Authentication and Authorization for the front-end web developer

Biraj Paul

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 22.6.2020

Supervisor

Prof. Tuomas Aura

Advisor

Niall O'Donoghue, M.Sc., Gofore

Aalto University
School of Science

Copyright © 2020 Biraj Paul

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of the master's thesis

Author Biraj Paul

Title Authentication and Authorization for the front-end web developer

Degree programme CCIS

Major Secure Systems **Code of major** SCI3042

Supervisor Prof. Tuomas Aura

Advisor Niall O'Donoghue, M.Sc., Gofore

Date 22.6.2020 **Number of pages** 61 **Language** English

Abstract

Traditional web pages are hosted and served through a web server that are executed in a web browser in the user's devices. Advancement in technologies used to create web pages has led to a paradigm shift in web development, leading to concepts such as front-end and back-end. Browser-based technologies, particularly JavaScript, has seen enormous advancements in functionalities and capabilities. This led to a possibility of creating standalone web applications capable of running in the browser and relying on the back-end server only for data. This is corroborated by the rise and popularity of various JavaScript frameworks that are used by default when creating web applications in modern times. As code running on a web browser can be inspected by anyone, this led to a challenge in incorporating authentication and authorization. Particularly because storing user credentials and secrets on the web browser code is not secure in any way.

This thesis explores and documents authentication and authorization methods that can be securely implemented in a front-end web application. Token-based authentication and authorization has become widely accepted as the solution. OpenID Connect and OAuth 2.0 protocols were explored, which are the most commonly used token-based solution for authentication and authorization. Furthermore, three use-cases were described that used token-based solutions in real world client projects.

Keywords Token-based authentication, OpenID Connect, Front-end web development

Contents

Abstract 3

Contents 4

Abbreviations 5

1 Introduction 7

2 Background 9
- 2.1 Single-Page Application and Traditional Web Application 9
  - 2.1.1 Traditional Web Application 9
  - 2.1.2 Single-Page Application (SPA) 12
- 2.2 Authentication 13
- 2.3 HTTP Basic Authentication 14
- 2.4 API Keys 15
- 2.5 Token-based Authentication 16
- 2.6 Session Management 22
- 2.7 Access Control 27
- 2.8 SSO Integration 29
- 2.9 Single Logout 30
- 2.10 Token storage 34
- 2.11 Software Development Kits 35
- 2.12 Credentials Manager 38
  - 2.12.1 Credentials Management API 40
  - 2.12.2 Web Authentication API 40
- 2.13 Authentication and Authorization in Native applications 43

3 Use Cases 46
- 3.1 Industrial Control System 46
  - 3.1.1 Architecture 46
- 3.2 Social Media Center 49
- 3.3 Component Library 54
  - 3.3.1 Proposal 55

4 Summary 57

References 59

Abbreviations

ACL	Access Control List
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CDN	Content Delivery Network
CORS	Cross Origin Resource Sharing
CRUD	Create Read Update Delete
CSS	Cascading Style Sheet
CTAP	Client to Authenticator Protocol
DOM	Document Object Model
FCP	First Contentful Paint
FIDO	Fast Identity Online
HMAC	Hash-based Message Authentication Code
HS256	HMAC with SHA-256
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IdP	Identity Provider
IoT	Internet of Things
IETF	Internet Engineering Task Force
JS	JavaScript
JSON	JavaScript Object Notation
JWT	JSON Web Token
JWKS	JSON Web Key Set
JWK	JSON Web Key
MD5	Message-Digest Algorithm 5
MVC	Model-View-Controller
NFC	Near Field Communication
NPM	Node Package Manager
OIDC	OpenID Connect
OP	OpenID Connect Provider
OS	Operating System
PaaS	Platform as a Service
PKCE	Proof Key for Code Exchange
RBAC	Role-based Access Control
REST	Representational State Transfer
RP	OpenID Connect Relying Party
RPC	Remote Procedure Call
RS256	RSA (Rivest–Shamir–Adleman) Signature with SHA-256
SDK	Software Development Kits
SEO	Search Engine Optimization
SET	Security Event Token
SHA256	Secure Hash Algorithm 256-bit
SPA	Single Page Application

SSL	Secure Socket Layer
SSO	Single Sign On
TLS	Transport Layer Security
TPM	Trusted Platform Module
UUID	Universally Unique Identifier
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
USB	Universal Serial Bus
UI	User Interface
W3C	World Wide Web Consortium
XHR	XMLHttpRequest
XML	Extensible Markup Language
XSS	Cross Site Scripting
iOS	iPhone Operating System
kB	kilobyte

1 Introduction

The 21st century saw widespread adoption and availability of internet since its introduction in the early 1980s. Since its foundation in 1994, the World Wide Web Consortium (W3C) has overseen the development and refinement of web standards and technologies that helped shape the internet into its current form. Primarily, the internet served as a medium of information that people could access from their personal computers. In the modern age, it has evolved into a medium that allows organisations to run businesses which customers can access through their mobile devices from anywhere.

Information on the internet is accessed through a web browser, which is an application that runs on the user's devices. Through a browser, the user navigates to a web site, which hosts the web pages. A web page is identified through a unique Uniform Resource Locator (URL). A URL identifies the domain of a web site and is distinct from other URLs. A web page is written in the Hypertext Markup Language (HTML). It is parsed by the browser and displayed according to the standards and rules that govern HTML. An HTML page is styled with Cascading Style Sheets (CSS), while interaction capabilities are provided by JavaScript (JS). During its early adoption, when these technologies were still at their nascent stages, the web pages were very minimalistic and contained minimal use of JavaScript. Most of the business logic of the web pages which are also referred to as web applications, were contained in the code that was hosted in the servers.

Since then, the capabilities of web browsers and the related technologies have grown tremendously. Particularly, the advancements in JavaScript allow web application developers to do a wide range of advanced operations. This led to a shift in the web development paradigm where a web page got divided into the front-end and back-end. The front-end represents the presentational part of a web page, which is displayed to the user in a browser, and the back-end represents the business logic, which is hidden away from the public. Furthermore, the capabilities of JavaScript for making asynchronous server calls changed how web applications are developed. It resulted in protocols such as REST and RPC APIs gaining mainstream adoption. Instead of making a server call for every page, the whole web application is loaded into the browser in the initial call. It resulted in the decrease of server calls and provided a much richer user experience where the interaction of the user with the web page is not blocked while the application loads data and resources asynchronously from the back-end server. This is corroborated by the usage of front-end libraries such as React, Angular and Vue, which are used by default when developing a web application in the present time.

Authentication and authorization operations have always been a part of the back-end in a web application. Since the back-end code is not available to the public, it is the most logical choice for such operations. Storing the user's information and secret securely was not possible in the browser, as anyone is able to see the code and data. With the introduction of token-based authentication and authorization, the possibility of moving those operations in to the web browser became feasible. Web browsers provided better security than before for storing the tokens and sensitive

information. Moreover, the security of the transport layer also increased significantly through the SSL/TLS protocol. This allows tighter de-coupling between the front-end application and the back-end servers. In some cases, it even allows an application to run without a back-end server, instead being statically hosted through a Content Delivery Network (CDN).

In this thesis, we explore and document the authentication and authorization methods that front-end developers should be aware of. We look at token based authentication and authorization approaches, particularly the OpenID Connect (OIDC) protocols. Finally, we document three use cases that the author worked on during his professional experience, which utilize the documented token based authentication and authorization methods.

2 Background

A conventional web application has two layers — the front-end and the back-end. Most web applications implement the business logic based on CRUD philosophy. This can be briefly described as:

- **C**reate — Creating data that is provided by the client.
- **R**ead — Read data that is stored in the data storage.
- **U**pdate — Update the stored data upon client input.
- **D**elete — Delete the stored data upon client input.

The front-end is the presentational layer of the web application that is responsible for displaying the user interface through which a user interacts with the web application. The user triggers events by interacting with the elements of the user interface. The front-end application is responsible for handling the events and passing them to the back-end using CRUD events.

The front-end communicates with the back-end through Application Programming Interfaces (API). The back-end typically exposes API end-points for a specific CRUD task, such as creating, or reading a data item. The front-end queries the end-points along with the data attributes, which are needed by the back-end to perform the specified action. Here, the process of authorization and authentication comes into the picture to protect the data and the services at the back-end. The web application needs to employ measures such that only known and authorized users are able to read, delete or make changes to the data. As the back-end reacts to the requests made by the front-end, the front-end must implement the processes and mechanisms through which a user can be authenticated and authorized with the back-end.

2.1 Single-Page Application and Traditional Web Application

Web applications are computer programs that are accessed with and run in a web browser. They use the Hypertext Markup Language (HTML) for presenting the content, Cascading style sheets (CSS) for styling the content and JavaScript (Js) for making the HTML elements interactive. Web applications are always hosted by a service called the back-end service, identified by a Uniform Resource Identifier (URI), which listens to requests, upon which the server delivers the web application resources to the requesting front-end application in the browser. Upon receiving the requested content from the back-end service, the browser parses the content and displays the webpage to the user in the browser. Considering this flow, web applications can be divided into two categories.

2.1.1 Traditional Web Application

A traditional web application is characterized by the back-end service processing and serving every page that is hosted by the web server. HTML documents provide

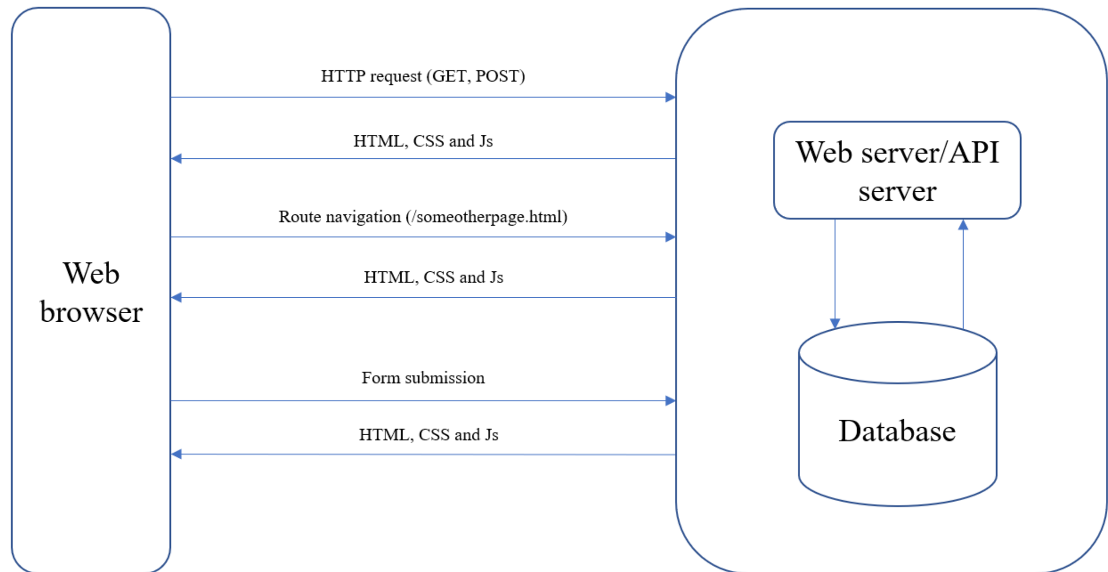

Figure 1: Traditional web application architecture.

hyperlinks (*a* tags), which are used to provide links to various pages of the web content. Every click to an *a* tag directs the browser to the back-end service, which processes the request, fetches and constructs the requested page, and sends it back to the web browser.

