDay!=null){
        // La fecha de inicio y fin del evento es la misma
        DateTime startDateTime = new DateTime(iniForm);
        startEventDateTime.setDate(startDateTime).setTimeZone("Europe/Madrid");
        endEventDateTime.setDate(startDateTime).setTimeZone("Europe/Madrid");
}
//Timed Event
else {
        DateTime startDateTime = new DateTime(iniForm + "T" + iniTimeForm +
"+01:00");
        DateTime endDateTime = new DateTime(endForm + "T" + endTimeForm +
"+01:00");
        startEventDateTime.setDateTime(startDateTime)
                .setTimeZone("Europe/Madrid");
        endEventDateTime.setDateTime(endDateTime).setTimeZone("Europe/Madrid");
}
//Recurring Events: Construcción de la regla de recurrencia
if(recurring!=null){
        Integer cont=0;
        String regla = "RRULE:";
        regla += "FREQ=" + freq + ";";
        // All Day Event Recurring
        if(allDay!=null){
                String rrEnd = endForm.replace("-", "");
                regla += "UNTIL="+rrEnd+";";
        }
        //Timed Event Recurring
        else {

        DateTime endDateTime = new DateTime(iniForm + "T" + endTimeForm +
"+01:00");
        endEventDateTime.setDateTime(endDateTime).setTimeZone("Europe/Madrid");
        //Eliminamos guiones y dos puntos de la fecha y d la hora
        String rrEnd = endForm.replace("-", "");
        String rrEndTime = endTimeForm.replace(":", "");
        regla += "UNTIL=" +rrEnd+ "T" +rrEndTime+ "z;";
        }

        regla += "INTERVAL=" + repetir;

        if(freq.compareTo("WEEKLY")==0){
                regla += ";BYDAY=";
                if(datos_form.containsKey("D")){
                        if(cont<1){     regla +="SU";}
                        else{ regla +=",SU";}
```

```
                        cont++;
                    }
                    if(datos_form.containsKey("L")){
                        if(cont<1){    regla +="MO";}
                        else{ regla +=",MO";}
                        cont++;
                    }
                    if(datos_form.containsKey("M")){
                        if(cont<1){    regla +="TU";}
                        else{ regla +=",TU";}
                        cont++;
                    }
                    if(datos_form.containsKey("X")){
                        if(cont<1){    regla +="WE";}
                        else{ regla +=",WE";}
                        cont++;
                    }
                    if(datos_form.containsKey("J")){
                        if(cont<1){    regla +="TH";}
                        else{ regla +=",TH";}
                        cont++;
                    }
                    if(datos_form.containsKey("V")){
                        if(cont<1){    regla +="FR";}
                        else{ regla +=",FR";}
                        cont++;
                        }
                    if(datos_form.containsKey("S")){
                        if(cont<1){    regla +="SA";}
                        else{ regla +=",SA";}
                        cont++;
                    }
                }
                //INSERTO REGLA DE RECURRENCIA
                event.setRecurrence(Arrays.asList(regla));
        }

        event.setSummary(nombre);
        event.setStart(startEventDateTime);
        event.setEnd(endEventDateTime);

        //INSERTO EN GOOGLE CALENDAR
        Event createdEvent = calendario.events().insert(calendar, event).execute();

        //INSERTO EVENTO EN BD
        String startBD = iniForm + " " + iniTimeForm;
        String endBD = endForm + " " + endTimeForm;

        String dispositivo = datos_form.get("dispositivo");
        String accion = datos_form.get("accion");
        Float value = null;
        if(!datos_form.get("value").equals("")){
                value = Float.parseFloat(datos_form.get("value"));
        }
        //CREANDO OBJETO HOME EVENT
        HomeEvent homeEvent = new HomeEvent(createdEvent.getId(), startBD, endBD,
                dispositivo, accion, value, 0, "UNKNOW");
        HomeEventDAO heDAO = new HomeEventDAO();

        heDAO.createEvent(homeEvent);

        return new StringRepresentation(createdEvent.getId(), MediaType.TEXT_HTML);
}
```

Como se ha repetido a lo largo de este trabajo, Google Calendar no permite establecer campos nuevos con contenido personalizado en un evento determinado. Es por ello que resulta necesario disponer de una base de datos, en la que guardar el evento y la acción a ejecutar sobre un dispositivo de la vivienda inteligente. En el apartado “5.2.4. Eventos domóticos” se explicará la definición de un nuevo objeto llamado *HomeEvent*, un evento que compartirá el mismo identificador del evento que se acaba de crear en Google Calendar y que indicará la acción domótica a realizar.

Para insertar este nuevo tipo de evento en la base de datos, se especifica también un objeto llamado *HomeEventDAO* que facilitará las operaciones sobre eventos *HomeEvent* en la base de datos, abstrayendo de esta forma la lógica de la aplicación de la persistencia de los datos. La implementación de los métodos de este objeto se detallará, también, en el apartado “5.2.4. Eventos domóticos”.

- OPTIONS

De la misma manera que los anteriores recursos, éste también debe implementar un método OPTIONS que sirva de fuente de información a un cliente de qué métodos están permitidos en el recurso.

```
@Options
public void doOptions(Representation entity) {

        Map<String, String> headers =
                Utils.getMessageHeaders(getRequest()).getValuesMap();
        String origin = headers.get("Origin");

        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Origin", origin);
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Methods", "GET,POST,OPTIONS");
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Headers", "Content-Type");
        Utils.getMessageHeaders(getResponse())
        .add("Access-Control-Allow-Headers", "Authorization");
}
```

5.2.4. Eventos domóticos

Una vez finalizada la implementación de todos los recursos que componen la capa de servicios REST, se aborda en este apartado la definición de una nueva estructura de datos, que se ha denominado *HomeEvent*. Este tipo de objeto es una abstracción de la tabla evento, creada para ampliar la funcionalidad que ofrece Google Calendar añadiendo campos relacionados con la vivienda inteligente.

Así pues, un objeto *HomeEvent* representará un evento insertado en la base de datos, por lo que sus atributos deben ser los mismos que los de la tabla evento, es decir, sus columnas. Se precisa conocer el identificador del evento, el fecha de inicio y fin, el identificador del dispositivo de la vivienda inteligente, la acción que se desea realizar sobre él, el *value* que dependerá de la acción, el estado anterior en el que se encontraba el dispositivo y si el evento está activo (ocurriendo) o no en el momento de la consulta.

Por tanto, el constructor de este tipo de datos se define como sigue en la clase HomeEvent.java, junto a los métodos *set()* y *get()* de cada atributo. Esta clase se encuentra dentro del paquete *“domain”*:

```
package domain;
public class HomeEvent {

        private String eventId;
        private String startTime;
        private String endTime;
        private String dispositivoId;
        private String accionId;
        private Float value;
        private int activo;
        private String previousState;

        public HomeEvent(String eventId, String startTime, String endTime, String
```

```
dispositivoId, String accionId, Float value, int activo, String previousState) {

        this.setEventId(eventId);
        this.setStartTime(startTime);
        this.setEndTime(endTime);
        this.setDispositivoId(dispositivoId);
        this.setAccionId(accionId);
        this.setValue(value);
        this.setActivo(activo);
        this.setPreviousState(previousState);

        }
        //METODOS SET Y GET PARA MODIFICAR Y RECUPERAR LOS VALORES DE LOS ATRIBUTOS… … …
}
```

Para realizar la conexión a la base de datos de una manera más sencilla y funcional, se implementará una clase dedicada exclusivamente para ello, llamada *ConnectionManager*. Esta clase se coloca en el paquete *"dal"*, dedicado a labores relacionadas con los datos. El constructor carga la clase com.mysql.jdbc.Driver, que es el *driver* necesario para realizar la conexión con la base de datos, cuya dirección de acceso se indica en la variable URL, especificando el nombre de la base de datos y el nombre y contraseña de un usuario.

Por último, se implementa un método *connect()* que simplemente se encarga de abrir la conexión con la base de datos y devolver esta conexión para poder realizar operaciones de consulta o inserción.

