Universitat Oberta
de Catalunya
uoc.edu

Security in API and API managers

Miguel Amengual Bauza
Máster universitario en Seguridad de las TIC
Protocolos y aplicaciones de seguridad

Manuel Jesus Mendoza Flores
Victor Garcia Font

4 de junio de 2019

Esta obra está sujeta a una licencia de Reconocimiento 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Security in API and API managers*
Nombre del autor:	*Miguel Amengual Bauza*
Nombre del consultor/a:	*Manuel Jesus Mendoza Flores*
Nombre del PRA:	*Victor Garcia Font*
Fecha de entrega:	06/2019
Titulación:	*Máster universitario en Seguridad de las TIC*
Área del Trabajo Final:	*Protocolos y aplicaciones de seguridad*
Idioma del trabajo:	*Español*
Palabras clave	*API security, API standards, microservices.*

Resumen del Trabajo:

La finalidad de este trabajo es estudiar cuales son los factores que hacen que una API sea segura. Así como conocer los métodos, estándares y herramientas que facilitan esta labor.

En primer lugar, se estudian los ataques más comunes y como mitigarlos. Por ejemplo, para mitigar el impacto de un ataque DoS se puede usar un API Gateway para devolver información de la cache y un WAF para bloquear llamadas que no tengan la estructura deseada.

En segundo lugar, se presentarán estándares para la definición de la API como OAS o API Blueprint, dada la importancia de pensar en la seguridad desde el diseño.

A continuación, se ve OAuth2 como estándar de uso extendido para solucionar los problemas de autorización entre aplicaciones que quieren compartir datos.

Además, hablamos de una nueva forma de construir aplicaciones a partir de microservicios, los cuales se comunican entre ellos a través de sus APIs.

Finalmente, se introduce el concepto de API Management System como un proceso que engloba las tareas de publicar, promocionar y supervisar APIs en un entorno seguro.

La metodología seguida ha centrado estos primeros capítulos del trabajo en el estudio teórico de los conceptos para luego poder aplicarlos al caso práctico. Este se ha resuelto satisfactoriamente ofreciendo una solución basada en la arquitectura de microservicios, con un API Gateway y haciendo uso de OAuth2.

Podemos concluir que este trabajo ha sido muy útil para ofrecer una visión global de la seguridad de las APIs y se han conseguido los objetivos iniciales.

Abstract:

The purpose of this thesis is to study what are the factors that make an API safe. As well as knowing the processes, standards and tools that facilitate this work.

First of all, we study the most common attacks and how to mitigate them. For example, to mitigate the impact of a DoS attack, we can use an API Gateway to return information from the cache and a WAF to block calls that do not have the desired structure.

Secondly, we will introduce standards for the definition of the APIs. These are OAS and API Blueprint. They emphasize the importance of thinking about security from the design.

Then, we will see OAuth2 as the most used standard to solve the authorization issues between applications that want to share data.

In addition, we talk about a new way to build applications using microservices, which communicate with each other through their APIs.

Finally, the API Management System concept is introduced as a process that include the tasks of publishing, promoting and monitoring APIs in a secure environment.

The methodology used has been to focus these first chapters of the thesis on the theoretical study of the concepts. And then apply them to the practical case. This has been satisfactorily resolved by offering a solution based on the microservices architecture, with an API Gateway and using OAuth2.

We can conclude that this work has been really useful to offer a global vision of API security and the initial objectives have been achieved.

Índice

1. Introducción 1
  1.1 Contexto y justificación del Trabajo 1
  1.2 Objetivos del Trabajo 3
  1.3 Enfoque y método seguido 3
  1.4 Planificación del Trabajo 4
  1.5 Breve sumario de productos obtenidos 8
  1.6 Breve descripción de los otros capítulos de la memoria 8
2. Aspectos básicos de seguridad en APIs 9
  2.1 Ataques de denegación de servicio 10
  2.2 Ataques Man in the Middle 11
  2.3 Ataques a los parámetros de entrada 13
  2.4 Ataques técnicos, exploits 13
  2.5 Ataques a los métodos de autenticación 14
  2.6 Herramientas para mitigar ataques a APIs 15
3. Estándares de diseño de APIs 18
  3.1 OAS 3.0 18
  3.2 API Blueprint 23
4. OAuth 2.0 27
  4.1. OpenID Connect 31
5. Arquitectura de microservicios 33
6. API Managament System 38
7. Caso de estudio 41
8. Conclusiones y trabajo futuro 47
9. Glosario 49
10. Bibliografía 51
11. Anexos 52
  11.1 Documentación Product API 52
  11.2 Documentación Inventory API 56
  11.3 Documentación List API 63
  11.4 Implementación Product API 68
  11.5 Implementación Inventory API 70
  11.6 Implementación List API 73

Lista de figuras

Figura 1. Diagrama uso de APIs 1
Figura 2. Crecimiento publicación de APIs 2
Figura 3. Preocupaciones sobre seguridad en las APIs 9
Figura 4. Métodos para securizar las APIs 10
Figura 5. Ataque DDoS 11
Figura 6. Ataque Man in the middle 12
Figura 7. Ataque SSL Strip 12
Figura 8. Web Application Framework 16
Figura 9. API Gateway 17
Figura 10. Secciones OAS 3.0 19
Figura 11. OAS 3.0 - Info 20
Figura 12. OAS 3.0 - Servers 20
Figura 13. OAS 3.0 - Security (Definición) 20
Figura 14. OAS 3.0 - Security (Aplicación) 21
Figura 15. OAS 3.0 - Paths 21
Figura 16. OAS 3.0 - Tags (Aplicación) 21
Figura 17. OAS 3.0 - Tags (Definición) 22
Figura 18. OAS 3.0 – External Docs 22
Figura 19. OAS 3.0 - Components 22
Figura 20. Secciones API Blueprint 23
Figura 21. API Blueprint - Metadata 23
Figura 22. API Blueprint - API Name 23
Figura 23. API Blueprint - Resource Groups 24
Figura 24. API Blueprint - Resources 24
Figura 25. API Blueprint - Actions (Parameters) 24
Figura 26. API Blueprint - Actions (Attributes) 24
Figura 27. API Blueprint - Actions (Request) 25
Figura 28. API Blueprint - Actions (Response) 25
Figura 29. API Blueprint - Data Structures 26
Figura 30. Ejemplo de uso de OAuth2 PolarFlow - Strava 29
Figura 31. Flujo proceso autenticación OAuth2 30
Figura 32. Arquitectura aplicación CRM monolítica 33
Figura 33. Arquitectura aplicación CRM con microservicios 35
Ilustración 34. API Management System 38
Figura 35. Microservicios con API Gateway 40
Figura 36. OAS3 - Simple Product API 41
Figura 37. OAS3 - Simple Inventory API 42
Figura 38. OAS3 - Simple List API 42
Figura 39. Arquitectura caso práctico 43
Figura 40. Proceso de Login con Facebook Connect 44
Figura 41. Flujo de autenticación caso práctico 44
Figura 42. Web - Creación producto 45
Figura 43. Web - Creación Inventario 45
Figura 44. Web - Añadir producto al Inventario 45
Figura 45. Web - Modificar cantidad de un producto del Inventario 46
Figura 46. Web - Ver lista de la compra 46

1. Introducción

1.1 Contexto y justificación del Trabajo

Vivimos en un mundo en que constantemente se están añadiendo nuevos servicios a la red, permitiéndonos realizar múltiples tareas que antes solo se podían realizar de forma física.

Además, cada vez más, se espera que un servicio se pueda comunicar con otro, interconectando las diferentes tareas que realizamos conjuntamente para tenerlas en un mismo sitio.

Esta comunicación entre servicios suele realizarse a través de APIs. El término API es un acrónimo de Application Programming Interface, Tal como se define en [1] (Brajesh 2017), una API es una interfaz que establece un contrato para que las aplicaciones se puedan comunicar entre ellas sin la interacción del usuario. Por ejemplo, en el momento de reservar una habitación de hotel desde un portal de viajes online, este envia la información de la reserva al sistema de reservas del hotel elegido para bloquear la disponibilidad de la habitación. Además, también manda la información de la tarjeta de crédito a una aplicación de pago electrónico. La aplicación de pagos interactua con la aplicación del banco para validar la información y procesar el pago. Si el pago se ha realizado correctamente, la habitación se reservará con exito. La interacción entre el portal de viajes, el sistema de reservas del hotel y la aplicación de pagos se hace mediante sus APIs. Sin embargo, como usuario solo se percibe la parte de elegir la habitación y la de introducir la información de la tarjeta de crédito.

Una API proporciona una interfaz para las aplicaciones que quieran consumirlas. De esta forma, las aplicaciones pueden interacturar con la información de los servicios que se encuentran detrás de estas APIs (Figura 1).

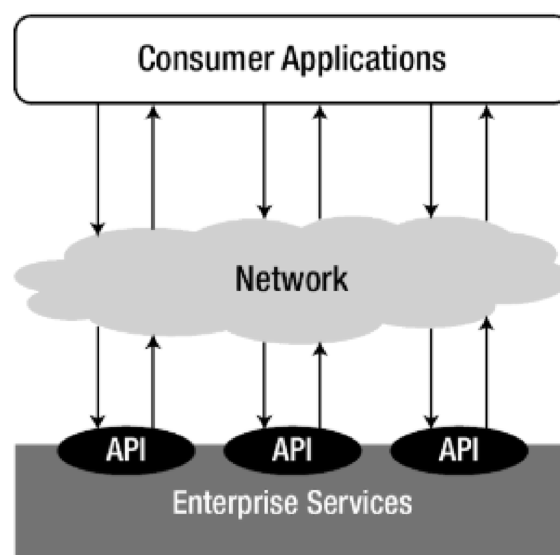


Figura 1. Diagrama uso de APIs

Este es uno de los motivos por el que muchas aplicaciones web han adoptado el modelo de API, mediante el cual exponen algunas de sus funciones para que puedan ser usadas por otros servicios. Haciendo pública su API, un servicio puede encontrar una nueva utilidad diferente de su principal cometido, por el cuál fue concebido.

El uso de APIs ha crecido exponencialmente desde 2010, tal como se puede observar en la siguiente figura, extraída de [6] (ProgrammableWeb):

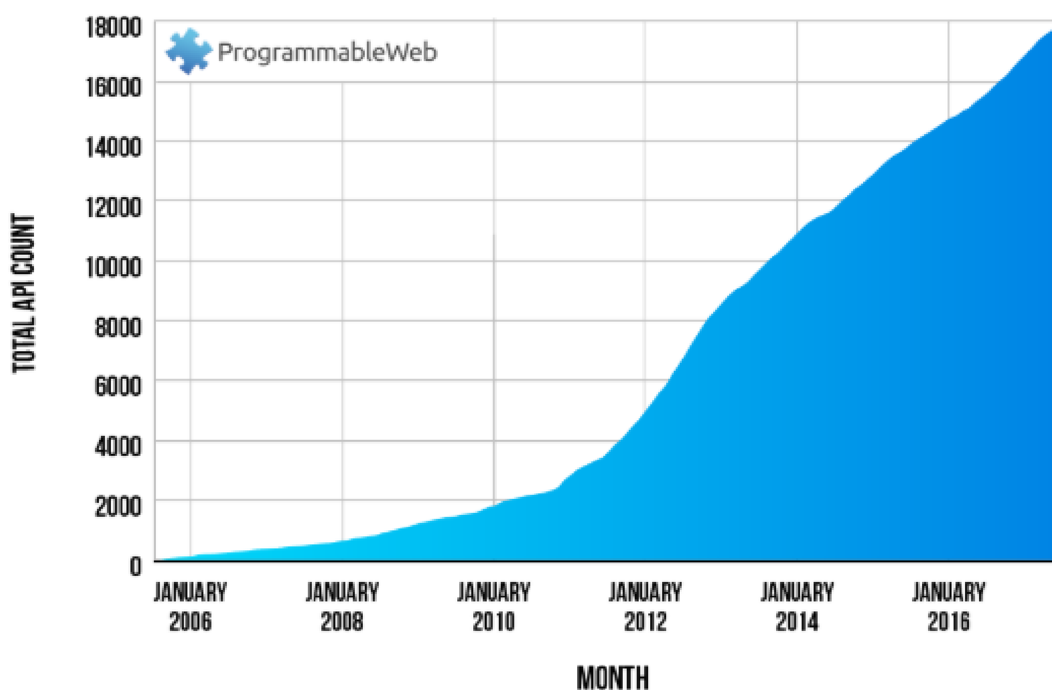


Figura 2. Crecimiento publicación de APIs

Este crecimiento viene asociado a una serie de ventajas que aporta el uso de APIs. En primer lugar, facilitan las integraciones con terceros ya que son fáciles de consumir y se pueden usar desde cualquier plataforma, ya sea una aplicación web, una aplicación móvil o un *dispositivo IoT* (1).
Además, son rápidas de desplegar, agilizando el desarrollo y los cambios. Otro factor importante es el bajo coste asociado a desplegar una API en un *entorno Cloud* (2). Finalmente, al permitir la comunicación entre aplicaciones, permite dividir grandes aplicaciones en funciones o *microservicios* (3) que puedan reutilizarse en diferentes ámbitos.

En el momento de publicar una API, se tiene que garantizar que esté protegida y sea segura para que tenga una utilidad real. En caso contrario, no podrá contener información sensible ni personalizada al usuario que hace la petición. Además, una API totalmente abierta sería víctima potencial de ataques de denegación de servicio.

Durante muchos años ha habido muchas soluciones privadas yendo y viniendo en los distintos aspectos de seguridad referentes a APIs. Las empresas líderes del mercado, así como organizaciones tecnológicas han ido adoptando soluciones similares que han acabado formalizadas en estándares. Un ejemplo

es OAuth, que surge como respuesta a las necesidades básicas de autenticación y autorización, reemplazando muchas de las soluciones propias por una tecnología estándar, con los beneficios que esto aporta.

Entre los objetivos de este estudio está investigar acerca de las soluciones y estándares más utilizados para ofrecer seguridad a las APIs. Así como también, conocer los API managers, porque motivo surgen, y su utilidad en un entorno dónde muchas organizaciones están evolucionando hacia el uso de *microservicios* (3), compartiendo un sistema de autenticación común para todos ellos.

Finalmente, se propondrá una solución para un caso práctico. Esta consistirá en exponer un conjunto de APIs de distinto ámbito bajo un sistema de autenticación común. También se realizará una aplicación web para mostrar como sería el acceso a cada una de las APIs.

1.2 Objetivos del Trabajo

- Conocer los principales riesgos a tener en cuenta a la hora de proteger una API. Así como métodos para mitigar su impacto.
- Conocer de los métodos y los estándares más comunes a la hora de desarrollar una API.
- Conocer el origen y la evolución de OpenAPI Specification (OAS) y de OpenAPI Initiative (OAI).
- Conocer los estándares OAS3 y API Blueprint.
- Conocer como la industria de las tecnologías de la información está gestionando y dirigiendo los estándares de seguridad.
- Conocer cómo se puede incluir la seguridad desde el diseño de la API.
- Conocer el estándar OAuth2 y su papel como mecanismo de autenticación y de autorización.
- Conocer el funcionamiento de un API Manager y cuando se usan.
- Conocer diferentes API Managers y compáralos entre ellos.
- Conocer la arquitectura de microservicios basados en API.
- Conocer el concepto Seguridad como servicio.
- Realizar un ejemplo completo de como varios microservicios pueden compartir un mismo sistema de autenticación y autorización.

1.3 Enfoque y método seguido

El último objetivo de este proyecto es proponer una solución para proporcionar seguridad a diferentes servicios en forma de API de una misma organización. Para tal objetivo, se puede desarrollar un producto nuevo, usando o no estándares. O se puede optar por usar y adaptar un producto existente. Esta decisión se tomará en base al conocimiento adquirido desarrollando los próximos capítulos.

El punto de partida será el estudio de los principales riesgos de seguridad que afectan a las APIs y su entorno.

A continuación, estudiaremos los estándares más comunes para desarrollo de APIs. Así como también los productos que encabezan el mercado en la actualidad.

En segundo lugar, haremos una búsqueda de las herramientas disponibles tal como API managers y valoraremos la viabilidad de usar alguno de ellos como punto de partida.

De esta forma, durante la primera fase iremos cumpliendo la mayoría de los objetivos de este proyecto que son de investigación y comparativa de soluciones disponibles. Y durante la segunda fase, asentaremos estos conocimientos y los aplicaremos de forma práctica.

1.4 Planificación del Trabajo

Los recursos necesarios serán documentación oficial de los diferentes estándares y soluciones propietarias. Así como libros y artículos que contengan información general, tendencias y comparativas.

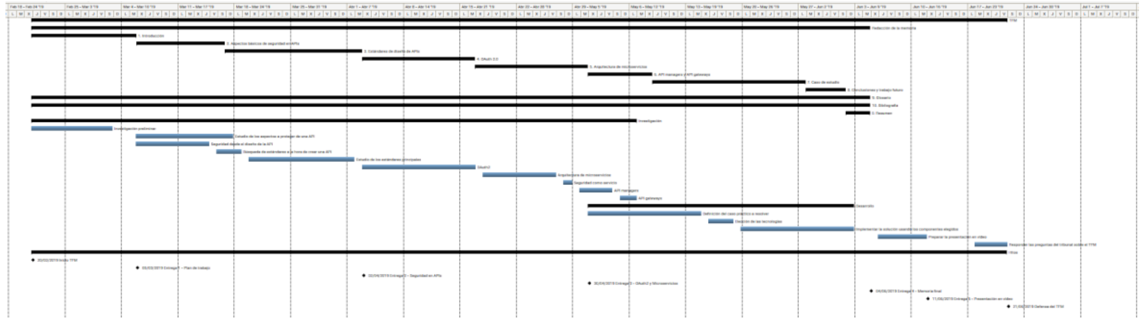
Las tareas de alto nivel con las que hemos dividido el proyecto son las siguientes:

Nombre	Duración	Esfuerzo	Inicio	Fin	Predecesoras
⊟TFM	122.25días	240horas	20/02/2019	21/06/2019	
⊟Redacción de la memoria	105días	71horas	20/02/2019	04/06/2019	47II
⊞1. Introducción	13.8días	12horas	20/02/2019	05/03/2019	30II
⊞2. Aspectos básicos de seguridad en APIs	11días	6horas	05/03/2019	16/03/2019	3
⊞3. Estándares de diseño de APIs	17días	5horas	16/03/2019	02/04/2019	8
⊞4. OAuth 2.0	14días	10horas	02/04/2019	16/04/2019	10
⊞5. Arquitectura de microservicios	14días	10horas	16/04/2019	30/04/2019	12
⊞6. API managers y API gateways	8días	6horas	30/04/2019	08/05/2019	14
⊞7. Caso de estudio	19días	10horas	08/05/2019	27/05/2019	16
⊞8. Conclusiones y trabajo futuro	5días	6horas	27/05/2019	01/06/2019	18
⊞9. Glosario	105días	2horas	20/02/2019	04/06/2019	
⊞10. Bibliografia	105días	2horas	20/02/2019	04/06/2019	
⊞0. Resumen	3días	2horas	01/06/2019	04/06/2019	20
⊟Investigación	76.25días	115horas	20/02/2019	06/05/2019	47II
Investigación preliminar	11días	12horas	20/02/2019	02/03/2019	
Estudio de los aspectos a proteger de una API	13días	10horas	05/03/2019	17/03/2019	48
Seguridad desde el diseño de la API	10días	20horas	05/03/2019	14/03/2019	31II
Búsqueda de estándares a la hora de crear una API	4días	6horas	15/03/2019	18/03/2019	32
Estudio de los estándares principales	14días	16horas	19/03/2019	01/04/2019	33
OAuth2	15días	20horas	02/04/2019	16/04/2019	34
Arquitectura de microservicios	10días	11horas	17/04/2019	26/04/2019	35
Seguridad como servicio	2días	4horas	27/04/2019	28/04/2019	36
API managers	5días	10horas	29/04/2019	03/05/2019	37
API gateways	3días	6horas	04/05/2019	06/05/2019	38
⊟Desarrollo	34días	35horas	30/04/2019	02/06/2019	50
Definición del caso práctico a resolver6	15días	7horas	30/04/2019	14/05/2019	
Elección de las tecnologías	4días	2horas	15/05/2019	18/05/2019	41
Implementar la solución usando los componentes ele	15días	26horas	19/05/2019	02/06/2019	42
Preparar la presentación en vídeo	7días	14horas	05/06/2019	11/06/2019	51FI+1día
Responder las preguntas del tribunal sobre el TFM	5días	5horas	17/06/2019	21/06/2019	53FF
⊟Hitos	121.25días	0hora	20/02/2019	21/06/2019	
Inicio TFM	0hora	0hora	20/02/2019	20/02/2019	
Entrega 1 - Plan de trabajo	0hora	0hora	05/03/2019	05/03/2019	
Entrega 2 - Seguridad en APIs	0hora	0hora	02/04/2019	02/04/2019	
Entrega 3 - OAuth2 y Microservicios	0hora	0hora	30/04/2019	30/04/2019	
Entrega 4 - Memoria final	0hora	0hora	04/06/2019	04/06/2019	
Entrega 5 - Presentación en vídeo	0hora	0hora	11/06/2019	11/06/2019	
Defensa del TFM	0hora	0hora	21/06/2019	21/06/2019	

