**T.C.**

**ISTANBUL KULTUR UNIVERSITY**

**INSTITUTE OF GRADUATE STUDIES**

# IMPLEMENTATION AND ANALYSIS OF INTERSERVICE AUTHORIZATION AND AUTHENTICATION METHODS IN MICROSERVICE-BASED SYSTEMS

**MASTERS THESIS**

**METIN BIÇAKSIZ**

**2100006025**

**Department: Computer Engineering**

**Program: Computer Engineering**

**Thesis Supervisor: Assis. Prof. Dr. Öznur ŞENGEL**

**JUNE 2024**

**T.C.**

**ISTANBUL KULTUR UNIVERSITY**

**INSTITUTE OF GRADUATE STUDIES**

# IMPLEMENTATION AND ANALYSIS OF INTERSERVICE AUTHORIZATION AND AUTHENTICATION METHODS IN MICROSERVICE-BASED SYSTEMS

**MASTERS THESIS**

**METIN BIÇAKSIZ**

**2100006025**

**Department: Computer Engineering**

**Program: Computer Engineering**

**Thesis Supervisor: Assis. Prof. Dr. Öznur ŞENGEL**

**Jury Members:**

**Assis. Prof. Dr. Öznur Şengel**

**Assoc. Prof. Dr. Akhan Akbulut**

**Assoc. Prof. Dr. Can Eyüpoğlu**

**JUNE 2024**

## PREFACE

I would like to express my profound gratitude to my mentor, Assis. Prof. Dr. Öznur ŞENGEL, for her invaluable guidance and support throughout this journey. Heartfelt thanks to my daughter, Dafne and my wife, Katerina, for their love and patience, which have been my constant source of strength. I am deeply appreciative of the esteemed faculty at Istanbul Kultur University and its teaching staff for their knowledge and encouragement. Lastly, I extend my sincere thanks to my family, friends, and colleagues for their unwavering support and belief in me. This thesis is a testament to the collective efforts and encouragement from all those who have been part of this journey.

06/2024

Metin BIÇAKSIZ

**TABLE OF CONTENT**

Page

## LIST OF FIGURES

**Figure** **Page**

## LIST OF TABLES

**Table** **Page**

## ABBREVIATIONS

| Abbreviation | Description |
|---|---|
| **API** | Application Programming Interface |
| **MFA** | Multi-Factor Authentication |
| **JWT** | JSON Web Token |
| **HTTP** | Hyper Text Transfer Protocol |
| **HTTPS** | Hyper Text Transfer Protocol Secure |
| **RBAC** | Role Based Access Control |
| **ABAC** | Attribute Based Access Control |
| **OAuth** | Name of the security providing framework (Open Authorization) |
| **gRPC** | Remote Protocol Call by Google |
| **SHA** | Secure Hash Algorithm |
| **CSSO** | Continuous Single Sign-On |
| **RTT** | Round Trip Time |
| **MD** | Message Digest |
| **KDF** | Key Derivation Function |
| **CAS** | Content Addressable Storage |
| **SSL** | Secure Socket Layer |
| **TLS** | Transport Layer Security |
| **HMAC** | Hash-based Message Authenticity Code |
| **RSA** | Rivest, Shamir, Adleman (the inventors of the algorithm) |
| **ECDSA** | Elliptic Curve Digital Signature Algorithm |

| | |
|---|---|
| **University** | : T.C. İstanbul Kultur University |
| **Institute** | : Institute of Graduate Studies |
| **Department** | : Computer Engineering |
| **Program** | : Computer Engineering |
| **Supervisor** | : Assis. Prof. Dr. Öznur ŞENGEL |
| **Degree Awarded and Date** | : Master's – June 2024 |

## ABSTRACT

In the evolving landscape of software architecture, the microservice paradigm has emerged as a robust solution for creating scalable and maintainable systems. A critical challenge within this architecture is ensuring secure inter-service authentication and authorization, particularly in synchronous communication. This thesis delves into the intricacies of the JSON Web Token (JWT) mechanism, a prevalent method for securing these communications.

Extensive literature research was conducted to assess current approaches, highlighting both ineffective practices and innovative strategies. A predominant trend identified is the adoption of the SHA2-512 hashing algorithm for token signing. Despite its popularity, SHA2-512 has demonstrated vulnerabilities to length extension attacks, posing significant security risks.

To address these concerns, this research advocates for the implementation of the SHA3-512 hashing algorithm within the JWT mechanism. Even though SHA3 is not natively supported by JWT libraries within the Spring Boot framework which was used for the development of the application, another mechanism which allowed us to apply SHA3 in the token signing was adopted. SHA3-512 offers enhanced resilience against such attacks owing to its fundamentally different cryptographic structure. Through this approach, we aim to bolster the security framework of microservice architectures, mitigating risks and enhancing the protection of inter-service communications. The findings and proposals presented in this thesis provide a crucial step towards more secure microservice ecosystems.

**Keywords:** SHA2, SHA3, JWT, Microservice Architecture, synchronous communication, inter-service authentication and authorization

**Üniversite** : İstanbul Kültür University
**Enstitü** : Lisansüstü Eğitim Entitüsü
**Anabilim Dalı** : Bilgisayar Mühendisliği
**Programı** : Bilgisayar Mühendisliği
**Tez Danışmanı** : Dr. Öğr. Üyesi Öznur ŞENGEL
**Tez Türü ve Tarihi** : Yükseklisans – Haziran 2024

## ÖZET

Yazılım mimarisinin gelişen dünyasında, mikro hizmet paradigması ölçeklenebilir ve sürdürülebilir sistemler oluşturmak için sağlam bir çözüm olarak ortaya çıkmıştır. Bu mimarideki önemli bir zorluk, özellikle senkron iletişimde, hizmetler arası kimlik doğrulama ve yetkilendirmeyi güvence altına almaktır. Bu tez, bu iletişimleri güvence altına almak için yaygın bir yöntem olan JSON Web Token (JWT) mekanizmasının ayrıntılarına iniyor.

Güncel yaklaşımları değerlendirmek adına kapsamlı bir literatür araştırması gerçekleştirildi ve etkisiz uygulamalar ile yenilikçi stratejiler vurgulandı. Belirlenen baskın bir eğilim, token imzalama için SHA2-512 karma algoritmasının benimsenmesi yönündedir. Popülerliğine rağmen, SHA2-512'nin uzunluk uzatma saldırılarına karşı savunmasız olduğu ve önemli güvenlik riskleri oluşturduğu gösterilmiştir.

Bu endişeleri gidermek için bu araştırma, JWT mekanizmasında SHA3-512 karma algoritmasının uygulanmasını savunmaktadır. SHA3, Spring Boot çatısı altındaki JWT kütüphaneleri tarafından doğal olarak desteklenmese de, uygulamanın geliştirilmesinde SHA3'ün token imzalamada uygulanmasına olanak tanıyan başka bir mekanizma benimsenmiştir. SHA3-512, kriptografik yapısı temel olarak farklı olduğu için bu tür saldırılara karşı daha güçlü bir direnç sunar. Bu yaklaşım sayesinde, mikro hizmet mimarilerinin güvenlik çerçevesini güçlendirerek riskleri azaltmayı ve hizmetler arası iletişimin korunmasını arttırmayı hedefliyoruz. Bu tezde sunulan

bulgular ve öneriler, daha güvenli mikro hizmet ekosistemlerine önemli bir adım olarak değerlendirilmektedir.

**Anahtar Kelimeler:** SHA2, SHA3, JWT, Microservice Mimarisi, senkron iletişim, servislerarası / hizmetlerarası doğrulama ve yetkilendirme

# 1. INTRODUCTION

Since the inception of computer era, humans have tried better ways to communicate and make use of the computer in easier, better and more productive ways. To do so, they have come up with different software languages, and different software architectures. Regardless of the languages used in software development, the architecture of the software system plays and pivotal role. In modern software development, even though there are more than a few software architecture types, the most common ones are monolith architecture, microservices architecture, cloud-native architecture, serverless architecture, event-driven architecture, reactive architecture and domain driven architecture.

In Monolith architecture, the software program is designed as a single, unified unit that contains all the necessary and required components and modules for the business logic and functionality. There is only one code base and it is developed and deployed as one. On the other hand, microservices architecture sees each functionality as a separate unit and as a result, there are smaller, independent services which have their own code base, data storage and deployment pipeline.

Microservice architecture is a way of designing and developing a collection of small, independent and loosely coupled software code bases which are called as services which communicate with each other. Even though it has become popular in recent years, it has long history of evolution. One of the earliest trials was done by IBM in 1997 when they were trying to create a reusable and small software application which was enterprise java bean (EJB) specification. EJBs were designed to facilitate the development and deployment of enterprise Java applications by providing a standard

for encapsulation and business logic. Since it has drawbacks as well, another architectural style came up in 2000, which was called service-oriented architecture (SOA). Along with interoperability and integration among different services, it also introduced concepts such as service discovery, orchestration, governance for complexity and diversity of the distributed systems.

Dr. Peter Rogers was the first to use the term "micro web services" during a conference in 2005. His model proposed microservices that adopt the principles of web and REST to service-oriented architecture. The term "microservices" was created in 2011 by a group of software architects while they were discussing their experience and practices in working with distributed systems. They realized that they were using similar approaches which was different from the traditional ones and decided to name it as *microservices.* Since then, microservices has gained popularity among software developers and been adopted by many organizations and developers based on their needs and requirements.

Microservice architecture is an architectural approach which is widely used in software development where the goal is to design and develop sophisticated applications as a collection of small, loosely coupled, independently deployable units. Unlike monolith applications which keeps all the functionality in one code base, microservice architecture breaks down the application into smaller, self-contained units which can communicate with each other through APIs (Application Programming Interface) or other mechanisms.

Main characteristics and principles of microservice architecture are:

1. Service independence: Each microservice has its own functional responsibility and it operates independently from others. This enables independent development, deployment and scalability.
2. Decentralization: Microservices are generally developed by small, cross-functional group of developers where decisions can be made autonomously reducing centralized control.
3. API-based Communication: Services communicate with each other using well-defined APIs, often over HTTP or other lightweight protocols. This allows them to work together while remaining decoupled or loosely coupled.

4. Scalability: Because each service is independent from others, it easier to scale only the required parts of the application while improving the resource utilization.
5. Resilience: Unlike monolith applications, in microservice architecture, if one service fails, this doesn't necessarily mean that the whole application will fail. The failure of one service does not stop other services.
6. Diversity of Technology: Teams for each service can choose their own technology stack based on the requirements and needs of the service functionality.
7. Continuous Deployment: since microservices are generally deployed independently, it allows more frequent updates and the risk of effecting the entire system is reduced.
8. Data Management: Each service may have its own database, and different approaches are used for data consistency and synchronization between individual services.

Microservice architecture can offer many advantages, such as improved agility, easier maintenance and updates, and better fault isolation. However, it also introduces challenges, including increased complexity in terms of service coordination, inter-service communication, and data management.

In this study, we will be mainly concentrating on the microservices architecture and more specifically interservice communication and, authorization and authentication mechanism in interservice communication in microservice architecture. We will discuss the already in-use authentication and authorization mechanisms and their security levels. By doing so, we are aiming to contribute to the literature in the field by comparing the available mechanisms and methods and offering innovative mechanisms and approaches.

Authentication and authorization are two fundamental security concepts that work together to control access to the system resources and data. They might sound similar; however, they address different aspects of security. Authentication is the process of verifying someone's identity. When one logs into a website or app, he/she is going through authentication. The system checks his/her credentials, like username and

password, to make sure he/she is the one who she/he claims to be. Authorization determines what a user can access after their identity is confirmed. Even if the user is authenticated, he/she still might not have access to everything. Authorization controls what actions the specific user can perform within the system. Both authentication and authorization are crucial parts of online security. They ensure that only authorized users can access sensitive information and perform actions within the system.

In this study, we aim to achieve a more resilient interservice authentication and authorization mechanisms in microservice architectures. We have specifically worked on an innovative approach which bring about a new perspective to the to the topic while opening a new chapter in the literature. While describing different methods for authorization which are already in use, we will describe our approach and solution to the issue.

## 2. LITERATURE SEARCH

Since the advent of software development, a plethora of approaches have been proposed by developers and scholars alike. The world of software development has undergone a remarkable transformation since its inception. Early websites were static, offering a limited set of information with minimal user interaction. Over time, the web evolved into a dynamic platform, enabling real-time data exchange and interactive functionalities. This evolution necessitated a shift in the underlying architectural paradigms used to build web applications.

### 2.1. The Era of Monolith Architecture

The initial approach to web application development involved a monolithic architecture. A monolith is a single, self-contained application that encompasses all functionalities within a single, unified codebase. This centralized structure offered several advantages in the early days such as simplicity because developing and deploying a single application was easier to manage compared to distributed systems. All components resided within the same codebase, facilitating tight integration and data coherence. By scaling (vertical) the hardware resources of the server hosting the monolith, developers could achieve increased capacity.

However, as web applications grew in complexity and user base, the limitations of the monolithic architecture became apparent. These limitations included: Adding new functionalities or scaling (horizontally) specific services within the application proved cumbersome. Codebases ballooned, making maintenance and updates a time-consuming and error-prone process. Implementing changes or deploying new features required modifying the entire application, hindering development velocity.