A traditional web application has several advantages which are listed below:

- The First Contentful Paint (FCP), which is defined as the time taken by the web browser to display the web page for the first time after requesting it from the back-end service, is low. This is because the back-end service only sends a small subset of the entire web application, which corresponds to the resources associated with the first page.

- It provides easy Search Engine Optimization (SEO), which helps web search engines to index the pages, thus providing better possibility for the search engine to list them when a user searches for any term relevant to the web application.

- It has a minimal set of requirements for the browsers to display the webpage. Especially when the browser has JavaScript turned off, a traditional web application can provide most of the core functionality.

Despite the advantages listed above, traditional web applications have taken the role of a legacy system. It was the primary approach to developing web applications

in older times. It has several disadvantages when compared to a SPA, which is the primary approach taken towards web application development in the modern times:

- It is network heavy, which means that every page requires a network request to the back-end service. In environments where network bandwidth is limited, it results in a decline in the user experience.
- It requires more effort in terms of security as every page in the web application has to be secured.
- It results in lower performance in terms of speed when compared to a SPA. The interaction time with the user also increases as the user has to wait for the browser to fetch the navigated web page every time a new page is requested.

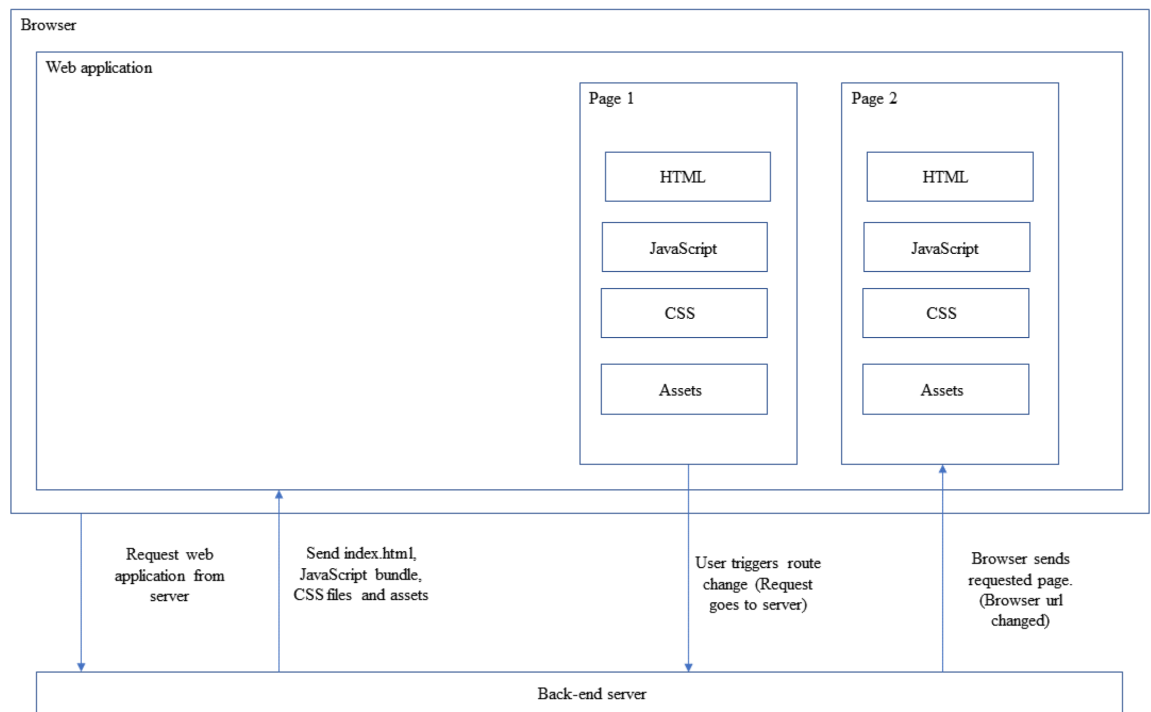

Figure 2: Traditional web application route change.

When the browser makes the initial request to the back-end server, the server responds with the *index.html* page and all the scripts, style sheets and the assets along with it. When the user click an *a* tag, the browser triggers a new request to the URL specified by the *a* tag's *link* attribute. If the *link* is absolute, then the exact URL is used for the request. If the *link* is relative, then the browser contruct's the full URL by taking the domain of the web page, and appending the *link* value to it. Page change route links are usually provided as relative URLs. Once the back-end server receives the request for the route change, it processes the request,

and contructs the new page, and sends the html page along with all the scripts, style sheets and other assets associated with the page. This operation is followed for every page to which the user navigates and after each form submission.

2.1.2 Single-Page Application (SPA)

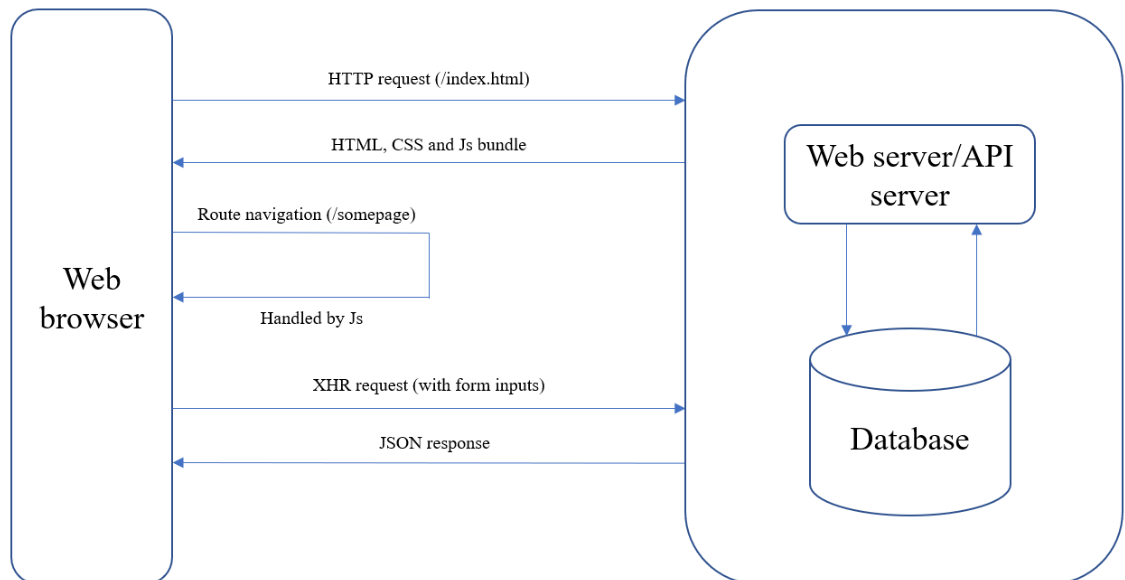

Figure 3: Single-page application architecture.

As the name suggests, an SPA consists of a single web page which is loaded only once by the web browser. The SPA architecture follows the Model-View-Controller (MVC) pattern. Views represent the pages or sections that are displayed to the user. Models represent the data that is associated with a View. Controller acts as the bridge between the Views and Models. An important characteristic of an SPA is the mapping of URIs to Views. The controller maps a URI with a particular View, which is displayed along with the associated data whenever it is requested. [1] [2]

SPA has several advantages when compared to traditional web applications, which are listed below.

- It provides faster performance and improved user experience. Since the whole web page is controlled by JavaScript, it provides faster response to user interactions and page loads.

- It requires less network bandwidth. The FCP is higher than a traditional web application, since the whole JavaScript bundle is loaded into the browser initially. However, subsequent traversal within the routes require less interaction

with the back-end service. The routes are handled by the Controller, and the data associated with a page within an active route is asynchronously fetched from the back-end service using XHR requests.

- It provides offine capabilities as the browser can cache the whole application.

Since SPAs can exist without a back-end server, web developers often take the approach of serving them through Content Delivery Networks (CDNs). SPAs take advantage of increased capability and browser support for Asynchronous JavaScript and XML (AJAX) calls for making server side calls from within the JavaScript. Furthermore, the development and wide adoption of Representation State Transfer (REST) APIs made SPAs an attractive option for web developers to build the web application with. The stateless nature of REST APIs allows decoupling between the web application and the API service, thus allowing them to perform their tasks independantly.

For complex applications, SPAs are often developed and served in conjunction with a back-end server within the same domain. The back-end server hosts the SPA and implements REST API routes, which are used to send and fetch data asynchronously from the SPA.

In the context of web application development, a front-end developer is primarily concerned with creating the user-facing side of the application. This is often called the public client, as all the code resides within the user browser and is available for anyone to inspect. A back-end application supports the front-end services by taking care of resource access and resource management, which are not visible to the end users. The front-end services and the back-end services can also exist at different end-points. The term *front-end server* is often used for a service that provides publically available URIs over a network that gives access to the SPA.

When the user triggers a route change, the JavaScript bundle handles the request, loads the requested page, and loads it in to the browser. It also updates the browser URL to reflect the route change. The route change does not trigger a new request to the server. The JavaScript bundle follows two different strategies to achieve this. One strategy is known as *Hash* strategy where the JS bundle uses the hash symbol in the URL to trigger the route change. By default, the browser does not send a new request to the server if any part of the URL string after the hash changes. For example, a route change to load a dashboard page changes the URL from *https://www.example.com* to *https://www.example.com/#/dashboard*. Another strategy is by using the newer HTML5 history APIs. By using the *pushstate* and *replacestate* functions, the JavaScript bundle is able to change the browser URL without triggering a new request to the server.

