Managing a Secure Refresh Token Implementation with JSON Web Token in REST API

Ehab Rushdy [1], Walid Khedr [2], Nihal Salah [3]

[1] Associate Professor of Information Technology, Faculty of Computers & Informatics, Zagazig University, ehab.rushdy@gmail.com

[2] Professor of Information Technology, Faculty of Computers & Informatics, Zagazig University, khedrw@yahoo.com

[3] Department of Information Technology, Faculty of Computers & Informatics, Zagazig University, nihal.radwan@hotmail.com

Abstract

JSON Web Token (JWT) is a compact and self-contained mechanism, digitally authenticated and trusted, for transmitting data between various parties. They are mainly used for implementing stateless authentication mechanisms. The Open Authorization (OAuth 2.0) implementations are using JWTs for their access tokens. OAuth 2.0 and JWT are used token frameworks or standards for authorizing access to REST APIs because of their statelessness and signature implementation and JWT tokens are based on JSON and used in new authentication and authorization protocols in OAuth 2.0 because of their small size. When refresh tokens are stored in cookies, the size limit of a cookie or URL may be quickly exceeded. There may be refresh tokens for accessing users and getting the refresh token is a bit more complicated and refresh tokens in the browser require additional security measures and the attacker steals a refresh token and attempts to use it after the application has already used it. This implies that the attacker was able to steal a refresh token from the application. If the refresh token can be stolen, then so can the access token, even short token lifetimes can still lead to major abuse scenarios. In this article, we discuss the security properties of refresh tokens in the browser and the pattern to secure JWT tokens in the web front-end better. We propose a Backend for Frontend (BFF) pattern, where the token handling is deferred to the server-side component to a secure token that provides a lot of flexibility to the client-side.

Keywords: Authorization, Authentication, JWT, BFF, Security

1.Introduction

Every the Representational State Transfer (REST) [1] Application Programming Interface (API) request from the client to a server must contain all the necessary information necessary to serve the request. The server maintains neither state nor context. For authorize access to protected REST APIs.

OAuth 2.0 [2] and JWT [3] are two of the most widely used token frameworks or standards for authorizing access to REST APIs.

1.1 Application programming interfaces

Application programming interfaces (APIs) [4]created using the Representational State Transfer (REST) protocol have become a nearly universal standard today for connecting mobile apps and websites with servers and third-party systems that the service providers expose several services as web-accessible APIs for users to build applications or consume services. One of the key aspects of designing an API platform is controlling who has access to data.

An API platform should be capable of permitting completely different levels of user access ways wherever authorized users could get unlimited access to secure APIs while non-authorized users will only access APIs that are public. From the API platform perspective, granting access to completely different levels of users should be created as easy as possible. As a result, many organizations provide different authorization strategies to access their APIs on behalf of the user. This has created a major requirement for a common global standard for securing APIs. This is where OAuth 2.0 stands out among other standards.

Every API request from the client to a server contains all the necessary information necessary to serve the request. The server maintains neither state nor context. To create a common API security model, suppose all endpoints require an OAuth 2.0 access token issued from a common identity provider and have the appropriate API security token checks in place and authorize access to protected REST APIs that OAuth 2.0 and JSON Web Token are two of the most widely used token frameworks or standards for authorizing access to APIs.

1.2 The OAuth2.0 Protocol

OAuth2.0 [5] is an open standard for authorization and provides a method for clients to access server resources on behalf of a resource owner, such as a different client or an end-user. It also provides a way for applications to gain limited access to a user's protected resources without the need for the user to disclose their login credentials to the application. In OAuth2.0, the client requests access to resources controlled by the resource owner and hosted by the resource server by giving a different set of credentials (not the resource owner’s credentials) to access the resource. In OAuth2.0, the client will obtain an access token from the authorization server, which can be used to access server resources on behalf of the resource owner.

1.2.1 The OAuth 2.0 flow

There are four types of roles specified in OAuth2.0 :

1. Client: Also known as "the app". It can be an application running on a mobile device or a web app. It is the application requesting access to protected resources keep on the resource server. It additionally obtains authorization from the resource owner.

2. Resource owner: It's referred to as a person, it is called an end-user who is capable of authorizing access to a protected resource or a service.

3. Resource server: Data owned by the resource owner such as sensitive data and it can accept and respond to protected resource requests

4. Authorization server: This server supplies access tokens to the client and it is responsible for validating (authorization grants) and issuing the (access tokens) that give the client application access to the end user's data on the resource server.

1.2.2 OAuth 2.0 grant types

An authorization grant type is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. In the abstract protocol flow above, the first four steps cover getting an authorization grant and access token. OAuth defines four grant types, each of which is useful in different cases can be seen in Table 1.

Table 1: OAuth 2.0 Grant Types

Grant type	Used for
Client Credentials	When two machines need to talk to each other, e.g., two APIs
Authorization Code	This is the flow that occurs when login to a service using Facebook, Google, GitHub etc.
Implicit Grant	Like the Authorization Code Grant Type, but user-based.
Password Grant	The user's username and password are exchanged directly for the OAuth2 tokens. which is more suitable for server apps used by a single user account. This is by far the easiest authentication scheme to implement that this paper focuses on this type.

1.3 Json web token

Json web token [6] is an open standard that defines a compact and self-contained way for securely transmitting information between parties. It is an authentication protocol where we allow encoded claims (tokens) to be transferred between two parties (client and server) and the token is issued upon the identification of a client, with each subsequent request we send the token.