### 2.2. The New Era: Microservice Architecture

The limitations of monolithic architecture paved the way for the emergence of the microservices architecture. This approach advocates the decomposition of a large

application into smaller, independent, and loosely coupled services. Each microservice encapsulates a specific business function and operates autonomously, utilizing its own technology stack and database. Communication between microservices is facilitated by the use of well-defined APIs.

Individual microservices can be scaled independently based on their specific resource requirements which is called horizontal scaling. Smaller codebases are easier to maintain and update, which promotes faster development cycles. Changes and new features can be implemented within specific microservices without affecting the entire application in an agile manner. The failure one of a single microservice does not result in the complete failure of the entire application which in return increases the resilience of the whole system.

### 2.3. The Security Landscape: Authentication and Authorization

As web applications become increasingly complex and handle sensitive user data, robust security measures are of paramount importance. Two fundamental security concepts in this context are authentication and authorization. Authentication verifies the identity of a user or service attempting to access an application or resource. Common authentication mechanisms include username/password combinations, multi-factor authentication (MFA), and token-based authentication using technologies like JWT (JSON Web Token). Authorization determines what actions a user or service is permitted to perform after successful authentication. It enforces access control policies and ensures that users can only access or modify resources based on their assigned permissions.

### 2.4. Monoliths vs. Microservices: Authentication and Authorization Considerations

The transition from monolithic to microservice architecture necessitates a reassessment of authentication and authorization strategies. In a monolith, these functionalities are typically centralized, leveraging a single-user database and authorization logic. Authentication in monolith stores and validates user credentials, after which users are granted access to all functionalities within the application based on their assigned role. Access control policies are defined within the monolith's

codebase for authorization in monoliths, regulating user permissions to specific resources or functionalities. A security breach in the central user database or authorization logic compromises the entire application. Since the only entry point carries the responsibilities, once it fails, the whole system will suffer, and this is against the whole logic of microservice architecture. Imagine you have an e-commerce platform built using a microservice architecture. The platform includes several microservices. In this setup, an API Gateway is often employed to act as a single-entry point for all client interactions with these microservices. While the API Gateway provides a convenient and centralized way to manage routing, load balancing, and security, it also becomes a single point of failure: If the API Gateway goes down or is compromised, users may not be able to access any of the services (User, Order, Product Catalog, or Payment) because all requests pass through the gateway.

An attack or failure on the API Gateway could expose sensitive data from multiple microservices, as it needs to handle various forms of authentication and authorization. The security mechanisms implemented in the API Gateway, like rate limiting or request validation, are critical. If these mechanisms are breached, attackers might exploit the API Gateway to launch larger-scale attacks on the underlying microservices. As the user base expands, the process of scaling the authentication and authorization functionalities becomes increasingly challenging. As the platform grows in popularity, the traffic to these microservices increases. To handle the load, each microservice is scaled up, potentially across multiple instances or even deployed in different geographic locations.

With scaling, the number of interactions between microservices increases. For instance, when a user places an order, the Order Processing Service needs to interact with the User Authentication Service to confirm the user's identity, the Product Catalog Service to check availability, and the Payment Service to process the transaction. Each of these interactions is a potential attack vector. Scaling up means that more endpoints and more instances of the services are exposed, increasing the potential attack surface. Ensuring secure communication between microservices becomes more challenging. Each service needs to confirm the identity and permissions of the requests coming from other services. This often involves using tokens, certificates, or other mechanisms which must be securely managed and verified. Ensuring that the data exchanged

between services has not been tampered with in transit, often requiring robust encryption and secure transfer protocols (e.g., HTTPS). In a highly scaled environment, services need to discover and interact with each other efficiently. Using service discovery mechanisms can present additional security challenges, such as protecting the discovery process from unauthorized access. Modifications to user management or authorization policies necessitate alterations to the entire codebase, which impedes agility. When the development team applies any change in the user management or the authorization mechanism, this will necessitate some additional changes on the gateway or the whole system depending on the architectural choice. This will make all the services behave as if they were monolith, which is against the whole idea behind the microservice architecture.

### 2.5. Microservices and Decentralized Security

The microservices architecture necessitates a decentralized approach to authentication and authorization. Traditional authorization methods can be trickier to implement in a microservices architecture due to its distributed nature. Here are some common authorization types that are well-suited for microservices:

Role-Based Access Control (RBAC): This is a widely used approach where users are assigned roles, and each role is granted specific permissions for resources within a microservice. This allows for granular control over access.

Attribute-Based Access Control (ABAC): This method goes beyond roles and considers additional attributes like user location, device type, or time of day when making authorization decisions. It offers more flexibility but can also become complex to manage.

OAuth 2.0: This is an authorization framework that enables secure delegation of access. It allows a user to grant access to a third-party service without sharing their credentials directly. This is particularly useful for microservices that need to interact with external resources.

OpenID Connect (OIDC): This is an identity layer that sits on top of OAuth 2.0. It

provides additional user information like name and email address along with the access token. This can simplify user identity management across microservices.

JSON Web Tokens (JWT): These are tokens that contain information about a user and their permissions. They are self-contained and can be verified by any microservice that trusts the issuer. This enables stateless authentication, reducing the need for session management.

These methods can be implemented in different ways within a microservice architecture. Some common authorization patterns include API gateway, per-service and global authentication. A centralized API gateway can handle authentication and authorization for all incoming requests, offloading this responsibility from individual microservices. Each microservice can implement its own authorization logic, enforcing access control based on the specific resources it manages. A dedicated service can handle user authentication and provide tokens or attributes that microservices can use for authorization decisions.

### 2.6. Communication between microservices

In a microservices architecture, where applications are divided into smaller, independent services, communication between these services becomes crucial. This communication is referred to as inter-service communication and plays a vital role in ensuring the overall functionality of the application.

There are two main approaches to inter-service communication: synchronous and asynchronous. In synchronous communication, a service makes a direct call to another service's API, utilizing protocols such as HTTP or gRPC. The calling service then awaits a response from the called service before proceeding. This is analogous to the manner in which we interact with a website, whereby a request is sent, and a response is awaited in order to load the page.

In asynchronous communication, services don't wait for a direct response. Instead, messages are sent using message brokers like Kafka or RabbitMQ. The receiving service can then process the message at its own pace. This approach is helpful for tasks

that don't require an immediate response or when the calling service doesn't need to be aware of the processing outcome. The objective of this study is to examine synchronous communication approaches utilizing HTTP requests. It is important to note that the requests will not be sent directly to the target service, instead, they will be sent through an API gateway as the intermediary. Besides, the implementation of synchronous communication is easier.

Murilo et al. [1] mentioned possible problems and used OAuth 2.0, OpenID Connect, API gateway, and JWT mechanisms in the SLR paper. Their findings suggest that OAuth 2.0, Json Web Token (JWT), and the use of API Gateway are the most commonly used ones to mitigate the problem.

Sanger and Abeck [2] studied authentication and authorization mechanisms in microservice architecture. They carried out their studies on the MuleSoft application and used Okta for identity and access management. For authorization, they have used JWT and API proxies. In order to increase the system security, they have used JWT with SHA-512, which generates 126-character hashes.

Banati et al. [3] studied authentication and authorization mechanisms and orchestrator mechanism, which can manage the tokens (create and delete) needed to authenticate and authorize the user. They have examined various types of authentications and authorization especially in healthcare applications where data is vital and should be available for doctors. They have concluded that by using the authentication and authorization orchestrator (AAO), the load on each service is decreased, and the main logic of microservices, which is assigning one responsibility to each service, is protected.

Martin et al. [4] suggested a different method for authentication and authorization, which claims that a user will have access to the whole system once logged in. This creates a huge risk for the system and data if the admin account, which has access to all resources, is compromised. This means Single-Sign-On mechanism is not ideal. To overcome this problem, they proposed an approach based on an authentication mechanism called Continuous-Single-Sign-On (CSSO). The user is authenticated throughout his/her session, not only during the first login. Users do not need to worry

about losing their keys with this system since the authentication is an ongoing process in this system.

Suomalainen [5] studied taking precautions against a user who is already in the system and worked on various challenges and risk for the system; however, our main concern about authentication and authorization is to prevent the attacker from entering the system. In this study, he studied the cases where the user already entered the system and has the ability to reach the sources.

Triartono et al. [6] proposed a mechanism that integrates OAuth 2.0 with role-based access control. They also tested the system's performance by response time, data transferred, and throughput. This approach has achieved considerable results in performance tests.

Shulin and JiePing [7] suggested an authentication and authorization mechanism that integrates OAuth 2.0 and JWT. They also proposed the use of the Zuul service for unified authentication.

Xu et al. [8] claimed that in distributed systems, it is difficult to protect the data sources and propose the use of API gateway technology to overcome the problem. During the test, they evaluated the round-trip time (RTT) and claimed that the application was ready for a life application based on their findings.

In this study, we have used stateless authentication and authorization mechanisms rather than a stateful approach. In the stateful approach, once the user credentials are compromised, it will be too difficult to identify and mitigate the access problem. Besides, the implementation of a stateful authentication and authorization mechanism is complicated and difficult. As a result, we have decided to work on the stateless authentication and authorization. After careful literature searches and studying real-world application examples, we have decided to study the JWT mechanism, more specifically the hashing algorithms of this mechanism.

According to our findings in the literature search, JWT mechanisms studied before were using SHA-256 or SHA-512 hashing algorithms. Even though there are more

resilient hashing algorithms, the SHA3 algorithm has never been used in the JWT mechanism since it is not natively supported. In this study, we have tried to implement the SHA3-512 algorithm to our API gateway and Auth Service to offer a more resilient authentication and authorization mechanism for microservice architecture. JWT has three parts: the head, the payload, and the signature. While applying our approach, we had to apply the SHA3 algorithm to the payload section. To see the results, we have used the same code based on two different algorithms (SHA2-512 and SHA3-512) and compared the results. Even though there are no big differences in latency performances, the resilience has improved, as SHA3 was proven in 2015 when it was first introduced.

### 2.6.1. Synchronous and Asynchronous Communication in Microservices

The microservices architecture has become a popular design paradigm for building complex, scalable software applications. By decomposing functionality into independent, loosely coupled services, developers gain agility, maintainability, and resilience. However, the success of this approach hinges on effective communication between these microservices. This article explores the two primary communication patterns employed in microservices: synchronous and asynchronous communication. We'll delve into their characteristics, advantages, disadvantages, and ideal use cases, equipping you to make informed decisions for your microservice architecture.

#### 2.6.1.1. Synchronous Communication: A Familiar Dance

Synchronous communication, a cornerstone of traditional client-server interactions, operates in a request-response fashion. A service initiates communication by sending a request to another service, known as the target service. The calling service then blocks, waiting for a response from the target service before proceeding. This response can be data, confirmation of completion, or an error message.

Imagine an e-commerce platform where a user adds an item to their cart. The UI service (the calling service) sends a request to the inventory service (the target service) to check product availability. The UI service waits for the inventory service's response – "In stock" or "Out of stock" – before updating the user interface accordingly.

There are some advantages of this way of communication. Synchronous communication is straightforward to implement and understand. Developers familiar with traditional client-server architectures can readily adapt to this approach. The calling service receives a response immediately, allowing for a cohesive user experience. In our e-commerce example, the user doesn't have to wait indefinitely to know if an item is available. Since the calling service waits for the response, errors can be handled synchronously. This simplifies debugging and error management within a single thread of execution.

There are also disadvantages of it, too. The calling service execution is blocked until a response is received. This can lead to performance bottlenecks if the target service experiences delays. In our example, a slow inventory service response could delay the entire checkout process. Synchronous communication can become challenging to scale as the number of microservices and request volume increase. Multiple concurrent requests can overwhelm the target service, leading to performance degradation. Tight coupling exists between the caller and target services, as the caller's execution depends on the target service's response. This can hinder independent deployment and maintenance of services.

Here are some of the use cases of synchronous communication.

1. It is ideal where strong consistency and transaction guarantees are critical, such as financial transactions.
2. It is suitable for real-time applications requiring immediate feedback, like web applications – user data submission or query responses.
3. In scenarios that involve simple data fetch operations where the overhead of asynchronous communication is unnecessary.

#### 2.6.1.2. Asynchronous Communication: A Decoupled Symphony

Asynchronous communication introduces a decoupling layer between services. Instead of waiting for a response, the calling service sends a message to the target service and continues execution without blocking. The message is typically placed in a queue or message broker, a dedicated service that facilitates asynchronous communication. The target service retrieves the message from the queue at its own pace and processes it independently. The calling service may or may not receive a direct response,

depending on the specific implementation.

Consider a scenario where a user places an order on the e-commerce platform. The order service (calling service) sends an order confirmation message to a message queue. It then continues processing the request (e.g., sending payment confirmation emails) without waiting for confirmation from downstream services. Meanwhile, a separate worker service (target service) consumes messages from the queue and fulfills the order (e.g., generating invoices, shipping items).

Asynchronous communication brings some advantages, and it decouples the caller from the target service, allowing independent scaling. High request volumes at the calling service won't directly impact the target service, as messages are buffered in the queue. If a target service experiences failures, the overall system remains available. Messages remain in the queue and can be retried later, ensuring eventual completion of tasks. Asynchronous communication facilitates loose coupling, making the microservices architecture more resilient to failures in individual services.

It also brings some disadvantages to us. Implementing asynchronous communication patterns requires additional infrastructure like message brokers and queue management logic. This adds complexity to the architecture compared to synchronous communication. The calling service might not receive immediate feedback on the success or failure of the operation initiated at the target service. This can require additional mechanisms for tracking and handling errors. While data consistency is eventually guaranteed, there can be temporary inconsistencies between services during processing. Careful design and coordination are needed to ensure data integrity.

