**LABORATORIO No.10**



**ELABORADO POR:**

**SAMUEL ROJAS YOPASA**

**MAURICIO ALEJANDRO MONROY JIMÉNEZ**

**PROFESOR**

**DIEGO ANDRES TRIVIÑO GONZALEZ**

**ARQUITECTURA DE SOFWARE**

**16-11-2024**

* ¿Qué es un Azure Function?

Azure Function es un servicio de computación en la nube que permite ejecutar pequeños fragmentos de código (funciones) en respuesta a eventos específicos, sin necesidad de gestionar infraestructura. Es parte del paradigma **serverless** y es ideal para tareas como procesamiento de datos, integraciones, y automatización.

* ¿Qué es serverless?

El término "serverless" hace referencia a un modelo de computación en la nube donde el desarrollador no tiene que gestionar ni administrar servidores. Este modelo permite abstraer la infraestructura subyacente para que los desarrolladores puedan centrarse únicamente en el código de la aplicación, sin tener que preocuparse por los detalles de los servidores, la configuración, el escalado o el mantenimiento.

Características de un modelo serverless:

* **Abstracción de la infraestructura:** La infraestructura está completamente gestionada por el proveedor de servicios en la nube. El desarrollador escribe código y lo despliega sin tener que preocuparse por la infraestructura que lo soporta.
* **Escalabilidad automática:** Escalan automáticamente según la demanda. Si la aplicación recibe muchas solicitudes, el proveedor de la nube asigna automáticamente más recursos para manejarlas. Si la demanda disminuye, los recursos se ajustan hacia abajo sin intervención manual.
* **Pago por uso:** Los costos están basados en el uso real de los recursos. No se paga por mantener servidores activos todo el tiempo. En lugar de eso, se paga en función del tiempo de ejecución de las funciones, la cantidad de solicitudes, o la cantidad de recursos como CPU y memoria que se usa.
* **Funcionalidades bajo demanda:** Las aplicaciones serverless suelen estar organizadas en pequeños fragmentos de código llamados funciones, que se ejecutan en respuesta a eventos.
* ¿Qué es el runtime y que implica seleccionarlo al momento de crear el Function App?

El runtime es el entorno en el que se ejecutan las funciones de Azure. Este define el lenguaje y las capacidades disponibles para la aplicación.

Al crear un Function App, se selecciona:

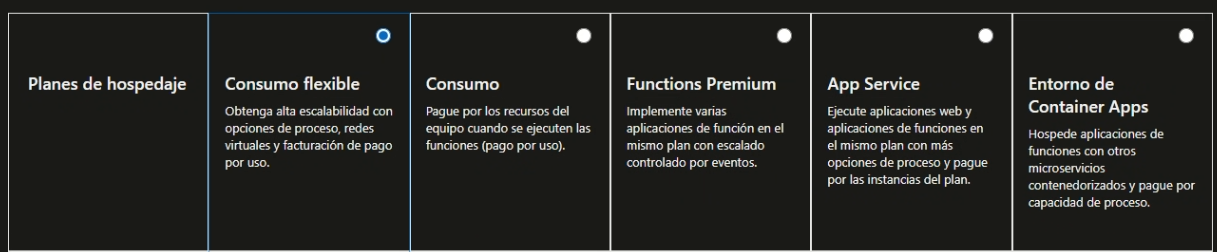
* **Versión del runtime:** Por ejemplo, .NET, Node.js, Python, Java, etc.
* **Sistema operativo:** Windows o Linux.

Implicaciones:

* Define las dependencias que se pueden usar (por ejemplo, librerías específicas del lenguaje).
* Afecta el rendimiento y las opciones de integración.
* Una vez desplegado, cambiar el runtime puede requerir reconstruir la aplicación.
* ¿Por qué es necesario crear un Storage Account de la mano de un Function App?

