Hecho por: Omar Emilio Durán Cortez // 26780 (C8)

Assignment 4

ICE191 Software Architecture / Cloud Computing

1. Describe the concept of throttling in APIs. **15 points.**

Por definición formal de acuerdo con Microsoft (2022) el *throttling* en las APIs consiste en reducir la cantidad de solicitudes que puede procesar una API en un periodo de tiempo para prevenir el uso excesivo de recursos. La misma documentación menciona que hay dos formas de que se cause, la primera es porque un usuario e particular está haciendo demasiadas solicitudes a alguna API en un corto periodo de tiempo, la segunda se causa por un uso masivo por parte de todos los usuarios que puedes usar esa API.

En el caso de que se presente el primer caso, algo comun que se puede recibir si se hacen demasiadas es el código 429 HTTP, es cuál según documentación oficial de Mozilla, es “Too Many Requests” diciendo que se han hecho muchas solicitudes. Esta respuesta puede venir con algún tiempo de espera antes de volver a intentar.

Según documentación oficial de AWS de Amazon API Gateway, en el caso de sus servicios para creaciones de API para desarrolladores, ya existen límites predeterminados que no puede cambiar el usuario. Si se fuese a alojar la API “on -site” tú mismo tendrías que lidiar con estos problemas por ti mismo o tu servidor que provee la API se puede caer al estar recibiendo muchas solicitudes.

Existe una técnica que se puede usar en conjunción a esta, se llama “rate limiting”. Mientras que *throttling* es para hacer un control de número de solicitudes por usuario en X tiempo, según Adobe, “Rate Limiting” es más para cuando se trata de tiempo real.

O sea, podría hacerle *throttle* a una API solo dándole 10000 accesos por semana a un usuario o haciéndolo lento en sus solicitudes si hay una alta cantidad de solicitudes por todos los usuarios, mientras que le haría *rate limit* si excedió una razón de solicitudes/segundo (ej: 5 máximo por segundo), forzándolo a hacer menos intensamente sus llamados.

2. Describe the concept of pagination in APIs. **15 points.**

De acuerdo con Atlassian (s.f), la paginación de APIs es el acto de convertir en páginas los resultados de alguna solicitud a una API. ¿Pero cómo que paginas? Haré una comparativa. Digamos que estamos en Google y hacemos una búsqueda, en ese caso habrá una página con resultados, luego otra, y otra…. Y así. Con sitios web dinámicos se podría hacer una lista infinita de resultados y solo ir mostrando parcialmente los resultados, pero con un sitio web estático se mostraría todo, y millones de resultados podría ser algo excesivo para procesar.

Esto mismo ocurre con APIs, para facilitar el manejo de resultados, algunas API paginan sus resultados. Es a la discreción del desarrollador de la API implementar la forma en la que se haría la paginación. De acuerdo con abstractapi (2021) hay dos formas principales, una es con límites y *offsets*, aquí en los *query parameters* (?param=valor) se especifica a partir de que página iniciar los resultados, y cuantas páginas se desean.

El siguiente método aportado es en base a llaves, en este caso el parámetro del *query* sería un valor para buscar en una tabla de SQL, mucho cuidado si se implementa así porque podrías hacer tu API muy vulnerable a ataques de inyección SQL y regresar datos que no debería ver el usuario.

El tercer método se parece al primero, pero en vez de escoger la página de donde se inicia, se escoge a partir de que elemento (ID) se empezará a mostrar los resultados. O sea que podría poner un límite de 50 elementos y pedir que me muestra a partir del elemento con un ID tal.

3. Describe the concept of callback function. **15 points.**

Las funciones *callback* son parte de la magia que nos traen los lenguajes de programación. Por definición formal según Mozilla (s.f) estas funciones son funciones que son utilizadas como parámetros de otras funciones. O sea, que, si tenga una función que me da el resultado de la suma de dos más dos, y uso esa función en otra que manda números a un archivo csv, entonces la función que hizo la suma sería la *callback.*

Cuando se usa una función como argumento su código se correrá, y, se puede utilizar su resultado como parámetro ya "normal" para la función no *callback*, o simplemente aprovechar la llamada para que haga otra cosa.

De acuerdo con la misma documentación, una forma usada para este tipo de funciones es usarse en código asíncrono (o sea, que no sigue un orden secuencial). Un ejemplo famoso son las promesas de Javascript. Estas reciben dos argumentos, ej:

```
const promesaLlamadora = new promise((resolve, reject) => {resolve(códigoDeResolver)});
```

resolve es el parámetro de una ejecución exitosa, reject es en caso de fallo.

```
promesaLlamadora.then((valor) => másCodigoAquí))
```

Al usar ".then" "valor" será el resultado de lo que nos de resolve, el cual es la función *callback* dentro de la promesa. Claro, si es que sale exitosa la promesa.