```
package dal;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionManager {


        private String URL;
        public ConnectionManager() {
                try {
                        Class.forName("com.mysql.jdbc.Driver").newInstance();
                        URL =
        "jdbc:mysql://localhost:3306/smarthome?user=smarthome&password=smarthome";
                } catch (InstantiationException e) {
                        e.printStackTrace();
                } catch (IllegalAccessException e) {
                        e.printStackTrace();
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
        }

        public Connection connect(){
                Connection conn = null;
                try {
                        conn = DriverManager.getConnection(URL);
                } catch (SQLException ex) {
                        System.out.println("SQLException: " + ex.getMessage());
                        System.out.println("SQLState: " + ex.getSQLState());
                        System.out.println("VendorError: " + ex.getErrorCode());
                }
                return conn;
        }
}
```

Una vez definido el gestor de conexiones a la base de datos, se aborda ahora las operaciones que se van a poder realizar sobre un evento domótico o *HomeEvent*. Este

punto es importante, ya que de esta forma la aplicación servidor puede insertar los nuevos eventos procedentes de peticiones al recurso *EventosResource* del API REST. Además, también será capaz de recuperar un evento determinado para comprobar si está activo y desactivarlo o viceversa. Si se activa un evento, se debe almacenar el estado que tenía el dispositivo antes de ejecutar la acción asociada, por lo que también se necesitará un método para guardar el estado anterior.

El objeto encargado de realizar estas tareas se denomina *HomeEventDAO*. Esta clase se incluye en el paquete *"dal"* junto con la clase *ConnectionManager*, separando así la capa de acceso a datos de la capa lógica de la aplicación. Esta clase la componen cuatro métodos:

- *getEvent (String idEvento)*

Como su nombre indica, este método se encarga de realizar una consulta en la tabla evento para recuperar el evento domótico que tenga un identificador determinado. En primer lugar se abre la conexión con la base de datos y se define la consulta a realizar. Si la búsqueda es satisfactoria, se crea el objeto *HomeEvent*, se cierra la conexión con la base de datos y se devuelve el evento.

```
public HomeEvent getEvent(String eventid) {

        ConnectionManager cm = new ConnectionManager();
        Connection cn = cm.connect();
        try {
                Statement stmnt = cn.createStatement();
                ResultSet rs = stmnt.executeQuery("SELECT * FROM evento WHERE 
evento_id='"+eventid+"'");

                while (rs.next()){
                        HomeEvent evento = new HomeEvent(rs.getString("evento_id"),
                                rs.getString("startTime"), rs.getString("endTime"),
                                rs.getString("dispositivo_id"), rs.getString("accion_id"),
                                rs.getFloat("value"), rs.getInt("activo"),
                                rs.getString("estado_anterior"));
                                cn.close();
                                return evento;
                        }


                } catch (SQLException e) {
                        System.out.println("SQLException: " + e.getMessage());
                    System.out.println("SQLState: " + e.getSQLState());
                    System.out.println("VendorError: " + e.getErrorCode());
                }
                return null;
        }
```

- *createEvent(HomeEvent evento)*

Este método es el que se encarga de insertar los nuevos eventos que llegan al recurso *EventosResource* de la capa de servicios REST. El procedimiento a seguir es el mismo que con el anterior método: se abre la conexión con la base de datos, se comprueba si la acción del evento domótico tiene asignado algún valor, se inserta el nuevo registro y se cierra la conexión.

```
public void createEvent(HomeEvent evento) {
        ConnectionManager cm = new ConnectionManager();
        Connection cn = cm.connect();

        try {
                if (evento.getValue()!=null){
```

```
                        Statement stmnt = cn.createStatement();
                        int rs = stmnt.executeUpdate("INSERT INTO evento (evento_id,
                        startTime, endTime, dispositivo_id, accion_id, value,
                        activo) VALUES ('"+ evento.getEventId() +"','"+
                        evento.getStartTime() +"','"+ evento.getEndTime() +"','" +
                        evento.getDispositivoId() +"','"+ evento.getAccionId()+"','"+
                        evento.getValue() +"','" + evento.getActivo()+"')");
                        cn.close();
                        }
                else {
                        Statement stmnt = cn.createStatement();
                        int rs = stmnt.executeUpdate("INSERT INTO evento
                        (evento_id, startTime, endTime, dispositivo_id,
                        accion_id, activo) VALUES ('"+ evento.getEventId() +"','"+
                        evento.getStartTime() +"','"+ evento.getEndTime() +"','" +
                        evento.getDispositivoId() +"','"+ evento.getAccionId()+"','" +
                        evento.getActivo() +"')");
                        cn.close();
                }
        } catch (SQLException e) {
                System.out.println("SQLException: " + e.getMessage());
                System.out.println("SQLState: " + e.getSQLState());
                System.out.println("VendorError: " + e.getErrorCode());
        }
}
```

- *setEventActivo(HomeEvent evento, int i)*

Este método activa o desactiva el evento domótico indicado. Un evento activo (1) significa que el evento está ocurriendo en ese instante de tiempo.

```
public void setEventActivo(HomeEvent evento, int i) {
        ConnectionManager cm = new ConnectionManager();
        Connection cn = cm.connect();
        try {
                Statement stmnt = cn.createStatement();
                int rs = stmnt.executeUpdate("UPDATE evento SET activo='"+i+"' WHERE
                evento_id='"+evento.getEventId()+"'");
                cn.close();
        }catch (SQLException e) {
                System.out.println("SQLException: " + e.getMessage());
                System.out.println("SQLState: " + e.getSQLState());
                System.out.println("VendorError: " + e.getErrorCode());
        }
}
```

- *setPreviousState (HomeEvent evento, String previousState)*

Este método se encarga de almacenar el estado en el que se encontraba el dispositivo de la vivienda inteligente involucrado en el evento. Esta funcionalidad permitirá que cuando un evento finalice, el dispositivo pueda volver a su estado anterior.

```
public void setPreviousState(HomeEvent evento, String previousState) {
        ConnectionManager cm = new ConnectionManager();
        Connection cn = cm.connect();
        try {
                Statement stmnt = cn.createStatement();
                int rs = stmnt.executeUpdate("UPDATE evento SET
                estado_anterior='"+previousState+"' WHERE
                evento_id='"+evento.getEventId()+"'");
                cn.close();
        }catch (SQLException e) {
                System.out.println("SQLException: " + e.getMessage());
                System.out.println("SQLState: " + e.getSQLState());
                System.out.println("VendorError: " + e.getErrorCode());
        }
}
```

5.2.5. Gestión de eventos en tiempo real

El objetivo principal de este proyecto es programar eventos domóticos en un calendario. Pero, ¿cómo conoce la aplicación servidor si hay algún evento a punto de ocurrir?; y, ¿qué ocurre cuando se produce un evento? El API de Google Calendar no ofrece un procedimiento para determinar si existe algún evento cuya fecha de inicio sea "ahora", sino que es necesario indicar la fecha y hora de inicio para recuperar todos los eventos que empiecen a partir de ese momento, la fecha y hora de finalización para obtener todos los eventos hasta ese instante o ambas fechas para obtener los eventos dentro de ese intervalo de tiempo.

Debido a esta carencia, se decide implementar un método que consulte cada minuto si existe un evento en un calendario de Google Calendar que comience entre ese instante y un minuto después. Si la consulta es satisfactoria y hay un evento programado, se recupera su identificador y se busca en la base de datos un evento doméstico *HomeEvent* que tenga el mismo código identificador, pues contiene información sobre el dispositivo inteligente involucrado en el evento y la acción a realizar sobre él. Se procede ahora a explicar detalladamente el algoritmo que seguirá el proceso encargado de realizar las peticiones al calendario cada minuto.

El método se dispara cada 60 segundos y nada más comenzar se solicita al API de Google Calendar los eventos existentes en el periodo de tiempo comprendido entre el momento de la ejecución y un minuto después. Si no existe ningún evento, el proceso termina y se volverá a lanzar cuando haya pasado un minuto desde la primera ejecución, realizando de nuevo una petición de eventos.

Si hay un evento, significa que se ha programado una acción para que ocurra en ese instante, por lo que se busca en la base de datos el evento domótico con el mismo identificador, de forma que se conoce con qué dispositivo de la vivienda inteligente se ha de interactuar. Podría darse el caso en el que debido a un error en la inserción, un evento pudiera estar definido en Google Calendar pero no en la base de datos. Por ello, se debe alertar al usuario registrando la incidencia.

Una vez recuperado el evento domótico, se comprueba si está activo o no. Si no lo está, se cambia su estado a activo, se lanza la petición al API REST de la vivienda inteligente indicando el dispositivo y la acción; y se almacena el estado del dispositivo previo a la acción. Si el evento estuviera activo, quiere decir que está ocurriendo en ese momento, por lo que se debe controlar cuánto tiempo falta para que termine y pueda ser desactivado, volviendo el dispositivo a su estado anterior.

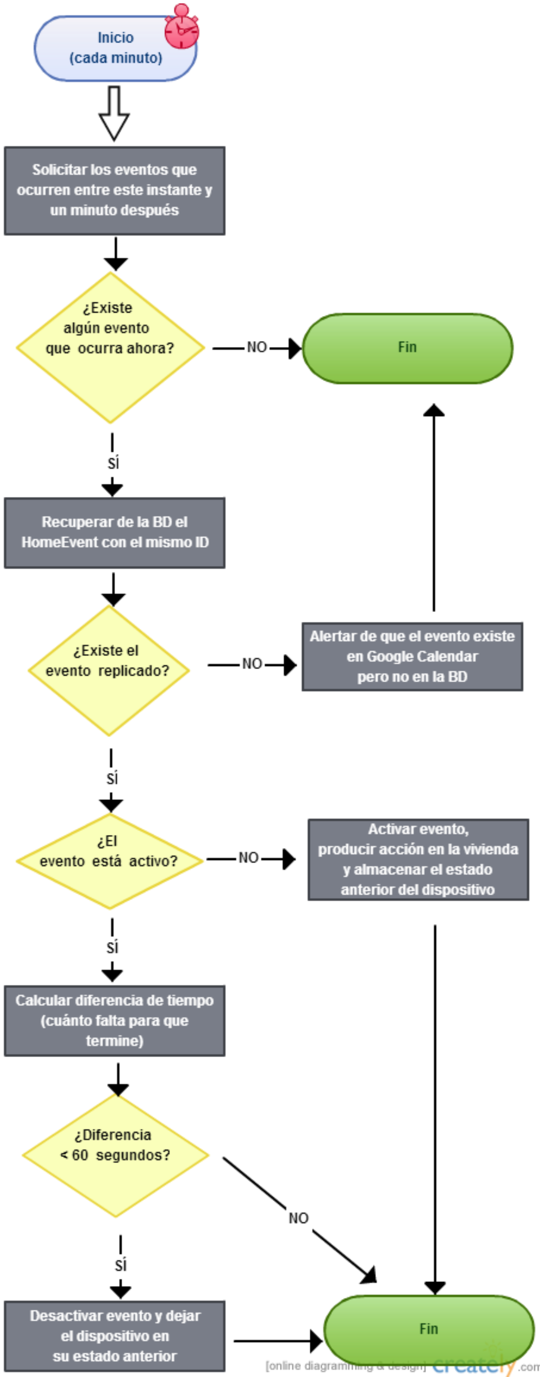

Ilustración 17: Algoritmo de control de eventos

La implementación de este algoritmo la realiza la clase *WatchForEvent*, que se incluye en el paquete *"oauth"* de la aplicación. La llamada a esta clase se realiza nada más arrancar la aplicación servidor, por lo que hay que añadir las siguientes líneas en la clase *SmartCalendarApplication.java*.