Here are some use cases of asynchronous communication.

1. It is suitable for scenarios with high traffic and workload, such as order processing systems in e-commerce.
2. It is ideal for tasks that can be processed asynchronously without immediate feedback, like sending emails or batch data processing.
3. It is used in systems built around event-driven principles, where services react to changes by processing and acting upon events as they occur.

### 2.6.2. Synchronous vs. Asynchronous Communication: Key Comparisons

To better understand the practical applications and implications of synchronous and asynchronous communication methods in microservices architecture, it is essential to compare them across several dimensions.

Synchronous communication tends to create tight coupling between services due to direct, immediate communication pathways, whereas asynchronous promotes loose coupling, making it easier to replace, upgrade, or scale individual services without impacting others. In synchronous communication, services are directly dependent on each other for timely responses, increasing inter-dependencies. However, in asynchronous method, services operate independently, relying on message brokers or queues to manage communication. Synchronous is suspected to be increasing latency, especially under high load conditions or network issues. On the other hand, asynchronous generally reduces latency impacts on the client-side, as processing happens independently. Synchronous limits scalability as the client must wait for the server, requiring significant resources for high concurrency. Asynchronous enhances scalability by allowing multiple requests to be handled simultaneously without waiting. In synchronous communication, a failed request can halt the operation, requiring complex error handling and retries. Asynchronous is better suited for failure handling as messages can be retried or stored for later processing, improving system resilience. In synchronous method, any downtime in a service can immediately affect all dependent services. Asynchronous method allows services to be able to function independently of each other's availability, as messages can be queued for processing when the system is back online.

Synchronous communication model is simpler to implement and debug due to the direct request-response model. However, asynchronous requires more complex setup and maintenance of messaging infrastructure, as well as thoughtful handling of asynchronous workflows. Synchronous is faster to develop and deploy new functionality due to straightforward interactions. Asynchronous may slow initial development due to more complex infrastructure requirements but can pay off in improved system robustness and scalability. Synchronous communication ensures strong consistency through immediate data updates, providing up-to-date data across

services. Asynchronous typically provides eventual consistency, meaning data might not be instantly synchronized across all services but will eventually become consistent.

### 2.6.3. Best Practices in Implementing Microservices Communication

Effectively implementing synchronous and asynchronous communication in a microservices architecture requires careful planning and adherence to best practices. Synchronous communication implements timeouts to avoid indefinite waiting periods and use circuit breakers to prevent cascading failures by halting operations when a service becomes unresponsive. It uses load balancers to distribute incoming requests evenly across multiple instances of a service to enhance availability and scalability. It designs idempotent operations in your services to ensure that repeated requests (due to outages or retries) don't lead to inconsistent results.

Asynchronous communication ensures reliable message delivery by using message brokers with mechanisms for acknowledgments, retries, and dead-letter queues for failed messages. It adopts event sourcing where state changes are logged as a series of events, enabling services to rebuild their state by processing these events. It implements idempotent message handling and deduplication logic to process the same message multiple times without adverse effects. It employs robust monitoring and alerting systems to track message flows, failures, and process bottlenecks, ensuring quick detection and response to issues. It employes schema versioning and backward-compatible changes to facilitate easier evolution of service interfaces without breaking existing functionality.

An e-commerce platform as a practical case study requires functionalities such as user authentication, product catalog management, order processing, payment, and notifications. Synchronous communication can be ideal here to provide immediate feedback during login or registration, ensuring users are authenticated in real-time. This can utilize asynchronous communication to handle operations like batch updates, inventory adjustments, and recommendations to ensure high throughput and eventual consistency. Order placement can use synchronous communication to ensure that user actions like adding to cart and checking out are immediately validated, while order fulfillment tasks such as sending confirmation emails, inventory updates, and shipping notifications can be handled asynchronously. Payment gateway integrates with third-

party payment services using synchronous communication for instant confirmation and transaction guarantees. For notifications, asynchronous communication for sending order updates, promotional messages, and alerts, leveraging messaging queues to ensure delivery even if the email or SMS service is temporarily unavailable.

### 2.7. A Comparative Analysis of Cryptographic Hashing Algorithms

In the digital realm, data integrity and authenticity are paramount. Cryptographic hash functions play a pivotal role in ensuring these qualities by generating a fixed-size string, known as a hash, from an arbitrary input. This hash acts like a digital fingerprint, uniquely representing the data without revealing its contents. Any alteration, even at the bit level, to the data will result in a completely different hash, exposing potential modifications or breaches.

However, not all hash functions are created equally. In this chapter, we will go deeper into the world of commonly used hashing algorithms, analyzing their strengths, weaknesses, and ideal use cases to provide you with the knowledge to select the most appropriate algorithm for your security needs.

Key characteristics of secure hash functions are related collision, preimage, and second preimage features. It would be incredibly difficult, if not impossible, to generate two different inputs that produce the same hash value (collision). Given a hash value, it should be computationally infeasible to determine the original data (preimage) that generated it. Given an input and its corresponding hash, it should be extremely difficult to find another input that generates the same hash (second preimage). Small changes in the input data should result in significant changes in the output hash.

#### 2.7.1. Message-Digest Algorithm 5 (MD5):

The MD5 is a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value from an arbitrary amount of input data. Below is an explanation of how MD5 works along with its main components. MD5 processes a variable-length message into a fixed-length output of 128 bits, typically rendered as a 32-character hexadecimal number. The basic steps of the MD5 algorithm are as follows:

Step 1: The message is padded so that its length (in bits) is congruent to 448 modulo

512. Padding is done by adding a single '1' bit, followed by enough '0' bits to make the message length 448 mod 512 bits. Then, the length of the original message (before padding) is appended as a 64-bit integer.

Step 2: A 128-bit buffer, divided into four 32-bit variables, is initialized with specific constants:

A = 0x67452301

B = 0xefcdab89

C = 0x98badcfe

D = 0x10325476

Step 3: The padded message is broken into 512-bit blocks. Each block is processed in a 4-round loop. Each round consists of 16 operations and involves one of the four 32-bit words (A, B, C, D).

Step 4: Four non-linear functions, given in Equations 2.1, 2.2, 2.3, and 2.4, are used to manipulate the data.

$$F(X, Y, Z) = (X \,\&\, Y) \mid (\sim X \,\&\, Z) \quad (2.1)$$

$$G(X, Y, Z) = (X \,\&\, Z) \mid (Y \,\&\, \sim Z) \quad (2.2)$$

$$H(X, Y, Z) = X \oplus Y \oplus Z \quad (2.3)$$

$$I(X, Y, Z) = Y \oplus (X \mid \sim Z) \quad (2.4)$$

Each round uses a different function and operates on chunks of 512 bits.

Step 5: Each 512-bit block is processed in a series of steps involving bitwise operations, additions, and modular arithmetic to transform the buffer values.

Step 6: The output is concatenated to form the final 128-bit hash value. After all the message blocks are processed, the final state of the buffer (A, B, C, D) is concatenated to form the final hash value. Each of the four 32-bit words in the buffer is transformed from little-endian to big-endian format and then concatenated to form the 128-bit MD5 hash.

The MD5 hash algorithm is used in legacy systems or applications where computational efficiency is a priority and strong collision resistance isn't essential

(e.g., checking file downloads for basic integrity). Table 2.1 provides a list of the advantages and disadvantages of MD5 hashing algorithm.

Table 2.1 The advantages and disadvantages of MD5 hashing algorithm

| PROS | CONS |
|---|---|
| It is simple to apply. | MD5 has been shown to be vulnerable to collision attacks, where two different inputs produce the same hash value. This makes it unsuitable for security-critical applications. |
| MD5 is widely used and supported by many applications and systems | It is considered to be broken and insecure for cryptographic purposes due to these vulnerabilities. |
| It is historically efficient. | MD5 has been deprecated by many security experts and organizations in favor of more secure hashing algorithms like SHA-256. |
| MD5 produces a fixed-size hash value, regardless of the input data size. | The hashing is one directional which means once the string is hashed or encoded, it can not be decoded back. |

### 2.7.2. Secure Hash Algorithm 1 (SHA-1):

SHA-1 is a cryptographic hashing algorithm that produces a fixed-size hash value from input data of any size. The working logic of the SHA-1 algorithm is as follows:

Step1: The input data is padded to ensure that its length is a multiple of 512 bits. Padding includes adding a single '1' bit followed by enough '0' bits to reach the desired length, as well as a 64-bit representation of the original message length.

Step 2: The padded data is then divided into blocks of 512 bits each.

Step 3: A message schedule array of 80 32-bit words is created to process the input blocks.

Step 4: The algorithm uses five 32-bit initial hash values (known as H0 to H4) to start the hashing process.

Step 5: The hashing process involves several rounds of computation using bitwise operations such as XOR, AND, OR, and shifts, along with logical functions like majority, parity, and rotate operations.

Step 6: Each block of data undergoes a series of bitwise operations and transformations in the compression function to update the hash values.

Step 7: After processing all blocks, the final hash value is obtained by concatenating the five 32-bit hash values (H0 to H4) in the order they were calculated.

Step 8: The resulting hash value is a 160-bit (20-byte) hexadecimal string, which serves as a unique representation of the input data.

Table 2.2 The advantages and disadvantages of SHA-1

| PROS | CONS |
|---|---|
| Historical Prevalence: SHA-1 was widely adopted for many years, establishing it as a common standard for many systems. This meant that a lot of infrastructure was designed to work with SHA-1, leading to a period of easy interoperability. | Vulnerabilities: SHA-1 has been proven vulnerable to collision attacks. This means that two different data inputs can produce the same hash output, which compromises the integrity of the algorithm for security purposes. |
| Speed: SHA-1 is relatively fast to compute, which led to its popularity for applications where speed was an important factor (such as real-time data processing or systems with limited computational resources). | Deprecated for Security Use: Due to the vulnerabilities, SHA-1 is now considered deprecated for security-sensitive applications. Organizations like NIST (National Institute of Standards and Technology) recommend against its use for cryptographic security. |
| Design Simplicity: The design of SHA-1 is relatively straightforward, making it easy to understand and implement in software and hardware. | Replacement by More Secure Algorithms: Newer algorithms like SHA-256 and SHA-3 are designed to be more secure and are recommended for current use. |
| | Loss of Trust: Since the discovery of practical attacks against SHA-1, its trustworthiness in the cryptographic community has substantially diminished, leading to an increased risk when used in new systems. |
| | Compromised Certification: Digital certificates that rely on SHA-1 are at risk of being compromised, which affects the overall security of the system they are meant to protect. |

Overall, the SHA-1 algorithm aims to produce a unique and fixed-size hash value that represents the input data securely and is widely used in various applications for data integrity verification and digital signatures. Table 2.2 lists the advantages and disadvantages of SHA-1 hashing algorithm.

Despite its weaknesses, SHA-1 has been used in a variety of applications, particularly before stronger hashing algorithms became standard. Some of its historical and, in a few cases, ongoing use cases include the ones given in Table 2.3.

Table 2.3 Use cases of SHA-1

| Use Case | Definition |
|---|---|
| Version Control Systems | SHA-1 was used by systems like Git for identifying commits. Each commit is associated with a SHA-1 hash that ensures the integrity of the data as it was at the time of the commit. |
| Digital Certificates | It was used within SSL/TLS certificates as a means of ensuring that the public key contained within the certificate corresponds to the private key held by the entity presenting the certificate. |
| Software Distribution | SHA-1 hashes have been provided alongside software downloads so that users can verify the integrity of the software package, ensuring no corruption or tampering has occurred during download. |
| Cryptographic Signature Verification | Secure email, software updates, and digital document signing often used SHA-1 as part of the digital signature process to ensure the integrity and authenticity of the message or document. |
| Data Integrity | Various backup systems and databases might use SHA-1 hashes to check the integrity of the data on regular intervals to identify any corruption or changes. |
| Password Storage | Some systems used SHA-1 hashes to store passwords. Though this is not secure by modern standards, it was common in legacy systems. |

However, due to the vulnerabilities found in SHA-1 (namely the fact that it is no longer collision-resistant), it is being phased out in most applications. Modern applications that require cryptographic security are encouraged to use stronger hashes such as SHA-256 or SHA-3. In many cases where SHA-1 might still be in use, it is generally due to

legacy reasons where systems have not yet been updated or because the environment does not require high security (though such use is becoming increasingly rare and is not recommended).

### 2.9.3. Secure Hash Algorithm 2 (SHA-2) Family:

The SHA-2 family of cryptographic hash functions includes several variants, primarily differing by the output bit length: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. The most commonly used are SHA-256 and SHA-512; here's a brief overview of the working logic of these algorithms, which share a similar structure.

In the initialization step, each variant of SHA-2 starts with a distinct set of initial hash values. These are constants derived from the fractional parts of the square roots of the first eight prime numbers, and the count of these values depends on the variant (e.g., SHA-256 starts with eight initial hash values). In the next step, which is preprocessing, the input message undergoes preprocessing, which involves three stages: First, the original message is padded so its length is 448 modulo 512 for SHA-256, or 896 modulo 1024 for SHA-512. Padding involves appending a '1' bit, followed by a series of '0' bits, and finally a 64-bit (for SHA-256) or a 128-bit (for SHA-512) length field denoting the original message length. Then, the padded message is divided into fixed-sized blocks (512 bits for SHA-256 and 1024 bits for SHA-512). Afterwards, each block is broken into 32-bit words for SHA-256 or 64-bit words for SHA-512. These words are then expanded into a larger message schedule array, which consists of 64 words for SHA-256 or 80 words for SHA-512. Each message block is processed using a main loop that updates the hash values. The main loop uses a series of logical functions and operations.