4. Describe the concept of cold start in AWS Lambda. **15 points.**

AWS Lambda es el servicio de AWS que permite correr código en la nube y puede usarse en conjunción a otros servicios de AWS.

Un concepto importante en la teoría de este servicio es el *cold start.* Este se refiere según James Beswick (2021) al tiempo en el que una lambda tarde en conseguir tu código y crear el ambiente de ejecución. Esto es parcialmente análogo a encender un carro en frío, es un proceso más difícil porque las temperaturas hacen que haya menos energía y se requiera más para arrancar y pueda ser que se tarde más en arrancar si la batería no está a la altura.

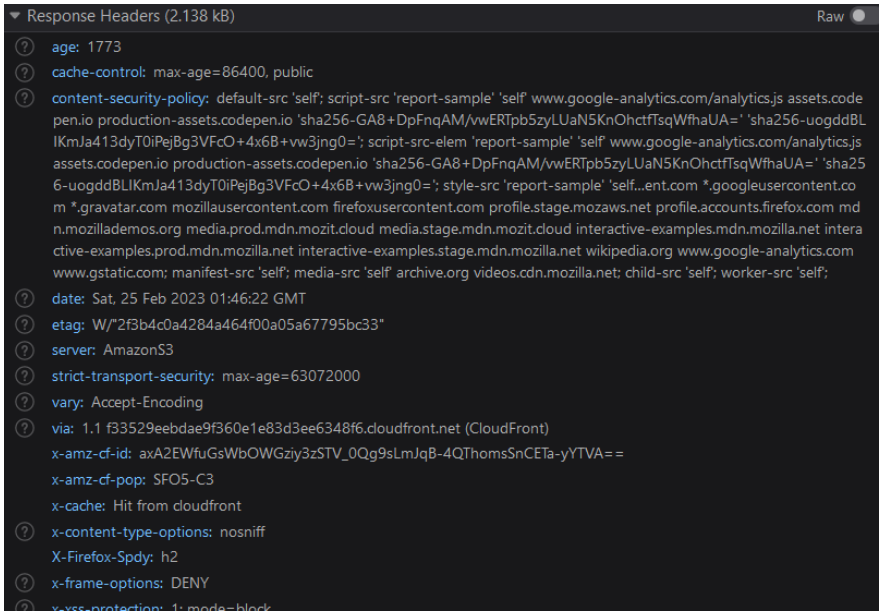
¿Pero que tiene eso que ver? Como documento del mismo autor menciona hay dos partes del proceso, primero se descarga el código y se mete en una cubeta de S3. Luego crea un ambiente de ejecución con la memoria y recursos específicos para el lenguaje de ejecución de la lambda. Al acabar este proceso se procede a la ejecución del código. Afortunadamente, este tiempo no se cobra (menos mal).

Cold start no es un concepto propio de AWS, ni un sinónimo, hasta en Azure, según documentación oficial de Microsoft también se le llama así. Lo que dice aquí que es que la causa de este fenómeno es que las funciones (la equivalencia a lambdas de AWS) no han sido usadas en algún tiempo.

Cabe destacar según Beswick (2021) que cualquier actualización de código va a provocar otro *cold start* porque se tiene que volver a descargar el código. Otro causante puede ser escalación de recursos, dado que se necesitan crear nuevas instancias de la lambda, se requiere crear el ambiente de ejecución para cada una de ellas.

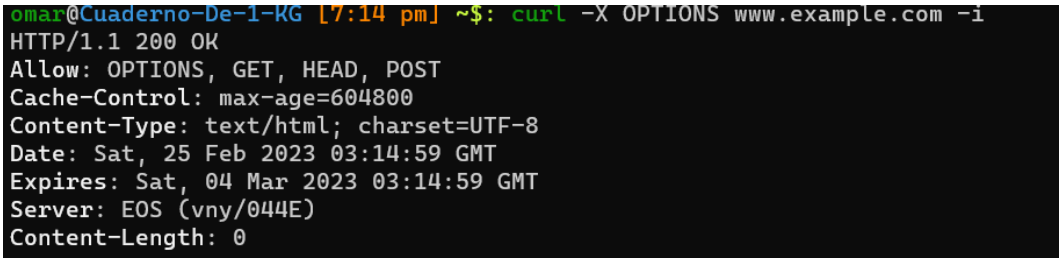
5. Describe each HTTP methods. **15 points.**