1.3.1 Structure of JWT

Every JWT is generated with the same structure. There are three parts: The header, the payload, and the signature, separated by a period character (.). Each section (except signature) is comprised of base64urlencoded JSON containing specific information for that token as shown in Figure 1 :

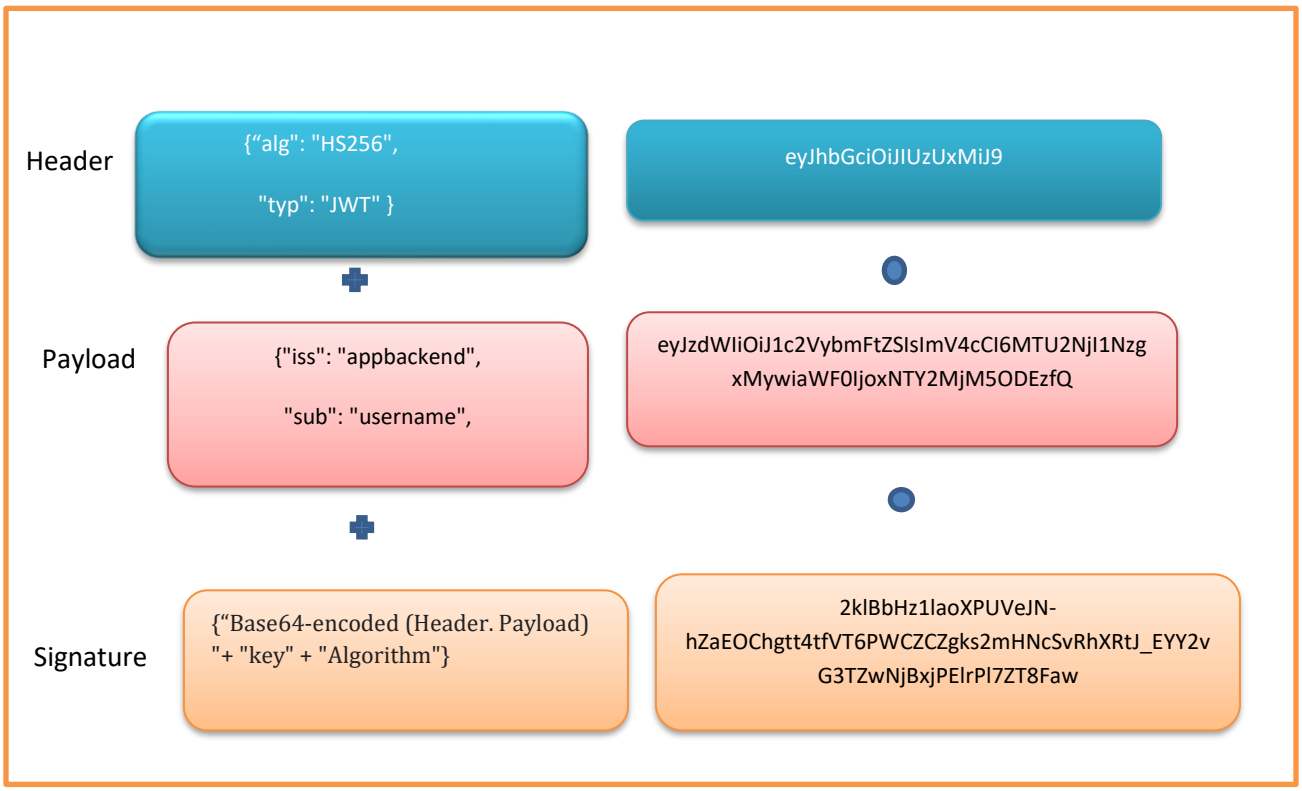

Figure 1. Structure of JSON Web Token: [7]

-The first section is known as the header. It contains the algorithm used to generate the signature (e.g., HS256 or RSA256) and the type of token used (JWT).

-The second section is the payload (also known as claims) and the core of any JWT: contains verifiable security statements, such as the identity of the user and the permissions they are allowed. There are several different standard claims for the JWT payload such as: “iss” the issuer that issued the JWT, “sub” the subject claim identifies the principal that is the subject of the JWT ,“aud”: token audience (who the token is intended to process the JWT),(The URI of the authentication service instance to be used), “exp” the expiration time and the "iat" (issued at) claim identifies the time at which the JWT was issued , The "jti" (JWT ID) claim provides a unique identifier for the JWT and The "nbf" (not before) claim identifies the time before which the JWT must not be accepted for processing. These fields can be useful when creating JWT.

-The final section is the signature used to validate the token. It contains the encoded header, the encoded payload, the secret, the algorithm that has been defined in the header and sign that as shown in Figure 2.

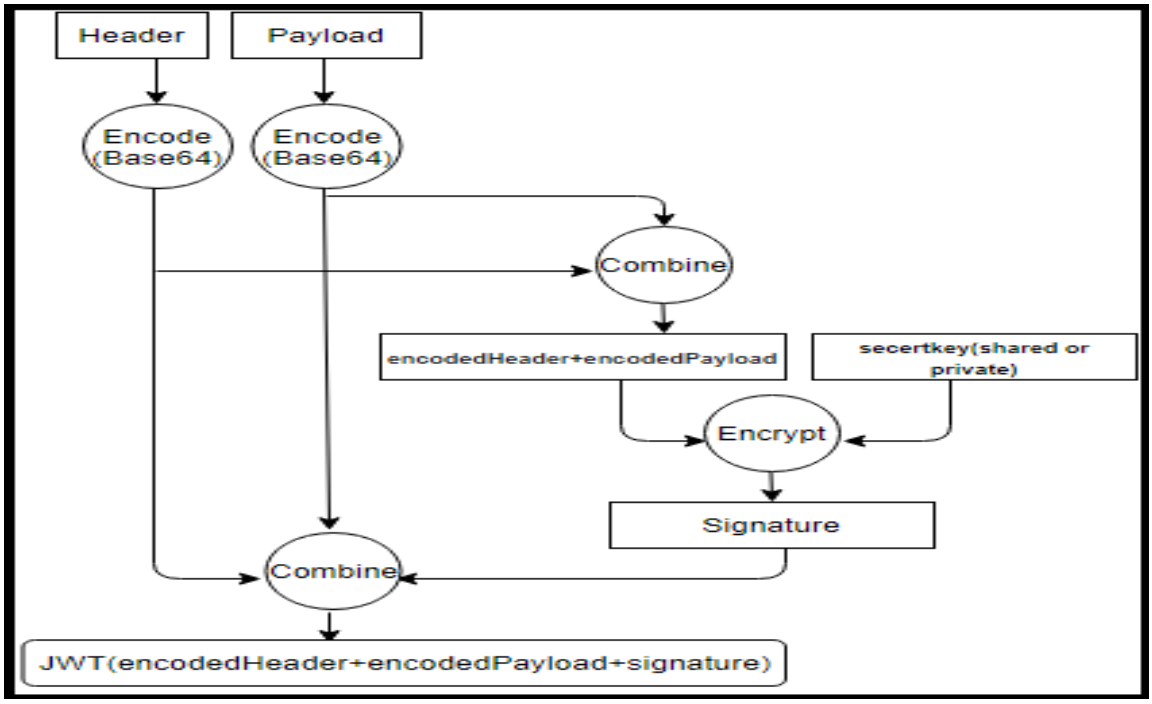

Figure 2. Signing authentication token

This is what makes a JWT secure and ensures the integrity of JWT during transport. The output of these steps yields the final part of the token as shown in Figure 3:

eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ1c2VybmFtZSIsImV4cCI6MTU2NjI1NzgxMywia
WF0IjoxNTY2MjM5ODEzfQ.2klBbHz1laoXPUsVeJN-
hZaEOChgtt4tfVT6PWCZCZgks2mHNcSvRhXRtJ_EYY2vG3TZwNjBxjPElrPl7ZT8Faw