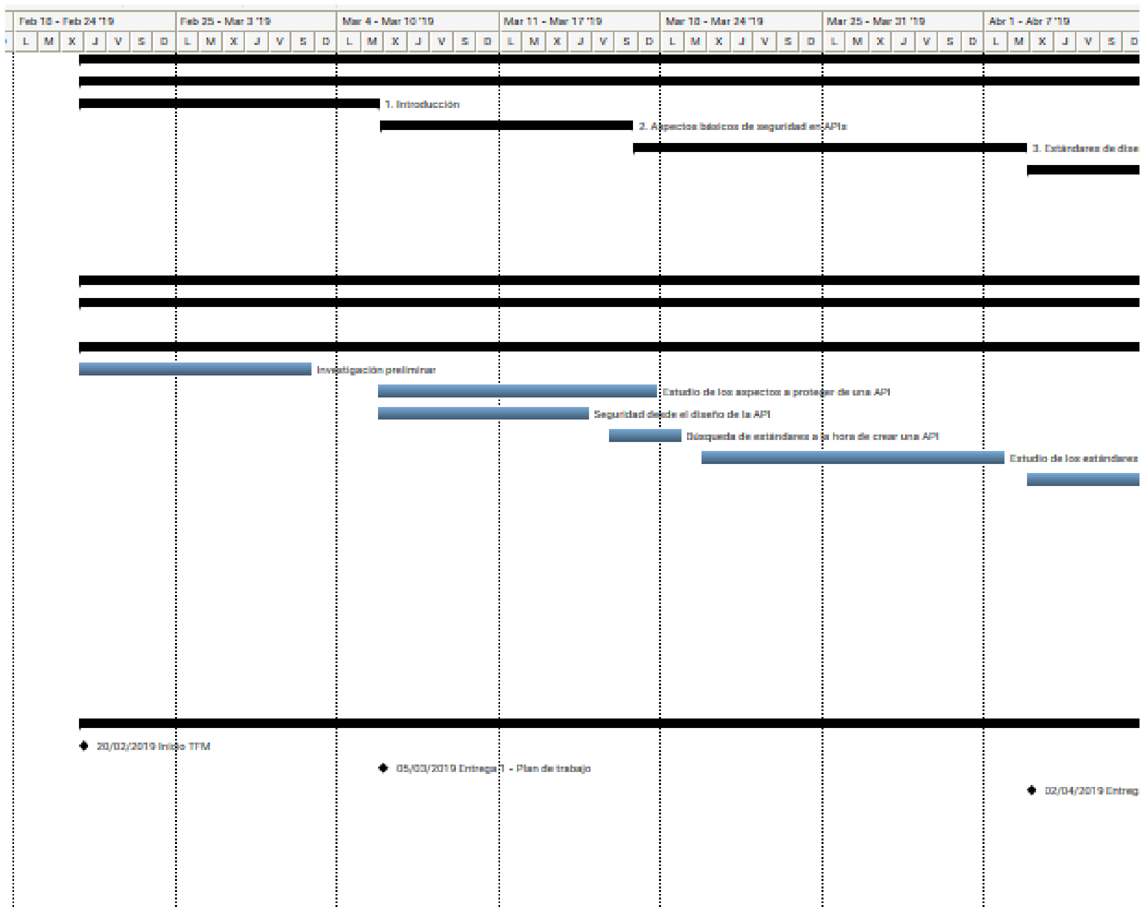
En cuanto a la clasificación de las tareas, hemos hecho 3 bloques: memoria, dónde se computa el tiempo en redactar esta memoria; investigación, tiempo para buscar información, clasificarla y entenderla; y desarrollo, tiempo para el desarrollo técnico de la solución para el caso práctico.

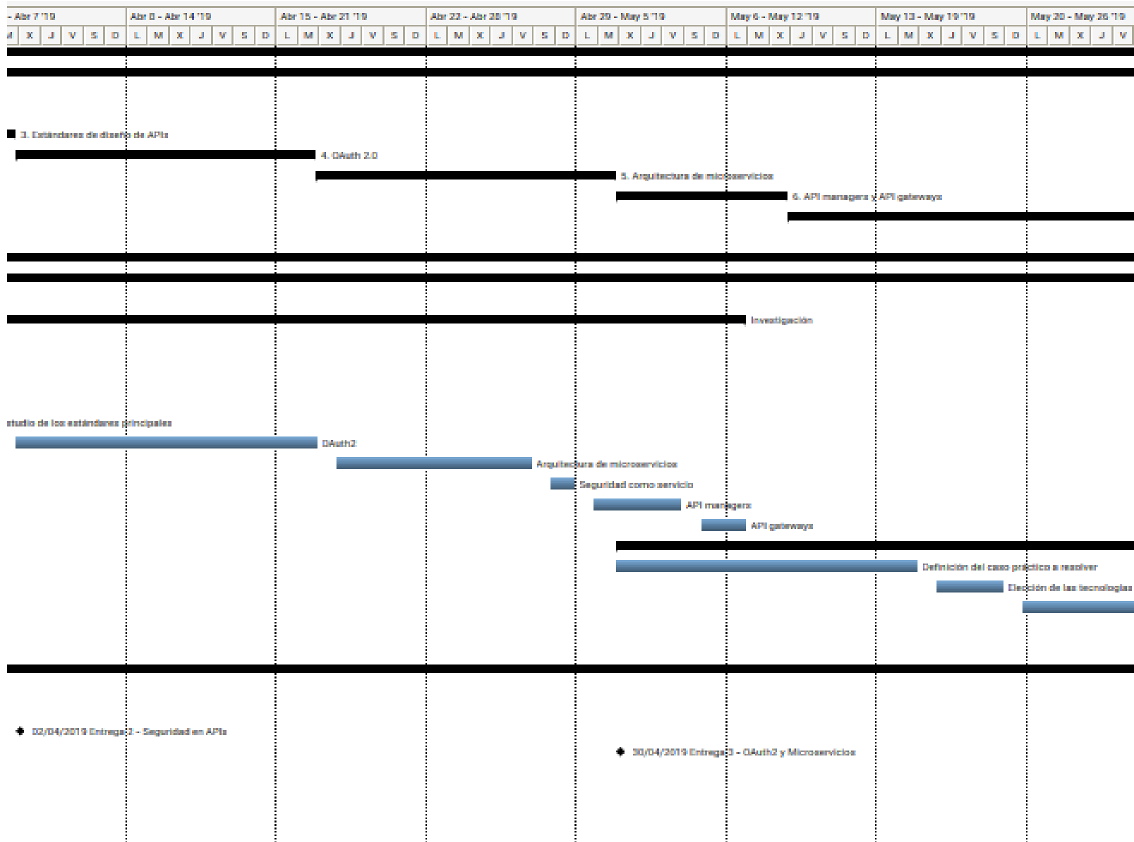
A la hora de confeccionar este Gantt lo que hemos hecho ha sido asignar el esfuerzo que conlleva cada tarea en horas, y a continuación asignar un recurso cuyo calendario es de 2 horas diarias de trabajo. De esta forma, hemos ido estableciendo la duración en días de cada tarea, teniendo en cuenta que si se trabaja en 2 cosas a la vez esta duración se alarga.

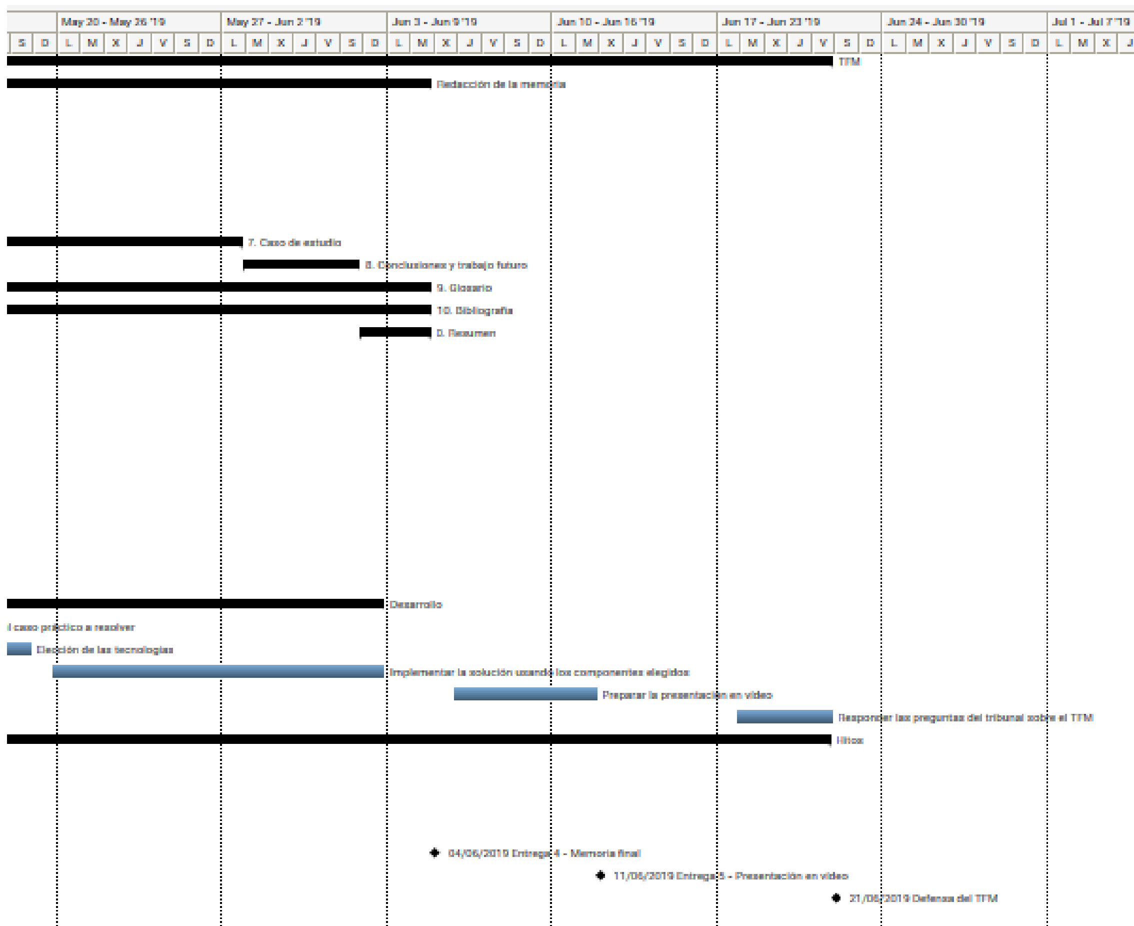
En la página siguiente vemos como ha quedado el diagrama:



Para poder ver con más detalle cada una de las partes, hemos dividido el diagrama en 3 partes y las hemos ampliado:




- Abr 7 '19
Abr 8 - Abr 14 '19
Abr 15 - Abr 21 '19
Abr 22 - Abr 28 '19
Abr 29 - May 5 '19
May 6 - May 12 '19
May 13 - May 19 '19
May 20 - May 26 '19
3. Estándares de diseño de APIs
4. OAuth 2.0
5. Arquitectura de microservicios
6. API managers y API gateways
Investigación
studio de los estándares principales
OAuth2
Arquitectura de microservicios
Seguridad como servicio
API managers
API gateways
Definición del caso práctico a resolver
Elección de las tecnologías
02/04/2019 Entrega 2 - Seguridad en APIs
30/04/2019 Entrega 3 - OAuth2 y Microservicios


May 20 - May 26 '19
May 27 - Jun 2 '19
Jun 3 - Jun 9 '19
Jun 10 - Jun 16 '19
Jun 17 - Jun 23 '19
Jun 24 - Jun 30 '19
Jul 1 - Jul 7 '19
TFM
Redacción de la memoria
7. Caso de estudio
8. Conclusiones y trabajo futuro
9. Glosario
10. Bibliografía
0. Resumen
Desarrollo
l caso práctico a resolver
Elección de las tecnologías
Implementar la solución usando los componentes elegidos
Preparar la presentación en vídeo
Responder las preguntas del tribunal sobre el TFM
Hitos
04/06/2019 Entrega 4 - Memoria final
11/06/2019 Entrega 5 - Presentación en vídeo
21/06/2019 Defensa del TFM

1.5 Breve sumario de productos obtenidos

Los productos obtenidos son esta memoria, con los diferentes capítulos. La documentación de las APIs en formato OAS 3.0 y el código fuente de las APIs y de la aplicación web.

1.6 Breve descripción de los otros capítulos de la memoria

En primer lugar, en el capítulo 2, vamos a estudiar los aspectos básicos en cuanto a seguridad de las APIs. Para ello, veremos los vectores de ataques más comunes y como se puede mitigar o reducir su impacto. También veremos herramientas que nos facilitan mucho este trabajo. Una vez introducidos los principales riesgos, veremos como debe ser un buen diseño de una API. Para ello, estudiaremos los estándares principales en el capítulo 3.

En el capítulo 4 vamos a hablar de OAuth2 dado su gran uso en el mercado y el importante papel que juega hoy en día para poder autorizar la comunicación entre diferentes aplicaciones de forma segura. Al surgir esta posibilidad de interconexión entre aplicaciones, tiene cada vez más sentido hacer múltiples aplicaciones con un alcance bien definido y conectarlas en lugar de hacer una gran aplicación monolítica. Hablaremos sobre la arquitectura de microservicios en el capítulo 5.

En el último capítulo teórico, el 6, hablaremos sobre el papel de los API Gateway y API Managers. Finalmente, en el capítulo 7 haremos un caso práctico de principio a fin. En el que diseñaremos un conjunto de APIs usando el estándar OAS, las implementaremos y haremos una aplicación que las utilice. Para usarlas el usuario deberá estar autenticado y para ello usaremos el estándar OAuth2.

2. Aspectos básicos de seguridad en APIs

Las APIs orientadas al público son una preocupación clave de seguridad, dado que son un vector directo de entrada a los datos confidenciales que hay detrás de las aplicaciones.

De acuerdo a una encuesta por [3] (Imperva 2018) a 250 profesionales pertenecientes al campo de las tecnologías de la información, las principales preocupaciones en cuanto a seguridad de las APIs son las que aparecen en la Figura 3.

Figura 3. Preocupaciones sobre seguridad en las APIs

Podemos ver que la principal preocupación está relacionada con *bots* (4) y ataques de denegación de servicio con un 39.2%. En segundo lugar, con un 24.4% preocupa cómo se aplica la autenticación.
Les siguen con porcentajes similares (14.8% y 13.6%) la inspección del contenido de la API para detectar ataques y la necesidad de analizar los recursos usados por las APIs.

En esta misma encuesta, se preguntó que métodos se usan para protegerse ante posibles ataques (Figura 4). Dos de cada tres personas reconocen hacer uso de un NAT. Con un porcentaje similar, los encuestados optan por el uso de Web Application Firewall y API Gateway. Casi la mitad asegura también usar métodos propios de seguridad a nivel de aplicación.

Figura 4. Métodos para securizar las APIs

En los siguientes apartados hemos desglosado los principales riesgos a los que está expuesta una API. Los hemos clasificado en 5 ámbitos: ataques DoS, ataques Man in the middle, ataques a los parámetros de entrada, ataques técnicos y ataques a los métodos de autenticación.
En cada uno de ellos, iremos dando pautas que nos pueden ayudar a estar más protegidos frente a estos ataques.

Finalmente, en el último punto analizaremos dos herramientas muy usadas para mitigar estos ataques: WAF y API Gateway.

2.1 Ataques de denegación de servicio

Un ataque de denegación de servicio (DoS) es un intento malintencionado de hacer que un servicio en línea deje de estar disponible para los usuarios. Consiste en inundar el objetivo con tráfico malicioso con la intención de interrumpir o suspender temporalmente los servicios del servidor dónde está alojado.

A menudo, un ataque DoS se lanza desde numerosos dispositivos comprometidos en lo que se conoce como *botnet* (5). En este caso el ataque se conoce como ataque de denegación de servicio distribuido (DDoS).

Como podemos ver en la Figura 5, extraída de [9] (Scudlayer), además del trafico normal que recibe el servidor se le suma el trafico malicioso que hace que el servidor se quede sin recursos y el servicio se vea interrumpido.

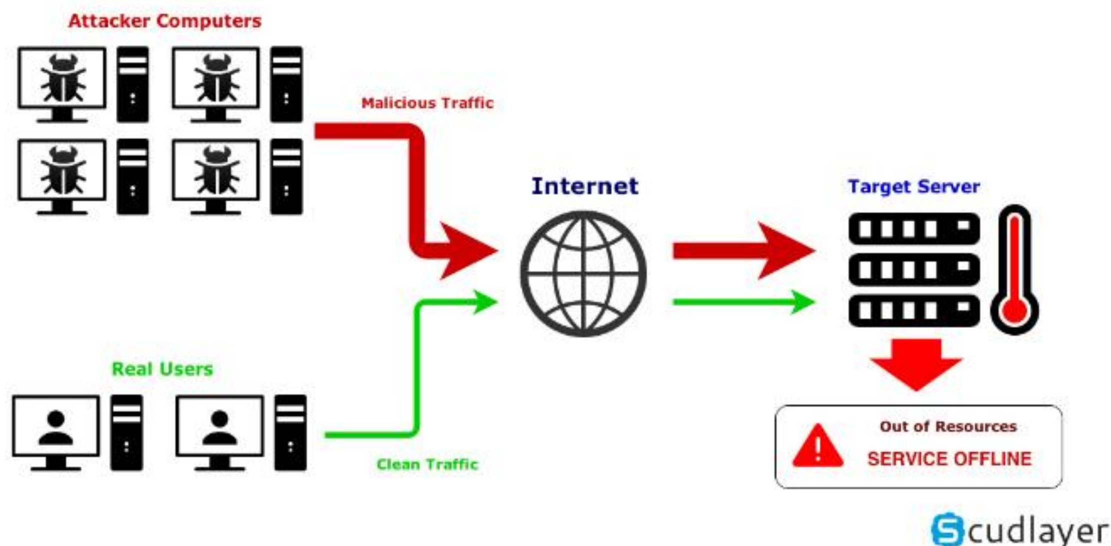


Figura 5. Ataque DDoS

Un ataque DDoS sobre una API, del mismo modo que sobre otro servicio, intenta sobrepasar su memoria y capacidad realizando múltiples conexiones concurrentes, o enviando/solicitando grandes cantidades de información en cada solicitud.

Para mitigar el peligro de estos ataques se puede usar un servicio de API Gateway para permitir almacenamiento en caché (Caching), limitar las peticiones (Rate Limit) y controlar los picos de peticiones (Spike Arrest). Además, se pueden limitar las peticiones provenientes de un mismo origen (Throttling).

Aunque se haga lo anterior, es una buena práctica utilizar colas de trabajo al lidiar con grandes cantidades de información. De esta forma, procesaremos tanto como sea posible en segundo plano y devolveremos una respuesta rápidamente para evitar un bloqueo HTTP.

Otra forma de disminuir la carga a procesar por el servidor y reducir el riesgo que se quede sin recursos es utilizar un *CDN* (6) para el contenido estático, principalmente para la descarga de ficheros pesados.