```
public Restlet createInboundRoot() {
   //IMPLEMENTACIÓN
   //… …
   //DISPARA LA BÚSQUEDA DEL EVENTO ACTUAL CADA MINUTO
   Timer timer = new Timer();
   timer.schedule(new WatchForEvent(), 0, 60000);
   return router;
}
```

Con la ayuda de un temporizador, se programa que cada minuto se dispare la clase WatchForEvent, encargada de buscar los eventos que puedan estar ocurriendo. Esta clase debe extender de *TimerTask* y destacan dos atributos: refreshToken y calendarID.

```
public class WatchForEvent extends TimerTask {

private static final HttpTransport HTTP_TRANSPORT = new NetHttpTransport();
private static final JsonFactory JSON_FACTORY = JacksonFactory.getDefaultInstance();
private static final String APPLICATION_NAME = "SmartCalendar";
private String refreshToken = "1/dQOSOI1W8gdCaF5438GnosSNBKHunGqGDjFEK2IQzGk";
private String calendarID = "37cdg5r3h11s6jss8k95b7v3uk@group.calendar.google.com";
private HomeEventDAO heDAO = new HomeEventDAO();
```

La primera vez que el usuario consiente que la aplicación acceda a su información privada, se envía un token de refresco junto al token de acceso. Este token permite el acceso a la información privada del usuario sin la necesidad de que esté presente, se conoce como acceso *offline*. El *refresh token* se obtuvo cuando la implementación del protocolo OAuth2 se realizó por primera vez en la aplicación servidor, antes de decidir utilizar la autenticación mediante la cabecera Authorization de HTTP. Se creó una tabla en la base de datos para almacenar a los usuarios que iniciaban sesión en el servicio, de forma que uno de los campos que se almacenaba era el *refresh token*.

Sin embargo, como se verá en el apartado “6. Problemas encontrados”, al realizar la implementación de la autenticación y autorización mediante OAuth2 en la aplicación cliente, se perdió la funcionalidad del token de refresco, afectando así a la escalabilidad del proyecto. En cualquier caso, la implementación de la clase *WatchForEvents* funciona para un usuario y calendario determinados y sigue los procedimientos del algoritmo explicado anteriormente.

```
public void run(){
   DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss+02:00");
   Date date = new Date();
   Date dateMax = new Date();
   dateMax.setTime(date.getTime() + 60000); //Eventos un minuto después

   DateTime dateMin = new DateTime(dateFormat.format(date));
   DateTime dateTimeMax = new DateTime(dateFormat.format(dateMax));
   //REQUEST EVENTS
   try {
       /**Proceso de obtención del token de acceso y su posterior intercambio por una
       credencial de acceso de tipo GoogleCredential**/

     Calendar calendario = new Calendar.Builder(HTTP_TRANSPORT,JSON_FACTORY,
     credential).setApplicationName(APPLICATION_NAME).build();

     Events eventos = calendario.events().list(calendarID).setTimeMin(dateMin)
     .setTimeMax(dateTimeMax).setSingleEvents(false).execute();

     List<Event> eventsFeed = eventos.getItems();
     if(eventsFeed.isEmpty()){
       System.out.println("No hay eventos que ocurran ahora");
     }
     else{
       System.out.println("Hay eventos que ocurren ahora");
       for (Event evento : eventsFeed){
         String eventId = evento.getId();

         //BUSCO EVENTO EN BD
         HomeEvent homeEvent = heDAO.getEvent(eventId);
         if (homeEvent!=null){
            System.out.println("¿Evento activo?: " + homeEvent.getActivo());

            if (homeEvent.getActivo()==0){
```

```
                //EVENTO NO ACTIVO, LO ACTIVO Y EJECUTO ACCION EN VIVIENDA
                Representation resp =
                HomeActions.triggerHomeAction(homeEvent.getDispositivoId(),
                 homeEvent.getAccionId(), homeEvent.getValue());
                JSONObject json = new JSONObject(resp.getText());

                if(json.has("previous-state")){
                  //GUARDO EL ESTADO ANTERIOR
                  heDAO.setPreviousState(homeEvent, json.getString("previous-state"));
                }
                heDAO.setEventActivo(homeEvent, 1);
             }else{
              System.out.println("Espero a que termine...");
              //EVENTO ACTIVO
              //¿CUÁNTO QUEDA PARA QUE TERMINE EL EVENTO?
              long ahora = TimeUnit.MILLISECONDS.toSeconds(date.getTime());
              long dateEvnt = TimeUnit.MILLISECONDS.toSeconds(end);
              long dif = dateEvnt-ahora;
              System.out.println("Faltan " + dif + " segundos");
              if(dif < 60) {
                //SI QUEDAN MENOS DE 60s PARA QUE FINALICE, DESACTIVO EL EVENTO
                  heDAO.setEventActivo(homeEvent, 0);
                  String previousState = homeEvent.getPreviousState();
                  String accion = "";

                  if(previousState.compareTo("UNKNOWN")==0){
                       System.out.println("Estado anterior: UNKNOWN");
                       String previousAccion = homeEvent.getAccionId();
                       if(previousAccion.compareTo("biaON")==0){
                       accion = "biaOFF";
                       }else if(previousAccion.compareTo("biaOFF")==0){
                       accion = "biaON";
                       }else if(previousAccion.compareTo("biaTOGGLE")==0){
                       accion = "biaTOGGLE";
                       }else if(previousAccion.compareTo("movaOPEN")==0)  {
                       accion = "movaCLOSE";
                       }else if(previousAccion.compareTo("movaCLOSE")==0)  {
                       accion = "movaOPEN";
                       }
                  }else {
                       System.out.println("Estado anterior: " + previousState);
                       if(previousState.compareTo("bisON")==0){
                         accion = "biaON";
                       }else if(previousState.compareTo("bisOFF")==0){
                         accion = "biaOFF";
                       }else if(previousState.compareTo("movsCLOSING")==0){
                         accion = "movaOPEN";
                       }else if(previousState.compareTo("movsOPENING")==0){
                         accion = "movaCLOSE";
                       }
                  }

