Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 7 Issue 01, January-2018

Microservices API Security

Jyothi Salibindla
Karsun Solutions LLC
Herndon, VA USA

***Abstract*—Microservices have become the hot term since 2014. This particular programming methodology involves an architectural approach that emphasizes the decomposition of applications into single purpose, with implementation via the Rest API. A new software development style that has grown from recent trends in software development/management practices, Microservices are composable, fine-grained modules which can be independently deployable and scaled both vertically and horizontally. This empowers the development, manageability and speed to the changing market by adapting to Agile methods, DevOps culture, cloud, Linux containers, and Continuous Integration/ Continuous Development methods. This paper will focus on the necessity for this methodology as well as it's corresponding API security options needed for securing all of the back-end Microservices**

Keywords— Microservices, API, Cross Cutting Concerns, OOB; AOP; SoC;Token;Json Web Token

I. INTRODUCTION

Problems in software manifest themselves in several ways including budget overflows and project delays. Previous methodologies in software development have been very inefficient, inferior in quality (bugs), and often do not meet the requirements of its client. Projects were unmanageable and the programming code has been difficult to maintain due to various reasons. However, this situation has been changed by the improvements in computing power, so much so that it has outpaced the ability of programmers to effectively utilize those capabilities. Various processes and methodologies have been developed over the past few decades to improve software quality management such as procedural programming and object-oriented programming. Despite the improvement of development methodologies, software projects that are large, complicated, poorly specified, and involve unfamiliar aspects, are still vulnerable to large, unanticipated problems.

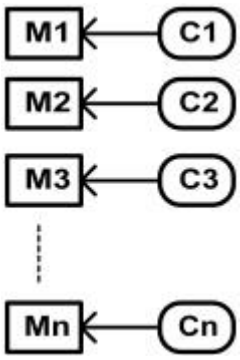
II. HISTORY

Since the 1960s, several advances in programming theory have occurred. The discipline of programming has progressed through several phases, with each new advance being touted as the next "better" way of programming. From structured programming, functional decomposition, object-oriented programming, aspect programming. As per the Barbara Liskov, "The connections between modules are the assumptions which the modules make about each other." Modern software systems are complex, and therefore are vital to implement a new strategy with modular approach, built on the idea of Separation of Concerns (SoC). SoC is traditionally achieved through modularity and encapsulation, with the help of information hiding. Layered designs in information systems are based on Separation of Concerns (e.g., presentation layer, business logic layer, data access layer, database layer).

In order to manage complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability, the problem (software system) should be decomposed into modules such that each module has one concern. A concern is an identifiable requirement which would evolve/change independently. Separation of Concerns is a guiding principal in software development centered on the idea that programs should have distinct sections, with each section being responsible for its own concern.

III. IMPLEMENTATION

A. *Solution Space*



In Object-oriented methods, the separated concerns are modeled as objects and classes, which are generally derived from the entities in the requirement specification and use cases. In structural methods, concerns are represented as procedures. Is it possible to get 1:1 mapping between problems (concerns) and solutions (code modules)? Unfortunately, the answer is "No". The key unit of modularity in OOPs is the class. Most of the times, one class is not enough to provide the Module functionality. Even though Object Oriented Programming (OOP) is easy to understand and provides decent decoupling between the objects, it is not enough to address the common concerns that are applicable throughout the application as it affects the entire solution. Implemented Patterns will be scattered/ tangled throughout system in the OOP.