Figure 3. Final part of JWT

1.3.2 Common JWT signing algorithms

There are two main ways to sign JWTs cryptographically: using symmetric or asymmetric keys. Both types are commonly used, and provide the same guarantees of authenticity. The most common algorithms are HMAC SHA-256 symmetric algorithm and RSA-256 an asymmetric algorithm can be seen in Table 2.

Table 2. Compression Between SHA256 and RSA256[8]

	SHA256	RSA256
Stand For	HMAC with SHA-256	RSA encryption plus SHA-256 hashing
Key type	Symmetrical secret key	Asymmetrical Public / private key pair RSA
Use cases	That's an algorithm that encrypts and hashes the message (JSON data) at the same time using symmetrical secret key. The same key is used for encryption and decryption of the message.	RSA is an asymmetric encryption algorithm, which means it operates on a pair of keys – public and private. Private key is used to encrypt a token, and public one – to decipher it.

1.3.3 Access token

Access tokens carry the necessary information to access a resource directly, when a client passes an access token to the authorization server managing a resource, that authorization server can use the information contained in the token to decide whether the client is authorized or not. Access tokens usually have an expiration date and are short-lived.

1.3.4 Refresh token

Refresh tokens help in getting new access tokens without asking users to sign in again. It is handled by the application itself. We provide Client-Side application OAuth 2.0 with two different tokens; one is an access token and the other is a refresh token.

The main reason to use refresh tokens in web applications is to reduce the lifetime of an access token. When the client gets an access token with a lifetime of five to 10 minutes, that token will likely expire while the user is using the application.

1.3.5 Cross Site Scripting (XSS) attacks

Cross site scripting [9] is the method where the attacker injects malicious script into trusted website application running on the victim's browser and attempt to inject JavaScript through form inputs. The injected code reads and transmits auth JWT tokens and cookies to the attacker. Local Storage can be read just as easily from the same injected code, so if the authentication data was stored in local storage, it could be easily grabbed. Web Storage (local Storage/session Storage) is accessible through JavaScript on the same domain. This means that any JavaScript running on application will have access to web storage, and because of this can be vulnerable to (XSS) attacks can be executed on the sessions that do not need any authentication.

1.3.6 Cross-site request forgery attack

Cross Site Request Forgery (CSRF/XSRF) [10] is one of the most popular ways of exploiting a server. It attacks the server by forcing the client to perform an unwanted action and this attack is not used to steal auth JWT tokens and attempts to trick users into performing an action using an existing session of a different website. Since cookies with JWT are sent by the browser client by default, an attacker might trick user into clicking on a malicious link and submitting an authenticated request to client app.

1.3.7 Cross-Origin Resource Sharing (CORS)

Using Cross-Origin Resource Sharing (CORS) [11] to secure user data that CORS handles this vulnerability well, and disallows the retrieval and inspection of data from another Origin, CORS will prevent the third-party JavaScript from reading private data , and will fail AJAX requests with security errors. CORS is an important consideration for using developing browser applications with JavaScript because most requests to resources are sent to an external domain (including Authentication Server and Resource Server) for a web service API, a main web application gathered the information from these servers.

1.4 Backend for Frontend design pattern

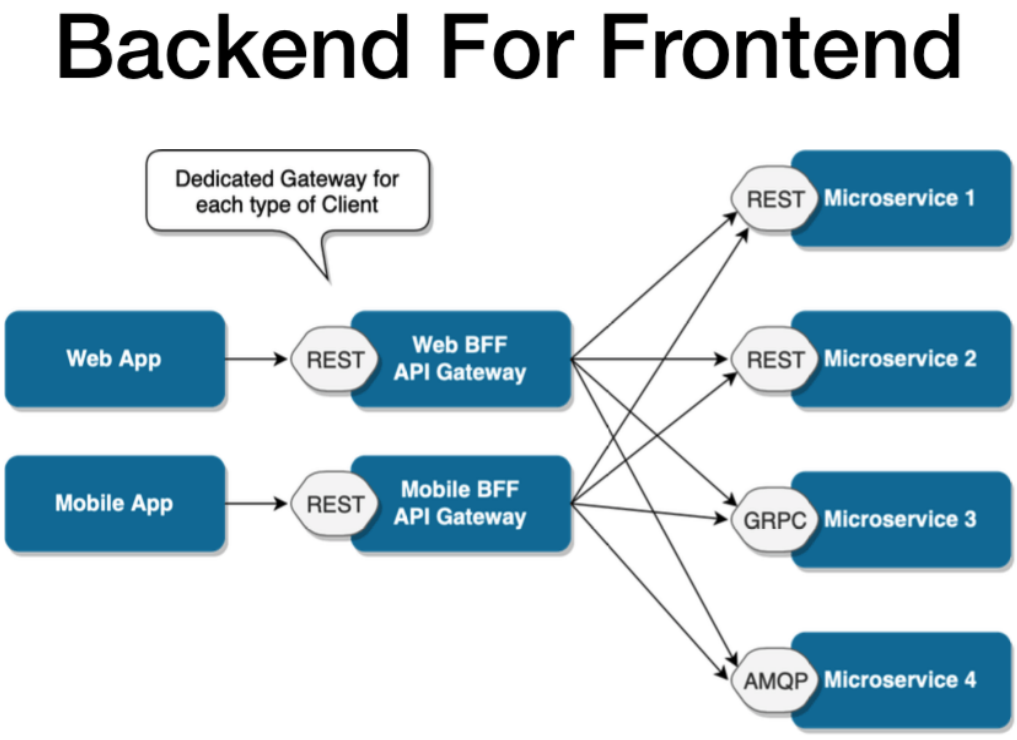

Figure 4 . BFF design pattern :[12]

BFF [12] is essentially a variant of the API Gateway pattern. It also provides an additional layer between microservices and clients. But rather than a single point of entry, it introduces multiple gateways for each client.

