

Aplicando seguridad a una API REST con JSON Web Tokens

Alumno: Eduardo Marcelo Salas González

Plan de Estudios: Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones MISTIC.

Área del trabajo final: Sistemas de autenticación y autorización

Consultor del TFM: Àngel Linares Zapater

Profesor responsable de la asignatura: Victor Garcia Font

Fecha de Entrega: diciembre 2020



Esta obra está sujeta a una licencia de
Reconocimiento-NoComercial-CompartirIgual
[3.0 España de Creative Commons](https://creativecommons.org/licenses/by-nc-sa/3.0/es/)

FICHA DEL TRABAJO FINAL

Título del trabajo:	<i>Aplicando seguridad a una API REST con JSON Web Tokens</i>
Nombre del autor:	<i>Eduardo Marcelo Salas González</i>
Nombre del consultor:	Àngel Linares Zapater
Nombre del PRA:	<i>Víctor García Font</i>
Fecha de entrega:	12/2020
Titulación:	Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones
Área del Trabajo Final:	Sistemas de autenticación y autorización
Idioma del trabajo:	<i>Castellano</i>
Palabras clave	<i>JWT, API-REST, KONG, KONGA</i>
Resumen del Trabajo:	
<p>La finalidad de este trabajo es poder presentar algunos ejemplos de configuración de seguridad utilizando JWT en las API REST que son publicadas a internet. Para ello se ha implementado en una fase inicial una prueba de concepto de la creación de una API REST codificada en JavaScript y desplegada en Node.js. En esta primera fase del proyecto los tokens JWT son firmados de forma simétrica (palabra secreta) en donde luego se ha ido incrementando la seguridad añadiendo claves asimétricas para obtener tokens más robustos.</p> <p>En una segunda fase de implementación seguiremos mejorando la infraestructura añadiendo Kong como API Gateway que permite una nueva capa de abstracción a la API creada en la fase anterior. Además, Kong en su <i>versión community</i> nos ofrece una serie de plugins que enriquecen las diferentes gestiones para las API. Finalmente, para facilitar el trabajo de gestionar diferentes API se ha instalado la herramienta Konga (dashboard) que es el complemento perfecto para una gestión de forma gráfica y cómoda para Kong.</p>	

Abstract:

The purpose of this work is to be able to present some examples of security configuration using JWT in the REST API that are published on the Internet. For this, a proof of concept has been implemented in the initial phase of the creation of a REST API development in JavaScript and deployed in Node.js. In this first phase of the project, the JWT tokens are signed symmetrically (secret key), where security has been increased later by adding asymmetric keys to obtain more robust tokens.

In a second implementation phase, we continue to improve the infrastructure adding Kong as an API Gateway that allows a new abstraction layer to the API created in the previous phase. In addition, Kong in its *community version* offers us a series of plugins that enrich the different procedures for the APIs. Finally, to facilitate the work of managing different API, the Konga tool (dashboard) has been installed, which is the perfect complement for graphical and comfortable management for Kong.

ÍNDICE

1.	Introducción	1
1.1.	Contexto y justificación del Trabajo	1
1.2.	Objetivos del Trabajo	2
1.3.	Enfoque y método seguido.....	3
1.4.	Planificación del Trabajo	4
1.5.	Planificación temporal de las tareas y sus dependencias.....	5
1.6.	Análisis de riesgo	7
1.7.	Estado del arte.....	8
1.7.1	Arquitectura API REST	8
1.7.2	Estándares para la definición de Tokens	8
1.7.3	Herramientas Gateway para API REST	9
1.8.	Recursos necesarios y coste de realización del proyecto.....	9
2.	Investigación y Análisis.....	10
2.1.	Arquitectura API REST.....	11
2.2.	Análisis de los estándares para API REST.....	12
2.2.1	JSON Web Token	12
2.2.2	JSON Web Signature.....	15
2.2.3	JSON Web Encryption	15
2.2.4	JSON Object Signing and Encryption (JOSE)	15
2.3.	JWT fallos de seguridad y buenas prácticas	20
2.3.1	Ataque "alg: none"	20
2.3.2	Claves débiles de HMAC	20
2.3.3	Ataques de sustitución.....	21
2.4.	Herramientas API Gateway y Dashboard.....	23
2.4.1	Kong API Gateway.....	24
2.4.3	Tyk.io API Gateway	25
2.4.3	Apigee API Gateway	25
2.4.4	Elección de API Gateway.....	25
2.4.5	Konga Dashboard	26
2.5.	Otras Herramientas	28
2.5.1	Postman	28
2.5.2	API REST Pública.....	28
2.5.3	Swagger Editor	29
2.5.4	RoboMongo	29
2.6.	Arquitectura API REST.....	30
3.	Implantación	32
3.1	Implantación infraestructura.....	32
3.1.1	Instalación Servidor	32
3.1.2	Node.js + Módulos	32
3.1.3	Instalación MongoDB.....	34
3.2	Codificación API REST	35
3.2.1	Codificación endpoints.....	36
3.2.2	Documentación API	37
3.2.3	Ejemplos de endpoints.....	38
3.2.4	Añadir Seguridad a los endpoints	42
3.3	Administración API REST	46
3.3.1	Instalación y configuración API Gateway Kong	46
3.3.2	Instalación y configuración Dashboard Konga	50
4.	Conclusiones	56
4.1	Evaluación de los objetivos.....	57
4.2	Trabajo Futuro	58
5.	Glosario	59
6.	Bibliografía.....	61

7.	Anexos	63
7.1	Código fuente TFM	63
7.2	Ficheros de configuración	70
7.3	Definición endpoint con Swagger Editor	70

LISTA DE FIGURAS

Figura 1	- JWT Autenticación/Autorización [13].....	10
Figura 2	- Token JWT separados por punto [16].....	12
Figura 3	- JWT formato JSON.....	12
Figura 4	- Public claims.....	13
Figura 5	- Tipos de claims.....	14
Figura 6	- JWT como clase abstracta [19].....	14
Figura 7	- Algoritmos disponibles para el claim "alg"	16
Figura 8	- Serialización Compacta [22]	16
Figura 9	- Serialización JSON [22].....	17
Figura 10	- JWS Serialización Compacta [19].....	17
Figura 11	- Ejemplo JWS Header.....	18
Figura 12	- JWE Serialización Compacta [19].....	18
Figura 13	- Ejemplo JWE Header.....	19
Figura 14	- Peticiones de forma directa [23].....	23
Figura 15	- API Gateway peticiones [23].....	24
Figura 16	- API Gateway con balanceo de servicios [23]	24
Figura 17	- Cuadrante mágico API Gateway [24]	26
Figura 18	- Konga Interfaz web [10].....	27
Figura 19	- Herramienta Postman [25].....	28
Figura 20	- API REST Publica.....	28
Figura 21	- Swagger Editor	29
Figura 22	- Robomongo	29
Figura 23	- Diseño arquitectura fase 1	30
Figura 24	- Diseño arquitectura fase 2	31
Figura 25	- Iniciando proyecto Node.js.....	33
Figura 26	- Iniciando App.....	33
Figura 27	- Comprobando App.....	33
Figura 28	- Base de Datos de Colecciones	34
Figura 29	- Ver colecciones	34
Figura 30	- Llamada endpoint login.....	35
Figura 31	- Web jwt.io debugger	35
Figura 32	- Añadiendo palabra secreta	36
Figura 33	- Swagger Editor EndPoint definidos.....	37
Figura 34	- Swagger Editor POST.....	37
Figura 35	- Swagger Editor DELETE	38
Figura 36	- Swagger Editor PUT	38
Figura 37	- Endpoint Registrar usuario	39
Figura 38	- MongoDB Insert nuevo registro	39
Figura 39	- Endpoint Consultar contenido público.....	40
Figura 40	- Endpoint Borrar usuario	40
Figura 41	- Registro eliminado	40
Figura 42	- MongoDB Usuario eliminado	41
Figura 43	- Insertar Token a la petición.....	41
Figura 44	- Enviar petición tipo PUT	41
Figura 45	- MongoDB actualización colección	41
Figura 46	- API REST con HTTPS.....	42
Figura 47	- Nuevo Token	44

Figura 48 - Visualizar contenido Token en jwt.io	45
Figura 49 - Validando Token.....	45
Figura 50 - Kong Servicio Admin API.....	47
Figura 51 - Kong Servicio Proxy	47
Figura 52 - Kong Ver servicios Activos	48
Figura 53 - Kong Crear servicio	48
Figura 54 - Creando Routes.....	48
Figura 55 - GET desde Kong	49
Figura 56 - GET desde reqres.in.....	49
Figura 57 - Konga iniciar servicio.....	51
Figura 58 - Konga primer inicio	51
Figura 59 - Konga acceso a la aplicación.....	51
Figura 60 - Konga crear conexión	52
Figura 61 - Konga datos conexión Kong	52
Figura 62 - Konga Dashboard.....	53
Figura 63 - Konga apartado Services.....	53
Figura 64 - Konga nuevo servicio creado.....	54
Figura 65 - Konga rutas para API-TFM	54
Figura 66 - Llamada del tipo POST por Kong	55
Figura 67 - Llamada del tipo GET por Kong.....	55

ÍNDICE TABLAS

Tabla 1 - Planificación.....	4
Tabla 2 - Diagrama de Gantt.....	5
Tabla 3 - Recursos y costes.....	9
Tabla 4 - JWS claims	18
Tabla 5 - Endpoints Autenticación.....	36
Tabla 6 - Endpoints Autorización	36

ÍNDICE SNIPPET-CODE

Snippet-Code 1 - Cambio de Puerto APP	43
Snippet-Code 2 - Clave Privada.....	43
Snippet-Code 3 - Clave Pública	44

1. Introducción

1.1. Contexto y justificación del Trabajo

Sin lugar a duda, hoy en día estamos viviendo en una era en donde la transferencia, el intercambio de mensajería es una parte importante para las compañías y administraciones. Estas compañías y/o administraciones publican sus servicios para que otros se puedan conectar e interactuar y viceversa. Consiguiendo una colaboración mucha más ágil entre ellas, creando nuevos modelos de negocio, etc. Podemos encontrar claros ejemplos con bancos, correos, YouTube, Twitter en donde publican sus API (Application Programming Interface) para que puedan ser consumidas. En muchas ocasiones llegando a crear nuevas aplicaciones o servicios por parte de los programadores.

Con la utilización de las API las compañías tienen la oportunidad de poder abstraer parte de su infraestructura en donde podrán publicar un conjunto de procedimientos y funciones para sus clientes y/o usuarios finales.

Podemos observar que muchas de estas API ya son parte importante dentro de los recursos de una compañía. Una mala configuración o la interrupción de estos servicios podría ocasionar pérdidas económicas o aún peor, perder la confianza de los clientes actuales y en donde podría afectar a la reputación de la compañía hacia posibles potenciales clientes que necesiten contratar algún servicio.

Hay que tener en cuenta que muchas de las peticiones que se pueden realizar a estas API pueden ser peticiones de información sensible o modificaciones en los datos. Lo que nos lleva a pensar que dejar expuestos estos servicios en donde no requieran autenticación ni autorización para acceder a los recursos podría conllevar a graves problemas. Necesitamos desplegar determinadas medidas de seguridad para evitar que cualquier usuario, robot o buscadores puedan consultar dicha información sin previa autorización.

En este trabajo fin de máster (TFM) se investigará sobre el uso de JSON Web Tokens (JWT) para la autenticación y autorización de las diferentes peticiones que se puedan realizar de una forma segura. Finalmente, como resultado de esta investigación se procederá al desarrollo una API REST (REpresentational StateTransfer) junto con JWT para así consolidar los conocimientos adquiridos en la etapa de investigación, además de aplicar los conocimientos obtenidos en las diferentes asignaturas del máster.

Por otro lado, podemos observar que las API REST hoy en día están en el punto de mira por parte de los ciberdelincuentes y centrarán sus esfuerzos en tratar de conseguir el acceso a estos tokens e intentar explotar alguna vulnerabilidad o simplemente conseguir una mala configuración para ganar acceso a los recursos no autorizados.

Uno de los objetivos que persigue este TFM es crear configuraciones robustas a la hora de firmar los tokens con diferentes algoritmos y claims que ayudarán a incrementar la seguridad a la hora de enviar los tokens por la red. Junto con el estudio de la identificación de los posibles riesgos que pueda tener una API REST en una instalación incorrecta o incompleta. Para conseguir estos objetivos se irán aplicando diferentes capas de seguridad a nuestra API en cuanto a arquitectura, así como también en la codificación de los diferentes endpoints para la API REST.

Así pues, como resultado final para este TFM nos permitirá comprender las diferentes opciones de configuración que existen a la hora de implementar una API REST junto con una arquitectura robusta antes de ser publicada a internet.

1.2. Objetivos del Trabajo

Objetivos de Investigación

- Investigar las características de las API REST para el estilo RESTful. Utilizando los verbos que nos proporciona el protocolo HTTP (POST, GET, PUT, DELETE)
- Investigar y estudiar el funcionamiento de los JSON Web Tokens (JWT)
- Investigar y analizar los potenciales fallos de seguridad en JWT
- Investigar y estudiar la utilización de Kong como API Gateway y Konga como dashboard para Kong

Objetivos de implantación

- Detallar la arquitectura técnica necesaria para implementar JWT en el entorno definido y detallar técnicamente cada uno de sus componentes.
- Instalación de un servidor Linux con Node.js, PM2, módulos para JWT y MongoDB.
- Implementar JWT en la codificación de la API RESTful
- Aprender a utilizar los diferentes claims que define el RFC de JWT.
- Codificación de los diferentes endpoints de la API con Node.js y utilizando MongoDB como base de datos NoSQL.
- Instalación de Kong (API Gateway) y Konga (dashboard) para la gestión de la API
- Puesta en marcha del producto final y la realización de pruebas para diferentes roles/usuarios que podrán utilizar la API diseñada.

Como se puede observar en la definición de los objetivos, el TFM tiene una componente de búsqueda de información sobre el funcionamiento de las API REST con JWT e investigación de los posibles fallos de seguridad que podrían ocurrir por una mala configuración. Finalmente se podrá observar el funcionamiento completo de una API REST para ser consumidas por los diferentes tipos de dispositivos que existe en el mercado. Así como las diferentes operaciones que se definirán en los endpoint.

1.3. Enfoque y método seguido

El enfoque que se llevará a cabo en este TFM es inicialmente un trabajo de investigación y estudio de los conceptos que intervienen, así como conocer el estado del arte de todos los componentes que interactúan en el funcionamiento de una API REST con JWT. En donde finalmente obtendremos una API REST con tokens robustos que puedan ser utilizados de forma segura. Además, con las herramientas que se implantarán nos ayudarán a comprender el funcionamiento de cada una de las partes de un JWT, así como las interacciones entre las diferentes herramientas que utilizaremos para el TFM.

El método que se utilizará para el desarrollo del TFM se va a diferenciar en tres partes:

- Una parte teórica de investigación y estudio sobre las herramientas que se utilizarán para el funcionamiento de los JWT. Por lo tanto, se realizarán comparativas de las diferentes librerías de seguridad que podríamos utilizar y seleccionar la que mejor se adapte al proyecto. Tras finalizar este apartado todos los resultados se materializarán en el documento PEC2.
- Y una segunda parte más práctica que se llevará a cabo con la instalación de las herramientas y respectivas configuraciones. Que se materializará en una tercera entrega como indica el plan de trabajo en la PEC3. En este apartado para la parte de codificación se utilizará una metodología ágil, ya que se necesita un producto (API) que sea funcional a medida que vayan pasando las semanas. Como podemos observar en la planificación del proyecto tenemos varias tareas en el backlog para desarrollar y junto con los sprints podremos ir observando el avance del producto entregable.
- Y finalmente una última entrega con la elaboración de la memoria final del TFM con su vídeo de demostración.

1.4. Planificación del Trabajo

Nombre de tarea	Comienzo	Fin	Duración
1 Planificación			
1.1 Definir problema a resolver	mié 16/09/20	jue 17/09/20	2 días
1.2 Definir objetivos	vie 18/09/20	lun 21/09/20	2 días
1.3 Definir propuesta metodología	mar 22/09/20	mar 22/09/20	1 día
1.4 Elaboración cronograma del TFM	mié 23/09/20	jue 24/09/20	2 días
1.5 Entrega PEC1			<i>Hito</i>
1.5.1 Redacción y entrega plan de trabajo	vie 25/09/20	mar 29/09/20	3 días
2 Investigación			
2.1 Estudio JWT, JWS, JWE	mié 30/09/20	jue 01/10/20	2 días
2.2 Estudio API REST	vie 02/10/20	lun 05/10/20	2 días
2.3 Estudio potenciales fallos seguridad en JWT	mar 06/10/20	lun 12/10/20	5 días
2.4 Diseñar arquitectura que utilizará el TFM	mar 13/10/20	mar 13/10/20	1 día
2.5 Investigar utilización Gateway Kong para API	vie 16/10/20	lun 19/10/20	2 días
2.6 Investigar utilización Konga Dashboard API	mar 20/10/20	mié 21/10/20	2 días
2.7 Entrega PEC2			<i>Hito</i>
2.7.1 Recopilar información y redacción PEC2	jue 22/10/20	mar 27/10/20	4 días
3. Implantación			
3.1 Implantar la arquitectura TFM			
3.1.1 Instalación servidor Linux	mié 28/10/20	vie 30/10/20	3 días
3.1.2 Instalación NodeJS + Módulos + PM2	mié 28/10/20	vie 30/10/20	3 días
3.1.3 Instalación MondoDB	mié 28/10/20	vie 30/10/20	3 días
3.2 Codificación API REST			
3.2.1 Codificación endpoints	lun 02/11/20	vie 06/11/20	5 días
3.2.2 Configurar seguridad para los endpoints	lun 09/11/20	mié 11/11/20	3 días
3.3 Administración API REST			
3.3.1 Instalación y configuración Kong	jue 12/11/20	mar 17/11/20	4 días
3.3.2 Instalación y configuración Konga	jue 12/11/20	mar 17/11/20	4 días
3.4 Entrega PEC3			<i>Hito</i>
3.4.1 Recopilar información y redacción PEC3	mié 18/11/20	mar 24/11/20	5 días
4. Presentación y defensa del TFM			
Entrega PEC4			<i>Hito</i>
4.1 Elaboración memoria final	mié 25/11/20	mar 29/12/20	25 días
Entrega PEC5			<i>Hito</i>
4.3 Elaboración vídeo presentación del TFM	mié 30/12/20	mar 05/01/21	5 días
Defensa TFM			<i>Hito</i>
4.6 Defensa TFM Videoconferencia	mié 06/01/21	vie 15/01/21	8 días

Tabla 1 - Planificación

1.5. Planificación temporal de las tareas y sus dependencias.

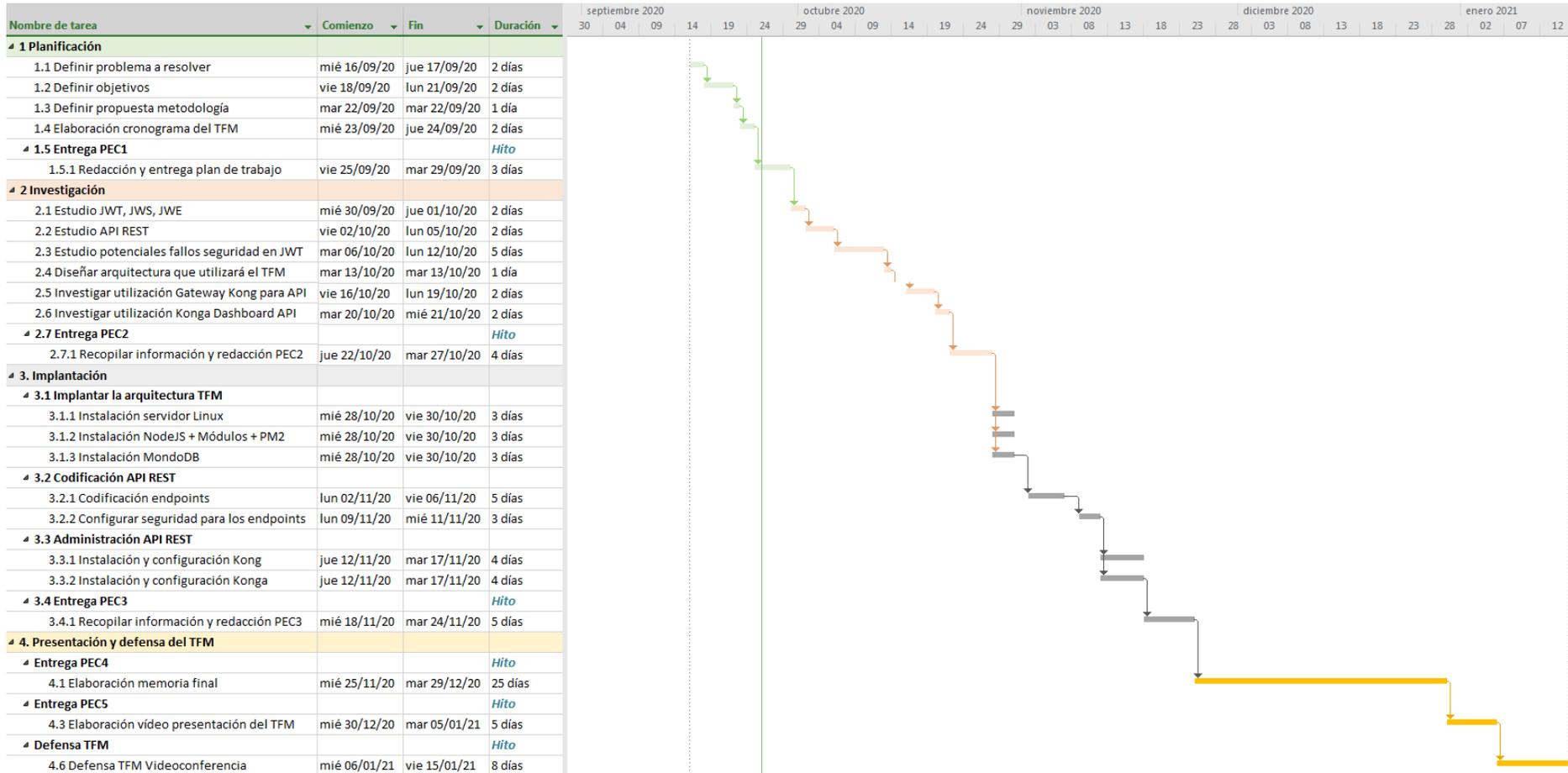


Tabla 2 - Diagrama de Gantt

1.6. Análisis de riesgo

R1. Problema con los tiempos de los objetivos

Definición del Riesgo:

Existe el riesgo de no poder finalizar adecuadamente el proyecto por el alcance de algunos de los objetivos planteados en plan de trabajo. Sobre todo, porque implica mucho tiempo de investigación y análisis de algunos de los objetivos. A continuación, podemos observar algunos objetivos que podrían aumentar los tiempos para el proyecto:

- La investigación y el análisis de los potenciales fallos de seguridad en JWT.
- Instalación y puesta en marcha del servidor Linux y sus componentes (JWT, JWS, JWE).
- Configuración e instalación de herramientas de tercero como Kong y Konga.

Mitigación del riesgo:

Como se puede observar son objetivos ambiciosos para el desarrollo del proyecto, por esto necesitamos que sean realistas. Como el tiempo apremia centraremos los esfuerzos en lo práctico durante la implantación no entrando en los detalles de cada uno de los módulos, configuraciones e instalaciones. Para el caso de los objetivos de investigación y análisis se intentará obtener los conceptos específicos que nos permitan desarrollar adecuadamente el objetivo.

R2. Problemas a la hora de reproducir los fallos de seguridad

Definición del riesgo:

Como uno de los objetivos importante dentro del proyecto es la investigación y análisis de los posibles fallos de seguridad en una implantación de JWT debido a una incorrecta o incompleta configuración. El riesgo principal que puede afectar a este objetivo es que se invierta mucho tiempo en intentar reproducir el fallo. Así como también intentar replicar las configuraciones e infraestructura con el consiguiente retraso en la planificación del proyecto.

Mitigación del riesgo:

Durante la fase de investigación de los posibles fallos que podamos encontrar se valorará la complejidad de reproducirlo. En caso contrario cuando se observe un incremento en los tiempos estipulados en el plan de trabajo dicho fallo o vulnerabilidad será documentada de forma teórica en la memoria del proyecto. En donde se indicará la forma de actuar ante dicha vulnerabilidad o fallo.

R3. Incompatibilidades entre herramientas.

Definición del riesgo:

Dentro de los objetivos del proyecto coexistirán varias herramientas con lo que conlleva una alta probabilidad que pueda existir algún problema de incompatibilidades de versiones entre herramientas. Riesgo que implicarían un alto coste en tiempo y recursos intentando encontrar el problema de la incompatibilidad.

Mitigación del riesgo:

Antes de instalar ninguna aplicación se tendrá que realizar un análisis de cada una de ellas. Comprobar matrices de compatibilidades entre versiones y ver que esfuerzo implicaría la instalación de la herramienta. En caso de tener un esfuerzo excesivo se intentará buscar alternativas menos costosas y que nos puedan dar el mismo resultado.

1.7. Estado del arte

En este apartado se presenta un conjunto de terminologías, herramientas y tecnologías que se tomarán como base para el desarrollo del TFM, aunque de forma simple sin entrar en detalles, ya que en los siguientes capítulos se ahondará en cada uno de ellos.

Inicialmente se presentarán las API REST que nos permitirá ir conectando conceptos como la arquitectura REST, el protocolo HTTP, endpoints, etc. También analizaremos las diferentes formas de incrementar la seguridad a los JWT.

1.7.1 Arquitectura API REST

El estilo de arquitectura REST fue diseñado por Thomas Fielding en su tesis doctoral ([Fielding 2000](#)), en donde detallada cada uno de los conceptos que veremos a continuación:

Representational State Transfer (REST) ([Wikipedia 2015](#)): se podría definir a un nivel más amplio como cualquier interfaz que utilice el protocolo [HTTP](#) para manipular u obtener datos. Apoyándose en cualquier formato [XML](#), [JSON](#), etc.