Un Storage Account es necesario porque Azure Functions depende de Azure Storage para varias tareas críticas:

* **Logs y monitoreo:** Almacena información de ejecución y diagnósticos.
* **Escalado:** Coordina instancias para manejar carga (durante escalado automático).
* **Estado interno:** Guarda el estado de las funciones duraderas (Durable Functions).
* **Archivos y configuraciones:** Guarda los binarios y configuraciones de la Function App.
* ¿Cuáles son los tipos de planes para un Function App?, ¿En qué se diferencias?, mencione ventajas y desventajas de cada uno de ellos.



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Plan** | | Escalado | Costo | |  | | --- | | **Cold Start** | | Límite de Ejecución | Casos de Uso |
| |  | | --- | | Consumo | | Automático | Bajo | Sí | 5-10 minutos | Cargas impredecibles o ligeras. |
| |  | | --- | | Consumo Flexible | | Personalizable | Moderado | Opcional | |  | | --- | | 5-10 minutos | | Mayor disponibilidad con escalado preciso. |
| |  | | --- | | Premium | | Automático | Alto | |  | | --- | | No | | Ilimitado | Funciones críticas o de alta demanda. |
| |  | | --- | | Dedicado (App Service) | | Manual | Variable | No | Ilimitado | Cargas constantes en recursos compartidos. |
| |  | | --- | | Contenedor | | Automático | Variable | No | Depende del contenedor | Portabilidad y personalización avanzada. |
| |  | | --- | | Entorno ASE | | Manual | |  | | --- | | Muy alto | | No | Ilimitado | Requisitos empresariales y de seguridad. |

* ¿Por qué la memoization falla o no funciona de forma correcta?

La memoization no funciona porque cada llamado a las Function App es independiente y no comparte estado con otras instancias, por tanto, no podrá acceder al diccionario con los resultados previamente calculados.

* ¿Cómo funciona el sistema de facturación de las Function App?

El sistema de facturación para las Azure Function Apps depende de varios factores, entre los que destacan el modelo de hospedaje elegido y el uso específico de las funciones, algunos de ellos son:

* **Número de ejecuciones:** El cargo se basa en la cantidad de veces que se invoca la función.
* **Duración de la ejecución:** Se cobra por el tiempo que las funciones están en ejecución, medido en milisegundos.
* **Recursos utilizados:** Se cobra por la cantidad de memoria y CPU que consume la función durante su ejecución.
* **Modelo de precios:** El precio varía dependiendo del plan que se elija (Consumo, Premium o App Service), siendo el Plan de Consumo el más económico.
* Informe

La creación de la Function App en Azure y la instalación de la extensión Azure Functions para Visual Studio Code se llevó a cabo siguiendo el procedimiento descrito en el enunciado del laboratorio. Sin embargo, durante el proceso de despliegue de la función, se presentaron algunas dificultades técnicas que requirieron ajustes adicionales. Para solucionar estos problemas y asegurar un despliegue exitoso, se realizaron las siguientes acciones:

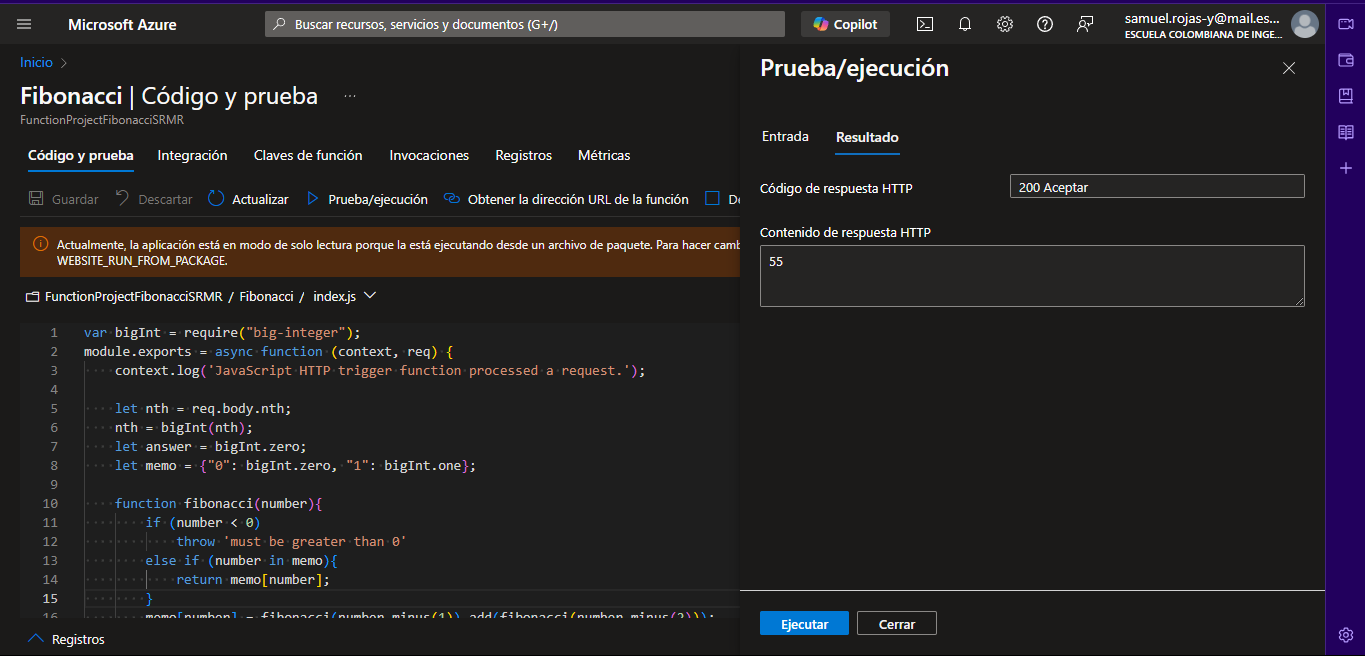
* + Instalación de Node.js y configuración de las variables de entorno:
  + Se verificó que Node.js estuviera correctamente instalado en el sistema, ya que es una dependencia fundamental para la ejecución de las funciones en Azure. Después, se procedió a configurar las variables de entorno necesarias para que el entorno de desarrollo pudiera acceder a las herramientas de Node de manera adecuada.
  + Instalación de Azure Functions Core Tools: Para habilitar la ejecución local de funciones de Azure, fue necesario instalar las Azure Functions Core Tools. Esto se realizó mediante el comando npm i -g azure-functions-core-tools@4 --unsafe-perm true, que permitió instalar la versión más reciente de las herramientas necesarias para trabajar con Azure Functions en el entorno local.
  + Modificación en el archivo host.json: En algunos casos, las configuraciones predeterminadas en el archivo host.json no eran completamente compatibles con el entorno local. Por lo tanto, se realizó un ajuste en una de las versiones de este archivo para garantizar que el despliegue y la ejecución de las funciones fueran correctos.

A computer screen with white text

Description automatically generated

Después de realizar estos ajustes, el proceso de despliegue se completó correctamente y se pudo verificar el funcionamiento de la función desde el portal de **Azure**, lo que permitió

confirmar que la aplicación estaba activa y accesible.



Una vez resuelto el despliegue de la función, se procedió a la configuración de las peticiones que se realizarían a la función a través de Newman, que es una herramienta de línea de comandos para ejecutar colecciones de Postman. Para ello, se creó un archivo JSON denominado postmanRequests.json, el cual contiene todas las configuraciones necesarias para realizar las solicitudes POST a la función de Fibonacci implementada en Azure. Este archivo incluye la URL de la función, los parámetros de la solicitud, así como las configuraciones de las pruebas automatizadas para medir el rendimiento y la respuesta de la función bajo diferentes condiciones:



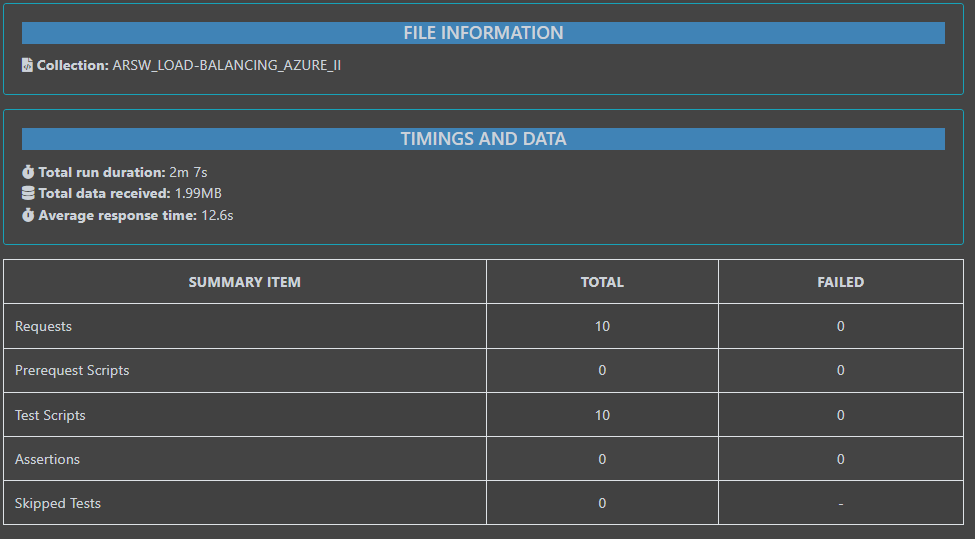
El archivo es una colección de Postman que contiene la configuración necesaria para ejecutar las peticiones a nuestra función de Fibonacci en Azure. A continuación, explicamos cómo este archivo facilita el uso de Newman y cómo se configuran las peticiones:

* 1. Estructura del Archivo JSON
  + **Info del JSON:** La propiedad info contiene metadatos sobre la colección, incluyendo su identificador único (\_postman\_id), nombre (name), y el esquema que define la versión de la colección.
  + **Item de Petición:** Dentro de item, tenemos una petición POST que se va a realizar a la URL de nuestra función de Fibonacci en Azure. La URL es: https://functionprojectfibonaccisrmr.azurewebsites.net/api/Fibonacci
  1. Configuración de la Petición
  + **Método POST:** Se realiza un POST hacia el endpoint de la función Fibonacci en Azure.
  + **Encabezado:** Se especifica el tipo de contenido como application/json en el encabezado Content-Type, lo que indica que los datos enviados en el cuerpo de la solicitud serán en formato JSON.
  + **Cuerpo de la Petición:** El cuerpo de la solicitud está configurado en formato raw y contiene los datos necesarios para calcular un número en la secuencia de Fibonacci. En este caso, el valor de nth está establecido en 10000, lo que significa que estamos pidiendo el valor de Fibonacci en la posición 10,000 de la secuencia.

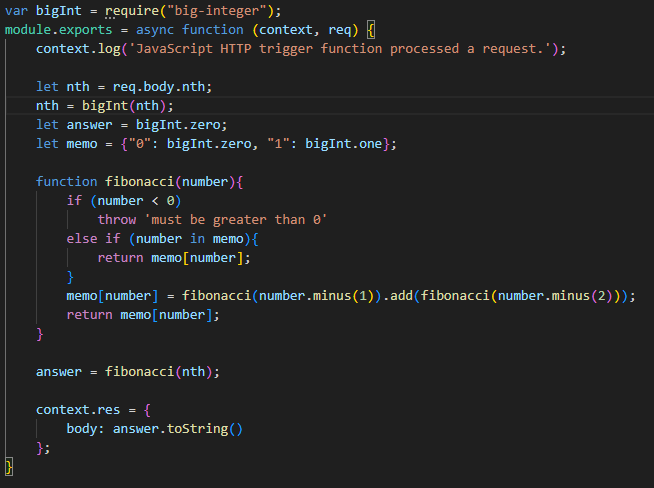
Ahora para las peticiones concurrentes usamos el comando:

* + newman run "postmanRequests.json" -n 10 -r htmlextra --reporter-htmlextra-export ./report10000After.html

Este comando ejecuta la colección de Postman definida en el archivo postmanRequests.json 10 veces de manera concurrente. La opción -n 10 especifica que se enviarán 10 peticiones al mismo tiempo. Además, se incluye el reporter htmlextra para generar un informe en formato HTML, que se guarda en la ruta ./report10000After.html. Este informe incluye detalles sobre el tiempo de ejecución, los resultados de las peticiones y cualquier error que pueda haber ocurrido durante las pruebas.



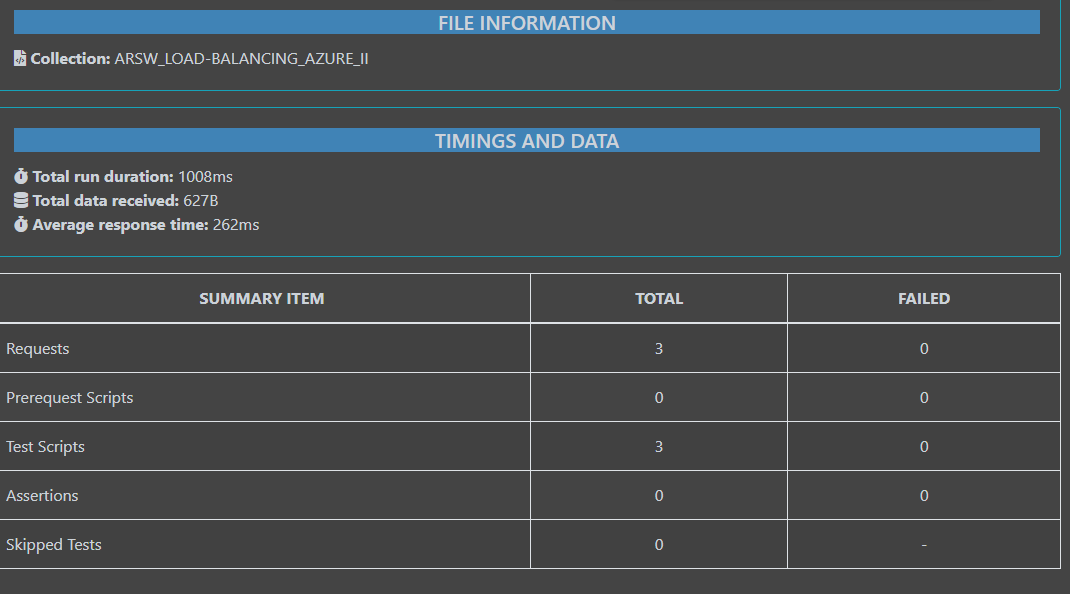
Una vez realizadas las pruebas iniciales en Postman para evaluar el rendimiento de la función de Fibonacci, se implementó una nueva versión de la función que utilizaba memoization. Esta técnica de optimización permite almacenar los resultados de cálculos anteriores para evitar la repetición de los mismos, mejorando significativamente el rendimiento cuando se calculan números de Fibonacci de manera repetida.



