Universitat Oberta de Catalunya

uoc.edu

“Protección de APIs REST “

Daniel Hernando Calleja
Máster Interuniversitario de Seguridad en las Tecnologías de la Información y de las comunicaciones (MISTIC)
Sistemas de autenticación y autorización

Pau del Canto Rodrigo
Victor Garcia Font

Junio 2021

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Seguridad en APIs REST*
Nombre del autor:	*Daniel Hernando Calleja*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	*Nombre y dos apellidos*
Fecha de entrega (mm/aaaa):	06/2021
Titulación:	*Máster Interuniversitario de Seguridad en las Tecnologías de la Información y de las comunicaciones (MISTIC)*
Área del Trabajo Final:	*Sistemas de autenticación y autorización*
Idioma del trabajo:	***Castellano***
Palabras clave	*API, REST, OAUTH*

Resumen del Trabajo:

La finalidad de este trabajo es analizar la seguridad en el contexto de las APIs REST, identificar las principales amenazas a las que se enfrenta este tipo de arquitecturas y enumerar algunas de las posibles soluciones ante las mismas.

Este análisis se ha llevado a la práctica mediante una prueba de concepto en la que se ha definido, codificado y protegido un API REST creado al efecto. Para la protección del mismo se ha utilizado un producto de API Managment, WSO2 API Manager, situándose como intermediador de los accesos que el API proporciona. Este producto permite añadir una capa de seguridad entre el consumidor y el productor del API independiente a los mismos, de manera que se ha instalado y configurado para proteger el API frente a ejemplos prácticos de los principales ataques de los que este tipo de arquitecturas REST es objetivo.

Para la construcción del API REST se han utilizado estándares tecnológicos como son OpenAPI 3.0 o Spring Boot, mientras que para la protección del API se han utilizado mecanismos de seguridad como HTTPS y JWS. Para la autentificación y la autorización de acceso al API se ha utilizado el estándar OAuth, aplicando todas estas opciones de manera transversal mediante la configuración del API Manager.

Las conclusiones obtenidas han sido satisfactorias, ya que se han podido acometer mediante la solución propuesta acciones de protección frente a las vulnerabilidades identificadas como más importantes.

Abstract:

The purpose of this work is to analyze the security context in REST APIs, identify the main vulnerabilitys faced by this type of architecture and list some of the possible solutions to them.

This analysis has been carried out through a proof of concept in which a REST API created for this purpose has been defined, codified and protected. For its protection, an API Management product, WSO2 API Manager, has been used, positioning itself as an intermediary for the accesses that the API provides. This product allows adding a security layer between the consumer and the producer of the API independent to them, so that it has been installed and configured to protect the API against practical examples of the main attacks of which this type of REST architecture is objective.

For the construction of the REST API, technological standards such as OpenAPI 3.0 or Spring Boot have been used, while for the protection of the API, security mechanisms such as HTTPS and JWS have been used. For the authentication and authorization of access to the API, the OAuth standard has been used, applying all these options in a transversal way through the configuration of the API Manager.

The conclusions obtained have been satisfactory, since it has been possible to undertake protection actions against the vulnerabilities identified as most important through the proposed solution.

Índice

Lista de Figuras

.1. Introducción

.1.1 Contexto y justificación del Trabajo

El punto de partida de este trabajo es investigar cual es el contexto actual de las APIs REST en el ámbito de las tecnologías de la información, centrando la atención en la necesidad de protección de las mismas. Cada vez más funcionalidad está expuesta a través de APIs, con lo que su seguridad se está convirtiendo en un factor de vital importancia.

En los últimos años REST se ha impuesto como opción de arquitectura para exponer funcionalidad en APIs. Las empresas e instituciones han apostado por este frente a otros modelos de definición y exposición de APIs como SOAP. Todos los actores importantes en las tecnologías de la información han ido exponiendo sus APIs a través de REST (desde gigantes de Internet como Google o Facebook a webs gubernamentales), y han ido surgiendo retos a resolver a medida que se popularizaba su uso.

En la siguiente figura se puede ver la información que se estima ha sido expuesta en brechas de seguridad en los últimos años. Mucha de esta información se ha obtenido con ataques a los APIs que las exponen, en muchos casos APIs REST.

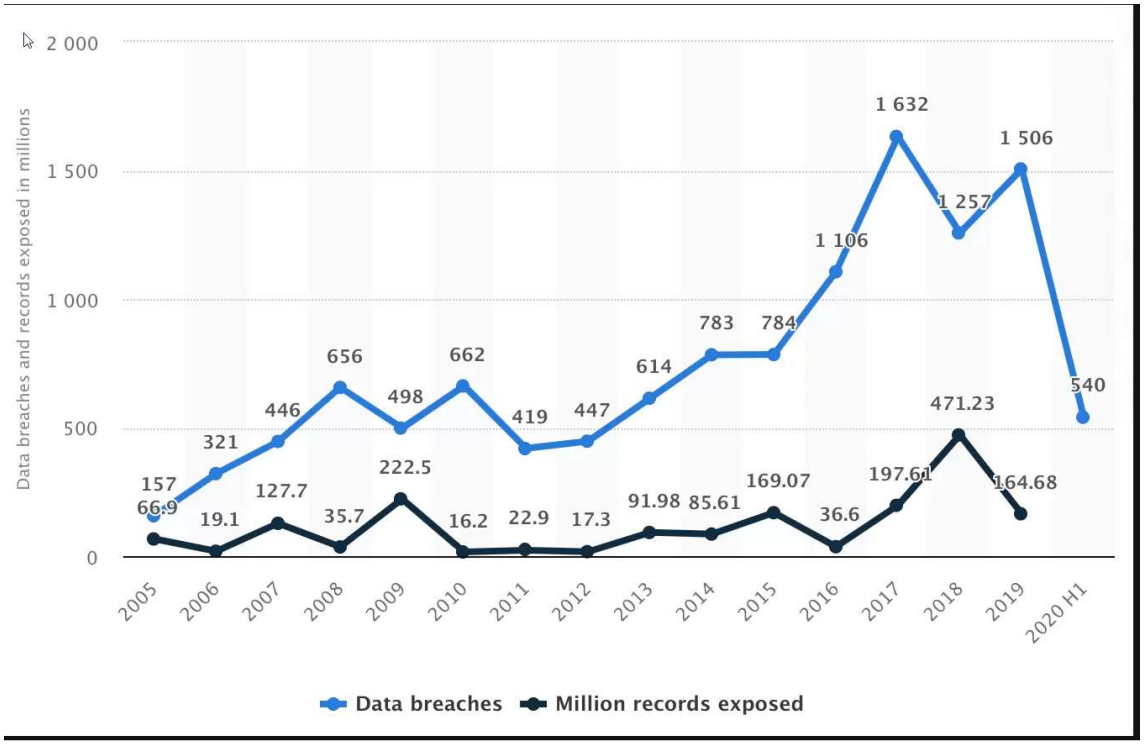

Ilustración 1: Millones de registros expuestos en brechas de seguridad

Todos los actores importantes de la industria exponen información vía APIs y, la mayoría de ellas, lo hacen bajo una arquitectura REST.

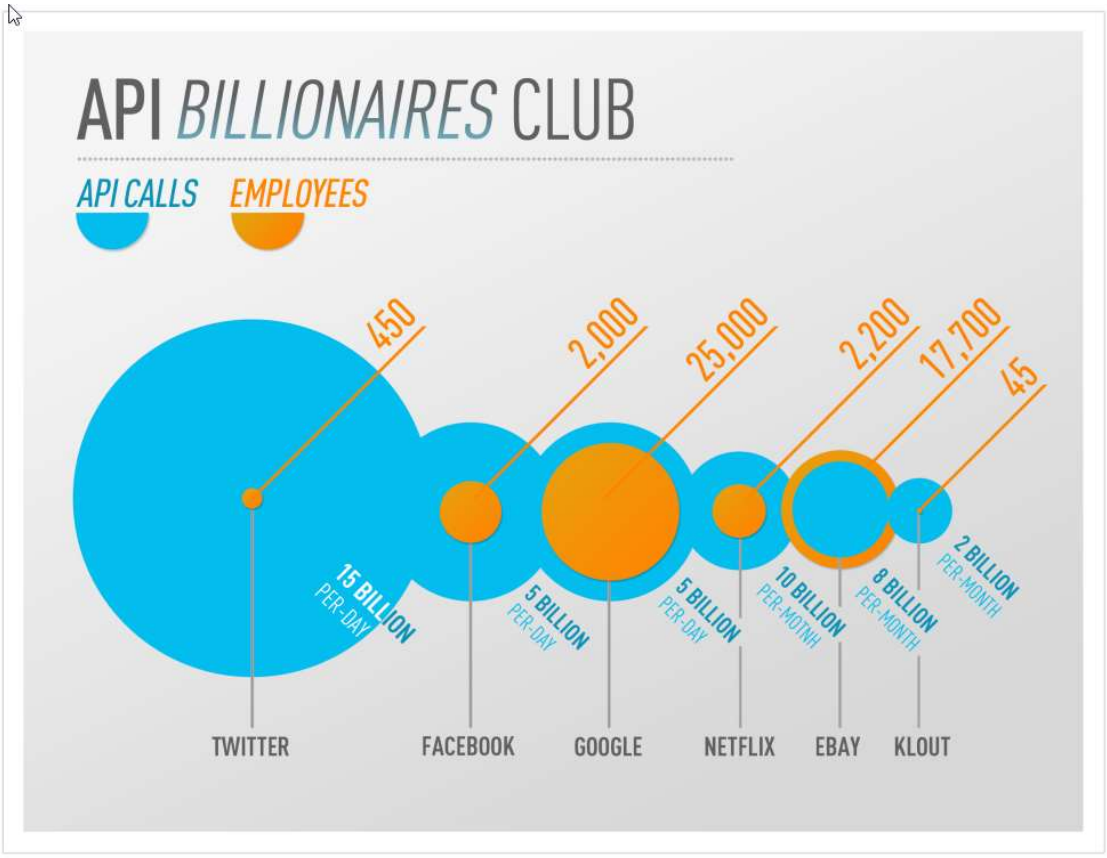

Ilustración 2: Principales APIs REST expuestas por numero de accesos

En base a estos datos, es fácil extraer la conclusión de los errores en la protección de los APIs REST pueden resultar un grave problema. Partiendo de unos requisitos de seguridad generales, se pretende aplicar un conjunto de mecanismos de seguridad que permitan proteger un API REST de de una lista de las principales amenazas a las que están expuestas dichas APIs. En este sentido, el trabajo tratará de plasmar un posible camino para protegerlas, aportando un análisis de las diferentes alternativas existentes a día de hoy para conseguir esto y se justificará la elección o el descarte de cada una de ellas.

Para complementar el resultado de dicho análisis, se realizará una prueba de concepto de protección de un API REST que se definirá en el marco del proyecto y se pondrá a prueba la solución propuesta para la protección contra las amenazas descritas en la fase de análisis mediante un conjunto de pruebas.

.1.2 Objetivos del Trabajo

El propósito de este trabajo es analizar las posibles opciones de protección de un API REST que se diseñará e implementará a modo de prueba de concepto. Este objetivo general se apoyará a su vez en la consecución de objetivos más concretos:

- Conocer su situación actual de implantación y futuro en las tecnologías de información y comunicación de los APIs REST.
- Explicación de los conceptos de API y REST y para qué necesidades escenarios se están utilizando.
- Conocer qué estrategias se están utilizando para la gestión y gobierno de APIs.
- Analizar los principales riesgos y amenazas a las que se exponen los APIs REST
- Analizar los mecanismos y soluciones para la protección de APIs REST.
- Definir un API REST e implementarla en un lenguaje de programación.
- Implementación de ejemplos reales de posibles ataques sobre el API REST definida.
- Implementar prueba de concepto de protección sobre los ataques identificados sobre el API implementado y comprobar la protección obtenida.

.1.3 Enfoque y método seguido

.

Este trabajo de final de máster tiene dos vertientes diferenciadas. La primera, es de documentación e investigación, más teórica, en la que se define el caso a cubrir y cómo podríamos hacerlo. En cambio, hay una segunda más práctica. Se creará una API REST que plantee posibles escenarios en los que se vean algunos de los posibles ataques de los que el API podría ser objetivo y cómo solucionarlos. Las dos partes se resuelven de manera muy diferente, ya que la parte teórica se resolverá mediante la elaboración de un análisis y se generará una documentación al respecto, mientras que la parte práctica, tendrá como resultado un software desarrollado para poner a prueba la solución.

Para la resolución de este trabajo, me voy a centrar en dividir la carga en grandes bloques temáticos que permitan adentrarse en el problema a resolver, comprender el mismo y ver, en la práctica, posibles soluciones. Estos bloques temáticos son dependientes entre sí, con lo que el orden de acometida de los mismos es determinante para completar el trabajo con éxito. El enfoque será el de ir desgranando la parte teórica y, una vez planteada, codificar en la parte práctica un ejemplo o prueba de concepto que sirva para poner a prueba la solución tecnológica propuesta. Los bloques temáticos serían los siguientes:

1. Definición de los conceptos y elementos de trabajo. Aquí se cubrirá desde lo que es un API, a describir lo que es REST. También se detallarán los estándares, productos y posibles soluciones en los que apoyarse cuando se implementa una API REST. Se mostrarán diferentes alternativas y cual es la situación actual de las mismas en las tecnologías de la información.

2. Análisis de riesgos, amenazas y vulnerabilidades de seguridad en APIs REST. Se enumerarán los principales en base a investigar cuales son los mismos, describiendo cada uno de ellos.

3. Implementación del caso de uso que hará las veces de prueba de concepto. Se definirá un API REST y se implementará. Se definirá la arquitectura y mejor solución para exponer su funcionalidad y se realizarán ataques de prueba intentando explotar las vulnerabilidades detalladas en el punto anterior.

4. Se plantearán alternativas a protegerse de los ataques descritos en los puntos anteriores y se escogerán soluciones a los mismos sobre el API implementada.

5. Se valorará la utilización de una herramienta de API Management como medio para otorgar protección al API creado. Si se opta por usar una de ellas, se valorarán algunas y se escogerá una que permita alcanzar los objetivos propuestos.

6. Se extraerán conclusiones de la cobertura de requisitos de seguridad que da la solución propuesta.

Para el desarrollo de esta parte práctica, no me planteo crear un producto nuevo que solvente algo que no existe ahora mismo en el ámbito de los sistemas de información. La API a definir e implementar si será propia, creada al efecto de la prueba de concepto, pero me voy a apoyar en diversos elementos de software que hay en torno a las APIs REST para la implementación de la misma. Así, voy a utilizar estándares de mercado, frameworks de desarrollo que proporcionen librerías de ayuda para el trabajo con APIs REST y productos de gestión y gobierno de APIs. El enfoque es poner todos estos elementos en juego para conseguir los requisitos de seguridad que una API REST debería cumplir y mostrar de manera práctica un camino para poder hacerlo.

Creo que este enfoque es el adecuado pues el objetivo no es la funcionalidad que reside en el API, sino cómo se protegería la misma. Así mismo, las APIs REST tienen un nivel de implantación en las tecnologías de la información tan profundo, que no es necesario crear nada nuevo. El objetivo es elegir entre las posibilidades que se ofrecen con el objetivo de dar con una solución válida a la necesidad de proteger un API REST. Utilizaré por tanto librerías y frameworks de mercado para la implementación y estándares para la definición. También

me planteo la posibilidad de utilizar alguna herramienta para el gobierno de APIs o API Management.

De cara al dimensionamiento de la carga de trabajo y la periodización del mismo, voy a hacer coincidir las fechas de las entregas planificadas en el aula virtual de la asignatura con los bloques definidos. Todo ello quedará plasmado en la planificación de trabajo propuesta en el siguiente punto.

.

.1.4 Planificación del Trabajo

Se ha generado el listado de tareas en la herramienta de planificación. Como se puede observar en la siguiente figura, se han explotado los 5 grandes bloques temáticos en subtareas que dan algo más de detalle y se han introducido las 5 entregas a realizar de acuerdo al plan docente de la asignatura como parte de las tareas a realizar.

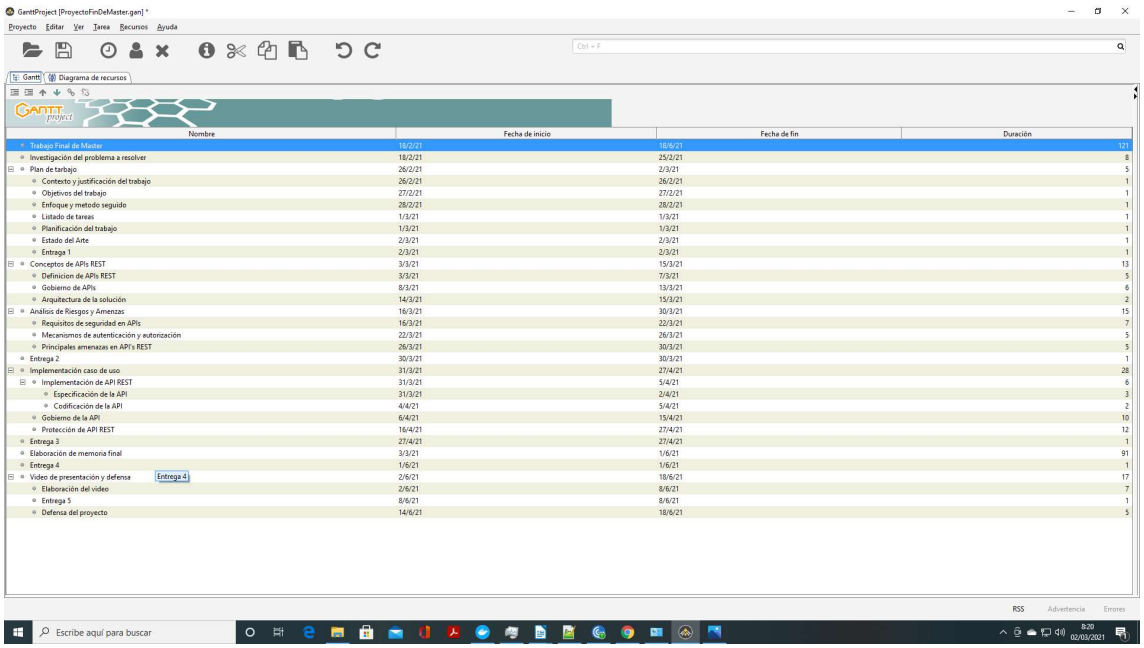

Ilustración 3: Lista de tareas del plan de trabajo

Partiendo de la lista de tareas anterior, se han distribuido en el tiempo acorde al ritmo de trabajo necesario para cumplir con la asignatura. Se ha tenido en cuenta que la dimensión del trabajo no puede exceder en 250 horas, que serían la carga derivada de los 9 créditos que tiene la asignatura. Así, se han distribuido las jornadas incluyendo sábados y domingos, dado que son fechas en las que se trabajará igualmente y tenemos una carga aproximada de 75 horas por cada mes. El primer mes se acometerán las tareas más teóricas de la asignatura, el segundo las más prácticas y el tercero se dedicará a la redacción de la memoria y generación del contenido de presentación del trabajo y defensa del mismo. Destacar que se han creado dependencias entre las fases y que se han introducido tareas de 'Entrega x' a modo de hitos, haciéndolas coincidir con la fecha límite de presentación de las mismas en el aula virtual. Hay una tarea en la que se ha evitado vinculación que es la

elaboración de la memoria, pues la idea es ir completándola continuamente con el material que se aporta en cada una de las otras fases.

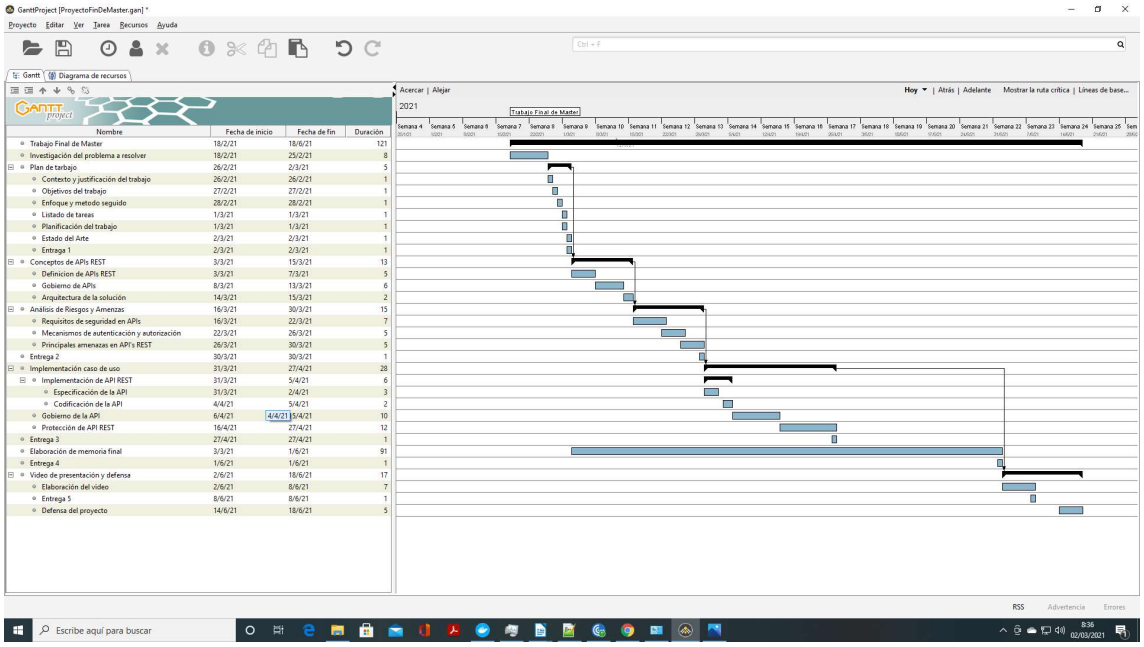

Ilustración 4: Diagrama de Gantt del plan de proyecto

.1.5 Breve sumario de productos obtenidos

Para la realización del Trabajo Final de Máster se exige de un conjunto de entregables a lo largo del ciclo de vida del proyecto. Estos son los siguientes:

PEC1: Se trata del plan de trabajo de la asignatura, donde se especifican los objetivos del proyecto, como se plantea cubrir los mismos y cúal será su desarrollo temporal.

PEC2: En esta entrega se detalla el análisis de la solución. Esto engloba desde una investigación de las tecnologías y arquitecturas que se usan para acometer el trabajo hasta la enumeración de las necesidades de protección de los APIs y sus principales amenazas.

PEC3: Se realiza una prueba de concepto en la que se define un API, se codifica y se protege mediante la solución propuesta en la entrega anterior.

PEC4: Se une todo el trabajo de las entregas anteriores generando una memoria final del proyecto a modo de documento académico en el que se sintetiza el trabajo realizado y conclusiones del mismo.

PEC5: Se realiza un video presentación de unos 15 minutos de duración donde se abordarán los detalles de implementación de la solución adoptada.

.1.6 Breve descripción de los otros capítulos de la memoria

En el capítulo dos se aborda toda la fase de análisis del proyecto. Para hacer este análisis, se ha estructurado el capítulo en cinco secciones. En la primera sección se introduce la definición conceptual de las arquitecturas y tecnologías involucradas en el análisis. Aquí por tanto se explicará que es un API, que es REST o que abarca el estándar OpenApi. Hay una segunda sección que aborda que significa el gobierno de APIs y cuales son sus ventajas. En la tercera sección, se habla de los productos de API Managment, como solución de mercado al gobierno de APIs, enumerando los factores que han facilitado su implantación en muchos escenarios. Posteriormente se abre una cuarta sección en la que se analizan un conjunto de productos de API Management disponibles. Por último, hay una quinta sección en el capítulo donde se detalla cual es el producto de API Management escogido, en este caso WSO2 API Manager, y la arquitectura propuesta para encajar con el resto de piezas para conseguir un gobierno de APIs con el producto.

En el capítulo tres se trata la seguridad de los APIs REST, haciendo inciso en los riesgos y amenazas de las que esta arquitectura suele ser objeto. Es decir, una vez se ha marcado la arquitectura de la solución propuesta, se van a detallar las ventajas y desventajas frente a la seguridad que esta solución tiene. Para ello se ha dividido el texto en tres grandes secciones donde se describen en primer lugar los requisitos de seguridad, después los mecanismos de seguridad escogidos para cumplir con dichos requisitos y por último se enumeran las principales amenazas y riesgos frente a los que se querrá dar protección al API REST.

Ya en el capítulo cuatro se detalla toda la parte práctica del proyecto, esto es, como se ha llevado la prueba de concepto a cabo. Su estructura es similar a la del marco propuesto en la definición, con una primera sección donde se muestra como se ha implementado el API, desde la codificación a la prueba del mismo. Después, se exponeprobando como este API bajo el gobierno del API Manager, publicándolo en el mismo y probando cómo se accede al mismo. La última sección enumera todas las acciones a nivel de configuración que hay que hacer en el producto para protegerlo de las amenazas especificadas en la fase de análisis. Se pondrá a prueba la efectividad de estas medidas para comprobar si tienen o no éxito.

Después de estos tres capítulos que abarcan el grueso del trabajo realizado, hay un capítulo cinco en el que se extraen las conclusiones de lo conseguido en este proyecto y posibles líneas de futuro y posteriormente los capítulos dedicados a la bibliografía, glosario y anexos.

.2. Marco Conceptual de APIs REST

Para poder hacer un análisis que permita proponer una solución al problema de la protección de los APIs REST, primero hay que hacer una breve introducción conceptual en la que se oriente sobre las tecnologías y estándares que sostienen los APIs REST y como se puede poner orden en el gobierno de dichas API's. Una vez esto esté explicado, se puede argumentar cúal será la arquitectura propuesta para llevar a cabo este trabajo.

.2.1 Definición de APIs REST

En esta sección se detallarán los conceptos y estándares claves sobre los que cimentar la solución propuesta.

2.1.1 Definición de API

El marco conceptual de los APIs es extenso. Desde que empezaron a utilizarse, los APIs han ido evolucionando y adaptándose al siempre cambiante entorno de los sistemas de información. Poco tienen que ver los APIs definidos en la década de 1980 con los que hoy en día se definen.

API es un acrónimo de Interfaz de Programación de Aplicaciones (en inglés Application Programming Interface). Una API o interfaz de programación de aplicaciones es un conjunto de definiciones y protocolos que se usa para diseñar e integrar el software de aplicaciones. El concepto hace referencia a los procesos, las funciones y los métodos que brinda una determinada biblioteca de programación a modo de capa de abstracción para que sea empleada por otro programa informático. Este es un primer punto importante en lo que se refiere a la seguridad de los mismos: son programas informáticos pensados para ser utilizados por otro programa. Suelen considerarse como el contrato entre un proveedor de información y un usuario, donde se establece el contenido que se requiere del consumidor (la llamada) y el que necesita el productor (la respuesta).

2.1.2 Definición de REST

REST no es un protocolo ni un estándar, sino que se trata de un conjunto de principios de arquitectura para el diseño de APIs. Fue definido por Roy Fielding en su disertación 'Architectural Styles and the Design of Network-based Software Architectures'. Dado que se trata de una propuesta de solución de arquitectura, los desarrolladores de las APIs pueden implementarlo de distintas formas, de manera que algunos aspectos están sujetos a la interpretación del diseñador de la arquitectura. En cualquier caso, tiene un conjunto de características concretas que todo API REST ha de intentar cumplir. Estos serían los criterios que definirían una API RESTFull o API basada en protocolo REST.