A set of working variables are initialized with the current hash values. Each word of the expanded message schedule is processed sequentially. The processing includes bitwise operations, conditional operations, and additions based on a round constant unique to each round of the algorithm. The working variables are combined and manipulated with these operations. After completing the main loop, the working variables are added to the current hash values to produce the updated hash values. This process is repeated for every block. After all message blocks have been processed, the

output is produced by concatenating the final hash values. For SHA-256, this results in a 256-bit (32-byte) hash, and for SHA-512, you get a 512-bit (64-byte) hash.

Table 2.4 The advantages and disadvantages of SHA-2

| PROS | CONS |
|---|---|
| Security: SHA-2 algorithms are considered secure against known attacks. This includes strong resistance to preimage attacks (finding a message with a specific hash) and collision attacks (finding two messages that hash to the same value). | Performance on 32-bit Platforms: SHA-256 and SHA-512 might not perform as well on 32-bit systems since they are optimized for 64-bit operations. However, this is less of a concern as 64-bit processors have become the standard in most devices |
| Widely Adopted: SHA-2 is the current standard for secure hashing functions and is used in a variety of industry applications, including SSL/TLS certificates for Internet security, blockchain technologies, and secure file transfer protocols. | Increasing Computational Resources for Brute force Protection: As our computational power increases, hash functions need to be resilient to brute force attacks, requiring the creation of even more complex algorithms. This is more of a future-proofing concern |
| Multiple Variants: The SHA-2 family includes a range of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512 and its variations), offering flexibility in terms of speed and security level. | Complexity: The internal workings of SHA-2 are more complex than its predecessor, SHA-1, which can make it more challenging to understand and correctly implement from a developer's perspective. |
| Government Approval: SHA-2 algorithms have been approved for use by various government agencies around the world, including the National Institute of Standards and Technology (NIST) in the United States. | Not Quantum-Safe: With the potential future development of quantum computing, SHA-2, like all current cryptographic hash functions, may be vulnerable to attacks by quantum computers. |
| Improved Security Over SHA-1: SHA-2 addresses the security issues found in SHA-1, making it a suitable replacement to maintain the integrity and security of data. | Not Immune to Future Threats: While SHA-2 is currently secure, the advancement of technologies like quantum computing may render it vulnerable in the future. However, this is a limitation that all current cryptographic hash functions share. |

The SHA-2 family is designed to work in a way that a small change in the input message will result in a significantly different hash, exhibiting a property known as

"avalanche effect." This makes it computationally infeasible to generate the same hash from two different inputs (collision resistance) or to deduce the original input from the hash value (pre-image resistance). Due to their robustness, SHA-2 algorithms remain secure and widely utilized in various applications, including digital certificates, cryptographic currency systems, and securing passwords. The SHA-2 family of hashing algorithms is widely respected and remains integral to modern cryptographic systems. Table 2.4 shows some of the primary advantages and disadvantages associated with it.

SHA-2 is a family of cryptographic hash functions designed by the National Security Agency (NSA). The most common members of this family are SHA-224, SHA-256, SHA-384, and SHA-512. Table 2.5 provides some of the most common use cases of this algorithm.

Table 2.5 Use cases of SHA-2

| Use Case | Definition |
|---|---|
| Data Integrity Verification | File verification ensures files have not been altered. Hash values are often provided with downloads to verify the integrity of the file.<br>Backup verification confirms that backups have remained unchanged. |
| Password Hashing | Storing hashing passwords before storing them in databases to ensure they cannot be easily retrieved or compromised. |
| Digital Signatures | Authentication provides a method for verifying the authenticity and integrity of a message, software, or digital document.<br>Non-repudiation ensures that a signer cannot deny the validity of their signature on a document. |
| TLS/SSL Certificates | SHA-2 is used in the generation of digital certificates that secure web communications. |
| Blockchain and Cryptocurrencies | Hash functions are fundamental to the integrity and security of blockchain transactions.<br>Hashing algorithms are used in the proof-of-work mechanism for mining cryptocurrencies like Bitcoin. |
| Code Signing | Software distribution verifies that software or firmware has not been tampered with and is from the official source. |

| | |
|---|---|
| Data Deduplication | Storage optimization identifies duplicate data segments by comparing their hash values, saving storage space. |
| Random Number Generation | Use hash functions as part of generating cryptographically secure random numbers. |
| Digital Forensics | Evidence integrity ensure digital evidence has not been altered by hashing files and verifying consistency over time. |
| Message Authentication Codes (MACs) | Data integrity and authentication ensure the authenticity and integrity of a message with a secret key. |

### 2.7.4. BLAKE2 (BLAKE2s and BLAKE2b):

BLAKE2 is a cryptographic hash function designed to be faster than MD5, SHA-1, and SHA-2, while providing a higher level of security than these older algorithms. The working logic of BLAKE2 is based on a combination of the HAIFA structure (Hirose's variant of the Merkle–Damgård construction) and the ChaCha stream cipher. Here is an overview of its core components and how it works:

A set of predefined constants that serve as the initial state of the hash function. The input data is divided into fixed-size blocks (e.g., 512 bits for BLAKE2b and 256 bits for BLAKE2s). A key part of the compression function, derived from the ChaCha stream cipher, that processes the message blocks along with the internal state. Predefined permutations specify which words are mixed together, adding diffusion.

Initialization starts with an initial state derived from the initialization vector (IV) and any optional parameters (like a key, a salt, or a personalization string). The message is divided into blocks. Each block is processed with a mixing function that incorporates both the message block and the internal state. The mixing function (G function) operates on pairs of words, mixing them and ensuring high diffusion of the bits. For each block, the state is processed with the G function, which performs a series of XOR, addition, and rotation operations on pairs of words from the state. This mixing ensures that small changes in the input lead to significantly different hash outputs. The words in the state are permuted according to predefined patterns before the next round of G mixing. This ensures that all parts of the state are well-mixed and that the hash function has good diffusion properties. After processing all blocks, a finalization step combines the internal state values to produce the final hash. Outputs the specified number of bits,

truncating if necessary (e.g., BLAKE2b can produce up to 512-bit hashes but can be truncated to shorter lengths as needed).

There are two BLAKE2 variants: BLAKE2b optimized for 64-bit platforms and can handle larger bit-lengths efficiently and BLAKE2s optimized for 8- to 32-bit platforms and is more efficient for smaller bit-lengths. Table 2.6 provides a list of advantages and disadvantages of BLAKE2 hashing algorithm.

Table 2.6 The advantages and disadvantages of BLAKE2 hashing algorithm

| PROS | CONS |
|---|---|
| High performance | Complexity compared to older hash functions |
| Faster than MD5, SHA-1 and SHA-2 | More complex than MD5 or SHA-1 |
| Optimized for various platforms (BLAKE2b for 64-bit, BLAKE2s for 8- to 32-bit) | |
| Strong Security | Relatively New |
| Comparable to SHA-3 | Newer compared to SHA-2, less available in older systems |
| Resistant to common cryptographic attacks | Less real-world scrutiny compared to SHA-256 |
| Versatility | Trade-Offs in Design |
| Supports keying (for HMAC), salting, and personalization | Specialized variants require choosing the appropriate one for the platform |
| customizable hash length | |
| Ease of Integration | |
| Simple API | |
| Can be used as a drop-in replacement for other hash functions | |
| Standardization | |
| Adopted in many modern cryptographic libraries and standards | |

BLAKE2 is a cryptographic hash function known for its high performance and security. Its versatility makes it suitable for a wide range of applications. Table 2.7 provides a list of use cases of BLAKE2 hashing algorithm. BLAKE2's efficiency and

cryptographic strength make it a preferred choice in many areas where fast and secure hash functions are required.

Table 2.7 Use cases of BLAKE2 hashing algorithm

| Use Case | Definition |
|---|---|
| Data Integrity Verification | Ensuring that files or data have not been altered during transfer or storage.<br>Generating and verifying checksums for software distribution. |
| Password Hashing | Securing stored passwords by transforming them into a non-reversible hash.<br>Commonly used in combination with other algorithms (e.g., Bcrypt, Argon2) for enhanced security. |
| Cryptographic Applications | Used in digital signatures and public key infrastructures to ensure message integrity.<br>Part of HMAC (Hash-based Message Authentication Code) implementations for verifying the authenticity and integrity of messages. |
| File Synchronization and Deduplication | Identifying and managing duplicate files by comparing their hash values.<br>Used in backup and file synchronization software to save storage space and bandwidth. |
| Key Derivation Functions (KDF) | Generating cryptographic keys from passwords or other data.<br>Employed in protocols like PBKDF2 to create keys for encryption from a user's password. |
| Content Addressable Storage (CAS) | Addressing and retrieving data by its hash value for efficient storage solutions.<br>Ensures that identical content shares the same storage reference. |
| Version Control Systems | Used in systems like Git to create unique identifiers for commits, ensuring data consistency and traceability. |
| Security Protocols | Implementation in various cryptographic protocols and frameworks, including SSL/TLS for secure communications. |
| Blockchain and Cryptocurrencies | Ensuring data integrity and block validation in certain blockchain implementations. |

### 2.7.5. Secure Hash Algorithm-3 (Keccak)

SHA-3 is a cryptographic hash function developed by the National Institute of

Standards and Technology as a part of the Secure Hash Algorithm competition. The competition was initiated in 2007 to develop a new hash function that could replace SHA-2, which was starting to show some weaknesses. After a rigorous selection process, the algorithm Keccak, designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, was chosen as the winner and officially designated as SHA-3 in 2015. It was approved by NIST as a federal standard for cryptographic applications.

SHA-3 is designed to be secure, efficient, and resistant to various types of attacks. It is widely used in digital signatures, certificates, and other cryptographic applications to ensure data integrity and authenticity. The algorithm has been widely adopted and continues to be a critical component of cybersecurity systems worldwide.

The digital landscape is constantly evolving, and the need for robust cryptographic tools to ensure data security grows ever stronger. Cryptographic hash functions play a crucial role in this domain, generating unique "fingerprints" of digital data. Enter SHA-3 (Keccak), the latest member of the Secure Hash Algorithm family, designed to address potential weaknesses in its predecessors and offer unparalleled security. This article delves into the intricate details of SHA-3, exploring its construction, strengths, and potential use cases.

Prior to SHA-3, the cryptographic community witnessed concerns regarding the collision resistance of MD5 and SHA-1 algorithms. These vulnerabilities could potentially allow attackers to forge digital signatures or tamper with data undetected. Recognizing this need for a more secure solution, NIST (National Institute of Standards and Technology) launched a competition in 2007 to develop a new cryptographic hash function. Keccak, designed by Guido Bertoni et al., emerged victorious in 2012, paving the way for a new era of secure hashing (SHA-3).

SHA-3 (Secure Hash Algorithm 3) is a member of the Secure Hash Algorithm family, which functions differently from the previous standards (SHA-1, SHA-2). The SHA-3 algorithm is based on a cryptographic construction called Keccak, which is fundamentally different from the Merkle–Damgård construction used by its predecessors. Here's an outline of its working logic:

The core idea of SHA-3 is the sponge construction, which comprises two main phases: absorbing and squeezing. SHA-3 initializes the state with a fixed size (often 1600 bits). The input message is padded and divided into fixed-size blocks. Each block is XORed into a subset of the state, and the state is then transformed via a permutation function called Keccak-f. This permutation function operates on the state by performing a series of rounds. Each round consists of five steps: Theta, Rho, Pi, Chi, and Iota. These steps involve bitwise operations (XOR, AND, NOT) and permutations.

Theta: Ensures diffusion by mixing bits across all lanes (partitions of the state).

Rho: Permutes the bits within positions.

Pi: Permutes the order of the lanes.

Chi: Non-linear operation that ensures confusion, making it resistant to structural attacks.

Iota: Adds a round-constant to the state to prevent symmetries.

After all blocks are absorbed and the final state is reached, the output hash is generated by extracting bits from the state in chunks. If additional output bits are needed, the permutation function is applied again, and more bits are extracted until the desired hash length is achieved. The message is padded in a specific way to fit into the fixed-size blocks: Pads the message with a '1', followed by some '0's, and then appends a '1' at the end of the block (or multiple blocks).

Security Characteristics

It is computationally infeasible to find two different inputs that produce the same hash output (Collision Resistance). It is computationally infeasible to reconstruct the original input from its hash output (Pre-image Resistance). It is computationally infeasible to find a different input that has the same hash output as a given input (Second Pre-image Resistance).

SHA-3's unique properties and its use of the sponge construction make it highly secure and flexible. Its resistance to known cryptographic attacks, like length extension attacks and collision attacks, makes it a robust choice for a variety of applications requiring secure hashing. Table 2.8 and 2.9 show some of the advantages and disadvantages and use cases of the SHA3 hashing algorithm, respectively.