2.2 Authentication

Authentication refers to the process of identifying an entity in a secure way. This entity can be a user, a device or a service. Authentication in information systems is needed so that protected services and resources are not accessed by someone or something without proper authorization. In very simple terms, authentication

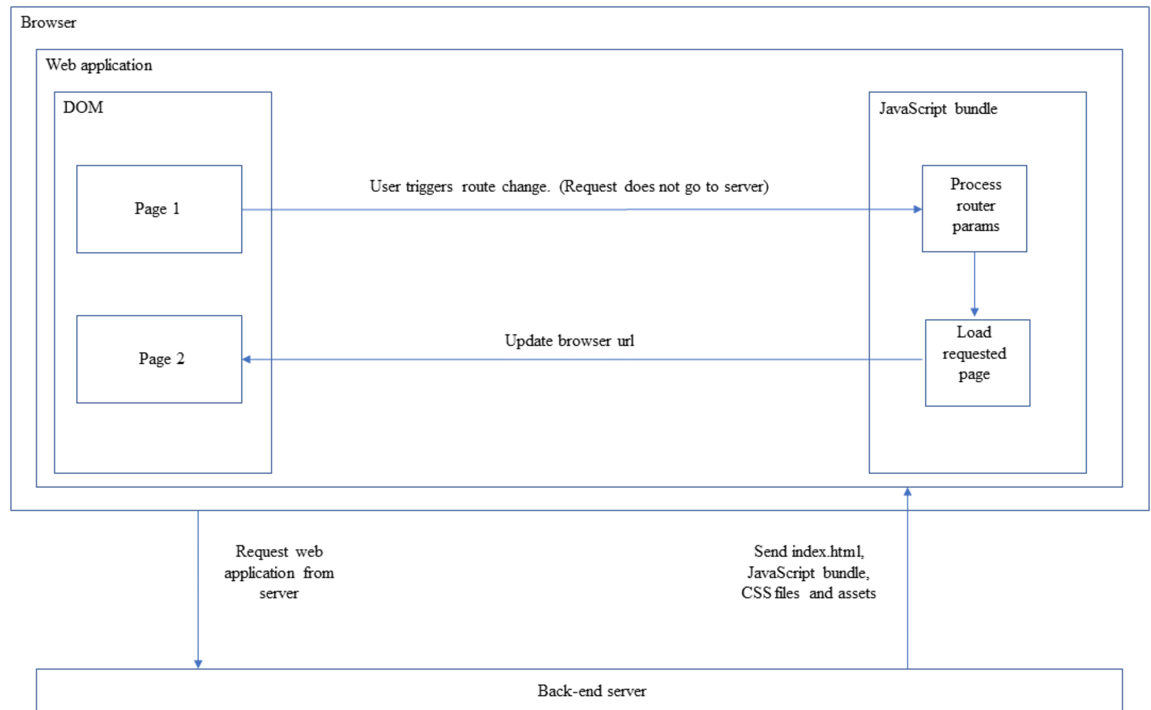

Figure 4: Single-page application route change.

answers the question "Who are you?" and authorization answers the question "Are you allowed to make that request or access that resource?". The process of answering these questions requires secure verification of the entity in question.

In a web application architecture, the role of authentication comes into play when a client or a user wants to access an API exposed by an API server. Below, we outline some of the common authentication mechanisms that are used for web applications.

2.3 HTTP Basic Authentication

The Internet Engineering Task Force (IETF) specifies several authentication and authorization mechanisms that web clients can employ. The most basic form of authenticating a user is by passing the user provided username and password in the HTTP Authorization header field. It is based upon a challenge-response mechanism where the web client responds to a challenge by the server when accessing an end-point. There are two schemes that can be applied in this form of authentication. The first scheme is called Basic scheme, in which the web client sends the username and password encoded in the base64 encoding after concatenating them in this format: username:password [3]. The second scheme is called the digest scheme where the client responds to the challenge by the web server in the form of a hashed string that consists of the username, password and a unique server nonce value [4]. Initially MD5 was used as the hashing algorithm, but later revisions of the specification allow the strong SHA-256 algorithm. Although the digest authentication offers better security

than basic authentication, it is not widely used in web client authentication.

Next, we will explain in more detail how the HTTP basic authentication is implemented. The server prompts the user to provide their authentication credentials. This is done with the HTTP header field *www-authenticate*. The values for this field are comma separated *name=value* pairs. One of the values that is identified in this header is the *realm*. The *realm* defines a space reserved within the server upon which the client request is authenticated. It identifies a space and scope within which the resources of the server are protected. A client wishing to authenticate itself to the server needs to provide the requested information in the HTTP *Authorization* header field. In this process, the user provides a plain-text username and password to the client application. The client application uses the credentials to contruct a base64-encoded string after concatenating the username and password with a colon in between. It is also possible for a proxy to relay the authentication challenge to the client by using the header *Proxy-Authenticate*. The client responds to this challenge with a *Proxy-authorization* field in the header. In both the request and response headers, the scheme for the authentication need to be specified. In this case, it is passed as *Basic*.

This authorization mechanism by itself does not inherently provide any security benefits and, thus, is vulnerable to security attacks. The base64 encoding of the credentials is sometimes mistaken for an encryption mechanism, but in reality, the code can be decoded back to obtain the plain-text values. Instead, it relies on the security and encryption of the channel through which the data is transmitted. One such way is by using HTTPS, which is the HTTP protocol encrypted with SSL/TLS, which has become the de-facto standard for secure web communication protocol.

2.4 API Keys

API keys are randomly generated and secure shared secrets between a client and an API producer. The clients can be individual users or projects consuming the API. When used for authenticating users, the API server generates a unique secret for each user. An API server in this context is the same entity as the back-end server. It handles the generation of API keys, storing them in the database, and checking every incoming API request for the correct API key. An API key should be random such that it cannot be guessed. A good approach is to use the user's Universally Unique Identifier (UUID), which is a random string generated by the application for every registered user. To prevent forgery of the key, it can optionally be signed with a shared secret key. The API server sends this generated API key to the user through an HTTP redirect after initial authentication or registration of the user. Depending on the implementation, the API keys can be sent to the client as a cookie or as a redirect parameter. If sent as a cookie, the secret is stored in the user's browser for the domain corresponding to the API server. Every subsequent request to the API server will include the cookie in its request parameters, and it is used by the API server to authenticate the user. When sent as a redirect parameter, the client application should store the API key in browser storage. From the front-end client application, the API key is passed with every request to the web API. The key can

be passed either as a request parameter within the contructed URL in a HTTP GET request, as a *x-api-key* header field in a HTTP POST request.

```
GET https://api.example.com?api_key=<API-KEY>

GET https://api.example.com
x-api-key: <API-KEY>
```

While API keys provide an easy way to authenticate clients and maintain their sessions, they are considered relatively insecure. Since it is a shared secret, exposure of the key can allow malicious users to impersonate the registered user. Alternatively, API keys are commonly used to identify projects rather than individual users. In this scenario, a front-end service exists between the client devices and the API server, which acts as a proxy between the two. Since the front-end server cannot be accessed by the users, it can securely store the API key. It adds the API key for every forwarded request to the API server and relays the response back to the client. The front-end server maintains its own implementation of authenticating and identifying individual users.

When distributing API keys by API servers to web developers, the web developer is asked to register the application, upon which they receive a unique API key. This API key is shown with a message that it will be shown only once, and the web developer should save it securely. A common mistake that a web developer does is embedding the API key into the code itself on the front-end server. This will result in the API key getting exposed to the public, if the code is publicly shared in a code hosting repository. Instead, a better approach is to use environment variables to store the API keys, and using the environment variables in the code. One of the most popular approach for hosting a web application is to use a cloud provider's Platform as a Service (PaaS). Most PaaS providers come with inbuilt environment variable configuration, which allows the web developer to define the API keys in the cloud provider dashboard itself. Naturally, an even bigger mistake would be to embed the application's API key into the client-side JavaScript or to otherwise expose it to the web browser. [5] [6]

2.5 Token-based Authentication

Token-based authentication is another solution, which has gained popularity in recent years. The previously discussed authentication mechanisms follow a stateful approach, which results in issues with scalability and heterogeneity. Token-based authentication allows a stateless authentication flow where the server does not have to maintain any state related to the client.

In the token-based authentication approach, the front-end application connects to a separate Identity Provider (IdP) by redirecting the user to a user interface provided by the IdP. The IdP authenticates the user, typically with a username and password. Additionally, the IdP can employ an extra layer of identification, such as a one-time password sent to the user's registered mobile device. Once authenticated, the IdP sends an ID token, which claims the identity of the user, back to the client. The

IdP also sends an access token to the client for accessing the protected APIs. The front-end application needs to store the access token so that it can be used for future access to protected resources.

OpenID Connect (OIDC) is a standardized protocol that is used to implement token-based authentication [7]. It is built on top of OAuth 2.0 specifications that defines the steps to obtain the ID token. OIDC defines the additional *openid* scope value which must be sent in the authentication request [8]. The authorization server returns the ID token in the form of a JWT. The ID token primarily serves as an assertion for the authenticated state of the user. The ID token contains the identity information about the authenticated user in the form of *claims*. OIDC defines a set of rules for validating the ID token before the information in it can be trusted and used. It is important to note that the *claims* that are contained in an ID token can be accessed through the */userinfo* end-point of an authorization server [7]. The OpenID client must present the access token to obtain the information from this end-point. ID tokens are useful because it represents the current logged in status of the user through a number of additional claims, such as the issuer identifier, the subject identifier, the audience who the ID token is intended for, the expiration time, and the end-user authentication time. Another important authentication request parameter is the *response_type* which determines the tokens that will be returned. The *response_type* should contain the value *id_token* if an ID token is needed. [9]

OIDC specifies two approaches based on the type of the application. In an architecture where most of the application logic is handled in the server, such as a traditional web app executing in a server or a single-page application that is backed by a back-end middleware, OIDC specifies the authorization code flow. In this approach, most of the authentication logic is handled in the back-end middleware or server. It is the OIDC client in this case. In a high level overview, the OIDC client receives an authorization code in response to an authentication request sent to the IdP's authentication end-point. The OIDC client then uses this authorization code to get the ID token and access token from a token end-point. This flow is defined in the OAuth 2.0 specification [8]. The authorization server must always return an access token. OIDC builds on top of this by requiring the authorization server to send the ID token along with the access token. [10]

Based on the architecture of the application, the necessity of the access token may differ. For example, for an application that only uses an authorization server for identity assertion of its end user, the *claims* present in the ID token is sufficient. The application may not need an access token as it is not accessing the user's resources from the resource server. The access token will still be issued if authorization code flow is used and it can be used only for the scopes that were granted. In this case, the access token can access the */userinfo* end-point to get information about the same claims that is present in the ID token.

The Authorization code flow steps are detailed as follows [7]:

- The client initiates the authentication with the IdP by a user interaction, such as clicking a button.
- The client invokes the */authentication* end-point at the IdP. OIDC specifications