2.2 Ataques Man in the Middle

Una conexión no cifrada entre el cliente y el servidor API puede exponer una gran cantidad de datos confidenciales a los atacantes. Este puede interceptar la conexión, situándose en medio del servidor y el cliente. Tal como se puede ver en la Figura 6, toda la información pasaría por las manos del atacante desvelando información sensible del usuario sin que este se dé cuenta.

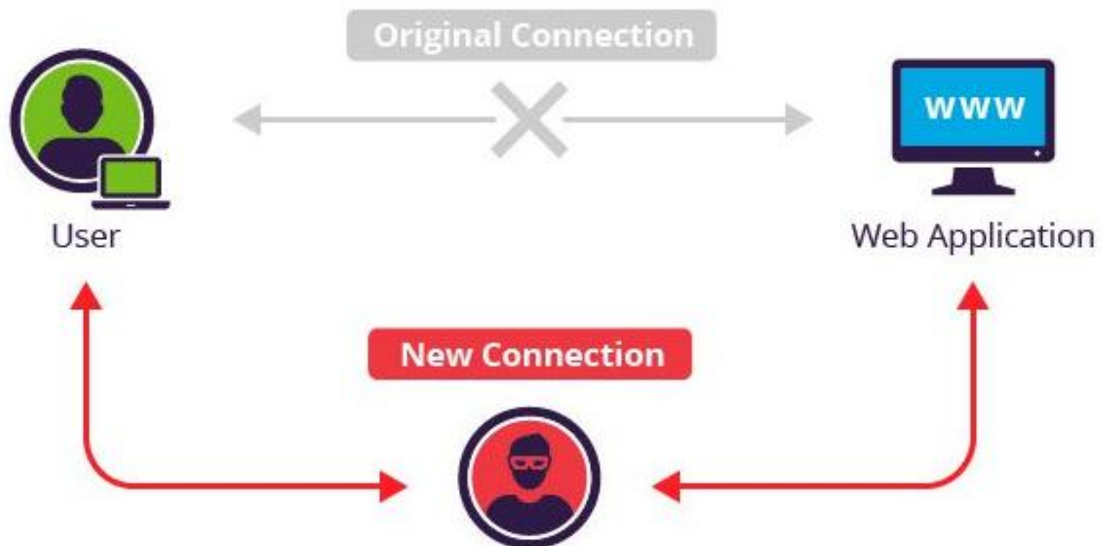


Figura 6. Ataque Man in the middle

Para evitar estos ataques se debe usar HTTPS en el lado del servidor. De esta forma, aunque se intercepte la comunicación los datos seguirán permaneciendo secretos, ya que están cifrados.

Además, se aconseja usar la cabecera HSTS con SSL para evitar SSL Strip Attack (Figura 7). Este ataque es una variante del Man in the middle que consiste en intervenir en la redirección de HTTP al protocolo seguro HTTPS e interceptar la solicitud del usuario al servidor. De esta forma, el atacante continuará estableciendo una conexión HTTPS entre él y el servidor, y una conexión HTTP no segura con el usuario.

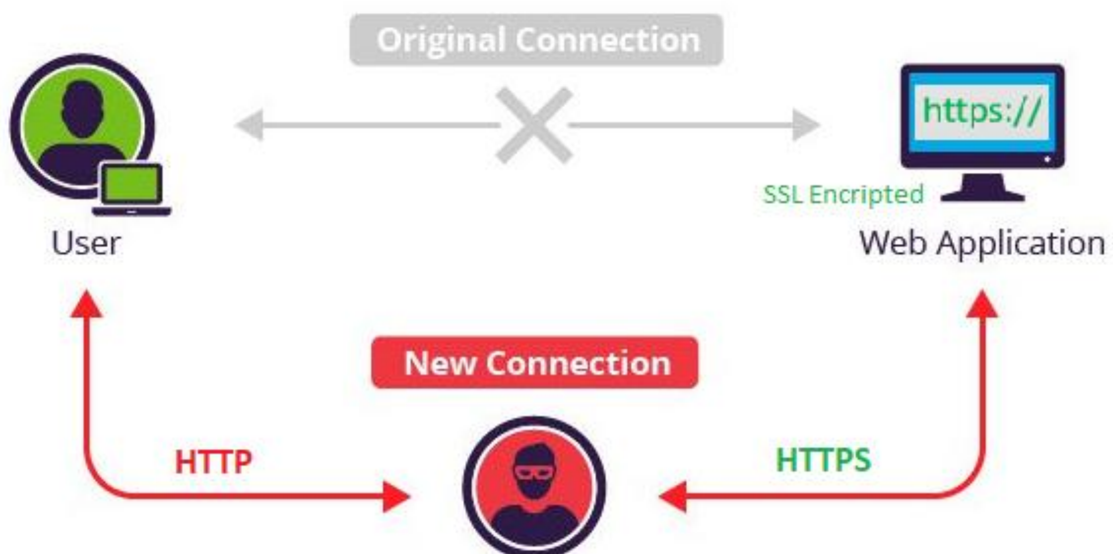


Figura 7. Ataque SSL Strip

La presencia de la cabecera HSTS indica al navegador que solo debe comunicarse con el servidor usando HTTPS.

2.3 Ataques a los parámetros de entrada

Uno de los métodos de explotación más comunes utilizados por los atacantes es poner a prueba la seguridad de las aplicaciones manipulando los parámetros de entrada. Se intentan cambiar los parámetros que se pasan entre el cliente y el servidor para intentar modificar el comportamiento normal de la API.

Un ejemplo sería cambiar el id de usuario de un campo oculto de un formulario para intentar modificar información de otra persona. O cambiar el valor del precio de un producto antes de enviar una petición de compra.

La API debe controlar los parámetros que pueden variar y los que no y asegurarse de que el usuario que hace la petición tiene acceso a la sección a la cuál intenta acceder.

Es una buena práctica es usar *UUID* [(7)] en lugar de identificadores auto incrementales para no dar la posibilidad a los atacantes de adivinar rutas a recursos a los cuales no deben tener acceso.
Del mismo modo, se debe evitar acceder a los recursos propios de un usuario mediante su identificador. Es mejor usar "/me/orders" en lugar de "/user/654321/orders"

Nos podemos proteger contra intentos ilegítimos de manipulación de los parámetros con un WAF. Con este servicio se pueden modelar como debe ser cada una de las llamadas a la API y comparar cualquier petición con la estructura de la API modelada para asegurar que los parámetros de entrada sean consistentes con la definición.

2.4 Ataques técnicos, exploits

Es posible que los atacantes inyecten en las peticiones contenido malicioso que podría conducir a ataques. Dichos ataques técnicos podrían ser XSS, SQL-Injection o Remote Code Execution, entre muchos otros.

Para mitigar estos ataques es muy importante validar las entradas y solo permitir los tipos de datos esperados.
También es importante ofrecer el mínimo detalle de la implementación y las tecnologías usadas a los atacantes potenciales ya que con esta información se pueden buscar vulnerabilidades concretas a estas tecnologías y explotarlas. Para tal efecto, se deben eliminar cabeceras que den información de la tecnología como X-Powered-By, Server, X-AspNet-Version, etc.

Así como validar que cada valor del parámetro es del tipo adecuado, se puede hacer una comprobación previa validando el content-type de la cabecera Accept (Content Negotiation). Solo se deben permitir sólo los formatos soportados. Por ejemplo, para las peticiones que se espera recibir un *JSON* [(8)]

solo aceptar las que tengan content-type con valor application/json. En caso contrario, responder con el tipo 406 (Not Acceptable).

Además, existen otras cabeceras que nos pueden servir para prevenir algunos ataques comunes, por ejemplo:

- Enviar la cabecera Content-Security-Policy: default-src 'none'. El principal objetivo es mitigar ataques XSS. Los ataques XSS se aprovechan de la confianza del navegador en el contenido que recibe del servidor. El navegador de la víctima ejecutará los scripts maliciosos porque confía en la fuente del contenido, aun cuando dicho contenido no provenga de donde se supone. Al especificar esta cabecera, solo se ejecutarán scripts de los archivos fuentes especificados en esa lista blanca de dominios, ignorando completamente cualquier otro script.
- Enviar la cabecera X-Frame-Options: deny. Esta cabecera puede usarse para evitar ataques de *clickjacking* (9), asegurándose que su contenido no es embebido en otros sitios mediante un *iframe* (10).

2.5 Ataques a los métodos de autenticación

Todos los servicios web deben implementar algún tipo de autorización. Cuando una API expone datos confidenciales y permite a los usuarios realizar acciones con información sensible, es aún más importante que se autorice cada solicitud antes del procesamiento.
En caso que algunas acciones requieran autorización y otras no, lo más seguro es rechazarlas todas de forma predeterminada y permitir solo las acciones que no requieran un permiso.

Para reducir el riesgo de vulnerabilidades en el proceso de autenticación es recomendable usar estándares. Tanto para la autenticación, como para la generación de tokens y el almacenamiento de contraseñas. Existen muchos menos riesgos al usar estándares, ya que han pasado unas estrictas pruebas de seguridad. Además, al ser de uso generalizado si en el futuro sale algún tipo de vulnerabilidad rápidamente saldrá una solución o alternativa.
Ejemplos de estándares que podemos usar son OAuth2 y JWT. Cabe destacar la importancia de evitar el uso de Basic Auth. Los principales riesgos de usar Basic Auth están relacionadas con el hecho que la contraseña es enviada y almacenada en el cliente en base64, por lo que es fácilmente descifrable si se consigue. Aunque el envío se pueda proteger usando SSL, existe la posibilidad de que sea robada al usarse en un ordenador público, reutilizada para hacer otras peticiones de forma silenciosa (*CSFR* (11)), extraída de información de los logs, etc.

Además, debemos tener en cuenta que:

- Hay que usar políticas para limitar el número de intentos (Max Retry) y funcionalidades de bloqueos (jailing) en el formulario de inicio de sesión.
  La función es la de mitigar ataques de fuerza bruta, dónde el atacante vaya probando combinaciones de contraseñas hasta adivinar la de la víctima potencial. Para tal efecto, es conveniente limitar el número de

intentos fallidos en un intervalo de tiempo. Así como bloquear los intentos futuros des de esa IP [12] u otros mecanismos durante un intervalo tiempo.

- Usar el método HTTP apropiado a cada operación: GET (lectura), POST (creación), PUT/PATCH (reemplazo/actualización), y DELETE (borrado). Hay que responder con el tipo 405 (Method Not Allowed) si el método en la petición no es apropiado para el recurso.
- No utilices información sensible (credenciales, contraseñas, tokens de usuario, o claves de API) en la URL, en su lugar usa la cabecera estándar Authorization.
- Evitar enviar información sensible cómo credenciales, contraseñas y tokens de seguridad en las respuestas.

Los vectores de ataques vistos en los puntos anteriores hacen referencia a la interacción entre el cliente y el servidor de aplicaciones. Sin embargo, es aconsejable aplicar unos métodos de seguridad generales como, por ejemplo, almacenar la información de nuestros usuarios en un lugar seguro y cifrar toda la que sea confidencial para evitar que sea descubierta.
Además, existen una serie de consideraciones sobre la implementación y despliegue de la API que, sin ir dirigidas a protegerse de un ataque concreto, ayudan a que una API sea más segura. Estas son:

- Auditar el diseño e implementación con *tests unitarios* [13]. Además de ser una forma mucho más rápida la detección de *bugs* [14], facilita enormemente el proceso de testing y reduce el número de errores que llegan al entorno de producción.
- Usar procesos de revisión de código y evitar la auto aprobación. Uno mismo al revisar su código puede pasar por alto algún error, mejor que sea otra persona la que apruebe que el código es aceptable.
- Deshabilitar el modo Debug. Aunque puede parecer obvio es aconsejable asegurarnos de que el modo debug se ha desactivado antes de poner la aplicación en producción, ya que el rendimiento puede ser menor, se puede exponer información sensible o proporcionar acceso a recursos de prueba.
- Diseñar un proceso de rollback para tus despliegues. Ante cualquier error inesperado que salga después de un despliegue es conveniente poder volver a la versión anterior mientras se investiga y corrige el error.

2.6 Herramientas para mitigar ataques a APIs

Durante los puntos anteriores han salido dos herramientas para ayudar a protegernos de ciertos ataques, WAF y API Gateway.

Un WAF (Web Application Framework) es un servidor de seguridad que ayuda a proteger una aplicación web de vulnerabilidades comunes que podrían afectar la disponibilidad de la aplicación, comprometer la seguridad o consumir excesivos recursos. Por lo general, protege las aplicaciones web de ataques como inyección de código SQL y ejecución de comandos en sitios cruzados (XSS). Además, puede filtrar solicitudes web en función de la dirección IP, el

área geográfica, el tamaño de la solicitud o en función de expresiones comunes mediante el uso de reglas. Estas condiciones se pueden introducir sobre encabezados o sobre el cuerpo de la solicitud, lo que permite crear reglas complejas para bloquear ataques provenientes de bots malintencionados o rastreadores de contenido.

Una WAF es una defensa de nivel 7 de protocolo (en el modelo OSI), y aunque no está diseñada para defenderse contra todo tipo de ataques puede ser útil ante una variedad de vectores de ataque.

Al implementar un WAF en frente de una aplicación web, se coloca un escudo entre la aplicación web e Internet pudiendo filtrar y monitorear el tráfico HTTP (Figura 8). Mientras que un servidor proxy protege la identidad de una máquina cliente utilizando un intermediario, un WAF es un tipo de proxy inverso, que protege al servidor de la exposición al hacer que los clientes pasen a través del WAF antes de llegar al servidor.

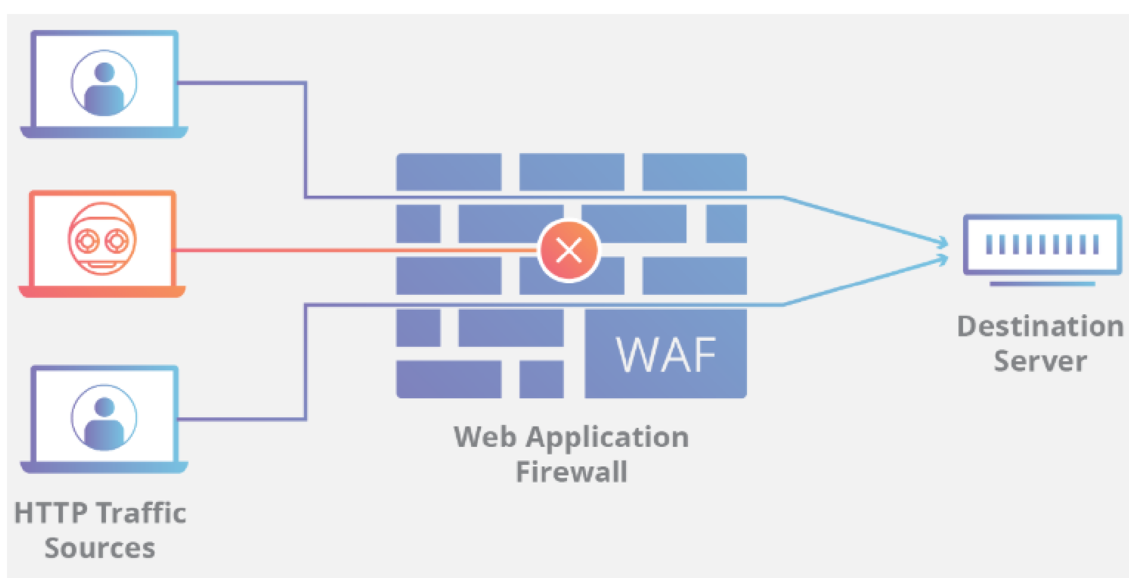


Figura 8. Web Application Framework

Un WAF opera a través de un conjunto de reglas a menudo llamadas políticas. Estas políticas tienen como objetivo proteger contra vulnerabilidades en la aplicación al filtrar el tráfico malicioso. El valor de un WAF se debe en parte a la velocidad y facilidad con que se puede implementar la modificación de políticas, lo que permite una respuesta más rápida a diferentes vectores de ataque; durante un ataque DDoS, la limitación de velocidad se puede implementar rápidamente modificando las políticas WAF.

En cuanto al concepto de API Gateway, aunque se estudiará con más detalle en el capítulo 6. API Managament System, veamos su función principal para protegernos de los ataques vistos anteriormente.

Se trata de un servidor que actúa como único punto de entrada al sistema. Además, es responsable del enrutamiento de solicitudes, la composición y la traducción de protocolos. Todas las solicitudes de los clientes pasan primero por el API Gateway. A continuación, las solicitudes se enrutan al servidor

apropiado (Figura 9). Esta característica lo hace realmente útil cuando se quiere trabajar con microservicios ya que encapsula la arquitectura interna del sistema.

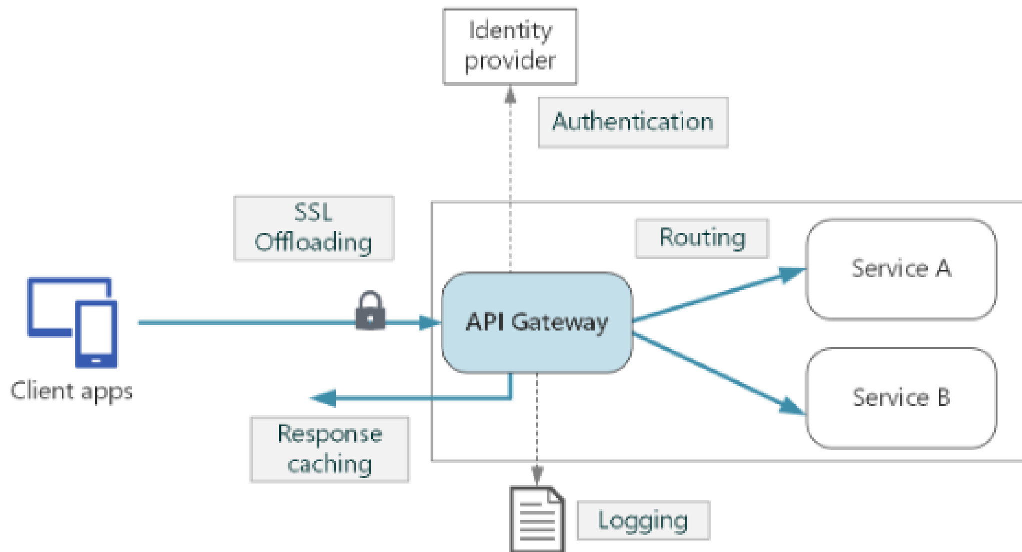


Figura 9. API Gateway

Lo relevante para este capítulo es conocer que se puede dotar la API Gateway de responsabilidades como la autorización y el control de acceso, monitoreo, balanceo de carga, almacenamiento en caché, configuración y gestión de solicitudes, y manejo de respuestas estáticas. De esta forma, se simplifica el código fuente de las API en sí, ya que estas preocupaciones están externalizadas.

Además, las API Gateway admiten varios mecanismos para controlar el acceso a la API: se puede crear políticas basadas en recursos para permitir o denegar el acceso. Así como crear planes de uso que permiten proporcionar claves de API a sus clientes y, después, realizar un seguimiento y limitar el uso de llamadas a métodos de API para cada clave de API.

3. Estándares de diseño de APIs

Crear una API es un trabajo duro. Además de largas reuniones con las partes interesadas, la elección de las tecnologías adecuadas para su creación y la creación de un modelo de distribución de datos adecuado, hay otros detalles más precisos que pueden pasarse por alto. Como por ejemplo la claridad y facilidad de uso.

Una mala experiencia de usuario mientras se consumen las API da lugar a una mala reputación y puede hacer que se deje de confiar en los servicios ofrecidos. Por lo tanto, es importante planificar como va a ser la API antes de empezar a implementarla. Aquí es donde entran en juego los estándares de diseño de API, como OpenAPI Specification y API Blueprint.

Diseñar una API significa proporcionar una interfaz efectiva que ayude a los consumidores de la API a comprenderla y usarla. Cada producto necesita un manual de uso, y una API no es una excepción.

Tener un buen diseño antes de empezar a desarrollar la API tiene las siguientes ventajas:

- Ayuda a una mejor implementación.
- Facilita el desarrollo incremental.
- Facilita una mejor documentación.
- Mejora la experiencia del desarrollador.

El diseño de la API debe definir la estructura de los recursos disponibles y la explicación de estos. Se pude considerar el diseño de una API como un contrato de cómo será el comportamiento de esta. De esta forma, al definir la API nos comprometemos a que las opciones que se especifican tienen que ser usadas tal como se ha definido y tendrán el comportamiento descrito.