Table 2.8 The advantages and disadvantages of SHA-3

| PROS | CONS |
|---|---|
| SHA-3 is designed to be secure against various cryptographic attacks, including collision attacks and pre-image attacks. Its resistance to attacks makes it a reliable choice for ensuring data integrity and authenticity. | One potential disadvantage of SHA-3 is that it is not backward compatible with older hash functions such as SHA-1 and SHA-2. This may require some modifications in systems and applications that were previously using the older algorithms. |
| SHA-3 is designed to be efficient in terms of computational resources and performance. It is optimized for both hardware and software implementations, making it suitable for a wide range of applications. | While SHA-3 has been widely adopted in the cryptographic community, there may still be some reluctance or skepticism from organizations or developers who are more comfortable with older hash functions like SHA-2. It may take time for widespread adoption of SHA-3 in all applications. |
| SHA-3 has been officially adopted as a federal standard by NIST, making it widely recognized and accepted in the cryptographic community. It is used in various security protocols, digital signatures, and cryptographic applications. | Although SHA-3 is designed to be efficient, there may still be some performance impact when compared to other hash functions, especially in resource-constrained environments or high-throughput applications. Developers need to consider performance considerations when implementing SHA-3 in their systems. |

Table 2.9 The use cases and definitions of SHA-3

| Use Case | Definition |
|---|---|
| Data Integrity Verification | Organizations often need to ensure the data they transmit or store has not been altered. For example, when software companies distribute updates, they provide a SHA-3 hash of the update file. Users can download the file and compute its SHA-3 hash locally. By comparing their computed hash against the provided one, they can verify the file's integrity, ensuring it hasn't been tampered with during transmission. |
| Digital Signatures | In digital communication, authenticity and integrity of messages are crucial. Consider a legal document signed digitally. The document's content is hashed using SHA-3, and the resulting digest is then encrypted with the sender's private key, creating a digital signature. Recipients can |

| | |
|---|---|
| | decrypt the signature using the sender's public key and then compare the decrypted hash to the independently calculated SHA-3 hash of the document. If they match, the recipient can be confident that the document is authentic and unaltered. |
| Cryptographic Hash Functions in Blockchains | Blockchain technology relies heavily on hash functions for security and linking data blocks. For instance, newer blockchain platforms might utilize SHA-3 instead of SHA-256 for better security margins. Each block contains the SHA-3 hash of the previous block, forming a chain. Any alteration in a block would change its hash, breaking the chain, which immediately indicates tampering. |
| Password Hashing | When users create accounts on a secure platform, their passwords should be stored securely. The platform can hash each password using SHA-3 before storage. During login, the user's entered password is hashed again with SHA-3 and compared to the stored hash. Successful comparison authenticates the user, while the use of hashing protects the password from being directly compromised if the database is breached. |
| Random Number Generation | Random number generation is critical in cryptographic applications. For example, in creating secure cryptographic keys, an application might use SHA-3 as part of its algorithm to ensure a high level of entropy and unpredictability. This strengthens the security of generated keys against potential attackers trying to predict or reproduce the random sequence. |
| Message Authentication Codes (HMAC) | When sending sensitive information over a network, it's crucial to ensure both the integrity and authenticity of the data. An HMAC using SHA-3 (HMAC-SHA-3) can be employed. The sender creates a secure message by combining the data with a secret key and hashing it using SHA-3. The resulting MAC is sent along with the message. The receiver, who also possesses the secret key, performs the same HMAC-SHA-3 operation and compares the result with the received MAC. If they match, it confirms that the message is authentic and has not been tampered with. |
| Certificate Fingerprints | In the realm of Public Key Infrastructure (PKI), certificate authorities (CAs) issue digital certificates that authenticate entities such as websites or individuals. These certificates contain fingerprints, which are essentially hashes of the certificate content. For example, a website's SSL certificate might include a SHA-3 fingerprint. Users' web browsers compare the fingerprint on the certificate with a locally computed SHA-3 hash. A match verifies that the certificate is genuine and the website can |

| | |
|---|---|
| | be trusted. |
| Proof of Existence | Proof of existence services allow users to prove that a document existed at a certain time without revealing the document itself. Users can hash their document using SHA-3 and then register the hash on a public ledger or blockchain. For instance, an author could hash a manuscript using SHA-3 and store the hash in a blockchain. Later, they can prove the document's existence at the time of hashing by showing that the hash corresponds to the document, without revealing its content, thus protecting their intellectual property. |

### 2.7.6. Choosing the Right Hashing Algorithm: A Strategic Decision

The optimal hashing algorithm selection hinges on a careful consideration of security requirements, performance needs, and compatibility with existing infrastructure. For password storage and message authentication: Prioritize collision resistance and security with algorithms like SHA-256 or SHA-3.

For legacy systems or non-critical data integrity checks: MD5 or SHA-1 might be acceptable if their limitations are understood. For performance-critical applications: Consider SHA-2 or BLAKE2, balancing security with efficiency. Remember, constantly evolving security threats necessitate staying updated on the latest vulnerabilities and recommendations from reputable security agencies.

## 2.8. What is JSON Web Token (JWT)?

JWT is an open standard (RFC 7519) used for securely transmitting information between nodes as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

In the conventional methods, user credentials used to be sent through requests, which made it easier to hack into. With JWT tokens, the credentials are hashed and more difficult to tamper. JWTs are widely used for authentication and authorization purposes. They are commonly used in stateless authentication mechanisms because of their ability to securely transmit user information and claims between a client and a server.

### 2.8.1. Structure of JWT

A JWT typically consists of three parts separated by dots (.): Header, Payload, and Signature. A JWT looks like hhhhhhh.pppppppp.sssssssss

The header contains metadata about the token, including the type of token (JWT) and the signing algorithm used (e.g., HMAC SHA256 or RSA). Some other required information can be included, too.

Payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims:

- Registered claims are predefined claims such as iss (issuer), exp (expiration time), sub (subject), and aud (audience).
- Public claims can be defined at will, but to avoid collisions, they should be registered or use a namespace.
- Private claims are custom claims created to share information between parties that agree on using them.

The signature is created by taking the encoded header, the encoded payload, a secret, and the algorithm specified in the header and applying a digital signature to them.

### 2.8.2. Advantages of Using JWT

JWTs contain all necessary information within the token itself, reducing the need for maintaining session state on the server. Since the server does not need to store session information, this can enhance scalability for distributed systems. JWTs are compact in size, making them suitable for being passed in the URL, POST parameters, or inside an HTTP header. They contain all necessary information about the user, thus reducing the number of database lookups. JWTs are signed, ensuring the integrity of the token. They can be verified using the secret key (HMAC) or public key (RSA or ECDSA). Although JWTs are usually signed, they can also be encrypted for added security. JWTs can be used across different technologies and platforms due to their JSON format. In a microservices architecture, JWTs can be used to authenticate users across different services without needing a centralized authentication server for each request.

### 2.8.3. Disadvantages of Using JWT

The size of the JWT can become large, especially when including many claims, which can lead to increased bandwidth usage. If not managed properly, tokens can be susceptible to replay attacks. It's essential to set an appropriate expiration time. JWTs are stateless, meaning once issued, they cannot be easily revoked without implementing additional mechanisms, like maintaining a token blacklist. Using weak algorithms or improperly handling the signature verification process can introduce vulnerabilities. Implementing JWT securely requires understanding the correct use of cryptographic algorithms and proper handling of token validation and expiration.

JWTs provide a powerful and flexible way to handle authentication and authorization in modern web applications. Their stateless nature, security features, and efficiency make them an excellent choice for systems requiring scalable and interoperable solutions. However, it's crucial to be aware of the potential security and implementation complexities to fully leverage the benefits of JWTs while mitigating their risks.

Securing JSON Web Tokens (JWTs) is crucial for maintaining the integrity and confidentiality of your authentication system. Here are some strategies to secure JWTs and prevent common exploitation methods:

Use strong and widely accepted algorithms like HS512 (HMAC with SHA-512) for symmetric key signing or RS256 (RSA with SHA-256) for asymmetric key signing. Avoid using weaker algorithms or allowing algorithm downgrading. Store signing keys in a secure, access-controlled environment. Rotate keys regularly and ensure old keys are properly retired. Set a reasonable expiration time (exp claim) to minimize the window in which a stolen token is valid. Issue short-lived access tokens and use refresh tokens to obtain new access tokens, reducing the impact of a compromised token. Ensure that tokens are correctly validated on each request, checking the signature, expiration, and other relevant claims (like issuer (iss), audience (aud), and not-before (nbf)). Always transmit JWTs over HTTPS to protect against interception via man-in-the-middle attacks. Prefer storing tokens in secure, HTTP-only cookies over localStorage or sessionStorage to mitigate XSS attacks. Configure Cross-Origin

Resource Sharing (CORS) properly to prevent unauthorized domains from accessing your API. Use well-maintained libraries for JWT verification instead of implementing custom verification logic. Ensure that the aud (audience) and iss (issuer) claims are set and verified to ensure the token is intended for the correct recipient and issued by a trusted source.

### 2.8.4. Common Ways Hackers Exploit JWT Tokens

Hackers use various techniques such as token theft interception, cross-site scripting (XSS), replay attacks, algorithm confusion, key guessing, tampering, key leakage, improper validation, and exploiting vulnerabilities in JWT libraries to obtain JWT tokens. Token Theft Interception designates capturing tokens in transit if HTTPS is not used. The theft of tokens via cross-site scripting attacks is a potential vulnerability when tokens are stored in an accessible location, such as localStorage. Replay attacks are defined as the reuse of a valid token to perform unauthorized actions within the token's valid time frame. Algorithm confusion exploits incorrect handling of the alg header, such as changing alg to none or a weaker algorithm if the server does not validate the alg properly. Weak key guessing is used to guess or brute-force weak signing keys. Tampering indicates modifying the payload of the token (e.g., changing the user role) if the token signature is not properly verified.

The third parties can gain access to the server or environment where the signing keys are stored due to poor key management practices. Improper validation permits the inappropriate utilization of tokens, particularly in instances where critical claims (e.g., exp, iss, aud) remain unvalidated. They also take advantage of vulnerabilities in outdated or improperly used JWT libraries.

It is imperative to implement robust security measures to safeguard the JWT token. This entails utilising robust encryption techniques and ensuring the confidentiality and integrity of the associated keys. Furthermore, it is essential to establish effective token expiry and validation mechanisms, as well as guarantee the secure storage and transmission of tokens. Additionally, it is crucial to conduct regular audits and updates to security practices to address emerging vulnerabilities and threats. By following these best practices, you can significantly reduce the risk of JWT-related security issues in your application.

## 3. MATERIALS AND THE CODE BASE

In this study, the application has six loosely coupled micro services where each one has an individual function or responsibility within the system. The application is a typical restaurant system where users can order food and see the status of their order, restaurants can see the orders and process them and change the status of the order and finally the admin user who can control everything in the system including creating users or restaurants. The goal of this application is to see the difference between different signing algorithms in JWT (JSON Web Token) mechanism. The system was developed in Spring Boot framework based on Java language, as the database, we have used MySQL in the authentication service. For the ease of development purposes, the database was set in a way that it will be created at the service startup and will be dropped once the service is turned off.

The system uses Eureka dependency created by Netflix company to register all the services in the same network as individual clients. This helps the system work smoothly and different microservices discover each other easily. In the development of the system, Spring Boot framework which was created in Java language has been used. The system allows the developer to inject different dependencies to facilitate the development process and provides very useful annotations which helps the developer save a lot of valuable time by providing important boilerplate code annotations.

Figure 3.1 shows the basic components of the application and the relationship between them. The client sends a request to the API gateway. Then the gateway sends the user credentials that it gets from the user to the auth-service where the credentials are checked against the registries in the database. If the credentials are valid, the system creates a token and sends it back and this token is validated again in the API gateway. If the token s valid and then another filter is used to see if the user credentials which are passed within the payload of the JWT are valid and authorized ones to reach and consume specific endpoints. If everything is fine, the end user is allowed to reach the

backend service.

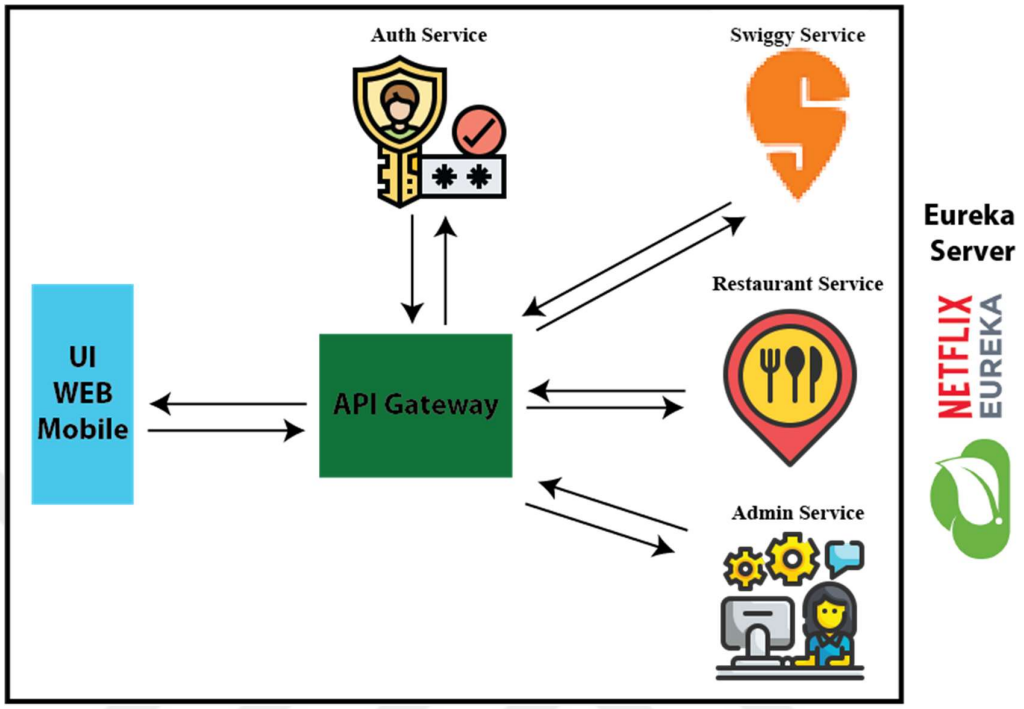


Figure 3.1 The components of the architecture

### 3.1. Admin Service Application

The Admin Service Application is a software system designed to manage administrative tasks. It allows administrators to access a specific service by navigating to a designated website address. When the application is turned on, it waits for someone (like an administrator) to visit. When an admin visits the `/admin` address, the "controller" (Admin Controller) receives the visitor. The controller prepares a welcoming message, "Welcome to Admin service," and gives it to the client.

