Assignment 4 26 de Febrero del 2023

Daniel Agraz Vallejo Cloud Computing

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

El concepto de Throttling en APIs es una técnica utilizada en la producción de una aplicación que busca limitar la cantidad de solicitudes que se pueden realizar a una API en determinado periodo de tiempo. Su objetivo principal es prevenir que la API sea abrumada por demasiadas solicitudes a la vez. También se considera una política que el servidor aplica y el cliente respetar.

Por otro lado, su contraparte Rate-Limiting de una API, es una técnica utilizada para limitar el número de solicitud que puede realizar un único usuario o aplicación durante un período determinado de tiempo. Con esta técnica se puede devolver un mensaje de eroror o un código de estado cuando el usuario o la aplicación supera el límite establecido.

Un ejemplo de API Throttling sería con la API de Google Maps, ya que utiliza un sistema basado en cuotas que limita el número de solicitudes realizadas a su API. A cada solicitud de API se le asigna un determinado número de “créditos” en función del tipo de solicitud realizada. Una vez agotada la cuota de créditos del usuario, la API devuelve un mensaje de error.

Algunas ventajas que presenta está técnica de API Throttling son las siguientes.

- **Mejor rendimiento de la API:** previene que la API sea desbordada por demasiadas solicitudes, a lo que se reduce el número de solicitudes que recibe el sistema, por tanto, puede mejorar el rendimiento general de la API.

- **Protege la API de Abusos:** ayuda a evitar que usuarios o aplicación maliciosas como bots sobrecarguen la API con demasiadas solicitudes, lo que cuida la integridad de la API.

- **Ayuda a Priorizar las Solicitudes:** establece priorizar las solicitudes que se pueden procesar en un momento dado, para garantizar que las solicitudes más importantes se procesen primero.

No obstante cuenta con sus respectivas desventajas:

- **Reduce la Experiencia de Usuario.** El uso de esta técnica puede resultar en tiempos de respuesta más lento de lo normal para los usuarios, lo que conduce a una experiencia de usuario más pobre.

- **Mayor Complejidad:** La implementación del API Throttling puede ser compleja e incluso requerir de infraestructura y herramientas externas.

- **Falsos Positivos:** Pueden provocar que se bloqueen o retrase solicitudes legítimas lo que puede dar lugar a falsos positivos.

- **Limita la Flexibilidad de Uso:** Esta técnica puede limitar la flexibilidad de la API y dificultar su personalización para satisfacer las necesidades de usuarios o aplicaciones específicas.

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

En el contexto de APIs, Pagination se refiere al proceso de fragmentar un conjunto de datos grande en pedazos más pequeños o en páginas que pueden ser solicitadas y extraídas por la API.

Esta técnica se emplea cuando la respuesta de una API contiene muchos datos que no pueden ser obtenidos o procesados a la vez, entonces la información se tiene que estructurar en páginas con una capacidad de resultados. Un ejemplo de Pagination es el motor de búsqueda de Chrome que cuando se realiza una consulta, obtienes un determinado número de páginas por pestaña, ya que cargar todos los resultados en

una sola página tomaría mucho tiempo en resolver la petición, así como cada solicitud de un cliente pondría una inmensa carga en el servidor, que tomaría tiempo compilar.

Otro ejemplo es, supongamos que tenemos un API Endpoint que devuelve una lista de usuarios: **GET /usuarios**. Por defecto, este Endpoints es capaz de devolver un listados de todos los usuarios del sistema, lo que sería un conjunto de datos muy grande, una vez que se almacenen muchos usuarios. Para evitar abrumar al cliente con demasiados datos a la vez, se puede implementar la paginación para devolver los datos en trozos pequeños.

Un enfoque común para la paginación es utilizar parámetros de consulta para especificar el tamaño de la página y el número de la página actual. Por ejemplo. **GET / users?page=1&size=20.** Esta petición devuelve los 20 primeros usuarios del sistema. Si hay más de 20 usuarios, la API puede incluir metadatos adicionales en la respuesta para indicar que hay más páginas disponibles.