B. *Cross-cutting Concerns*

Depending upon the implementation language and the corresponding abstractions provided by the language, some concerns cannot be easily separated, and thus, are forced to map such concerns over many modules. Certain instances in which this situation can be applied to are security, thread safety, transaction management, logging and tracing, profiling, caching, pooling, persistence, etc.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 7 Issue 01, January-2018

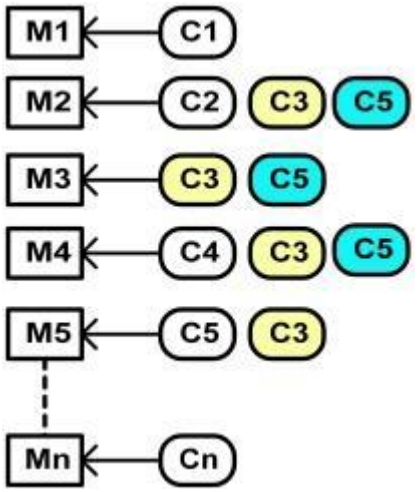


Fig. 1. Module – Concern Mapping

Cross-cutting concerns are modularized using a new abstraction called Aspect. It allows the developers to concentrate on the functional capabilities rather than focusing on individual cross-cutting concerns.

C. *Aspect Orientation*

Aspect Orientation is way to modularize cross-cutting concerns. This model helps with identifying the system modules that could be factored out into separate modules (Fig. 2: A3, A5). These separated components/modules would then be included in the system so that they could be used by various other modules.

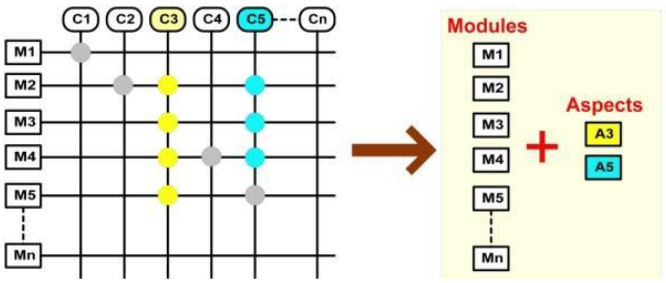


Fig. 2. Aspect Oriented Model

D. *Composition*

In order to divide the problem into composable units that perform one identifiable task, it is pivotal to keep all of the operations in a unit at the same level of abstraction. This will naturally result in systems with many small units, which in turn are far easier to test. Names become documentation, and this process will make previously-hidden reusable assets discoverable.

E. *Single Responsibility Principle*

This principle states that every module should have a single responsibility, with which all its services should be narrowly aligned with that responsibility.

IV. MICROSERVICE ARCHITECTURE

Microservices are a variant of the service-oriented architecture (SOA) style which structures an application as a collection of loosely coupled services. In Microservices architecture, services should be fine-grained and the protocols should be lightweight. The modular architectural style has been shown to be particularly well suited to cloud-based environments and its popularity has been exponentially rising.

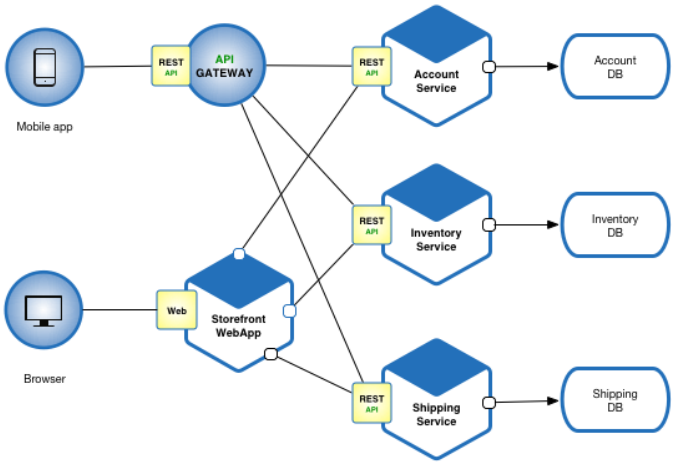


Fig. 3. Example e-commerce application

Benefits of the Microservices Architecture:

- Each Microservice is relatively small
- Easier for a developer to understand
- The IDE is faster making developers more productive
- The application starts faster, which makes developers more productive, and speeds up deployments
- Each service can be deployed independently of other services which makes it easier to deploy new versions of services frequently
- Easier to scale development. It enables the ability to organize the development effort around multiple teams
- Improved fault isolation. For example, if there is a memory leak in one service then only that particular service will be affected.
- Eliminates any long-term commitment to a technology stack