Main Application Start Point is where the system admin manages all the admin responsibilities and fulfill the tasks. Admin Controller is used to provide the http endpoints of the service to the rest of the system. Admin Service carries and deals with all the business logic of the service. It prepares the welcoming message that the receptionist will give to the visitors.

### 3.2. Authentication Service with JWT

The Authentication Service is a digital security system that manages user identities

and permissions. It ensures that only authorized users can access certain parts of an application and protects the system from unauthorized access.

User Management manages user information like names, usernames, passwords, and roles (e.g., User, Restaurant, Admin). Just like a membership card providing access to different areas, roles determine what a user can do within the system. For instance, 'Admin' has more privileges than a 'User.'

Token Repository as a secure storage or logbook that keeps track of tokens issued to users. Tokens are like digital keys. There are two types: Access Tokens for short-term use and Refresh Tokens for obtaining new Access Tokens without logging in again. User tokens are managed and checked to ensure they are valid and not logged out (like checking if a key has been revoked). Each user can have multiple tokens. Authentication response sends back a pair of tokens and a message, much like getting a receipt along with your new key when a user successfully logs in. Token filters act like security guards at entry points, ensuring every request has a valid token before allowing access. If a request doesn't have a valid token, access is denied, much like a guard not letting you enter a room without the proper keycard. Security configurations define who can access what. For instance, certain URLs are restricted to only 'Admin' users. It handles user registration, login, token refresh, and logout processes securely, similar to procedures at a secure facility. Controllers serve as public interfaces that users interact with Demo Controller is used to demonstrate secured URLs and Authentication Controller provides endpoints for user registration, login, and token refresh.

Users provide their details (like signing up for a membership). Upon success, they receive digital keys (tokens). When users attempt to access restricted areas of the application, their tokens are checked to see if they have valid access (like showing a security pass at checkpoints). When users log in or refresh their session without logging out, new tokens are issued. If a user logs out, their tokens are marked as invalid. For each interaction, like accessing data or services, the system ensures only valid and appropriate tokens are allowed through, maintaining a high security standard.

It ensures that only authorized users can access sensitive information and functionality,

differentiates access levels and permissions based on user roles, providing flexibility and security, allows seamless authentication and session management for users, making the experience smoother without repeatedly logging in.

The system protects data and resources by verifying user identity and permissions. Digital keys (tokens) control what each user can access, ensuring security and proper resource allocation. Different user roles provide the flexibility to manage various types of users and their access levels appropriately. This service essentially acts as a vigilant gatekeeper, ensuring that only rightful users can enter and interact with the application, thereby maintaining the security and integrity of the system.

### 3.3. API Gateway Application

Application Startup (Starting the Service) initializes and starts the API Gateway service, making it ready to handle requests. When the service starts, it gets ready to handle various tasks related to managing requests from users and other services. Security and Authentication (Ensuring Safe Access) uses special coded tokens to authenticate users and ensure they have permission to access certain resources. It checks if the provided security token is valid. It identifies the role of the user based on the provided token. It uses secret keys to sign and verify tokens for security purposes. This component specifies which parts of the service are open to everyone and which parts require special permissions. It lists specific paths (like registration or token generation) that are accessible without security checks, helps determine if incoming requests need to pass through security checks. The service checks every incoming request to ensure that it has the necessary security token if accessing secure areas. It verifies if requests include an authorization header, strips and processes the token from the request header and responds with errors if the token is missing or invalid, preventing unauthorized access. Configuration (Setting Up Dependencies) part of the service sets up necessary tools that help make requests to other services within the system and prepares a tool that makes it easier to handle requests to other services.

In essence, this API Gateway Application ensures that requests to an online service are properly authenticated and routed, protecting against unauthorized access while specifying which routes are freely accessible. This helps maintain a secure and orderly

flow of information in the system.

### 3.4. Restaurant Service Application

The Restaurant Service Application is designed to smoothly manage restaurant orders by allowing users to check their order status and receive relevant information about their orders. The application ensures that all necessary operations are handled, from starting the service and processing user requests to managing order data and providing structured information about each order. Application Startup initializes and starts the restaurant service, making it ready to handle operations related to restaurant orders. Key action ensures that the service is registered in a central directory so that other services can discover and communicate with it. The controller component acts as the main point of interaction for users, allowing them to send requests and receive responses. Greeting message provides a simple welcome message to users and allows them to check the status of their orders by providing an order ID. The Data Access Object part manages the retrieval and storage of order information. Order Generation simulates a database of orders with various details like order name, quantity, price, date, status, and delivery estimate. Get Order retrieves the details of a specific order based on the provided order ID. Data Transfer Objects are templates used to create and structure the data for orders. Order Details contains information such as order id, name, quantity, price, order date, status, and estimated delivery time. The service component contains the main business logic for handling orders. Greeting provides a welcome message for the restaurant service. Order Information Retrieval connects with the data access object to get details of a specific order.

### 3.5. Swiggy Service Application

This application is designed to manage and track food delivery orders. The `SwiggyServiceApplication` class is the entry point. This is the part of the application that gets things started when you run the program. It initiates the whole application and gets everything up and running. Swiggy App Service is where the main features related to the Swiggy service are implemented. Greeting Message provides a welcoming message for anyone using the service. Order Status Check allows users to check the status of their food orders. The `OrderResponseDTO` is a special object used to transfer data about orders. It contains order details like ID, name, quantity, price,

order date, status, and estimated delivery time. Swiggy App Controller part acts like a receptionist. It takes in requests from users and delegates them within the service. When users navigate to the home section, it shows them a greeting message. When users want to check the status of an order, they provide the order ID, and this controller fetches the details for them. The `SwiggyAppConfig` class manages configuration settings. Load Balancer ensures that the application can handle multiple requests smoothly by distributing them evenly. Rest Template tool helps the application communicate with other services over the system. Restaurant Service Client part deals with contacting external services, like the actual restaurants to fetch order statuses and fetches order status from the restaurant service: It sends a request to the restaurant's system to get the latest status of an order.

The "Swiggy Service " is a well-organized service composed of multiple parts working together to provide food delivery services. It begins by launching the main application, handles various operations related to orders, communicates with external restaurant services, and provides users with essential functions like greeting messages and order status checks, all while ensuring smooth and balanced operation through efficient configuration and connectivity management. This setup ensures that users have a seamless experience when interacting with the Swiggy application.

### 3.6. Service Registry Service

This is a small program written in the Java programming language. Its main goal is to set up a "Service Registry". This program helps in organizing and keeping track of different services that run in a network. Think of it like an attendance list that keeps track of which services are available and ready to be used. It's useful in complex systems where many different services need to work together seamlessly.

The main feature of Service Registry is to act as a registry or a directory. It allows services (like small parts of a larger application) to register themselves and discover other services. Eureka Server is a special type of server that is good at keeping track of all these services. The program uses a tool called "Eureka" to do this job. When you run this program, it sets up a Service Registry using Eureka. Main Function is a function (you can think of it as the starting point) that runs the whole thing. The

program uses a framework called "Spring Boot" to get everything up and running quickly without needing to worry about a lot of complicated configurations.

In large systems, different parts (or services) need to talk to each other. This program helps in making sure that these parts can find each other easily and work together smoothly. It makes managing complex systems a lot easier by keeping an organized list of all available services.

# 4. HOW IS THE SYSTEM TESTED

In order to test and compare different hashing algorithms, the same system with different specifications were developed. In the first round, the auth service was developed with SHA2-512 as the hashing algorithm for token signing. After carrying out all the tests and trying all the test cases, the auth service was modified so that we could use SHA3-512 in order to sign the token generation and verification. In both scenarios, the database was designed in a way that it will drop all the tables once it is shut down and re-create them again at the next startup.

The system was tested against 3 different user roles (Admin, Restaurant, User) to check the authentication and authorization levels. For each user role, we have tried to reach each micro service and based on each role and its privileges, some of them were not successful. The flow of these tests and the lifecycle of the code are shown in figures 4.1, 4.2, and 4.3.

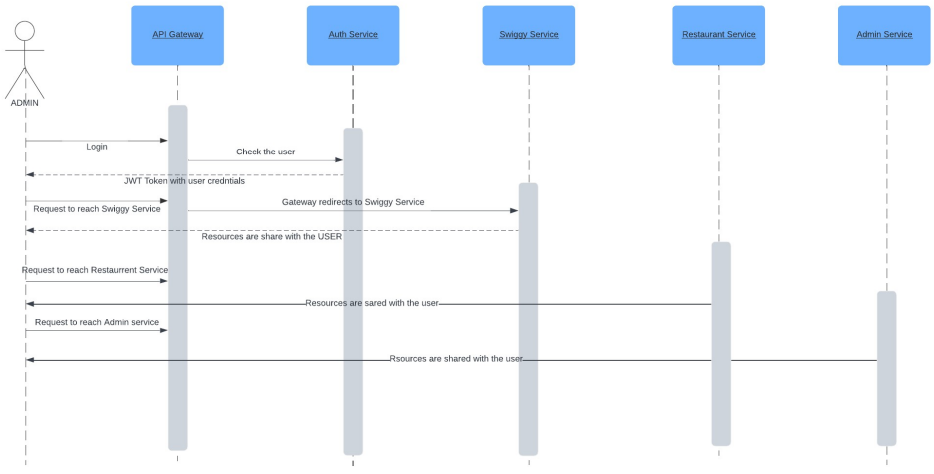


Figure 4.1 The sequence of events with admin role privilege

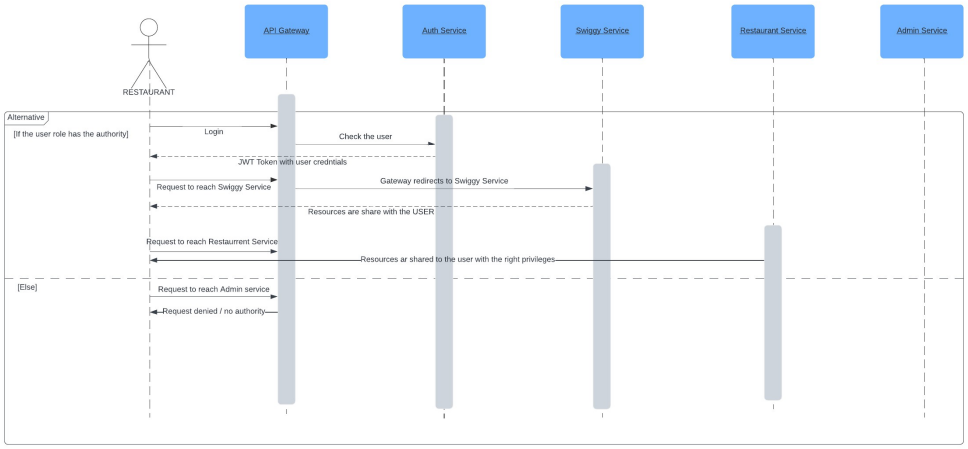


Figure 4.2 The sequence of events with restaurant role privilege

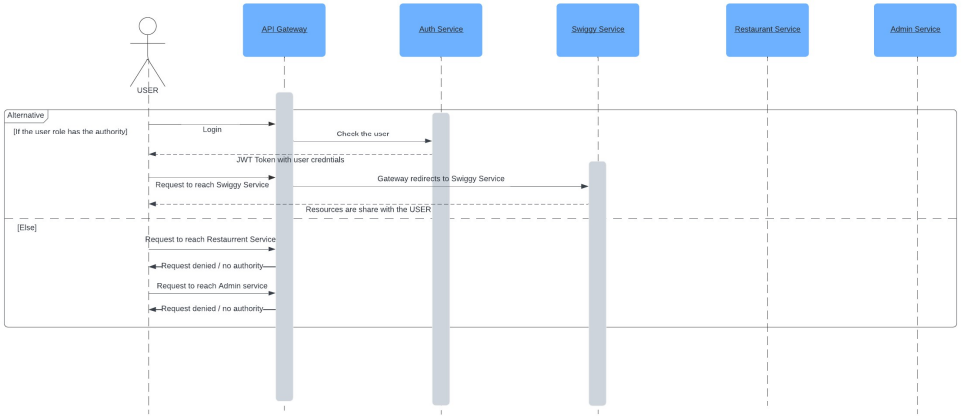


Figure 4.3 The sequence of events with user role privilege

The system first was taken to level tests to check if the authentication and authorization mechanisms work properly. In order to check this mechanism, one user for each role was created and each of them was tested to see if they can consume the endpoints they have authorities for and if they cannot reach the ones out of their privileges. The role with minimum privileges in the system is USER role and the Figures 4.4, 4.5 and 4.6 illustrate the attempts to reach separate services. Figure 4.4 shows the USER attempting to consume Swiggy service resources and successfully fulfilling it. Figure 4.5 shows an attempt by USER role to reach Restaurant service resources and failing to do so. Likewise, Figure 4.6 illustrates the USER role attempting to reach Admin service resources and failing to do so because of not enough authority.