With BFF, you can add an API tailored to the needs of each client, removing a lot of the bloat caused by keeping it all in one place. The resulting pattern can be seen in Figure 4.

Our proposed solution would be to have different APIs for mobile and web. The idea was that having the team working on the client owns the API would allow for them to move much quicker as it required no coordination between parts.

A BFF is, in simple terms, a layer between the user experience and the resources it calls on. When a mobile user requests data, in a BFF situation, their request is translated through the BFF and into a general layer below it.

The BFF pattern comes with many potential benefits [13]:

- **Call the backends in parallel:** We can call the backends in parallel and perhaps respond faster to the end-user.
- **Filter the responses**: We can remove internal or sensitive data, such as an unwanted customer flag, or unnecessary data that just adds complexity to the frontend and drains battery power when parsing on mobile devices.
- **Transform the responses**: We can transform the responses to the frontend into something more usable, such as translating internal codes into something more descriptive.
- **Enhanced security**: We can protect the insecure backend solutions, so the frontend doesn't need to deal with different authentication methods in different backend systems.
- **Handle different protocols**: The BFF can call SOAP, REST, and other types of services, but still use a single protocol when interacting with the client.
- **Encapsulate advanced business logic**: Issuing a new insurance policy may be a complex operation that requires multiple service calls, but this can be simplified for the clients into a single endpoint with only the absolute minimum data in a flat structure.
- **Caching across clients**: Some of the backends may be slow so caching across different clients may be an effective way to deliver acceptable response times to the end-users. Moreover, if we need to call usage-based external APIs, it may also be worth caching those responses across clients to reduce the cost.
- **Protect the client from changes to the backend**: Changes in the backends APIs need not to result in changes in the UI, the changes can be handled in the BFF.

1.4.1 Backend for Frontend (BFF) pattern

Front-end and back-end development are the two major areas of application development. The front-end is the presentation layer of an application, which is directly accessible to the user and displayed in the user interface. The back-end or server-side is not accessible to end-users. It acts as the backbone of the application.

BFF [14] is an additional layer between clients and the resources it calls on. The key component of this pattern is an application that connects the front-end of the application with the backend.

Building a BFF allows to intelligently make batch calls to other multiple back-end or servers and return the data all at once, or return a more convenient representation by transforming and formatting the data.

BFF architecture can be used to create multiple back-end for client-facing mobile or web applications. This code pattern helps iterate features faster and have control over multiple back-end for applications without affecting the experience for a corresponding mobile or web application.

BFF [15] is shared among hundreds or even thousands of client instances, each operating on behalf of a different user. The BFF keeps track of these users with a cookie-based session. Like many other patterns, using the BFF in an application depends on the context and the proposed architecture we plan to follow. However, our application depends on servers and consumes many external APIs and other services, it is better to use a BFF to streamline the data flow and provide a lot of efficiency to our application.

1.4.2 The role of a BFF

1. User experience platforms like Mobile and Web Applications communicate to their own BFF server, in order to gather the appropriate APIs and service requests needed.
2. Each BFF calls upon the necessary services that are requested by the frontend.
3. Separation of concerns that Frontend requirements will be separated from the backend concerns. This is easier for maintenance.
4. Certain sensitive information can be hidden, and unnecessary data to the frontend can be omitted when sending back a response to the frontend, which will make it harder for attackers to target the client application.

Build a "Backend for Frontend" [16] (e.g. a BFF or Dispatcher) that acts as a single API for a client. Implement different BFFs for different types of clients, each with an API that is customized to what that client type needs.

Each Backend for Frontend will then interface Results: A BFF can perform the following actions:

- Organization – It can organiz several calls to business microservices that result from a single client action
.
- Translation – It can translate the results of a microservice into a channel-specific representation that more cleanly maps to the needs of the user experience of that client.
- Filtering – It can filter results from a business microservice that are not needed by a particular client type.

1.4.3 Endpoints

This specification introduces "BFF-token" specialized endpoint that the backend exposes to support the frontend in acquiring tokens and user-session information. For the purpose of facilitating the implementation of the pattern with minimal configuration requirements, the endpoint is meant to be used by the applications' frontend, and the frontend only. As such, the backend must verify the call is occurring in the context of a secure session (by mandating the existence of a valid session cookie received via HTTPS).

1.4.4 The BFF-token Endpoint

The "BFF-token" endpoint is exposed by the backend to allow the frontend to request access tokens. The backend MUST support the use of the HTTP "GET" method for the "BFF-token" endpoint and the backend ignores unrecognized request parameters.

2.Related work

Various performed researches and manufactured models have been proposed for the Security Analysis of OAuth 2.0. Several works have been done regarding OAuth security and its application in constrained environments.

The work [7] proposed an approach that combines OAuth 2.0 with JWT in REST API and test the algorithm performance on JWT and compare the performance of the two HS256 (HMAC with SHA-256) and RS256 (RSA Signature with SHA-256) algorithms in the parameters of the speed of generating tokens, the size of tokens, time data transfer tokens and security of tokens against attacks.

The use of token-based authentication using JWT has already been investigated [17] which focused on the security and privacy challenges of cloud computing with specific reference to user authentication and access management for cloud SaaS applications. The suggested model uses a framework that harnesses the stateless and secure nature of JWT for client authentication and session management.

The work [18] explained scenarios of OAuth 2.0 Protocol with an example for establishing identity management standards across services, provides an alternative to sharing our usernames and passwords, and exposing ourselves to attacks on our online data and identities.

The work [19] focuses on security vulnerabilities of the OAuth 2.0 protocol that proposed an attacker model to perform systematic analysis of the root causes of common attacks like replay attacks, impersonation attacks and, forced login CSRF attacks.

The work [20] this work provided an additional layer of protection against CSRF attacks for OAuth 2.0 services and proposed a new practical technique used to mitigate CSRF attacks against both OAuth 2.0 and OpenID Connect.

The work [21] proposed a scheme based on OAuth 2.0 and JSON Web Token for securing existing health care services in the IOT cloud platform.

The work [22] provided the stateless and compact feature of JWT for authentication and access authorizations for cloud users.