- La comunicación entre el cliente y el servidor carece de estado. No se almacena la información del cliente entre las solicitudes; cada una es independiente y está desconectada del resto.
- La interfaz ha de ser uniforme y ha de contener mensajes descriptivos. REST está ligado a HTTP como protocolo de comunicación entre el cliente y el servidor, todo servicio REST se apoya en él.
- La URI es el identificador único del recurso, es decir, con solamente la información que se traslada en la URI podemos identificar la entidad sobre la que se está actuando.
- Las operaciones adquieren semántica de los verbos HTTP. Así, una acción sobre un recurso (una URI) quedará determinada por el método por el cual se accede a la misma. Así, un método GET implica una consulta sobre la entidad identificada en la URI y un PUT creación de una nueva entidad. Los verbos GET,POST,PUT y DELETE son los más utilizados ya que cubren el espectro de operaciones CRUD sobre una entidad.
- Se utilizan el resto de posibilidades que aporta HTTP (tipo del contenido, códigos de respuesta, autenticación) para que el cliente sepa cómo tratar la información obtenida en las respuestas o cómo aportar información en sus peticiones.
- El cliente no tiene por qué saber la tecnología o lenguajes de programación en los que el servidor pueda implementarse. Solamente conoce el formato de la información a intercambiar.
- Los datos han de poder almacenarse en caché y optimizan las interacciones entre el cliente y el servidor.
- Debe contener hipermedios, lo cual significa que cuando el cliente acceda a cierto recurso, si este hace referencia a otro, se debe poder utilizar hipervínculos para buscar las demás acciones disponibles en ese momento (paradigma HATEOAS)
- El sistema en capas que se pueda utilizar en el servidor para proporcionar una respuesta, serán invisibles para el cliente (los encargados de la seguridad, del equilibrio de carga, etc.).
- Existe la posibilidad opcional de responder con código bajo demanda, de manera que en la propia respuesta se traslade cierta lógica de ejecución para que el cliente la ejecute en su lado (código JavaScript, por ejemplo).
- El contrato ha de mantenerse frente a los cambios de implementación. Esto implica el uso de versiones del API, de manera que no se afecte a los desarrolladores frente a cambios no retrocompatibles. Las versiones del API quedarán plasmadas en la propia URI de acceso a la misma.

Las principales ventajas de REST son que aporta escalabilidad, flexibilidad e independencia a ambos lados de la comunicación. Al no estar ligado a complejos protocolos, es adaptable a múltiples plataformas. Consume muy pocos recursos y se adapta muy bien a la infraestructura de Internet que ya se apoya en el protocolo HTTP.

Hay que entender el por qué del triunfo de las APIs REST en los últimos años. Antes de REST, el estándar de facto para el consumo de APIs era SOAP o XML/RPC. Este era un protocolo muy completo, con una cobertura de todos los

posibles aspectos de un API (seguridad, transporte, enrutamiento, definición de entidades, etc), pero muy férreo y muy complejo. Tenía un alto escalón de aprendizaje. Era relativamente complejo entender todo lo necesario para definir un API y conseguir que la misma se ejecutase. Además, todos los actores importantes del mercado informático tomaron posiciones alrededor de esta tecnología e hicieron sus propias implementaciones de la misma (cada uno hizo su propio parser de SOAP/XML con sus propias reglas pese a que había un estándar definido por el W3C) , a menudo no totalmente compatibles. Esto hizo que REST, al estar construido sobre tecnologías plenamente implantadas desde hacía años (HTTP, JSON), no exigiese aprender lenguajes nuevos ni era propiedad de nadie. Además consumía pocos recursos y era escalable a dispositivos con poca capacidad de proceso.

No todo podían ser ventajas y tenía algunas debilidades. Como sobre reacción a SOAP y su complejidad, se quiso hacer de REST muy simple. Tan simple que los contratos de los APIs no se definían, se dejaba a criterio del diseñador como informar a los clientes de cómo usar su API. Esto derivó en la que se convirtió en principal desventaja de REST: no había forma de automatizar la construcción de los clientes de las APIs como si ofrecían otras alternativas como las antes mencionadas. Esto dificultaba los desarrollos de APIs y las pruebas sobre los mismos. Tampoco había nada en REST que definiese cómo se había de documentar el funcionamiento de los APIs.

Esto era un problema grave en términos de productividad e interoperatividad, con lo que, aunque la arquitectura REST no provee de un lenguaje autodescriptivo para las APIs, se fue imponiendo esta necesidad. De ahí surgieron estándares de facto como inicialmente con el nombre de Swagger o, más recientemente, especificación OpenAPI. Estas son básicamente reglas de autodescripción mínima de los API basados en lenguajes de estructuras de datos muy flexibles, estandarizados y nativos a la tecnología HTTP como son JSON y YAML.

2.1.3 Swagger y OpenApi

El proyecto de Swagger (propiedad de la empresa SmartBear) se donó a la iniciativa OpenAPI en 2015 y desde entonces se conoce como OpenAPI. Ambos nombres se usan indistintamente. Sin embargo, mientras OpenAPI hace referencia a la especificación. Swagger hace referencia a la familia de productos comerciales y de código abierto de Smartbear que funcionan con la especificación OpenAPI. Los productos de código abierto, como OpenAPIGenerator, también se encuentran bajo el nombre de la familia de Swagger, a pesar de no ser publicados por Smartbear. En resumen:

- OpenAPI es una especificación.
- Swagger es una herramienta que usa la especificación OpenAPI. Por ejemplo, con sus productos OpenAPIGenerator y SwaggerUI.

OpenAPI es una especificación independiente del lenguaje que sirve para describir APIs REST. Permite a los equipos y a los usuarios comprender las

capacidades de una API REST sin acceso directo al código fuente. Sus principales objetivos son los siguientes:

- Minimizar la cantidad de trabajo necesaria para conectar los servicios desacoplados.
- Reducir la cantidad de tiempo necesario para documentar un servicio con precisión.
- Automatizar cierta parte de las actividades de construcción y pruebas sobre APIs REST.

El ejemplo tipo en el que se pueden ver los conceptos tratados sería este que se aporta en la propia URL de la especificación

----

```
openapi: 3.0.0
info:
  title: Sample API
          description:     Optional     multiline     or     single-line     description     in
[CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
-------------------------------------
```

Como se puede ver, en este caso se ha utilizado YAML para describir el API. Hay metadatos para describir la funcionalidad, la versión, los servidores donde reside el API (tanto de producción como de pruebas), y las descripciones de las operaciones, parámetros de las mismas, formato del contenido y sus posibles código de respuesta.

.2.2 Gobierno de APIs REST

La estandarización de los APIs a nivel descriptivo puede resolverse con OpenAPI, pero la proliferación de arquitecturas de microservicios basados en APIs en el mercado informático hizo patentes nuevas necesidades que la arquitectura REST por sí misma tampoco cubría. Derivado de esto, surgieron problemas relacionados con la organización de los APIs, su mantenimiento, su exposición, su consumo, su seguridad, etc. Con la llegada de la denominada ‘economía de APIs’, las empresas pasaron a exponer información de negocio mediante APIs e introducirlas en su modelo de negocio como un activo más.

Esto provocó una necesidad de mayores capacidades de control y escalado. Se necesitaba de métricas para saber cuánta información exponía un API o quien lo consumía, ya que en muchos casos se monetiza su consumo. El número de APIs REST en las organizaciones también ha repercutido en que sea necesario un sitio centralizado de gobierno de las mismas, para que no se provoque el caos cuando el número de ellas asciende a cientos, o incluso miles. La diversidad de dispositivos que pueden ejercer como cliente de las APIs REST así como el nivel de exposición de las mismas (si es para programas internos, si es de exposición pública, etc) exigen de un control que la arquitectura original REST no estaba planteada para resolver y en la que estos aspectos no tuvieron peso en su diseño.

El gobierno de APIs es más necesario en tanto en cuanto más grande sea la organización, mayor sea el número de APIs expuestas, más sean los programas que las invocan y más críticos sean los servicios que proporcionan. El ecosistema de aplicaciones es tan diverso que una misma organización puede tener que lidiar con el problema de proporcionar un método ágil para la exposición de funcionalidad mediante APIs REST entre programas que residen dentro de la misma organización, como para hacerlo a dispositivos móviles que ejecutan software de terceros que se nutren de sus APIs. En los casos más extremos estos dispositivos pueden ser móviles, tabletas, asistentes personales o dispositivos IoT. Un mismo API puede ser expuesto a todos estos diversos clientes y hay que dar soluciones de diferente calado dependiendo del programa origen de la invocación.

.2.3 API Managers

Para cubrir todas estas necesidades de gobierno de APIs REST en la última década han surgido numerosos productos comerciales y de software libre que permiten el gobierno de las APIs en un repositorio centralizado, externo al propio API, que permita ejercer las labores de control sobre los APIs de manera transversal y transparente a la codificación del mismo.

Esta arquitectura tiene múltiples ventajas y viene a completar lo que REST propone, ya que la inclusión de estos productos no afecta para nada ni a los consumidores de servicios ni a los productores. Las APIs pueden ser clasificadas en base a políticas definidas en el API Manager y ofrecen una solución completa para llevar los siguientes controles sobre las APIs REST:

- Ciclo de vida de las APIs. Permiten desde la definición de la misma dentro del mismo producto, hasta la importación de una API mediante su definición OpenApi. También permiten manejar el ciclo de publicación de las mismas, versionado, adjuntar documentación o ejemplos de pruebas.
- Organización de las APIs. Permiten clasificarlas, tanto desde el punto de vista del productor, como del consumidor. Dependiendo del rol que cada uno tenga, puede operar con las APIs que exponga o que consuma, asignándole políticas de consumo, de seguridad, etc.
- Monetización de las APIs. Permiten aplicar planes de precios a las APIs en caso de querer comercializar su uso en base a distintos SLA’s.

- Trazabilidad y estadísticas. Ofrecen logs para la auditoría, resolución de problemas o cuadros de mando con finalidad estadística que permita tener idea del rendimiento de la plataforma.
- Enrutamiento. Desde el Api Manager se debe poder encaminar las peticiones a los servidores que de verdad poseen la implementación del API. Esto aplica desde el simple enrutamiento hasta el backend que sirve el API REST hasta funcionalidades más avanzadas como el balanceo de carga, la política en caso de fallo o políticas de circuit breaker.
- Mediación de APIs. Se pueden programar acciones de middleware entre el consumidor del mensaje y el productor. Pueden ser acciones personalizadas relacionadas con la seguridad o con la transformación de peticiones.

Para realizar estas funciones, los API Managers suelen contener varios módulos que el usuario puede o no utilizar. Estos módulos pueden ser diversos en función del producto escogido, pero casi todos los productos constan al menos de los siguientes elementos:

- Portal del suscriptor de APIs o Portal del desarrollador. Este módulo dará al usuario la posibilidad de consultar los APIs que tiene disponibles para consumir, así como la de crear credenciales, consultar documentación de las API expuestas o descargar ejemplos de invocación. Dependiendo del ámbito de exposición de las APIs puede ser un portal de acceso restringido, pero lo normal es que se exponga cuanto menos a terceros o directamente su acceso sea público bajo registro.

- Portal del publicador de API's. Este módulo se proporciona para el perfil de usuario de publicador de servicios, es decir, es un perfil para la administración de las APIs expuestas. Permite controlar la definición, versionado, ciclo de vida, monetización, etc, es decir, es desde donde se configura el gobierno de los API's. Suele tener un perfil de acceso restringido a la propia empresa.

- Gateway de servicios. Este módulo es el que aplica la configuración definida en el módulo de publicación de API's, es decir, la configuración y la aplicación de la misma se realiza en módulos distintos. Este módulo tiene la finalidad de exponer las APIs a los subscriptores, aplicando las políticas de consumo que le aplique a cada uno de ellos.

Hay productos que tienen algunos módulos más, pues estos han crecido mucho en los últimos años, pero por lo general estos 3 módulos son comunes. Luego existe la posibilidad de tener módulos de estadísticas, controladores de tráfico, repositorios de identidad, etc. Ya depende de la complejidad del producto.

.2.4 Productos de API Management o API Gateway

Como ya se ha comentado previamente, existen muchos productos en el mercado que tienen como finalidad el gobierno de APIs. Para recoger una muestra significativa, se puede visualizar este gráfico de la consultora Forrester que muestra un cuadrante con el posicionamiento de estos productos en el mercado en base a sus criterios de calificación. Es el Forrester Wave: API Management Solutions, correspondiente al tercer trimestre de 2020.

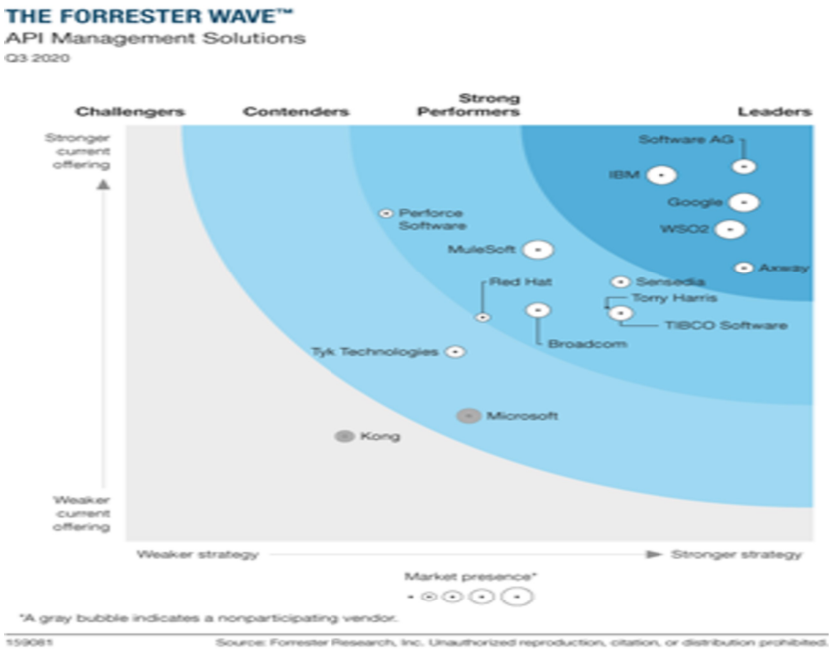

Ilustración 5: Cuadrante de Consultora Forrester sobre soluciones de Api Managment

Hay muchas opciones y están muchos de los grandes fabricantes de software proponiendo su producto, lo que deja muy a las claras el nivel de implantación de estas soluciones en la industria. De entre todas estas opciones, voy a evaluar varias con el objetivo de seleccionar una que sirva para los propósitos de este trabajo. Así, he adoptado un primer criterio de selección, que es que el producto tenga una versión de código abierto o me permita ejecutar una licencia temporal durante el periodo que cubre este trabajo.

2.4.1 Kong

Kong nació en 2011 como un API Gateway privado de la empresa Kong Inc. (originalmente Mashup) apoyado en el servidor HTTP Nginx con un enfoque claro: ofrecer alto rendimiento. Más tarde en 2015 se convierte en un proyecto open-source.
Hoy en día su uso está extendido de forma global en más de 5000 organizaciones.

Disponemos de dos modalidades de Kong:

- Community Edition: La versión CE viene bastante bien equipada, con soporte de plugins open-source, balanceo de carga y service discovery entre otros, pero no incluye un panel de administración, por lo que configuraremos Kong vía REST o bien apoyándonos en algunos de los dashboard open-source que existen como Konga o Kong Dashboard.
- La versión Enterprise Edition nos ofrece muchas más funcionalidades de serie como el dashboard de administración, plugins de seguridad, métricas y soporte 24×7 entre otros.

Kong puede ser instalado de diversas formas tanto en infraestructura propia como en la nube. Disponemos desde paquetería nativa de Debian, Red Hat, y OS X, hasta contenedor de Docker y CloudFormation para AWS entre otros.

Como ventajas está el demostrado buen rendimiento y su facilidad de escalabilidad, ya que se distribuye en modelo contenedor. La principal desventaja radica en que la versión de código libre tiene funcionalidades limitadas, principalmente en las interfaces gráficas de administración y el módulo de seguridad basada en Oauth, con lo que en principio la descarto, ya que creo que este módulo es muy interesante para resolver el trabajo.

2.4.2 TYK

Tyk es un API Gateway open-source que nació en 2014 antes incluso del API Gateway como servicio de AWS. Tyk está desarrollado en Golang y utiliza el servidor HTTP del propio lenguaje.

Si queremos utilizar Tyk disponemos de varios sabores: Cloud, Hybrid (GW en infraestructura propia) y On-Premise.

Cabe destacar que en la versión cloud podemos tener un API Gateway con un tráfico de 50.000 peticiones diarias de forma gratuita, mientras que la modalidad on-premises nos permite tener una instancia corriendo de manera gratuita.
Existen varias formas de instalar Tyk: paquetería estándar de Ubuntu y RHEL, tarball o contenedor de Docker.

Al contrario que Kong, una de las ventajas de Tyk es que ofrece una versión completa incluso en modalidad On Premise, con todos los módulos incluidos. Puedes elegir instalarlo en la modalidad Cloud o como paquetería en tu entorno RHEL. Como desventaja principal, he encontrado poca documentación con la que empezar a trabajar con él y las interfaces de administración parecen muy poco trabajadas.

2.4.3 WSO2 API Manager

WSO2 API Manager combina fácilmente la gestión de accesos al API y la gobernación y el análisis de las mismas. El producto tiene una historia de más de 10 años en el mercado y es uno de los productos más asentados. Proporciona versión Open Source con todos los módulos disponibles, en modalidades contenedor u On Premise. Permite el desarrollo de mediaciones, para poder construir controles personalizados de seguridad y estructura híbrida de Microgateways. Esto hace de ella la solución más completa de las tres analizadas para mi caso de uso, pues la principal desventaja frente a Kong o Tyk es que, en caso de tener que escalar el número de gateways, puede ser significativamente más caro y complejo que las otras dos alternativas.

Existen otros productos muy extendidos y asentado, como Microsoft Azure Gateway, RedHat 3Scale, Mulesoft, Google Apigee o IBM Gateway, pero ninguno de ellos dispone de modalidad gratuita de licenciamiento, con lo que los he descartado para los propósitos de este trabajo. Teniendo en cuenta todos estos factores, utilizaré WSO2 API Manager como solución para el gobierno de APIs REST y su securización.

.2.5 Arquitectura de la solución propuesta

Una vez definidas las tecnologías y productos que voy a tener en cuenta para abordar este trabajo, hay que especificar la arquitectura conceptual que va a permitir simular la que podría ser una arquitectura de una solución API REST tipo, como la existente en muchas entidades y empresas.

En la siguiente figura he construido un diagrama de elementos que describe cuáles serían las piezas fundamentales de la infraestructura.

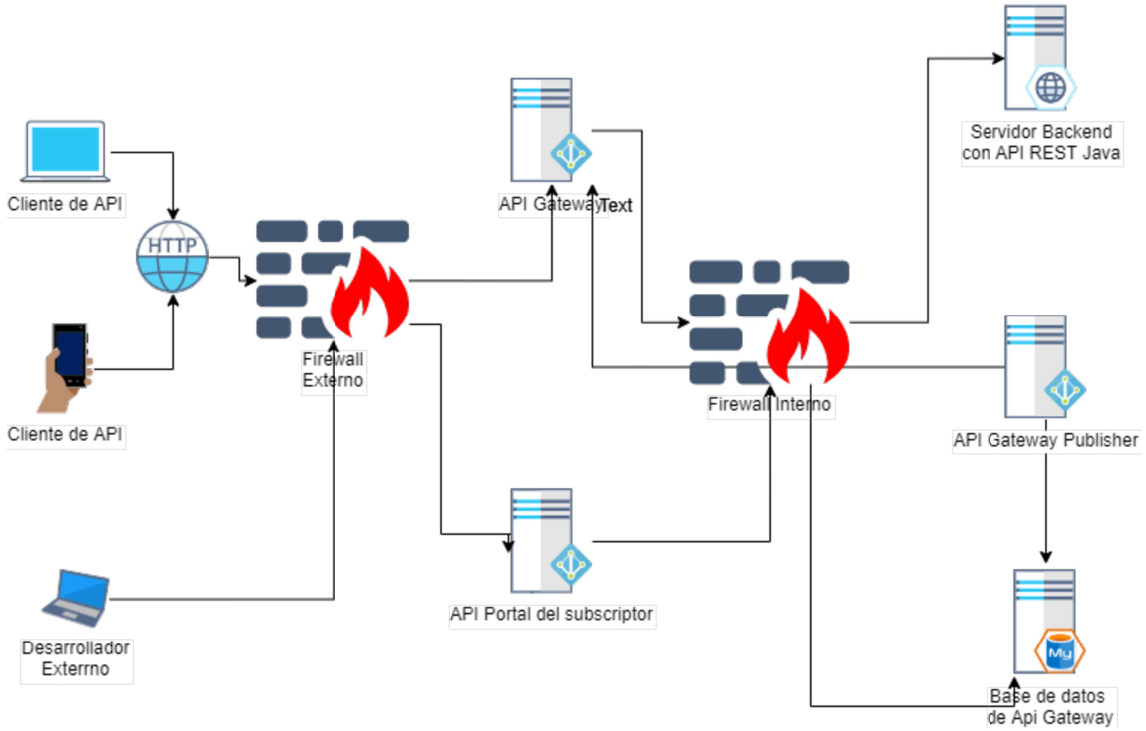

Ilustración 6: Diagrama de arquitectura de solución propuesta

De manera general, podemos resumir en que asume un modelo de capas que cumpla con la arquitectura REST ya que el consumidor y el productor del API REST están desacoplados. La invocación a la misma será bajo HTTP desde internet.

En la parte externa, vamos a tener dos conjuntos de roles, perfiles o uso del Api Manager. El primer conjunto, pertenece al de un consumidor del API REST. Este consumidor ejecutará su petición HTTP en un dispositivo capaz de hacerlo (puede ser un PC con un navegador, una aplicación móvil o un dispositivo IoT) . Dicha petición llegará al módulo de Gateway del API Manager. Este comprobará la URI solicitada y comprobará si tiene algún API que atienda a los parámetros de la petición. En caso de que no sea así, rechazará la petición con un código genérico HTTP. En el caso contrario, encaminará la petición al servidor de backend en el que se encuentra la implementación del API REST y esperará la respuesta de este. Una vez esta se produzca, la encaminará al dispositivo consumidor del API.

Existe un segundo perfil, en este caso el del desarrollador o colaborador que quiere que una aplicación suya consuma un API expuesto. Para ello, accederá desde un PC al portal del suscriptor (en este producto se llama portal del desarrollador), donde podrá, previo registro, ver que APIs REST se exponen, con qué políticas y la documentación de las mismas. Podrá solicitar crear una aplicación. Bajo este concepto, podrá crear unas contraseñas para su aplicación en caso de que el API esté protegida y suscribir su aplicación al consumo de dicha API.

Estos dos elementos estarían en una zona accesible desde el exterior, ya que son los módulos concebidos para consumirse desde internet. El gateway estaría en una zona de DMZ ya que es un elemento puro de ejecución, como puede ser un servidor HTTP cualquiera. El módulo del portal del desarrollador podría ubicarse también en una zona de DMZ o habilitar el acceso al mismo a través de algún tipo de proxy, como se decida. En el diseño he asumido que ambos estarían detrás de un firewall pues son elementos que sería necesario proteger dada su exposición.

Los otros elementos de la arquitectura ya no tendrían que estar expuestos al exterior, por eso los he situado detrás de otro firewall que los separe de la zona de DMZ. Por un lado tendremos el API Publisher en el que un perfil administrador gobernará las APIs REST publicadas en tiempo y forma. Habrá que habilitar la comunicación entre este módulo y el módulo del API Gateway, ya que el Api Publisher enviará las políticas y reglas al Gateway para que éste las aplique.

El servidor de backend que contiene la implementación del API REST y por tanto su lógica de negocio también estaría en esta zona y también habrá que habilitar su conexión con el Gateway. Por último, toda la configuración del producto se almacena en una base de datos que también situaríamos en esta zona interna, aunque será necesario habilitar la comunicación de los módulos de Portal del Desarrollador y Api Publisher con la misma.

Para conseguir una alta disponibilidad en la infraestructura, se podría optar por poner un balanceador de carga delante de todos estos elementos y duplicar la infraestructura al completo, excepto la base de datos que sería común a todos estos módulos.

Esta sería la arquitectura para este tipo de soluciones. Por motivos de plazos temporales y quedar fuera de la posibilidades del alcance de este trabajo, se simplificará la arquitectura haciendo que todos estos módulos residan en la misma máquina y no exista la infraestructura de red ni los firewalls propuestos. Lo que si se instalarán son todos los elementos y se configurarán de acuerdo al ejercicio teórico de que se implantase la solución sobre la arquitectura propuesta.

En lo que se refiere a la API REST, se implementará en lenguaje de programación Java, utilizando un framework o librerías de ayuda a la codificación de APIs REST denominado Spring Boot. Este software se ejecutará sobre una máquina virtual Java sobre un sistema operativo Windows.

.3. Análisis de riesgos y amenazas

El enfoque que tomaré en cuanto a la seguridad de la arquitectura, apuesta por aplicar la misma de manera transversal en el API Gateway. El framework Spring Boot tiene posibilidades de aplicar securización de manera programática y declarativa a los servicios desarrollados. Se pueden cubrir muchos de los requisitos de seguridad de un API REST con las ayudas que aporta el framework. Pero el aplicar la seguridad en el API Gateway frente a aplicarla en código tiene varias ventajas.

- Se aplica la seguridad en un punto común. El nivel de seguridad aplicado es potencialmente el mismo para todos los APIs que se expongan por el Gateway, no dependiendo del nivel de seguridad ejercido en la programación de cada servicio.
- La seguridad es declarativa en la herramienta, no se necesita un programador para aplicar las medidas de seguridad. Se cubren muchos de los estándares de mercado relacionados con la seguridad de APIs.
- El API Gateway ofrece protección para ataques más comunes contra las APIs y parches de seguridad en caso de licenciarlo.
- El API Gateway ofrece la posibilidad de programar medidas personalizadas de protección que pudieran no cubrirse en su versión por defecto.
- Se pueden realizar estrategias complejas de protección utilizando varios API Gateways (estrategia de Microgateways), en los que, según ciertos criterios de clasificación, a unos APIs se les exponga por un API Gateway con ciertas reglas de seguridad y a otros por otro.

Si bien el caso ideal es que las comprobaciones de seguridad se hagan también en el servidor backend, en la arquitectura propuesta vamos a aplicar las políticas de seguridad en la capa del API Gateway. Por otra parte, hay que entender qué ataques contra la lógica de la aplicación en la que el Gateway no pueda intervenir sólo pueden ser controlados en la implementación de la API REST.

Hay soluciones híbridas en las que se aplica un nivel de seguridad en el API Gateway, y otro diferente en la programación de las APIs, más laxo al estar protegido por el Gateway en primer lugar. En estas soluciones se suele proteger mediante algún mecanismo de relación de confianza la comunicación entre el API Gateway y el código fuente de los APIs, de manera que, si bien el grueso de controles los aplica el API Gateway, el código fuente de los APIs sólo aceptará peticiones que provengan del mismo (esto se puede hacer de forma programática, uso de cabeceras acordadas, o protegiendo la comunicación con un canal seguro entre ambos elementos). Los previamente mencionados Micrograteways también son soluciones muy flexibles en cuanto a seguridad.

.3.1 Requisitos de Seguridad

Una API REST no deja de ser un programa informático, de manera que hemos de tener presente que, para poder considerarlo seguro, habrá que aplicar los conceptos generales en seguridad de la información, los cuales agrupamos como requisitos de seguridad. Algunos de estos requisitos son la disponibilidad, confidencialidad, integridad,autenticidad, y trazabilidad en la solución adoptada. A continuación se explica brevemente en qué consisten estos requisitos y se menciona cuales son los mecanismos de seguridad que se utilizarán en la arquitectura propuesta para poder llegar a cumplir todos los requisitos enumerados.

3.1.1 Disponibilidad

En la arquitectura propuesta, la disponibilidad se puede afianzar con medidas como la redundancia de infraestructuras, el balanceo de carga o la aplicación de políticas del control del número de peticiones, ya sea globales o desde un determinado origen. Todas estas medidas serían preventivas para maximizar las probabilidades de que el sistema responda, ya que en muchas ocasiones existen acuerdos de SLA's con consumidores de las APIs o la información proporcionada es crítica para ciertas áreas del negocio de la empresa. En la arquitectura propuesta, utilizaremos las posibilidades que nos aporta el Gateway para proteger la disponibilidad del API. Estas posibilidades son la capacidad de controlar el número de peticiones que se producen y en que lapso de tiempo, para asegurar que ningún cliente consume excesivos recursos que puedan comprometer la disponibilidad del API. El producto también ofrece opciones como el balanceo de carga entre varios nodos, la especificación de un nodo alternativo de respuesta de emergencias o la desactivación temporal del API en caso de que se superen los parámetros de consumo especificados para el mismo.