Algunas ventajas que tiene emplear esta técnica de paginación son:

- **Mejor rendimiento de la API.** La paginación de la API puede mejorar el rendimiento de la API al reducir la cantidad de datos que deben transmitirse por la red y la cantidad de procesamiento que necesita el servidor.

- **Mejor Experiencia de Usuario:** Al devolver los datos en trozos más pequeños y manejables, la paginación de la API puede mejorar la experiencia del usuario y reducir el riesgo de bloqueos o tiempos de espera del lado del cliente.

- **Reducción de la carga del servidor:** La paginación de API puede ayudar a reducir la carga del servidor al distribuir la carga de procesamiento en múltiples solicitudes más pequeñas.

• **Mayor Escalabilidad:** La paginación de API puede mejorar la escalabilidad de una API al permitirle manejar conjuntos de datos más grandes y más usuarios simultáneos.

Del lado de las desventajas se encuentran:

- **Menor Coherencia de Datos.** da lugar a incoherencias en los datos, ya que se pueden añadir o eliminar datos entre páginas, lo que provoca que se devuelvan datos incoherentes al cliente.

- **Rendimiento más Lento:** permite que el sistema tenga un rendimiento más lento para los clientes que necesitan recuperar grandes cantidades de datos, ya que pueden ser necesarias varias solicitudes para recuperar todos los datos.

- **Limita la Flexibilidad:** limitar la flexibilidad de la API y dificultar su personalización para satisfacer las necesidades de usuarios o aplicaciones específicas.

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

Un Callback Function es una función que es capaz de pasarse como un argumento para otra función. Su principal propósito es permitir que una función llame a otra función cuando un determinado evento ocurre. Por ejemplo, en JavaScript existen la funciones de Orden Superior que se son capaces de recibir una función como argumento, además que dentro del lenguaje se considera como objetos de primera clase, tal que, una función puede ser un argumento para otra función, asignado a variables e incluso obtener una la llamada de una función como output.

Por parte de las APIs, un Callback Function suele ser empleado como un argumento para la llamada a una API asíncrona y se ejecuta, una vez que la llamada a la API es completada. El motivo del por qué es útil tener una Callback Function en una API es para permitir que el API del cliente sea capaz de manejar los resultados de la llamada a la API de manera asíncrona, sin interrumpir el hilo de ejecución principal. Un ejemplo de esto, es cuando un Callback API se manda a llamar, el proveedor de la API debe manejar la solicitud del cliente y ofrecer una respuesta que cumpla con lo se espera.

Un ejemplo de esto es una implementación en Javascript que realiza una solicitud a un API y que llama a una función Callback para obtener en consola, los datos de la respuesta.

```
// Se define la función que va a realizar una solicitud a la API
function apiRequest(callback) {
  // make an API request
  fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => callback(data))
    .catch(error => console.error(error))
}

// Se define la CallBack Function que pasara como argumento en apiRequest
function handleApiResponse(data) {
  console.log(data);
}

// se realiza la solicitud y se pasa como argumnto la función handleApiResponse
apiRequest(handleApiResponse);
```

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

Un Cold Start en el servicio de AWS Lambda, se refiere al incremento del tiempo de respuesta cuando una recién creada instancia Lambda gestiona su primera solicitud. Este estado de Cold Start ocurre cuando se recibe la primera solicitud después del despliegue de la instancia.

Después de que la solicitud haya sido procesada, la instancia se mantiene operando con el propósito de ser reutilizada para solicitudes futuras. El tiempo de vida de una instancia Lambda inactiva es de entre 5 y 7 minutos.

Para mitigar el impacto del Cold Start, AWS proporciona características:

- **Concurrencia Provisionada:** permite precalentar un número determinado de instancias de una función para garantizar que estén disponibles para atender solicitudes inmediatamente.

- **Warm-Up Scripts:** pueden ejecutarse periódicamente para mantener calientes los recursos de la función, lo que reduce la probabilidad de que se produzcan inicios en frío.