3.Background

This section describes some issues and solves all these problems that one snippet of malicious JavaScript code, coming in through Cross-Site Scripting (XSS) vulnerability [9] or a compromised remote code file, can quickly result in the theft of a refresh token.

Malicious JavaScript poses a threat to browser-based applications to steal tokens. This nightmare scenario gives the attacker long-term access to an API on behalf of the user we need to avoid at all costs.

Also, we want to avoid dealing with Cross-Origin Resource Sharing (CORS) [11] policies and Cross-Site Request Forgery (CSRF) [10] tokens for mitigating CSRF attack surfaces at the authentication service by sending tokens in HTTP body instead of cookies.

There is one main problem that surfaces with this sort of deployment and it comes down to Cookies. The most secure way for us to store our session on the client will be httpOnly Cookie. The issue with this is related to the fact that when refresh tokens are stored in cookies, the size limit of a cookie or URL may be quickly exceeded that depends on the application's architecture and especially the data that is stored in the token.

There may be refresh tokens for accessing users and getting the refresh token is a bit more complicated and refresh tokens in the browser require additional security measures and the attacker steals a refresh token and attempts to use it after the application has already used it. This implies that the attacker was able to steal a refresh token from the application.

Refresh tokens need additional protection. Concretely, refresh tokens exposed to the browser should be protected with a Backend for our Frontend (BFF) pattern.

The storing the application credentials in the application configuration files and the access tokens in the browser's local or session storage and the access tokens become accessible if malicious cookies or malware programs have been accepted that can attack the application for sensitive information.

Therefore, we introduced a Backend for Frontend (BFF) which handles cookies. This involves having a service that runs in a web server that acts as the Authorzation server for the application that serves the end-users.

This actually solves the problems that we have secure JWT handling. Authentication service doesn't have to deal with cookies, BFF pattern, where the token handling is deferred to a server-side component or API. The solution to secure a sensitive refresh token is using a backend-for-frontend.

4. Methodology

This section proposes a pattern in that BFF also assumes a minimal security role. It accepts requests from the client application, augments them with JWT OAuth 2.0, and forwards the request to the API. Similarly, any response from the API is forwarded to the client application.

The BFF becomes the OAuth 2.0 client application. Since the BFF runs on a backend system, it can be configured as a confidential client.

The BFF is the client, so it initializes the OAuth 2.0 flow that runs in the user's browser, which it exchanges for tokens with the authorization server using the client application as shown in Figure 5. With the access token and refresh token, the BFF can forward API requests, and using the refresh token requires the BFF to authenticate to the authorization server.In this approach, we create a Backend for Frontend (BFF) architecture service that this pattern helps for giving more control over the APIs without affecting the user experience as shown in Figure 6.

We keep a short expiry time for JWT access token to make it more secure but we don't need users to sign in every 5 minutes. So, we use refresh token for this. Every time your access token expires the application can use a refresh token to verify the user and issue a new JWT access token.

4.1 BFF pattern flow

This document describes what the protocol flow is concerned, the topology can be broken down into five Roles as shown in Figure 5:

- **Frontend**: This represents the application code executing in the user agent, controlling presentation, and invoking one or more resource servers.
- **Backend**: The backed represents code executing on a server, in particular on the same domain from where the frontend code has been served.
- **Client**: It can be an application running on a mobile device or a web app. It is the application requesting access to protected resources keep on the resource server.
- **Resource Server**: This represents OAuth2 resource server exposing the API the frontend needs to invoke.
- **Authorization Server**: This represents OAuth2 authorization server handling authorization for the API the frontend needs to invoke.

4.2 Authentication flow

This section provides a high-level description of the way in which the frontend can obtain and use access tokens with the help of its backend.

As a prerequisite for the flow described below as shown in Figure 5, the backend must have established a secure session with the user agent, so that all requests from that user agent toward the backend occur over HTTPS and carry a valid session artifact (such as a cookie) that the backend can validate.

