**TERMS AND CONDITONS**: **1. DO NOT SHARE THE PROJECT TO ANYONE.**

**2. DO NOT SELL THE PROJECT IF YOU SELL WE HAVE TO TAKE ACTIONS.**

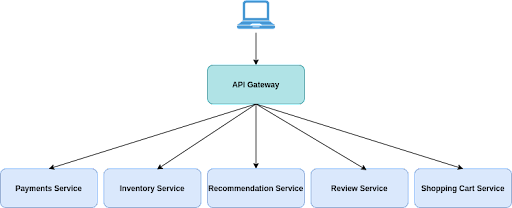
**3. DO NOT UPLOAD ON ANY GIT .**

**KINDLY REQUEST YOU TO FOLLOW THE RULES**

**Working Of Claims Management Java Project**

**Api Gateway Microservice**

The API Gateway serves as a reverse proxy to accept API calls from the client application, forwarding this traffic to the appropriate service. The API gateway also enforces security and ensures scalability and high availability.



API Gateway architecture this article will look at why API Gateways are critical to a microservice architecture, and we’ll cover the common features of an API gateway. Along the way, we’ll learn about the challenges that an API gateway can overcome and some of the caveats to bear in mind when using an API gateway.

Why Do We Need API Gateways?

The various advantages of a microservice architecture subsequently present unique challenges that API gateways are purpose-built to address.

Centralized access to decentralized microservices

A microservice architecture modularizes the many functions of an application so that each service can focus on implementing a specific business rule. This design pattern makes it easy to develop, test, deploy, and maintain different capabilities of an application. However, this approach also means increased complexity for clients to access those services.

However, an API gateway can handle several API calls simultaneously and route them to different backend services. It can also decompose a single client call into multiple requests to other microservices and aggregate the results when they respond.

Management and discovery for scalable, distributed services

The elastic nature of the cloud allows for the horizontal scaling of services as demand increases. However, achieving this requires efficient load balancing, easier ways to discover services, and resiliency features like retries and timeouts.

API Gateways can load balancing between replicas of the same service, which allows better resource utilization. Traditional proxies employ basic load balancing algorithms like Random or Round Robin. At the same time, API Gateways can use more sophisticated algorithms like weighted connections, least connections, or even custom implementations that leverage the service registry. These techniques can provide efficient traffic routing. It can do all this while implementing features like retries when a service is down, rerouting requests to healthy service instances, or graceful error handling.

**User Microservice and Authorizartion**

Designing APIs for Claims or Claims for APIs

In a RESTful microservice architecture each API is independent and each request should contain all data needed to perform the request. Ideally the API rarely should have to contact other services to retrieve the information it needs to authorize the request.

By carefully designing the Access Token it's possible to achieve a high level of containment for the services.

Authorization

Much of what the API will need to figure out is: Is the Client and the User allowed to perform this request.

Broadly speaking this involves figuring out the following:

* Is the client authorized to call the API at all
* Is the user authenticated
* Does the user have enough permissions
* Does the data the request operates on belong to the user
* If not, is the user allowed anyway

Many times system designers wants a few levels of access. User access, Admin Access and possible Customer Support access. With a claims based architecture using OAuth it's possible to achieve these levels system wide without making custom solutions in each API.

Authorizing the Client

As discussed in the [Scope article](https://curity.io/resources/learn/scopes-and-how-they-relate-to-claims/), the Scope can be used for coarse grained access control. From the APIs perspective the API should validate that the Access Token contains the desired scopes. The OAuth server will not issue scopes to a client that is not entitled to get them. So the API can simply rely on that fact.

Microservice Example with Scopes

Consider the following service that allows a client to send SMSs. It consists of three APIs. An invoice API where internal clients can create invoices, and external clients can read them, an SMS API where the external Clients can send SMSs and a Status API where it's possible to see the system status for the service.

Lets assume we create the scopes: invoice-read, invoice-write, sms and status. A client needs to be authorized with these scopes to be able to contact the API. So it's easy for the API to determine if it should let the request through the door.

However to actually perform anything it needs to know more. What account-nbr does the user have, where the billing should occur. When sending SMS it needs to know what account-type is it: premium or standard, to know what rate limits apply, and also the account-nbr to know where to increase the bill.

***Eureka Microservice***

Eureka Server is **an application that holds the information about all client-service applications**. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

**Connecting Microservices to Eureka naming server**

1. Step 1: Select the currency-conversion-service project.
2. Step 2: Open the pom. ...
3. Step 3: Open CurrencyConversionServiceApplication. ...
4. CurrencyConversionServiceApplication.java.
5. Step 4: Open the application. ...
6. application.properties.

**Policy Microservice**

**5 core components of microservices architecture**

* Microservices. Microservices make up the foundation of a microservices architecture. ...
* Containers. ...
* Service mesh. ...
* Service discovery. ...
* API gateway.

**What is policy administration system?**

A policy administration system is **a software solution that executes insurance functions including rating, quoting, binding, issuing, endorsements, and renewals**. Historically, these systems were inflexible, non-customizable, expensive, and an all-or-nothing type solution



**Member Module Microservice**

Microservice architecture or microservices, is a method of designing software in which applications are divided into several smaller, individual and independent services or modules. Each service is flexible, robust, composable and complete, that is necessary to run a specific business function. Each microservice can be implemented in a different programming language.  
  
Microservices encourages loosely coupled and highly cohesive architecture. Coupling refers to the degree of interdependence between software modules and cohesion indicates how well the individual elements within an application work together. This has gained popularity as it helps Enterprise applications easier to develop, build, deploy, test and scale.