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

HTTP es un protocolo “Stateless”, dado que una vez que la solicitud inicial es completada, la comunicación Servidor-Cliente se pierde. HTTP te permite enviar recursos y datos a través de la web. Este protocolo tiene métodos HTTP (HTTP Verbs) que son usados para realizar diferentes acciones en un servidor web. A continuación se presentan estos métodos:

• **GET** : Es de los métodos más utilizados dentro del protocolo HTTP, se emplea para solicitar la representación sobre un recurso específico. Este tipo de método se utiliza solo para obtener datos.

El método GET está definido como "seguro" e "idempotente", lo que significa que múltiples peticiones idénticas deberían tener el mismo efecto que una única petición.

Un ejemplo de GET, es el siguiente:

```
app.get("/", (req, res) => {
    res.sendFile(`${__dirname}/signup.html`);
})
```

En este fragmento de código se da entender que cuando el usuario este en la raíz (/) de la página, se le va enviar los recursos de una página HTML para que el browser le haga render.

• **POST :** Al igual que GET, POST es un método de uso común dentro de HTTP, se utiliza principalmente para enviar una entidad a un recurso especifico, que causa un cambio en el servidor. Se emplea a menudo para crear o actualizar recursos en el servidor.

Cuando un cliente envía una solicitud POST a un servidor, envía una representación de una entidad en la carga útil de la solicitud al servidor para que sea procesada. El servidor realiza entonces la acción apropiada basándose en la información de la entidad y devuelve una respuesta indicando el estado de la petición.

Un ejemplo de POST es el siguiente:

```
app.post("/", (req, res) => {

    const fName = req.body.firstName;
    const lName = req.body.lastName;
    const email = req.body.email;

    let data = {
        members: [
            {
                email_address: email,
                status: "subscribed",
                merge_fields: {
                    FNAME: fName,
                    LNAME: lName
                }
            }
        ]
    };

    var jsonData = JSON.stringify(data);
    const url = 'https://us21.api.mailchimp.com/3.0/lists/bdcdf6f7c8';
    const options = {
        method: "POST",
        auth: "daniel16:Zcac1d69370fc2e192d526cedf336956d-us21"
    }
    const request = https.request(url, options, (response) => {
        if (response.statusCode === 200){res.sendFile(`${__dirname}/success.html`)}
        else {res.sendFile(`${__dirname}/failure.html`)}

        response.on("data", (data) => {
            console.log(JSON.parse(data));
        })
    })
    request.write(jsonData);
    request.end();
})
```

Aquí el cliente le pica al botón de enviar a un formulario, se realiza una operación de post en el servidor donde se emplea un API endpoint para indicarle donde se requiere que se haga la escritura de los datos.

• **PUT :** Busca actualizar un recurso existente en el servidor web. Generalmente, se emplea para actualizar un recurso con nueva información o reemplazarlo por un recurso nuevo.

El uso del método PUT debe tener efectos secundarios en el servidor, como la actualización de un recurso existente. No debe utilizarse para crear nuevos recursos, ya que el recurso ya debería existir en el servidor.

• **DELETE :** Se utiliza para borrar un recurso del servidor web como un archivo, registro, o otro recurso que forme parte del servidor.

El uso del método DELETE debe tener efectos secundarios en el servidor, como borrar un recurso existente. No debe utilizarse para operaciones seguras o para recuperar información. Además, las peticiones DELETE son idempotentes, lo que significa que múltiples peticiones idénticas deberían tener el mismo efecto que una única petición.

• **HEAD :** Este método solo solicita la información del apartado “Header” de un recurso en el servidor web, sin extraer el “Body” del recurso. A menudo se utiliza para comprobar el estado de un recurso y recuperar metadatos sobre el recurso, como el tamaño del recurso o la hora de la última modificación.

• **OPTIONS:** Describe las opciones de comunicación disponibles para cierto recurso. Este método permite que el cliente determine las opciones y requerimientos relacionados a un recurso.