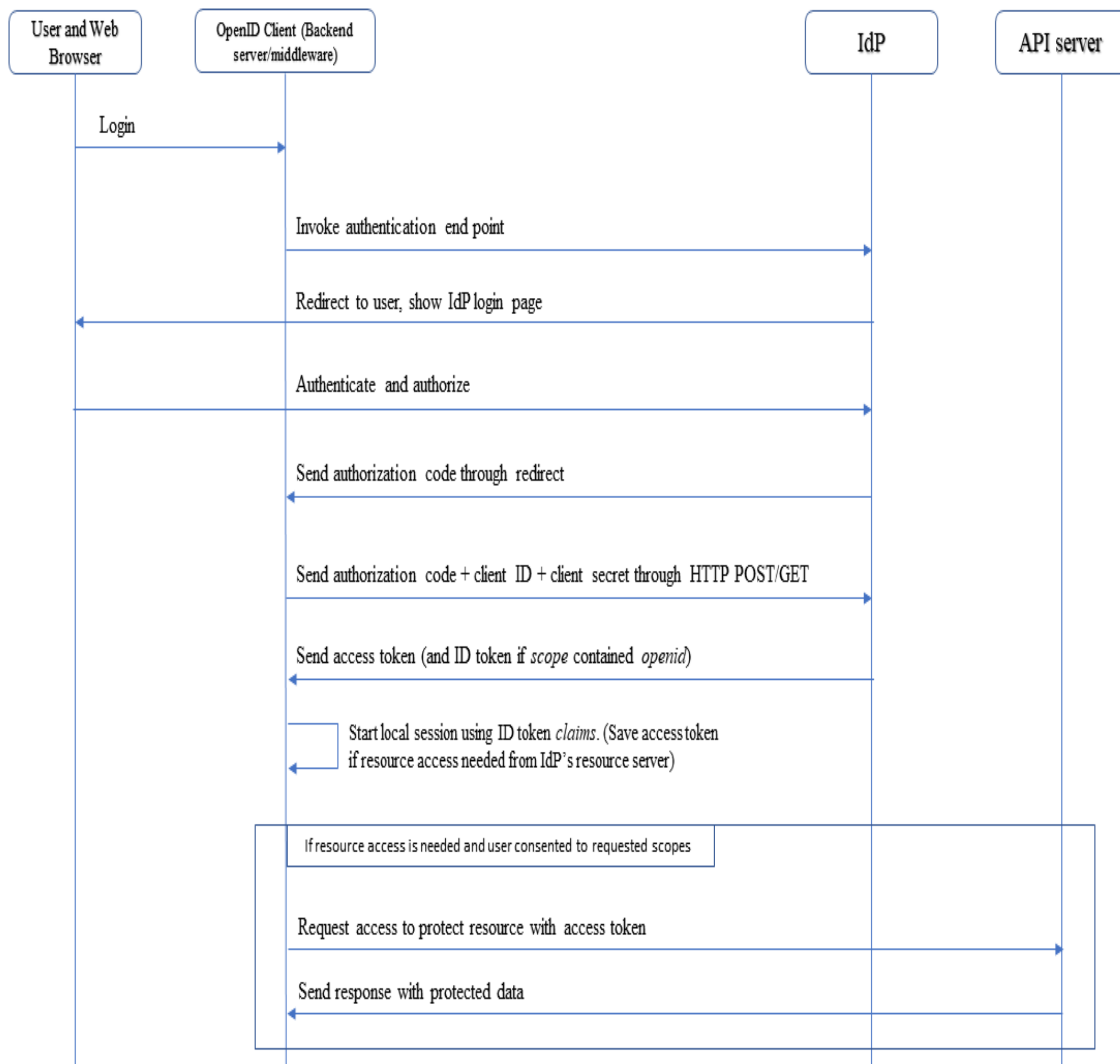

Figure 5: Authorization code flow.

requires the *scope* value to contain *openid* if an ID token is required. The *response_type* in this flow should be *code*.

- The IdP redirects the user's browser to the IdP login page.

```
HTTP/1.1 302 Found
Location: https://example.idp.com/login
    ?response_type=code
    &scope=openid
    &client_id=<CLIENT-ID>
    &state=<RANDOMLY-GENERATED-STATE-VALUE>
    &redirect_uri=https%3A%2F%2Fexample.client.com%2Fcallback
```

The user is presented with a screen where they authenticate themselves with their credentials. Furthermore, the page also presents a list of actions and resources that the client asks access to. The user must consent to the requests before the IdP can grant the access token to the client.

- The IdP redirects back to the *callback* URL that the client registered with the IdP. The IdP sends a one-time authorization code as a parameter of this redirect.

```
HTTP/1.1 302 Found
Location: https://example.client.com/callback
    ?code=<AUTHORIZATION-CODE>
    &state=<SAME-STATE-VALUE-THAT-THE-CLIENT-SENT-BEFORE>
```

- The client processes the redirect request and sends an HTTP POST request to the IdP to its *token* end-point. The authorization code, client ID and client secret is sent as parameters to this request.

```
POST /token HTTP/1.1
Host: example.idp.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic <CLIENT-ID-AND-SECRET>

    grant_type=authorization_code
    &code=<AUTHORIZATION-CODE>
    &redirect_uri=https%3A%2F%2Fexample.client.com%2Fcallback
```

- The IdP processes the request, verifies the authorization code, client ID and client secret, and replies back to the request with the ID and access token. It also sends the refresh token in this step if it was requested by the client.

- The client processes the reply, parses the ID token and creates a new session for the user with the information from the ID token and redirects the user to the application's dashboard page. If the client requires access to protected resources, it saves the access token.

An important step in this flow is client authentication with the *token* end-point. OIDC defines a few methods that should be supported by the IdP. In each one of them, the client sends different information to the IdP in the client secret field mentioned above. They are detailed as follows [7]:

- *client_secret_basic* — In this method, the client uses HTTP Basic Authentication mechanism to authenticate to the IdP *token* end-point. The client must send the *client_secret* that it received from the IdP when registering with the IdP.

- *client_secret_post* — This method is the same as *client_secret_basic* except that the *client_id* and *client_secret* is sent as request parameters in the request body in a HTTP POST request.
- *client_secret_jwt* — In this method, the client creates a Hashed Message Authentication Code (HMAC) using an algorithm such as HMAC SHA-256. The HMAC is created using the *client_secret* that it receives from the IdP. The client is required to include the following claims in the JWT: *iss*, *sub*, *aud*, *jti* and *exp*. The *iss* (issuer) and *sub* (subject) claims should be the *client_id* of the client. The *aud* (audience) claim should be the URL of the token end-point of the IdP, which identifies the IdP as the audience for this JWT. The *jti* (JWT ID) should be a unique identifier for the token such that this token is used only once. The *exp* (expiry) is the expiration time which states the time after which the token should not be used. The client should send this token as a *client_assertion* parameter in the request body, along with a *client_assertion_type* parameter whose value should always be *urn:ietf:params:oauth:client-assertion-type:jwt-bearer*.
- *private_key_jwt* — This method follows the same principle as *client_secret_-jwt*. The only difference is that instead of creating an HMAC, the client signs the JWT using its private key. The client should have registered a public key with the IdP.
- *none* — In this method, the client does not authenticate itself against the *token* end-point.

OpenID connect also specifies the Implicit code flow, which is specifically catered to web apps running in the browser. Single-page applications belong to this category. In the Implicit code flow, the ID token is directly sent back to the client from the IdP without any authorization code. The *response_type* in this flow should be *id_token*. If the access token is desired alongside the ID token, then the *response_type* should be *id_token token*. Since HTTP is a stateless protocol, the IdP has to send the tokens back to the web app through an HTTP redirect. This exposes the ID and access token through the browser history, which can be snooped by malicious programs in the form of plugins in the user agent. ID tokens themselves are not used for accessing any protected resources or for authentication. They merely assert the authentication status of the end-user, and contains the end-user's information. So, the risk involved with its leak is lower than the access token leak. For this reason, the Implicit flow is not recommended if the client needs the access token along with the ID token. This risk is also present when using Authorization code flow in a browser based app. The *client secret* cannot be securely stored because the browser code and data can be viewed by everyone.

To avoid these risks, IETF introduced the authorization code flow along with Proof Key for Code Exchange (PKCE) for browser-based applications and native applications that cannot obtain and store a client secret securely [11]. The PKCE plays the role of the client secret in the authorization code flow. Originally it is

a part of the OAuth 2.0 specification which, by extension, gets applied to OIDC specification. If the initial authentication request does not contain the *openid* value in the *scope* parameter, then only the access code is returned. However, if the *openid* value is added to the *scope* parameter, then the ID token is returned along with the access token. To implement authentication with this approach, the front-end application has to follow the steps below [11]:

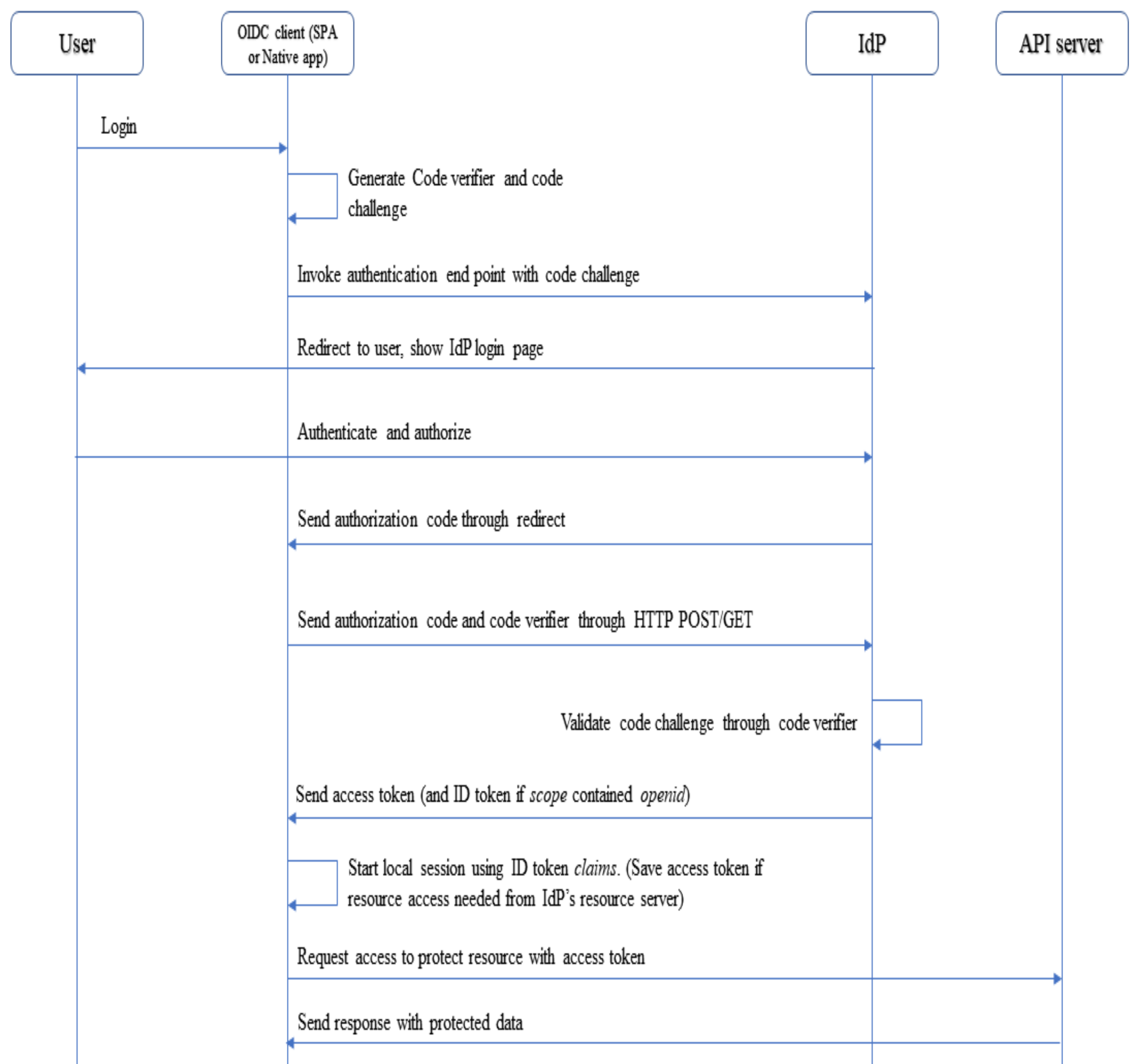

Figure 6: Authorization code flow with PKCE.