                  if(accion.compareTo("")!=0){
                     System.out.println("Volver al estado anterior con la accion: " +
accion);
                     Representation resp =HomeActions.triggerHomeAction(
                          homeEvent.getDispositivoId(),
                          accion, homeEvent.getValue());
                  }
                  System.out.println("Evento: " + evento.getSummary() + " ha terminado");
                }
              }
            }
           else {
             System.out.println("Evento en tu cuenta de Google pero no en la BD");
           }
         }
       }
      } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
      } catch (JSONException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
   }
  }
}
```

De esta implementación cabe destacar el uso del método *triggerHomeAction (String identificadorDispositivo, String identificadorAcción, Float value)*. Este método es el encargado de realizar la petición a la vivienda inteligente que se detalla en el siguiente apartado.

Cuando queda menos de un minuto para que el evento finalice, se procede a dejar el dispositivo en el estado en el que se encontraba antes de efectuar la acción. Cuando el servidor de la vivienda inteligente arranca por primera vez, la mayoría de los dispositivos se encuentran en un estado desconocido (UNKNOW), por lo que el regreso al estado anterior se realiza en función de la última acción realizada sobre el dispositivo, ejecutando una acción contraria. Cuando un dispositivo sí que dispone de estado anterior, se ejecuta la acción necesaria para volver a ese estado (si el estado anterior es encendido quiere decir que ahora se encuentra apagado, por lo que la acción será de encendido).

5.2.6. Desencademiento de acciones en la vivienda inteligente

Durante la fase de análisis se explicó que una de las características del *framework* Restlet era su capacidad de crear aplicaciones no solamente aplicaciones servidor, si no también aplicaciones cliente. Para interactuar con la vivienda inteligente, la aplicación servidor actuará como cliente, realizando las peticiones necesarias cuando ocurra un evento. Este método se incluye en la clase *HomeActions*, situada en la capa *"domain"* de la aplicación.

```
public static Representation triggerHomeAction(String dispositivoId, String accionId,
               Float value) throws JSONException {

        ClientResource cr = new ClientResource("http://localhost:8182/devFunc/" +
dispositivoId);
        Client client = new Client(new Context(), Protocol.HTTP);
        client.getContext().getParameters().add("useForwardedForHeader","false");

        cr.setNext(client);

        JSONObject body = new JSONObject();
        body.put("action", accionId);
        if (value!=null){
                body.put("value", value);
        }

        Representation resp = cr.put(body, MediaType.APPLICATION_JSON);
        return resp;   }
```

Este método recibe tres parámetros: el identificador del dispositivo, el identificado de la acción y su valor asociado. La capa de servicios REST de la vivienda inteligente solo permite los métodos HTTP GET y PUT, siendo este último el que se utiliza para realizar cambios en los dispositivos. Se crea un objeto *ClientResource* –a diferencia del tipo *ServerResource* del que extienden los recursos del API REST del servidor– que va a realizar la petición sobre la DeviceFunctionality (dispositivo) que recibe el método como parámetro. Como se trata de una petición PUT, se debe indicar en el cuerpo de la llamada (en formato JSON) el nombre de la acción a realizar y el value si lo hubiera. Al provocar una acción en la vivienda inteligente se devuelve una representación del dispositivo afectado, pudiendo obtener su estado anterior.

5.3. Cliente

Tras la implementación de la aplicación servidor, ya se dispone de una herramienta que facilita la interacción entre Google Calendar y una vivienda inteligente. A partir de ahora, se pueden crear sucesivas aplicaciones cliente que se comuniquen únicamente con la capa de servicios REST que implementa.

Para este proyecto, se decide realizar una aplicación web cliente que facilite la inserción de eventos periódicos en un calendario específico. Cuando se desarrolla una aplicación basándose en una API, se debe prestar mucha atención a los requisitos del servicio ofrecido, tanto en cuestión de formato de los datos como en materia de implementación. En otras palabras, el servicio obliga a seguir sus pautas para conseguir la interacción, como se ha realizado a lo largo de este trabajo con Google Calendar. De modo que, para que la aplicación servidor recoja de forma correcta la información relacionada con un evento, todos los campos de los formularios que se van a definir en la aplicación cliente deben llamarse de la misma forma.

5.3.1. Modelo-Vista-Controlador

En la fase de diseño del servidor se decidió separar las clases en capas, de manera que cada capa se compusiera de elementos con una funcionalidad en común. El patrón MVC define tres componentes: el modelo, la vista y el controlador; siguiendo la misma filosofía anterior, diferenciar la lógica y los datos de la interfaz de usuario. El modelo es la representación de la información que va a manejarse y sobre él se pueden realizar acciones de creación, actualización y borrado. Estas operaciones las gestiona el controlador, que es el encargado de conectar el modelo con las vistas, ya que una vista es la representación visual de un modelo (Wikipedia s.f.).

El *framework* PHP Yii está basado en este patrón de diseño y dispone de una herramienta para generar los modelos a partir de la tabla de la base de datos, el controlador y las vistas a partir del modelo. Esta herramienta se utiliza para generar los modelos de las tablas dispositivo y acción, de manera que se puedan administrar de manera sencilla desde sus respectivas vistas. Será necesario incluir los datos de acceso a la base de datos en el fichero de configuración de la aplicación para poder realizar la generación de código:

```
'db'=>array(
        'connectionString' => 'mysql:host=localhost;dbname=smarthome',
        'emulatePrepare' => true,
        'username' => 'root',
        'password' => '',
        'charset' => 'utf8',
),
```

La aplicación web ya está lista para ser desplegada en cualquier servidor web. En este proyecto se ha utilizado Apache, instalado junto a MySQL en la fase previa de la implementación. Se crea en el directorio *htdocs* del servidor web Apache la carpeta *smarthome* de manera que la aplicación web será accesible desde la URL:

http://localhost/smarthome/

La estructura de la aplicación cliente queda de esta forma:

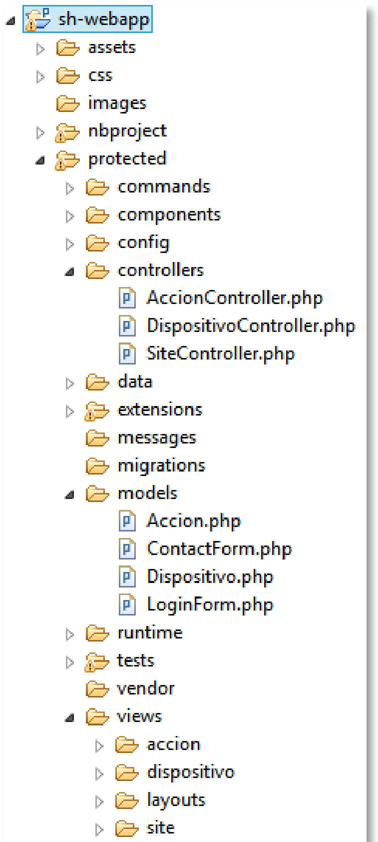

Ilustración 18: Estructura de la aplicación web

5.3.2. Autenticación y autorización

Tras comprender que una API REST debe ser un servicio en el que cada petición tiene que tratarse con independencia de las anteriores, es decir, sin utilizar ningún mecanismo para comprobar el estado del cliente; se decide realizar en esta aplicación la autenticación y autorización con OAuth2.

Para ello se ha añadido el una extensión a la aplicación generada por Yii, llamada eauth (Zemskov s.f.). Este plugin permite crear un sistema de inicio de sesión con una cuenta de Google mediante el protocolo OAuth2 y poder acceder además a las APIs que indiquemos. El primer paso es registrar la aplicación cliente en la consola para desarrolladores de Google, de la misma forma que se procedió para registrar la aplicación servidor. Esta vez, en el campo URL se debe indicar dónde se va a recibir el código de acceso. La documentación del *plugin* eauth indica que debe ser la página dónde se vaya a realizar el inicio de sesión y añadiendo un parámeto *service* a la URL, por lo que será:

http://localhost/smarthome/site/login?service=google_oauth

Una vez se dispone de las credenciales de acceso para la aplicación cliente, se introducen en el fichero de configuración:

```
'eauth' => array(
        'class' => 'ext.eauth.EAuth',
        'popup' => true,
        'cache' => false,
        'cacheExpire' => 0,
        'services' => array(
                'google_oauth' => array(
                'class' => 'GoogleOAuthService',
                'client_id' =>
'80880626630bloh2k14k3cr625db6kglmv2g8b6fpck.apps.googleusercontent.com',
                'client_secret' => 'w-hyQQ_cLwklQxPMO0*********',
                ),
        ),
),
```

Este plugin almacena en una variable de sesión el objeto que contiene todos los elementos relacionados con OAuth, y en este caso interesa obtener el token de acceso que será el que se envíe en las peticiones a la capa de servicios REST del servidor.