De acuerdo con documentación oficial de Mozilla (s.f) los métodos HTTP son los siguientes:

- GET: De acuerdo con la misma documentación, el método GET es para obtener un recurso de algún sitio web. Se recomienda (pero no necesariamente obliga) que no se envié información en el cuerpo (*body)* de la solicitud dado que podría ser rechazado o ignorado, el método GET no es para ese tipo de cosas.
    - Si se quiere especificar en alguna API lo que se desea deberían usarse los parámetros en el *query* (o sea: www.xd.com/id?=222)
- HEAD: De acuerdo con la misma documentación, HEAD es muy parecido a GET, de hecho, es lo mismo pero la diferencia se encuentra en el nombre, la cabeza, o más bien, los *headers.* HEAD solo envía *headers* idénticos a los que se mandarían en una solicitud GET, pero sin el cuerpo *(body).*
    - Por esta razón, una solicitud de HEAD a un sitio web no nos ayudaría a visualizarlo, pero sí nos otorgarían metadatos importantes como los de esta imagen:
    - 

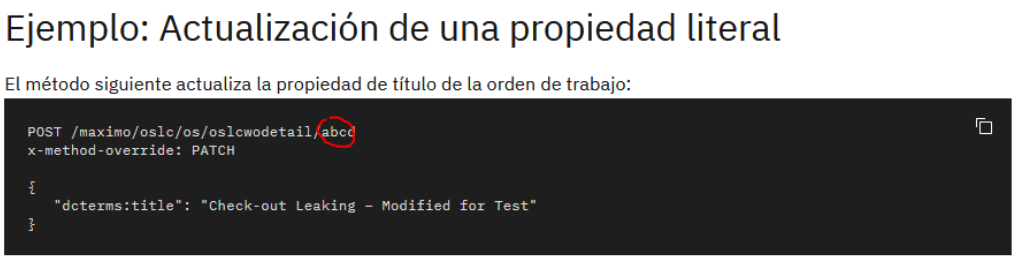
    - Esta imagen es de la pestaña de red de Firefox, mostrando los *headers* de respuesta de la página de la documentación de este método, incluye datos como el servidor de origen, vía en la que se nos transmitió, entre otros.

- POST: Mientras que GET y HEAD son para pedir datos, POST es para dar datos, un ejemplo muy sencillo es llenando una forma en un sitio web. De acuerdo con la misma documentación, POST envía datos al servidor de donde proviene el sitio y se puede especificar el tipo con el *header* “content-type”
    - Los tipos de contenidos que se pueden usar deben especificarse en base a un estándar referido como “MIME type”. Esto los define el *Internet Assigned Numbers Authority.* Su sintaxis es *tipoDeContenido/Formato* (ej: aaplication/A2L. text/rtf).
    - Este detalle mencionado, según la misma documentación se puede controlar en el HTML con el atributo “enctype” en una forma, y “formenctype” si se usa una etiqueta *input*.
- PUT: PUT es muy parecido a POST. Según W3 (s.f), PUT a diferencia de POST es idempotente, o sea que puede hacer N veces la solicitud y siempre pasará lo mismo ¿ok pero que hace PUT?
    - PUT es para la creación de recursos al igual que POST según el mismo documento de W3, en caso de que ya existe el recurso de actualiza
    - Las diferencias entonces de acuerdo con el mismo documento son según Geeks for Geeks (2021) que PUT debe usarse más para actualizaciones que creaciones (no es que no se pueda creaciones). PUT es para modificaciones de recursos en alguna colección (ej: modificar lista.txt del directorio /hola/) mientras que POST es para añadir otro recurso, hacer algo de esa índole con POST como si fuera PUT dará error diciendo que no existe tal recurso.
    - El usuario de stackoverflow Brian Bondy da un criterio relevante para decidir si usar PUT o POST y está relacionado con lo dicho al final del punto previo, si el servidor decide donde poner los datos de la solicitud es mejor usar POST, sino entonces PUT. Además, recomienda
- DELETE: Mientras que los previos son para creación y obtención de recursos, DELETE es para el borrado. De acuerdo con la misma documentación este método es para borrar recursos. A diferencia de GET y HEAD si se puede poner data en el *body*.