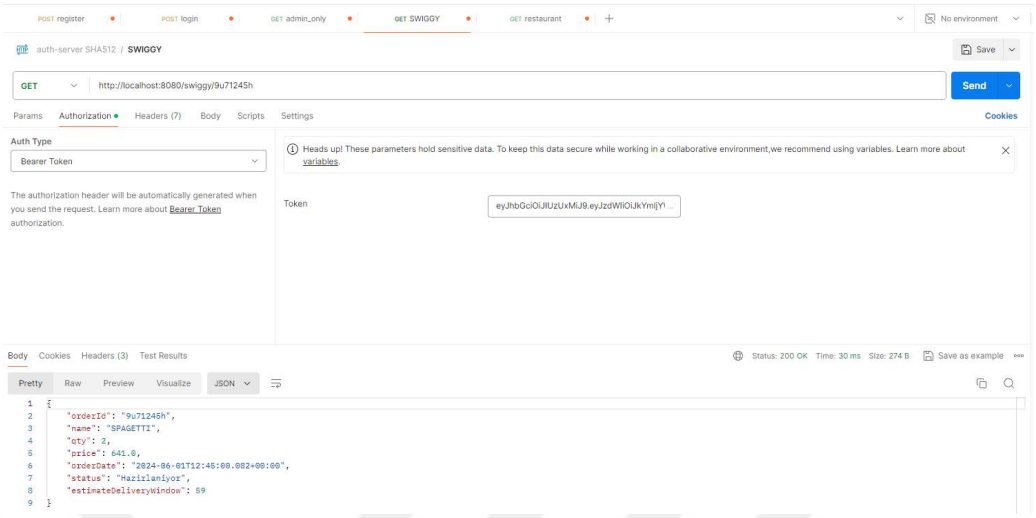


Figure 4.4 USER role's successful attempt to reach Swiggy Service

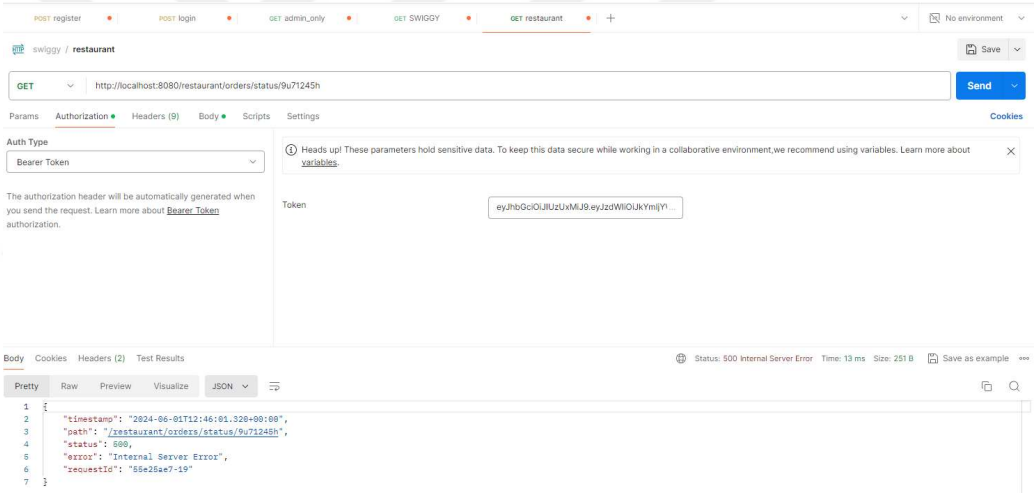


Figure 4.5 USER role's unsuccessful attempt to reach Restaurant Service

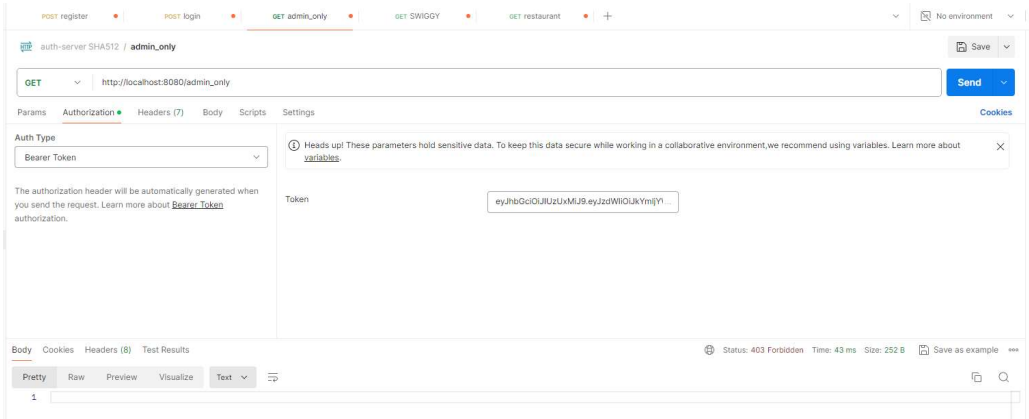


Figure 4.6 USER role's unsuccessful attempt to reach Admin Service

After authentication and authorization level tests, the system was taken to load testing and penetration testing for both SHA2 and SHA3 configurations in both Auth Service and API gateway configured accordingly. The system was tested in a simulation for 50 and 1000 users for both of the configurations. The load tests were delivered by using open source Apache JMeter application. At the end of the tests, there were some interesting results.

The configuration of the system for SHA2 hashing yielded the response time results illustrated in Table 4.1 and Table 4.2 for 50 and 1000 users, respectively. The system response times per milliseconds (ms) for 50 users were minimum 17.56 ms and maximum 1778.6 ms. The average response time of the system was 611.02 ms. The response time per milliseconds for 1000 users were minimum 18.34 ms and maximum 27175.95 ms. The average response time was 9086.32 ms.

Table 4.1 The response time of SHA2 configuration with 50 users

| **Request Label** | **Response Time (ms)** | | | |
|---|---|---|---|---|
| | **Average** | **Min** | **Max** | **Median** |
| SHA2 | 611.02 | 5 | S2970 | 35.50 |
| SHA2-5 | 1778.46 | 740 | 2970 | 1724.50 |
| SHA2-7 | 37.04 | 8 | 124 | 18.50 |
| SHA2-8 | 17.56 | 5 | 110 | 9.50 |

Table 4.2 The response time of SHA2 configuration with 1000 users

| **Request Label** | **Response Time (ms)** | | | |
|---|---|---|---|---|
| | **Average** | **Min** | **Max** | **Median** |
| SHA2 | 9086.32 | 4 | 51129 | 24.00 |
| SHA2-5 | 27175.95 | 2262 | 51129 | 28179.00 |
| SHA2-7 | 64.67 | 5 | 1721 | 14.00 |
| SHA2-8 | 18.34 | 4 | 810 | 6.00 |

Figure 4.5 shows the success rate of the SHA2 configuration with 50 users. During the test with 50 users, the system did not fail at all and achieved 100% success. However, the failure rate with 1000 users was 2.17% as illustrated in Figure 4.6.

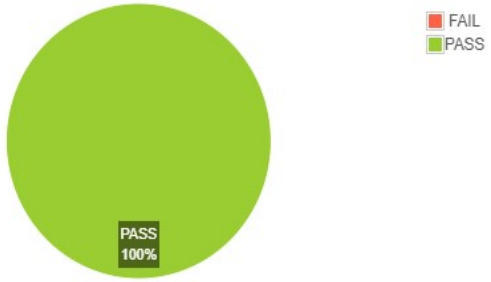


Figure 4.7 The execution error rate of the SHA2 configuration with 50 users

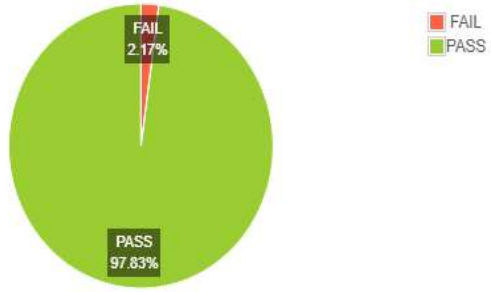


Figure 4.8 The execution error rate of the SHA2 configuration with 1000 users

As seen in Table 4.3, the execution column gives us the number of executions for each sampler with SHA2 configuration. Since we had 50 threads and 1 iteration, it means each virtual user executes all samplers 50 times. During these executions, the network experienced different number of transactions, the lowest to be 14.90 transactions per second and the highest to me 19.70 transactions per second. When the number of executions hits 150, these transactions increased to 41.70 per second. The size of data transferred changed throughout the test. The received KB/Sec was minimum 4.40 and maximum 11.21. When the execution number reaches 150, this number reached to 18.19. The amount of sent data through the network with 50 executions was minimum 8.86 KB/Sec and maximum 9.24, and 17.31 when the execution hits 150.

Table 4.4 illustrates the network traffic and data sent with 1000 users in the system with SHA2 configuration. When the number of threads was set to 1000, the network experienced different number of transactions, the lowest to be 18.39 transactions per second and the highest to me 19.84 transactions per second. When the number of executions hits 3000, these transactions increased to 54.91 per second. The size of data transferred changed throughout the test. The received KB/Sec was minimum 5.48 and maximum 13.23. When the execution number reaches 3000, this number reached to

54.91. The amount of sent data through the network with 1000 executions was minimum 5.69 KB/Sec and maximum 9.46, and 22.79 when the execution hits 3000.

Table 4.3 The network results of SHA2 configuration with 50 users

| Request Label | Execution | Throughput | Network (KB/sec) | |
|---|---|---|---|---|
| | Sample | Transactions/s | Received | Sent |
| SHA2 | 150 | 41.70 | 18.19 | 17.31 |
| SHA2-5 | 50 | 14.90 | 11.21 | 4.61 |
| SHA2-7 | 50 | 19.30 | 5.37 | 8.86 |
| SHA2-8 | 50 | 19.39 | 5.40 | 9.24 |

Table 4.4 The network results of SHA2 configuration with 1000 users

| Request Label | Execution | Throughput | Network (KB/sec) | |
|---|---|---|---|---|
| | Sample | Transactions/s | Received | Sent |
| SHA2 | 3000 | 54.91 | 23.37 | 22.79 |
| SHA2-5 | 1000 | 18.39 | 13.23 | 5.69 |
| SHA2-7 | 1000 | 19.68 | 5.48 | 9.03 |
| SHA2-8 | 1000 | 19.84 | 5.54 | 9.46 |

The configuration of the system for SHA2 hashing yielded the response time results illustrated in Table 4.5 and Table 4.6 for 50 and 1000 users, respectively. The response time of the system per millisecond for 50 users were minimum 15.28 ms and maximum 2160.58 ms. The average response time was 740.93 ms. The response time per milliseconds for 1000 users were minimum 16.84 ms and maximum 27069.53 ms. The average response time was 9042.56 ms.

Table 4.5 The response time of SHA3 configuration with 50 users

| Request Label | Response Time (ms) | | | |
|---|---|---|---|---|
| | Average | Min | Max | Median |
| SHA3 | 740.93 | 7 | 3379 | 27.50 |
| SHA3-1 | 2160.58 | 1017 | 3379 | 2190.00 |
| SHA3-3 | 46.92 | 9 | 378 | 23.00 |
| SHA3-4 | 15.28 | 7 | 33 | 12.50 |

Table 4.6 The response time of SHA3 configuration with 1000 users

| Request Label | Response Time (ms) | | | |
|---|---|---|---|---|
| | Average | Min | Max | Median |
| SHA3 | 9042.56 | 4 | 49292 | 25.00 |
| SHA3-1 | 27069.53 | 3910 | 49292 | 28243.00 |
| SHA3-3 | 41.32 | 5 | 465 | 13.00 |
| SHA3-4 | 16.84 | 4 | 524 | 7.00 |

The test results show that the increase in the number of users using the system causes a negligible increase in the error rate of the system. Figure 4.7 shows the success rate of the SHA3 configuration with 50 users. During the test with 50 users, system failure rate was 5.33%, and the failure rate with 1000 users was 5.93% as illustrated in Figure 4.8.

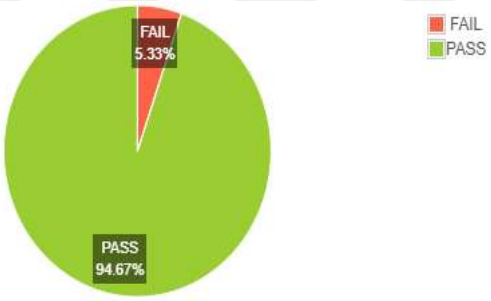


Figure 4.9 The execution error rate of the SHA3 configuration with 50 users


Figure 4.10 The execution error rate of the SHA3 configuration with 1000 users

As seen in Table 4.7, the execution column gives us the number of executions for each sampler SHA3 configuration. Since we had 50 threads and 1 iteration, it means each virtual user executes all samplers 50 times. During these executions, the network experienced different number of transactions, the lowest to be 13.03 transactions per

second and the highest to me 17.73 transactions per second. When the number of executions hits 150, these transactions increased to 36.91 per second. The size of data transferred changed throughout the test. The received KB/Sec was minimum 4.81 and maximum 9.80. When the execution number reaches 150, this number reached to 16.01. The amount of sent data through the network with 50 executions was minimum 4.04 KB/Sec and maximum 8.45, and 15.32 when the execution hits 150.