El contrato define el protocolo, el formato de las peticiones y las respuestas. Así como los tipos de datos que tienen que usar los componentes de software para poder interactuar.
Se define la funcionalidad, que es independiente de las tecnologías usadas. Podría cambiar la implementación, pero la definición debe permanecer constante. El contrato ayuda a incrementar la confianza y en consecuencia el uso del componente.

3.1 OAS 3.0

La OAS (Open API Specification) tiene su origen en la especificación Swagger 2.0, que se incluyó en la Open API Initiative, creada por un consorcio de la industria para estandarizar las descripciones de las API REST, y cambió su nombre a OAS.

Actualmente, OAS es una de las formas más conocidas de diseño de API. En ella, se especifican las reglas y la sintaxis necesarias para describir la interfaz

de la API. Además, promueve un enfoque de *design first* [15], en lugar de un enfoque de implementación primero. Se diseña, en primer lugar, el contrato API (la interfaz) y luego se escribe el código que implementa el contrato.

Al momento de escribir este trabajo, nos encontramos en la tercera versión de la especificación (OAS 3.0). Desde sus inicios, OAS ha evolucionado para satisfacer las necesidades de los equipos de desarrollo de API y continúa introduciendo actualizaciones para simplificar el uso de la especificación.

Las distintas secciones de un contrato de API diseñado usando OAS 3.0 son las siguientes:

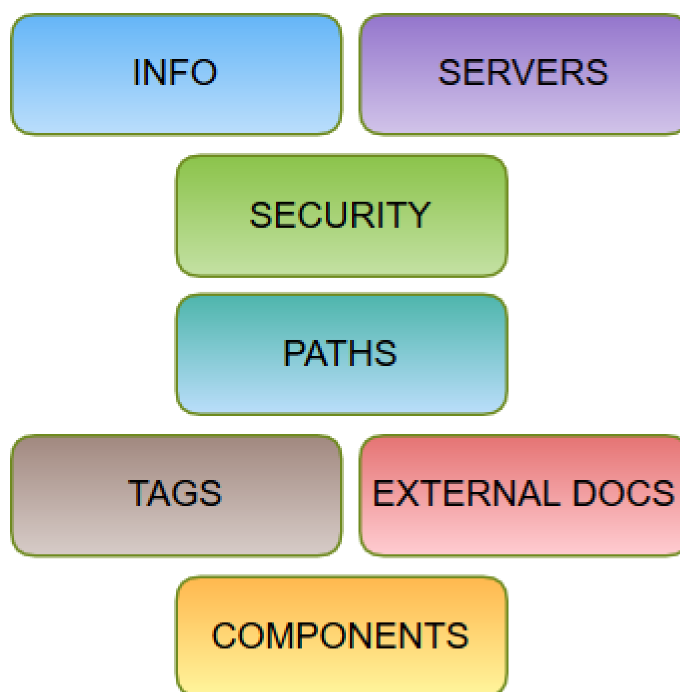


Figura 10. Secciones OAS 3.0

A continuación, se detalla la información contenida en cada sección:

- Info: Contiene los metadatos asociados con el contrato de la API. Debe dar una visión general de alto nivel de lo que hace la API.
  Las partes necesarias de esta sección son el título, la versión y la descripción de la API. Esta sección también puede tener campos como información de contacto, información de licencia y enlaces de términos de servicio.

```yaml
info:
  version: 1.0.0
  title: Swagger Petstore
  description: This is a sample Petstore server.
  license:
    name: MIT
  contact:
    email: apiteam@swagger.com
```

Figura 11. OAS 3.0 - Info

- Servers: Proporciona información sobre dónde se encuentran los servidores de la API, a través de su URL. A diferencia de la versión 2.0 de la especificación, que solo permitía que su definición de API tuviera una URL de servidor, OAS 3.0 admite múltiples servidores. Esto es útil porque en el mundo real, las API existen en múltiples entornos, y la lógica de negocio puede cambiar según el entorno.

```yaml
servers:
  - url: http://petstore.swagger.io/v1
  - url: http://petstore.swagger.io/v2
```

Figura 12. OAS 3.0 - Servers

- Security: El formato de descripción de OpenAPI es compatible con varios esquemas de autenticación y autorización para mitigar que usuarios desconocidos y no registrados accedan a la API. El OpenAPI soporta:
  - Esquemas de autenticación HTTP
  - Claves API en encabezados, cookies o parámetros
  - OAuth2
  - OpenID

  Hay dos pasos para implementar la seguridad en el diseño de su API. El primer paso es definir las implementaciones de seguridad, y el segundo es aplicarlas.

  Todos los esquemas de seguridad que se vayan a usar se tienen que definir en la sección componentes / securitySchemes.

```yaml
components:
  securitySchemes:
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: https://example.com/api/oauth/dialog
          scopes:
            write:pets: modify pets in your account
            read:pets: read your pets
```

Figura 13. OAS 3.0 - Security (Definición)

Una vez definidos de esta forma los esquemas de seguridad, se pueden aplicar a toda la API o a llamadas individuales añadiendo la sección security en la raiz o dentro de una llamada.

```yaml
paths:
  /pets/{id}:
    get:
      description: Returns a pet based on a single ID
      operationId: find pet by id
      security:
        - petstore_auth:
          - read:pets
```

Figura 14. OAS 3.0 - Security (Aplicación)

- Paths: Esta sección muestra los diversos endpoints que expone su API y los métodos HTTP correspondientes. Bajo cada método que se detalla la descipción, el tipo de entrada y salida. Así como los parámetros de entrada y las respuestas.

```yaml
paths:
  /pets:
    post:
      description: Creates a new pet in the store.  Duplicates are
        allowed
      operationId: addPet
      requestBody:
        description: Pet to add to the store
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/NewPet'
      responses:
        '200':
          description: pet response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
```

Figura 15. OAS 3.0 - Paths

- Tags: Los tags son categorías para permitir agrupar varias operaciones. Esto permite a los consumidores de la API segmentar mejor e identificar para qué quieren usar la API. Los tags se pueden agregar a cada path.

```yaml
paths:
  /pet/findByStatus:
    get:
      description: Finds pets by Status
      tags:
      - pets
```

Figura 16. OAS 3.0 - Tags (Aplicación)

Además, se puede dar una mejor descripción de lo que significa cada una de estas etiquetas agregando una sección de tags opcional en el nivel raíz de la definición de API.

```
tags:
  - name: pets
    description: Everything about your Pets
  - name: store
    description: Access to Petstore orders
```

Figura 17. OAS 3.0 - Tags (Definición)

- External Docs: OAS 3.0 permite hacer referencia a documentación externa. Su función es ofrecer información adicional para facilitar el consumo y la integración con la API.

```
externalDocs:
  description: Find out more about Swagger
  url: https://editor.swagger.io
```

Figura 18. OAS 3.0 – External Docs

- Components: A medida que se exponen más recursos y operaciones en la API, la definición puede llegar a ser realmente larga. Se puede repetir muchos parámetros existentes o descripciones de respuesta en diferentes rutas y operaciones. Por lo que tener que reescribirlos cada vez hace que sea un exceso de trabajo innecesario y los hace propensos a descripciones inconsistentes. El objeto components puede contener un conjunto de objetos reutilizables del diseño de la API. Estos objetos pueden ser esquemas, respuestas, parámetros, ejemplos, etc. Una vez definidios se referenciarán desde cualquier parte de la definición.

```
components:
  schemas:
      Pet:
        type: object
        properties:
          id:
            type: integer
            example: 65
          name:
            type: string
            example: doggo
          age:
            type: integer
            example: 4
```

Figura 19. OAS 3.0 - Components

3.2 API Blueprint

API Blueprint es un lenguaje de alto nivel para describir API web. La sintaxis es una combinación de la sintaxis de *Markdown* (16) y MSON y los archivos se guardan con una extensión "*.apib*". MSON es una extensión de Markdown para describir objetos de datos.

El objetivo de este formato es usar la filosofía de *desgin first* para las API REST, sin embargo, se puede usar este formato para documentar las API existentes.

Las secciones de las que se compone API Blueprint son:

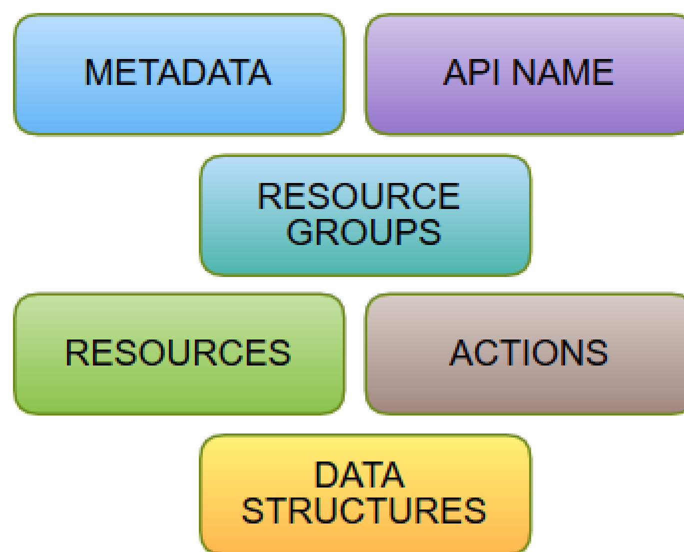


Figura 20. Secciones API Blueprint

- Metadata: Describe la versión usada de API Blueprint. También se puede indicar el servidor en el que esta alojada.

```
FORMAT: 1A
HOST: http://polls.apiblueprint.org/
```

Figura 21. API Blueprint - Metadata

- API Name: El primer encabezado sirve como nombre de la API. Los encabezados comienzan con uno o más símbolos # seguidos de un título. El número de # que se use determinará el nivel del encabezado.
  Después del encabezado hay una descripción de la API. Se pueden usar más encabezados para dividir la sección de descripción.

```
# tfm-mistic

Polls is a simple API allowing consumers to view polls and vote in them.
```

Figura 22. API Blueprint - API Name

- Resource Groups: Describe una colección de endpoints relacionados. Cada colección está compuesta de Resources. Se definen usando la palabra Group al inicio de un encabezado seguido del título de la colección.

```
# Group Questions

Resources related to questions in the API.
```

Figura 23. API Blueprint - Resource Groups

- Resources: Describe un endpoint concreto. Se especifica el nombre y el path en el que se encuentra.

```
## Questions Collection [/questions]
```

Figura 24. API Blueprint - Resources

- Actions: Describe una acción (tipo de método HTTP) sobre un endpoint. Para completar la información puede tener las siguientes partes, estas se definen como un objeto de lista en lenguaje markdown, esto es precedido por el carácter +.
    - Parameters: describe los parámetros específicos de la acción que se encuentra en la propia URL.

```
## Questions Collections [/questions{?limit}]

### List All Questions [GET]
Retrieves the list of questions.

+ Parameters
    + limit (optional, number) ... Maximum number of questions to retrieve
        + Default: `20`
```

Figura 25. API Blueprint - Actions (Parameters)

    - Atributes: describe la estructura de datos de la petición. Si se define, todas las peticiones (Request) heredaran esta estructura a menos que se especifique lo contrario:

```
+ Attributes
    + author: john@appleseed.com (string) - Author of the question
```

Figura 26. API Blueprint - Actions (Attributes)

    - Request: Contiene al menos un ejemplo de petición. Contiene la información que debe contener el body enviado al servidor:

```
+ Request (application/json)

        {
            "question": "Favourite programming language?",
            "choices": [
                "Swift",
                "Python",
                "Objective-C",
                "Ruby"
            ]
        }
```

Figura 27. API Blueprint - Actions (Request)

- Response: Contiene al menos un tipo de respuesta a la petición. Contiene el un código de estado HTTP de la respuesta. También puede contener los headers de la respuesta y el body:

```
+ Response 201 (application/json)

    + Headers

            Location: /questions/2

    + Body

            {
                "question": "Favourite programming language?",
                "published_at": "2015-08-05T08:40:51.620Z",
                "choices": [
                    {
                        "choice": "Swift",
                        "votes": 0
                    }, {
                        "choice": "Python",
                        "votes": 0
                    }, {
                        "choice": "Objective-C",
                        "votes": 0
                    }, {
                        "choice": "Ruby",
                        "votes": 0
                    }
                ]
            }
```

Figura 28. API Blueprint - Actions (Response)

- Data Structures: Describe el formato de datos de objetos usadas para peticiones y respuestas de la API. Definiéndolos en esta sección, separada de Actions, se pueden reutilizar fácilmente, referenciándolos en varias acciones.

```
# Data Structures

## Message (object)

+ text (string) - text of the message
+ author (Author) - author of the message

## Author (object)

+ name: John
+ email: john@appleseed.com
```

Figura 29. API Blueprint - Data Structures

Del mismo modo, que hemos visto con OAS 3.0, API Blueprint, establece un contrato. En este, se puede observar el comportamiento esperado de la API para cada situación, documentando todas las opciones disponibles para el desarrollador y facilitando la tarea de realizar una implementación que haga uso de la API.

Ambos formatos son de código abierto y tienen una gran comunidad y herramientas en torno a ellos, con amplias opciones de extensiones e integraciones.
Un hecho distintivo es el formato usado, mientras OAS usa el formato YAML (17) o JSON, API Blueprint usa una sintaxis similar a Markdown, MSON.

Además, de estos existen otros estándares de definición de API como RAML, todos ellos nos permiten aplicar el concepto de design first y nos ayudan a tener una API bien documentada, más fácil de implementar y de consumir.
La elección de un estándar u otro depende de nuestras preferencias y del tipo de proyecto. Los dos estudiados en este apartado son los más usados en la industria actualmente y, por tanto, los más recomendados.

4. OAuth 2.0

Ya hemos visto la importancia de esforzarnos para que una API sea segura. La estrategia de securización de APIs debe encontrar el equilibrio entre la seguridad y la usabilidad. Una estrategia demasiado compleja disuade a usuarios malintencionados, pero también obstaculiza su uso por parte de usuarios legítimos. Como hemos comentado, el uso de estándares ayuda a conseguir ese equilibrio. No solo el acceso a las APIs necesita control, también encontramos el problema de la delegación de la autorización para acceder a los datos. OAuth es un estándar que surge como respuesta a esas necesidades. Permite a las aplicaciones otorgar un acceso limitado (scopes) a los datos de los usuarios, sin tener que proporcionar las credenciales de dicho usuario, desacoplando la autenticación y la autorización de los datos.

OAuth 2.0 es un protocolo de seguridad que se utiliza para proteger una gran cantidad de APIs web en todo el mundo, desde proveedores a gran escala como Facebook y Google hasta pequeñas API en nuevas empresas. Se utiliza para conectar sitios web entre sí y potencia las aplicaciones nativas y móviles que se conectan a servicios en la nube. OAuth2 es el método de seguridad dominante en la web hoy en día, y su ubicuidad ha nivelado el campo de juego para los desarrolladores que desean proteger sus aplicaciones.

Se trata de un protocolo de delegación, un medio para permitir que alguien que controla un recurso permita que una aplicación de software acceda a ese recurso en su nombre sin necesidad de hacerse pasar por él. Para tal efecto, la aplicación solicita la autorización del propietario del recurso y recibe tokens que puede usar para acceder al recurso. Todo esto sucede sin que la aplicación tenga que hacerse pasar por la persona que controla el recurso, ya que el token representa explícitamente un derecho de acceso delegado.
Además, los tokens de OAuth pueden limitar el acceso de la aplicación solo a las acciones que el propietario del recurso delegue.

Por ejemplo, un usuario que tiene un servicio de almacenamiento de fotografías en la nube y un servicio de impresión de fotografías desea poder imprimir las fotografías que ha almacenado en su servicio de almacenamiento. Afortunadamente, su servicio de impresión en la nube puede comunicarse con su servicio de almacenamiento en la nube utilizando una API y acceder a las fotografías. Sin embargo, los dos servicios son administrados por compañías diferentes, lo que significa que su cuenta con el servicio de almacenamiento no tiene conexión a su cuenta con el servicio de impresión. En casos como este, OAuth es una buena opción para resolver el problema, pudiendo delegar el acceso a sus fotos a través del servicio de impresión de fotos, todo sin dar su contraseña.

Esta casuística se presenta en la definición de [4] (Internet Engineering Task Force (IETF)), que es la siguiente:
El marco de autorización OAuth 2.0 permite que una aplicación de terceros obtenga acceso limitado a un servicio HTTP. Este acceso puede darse de dos formas: en nombre del propietario del recurso al organizar una interacción de

aprobación entre el propietario del recurso y el servicio HTTP, o al permitir que la aplicación de terceros obtenga acceso por ella misma.

En el proceso de autenticación usando OAuth2 intervienen cuatro componentes, son los siguientes:

- Propietario del recurso: el propietario del recurso tiene acceso a una API y puede delegar el acceso a esa API. Es el encargado de definir qué aplicaciones pueden llamar a la API, qué usuarios pueden acceder y qué acciones pueden realizar. El propietario del recurso suele ser una persona y generalmente se asume que tiene acceso a un navegador web.
- Recurso protegido: El recurso protegido es el componente al que tiene acceso el propietario del recurso. Este puede ser de muchas formas diferentes, pero en la mayoría de los casos se tratará de una API web. A pesar de que el nombre "recurso" hace que parezca que es algo para descargar, estas API pueden permitir la lectura, la escritura, así como otras operaciones.
- Cliente: es la aplicación o componente de software que accede al recurso protegido en nombre del propietario del recurso. Es el encargado de realizar las peticiones a la API que constituye el recurso protegido.
- Servidor de autenticación OAuth2: es un servidor HTTP que actúa como componente central. Es el encargado de autenticar el propietario del recurso y el cliente. Proporciona mecanismos para permitir a los propietarios del recurso autorizar clientes para que actúen en su nombre.

Los cuatro componentes interactúan entre sí para autenticar el usuario. El usuario presenta sus credenciales ante el servidor de autenticación. Si se autentica correctamente, se le da un token que debe enviarse con cada solicitud a los diferentes servicios a los que ha dado permisos. Lo habitual es que los tokens tengan una caducidad, por eso existe la posibilidad de validar el token con el servidor de autenticación y recuperar los roles que un usuario les ha asignado.
La gran utilidad de OAuth 2.0 es que permite al usuario autenticarse ante diferentes servicios sin tener que presentar constantemente las credenciales.

En el ejemplo de impresión, imaginemos que un usuario ha subido sus fotos de vacaciones al sitio de almacenamiento de fotos y ahora desea imprimirlas. La API del sitio de almacenamiento es el recurso, y el servicio de impresión es el cliente de esa API. El usuario, como propietario del recurso, debe poder delegar parte de su autoridad a la impresora para que pueda leer sus fotos. Es probable que no desee que la impresora pueda leer todas sus fotos, ni tampoco que la impresora pueda eliminar fotos o cargar otras nuevas por su cuenta. Toda esta casuística puede resolverse usando OAuth2.

Otro ejemplo, seria el de aplicaciones pensadas para guardar un registro de actividad física. En la

Figura 30 mostramos el ejemplo de conexión de PolarFlow (Cliente), que al sincronizarse una actividad física con el reloj Polar asociado a la cuenta, transferirá los detalles de la actividad como duración, recorrido realizado,

pulsaciones y ritmos al servicio Strava (recurso protegido) para que sea compartido con los amigos en ese servicio.