Dentro del estilo arquitectónico de REST define seis restricciones que se indican a continuación: *cliente-servidor*, *sin estado (stateless)*, *caché*, *interfaz uniforme*, *sistemas de capas*, *código bajo demanda*. Todas estas restricciones se explicarán con más detalle en el siguiente capítulo.

Protocolo HTTP: hoy en día la mayoría de API REST utilizan el protocolo HTTP para las peticiones como capa de transporte para sus comunicaciones. Podemos observar cómo API REST utiliza los verbos de operaciones GET (consultar), POST (añadir), PUT (editar) y DELETE (eliminar).

Application programming interface (API) ([Wikipedia 2017](#)): es una capa de abstracción que ofrece un conjunto de subrutinas, funciones y procedimientos para ser utilizado por otro software.

1.7.2 Estándares para la definición de Tokens

En este apartado se analizarán los diferentes estándares que existen para la definición de seguridad para los tokens que utilizaremos en las API REST. En los siguientes capítulos tras el análisis de estos estándares seleccionaremos el más adecuado para el desarrollo del TFM.

JSON Web Tokens (JWT) ([Wikipedia 2018](#)): estándar basado en el formato del tipo JSON que permite la creación de tokens permitiendo transmitir información como por ejemplo identidad, roles, privilegios.

JSON Web Signature (JWS) ([Wikipedia 2017](#)): estándar propuesto para la firma de un JWT que nos permitirá validar la información junto con una “palabra secreta” indicando que el mensaje enviado no ha sido alterado desde que se firmó.

JSON Web Encryption (JWE) ([Jones 2015](#)): nos permite encriptar el contenido que se envía previniendo que terceros tenga acceso a la información que se está transmitiendo.

Open Authorization 2.0 (OAuth) ([Wikipedia 2009](#)): es un estándar abierto para la autorización que nos permite obtener acceso limitado a sitios webs, aplicaciones móviles o aplicaciones de escritorio. Permitirá a un usuario otorgar acceso a una aplicación de terceros a los recursos protegidos del usuario, sin revelar sus credenciales o incluso su identidad.

1.7.3 Herramientas Gateway para API REST

Existen en el mercado una amplia gama de opciones para la gestión de las API. Herramientas que nos permiten funcionalidades como la monitorización del tráfico, aplicar políticas de seguridad, políticas de consumo, rendimiento, cuadro de mandos (dashboard), etc. Dentro del estudio de estas herramientas se ha optado por investigar las siguientes ([Pérez-Bermejo, Evgeniev 2019](#)):

Kong API Gateway¹: proyecto open-source que proporciona una capa de abstracción flexible que gestiona de forma segura la comunicación entre clientes y microservicios a través de su API. Kong se suele instalar con Konga² para que el manejo de las configuraciones de las API sea más cómodo y ágil para el usuario administrador.

Tyk API Gateway³: proyecto open-source que proporciona rapidez y escalabilidad, su plataforma permite la gestión de API de forma gráfica a través su panel web.

Apigee API Gateway⁴: proyecto adquirido por Google en 2016. No es de código abierto y se basa en Java empresarial. Inicialmente comenzaron como una aplicación XML / SOA, pero pasaron al espacio de administración de API.

1.8. Recursos necesarios y coste de realización del proyecto.

Para llevar a cabo el TFM, se hará uso de los siguientes materiales. También se especifica el coste por cada uno de los recursos necesarios.

Recurso	Detalle	Coste
Ordenador portátil	Instalación de máquina virtual para la API	500,00€
Conexión Fibra	Conexión a internet	50,00€
Otros	Gastos electricidad	30,00€
Tiempo Empleado	Horas 225 * 7€	1575,00€
	Total	2155,00€

Tabla 3 - Recursos y costes

¹ <https://konghq.com/kong/>

² <https://pantse1.github.io/konga/>

³ <https://tyk.io/>

⁴ <https://docs.apigee.com>

2. Investigación y Análisis

En este apartado se presentarán los conceptos relacionados para la implementación de una API REST con JWT.

Para el desarrollo del proyecto haremos alusión a dos conceptos importantes que nos permitirá un mayor entendimiento de la materia. Estos conceptos son *autenticación* y *autorización*.

Autenticación: proceso que implica la verificación de un individuo quién dice ser que es. Esto puede implicar verificar un nombre de usuario/contraseña o verificar que un token esté firmado y no esté caducado. La autenticación no denota que esta persona pueda acceder a un recurso en particular.

Autorización: es un proceso que determina si el individuo tiene derechos, permisos o privilegios para realizar algún tipo de acción sobre algún recurso.

Para este proyecto utilizaremos el mecanismo de *autenticación por conocimiento* ([Myers 2016](#)) en donde el usuario que utilizará la API REST utilizará información que solo él conoce.

En la siguiente figura se puede observar un flujo básico de autenticación y autorización mediante JWT.

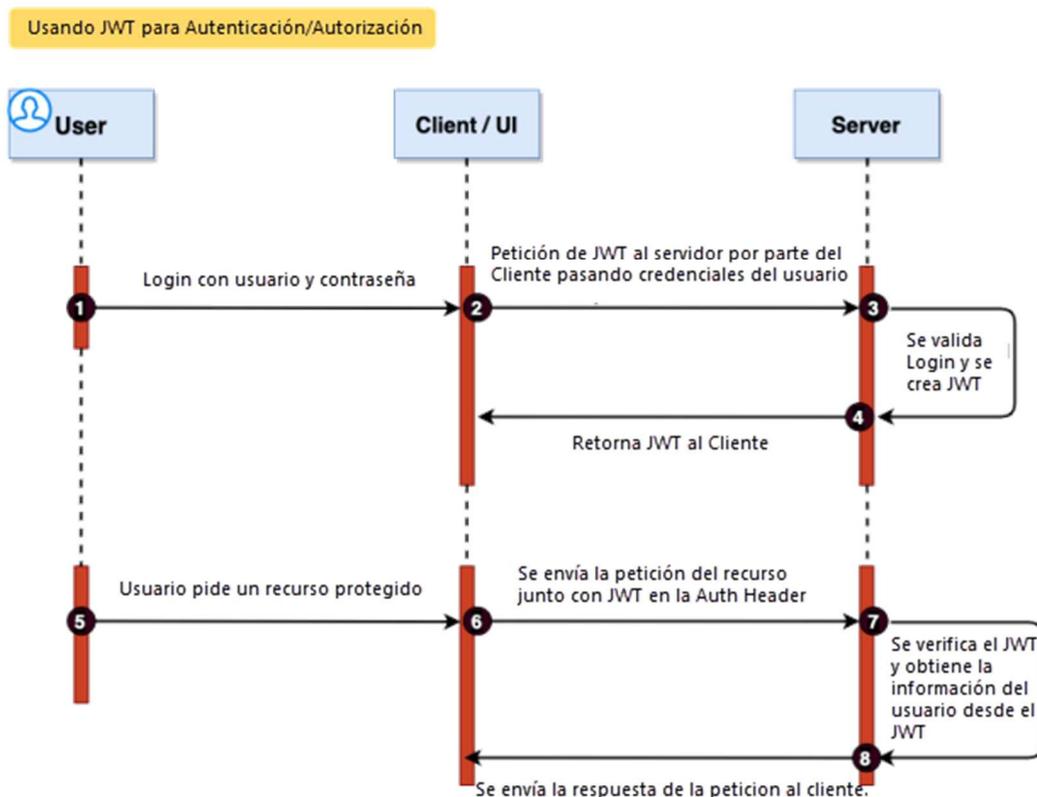


Figura 1 - JWT Autenticación/Autorización. ([Sunit Chatterjee, 2020](#))

2.1. Arquitectura API REST

Muchas veces las siglas, por ejemplo: “REST (Representational State Transfer)” no nos dan una idea clara del funcionamiento de las cosas. En el blog de Michael Miele ([Miele, 2018](#)) podemos encontrar un ejemplo adaptado a las páginas web que nos permitirá verlo desde otro punto de vista. Ejemplo:

*“Un sitio web está compuesto de recursos. Un recurso es cualquier elemento que valga la pena exponer. Por ejemplo, Sunshine Bakery puede definir un recurso de **croissant de trufa de chocolate**. Los clientes pueden acceder a ese recurso con esta [URL: www.sunshinebakery.com/croissant/chocolate-truffle](http://www.sunshinebakery.com/croissant/chocolate-truffle). Tras acceder a la URL nos devuelve una representación (**representation**) del recurso, por ejemplo, [croissant-chocolate-truffle.html](#). La representación pone la aplicación cliente en un estado (**state**). Si el cliente hace clic en un hipervínculo en la página, la nueva representación pone la aplicación del cliente en otro estado. Como resultado, la aplicación cliente cambia (**transfers**) el estado con cada representación de recurso”*

Con este ejemplo podemos ver una visión global de cómo trabaja REST en una página web o para otros sistemas.

Continuando con la utilización de la arquitectura REST la cual utiliza una serie de restricciones (constraints) durante la comunicación que realiza entre cliente y servidor. A continuación, veremos una pequeña descripción de estas restricciones. Roy Thomas Fielding ([Fielding, 2000](#)) define las siguientes restricciones:

- **Cliente-servidor:** se basa en el concepto de separación de preocupaciones ([SoC](#)). En donde existe una clara separación de preocupaciones entre cliente y servidor. Por ejemplo, el servidor se encarga del back-end (almacenamiento de datos) y el cliente maneja el front-end (interfaces de usuario).
- **Sin estado (stateless):** nos indica que el servidor no puede guardar los estados de los clientes más allá de la primera solicitud. El cliente al mantener la sesión cada vez que envíe una solicitud esta debe de incluir toda la información contextual requerida para poder ser tratada por el servidor.
- **Caché:** al servidor en cuestión tendrá la funcionalidad de poder cachear las peticiones en caso de reutilización de alguna petición por parte de algún cliente.
- **Interfaz uniforme** que está compuesta de las siguientes características:
 - o **Identificación de recursos:** se identificarán los recursos a través de una [URI](#). Por lo tanto, podemos decir que un recurso es cualquier cosa que pueda identificarse mediante una [URI](#).
 - o **Manipulación de recursos a través de representaciones:** al realizar una solicitud de un recurso, el servidor responde con una representación del recurso. Esta representación captura el estado actual del recurso en un formato que el cliente puede comprender y manipular. Estos metadatos tienen habitualmente un formato [JSON](#), [XML](#).

- **Mensajes autodescriptivos:** un mensaje debe tener suficiente información para que el servidor sepa cómo procesarlo (es decir, el tipo de solicitud, tipos de [MIME](#), etc.)
- **Hipermedia como motor del estado de la aplicación:** debemos utilizar enlaces (es decir, hipermedia) para navegar por la aplicación. El acceso a una API debería ser similar al acceso de una página web.
- **Sistemas de capas (Layered system):** el objetivo principal de esta restricción es que el cliente nunca llame directamente al servidor de aplicaciones sin haber pasado primero por un intermediario. Por ejemplo, se pueden añadir servidores proxy o Gateway para controlar las peticiones de los clientes.
- **Código bajo demanda:** esta restricción es opcional en donde el servidor puede devolver código ejecutable como por ejemplo los applets de java.

2.2. Análisis de los estándares para API REST

2.2.1 JSON Web Token

JSON Web Token es un estándar abierto definido en el RFC 7519⁵ que permite crear tokens de acceso para aplicaciones y poder transmitir la información de forma segura entre cliente y servidor. El servidor se encarga de firmar el token con una clave predefinida en donde el cliente y servidor podrán validar la autenticidad del token enviado. Los tokens presentan un formato del tipo JSON para el intercambio de datos.

Los JWT están codificados en [Base64url Encoding](#) con las técnicas que se especifica en su RFC 7519. El token lo podemos visualizar como se muestra en la figura 2, en donde cada sección está separada por un punto.

<p>eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ5b3VyLWFWaS1rZXkiLCJqdGkiOiIwLjQ3MzYyOTQ0NjIzNDU1NDIxIiwiaWF0IjoxNDQ3Mjc2MDk2LCJleHAiOjE0NDcyNzIxNTZ9.fQGPSV</p>	Header	<pre>{ "alg": "HS256", "typ": "JWT" }</pre>
<p>Jpc3MiOiJ5b3VyLWFWaS1rZXkiLCJqdGkiOiIwLjQ3MzYyOTQ0NjIzNDU1NDIxIiwiaWF0IjoxNDQ3Mjc2MDk2LCJleHAiOjE0NDcyNzIxNTZ9.fQGPSV</p>	Payload	<pre>{ "sub": "1234567890", "name": "Eduardo Salas", "iat": 1516239022 }</pre>
<p>Mjc2MDk2LCJleHAiOjE0NDcyNzIxNTZ9.fQGPSV 85QPhbNmuu86CIgZiluKBvZKd-NmzM6vo11DMsw</p> <p><i>Figura 2 - Token JWT separados por punto (elaboración propia, baso en la web de jwt.io [16])</i></p>	Signature	<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), TFM-U0C-2020) secret base64 encoded</pre> <p><i>Figura 3 - JWT formato JSON. (elaboración propia, baso en la web de jwt.io)</i></p>

⁵ <https://tools.ietf.org/html/rfc7519>

Header (cabecera): en esta sección se especifica el tipo de token y el algoritmo hash que utilizará para firmar el token. En la figura 3, se puede observar que el token está firmado utilizando HMAC-SHA256. (“alg”:“HS256”).

Payload (carga útil): en esta sección es donde se almacenará toda la información útil que el token utilizará. En la figura 3 se puede observar como el par “clave:valor” (claims) registra la información que se desea transmitir en el token.

En la definición de JWT en RFC 7519, podemos comprobar que existen tres tipos de claims ([Jones, 2015](#)):

- **Registered Claim Names:** estos claims no son de carácter obligatorio, pero si recomendables. Proporcionando un grupo de claims útiles e interoperables. Se definen de la siguiente forma:
 - “iss” (Issuer): identifica al emisor del token. Este valor distingue entre mayúsculas y minúsculas
 - “sub” (Subject): identifica al sujeto. Este valor distingue entre mayúsculas y minúsculas
 - “aud” (Audience): identifica la audiencia a la que está destinado el JWT
 - “exp” (Expiration Time): es la hora de expiración, a partir de la cual, el token no será válido.
 - “nbf” (Not Before): identifica el tiempo antes del cual no se acepta el token para su procesamiento.
 - “iat” (Issue At): identifica el momento en que se emitió el JWT.
 - “jti” (JWT ID): es el identificador único del token.
- **Public Claim Names:** estos claims pueden ser personalizados, pero teniendo en cuenta que el nombre seleccionado no provoque colisiones con los ya registrados de uso común. Se pueden representar por una URL o por un nombre. Y para evitar dichas colisiones, es recomendable consultarlos o registrarlos en el sitio web de la IANA JSON Web Token Registry⁶. En la figura 4 se muestran los claims públicos más comunes ya registrados.

Claim Name	Claim Description
name	Full name
given_name	Given name(s) or first name(s)
family_name	Surname(s) or last name(s)
middle_name	Middle name(s)
nickname	Casual name
preferred_username	Shorthand name by which the End-User wishes to be referred to
profile	Profile page URL
picture	Profile picture URL
website	Web page or blog URL
email	Preferred e-mail address

Figura 4 - Public claims ([John Bradley et al.,2015](#))

⁶ <https://www.iana.org/assignments/jwt/jwt.xhtml>

- **Private Claim Names:** estos claims solo lo conocen el productor y el consumidor de un JWT. Los nombres de los claims privados no son resistentes a las colisiones.

```

PAYLOAD: DATA
{
  Registered Claim
  {
    "iss": "srv-tfm.edu",
    "iat": 1516239022,
    "exp": 1517219032,
    "jti": "abje2899fdji33",
  }
  Public Claim
  {
    "name": "Eduardo Salas",
    "role": "admin",
    "email": "emsalag@uoc.edu",
  }
  Private Claim
  {
    "TFM" : "Aplicando seguridad a una API REST con JW
  }
}

```

Figura 5 - Tipos de claims (elaboración propia, basado en web jwt.io)

Signature (firma): en esta sección se permite al emisor del token firmarlo utilizando un hash definido en la sección `header` para así mantener la integridad del token. En la figura 3 se muestra como las secciones son codificadas en base64Url `header + payload + clave secreta` ("TFM-UOC-2020") para firmar el token.

Por realizar un símil a [POO](#) podemos indicar que JWT es una clase abstracta en donde JWS y JWE heredan las propiedades de JWT como se aprecia en la figura 6. Tenemos que entender que JWT por sí mismo es un concepto que explica cómo debemos de actuar a la hora de transferir los datos. Para tener una implementación de JWT echaremos mano de JWS, JWE o ambos a la vez.

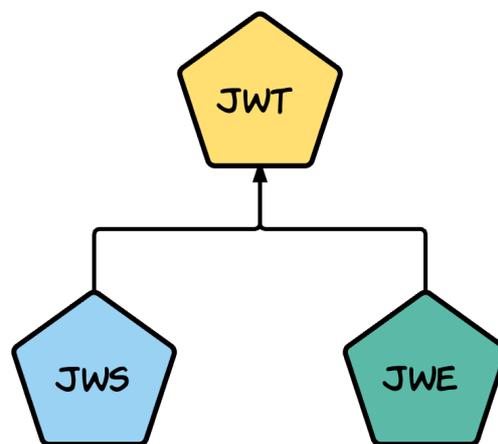


Figura 6 - JWT como clase abstracta. (Prabath Siriwardena, 2016)

Una vez repasado este símil que involucra a estos conceptos vamos a detallar las características principales para JWS y JWE.

2.2.2 JSON Web Signature

Como se ha visto en los apartados anteriores JWT por sí mismo no es capaz de implementar un token de forma completa. Aunque podemos representar un JWT sin el claim "alg" con un valor "none" en donde esto conllevaría a un grave fallo de seguridad. Por este motivo vamos a decir que JWT es un mecanismo para transferir la carga útil (payload) entre dos partes con garantía de integridad. En donde JWS en sus especificaciones define múltiples formas de firmar la carga útil y múltiples formas de serializar el contenido para transferirlo a través de la red.

A continuación, podemos ver un ejemplo de cómo se forma un JWS:

```
HMACHA256 (
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload), secret)
```

- El algoritmo que utilizará el token vendrá definido en el `header` (figura 3), por defecto JWT utiliza HMAC SHA256 para firmar el token, podemos encontrar una variedad de algoritmos para utilizar durante la firma.
- Se aplicará `base64URLEncode` tanto el `header` y `payload`
- Y finalmente tendremos un "secreto" que utilizará el servidor para validar el token.

2.2.3 JSON Web Encryption

Además de poder firmar un token también tenemos la posibilidad de poder cifrar un token gracias a la especificación de JWE en su RFC 7516⁷. El cual nos permitirá cifrar el contenido JSON añadiendo así confidencialidad al token.

Otras de las configuraciones que podemos encontrar es que con JWE se puede firmar e incluir en un JWS. Con esto, se obtiene cifrado y firma (obteniendo así confidencialidad, integridad, autenticación).

Para aplicar esta configuración primero deberíamos de firmar y luego encriptar tal y como indica el RFC 7516.

Hasta ahora se ha visto una configuración simple tanto para los estándares JWS y JWE.

2.2.4 JSON Object Signing and Encryption (JOSE)

JWT define el formato del token y utiliza especificaciones complementarias para manejar la firma y el cifrado de una forma más básica como se podía ver en los puntos anteriores. Ahora una forma de aunar todos estos estándares es mediante una colección de especificaciones que se conoce como JOSE (JavaScript Object Signing and Encryption) y tras consultar su RFC 7520⁸ podemos indicar que consta de los siguientes componentes:

- **JSON Web Signature** (JWS – RFC 7515): define el proceso para firmar digitalmente un JWT

⁷ <https://tools.ietf.org/html/rfc7516>

⁸ <https://tools.ietf.org/html/rfc7520>

- **JSON Web Encryption** (JWE – RFC 7516): define el proceso para cifrar un JWT
- **JSON Web Algorithm** (JWA – RFC 7518): define una lista de algoritmos para firmar digitalmente o cifrar.

El parámetro `alg` en el encabezado representa el algoritmo de la firma. Y tras consultar el RFC podemos ver la siguiente tabla en la documentación.

"alg" Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS1-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS1-v1_5 using SHA-384	Optional
RS512	RSASSA-PKCS1-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended+
ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional
none	No digital signature or MAC performed	Optional

Figura 7 - Algoritmos disponibles para el claim "alg" ([Jones, 2015](#))

- **JSON Web Key** (JWK – RFC 7517): define cómo se representa una clave criptográfica y conjuntos de claves.

Antes de pasar a visualizar algún ejemplo de las cabeceras que implementa la librería JOSE vamos a ver las dos formas de serializar las firmas ([Prabath Siriwardena, 2020](#)):

- La **serialización compacta** es una representación que está diseñada para ser utilizada en contexto web. Es decir, Base64 URL Safe separadas por el punto "."

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||
BASE64URL(JWS Payload) || '.' ||
BASE64URL(JWS Signature)
```

Figura 8 - Serialización Compacta. ([Nuwan Dias, Prabath Siriwardena, 2020](#))

- La **serialización JSON** representa las estructuras JWS como objetos JSON y, como resultado, permite que se incluyan varias firmas en la solicitud.

```

{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOiEzMDA4MTkzOD",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header": {
        "kid": "2014-06-29"
      },
      "signature": "cC4hiUPoj9Eetdgtv3hF80EGrhuB"
    },
    {
      "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header": {
        "kid": "e909097a-ce81-4036-9562-d21d2992db0d"
      },
      "signature": "DtEhU3ljbEg8L38VWafUAq0yKAM"
    }
  ]
}

```

Figura 9 - Serialización JSON. ([Nuwan Dias, Prabath Siriwardena, 2020](#))

En donde podemos observar en la figura 9, en la parte superior:

- **Payload:** Cadena codificada en Base64url del objeto de carga útil JWT real.
- **Signatures:** es un array de objetos JSON que llevan las firmas. Estos objetos se definen a continuación.

El array de firmas contiene:

- **Protected (protegido):** una cadena codificada en Base64url del encabezado JWS. Los claims contenidos en este encabezado están protegidos por la firma. Este encabezado es necesario solo si no hay encabezados desprotegidos. Si hay encabezados no protegidos, entonces este encabezado puede estar presente o no.
- **Header:** es un objeto JSON que contiene claims de encabezado. Este encabezado no está protegido por la firma. Si no hay un encabezado protegido, este elemento es obligatorio. Si hay un encabezado protegido, este elemento es opcional.
- **Signature:** una cadena codificada en Base64url de la firma JWS

Una vez visto las formas de serialización se verá algunos ejemplos de JWS y JWE:

Ejemplo estructura de un JWS con una serialización compacta



Figura 10 - JWS Serialización Compacta ([Prabath Siriwardena, 2016](#))

En la siguiente tabla se definen todos los parámetros que se pueden utilizar en el header para una representación compacta del tipo JWS

Claims	N. Completo	Descripción	Obligatorios
alg	algorithm	Especifica el algoritmo de firma. Cuando no es ninguna, significa que no se utiliza ninguna firma para proteger la integridad.	si
jku	JWK set URL	El URI de la clave pública correspondiente a la clave utilizada para la firma	no
jwk	json web key	Clave pública correspondiente a la clave utilizada para la firma	no
kid	key id	ID de la clave utilizada para firmar	no
typ	Type	Indica el tipo de medio de todos los JW, JOSE significa compacto, JOSE + JSON significa JSON	no
cty	Content Type	Media tipo de carga	no
crit	Critical	El receptor debe comprender y procesar los parámetros del encabezado de extensión enumerados en este campo; de lo contrario, los JW no son válidos y el campo tiene un formato array.	no
x5u	X.509 URL	URL que apunta a la cadena de certificados X.509 utilizada para firmar el JWT	no
x5c	X.509 Certificate Chain	Array JSON de la cadena de certificados de formato DER codificado en Base64 que se utiliza para firmar el JWT	no
x5t	X.509 Certificate SHA-1 Thumbprint	Huella digital SHA-1 del certificado X.509 utilizado para firmar el JWT	no
x5t#S256	X.509 Certificate SHA-256 Thumbprint	Huella digital SHA-256 del certificado X.509 utilizado para firmar el JWT	no

Tabla 4 - JWS Claims

```

HEADER: ALGORITHM & TOKEN TYPE

{
  "alg": "RS256",
  "typ": "JWS",
  "kid": "keyABC",
  "jku": "https://myjwk.com/jwks"
}

```

Figura 11 - Ejemplo JWS Header. (elaboración propia, basado en web jwt.io)

Ejemplo estructura de un JWE con una serialización compacta



Figura 12 - JWE Serialización Compacta. (Prabath Siriwardena, 2016)

Para el caso de una representación JWE utilizaríamos los siguientes parámetros:

- **JOSE Header (encabezado protegido):** un encabezado análogo al encabezado JWS. Contiene la información necesaria para procesar el JWT.

- **Encrypted Key (clave cifrada)**: clave utilizada para cifrar datos en el token. Esta clave está encriptada con una clave especificada por el usuario.
- **Initialization Vector (vector de inicialización)**: demandado por el algoritmo criptográfico
- **Ciphertext (datos cifrados)**: datos reales cifrados
- **Authentication tag (etiqueta de autenticación)**: datos generados por el algoritmo para la validación