```
<?php
$session = Yii::app()->session;
$token = "";
if (isset($session['obj_eauth'])){
        $eauth = $session['obj_eauth'];
        $token = $eauth->access_token;
}
?>
```

5.3.3. Peticiones AJAX autenticadas

Una vez recuperado el *token* de acceso, ya se puede utilizar en la cabecera de una petición a cualquier recurso del API del servidor. Una llamada AJAX es una petición asíncrona a un servidor desde un cliente. El código JavaScript se ejecuta en el navegador de usuario mientras se mantiene la comunicación asíncrona con el servidor. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones (Wikipedia s.f.).

Se hace uso de la librería JavaScript jQuery para realizar las llamadas AJAX, que como mínimo están compuestas de la dirección URL del servidor al que realizar la petición, el método HTTP que se va a utilizar, el formato de la representación q se espera y dos funciones que controlen el éxito (donde se devuelve el resultado de la petición) y el error de la petición.

```
var token = "<?php echo $token;?>";
$.ajax({                url:'http://localhost:8080/smartcalendarREST/calendarios',
        type: 'GET',
        accepts: 'json',
        beforeSend: function (xhr){
             xhr.setRequestHeader('Authorization', token);
        },
        success: function(response){
        $('#calendarios1').empty();
        for (var i=0; i<response.calendarios.length; i++) {
                $('#calendarios1').append('<option value="' +
                response.calendarios[i].id + '">' +
                response.calendarios[i].summary + '</option>');
        }
        },
        error: function (xhr, ajaxOptions, thrownError) {
        console.log(xhr.status + ' | ' + thrownError);
        $('#alert').show().fadeOut(3600);
      }
});
```

El parámetro *beforeSend* permite realizar acciones antes de realizar la petición, por lo que se utiliza para definir la cabecera HTTP Authorization con el token de acceso. Esta llamada sirve para solicitar al recurso /calendarios del servidor todos los calendarios del usuario identificado. Si la petición es satisfactoria, se rellena un elemento *select* con todos los calendarios del usuario y si falla, se muestra un mensaje de error.

Si en vez de una petición GET hubiera sido una petición de tipo POST, se añaden los parámetros *data,* que contiene los datos que se van a enviar al servidor; y contentType, que indica en qué formato se encuentran.

```
$.ajax({
        type: 'POST',
        url: 'http://localhost:8080/smartcalendarREST/calendarios',
        data: $("#form-crear").serialize(),
        contentType: 'application/json',
        beforeSend: function (xhr){
                xhr.setRequestHeader('Authorization', token);
                },
        success: function(resp){
                $( '#event-success #text' ).text( ' Calendario creado satisfactoriamente.
ID: ' + resp);
                $('#event-success').show().fadeOut(3600);
                rellenaCalendarios1();
        },
        error: function (xhr, ajaxOptions, thrownError) {
                $('#event-error').text('Error creando el calendario').show();
                console.log(xhr.status + ' | ' + thrownError);
        }
});
```

Para crear los eventos se implementa un formulario de creación de forma que el usuario puede especificar todos los campos necesarios de un evento. Cabe destacar que el valor del atributo *name* de los elementos HTML que recogen la información es el mismo que utiliza el servidor para procesar los campos recibidos. En otras palabras, cada etiqueta <*input*> (tanto de tipo texto como *checkbox* o *radio*) y <*select*> debe respetar la terminología utilizada en la aplicación servidor, de modo que los atributos *name* de cada campo se correspondan con los utilizados en el servidor.

```
<form id="form" role="form" class="form-horizontal" >
<div class="form-group">

<h2>Elige un calendario</h2>
<fieldset id="f1">
<div class="col-sm-offset-2 col-sm-10">
<select id="calendarios" class="form-control calendarios" name="calendar"></select>
</div>
</fieldset>
</div>

<div class="form-group">
   <h2>Crea un evento para este calendario</h2>
   <fieldset>

   <label class="col-sm-2 control-label">Nombre</label>
   <div class="col-sm-10 mrg-bot-10">
   <input class="form-control" name="nombre" type="text" />
</div>

<div class="col-sm-offset-2 col-sm-10 mrg-bot-10">
<div class="checkbox">
   <label class="control-label"><input id="checkBox" name="allDay"
   type="checkbox">Todo el día</label>
</div>
```

```
</div>

<label class="col-sm-2 control-label">Del</label>
<div class="col-sm-3 mrg-bot-10">
<input class="form-control date" name="inicio" type="text" />
</div>

<label class="col-sm-3 control-label time">desde las</label>
<div class="col-sm-3 mrg-bot-10 time">
<input class="form-control" name="iniTime" type="text" />
</div>

<label class="col-sm-2 control-label">Al</label>
<div class="col-sm-3 mrg-bot-10">
<input class="form-control date" name="fin" type="text" />
</div>

<label class="col-sm-3 control-label time">hasta las</label>
<div class="col-sm-3 mrg-bot-10 time">
<input class="form-control" name="finTime" type="text" />
</div>

<div class="col-sm-offset-2 col-sm-10 mrg-bot-10">
<div class="checkbox">
<label class="control-label"><input id="checkBoxRE" name="recurring"
type="checkbox">Repetir</label>
</div>
</div>

<div class="recurring">
<label class="col-sm-2 control-label recurring">Se repite</label>
<div class="col-sm-4 mrg-bot-10">
<select id="repetir" class="form-control" name="freq">
<option value="DAILY">Cada día</option>
<option value="WEEKLY">Cada semana</option>
</select>
</div>

<label class="col-sm-3 control-label recurring">Repetir cada</label>
<div class="col-sm-2 mrg-bot-10">
<select class="form-control" name="repetir">
<?php for($i=1;$i<=30;$i++) {
   echo '<option value="'.$i.'">'.$i.'</option>';
}?>
</select>
</div>

<label class="col-sm-1 control-label repeating"></label>
<div class="week">
<label class="col-sm-2 control-label">Repetir</label>
<div class="col-sm-8 mrg-bot-10">
<label class="checkbox-inline"><input id="L" name="L" type="checkbox">L</label>
<label class="checkbox-inline"><input id="M" name="M" type="checkbox">M</label>
<label class="checkbox-inline"><input id="X" name="X" type="checkbox">X</label>
<label class="checkbox-inline"><input id="J" name="J" type="checkbox">J</label>
<label class="checkbox-inline"><input id="V" name="V" type="checkbox">V</label>
<label class="checkbox-inline"><input id="S" name="S" type="checkbox">S</label>
<label class="checkbox-inline"><input id="D" name="D" type="checkbox">D</label>
</div>
</div>
</div>
</fieldset>
</div>

<div class="form-group">
<h2>Acción sobre la vivienda inteligente</h2>
<fieldset>
<div class="categorias">

<label class="radio-inline">
<input type="radio" name="cat" value="climatizacion">
Climatización
</label>

<label class="radio-inline">
<input type="radio" name="cat" value="iluminacion">
Iluminación
```

```
</label>

<label class="radio-inline">
<input type="radio" name="cat" value="ventanas">
Ventanas
</label>
</div>

<label class="col-sm-2 control-label">Dispositivo</label>
<div class="col-sm-10 mrg-bot-10">

<select id="dispositivos" class="form-control" name="dispositivo">
<option value="selecciona">Selecciona una categoría</option>
</select>
</div>

<label class="col-sm-2 control-label">Acción</label>
<div class="col-sm-10 mrg-bot-10">

<select id="acciones" class="form-control" name="accion">
<option value="selecciona">Selecciona una categoría</option>
</select>
</div>

<label class="col-sm-2 control-label">Value</label>
<div class="col-sm-10 mrg-bot-10">
<input class="form-control" name="value" type="text" />
</div>
</fieldset>
</div>

<div class="form-group">
<div class="col-sm-offset-2 col-sm-10">
<button type="submit" class="float-right btn btn-info">Crear</button>
</div>
</div>
</form>
```

Llama la atención también que varios elementos *<select>* no tienen contenido. Esto es así debido a que se completan haciendo uso de llamadas AJAX al API para obtener los calendarios y a los controladores para recuperar de la base de datos los dispositivos y las acciones que se pueden realizar sobre cada uno. Una vez obtenida la información, se hace uso de funciones jQuery que la muestran completando los elementos *<select>*. De esta forma se mejora mucho la funcionalidad y la experiencia de usuario navegando por la aplicación.

Cuando el usuario completa el formulario de creación de un evento, se envía el contenido al API del servidor haciendo uso de la siguiente llamada AJAX:

```
   $('#form').submit(function(){

               $.ajax({
                   type: 'POST',
                   url: url,
                   data: $("#form").serialize(), // serializes the form's elements.
                   beforeSend: function (xhr){
                               xhr.setRequestHeader('Authorization', token);
                       },
                   dateType: 'text',
                   success: function(resp){
                          $('#event-success #text')
                          .text('Evento creado satisfactoriamente. ID: ' + resp);
                          $('#calendario').empty();
                          $('#calendario').append('<iframe
src="https://www.google.com/calendar/embed?showPrint=0&amp;showCalendars=0&amp;showTz=0&amp;height=400&amp;wkst=2&amp;hl=es&amp;bgcolor=%23FFFFFF&amp;src='+ cal
+'&amp;color=%23AB8B00&amp;ctz=Europe%2FMadrid" style=" border-width:0 " width="600"
height="500" frameborder="0" scrolling="no"></iframe>');
                       },
```

```
                    error: function (xhr, ajaxOptions, thrownError) {
                                $('#event-error').show();
                            console.log(xhr.status + ' | ' + thrownError);
                    }
            });
            /*e.preventDefault();*/
            return false;
    });
```

La variable “url” está definida en una parte superior del código y se inicializa cuando el usuario elige el calendario en el que se va a insertar el nuevo evento para construir una URL del tipo:

http://localhost:8080/smarthomeREST/calendarios/{CalendarioID}

Si el evento se inserta sin ningún tipo de problema, se vacía la capa HTML que tiene como identificador “calendario” e incrusta el calendario seleccionado, que dispondrá ya del evento.

5.3.4. Interfaces de administración

La herramienta de generación automática de código de Yii facilita el desarrollo rápido de aplicaciones web. Nada más crear una aplicación ya se dispone de una estructura completa, con página de inicio, un formulario de contacto e incluso un sistema de autenticación al que solo hay que añadir los usuarios. El acceso a la administración de dispositivos y acciones solo estará disponible para el usuario administrador. En la cabecera de la aplicación se incluyen los enlaces a los paneles de gestión, permaneciendo ocultos para los usuarios comunes.

Ilustración 19: Cabecera usuario de Google

Ilustración 20: Cabecera usuario administrador

A continuación se muestran los paneles de administración de los dispositivos inteligentes. Estas vistas son iguales para las acciones, la única que cambia es la de crear una nueva, ya que los campos son distintos. Desde la pantalla de administración se obtiene mucha información sobre un dispositivo de un solo vistazo y se facilitan enlaces de acceso a las distintas operaciones CRUD que podemos hacer con el modelo.

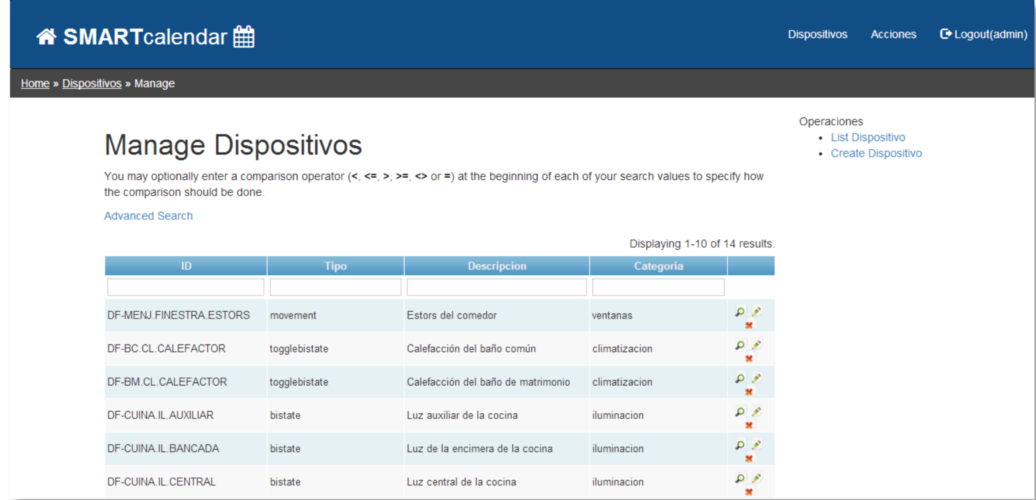

Ilustración 21: Panel de administración de dispositivos

Si hacemos clic en el icono de la lupa, accedemos a la vista detalle de un dispositivo, en el que se resumen los campos del modelo y sus valores. En la columna de la derecha se muestran el resto de operaciones que se pueden realizar sobre el dispositivo.

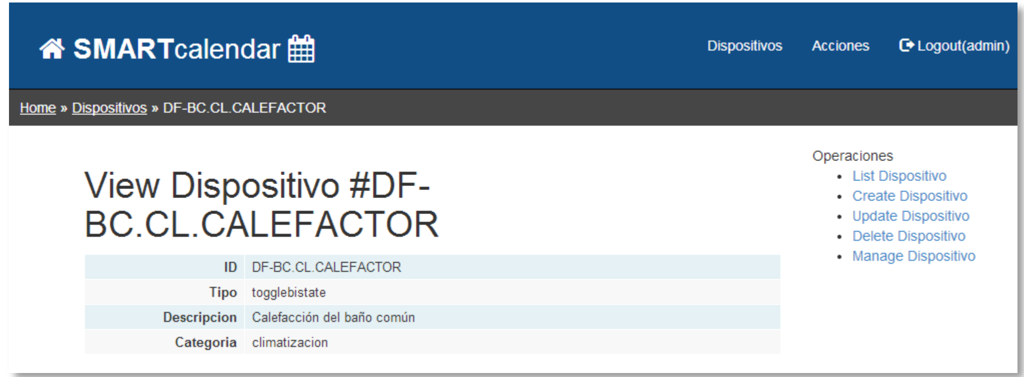

Ilustración 22: Vista detalle de un dispositivo

La vista UPDATE permite actualizar la información sobre el dispositivo. Al hacer clic en el botón guardar, se envían todos los campos al controlador del modelo dispositivo y este se encarga de actualizar los cambios directamente en la base de datos.

Ilustración 23: Vista UPDATE de un dispositivo