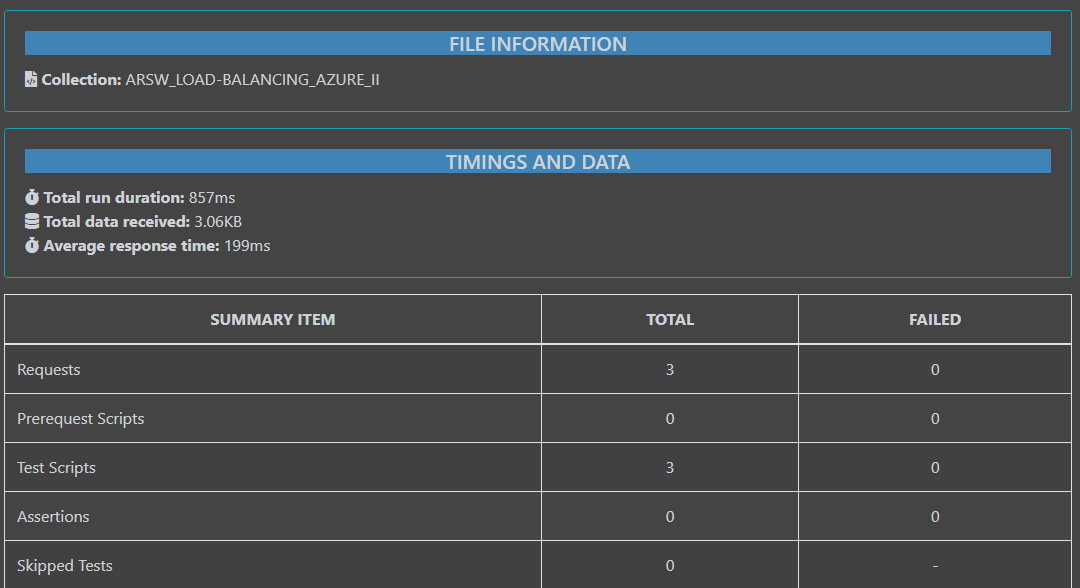
Con la nueva versión optimizada de la función, se ejecutaron nuevamente las pruebas usando Newman. En esta ocasión, se modificó el archivo postmanRequests.json para utilizar diferentes valores de entrada para los cálculos de Fibonacci: 1000, 5000 y 10000. Para cada uno de estos valores, se realizaron 3 peticiones concurrentes, lo que permitió comparar el rendimiento de la función en condiciones de carga.

Los resultados obtenidos para cada valor de Fibonacci fueron los siguientes:

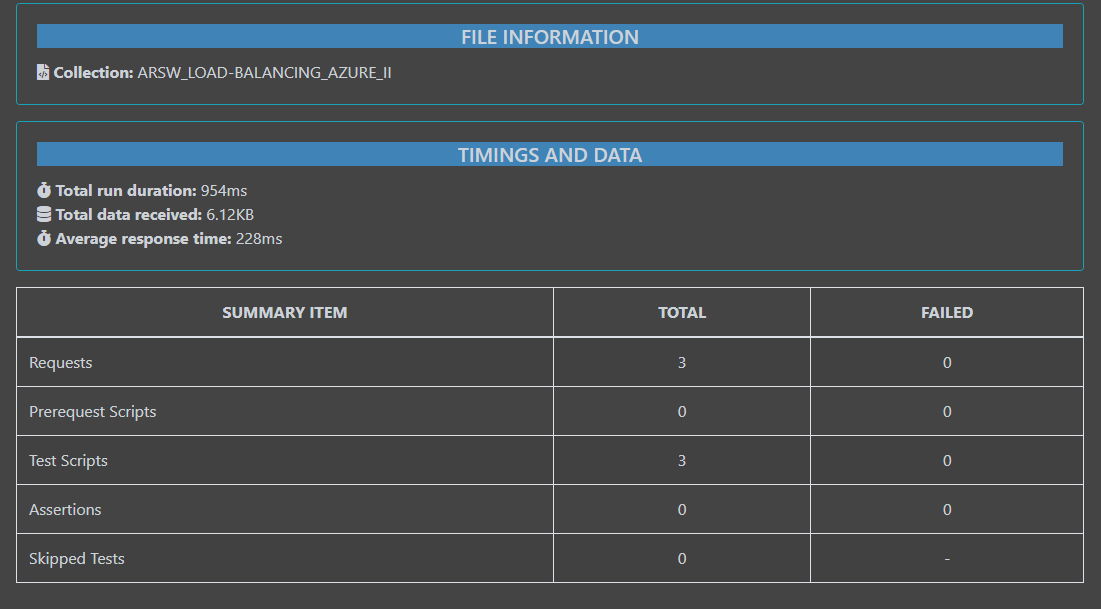
* 1000:



* 5000:



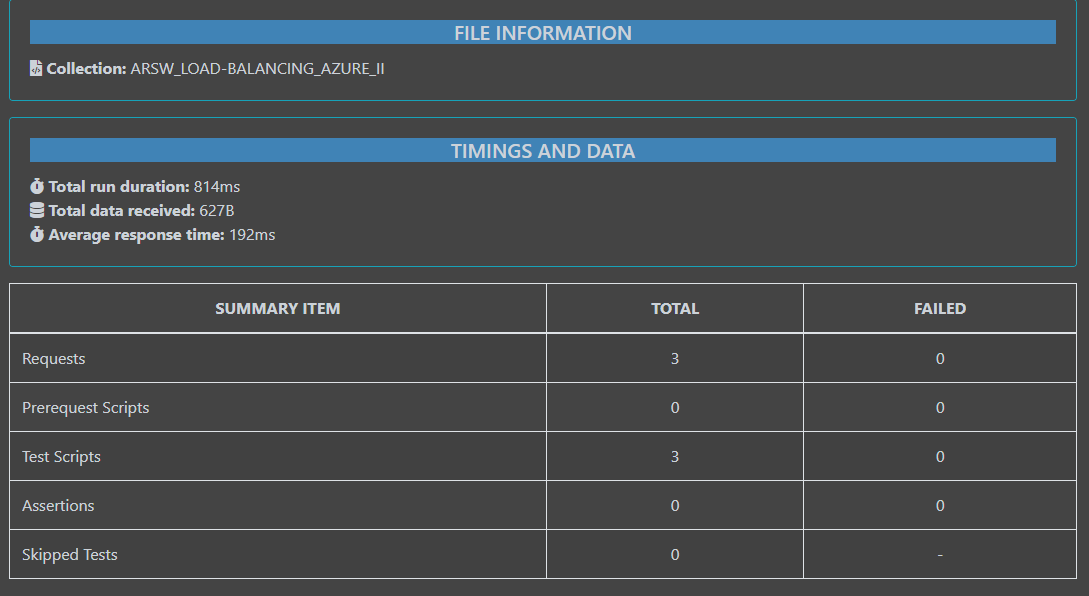
* 10000:



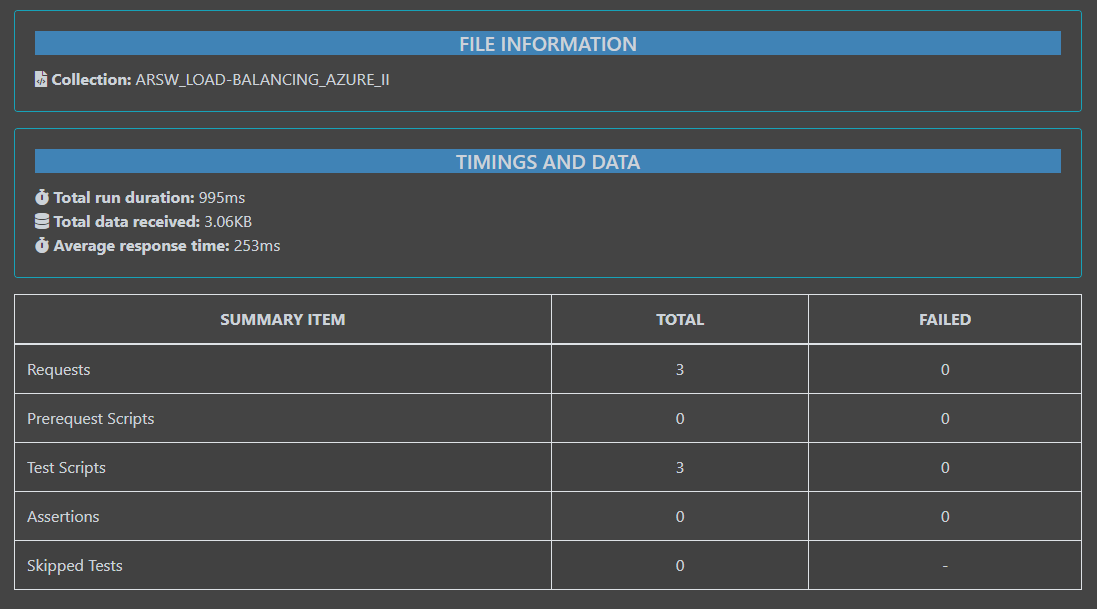
Tras realizar estas primeras pruebas, se dejó reposar la función durante 5 minutos sin realizar ninguna operación sobre ella. Posteriormente, se repitieron las mismas pruebas para los valores de 1000, 5000 y 10000 de Fibonacci, utilizando las mismas condiciones de carga y concurrentes.