```

HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "RSA1_5",
  "enc": "A256GCM",
  "iv": "__79_Pv6-fg",
  "x5t": "7noOPq-hJ1_hCnvWh6IeYI2w9Q0"
}

```

Figura 13 - Ejemplo JWE Header. (elaboración propia, basado en web jwt.io)

El siguiente ejemplo de encabezado JWE declara que:

- La clave cifrada (alg), cifra para el destinatario aplicando el algoritmo RSA1_5 (RSA-PKCS1_1.5) para crear la clave cifrada JWE
- El texto plano se cifra utilizando el algoritmo AES-256-GCM para crear el texto cifrado JWE
- Se puede observar en la figura 13 que se está utilizando el Vector de inicialización (iv) de 64 bits especificado con la codificación base64url
- La huella digital del certificado X.509 (x5t) que corresponde a la clave utilizada para cifrar el JWE tiene la codificación base64url 7noOPq-hJ1_hCnvWh6IeYI2w9Q0.

2.3. JWT fallos de seguridad y buenas prácticas

Durante la fase de investigación para este apartado se ha visto que la temática es muy amplia y difícil de abordar en un solo apartado. Entonces se han seleccionado los ataques o vulnerabilidades más representativas para este apartado.

Por lo tanto, muchos de estos ataques están relacionados con la implementación, más que con el diseño, de JSON Web Tokens. Lo que quiere decir que esto no los hace menos críticos. En resumen, una incorrecta implementación puede acarrear la creación de nuevas vulnerabilidades.

En los puntos anteriores se ha visto la estructura de un JWT con los diferentes claims que se pueden ir añadiendo y las diferentes configuraciones que se puede ir formando. En este apartado vamos a detallar los errores típicos que se pueden encontrar durante la creación de un JWT. También vamos a añadir algunas consideraciones de seguridad para su buen uso. No se pretende indagar a fondo en los detalles técnicos, ya que esto llevaría mucho tiempo desviando el objetivo principal del proyecto. Entonces según [Peyrott, 2018](#)

2.3.1 Ataque "alg: none"

Como se ha mencionado anteriormente, los JWT llevan dos objetos JSON con información importante, el encabezado y la carga útil. El encabezado incluye información sobre el algoritmo utilizado por el JWT para firmar o cifrar los datos que contiene.

En este tipo de ataque, un usuario malintencionado podría intentar usar un token sin firma.

Por ejemplo, durante la codificación de una función que se basa en el claim "alg" del encabezado. Es en este punto donde el ataque pueda llevarse a cabo. En el pasado, muchas bibliotecas confiaban en esta afirmación para elegir el algoritmo de verificación. Entonces el atacante modifica el valor del claim a "alg:none" lo que significa que no hay un algoritmo de verificación y el paso de verificación siempre tenía éxito.

Como se puede ver, este fue un ejemplo clásico de ataque en el pasado. Por esta razón, muchas bibliotecas informan hoy en día que los tokens "alg:none" no son válidos. Existen otras posibles mitigaciones para este tipo de ataque, pero la más importante es verificar siempre el algoritmo especificado en el encabezado antes de intentar verificar un token. ([Peyrott, 2018](#))

2.3.2 Claves débiles de HMAC

Los algoritmos HMAC se basan en un secreto compartido para producir y verificar firmas. Los secretos compartidos de HMAC, tal como los utilizan los JWT, están optimizados para la velocidad. Esto permite que muchas operaciones de firma/verificación se realicen de manera eficiente, pero facilita los ataques de fuerza bruta. Por lo tanto, la longitud del secreto compartido para HS256/384/512 es de suma importancia. De hecho, JSON Web Algorithms define la longitud mínima de la clave para que sea igual al tamaño en bits de la función hash utilizada junto con el algoritmo HMAC.

Una buena opción para mitigar este ataque es cambiar a RS256 u otros algoritmos de clave pública, que son mucho más robustos y flexibles.

Existen herramientas como jwt-cracker⁹ que permiten realizar fuerza bruta para obtener un token válido. ([Peyrott, 2018](#))

⁹ <https://www.npmjs.com/package/jwt-cracker>

2.3.3 Ataques de sustitución

Los ataques de sustitución son una clase de ataques en los que un atacante logra interceptar al menos dos tokens diferentes. El atacante usará uno o ambos de estos tokens para fines distintos al que estaban destinados.

Hay dos tipos de ataques de sustitución:

A un destinatario diferente: este ataque funciona capturando un token para un destinatario, le llamaremos destinatario “oficial” en donde este token es enviado a otro destinatario que “no es” el oficial. Entonces, para este caso va a existir un servidor de autorización que emite tokens para un servicio de terceros. El token de autorización es un JWT firmado en donde en la carga útil tenemos: “rol:admin”, “sub:fulanito”

Este token se puede utilizar contra una API para realizar operaciones autenticadas. Además, al menos cuando se trata de este servicio, el usuario `fulanito` tiene permisos de administrador (rol admin). Sin embargo, podemos observar que existe un problema con el token: no hay un destinatario previsto ni un emisor. ¿Qué pasaría en el caso que existiera otra API diferente, diferente del destinatario previsto para el que se emitió este token?, ¿se usara la firma como la única verificación de validez? Puede darse la casualidad de que el mismo usuario `fulanito` exista en la base de datos para ese servicio o API. Es cuando el atacante podría enviar este mismo token a ese otro servicio y obtener los privilegios como administrador.

Para evitar estos ataques, la validación de tokens debe basarse en claves o secretos únicos por servicio o en afirmaciones específicas. Por ejemplo, este token podría incluir una declaración de auditoría que especifique en el claims audiencia (`aud`) prevista. De esta forma, incluso si la firma es válida, el token no se puede utilizar en otros servicios que comparten el mismo secreto o clave de firma. (Peyrott, 2018)

Mismo destinatario (JWT cruzado): este ataque es similar al anterior, pero en lugar de depender de un token emitido para un destinatario diferente, en este caso, el destinatario es el mismo. Lo que cambia, en este caso, es que el atacante envía el token a un servicio diferente al previsto (dentro de la misma empresa o proveedor de servicios).

Imaginemos un token con la siguiente carga útil:

```
{
  "sub": "fulanito",
  "perm": "write",
  "aud": "mycompany/usuarios-bbdd",
  "iss": "mycompany"
}
```

Este token parece mucho más seguro. Tenemos un claim de emisor (`iss`), un claim de audiencia (`aud`) y un claim de permisos (`perm`). La API para la que se emitió este token verifica todas estas afirmaciones, incluso si la firma del token es válida. De esta manera, incluso si los atacantes logran tener en sus manos un token firmado con la misma clave privada o secreto, no pueden usarlo para operar en este servicio si no está destinado para él.

Sin embargo, `mycompany` tiene otros servicios públicos. Uno de estos servicios, es el servicio “`mycompany/elementos-bbdd`”, que ha sido actualizado recientemente por parte del equipo de desarrollo para verificar los claims junto con la firma del token. Sin embargo, durante las actualizaciones, no se validó correctamente el claim `aud`. En lugar de buscar una coincidencia exacta, decidieron verificar la presencia de la cadena `mycompany`. Resulta que el otro servicio, el hipotético servicio `mycompany/usuarios-`

bbdd, emite tokens que también pasan esta verificación. En otras palabras, los atacantes podrían utilizar el token destinado al servicio de base de datos de usuarios en su lugar para el token del servicio de base de datos de elementos. Es decir, que los atacantes tendrán permisos de escritura en la base de datos de `elementos-bbdd` cuando solo deberían tener permisos de escritura para la base de datos de `usuarios-bbdd`. ([Peyrott, 2018](#))

A continuación, se listan una serie de recomendaciones a la hora de trabajar con tokens:

- Utilizar siempre HTTPS para el envío de tokens
- Los tokens emitidos siempre deberían de ir firmados
- Utilizar algoritmos robustos
- Añadir a los tokens la fecha de expiración (`exp`) así como también un identificador único (`jti`)
- Añadir los `claims` de emisor (`iss`) y destinatarios (`aud`)
- Evitar incluir datos sensibles sin cifrar

Así como también seguir buenas prácticas para la validación del token

- Nunca aceptar tokens sin firma
- Realizar validaciones de los `claim` de la cabecera
- Realizar validaciones tanto emisor como destinatarios

2.4. Herramientas API Gateway y Dashboard

Como podíamos leer en la introducción del TFM las API son la fuerza impulsora detrás de muchas aplicaciones grandes y pequeñas. Ya sea que se haya publicado una API pública o realizando funciones de integraciones con diferentes servicios. Podemos observar que muchas de estas API están puestas en producción detrás de un Gateway presentando así una alta disponibilidad para sus microservicios. Podemos definirlo como un tipo de servidor proxy quien gestiona las peticiones y realiza funciones como autenticación, limitación de velocidad, enrutamiento apropiado para los microservicios, balanceo de carga entre múltiples servidores internos, entre otras cosas.

Y tras el estudio de la arquitectura de una API REST no podemos olvidarnos de aplicar una de las restricciones que veíamos en el apartado 2.1 que era “*interfaz uniforme*” en donde una de sus características era implantar un *sistema de capas*. A continuación, se nombrarán algunas características que benefician el uso de una API Gateway.

Desacoplamiento: en el caso que algún usuario se comunique directamente con muchos servicios separados, cambiar el nombre o mover esos servicios puede ser un desafío, ya que el usuario está acoplado a la arquitectura y organización subyacentes. Al utilizar un Gateway para la API permitirá enrutar en función de la ruta, el nombre de host, los encabezados y otra información clave, lo que le permite desacoplar los endpoints de la API.

Reducir las peticiones de ida y vuelta: es posible que algunos endpoints de la API necesite unir datos entre varios servicios. Con un Gateway se pueden realizar esta agregación para que el usuario no necesite un complicado encadenamiento de peticiones y así poder reducir la cantidad de peticiones de ida y vuelta que se realizan.

Seguridad: los Gateway proporcionan un servidor proxy centralizado para administrar la limitación de velocidad, la detección de bots, la autenticación, [CORS](#), entre otras cosas.

Preocupaciones transversales: los registros de logs, el almacenamiento en caché y otras inquietudes transversales pueden manejarse en dispositivos centralizados en lugar de implementarse en todos los microservicios.

En la siguiente figura se aprecia un sistema sin Gateway donde los clientes consultan de forma directa a los microservicios o API.

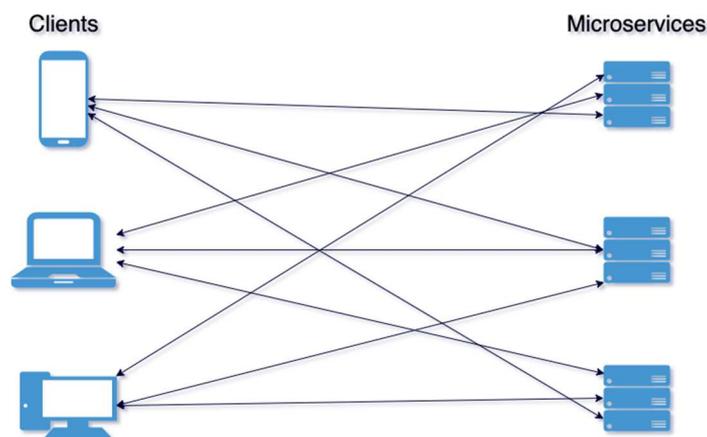


Figura 14 - Peticiones de forma directa. (Ivanov, 2020)

A partir de las siguientes figuras ya se puede observar un nuevo elemento en la arquitectura que son las API Gateway que actúa como un servidor proxy. Por lo tanto, el servidor Gateway se encargará de gestionar las diferentes peticiones que tiene de cada usuario.

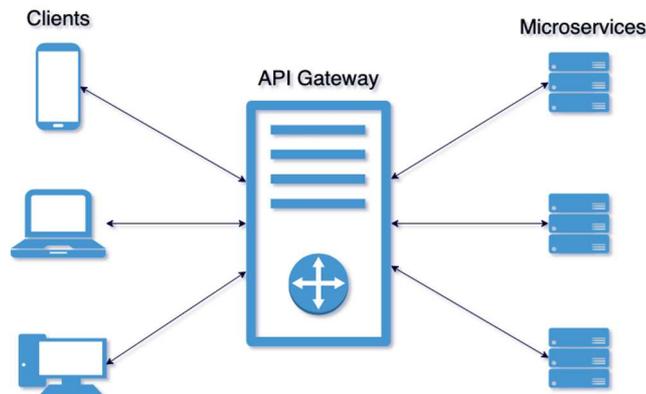


Figura 15 - API Gateway peticiones. (Ivanov, 2020)

Y finalmente se aprecia una configuración más avanzada con una API Gateway en donde existen servicios balanceados.

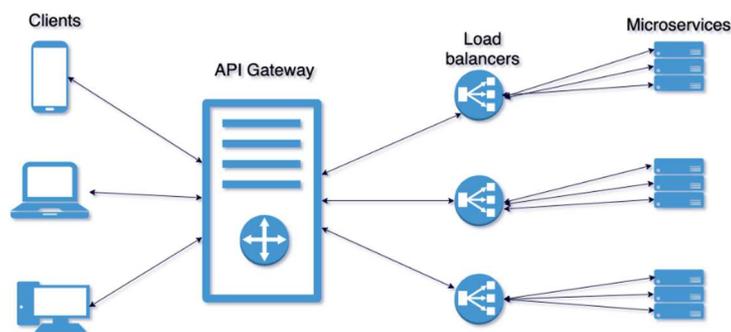


Figura 16 - API Gateway con balanceo de servicios. (Ivanov, 2020)

2.4.1 Kong API Gateway

Kong es un API Gateway de código abierto que se basa en [Nginx](#). Aunque Kong es de código abierto, KongHQ proporciona licencias de mantenimiento y soporte para grandes empresas. Si bien las características básicas se tienen con la versión de código abierto, ciertas características como la interfaz de usuario del administrador, la seguridad y el portal para desarrolladores están disponibles solo con una licencia de pago.

Existen varias formas de instalar Kong por ejemplo utilizando [Docker](#) para tener una implementación rápidamente. Kong tiene una complejidad moderada cuando se trata de implementación. Requiere ejecutar [Cassandra](#) o Postgres como gestores de bases de datos.

En su instalación por defecto, proporciona muchas características esperadas de API Management, como la creación de claves de API, enrutamiento a múltiples microservicios, etc.

Kong tiene el concepto de servicios, rutas y consumidores que brindan mucha flexibilidad al tratar con cientos de microservicios que componen una API.

2.4.3 Tyk.io API Gateway

Al igual que Kong, Tyk también es de código abierto, pero está bajo licencia MPL, que es menos permisiva que la licencia Apache 2.0 de Kong. Al mismo tiempo, el usuario empresarial de Tyk usa exactamente el mismo producto que un usuario de la versión community.

Tyk se basa en [Go](#), que, como lenguaje del sistema, está diseñado para un alto rendimiento y paralelismo.

Para su implementación Tyk ofrece una solución [SaaS](#) alojada en la nube o implementada en las instalaciones (on-premises). La versión de código abierto es relativamente simple de implementar y solo requiere [Redis](#), mientras que Kong requiere ejecutar clústeres tanto de [Cassandra](#) como de Postgres.

Tyk tiene características como administración de claves, cuotas, límite de tarifa, versión de API, control de acceso, pero no características de facturación integradas. Tyk tiene una API REST y un panel web para realizar tareas administrativas. Si bien tienen una lista de extensiones, Tyk no tiene una comunidad tan grande ni el centro de complementos que tiene Kong.

2.4.3 Apigee API Gateway

Apigee fue fundada en 2004 y adquirida por Google en 2016. No es de código abierto y se basa en Java empresarial. Inicialmente comenzaron como una aplicación XML / SOA, pero pasaron al espacio de administración de API. Están menos centrados en microservicios y API internas.

Debido a que Apigee tiene una arquitectura compleja de múltiples nodos, la implementación tiene un nivel de complejidad mucho mayor en relación con Gateway de código abierto. Apigee requiere ejecutar un mínimo de 9 nodos en las instalaciones e incluye ejecutar Cassandra, [Zookeeper](#) y Postgres, lo que obliga a que la implementación sea más compleja.

A diferencia de otros, Apigee admite la facturación integrada de extremo a extremo para monetizar sus API directamente. El portal de administración está construido sobre Drupal.

2.4.4 Elección de API Gateway

Para tener una comparativa y ver la tendencia de mercado sobre las diferentes API Gateways que existen en el mercado se ha realizado una consulta al *cuadrante mágico de Gartner* para tener un referente y poder tomar una decisión a la hora de implantar una API Gateway para el proyecto.

Se puede observar en la figura 17 que tanto Kong como Apigee están dentro del cuadrante de líderes los cuales los sitúa como posibles candidatos. Ambos tienen una gran cuota de mercado e innovadores. Sin embargo, tenemos a Tyk.io que se queda en el cuadrante de Visionario (visionaries) en donde Gartner las considera un producto innovador, pero con una baja cuota de mercado.

Magic Quadrant

Figure 1. Magic Quadrant for Full Life Cycle API Management



Figura 17 - Cuadrante mágico API Gateway. (Gartner, 2020)

Definitivamente la API Gateway Kong es nuestra apuesta para el TFM. Al ser de código abierto y al estar bien valorada por Gartner. Es una API Gateway moderna, en donde está diseñada para administrar múltiples microservicios, posibilidad de trabajar con diferentes plugins, capas de almacenamiento en caché y verificación de JWT.

Aunque Tyk.io tiene muchas de las funcionalidades que pueda prestar Kong se descarta porque necesitamos un producto puntero y que además esté reconocido como líder dentro de las API Gateway. Otro factor que ha inclinado la balanza hacia Kong es que ofrece mejores rendimientos a la hora de procesar peticiones.

La API Gateway Apigee queda fuera de nuestro alcance por ser de pago y por su elevada dificultad en la implantación. Recordemos que uno de los factores críticos definidos inicialmente era evitar tener un elevado tiempo durante la implantación.

2.4.5 Konga Dashboard

Tras la elección de Kong como API Gateway en donde la administración de los diferentes módulos se realiza a través de *Kong Admin API*¹⁰ utilizando herramientas de terceros como [Curl](#) o Postman. Así que se ha optado por una interfaz más amigable para trabajar de forma más cómoda. Tras investigar y navegar por la red se ha dado con

¹⁰ <https://docs.konghq.com/2.1.x/admin-api/>

un portal web Konga que nos permite interactuar de forma gráfica con las diferentes características de Kong.

Konga es una GUI multiusuario y de código abierto que facilita las tareas de administración de Kong. Puede administrar múltiples instancias de Kong. Además de poder integrarla con bases de datos MySQL, Postgres o MongoDB.

En la siguiente figura podemos observar la interfaz gráfica que presenta Konga.

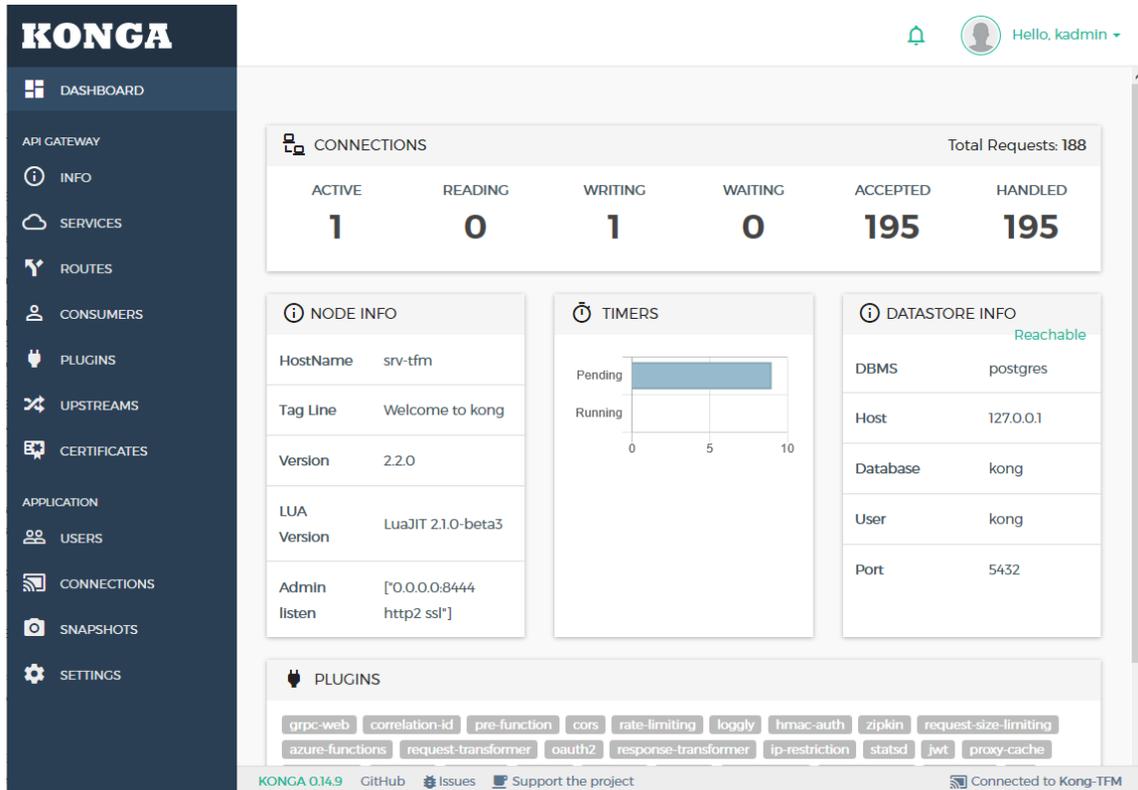


Figura 18 - Konga Interfaz web

2.5. Otras Herramientas

2.5.1 Postman¹¹

Postman es una herramienta que nos permite realizar peticiones a API RESTful creadas por otros o probar las que uno mismo ha creado. Ofrece una interfaz de usuario amigable con la que realizar solicitudes HTML, así pues, evitando de escribir un montón de código solo para probar la funcionalidad de una API.

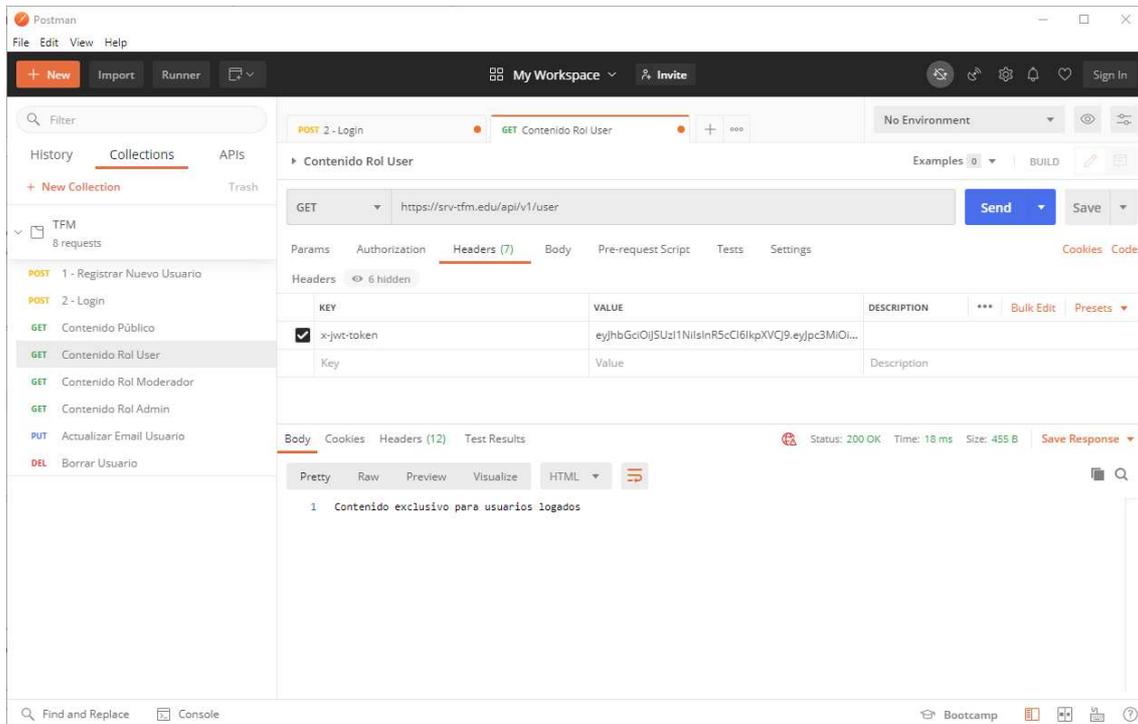


Figura 19 - Herramienta Postman

2.5.2 API REST Pública

La API REST del sitio web <https://regres.in/> nos permitirá realizar pruebas con una API real. Se ajusta a los principios REST y simula escenarios de aplicaciones reales, como probar un sistema de autenticación de usuarios.

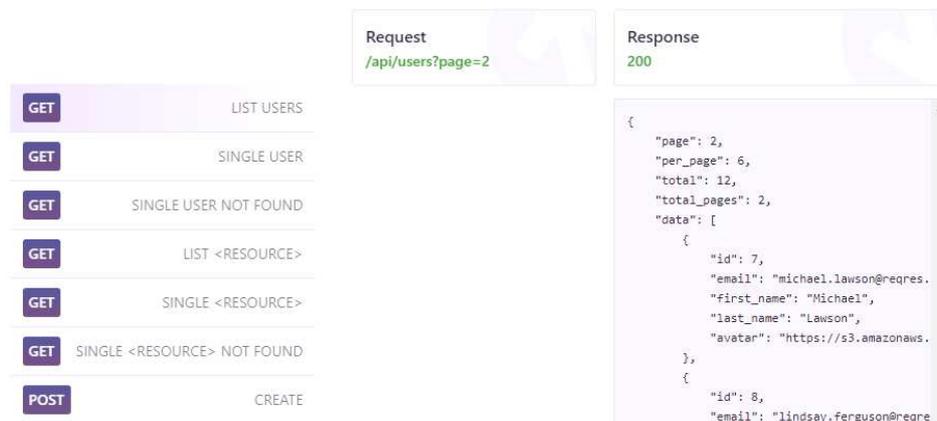


Figura 20 - API REST pública.

¹¹ <https://www.postman.com>

2.5.3 Swagger Editor¹²

Se utilizará esta herramienta para generar la documentación de la API. El editor proporciona ayuda visual para las especificaciones de [OpenAPI](#), maneja errores y proporciona autocompletados en tiempo real.

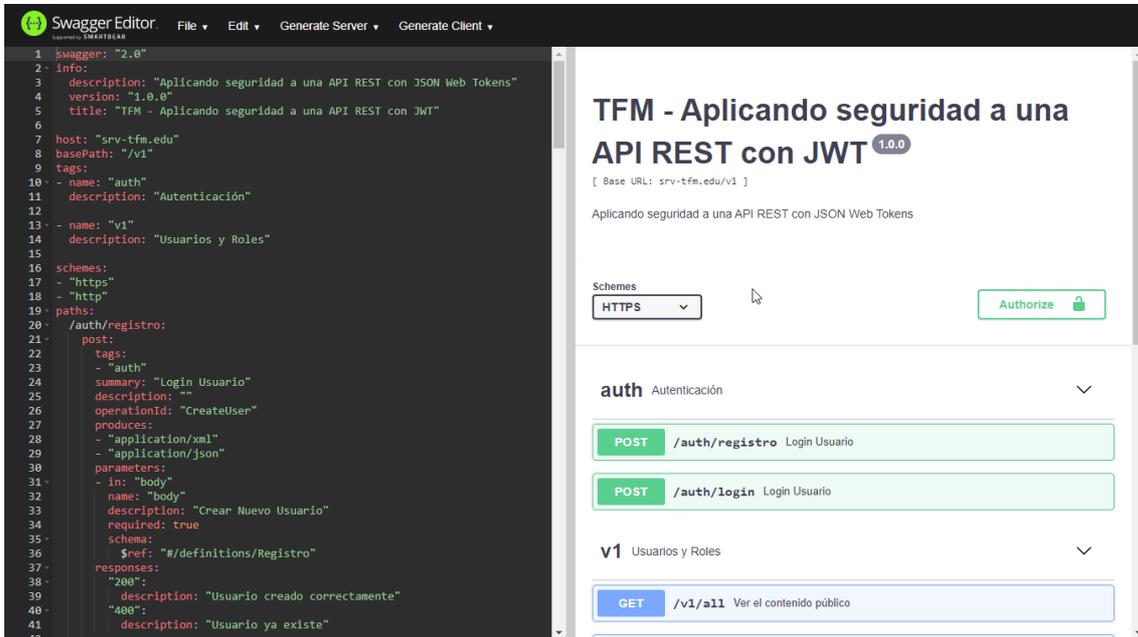


Figura 21 - Swagger Editor

2.5.4 RoboMongo¹³

Con esta herramienta podremos administrar de forma gráfica las bases de datos de mongoDB que instalaremos y utilizaremos durante el desarrollo del TFM.

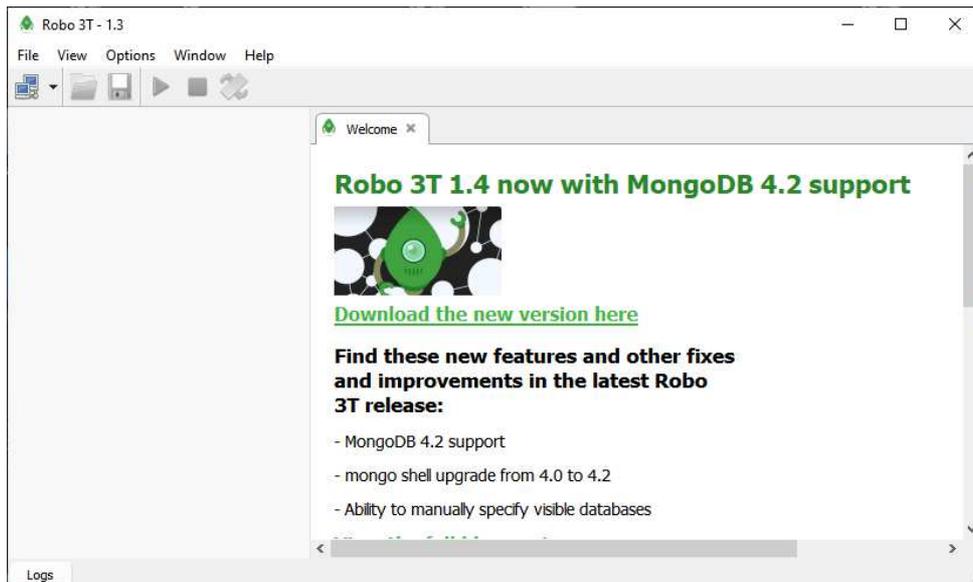


Figura 22 - Robomongo

¹² <https://editor.swagger.io/>

¹³ <https://robomongo.org/>

2.6. Arquitectura API REST

Para la implementación de la arquitectura API REST del proyecto, se realizará en dos fases. Una primera fase en donde los clientes conectarán directamente con la API REST. Como se puede observar en la figura 23.

En donde se muestra que a través de rutas definidas con ExpressJS y mediante el CORS Middleware se comprobará la solicitud HTTP que coincida con una ruta antes de llegar a la capa de seguridad. La capa de seguridad está formada por un middleware de Autenticación JWT que verificar el registro y un middleware de Autorización JWT que verificará los roles del usuario contra MongoDB.

Se enviará un mensaje de error como respuesta HTTP al cliente cuando el middleware arroje algún error. Los controladores interactúan con la base de datos MongoDB a través de la biblioteca Mongoose y envían una respuesta al cliente.

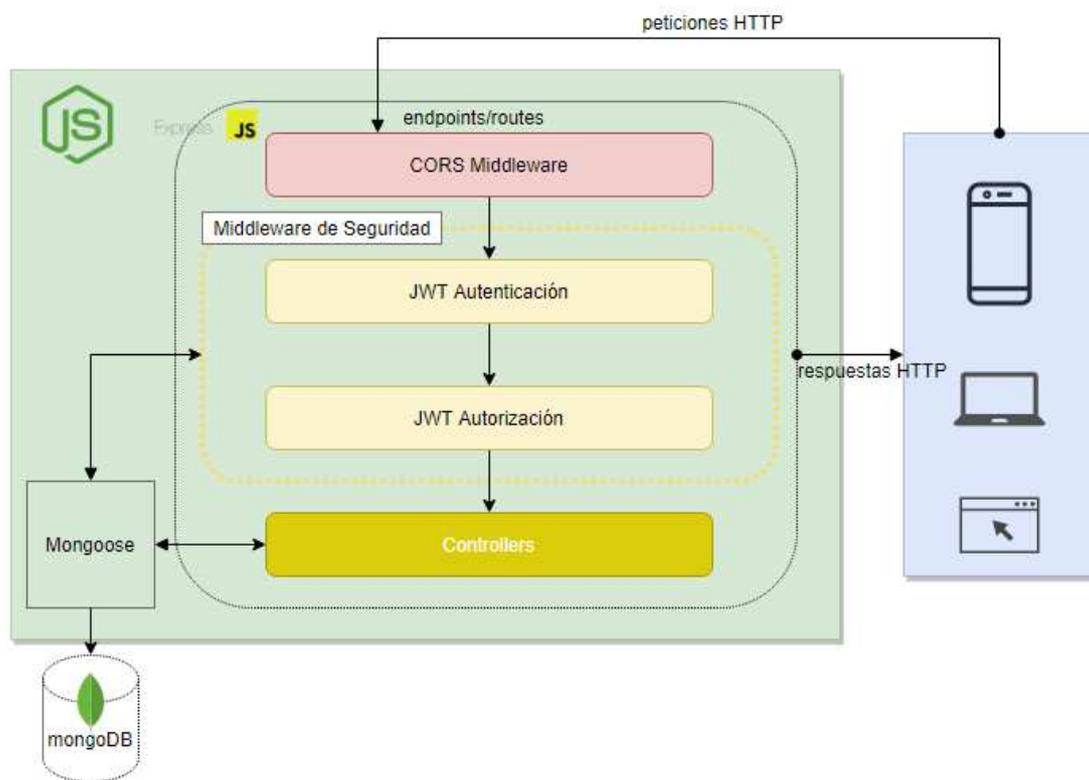


Figura 23 - Diseño arquitectura fase 1. (elaboración propia, basado en imagen de [Santiago, 2018](#))

Tras haber comprobado y realizado todas las pruebas con la API REST de la fase uno pasaremos a la siguiente fase que será implantar la API Gateway (Kong) como se muestra el esquema de la figura 24. En donde Kong nos permitirá abstraer la parte de API diseñada en Node.js y ExpressJS realizada durante la fase uno.

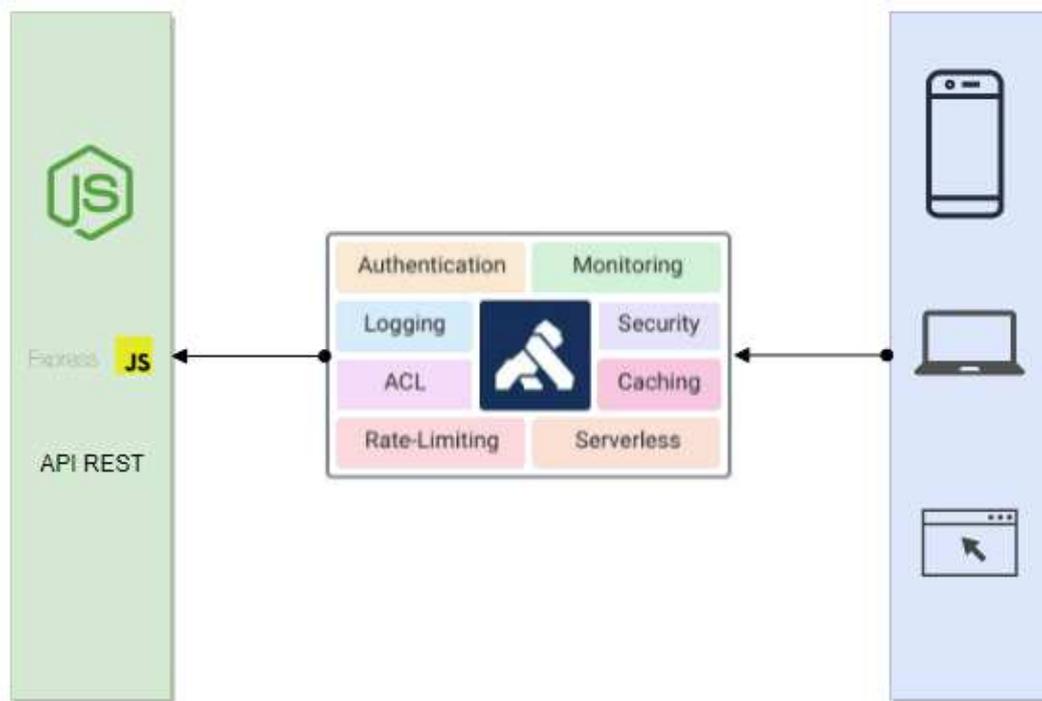


Figura 24 - Diseño arquitectura fase 2. (elaboración propia, basado en imagen konghq¹⁴)

¹⁴ <https://konghq.com/>

3. Implantación

3.1 Implantación infraestructura

En este apartado se describen todos los pasos necesarios para implantar la arquitectura representada en el apartado 2.6. Inicialmente instalaremos el software necesario para poder implementar una API REST con JWT para la fase uno. Y tras comprobar que todo está funcionando según lo previsto pasaremos a la fase dos de implantación que hace referencia a las herramientas de API Gateway (Kong) y Dashboard (Konga).

3.1.1 Instalación Servidor

Para comenzar con el despliegue de las herramientas necesarias para el desarrollo de la arquitectura necesitaremos poner en marcha un servidor Linux. (Debian 10.6)

Finalmente, tras la instalación del servidor con nombre de host "srv-tfm" llevaremos a cabo la instalación de las diferentes herramientas y software necesario para el TFM.