Drawbacks of the Microservices Architecture:

- Developers must deal with the additional complexity of creating a distributed system.
- Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
- Testing across services can be more difficult, compared to monolithic applications
- Developers must implement the inter-service communication mechanism.
- Implementing use cases which span multiple services without using distributed transactions can be arduous
- Implementing use cases which span multiple services also requires careful coordination between the teams

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 7 Issue 01, January-2018

- Deployment complexity
- Increased resource consumption

V. SCALING BY FUNCTIONAL DECOMPOSITION

One of the major driving forces behind any kind of architectural solution is scalability.

The horizontal application scaling (which has been possible even with monolithic architecture), and the Z axis represents the scaling of the application by splitting similar things. The Z axis idea can be better understood by using the sharding concept, where data is partitioned and the application redirects requests to corresponding shards based on user input.

The Y axis, however, is crucial in understanding the scope of decomposition. This axis represents functional decomposition. In this kind of strategy, various functions can be seen as independent services. Instead of deploying the entire application only once the entire development is done, developers can deploy their respective services independently without waiting for the other development teams to finish their modules. This not only improves developer time management, but also offers them much more flexibility to change and redeploy their modules without needing to worry about the rest of the application's components. There are a couple of different ways of decomposing the application into services. One approach is to use verb-based decomposition and define services that implement a single use case such as "checkout". The other option is to decompose the application by noun and create services responsible for all operations related to a particular entity such as "customer management". An application might also use a combination of verb-based and noun-based decomposition.

Using Z-axis scaling each server runs an identical copy of the code. In this respect, it's similar to X-axis scaling. The big difference however, is that with the Z-axis circumstance, each server is responsible for only a subset of the data. Z-axis splits are commonly used to scale databases. Data is partitioned (a.k.a. sharded) across a set of servers based on an attribute of each record.

VI. API SECURITY

One of the challenges to building any RESTful API is having a well thought out authentication and authorization strategy. Cross-cutting concerns like authentication, security, and logging are always challenging and involve many stakeholders.

A. Basic Authentication

A client can authenticate to the API Gateway with a username and password combination using HTTP Basic Authentication. Basic Authentication is not considered to be a secure method of user authentication (unless used in conjunction with some external secure system such as SSL), as the user name and password are passed over the network as cleartext. With HTTP Basic Authentication, the client's username and password are concatenated, base64-encoded, and passed in the Authorization HTTP header as follows:

Authorization: ***Basic*** *dm9yZGVsOnZvcmRlbA==*

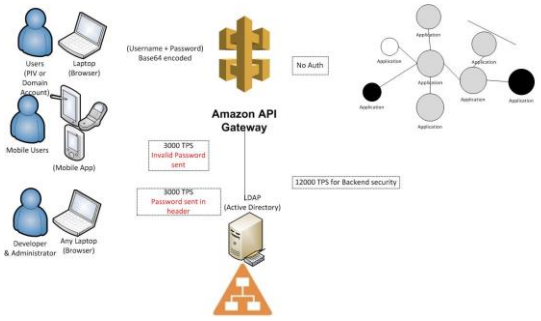


Fig. 4. Basic Authentication

A majority of the Application Architectures will not secure the backend services. As a common approach, the client will be authenticated one time and allow the access to the HTTP resources which will then internally call other resources and return the results to the client. A major concern of this process is that there is no authentication and authorization implemented on the client calls on the backend.

B. IP Whitelisting

Access to the APIs (Microservices) can be restricted by setting up a whitelist of IP addresses, however the problem with IP Whitelisting in the cloud environment is the elasticity; the servers IP Address will change and come with new IP Address after every maintenance window in the cloud.

C. OAuth2.0 Authentication

OAuth 2.0 is the industry-standard protocol for authorization. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service. This can be done either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

An OAuth authentication flow defines a series of steps used to coordinate the authentication process between the application and HTTP service. Supported OAuth flows include:

1. Username-password flow: where the application has direct access to user credentials.
2. Web server flow: where the server can securely protect the consumer secret.
3. User-agent flow: used by applications that cannot securely store the consumer secret.