- CONNECT: Conexión es especial, no es para enviar, no es para recibir, sino para ambos. De acuerdo con la misma documentación esta es para iniciar una comunicación entre quien solicita y el solicitado.
    - Según W3cubDocs (s.f) un ejemplo en el que se usa es en el acceso a sitios que usan SSL (sitios que usan HTTPS).
    - Según el RFC 2616 de la Internet Engineering Task Force, este método utiliza proxys web (servidores intermediarios) para que se conviertan en túneles para el uso de SSL
- OPTIONS: De acuerdo con la misma documentación, OPTIONS es para solicitar las opciones (métodos) de comunicación permitidas respecto a un servidor. No acepta datos en el *body.*
    - 

    - Al hacerse la solicitud saldrían los datos que se ven en pantalla (el relevante siendo el segundo renglón). Se uso la utilería curl para hacer un *request* personalizado, -X es para escoger el tipo de *request*, e -i es para que se incluya el *header* en el output, necesario para que salga este resultado, si no se usa, no sale nada
- TRACE: TRACE es un método enfocado hacia desarrolladores, se usa según PortSwigger (s.f) para propósitos de diagnóstico. Es como un eco, dará de respuesta los mismos *headers* que le enviaste.
    - La misma fuente recomienda no utilizarlo en servidores de producción (el contrario a ambientes de desarrollo) porque podría dar información adicional a la que está recibiendo como *headers* con datos de autorización
- PATCH: De acuerdo con la misma documentación, PATCH es para edición de contenidos, más específicamente, para actualizaciones. Hasta el mismo recurso le hace analogía al UPDATE en CRUD.

- PATCH, en su solicitud contiene instrucciones de que se debe modificar en un recurso
- Aquí está un ejemplo de documentación de IBM



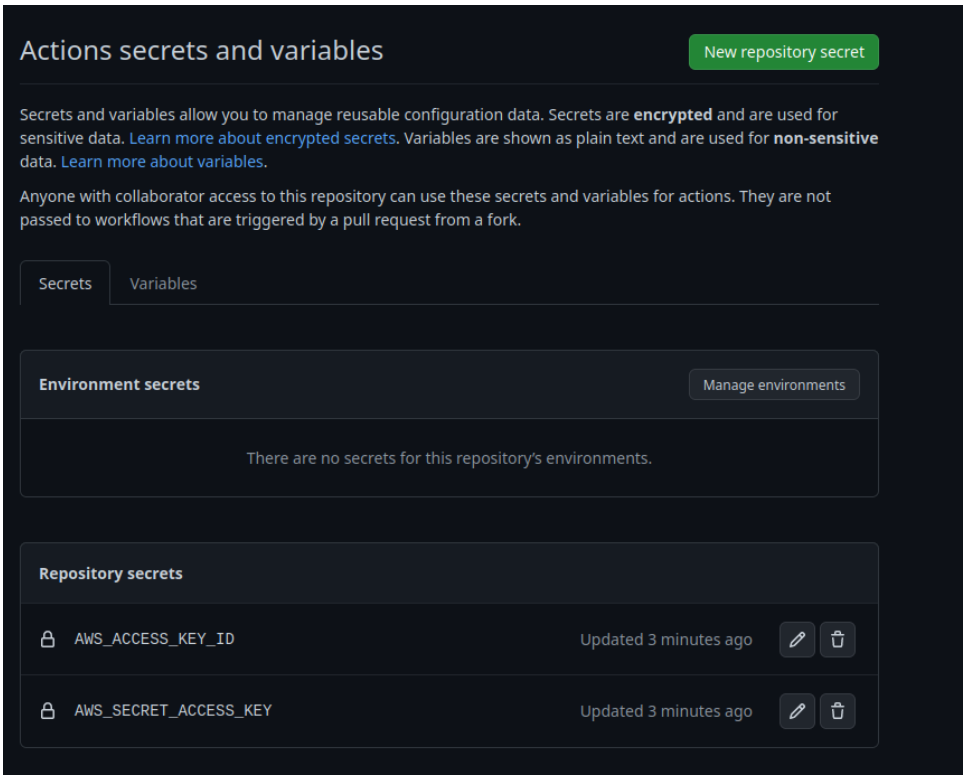
- La documentación de donde proviene esta imagen dice que propiedades que no fueros especificadas en la solicitud no fueron modificadas.
  - Por ende, una buena aplicación de esta solicitud es configuraciones de usuario, donde en un menú de puede que no se cambien todas a la vez, sino algunas.

6. Describe how you can automate a deployment of a static website to S3. **15 points**

Me basaré en la guía de Seok Jun Hong (2021) para este procedimiento.