- The application initiates the authentication with the IdP by a user action such as clicking a button.
- The application generates two artifacts *code_verifier* and *code_challenge*. The *code_verifier* is a randomly generated 128 bit string consisting of alphanumeric characters along with a few special characters. The *code_challenge* is generated

from the *code_verifier* by encrypting it with a cryptographic hash algorithm, typically SHA256, and then encoding it in base64URL format.

- The application sends the *code_challenge* along with a *redirect_uri* to the authentication server in an authentication request using HTTP redirect. *scope* should contain *openid* as one of its values if an ID token is needed. The *response_type* should be *code*.

- The authentication server authenticates the user, typically through a username and password. After the user is authenticated, it prompts the user for consent to grant access to the user's resources to the requesting client application.

- Once the consent is given, the authorization server saves the *code_challenge* and sends a cryptographically generated random authorization code to the client application using the *redirect_uri*.

- The client application gets the authorization code and sends a POST request to the authorization server requesting the ID token and access token. The client needs to send the *code_verifier* and the authorization code in this request.

- The server verifies the *code_verifier* with the *code_challenge* that it stored in the earlier request. Additionally, it verifies the authorization code. If the verification succeeds, the server replies to the POST request with the ID token and the access token, and the server also deletes the *code_verifier* as used.

- The client application parses the ID token and creates a personalized dashboard with the *claims* information. It stores the access token in its internal storage for use during future resource access requests.

2.6 Session Management

The ID token and access token that are issued by the Identity provider are often short-lived due to security considerations. This creates a situation where the client needs to request for new tokens from the authorization server once the existing tokens expire. The OIDC client should handle the sessions for both the ID token and the access token. The tokens may have different expiry times. It is important to distinguish and identify the uses of the two tokens for the application's use cases. The audience of an ID token is the OIDC client and the audience of an access token is the resource server. An ID token gives to the client information about the user's identity and authentication status. This information is provided in the *claims* field of the ID token payload. This token is used to generate a custom profile with the *claims* fields. Access tokens, on the other hand, are used by the client to access resources on behalf of the user. OIDC specification defines user-sessions as a continuous period of time during which the OIDC client has a valid authenticated status with the IdP. User-session management relates to checking this authenticated status. If a session expires, all existing tokens are invalidated, and new tokens are issued after the user

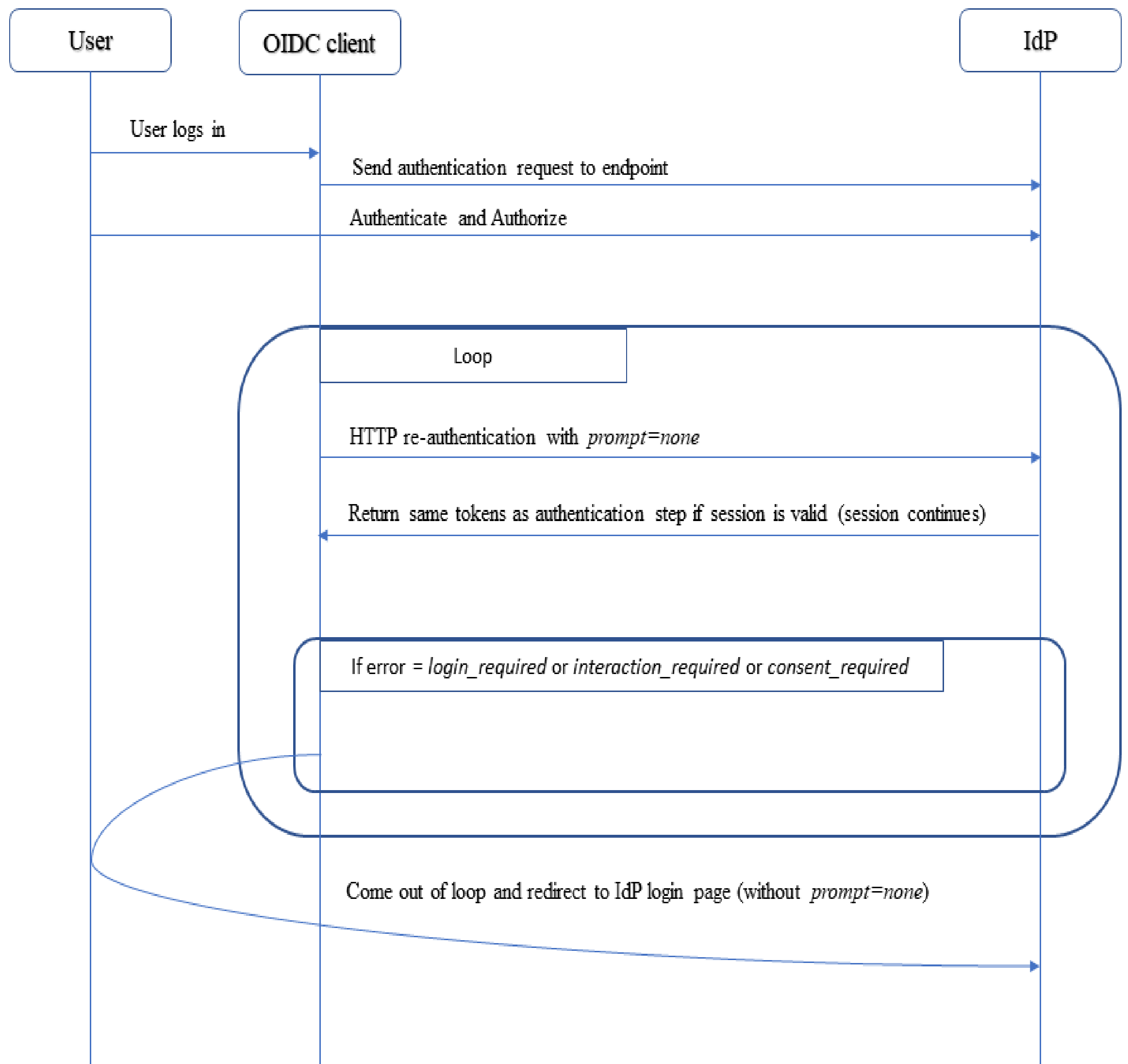

Figure 7: Session management using silent re-authentication.

re-authenticates. This includes the access token even if the access token was still valid according to its expiration time.

There are two primary approaches to maintaining a session. One approach is called silent re-authentication [7]. The web application follows the usual authentication flow providing the same parameters to the IdP authorization end-point. This approach requires the client to periodically query the OpenID provider about the end user session status. For example, in OpenID Connect, the *prompt=none* field in the request tells the identity provider that the authentication should not ask for end user interaction. If the session is still valid in the IdP, the IdP will reply with the requested tokens. Additionally, OIDC defines error response codes that the IdP should include in the response parameter *error_description* if the silent authentication did not succeed. These error responses are *login_required*, *interaction_required* and

consent_required.

- *login_required* — This code is sent when the user is not logged in with the IdP.
- *interaction_required* — This code is sent when the user is logged in with the IdP and has authorized the application with the IdP, but the IdP needs to perform some additional interaction with the user before refreshing the ID token.
- *consent_required* — This code is sent when the user is logged in with the IdP but the IdP requires end user consent before granting the application access to the user's identity and access to resources.

The OIDC client should follow the above mentioned responses with a redirect to the IdP's authorization endpoint without the *prompt=none* because the IdP requires user interaction to complete the authentication.

A second approach is to use hidden iframes connected to the IdP, which is polled by the web-client at regular intervals to get the session status [12]. This is done through cross origin communication between the OIDC client and the IdP. The OIDC client loads an invisible iframe with a URL source provided by the IdP. This allows the IdP to run in its own security context within the OIDC client. The OIDC client uses the *Window.postMessage* API to send messages to the IdP iframe, which uses the same API to reply back to the OIDC client. Messages within each iframe are received with a *message* event listener, which is triggered when the iframe receives a message through the *postMessage* API. OIDC defines two metadata endpoints in the form of URLs that the IdP should implement to support this functionality. The first URL, *check_session_iframe*, is provided as a source to the invisible iframe that loads the IdP implemented page into the iframe. The second URL, *end_session_endpoint*, is provided by the IdP so that the OIDC client can redirect to it when it wants to end a session such as when the user wants to log out.

To keep track of the session state, OIDC defines a parameter called the *session_-state*. It is calculated from the *client_id*, *origin URL*, *browser state* and a salt, which are then hashed cryptographically. It is calculated by the IdP and sent to the OIDC client along with the initial authentication response. The *browser state* is maintained by the IdP and stored at the OIDC client in the form of a cookie or in the browser local storage. It is IdP-implementation dependant which is updated during meaningful events such as login, logout etc. It is not recommended for the IdP to change the *browser state* too often, as it will trigger a change in *session_state* and additional network traffic to reflect the change at the OIDC client. It is stored under the IdP domain so that it is not accessible outside its domain. The IdP accesses it through the page that is run in the invisible iframe under its own domain. The *origin URL* refers to the origin URL of the initial authentication response. This is typically a reference to the IdP's origin URL. [12]

The OIDC client loads an invisible iframe with a URL from its own domain to communicate with the IdP iframe. This iframe sends the *client_id* and the *session state* with each session check request made to the IdP iframe. The OIDC client