Table 4.8 illustrates the network traffic and data sent with 1000 users in the system with SHA3 configuration. When the number of threads was set to 1000, the network experienced different number of transactions, the lowest to be 18.83 transactions per second and the highest to me 20.43 transactions per second. When the number of executions hits 3000, these transactions increased to 56.19 per second. The size of data transferred changed throughout the test. The received KB/Sec was minimum 5.62 and maximum 13.50. When the execution number reaches 3000, this number reached to 56.19. The amount of sent data through the network with 1000 executions was minimum 5.83 KB/Sec and maximum 9.74, and 23.32 when the execution hits 3000.

Table 4.7 The network results of SHA3 configuration with 50 users

| Request Label | Execution | Throughput | Network (KB/sec) | |
|---|---|---|---|---|
| | Sample | Transactions/s | Received | Sent |
| SHA3 | 150 | 36.91 | 16.01 | 15.32 |
| SHA3-1 | 50 | 13.03 | 9.80 | 4.04 |
| SHA3-3 | 50 | 17.67 | 4.81 | 8.11 |
| SHA3-4 | 50 | 17.73 | 4.91 | 8.45 |

Table 4.8 The network results of SHA3 configuration with 1000 users

| Request Label | Execution | Throughput | Network (KB/sec) | |
|---|---|---|---|---|
| | Sample | Transactions/s | Received | Sent |
| SHA3 | 3000 | 56.19 | 23.79 | 23.32 |
| SHA3-1 | 1000 | 18.83 | 13.50 | 5.83 |
| SHA3-3 | 1000 | 20.43 | 5.62 | 9.38 |
| SHA3-4 | 1000 | 20.43 | 5.68 | 9.74 |

After load testing, both configurations were taken to penetration testing. The tests were delivered by using an open source application named ZAP Proxy which was developed by OWASP (Open source web application security project). During the penetration testing, the application was set to simulate SQL injection attack, none algorithm attack, algorithm confusion attack, trusting JWK provided with the Token, empty JWT, null byte injection attack and weak JWT secrets attacks. For both configurations, the ZAP Proxy first goes through a process called Spidering during which it finds all the endpoints in the system. Then, the automatic scanning is started to find vulnerabilities. The application simulates more than 1500 automatic attempts for possible vulnerabilities for both configurations.

For SHA2 configuration, there were 3 different versions of alerts in the report. There was only one high level vulnerability found which was SQL Injection attack vulnerability (Table 4.9). This will mean, the attacker can inject information into the database. The medium level alert was about Spring Actuator information leak warning (Table 4.9). This is about Spring Framework and the dependency it provides named Actuator. Then, there were 96 informational warnings which were not harmful for the system but recommendations for improvements.

Table 4.9 The number of alerts for each level and confidence of the SHA-2 configuration

| | | **Confidence** | | |
|---|---|---|---|---|
| | | **High** | **Medium** | **Low** |
| **Risk** | **High** | 0.0% | 33.3% | 0.0% |
| | **Medium** | 0.0% | 33.3% | 0.0% |
| | **Low** | 0.0% | 0.0% | 0.0% |

For SHA3 configuration, there were basically 1565 attempts by the application and there were only 2 levels of warnings. These warnings are medium level (Table 4.10) and informational warnings, the same as the SHA2 configuration. The only difference is that SHA3 configuration did not have any vulnerabilities against SQL Injection attacks. Since the system configurations for both versions are the same except the Auth-Service and API gateway, this is an expected result. The only difference, which

is an important one, is the SQL Injection vulnerability which happens in Auth-Service in SHA2 configuration.

Table 4.10 The number of alerts for each level and confidence of the SHA-3 configuration

| | | Confidence | | |
|---|---|---|---|---|
| | | **High** | **Medium** | **Low** |
| **Risk** | **High** | 0.0% | 0.0% | 0.0% |
| | **Medium** | 0.0% | 50.0% | 0.0% |
| | **Low** | 0.0% | 0.0% | 0.0% |

## 5. CONCLUSION

During the study and the testing phase, we haven't found meaningful differences at latency and throughput between SHA2-512 and SHA3-512, however, according to the test results by NIST, SHA3-512 is more resilient against specific types of attacks such as length extension attacks against which SHA2 has been reported to have been compromised. Even though SHA-3 is not a newly-developed algorithm and it was developed as a backup of SHA-2 algorithm, it has not been used as JWT signing algorithm. It definitely provides benefits over the former ones and has been accepted as a very strong algorithm for the time being.

Although there are variants of SHA-3 as other SHA algorithms, we have chosen SHA3-512 among them because it provides the longest hash result. As a result, it provides the strongest security compared to SHA3-256 or SHA3-348.

According to the findings at the end of our load test via Apache Jmeter, it came out that depending on the server configurations and system resources, response times of both SHA2 configuration and SHA3 configurations show similar results with both 50 users and 1000 users at the same time even though it was expected for SHA3 to have higher response times because of its higher security margin and the permutation functions. The most remarkable difference in both systems was the failure rates based on the number of users in the system. While SHA2 configuration had no failures with 50 users, it had 2.17% failure rate with 1000 users. On the other hand, SHA3 configuration had 5.33% failure rate with 50 users and 5.93% with 1000 users. The difference in SHA2 was 2.17 % while it was 0.6 in SHA3 configuration. We can conclude that failure rates increase more drastically in SHA2 configuration than in SHA3 configuration. This means SHA3 failure rates do not increase as much as SHA2

with the increasing user load on the system. The penetration test results also suggest that SHA3 is a more resilient configuration because SHA2 tests resulted in an SQL Injection vulnerability while there was no high risk vulnerability found in SHA3 configuration.

We recommend that academia and the business world to start considering upgrading the security systems from SHA2 to SHA3 since the computational power of computers has increased and older hashing algorithms have already been compromised.

## 6. DISCUSSION

The main discussion and recommendation for further studies would be to work on creating a brand-new library to support SHA3 hashing algorithm in token generation and token signing. Since the available libraries do not directly support this hashing algorithm, developers and security experts will have to implement it indirectly with external third-party libraries such as MD, Jose Nimbus or Bouncy Castle. Instead, there should be either native support in JWT mechanism for SHA-3 or a new token generation model could be developed around this hashing algorithm.

If the signature algorithm library is to be extended to make the JWT system more secure, SHA3 should be the first one to be added in the natively supported algorithms list, so that developers will be able to use this in an easier way and create more resilient and secure systems without needing to implement it indirectly to their systems. This way, even though computational power of devices increases, the systems will continue to be safe for a long time.

If latter option is to be adopted which is creating a new model for token generation, it will require creating a whole new system and it would be very costly in terms of time, energy and money. Instead, improving the already existing model to the highest possible security level would be a smarter choice.

## 7. REFERENCES

[1] M. G. d. Almeida and E. D. Canedo “Authentication and Authorization in Microservices Architecture: A Systematic Literature Review” available at https://doi.org/10.3390/app12063023

[2] N. Sčnger and S. Abeck “Authentication and Authorization in Microservice-Based Applications” in INFORMATIK 2022, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2022 207

[3] A. Banati, E. Kail and M. Kozlovszky “Authentication and authorization orchestrator for microservice-based software architectures” in 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)

[4] B. Martin, A. Quimatio, N. T. Auvarique, T. Fidele and J. N. Marcellin “Continuous Single-Sign-On (CSSO) method for authentication and authorization in microservices architectures” available at https://doi.org/10.21203/rs.3.rs-1579462/v1

[5] Joel Suomalainen “DEFENSE-IN-DEPTH METHODS IN MICROSERVICES ACCESS CONTROL” Master’s Thesis at Tampere University, Feb 2019

[6] Z. Triartono, R. M. Negara and Sussi “Implementation of Role-Based Access Control on OAuth 2.0 as Authentication and Authorization System” at Proc. EECSI 2019 – Bandung, Indonesia

[7] Y. ShuLin and H. JiePing “Research on Unified Authentication and Authorization in Microservice Architecture” at 2020 IEEE 20th International Conference on Communication Technology

[8] X. Rongxu, J. Wenquan and K. Dohyeun “Microservice Security Agent Based on API Gateway in Edge Computing” at Sensors 2019, 19, 4905; doi:10.3390/s19224905

[9] K. A. Torkura, M. I. H. Sukmana, A. V.D.M. Kayem, F. Cheng and C. Meinel, “A Cyber Risk-Based Moving Target Defense Mechanism for Microservice Architecture” in 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications

[10] N. Chondamrongkul, J. Sun and I. Waren “Automated Security Analysis for Microservice Architecture” in 2020 IEEE International Conference on Software Architecture Companion (ICSA-C)

[11] R. Bhattacharya and T. Wood “BLOC: Balancing Load with Overload Control In the Microservices Architecture” in 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)

[12] C. C. Hernandez-Aparicio, J. O. Ocharan-Hernandez, K. Cortes-Verdin and M. A. Arenas-Valdes “Architectural Languages for the Microservices Architecture: A systematic mapping Study” in 2022 10th International Conference in Software Engineering Research and Innovation

[13] K. Jander, L. Braubach and A. Pokahr “Defense-in-depth and Role Authentication for Microservice Systems” in the 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018)

[14] A. Barabanov and D. Makrushin “Authentication and Authorization in Microservice-Based Systems: Survey of Architecture Patterns” DOI: 10.21681/2311-3456-2020-04-32-43 January 2020 available at https://www.researchgate.net/publication/346523340_Authentication_and_Authorization_in_Microservice-Based_Systems_Survey_of_Architecture_Patterns

[15] O. Tereshchenko and N. Trintina “Development Principles of Secure Microservices” CPITS-II-2021: Cybersecurity Providing in Information and Telecommunication Systems, October 26, 2021 available at CEUR-WS. Org/Vol-3188/paper2.pdf

[16] A. Venckauskas, D. Kukta, S. Grigaliunas and R. Bruzgiene “Enhancing Microservices Security with Token-Based Access Control Method” at Sensors 2023, 23, 3363. https://doi.org/10.3390/s23063363

[17] L.D.S.B Weerasinghe and I Perera “Evaluating the Inter-Service Communication on Microservice Architecture” in 2022 7th International Conference on Information Technology Research (ICITR) | 979-8-3503-3203-2/22/$31.00 ©2022 IEEE
DOI: 10.1109/ICITR57877.2022.9992918

[18] S. T. FLORÉN “Implementation and Analysis of Authentication and Authorization Methods in a Microservice Architecture” Degree Project in Computer Science and Engineering at KTH Institute of Technology June 2021

[19] Srijith, K. Bantia, N. Govardhan “Inter-Service Communication among Microservices using Kafka Connect” at 2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS) | 978-1-6654-1032-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSESS54813.2022.9930270

[20] X. He and X. Yang “Authentication and Authorization of End User in Microservice Architecture” in IOP Conf. Series: Journal of Physics: Conf. Series 910 (2017) 012060

[21] Y. Fei, Z. Shengjie and D. Hao “Microservice Security Framework for IoT by Mimic Defense Mechanism” at Sensors 2022, 22, 2418. https://doi.org/10.3390/s22062418

[22] T. Yarygina and A. H. Bagge “Overcoming Security Challenges in Microservice Architectures” at 2018 IEEE Symposium on Service-Oriented System Engineering

[23] K. Jander, L. Braubach and A. Pokahr “Practical Defense-in-depth Solution for Microservice Systems” at Journal of Ubiquitous Systems & Pervasive Networks Volume 11, No. 1 (2019) pp. 17-25

[24] N. Mateus-Coelho, M. Cruz-Cunha and L. G. Ferreira “Security in Microservices Architectures” at CENTERIS - International Conference on Enterprise Information Systems / ProjMAN - International Conference on Project Management / HCist - International Conference on Health and Social Care Information Systems and Technologies 2020

[25] R. Chandramouli “Security Strategies for Microservices-based Application Systems” at NIST Special Publication 800-204

[26] R. Alboqmi, S. Jahan and R. F. Gamble “Toward Enabling Self-Protection in the Service Mesh of the Microservice Architecture” at 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)

[27] X. Li, Y. Chen and Z. Lin “Towards Automated Inter-Service Authorization for Microservice Applications” available at DOI: 10.1145/3342280.3342288 Conference Paper: August 2019

[28] M. Gordesli and A. Varol “Comparing Interservice Communications of Microservices for E-Commerce Industry” at 2022 10th International Symposium on Digital Forensics and Security (ISDFS) | 978-1-6654-9796-1/22/ ©2022 IEEE | DOI: 10.1109/ISDFS55398.2022.9800784

[29] R. A. Al-Wadi and A. A. Maaita “Authentication and Role-Based Authorization in Microservice Architecture: A Generic Performance-Centric Design” at Article *in* Journal of Advances in Information Technology · January 2023 DOI: 10.12720/jait.14.4.758-768

[30] K. Lin, B. Liu, Y. Chen, H. Dai and Z. Liang “Intelligent Edge Computing Gateway Implementation Method based on Microservice Architecture” at 2022 International Conference on Informatics, Networking and Computing (ICINC)