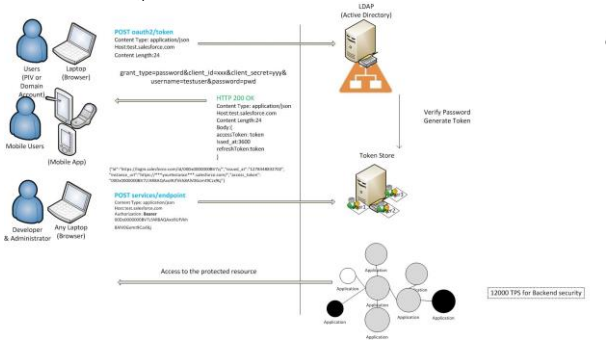


Fig. 5. OAuth2.0 Authentication

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 7 Issue 01, January-2018

In the OAuth2.0 Authentication, the user will still need to send the username and password in the request body similar to basic authentication but also introduce tokens. Tokens will be stored and maintained in the Token store on the server side. The same token can be used to call the service any number of times until it is expired. When the access token is expired, the user can use the refreshToken to get the new accesstoken. The glaring problem with the OAuth2.0 specification is that it constantly introduces more tokens. These tokens are essentially equivalent to passwords, and therefore, the server needs to maintain the state of each token. In addition to all the new tokens, all of the invalid token requests will increase the load on the token store, allocating a higher overall server load dedicated just to the tokens.

D. JSON Web Token (JWT) Authentication

A JWT token is actually a full JSON Object that has been base64 encoded and then signed with either a symmetric shared key or using a public/private key pair. The JWT can contain such information including the subject or user ID, when the token was issued, and when it expires. By signing with a secret, the JWT ensures that only the defined user can generate a unique token, which cannot be open for tampering (such as modifying the user ID or when it expires). While the JWT is signed however, the JWTs are usually not automatically encrypted (JWT Encryption is an optional feature). This means that any data which is in the token can be read by anyone who has access to the token.

One of the benefits of JWTs is they can be used without a backing store. All the information required to authenticate the user is contained within the token itself. In a distributed microservice world, it makes it easy to not rely on centralized authentication servers and databases. The individual microservice only needs some middleware to handle verifying the token (JWT libs are openly available for everything from Express to JVM MVC Frameworks) in addition to the secret key needed to verify. Verifying consists of checking the signature and a few other parameters such as the claims and when the token expires. JWTs are usually medium life tokens with an expiration date that may be set anywhere from a few weeks to significantly longer, customizable to the client's request.Verifying that a token is correctly signed only takes CPU Cycles and requires no IO or network access and is therefore easy to scale on modern web server hardware.

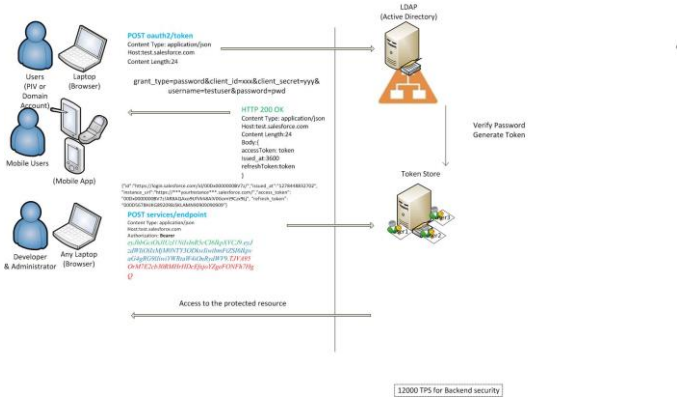


Fig. 6. OAuth2.0+ JWT Authentication

A JWT Token Consists of three parts: header, claims, and signature. All of these three parts are base64-encoded, combined into one String and included in the Authorization Header.