- **CONNECT:** Solicita que el destinatario establezca un túnel de conexión con el servidor de origen de destino identificado por el objetivo de la solicitud y, si tiene éxito, a partir de entonces restringirá su comportamiento al reenvío ciego de datos, hasta que se cierre el túnel.

- **TRACE :** Este método realiza un bucle de retorno de mensajes a lo largo de la ruta hacia el recurso de destino.

• **PATCH :** Este método busca aplicar modificaciones parciales a un recurso, se emplea para realizar actualizaciones parciales a un recurso existente. Las peticiones PATCH no son necesariamente idempotentes, lo que significa que varias peticiones idénticas pueden tener efectos diferentes.

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

Para automatizar un deployment de un sitio web estático que esta contenido en una instancia S3, se tiene crear un Pipeline asegure la extracción del contenido fuente, que sea capaz de construir y desplegar toda la aplicación cuando exista un cambio.

Entonces para lograr este proceso, se requiere de un CI/CD (Continous Integration/ Continous Delivery) Pipeline que es un flujo de trabajo ágil, que se concentra en priorizar un constante proceso de entrega frecuente y confiable. Esta metodología de producción es iterativa, por lo que, permite a los equipos de DevOps escribir código, integrarlo, ejecutar pruebas, entregar versiones y desplegar cambios en el software de forma colaborativa y en tiempo real.

Generalmente en esta metodología se habla se 4 fases:

- **Build:** Esta fase forma parte del proceso de CI, que involucra la creación y compilación de código.

- **Test:** En esta etapa, se prueba el código. Se realizan pruebas unitarias, automatizadas y de integración. Esto con la finalidad de asegurar de que el código funcione correctamente.

- **Deliver:** El código fuente se aprueba y se envía a un entorno de producción. En esta etapa se automatiza en el despliegue continuo y solo se automatiza en la entrega continua tras la aprobación del desarrollador.

- **Deploy:** Los cambios se despliegan y el producto final hace la transición de desarrollo a producción. En CD, el código se almacena en repositorios y se mueve a un ambiente de producción supervisado.

La creación del Pipeline se llevo a cabo con la herramienta de desarrollador CodePipeline, que ofrece un servicio de despliegue continuo que automatiza las fases del desarrollo de un proyecto de software.

Dentro de la documentación de CodePipeline se nos ofrece una definición de lo que es un **Pipeline:**

> *Pipelines* are models of automated release processes. Each pipeline is uniquely named, and consists of stages, actions, and transitions.

Gracias al servicio de CodePipeline, las acciones válidas que se pueden incluir en el Pipeline son:

- Source
- Build
- Test
- Deploy
- Approval
- Invoke

Debido al tamaño del contenido que se almacena en el S3 , se presentará un Pipeline en el que se establezcan dos etapas: **Source**, que se encarga de extraer el contenido en un determinado origen y **Deploy,** que busca efectuar los cambios realizados en la instancia de S3.

La generación del Pipeline se realiza con el comando:

```
aws codepipeline create-pipeline
--pipeline file:///Users/kekaz16/Documents/Semestre6/
CloudComputing/myPipeStructure.json
```

Dicho comando requiere de un argumento **pipeline**, que es el que indica la ruta del archivo que representa la estructura de acciones y etapas que se van a realizar en el Pipeline.

La estructura del archivo es en forma JSON, y un ejemplo de este sería:

Así se ve el archivo de configuración, a continuación se procederá a explicar las llaves más relevantes de la estructura del Pipeline.

- **name:** Es el nombre que se le otorga al Pipeline, en mi caso se puso como myFirstPipeline.

- **roleArn:** Es el nombre del Amazon Resource Name a un usuario IAM con los permisos necesarios para acceder a las herramientas de desarrollador. Para esta llave se puso el ARN que se tiene como usuario, ya que no se tiene asignado un rol de IAM para ejecutar este pipeline.
- **artifactStore:** Va a representar la información sobre el bucket S3 donde se almacenan los “artifacts” es el archivo que se produce cada vez que se traiciona de una etapa a otra (e.g De Source Stage a Deploy Stage, va a existir un artifact con los detalles del proceso) para el pipeline.
  - **type:** indica el tipo de “artifact store” donde se van a guardar los archivos de razón artifact, en este caso se utilizó S3.
  - **location:** es el nombre del bucket S3 que se va a utilizar para el almacenar los artifacts que produzca el pipeline.