Los resultados obtenidos tras el periodo de reposo fueron los siguientes:

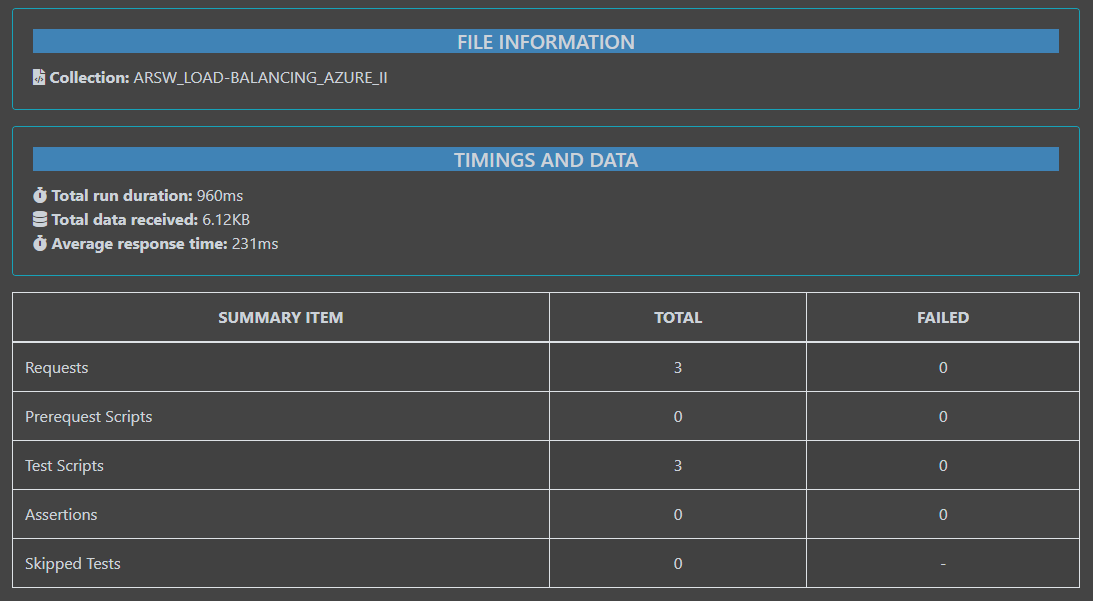
* 1000:



* 5000:



* 10000:



Al comparar los resultados de ambas series de pruebas (las realizadas de manera inmediata y las realizadas después de un descanso de 5 minutos), se observó que no hubo una mejora significativa en el rendimiento tras el tiempo de espera. Para cada valor de Fibonacci, los tiempos de respuesta y la eficiencia del cálculo fueron muy similares, lo que sugiere que la optimización mediante memoization fue efectiva durante tiempos cortos, pero no al tener un largo periodo de inactividad.