```
root@srv-tfm:~# uname -a
Linux srv-tfm 4.19.0-12-amd64 #1 SMP Debian 4.19.152-1 (2020-10-18) x86_64
GNU/Linux6.14.8
```

3.1.2 Node.js + Módulos

Una vez instalado el servidor Linux continuaremos con la instalación del software necesario para empezar la codificación de la API. A continuación, se lista el software necesario:

- Node.js
- Módulos Node.js: express, mongoose, body-parser, cors, jsonwebtoken, bcryptjs, pm2

Se muestra los comandos ejecutados en la terminal y versiones instaladas.

```
root@srv-tfm:~# apt-get install curl software-properties-common
root@srv-tfm:~# curl -sL https://deb.nodesource.com/setup_14.x | bash -
root@srv-tfm:~# apt-get install nodejs

tfm@srv-tfm:~$ node -v
v14.15.0

tfm@srv-tfm:~$ npm -v
6.14.8

tfm@srv-tfm:~$ npm i -g pm2

root@srv-tfm:/opt/TFM-JWT# npm install express mongoose body-parser cors
jsonwebtoken bcryptjs --save
```

Tras la instalación del software ejecutaremos el comando “npm init” para la creación de una aplicación con Node.js. Esto nos creará un fichero [package.json](#) en donde se guarda toda la configuración para esta nueva “App”. En la siguiente figura se muestra toda la parametrización que ha ido pidiendo.

```

root@srv-tfm:/opt/TFM-JWT# npm init
package name: (tfm-jwt)
version: (1.0.0)
description: Protegiendo una API REST que utiliza JSON Web Tokens
entry point: (index.js) server.js
test command:
git repository:
keywords: NodeJS, JWT, mongoDB, Authentication, Authorization
author: emsalas
license: (ISC)
About to write to /opt/TFM-JWT/package.json:
{
  "name": "tfm-jwt",
  "version": "1.0.0",
  "description": "Protegiendo una API REST que utiliza JSON Web Tokens",
  "main": "server.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [
    "NodeJS",
    "JWT",
    "mongoDB",
    "Authentication",
    "Authorization"
  ],
  "author": "emsalas",
  "license": "ISC"
}
Is this OK? (yes) yes

```

Figura 25 - Iniciando proyecto Node.js

En este punto tenemos una App vacía. A modo de ejemplo creamos un fichero `server.js` para poder levantar una instancia y poner la aplicación a la escucha en el puerto 8080. A continuación, un ejemplo sencillo de cómo iniciar una instancia con Node.js.

```

root@srv-tfm:/opt/TFM-JWT# cat server.js

var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('TFM UOC 2020 - Aplicando seguridad a una API REST con JWT\n');
}).listen(8080, "192.168.1.70");
console.log('Server running at http://srv-tfm.edu:8080/');

```

Con el comando `pm2` arrancamos la App (TFM_SegJWT)

```

root@srv-tfm:/opt/TFM-JWT# pm2 start server.js --name TFM_SegJWT --log-date-format "DD-MM HH:mm:ss.SSS"
[PM2] Starting /opt/TFM-JWT/server.js in fork_mode (1 instance)
[PM2] Done.

```

id	name	namespace	version	mode	pid	uptime	@	status	cpu	mem	user	watching
0	TFM_SegJWT	default	1.0.0	fork	8544	0s	0	online	0%	11.4mb	root	disabled

Figura 26 - Iniciando App

Y realizamos la comprobación en el navegador.

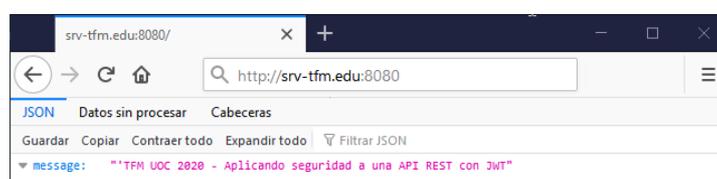


Figura 27 - Comprobando App

3.1.3 Instalación MongoDB

Otro software necesario para llevar a cabo el proyecto es la instalación de la NoSQL base de datos para ir almacenando las colecciones que utilizaremos para la API REST. Información que será consumida mediante los endpoints definidos en el apartado de codificación. Ejecutaremos los siguientes comandos en la terminal:

```
root@srv-tfm:# apt install dirmngr gnupg apt-transport-https software-properties-common ca-certificates curl

root@srv-tfm:# curl -fsSL https://www.mongodb.org/static/pgp/server-4.2.asc | apt-key add -

root@srv-tfm:# add-apt-repository 'deb https://repo.mongodb.org/apt/debian buster/mongodb-org/4.2 main'