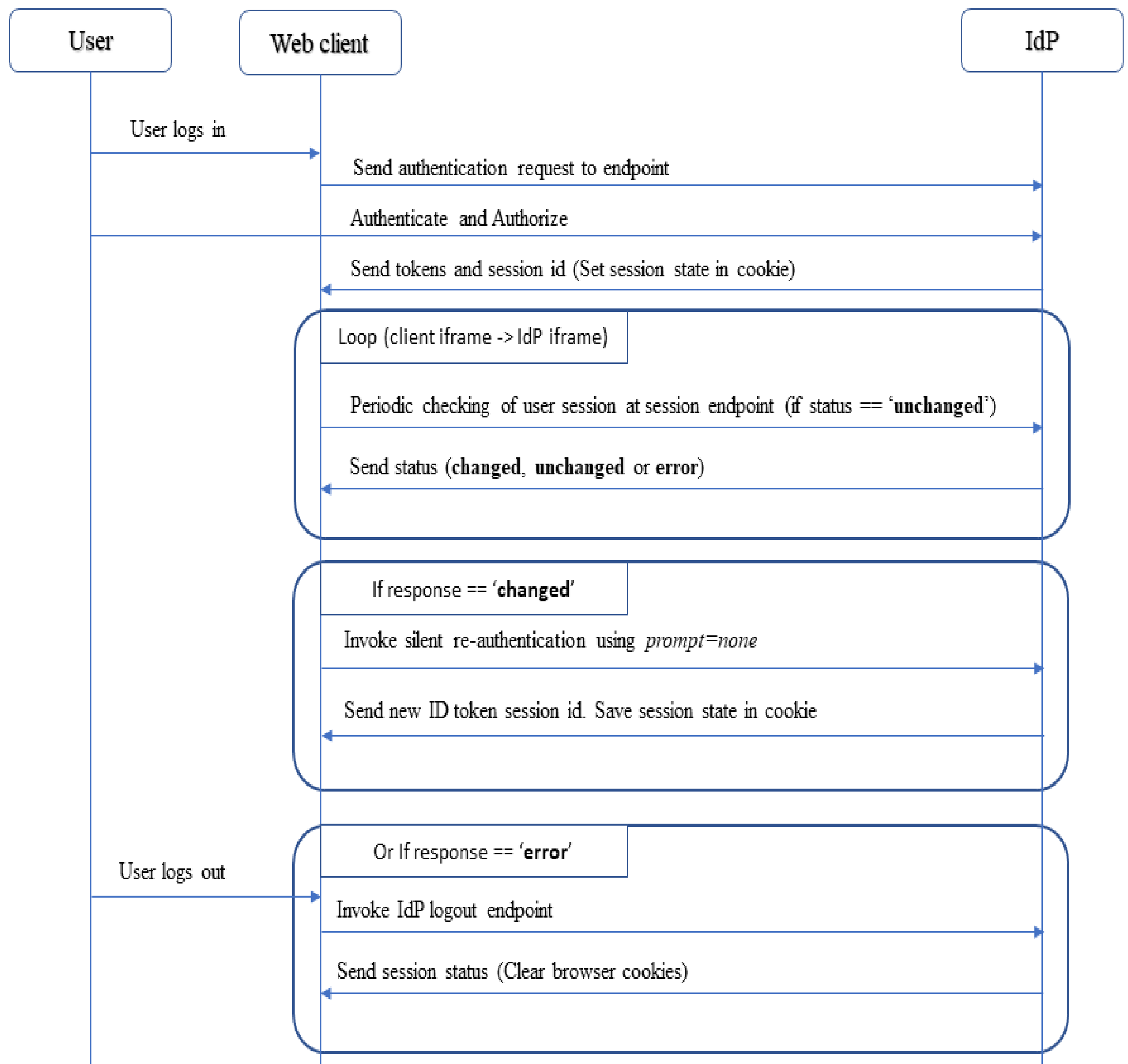

Figure 8: Session management using hidden iframes.

iframe should know the *ID* of the IdP iframe to send the request. The IdP utilizes the *client_id* sent by the OIDC client iframe, the origin URL of the IdP, and the *browser state* stored in the browser to recalculate the *session state*. Then it matches it against the *session state* passed by the OIDC client. Based on the match, the IdP replies with three possible status values.

- *changed* — If the two *session states* do not match, the IdP sends this response. Upon receiving the response, the OIDC client iframe should initiate a silent re-authentication with the IdP using *prompt=none*. The OIDC client iframe needs to send the old ID token as a *id_token_hint* parameter in the request. If the IdP responds back with an ID token for the same user, then the OIDC client stores the ID token and updates the *session state* value received along with the response. If the OIDC client does not receive an ID token or receives

an ID token for a different user, then it indicates the end of a session, and the OIDC client should handle it as a logout for the current user.

- *unchanged* — If the two *sessions states* match, the IdP replies with this response. The OIDC client does not need to take any further action in this case, as it indicates that the user session is still valid.
- *error* — If the IdP cannot calculate the *session state* from the parameters received through the OIDC client request, it replies with this response. The web application should handle this case as a logout for the user session. Upon receiving this error response, OIDC mandates that the OIDC client should not perform a re-authentication with *prompt=none* parameter so that it does not cause potential infinite loop requests with the IdP.

To end the session with the IdP during a user log out, the OIDC client redirects the browser to *end_session_endpoint* URL. The OIDC client is required to pass the current ID token as an *id_token_hint* parameter. Optionally, the OIDC client can also pass a post logout redirect URL as a *post_logout_redirect_uri*. If provided, the IdP redirects the browser to this URL after logging the user out. The URL should be registered with the IdP before it can be used.

The OAuth 2.0 specification introduced refresh tokens, which can be used to request the IdP for a new access token once the current one expires [8]. Since the OpenID Connect specification was built upon the OAuth 2.0 specification, it introduced the *openid* scope parameter, which must be sent if an ID token is needed along with the access token. This scope parameter can be sent both in the initial authentication request and when using the refresh token. Refresh tokens are often long lived, which allows the web application to maintain an active session without forcing the user to go through the authentication and authorization flow all over again.

Since refresh tokens can be used to directly acquire access tokens, it is of paramount importance that they stored securely. In a public client such as a single-sage application, the refresh token can be stored in the browser localStorage or in cookies. Furthermore, refresh tokens are granted for a single use, which means a new refresh token is sent by the authorization server along with the access token. This still does not prevent malicious programs in the browser from getting access to the refresh token, paricularly if stored in the localStorage.

To maintain the user session, the web application needs to request new tokens before the existing tokens expire. All tokens are opaque to the web application, which means the web application cannot read the access token and get the expiration details from it. Authorization servers are often configured to send the parameter *expires_in*, which contains the amount of time after which the tokens would expire. The web application needs to read and store this value and implement an automatic request well before the tokens expire.

2.7 Access Control

In a traditional web application, access control is generally a part of the back-end where every route checks whether the user is allowed to access that route. However, in SPAs, where the whole application runs in the user's browser, implementing access control strategy in the front-end comes into the picture as an important task.

From a security standpoint, access control in a browser-based application relates to simplifying and enhancing the user's experience while using the application. Since browser-based applications can be inspected and modified, the control checks must be performed in the back-end. Implementing redundant access control checks in the front-end has several advantages as detailed below:

- It prevents unwanted asynchronous AJAX calls to the server. A web application loads certain parts of the user interface and populates it with data asynchronously. The application is optimized for performance by avoiding server calls for data that the user does not have access to.
- It prevents displaying UI elements for performing operations that the user is not authorized to. The web application performs better through enhanced page load and response times, and provides coherence in user experience by removing points of error.
- It enhances the user experience by displaying proper messages to the users when performing unauthorized actions.

The most commonly used access control method in web applications is Role-Based Access Control (RBAC). While configuring authorization servers, each user is assigned a Role, Group or Permission. A user can be assigned a role, which defines a specific set of permissions that the user is allowed to do. For example, a user with the role *Admin* can have the permission to ADD or DELETE users. However, a user with the role *Visitor* may only have the READ permission. Users can also be assigned to groups, where each group has a set of assigned permissions that its members are allowed to do.

Access control can be implemented using custom scopes, which refer to the allowed actions against a particular resource. They are typically represented in the form *resource:action*. When requesting access tokens from the authorization server, the front-end application has to include in the request a *scope* parameter, which is a space-separated list of strings. The authorization server replies back with the allowed permissions in the *scope* parameter in the access token. Furthermore, the authorization server can also display a UI prompt to the user where the user can consent or dissent to the permissions requested by the client application. Based on the user action, the authorization server can modify the set of permissions before sending the list of scopes back to the web application. The list of defined custom scopes is decided by the API developer. They are usually provided to the application developer through the API documentations.

In SPAs, routes and UI elements should be protected based on the permissions obtained in the scope. This can be achieved by maintaining a list of all roles and

allowed permissions in the web application. Based on the obtained scope, the front-end application can implement route guards that prevent a user from navigating to the protected route. For example, a user with the *Visitor* role should not be able to access the */admin* route. Furthermore, UI elements should be disabled or hidden from certain common pages based on the user roles. For example, in a *Dashboard* page, there can be a *Add user* button, which should be visible only to users in the *Admin* role. [13]

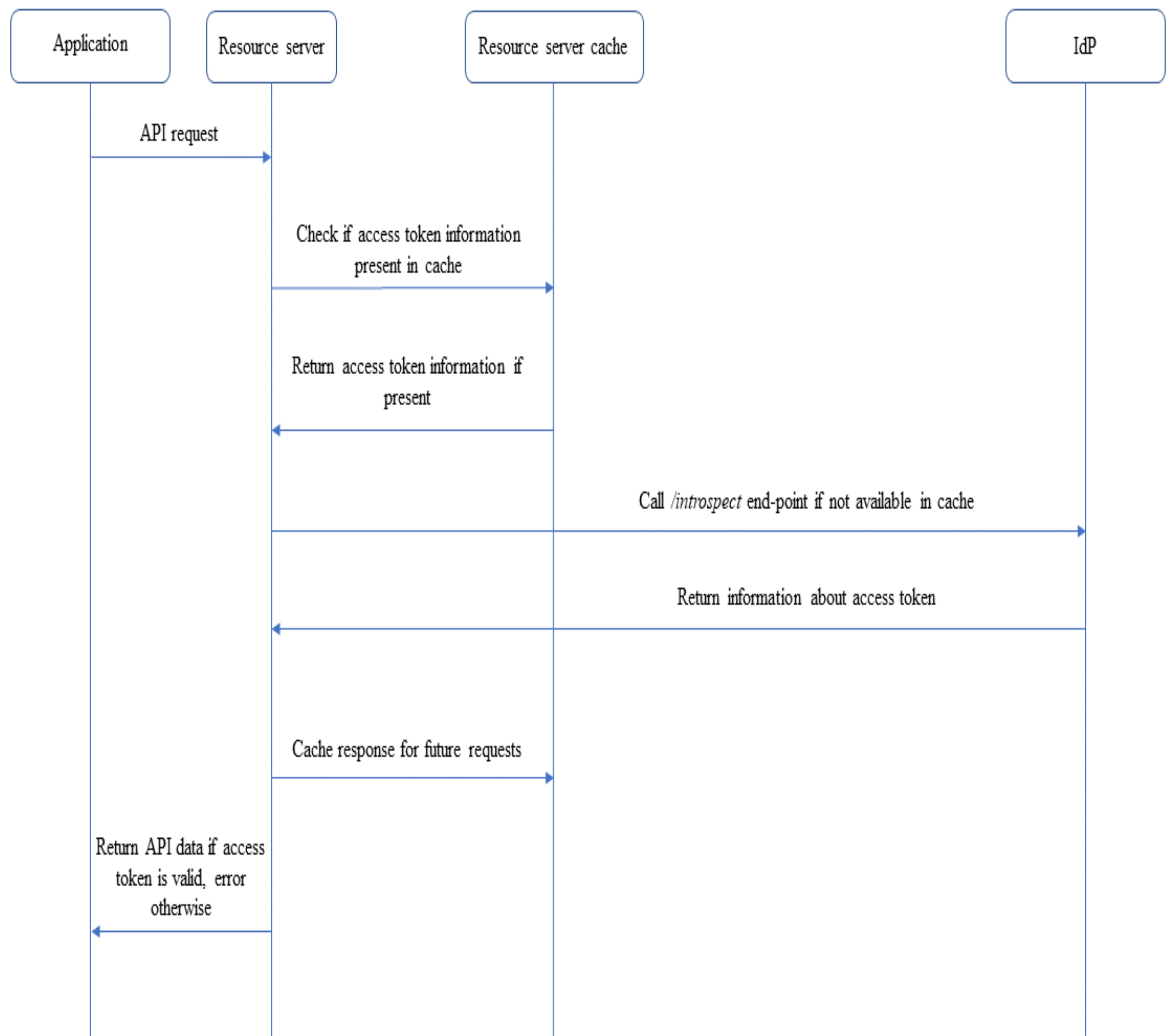

Figure 9: Access token validation check.