3.1.2 Confidencialidad.

Consiste en que se ha de asegurar que la información está disponible sólo para los usuarios que estén autorizados a acceder a la misma. En el caso del API REST, en la arquitectura propuesta se garantizaría la confidencialidad aplicando un conjunto de políticas sobre el API, de manera que puede llegar a autorizar el consumo del API sólo a los usuarios autorizados (también sólo a ciertas operaciones, sí se necesita ese nivel de filtrado). Esto se puede conseguir en el API Manager mediante la aplicación de algunos mecanismos de autorización que este soporta, controlando el acceso a la información en base al perfilado del usuario. El producto permite trabajar con algunos estándares de mercado para la autorización como son los flujos OAuth 2.0 y OpenID.

3.1.3 Integridad

La integridad se consigue en una aplicación cuando se puede garantizar que la información que la aplicación maneja se transmite hasta el receptor siendo esta correcta. Además se asegura que la información no es modificada, de manera

que sus datos se mantienen tal cual se generaron, sin modificaciones, manipulaciones o alteraciones por parte de terceros.

El protocolo HTTP tiene en la versión segura del protocolo, HTTPS, capacidades para garantizar la integridad de la información punto a punto. El API REST expuesto mediante el Gateway sólo escuchará peticiones bajo esta versión segura HTTPS y se deshabilitará la escucha de peticiones al API por HTTP.

3.1.4 Autenticidad.

Hay que conseguir que los usuarios puedan acreditar su identidad de una manera confiable. Existen diferentes formas de conseguir esto y el API Manager puede implementarlas. Existe la opción por ejemplo de utilizar HTTPS con certificados cliente hasta el paso de credenciales mediante la cabecera HTTP Authorization. En este trabajo abordaremos esta última ya que es la más común en el marco de los APIs REST. Dentro de esta cabecera Authorization, utilizaremos la generación de Tokens en base a ciclos de autentificación OAuth 2.0. Existen diferentes opciones en la especificación y se escogerá el que mejor aplique al caso propuesto, intentando conseguir mostrar los diferentes casos de uso que cubren.

3.1.5 No repudio.

El no repudio o irrenunciabilidad es un requisito de seguridad que es muy necesario en algunos procesos informáticos y las APIs REST no son una excepción. Bajo este requisito se permiten probar de manera inequívoca la participación de las partes en cierto proceso de la comunicación, de manera que ninguna de ellas pueda negar a posteriori su implicación en el mismo. En el API Manager elegido, se permite el firmado de tokens en la comunicación mediante JWS, de manera que se pueda permitir cumplir con este requisito.

3.1.6 Trazabilidad.

Es necesario que todas las acciones que se producen en el API puedan llegar a ser registradas en el sistema que lo ofrece para poder controlar las posibles amenazas a la seguridad del mismo. Se han de trazar las peticiones así como la información relevante a la seguridad de la comunicación (direcciones IP desde las que se accede, cabeceras de los protocolos utilizados, fecha y hora de cada petición y respuesta, etc). En la solución propuesta de API Manager, existen diferentes ficheros de trazabilidad en el que registrar desde la auditoría de acceso a cada uno de los módulos, hasta el registro de peticiones y respuestas. Se configurará el producto para mostrar estas capacidades asumiendo que no siempre son aplicables en su mayor nivel de profundidad a un entorno productivo por razones de rendimiento. Así mismo, la implementación del API REST, tendrá su trazabilidad en sus propios ficheros de log.

.3.2. Mecanismos de seguridad

Para conseguir cubrir los requisitos de seguridad, hay un conjunto de opciones tecnológicas utilizadas como mecanismos con los que conseguir los objetivos marcados en los requisitos de seguridad. Algunos de estos mecanismos cubren varios requisitos a la vez. Al igual que en otros puntos de este trabajo, las opciones disponibles son muchas, pero se va a proceder a detallar brevemente algunos de los mecanismos de seguridad utilizados en el contexto del caso de uso a resolver.

3.2.1 HTTPS

HTTPS es un protocolo de aplicación seguro de transferencia de hipertexto basado en HTTP, siendo la versión segura de este último. A nivel técnico, usa TLS para crear un canal de cifrado más apropiado para el intercambio de información sensible de lo que es HTTP, en el que el texto compone la información a transferir pasa por los diversos nodos de la capa de transporte como texto plano.

La securización del canal se consigue a través del uso de certificados digitales y criptografía de clave asimétrica. Básicamente, existe la posibilidad de que tanto el sitio web que expone el recurso HTTPS como el programa que lo consume, confíen en una entidad certificadora que es la que expide los certificados digitales que acredita la identidad de los intervinientes. Con TLS, se producirá un cifrado de la comunicación con dichos certificados basada en sus respectivas claves públicas y privadas. Se pueden usar certificados del lado cliente y del lado servidor, siendo el uso del primero opcional y, por razones de simplicidad en la distribución de certificados cliente, el método más habitual hoy por hoy. Se suele cifrar el canal con un certificado de servidor y el cliente presentar algún tipo de credencial (usuario y contraseña en muchos casos) para acreditar su identidad.

HTTPS permite cubrir varios requisitos de seguridad de los anteriormente expuestos (autenticidad, confidencialidad, integridad) y está soportado en todas sus variantes (uso de certificados cliente y servidor o sólo servidor) por el API Manager WSO2 bajo el que se expondrá el API en este trabajo.

3.2.2 JWT, JWS y JWE

En los últimos años, los APIs REST se han tenido que enfrentar a diversos desafíos en cuanto al cumplimiento de los requisitos de seguridad. Si bien algunos están resueltos por HTTPS, lo cierto es que a veces las necesidades exceden lo que HTTPS puede hacer o, simplemente, hacerlo con HTTPS origina soluciones complejas (como puede ser la logística de la distribución de certificados cliente, por ejemplo). Dada la creciente necesidad de que los APIs cumpliesen todos los requisitos de seguridad y la dificultad de proporcionar algunos de ellos bajo HTTPS, hay varias tecnologías en las que apoyarse para conseguirlo. Por ejemplo, en el ámbito de la autenticidad, es bastante común el

uso de tokens de seguridad que se proporcionan a los clientes para que sean traspasados en todas las comunicaciones. Estos tokens pueden tener cierta estructura y, una solución muy extendida, es el uso de JTW, un estándar abierto basado en lenguaje JSON propuesto por el IETF bajo la especificación RFC 7519. Bajo estos tokens JWT, se permite la propagación de identidad (identificador y posibles datos del usuario) y privilegios en el API(los denominados claims). Bajo esta especificación se define el formato del token para ser válido, se proponen secciones en su estructura y trata de cubrir con ellas un variado número de escenarios. Algunas de las secciones sirven para dotar a esta tecnología de token de mecanismos de firma digital y algoritmos de cifrado de la información. Esto se consigue apoyándose en otros estándares también de lenguaje JSON, que son JWS y JWT.

Con JWS se puede otorgar al token la posibilidad de ser firmado digitalmente, de tal manera que se obtengan algunos requisitos de seguridad con esta acción como la integridad, la autenticidad o el no repudio.

Con JWE se puede cifrar digitalmente el contenido de los tokens con un variado número de algoritmos de criptografía simétrica y asimétrica, de manera que se consiga el requisito de la confidencialidad en la comunicación.

El uso de tokens JWT, ya sea firmados o cifrados, se ha extendido de manera muy importante en las soluciones de securización de APIs REST. Suelen combinarse con HTTPS para que, con todos estos mecanismos, se pueda cumplir con todos los requisitos de seguridad necesarios para el uso seguro de un API.

3.2.3 Mecanismos de autentificación y autorización en APIs REST

Hasta este punto, se han descrito de manera breve los mecanismos de seguridad que permiten cumplir con los requisitos de seguridad necesarios. Ahora bien, dentro de estos mecanismos, existen diferentes maneras o formas de que, utilizando las tecnologías previamente expuestas, se facilite la posibilidad de que los intervinientes acrediten su identidad y se otorguen permisos en base a la misma.

Hay que hacer hincapié en la diferencia que existe entre autentificación y autorización, pues en ocasiones se entremezclan ya que un mismo mecanismo de seguridad permite cubrir ambas características.

La autentificación es el proceso o acción de verificar la identidad de un usuario o proceso y se basa en proporcionar un conjunto de credenciales para que el servidor o aplicación pueda verificar que cada cual pueda comprobar que el otro extremo de la comunicación es quien dice ser y que la información que se proporciona es válida.

La autorización parte del proceso de autentificación, extendiéndolo y garantiza que, una vez que se determine que las credenciales de usuario son correctas,

se aplicará el control de acceso para determinar a qué permisos sobre los recursos se tiene en el sistema. En el caso que nos ocupa, se asignan privilegios de acceso a los recursos de un API en función de un perfilado que el mismo adquiere. El perfilado tiene cierta granularidad apoyándose en la arquitectura REST, de manera que es posible autorizar al uso de ciertas operaciones de las expuestas en el API y denegar otras.

Hay muchos mecanismos de seguridad en el mercado, y muchas formas de autentificarse y autorizarse frente al consumo de un API REST. En este trabajo los mecanismos elegidos son HTTPS, JWS y JWE ,que sirven para estos fines y suelen combinarse entre ellos en función del escenario. Estos escenarios pueden incluir de multitud de factores, como el tipo de dispositivo, el tipo de uso del mismo, el nivel necesario de confidencialidad de la información, el tipo de operación a la que se accede, etc. Lo cierto es que no hay una sola manera de asegurar un API usando estos mecanismos. Es por ello que se van a tratar de exponer algunos de los principales métodos existentes.

Los 4 métodos principales de autentificación APIs REST son:

Autentificación básica
Autentificación basada en token
Autentificación basada en clave API
OAuth 2.0 (Autorización abierta)

Todos estos métodos de autenticación expuestos están soportados por el producto API Manager elegido para implementar la parte práctica del trabajo. Se comentará en la misma como aplicarlos al API REST que se va a proteger.

Hay un detalle muy importante en el flujo de información que tiene que ver con la propia arquitectura REST, en la que no existe estado. Por lo tanto, no hay un mecanismo de sesión que resida en el propio protocolo. Esto se traduce en que, de una u otra manera, en todas las peticiones el consumidor tendrá que presentar una credencial y, para todas las peticiones, el productor tendrá que validarlas. Frente a otros protocolos o arquitecturas que sí que albergan estado, es un aspecto a tener muy en cuenta y frente al que se pueden implementar diferentes soluciones.

3.2.3.1 Autentificación básica

Es la forma más sencilla de autenticar a un consumidor frente al API. Se utiliza la cabecera HTTP Authorization en la que se codifica en BASE64 el identificador de usuario y su contraseña. Este método no cubre la autorización basada en esta cabecera, el usuario tiene acceso a todo el API o no lo tiene. Es el método más simple y más antiguo. Como principal desventaja tiene el que el identificador de usuario y su contraseña viaja en todas las peticiones. En caso de que se capture una, esas credenciales podrían llegar a valer en otros puntos de la infraestructura. Si el identificador es el mismo para el API que, imaginemos, un portal del empleado, el atacante tendría credenciales para acceder a este.

3.2.3.2 Autentificación basada en token

Al igual que en el método anterior, el usuario adjunta sus credenciales en la cabecera Authorization. Ahora bien, sólo lo hace una vez, al hacer la autenticación frente al API. Como respuesta de esta autenticación, se proporcionará un Token. Este token es el que posteriormente se adjuntará en todas las peticiones posteriores en la cabecera HTTP Authorization. Al ser un Token, esto aporta muchas ventajas frente al método anterior.

El Token puede ir firmado e incluso cifrado, de manera que se puedan pasar al cliente ciertos parámetros propios de la autenticación y autorización. Se puede introducir información de usuario y perfilado del mismo en el Token, de manera que no se pueda cambiar y no haya que volver a hacer los controles en cada petición. Lo normal es que al Token se le de un tiempo de vida, de manera que en caso de captura del mismo, sólo sería válido antes de su fecha de expiración. Además, la autenticación basada en Tokens permite centralizar la autenticación y autorización a todas las APIs en un único URI, al que se le pasan las credenciales y haría las funciones de IDP, esto es, habría un único punto de autenticación y autorización para todas las APIs de un proveedor, con lo que no habría que implementar la generación del token en todos los servicios de manera individual.
Este método está muy extendido por sencillez y rendimiento. Alrededor del mismo han surgido algunas estandarizaciones del contenido de los Tokens como son JWT y OpenID, de manera que se pueda tener un formato uniforme y sea más sencillo que las aplicaciones que consumen APIs puedan acceder a la información de los Tokens.

3.2.3.3 Autentificación basada en clave API

Frente a los dos métodos anteriores, en este se prescinde de la autenticación en el proceso del consumo del API. Esta se realiza antes. Lo que se hace es, ya sea en una interfaz de usuario al efecto o mediante el intercambio de un secreto compartido entre cliente y servidor, generar una clave única de aplicación. Este concepto es el que se suele aplicar en las comunicaciones B2B, donde existe una aplicación que se subscribe a un servicio mediante una credencial y un API que autentica y autoriza el consumo del mismo por esa clave o key de aplicación.

Este método es más seguro que los anteriores pues el secreto compartido sólo se genera una vez, luego el proceso de autenticación se saca del ciclo general de peticiones, sólo queda la autorización. A cambio, no suele caducar hasta que se crea una nueva credencial compartida (en caso de robo de key, puede valer mucho tiempo), y suele exigir de productos de tipo API Gateway en el lado del servidor, pues si no el mantenimiento de las keys puede ser muy laborioso y la seguridad de los mismos puede verse seriamente comprometida. Además, el escenario C2B es muy difícil de implementar en este método.

3.2.3.4 Autentificación basada en OAuth 2.0

Este se ha convertido en el método estándar de facto para la autenticación y autorización de APIs REST, mezclando soluciones aplicadas en los dos

métodos anteriores y algunas más para cubrir muchos casos de uso. Es el método implementado por todas las grandes compañías de internet para el consumo de sus APIs. Fue pensado para cubrir varios escenarios o Flows, dependiendo del tipo de aplicación a la que se proporciona la credencial y cómo interactúa esta con el API y con el usuario. Cubre escenarios de B2B y C2B, con lo que es la solución más completa, aunque también la más compleja y la que necesita de un mayor número de elementos. En la siguiente figura se especifica de manera general cual es el flujo de peticiones a la hora de autenticar y autorizar a una aplicación cliente el consumo de una API.

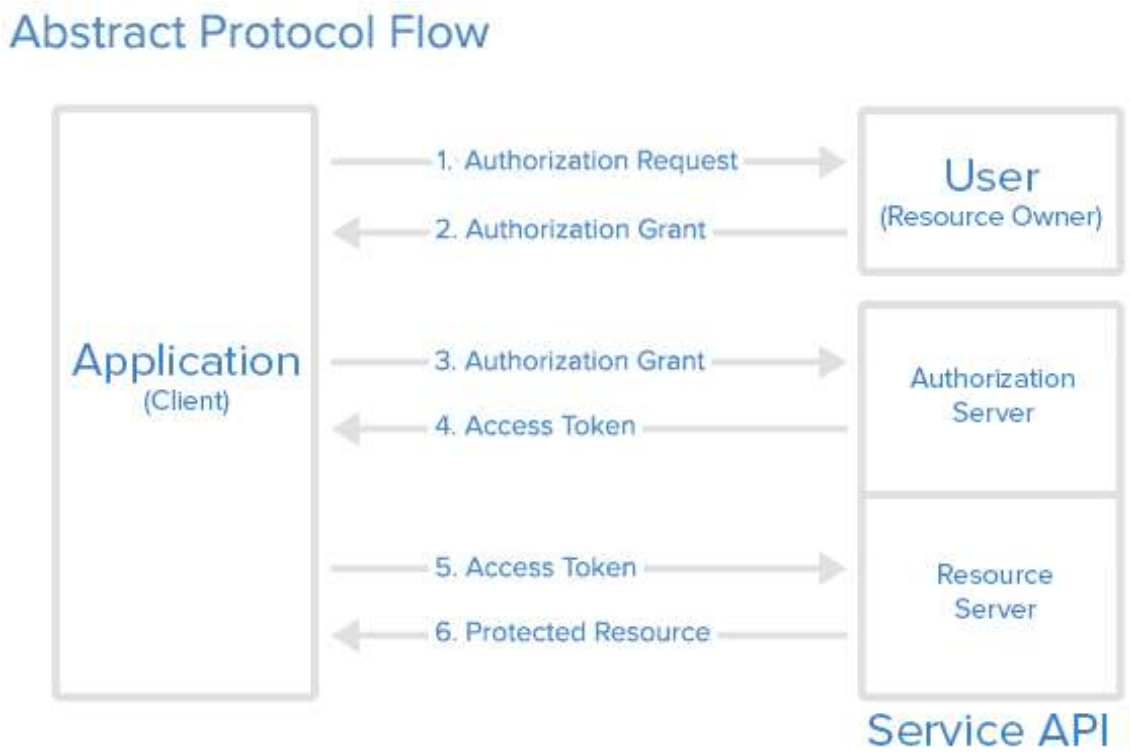

Ilustración 7: Flujo genérico de autentificación mediante OAuth

En la figura anterior se describe un posible flujo de autenticación en el que intervienen una serie de actores. Estos son los siguientes:

- Application. Es la aplicación que consumirá el API. No confundir con el usuario, que es el propietario de los datos. La aplicación lo que necesita es consumir un API en nombre del usuario, sin poseer la credencial de la cuenta del mismo.

- User: es el usuario de la aplicación. Ha de poseer una credencial frente al authorization server, ya que se le solicitará en el proceso. El usuario será redirigido al autorization Server una vez reciba la solicitud de Grant por parte de la aplicación.

- Authorization Server: es el IDP, pero es un punto de autenticación único (URI de autentificación). Frente a este punto de autenticación único el usuario presentará sus credenciales, el clientID de la aplicación a la que

permite el acceso al API y el grant o flujo de autenticación OAuth que se quiere utilizar. Si este proceso es exitoso, se generará un token OAuth por parte del Authorization Server que será devuelto al usuario y este será redirigido a la aplicación para esta haga uso de dicho token.

- Resource Server: es la URI de la API Rest a consumir por la aplicación en nombre del usuario. Para poder acceder a esa URI, hay que presentar el token obtenido en el paso anterior.

OAuth es un estándar muy completo, cubre muchos escenarios. En función del mismo, hay algunas entidades de las descritas que pueden intervenir o no en el ciclo de autenticación y autorización. En el caso que se va a resolver en esta práctica, el escenario sería el de una aplicación que en modelo B2B pretende consumir un API. En este caso, las credenciales están en posesión de la aplicación ya de origen, pues es ella la que genera el Token. Por ejemplo, en API Manager, esto lo hace en el portal del desarrollador. Este flujo se le denomina Client Credentials en OAuth, y en él intervienen la aplicación, el authorization server y el resource server, pero no el usuario (en realidad el usuario es el propio desarrollador, solamente que la fase de autenticación frente al IDP con su identificador de usuario y contraseña se realiza fuera del flujo OAuth, el el módulo del portal del desarrollador).

En general, en todos los flujos OAuth existen unos parámetros que se intercambian entre los diferentes actores y que son los siguientes:

- ClientID: identificador de la aplicación que quiere hacer uso del API. Este identificador es proporcionado por el Authorization Server y se crea en el proceso de registro de la aplicación en el mismo (en el ejemplo tratado, al registrar la aplicación en el API Manager). No cambiaría nunca.

- Grant: Tipo de permiso o flujo OAuth que se pretende ejecutar. En este caso, pasaría el valor client_credentials.

- Client Secret: cuando el desarrollador crea id de aplicación, se le crea este identificador a modo de secreto compartido entre la aplicación y el authorization server. A diferencia del ClientID, podría solicitarse su regeneración en caso de que se piense que puede haber sido vulnerado.

- Scope: tipo de permiso que se solicita tener frente al API. Es decir, es el parámetro que servirá para la autorización.

Cada vez que se quiera consumir el API y no se disponga de un token para ello, habrá que enviar al Authorization Server estos 4 parámetros. Este responderá con un Token. En función de la configuración que se aplique, este token puede contener información (Token JWT) o puede ser un token opaco (no traslada información, solamente es una cadena de caracteres que hace las veces de Token). Ese Token, en función del tipo de flujo Oauth, tendrá una caducidad. Hay que destacar que en OAuth hay dos tipos de Tokens en función de la caducidad.

1. Authorization Token. Se genera cuando se reciben en el Authorization Server los parámetros necesarios para un flujo de autenticación correcta. Puede o no tener una caducidad temporal y puede tener formato y contenido propio (por ejemplo, un Token JWT con datos como el ID de usuario, firmado con JWS y cifrado con JWE) o ser un Token opaco). Se ha de pasar siempre al Resource Server cuando se quiere acceder a consumir un recurso del API.

2. Refresh Token. Este token es siempre opaco. Se puede construir por parte del Authorization Server en caso de que se permita su utilización. Su función es la de ser un Token sin caducidad (aunque si es revocable). Básicamente, cuando se ha autenticado la aplicación y obtenido este Token de refresco, la aplicación lo puede guardar si tiene estructura para ello (cookies, memoria de la aplicación, etc). Si la aplicación no dispone aun de un Authorization Token o este está caducado, puede mandar este token de refresco al Authorization Server. Si es válido, el Authorization Server devolverá un Authorization Token nuevo a la aplicación para que lo use a partir de entonces para acceder al API del Resource Server. El propósito de todo esto es que cada vez que caduque un Authorization Token, no se tengan que pasar de nuevo todos los parámetros de la autenticación original, exponiendo los valores de los mismos. Estos sólo se pasan una vez y luego se usa permanentemente el Refresh Token para usos posteriores.

   Es importante destacar que el token de refresco no sirve para autenticarse contra el resource server. Con el nunca se podrá consumir el API, ya que el resource server sólo reconoce Tokens de tipo Authorization Token.

.3.3. Riesgos y Amenazas contra APIs Rest

Con la cantidad de tráfico que soportan los APIs (algunos estudios hablan de más del 80% del tráfico de datos de internet), el número de ataques sobre los mismos también es muy alto. Contra más volumen de tráfico se soporta, más ataques se producen. La consultora Gartner prevé que en 2022 las vulnerabilidades que presenten las APIs serán el vector de ataque más frecuente al que se enfrenten las organizaciones. Teniendo en cuenta que las APIs REST son el tipo de API más extendido actualmente en internet con mucha diferencia, las amenazas contra este tipo de software crece año a año.

A continuación se muestra una lista de las principales brechas de seguridad en 2020 relacionadas con las APIs. Como se puede ver, encontramos que muchos de los gigantes de internet están entre las víctimas.

- Ledger Customer Data Breach
- YouTube API flaws
- Twitter Fleet APIs allow Access to Older Tweets
- Tesla Backup Gateway APIs (multiple vulnerabilities)
- Prestige Software S3 API access exposure
- AWS Resource-Based Policy APIs had information leakage

- Thrillophilia API Flaw Exposing Millions of Customer records
- Waze Allowing Random User Tracking
- Gitlab Backdoor API Exposes Private Projects
- Shopify Insider Leveraged Order APIs to Obtain Millions of Customer records
- Mercedes Benz Car Control Vulnerabilities (multiple)
- MGM Grand hotel and casino data breach
- Cisco Data Center Network Manager Authentication Bypass Vulnerability
- Google Analytics API used as Attack Vector to exfiltrate data
- Google Firebase blunder exposes data through estimated 24,000 Android apps
- Account Takeover Vulnerability in Microsoft Teams
- Personal data of 1.41m US doctors sold on hacker forum
- Third party app used by top EU merchants exposed 8 million sales records
- Facebook Lets Any User Change the User Name of Any Facebook Page
- CapitalOne Fined 80 Million

Los ataques tienen objetivos muy diversos, desde financieros, datos sanitarios, control sobre dispositivos IoT como pueden ser coches o grandes servicios de internet como tiendas de aplicaciones. Queda claro que la protección de los APIs es una necesidad imperiosa y que no hay compañías que estén exentas de ser un objetivo potencial de ataques.

Dentro de las recomendaciones frente a ataques, la fundación OWASP tiene como objetivo canalizar los esfuerzos en la seguridad de aplicaciones informáticas. Las APIs están entre ellas, como es natural. Anualmente, elabora un ranking de los 10 riesgos de seguridad de APIs más críticos en la web. Estos serían los correspondientes a 2019 por orden de importancia.

1. Autorización a nivel de objeto. El atacante recibe cierta información como identificadores de objeto que luego altera el peticiones posteriores con el objetivo de acceder a datos para los que no está autorizado.

2. Autenticación de usuario. En ocasiones el mecanismo de autenticación no es correcto o, en los casos más extremos, directamente no existe, es decir, hay APIs que son de acceso público cuando no deberían serlo por los datos que potencialmente exponen. Aquí los ataques son diversos, desde conseguir Tokens de otros usuarios, vulnerar el flujo de autenticación o errores en la expedición de Tokens como la no caducidad de los mismos o recuperaciones incorrectas de Tokens en procesos de olvido o recuperación de contraseñas.

3. Exceso de datos expuestos. Muchas veces los APIs exponen objetos completos derivados de una mayor simplicidad en el desarrollo de las APIs y se proporciona acceso a información indebida.

4. Limitación de peticiones y recursos. Hay veces que la necesidad hace que un API sea de acceso público, que no se tenga necesidad de autenticación para acceder a la funcionalidad de la misma, pero eso no implica que no haya que controlarla. Si no se controla el número de peticiones desde un mismo origen o a un mismo recurso, se puede

provocar que, con pocos recursos por parte del atacante, se pueda conseguir la denegación del servicio. Otros posibles controles son los tiempos máximos de respuesta, tamaños del peticiones o volumen de las respuestas, si no se implementa una paginación, por ejemplo.

5. Autorización a nivel de operación. Las APIs se exponen, pero no para el mismo nivel de acceso a todos los usuarios. Si no se implementa una autorización de acceso a las operaciones, se puede provocar que un usuario que tiene permisos legítimos de operaciones de consulta, por ejemplo, cambie las peticiones para pedir otra función (en REST es particularmente sencillo, a veces sólo cambia el verbo HTTP sobre la misma URI), y ejecute una función para la que no está autorizado.

6. Asignación directa de datos. En los frameworks de programación, para facilidad de uso de los mismos por parte del desarrollador, en muchas ocasiones se hacen asociaciones directas entre los objetos que vienen en las peticiones, con objetos de negocio de la aplicación. Esto es muy cómodo cuando se trabaja con objetos con muchos parámetros. Esto puede provocar que en ocasiones, datos que sean actualizables en una petición lícita, se mezclan con otros que no deberían poder actualizarse. Si no hay un buen control de la asociación entre objetos de la petición y los objetos de negocio, se puede conseguir escribir información en los objetos de negocio de la aplicación sin que se debiera poder hacerlo.

7. Configuración de seguridad deficiente. Los atacantes encuentran un defecto en la configuración de la seguridad del API en alguno de sus puntos. Estos defectos pueden ser, por ejemplo, parches no actualizados, no deshabilitar verbos HTTP que no se trabajan, no usar HTTPS, permitir peticiones de dominio cruzado o CORS, información excesiva en caso de errores, etc. Las consecuencias de dichos fallos de configuración pueden ser muy diversas, desde acceso a datos no permitidos, suplantación de usuarios, etc.

8. Inyección de código. Si no se filtran los parámetros que se reciben, estos pueden alterarse para construir sentencias o comandos en la parte del servidor backend que tengan consecuencias no deseadas, desde pérdidas de información, hasta escaladas de privilegios en el sistema. Los atacantes modifican los parámetros de las peticiones para incluir entre ellos sentencias SQL, LDAP, comandos de sistema operativo, etc, que, si no se realizan los filtros adecuados, pueden terminar ejecutándose en el servidor de forma fraudulenta.