Por último, la vista CREATE permite insertar un nuevo dispositivo en la base de datos. Al igual que en la acción UPDATE, es el controlador del modelo el que gestiona y procesa la petición de inserción de un dispositivo.

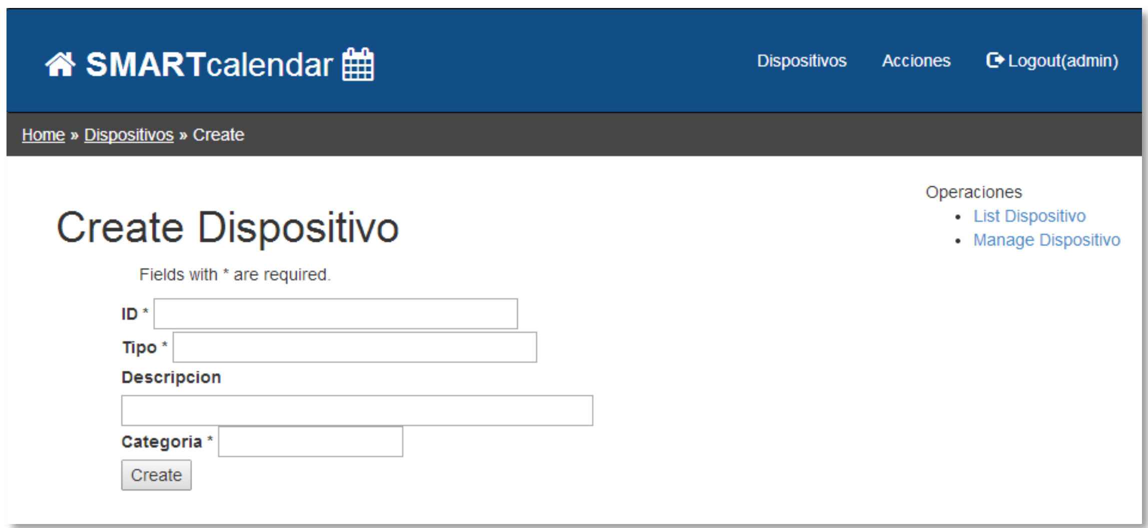

Ilustración 24: Vista CREATE de un dispositivo

5.3.5. Interfaces de usuario

Al tratarse de una aplicación web muy simple, solo se diferencian tres tipos de interfaces. Como se indicó en el apartado de análisis, la maquetación de gran parte del cliente se ha realizado utilizando el *framework* Bootstrap, que facilita el desarrollo de páginas web optimizadas para dispositivos móviles. Todas las llamadas que se realizan a recursos de la capa de servicios REST del servidor son peticiones AJAX.

- **Login:** Ofrece la posibilidad de que el usuario se identifique con su cuenta de Google o como un usuario administrador

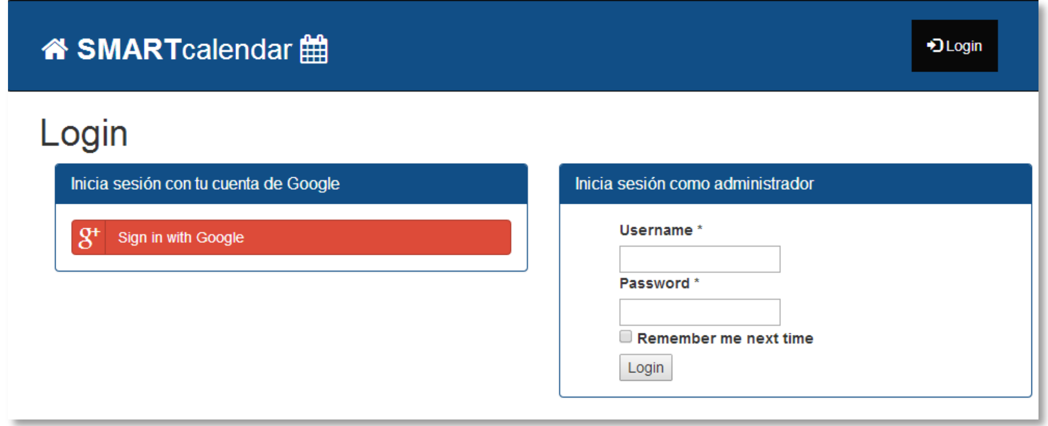

Ilustración 25: Inicio de sesión

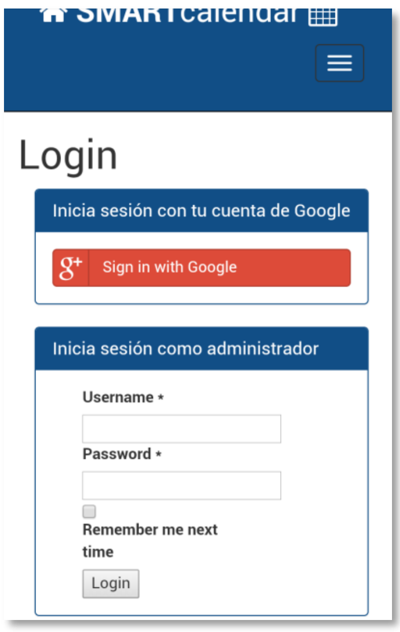

Ilustración 26: Inicio de sesión (versión móvil)

- **Crear y eliminar un calendario:** Utilizando el API de la aplicación servidor, se puede crear un calendario nuevo o elegir de entre los ya existentes y eliminarlo

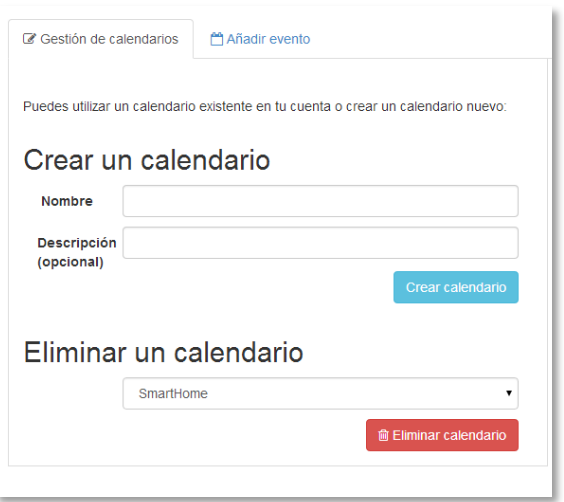

Ilustración 27: Gestión de calendarios

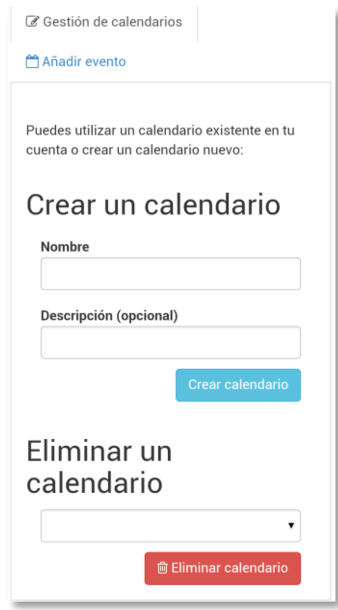

Ilustración 28: Gestión calendarios (versión móvil)

- **Crear evento:**

Es la pantalla más importante de la aplicación ya que, desde el formulario que se facilita, el usuario puede crear los eventos domóticos e insertarlos en su calendario preferido. La creación de un nuevo evento consiste en tres sencillos pasos. Primero, el usuario tiene que seleccionar sobre qué calendario de su cuenta de Google Calendar quiere crear los eventos domóticos. En la parte derecha se visualiza el calendario seleccionado para tener una mayor planificación de los eventos.

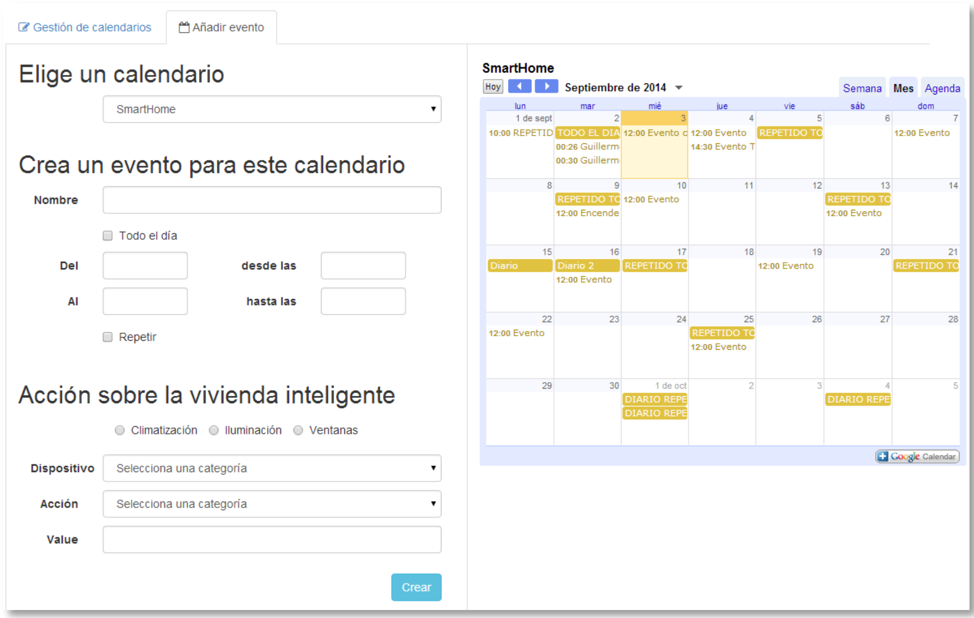

Ilustración 29: Formulario de creación de eventos

Una vez seleccionado, el usuario especifica el título, si dura todo el día o no, las fechas y horas de inicio y fin y la recurrencia del evento, si la hubiera. Al hacer clic en

"Repetir" se muestran las opciones necesarias para crear la regla de recurrencia, permitiendo elegir una frecuencia diaria o semanal, indicando para esta qué días de la semana se realiza.

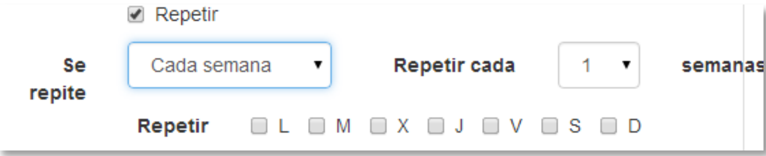

Ilustración 30: Definición de a recurencia del evento

La versión para dispositivos móviles ofrece la misma funcionalidad, solo que se ha omitido el calendario por cuestiones de espacio, quedaba demasiado pequeño y no era usable.

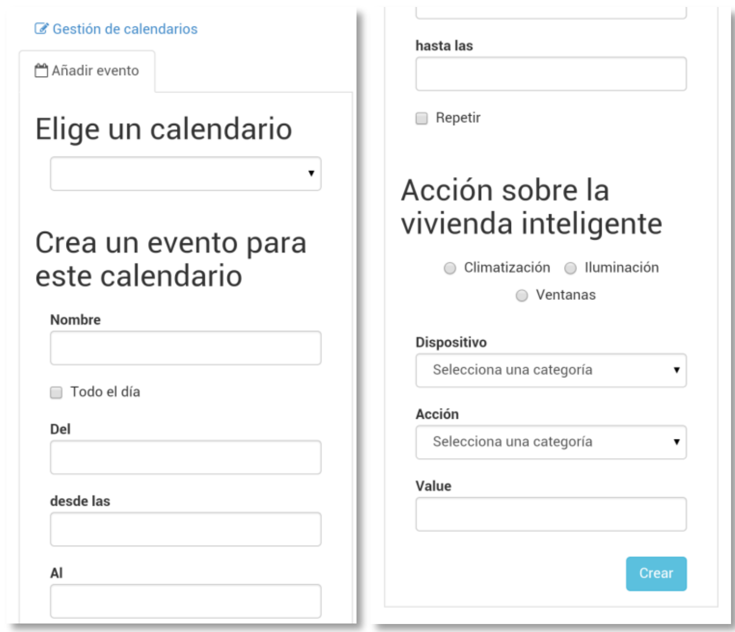

Ilustración 31: Formulario de creación de eventos (versión móvil)

6. Problemas encontrados

Como se ha indicado en la implementación de la clase *WatchForEvents* encargada de realizar la consulta de eventos en un calendario de manera "silenciosa", uno de los mayores problemas que se ha tenido en el desarrollo de este proyecto ha sido no poder obtener un token de refresco desde la aplicación cliente. Este problema ha afectado a la escalabilidad del proyecto.

El fallo se produce cuando se fuerza a aplicación a que el acceso se realice sin la presencia del usuario, indicando el parámetro access_type=offline en la petición de inicio de sesión. Se ha tratado de solventar este fallo de la aplicación, pero no ha sido posible depurarlo ya que el error no aporta mucha más información.

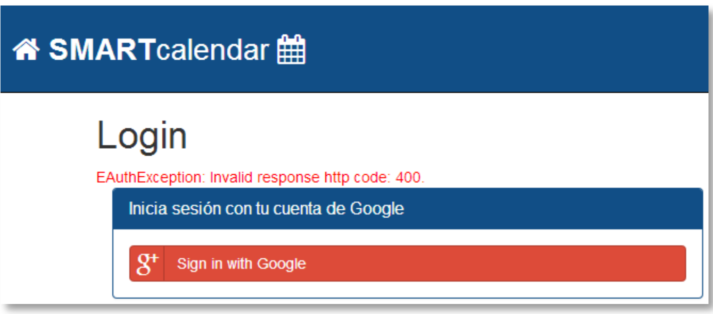

Ilustración 32: Error en el inicio de sesión

Este error se produce al alterar una de las clases del plugin para que se incluya en la petición de inicio de sesión el parámetro de acceso sin conexión y forzando, además, a mostrar de nuevo el diálogo de consentimiento de la aplicación:

```
protected function getAccessToken($code) {
        $params = array(
                'client_id' => $this->client_id,
                'client_secret' => $this->client_secret,
                'grant_type' => 'authorization_code',
                'code' => $code,
                'redirect_uri' => $this->getState('redirect_uri'),
                'access_type' => 'offline',  /*AQUÍ  */
                'approval_prompt'=> 'force',/*FALLA */
        );
        return $this->makeRequest($this->getTokenUrl($code), array('data' => $params));
}
```

También se ha tenido problemas con el tratamiento de las fechas debido a los múltiples tipos de formato manejados. El almacenamiento de la fecha y hora de un evento en la base de datos se realiza en formato *datetime* de MySQL (yyyy-mm-dd hh:mm:ss), mientras que para los eventos en Google Calendar se sigue el formato de acuerdo al RFC3339 (yyyy-mm-dd"T"hh:mm:ss+UTC) (Google s.f.). Además, todos los campos del formulario son tratados como cadenas de texto, por lo que se ha tenido que hacer uso de la clase DateFormat para especificar el formato de las fechas.

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss+02:00");
DateFormat dateFormatBD = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

7. Conclusiones

Para concluir este trabajo, se realiza una síntesis de los resultados obtenidos y su evaluación. El objetivo principal se ha conseguido: el usuario puede programar eventos en un calendario y cuando ocurren se desencadena una acción sobre un dispositivo de la vivienda inteligente. Debido a una falta de análisis previo, la decisión de instalar la extensión eauth en la aplicación cliente ha afectado a la escalabilidad del proyecto, provocando que solo se pueda acceder desde una cuenta y a un calendario.

Los requisitos funcionales de las aplicaciones también están cubiertos en gran medida. El servidor implementa una capa de servicios REST con la que interactuar mediante métodos HTTP completamente funcional. Este actúa como rueda de engranaje entre la vivienda y Google Calendar, solucionando el problema de interoperabilidad entre dos sistemas que aparentemente no pueden comunicar. Realiza un control en tiempo real de los eventos que están a punto de ocurrir, provocando de esta manera la acción en la vivienda inteligente.

Por otro lado, la aplicación cliente sirve de herramienta al usuario para poder planificar eventos. El uso de reglas de recurrencia aumenta el potencial de esta herramienta, permitendo la planificación diaria o semanal de los eventos. Además, cumple otro de los requisitos: permitir la inserción de nuevos dispositivos y acciones así como administrarlos. Por último tiene un diseño adaptativo para poder utilizarse en dispositivos móviles, mejorando así la usabilidad de la aplicación.

Personalmente, he adquirido una gran experiencia en cuanto al desarrollo de interfaces de programación de aplicaciones. Las posibilidades que ofrece un API son infinitas y simplifica la comunicación entre aplicaciones, lo que ha despertado en mí un interés muy alto. También he reforzado mucho los pocos conocimientos en HTML, CSS, JavaScript y PHP que tenía.

Finalmente y a pesar de los problemas encontrados, puedo decir que el resultado final de este proyecto, tanto funcional como conceptual, es más que gratificante y estoy satisfecho.

Bibliografía

Bootstrap Components. http://getbootstrap.com/components/.

Cors, Joan Fons i. «SmartHome Server: Infraestructura para prototipar aplicaciones sobre una Vivienda Inteligente vía REST.» 2014.

Dawson, F., y D. Stenerson. «Internet Calendaring and Scheduling Core Object Specification.» *IETF*. Noviembre de 1998. https://www.ietf.org/rfc/rfc2445 (último acceso: 2014).

Google. «Calendars.» *Google Calendar API*. https://developers.google.com/google-apps/calendar/v3/reference/calendars (último acceso: Julio de 2014).

—. «Events: insert.» *Google CalendarAPI*. https://developers.google.com/google-apps/calendar/v3/reference/events (último acceso: Julio de 2014).

—. *Google Calendar API*. https://developers.google.com/google-apps/calendar (último acceso: Julio de 2014).

—. «Using OAuth 2.0 to Access Google APIs.» *Google Accounts Authentication and Authorization*. https://developers.google.com/accounts/docs/OAuth2?hl=ES (último acceso: 2014 de Julio).

Müller, David. «Cross Domain AJAX Guide.» *David Müller: Webarchitektur*. 10 de Diciembre de 2012. http://www.d-mueller.de/blog/cross-domain-ajax-guide/ (último acceso: Julio de 2014).

Peierls, Tim. «How to set response header.» *Restlet-discuss*. 09 de Febrero de 2012. http://restlet-discuss.1400322.n2.nabble.com/How-to-set-response-header-td7270489.html.

Restlet. «Restlet 2.2 - Tutorial.» *Restlet*. http://restlet.com/learn/tutorial/2.2/ (último acceso: Julio de 2014).

Sahni, Vinay. «Best Practices for Designing a Pragmatic RESTful API.» *Vinay Sahni*. 29 de Mayo de 2013. http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api (último acceso: Julio de 2014).

Sandoval, Jose. *RESTful Java Web Services*. Birmingham: Packt Publishing Ltd., 2009.

The jQuery Foundation. «jQuery().ajax.» *jQuery API Documentation*. http://api.jquery.com/jquery.ajax/ (último acceso: Julio de 2014).

Wikipedia. «AJAX.» *Wikipedia*. http://es.wikipedia.org/wiki/AJAX (último acceso: Julio de 2014).

—. «Modelo-Vista-Controlador.» *Wikipedia*. http://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador (último acceso: Julio de 2014).

World Wide Web Consortium. «Cross-Origin Resource Sharing.» *World Wide Web Consortium (W3C)*. 16 de Enero de 2014. http://www.w3.org/TR/cors/ (último acceso: Julio de 2014).

—. «RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1.» *Vorld Wide Web Consortium (W3C)*. Junio de 1999. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

XAMPP Apache + MySQL + PHP + Perl. https://www.apachefriends.org/es/index.html (último acceso: Julio de 2014).

Yii framework. (último acceso: Julio de 2014).

Zemskov, Maxim. *Yii-eauth*. https://github.com/Nodge/yii-eauth (último acceso: Julio de 2014).