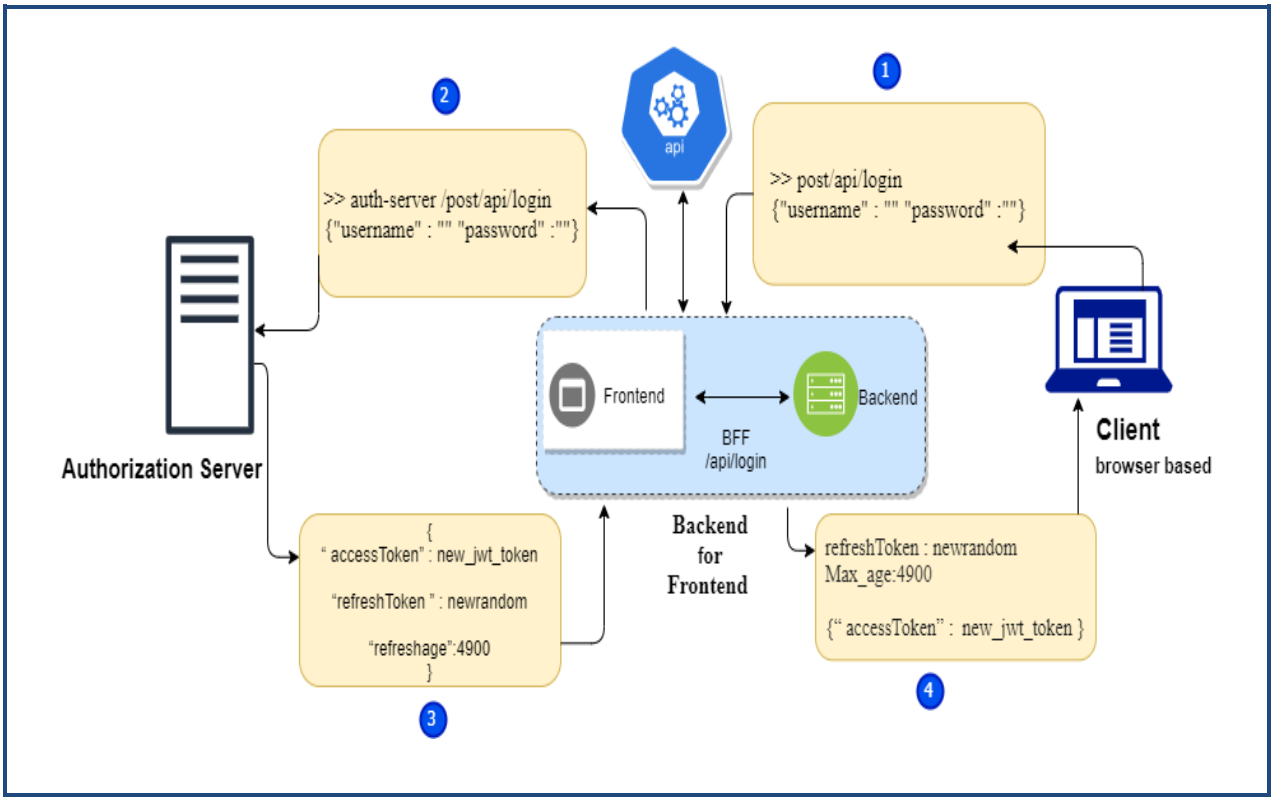

Figure 5. Login workflow

1. The web client sends credentials in HTTP body to our BFF via a relative /api//login path.
2. BFF forwards this request to the Authorization server without modifications.
3. Authorization service responds with accessToken, refresh token, and refreshAge in HTTP body in httpOnly cookie.
4. BFF creates an HttpOnly cookie from refreshToken with an age of refreshAge. Then it responds to the web client setting this cookie and returning accessToken in the HTTP body,15 minutes go by and the access token expires. Requests made with this expired token now fail with 401 errors.

4.3 Refresh token flow

This section provides a high-level description of the way in which the frontend can obtain and use refresh access tokens with the help of its backend as shown in Figure 6:

1. Web client OAuth2.0 sends refresh_token (stored in cookie) via HTTP Only cookie to BFF via a path /api/token/refresh path.

2. BFF uses the session information in the request to extract refresh_token from cookie and forwards it to the Authorization server in the HTTP body. If JWT access token is still valid, the request can be forwarded directly by BFF.

3. Authorization server responds with a new access_token, refresh_token, and refresh age in the HTTP body.

4. BFF creates an HTTP Only cookie from the refresh_token with an age of refresh age then it responds to the web client setting this cookie and returning access token in the HTTP body. If the access token has expired, the BFF uses the user's refresh token to obtain a fresh access token before forwarding the request.

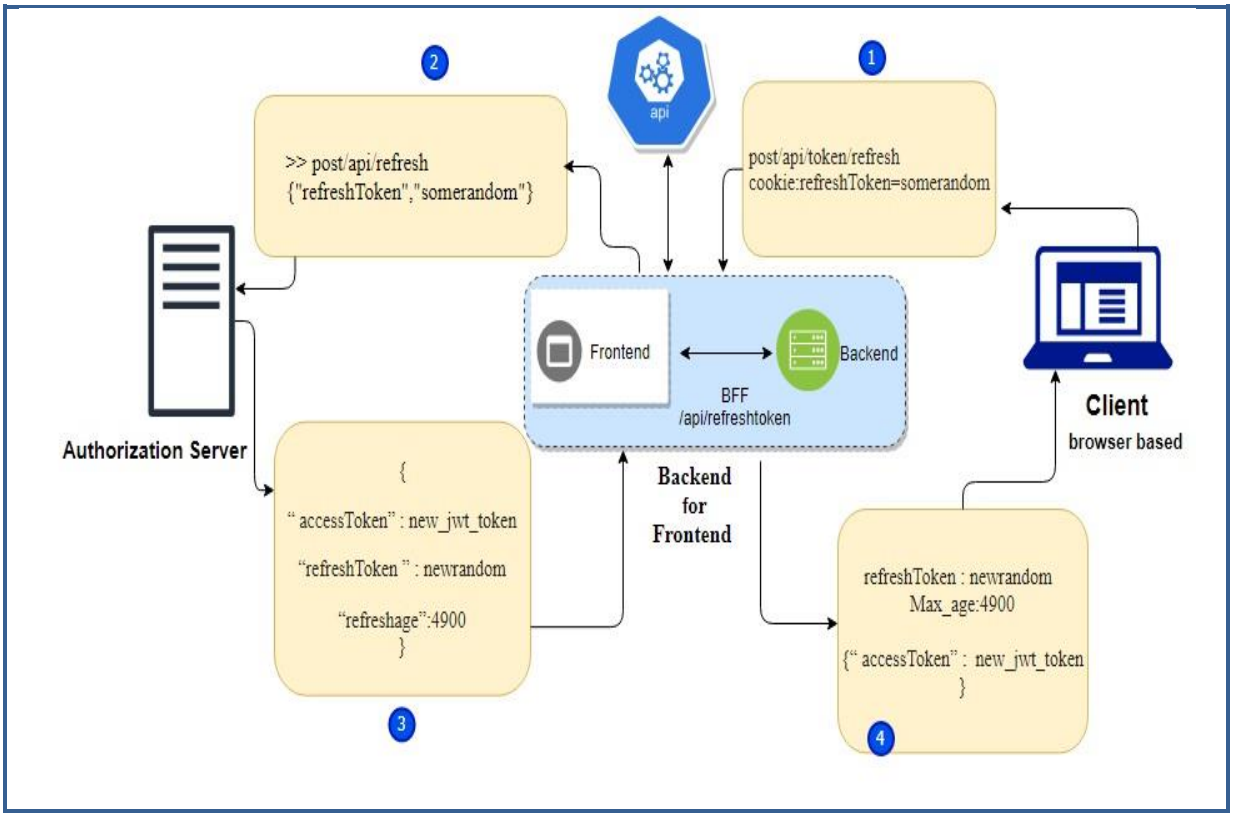

Figure 6. Refresh token workflow

4.4 Refresh token invalidation

Therefore, we have to delete it from the Authorization server's refresh_token store and clear it from the browser cookie once refresh_token invalidate as shown in Figure 7.

This token only lasts 15 minutes. In order to have a session longer than 15 minutes, we need the ability to refresh this token using our refresh token. We can do this with a /refresh route.

1. Web client OAuth 2.0 sends refresh_token via HTTP Only cookie to our BFF via a relative /api/v1/invalidate path.
2. BFF extracts refresh_token from the cookie and forwards it to the Authorization service in the HTTP body.
3. Authorization server clears refresh_token from the store.
4. BFF clears refresh_token from browser cookie.