root@srv-tfm:# apt install mongodb-org

root@srv-tfm:/# mongod --version
db version v4.2.10
```

Con la herramienta RoboMongo podemos conectar de forma fácil y gestionar las colecciones.

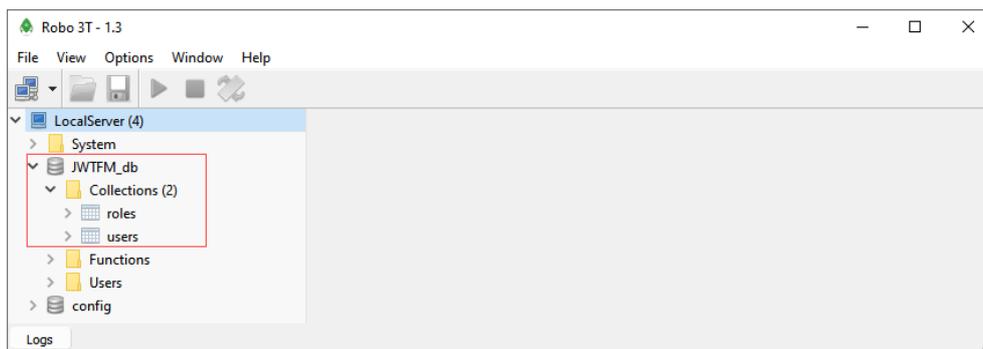


Figura 28 - Base de Datos de Colecciones

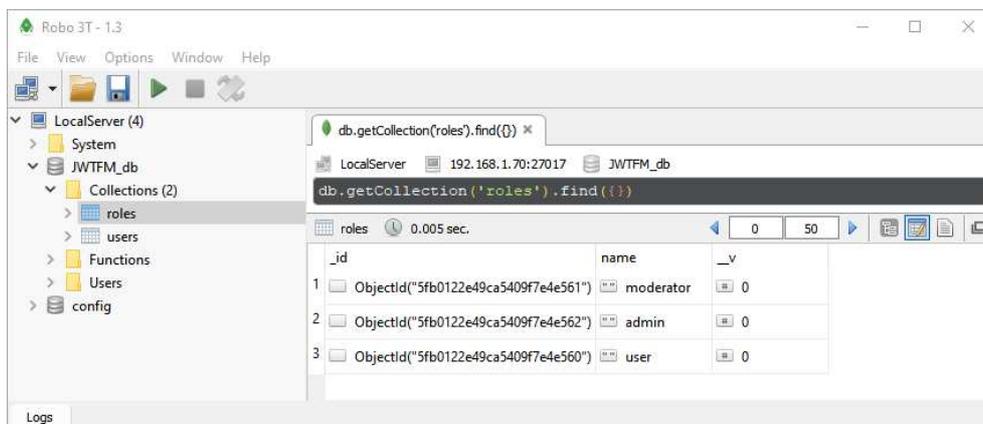


Figura 29 - Ver colecciones

3.2 Codificación API REST

Para esta primera entrega de la API REST se ha desarrollado un endpoint del tipo autenticación. Que mediante llamadas del tipo POST e indicando credenciales podrán ser validadas contra la MongoDB para comprobar que dicho usuario está autorizado a utilizar la API. En la siguiente figura se muestra la llamada al endpoint “/api/auth/login” junto con las credenciales del usuario. Como resultado el servidor devolverá un token.

El objetivo de esta primera entrega es comprobar el correcto funcionamiento de los tokens y el código realizado con Node.js y ExpressJS.

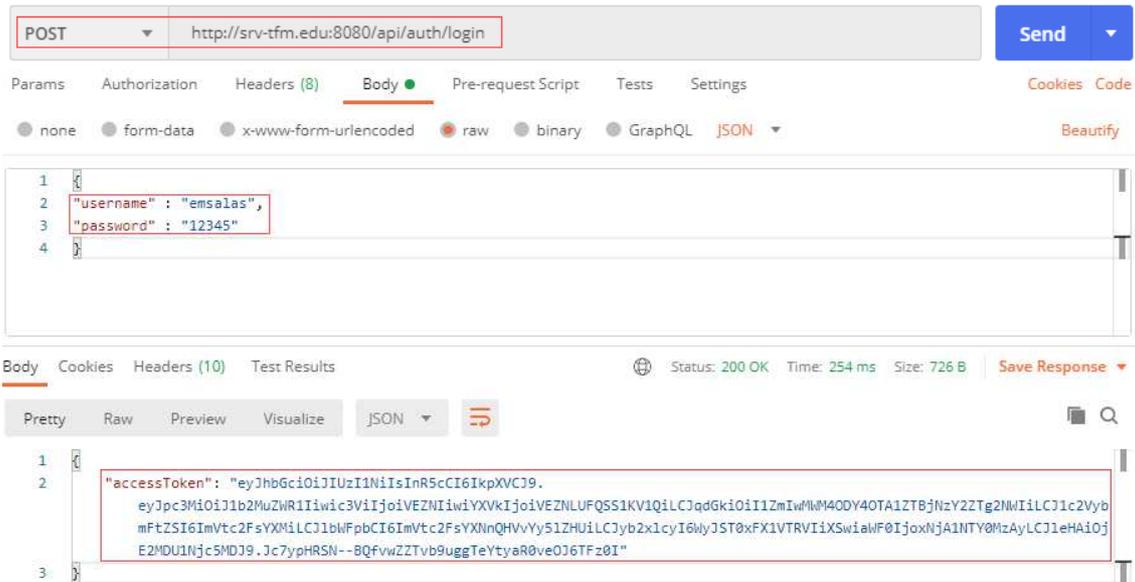
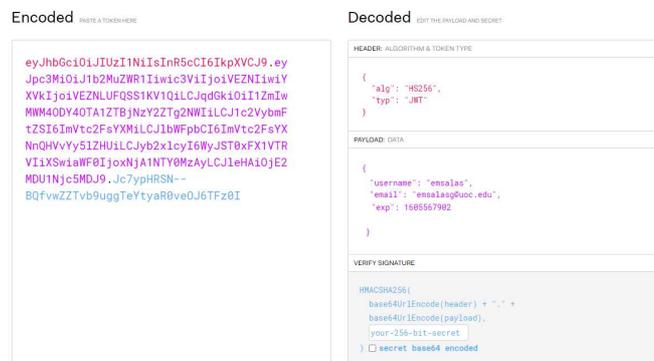


Figura 30 - Llamada endpoint login

Tras ejecutar el POST con la aplicación Postman al endpoint en cuestión obtendremos el token JWT. Tenemos la opción de poder validar y ver su contenido en la web de jwt.io¹⁵.

En la web podemos ver que existe un apartado de *debugger* en donde se observa que el token cumple con la estructura indicada en los apartados anteriores. El token lleva una definición básica en donde en el apartado de `payload` solo tiene cargados algunos `claim` del tipo público y privado. De forma opcional podemos añadir la firma (`secret`) para validar el token.



© Invalid Signature

Figura 31 - Web jwt.io debugger

¹⁵ <https://jwt.io/>

En el caso de añadir la palabra secreta en la web podemos ver como el token lo registra como “verified”



Figura 32 - Añadiendo palabra secreta

3.2.1 Codificación endpoints

Siguiendo con la codificación de la API REST para esta segunda entrega se van a desarrollar los diferentes endpoint para dejar una API REST funcional. Como resultado final tendremos una API REST en donde se definirán usuarios y roles. En donde cada usuario podrá visualizar contenidos según su rol. Esta API podría ser utilizada en diferentes ámbitos en donde se necesite diferenciar entre usuarios y roles.

En la siguiente tabla se puede observar los endpoints desarrollados para la API:

Métodos	Endpoint *	Token/Rol	Acción
POST	/api/auth/registro	-	Registrar a un nuevo usuario
POST	/api/auth/login	Token / Rol	Autenticación de usuario contra la API

Tabla 5 - Endpoints Autenticación

Métodos	Endpoint *	Token/Rol	Acción
GET	/api/v1/all	-	Obtener todas las entradas catalogadas como públicas
GET	/api/v1/user	Token	Visualizar contenido con el rol de usuario
GET	/api/v1/moder	Token / Rol Moderador	Visualizar contenido con el rol de moderador
GET	/api/v1/admin	Token / Rol Admin	Visualizar contenido con el rol de administrador
PUT	/api/v1/email	Token / Rol Moderador	Actualizar email de un usuario.
DELETE	/api/v1/user	Token / Rol Admin	Eliminar usuario

Tabla 6 - Endpoints Autorización

* En el link de los endpoints está vinculado al anexo [7.1 Código fuente](#)

3.2.2 Documentación API

Para documentar la API REST se ha utilizado la herramienta “Swagger Editor” que nos permite generar de forma más ágil la documentación. Podemos ver como se definen los parámetros y las diferentes respuestas con los códigos HTTP. Además, en el [Anexo 7.3](#) se puede ver la estructura completa del fichero de definición.

En las siguientes capturas se muestran ejemplos de la documentación de los endpoint creados.

TFM - Aplicando seguridad a una API REST con JWT

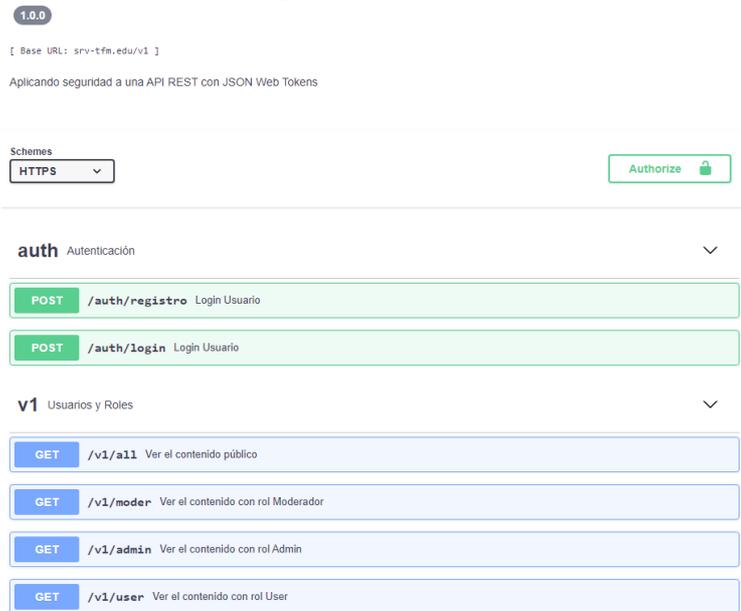


Figura 33 - Swagger Editor endpoint definidos

Con la ayuda del editor nos va autocompletando, facilitando la tarea de la creación de la documentación.

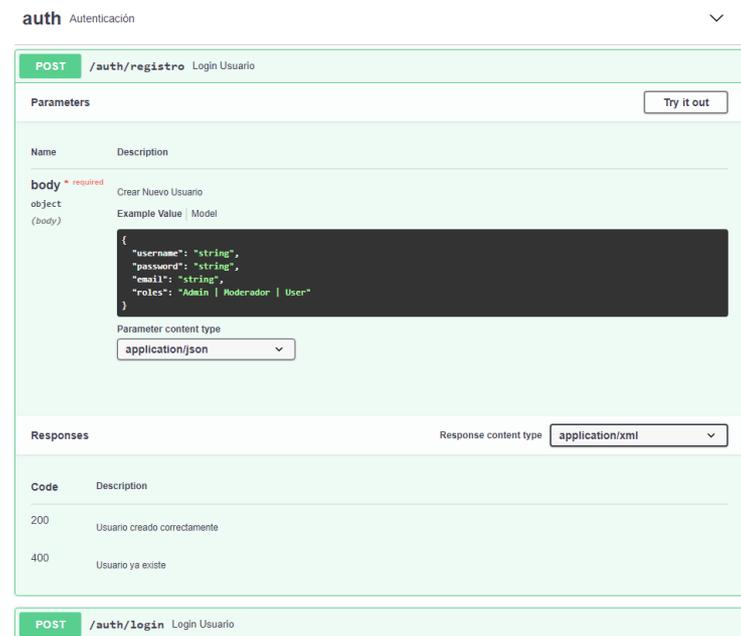


Figura 34 - Swagger Editor POST

En la siguiente figura se puede apreciar los parámetros que son necesarios. En este caso: JWT y el Body de la petición. Además de ver todos los estados que pueda devolver la petición.

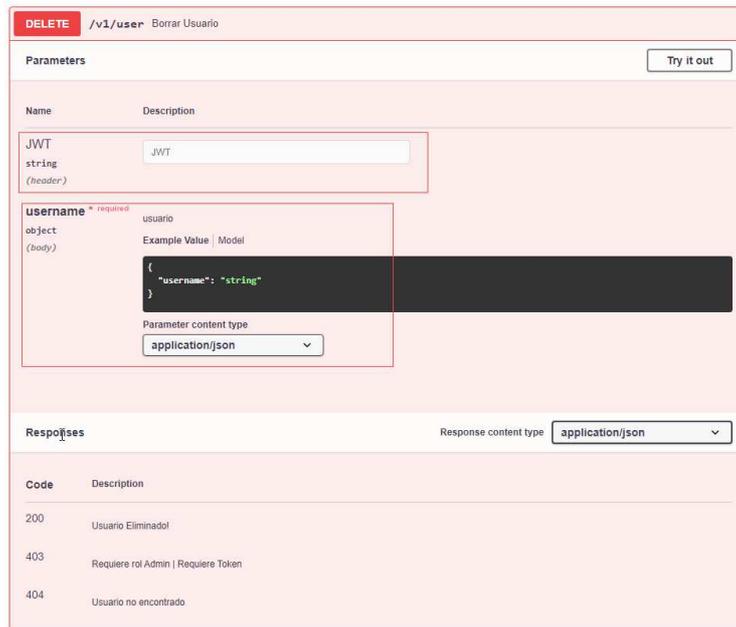


Figura 35 - Swagger Editor DELETE

Otro ejemplo cuando se pretende actualizar un email de un usuario.

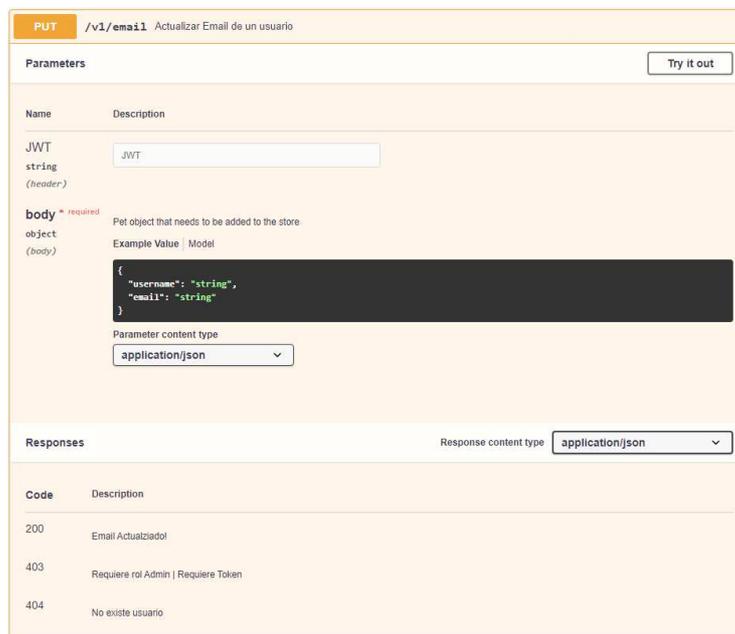


Figura 36 - Swagger Editor PUT

3.2.3 Ejemplos de endpoints

A continuación, veremos diferentes ejemplos de peticiones a la API REST con la aplicación de Postman.

Según la [tabla 5](#), como punto de partida crearemos un usuario que pueda utilizar la API. Para este ejemplo, este usuario tendrá un rol del tipo “user”.

✓ Tipo POST /api/auth/registro

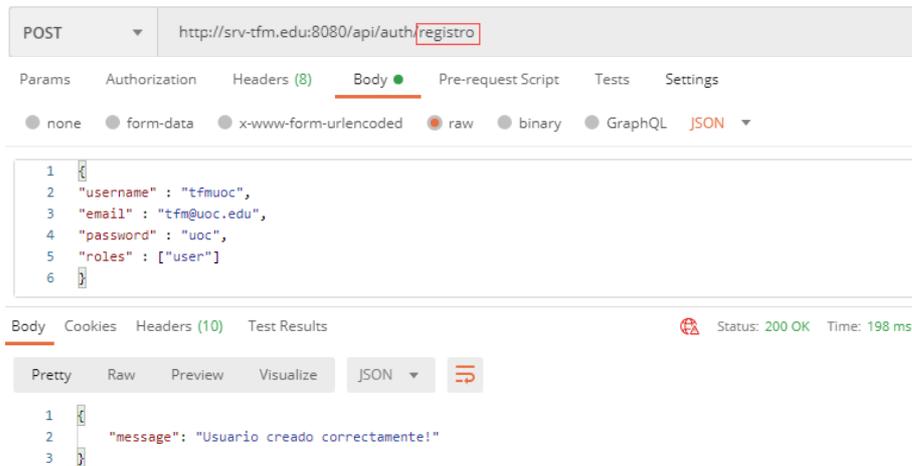


Figura 37 - Endpoint Registrar usuario

Comprobamos en la colección de Users que se ha añadido el nuevo usuario.

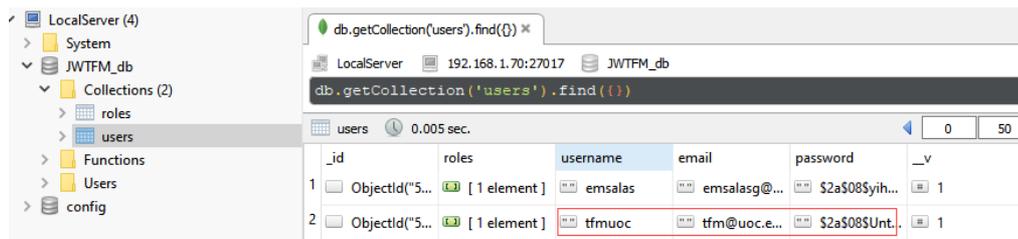


Figura 38 - MongoDB Insert nuevo registro

✓ Tipo POST /api/auth/login.

Tras crear el usuario podremos autenticarnos para obtener un Token. Como se puede ver en la figura 30. Y como se podía ver en la figura 31 podríamos visualizar el contenido visitando la web de jwt.io.

Continuando con la [tabla 6](#), cada usuario estará autorizado para realizar diferentes acciones en la API. A continuación, veremos algunos ejemplos:

- ✓ Tipo GET: consultar contenido público. Para este caso no hace falta obtener un token para consultar el contenido.

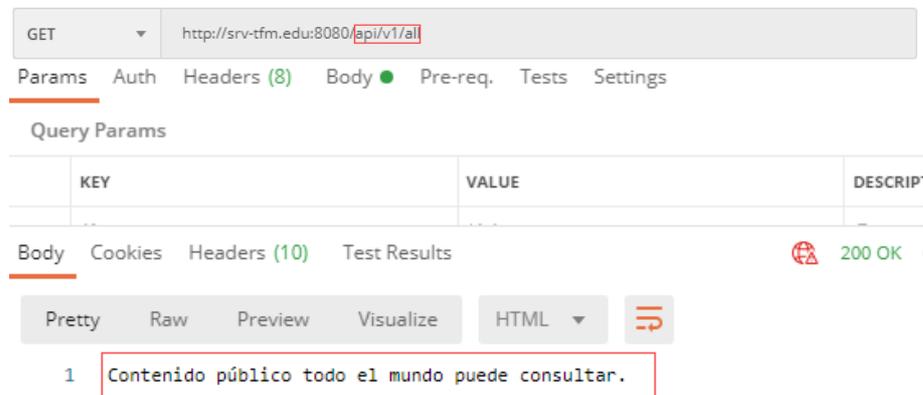


Figura 39 - Endpoint Consultar contenido público

- ✓ Tipo DELETE: eliminar usuario. Para este caso la primera comprobación que realizará la API REST es comprobar el rol del usuario para ejecutar la petición del tipo DELETE. Como se puede observar en la figura el usuario que está intentando realizar dicha acción no tiene permisos. Por lo tanto, la API le devuelve un mensaje informativo.

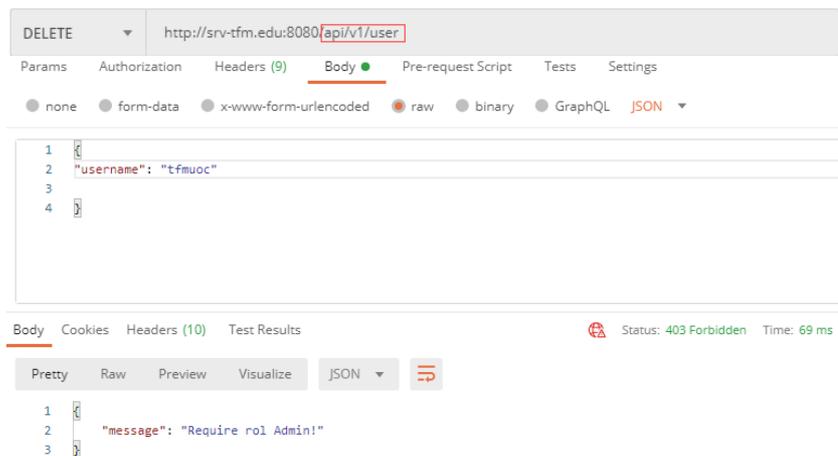


Figura 40 - Endpoint Borrar usuario

Si volvemos a pedir un token con un usuario con rol del tipo Admin ya podríamos borrar el usuario. Volvemos a lanzar la petición en esta ocasión con éxito.

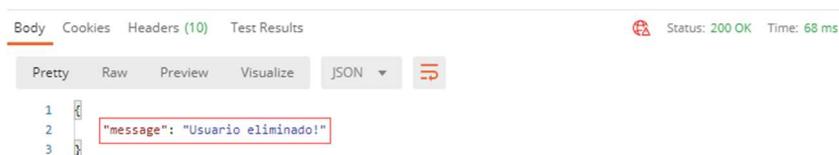


Figura 41 - Registro eliminado

Al comprobar la colección Users podemos ver que efectivamente se ha borrado el usuario.

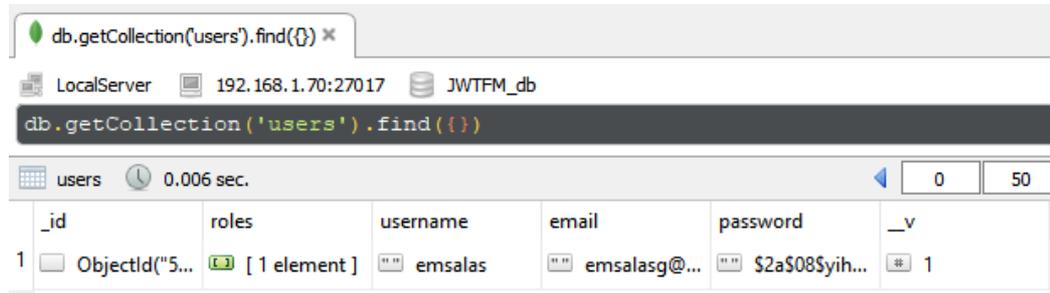


Figura 42 - MongoDB Usuario eliminado

- ✓ Tipo PUT: actualización de correo electrónico. Añadimos el token a la petición, ya que esta petición necesita un rol moderador (ver tabla 4).

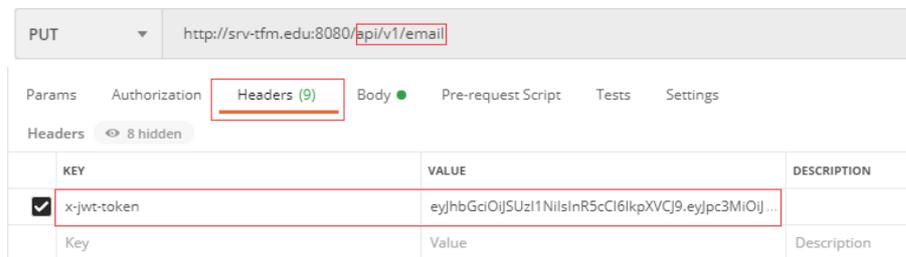


Figura 43 - Insertar Token a la petición

Podemos ver que efectivamente se ha actualizado el correo del usuario.

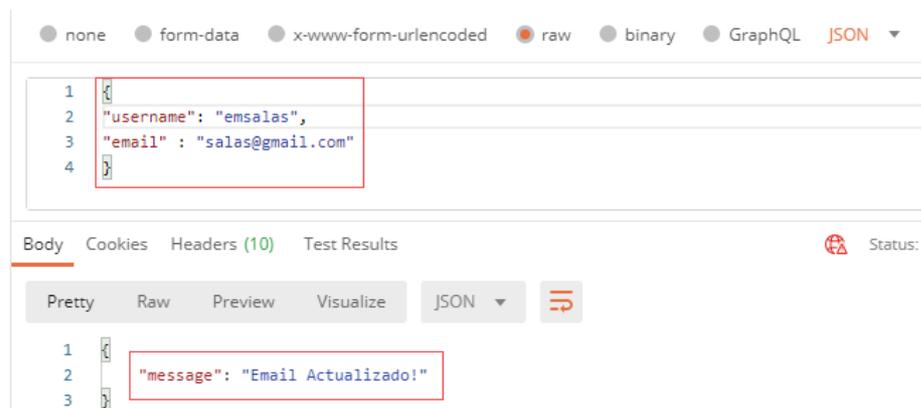


Figura 44 - Enviar petición tipo PUT

Comprobamos que se ha actualizado correctamente el correo.

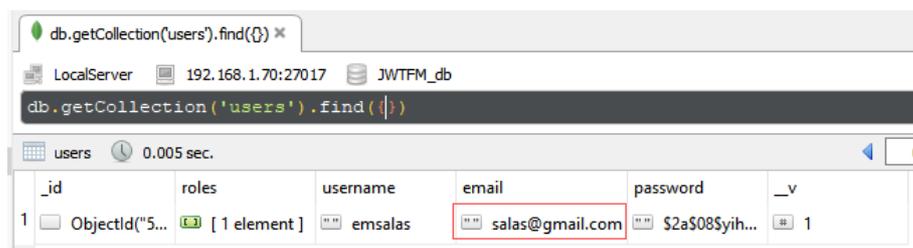


Figura 45 - MongoDB actualización colección

3.2.4 Añadir Seguridad a los endpoints

Como se vio en el apartado 2.3 seguiremos las recomendaciones para proteger la API REST y sus tokens. Recomendaciones que afectan al todo el conjunto de la infraestructura.

En primer lugar, vamos a añadir un certificado (autofirmado) para que los datos que envían a través de la red vayan cifrados. Crearemos un certificado con el siguiente comando en nuestro servidor

```
root@srv-tfm:/opt/TFM-KEY-EXPRESS# openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout selfsignedTFM.key -out selfsignedTFM.crt
```

Tras copiar los certificados al proyecto añadiremos la configuración al servidor. Podemos comprobar si realizamos una petición del tipo GET podemos ver que nos sale la advertencia de certificado autofirmado. A partir de este momento todas las peticiones irán cifradas.

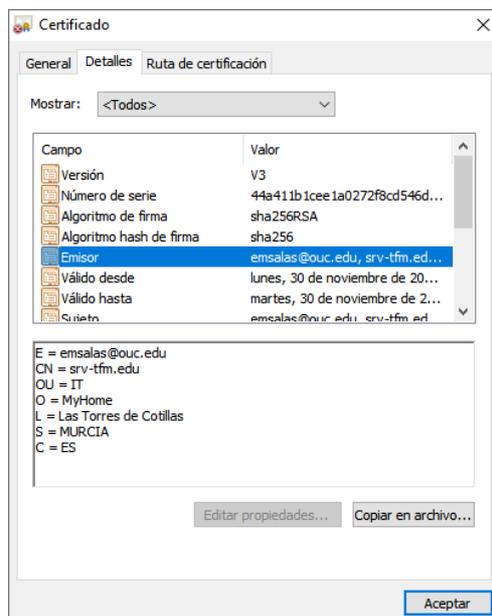
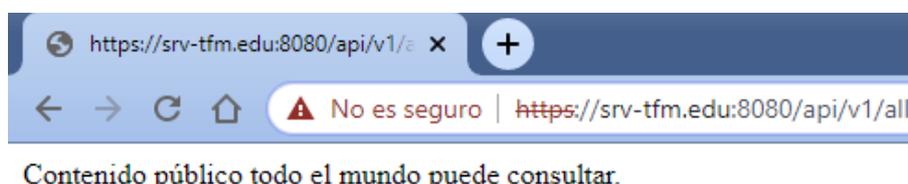


Figura 46 - API REST con HTTPS

Como segunda medida, vamos a crear el token JWT con un cifrado asimétrico generando la [clave pública](#) y [privada](#). En donde cada token será firmado por la clave privada que contiene el servidor. Como podíamos ver en la entrega inicial los tokens iban cifrados con una "palabra secreta" que estaba añadida directamente al código de la aplicación [[ver fichero de configuración](#)].

Entonces, el primer paso es crear las claves. Para esto se han ejecutado los siguientes comandos en la terminal

```
# Crear Key Privada RSA
root@srv-tfm:/opt/TFM-KEY-JWT# ssh-keygen -t rsa -b 4096 -m PEM -f
jwtRS256.key

# Crear Key Pública.
root@srv-tfm:/opt/TFM-KEY-JWT# openssl rsa -in jwtRS256.key -pubout -outform
PEM -out jwtRS256.pub
```

El siguiente paso será modificar los ficheros JavaScript con la nueva configuración. En el siguiente *snippet-code* podemos observar el cambio de puerto.

Fichero JS: TFM-JWT/[server.js](#)

```
56 // v1 HTTP Puerto 8080
57 // const PORT = process.env.PORT || 8080;
58 // v2 HTTPS Puerto 8443
59 const PORT = process.env.PORT || 8443;
```

Snippet-Code 1 - Cambio de Puerto APP

Otro cambio que se debe de realizar es añadir al código para que la API pueda leer la clave privada y la utilice para firmar los tokens.

Como se puede apreciar en el *snippet-code 2* se añade la nueva configuración añadiendo la clave privada que fue creada en los pasos anteriores.

Fichero JS: TFM-JWT/[auth.controller.js](#)

```
// VERSION 2 ADD KEY
const privateKey = fs.readFileSync('cert/jwtRS256.key');

// DEFINIR CLAIMS
var issClaim = "srv-tfm.edu";
var subClaim = "TFM";
var audClaim = "TFM-API-JWT";
var algClaim = "RS256";

// FIRMA v1 + ADD CLAIMS
//var token = jwt.sign({ iss:issClaim, sub:subClaim,
userRoles,}, config.secret, {
//expiresIn: '1h'
//});

// FIRMA v2 + ADD CLAIMS
var token = jwt.sign({
// REGISTERED CLAIM
iss:issClaim,
sub:subClaim,
aud:audClaim,
jti: user.id,
// PUBLIC CLAIM
username: user.username,
email: user.email,
// PRIVATE CLAIM
roles: userRoles
},
privateKey, {
algorithm: algClaim,
expiresIn: '1h'
});
```

Snippet-Code 2 - Clave Privada

Y el cambio en el código para la clave pública. Aplicamos los cambios como indica el *snippet-code 3*.

Fichero JS: TFM-JWT/[ajwt.mdlw.js](#)

```
const fs = require('fs');
const publicKey = fs.readFileSync('cert/jwtRS256.pub');

// COMPROBAR TOKEN
verifyToken = (req, res, next) => {
  // VERSION CON CLAVE SECRETA
  /*jwt.verify(token, config.secret, (err, decoded) => {
    if (err) {
      return res.status(401).send({ message: "Unauthorized!" });
    }
    req.userId = decoded.id;
    next();
  });*/
  // VERSION CON CLAVE PUBLICA
  try {
    const decoded = jwt.verify(token, publicKey, {
      algorithm: 'RS256',
    });
    req.userId = decoded.jti;
    next();
  } catch (error) {
    console.log ("error:" + error);
    return res.status(401).send('Unauthorized');
  }
};
```

Snippet-Code 3 - Clave Pública

Tras los cambios en el código se puede apreciar que tras realizar login el token que devuelve el servidor tiene un tamaño superior al de la configuración inicial que utilizaba un cifrado simétrico del tipo palabra secreta (ver [figura 29](#)).

The screenshot shows a REST client interface for a POST request to `https://srv-tfm.edu:8443/api/auth/login`. The request body is a JSON object with `username: "emasalas"` and `password: "12345"`. The response status is 200 OK, and the response body is a large JSON object containing an `accessToken` string. The token is a long alphanumeric string starting with `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.`

Figura 47 - Nuevo Token

Y si nos vamos a la web de [jwt.io](#) podemos ver como detecta el claim `alg:RS256`, en donde se habilita la casilla de `Public Key` para introducir la [clave pública](#) y validar el token.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9
.eyJpc3MiOiJzcnYtdGZtLmVkdSI6InN1YiI
6I1RGTsImF1ZCI6I1RGTs1BUektS1dUIiW
ianRpIjojNWZiMDFjODg2ODkwNWUwYzcyNmU
4NjViIiwidXN1cm5hbWUwOiJ1bXNhbGFzIiw
iZW1haWwiOiJzYWxhc0BnbWVpbC5jb20iLCJ
yb2xlcYI6WyJST0x0FETU10I10sIm1hdCI
6MTYwNjkzNzI3NSwiZXhwIjoxNjA2OTQwODc
1fQ.kqxtUe_E3hatB3bMUZuD2g1teVlw9YsX
UL6F-
WUZjrW9mFrBdgrtMEH4EF2aJm72V13Jx8x5g
GN9sb8TX2Ab5U6gkFV0_uWhrX6BZjHn4whzz
vprhGTAPKvKe035gtrCReIKXxq6KQzV7kgbM
V3YnF9c0tjHQhvtJ68CTf1DrkvfWSmZsSLpD
hZv8oo0jGrQajkSq_MN7EXuQh0IEGUSDNHLn
pYKJFWUpoADtj11PzrzqSdZa38en_9bIOFYA
cVyGjwR-
19z6hYIs05ncae51laxQNYrW6pPdb_QeUIcY
g5yCgiGFF0RE3IFbLaGCss0KDZadgZefv-
Tp1mF4AW3xm0MQorRoJjVtHw_93SKDaobEWI
Ewi0rk1VEFJSD0E5p-
sQvUa06F1PC0rLBMWjAmkKSeVrSzP1vZLz
gkzb0gJosyQD-
pHvNfUL7mI3KkBT2x7sKoEJXiaaQZT48RVC
FjtsHM85fdYP12s2F01DMFJr-
Uc5yNb9QfrMUCgy1FzKnitaRqGSnQWr50U_
oQ16B1E_hDtBnEs6CcioT0uEvMunwZ7PX03s
Q1i2os4Q666FeikKSE94unSxVzMHCK3h3-
PtMHF1HpgTZssoQ3eJ-
pmkX8_A2oUqypbN_WSeuNZh5tUMGQofkV-
```

Decoded EDIT THE PAYLOAD AND SECRET

```
HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "RS256",
  "typ": "JWT"
}

PAYLOAD: DATA
{
  "iss": "srv-tfm.edu",
  "sub": "TFM",
  "aud": "TFM-API-JWT",
  "jti": "5fb01c8868905e8c76e865b",
  "username": "emsalas",
  "email": "salas@gmail.com",
  "roles": [
    "ROLE_ADMIN"
  ],
  "iat": 1606937275,
  "exp": 1606940875
}

VERIFY SIGNATURE
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload)
)
Public Key or Certificate. Enter it in plain text only if you want to verify a token
Private Key. Enter it in plain text only if you want to generate a new token. The key never leaves your browser.
```

⊗ Invalid Signature

Figura 48 - Visualizar contenido Token en jwt.io

Tras introducir la clave pública podemos comprobar que el token lo da por válido.

```
xaz6NDh8kIi_jeuGC5zRorhro6HJBZ3e6hzopud
qG3f0wUFSKnMLuVJbi0ArVq2V-
zr4byNMMJeWw51KAnI5Bg1-
aQC5LKUFMt7ZuPWI37idwCHMySbCGaJ6BZyT8fv
hSLnChwxfXF0ZNSHrQ1YQJwHqgD1Yg0EkxKU8qcD
E1llwdLnwQEatUxAdYhJfFbLLCqZzh014LFEqROa
CeGegNmxtpG5xmEENDFJlcWo0tSH40mgAZ7f2gi
keV1P00zyYnDF-
HsT77iqEDrh3RctxTHAFVW9MPedouH7ISPVV11j
HFkFSGsqvSeEpK4_YbH2xNe4r0zpVbthB26c_aE
ZNYsHeFC0Zgp5xdPr rZsAw900rRWSyFfw8oYsPH
U1rDJQnB57cYpk3zLc9Bf1SAoaQx4hGE1VfH4k
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload)
)
-----BEGIN PUBLIC KEY-----
MIICjANBghkqhkiG9w0BAQEFAAOC
AgBAMIICCGKCAgEA94ogBpx7P0hQ
Tg901x11
Private Key. Enter it in plain text only if you want to generate a new token. The key never leaves your browser.
```

⊙ Signature Verified

Figura 49 - Validando Token

```
PAYLOAD: DATA
{
  "iss": "srv-tfm.edu",
  "sub": "TFM",
  "aud": "TFM-API-JWT",
  "jti": "5fb01c8868905e8c76e865b",
  "username": "emsalas",
  "email": "salas@gmail.com",
  "roles": [
    "ROLE_ADMIN"
  ],
  "iat": 1606937275,
  "exp": 1606940875
}
```

Si realizamos un zoom al Payload de la figura 48, se puede apreciar que se han utilizado los claims para crear un Token único (jti), para un sujeto (sub) y fin específico (aud). Añadiendo claims del tipo público como username o email y otro del tipo privado que hace referencia a roles. Además de tener su fecha de expiración (exp) y emisión (iat).