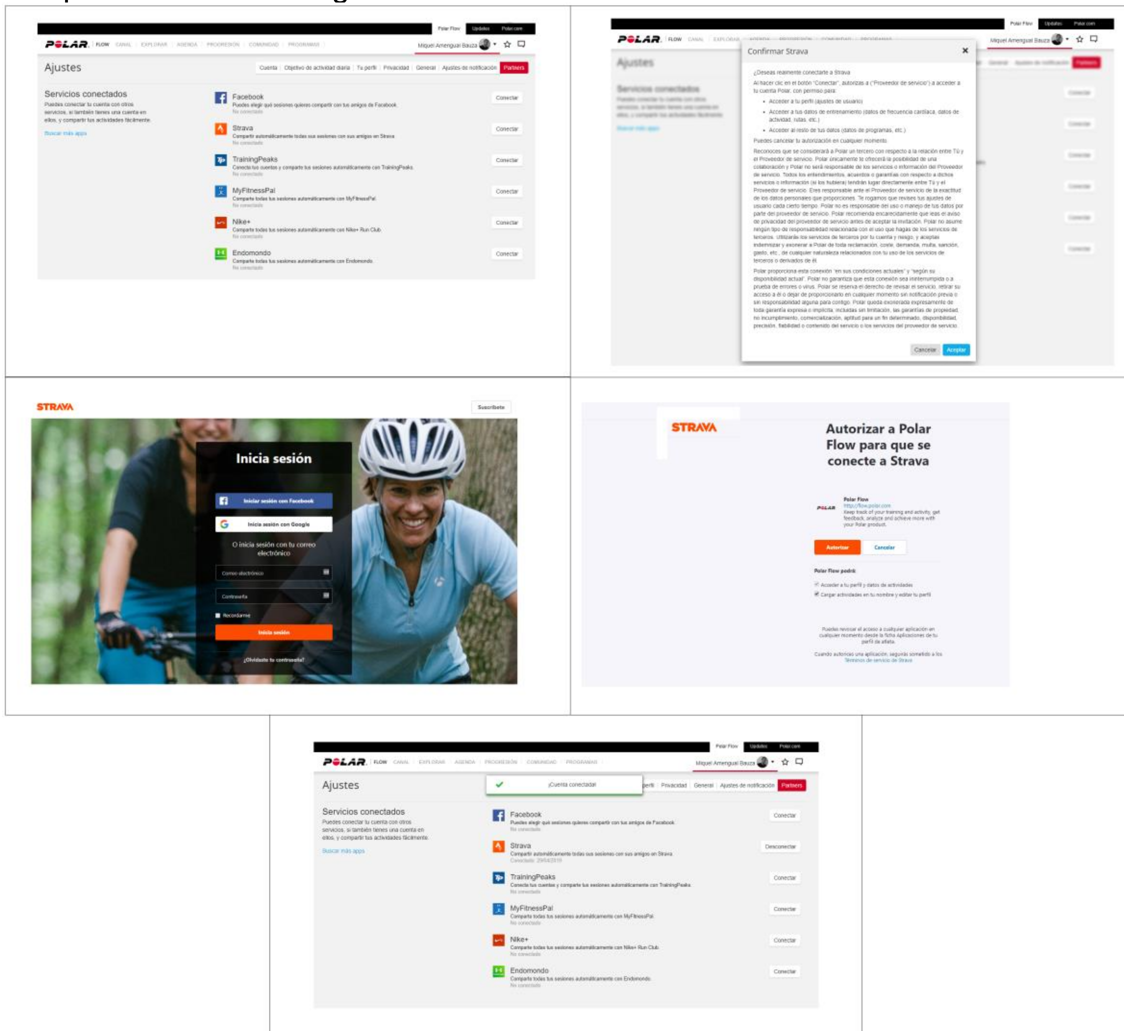


Figura 30. Ejemplo de uso de OAuth2 PolarFlow - Strava

Podemos ver que a nivel de usuario se compone de 5 fases, elegir el servicio al que queremos delegar acceso para que la aplicación pueda interactuar con él. En segundo lugar, nos expone cómo será la comunicación y los datos que proporcionará al servicio Strava. A continuación, nos redirige a la página de Strava para que nos autentiquemos. Una vez introducidas las credenciales, Strava nos pregunta si queremos delegar acceso a Polar para que acceda a los datos de actividades y cargue actividades en nuestro nombre.
Una vez autorizado, el proceso termina redirigiéndonos de nuevo a la aplicación de PolarFlow.

El proceso completo, consta de algunas partes que no son visibles por el usuario. Podemos observarlos en la Figura 31, la cual hemos extraído de [8] (Richer 2017):

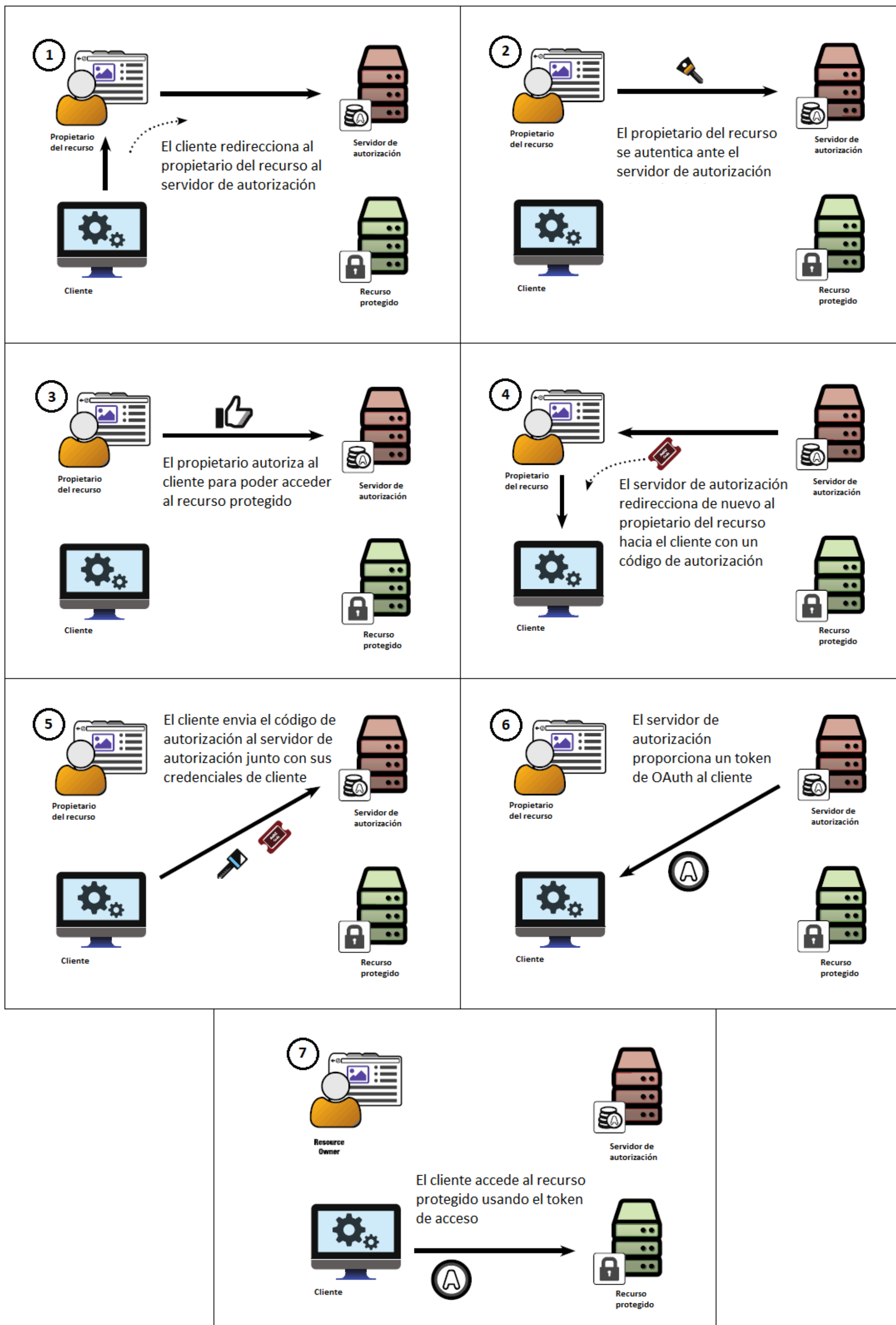


Figura 31. Flujo proceso autenticación OAuth2

Podemos ver los 7 pasos que autorizan al cliente a poder acceder al recurso protegido:

1. El cliente redirecciona al propietario del recurso al servidor de autorización para que se autentique.

2. El propietario del recurso se autentica introduciendo sus credenciales del recurso protegido.
3. El propietario del recurso autoriza al cliente para poder realizar acciones sobre el recurso protegido.
4. El servidor de autorización redirecciona de nuevo al propietario del recurso hacia el cliente, con un código de autorización.
5. El cliente envía el código de autorización al servidor de autorización junto con las credenciales de aplicación.
6. El servidor de autorización proporciona un token OAuth al cliente para que actúe en nombre del propietario del recurso.
7. El cliente puede realizar las acciones por las que se ha autorizado haciendo uso del token.

En resumen, OAuth2 es un buen protocolo, aunque está lejos de ser perfecto. Como ocurre con todas las cosas en tecnología, es posible que lo veamos reemplazado por otro en el futuro. Sin embargo, aún no ha surgido ningún competidor real en el momento de escritura de este trabajo. Es igual de probable que el reemplazo de OAuth2 termine siendo una extensión de OAuth2 en sí.

Sin embargo, tiene sus límites. OAuth2 no es un protocolo de autenticación, aunque se puede usar para crear uno, una transacción de OAuth por sí sola no le dice nada sobre quién es el usuario. Como hemos visto en el ejemplo de impresión fotográfica: la impresora fotográfica no necesita saber quién es el usuario, solo que alguien dijo que estaba bien descargar algunas fotos.
Aunque OAuth2 utiliza la autenticación en varios lugares, particularmente la autenticación del propietario del recurso y el software del cliente al servidor de autorización. Esta autenticación integrada no hace que OAuth2 sea un protocolo de autenticación.
También cabe remarcar que OAuth2 no está definido fuera del protocolo HTTP. Y dado que OAuth2 trabaja con tokens secretos y no proporciona firmas de mensajes, nunca debe utilizarse fuera de HTTPS.

4.1. OpenID Connect

A todos nos resulta familiar que al acceder a una aplicación aparezca la opción de acceder usando Facebook, Twitter, Google, Amazon, etc. La posibilidad de autenticarnos mediante una aplicación de terceros se puede realizar gracias a una extensión de OAuth2 realizada por ellos mismos para proveer mecanismos de autenticación. En estos casos se siguen los mismos conceptos que en el protocolo OIDC (OpenID Connect) desarrollado por la OpenID Foundation.

Tal como se dice en su web [5] (OpenID), podemos definir OpenID como una simple capa de identificación sobre el protocolo OAuth 2.0. Permite a los clientes verificar la identidad del usuario final en función de la autenticación realizada por un servidor de autorización, así como obtener información básica del perfil del usuario final en una forma interoperable y similar a REST.

El propósito de OIDC es tener un mismo inicio de sesión para múltiples sitios. Cada vez que necesita iniciar sesión en un sitio web mediante OIDC, se le redirige a su sitio de OpenID en el que inicia sesión, y luego se lo devuelve al sitio web.

Para ilustrar el funcionamiento de OIDC, veamos un ejemplo de los pasos que se realizan cuando le damos a iniciar sesión con Google:

1. Cuando se elige iniciar sesión con su cuenta de Google, se envía una solicitud de autorización a Google.
2. Google autentica sus credenciales o le solicita que inicie sesión si aún no lo ha hecho, y solicita su autorización (enumera todos los permisos que la aplicación desea, por ejemplo, leer su dirección de correo electrónico y le pregunta si está de acuerdo).
3. Una vez que se autentica y autoriza el inicio de sesión, Google envía de vuelta un access token y, si se solicita, un ID token.
4. La aplicación puede recuperar información del usuario (como su identificador único de Google o su correo electrónico) desde el ID token o usar el access token para invocar una API de Google.

Hemos hablado de dos tipos de tokens, los access token y los ID token:

- Los access token son credenciales que una aplicación puede usar para acceder a la API del proveedor. Su propósito es informar a la API que al portador de este token se le ha otorgado acceso delegado a la API y solicitar acciones específicas (según lo especificado por los ámbitos que se han otorgado).
- Los ID token son token web JSON (JWT) que contienen datos de identidad. Son consumidos por la aplicación y se usan para obtener información del usuario, como el nombre del usuario, el correo electrónico, etc.

5. Arquitectura de microservicios

Antes de que evolucionara el concepto de microservicios, la mayoría de aplicaciones web fueron desarrolladas utilizando un estilo arquitectónico monolítico. En una arquitectura monolítica, una aplicación se entrega como un único paquete de software desplegable.
Toda la interfaz de usuario, la lógica de negocio y el acceso a la base de datos se empaquetan juntas en un único artefacto de aplicación que es desplegado en un servidor de aplicaciones.

Si bien una aplicación puede implementarse como una sola unidad de trabajo, la mayoría de las veces habrá múltiples equipos de desarrollo trabajando en la aplicación. Cada equipo de desarrollo tendrá sus propias piezas discretas de la aplicación de las que es responsable y, a menudo, los clientes específicos que están sirviendo con su pieza funcional.

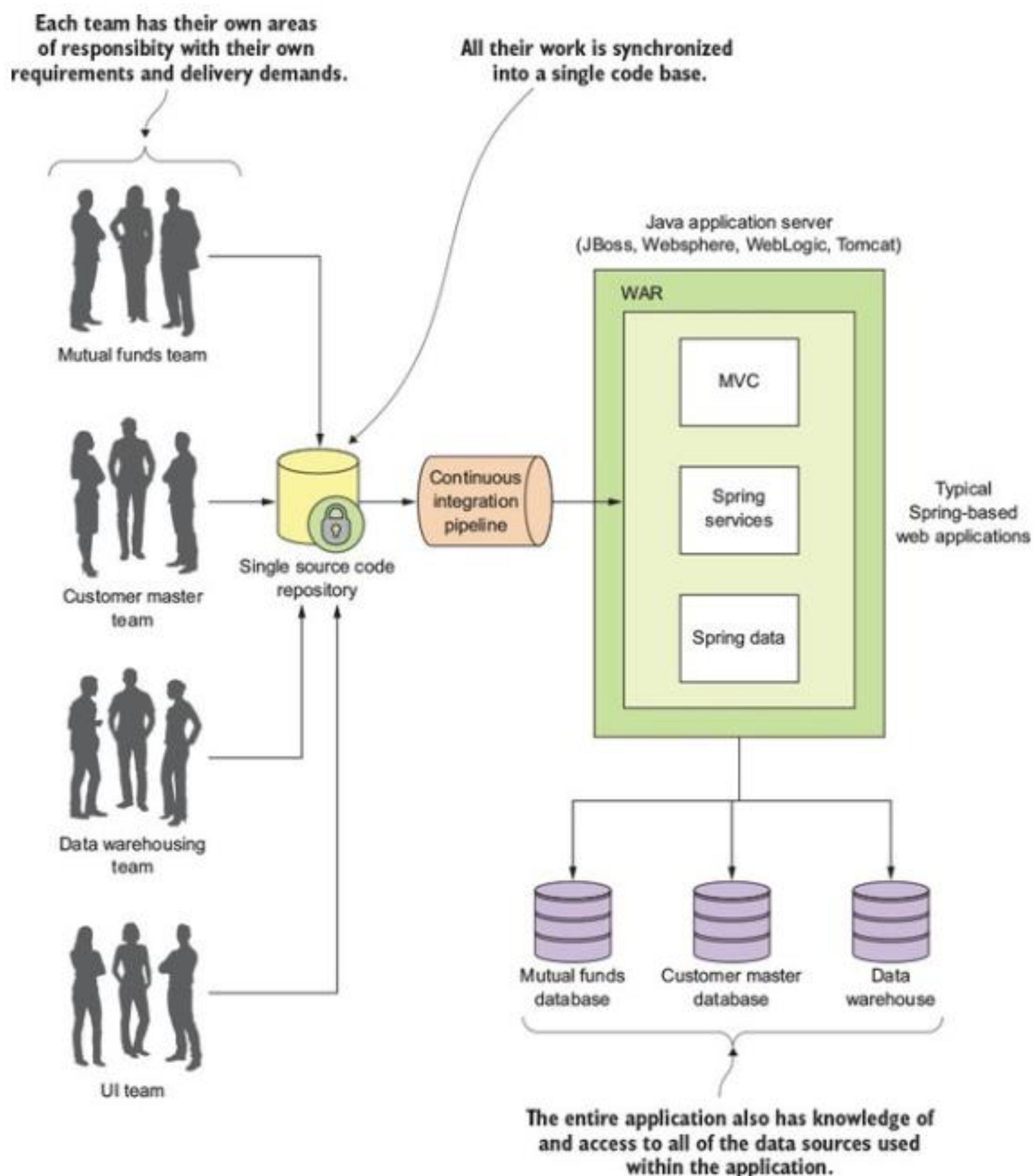


Figura 32. Arquitectura aplicación CRM monolítica

En la Figura 32, de [2] (Carnell 2017) podemos ver un ejemplo de una gran empresa de servicios financieros. Se trata de una aplicación interna de gestión de relaciones con el cliente (CRM) que implica la coordinación de varios equipos: el de la interfaz de usuario, el de gestión de clientes, el de análisis de datos y el de financiación.

De esta forma, se fuerza a que los diferentes equipos de desarrollo tengan que sincronizar su despligue ya que su código necesita ser desplegado y probado como una unidad completa.

El problema principal es que a medida que aumenta el tamaño y la complejidad de la aplicación *CRM* [(17)] monolítica, los costos de comunicación y coordinación de los equipos individuales que trabajan en la aplicación aumentan. Cada vez que un equipo individual necesite hacer un cambio, la aplicación completa se tiene que reconstruir, volver a probar y volver a desplegar.

El concepto de microservicio se introdujo originalmente en la conciencia de la comunidad de desarrollo de software alrededor de 2014 y fue una respuesta directa a muchos de los desafíos al tratar de escalar aplicaciones monolíticas. Tanto a nivel técnico como de organización. Un microservicio es un servicio pequeño, distribuido de manera flexible. Los microservicios permiten tomar una aplicación grande y descomponerla en componentes fáciles de administrar con responsabilidades definidas de manera limitada.

Los microservicios ayudan a combatir los problemas tradicionales de complejidad en el código de un aplicación grande al descomponer el código en piezas pequeñas y bien definidas. El concepto clave que se debe adoptar al pensar en los microservicios es descomponer y desagregar la funcionalidad de las aplicaciones para que sean completamente independientes entre sí.

Si tomamos la aplicación CRM que vimos en la Figura 32 y la descomponemos en microservicios, podría parecerse a lo que se muestra en la Figura 33, también extraída de [2] (Carnell 2017).

Usando una arquitectura de microservicio, nuestra aplicación de CRM se descompondría en un conjunto de microservicios completamente independientes entre sí, lo que permitiría a cada equipo de desarrollo moverse a su propio ritmo.

Se puede ver que cada equipo funcional tiene su código de servicio y su infraestructura de servicio independiente. Un equipo puede construir, implementar y probar sin necesidad de coordinarse con otros equipos, ya que su código y la infraestructura (servidor de aplicaciones y base de datos) ahora son completamente independientes de las otras partes de la aplicación.