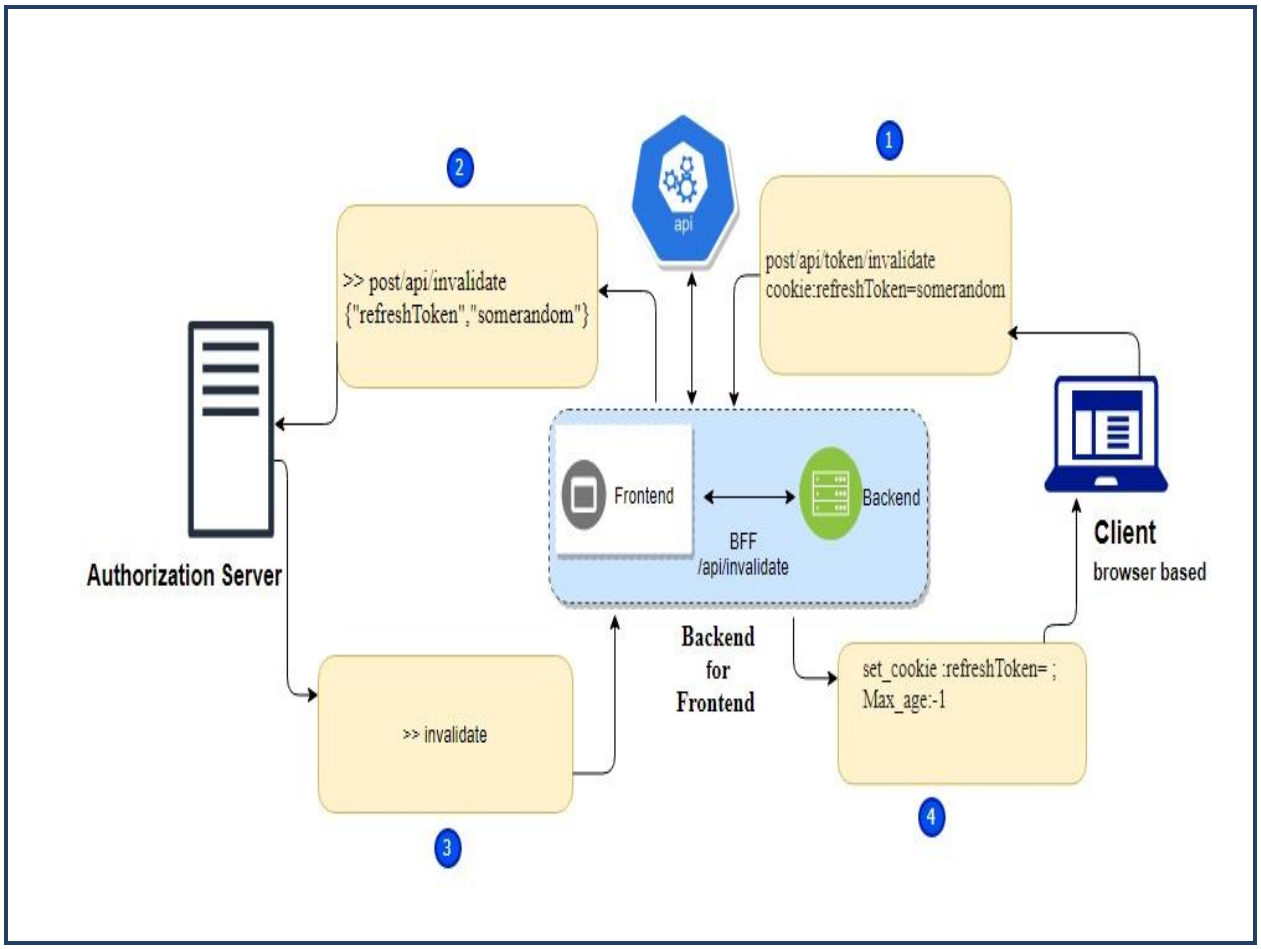

Figure 7. Refresh token invalidation workflow

4.5 Generating tokens

There are two types of JWT tokens that were created:

- **Access Token.** Used to make requests on behalf of a user. Any authenticated routes will verify the token in order to establish trust. To improve security this token a short expiration of 15 minutes.
- **Refresh tokens**. Used when the access token expires to fetch a new one. Has a long or no expiration.

4.6 Requesting access tokens to the backend

To obtain an access token, the frontend makes a request to the backend at the "BFF-token" endpoint URI. The flow includes the following steps, as shown in Figure 5.

1. The frontend generates the request and sends it to the "BFF- token" endpoint.

2. The backend examines the request, validating whether it includes a valid user session: if it doesn't, it rejects the request.

3. The backend extracts user information from the session, using whatever mechanism it deems suitable, and verifies whether it already has in storage a suitable access token satisfying the request.

4. If there is no suitable access token stored, the backend verifies whether it has the necessary artifacts to request it to the authorization server without requiring user interaction, by using a refresh token previously stored for the current user. If it does, the backed contacts the authorization server with a token request using the grant of choice as shown in Figure 6, if there is not suitable artifact required to perform a request toward the authorization server, the backend returned an error to the frontend as shown in Figure 7.

5. If the authorization server returns the requested token, the backend returns it to the frontend, as shown in Figure 5, if the authorization server denies the request, the backend returns an error to the frontend as shown in Figure 7.

5. Experimental results and discussion

The proposed scheme is done by making OAuth2.0 with JWT. It has been developed in JAVA and configured with Spring Security OAuth2 [23] by implementing a Backend for Frontend (BFF) to secure JWT handling. The Spring Boot framework tries to solve the problems associated with using Spring for Java web development.

The main program of the project is run in the web application, web server and back-end API server (including Authentication Server,Resource Server, and BFF service), we treat web browser as OAuth Client and users must authorize the device or browser to access their resource on the resource server, as well as using ajax-request in JavaScript to obtain a JWT from the authorization server and connect with REST API.

5.1 Evaluation the Backend for Frontend (BFF) architecture pattern for secure JWT token

A backend-for-frontend offers significant security benefits over browser-based applications. The BFF acts as the OAuth 2.0 client, handling JWT tokens, allowing it to apply security best practices for confidential clients.