9. Manejo incorrecto de activos. Las APIs evolucionan con rapidez en algunos escenarios, de manera que pueden estar vigentes varias versiones a la vez en un entorno productivo. Si no hay una política de retiro de versiones antiguas, es fácil que las vulnerabilidades detectadas sobre las versiones actuales no se corrijan en las versiones antiguas, ya sea porque tienen un uso residual o porque no se tiene conciencia de que nadie las está utilizando. Si no se tiene control sobre el ciclo de vida de las versiones de los APIs, se pueden utilizar por parte del atacante

versiones antiguas no parcheadas para explotar vulnerabilidades que se creían corregidas en las nuevas versiones.

10. Trazas o monitorización insuficiente. Si no se tienen alertas de motorización sobre usos poco frecuentes es muy difícil saber cómo están intentando atacar tu sistema. Igualmente, si no hay un nivel de trazabilidad correcto en las APIs, en caso de sufrir un incidente de seguridad, será muy difícil saber cómo se ha comprometido el sistema y cúal es la profundidad de la intrusión.

En esta clasificación están representados casos específicos de los que están considerados los cuatros principales tipos de vulnerabilidad que tiene los APIs: denegación de servicio, inyección de código fuente, robo de credenciales, escalada de privilegios y ataques de suplantación o intermediación de la comunicación.

En este trabajo, vamos a abordar cómo se podrían paliar estos ataques con la arquitectura propuesta. Algunos se podrán abordar con la herramienta de API Manager que vamos a utilizar en la protección del API REST expuesto, y otros no. Será necesario controlarlos en la codificación. Por razones de alcance del trabajo, se mostrará de manera práctica como se pueden paliar aquellos en los que mediante la configuración del API Gateway podamos obtener resultados y se mostrarán los mismos. Aquellas vulnerabilidades de las nombradas que queden en el control por parte del desarrollador del servicio backend del API, no se realizará una demostración práctica de su control.

.4. Implementación del caso de uso

Una vez asumidos los conceptos y vistos los posibles escenarios de amenazas a una API Rest, se procederá a poner a prueba la solución propuesta, consistente en la protección de un API REST mediante el producto de API Gateway WSO2 API Manager. Para ello, se construirá un API REST de prueba y se intermediará el acceso a la misma mediante el API Gateway. El API que se va a implementar, es un ficticio API de Pedidos, que ofrecerá las operaciones CRUD básicas para poder poner a prueba los principales verbos HTTP que se suelen utilizar en REST y tener diferentes niveles de acceso a los mismos en función del perfilado de la aplicación consumidora que lo invoque.

.4.1 Implementación del API REST

Se puede implementar el API REST con multitud de tecnologías o lenguajes de programación. Para el caso de uso se va a utilizar un conjunto de ellas. En la arquitectura de la solución, se va a hacer uso de las siguientes tecnologías, lenguajes de programación y productos.

- Como lenguaje de programación, se usará JAVA. Así que la solución implica que hay que instalar una JVM para ejecutar el programa resultante, en este caso, se ha optado por la OpenJDK, versión 11.0.8. Para compilarlo usaremos este mismo producto.

- Dentro de las múltiples librerías que desde JAVA se ofrecen para la ayuda a la codificación de APIs REST, se ha optado por utilizar Spring Boot. Este es un framework de trabajo muy extendido en la comunidad java, que ofrece múltiples librerías para la ayuda a la codificación de soluciones en lenguaje JAVA. En este caso de uso, se ha decidido utilizar el IDE de desarrollo basado en Eclipse, Spring Tool Suite, que ofrece una serie de opciones para el trabajo con Spring Boot que facilitan la tarea. Entre estas opciones destacan el uso de la gestión de dependencias basada en Apache Maven y el servidor de aplicaciones embebido en Spring Boot Apache Tomcat.

- Para la persistencia de los datos que guarda la API, se va a utilizar como base de datos relacional MySQL. Para el acceso a la misma desde la API, se utilizará la librería JDBC JAVA que aporta el fabricante.

- Para hacer las pruebas sobre la API, se va a utilizar el producto SOAPUi de SmartBear, que permite la implementación de casos de prueba sobre el API y sus operaciones con múltiples opciones.

4.1.1 Especificación de la API

Como se comentó previamente, se va a utilizar el estándar OpenAPI 3.0 como lenguaje descriptivo del API. Básicamente, se han descrito un conjunto de

operaciones CRUD para permitir interactuar con los Pedidos que el API gestiona. Así, se han creado operaciones para la consulta de todos los pedidos, la consulta de uno específico, la creación de un pedido, actualización de datos del mismo o el borrado de un pedido. Todas estas operaciones se han ubicado bajo una misma URI como la arquitectura REST propone y se han utilizado los verbos HTTP para darle significado semántico a las operaciones sobre una misma URI. El contenido del archivo quedaría así en YAML.

```
swagger: '2.0'
info:
  description: Api para la gestion de pedidos.
  version: '1.0'
  title: Api Pedidos
  termsOfService: 'urn:tos'
  contact: {}
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0'
host: 'localhost:8080'
basePath: /pedidos
tags:
  - name: pedidos API Rest
    description: API Rest para la gestion de pedidos REST
security:
  - default: []
paths:
  /:
    get:
      tags:
        - pedidos-rest
      summary: getPedidos
      operationId: getPedidosUsingGET
      produces:
        - '*/*'
      parameters: []
      responses:
        '200':
          description: OK
          schema:
            type: array
            items:
              $ref: '#/definitions/Pedido'
        '401':
          description: Unauthorized
        '403':
          description: Forbidden
        '404':
          description: Not Found
      security:
        - default: []
      deprecated: false
      x-auth-type: Application & Application User
      x-throttling-tier: Unlimited
      x-wso2-application-security:
        security-types:
          - oauth2
        optional: false
    post:
      tags:
        - pedidos-rest
      summary: creaPedido
      operationId: creaPedidoUsingPOST
      consumes:
        - application/json
      produces:
        - '*/*'
      parameters:
        - in: body
          name: pedido
          description: pedido
          required: true
```

```
        schema:
          $ref: '#/definitions/Pedido'
    responses:
      '200':
        description: OK
        schema:
          $ref: '#/definitions/Pedido'
      '201':
        description: Created
      '401':
        description: Unauthorized
      '403':
        description: Forbidden
      '404':
        description: Not Found
    security:
      - default: []
    deprecated: false
    x-auth-type: Application & Application User
    x-throttling-tier: Unlimited
    x-wso2-application-security:
      security-types:
        - oauth2
      optional: false
'/{idPedido}':
  get:
    tags:
      - pedidos-rest
    summary: getPedidoById
    operationId: getPedidoByIdUsingGET
    produces:
      - '*/*'
    parameters:
      - name: idPedido
        in: path
        description: idPedido
        required: true
        type: integer
        format: int64
    responses:
      '200':
        description: OK
        schema:
          $ref: '#/definitions/Optional«Pedido»'
      '401':
        description: Unauthorized
      '403':
        description: Forbidden
      '404':
        description: Not Found
    security:
      - default: []
    deprecated: false
    x-auth-type: Application & Application User
    x-throttling-tier: Unlimited
    x-wso2-application-security:
      security-types:
        - oauth2
      optional: false
  put:
    tags:
      - pedidos-rest
    summary: actualizaPedido
    operationId: actualizaPedidoUsingPUT
    consumes:
      - application/json
    produces:
      - '*/*'
    parameters:
      - name: idPedido
        in: path
        description: idPedido
        required: true
        type: integer
```

```
          format: int64
        - in: body
          name: pedido
          description: pedido
          required: true
          schema:
            $ref: '#/definitions/Pedido'
      responses:
        '200':
          description: OK
          schema:
            $ref: '#/definitions/Pedido'
        '201':
          description: Created
        '401':
          description: Unauthorized
        '403':
          description: Forbidden
        '404':
          description: Not Found
      security:
        - default: []
      deprecated: false
      x-auth-type: Application & Application User
      x-throttling-tier: Unlimited
      x-wso2-application-security:
        security-types:
          - oauth2
        optional: false
    delete:
      tags:
        - pedidos-rest
      summary: borraPedido
      operationId: borraPedidoUsingDELETE
      produces:
        - '*/*'
      parameters:
        - name: idPedido
          in: path
          description: idPedido
          required: true
          type: integer
          format: int64
      responses:
        '200':
          description: OK
        '204':
          description: No Content
        '401':
          description: Unauthorized
        '403':
          description: Forbidden
      security:
        - default: []
      deprecated: false
      x-auth-type: Application & Application User
      x-throttling-tier: Unlimited
      x-wso2-application-security:
        security-types:
          - oauth2
        optional: false
securityDefinitions:
  default:
    type: oauth2
    authorizationUrl: 'https://test.com'
    flow: implicit
definitions:
  ModelAndView:
    type: object
    properties:
      empty:
        type: boolean
      model:
        type: object
```

```
  properties: {}
modelMap:
  type: object
  additionalProperties:
    type: object
    properties: {}
reference:
  type: boolean
status:
  type: string
  enum:
    - 100 CONTINUE
    - 101 SWITCHING_PROTOCOLS
    - 102 PROCESSING
    - 103 CHECKPOINT
    - 200 OK
    - 201 CREATED
    - 202 ACCEPTED
    - 203 NON_AUTHORITATIVE_INFORMATION
    - 204 NO_CONTENT
    - 205 RESET_CONTENT
    - 206 PARTIAL_CONTENT
    - 207 MULTI_STATUS
    - 208 ALREADY_REPORTED
    - 226 IM_USED
    - 300 MULTIPLE_CHOICES
    - 301 MOVED_PERMANENTLY
    - 302 FOUND
    - 302 MOVED_TEMPORARILY
    - 303 SEE_OTHER
    - 304 NOT_MODIFIED
    - 305 USE_PROXY
    - 307 TEMPORARY_REDIRECT
    - 308 PERMANENT_REDIRECT
    - 400 BAD_REQUEST
    - 401 UNAUTHORIZED
    - 402 PAYMENT_REQUIRED
    - 403 FORBIDDEN
    - 404 NOT_FOUND
    - 405 METHOD_NOT_ALLOWED
    - 406 NOT_ACCEPTABLE
    - 407 PROXY_AUTHENTICATION_REQUIRED
    - 408 REQUEST_TIMEOUT
    - 409 CONFLICT
    - 410 GONE
    - 411 LENGTH_REQUIRED
    - 412 PRECONDITION_FAILED
    - 413 PAYLOAD_TOO_LARGE
    - 413 REQUEST_ENTITY_TOO_LARGE
    - 414 URI_TOO_LONG
    - 414 REQUEST_URI_TOO_LONG
    - 415 UNSUPPORTED_MEDIA_TYPE
    - 416 REQUESTED_RANGE_NOT_SATISFIABLE
    - 417 EXPECTATION_FAILED
    - 418 I_AM_A_TEAPOT
    - 419 INSUFFICIENT_SPACE_ON_RESOURCE
    - 420 METHOD_FAILURE
    - 421 DESTINATION_LOCKED
    - 422 UNPROCESSABLE_ENTITY
    - 423 LOCKED
    - 424 FAILED_DEPENDENCY
    - 425 TOO_EARLY
    - 426 UPGRADE_REQUIRED
    - 428 PRECONDITION_REQUIRED
    - 429 TOO_MANY_REQUESTS
    - 431 REQUEST_HEADER_FIELDS_TOO_LARGE
    - 451 UNAVAILABLE_FOR_LEGAL_REASONS
    - 500 INTERNAL_SERVER_ERROR
    - 501 NOT_IMPLEMENTED
    - 502 BAD_GATEWAY
    - 503 SERVICE_UNAVAILABLE
    - 504 GATEWAY_TIMEOUT
    - 505 HTTP_VERSION_NOT_SUPPORTED
    - 506 VARIANT_ALSO_NEGOTIATES
```

```
          - 507 INSUFFICIENT_STORAGE
          - 508 LOOP_DETECTED
          - 509 BANDWIDTH_LIMIT_EXCEEDED
          - 510 NOT_EXTENDED
          - 511 NETWORK_AUTHENTICATION_REQUIRED
      view:
        $ref: '#/definitions/View'
      viewName:
        type: string
    title: ModelAndView
  Optional«Pedido»:
    type: object
    properties:
      empty:
        type: boolean
      present:
        type: boolean
    title: Optional«Pedido»
  Pedido:
    type: object
    properties:
      detalles:
        type: string
      direccion:
        type: string
      id:
        type: integer
        format: int64
      nombre:
        type: string
    title: Pedido
  View:
    type: object
    properties:
      contentType:
        type: string
    title: View
x-wso2-auth-header: Authorization
x-wso2-cors:
  corsConfigurationEnabled: false
  accessControlAllowOrigins:
    - '*'
  accessControlAllowCredentials: false
  accessControlAllowHeaders:
    - authorization
    - Access-Control-Allow-Origin
    - Content-Type
    - SOAPAction
    - apikey
    - testKey
  accessControlAllowMethods:
    - GET
    - PUT
    - POST
    - DELETE
    - PATCH
    - OPTIONS
x-wso2-production-endpoints:
  urls:
    - 'http://localhost:8080/pedidos'
  type: http
x-wso2-basePath: /pedidos/1.0.0
x-wso2-transports:
  - http
  - https
x-wso2-application-security:
  security-types:
    - oauth2
  optional: false
x-wso2-response-cache:
  enabled: false
  cacheTimeoutInSeconds: 300
```

Se puede apreciar que existen las siguientes secciones:

- Info, donde se detalla desde la descripción del Api, hasta el modo de licenciamiento de la misma.
- Paths, es la sección donde se indica que URI o URIS utiliza el API, especificando el formato de la misma y que verbos escucha. Para cada verbo, se detallan los posibles códigos de respuesta HTTP y el contenido de la salida de los mismos.
- Definitions, donde se detallan los tipos de datos de las entidades tratadas por el modelo de datos del API

Dentro de la página web de la especificación OpenAPI (https://swagger.io/tools/open-source/), existe la posibilidad de obtener una representación gráfica del YAML. En este caso quedaría así:

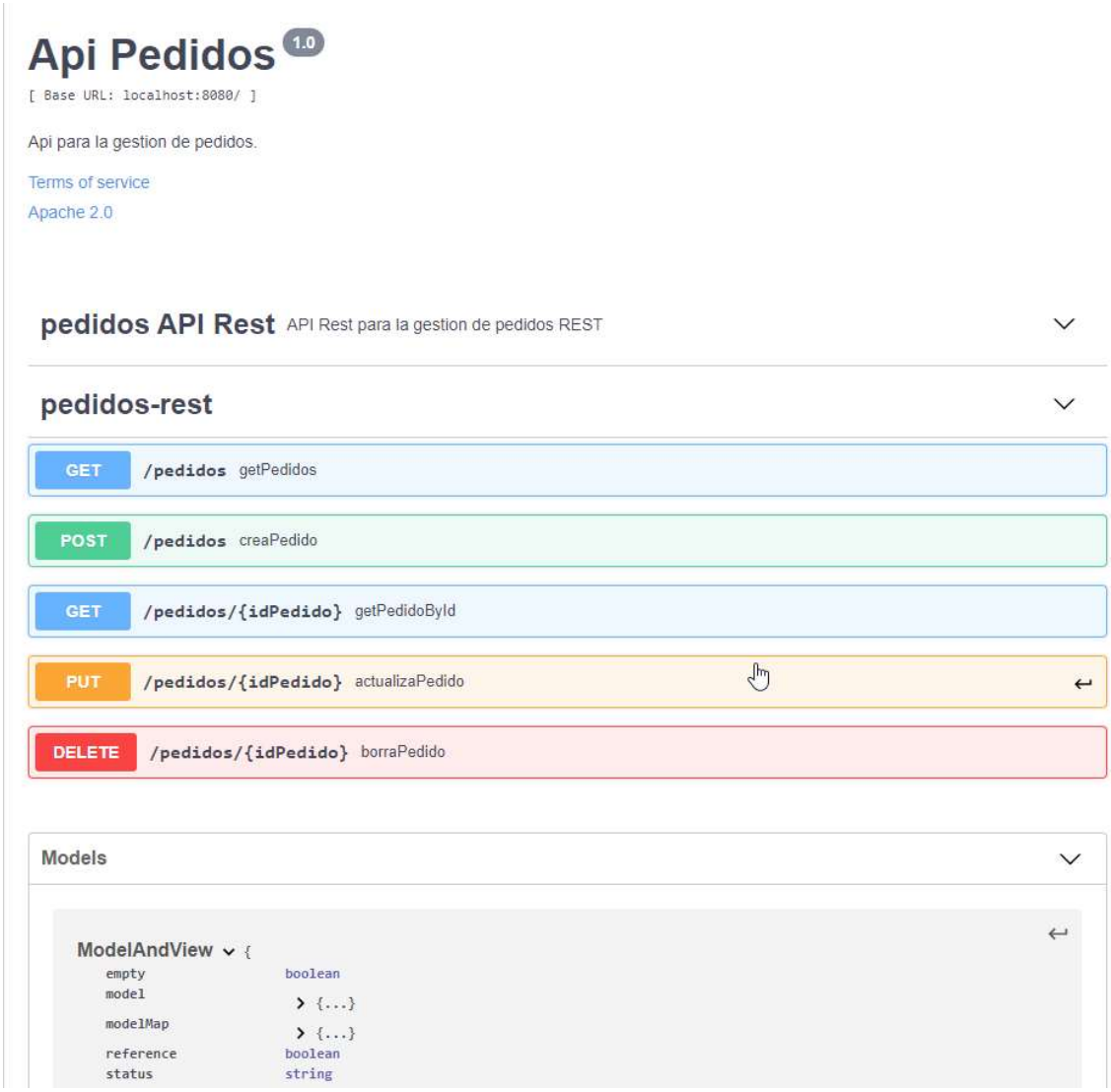

Ilustración 8: Captura de la interfaz gráfica que define el API

Se puede explotar cada operación, de manera que se detalla como se interactúa con la misma.

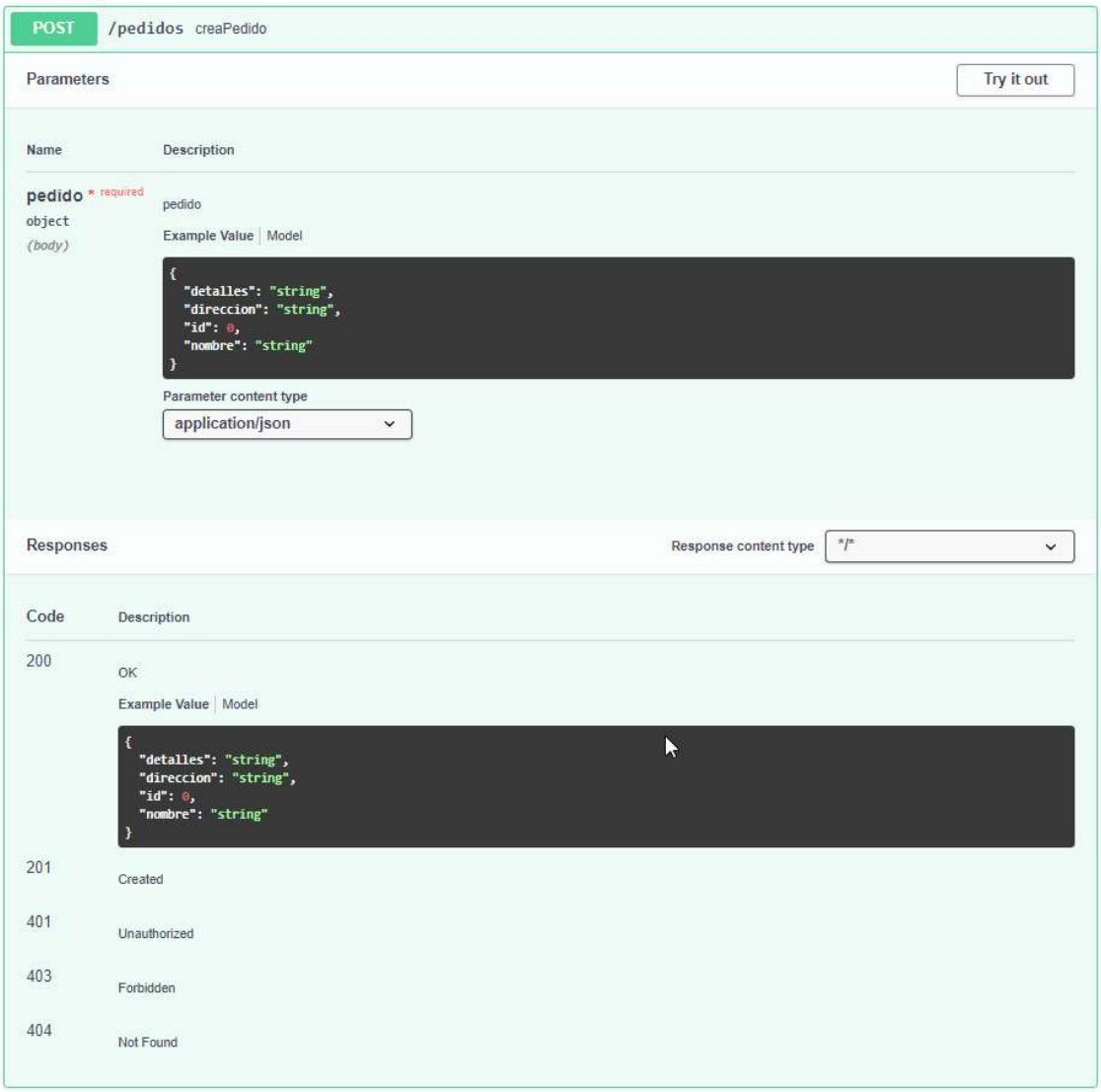

Ilustración 9: Pantalla detalle de una operación del API

4.1.2 Codificación de la API REST

Sin entrar completamente a los detalles de la implementación, si que se hará mención de algunos de los detalles más importantes de la misma, para explicar como se ha conseguido implementar el API REST.

En primer lugar, decir que para la gestión de dependencias de librerías, se ha utilizado apache Maven. Con este gestor hay un archivo (pom.xml) en el que se especifica las librerías externas en las que se va a apoyar el software que estamos construyendo. En este caso, he añadido dependencias para añadir el

servidor de aplicaciones embebido, el API de persistencia de JAVA para operar con bases de datos relacionales JPA y los drivers de conexión a la base de datos mysql. Esto se hace con estas líneas en el archivo pom.xml

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
</dependency>
```

Por otro lado, dentro de Spring Boot, existe un archivo en el que inicializar posibles objetos con los que interactuar. Es el archivo application.yml. En este caso, se van a especificar algunos parámetros que tienen que ver con la conexión a la base de datos y el modo de uso de la misma por parte de JPA. En el caso de uso implementado, este archivo tiene este contenido

```
spring:
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL5Dialect

  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/TFM?
zeroDateTimeBehavior=convertToNull&serverTimeZone=UTC
    username: root
    password: *******
```

Se puede ver que pedimos inicializar la persistencia vía JPA en un modo ddl-auto, lo que permitirá al software generar sentencias DDL para la definición de estructuras de almacenamiento (tablas, etc) en caso de que lo necesite. También se le especifica el dialecto SQL que ha de utilizar para conversar con el gestor de bases de datos relacionales, en este caso MySQL. Posteriormente, se le pide crear un pool de conexiones a la base de datos instanciando un patrón de diseño Datasource, en el que se le parametrizan los datos del driver de base de datos a utilizar, la cadena de conexión a la base de datos y las credenciales.

Esto es en cuanto a las dependencias de elementos externos. Hablando de la propia aplicación, se ha generado una estructura de paquetería en la que albergar las clases que la aplicación necesita para producir el contenido REST.

Ilustración 10: Estructura de paquetes Java en el API implementada

En el paquete general com.tfm.seguridadapis contiene una única clase, PedidosApiApplication, que inicializa el framework de Spring Boot. Su función es leer lo existente en el archivo application.yml y proporcionar la configuración al resto de clases de la aplicación.

En el paquete com.tfm.seguridadapis.entity se definirá la clase de entidad Pedido, que nos servirá como modelo de datos para trasladar la información al API o para persistir la misma en la base de datos mediante JPA. Esta clase llevará anotaciones que definan el tipo de dato de los campos que conforman la entidad, así como la tabla donde se guardará la misma (@Entity,@Table(name="pedidos"), @Id, @Column(name="id") o @GeneratedValue(strategy=GenerationType.***IDENTITY***)).

En el paquete com.tfm.seguridadapis.dao se sitúa la clase PedidosDAO. Esta clase implementa la interface JPA que permitirá a la aplicación ejecutar la lógica de persistencia en la base de datos de manera transparente a la aplicación.

Por último, en el paquete com.tfm.seguridadapis.rest, se ha situado la clase PedidosREST. Esta clase contiene la implementación del API. Mediante anotaciones, se va especifican tanto que se trata de un API REST (@RestController), como los paths que escucha este API (@RequestMapping("/pedidos")) . Posteriormente, en cada uno de los métodos, hay anotaciones que indican bajo que verbos HTTP y que parámetros se lanzaría cada uno de ellos(@GetMapping o @DeleteMapping(value="{idPedido}"). Esta clase hará uso de las clases de persistencia de base de datos PedidosDAO y la entidad Pedido para el traspaso de datos. Otro detalle a destacar es la facilidad del trabajo con REST gracias a las clases que aporta la librería ResponseEntity que permite traducir las entidades a JSON de manera transparente y devolver códigos de respuesta HTTP con sencillez.

Al arrancar la clase de inicio de la aplicación, vemos que sale esto por el log de la consola.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
[32m :: Spring Boot :: [39m              [2m (v2.4.4)[0;39m

[2m2021-04-26 10:11:30.103[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36mc.t.seguridadapis.PedidosApiApplication [0;39m [2m:[0;39m
Starting PedidosApiApplication using Java 15.0.2 on DESKTOP-0EIDUOQ with PID 44696 (C:\Users\
```

```
ogane\Documents\workspace-spring-tool-suite-4-4.10.0.RELEASE\PedidosApi\target\classes started
by ogane in C:\Users\ogane\Documents\workspace-spring-tool-suite-4-4.10.0.RELEASE\PedidosApi)
[2m2021-04-26 10:11:30.105[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36mc.t.seguridadapis.PedidosApiApplication [0;39m [2m:[0;39m No
active profile set, falling back to default profiles: default
[2m2021-04-26 10:11:30.623[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36m.s.d.r.c.RepositoryConfigurationDelegate[0;39m [2m:[0;39m
Bootstrapping Spring Data JPA repositories in DEFAULT mode.
[2m2021-04-26 10:11:30.653[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36m.s.d.r.c.RepositoryConfigurationDelegate[0;39m [2m:[0;39m
Finished Spring Data repository scanning in 25 ms. Found 1 JPA repository interfaces.
[2m2021-04-26 10:11:30.971[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36mo.s.b.w.embedded.tomcat.TomcatWebServer [0;39m [2m:[0;39m Tomcat
initialized with port(s): 8080 (http)
[2m2021-04-26 10:11:30.981[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36mo.apache.catalina.core.StandardService  [0;39m [2m:[0;39m
Starting service [Tomcat]
[2m2021-04-26 10:11:30.981[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m
[2m[           main][0;39m [36morg.apache.catalina.core.StandardEngine [0;39m [2m:[0;39m
Starting Servlet engine: [Apache Tomcat/9.0.44]
[2m2021-04-26 10:11:31.050[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mo.a.c.c.C.[Tomcat].[localhost].[/]      [0;39m [2m:[0;39m Initializing Spring embedded WebApplicationContext
[2m2021-04-26 10:11:31.050[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mw.s.c.ServletWebServerApplicationContext[0;39m [2m:[0;39m Root WebApplicationContext: initialization
completed in 918 ms
[2m2021-04-26 10:11:31.163[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mo.hibernate.jpa.internal.util.LogHelper [0;39m [2m:[0;39m HHH000204: Processing PersistenceUnitInfo [name:
default]
[2m2021-04-26 10:11:31.198[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36morg.hibernate.Version                   [0;39m [2m:[0;39m HHH000412: Hibernate ORM core version 5.4.29.Final
[2m2021-04-26 10:11:31.289[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mo.hibernate.annotations.common.Version  [0;39m [2m:[0;39m HCANN000001: Hibernate Commons Annotations
{5.1.2.Final}
[2m2021-04-26 10:11:31.361[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mcom.zaxxer.hikari.HikariDataSource      [0;39m [2m:[0;39m HikariPool-1 - Starting...
[2m2021-04-26 10:11:31.609[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mcom.zaxxer.hikari.HikariDataSource      [0;39m [2m:[0;39m HikariPool-1 - Start completed.
[2m2021-04-26 10:11:31.620[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36morg.hibernate.dialect.Dialect           [0;39m [2m:[0;39m HHH000400: Using dialect:
org.hibernate.dialect.MySQL5Dialect
[2m2021-04-26 10:11:32.008[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mo.h.e.t.j.p.i.JtaPlatformInitiator      [0;39m [2m:[0;39m HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
[2m2021-04-26 10:11:32.013[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mj.LocalContainerEntityManagerFactoryBean[0;39m [2m:[0;39m Initialized JPA EntityManagerFactory for
persistence unit 'default'
[2m2021-04-26 10:11:32.305[0;39m [33m WARN[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mJpaBaseConfiguration$JpaWebConfiguration[0;39m [2m:[0;39m spring.jpa.open-in-view is enabled by default.
Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to
disable this warning
[2m2021-04-26 10:11:32.392[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mpertySourcedRequestMappingHandlerMapping[0;39m [2m:[0;39m Mapped URL path [/v2/api-docs] onto method
[springfox.documentation.swagger2.web.Swagger2Controller#getDocumentation(String, HttpServletRequest)]
[2m2021-04-26 10:11:32.445[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mo.s.s.concurrent.ThreadPoolTaskExecutor [0;39m [2m:[0;39m Initializing ExecutorService
'applicationTaskExecutor'
[2m2021-04-26 10:11:32.558[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mo.s.b.w.embedded.tomcat.TomcatWebServer [0;39m [2m:[0;39m Tomcat started on port(s): 8080 (http) with
context path ''
[2m2021-04-26 10:11:32.558[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36md.s.w.p.DocumentationPluginsBootstrapper[0;39m [2m:[0;39m Context refreshed
[2m2021-04-26 10:11:32.574[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36md.s.w.p.DocumentationPluginsBootstrapper[0;39m [2m:[0;39m Found 1 custom documentation plugin(s)
[2m2021-04-26 10:11:32.593[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36ms.d.s.w.s.ApiListingReferenceScanner    [0;39m [2m:[0;39m Scanning for api listing references
[2m2021-04-26 10:11:32.715[0;39m [32m INFO[0;39m [35m44696[0;39m [2m---[0;39m [2m[           main][0;39m
[36mc.t.seguridadapis.PedidosApiApplication [0;39m [2m:[0;39m Started PedidosApiApplication in 2.867 seconds (JVM
running for 3.389)
```

Estas trazas aportan información como se arranca el servidor Apache Tomcat embebido en Spring Boot en el puerto 8080, la inicialización de JPA, la conexión a MySQL o la traza final que confirma el arranque de la aplicación PedidosApplication.