Every access request is checked in the resource server for validation. This is necessary because a malicious user can bypass the access control checks in the front-end web application. Typically, resource servers communicate with authorization servers for validating an access request. Authorization servers maintain Access Control Lists (ACL) which contains information about the users along with their

roles and permissions. OAuth 2.0 specifies a */introspect* end-point that resource servers can use to validate an access token [14]. It is recommended to not expose this end-point publically as it may return sensitive information about the client. A common approach is to use it in a private server which only the resource server can access. Additionally, it is recommended that this end-point is protected by an authentication mechanism such as HTTP Basic authentication. This request returns a JSON object that contains information about the access token and the client. The *active* field states if the access token is valid. Based on the implementation, *client id*, *scope*, *username* and *exp* fields are also returned. The resource server checks the *scope* field to determine if the access token has the permission to access the requested resource. If permission is not granted, then it returns an *unauthorized* error response to the client. The resource server can cache the values of the */introspect* end-point for performance considerations. The cache expiry times are determined by security and performance trade-offs. Shorter expiry times will result in better security. Higher expiry times may result in situations where the resource server validates an expired token.

2.8 SSO Integration

Single sign on (SSO) allows a user to use the same credentials in a number of applications within an organisation. User's credentials are registered and stored in the organisation's IdP. Every application within the organisation share a trust relationship with the IdP. For every application, the users are redirected to the IdP's login page where they authenticate themselves. After authentication, the IdP shares the authentication status with the application. With OIDC, the authentication status is shared through the ID token. [15] [16]

JWT is most commonly used for the token. The payload field of the ID token contains a number of claims that represent information about the user. This information is used by the front-end client application to create a personalized experience for the user. Before accessing the payload, the front-end application needs to decode the ID token and verify the signature of the JWT. According to the standards, the JWT payload is encoded in base64URL format [17]. Thus, the raw payload information can be found by just decoding the payload in base64URL.

Secondly, it is important to verify that the received token is legitimate. There are various steps that need to be taken by the front-end application to verify the authenticity of the token. The first step is to verify the signature of the JWT. The IETF specification supports various algorithms for signing the JWT [17]. Out of them, the most used ones are HS256 (symmetric) and RS256 (asymmetric). The algorithm used by the IdP to sign the JWT is included in the *alg* header field of the JWT. For public facing applications such as SPAs, HS256 is not applicable since it uses a shared secret key, which is used both for signing and verification. Public facing applications have no way to store this secret key in a secure manner. This option is applicable if the application architecture uses a back-end proxy API server that handles the tokens on behalf of the SPA client. RS256 is used instead for public facing SPA clients, and it uses asymmetric cryptography. The IdP server signs the

JWT using its private key before sending it to the client. The client uses the public key of the IdP server for verifying the signature.

To get the public keys, the IdP must expose an endpoint that provides the caller configuration information about the IdP. The OIDC specification defines this endpoint as */.well-known/OpenID-configuration* concatenated to the IdP host address [18] [19]. For example, if the host address of the IdP is *https://www.example.com*, then the configuration information URL is *https://www.example.com/.well-known/OpenID-configuration*. The JSON object received as a response to this API contains a *jwks_uri* field which points to the location of the IdP's JSON Web Key Set object (JWKS). The JWKS contains an array of JSON Web Keys (JWK), which represent the cryptographic keys used by the IdP to sign the JWTs [20]. Each JWK contains a *kid* and the public key amongst other fields. The client matches the *kid* value with the one that it receives in the JWT token and uses the public key for the verification process.

Apart from the signature, three other important fields need to be verified which are the Issuer *iss*, the Subject *sub* and the Audience *aud*, which are included in the token payload. The issuer field represents the IdP that issues the token. The subject field is a unique identifier for the end-user within the IdP (OIDC Provider), who is being authenticated. The audience is the entity who is expected to receive and verify the token. The OIDC specification state that the audience field must contain the client ID of the OIDC client. The OIDC client, in this case, is the back-end server or front-end application, which receives the tokens from the IdP. The client ID is issued to an OIDC client when it is registered with the IdP (OIDC provider). The audience field also allows custom values which are application specific and should be verified if required by the application.

Once verified, the front-end application can use the subject identity (*sub*) and other claims that are contained inside the JWT payload. The OpenID connect specification define a number of standard claims related to the identity of the user. These values can be used by the front-end application to create a local profile. To receive the claims in the ID token, the application needs to include the appropriate *scope* values while making the authentication request. An example is *scope=openid profile email address phone*. The list of scopes is space delimited. The *openid* scope value states that the client wants to use the OpenID protocol for authentication and expects an ID token in the response. The *profile* scope represents that the client application wants access to the public profile of the user, which includes the name, gender, picture and birthday. The front-end developer should only ask for the minimum necessary set of claims. The user is asked for their consent when they authenticate themselves with the IdP whether they allow the requesting web application to access their data. Hence, the requested scope may not fully reflect the claims that are actually returned. [21]

2.9 Single Logout

OIDC provides specifications that support single log out. The principal idea behind single log out is to end the user's session for all web applications to which the user

is logged in with the same IdP. It allows the ease of logging out once, instead of logging out of each web application individually.

When logging a user out of a web application, the application has to handle a few scenarios to ensure a successful log out. First, it needs to clear session variables from its own security context that it maintains for the web application. Then, it must perform a logout with the IdP, which clears the session from the IdP. Depending upon the use cases, the web application can perform a local logout or a global logout. In the local logout, the web application logs the user out of the currently active application that initiated the logout. In a global logout, the IdP triggers the logout of all applications that have an active session for the user.

For local logout, the OIDC client can implement a simple logout mechanism based on timeout logic. Every ID token contains an expiration field that specifies the time until when the ID token is valid. The OIDC client starts an internal timer, which triggers a logout when the ID token expiration time is reached. This mechanism will not work, however, if the OIDC client uses refresh tokens to persist the user session or uses periodic silent authentication to keep the user logged in.

The OIDC session management spec describes how a web application can perform a local logout [12]. The *Session management* section describes how a OIDC client maintains a session with the IdP. For a OIDC-client-initiated logout, the client first clears its own security session with the user. Then, it redirects the browser to an IdP-defined endpoint, which clears the IdP-maintained session for the user. The IdP needs to implement a service discovery endpoint, as specified by OIDC, through which the OIDC clients can find its logout URL. With the redirect request, the client sends the *id_token_hint*, *post_logout_redirect_uri* and *state* parameters. At the logout endpoint, the IdP should ask the user whether they want to log out of the IdP as well. This triggers a global logout depending on the choice by the user. For a IdP initiated logout, the OIDC client should periodically query the IdP about the session status for the user. This is done using two invisible iframes, one created by the OIDC client on from its own domain with a page containing session management functions, and one created by the OIDC client with the session URL provided by the IdP. The two iframes communicate with each other through the *window.postMessage* API. For every request, the IdP responds with one of the status values *changed*, *unchanged* or *error*. If the status is *changed*, the RP should initiate a silent re-authentication with the IdP. If the IdP responds with no ID token or an ID token for a different user, the OIDC client should handle this as a logout. If the status is *error*, the OIDC client should also handle it as a logout without trying any silent re-authentication.

For global logout, OIDC specifies two mechanisms: *Front channel logout* [22] and *Back channel logout* [23]. Front channel logout uses the browser to handle redirects that perform the logout steps. It is used by single-page application clients that handle session management in the front-end. Back channel logout is performed by the IdP through direct communication with the OIDC client. In this case, the client is an intermediate back-end web server that handles the authentication, authorization and session management with the IdP. For a front-end developer, understanding the front channel logout mechanism is important. [24] [25]

Figure 10 shows the front-channel logout flow between the IdP and the OIDC

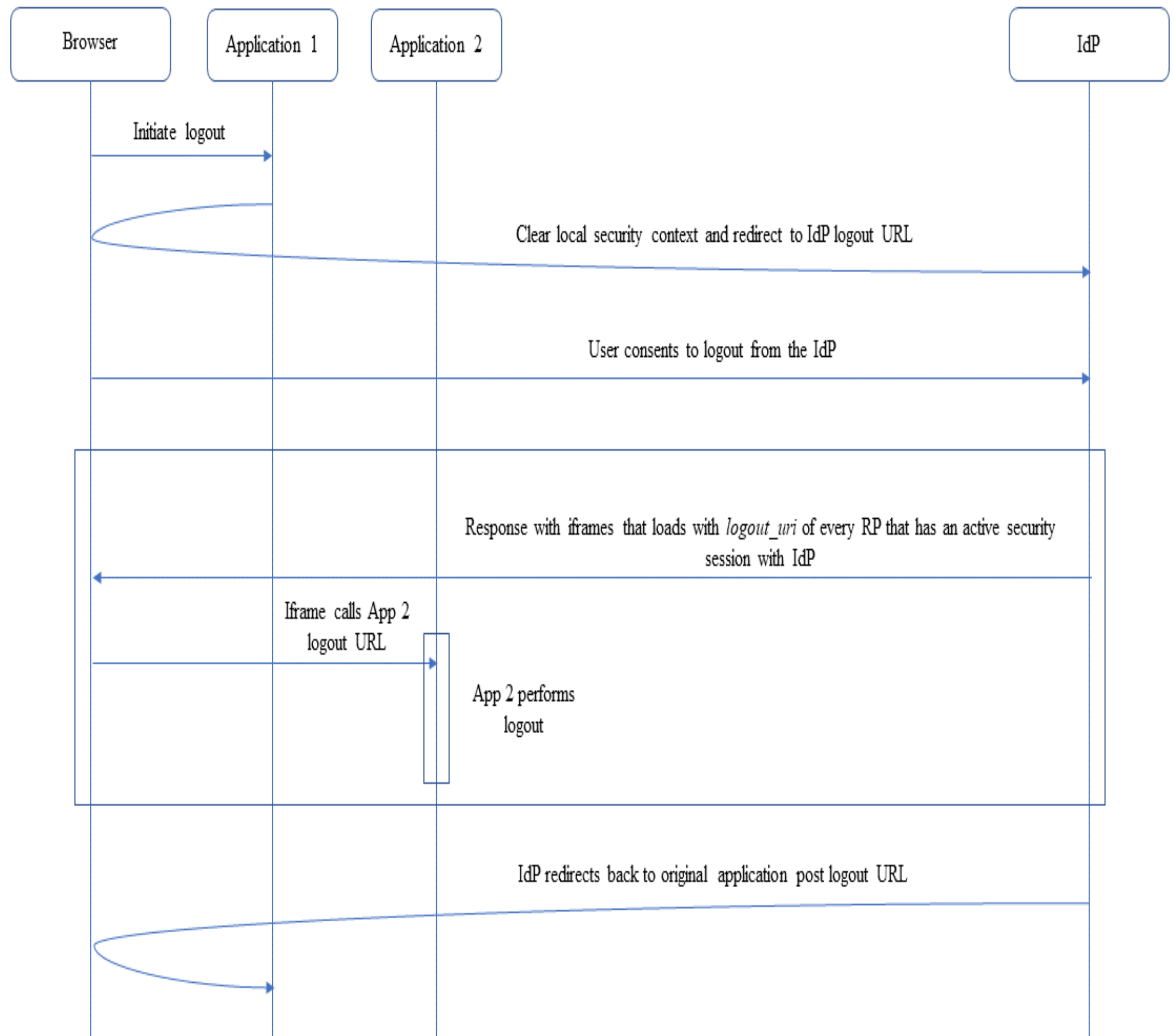

Figure 10: OIDC Front-channel logout.

clients. The OIDC clients are SPAs that are running in the user's browser. In the front-channel logout, the IdP has no direct way of contacting the client or server side of the other applications B and C to log the user out from those applications. Instead, the user's browser needs to send those logout notifications to the servers of applications B and C. To implement this, every application should register a *front_channel_logout* URI with the IdP. The user initiates the global logout from application A through the browser which triggers the IdP's global logout mechanism. The IdP's response to the browser contains an IdP-implemented global logout page that informs the servers of applications B and C about the logout. This is done by rendering an iframe within the global logout page. Due to the same origin policy, the global logout page creates a new iframe for each of the applications with their registered *front_channel_logout* URI. The servers then notify the client parts of

applications B and C in the user's browser and also delete the session state form the servers themselves. Finally, the IdP redirects back to application A's post-logout URL.