3.3 Administración API REST

En este apartado se centrará en la instalación del API Gateway Kong y su Dashboard Konga. Se indicarán los pasos a seguir para la instalación y la configuración de cada una de las herramientas.

3.3.1 Instalación y configuración API Gateway Kong

En primer lugar, vamos a instalar todas las dependencias para poder instalar Kong. Por lo tanto, deberemos de ejecutar los siguientes comandos en la terminal

```
root@srv-tfm:~# echo "deb https://kong.bintray.com/kong-deb
`lsb_release -sc` main" | tee -a /etc/apt/sources.list

root@srv-tfm:~# curl -o bintray.key
https://bintray.com/user/downloadSubjectPublicKey?username=bintray

root@srv-tfm:~# apt-key add bintray.key
root@srv-tfm:~# apt-get update
root@srv-tfm:~# apt-get install -y Kong
```

Para continuar con la instalación necesitaremos una base de datos Postgres. En donde se creará una base de datos para albergar los datos de Kong. Ejecutamos los siguientes comandos en la terminal

```
root@srv-tfm:~# su postgres
postgres@srv-tfm:$ psql
postgres=# CREATE USER kong; CREATE DATABASE kong OWNER kong;
postgres=# ALTER USER kong WITH PASSWORD 'kongTFM';
```

Una vez preparada la base de datos y la instalación procedemos a la configuración del fichero principal de Kong. Realizando una copia del fichero de ejemplo kong.conf.default. Por un fichero llamado [kong.default](#) en donde se ajustarán los parámetros necesarios. En este caso añadiremos el nombre de la base de datos, usuarios y contraseña creados en el paso anterior.

```
root@srv-tfm:~# cp /etc/kong/kong.conf.default /etc/kong/kong.conf
root@srv-tfm:~# nano /etc/kong/kong.conf
pg_user = kong
pg_password = kongTFM
pg_database = Kong
```

Con el fichero kong.conf ejecutaremos el script de inicio de Kong que aplicará toda la configuración en la base de datos creada. Y como último punto ejecutaremos el siguiente comando en la terminal para iniciar Kong.

```
root@srv-tfm:~# kong migrations bootstrap -c /etc/kong/kong.conf
root@srv-tfm:~# kong start -c /etc/kong/kong.conf
```

Tras la instalación, Kong registra dos servicios:

Servicio Proxy¹⁶: es donde Kong recibirá el tráfico de entrada de la API. Por defecto define los siguientes puertos:

- ✓ 8000 para proxying HTTP tráfico
- ✓ 8443 para proxying HTTPS tráfico

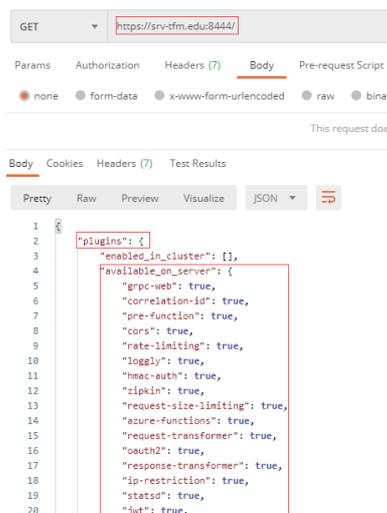
Servicio Admin API¹⁶: este será el puerto donde Kong publica su API de administración en donde se podrán realizar todas las operaciones. Para este caso define los siguientes puertos:

- ✓ 8001 para administrar la API de Kong vía HTTP
- ✓ 8444 para administrar la API de Kong vía HTTPS

Entonces, para este TFM se utilizarán los siguientes puertos:

- ✓ 443 para el Servicio Proxy
- ✓ 8444 para Servicio Admin API.

Finalmente, para comprobar que la configuración es la correcta procederemos a realizar una llamada al *servicio de Admin API* con la URL del servidor y el puerto 8444. Realizamos una llamada del tipo GET con la herramienta de Postman para comprobar que está el servicio iniciado. Como resultado se observa la configuración por defecto que tiene activa.



```
GET https://srv-tfm.edu:8444/

Body

This request does:

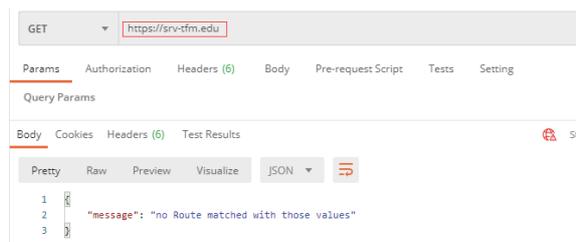
Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
{"plugins": {
  "enabled_in_cluster": [],
  "available_on_server": {
    "grpc-web": true,
    "correlation-id": true,
    "pre-function": true,
    "cors": true,
    "rate-limiting": true,
    "loggly": true,
    "hmac-auth": true,
    "zipkin": true,
    "request-size-limiting": true,
    "azure-functions": true,
    "request-transformer": true,
    "oauth2": true,
    "response-transformer": true,
    "ip-restriction": true,
    "statsd": true,
    "jwt": true,
  }
}}
```

Figura 50 - Kong Servicio Admin API

Y en caso de comprobar el *servicio de proxy* podemos ver que al no existir ninguna API registrada nos dará un mensaje indicando que no ha encontrado ninguna ruta.



```
GET https://srv-tfm.edu

Params Authorization Headers (6) Body Pre-request Script Tests Setting

Query Params

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON

1
2
3
{"message": "no Route matched with those values"}
```

Figura 51 - Kong Servicio Proxy

¹⁶ <https://docs.konghq.com/1.1.x/network/>

Hasta el momento la única forma de registrar API es utilizando su servicio de API Admin. Entonces con la ayuda de la herramienta de Postman podremos ir registrando API al entorno de Kong. Para comprobar que API tenemos registradas en Kong podemos utilizar la siguiente llamada para consultar. Podemos observar en la siguiente figura que aún no tenemos ningún servicio activo.

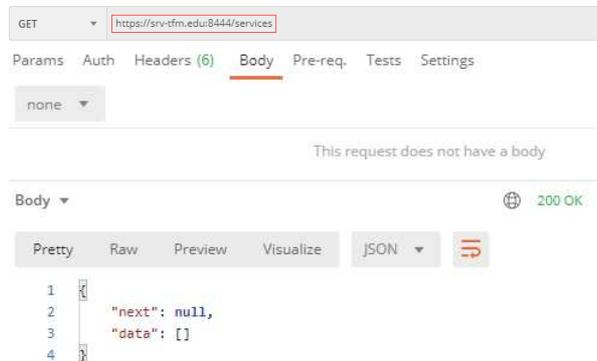


Figura 52 - Kong Ver servicios Activos

A modo de ejemplo se va a registrar una API REST al entorno de Kong. Utilizaremos la API REST que podíamos ver en el apartado 2.5.2 “reqres.in”. Para llevar a cabo esta tarea se realizará un POST al contexto de “services” de la API Admin de Kong con sus respectivos parámetros en el body de la petición. Como se puede observar en la figura vamos a añadir un servicio llamado “API-Test”.

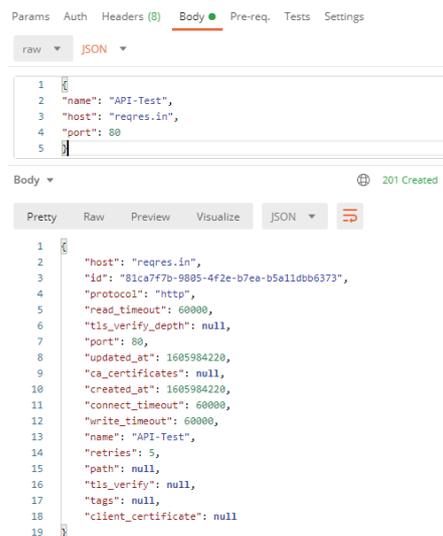


Figura 53 - Kong Crear servicio

Una vez creado el servicio tenemos que añadir los denominados “routes” que son los mapeos definidos entre las API y Kong. En el siguiente ejemplo crearemos una ruta que mapee la llamada de “/api/users” con la llamada del tipo GET

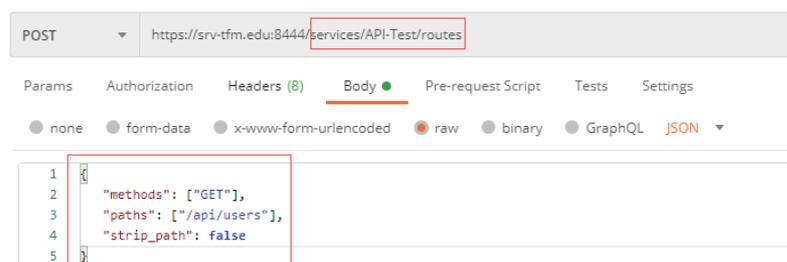
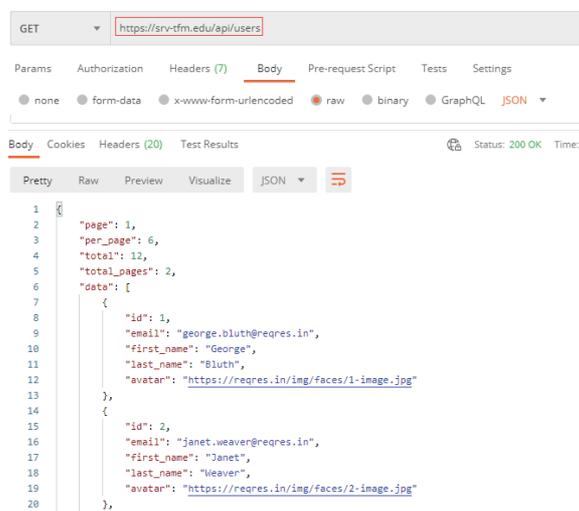


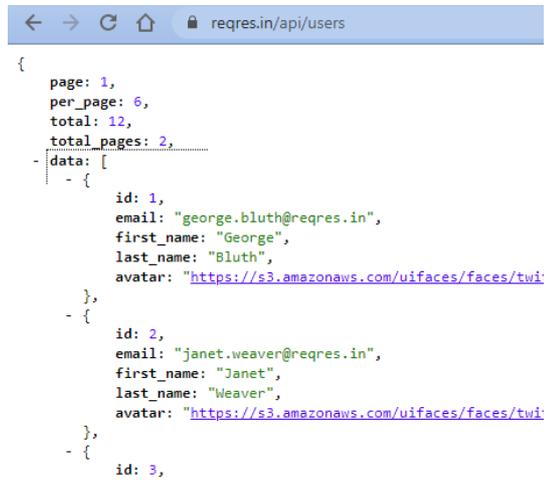
Figura 54 - Creando Routes

Ahora realizaremos la petición desde Postman para comprobar que las rutas funcionan. En las siguientes figuras podemos observar que tanto Kong como la web de reqres.in devuelven los mismos resultados.



```
1 {
2   "page": 1,
3   "per_page": 6,
4   "total": 12,
5   "total_pages": 2,
6   "data": [
7     {
8       "id": 1,
9       "email": "george.bluth@reqres.in",
10      "first_name": "George",
11      "last_name": "Bluth",
12      "avatar": "https://reqres.in/img/faces/1-image.jpg"
13    },
14    {
15      "id": 2,
16      "email": "janet.weaver@reqres.in",
17      "first_name": "Janet",
18      "last_name": "Weaver",
19      "avatar": "https://reqres.in/img/faces/2-image.jpg"
20    }
21  ]
22 }
```

Figura 55 - GET desde Kong



```
{
  "page": 1,
  "per_page": 6,
  "total": 12,
  "total_pages": 2,
  "data": [
    - {
      "id": 1,
      "email": "george.bluth@reqres.in",
      "first_name": "George",
      "last_name": "Bluth",
      "avatar": "https://s3.amazonaws.com/uifaces/faces/twi"
    },
    - {
      "id": 2,
      "email": "janet.weaver@reqres.in",
      "first_name": "Janet",
      "last_name": "Weaver",
      "avatar": "https://s3.amazonaws.com/uifaces/faces/twi"
    },
    - {
      "id": 3,

```

Figura 56 - GET desde reqres.in

3.3.2 Instalación y configuración Dashboard Konga

Para la instalación de Konga visitaremos la web del proyecto en GitHub¹⁷ y se clonará el proyecto para una instalación on-premises junto con una configuración de base de datos Postgres. A continuación, se describen todos los pasos que se necesitan para la instalación en nuestro servidor `srv-tfm`.

El primer paso será tener instaladas todas las dependencias y clonar el proyecto desde GitHub. Ejecutaremos los siguientes comandos en la terminal.

```
root@srv-tfm:~# npm install -g bower
root@srv-tfm:~# npm run bower-deps
root@srv-tfm:~# npm install -g gulp-cli
root@srv-tfm:~# git clone https://github.com/pantsel/konga.git
root@srv-tfm:~# cd konga
root@srv-tfm:~# npm i
```

Una vez clonado el proyecto procederemos a la creación de la base de datos con los siguientes comandos.

```
root@srv-tfm:~# su postgres
postgres@srv-tfm:~$ psql
postgres=# CREATE USER konga; CREATE DATABASE konga OWNER konga;
postgres=# ALTER USER konga WITH PASSWORD 'kongaTFM';
```

Para una instalación on-premises de Konga debemos de crear un fichero de configuración dentro del directorio del proyecto. Konga nos facilita un fichero de ejemplo llamado `“.env_example”` en donde realizando una copia del fichero `“.env”` ya podremos añadir las configuraciones oportunas para el sitio web. A continuación, se muestran los parámetros añadidos al fichero.

```
root@srv-tfm:~# cp .env_example .env
root@srv-tfm:~# nano .env
PORT=1337
NODE_ENV=production
KONGA_HOOK_TIMEOUT=120000
DB_ADAPTER=postgres
DB_URI=postgres://konga:kongaTFM@localhost:5432/konga
KONGA_LOG_LEVEL=debug
TOKEN_SECRET=TFM_JWT_t0k3n!
SSL_KEY_PATH=/opt/konga/certs/selfsignedTFM.key
SSL_CERT_PATH=/opt/konga/certs/selfsignedTFM.crt
NODE_TLS_REJECT_UNAUTHORIZED=0
```

Como último paso se debe ejecutar el script que se encargará de la creación de los objetos en la base de datos. Ejecutaremos en la terminal el siguiente comando.

```
root@srv-tfm:~# node ./bin/konga.js prepare --adapter postgres --
uri postgres://konga:kongaTFM@localhost:5432/konga
```

¹⁷ <https://github.com/pantsel/konga>

Finalmente procederemos a arrancar la aplicación como se muestra en la siguiente figura.

```
root@srv-tfm:/opt/konga# npm run production
> kongadmin@0.14.9 production /opt/konga
> node --harmony app.js --prod
-----
:: Tue Dec 01 2020 22:39:02 GMT+0100 (Central European Standard Time)
Environment : production
Host        : 0.0.0.0
Port       : 1337
-----
```

Figura 57 - Konga iniciar servicio.

Por defecto Konga levanta su web en el puerto 1337. Además, se ha configurado para que el acceso vaya mediante HTTPS. Entonces la URL para acceder será <https://srv-tfm.edu:1337> en donde para un primer inicio en la web deberemos de crear el usuario administrador. Como se puede ver en la siguiente figura.

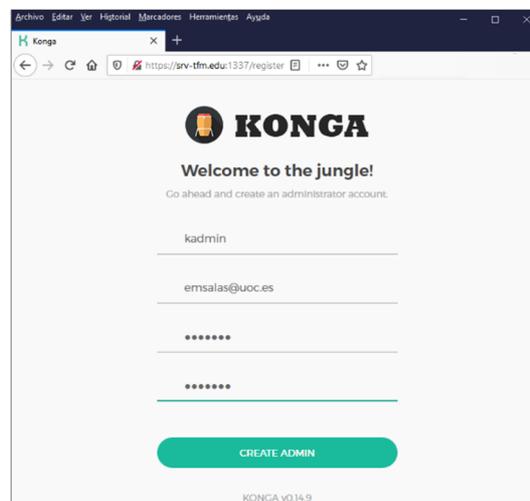


Figura 58 - Konga primer inicio

Tras la creación del usuario se podrá iniciar sesión en la aplicación.

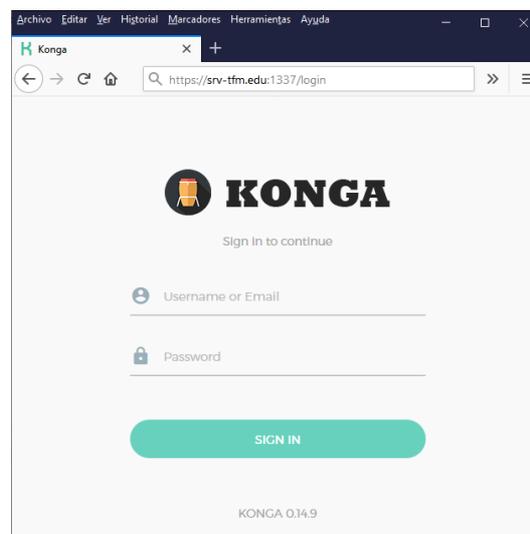


Figura 59 - Konga acceso a la aplicación

Nuestro siguiente objetivo es establecer conexión con la API de Kong. Como se puede ver en la captura al no existir ninguna conexión es la primera opción que nos presenta.

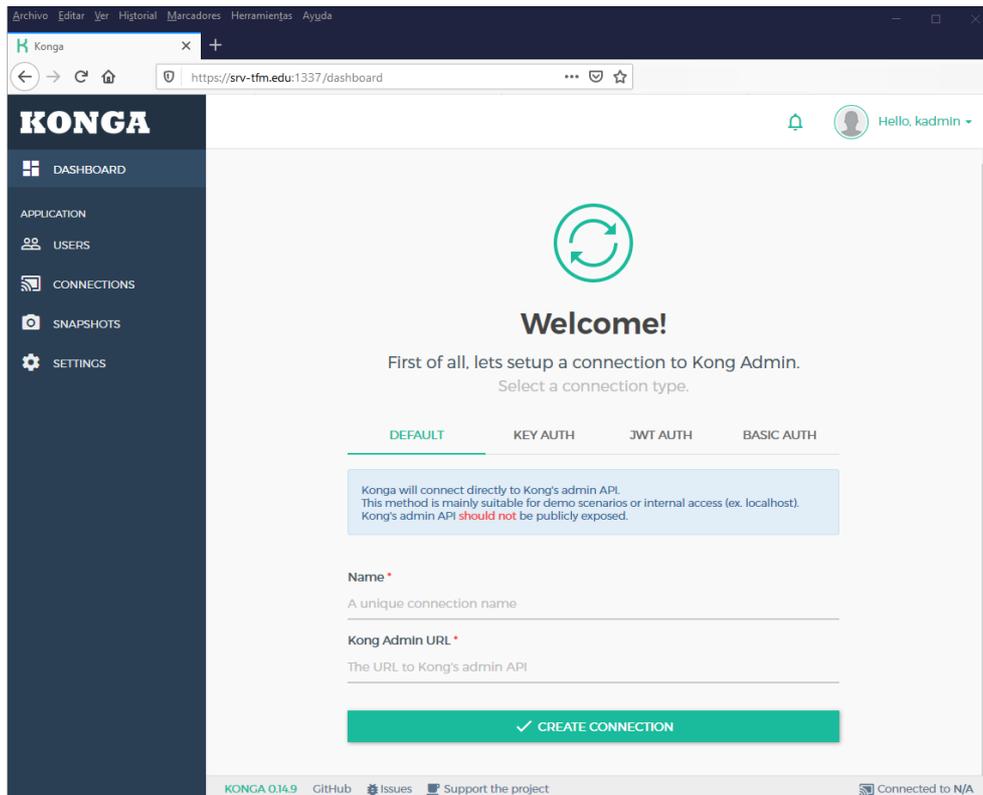


Figura 60 - Konga crear conexión

Como se ha visto en el apartado 3.3.1 utilizaremos los datos del servicio API Admin de Kong para realizar la configuración. Podemos ver como se queda la conexión en la siguiente figura.

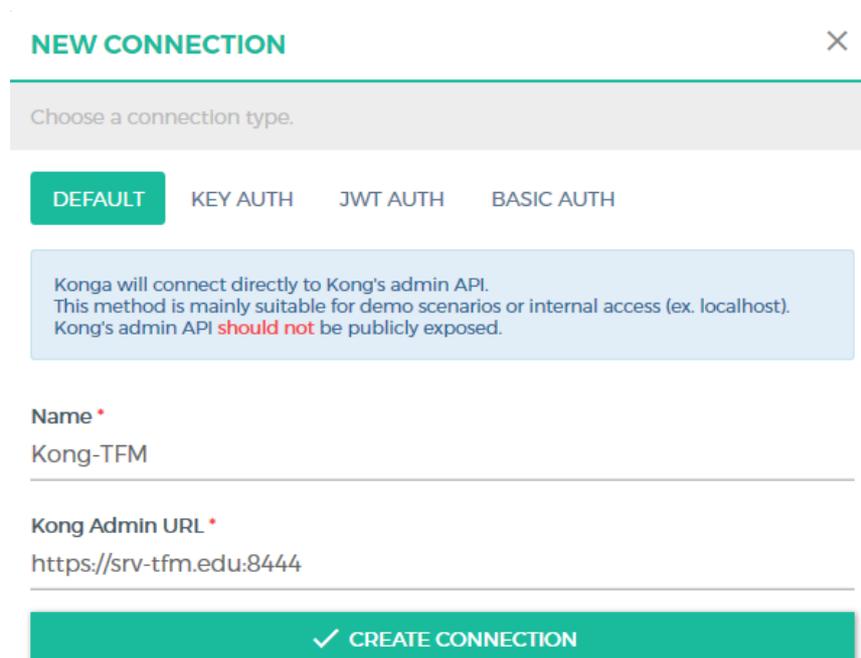


Figura 61 - Konga datos conexión Kong

Tras la realización de la conexión podemos observar que ya tendremos acceso a todas las funcionalidades de la web de Konga.

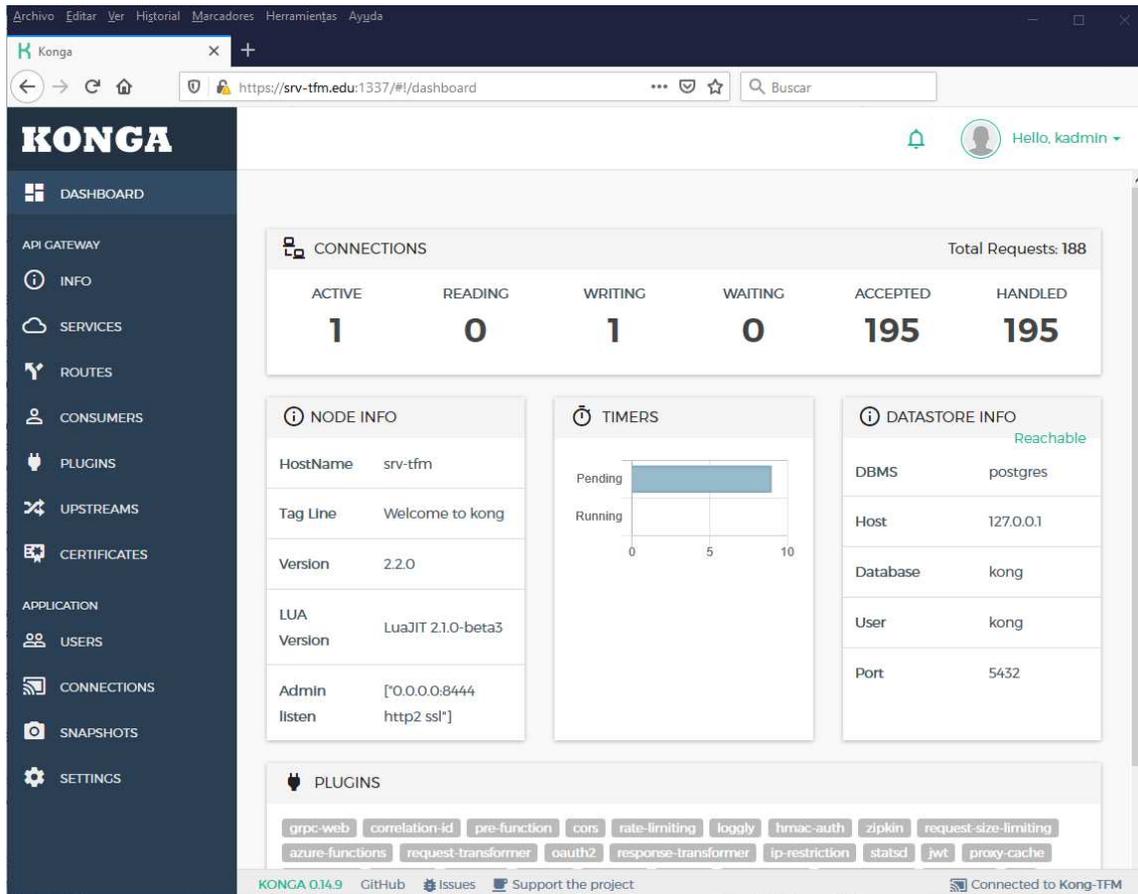


Figura 62 - Konga Dashboard

En el apartado Services es donde se irán registrando las API. En la siguiente figura se muestra el registro de la “API-Test” que se creó vía POST en la [figura 53](#)

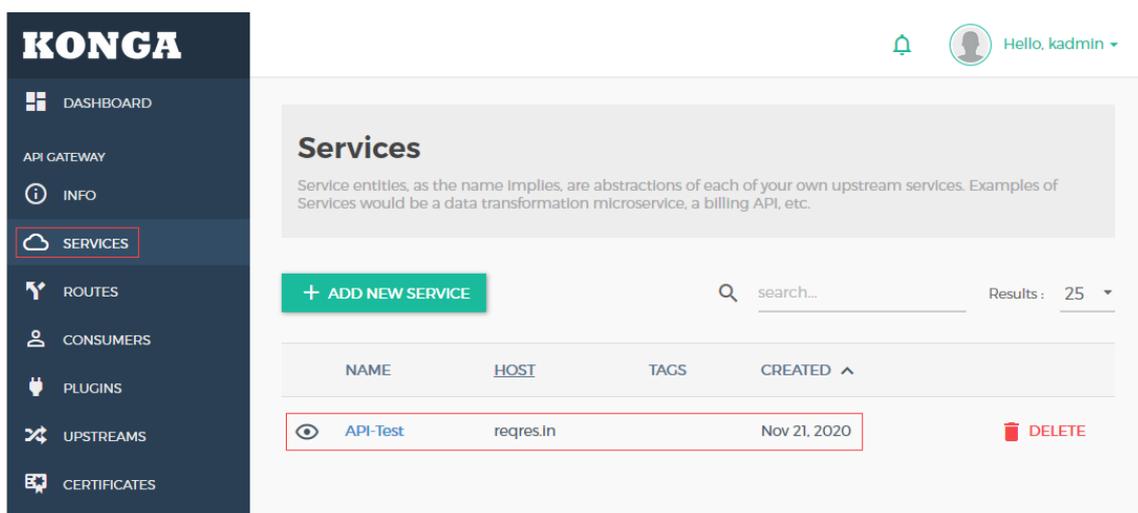


Figura 63 - Konga apartado Services

Continuando con la configuración de Konga el siguiente paso será añadir la API REST creada en Node.js y ExpressJS para que todas las peticiones las puedas gestionar Kong.