4.1.3 Prueba funcional del API

Para probar el API se ha generado un proyecto SOAPUI que permite probar las 5 operaciones CRUD codificadas. Se puede ver en el siguiente ejemplo como se hace una petición POST que creará un nuevo pedido en la base de datos de pedidos.

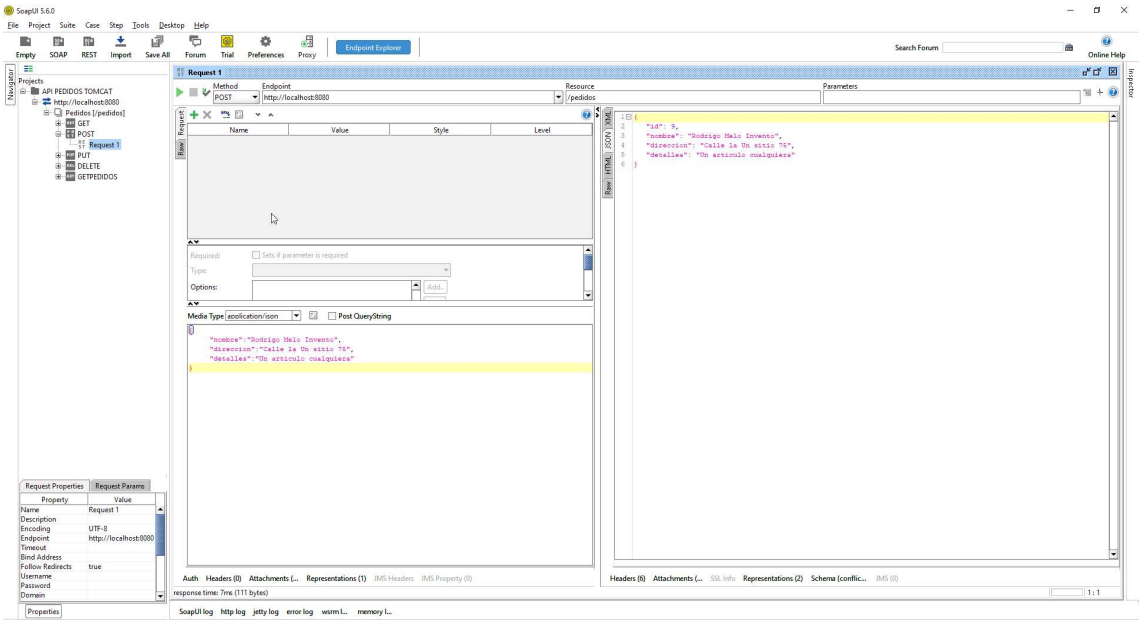

Ilustración 11: Ejemplo de petición POST al API implementada

Se ha creado el pedido con ID:9 con los valores pasados. Si vamos a la base de datos, podemos comprobar que el registro está en la tabla correspondiente.

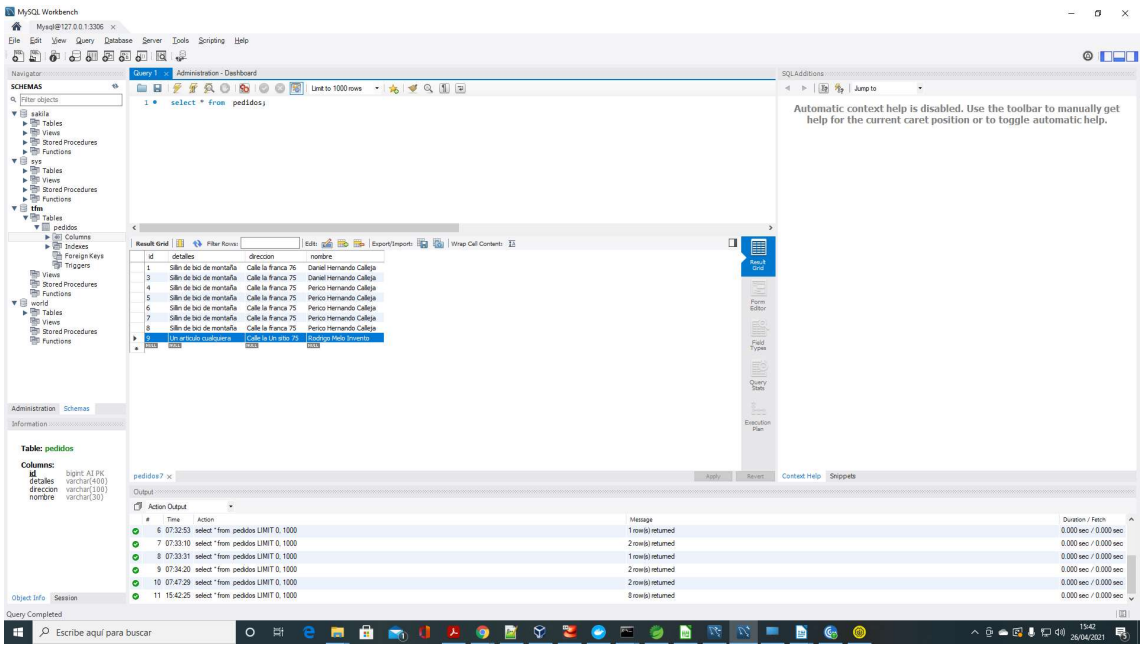

Ilustración 12: Estado de la base de datos tras llamada

Se prueban todo el resto de operaciones y funcionan con éxito. Desde la consulta de todos los pedidos existentes:

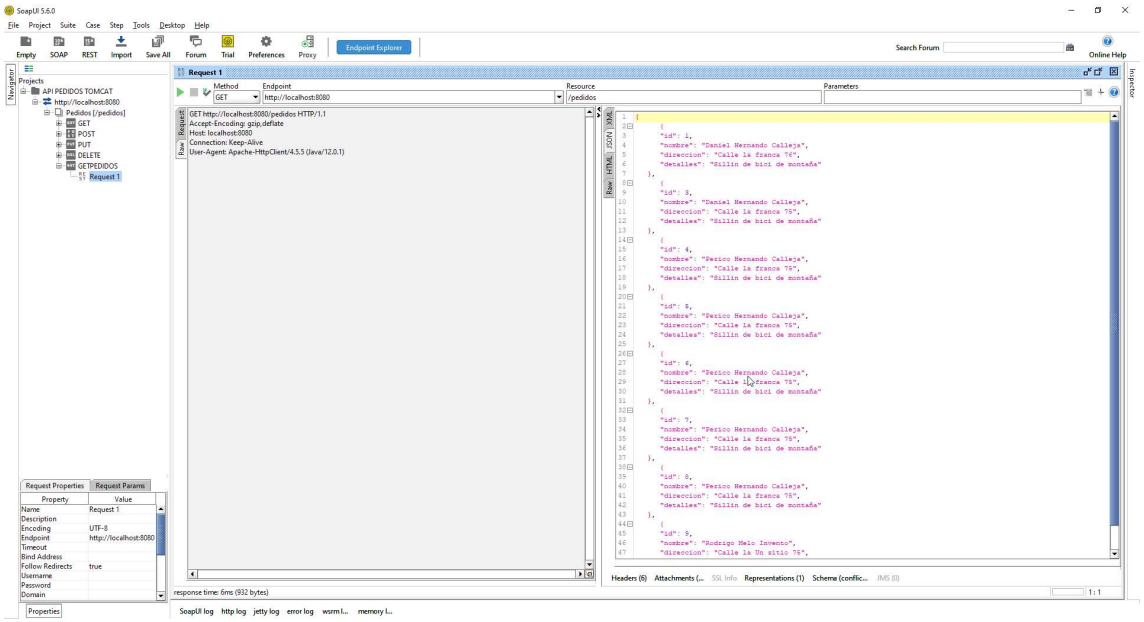

Ilustración 13: Ejemplo de prueba verbo GET

A la actualización mediante PUT, la consulta de un registro especifico por ID mediante GET o el borrado mediante el DELETE.

.4.2 Gobierno de la API

Ya tenemos el API desarrollada. Ahora, de acuerdo a los objetivos del proyecto, vamos a gobernarla mediante el producto de API Management WSO2 Api Manager, ya que el API no tiene ningún tipo de protección. Ni su uso está restringido mediante mecanismos de autenticación y autorización, ni está protegida frente a posibles ataques como los descritos previamente.

Para ello, he instalado el producto WSO2 con los 3 módulos necesarios para este caso de uso: API Gateway, Developer Portal y API publisher. En primer lugar, para poder gobernar el API hay que registrarlo en el producto. Esto se hace mediante el módulo de API Publisher. Al instalar el producto, este proporciona la consola administrativa del Publisher en la siguiente URL destinada al efecto (https://localhost:9443/publisher/). Una vez introducidas las credenciales, tenemos la posibilidad de registrar un API en el producto.

Una vez dentro de la consola, si pulsamos sobre la opción APIs, aparecerá un listado con las APIs registradas. Existirá también la posibilidad de registrar un API nuevo mediante el botón de Create API. Al pulsar sobre el mismo, ofrece diferentes opciones. En este caso de uso, dado que ya tenemos diseñado el API con su archivo YAML, existirá la posibilidad de importarlo

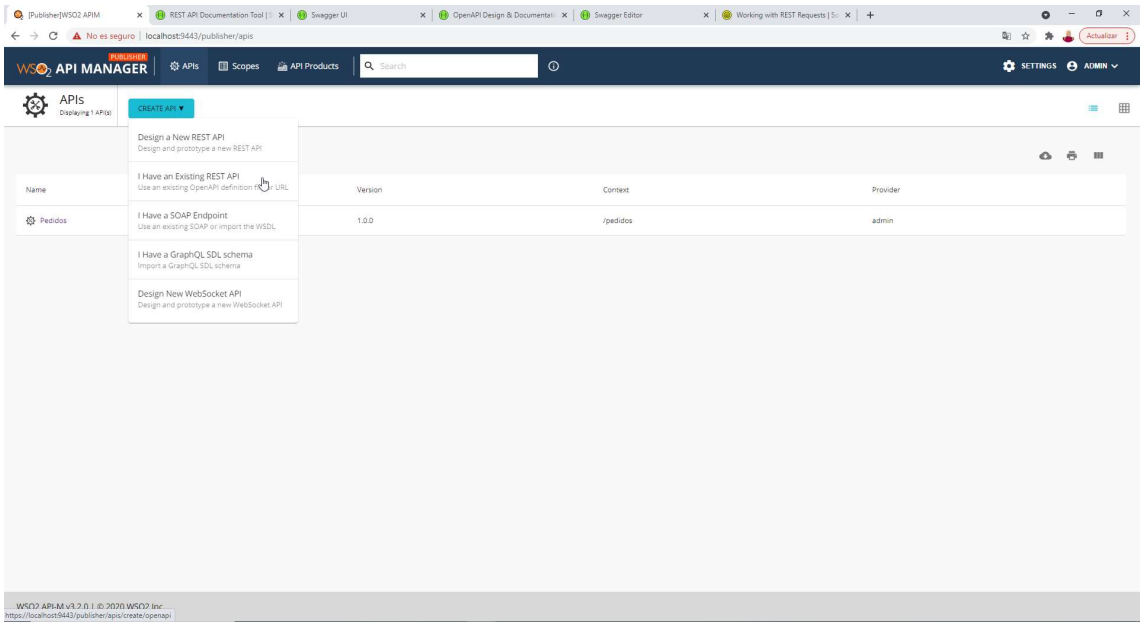

Ilustración 14: Registro de API en WSO2

Una vez importado el archivo, en la lista de APIs aparecerá el API, en este caso, el API Pedidos. Si pulsamos sobre el mismo, obtenemos todas las posibles opciones de configuración que es posible ejercer sobre el API registrado.

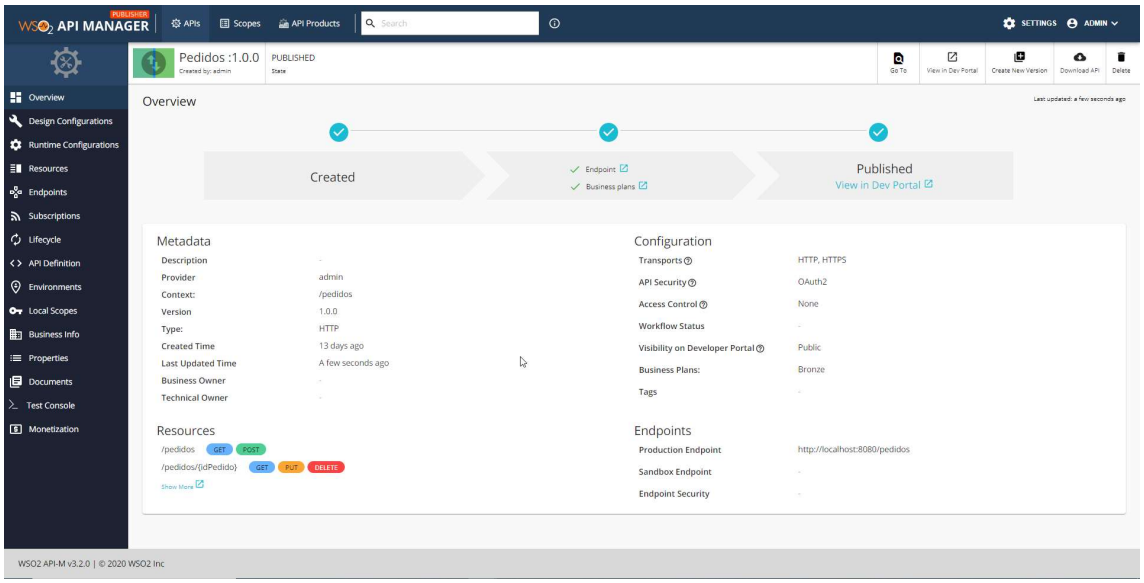

Ilustración 15: Descripción básica del API en la consola publisher WSO2

Como se puede apreciar, ya en la vista previa Overview se ofrece una visión bastante significativa de como está configurado el API. Se ve que se ofrece en el contexto /pedidos, que está etiquetado como versión 1.0.0 dentro del ciclo de vida y se encuentra en estado publicada. Además, vemos que tiene seguridad aplicada bajo OAuth2 pero no tiene control de acceso, que está visible en la zona pública del Developer Portal y que tiene un plan de negocio Bronze en cuanto a su limitación de consumo máximo. Además, vemos que ofrece los paths especificados en el YAML para el API y que tiene un EndPoint final que corresponde al servidor Tomcat que ha levantado la aplicación Spring Boot. Por último, vemos que se ofrece tanto por HTTP como por HTTPS.

Con esta configuración básica, ya es posible ejecutar una prueba y además hacerlo desde la propia consola administrativa del módulo de publisher. Al haber puesto el API en estado publicado, esta ha sido habilitada para ser expuesta en el módulo de API Gateway. Si vamos a la opción de menú Test Console, nos aparecerá una interfaz de usuario en el que probar todas las operaciones del API. Nos da a elegir entre HTTP y HTTPS y cualquiera de las operaciones.

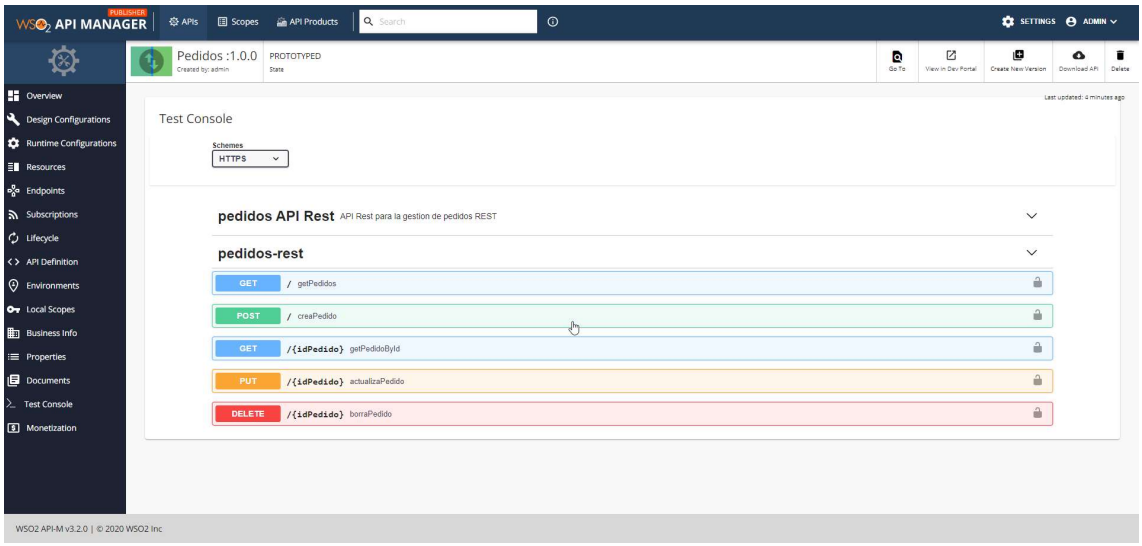

Ilustración 16: Sección de pruebas del API en la consola WSO2 Publisher

Se elige una opción, por ejemplo GET. Hay un botón de 'Try it out' que, una vez pulsado, nos habilita otro de 'Execute'. Si se pulsa en Execute, se mostrarán en pantalla varias cosas. Por un lado, un ejemplo de prueba mediante el comando CURL que es capaz de realizar la petición. Además, nos aparecerá debajo el resultado.

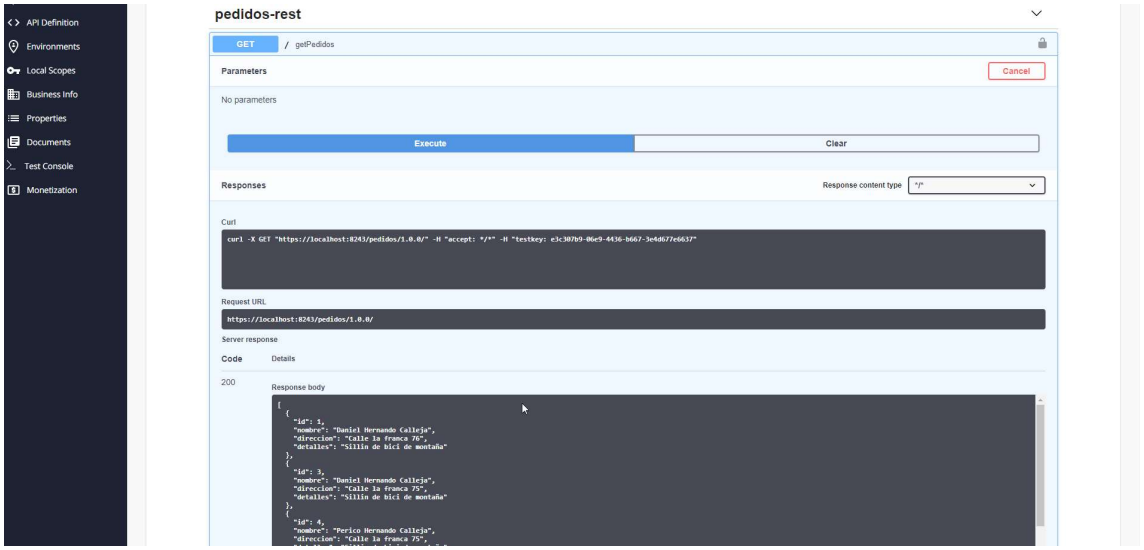

Ilustración 17: Ejecución de prueba mediante consola de WSO2 Publisher

Como se puede apreciar, frente a la exposición que se hacía via Tomcat, hay varios cambios en la petición. Por un lado, se lanza sobre el puerto HTTPS del Gateway (https://localhost:8243). Por otro, se puede ver que la URI ahora contiene la versión del API (https://localhost:8243/pedidos/1.0.0) y además vemos que se pasa una cabecera 'testkey' con un valor que ha generado la consola administrativa para la prueba. El resultado no difiere frente a la petición que se realizaba directa a Tomcat. Este parámetro testkey hace las veces de Token temporal para poder probar el API. Si se intenta probar el API sin este parámetro, por ejemplo, con un proyecto SOAPUI similar al que se construyó

contra Tomcat, veremos que no se permite consumir ninguna operación del API, devolviendo un código HTTP 401 Unauthorized.

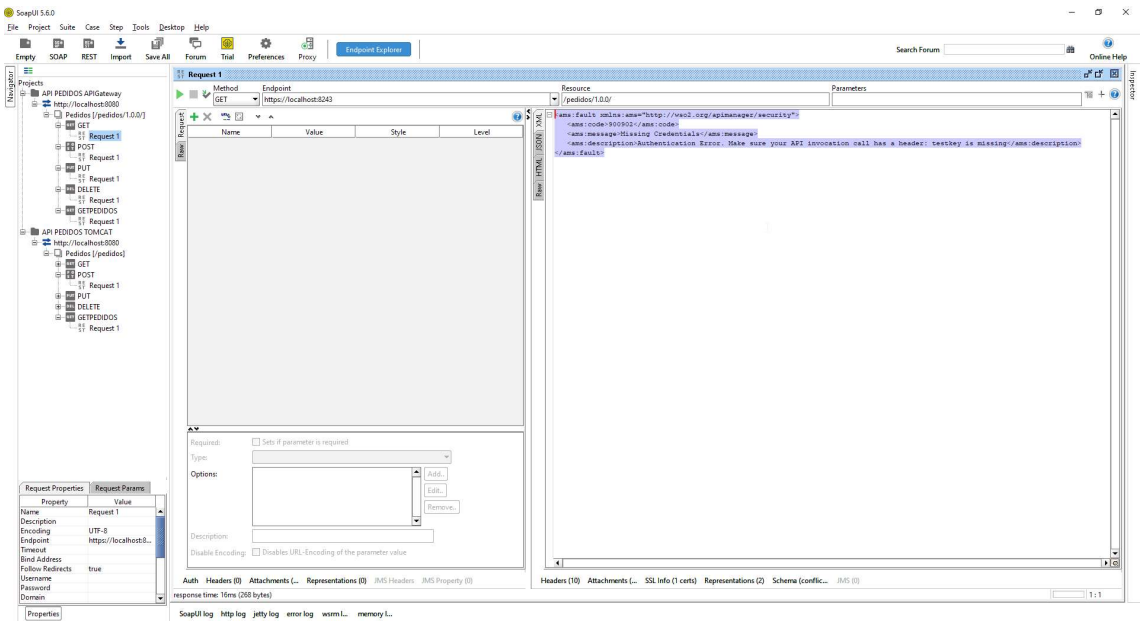

Ilustración 18: Ejecución de prueba mediante SOAPUI a través de WSO2 API Gateway

Se ha conseguido exponer en el API Gateway el API Pedidos gobernada mediante el API Publisher. Ahora quedaría el paso de dar de alta un perfil de desarrollador en el módulo de Developer Portal. En este caso, la consola del módulo está escuchando en la URI https://localhost:9443/devportal/. Si se accede a la misma, se ve que, dado que la API Pedidos se ha declarado como de visibilidad pública, esta expuesta información de la misma en la consola del developer sin tener que autenticarse. Esto no significa que se pueda consumir, sólo informa de que dicha API está disponible en el Gateway. Para consumirla, habrá que registrarse como usuario.

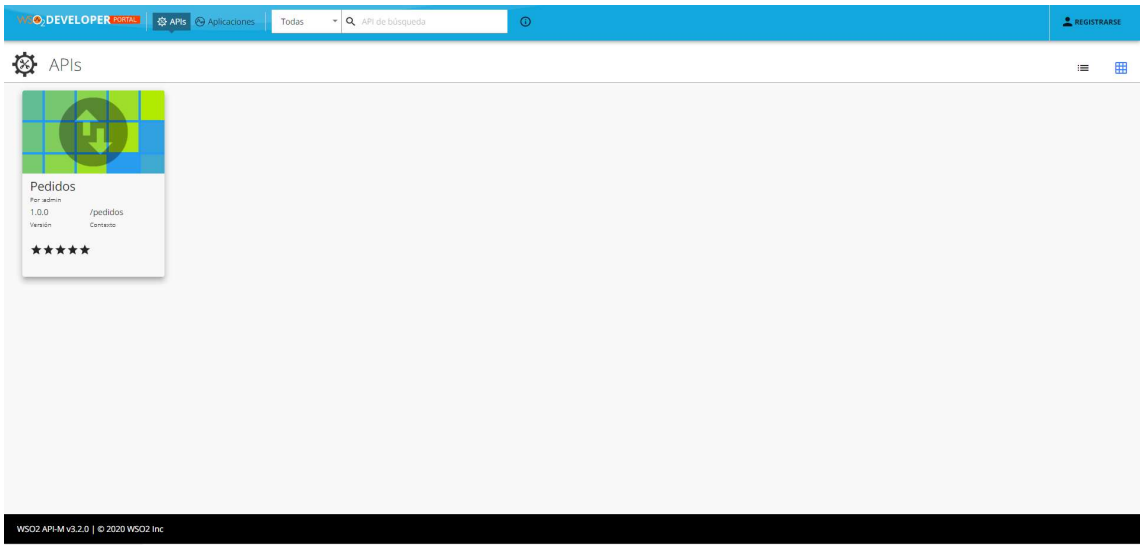

Ilustración 19: Consola del módulo WSO2 portal del developer

Si se pulsa sobre la API, se da información sobre la misma, como documentación o incluso se puede habilitar el dejar realizar una prueba de uso sobre la misma. Si se pulsa sobre el botón Registrarse, se puede registrar un usuario. Se ha realizado el registro de un usuario 'Desarrollador' y se ha accedido a la consola del Developer Portal con dichas credenciales.