- **stages:** La lista de etapas que va a tener el pipeline, que contiene información de las acciones que se van a realizar en cada etapa.
  - **name:** es el nombre de la etapa.
  - **actions:** es una lista con las acciones que se realizan en en la etapa.
    - **name:** es la declaración de la acción. Este nombre puede ser uno que te ayude a identificar que hace la acción.
    - **actionTypeId:** Especifica el tipo de acción que se va a realizar y el proveedor de dicha acción.
      - **category:** en este campo se selecciona la categoría de acción que se a realizar en la etapa, te pide que selecciones el proveedor de la acción. Como se menciono al principio de esta pregunta. Las categorías válidas de una acción son: Source, Build, Test, Deploy, Innove y Approval.
      - **owner:** Es el creador que llama a la acción.
      - **version:** es un string que describe la versión de la acción
      - **provider:** es el proveedor del servicio que se requiere para llevar a cabo la acción.
    - **runOrder:** es un entero que indica el orden en que las acciones se ejecutan

    - **configuration:** son los valores de la configuración de la acción. Cada etapa tiene su propia configuración, por lo que esta estructura depende de las etapas del pipeline. En este caso se mostrarán las llaves que se deben de llenar en la etapa de Source.
        - **S3Bucket:** el nombre de la instancia S3 donde se va a realizar la etapa de Source
        - **S3ObjectKey:** es el nombre del objeto S3 donde los cambios del código serán detectados.
        - **PollForSourceChanges:** controla si CodePipeline checa constante si hay cambios en el código fuente del bucket S3.
    - **outputArtifacts:** es el nombre o ID del resultado de la acción.
    - **inputArtifacts:** es el nombre o ID del artifact consumido por determinada acción.
    - **roleArn:** es el ARN del rol de un usuario IAM que permite que se realiza la acción establecida.
    - **region:** es la región AWS de la declaración de la acción, en ese se tiene como us-east-1.
    - **namespace:** es el nombre una variable que se asociar con la acción.
- **version:** es un número que indica la versión del pipeline .

Una vez que se crea el pipeline con su respecitvo archivo de configuración JSON, se tiene que iniciar la ejecución del pipeline para que empiece a operar. Para conseguir esto, se tiene que usar el comando:

```
aws codepipeline start-pipeline-execution
--name myFirstPipeline
```

Este comando requiere del nombre del Pipeline que se quiere ejecutar. El nombre del Pipeline se puede obtener del archivo de configuarción, que en este caso sería, myFirstPipeline. Y con eso se inicia un el Pipeline definido, y entra en un proceso en el que constante checa los últimos cambios que se han hecho en el código fuente para actualizar el bucket S3.

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

El reto que se presenta en Real-World Engineering Challenges #8: Breaking up a Monolith, trata sobre el proyecto migración masiva de la plataforma educativa Khan Academy. Este proyecto involucraba hacer una transición de 1 millón de líneas de código en Python 2 y reinvidicarlos en alrededor de 40 servicios. Este proceso tomo un lapso de 3.5 años para considerar la migración completa.

La estrategia de migración se llevo a cabo en 2 Fases.

- **Fase 1: Minimum Viable Experience (MVE).** En el artículo se comenta que su principal objetivo era lograr la misma funcionalidad que caracteriza a la plataforma, ya que como tal ya tenía un producto, solo había que extrapolarlo a otro lenguaje.

- **Fase 2: Endgame.** Esta fase se concentro en reescribir las herramientas internas del sistema, haciendo la transición de Python a Go. Esta fase fue la que más tiempo tomo debido a que cada se incluían más funciones a la plataforma.