Using the strategy of BFF, this means that:

- The malicious code can no longer access the tokens since they are only available to the BFF.
- Cookie security measures (Http Only attribute) prevent the malicious code from stealing the session with the BFF.
- The BFF is required to authenticate to the authorization server when exchanging refresh tokens.
- The BFF uses sender-constrained access tokens and sender-constrained refresh tokens.
- JWT Access tokens and refresh tokens are never uncovered to the browser.
- The BFF requires to follow cookie security best practices to guarantee the security of the cookie. Concretely, this means that to set a cookie with the name "BFFCookie," the following header has to be used: Set-Cookie: Host-BFFCookie; Secure; HttpOnly; SameSite.
- Using this combination of same-domain applications and OAuth, we prevent access tokens from ever appearing in the context of the browser, while still preventing CSRF.

BFFs are traditionally used to aggregate various APIs into a single coherent API to serve a client application. In our proposed pattern, the BFF also assumes a minimal security role. It accepts requests from the client application, augments them with OAuth 2.0 JWT tokens, and forwards the request to the API. Similarly, any response from the API is forwarded to the client application.

As a result, is increasingly common practice to use the BFF pattern: rely on the backend component for obtaining tokens from the authorization server, and sending back to the frontend the resulting access tokens for the direct frontend to API communication.

As long as the mechanism used for transmitting tokens from the backend to the frontend is secure, we provide a proposed pattern on how to implement this pattern in which a frontend component can delegate token acquisition to its back-end component.

5.2 Practical experiments

Testing is done with Postman who has a function as application used for testing the REST API that was created. We defined these routes with postman:

- /login For logging in with username and password
- /logout For logging out

- /refresh For retrieving a new Access Token
- /invalidate Authorization server clears refresh_token from the store

5.2.1 No valid session found

All requests to the backend must be performed in the context of a valid authenticated session, typically by presenting a session cookie over a TLS channel HTTPS. If the backend cannot find or validate a session, it must reject the request and return a message with an error parameter value of "invalidate" as shown in Figure 7.

5.2.2 The backend request to the authorization server fails

If the backend doesn't have the necessary artifacts (refresh token for the current user or requested resource) to request a suitable access token to the authorization server without requiring user interaction, it will reject the request and return a message as shown in Figure 7, with an error parameter value of " invalidate ".

5.3 Comparison using BFF pattern and storing JWT on the cookies for JWT's Storage

Table 3. Comparison using BFF pattern and storing JWT on the cookies

Method	Access credentials secure during authentication	Speed	Secure against CSRF
Using the cookies	Not vulnerable	Fast	vulnerable
BFF pattern	Not vulnerable	Average (double the number of requests)	Not vulnerable

6. Conclusion

In this article, we proposed Backend for Frontend (BFF) architecture pattern offers significant security benefits over browser-based applications and a lot of flexibility client side. The BFF acts as the OAuth 2.0 client, allowing it to apply security for store JWT token also the BFF provides to follow cookie security best practices to guarantee the security of the cookie. Additionally, the BFF can apply traffic analysis patterns to detect suspicious behavior that includes the attacker can perform a session riding attack by sending malicious API calls through the BFF pattern, but are safe and well proven when executed by a confidential client running on a backend.

References

1. Fielding, R.T. and R.N. Taylor, Principled design of the modern Web architecture. ACM Transactions on Internet Technology (TOIT), 2002. 2(2): p. 115-150.
2. Hardt, D., The OAuth 2.0 authorization framework. 2012, RFC 6749, October.
3. Jones, M., B. Campbell, and C. Mortimore, JSON Web Token (JWT) profile for OAuth 2.0 client authentication and authorization Grants. May-2015.[Online]. Available: https://tools. ietf. org/html/rfc7523, 2015.
4. Fielding, R., Representational state transfer. Architectural Styles and the Design of Netowork-based Software Architecture, 2000: p. 76-85.
5. Auth0. OAuth 2.0 Authorization Framework. 2013 April 19, 2021]; Available from: https://auth0.com/docs/protocols/protocol-oauth2.
6. Peyrott, S.E., The JWT Handbook. 2017.
7. Ehab rushdy, W.K., Nihal salah, Framework to secure the oauth 2.0 and json web token for rest api. Journal of Theoretical and Applied Information Technology -- Vol. 99. No. 09 -- 2021
8. Auth0. Introduction to JSON Web Tokens. Available from: https://jwt.io/introduction/.
9. Guo, X., S. Jin, and Y. Zhang. XSS vulnerability detection using optimized attack vector repertory. in 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. 2015. IEEE.
10. Wichers, D., Owasp top-10 2013. OWASP Foundation, February, 2013.
11. Sharing, W.C.C.-O.R. Cross-Origin Resource Sharing. 16 January 2014; Available from: https://www.w3.org/TR/cors/.
12. Adam, A. Microservices design patterns for CTOs: API Gateway, Backend for Frontend 2019; Available from: https://tsh.io/blog/design-patterns-in-microservices-api-gateway-bff-and-more/.
13. Abbott, M. BACKEND FOR FRONTEND (BFF) PATTERN. 2019; Available from: https://akfpartners.com/growth-blog/backend-for-frontend.
14. Syafariani, F., Application of Backend and Frontend Systems on Go-Baby Application in Bandung City. International Jurnal of Recent Technology and Engineering (IJRTE), 2019. 7(6s5): p. 125-131.
15. Newman, S. Backends for frontends. 2015; Available from: https://samnewman.io/patterns/architectural/bff/.
16. Brown, K. and B. Woolf. Implementation patterns for microservices architectures. in Proceedings of the 23rd Conference on Pattern Languages of Programs. 2016.
17. Ethelbert, O., et al. A JSON token-based authentication and access management schema for Cloud SaaS applications. in 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud). 2017. IEEE.
18. Leiba, B., Oauth web authorization protocol. IEEE Internet Computing, 2012. 16(1): p. 74-77.
19. Yang, F. and S. Manoharan. A security analysis of the OAuth protocol. in 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM). 2013. IEEE.
20. Li, W., C.J. Mitchell, and T. Chen. Mitigating CSRF attacks on OAuth 2.0 systems. in 2018 16th Annual Conference on Privacy, Security and Trust (PST). 2018. IEEE.

21. Solapurkar, P. Building secure healthcare services using OAuth 2.0 and JSON web token in IOT cloud scenario. in 2016 2nd International Conference on Contemporary Computing and Informatics (IC3I). 2016. IEEE.
22. Ethelbert, O., et al. A JSON token-based authentication and access management schema for Cloud SaaS applications. 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud) 2017; 47-53].