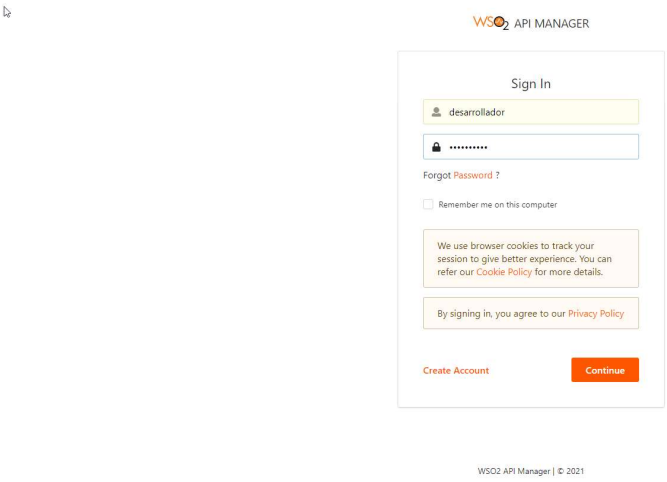

Ilustración 20: Pantalla de autenticación en la consola del desarrollador WSO2

Al entrar en la consola, vemos una pantalla en la interfaz muy similar a la obtenida en la zona pública. La diferencia es que ahora podemos generar Aplicaciones. Un Aplicación en Developer Portal es un concepto bajo el que se obtienen una credenciales de consumo a APIs expuestas en el Gateway bajo una solicitud de suscripción. Este concepto cubre el caso de uso en el que un desarrollador puede estar creando una aplicación y quiere consumir una o varias APIs en la misma. Para ello, crea la aplicación y se subscribe a las APIs deseadas con un plan de consumo. Esto se realiza en la consola mediante el botón Aplicaciones, Agregar Nueva Aplicación. Nos solicitará un nombre de

Aplicación, un plan de consumo por cada Token obtenido para un API, y una descripción. Se ha creado la Aplicación 'Pedidos Movil', con un plan de 10 peticiones por minuto para cada Token expedido.

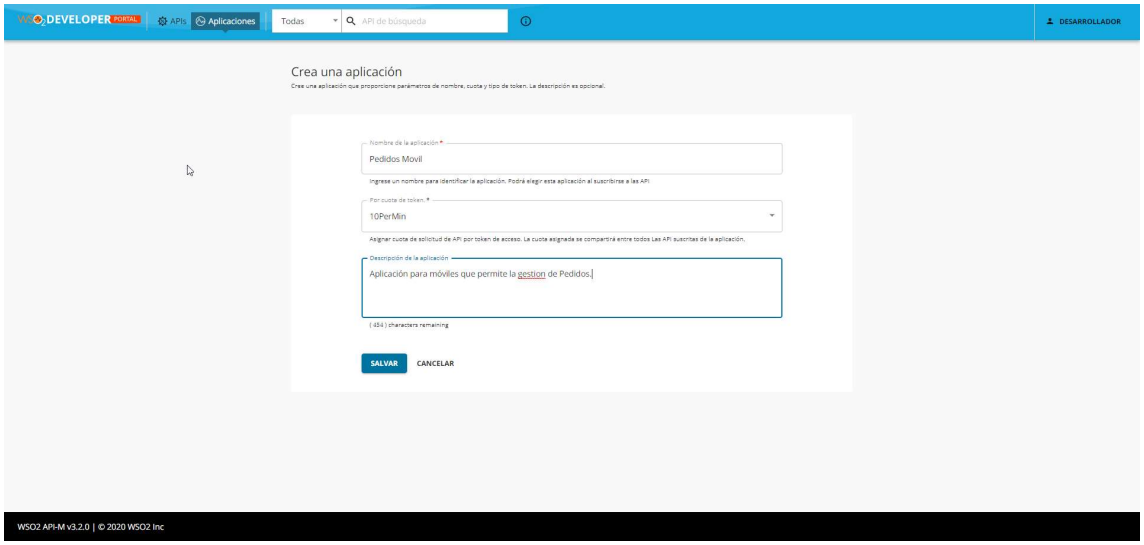

Ilustración 21: Pantalla de consola WSO2 para crear una aplicación

Ya está creada la aplicación, pero todavía no está subscrita a ningún API. Habrá que suscribirla al API de Pedidos para que la pueda consumir y crear unas credenciales para hacerlo. Para ello se pulsa sobre la aplicación Pedidos Movil y se va a la sección Suscripciones. Se pulsa sobre Suscribir API. Se ofrece la política de limitación de consumo por defecto del API (Bronze) y la Api Pedidos. Se pulsa en Suscribir y esta aplicación ya podrá consumir el API.

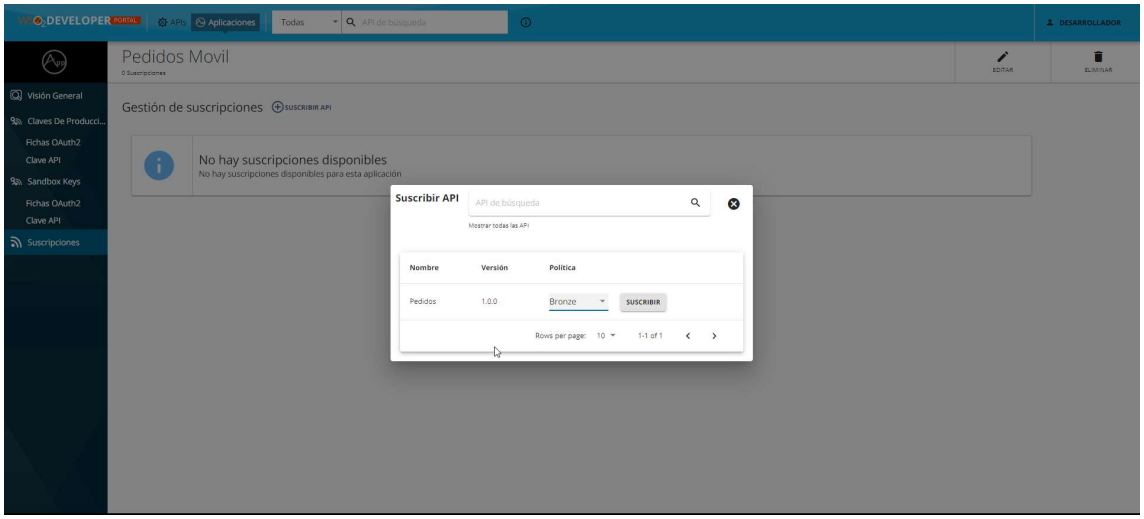

Ilustración 22: Pantalla de suscripción de una aplicación al consumo de un API

Puede consumirla por estar suscrita, pero dado que la configuración que tenemos aplicada por defecto en el API es la de protegerla con seguridad OAuth2, habrá que crearle unas credenciales. Según los flujos OAuth2 explicados previamente en este documento, estas credenciales siempre serán un Token generado que puede ser de dos tipos en base a su duración en el

tiempo: existirá el Token con caducidad (opción Fichas O¡Auth2) o el Token sin ella (Api keys).

Para trabajar Tokens OAuth2 existe en la consola la siguiente pantalla.

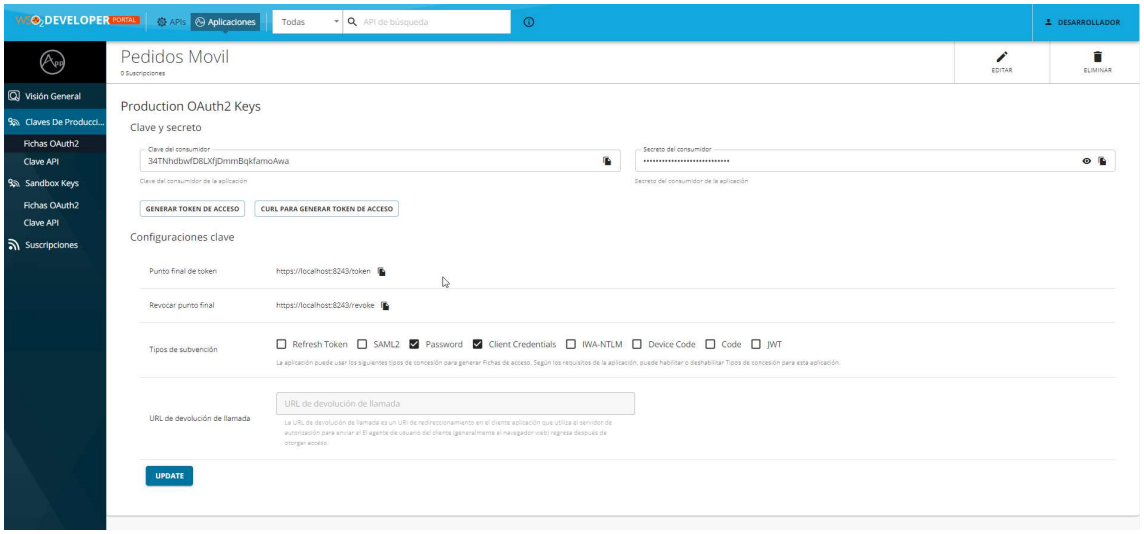

Ilustración 23: Pantalla de creación de credenciales de aplicación

En ella se puede trabajar conseguir un Token con caducidad temporal. Para ello, hay que proporcionar una serie de datos a la hora de generar un Token y este tendrá una duración limitada (por defecto 60 segundos). Existen los conceptos de Client Secret, Consumer Secret, Scope, y Grant dentro de Oauth2 que nos permitirán construir dicho Token, aquí ya adquieren valores y se pueden utilizar para construir el Token de consumo desde esta misma pantalla. También tiene el denominado 'Tipos de subvencion' que hace referencia al tipo de Grant pedido por el Token y por ende el contenido del mismo (puede tener un Token opaco que no contenga información de ningún tipo, un token JWT, proporcionar un código de refresco o refresh token, etc). Sin entrar en todas las posibles opciones se puede proceder a generar un Token mediante la consola que sirva para consumir el API desde un ejemplo. Para ello seleccionamos en Generar Token, elegimos el alcance o Scope por defecto del API (no se puso ninguno) y se obtiene como en la siguiente figura.

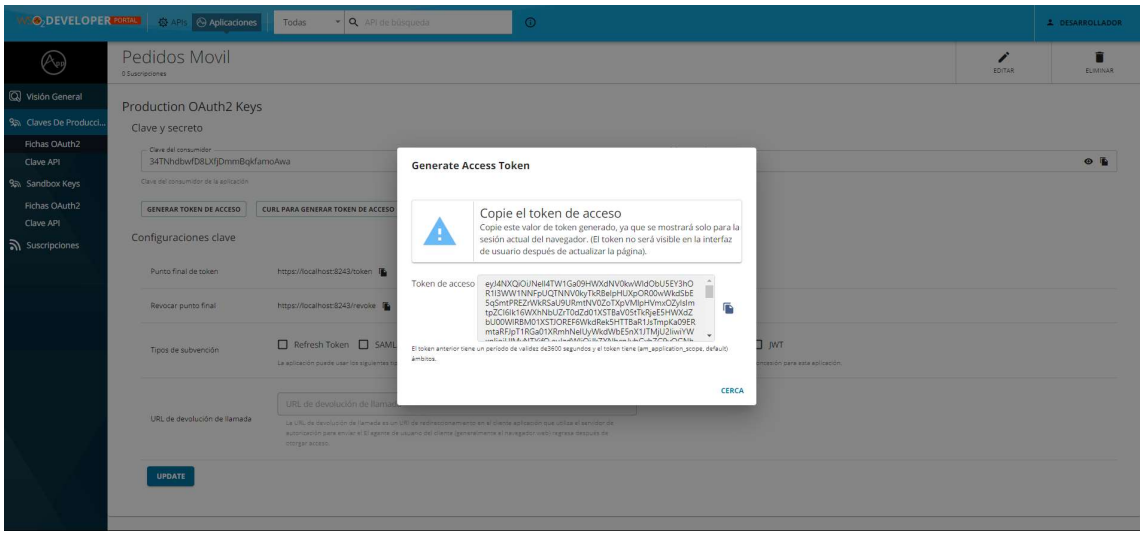

Ilustración 24: Ejemplo de Token generado mediante consola desarrollador WSO2

Se copia dicho token, que como se puede ver es una cadena de caracteres codificada en juego de caracteres base64 y se utiliza en la petición de ejemplo dentro de la cabecera HTTP Authorization para intentar consumir el API. Por tanto, recuperando en SOAPUI la petición previamente realizada y que daba código de retorno HTTP 401, se añade la cabecera Authorization con el formato que OAuth2 exige, es decir. 'Bearer TokenBase64'

Al generar la petición, se produce la respuesta esperada, es decir, la lista de todos los Pedidos

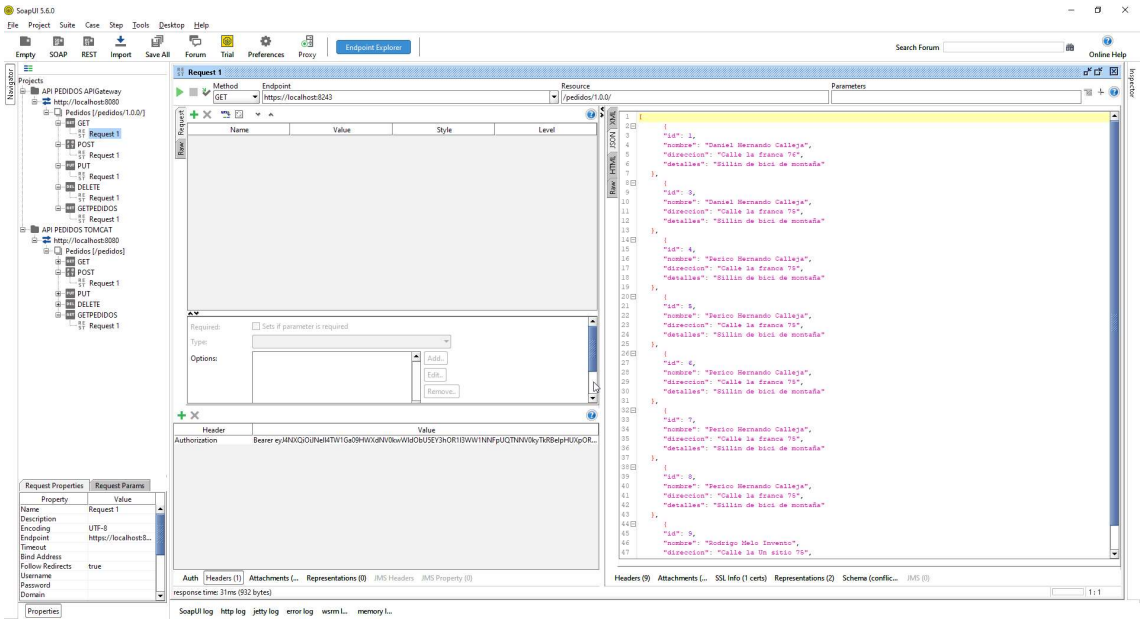

Ilustración 25: Prueba con SOAPUI de consumo del API con la credencial creada para la aplicación

Con este ejemplo de gobierno del API se puede comprobar que, solamente por pasarlo por el Gateway, con la configuración por defecto, ya se están obteniendo unos resultados en cuanto a la securización del API bastante

significativos. El API necesita de unas credenciales OAuth2 para ser consumido, está expuesto bajo protocolo HTTPS y tiene políticas de consumo máximo asignadas. En el siguiente punto, se explicará como mejorar la seguridad del API para conseguir los requisitos de seguridad expuestos en este documento como deseables y proteger el mismo contra las principales amenazas que se describieron previamente.

.4.3 Protección del API REST

WSO2 API Manager proporciona una cantidad muy extensa de opciones a la hora de proteger un API. En el punto anterior se ha podido comprobar que, por defecto, el producto está orientado a seguridad OAuth2 y aplicar un mínimo de opciones de seguridad a las APIs expuestas por defecto. Estas opciones lógicamente se pueden ampliar o incluso desactivar y dejar el API para consumo sin ningún tipo de protección tal como está expuesta en el Tomcat. Lógicamente, para llevar a cabo el caso de uso propuesto, en este caso se van a explorar las opciones que permiten proteger el API contra los ataques más frecuentes.

4.3.1 Exposición del API mediante HTTPS

El producto por defecto ofrece puertos de escucha HTTP y HTTPS para la exposición de las APIs mediante el Gateway. Para garantizar la confidencialidad e integridad de la comunicación, es necesario deshabilitar la exposición del API vía HTTP. Esto se realiza en la sección Runtime Configurations, Transport Level Security. Por defecto vienen marcados ambos protocolos, si se desmarca el HTTP, el API dejará de atenderse por dicho protocolo en el puerto 8280, como estaba habilitado anteriormente.

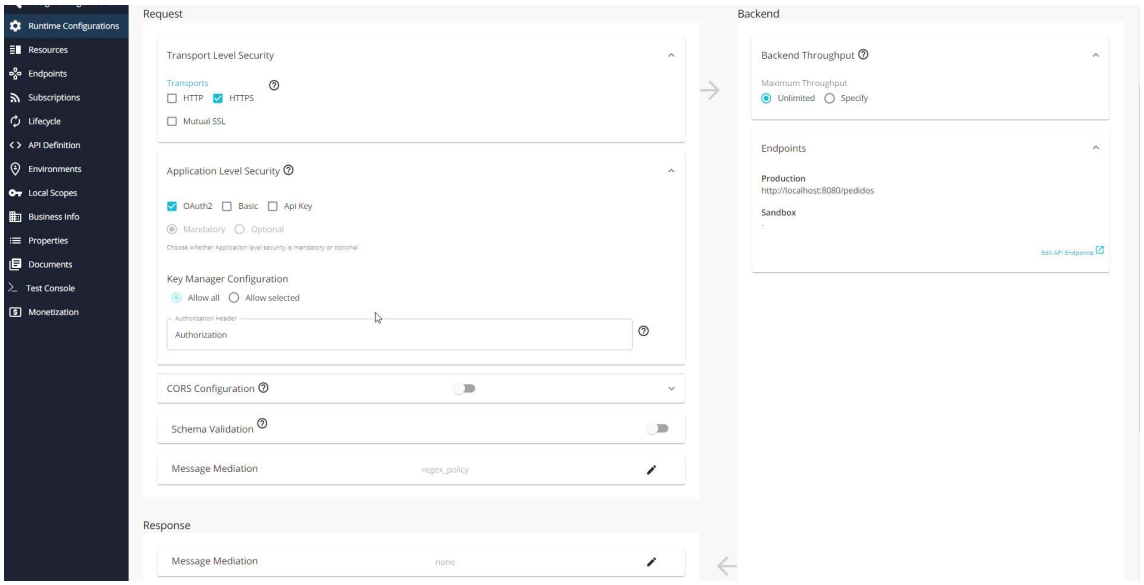

Ilustración 26: Ajuste de configuración para exponer el API vía HTTPS

Para comprobar que esto se ha aplicado correctamente, se realiza una petición al puerto 8280, produciéndose en código de error HTTP 403 Forbidden, con el mensaje Unsupported Transport [HTTP]

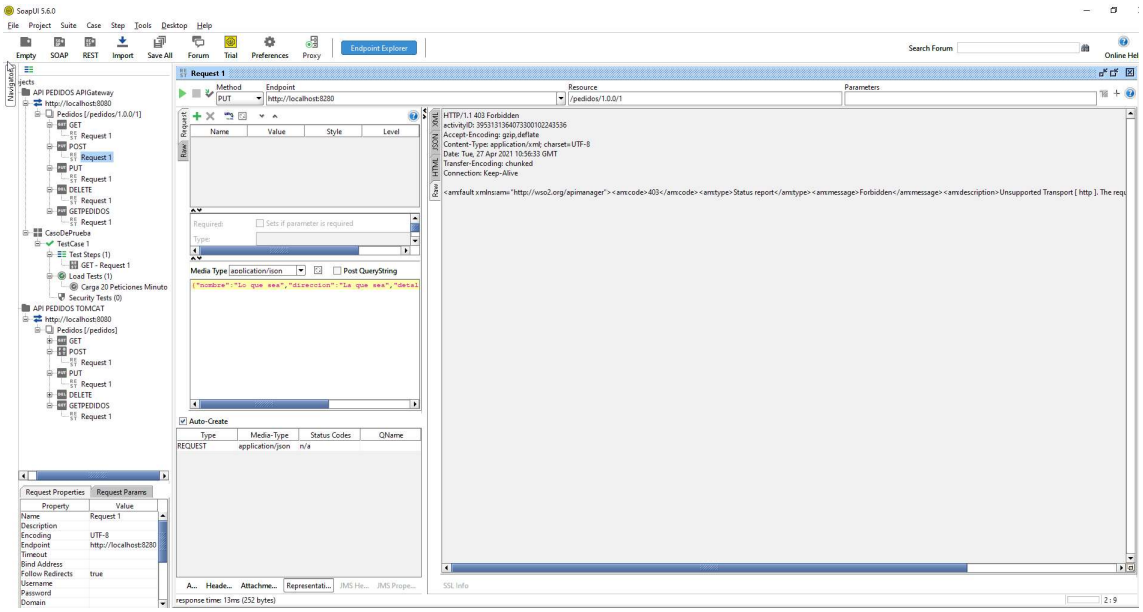

Ilustración 27: Comprobación de que el API ya sólo se expone por HTTPS

4.3.2 Ataques de denegación de servicio.

Como se ha visto en el ejemplo anterior, el Gateway siempre asocia una política de consumo de recursos a cada Token generado para unas credenciales concretas. El objetivo es limitar los recursos que un cliente puede llegar a consumir e, incluso, con las opciones de Monetización llegar a facturar por ello en base a un acuerdo o contrato. En el ejemplo, se ha configurado una Política Bronze para el API y, mediante la consola del Api Publisher se puede gobernar quien está suscrito al API y con que política.

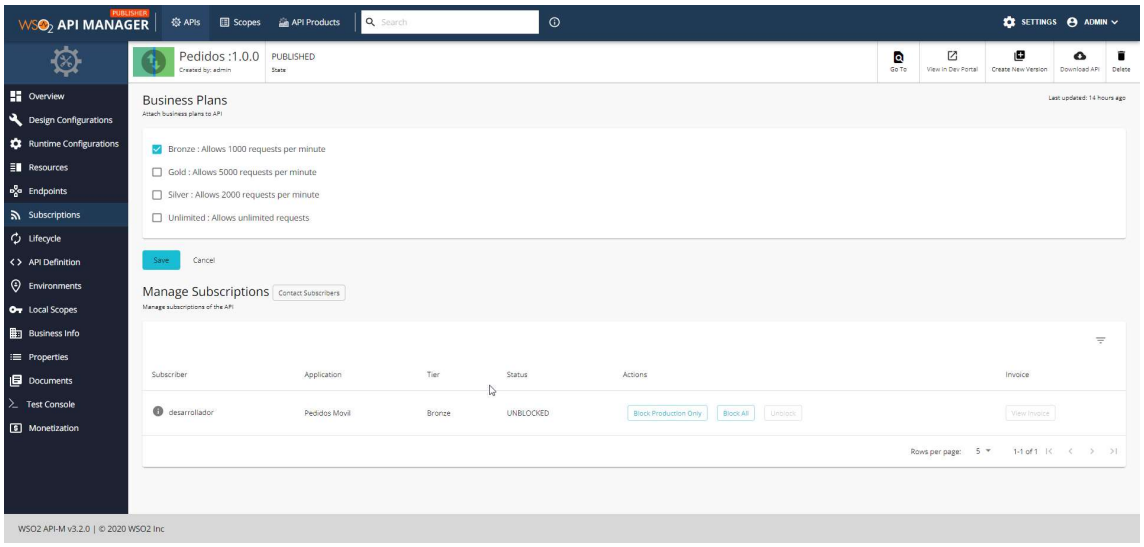

Ilustración 28: Opciones para el control del número de peticiones en WSO2

Se ve que el usuario 'desarrollador' que usamos en el portal del developer está suscrito mediante la aplicación Pedidos Movil con una Política Bronze de 1000 peticiones por minuto como límite. Existen otras políticas por defecto (Silver, Gold, Unlimited) con otros rangos o incluso se pueden crear políticas propias. Como se puede ver en el panel, se puede bloquear el API desde este punto del gobierno para una determinada aplicación. Si se comprueba esta opción pulsando en BlockAll e intentamos realizar la petición anterior, veremos como no es posible hacerlo.

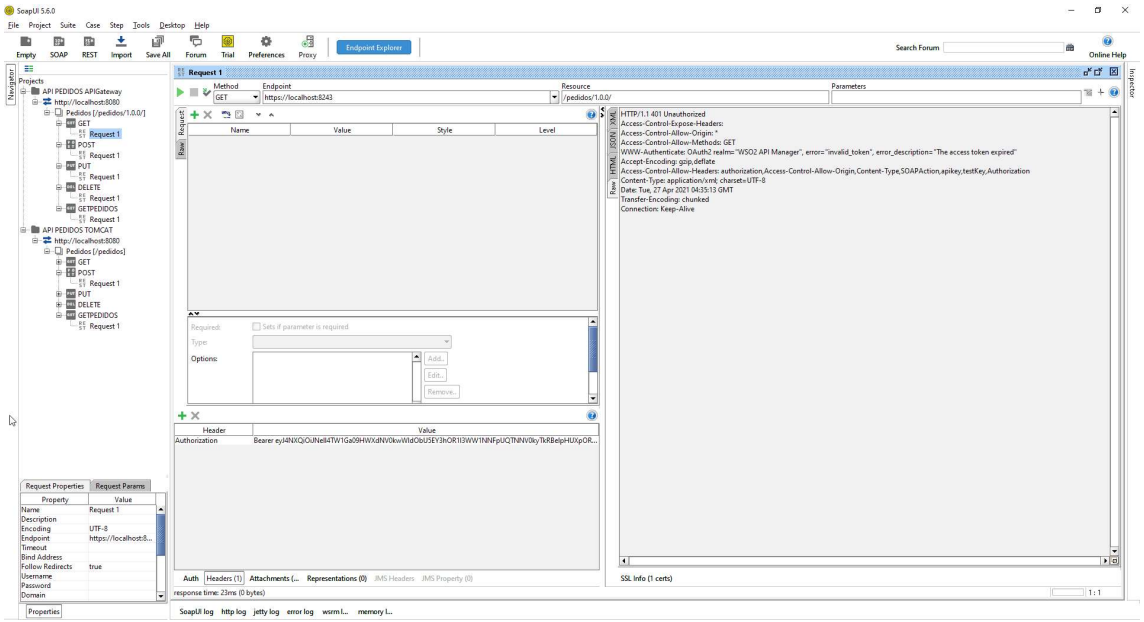

Ilustración 29: Comprobación de bloqueo de un API

Proporciona un código de Respuesta 401 UnAuthorized con una descripción de Access Token Exprired. Si pulsamos en Unblock de nuevo sobre la aplicación, la misma petición con el mismo Token da un código 200 con la respuesta esperada.

Esto permite el gobierno de las suscripciones de aplicaciones a las APIS de manera discrecional, con lo que es posible bloquear el consumo de una aplicación que se haya considerado peligrosa para la estabilidad del sistema. Pero lo interesante sería ver que dicha aplicación se bloquea automáticamente cuando se pasa del límite de consumo asignado. Esto se puede comprobar fácilmente ya que se ha creado un Token que, por encima de la política Bronze, especificaba un consumo máximo de 10 peticiones por minuto para el mismo. Así que, con la herramienta SOAPUI se ha generado un caso de prueba en el que se lanzan 20 peticiones en un minuto. Si se observa el resultado de la petición 20, por ejemplo, se ve que da un código HTTP 429, con el mensaje 'Message throttled out', que indica que la petición ha sido rechaza por exceso de consumo desde este Token de credencial. Si se espera el minuto correspondiente y se lanza otra petición con el mismo Token, el resultado es un código de respuesta 200 OK como sería de esperar.