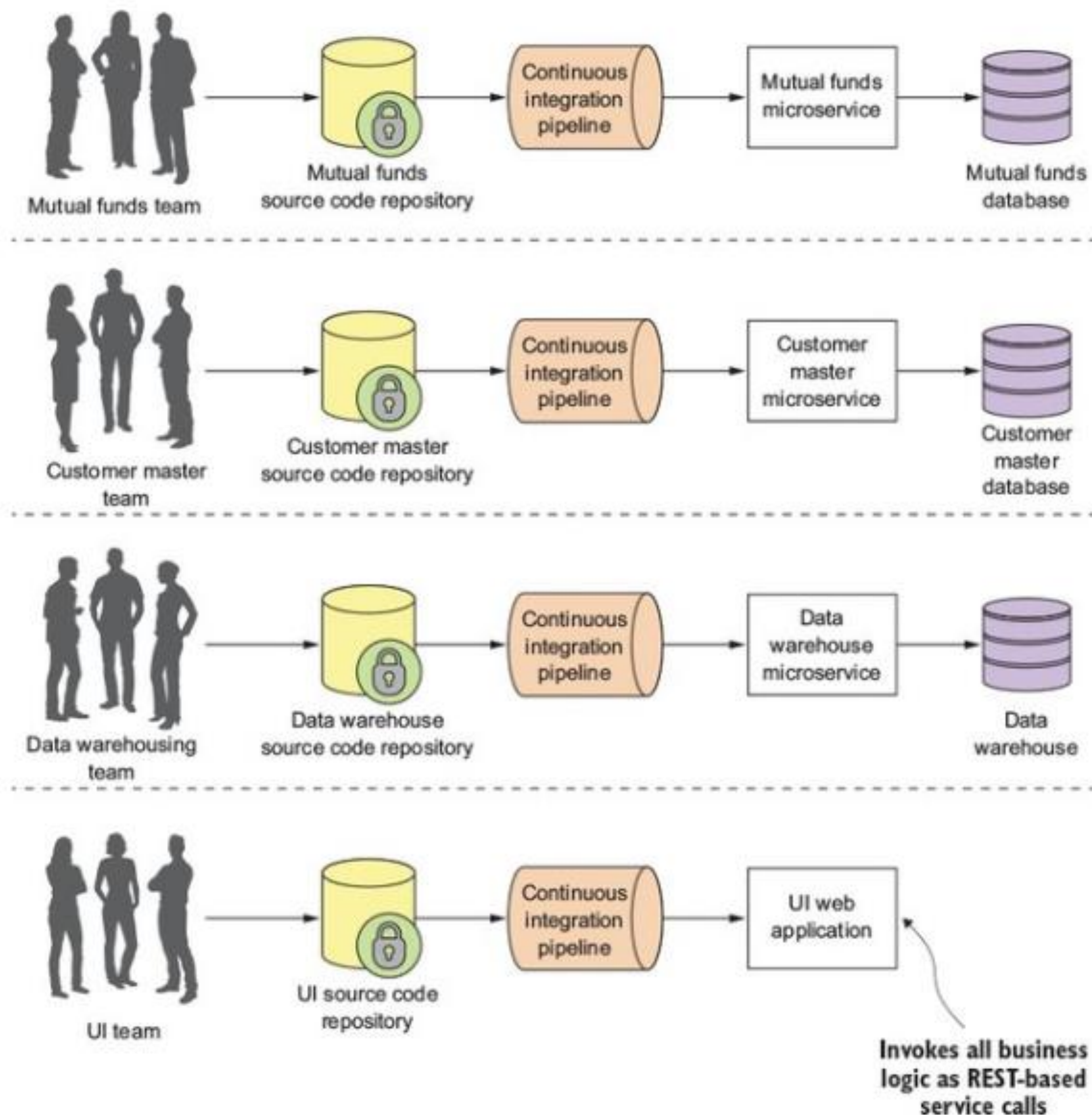


Figura 33. Arquitectura aplicación CRM con microservicios

Una arquitectura de microservicio tiene las siguientes características:

- La lógica de la aplicación se divide en componentes pequeños con límites de responsabilidad bien definidos que se coordinan para ofrecer una solución.
- Cada componente tiene un pequeño dominio de responsabilidad y se implementa de forma completamente independiente entre sí. Los microservicios deben ser responsables de una sola parte de un dominio empresarial. Además, un microservicio debe ser reutilizable en múltiples aplicaciones.
- Los microservicios se comunican según algunos principios básicos y emplean protocolos de comunicación ligeros como HTTP y JSON para intercambiar datos entre el consumidor de servicios y el proveedor de servicios.

- La implementación técnica subyacente del servicio es irrelevante porque las aplicaciones siempre se comunican con un protocolo de tecnología neutral (JSON es el más común). Esto significa que una aplicación creada utilizando una aplicación de microservicio podría construirse con múltiples lenguajes y tecnologías.
- Los microservicios, por su naturaleza pequeña, independiente y distribuida, permiten a las organizaciones tener pequeños equipos de desarrollo con áreas de responsabilidad bien definidas. Es posible que estos equipos trabajen hacia un solo objetivo, como entregar una aplicación, pero cada equipo es responsable solo de los servicios en los que está trabajando.

Como cualquier tecnología, la arquitectura de microservicios tiene ventajas e inconvenientes, para saber si es adecuado hay que tener en cuenta los requisitos de la aplicación.
Las ventajas que presenta el uso de microservicios son:

- Ofrece a los desarrolladores la libertad de desarrollar y desplegar servicios de forma independiente.
- Un microservicio puede ser desarrollado por un equipo bastante pequeño.
- El código para diferentes servicios se puede escribir en diferentes lenguajes.
- Fácil integración y despliegue al ser piezas pequeñas independientes.
- Fácil de entender y modificar para los desarrolladores, puede ayudar a un nuevo miembro del equipo a ser productivo más rápidamente.
- Cuando se requiere un cambio en una parte determinada de la aplicación, solo el servicio relacionado se tiene que modificar y volver a desplegar, sin necesidad de modificar y volver a desplegar toda la aplicación.
- Mejor aislamiento de errores: si un microservicio falla, el otro continuará funcionando. En cambio, un área problemática de una aplicación monolítica puede poner en peligro todo el sistema.
- Fácil de escalar e integrar con servicios de terceros.
- Sin compromiso a largo plazo con la tecnología usada, ya que se puede cambiar solo del microservicio que nos interese sin afectar al resto.

En cuento a las desventajas tenemos que:

- Debido a la implementación distribuida, las pruebas pueden ser complicadas y tediosas.
- Un número creciente de microservicios puede hacer que los desarrolladores no tengan información suficiente de la arquitectura global.
- La arquitectura trae una complejidad adicional, ya que los desarrolladores tienen que mitigar la tolerancia a errores y la latencia de la red.
- Al ser un sistema distribuido, puede resultar en una duplicidad de esfuerzos.
- Cuando aumenta el número de servicios, la integración y la gestión de productos completos pueden complicarse.
- Los desarrolladores tienen que lidiar con la complejidad adicional de un sistema distribuido. Es necesario implementar un mecanismo de comunicación entre los servicios

- El manejo de casos de uso que abarcan más de un servicio requiere comunicación y cooperación entre diferentes equipos.

En resumen, usadas de forma correcta, las arquitecturas basadas en microservicios permiten construir y escalar su aplicación a medida que aumenta la complejidad y el número de desarrolladores que trabajan en la aplicación. Sin embargo, es importante realizar un seguimiento cada vez que se agregue un nuevo microservicio al sistema o se realice una nueva conexión entre microservicios para ver que vamos por el buen camino. Hay que examinar el aumento de complejidad global y asegurarse de que está justificado.

6. API Managament System

Las empresas que exponen ciertos activos de datos o funciones definidas expresamente para su consumo a través de APIs necesitan que estas sean fácilmente accesibles y bien documentadas. Así como informar de los cambios y nuevas versiones.

Además, surge el concepto de API Economy, permitiendo a las empresas impulsar su crecimiento con el consumo y la monetización de las APIs. Dado que pueden obtener beneficios mediante la exposición de APIs de servicios que son valiosos para terceros y están dispuestos a pagar por su uso.

Con el fin de cubrir estas necesidades aparece el término API Managament System. Este proceso engloba las tareas de publicar, promocionar y supervisar APIs en un entorno seguro y escalable. Asimismo, incluye todos aquellos recursos enfocados a la creación, documentación y distribución de las APIs.

En general, un API Managament System está conformado de los siguientes componentes:

- API Gateway: componente cuya principal función es la de habilitar la interconexión entre los servicios y los consumidores, a través de las APIs publicadas en él.
- API Manager: componente cuya principal responsabilidad es la de ofrecer a los proveedores capacidades de alta configuración y publicación de sus APIs en el componente API Gateway. Es donde se incluye toda la gestión de claves y planes de uso para monetizar la API.
- Portal de APIs: componente dedicado a recopilar toda la información necesaria para los consumidores sobre las APIs publicadas en el API Gateway.

Podemos ver como se relacionan estos componentes en la siguiente figura:

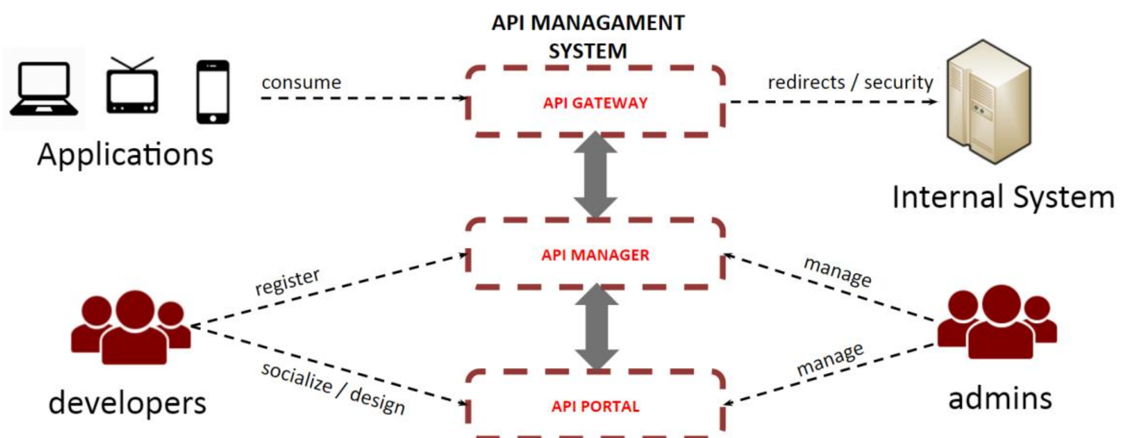


Ilustración 34. API Management System

A continuación, analizaremos los elementos que tiene cada componente:

Los elementos que componen un API Portal son:

- Comunidad de desarrollo: contiene publicaciones de noticias y comentarios referentes al uso de las APIs. Así como posibles configuraciones, errores comunes y soluciones adoptadas.
- Navegador interno: buscador de APIs registradas en el sistema, con varios filtros de consulta como estado, versión, mejor valoración, etc.
- Tienda: dónde se localizan las API publicadas, accesos directos a las comunidades de consumidores, herramientas de testing, monitorización, recomendaciones de usuarios, etc.
- Sistema de pruebas: sistema integrado de testeo de cada API.
- Documentación: repositorio de documentación referente a las APIs publicadas.
- Estadísticas de uso: sistemas de monitorización y análisis desde la perspectiva del consumidor: timing, status, etc.

En cuanto a los API Manager, están formados por:

- Gestor de publicación: herramienta para la publicación las APIs en el componente API Gateway definiendo su ruta de acceso.
- Gestor de edición: herramienta para el diseño tanto de la interfaz de la API como de la documentación.
- Gestor de ciclo de vida: permite gestionar los diferentes estados por lo que pasa una API, así como su versionado.
- Gestor de políticas de uso: herramienta para la configuración de reglas de uso tales como pagos por uso y gestión de las claves.
- Gestor de consumo: monitorización del uso de las APIs y sistema de configuración de alertas según los parámetros de consumo.
- Gestor de políticas de seguridad: gestiona todas las configuraciones de seguridad de una API.

Finalmente, los API Gateway se componen de:

- Políticas de routing: enrutamiento de mensajes a diferentes destinos dependiendo del contexto o del contenido del mensaje.
- Soporte muti-protocolo: protocolos soportados tanto para la publicación de APIs en el componente Gateway como para el enrutamiento a los servicios internos.
- Soporte multi-formato: componentes destinados a transformar los datos de un formato a otro, o de su enmascaramiento.
- Monitoring: Monitorización del tráfico de entrada y salida.
- Políticas de seguridad: otorga a las API características de autenticación, autorización y cifrado utilizando estándares o tecnologías conocidas como el cifrado de transporte mediante HTTPS o el estándar de autorización OAuth para interfaces REST.
- Políticas de uso: capacita a las APIs para gestionar políticas de consumo, rendimiento, fallos, etc.

Enlazando con el capítulo anterior, un API Gateway es una pieza fundamental para abstraer a la aplicación en frente de numerosos microservicios. De esta forma, solo tiene que conocer un solo recurso (URL del API Gateway). Este será el encargado de redirigir al microservicio correspondiente dependiendo de

la ruta, se puede ver esta idea en la Figura 35, extraída de [7] (Richardson 2015):

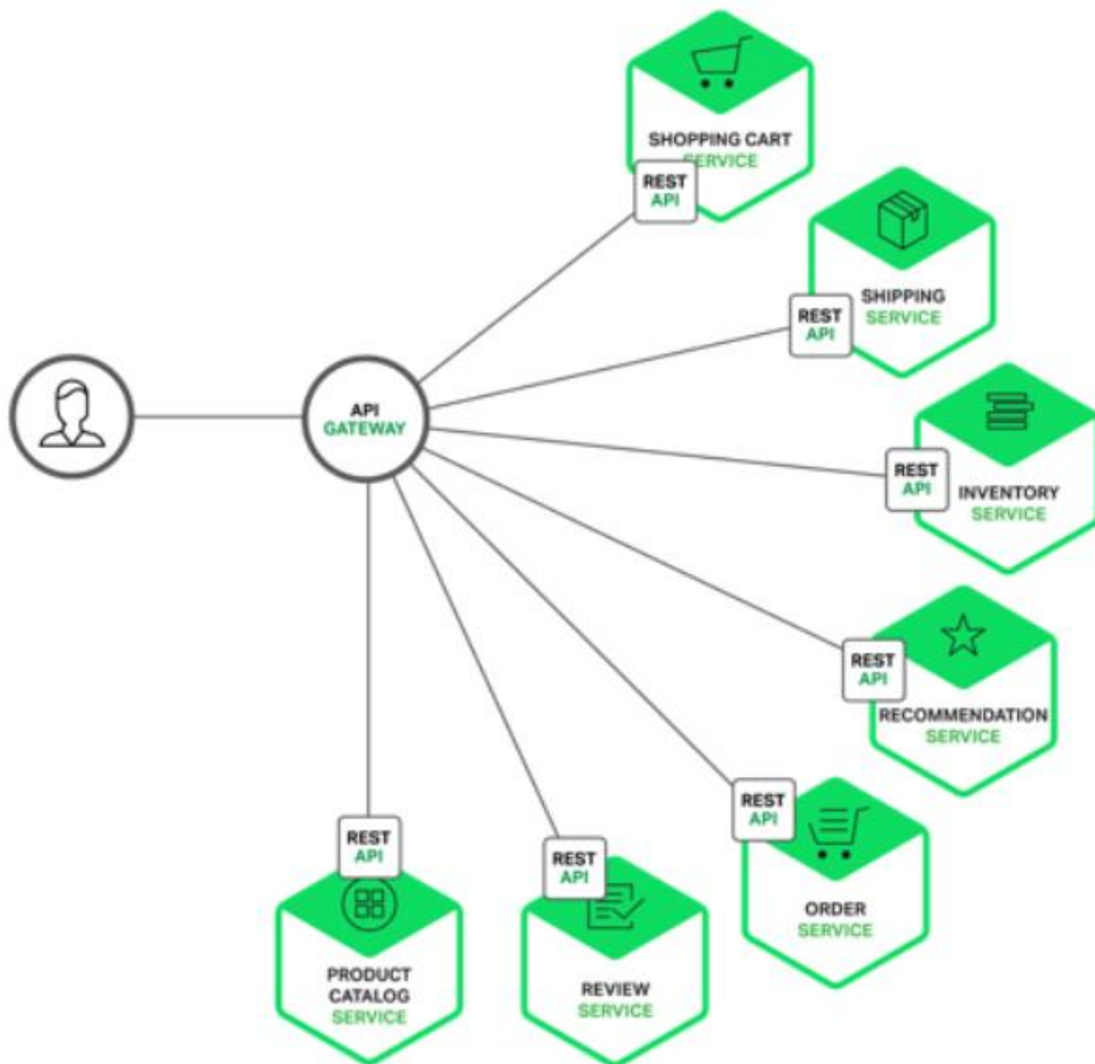


Figura 35. Microservicios con API Gateway

Usando una arquitectura como la anterior, la aplicación no necesita conocer la ubicación final del recurso al cual accede, por lo que se puede cambiar aplicaciones de servidores, separar un servicio en dos diferentes o agruparlos. También facilita la tarea del desarrollador, ya que solo tiene que mantener la API Gateway para que conozca las ubicaciones de los diferentes servicios.

7. Caso de estudio

Se quiere realizar una aplicación para poder gestionar el inventario de productos que se tienen en casa y así controlar lo que se necesita cuando decidimos ir a comprar.

Para ello, se aplicarán los conceptos y tecnologías vistas en los puntos anteriores. Vamos a optar por una arquitectura de microservicios, lo que posibilitará que los diferentes servicios puedan crecer de forma independiente e incluso que se encarguen de ellos diferentes personas.

Los tres microservicios que van a desarrollarse serán:

- ProductAPI: Para gestionar los productos que usamos habitualmente, se puede guardar un nombre, una descripción y un precio. En el futuro se podría añadir una foto, el código de barras y tiendas dónde lo podemos adquirir.
- InventoryAPI: Para agrupar los productos en diferentes categorías y definir el intervalo de cantidad deseado de cada producto. Además, se puede ir cambiando la cantidad que se tiene.
- ListAPI: Para crear listas de los productos y la cantidad que se necesita de cada uno. En el futuro se podría añadir el precio total que nos va a costar, descartar productos, así como marcar los productos que han sido comprados.

Para describir la funcionalidad usaremos el estándar OAS 3.0. En los anexos se pueden consultar estos documentos para ver todas las funciones que se permiten. Además, adjuntamos un resumen visual generado con https://editor.swagger.io/ a partir de dicha documentación. Los podemos ver en las siguientes figuras: Figura 36, Figura 37 y Figura 38.

Figura 36. OAS3 - Simple Product API

Simple Inventory API 1.0.0 OAS3

A sample API that allow perform basic operations onto inventories

Contact Miguel Amengual Bauza

Apache 2.0

Servers

https://glj3mba16a.execute-api.eu-west-1.amazonaws.com/api

Authorize

default

GET /inventory

POST /inventory

GET /inventory/{inventoryName}

DELETE /inventory/{inventoryName}

GET /inventory/{inventoryName}/product

POST /inventory/{inventoryName}/product

PUT /inventory/{inventoryName}/product/{productName}

DELETE /inventory/{inventoryName}/product/{productName}

Figura 37. OAS3 - Simple Inventory API

Simple List API 1.0.0 OAS3

A sample API that allow perform basic operations onto lists

Contact Miguel Amengual Bauza

Apache 2.0

Servers

https://glj3mba16a.execute-api.eu-west-1.amazonaws.com/api

Authorize

default

GET /list

POST /list

GET /list/{name}

PUT /list/{name}

DELETE /list/{name}

Figura 38. OAS3 - Simple List API

En cuanto a la implementación, las APIs se desarrollarán en lenguaje JavaScript (Node.js) y se desplegarán en un entorno serverless, en concreto AWS Lambda. El concepto serveless hace referencia a que el desarrollador no

se tiene que preocupar sobre el mantenimiento del servidor donde corre la aplicación, el proveedor se encarga de ejecutar un fragmento de código mediante la asignación dinámica de los recursos.
Los datos que se necesiten almacenar se guardaran en el servicio de base de datos NoSQL (18) de Amazon, AWS DynamoDB.

Para la autenticación de los usuarios se usará AWS Cognito. Se trata de un servicio se utiliza para crear identidades únicas para los usuarios, autenticarlas con proveedores de autenticación y guardar los datos de usuario en la nube de AWS. Los usuarios pueden iniciar sesión directamente con un nombre de usuario y una contraseña o a través de un tercero como Facebook, Amazon o Google. Tras ello, se nos otorga un token, que se incluirá en todas las llamadas.

Para abstraer a la aplicación de los diferentes microservicios y para poder implementar medidas de seguridad en un punto común se usará un API Gateway (AWS API Gateway). Se redirigirá la petición al servicio adecuado dependiendo de la ruta. Además, todas las llamadas necesitarán incluir un token de usuario en los Headers, que se verificará en el servicio de AWS Cognito. Ningún usuario podrá acceder a ninguna de las APIs sin estar autenticado.

Se puede ver todo lo descrito en la Figura 39.