Authorization:Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cb30RMHrHDcEfxjoYZgeFONFh7HgQ

Header

The header simply declares that this is a JSON Web Token and the algorithm used to generate the signature. Below block shows the JWT header sample

```
{
  "alg" : "AES256",
  "typ" : "JWT"
}
```

Claims

This is the heart of the token. Claims are meant as details about the user which are meant to transfer between parties. Whether the user wants to authenticate on one server and access protected resources from another server, or the API issues the token and then stores the token on a client-side app, the token can be generated to facilitate a variety of different circumstances.

Signature

It is possible to decode the token using base64 algorithm; however the signature would then be rendered invalid. The signature is generated using a private key to hash the Header and Claims. This way only the original token will match the signature.

This raises an important implementation detail. Only applications that have a private key, i.e. server-side apps, can trust the claims of the token. It is not advised to place a private key in a browser side application.

One of the downsides with JWTs is that banning users or adding/removing roles is more complex if the action is required immediately. The JWT has a predefined expiration date which may be set a defined date into the future. Since the token is stored client side, there is no way to directly invalidate the token even if the user is marked as disabled in the database, and must wait for the token to expire. This can influence the architecture, especially if designing a public API that could be starved by one power user or an e-commerce app where fraudulent users need to be banned. There are workarounds, for example, if all that is needed is banning compromised tokens or users, the user can create a blacklist of tokens or user IDs; however this may reintroduce a database back into the authentication framework. A recommended way to blacklist is to ensure each token has a jti claim (or a JWT Id which can be stored in the Db). Assuming that the number of tokens the user would like to invalidate is notably smaller than the number of users in the application, this process should then scale fairly easy. The user may even locally cache it in the same process as the API code, removing some dependency on a database server.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 7 Issue 01, January-2018

On the other hand, if the user has an enterprise app with many roles such as admin, project owner, service account manager and they want the token responsibility to have immediate effect, then development can be more complicated. For example, in the instance where an admin is modifying someone else's authorized roles such as his/her immediate reports, the modified user wouldn't even know their roles have changed without refreshing the JWT.

Another downside is the token can grow as more fields are added. In stateless apps, the token is sent for virtually every request, and therefore can be an impact on data traffic size. For example, the enterprise app may have many roles, which may add bloat and complications for what to store in a token. In mobile apps where smartphone owners are concerned for client-side latency and data usage, JWTs may add too much payload to each request.

E. HTTP Signatures

In the JWT, only the Authorization header is base64-encoded and signed, so if anyone were to get the JWT token and request, they can then update the HTTP Request body. To avoid this kind of data manipulation issues, HTTP Signatures allow the client the sign the entire HTTP Message, so that others can touch the request on the network. HTTP Signatures have been introduced by Amazon, Facebook and Google. Later in 2016, a new Work in Progress specification called Signing HTTP Messages came into practice. As per this specification, the benefit of signing the HTTP message for the purposes of end-to-end message integrity is so that the client can be authenticated using the same mechanism without the need for multiple loops.

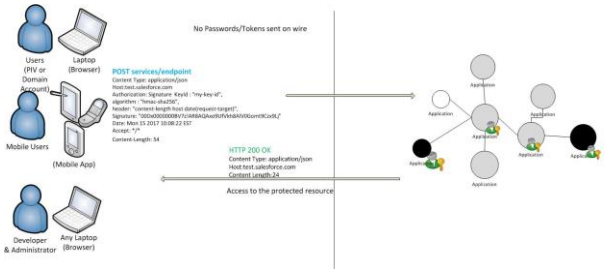


Fig. 7. HTTP Messages Authentication

REFERENCES

[1] https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

[2] https://www.thoughtworks.com/insights/blog/microservices-nutshell

[3] I.S. Jacobs and C.P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G.T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271-350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740-741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] A Design Methodology for Reliable Software Systems. FJCC, Dec.1972,

[9] https://tools.ietf.org/html/draft-cavage-http-signatures-09

[10] https://tools.ietf.org/html/draft-ietf-oauth-pop-architecture-08