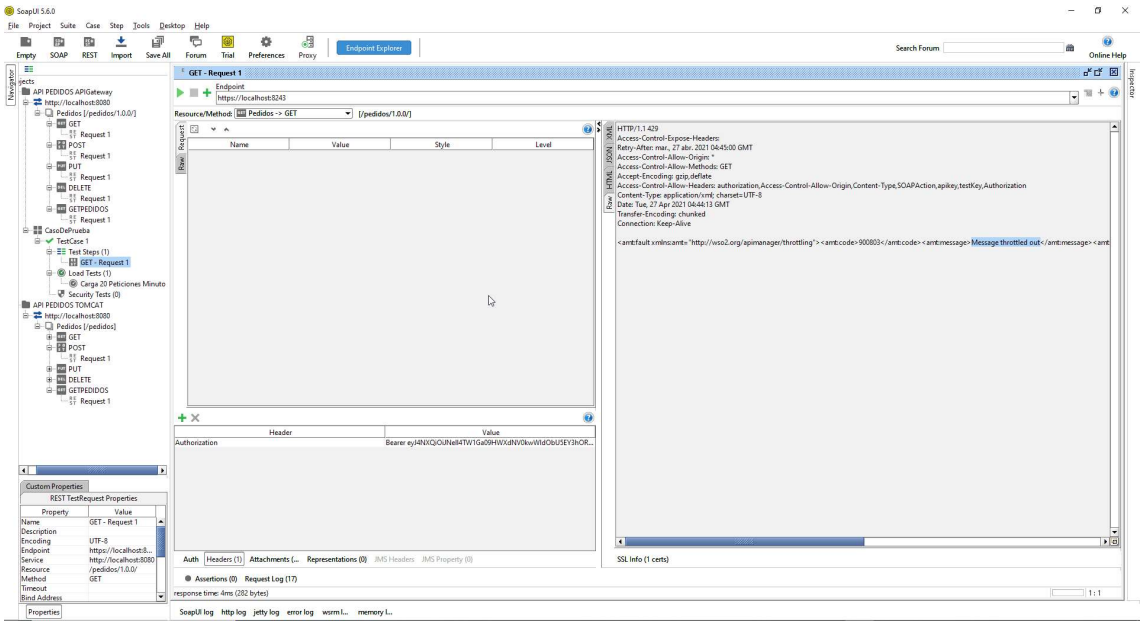

Ilustración 30: Prueba sobrepasando el límite de peticiones permitidas por minuto

Explorando estas opciones se pueden dibujar muchas configuraciones que permitirán tener mucha flexibilidad y control sobre los ataques de denegación de servicio de los que los APIs puedan ser objetivo.

4.3.3 Autenticación y autorización sobre recursos

Se ha podido comprobar previamente que se ha protegido el API contra el consumo no autenticado sobre el mismo. Ahora bien, más allá de la autenticación, se va proceder a estudiar como el API Gateway puede proteger el acceso a ciertos recursos del API de manera concreta. Por ejemplo, haciendo que una credencial de acceso o Token pueda consumir sólo unas operaciones del API y otras no. Para poder plasmar esto se debe perfilar el API, es decir, dar niveles de acceso diferentes al mismo. Esto en OAuth2 se

consigue mediante los Alcances o Scopes. En el ejemplo básico que se ha realizado hasta ahora, el API no tenía definidos niveles de acceso o Scopes. Se procede a definir dos scopes: básico y avanzado. Para ello, se pulsa sobre la opción Local Scopes y Create Local Scopes. Se dará acceso a la pantalla de creación de Scopes para el API.

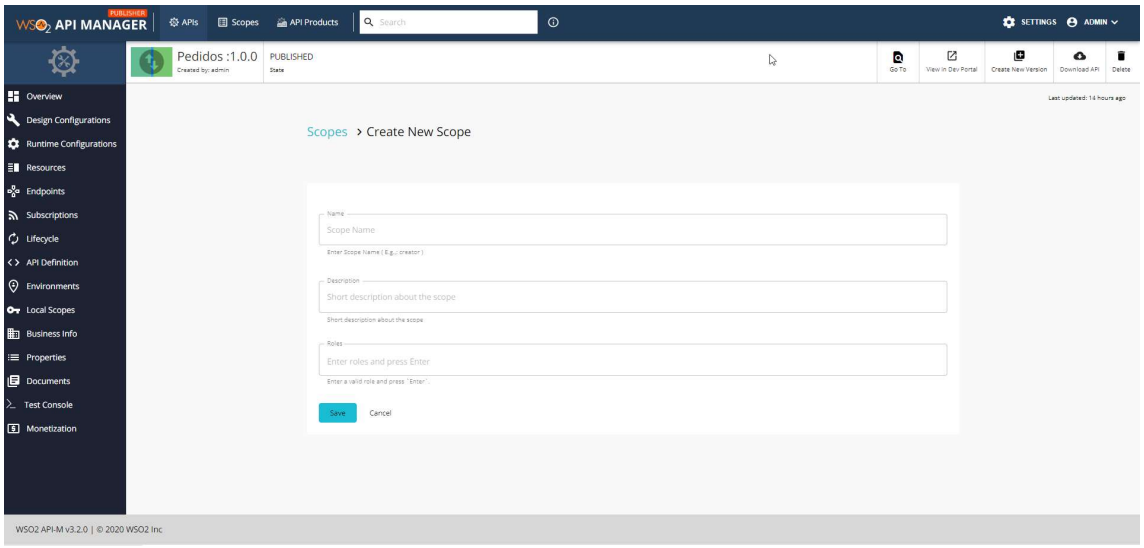

Ilustración 31: Creación de OAuth Scope en WSO2

Se crea uno Basico y un Avanzado. Por criterios de simplificación no se utilizará la opción Roles. Esta opción lo que permite es que se asigne en el repositorio de usuarios de la herramienta un grupo de tal manera que si el usuario peticionario del Scope no está en él, no se permite solicitar este Scope. Se va a suponer que dichos grupos existen y que el usuario 'Desarrollador' está en el grupo que permite solicitar ambos Scopes. Una vez generados los Scopes como se ve en la siguiente figura, se procederá a la autorización de operaciones en base a los mismos.

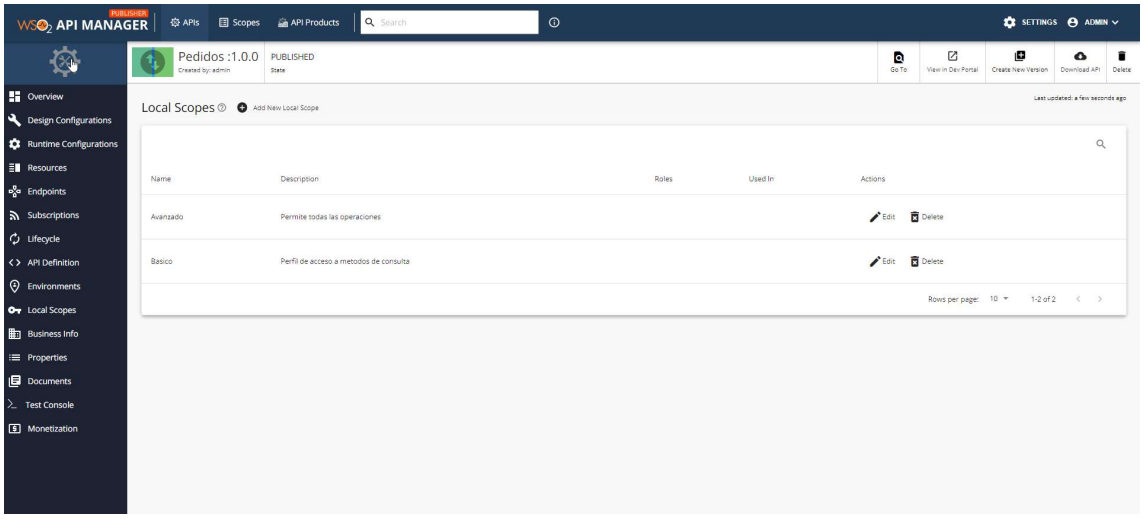

Ilustración 32: Lista de Scopes definidos en el API

Para ello existe en el Publisher la opción Resources. Si se pulsa sobre ella se obtendrá una descripción detallada de los verbos HTTP que expone el API, sobre que Paths, y permitirá la autorización de los mismos en base a los scopes.

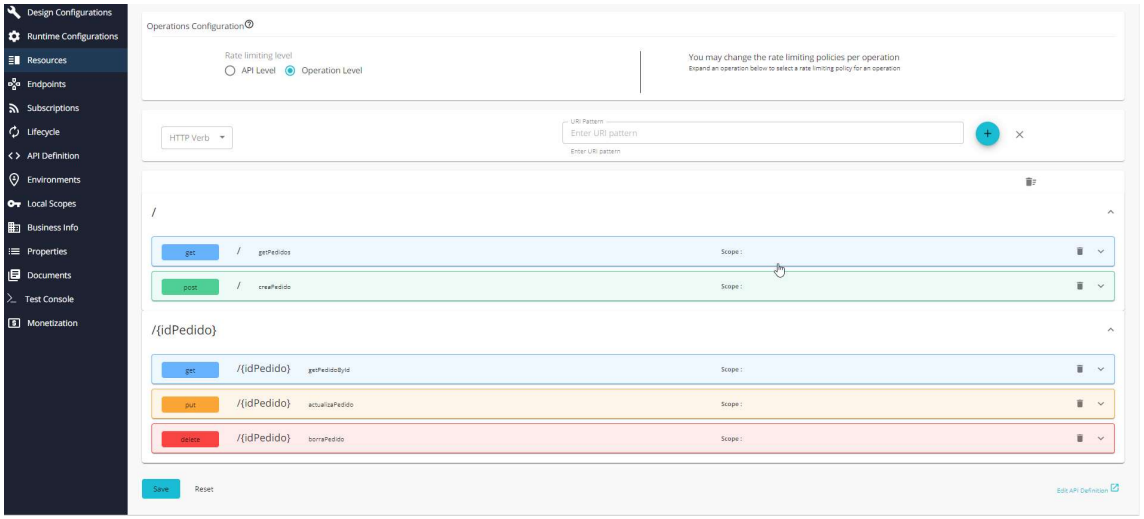

Ilustración 33: Asignación en consola WSO2 de un Scope a una operación del API

Si desplegamos cualquier verbo, por ejemplo el GET sobre el path genérico /, podemos comprobar que se ofrecen múltiples opciones de autorización. Por proseguir con el ejemplo marcado, se va a asignar el Scope Basico a las operaciones de consulta GET y el Scope Avanzado a todas las operaciones. Esto supondrá que un token consutruido con Scope Basico sólo podrá acceder a las operaciones GET y uno avanzado, podrá acceder a todas. Una vez aplicado, la misma pantalla reflejará la configuración propuesta.

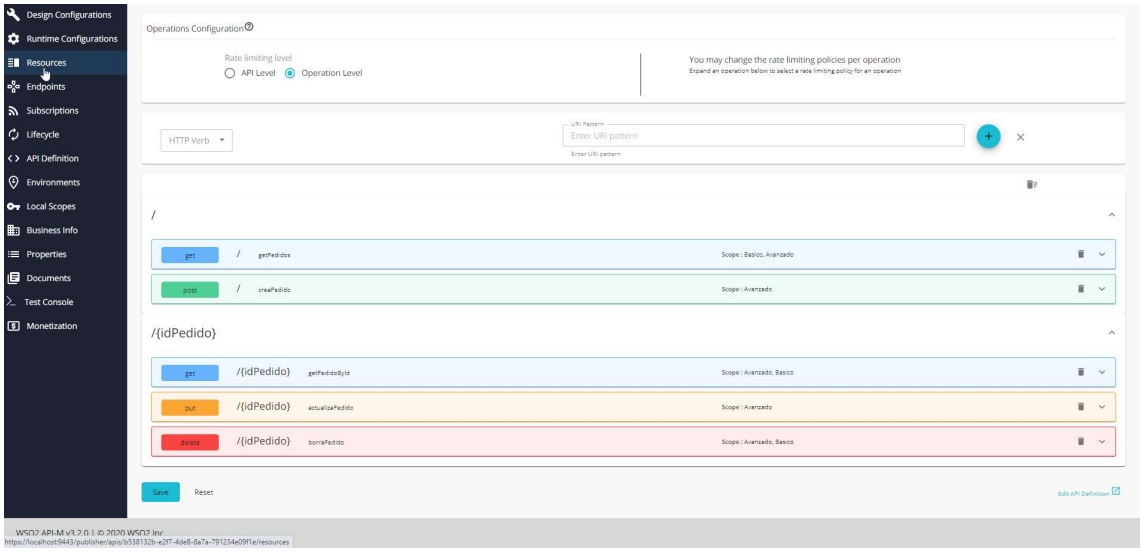

Ilustración 34: Pantalla de resumen de operaciones del API y sus Scopes asignados

Ahora, con la prueba que se tiene, se va a proceder a ejecutar la petición GET para la que se tenía un Token de acceso. Si se ejecuta directamente, se obtiene un código de Respuesta 403 Forbidden con el mensaje 'User is NOT

authorized to access the Resource: /. Scope validation failed.'. Esto es debido a que el Token que se obtuvo previamente no tenía Scope. Se procede por tanto a generar un Token con la consola del Developer que tenga el Scope Basico y se realiza de nuevo la petición. Se obtiene el resultado esperado con un código 200 Ok.

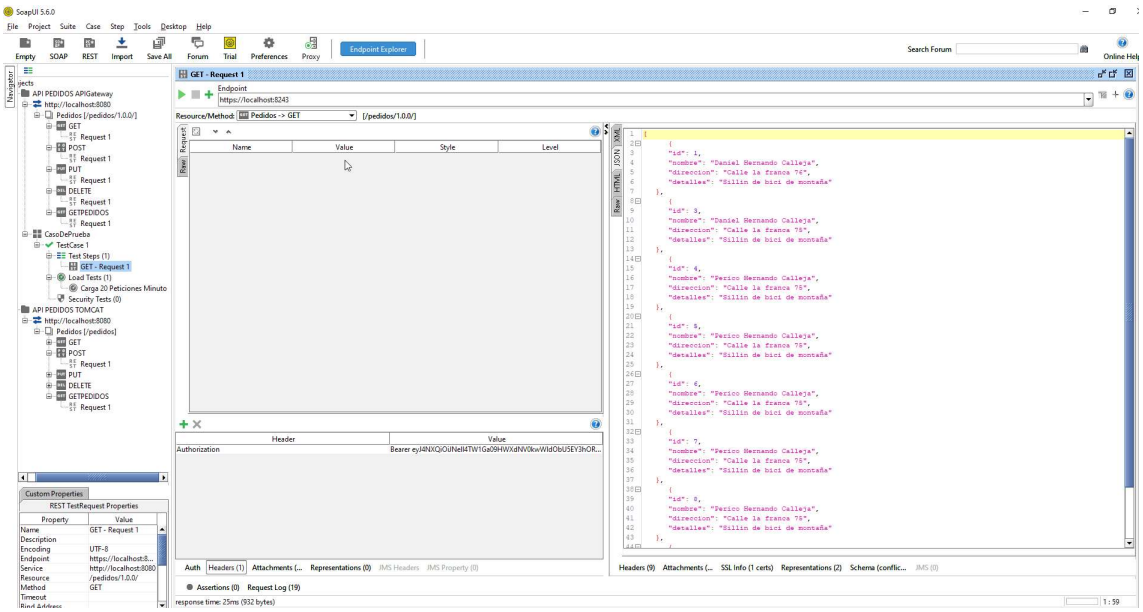

Ilustración 35: Prueba de petición a un Scope permitido para una credencial

Ahora bien, si se intenta invocar por ejemplo una operación POST, con este Token que sólo tiene Scope Basico, no debe permitir hacerlo. Al intentar probarlo, se puede comprobar que esto ha quedado aplicado. Se produce un código HTTP 403 Forbidden con mensaje de no autorización de consumo al recurso.

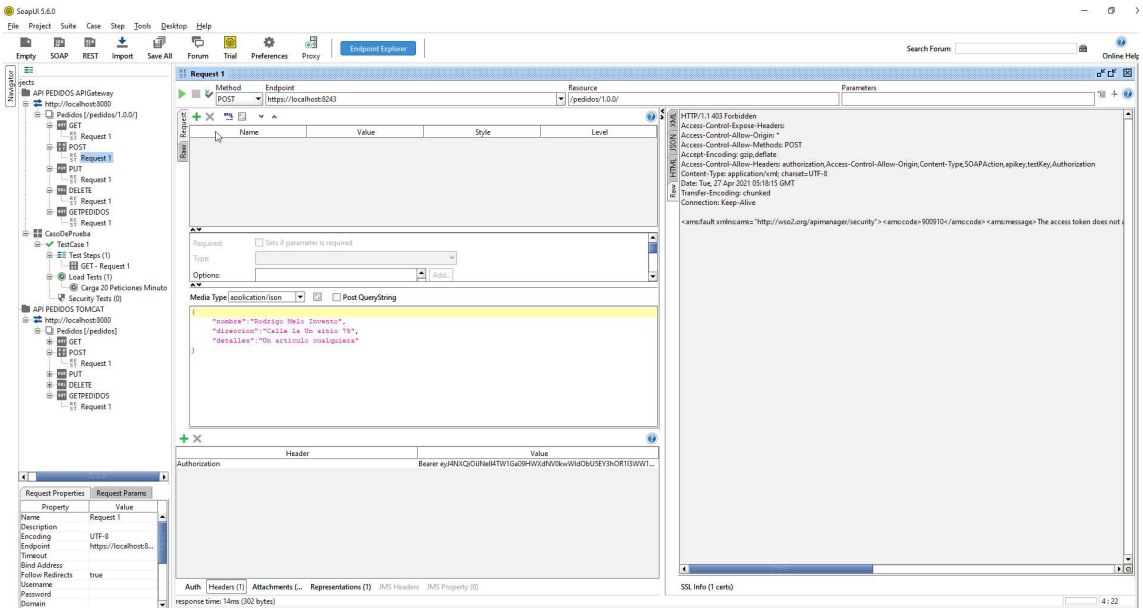

Ilustración 36: Prueba de acceso a una operación no soportada por el Scope de la credencial

Queda por comprobar si con un Token para el Scope Avanzado se podría consumir este recurso mediante método POST. Para ello, se procede a crear un Token con Scope Avanzado en el portal del Developer, y se prueba de nuevo con el mismo. La prueba es exitosa y nos deja consumir el recurso con método POST con este nuevo Token.

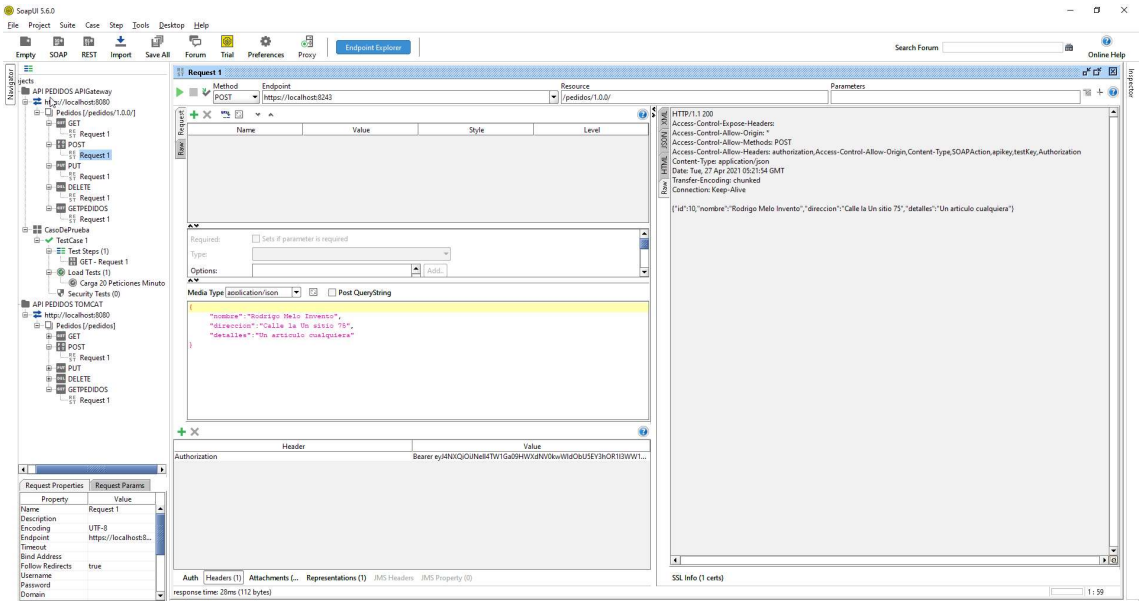

Ilustración 37: Prueba de operación una vez se asignan permisos sobre el Scope a la credencial

Con estas opciones que da el producto, se puede configurar una protección de autorización de consumo de recursos de grano muy fino. Trabajando los Roles en el repositorio de usuarios del producto y creando Scopes para las operaciones en base a los verbos HTTP y paths que corresponde a cada una,

se pueden limitar mucho los riesgos sobre el tipo de información que el API expone.

4.3.4 Peticiones de dominio cruzado o CORS

El producto ofrece la posibilidad de habilitar el consumo de un API solamente desde un determinado dominio o conjunto de dominios, el denominado CORS. Este es un mecanismo de seguridad que informa a los navegadores desde que dominios el API deja ser consumido. Esto se hace mediante las cabeceras HTTP Access-Control-Expose-Headers, Access-Control-Allow-Origin, Access-Control-Allow-Methods y Access-Control-Allow-Headers

Cuando un navegador hace una petición AJAX a un recurso sin que el usuario sea consciente de ello, la aplicación puede estar invocando un API a otro dominio sin que el usuario lo sepa. Para paliar posibles usos indebidos de esta frecuente necesidad en los sitios web, se creó un método por el que los navegadores realizan una petición con método HTTP Options al recurso del API para que le informe desde que dominios debería dejar ser invocado y con que métodos y cabeceras. Esto lo informa el API con las cabeceras HTTP expuestas anteriormente y lo hace para todas las peticiones recibidas. La idea es que el navegador pueda proteger al usuario de aplicaciones maliciosas que invoquen desde dominios no permitidos a un API. Los navegadores inhabilitan los resultados obtenidos por una petición Ajax en caso de que la información proporcionada por el API no sea la correcta.

Si se observa cualquiera de las peticiones previas, se puede comprobar que el API Gateway tiene una configuración CORS por defecto para todas las APIS. Se puede ver que para la última petición ejecutada, devolvió la siguiente información.

```
HTTP/1.1 200
Access-Control-Expose-Headers:
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers:                authorization,Access-Control-Allow-Origin,Content-
Type,SOAPAction,apikey,testKey,Authorization
```

Esto informa al navegador que este API deja ser atacado desde cualquier dominio (*). Además, informa de que deja ejecutar un método POST, y que para las peticiones se ha de dejar pasar las cabeceras HTTP authorization,Access-Control-Allow-Origin,Content-Type,SOAPAction,apikey,testKey,Authorization

Esta es un configuración al uso que puede ser cambiada fácilmente para, por ejemplo, informar a los navegadores que sólo se deja invocar desde un dominio. Para ello, hay que ir al API en el Publisher, a la sección Runtime Configurations. Esta sección en la que se pueden configurar estas cabeceras para el manejo del CORS. Se procede a poner por ejemplo la limitación de que en vez de permitir todos los dominios, sólo deje el dominio ‘clientes.algo.com’

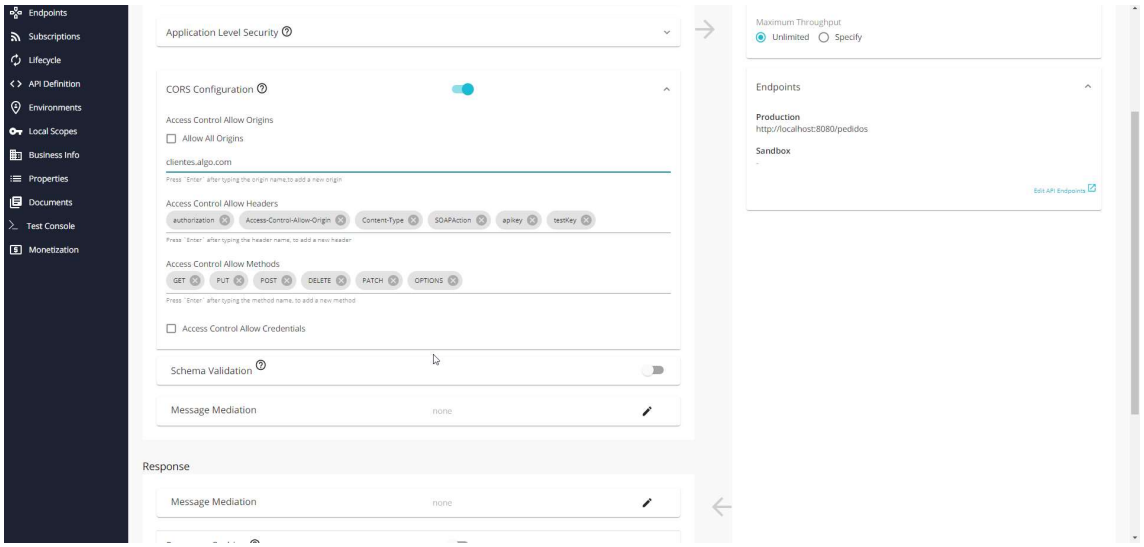

Ilustración 38: Pantalla de configuración de CORS en WSO2

Cuando realizamos la misma petición del ejemplo anterior y vemos las cabeceras que atañen al CORS, vemos que estas han cambiado, de manera que ahora informa de que el único dominio desde el que el navegador debería permitir peticiones CORS en clientes.algo.com

```
HTTP/1.1 200
Access-Control-Expose-Headers:
Access-Control-Allow-Origin: clientes.algo.com
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers:                authorization,Access-Control-Allow-Origin,Content-
Type,SOAPAction,apikey,testKey,Authorization
Content-Type: application/json
Date: Tue, 27 Apr 2021 05:50:27 GMT
Transfer-Encoding: chunked
Connection: Keep-Alive

{"id":21,"nombre":"Rodrigo Melo Invento","direccion":"Calle la Un sitio 75","detalles":"Un
articulo cualquiera"}
```

El producto permite aplicar configuración global de protección CORS y también personalizarla para cada una de las APIs expuestas.

4.3.5 Validación de esquema

Este ataque también tiene posibilidad de ser paliado de manera importante desde la configuración del producto. En el producto, se permiten gobernar APIs de tipo SOAP y de tipo REST. Para ello, trae diferentes posibilidades dentro de la sección Runtime Configuration. Particularmente, para el trabajo con REST y peticiones en formato JSON, que es lo que aplica al caso de uso del API de Pedidos, existe la posibilidad de aplicar una mediación. Las mediaciones son fragmentos de código JAVA que en producto permite introducir como intermediaciones entre el consumidor y el productor del API. Estas intermediaciones dejan operar con el contenido del mensaje, permitiendo cambiarlo, validarlo, etc. Las mediaciones se pueden programar en base a un API que el producto proporciona o aplicar algunas que el producto incorpora por defecto. En este caso, se va a proceder a aplicar una mediación que el

producto ofrece, json_validator, sobre las peticiones (las mediaciones dejan aplicarse en ambos sentidos, contenido de la petición y contenido de la respuesta). Esta mediación aplica el esquema definido para la API a las peticiones entrantes. Si ejecutamos la petción pasando contenido que no es acorde al esquema JSON o que no es directamente JSON, se producirá un error HTTP 400 de petición invalida.