Personalmente, considero que este proyecto de migración me pareció muy desafiante interesante para una organización como Khan Academy. Se entiende que exista esa necesidad de hacer una migración masiva, ya que la mayoría de sistemas optan por usar la arquitectura monolítica por conveniencia y facilidad a la hora de desarrollar, pero a largo plazo, resulta muy difícil de mantener, ya que todo esta fuertemente acoplado y se despliega todo a la vez. Fueron varias las circunstancias que llevaron a la organización a planear esta migración masiva, de las cuales las más relevantes fueron: el hecho de que el sistema estaba desarrollado en Python 2, y esta versión del lenguaje se iba a volver obsoleta, y también la reestructuración en la arquitectura del sistema, de realizar la transición de una aplicación Monolítica a una arquitectura de

Microservicios, que tiene gran oportunidad de sacarle provecho, por la robustez y magnitud del sistema.

La estrategia que utilizaron de Side-By-Side Testing me llamo mucho la atención, porque tomaron una metodología de hacer las cosas paso por paso, que lo mencionan en el artículo como el acercamiento “Field by field”. Me identifique con esta parte, ya que cuando realizas un proceso de “Porting”, que es cuando adaptas un bloque largo de código a otro tecnología; normalmente en este tipo de casos buscas evitar hacer todo el Port en un solo movimiento, ya que esto puede causar problemas que resulten muy difícil de debuguear, entonces lo más recomendable es hacerlo paso por paso, de tal manera que, que los ajustes sean incrementales y sean probados al instante, para así verificar, si la funcionalidad que se pretende replicar esta previsto como se desea. Este acercamiento me parece lo mejor que se puede hacer, aunque resulte un poco más tardado, pero te aseguras que el margen de error sea capaz de ser controlado, ya que se avanza poco a poco.

También se menciona algo que he escuchado varias veces y es el lenguaje de consultas para APIs GraphQL, que es de código abierto y que permite comunicar los servicios que se fueran desplegando para redirigir el tráfico e ir encapsulando el comportamiento del sistema en distintos componentes que en conjunto formaran la lógica del dominio de la plataforma.

El hecho de que conforme fueran desarrollando el Port de un servicio en Go, lo pudieran testear en producción junto al sistema Monolítico en Python, resultó una buena estrategia para conocer el desempeño del servicio Go, operando con un porcentaje de todo el tráfico que tiene la plataforma.

Otra cosa que me sorprendió fue que la administración del proyecto, que fue llevado a cabo bajo una metodología de Envío Incremental, con una fecha límite fija y un alcance delimitado. Kevin Dangoor, el arquitecto de software, consideró que mantener el ritmo en desplegar los servicios era importante, aunque fuera un entregable pequeño, tenía

valor. Y pienso que ya cuando el equipo agarra un ritmo de trabajo estable en el que están cómodos, llega un momento en que ya no se percibe tanto la presión de presentar un entregable.

Referencias Bibliográficas

AWS (2023). *create-pipeline.* https://awscli.amazonaws.com/v2/documentation/api/latest/reference/codepipeline/create-pipeline.html

AWS (2023). *CodePipeline pipeline structure reference.* https://docs.aws.amazon.com/codepipeline/latest/userguide/reference-pipeline-structure.html#action-requirements

AWS (2023). *Amazon S3 deploy action.* https://docs.aws.amazon.com/codepipeline/latest/userguide/action-reference-S3Deploy.html.

AWS (2023). *Amazon S3 source action.* https://docs.aws.amazon.com/codepipeline/latest/userguide/reference-pipeline-structure.html#actions-valid-providers

AWS (2023). *What is AWS CodePipeline.* https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html

Bilal, A. (2021). *What is API Pagination? .* https://rapidapi.com/guides/api-pagination

White, B. (2021). *What is a Callback API? .* https://stackoverflow.com/questions/38854121/what-is-a-callback-api

Shilkov, M (2021). *Cold Starts in AWS Lambda.* https://mikhail.io/serverless/coldstarts/aws/

MDN WebDocs (2023). *HTTPS request methods.* https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

IETF (2023). *HTTP Semantics.* https://httpwg.org/specs/rfc9110.html