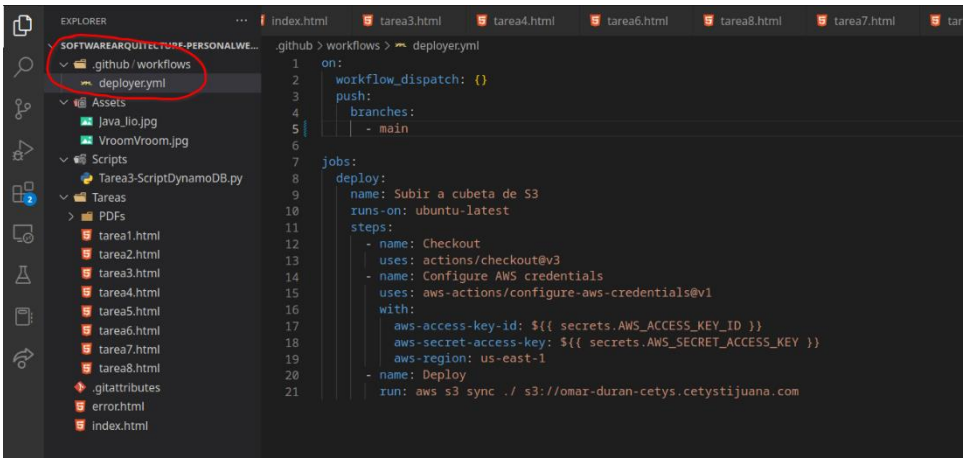
Los primeros pasos ya fueron hechos en otras tareas, create cubeta de s3, subirle archivos, hacerlo de acceso público, y convertirlo en sitio web estático.

El siguiente paso es ir al repositorio de GitHub donde están los archivos de tu sitio (y si no hay entonces haz uno) y ve a la sección de secretos en la sección de configuración



Aquí pondremos las credenciales de nuestro usuario de IAM de la clase (en un escenario más profesional hubiera creado uno meramente para esto con solo permisos para interactuar con s3).

Ahora dentro de nuestro repo debemos crear este archivo dentro de la ruta señalada en la imagen



Deployer.yml es un archivo (nombre es a libertad propia) con instrucciones para GitHub actions de qué hacer con el repo, al inicio "workflow_dispatch" es para que pueda hacer

el deploy manualmente por si las dudas, líneas 1,3,4, y 5, son para que se haga deploy automático cada vez que se haga push al Branch de nombre main.
Línea 7 es el trabajo que se hará, le puse nombre en la línea 9, la 10 es escoger el runner de GitHub, escogí que lo corra la plataforma misma (es gratis hasta cierto limite anual). Líneas 12 y 13 son para hacer checkout, que se descargue el código para subirlo. Líneas 14 a 19 son para usar credenciales de AWS provenientes de los secretos del repositorio. Líneas 20 y 21 corren un comando, en este caos el comando es el de sincronización de computadora a la cubeta de AWS.

Al acabar el archivo debemos hacer push al Branch main, con esto automáticamente cada vez que se haga un cambio se hará un deploy a la cubeta de S3.

Hola, este es el sitio web de Omar Duán, aquí encontrarás mis tareas las cuales estan en cada uno de los hipervinuclos ahí debajo como PDFs embedididos. Pasatela bien

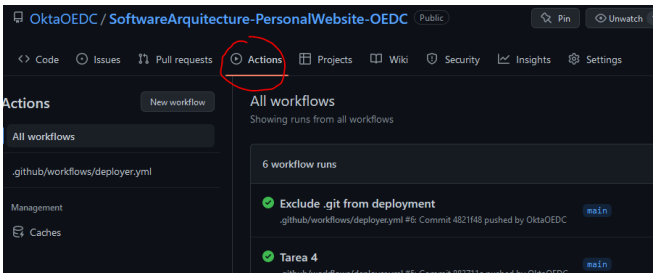
- Tarea 1
- Tarea 2
- Tarea 3
- Tarea 4
- Tarea 5 [Proximamente]
- Tarea 6 [Proximamente]
- Tarea 7 [Proximamente]
- Tarea 8 [Proximamente]

Vroom Vroom!!

Accede al código fuente de esta página web aquí :)

Boom, con esto mi sitio fue actualizado sin yo tener que abrir una terminal, hasta parece magia.