Como podíamos ver en el apartado anterior para poder registrar una API en Kong teníamos que utilizar herramientas como [Curl](#) o Postman lo cual nos llevaría mucho tiempo en el caso de registrar las diferentes rutas para cada API. Entonces, con la ayuda de Konga nos facilitará esta tarea de una forma más gráfica y acelerando el proceso de creación.

Por lo tanto, crearemos una nueva entrada de API en la web de Konga para nuestro proyecto de TFM. A continuación, se mostrarán una serie de figuras indicando el proceso.

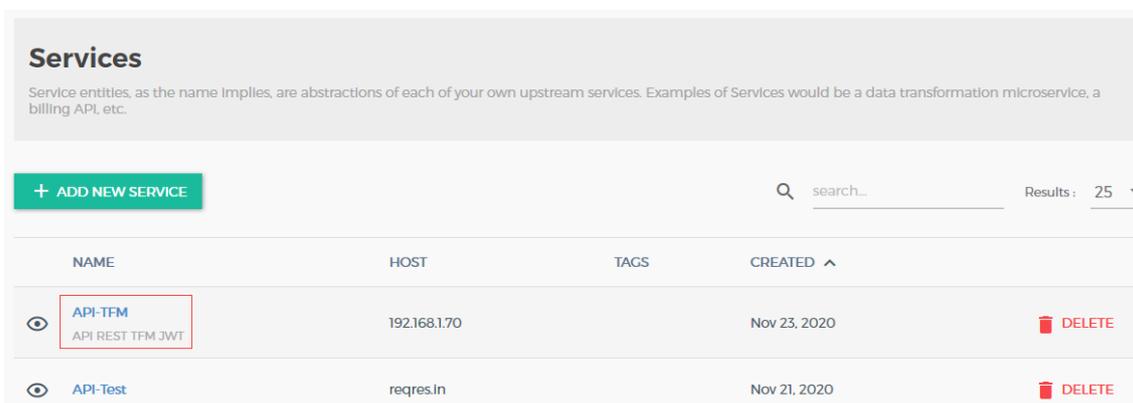


Figura 64 - Konga nuevo servicio creado.

Una vez creado el servicio accederemos al menú de la izquierda "Routes" en donde debemos asociar el servicio con las rutas a cada tipo de llamada. Podemos ver el detalle en la siguiente figura

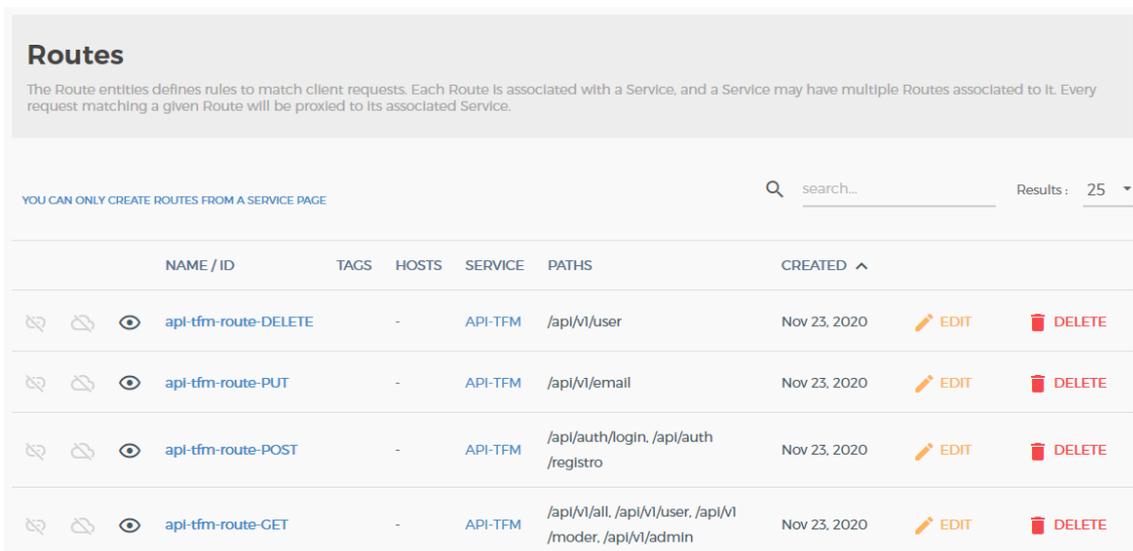


Figura 65 - Konga rutas para API-TFM

Tras finalizar la configuración de Servicios y Rutas podremos consumir la API desde el servicio proxy de Kong.

4. Conclusiones

Gracias al desarrollo del *trabajo fin de máster* se ha indagado más a fondo sobre los diferentes conceptos y funcionamiento de las API RESTful y sus herramientas. Así como la posibilidad de ir añadiendo seguridad en las diferentes etapas de su implantación.

Se ha realizado una prueba de concepto con la codificación de nuestra propia API REST, en donde, inicialmente con un ejemplo básico utilizando un algoritmo simétrico (palabra secreta) y luego cambiado la configuración para utilizar un algoritmo asimétrico.

Creo que hoy en día es difícil pensar que una gran organización tenga una API REST sin seguridad o que simplemente se pueda acceder directamente al servidor que la sirve.

Como se ha visto a lo largo del trabajo existen muchas herramientas que pueden ayudar a mantener una API REST de forma segura. No solo aumentando la seguridad en los tokens, sino que también en la infraestructura que la implementa. Hablamos de las API Gateway que permiten gestionar multitud API para un mismo dominio o diferentes.

Tenemos que pensar que las grandes empresas no solo trabajan con una solo API, sino que podrían gestionar multitudes según las líneas de negocio a las cuales se dedican. Y sin un gestor para administrarlas esta tarea sería casi imposible.

Además, se ha podido ver que con la ayuda de Kong y Konga facilita mucho el trabajo a la hora de gestionar varias API.

Otro punto importante que se ha podido observar durante la implantación es que muchas de las herramientas que han sido utilizadas vienen por defecto con una configuración básica (puertos estándares y sin certificados) lo cual podría ser un problema si se publican a internet con una configuración por defecto.

Con el incremento de los ciberataques hoy en día publicar algún servicio a internet debería de cumplir unas mínimas garantías para no sufrir ningún ataque o denegación de servicio que pueda dañar la imagen de la compañía.

Que mínimos una conexión por HTTPS y una arquitectura API Gateway para poder gestionar miles de peticiones a nuestros microservicios que sean accesibles desde la red.

También cabe destacar que durante el desarrollo del TFM me he encontrado con varios detractores de la utilización de JWT para API REST por fallos de seguridad que se pueden ocasionar. Pero también tengo que decir que muchos de estos fallos son producidos por una mala configuración (misconfiguration) o en muchos casos por publicar directamente (sin gateway) una API REST a internet.

Lo que quiere decir que no siguen el estándar de API RESTful publicada por el señor Thomas Fielding como podíamos leer en el apartado estado del arte.

Finalmente, este trabajo pretende dar un ejemplo de las múltiples configuraciones que puedan existir a la hora de implementar una API RESTful así como añadir una arquitectura robusta antes de publicar una API a internet.

4.1 Evaluación de los objetivos

En este apartado se realiza la valoración de los diferentes objetivos definidos en el plan de trabajo.

Objetivos de Investigación

- **Investigar las características de las API REST para el estilo RESTful. Utilizando los verbos que nos proporciona el protocolo HTTP (POST, GET, PUT, DELETE)**

Se ha investigado su funcionamiento y aplicado su uso durante la codificación de la API REST. Se pueden ver diferentes ejemplos en las llamadas a los endpoint en el apartado 3.2.3.

- **Investigar y estudiar el funcionamiento de los JSON Web Tokens (JWT)**

Se ha investigado el funcionamiento y estructura que tienen los JWT. Además de haberse implementado en la API del TFM.

- **Investigar y analizar los potenciales fallos de seguridad en JWT**

Durante la fase de investigación de este objetivo se puede decir que ha sido uno de los más largos dado a la gran cantidad de información dispersa en la red y además de difícil implementación (reproducir) en un entorno real por los diferentes factores que aplican. Este objetivo se ha trabajado de forma más teórica en el desarrollo del TFM.

- **Investigar y estudiar la utilización de Kong como Gateway y Konga para la administración de los servicios API REST**

Se ha investigado y estudiado la forma de usar la API Gateway de Kong, así como Konga para facilitar la configuración de las API.

Objetivos de implantación

- **Detallar la arquitectura técnica necesaria para implementar JWT en el entorno definido y detallar técnicamente cada uno de sus componentes**

Este objetivo se puede dar por cumplido, ya que se ha separado en dos fases. La primera fase teníamos la API REST codificada y accediendo de forma directa. Y en una segunda fase se implementa la API Gateway más el dashboard de Konga.

- **Instalación servidor Linux con Node.js, PM2, módulos para JWT y MongoDB**

El objetivo se puede considerar cumplido referente a la instalación y configuración del servidor y sus componentes.

- **Implementar JWT en la codificación de la API RESTful**

Este objetivo se ha llevado a cabo en la codificación de la API REST utilizando clave simétrica y asimétrica.

- **Aprender a utilizar los diferentes claims que define el RFC de JWT**

Tras el estudio de JWT durante el desarrollo del proyecto se han implementado los diferentes claims (registrados, públicos y privados) como se pueden observar en las figuras que hacen referencia a los tokens.

- **Codificación de los diferentes endpoints de la API con Node.js y utilizando MongoDB como base de datos NoSQL**

Objetivo cumplido en el desarrollo de los diferentes endpoint en donde se ha utilizado MongoDB para almacenar los datos de la API (roles y usuarios).

- **Instalación de Kong (Gateway) y Konga (dashboard) para la gestión de la API**

Tras la instalación y configuración de ambas herramientas en el servidor Linux podemos dar por cumplido el objetivo. Además de realizar configuraciones personalizadas a cada herramienta para ser utilizadas de forma segura (HTTPS)

- **Puesta en marcha del producto final y la realización de pruebas para diferentes roles/usuarios que podrán utilizar la API diseñada**

Finalmente se ha conseguido tener una API RESTful funcional con una arquitectura API Gateway la cual es capaz de gestionar diferentes peticiones para diversas API. Se comprueban que todos los endpoints funcionan correctamente tal y como funcionaban durante la fase uno de la arquitectura.

Si bien la valoración general es que se han cumplido todos los objetivos planteados, se puede decir que ha sido un trabajo de dedicación de muchas horas, pero finalmente se ha podido sacar provecho del trabajo de investigación para poder implementarlo de forma real en un proyecto de mi entorno laboral.

4.2 Trabajo Futuro

Tras dar por cumplidos los objetivos del plan de trabajo, podemos indicar que existen varias líneas de mejora para el proyecto. A modo de ejemplo:

- Se queda pendiente el estudio e investigación de la librería JOSE para Node.js. Ya que esta librería lleva todos los estándares (JWT, JWS, JWE). Así como su codificación en JavaScript y Node.js.
- Ampliar las funcionalidades de la API REST con nuevos endpoints.
- Se ha implementado el dashboard de Konga para poder facilitar el manejo de las API que están registradas en Kong. Por lo tanto, quedaría pendiente como trabajo futuro ampliar cada una de las características (plugins) de Kong y Konga como, por ejemplo: logging (syslog, http log), ACL, Rate Limiting, Proxy Cache, Analytics & Monitoring.
- Con respecto a la arquitectura se podría ampliar los servidores de Kong para crear una infraestructura en modo clúster HA.
- Y aplicar la utilización de la API para entornos web o dispositivos móviles.

5. Glosario¹⁸

HTTP: Hypertext Transfer Protocol (Protocolo de transferencia de hipertexto)

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

JSON: JavaScript Object Notation es un formato de texto sencillo para el intercambio de datos

API: Application programming interface

JWT: JSON Web Token

JWS: JSON Web Signature

JWE: JSON Web Encryption

URI: (Uniform Resource Identifier): es una cadena de caracteres que identifica los recursos de una red de forma unívoca.

MIME: (MUltipurpose Internet Mail Extensions) son una serie de convenciones o especificaciones dirigidas al intercambio a través de Internet de todo tipo de archivos (texto, audio, vídeo, etc.) de forma transparente para el usuario

SoC: (separation of concerns) separación de intereses, también denominada separación de preocupaciones

URL: (Uniform Resource Locator) es un identificador de recursos uniforme (Uniform Resource Identifier, URI) cuyos recursos referidos pueden cambiar, esto es, la dirección puede apuntar a recursos variables en el tiempo

Base64URL Encoding: es un sistema de numeración posicional que usa 64 como base. Es la mayor potencia que puede ser representada usando únicamente los caracteres imprimibles de ASCII. En el caso de URL Encoding los caracteres “+” y “/” son reemplazados por “-” y “_” respectivamente.

CORS (Cross-origin resource sharing): es un mecanismo que permite que se puedan solicitar recursos restringidos en una página web desde un dominio diferente del dominio que sirvió el primer recurso

Cassandra: se trata de un software NoSQL distribuido y basado en un modelo de almacenamiento de «clave-valor»

Apache ZooKeeper: es un proyecto de software libre de la Apache Software Foundation, que ofrece un servicio para la coordinación de procesos distribuidos.

Nginx: es un servidor web/proxy inverso ligero de alto rendimiento

¹⁸ Definiciones obtenidas desde la web de [Wikipedia](#).

Docker: automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

Redis es un motor de base de datos en memoria, basado en el almacenamiento en tablas de hashes (clave/valor).

Curl: software que se basa en una biblioteca (libcurl) y un intérprete de comandos (curl) orientado a la transferencia de archivos

OpenAPI: es una especificación para archivos de interfaz legibles por máquina para describir, producir, consumir y visualizar servicios web RESTful

Go: es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C, que intenta ser dinámico como Python y con el rendimiento de C o C++. Ha sido desarrollado por Google

SaaS: Software como un Servicio, (Software as a Service), es un modelo de distribución de software donde el soporte lógico y los datos que maneja se alojan en servidores de una compañía de tecnologías, a los que se accede vía Internet desde un cliente

6. Bibliografía

Roy Thomas Fielding. *Arquitectura REST (2000)* [en línea] [fecha de consulta: 03 de octubre de 2020].

Disponible en: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

Wikipedia REST [en línea] [fecha de consulta: 04 octubre de 2020].

Disponible en: https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional

Wikipedia Interfaz de programación de aplicaciones (2017) [en línea] [fecha de consulta: 04 octubre 2020].

Disponible en: https://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

Wikipedia JSON Web Token (2018) [en línea] [fecha de consulta: 04 octubre 2020].

Disponible en: https://es.wikipedia.org/wiki/JSON_Web_Token

JSON Web Signature (2017) [en línea] [fecha de consulta: 04 octubre 2020].

Disponible en: https://es.wikipedia.org/wiki/JSON_Web_Signature

M. Jones. *Estándar JSON Web Encryption* (mayo 2015) [en línea] [fecha de consulta: 04 de octubre 2020].

Disponible en: <https://tools.ietf.org/html/rfc7516>

OAuth 2.0 [en línea] [fecha de consulta: 04 de octubre 2020].

Disponible en: <https://es.wikipedia.org/wiki/OAuth>

Pérez-Bermejo, Alberto; Evgeniev, Martin. *Tyk y Kong: analizamos estos dos API Gateways* (11 de abril 2019) [en línea] [fecha de consulta: 04 de octubre 2020].

Disponible en: <https://www.bbva.com/es/tyk-kong-analizamos-estos-dos-api-gateways>

Lysa Myers. *Introducción a la autenticación: cómo probar que realmente eres tú* (4 de mayo 2016) [en línea] [fecha de consulta: 13 de octubre 2020].

Disponible: <https://www.welivesecurity.com/la-es/2016/05/04/autenticacion-como-probar-que-eres-tu/>

Sunit Chatterjee. *Working with JWT (Json Web Tokens)* (17 de agosto 2020) [en línea] [fecha de consulta: 15 de octubre 2020].

Disponible: <https://developerpod.com/2020/08/17/working-with-jwt-json-web-tokens>

Michael Miele. *Representational State Transfer (REST) APIs Part 1 of 4* (9 de octubre, 2018) [en línea] [fecha de consulta: 17 de octubre 2020].

Disponible: <http://acloudsky.com/representational-state-transfer-rest-apis-part-1-of-4/>

John Bradley, Brian Campbell, Michael B. Jones, Chuck Mortimore. *IANA JSON Web Token Registry* [en línea] [fecha de consulta: 18 de octubre 2020].

Disponible: <https://www.iana.org/assignments/jwt/jwt.xhtml>

Prabath Siriwardena. *JWT, JWS and JWE for Not So Dummies!* (27 de abril 2016) [en línea] [fecha de consulta: 19 de octubre 2020].

Disponible: <https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3>

Nuwan Dias, Prabath Siriwardena *Microservices Security in Action* (agosto 2020) [en línea] [fecha de consulta: 22 de octubre 2020]

Disponible: <https://livebook.manning.com/book/microservices-security-in-action/h-json-web-token-jwt-v-7/32>

Sebastian Peyrott. *A Look at The Draft for JWT Best Current Practices*. (11 de abril 2018)
[en línea] [fecha de consulta: 26 de octubre 2020]
Disponible: <https://auth0.com/blog/a-look-at-the-latest-draft-for-jwt-bcp/>

Viktor Yordanov Ivanov. *Introducción al API Gateway Pattern*. (27 de mayo 2020) [en línea] [fecha de consulta: 23 de octubre 2020]
Disponible: <https://www.adictosaltrabajo.com/2020/05/27/introduccion-al-api-gateway-pattern/>

Magic Quadrant for Full Life Cycle API Management [en línea] [fecha de consulta: 24 de octubre 2020]
Disponible: <https://www.gartner.com/doc/reprints?id=147SMI7U&ct=200922&st=sb>

Antonio Santiago. *Using async/await in ExpressJS middlewares* [en línea] [fecha de consulta: 23 de noviembre 2020]
Disponible: <https://www.acuriousanimal.com/blog/2018/03/15/express-async-middleware>

7. Anexos

7.1 Código fuente TFM

Código Fuente TFM-JWT	
server.js	<pre>const express = require("express"); const bodyParser = require("body-parser"); const cors = require("cors"); const dbConfig = require("../config/db.config"); const fs = require('fs'); const app = express(); // CODE HTTPS const https = require('https'); var key = fs.readFileSync(__dirname + '/cert/selfsignedTFM.key'); var cert = fs.readFileSync(__dirname + '/cert/selfsignedTFM.crt'); var options = { key: key, cert: cert }; const db = require("../models"); const Role = db.role; db.mongoose .connect(`mongodb://\${dbConfig.HOST}:\${dbConfig.PORT}/\${dbConfig.DB}`, { useNewUrlParser: true, useUnifiedTopology: true }) .then(() => { console.log("Successfully connect to MongoDB."); //initial(); }) .catch(err => { console.error("Connection error", err); process.exit(); }); // simple route app.get("/", (req, res) => { res.json({ message: "Protegiendo una API REST que utiliza JSON Web Tokens" }); }); // routes require("../routes/auth.routes")(app); require("../routes/user.routes")(app); // v1 HTTP Puerto 8080 const PORT = process.env.PORT 8080; // v2 HTTPS Puerto 8443 //const PORT = process.env.PORT 8443; //app.listen(PORT, () => { // console.log(`Server is running on port \${PORT}.`); //}); var server = https.createServer(options, app); server.listen(PORT, () => { console.log("Server TFM starting on port : " + PORT) });</pre>
package.json	<pre>{ "name": "tfm-jwt", "version": "1.0.0", "description": "Aplicando seguridad a una API REST con JWT", "main": "server.js", "scripts": { "test": "echo \\\"Error: no test specified\\\" && exit 1" }, "keywords": ["Node.js", "JWT", "mongoDB", "Authentication", "Authorization"], "author": "emsalasg", "license": "ISC", "dependencies": { "bcryptjs": "^2.4.3",</pre>

	<pre> "body-parser": "^1.19.0", "config": "^3.3.2", "cors": "^2.8.5", "express": "^4.17.1", "fs": "0.0.1-security", "jsonwebtoken": "^8.5.1", "mongoose": "^5.10.14" } } </pre>
user.routes.js	<pre> // File: routes/user.routes const jwtAuth = require("../middlewares/ajwt.mdlw"); const controller = require("../controllers/user.controller"); module.exports = function(app) { app.use(function(req, res, next) { res.header("Access-Control-Allow-Headers", "x-jwt-token, Origin, Content-Type, Accept"); next(); }); // LLAMADAS A LA API // GET ALL app.get("/api/v1/all", controller.allAccess); // GET USER app.get("/api/v1/user", [jwtAuth.verifyToken], controller.userBoard); // GET MODER app.get("/api/v1/moder", [jwtAuth.verifyToken, jwtAuth.isModerator], controller.moderatorBoard); // GET ADMIN app.get("/api/v1/admin", [jwtAuth.verifyToken, jwtAuth.isAdmin], controller.adminBoard); // PUT EMAIL app.put("/api/v1/email", [jwtAuth.verifyToken, jwtAuth.isModerator], controller.updateEmail); // DELETE USER app.delete("/api/v1/user", [jwtAuth.verifyToken, jwtAuth.isAdmin], controller.deleteUser); } </pre>
auth.routes.js	<pre> // File: routes/auth.routes const verifyUser = require("../middlewares/vuser.mdlw"); const controller = require("../controllers/auth.controller"); module.exports = function(app) { app.use(function(req, res, next) { res.header("Access-Control-Allow-Headers", "x-jwt-token, Origin, Content-Type, Accept"); next(); }); // ENDPOINT: REGISTRAR USUARIO app.post("/api/auth/registro", [verifyUser.checkDuplicateUsernameOrEmail, verifyUser.checkRolesExisted], controller.registro); // ENDPOINT: HACER LOGIN app.post("/api/auth/login", controller.login); } </pre>
user.model.js	<pre> // File: models/user.models // CARGA CLIENTE DE MONGO const mongoose = require("mongoose"); // CREAR NUEVO USUARIO const User = mongoose.model("User", new mongoose.Schema({ username: String, email: String, password: String, roles: [{ type: mongoose.Schema.Types.ObjectId, ref: "Role" }] })); module.exports = User; </pre>
role.model.js	<pre> // File: models/role.models // CARGA CLIENTE DE MONGO const mongoose = require("mongoose"); // CREAR NUEVO USUARIO const Role = mongoose.model("Role", new mongoose.Schema({ name: String })); module.exports = Role; </pre>
index.js	<pre> // File: models/index const mongoose = require('mongoose'); </pre>

	<pre> mongoose.Promise = global.Promise; const dbMongo = {}; dbMongo.mongoose = mongoose; dbMongo.role = require("../role.model"); dbMongo.user = require("../user.model"); dbMongo.ROLES = ["user", "admin", "moderator"]; module.exports = dbMongo; </pre>
vuser.mdlw.js	<pre> // File: middlewares/vuser.mdlw const db = require("../models"); const User = db.user; const uRoles = db.ROLES; checkExistenRoles = (req, res, next) => { if (req.body.roles) { for (x = 0; x < req.body.roles.length; x++) { if (!uRoles.includes(req.body.roles[x])) { res.status(400).send({ message: `Error Rol \${req.body.roles[x]} no existe!` }); return; } } } next (); }; checkUsernameEmailDuplicados = (req, res, next) => { // Username User.findOne({ username: req.body.username }).exec ((err, user) => { if (err) { res.status(500).send({ message: err }); return; } if (user) { res.status(400).send({ message: "Error! ¡Usuario está en uso!" }); return; } // Email User.findOne({ email: req.body.email }).exec((err, user) => { if (err) { res.status(500).send({ message: err }); return; } if (user) { res.status(400).send({ message: "Error! ¡Email está en uso!" }); return; } next (); }); }); }; </pre>
index.js	<pre> // File: middlewares/index // INDEX MIDDLEWARE const authJwt = require("../ajwt.mdlw"); const verifyUser = require("../vuser.mdlw"); // EXPORT JS FILES module.exports = { authJwt, verifyUser }; </pre>
ajwt.mdlw.js	<pre> // File: middlewares/ajwt.mdlw const jwt = require("jsonwebtoken"); const config = require("../config/auth.config.js"); const db = require("../models"); const fs = require('fs'); const publicKey = fs.readFileSync('cert/jwtRS256.pub'); const User = db.user; const Role = db.role; // COMPROBAR TOKEN verificarToken = (req, res, next) => { console.log ("verificando token"); console.log ("publicKey: " + publicKey); let token = req.headers["x-jwt-token"]; console.log ("Token:" + token); </pre>