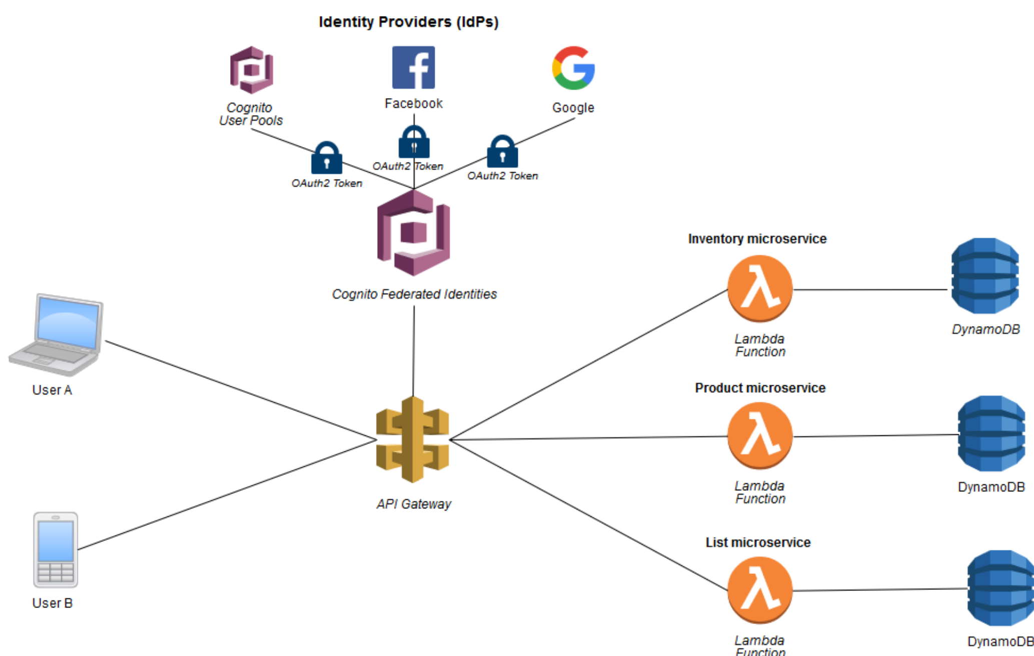


Figura 39. Arquitectura caso práctico

Veamos, por ejemplo, como es el proceso de login a través de Facebook (Figura 40).

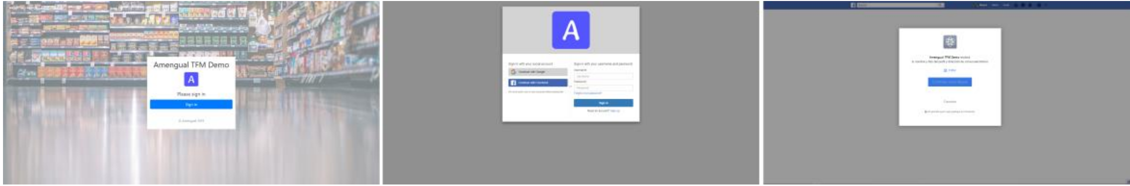


Figura 40. Proceso de Login con Facebook Connect

En la primera pantalla indicamos al usuario que debe iniciar sesión para ver el contenido. La segunda, se trata de una interfaz que nos proporciona el propio AWS Cognito, denominada Hosted UI, a partir de la configuración de nuestro grupo de usuarios. Y la tercera, es la interfaz de Facebook para aceptar los permisos solicitados.

Podemos ver en más detalle el flujo completo en la Figura 41:

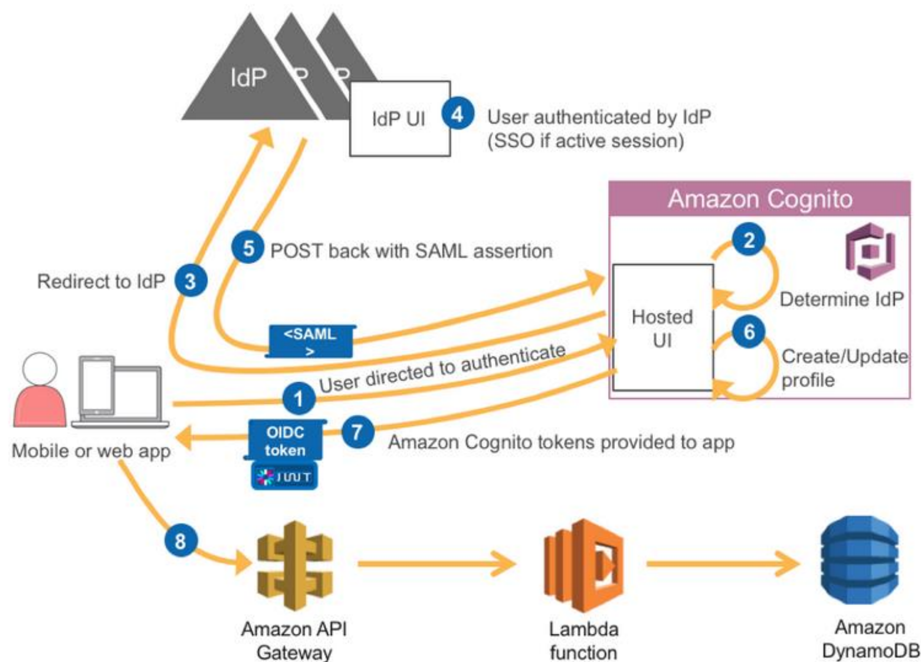


Figura 41. Flujo de autenticación caso práctico

1. La primera acción cuando el usuario accede a la aplicación es redirigirlo hacia la pantalla de login de AWS Cognito (Hosted UI).
2. A continuación, se elige el tipo de inicio de sesión.
3. El siguiente paso es redirigir al proveedor de identidad adecuado.
4. Si el usuario tiene sesión activa se le pide que acepte los permisos en caso que no los haya aceptado. En caso que no tenga la sesión activa antes debe iniciar sesión.
5. Se devuelve la información del usuario a AWS Cognito.
6. Con la información recibida, se crea o actualiza el usuario.
7. Amazon Cognito devuelve un conjunto de tokens a la aplicación, entre ellos el idToken, que contiene la identidad del usuario en formato JWT.
8. La aplicación almacena este token en la memoria del navegador y lo incluye en todas las llamadas que realiza a la API Gateway.

A continuación, veremos unas cuantas acciones que se pueden realizar con la aplicación web realizada:

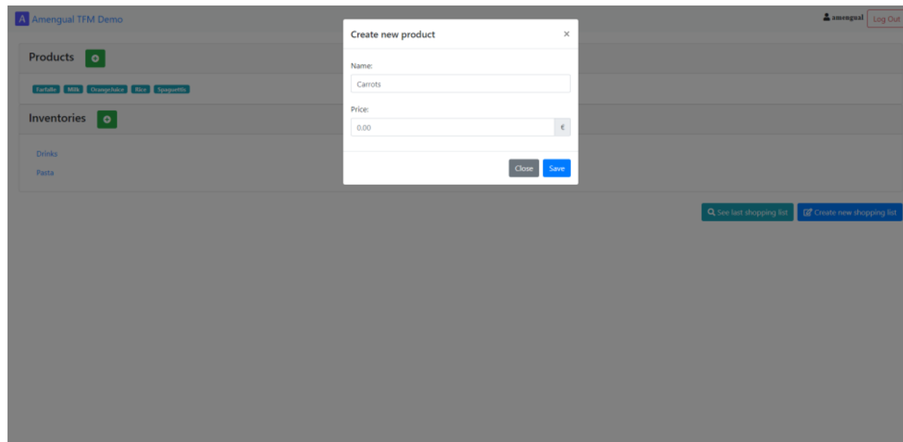


Figura 42. Web - Creación producto

- Creación de un producto indicando el nombre y el precio. (Figura 42)

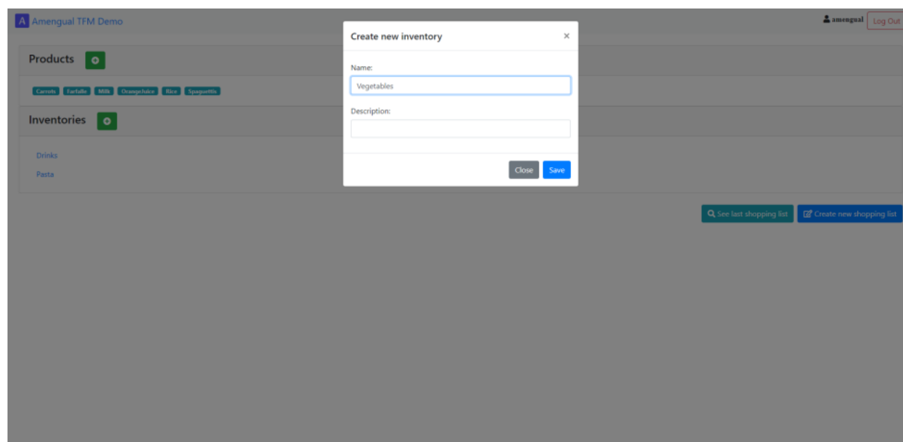


Figura 43. Web - Creación Inventario

- Creación de un inventario, entendido como una categoría en la que agrupar productos que mantengan algún tipo de relación. (Figura 43)

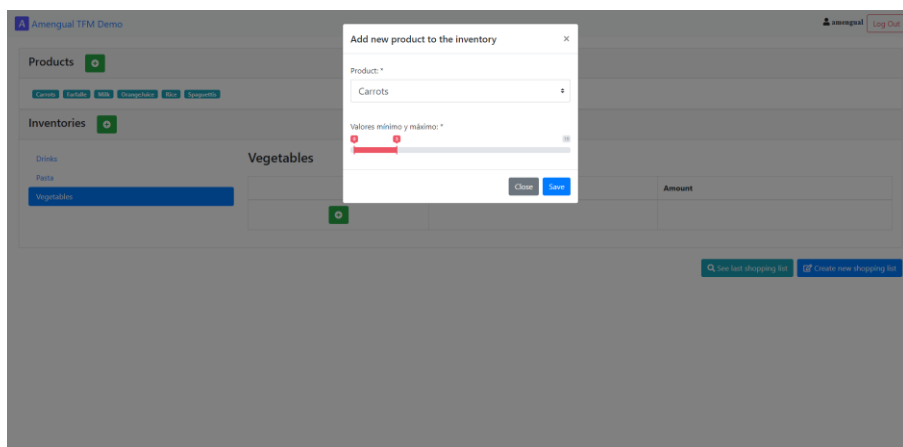


Figura 44. Web - Añadir producto al Inventario

- Añadir un producto a un inventario, especificando el rango deseado, con la cantidad máxima de este producto que se quiere tener en este inventario. (Figura 44)

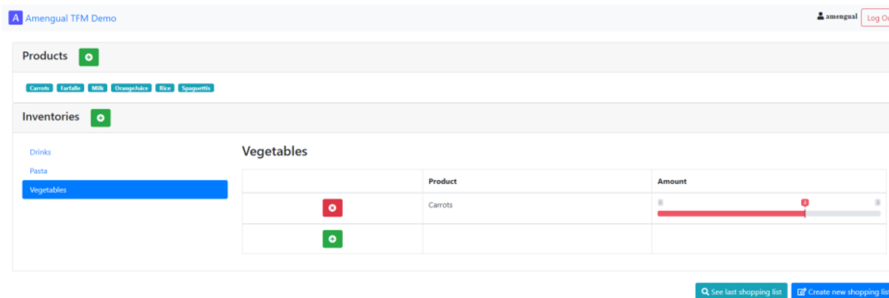


Figura 45. Web - Modificar cantidad de un producto del Inventario

- Modificar la cantidad que se tiene de un producto concreto en un inventario. (Figura 45)

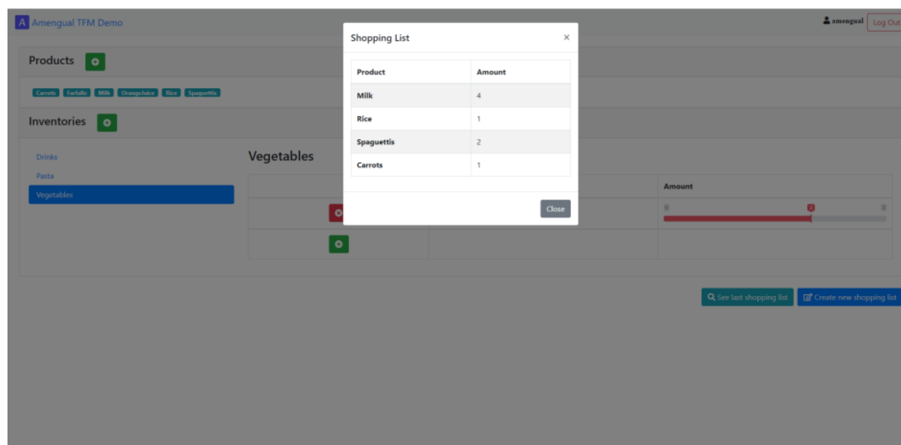


Figura 46. Web - Ver lista de la compra

- Ver la lista de la compra con la cantidad que falta de cada producto, sea cual sea su inventario, para llegar al máximo deseado para ese producto. (Figura 46)

La aplicación estará disponible a través del siguiente enlace: https://amengual-tfm-demo.s3-eu-west-1.amazonaws.com/login.html durante 6 meses desde la fecha de publicación de esta memoria. Por lo que se puede ver y probar la funcionalidad.
El código fuente de los diferentes microservicios puede verse en los anexos, mientras que el de la aplicación se puede ver en la propia página al tratarse de código de cliente.

8. Conclusiones y trabajo futuro

Con la realización de este trabajo se han asimilado y aplicado varios conceptos relacionados con la seguridad en APIs y API Managers. En general, podemos decir que la seguridad en el campo de las APIs es muy importante y así lo ven la mayoría de los profesionales de la industria.

Hemos visto la importancia de tener en cuenta los principales riesgos y hacer todo lo posible para reducir la probabilidad de que ocurran o mitigar su impacto. Esto se consigue con una buena implementación y la ayuda de herramientas diseñadas para ofrecer una mayor seguridad a nuestras APIs. Del mismo modo, mucho de los riesgos se reducen si hacemos uso de estándares.

Podemos concluir que este trabajo ha cumplido la mayoría de los objetivos planteados al principio. A continuación, exponemos cuáles se han cumplido y cuáles no por cada uno de los capítulos.

En el capítulo 2. Aspectos básicos de seguridad en APIs se ha cumplido el objetivo de conocer los principales riesgos a tener en cuenta a la hora de proteger una API. Así como métodos para mitigar su impacto.

En el capítulo 3. Estándares de diseño de APIs se han cumplido varios objetivos. En primer lugar, se han visto los estándares más comunes a la hora de desarrollar una API y la importancia de su uso. También se ha explicado el origen de OpenAPI Specification (OAS) y de OpenAPI Initiative (OAI) y se ha visto que se puede pensar en la seguridad desde el diseño de la API. Finalmente, se han estudiado los estándares OAS3 y API Blueprint con detalle, acompañado la explicación con ejemplos.
Sin embargo, no se ha profundizado mucho en como industria de las tecnologías de la información está gestionando y dirigiendo los estándares de seguridad, la razón fue que no encontramos mucha información de interés y la extensión del capítulo era suficiente.

El objetivo de conocer el estándar OAuth2 y su papel como mecanismo de autenticación y de autorización se ha superado en el capítulo 4. OAuth 2.0.

En el capítulo 5. Arquitectura de microservicios se ha cumplido el objetivo de conocer la arquitectura de microservicios basados en API.
Otro de los objetivos era profundizar sobre el concepto Seguridad como servicio, la razón principal era ver si era posible aplicarlo al caso práctico. Sin embargo, tras una breve investigación nos dimos cuenta de que teníamos un concepto equivocado y no era muy relevante para incluir en el trabajo.

En el capítulo 6. API Managament System, se ha visto el funcionamiento de los API Managers. Además, se han puesto en contexto dentro de un sistema de API Management System, que también incluye el concepto de API Gateway y API Portal, que, aunque no estaba en los objetivos se ha visto oportuno.
El objetivo de conocer diferentes API Managers y compáralos entre ellos se ha dejado de lado, el motivo es que queríamos empezar con el caso práctico dado el retraso en la planificación. Tras una primera búsqueda de distintos

productos, vimos que eran muy similares y requerían profundizar e incluso probar para poder describir mejor sus diferencias. Por lo que se decidió únicamente probar el de AWS, que usamos para el caso práctico.

Finalmente, el capítulo 7. Caso de estudio, ha consolidado muchos de los objetivos y se han asimilado muchos de los conceptos vistos de forma teórica. Se ha realizado un ejemplo de cómo varios microservicios pueden compartir un mismo sistema de autenticación y autorización.

La metodología creemos que ha sido adecuada, dado que se han adquirido muchos conocimientos que luego hemos podido aplicar en mayor o menor medida en el caso práctico.
En lo relativo al seguimiento de la planificación han surgido una serie de complicaciones. La idea inicial de dedicar 2 horas diarias era una estimación, a sabiendas de que algunos días se trabajaría más y otros menos. Lo importante era llevar un progreso semana a semana y lo principal cumplir con los hitos.

La primera incidencia surgió después de la primera entrega, dónde se detectó que en el punto 2 se explicaban de manera demasiado técnica y detallada las contramedidas de seguridad recomendadas. Además, no se hacía apenas referencia al riesgo que intentaban mitigar. Esto, nos hizo plantear una reestructuración de este punto, mientras se trabajaban en los próximos capítulos, lo que nos hizo tener que dedicar más horas de las previstas para llegar a la tercera entrega.
La segunda complicación vino dada por una mala previsión sobre el tiempo requerido para realizar el caso práctico, dado que hubo un primer tiempo para conocer y interconectar las herramientas a usar como AWS Cognito, AWS API Gateway y AWS Lambda. Esto hizo que al final las horas dedicadas fueron algo mayores a las previstas. La lección aprendida es que debimos haber dejado unos colchones de tiempo principalmente antes de cada entrega, para prevenir posibles desviaciones de tiempo.

En cuanto a cómo se podría ampliar el trabajo en el futuro, consideramos que sería interesante añadir un WAF a nuestra solución, así como ampliar las funcionalidades de nuestras APIs para permitir que la aplicación crezca.

9. Glosario

1. Dispositivo IoT: Las siglas IoT hacen referencia a Internet of Things. Un dispositivo IoT es aquella que está conectado a la red para ampliar su funcionalidad. Por ejemplo, una bombilla inteligente, que podemos encender y apagar desde nuestro dispositivo móvil.

2. Entorno Cloud: Tanto cloud como cloud computing son términos que se utilizan para describir el concepto de almacenar y acceder a la información en Internet. No se tiene conocimiento de donde están físicamente los datos, se accede a ellos a través de servicios de terceros.

3. Microservicios: Los microservicios son tanto un estilo de arquitectura como un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus componentes más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se integra en una única pieza, los microservicios son independientes y funcionan en conjunto para llevar a cabo las mismas tareas. Cada uno de estos elementos o procesos es un microservicio.

4. Bots: Se refiere a un tipo de programa informático autónomo que es capaz de llevar a cabo tareas concretas e imitar el comportamiento humano.

5. Botnet: Una botnet es una red de bots, se trata de un gran número de equipos informáticos que han sido "secuestrados" por malware, de forma que quedan a disposición de un hacker.

6. CDN: Es una red de Distribución de Contenido o Content Delivery Network (por sus siglas en inglés), se trata de un conjunto de servidores ubicados en diferentes zonas geográficas que contienen copias locales de los contenidos de los clientes. El objetivo es que los visitantes que están lejos reciban rápidamente el contenido del sitio web desde el servidor más cercano físicamente.

7. UUID: son las siglas de Universally Unique IDentifier, que en inglés significa, literalmente, 'identificador único universal'. Como tal, el UUID es un código identificador estándar empleado en el proceso de construcción de software. Es utilizado para crear identificadores únicos universales que permitan reconocer e distinguir un objeto dentro de un sistema, o el mismo objeto en diferentes contextos.

8. JSON: El acrónimo JSON deriva de la expresión de la lengua inglesa JavaScript Object Notation, que puede traducirse como Notación de Objeto de JavaScript. Se trata de un formato de texto ligero que permite intercambiar datos.

9. Clickjacking: también conocido como "ataque de compensación de UI", es cuando un atacante usa varias capas transparentes u opacas para engañar a un usuario para que haga click en un botón o enlace en otra página cuando intenta hacer click en la página del nivel superior. Por lo tanto, el

atacante está "secuestrando" los clicks destinados a su página y enrutando a otra página, muy probablemente propiedad de otra aplicación, dominio o ambos.

10. Iframe: es un elemento HTML que permite insertar o incrustar un documento HTML (por ejemplo, otra web) dentro del documento HTML principal.

11. CSRF: Las vulnerabilidades de falsificación de petición en sitios cruzados fuerzan al navegador validado de una víctima a enviar una petición a una aplicación Web vulnerable, la cual entonces realiza la acción elegida a través de la víctima.