Para ver si funcionó hay que ir a la sección de “Actions” en nuestro repositorio y buscar el workflow de nuestro archivo:



7. Read the Real-world Engineering Challenges #8: Breaking up a Monolith article and write a summary and opinions about it. **10 points.**

Esta lectura trata acerca del proceso de migración de la plataforma de Khan Academy, sitio de aprendizaje, de un modelo de desarrollo monolítico, a uno basado en microservicios.

Esto fue muy complejo para los desarrolladores involucrados, todo era un monolito en python 2 con todas sus APIs REST ahí. Este proceso fue altamente acomplejado porque Python 2 estaba terminando su soporte, así que también surgió la necesidad de cambiarse de lenguaje, el cual fue a GO debido a su mejor rendimiento según el artículo, mientras que para las APIs utilizaron GraphQL (un lenguaje para querys en APIs).

En vez de hacer un producto mínimo hicieron una experiencia mínima, o sea, que debido que no que iba a haber nada nuevo por sí solo no tendría sentido crear un MVP, sería como darle menos al sitio web, así que se enfocaron en lo menos que debían portear para darle el funcionamiento al sitio en base a microservicios, esto fue casi el 95% del esfuerzo requerido.

El proceso tuvo muchos cambios de desarrolladores con el tiempo, al inicio eran pocos, luego subieron mucho, y de nuevo bajaron conforme acababa el proyecto, pero muchos aprendieron tecnologías nuevas en el proceso y estaban acostumbrados a usar otras. Así que, aunque se invirtió tiempo extra fue por un lado positivo. Eso sí, se fue migrando primero lo pequeño, y luego lo grande, incluyendo los bugs para no invertir tiempo mayor al necesario.

Yo opino que esto es un buen artículo para leer para cualquiera que quiera aprender cómo es que ocurre el desarrollo de microservicios en el mundo real, esto es un muy buen ejemplo y algo que nunca pensé que pasaba detrás de escenas hace años cuando usaba Khan Academy hace años (y hoy aun).

Convertir una arquitectura de software a otra puede ser un proceso muy tardado y creo que muchos dolores de cabeza se pueden ahorrar si se plantean bien anticipadamente, como dicen algunos, el desarrollo de software es más socializar que programar, y esa socialización incluye buena toma de decisiones, organización y definir cuál es el mejor camino para un software.

Hecho por: Omar Emilio Durán Cortez // 26780 (C8)

Recursos Utilizados:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
  - Documentación de Mozilla para métodos HTTP
- https://www.iana.org/assignments/media-types/media-types.xhtml
  - Documentación de IANA respecto a tipos de media en HTTP
- https://www.rfc-editor.org/rfc/rfc2616#section-9.9
  - RFC 2616 respecto a métodos HTTP
- https://linux.die.net/man/1/curl
  - Manual de utilidad curl
- https://portswigger.net/kb/issues/00500a00_http-trace-method-is-enabled
  - Documentación de Portswigger de método HTTP Trace
- https://www.ibm.com/docs/es/mam/7.6.0.8?topic=SSLKT6_7.6.0.8/com.ibm.mif.doc/gp_intfrmwk/oslc/c_oslc_patch_method.htm
  - Documentación de IBM respecto a método HTTP PATCH
- https://www.geeksforgeeks.org/diffrence-between-put-and-post-http-requests/
  - Artículo de Geeksforgeeks respecto a métodos PUT y POST y sus diferencias
- https://stackoverflow.com/questions/630453/what-is-the-difference-between-post-and-put-in-http/630475#630475
  - Diferencias entre PUT y POST discutidas en un post de StackOverflow
- https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/
  - Descripción de AWS de cold start en servicio de lambdas
- https://azure.microsoft.com/es-es/blog/understanding-serverless-cold-start/
  - Documentación de Azure respecto a cold starts
- https://learn.microsoft.com/en-us/partner-center/developer/api-throttling-guidance
  - Documentación de Microsoft respecto a *throttling* de APIs
- https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html
- https://helpx.adobe.com/coldfusion/api-manager/throttling-and-rate-limiting.html
  - Artículo de Adobe respecto a límites y *throttling* de APIs
- https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

    - Documentación de Mozilla respecto a callbacks
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
    - Documentación de Mozilla respecto a promesas
- https://www.abstractapi.com/api-glossary/api-pagination
    - Descricpiónde AbstractAPI respecto a paginación de APIs
- https://faun.pub/deploying-website-to-aws-s3-w-github-actions-279998db5dae?gi=3379aa8dd7e6
    - Guía de Seok Jun Hong para usar github actions para hacer deploys a S3 con cada push