	<pre> if (!token) { return res.status(403).send({ message: "No se ha proporcionado Token!" }); } // VERSION CON CLAVE SECRETA /*jwt.verify(token, config.secret, (err, decoded) => { if (err) { return res.status(401).send({ message: "Unauthorized!" }); } req.userId = decoded.id; next(); });*/ // VERSION CON CLAVE PUBLICA try { const decoded = jwt.verify(token, publicKey, { algorithm: 'RS256', }); req.userId = decoded.jti; next(); } catch (error) { console.log ("error:" + error); return res.status(401).send('Unauthorized'); } }; esAdmin = (req, res, next) => { User.findById(req.userId).exec((err, user) => { if (err) { res.status(500).send({ message: err }); return; } Rol.find({ _id: { \$in: user.roles } }, (err, roles) => { if (err) { res.status(500).send({ message: err }); return; } for (x = 0; x < roles.length; x++) { if (roles[x].name === "admin") { next(); return; } } res.status(403).send({ message: "Require Rol Admin!" }); return; }); }); }; esModerador = (req, res, next) => { User.findById(req.userId).exec((err, user) => { if (err) { res.status(500).send({ message: err }); return; } Rol.find((err, roles) => { if (err) { res.status(500).send({ message: err }); return; } for (x = 0; x < roles.length; x++) { if (roles[x].name === "moderador") { next(); return; } } res.status(403).send({ message: "Require rol Moderador!" }); return; }); }); }; </pre>
user.controller.js	// File: controllers/user.controller const db = require("../models");

	<pre> const User = db.user; exports.allAccess = (req, res) => { res.status(200).send("Contenido público todo el mundo puede consultar."); }; exports.userBoard = (req, res) => { res.status(200).send("Contenido exclusivo para usuarios logados"); }; exports.adminBoard = (req, res) => { res.status(200).send("Contenido exclusivo para usuarios Admin"); }; exports.moderatorBoard = (req, res) => { res.status(200).send("Contenido exclusivo para usuarios Moderadores"); }; // MODIFICAR CORREO exports.updateEmail = (req, res) => { User.findOneAndUpdate({ "username": req.body.username }, { "\$set": { "email": req.body.email, } }).exec(function(err, User){ if(err) { console.log(err); res.status(500).send(err); } else { res.status(200).send({ message: "Email Actualizado!" }); } }); }); }; // BORRAR USUARIO exports.deleteUser = (req, res) => { User.findOne({username: req.body.username}, function (error, User){ console.log("Borrando al usuario " + User); User.remove(); res.status(200).send({ message: "Usuario eliminado!" }); }); }; </pre>
<p>auth.controller.js</p>	<pre> // File: controllers/auth.controller // LEER FICHERO DE CONFIG PARA LA PALABRA SECRETA v1 const config = require("../config/auth.config"); // LEER LOS MODELOS DE MONGO const db = require("../models"); // MODULO PARA LEER FICHEROS const fs = require('fs'); // VERSION 2 ADD KEY const privateKey = fs.readFileSync('cert/jwRS256.key'); // DEFINIR OBJETOS DESDE LA COLECCION DE MONGO const User = db.user; const Role = db.role; // CARGAR MODULO JWT var jwt = require("jsonwebtoken"); // CARGAR MODULO ENCRIPACION var bcrypt = require("bcryptjs"); // DEFINIR CLAIMS var issClaim = "srv-tfm.edu"; var subClaim = "TFM"; var audClaim = "TFM-API-JWT"; var algClaim = "RS256"; // ALTA USUARIO exports.registro = (req, res) => { const user = new User({ username: req.body.username, email: req.body.email, password: bcrypt.hashSync(req.body.password, 8) }); // GUARDAR REGISTRO user.save((err, user) => { if (err) { res.status(500).send({ message: err }); } return; }) // BUSCAR USUARIO ANTES DE CREARLO if (req.body.roles) { Role.find({ name: { \$in: req.body.roles } }, (err, roles) => { if (err) { res.status(500).send({ message: err }); return; } user.roles = roles.map(role => role._id); user.save(err => { if (err) { res.status(500).send({ message: err }); return; } }) }) } } </pre>

	<pre> res.send({ message: "Usuario creado correctamente!" }); }); } }); } else { Role.findOne({ name: "user" }, (err, role) => { if (err) { res.status(500).send({ message: err }); return; } user.roles = [role._id]; user.save(err => { if (err) { res.status(500).send({ message: err }); return; } res.send({ message: "Usuario creado correctamente!" }); }); }); } }); }); }; // LOGIN USUARIO exports.login = (req, res) => { User.findOne({ username: req.body.username }) .populate("roles", "-_v") .exec((err, user) => { if (err) { res.status(500).send({ message: err }); return; } if (!user) { return res.status(404).send({ message: "Usuario no encontrado." }); } // CONVERTIR LA PASS PARA LUEGO COMPARAR var passwordIsCorrect = bcrypt.compareSync(req.body.password, user.password); // COMPROBAR LA PASS PASADA POR PARAMETROS if (!passwordIsCorrect) { return res.status(401).send({ accessToken: null, message: "Password incorrecta!" }); } // ARRAY PARA METER LOS ROLES DE UN USUARIO var userRoles = []; // RECORRER LOS ROLES DE UN USUARIO QUE VIENE DE MONGO for (x = 0; x < user.roles.length; x++) { userRoles.push("ROLE_" + user.roles[x].name) // FIRMA v1 + ADD CLAIMS //var token = jwt.sign({ iss:issClaim, sub:subClaim, aud:audClaim, jti: user.id,username: user.username,email: user.email, roles: userRoles,}, config.secret, { //expiresIn: '1h' //}); // FIRMA v2 + ADD CLAIMS var token = jwt.sign({ // REGISTERED CLAIM iss:issClaim, sub:subClaim, aud:audClaim, jti: user.id, // PUBLIC CLAIM username: user.username, email: user.email, // PRIVATE CLAIM roles: userRoles }, privateKey, { algorithm: algClaim, expiresIn: '1h' }); // AQUI SE DEVUELVE EL TOKEN res.status(200).send({ accessToken: token }); }); }); }; </pre>
db.config.js	<pre> module.exports = { HOST: "192.168.1.70", PORT: 27017, DB: "JWTFM_db" }; </pre>
auth.config.js	<pre> module.exports = { secret: "KEY-Pr0t3g13nd0_un4_4P1-R3ST_JwT" }; </pre>

selfsignedTFM.key	<pre> }; -----BEGIN PRIVATE KEY----- MIIEvAIBADANBgkqhkiG9w0BAQEFAASCbKYwggSiAgEAAQIBAQCTfbx7nfQhQIDe mlbeG7osGk0ZEtRZyLFuimYSGvHpcjkjGHRPxsQDnkvtwOgxrlVqGssNm8FH0L C+yan/pjrw7fi7kUSVULHzu3BqvP2yp4WnHi7ZV/+r2afHeDyqosvJ0KTMu8B7Q6 kUNhSBLrriDTKpAAjbjyLgZm3aaZLiX0gLOcE8Y0txdqHVzB3W1H300ghQ83oY3b HxPNe6ycSw3vqKXvHsQIaZmELUkmlhwuVitc2QdEVQER6MxH5VjDjFkPXLXWXYw j1N2eM4ZsmJ7Em5nxs1f/Amy5O0rV/60ZiE7x1PhTPPiBf6phiFVTRH6cmGQKrO rREoJRxsAgMBAAECggEAFS08LFbW1vcJhHQwON+CrERcSSYeJmVnAlSmGzYiK0k6 p+F/gg5cSYWdkqhp8+DjgZXE4wmj3r8epr1R9539czx2z/BA+Sfz5mBhSp37gg7q TzJyLQFQXsdTemyU56aVghkit9xdOJJeRX3mhKLBiHhYXblWFC+0f5Ebn6eoFRs um397a/e19/1uLF1WCyqx7dkJtw9t1JI6oCGbQkhCsUZuGxrpjG218cri+xMg12d fuuaWdts62Qes/CPdfXIS+U4rWKBgQDFnXzDdg01c2W70HqlijFMU4F3Cuf0isgj 1wQIot/nndV1nYj12JPyxDMYXzhkWWXqieXs1QbD1eGIB24Y7FEaBlbyzABbMgd GQvpZRSztY53aSDBhty/QtQjLftaNVigqd3hHbm2nuP+YAADQ4FbT6wC00hNU8A M68eZvwiHwKBgCzk+HTACcLawPScJjLluVH6V1VtDxe7ArIWGr1WLFuXeghM44mL rgYDg8md8vTEsiyVSY9EIKtLrClPpntEb9iLMyIhBa9Jn4F61yJ5Qt+9EXnZAF 8+/2cPd+sow90ue00jrd5YsLvEptmxbnzMCw8TBTyU4kDswkUDv0hkbdAoGACskz xX3Ncjt9PE2Xwp6bVmxwJfIleL+am38EiSbaHMRy/fbFiI/3mljceHz0awhwbGH0 4RntBiwZ5I/f2ZJVf1RMOBsdkJPAWdrPzVjWmVYLQqHW80GtjfgtGUxiE1NxcS4u qObV2w9Whk4w8fXJI1B4K/xyOCDefqXmskQX4AUCgYABP/QApYq/KlJec3j1mLpU +VE4SALf4XnTQZnrYOZDr1cXKGWqD0bpa60VAlVXHyjndRL/AWIinGR1OPLbaOLR LRPc+7yrt+KtVq854YwCLASaeyHS08QbdzC4oBb+bbGXmjCPf1QaEGXFLWSX5RB uYeH+HvzX4pBTeaff0iJKA== -----END PRIVATE KEY----- </pre>
selfsignedTFM.crt	<pre> -----BEGIN CERTIFICATE----- MIIECTCAvGAgIBAgIURKQRsc7hoCcvjNVG0e/ieOz5tVYwDQYJKoZIhvcNAQEL BQAwGzmxCAJBAgNVBAQTAkVMTQ8wDQYDVQIDAZNvVJDSUEXHzAdBgNVBAAMCFkxh cyBUB3JyZXMgZGUuY290aWxsYXkxZDANBgNVBAoMBk15SG9tZTEMLmGALUECAIEC ASiWdQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAAK0VvHud9AeogN6aVt4buiwZ 0qhks1EnPisW6KZhTa8ekKOSAcEs/FJAoEs+3A6DGuUi9CCw2bwUfSUL7Jqf+Ov Dt8juRRJVQsf07cGq8/bKnhaceLt1X/6vZoUd4PKqixWPQmPy7wHtDqR2FIEuuu INMqkACNuPisZmbdprMuJfSAs5wTxg63F2odKMHdaIfc7SCFDzehjdsfE817rJxL De+ope8exAhpmyQtSsAWhc5Wk1zZB0RVARHozEflWMOMUP6ktdYhdjCOU3Z4zhmy YnsBmfGyV/8ChLk506u//rRmITvGU+FM8+IF/qmGIVVFMfQ2Y2Aq6tESiNxczEC AwEAAANTMFEwHQYDVR0OBBYEFKjO+S76EXXoaZ0jpncc9AP25Cvo2MB8GALUdLwQY MBAAFKjO+S76EXXoaZ0jpncc9AP25Cvo2MA8GALUdEwEB/wQFMAMBAF8wDQYJKoZI hvcNAQELBQADggEBAJWcIDmxYlCiTs1s9r6PeUdsBp0OgiEvsE3zBPPoogmyWVDJ vE+PgN33bdoc5S5ZBwdqtmVn5GD1m8D5R4R3wZ3gP1/ibEKb8CrdMgK8XuFv6cAe SFwYyI786GqiasBTKINNwkrGceptanylDycrnwCvhiNo/fq5H3BLEXZtNQCe26stj cV9244rnHucZXVMj4Bk4IqqTei0QoR04x7CpgakmXzH1L7WctFOLWecB/WbDUOSy FI1016UzrmRC8/uAh8CRQVvV/s8Mh93wM3gmp2Co+/rygxBRU/TXGJgg4sF01aOV mT9HbIT6Ghc3Uu4ArD+1k6viiOKPFO3xXz+EUy= -----END CERTIFICATE----- </pre>
jwtrS256.pub	<pre> -----BEGIN PUBLIC KEY----- MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCgEA94ogBPx7P0hQQtg901xi1 mUR2n0ORLoLnT1T3xN33ppiM1vCj14UougZNO41H/aJdgt0uNs5tO1jtp0zYLcUN Eq11NP7sWAe45h411QLX0jcnJEM16BmtSG6ae01+Y61dr63TpcV7WvTtIKD06v00 aNOjxMVJ0uFw0dlyQSySveczkZdr18N6k2H3z0t9/TFQeBoFdwNOKCpaRuV+EOaM DmVuC3rZ/SCWms7H/PJPGyIa8juku8yTM4HPOVj97GJKD2xPpYzYybVqEnLwFik QSB+EB9/eyrYeTrn2rKrPLopS0YFvUgWgz8Di1ZADYS9cJTMbiHQob+wtakmZeu hIa4NuXFV+99atVucLXPectjw2Unr2VJcszjIH1/4Y1TX+g1uoHEZVnBezRR8D6M +QIUOeaJmOEi8RW9DAYfRk3v6IUjoc7GB/NBT/HTbWlmb5v3tLtiN1V7wqyJQt2L BZcv1i380Xv025XPAbjv+Pg+emZHQyYy41ioTF7TUOm8iG2Yae6JPej261B8eDWL BJITNC924B4IaW3Ufj8s8JsrjBjJg1U0xxM7vpY27HpMmEZob94QYwXUi8Jy2O wVa23de/mK5aVquIBnu660kaYonDas66oGpPzrx1NctjfySeXqd9BhtMV7e19EI3 HjNoBaJ5ZpY7KRzRhfwKs18CAwEAAQ== -----END PUBLIC KEY----- </pre>
jwtrS256.key	<pre> -----BEGIN RSA PRIVATE KEY----- MIIEKQIBAAKCAgEA94ogBPx7P0hQQtg901xi1mUR2n0ORLoLnT1T3xN33ppiM1vCj 14UougZNO41H/aJdgt0uNs5tO1jtp0zYLcUNEqI1NP7sWAe45h411QLX0jcnJEM1 6BmtSG6ae01+Y61dr63TpcV7WvTtIKD06v00aNOjxMVJ0uFw0dlyQSySveczkZdr i8N6k2H3z0t9/TFQeBoFdwNOKCpaRuV+EOaMDmVuC3rZ/SCWms7H/PJPGyIa8juk u8yTM4HPOVj97GJKD2xPpYzYybVqEnLwFikQSB+EB9/eyrYeTrn2rKrPLopS0Y FvUgWgz8Di1ZADYS9cJTMbiHQob+wtakmZeuHia4NuXFV+99atVucLXPectjw2Un r2VJcszjIH1/4Y1TX+g1uoHEZVnBezRR8D6M+QIUOeaJmOEi8RW9DAYfRk3v6IUj oc7GB/NBT/HTbWlmb5v3tLtiN1V7wqyJQt2LBZcv1i380Xv025XPAbjv+Pg+emZ HqyYy41ioTF7TUOm8iG2Yae6JPej261B8eDWLBJITNC924B4IaW3Ufj8s8JsrjBj jJg1U0xxM7vpY27HpMmEZob94QYwXUi8Jy2OwVa23de/mK5aVquIBnu660kaYonD As66oGpPzrx1NctjfySeXqd9BhtMV7e19EI3HjNoBaJ5ZpY7KRzRhfwKs18CAwEA AQKCAgEA8vFc+4fQbgtKxVHmGjTWUdYfk2QXum5At1tt0IkIXThzAYCrDcpI6cnf guz1+seF+qmY8bEnAJViqQ+CG2mJqWAHudT50exaS37a1nQz3NyqySwte923c58Z fBwANbXdfOfdVvy6Uz/+lcc9TO00zDY9JKF8EruvPkvRmKV1+Wkv83P02QLBcmT DwIQZK5BjTzEPTdyXXkzoOEBGpsr+fUSS7IGv6d7weGv5VDq0nZme9aNsLwtVyK yp30OHE+vmR060SmVemHR+M1LUXstD1E9+BtXyLAERPbLFPjD9o9dV8MhXUOgZpz JnbbIpnvgcN2YX2xj1m7Lv36DcQQZLU0SUCBK+3opsGkBP2dG4YSZUotcRvITM/ gRRjlaaXHFkoKckxGY8OUZ6BQMjww3bq5wac5JATQE608Ts45D/+sIbG1Ght/n3e 11PXArcTjYBfYfyTOTHV9syHcgHsrgfQzcuo4CLLSyqELoDLeoiAdeEY7S8WYmL oLp0ZY2TgYGRGOJ2HUtbqeaC9Jz9B3Ia4MaQKhr4x6rCgSgNCG/xKp1cJzYqcs/ Mmgx4F6A3Ahet4Sfmb6yXeo2w2RSW3TWP8M0dih9AExB6SUF47Byms7B9RIEZA41 5x+VqcvQ8dT+B77XvBWJf0duxQT6ERqcdFvTKuX109sDLg9ogEGGyEBAPzxj/Cv Rlz69ActyEYJpltdq11cMpyGRI4ZrdGVzU+4KPFgtgPHPyJh5gzIUSqRf380M6Y SinEAjyY0fN3fYsfJ6L449NGycJ7mAAjqu/JSb6QeGLbTz6+b014aqE+Zd1ZQSQ w4yANWxw9w1E5JKVTya5UT2qeACQmIbyKw8iEUr5EeEwh7p4wgzakfU5m8ygg7 3DwW9u6npIRMscoSiw3JLVk0j0fDHnEi10iObsuxN7meVWDCaocCwU2WSPmiULhL tSLP0qm8BZ7eI+ZrkYhfQFERSn8RXi6bU51x26YB5Emt3wnDbP6b7yJfBKuIBNTA Xi8hcKrXmhX8FJUCggEBAPqH2Kc3BEeiFcrXKbEfoAK3k400Ebi2hsK9YWPQgnx </pre>

```
C7/LoQhS3JmJ/exdrKxyJ1TW/WfAVm8PpdyloPEeXI5Vp6TPi5FERapSbNTJFTI6
WmyA/m0BSGjUSV3mC/Fm1M0Y3mfG0cpLFeO+8LwwY2xv0jbWCaD+zGggyBc07kAk
7RjXokca+Q3Va5dUrZR4DqH7M/A7PDPfRt6Kg6klwsiATyE5caJksqV1x0Q7Jd+9
atvdbiSExx40B1ZM38PK0whPZ8PEhfVIOYq5feAmemsGKyFpqSh6Lxot/oh2fWo1
/CjNgHQq1jCGLaIJ45i9UxdNCVsA48/dpR09nGBx0WE6Yo6RwnUMvPDiJy26zBKJ
Rb8xyGQPA1g1GjIvGW8Faiqa0G+FaMnJog9BfWQPYH+nBS7ZOXCMAk15qm+0RRaC
CFos9YOCggEBAllogsrL4RwNIBqVujSH6QPu0jML6uq1jRyx+RHFCg71d1U9sVfo
YrgPuGawxila/vmIZTbFHo0Rjw4jBc/ZsDgdALLjZpBJWtBdHvXiSC7hbH/wLExQ
XvGgFpsR7N1UkH/STsJW1kZZ645yzinQ3/ownItkuKmb/O7U4wYX1zJstx0fZONk
v/8vQyH0SvwmHOMEXGxj8ejigYxU1s5Gq+66eBKxqMGh9LnP/dde+McHs5h/1Y2d
fVY9ByjGhAYspBumVEwVXVLTALUkECVkt5i75nHwCGwNjw8iSEbn+FvgqNiNT41
vQU5DcszLtN4Lkvx4EGsiXVysP8pqGckB/MCggEADok3teRgiQApnbiDGzJICXtd
/pwU3dgzXkLuMI1v/DNcllyY1qPoNZsghTnC/Fh6NejeJlAjw0hYu9lopEtZynzp
EkOhmuBocKkNyTKrRRVXVTZQdIWT30vebshCmbEV5xRDgzZ5oqyqS2ek0T71buea
kDWms+YDrIb4IrgcWMSVETwp7RbRCzVkvG/uQnTi8Mof0+HBdK4PDnSdx0v31YpV
5mAB2BD+V7iAEOs9e9PTZwpvd/S1VcxGgo5kRE00ZAaBP+M0GqkM82xDNfxKsO9/
8VvlcCreYufa/viZ8efW6AFvyh9cc+otroLKuy1XR4jbYUsCqxx4Q0dddd7w3LQ==
-----END RSA PRIVATE KEY-----
```

7.2 Archivos de configuración

kong.conf	<pre># ----- # Kong configuration file # ----- admin_listen = 0.0.0.0:8444 http2 ssl proxy_listen = 0.0.0.0:443 http2 ssl reuseport backlog=16384 #----- # DATASTORE #----- pg_user = kong # Postgres user. pg_password = kongTFM # Postgres user's password. pg_database = kong # The database name to connect to.</pre>
.env	<pre>PORT=1337 NODE_ENV=production KONGA_HOOK_TIMEOUT=120000 DE_ADAPTER=postgres DE_URI=postgres://konga:kongaTFM@localhost:5432/konga KONGA_LOG_LEVEL=debug TOKEN_SECRET=TFM_JWT_t0k3n! SSL_KEY_PATH=/opt/konga/certs/selfsignedTFM.key SSL_CERT_PATH=/opt/konga/certs/selfsignedTFM.crt NODE_TLS_REJECT_UNAUTHORIZED=0</pre>

7.3 Definición endpoint con Swagger Editor

```
swagger: "2.0"
info:
  description: "Aplicando seguridad a una API REST con JSON Web Tokens"
  version: "1.0.0"
  title: "TFM - Aplicando seguridad a una API REST con JWT"

host: "srv-tfm.edu"
basePath: "/v1"
tags:
- name: "auth"
  description: "Autenticación"

- name: "v1"
  description: "Usuarios y Roles"

schemes:
- "https"
- "http"
paths:
  /auth/registro:
    post:
      tags:
      - "auth"
      summary: "Login Usuario"
      description: ""
      operationId: "CreateUser"
      produces:
      - "application/xml"
      - "application/json"
      parameters:
      - in: "body"
```

```

    name: "body"
    description: "Crear Nuevo Usuario"
    required: true
    schema:
      $ref: "#/definitions/Registro"
  responses:
    "200":
      description: "Usuario creado correctamente"
    "400":
      description: "Usuario ya existe"

/auth/login:
  post:
    tags:
      - "auth"
    summary: "Login Usuario"
    description: ""
    operationId: "LoginUser"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - in: "body"
        name: "body"
        description: "usuario y contraseña"
        required: true
        schema:
          $ref: "#/definitions/Login"
    responses:
      "200":
        description: "Devuelve Token"

      "400":
        description: "Usuario no Existe"

/v1/all:
  get:
    tags:
      - "v1"
    summary: "Ver el contenido público"
    description: ""
    operationId: "showContent"
    consumes:
      - "application/xml"
    produces:
      - "application/json"
    responses:
      "200":
        description: "Contenido público todo el mundo puede consultar"

/v1/moder:
  get:
    tags:
      - "v1"
    summary: "Ver el contenido con rol Moderador"
    description: "Ver el contenido con rol Moderador"
    produces:
      - "application/json"

    parameters:
      - name: "JWT"
        in: "header"
        required: false
        type: "string"

    responses:
      "200":
        description: "Mostrando contenido para Rol Moderador"
      "403":
        description: "Requiere rol Moderador | Requiere Token"

/v1/admin:
  get:
    tags:
      - "v1"
    summary: "Ver el contenido con rol Admin"

```

```

description: "Ver el contenido con rol Admin"
produces:
- "application/json"

parameters:

- name: "JWT"
  in: "header"
  required: false
  type: "string"

responses:
"200":
  description: "Mostrando contenido para Rol Admin"
"403":
  description: "Requiere rol Moderador | Requiere Token"
"404":
  description: "Pet not found"

/v1/user:
get:
tags:
- "v1"
summary: "Ver el contenido con rol User"
description: "Ver el contenido con rol User"

produces:

- "application/json"
parameters:

- name: "JWT"
  in: "header"
  required: false
  type: "string"

responses:
"200":
  description: "Mostrando contenido para Rol User"
"403":
  description: "Requiere rol Moderador | Requiere Token"

delete:
tags:
- "v1"
summary: "Borrar Usuario"
description: ""
operationId: "deleteUser"
produces:
- "application/json"
parameters:

- name: "JWT"
  in: "header"
  required: false
  type: "string"

- name: "username"
  in: "body"
  description: "usuario"
  required: true
  schema:
    $ref: "#/definitions/delUser"
responses:
"404":
  description: "Usuario no encontrado"

"403":
  description: "Requiere rol Admin | Requiere Token"

"200":
  description: "Usuario Eliminado!"

/v1/email:

```

```

put:
  tags:
    - "v1"
  summary: "Actualizar Email de un usuario"
  description: ""
  operationId: "updatePet"
  consumes:
    - "application/json"

  produces:

    - "application/json"
  parameters:

    - name: "JWT"
      in: "header"
      required: false
      type: "string"

    - in: "body"
      name: "body"
      description: "Pet object that needs to be added to the store"
      required: true
      schema:
        $ref: "#/definitions/updateEmail"
  responses:

    "404":
      description: "No existe usuario"
    "403":
      description: "Requiere rol Admin | Requiere Token"

    "200":
      description: "Email Actualziado!"

securityDefinitions:
  JWT:
    type: "apiKey"
    in: "header"
    name: "x-jwt-token"

definitions:
  Login:
    type: "object"
    properties:
      username:
        type: "string"

      password:
        type: "string"
  xml:
    name: "Order"

  delUser:
    type: "object"
    properties:
      username:
        type: "string"

  updateEmail:
    type: "object"
    properties:
      username:
        type: "string"
      email:
        type: "string"

  Registro:
    type: "object"
    properties:
      username:
        type: "string"
      password:
        type: "string"
      email:
        type: "string"

```

```
roles:
  type: "string"
  enum: [Admin | Moderador | User]
xml:
  name: "Order"
```