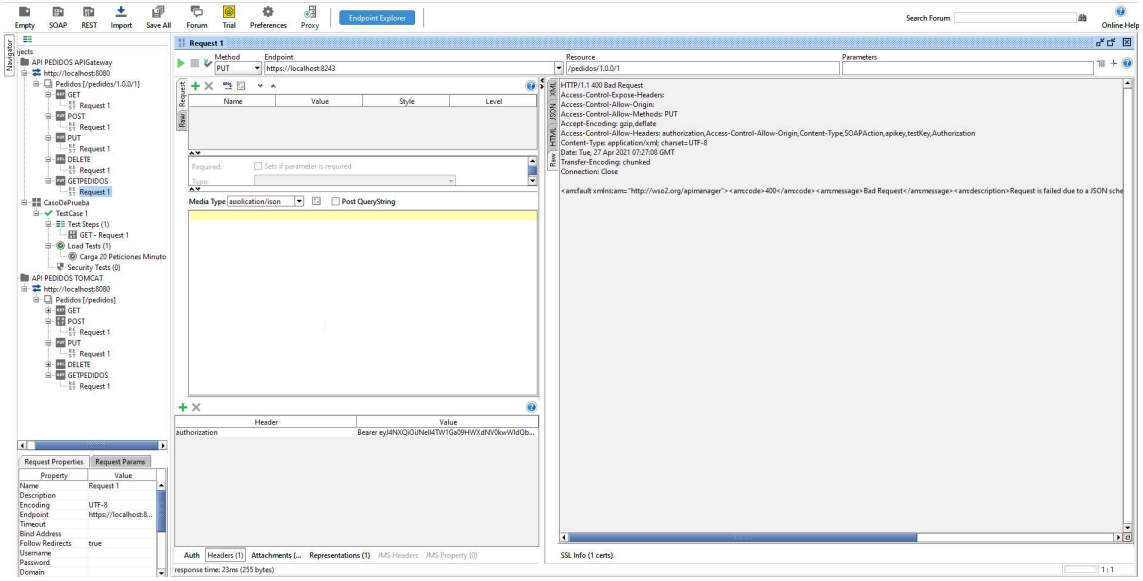

Ilustración 39: Prueba de petición con formato JSON no válido

4.3.6 Control de inyección SQL

Se puede asignar a las APIs una mediación de las que el producto aporta para prevenir los posibles ataques de inyección SQL. La mediación funciona con un fichero de patrones sobre expresiones regulares que, cuando se activa la mediación, chequea que el contenido de la petición no tiene palabras o sentencias que coincidan con el patrón o expresión regular. El fichero tiene una configuración por defecto con las siguientes expresiones.

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="regex_policy">
    <log level="custom">
        <property name="IN_MESSAGE" value="Regular_expression_policy"/>
    </log>
    <property name="threatType" expression="get-property('threatType')" value="SQL-Injection"/>
     <property name="regex" expression="get-property('regex')" value=".*'.*|.*alter.*|.*alter \
s*table.*|.*alter\s*view.*|
                    .*create\s*database.*|.*create\s*procedure.*|.*create\s*schema.*|.*create\
s*table.*|.*create\s*view.*|
      .*delete\s*from.*|.*drop\s*database.*|.*drop\s*procedure.*|.*drop\s*table.*|.*select\s*.*\
s*from.*|
    .*truncate\s*table.*|.*insert\s*into.*"/>
            <property   name="enabledCheckBody"   expression="get-property('checkBodyEnable')"
value="true"/>
        <property  name="enabledCheckHeaders"  expression="get-property('enabledCheckHeaders')"
value="true"/>
     <property name="enabledCheckPathParams" expression="get-property('enabledCheckPathParams')"
value="true"/>
    <class name="org.wso2.carbon.apimgt.gateway.mediators.RegularExpressionProtector"/>
</sequence>
```

Como se puede ver, algunas palabras de propias de sentencias SQL están dentro del filtro de expresiones regulares a aplicar. Además, es configurable

también decidir si se aplica sobre el cuerpo de la petición, los paths de las URIs o las cabeceras HTTP.

Se procede a hacer una prueba de inyección con el texto Insert into dentro del cuerpo de la petición. Cuando la petición es ejecutada, el Gateway devuelve un código de error HTTP 400 Bad Request con el mensaje 'SQL-Injection Threat detected in Payload' , luego el filtro de SQL se está aplicando.

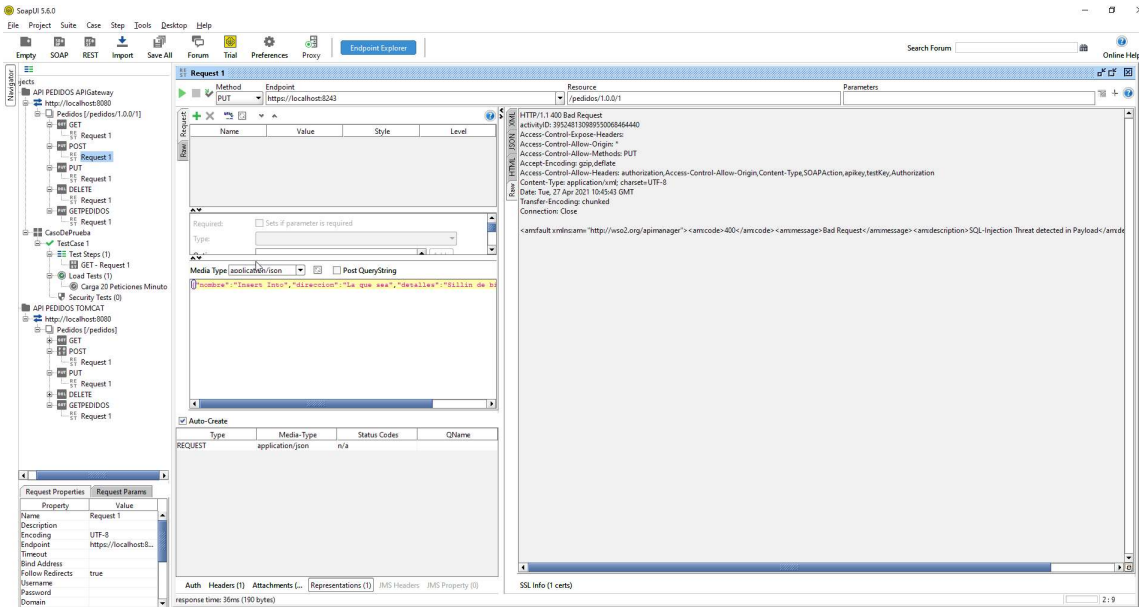

Ilustración 40: Prueba de petición que intenta una inyección SQL

4.3.7 Monitorización de peticiones

El producto trae una serie de opciones de trazabilidad y motorización bastante extensas. Existen diferentes ficheros de log en el directorio repository/logs de la aplicación. Los más importantes son los siguientes:

- audit.log – Fichero en el que se escriben eventos de auditoría como pueden ser validación del usuario en diferentes consolas, cambios en las mismas, etc.

- http_access_.log – Se registran todas las peticiones, con la IP del peticionario, recurso HTTP solicitado y código de respuesta proporcionado.

- wso2-apigw-errors.log – Se registran los errores producidos en cualquier petición o respuesta, volcando el contenido del error de ejecución.

- wso2carbon.log – Log general del producto. Se pueden habilitar niveles de traza (FATAL, ERROR, INFO, DEBUG) en un archivo de configuración al efecto.

Explotando estos registros de trazas es posible tener un control importante de los apis expuestos y que está ocurriendo con las peticiones que reciben, pero el producto tiene más opciones. Permite monitorizar las APIs con alertas, tiene políticas de Failover para posibles balanceos de carga en caso de indisponibilidad e incluso el módulo de estadísticas, que no se ha instalado en esta prueba de concepto, que permite monitorizar de manera gráfica lo que ocurre en el Gateway.

Una de las opciones más interesantes es la de poder, en un momento dado, activar un modo de trazabilidad profundo en el que podamos registrar todo el contenido de las peticiones que recibe un API. Para ello, hay que iniciar la consola de administración global del producto, denominada Carbon (https://localhost:9443/carbon/). Validándose en la misma con el usuario administrador, podemos activar la trazabilidad del contenido de las peticiones.

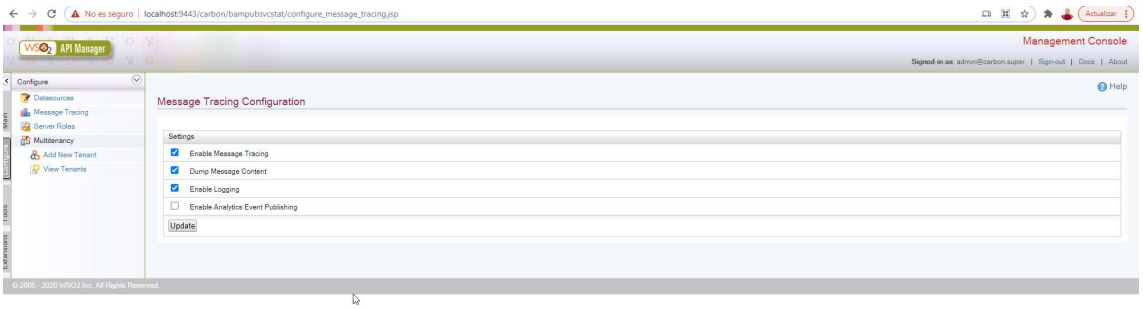

Ilustración 41: Pantalla de consola WSO2 donde habilitar la trazabilidad

Cuando activamos este registro, el contenido de las peticiones y respuestas aparecen en el fichero de log general. Haciendo una prueba, se puede comprobar que tipo de trazas genera.

```
TID:        [-1234]        []        [2021-04-27        09:45:02,985]                INFO
{org.apache.synapse.mediators.builtin.LogMediator}     -     {api:admin--Pedidos:v1.0.0}     To:
/pedidos/1.0.0/1, MessageID: urn:uuid:726e1ee9-c850-4ae3-a0af-2ce4eb10b04d, Direction: request,
IN_MESSAGE = IN_MESSAGE, Payload: {"nombre":"Daniel Hernando Calleja","direccion":"La que
sea","detalles":"Sillin de bici de montaña"}
TID:        [-1234]        []        [2021-04-27        09:45:02,986]                INFO
{org.apache.synapse.mediators.builtin.LogMediator}   -   {api:admin--Pedidos:v1.0.0}   Host   =
localhost:8243, Context = /pedidos/1.0.0/1, Http_Method = PUT, Resource = /1, Origin = chrome-
extension://aejoelaoggembcahagimdiliamlcdmfm, Content-Type = application/json, Message_Id =
urn:uuid:726e1ee9-c850-4ae3-a0af-2ce4eb10b04d
```

Se puede ver tanto las cabeceras como contenido de la petición, así como identificadores de mensaje. Esto permite muchas posibilidades de control relacionadas con la seguridad, como por ejemplo que, si se cree que se están recibiendo peticiones con contenido malicioso, poder conocer el contenido integro de las mismas.

4.3.8 Nivel de protección obtenido frente a las amenazas más frecuentes.

En este documento se han detallado un conjunto de amenazas y requisitos de seguridad. Algunas de las acciones realizadas anteriormente sobre el API expuesto sirven para intentar paliar muchas de estas amenazas y cumplir los requisitos expuestos previamente en el documento. Esto se realiza de manera transversal en la arquitectura de la solución, sin tener por qué modificar el código de las APIs, con una posibilidad de personalización muy fuerte del nivel de protección en función de diferentes criterios. Se pueden configurar políticas muy diferentes para cada API, de acuerdo a posibles estimaciones de riesgos potenciales que tenga cada cual.

Por razones de alcance de este proyecto, no se han podido implementar todas las posibilidades que el API Manager ofrece. Pero dentro de las amenazas previamente mostradas si podemos hacer un análisis de cuantas se han cubierto total o parcialmente con las medidas aplicadas en el documento.

1. Autorización a nivel de objeto. Se ha perfilado mediante seguridad Oauth 2.0 la autorización de consumo orientada a recurso, de manera que se ha perfilado mediante los Scopes que una determinada aplicación sólo admita una serie de verbos o un path. El api propuesto sólo tenía una entidad, pero en caso de trabajar con varias, se podría limitar el acceso a las mismas por el path y scope, impidiendo una escalada de privilegios sobre entidades a las que no se permite el acceso.

2. Autenticación de usuario. Se ha configurado la autenticación de acceso de consumo al API mediante seguridad OAuth. No se permite consumir el API sin un Token válido.

3. Exceso de datos expuestos. Pese a que un API pueda tener muchas opciones implementadas en su programación, si mediante el API Gateway eliminamos alguno de los métodos de su definición, jamás dejará pasar esas peticiones.

4. Limitación de peticiones y recursos. Se ha realizado una prueba en la que se conseguía limitar el número de peticiones en un intervalo de tiempo por Token expedido.

5. Autorización a nivel de operación. Se ha puesto en practica generando Tokens de diferentes niveles de acceso y comprobado que se aplican las restricciones por operaciones o verbos.

6. Asignación directa de datos. Este caso de uso es un tema muy relacionado con la lógica de programación, aunque se ha realizado una prueba de validación de esquemas que si podría paliar en parte estos ataques, ya que no permitirá aportar más información que le especificada en la definición del API. Es decir, si el atacante intenta pasar más parámetros dentro de la petición con la esperanza de sobrescribir ciertos valores que no residan en el esquema de entidades del mismo, no podrá hacerlo.

7. Configuración de seguridad deficiente. En este caso, se ha mejorado la seguridad que tenía el API en origen de manera notable y en dos fases. Una primera con opciones por defecto que protegían de ciertas amenazas, y una segunda en la que se han aplicado nuevas acciones que cubrían más amenazas.

8. Inyección de código. Se ha realizado una protección de la API frente a inyección SQL.

9. Manejo incorrecto de activos. Se ha aplicado versionado al API desde el API Gateway, de manera que este problema quede resuelto gracias al control del ciclo de vida de la API y sus versiones que el producto aporta.

10. Trazas y motorización insuficientes. Se ha ampliado la trazabilidad por defecto del producto para mostrar que se puede incluso monitorizar el contenido de los mensajes y respuestas en caso de ser necesario.

En cuanto a los requisitos de seguridad, se han cubierto los siguientes:

- Disponibilidad. Se ha realizado una prueba que controla el nivel de consumo de los APIs, de tal manera que, incluso salvando la autenticación y obteniendo un token válido, el atacante no podrá hacer ataques de denegación de servicio sobre el API pues está limitado su consumo máximo por minuto.

- Confidencialidad. Se aporta limitando el protocolo de comunicación a HTTPS, deshabilitando HTTP.

- Integridad. Al igual que en el caso anterior, la limitación aplicada de solamente permitir HTTPS, proporciona esta garantía (se supone que el caso de uso se aplicaría sobre un certificado digital válido para el DNS sobre el que se montase el Gateway).

- Autentificación. Se ha implementado seguridad OAuth2, de forma que todas las peticiones que reciba el API han de superar la autenticación basada en este protocolo.

- Trazabilidad. Se ha hecho un a prueba de trazabilidad de las peticiones, capturando las cabeceras y el cuerpo de cada petición.

En base a este análisis, el alcance propuesto para este trabajo ha sido cumplido, proporcionando con el API Gateway un nivel de protección bastante alto frente a los principales ataques que reciben las APIs REST. Sin el API Gateway, el API programada no tendría ninguna de estas protecciones y habría que buscar otros métodos (programáticos, por ejemplo) para poder exponer el API sin excesivos riesgos.

.5. Conclusiones

A lo largo de la elaboración de este trabajo he extraído muchas conclusiones y algunas lecciones aprendidas. De entre todas ellas destaco las siguientes:

- Hoy por hoy, admite poco debate la necesidad de proteger cualquier API REST expuesta por una organización. Se puede argumentar la conveniencia de ciertas medidas o requisitos de seguridad en función de los diferentes escenarios, pero no gobernar el acceso a las mismas es un grave error.

- Los productos de API Management son una herramienta que puede resultar muy útil para el gobierno y protección de APIs REST. Hay diferentes tipologías, licenciamiento y capacidades, existiendo la variedad suficiente como para que sirvan en un conjunto de escenarios muy diversos.

- WSO2 API Manager es un producto muy completo. No sólo ha servido para proteger el caso práctico que se ha planteado en este trabajo, si no que se intuye de todas las opciones de configuración que ofrece que puede cubrir muchos requisitos de seguridad con los mecanismos de protección que ofrece.

- Ofrecer la posibilidad de aplicar una capa de seguridad de manera transversal a los desarrollos, olvidándose del lenguaje de implementación de los mismos, es una ventaja muy grande desde el punto de vista de los equipos que se dedican a la seguridad informática. Poder atacar las vulnerabilidades desde un único punto y tener control de lo que se expone si depender de la tecnología de implantación subyacente al API REST también lo es.

- Puede ser una molestia para los equipos de desarrollo, especialmente en metodologías ágiles, que las APIs gobernadas necesariamente deban ser definidas mediante una interfaz Swagger u OpenAPI, pero creo que es una buena práctica tanto para el mantenimiento evolutivo y correctivo de las mismas como para poder aplicar granularidad a la autorización de acceso.

- El framework Spring Boot es muy indicado para el desarrollo de APIs REST. Gracias a las librerías que incorpora, ofrece una serie de problemas resueltos en el trabajo con JSON, en la definición de interfaces para las APIs y el trabajo con bases de datos que hace que los tiempos de implementación se reduzcan considerablemente.

- Aunque HTTPS es un mecanismo de seguridad que puede resolver potencialmente muchos requisitos de seguridad por si mismo, la combinación con tecnologías de Token como JWT hacen una solución mas escalable y simple sin perder necesariamente robustez.

- OAuth es un mecanismo de autentificación y autorización muy amplio en lo que a protección de APIs REST se refiere. Sus flujos o Grants permiten aportar a esta capa mucha granularidad cubriendo prácticamente todos los escenarios que se suelen dar en la interacción existente entre las aplicaciones consumidoras del API y el productor del mismo.

En cuanto al cumplimiento de los objetivos, recordando los objetivos planteados al inicio del proyecto, se analiza su gradosu grado de cumplimiento:

- **Conocer su situación actual de implantación y futuro en las tecnologías de información y comunicación de los APIs REST.** Se ha hecho una investigación sobre el nivel de uso que tienen, cual es el nivel de implantación que tienen en la industria y tratado de extrapolar lo importantes que pueden resultar en los próximos años.

- **Explicación de los conceptos de API y REST y para qué necesidades escenarios se están utilizando**. Se ha detallado el marco conceptual de estas arquitecturas y planteado los escenarios en los que se están utilizando en los sistemas de información, explicando cuales son los estándares de definición de APIs más utilizados y cúal es su formato.

- **Conocer qué estrategias se están utilizando para la gestión y gobierno de APIs.** Se ha descrito en que consiste el gobierno de APIs, la necesidad de que exista y las ventajas que otorga. Se han analizado diferentes soluciones de productos de API Management y escogido uno válido para los propósitos de este trabajo.

- **Analizar los principales riesgos y amenazas a las que se exponen los APIs REST**. Se ha realizado una investigación sobre cuáles son los 10 riesgos o vulnerabilidades más explotadas en los APIs REST, explicando en que consiste cada una de ellas.

- **Analizar los mecanismos y soluciones para la protección de APIs REST .** Se han descrito que mecanismos de seguridad se han utilizado para conseguir que los requisitos de seguridad detallados se cumplan y como se iban a aplicar al caso de uso propuesto.

- **Definir un API REST e implementarla en un lenguaje de programación**. Se ha generado un API REST ficticio apoyándose en los estándares descritos en la fase de análisis. Posteriormente se ha implementado en lenguaje de programación java y se ha comprobado su correcto funcionamiento.

- **Implementación de ejemplos reales de posibles ataques sobre el API REST definida**. Partiendo de los ataques descritos en la fase de análisis se han ejecutado ejemplos que probaban como el API originalmente podía ser vulnerable a los mismos.

- **Implementar prueba de concepto de protección sobre los ataques identificados sobre el API implementado y comprobar la protección obtenida.** Se ha descrito como configurar el producto WSO2 API Manager para paliar total o parcialmente los ataques descritos y repetido las pruebas para ver si la vulnerabilidad había sido paliada.

No ha quedado ninguno de los objetivos sin cumplir. La metodología ha resultado efectiva, las cargas de trabajo han demostrado estar bien dimensionadas, pues se han podido ir cumpliendo las fases del proyecto en las entregas parciales. No han sido por tanto necesarios cambios en la misma y se ha llegado al alcance perseguido en el trabajo, aunque debido a la profundidad que puede llegar a alcanzar el tema tratado, en ocasiones ha sido un poco difícil saber hasta que punto había que llegar.

Las líneas de trabajo futuro que han podido quedar pendientes serían, principalmente, profundizar en algunos de los temas tratados. Por ejemplo, en los mecanismos de autenticación y autorización, se ha abordado sólo uno de los escenarios que el estándar OAuth cubre. Sería muy interesante poner a prueba otros escenarios para ver qué grado de protección se obtendría con la solución propuesta. Otro de los puntos interesantes que admite mucha mejora es la posibilidad de apoyar la trazabilidad del producto WSO2 API Manager en alguna herramienta de gestión de eventos, ya fuese un SIEM o simplemente alguna herramienta de cuadro de mando basada en ELK o Kibana, por ejemplo, ya que el formato de los ficheros de traza que ofrece el producto son difícilmente explotables en formato y forma. Por último, otro aspecto muy interesante de cara a la seguridad habría sido probar las posibilidades que el producto WSO2 API Manager ofrece en la programación de mediaciones. Estas mediaciones permiten programar protecciones más específicas que pudieran no estar resueltas en las opciones que trae el producto implementadas. Trabajar el API del propio producto y ver cuán efectivas pueden resultar estas extensiones a la seguridad resuelta por el producto sería muy interesante de cara a cubrir otros ataques no especificados en el marco de este proyecto.

.

.6. Glosario

APIs: Api Programing Interface o intefaz de programación de aplicaciones

REST: representational state transfer

RESTFuII: API o servicio que respeta todos los fundamentos arquitectonicos de REST

URI: uniform resource identifier o Identificador de recursos uniforme. Cadena de
caracteres que identifica los recursos de una red de forma unívoca

HTTP: Hypertext Transfer Protocol o Protocolo de transferencia de hipertexto, ese un protocolo de comunicación que permite las transferencias de información a través de archivos

HTTPS: versión segura del protocolo HTTP, que con criptografía asimétrica proporciona varios opciones de protección de la comunicación frente a su versión no segura.

verbo HTTP: parte de la sintaxis del protocolo HTTP que permite especificar la acción sobre el recurso URI (ejemplos, GET,POST,etc)

JWT: JSON Web Token es un estándar abierto basado en JSON propuesto por IETF (RFC 7519) para la creación de tokens de acceso que permiten la propagación de identidad y privilegios o claims en inglés

JWE: JSON Web Encryption es un estándar IETF que proporciona una sintaxis estandarizada para el intercambio de datos cifrados, basada en JSON y Base64. Está definido por RFC7516

JWS: JSON Web Signature es un estándar propuesto por IETF para la firma de JSON arbitrario. Se utiliza como base para una variedad de tecnologías basadas en la web, incluyendo JSON Web Token.

CRUD: Create, Read, Update and Delete, acrónimo para referirse a las operaciones básicas de trabajo con datos (consulta, modificación, creacion y borrado)

HATEOAS: Hypermedia as the Engine of Application Stateo Hipermedia como motor del estado de la aplicación. Formá dinamica de adjuntar hipervinculos en las respuestas que permita al cliente acceder a información relacionada

SOAP: Simple Object Access Protocol, es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.

XML/RPC: XML Remote Procedure Call. Es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos y HTTP como protocolo de transmisión de mensajes.

XML: eXtensible Markup Language. es un metalenguaje que permite definir lenguajes de marcas desarrollado por el World Wide Web Consortium (W3C), utilizado para almacenar datos en forma legible

W3C: World Wide Web Consortium, es un consorcio internacional que genera recomendaciones y estándares que aseguran el crecimiento de la World Wide Web a largo plazo

JSON: JavaScript Object Notation, es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera (año 2019) un formato independiente del lenguaje.

Swagger: Es una especificación para archivos de interfaz legibles por máquina para describir, producir, consumir y visualizar servicios web RESTful.

OpenApi: Estandarización como software libre de la especificación Swagger

YAML: YAML Ain't Markup Language, YAML es un formato de serialización de datos legible por humanos inspirado en lenguajes como XML.

IoT: Internet of things, es un concepto que se refiere a una interconexión digital de objetos cotidianos con internet.

Docker: es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software.

AWS: Amazon Web Services es una colección de servicios de computación en la nube pública (también llamados servicios web) que en conjunto forman una plataforma de computación en la nube, ofrecidas a través de Internet por Amazon.com

OAuth: Open Authorization, es un estándar abierto que permite flujos simples de autorización para sitios web o aplicaciones informáticas.

RHEL: Red Hat Enterprise Linux, es una distribución comercial de GNU/Linux desarrollada por Red Hat.

On Premise: hace referencia a la instalación de un software en el Centro de datos corporativo de una empresa.

backend: hace referencia, en arquitectura distrribuidas, a las máquinas en la que la capa de lógica de negocio se ejecuta, quedando separada de la presentación de datos.

PC: Personal Computer, ordenador personal.

DMZ: zona desmilitarizada, o red perimetral es una red local que se ubica entre la red interna de una organización y una red externa, generalmente en Internet.

Firewall: cortafuegos , es la parte de un sistema informático o una red informática que está diseñada para bloquear el acceso no autorizado, permitiendo al mismo tiempo comunicaciones autorizadas.

proxy: Un proxy, o servidor proxy, en una red informática, es un servidor —programa o dispositivo—, que hace de intermediario en las peticiones de recursos que realiza un cliente (A) a otro servidor (C).

framework: Un entorno de trabajo es una estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software

SLA: Service Level Agreement. Acuerdo de nivel de servicio . Es un acuerdo escrito entre un proveedor de servicio y su cliente con objeto de fijar el nivel acordado para la calidad de dicho servicio

OpenID: OpenID es un estándar de identificación digital descentralizado, con el que un usuario puede identificarse en una página web a través de una URL (o un XRI en la versión actual) y puede ser verificado por cualquier servidor que soporte el protocolo.

.7. Bibliografía

Libros Consultados

- Richer, J. <<OAuth 2 in action>>. Manning publishers ISBN:978-1-61729-327-6. 2017

- Balaji Varanasi, <<Spring REST>> Apress. Sudha Belida Released. ISBN: 9781484208236. Junio 2015

Paginas web consultadas

- ¿Qué es una API de REST? https://www.redhat.com/es/topics/api/what-is-a-rest-api

- API REST: qué es y cuáles son sus ventajas en el desarrollo de proyectos https://www.bbvaapimarket.com/es/mundo-api/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos/

- Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST) https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_1

- OpenAPI: un formato de descripción abierto para los servicios de API - IONOS https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-openapi/

- OpenAPI-Specification/3.0.3.md at master · OAI/OpenAPI-Specification · GitHub https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md#oasDocument

- The Forrester Wave : API Management Solutions, Q3 2020 https://wso2.com/resources/analyst-reports/the-forrester-wave-api-management-solutions-q3-2020/?utm_source=go&utm_medium=cpc&utm_campaign=go_cpc_apim_spain_fw_210222&gclid=Cj0KCQjwmIuDBhDXARIsAFITC_7z4883UU-JvuDQFgoztT8WVv8uHU8LiRma33Ru30VS_ScVftlOwywaAlBYEALw_wcB

- Tyk y Kong: analizamos estos dos API Gateways | BBVA https://www.bbva.com/es/tyk-kong-analizamos-estos-dos-api-gateways/
- ¿Qué es la seguridad de las API? https://www.redhat.com/es/topics/security/api-security

- ¿Cuál es el mejor método de autentificación en un API REST? https://www.itdo.com/blog/cual-es-el-mejor-metodo-de-autentificacion-en-un-api-rest/

- An Introduction to OAuth 2 | DigitalOcean https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2

- API Data Breaches in 2020 - CloudVector https://www.cloudvector.com/api-data-breaches-in-2020/

- OWASP API Security - Top 10 | OWASP https://owasp.org/www-project-api-security/

- WSO2 API Manager Documentation | WSO2 Oficial site https://apim.docs.wso2.com/

- Building REST services with Spring |Spring official site https://spring.io/guides/tutorials/rest/

Videos consultados

- Oscar Blancate. INIT. “Como crear tu primer API REST”. URL https://youtu.be/K-KUfg-Cuc8

.