12. IP: es la sigla de Internet Protocol. Se trata de un estándar que se emplea para el envío y recepción de información mediante una red que reúne paquetes conmutados. No cuenta con la posibilidad de confirmar si un paquete de datos llegó a su destino.

13. Tests unitarios: Son una forma de comprobar el correcto funcionamiento de una unidad de código. Esto sirve para asegurar que cada unidad funcione correcta y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de respuesta, etc.

14. Bugs: Es un error o simplemente fallo en un sistema de software que desencadena un resultado indeseado.

15. Design first: El enfoque de Design First aboga por el diseño del contrato de la API antes de escribir cualquier código. Este es un enfoque relativamente nuevo, pero se está poniendo en práctica con rapidez, especialmente con el uso de formatos de descripción de API.

16. Markdown: es un lenguaje de marcado que facilita la aplicación de formato a un texto empleando una serie de caracteres de una forma especial. En principio, fue pensado para elaborar textos para la web con más rapidez y sencillez que si estuviésemos empleando directamente HTML.

17. CRM: Las siglas CRM (Gestión de relaciones con los clientes) es un término de la industria de la información que se aplica a metodologías, software y, en general, a las capacidades de Internet que ayudan a una empresa a gestionar las relaciones con sus clientes de una manera organizada.

18. NoSQL: es una amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico de SGBDR (Sistema de Gestión de Bases de Datos Relacionales) en aspectos importantes, siendo el más destacado que no usan SQL como lenguaje principal de consultas.

10. Bibliografía

[1] En *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*, de De Brajesh, 1-2. Apress, 2017.

[2] Carnell, John. *Spring Microservices in Action.* Manning, 2017.

[3] Imperva. "API Security Survey." *One Poll survey of 250 IT professionals on the state of application.* Enero 25, 2018. https://www.slideshare.net/Imperva/api-security-survey?qid=aab26aae-da6c-40b0-8c28-0ea3a0f526a8&v=&b=&from_search=4 (accessed Abril 2019, 16).

[4] Internet Engineering Task Force (IETF) . *RFC 6749 - The OAuth 2.0 Authorization Framework.* s.f. https://tools.ietf.org/pdf/rfc6749.pdf (último acceso: 21 de Abril de 2019).

[5] OpenID. *The Internet Identity Layer.* s.f. https://openid.net/connect/ (último acceso: 06 de 05 de 2019).

[6] ProgrammableWeb. *Programmable Web.* n.d. https://www.programmableweb.com/news/programmableweb-api-directory-eclipses-17000-api-economy-continues-surge/research/2017/03/13 (accessed Abril 12, 2019).

[7] Richardson, Chris. *NGINX Blog.* 2015. https://www.nginx.com/blog/building-microservices-using-an-api-gateway/ (último acceso: 29 de Abril de 2019).

[8] Richer, Justin y Sanso, Antonio. *OAuth 2 in Action.* Manning, 2017.

[9] Scudlayer. *What is a DDOS attack.* s.f. https://www.scudlayer.com/en/ddos-attacks/ (último acceso: 18 de Abril de 2019).

[10] Shieldfy Open Source. *API Security Checklist.* s.f. https://github.com/shieldfy/API-Security-Checklist/blob/master/README-es.md (último acceso: 12 de Marzo de 2019).

[11] En *Undisturbed REST: A Guide to Designing the Perfect API*, de Michael Stowe, 1-3. Mulesoft, 2015.

11. Anexos

11.1 Documentación Product API

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Simple Product API
  description: A sample API that allow perform basic operations onto products
  contact:
    name: Miguel Amengual Bauza
    email: amengual27@uoc.edu
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
servers:
  - url: https://glj3mba16a.execute-api.eu-west-1.amazonaws.com/api
paths:
  /product:
    get:
      description: Returns all products created by the authenticated user.
      operationId: findProducts
      parameters:
        - name: name
          in: query
          description: name of the product to search by
          required: false
          schema:
            type: string
        - name: limit
          in: query
          description: maximum number of results to return
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: product response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Product'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
```

```yaml
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
    security:
      - bearerAuth: []
  post:
    description: Creates a new product. Duplicates names are not allowed
    operationId: addProduct
    requestBody:
      description: Product to add to the user account
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Product'
    responses:
      '200':
        description: product response
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Product'
      '401':
        $ref: '#/components/responses/UnauthorizedError'
      default:
        description: unexpected error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
    security:
      - bearerAuth: []
/product/{name}:
  get:
    description: Returns the information of the product
    operationId: findProduct
    parameters:
      - name: name
        in: path
        description: name of product to fetch
        required: true
        schema:
          type: string
    responses:
      '200':
        description: product response
        content:
          application/json:
            schema:
```

```
                $ref: '#/components/schemas/Product'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
    delete:
      description: deletes a single product based on the name supplied
      operationId: deleteProduct
      parameters:
        - name: name
          in: path
          description: name of product to delete
          required: true
          schema:
            type: string
      responses:
        '204':
          description: inventory deleted
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
components:
  schemas:
    Product:
      required:
        - name
      properties:
        name:
          type: string
        description:
          type: string
        price:
          type: number
          format: float
        currency:
          type: string
    Error:
```

```
    required:
      - code
      - message
    properties:
      code:
        type: integer
        format: int32
      message:
        type: string
responses:
  UnauthorizedError:
    description: Access token is missing or invalid
securitySchemes:
  bearerAuth:
    type: http
    scheme: bearer
    bearerFormat: JWT
```

11.2 Documentación Inventory API


```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Simple Inventory API
  description: A sample API that allow perform basic operations onto inventories
  contact:
    name: Miguel Amengual Bauza
    email: amengual27@uoc.edu
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
servers:
  - url: https://glj3mba16a.execute-api.eu-west-1.amazonaws.com/api
paths:
  /inventory:
    get:
      description: Returns all inventories created by the authenticated user.
      operationId: findInventories
      parameters:
        - name: limit
          in: query
          description: maximum number of results to return
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: inventory response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Inventory'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
    post:
      description: Creates a new inventory. Duplicates names are not allowed
      operationId: addInventory
```

```
      requestBody:
        description: Inventory to add to the user account
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/NewInventory'
      responses:
        '200':
          description: inventory response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Inventory'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
  /inventory/{inventoryName}:
    get:
      description: Returns the information of the inventory
      operationId: findInventory
      parameters:
        - name: inventoryName
          in: path
          description: name of inventory to fetch
          required: true
          schema:
            type: string
      responses:
        '200':
          description: inventory response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Inventory'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
```

```
      security:
        - bearerAuth: []
    delete:
      description: deletes a single inventory based on the ID supplied
      operationId: deleteInventory
      parameters:
        - name: inventoryName
          in: path
          description: name of inventory to delete
          required: true
          schema:
            type: string
      responses:
        '204':
          description: inventory deleted
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
  /inventory/{inventoryName}/product:
    get:
      description: Returns the products of the inventory
      operationId: findInventoryProducts
      parameters:
        - name: inventoryName
          in: path
          description: name of inventory
          required: true
          schema:
            type: string
      responses:
        '200':
          description: inventory products response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/InventoryProduct'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
```

```
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
    post:
      description: Adds a new product to the inventory. Duplicates producs are not allowed
      operationId: addProduct
      parameters:
        - name: inventoryName
          in: path
          description: name of inventory to modify
          required: true
          schema:
            type: string
      requestBody:
        description: Product to add to the user inventory
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/InventoryProduct'
      responses:
        '200':
          description: product created
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
  /inventory/{inventoryName}/product/{productName}:
    put:
      description: updates the amount of a product from an inventory
      operationId: updateProduct
      parameters:
        - name: inventoryName
          in: path
          description: name of inventory to modify
          required: true
          schema:
            type: string
        - name: productName
          in: path
          description: name of product to update
```

```
      required: true
      schema:
        type: string
  requestBody:
    description: amount to the set to the product
    required: true
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/UpdateInventoryProduct'
  responses:
    '200':
      description: product updated
    '401':
      $ref: '#/components/responses/UnauthorizedError'
    default:
      description: unexpected error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
  security:
    - bearerAuth: []
delete:
  description: deletes a single product from an inventory
  operationId: deleteProduct
  parameters:
    - name: inventoryName
      in: path
      description: name of inventory to modify
      required: true
      schema:
        type: string
    - name: productName
      in: path
      description: name of product to delete
      required: true
      schema:
        type: string
  responses:
    '204':
      description: product deleted
    '401':
      $ref: '#/components/responses/UnauthorizedError'
    default:
      description: unexpected error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
```

```yaml
      security:
        - bearerAuth: []
components:
  schemas:
    Inventory:
      allOf:
        - $ref: '#/components/schemas/NewInventory'
        - required:
          - id
          properties:
            id:
              type: integer
              format: int32
    NewInventory:
      required:
        - name
      properties:
        name:
          type: string
        description:
          type: string
    InventoryProduct:
      required:
        - product
      properties:
        product:
          type: string
        amount:
          type: integer
          format: int32
        minAmount:
          type: integer
          format: int32
        maxAmount:
          type: integer
          format: int32
    UpdateInventoryProduct:
      properties:
        amount:
          type: integer
          format: int32
    Error:
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
```

```
        type: string
responses:
  UnauthorizedError:
    description: Access token is missing or invalid
securitySchemes:
  bearerAuth:
    type: http
    scheme: bearer
    bearerFormat: JWT
```

11.3 Documentación List API


```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Simple List API
  description: A sample API that allow perform basic operations onto lists
  contact:
    name: Miguel Amengual Bauza
    email: amengual27@uoc.edu
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
servers:
  - url: https://glj3mba16a.execute-api.eu-west-1.amazonaws.com/api
paths:
  /list:
    get:
      description: Returns all lists created by the authenticated user ordered by most recent first.
      operationId: findLists
      parameters:
        - name: completed
          in: query
          description: filter by completed or uncompleted lists
          required: false
          schema:
            type: boolean
        - name: limit
          in: query
          description: maximum number of results to return
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: list response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/List'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
```

```
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
    post:
      description: Creates a new list. Duplicates names are not allowed
      operationId: addList
      requestBody:
        description: List to add to the user account
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/List'
      responses:
        '200':
          description: list response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/List'
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
  /list/{name}:
    get:
      description: Returns the information of the list
      operationId: findList
      parameters:
        - name: name
          in: path
          description: Name of list to fetch
          required: true
          schema:
            type: string
      responses:
        '200':
          description: list response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/List'
        '401':
```

```
      $ref: '#/components/responses/UnauthorizedError'
    default:
      description: unexpected error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
  security:
    - bearerAuth: []
put:
  description: modify the completed attribute of the list
  operationId: modifyList
  parameters:
    - name: name
      in: path
      description: Name of list to modify
      required: true
      schema:
        type: string
  requestBody:
    description: value of completed attribute to set
    required: true
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/ChangeList'
  responses:
    '200':
      description: list updated
    '401':
      $ref: '#/components/responses/UnauthorizedError'
    default:
      description: unexpected error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
  security:
    - bearerAuth: []
delete:
  description: deletes a single list based on the name supplied
  operationId: deleteList
  parameters:
    - name: name
      in: path
      description: Name of list to delete
      required: true
      schema:
        type: string
  responses:
```

```
        '204':
          description: list deleted
        '401':
          $ref: '#/components/responses/UnauthorizedError'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      security:
        - bearerAuth: []
components:
  schemas:
    List:
      required:
        - name
      properties:
        name:
          type: string
        description:
          type: string
        products:
          type: array
          items:
            type: object
            properties:
              product:
                type: string
              amount:
                type: integer
                format: int32
        completed:
          type: boolean
    ChangeList:
      required:
        - completed
      properties:
       completed:
        type: boolean
    Error:
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```

```
responses:
  UnauthorizedError:
    description: Access token is missing or invalid
securitySchemes:
  bearerAuth:
    type: http
    scheme: bearer
    bearerFormat: JWT
```

11.4 Implementación Product API

```
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {

   var username = null;

   if (event && event.requestContext && event.requestContext.authorizer &&
event.requestContext.authorizer.claims &&
event.requestContext.authorizer.claims['cognito:username']) {
      username = event.requestContext.authorizer.claims['cognito:username']
   }

   if (username == null) {
      done(new Error(`Unable to get authorized username`));
   }

   var tableName = 'product';

   console.log('Received event:', JSON.stringify(event, null, 2));

   const done = (err, res) => callback(null, {
      statusCode: err ? '400' : '200',
      body: err ? err.message : JSON.stringify(res),
      headers: {
         'Content-Type': 'application/json',
         'Access-Control-Allow-Origin' : "*", // Required for CORS support to work
         'Access-Control-Allow-Credentials' : true // Required for cookies,
authorization headers with HTTPS
      },
   });

   if (event.resource === '/product') {

      switch (event.httpMethod) {

         case 'GET':

            var limit = event.queryStringParameters != null &&
event.queryStringParameters['limit'] != null ?
event.queryStringParameters['limit'] : 100;
            var nameLike = event.queryStringParameters != null ?
event.queryStringParameters['name'] : null;

            if (nameLike == null) {
               dynamo.query({TableName: tableName, KeyConditionExpression:
"username = :username",
```

```
            ExpressionAttributeValues: {":username": username}, Limit: limit},
done);
          }
          else {
            dynamo.query({TableName: tableName, KeyConditionExpression:
"username = :username and begins_with(#name, :nameLike)",
            ExpressionAttributeValues: {":username": username, ":nameLike":
nameLike}, ExpressionAttributeNames: {"#name" : "name"}, Limit: limit}, done);
          }
          break;

        case 'POST':

          var eventBody = JSON.parse(event.body);
          var product = {
            'name': eventBody['name'].replace(/[^\w]/gi, ''),
            'price': eventBody['price'] ? eventBody['price'] : 0.0,
            'currency': eventBody['currency'],
            'username': username
          };

          dynamo.putItem({TableName: tableName, Item: product}, done);
          break;

        default:
          done(new Error(`Unsupported method "${event.httpMethod}"`));
      }
  }
  else if (event.resource === '/product/{id}') {

      var product = {
        'name': decodeURIComponent(event.pathParameters['id']),
        'username': username
      };

      switch (event.httpMethod) {
        case 'GET':
          dynamo.getItem({ TableName: tableName, Key: product }, done);
          break;
        case 'DELETE':
          dynamo.deleteItem({TableName: tableName, Key: product}, done);
          break;
        default:
          done(new Error(`Unsupported method "${event.httpMethod}"`));
      }
  }
  else {
      done(new Error(`Unsupported resource "${event.resource}"`));
  }
};
```

11.5 Implementación Inventory API

```
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {

   console.log('Received event:', JSON.stringify(event, null, 2));

   var username = null;

   if (event && event.requestContext && event.requestContext.authorizer &&
event.requestContext.authorizer.claims &&
event.requestContext.authorizer.claims['cognito:username']) {
      username = event.requestContext.authorizer.claims['cognito:username']
   }

   if (username == null) {
      done(new Error(`Unable to get authorized username`));
   }

   const done = (err, res) => callback(null, {
      statusCode: err ? '400' : '200',
      body: err ? err.message : JSON.stringify(res),
      headers: {
         'Content-Type': 'application/json',
         'Access-Control-Allow-Origin' : "*", // Required for CORS support to work
         'Access-Control-Allow-Credentials' : true // Required for cookies,
authorization headers with HTTPS
      },
   });

   if (event.resource === '/inventory') {

      var tableName = 'inventory';

      switch (event.httpMethod) {
         case 'GET':
            var limit = event.queryStringParameters != null &&
event.queryStringParameters['limit'] != null ?
event.queryStringParameters['limit'] : 100;
            dynamo.query({TableName: tableName, KeyConditionExpression:
"username = :username",
            ExpressionAttributeValues: {":username": username}, Limit: limit},
done);
            break;
         case 'POST':
            var eventBody = JSON.parse(event.body);
            var inventory = {
               'name': eventBody['name'].replace(/[^\w]/gi, ''),
```

```
                'username': username
            };
            if (eventBody['description']) {
                inventory['description'] = eventBody['description'];
            }
            dynamo.putItem({TableName: tableName, Item: inventory}, done);
            break;
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`));
    }
}
else if (event.resource === '/inventory/{id}/product') {

    var tableName = 'inventory-products';
    var usernameInventory =  username + '.' + event.pathParameters['id'];

    switch (event.httpMethod) {
        case 'GET':
            dynamo.query({TableName: tableName, KeyConditionExpression:
"#id = :id", ExpressionAttributeValues: {":id": usernameInventory},
            ExpressionAttributeNames: {"#id": "username.inventory"}}, done);
            break;
        case 'POST':
            var eventBody = JSON.parse(event.body);
            var inventoryProduct = {
                "username.inventory": usernameInventory,
                "product": decodeURIComponent(eventBody['product']),
                "minAmount": eventBody['minAmount'],
                "maxAmount": eventBody['maxAmount'],
                "amount": eventBody['amount']
            };
            dynamo.putItem({TableName: tableName, Item: inventoryProduct},
done);
            break;
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`));
    }
}
else if (event.resource === '/inventory/{id}/product/{product}') {

    var tableName = 'inventory-products';
    var inventoryProduct = {
        "username.inventory": username + '.' + event.pathParameters['id'],
        "product": decodeURIComponent(event.pathParameters['product'])
    };

    switch (event.httpMethod) {
        case 'DELETE':
            dynamo.deleteItem({TableName: tableName, Key: inventoryProduct},
done);
```

```
            break;
        case 'PUT':
            var eventBody = JSON.parse(event.body);
            var amount = eventBody['amount'];
            dynamo.updateItem({TableName: tableName, Key: inventoryProduct,
UpdateExpression: "set amount = :amount",
            ExpressionAttributeValues:            {":amount":            amount},
ReturnValues:"UPDATED_NEW"}, done);
            break;
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`));
      }
   }
   else {
      done(new Error(`Unsupported resource "${event.resource}"`));
   }
};
```

11.6 Implementación List API

```
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {

   console.log('Received event:', JSON.stringify(event, null, 2));

   var username = null;

   if (event && event.requestContext && event.requestContext.authorizer &&
event.requestContext.authorizer.claims &&
event.requestContext.authorizer.claims['cognito:username']) {
      username = event.requestContext.authorizer.claims['cognito:username']
   }

   if (username == null) {
      done(new Error(`Unable to get authorized username`));
   }

   const done = (err, res) => callback(null, {
      statusCode: err ? '400' : '200',
      body: err ? err.message : JSON.stringify(res),
      headers: {
         'Content-Type': 'application/json',
         'Access-Control-Allow-Origin' : "*", // Required for CORS support to work
         'Access-Control-Allow-Credentials' : true // Required for cookies,
authorization headers with HTTPS
      },
   });

   var tableName = 'list';

   if (event.resource === '/list') {

      switch (event.httpMethod) {
         case 'GET':
            var limit = event.queryStringParameters != null &&
event.queryStringParameters['limit'] != null ?
event.queryStringParameters['limit'] : 100;
            dynamo.query({TableName: tableName, KeyConditionExpression:
"username = :username",
            ExpressionAttributeValues: {":username": username},
ScanIndexForward: false, Limit: limit}, done);
            break;
         case 'POST':
            var eventBody = JSON.parse(event.body);
            var list = {
               'name': eventBody['name'].replace(/[^\w]/gi, ''),
```

```
                'username': username
            };
            if (eventBody['description']) {
                list['description'] = eventBody['description'];
            }
            if (eventBody['products']) {
                list['products'] = eventBody['products'];
            }
            dynamo.putItem({TableName: tableName, Item: list}, done);
            break;
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`));
    }
  }
  else if (event.resource === '/list/{id}') {

    var list = {
        'name': decodeURIComponent(event.pathParameters['id']),
        'username': username
    };

    switch (event.httpMethod) {
        case 'GET':
            dynamo.getItem({ TableName: tableName, Key: list}, done);
            break;
        case 'PUT':
            var eventBody = JSON.parse(event.body);
            var completed = eventBody['completed'];
            dynamo.updateItem({TableName: tableName, Key: list,
UpdateExpression: "set completed = :completed",
            ExpressionAttributeValues: {":completed": completed},
ReturnValues:"UPDATED_NEW"}, done);
            break;
        case 'DELETE':
            dynamo.deleteItem({TableName: tableName, Key: list}, done);
            break;
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`));
    }
  }
  else {
    done(new Error(`Unsupported resource "${event.resource}"`));
  }
};
```