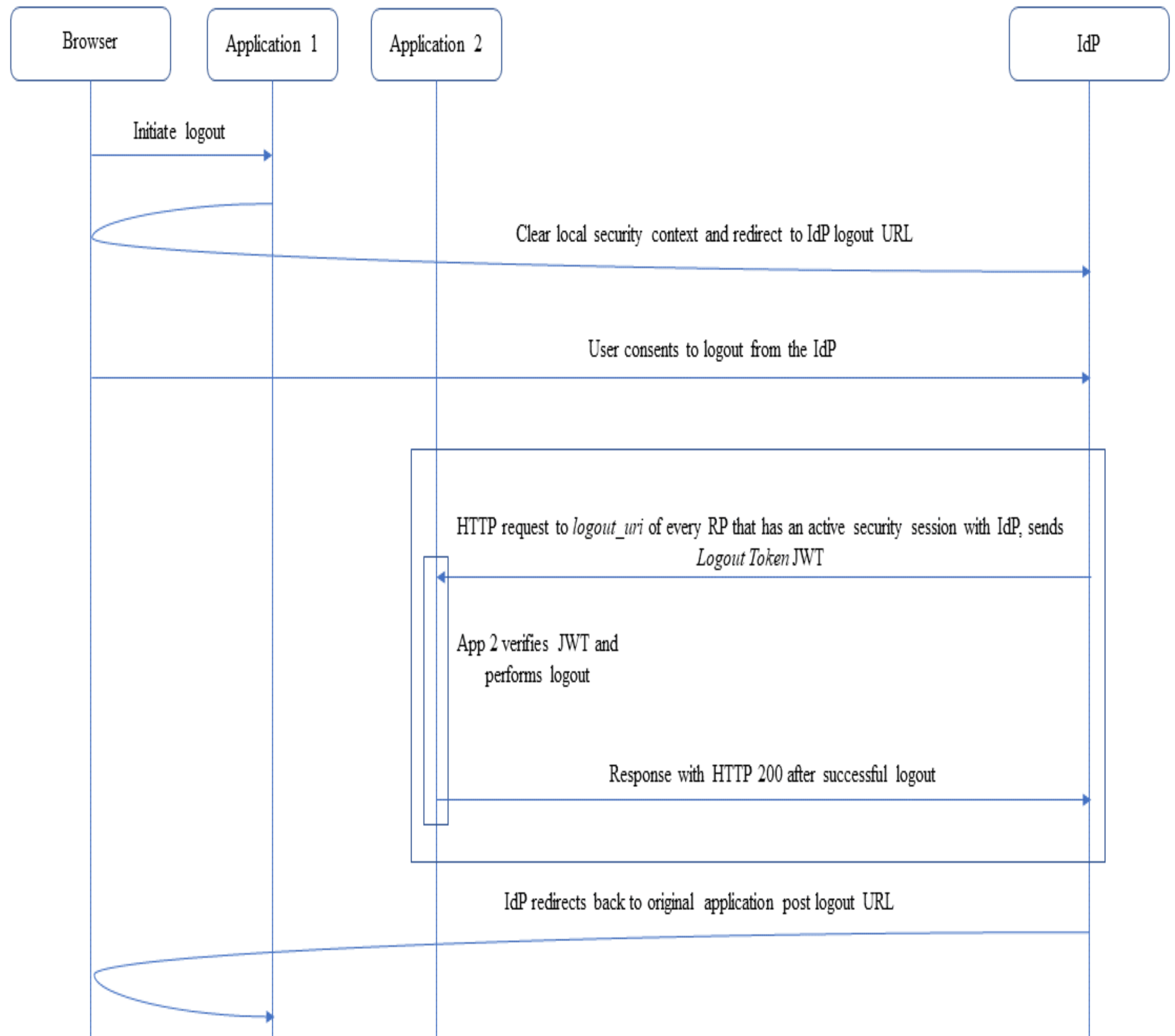

Figure 11: OIDC Back-channel logout.

An IdP initiates a back-channel global logout by sending a logout request to every additional client that has an established session to their registered *back_channel_-logout* URI. The IdP sends a logout token, which is known as a Security Event Token (SET) [26]. A SET is JWT token that represents a set of security and identity related events that occur for a client. The OIDC specification defines a set of required claims that should be present in the SET to identify it as a logout token. These are listed below:

- *iss* — The issuer of the SET, which is the identifier of the IdP.

- *aud* — The audience of the SET, which is the client ID of application the SET is intended for.
- *iat* — The *issued at time*, which tells the time the SET was issued at.
- *jti* — The JWT ID, which uniquely identifies the SET.
- *events* — This claim contains a JSON object, whose key should be *http://schemas.openid.net/event/backchannel-logout* and the value should be an empty JSON object *{}*

A back-channel logout request from the IdP to a client looks like this:

```
POST /backchannel_logout HTTP/1.1
    Host: client.com
    Content-Type: application/x-www-form-urlencoded

    logout_token=<SET logout token>
```

Upon receiving a back-channel logout request with a SET, the client should validate the SET token. The validation steps are similar to the validation steps of an ID token. Additionally, the client should verify that the SET contains an *events* claim with correct JSON object as defined above, and it did not receive any other logout token before with the same *jti* claim value. If any of the validation steps fail, the client should respond with a HTTP 400 Bad Request error. After the validation passes, the client performs its own implementation-specific logout process. The client can use the *sub* or *iss* claims in the SET to locate any session variables in its session and clear them out. After a successful logout, the client responds back to the IdP with a HTTP 200 response.

2.10 Token storage

Token storage is inherently more secure in a back-end server than in a front-end application running in a web browser. A front-end web application running in a web browser is called a public facing client because its source code and data are fully visible to the user from the web browsers. Back-end servers do not allow access to their code to the public. It is accessed only by the developers who have access to the development environment. The functionalities of a back-end server are provided through public APIs, which can be called by any public client. For each request to an API, the back-end server replies back with data that the user requested. The public client is not exposed to how the data is provided or where it is comes from. Therefore, storing sensitive information is considered secure in the back-end environment.

On the other hand, since the full source code of the front-end web application is visible to everyone, extra precautions must be taken when storing and using sensitive information from it. Shared secret keys, which are used in asymmetric key cryptography, are never stored in public facing clients.

The access token that is used in OIDC operation is a bearer token. Anyone with access to it can make a valid request to an API that accepts the token. Furthermore, it needs to be sent in as *Bearer* header field in the HTTP request. For a web application that has a back-end server, the tokens are securely stored in the back-end server. It modifies each outgoing API request by injecting the tokens, and relays the API replies back to the web applications. For web applications that do not have a back-end server, such as static web applications, the injection of access token into every request must be done by the JavaScript code.

Access and Refresh tokens can be stored in the web browser in the *localStorage* or *sessionStorage*. Browsers can store *key:value* pairs in these storage locations. Session storage persists the data across a single session. It allows access to data only within the window or tab in which the data was created. The session storage is cleared once the user closes the browser. Local storage persists the data even after the browser is closed. This data can be shared across different tabs and windows. Browsers store these values on a per-origin policy, which means they do not allow cross-domain access to these storages. However, any JavaScript running on the domain of the application can access the storage. This poses a security risk which is known as Cross Site Scripting (XSS). The XSS vulnerability can also be in a third party library, which is used in the web application. [27]

To avoid the risks associated with storing tokens in the browser storage, OIDC providers (OP) can choose to store the tokens in cookies. Cookies are also stored in the browser on a *per domain* basis, which means that a web page can only access a cookie which is associated with its domain. Cookies provide a number of flags that can be set to prevent them from being compromised. The OP can set a cookie's *Secure* flag to ensure the cookie is sent only with HTTPS requests, and the *httpOnly* flag to ensure the JavaScript cannot access it using the *document.cookie* DOM API. The *sameSite* cookie can also be set which ensures the cookie is sent only on same-origin requests. Browsers, however, have a size limit of 4 kilobytes for cookies. Therefore, any data or token larger than 4kB cannot be stored in a cookie. [28]

2.11 Software Development Kits

Software Development Kits (SDK) are tools that allow developers to use certain functionality. SDKs group operations and functionality pertaining to a certain logic and expose APIs that can be called to execute them. SDKs contain instructions to developers regarding their usage on the supported platforms. They are easy to integrate with existing applications, thus providing a convenient way for developers to introduce new functionality to their web applications. Authentication and Authorization SDKs are one example. The steps required to perform authentication and authorization are laid down by specifications. The translation from those specifications to code should result in the same functionality in every application. By grouping together these steps in one place and exposing them through APIs has several advantages as mentioned below:

- Developers do not have to write the same code for every application that follows

the same authentication and authorization steps. This significantly reduces the cost and time needed to develop and release an application.

- Developers do not have to worry about the steps that are performed to achieve a certain operation. Application developers do not have to be concerned about the methods of implementing certain operations such as token storage and session management. Storage and retrieval of tokens are handled by the SDK. Creation of iframes for session management and periodic polling of the IdP are handled by the SDK. The SDK contains robust error handling mechanisms that covers every boundary and edge cases. The SDK developers are responsible for maintaining the SDK and ensuring that the SDK works as intended without any flaws or vulnerabilities.
- Introducing new features and maintaining existing features can be done in one place instead of all the applications that use them. Having a single code eases debugging and fixing bugs.
- It provides a unified experience throughout similar applications. With the same configuration, the developer can ensure that the operations will work in similar ways across the applications.
- SDKs follow a uniform patterns in their APIs and usage. This abstraction of the core operations allows easier migration between different platforms.
- IdP implement and release the URL end-points in different versions. To maintain compatibility, new versions are published in a different URL. SDKs maintain the URL end-points that are used with a particular IdP. Since migrating to a newer version of the URL is internally handled by the SDK, the application developer does not have to worry about it. Without an SDK, the migration would need to be done for every application, which can be tedious and error prone. Furthemore, the IdP might change the operation with the new URLs. Without an SDK, the application developer would have to change the operation logic for every application.

Most identity providers have JavaScript SDKs available for use with a web client. Every SDK starts with loading the JavaScript bundle in the web page. This is done by the *script* tag with its *src* pointing to the bundle URL provided by the identity provider. This downloads the SDK on page load and exposes a global variable, through which the APIs of the SDKs are accessed. Web applications should consider Cross Origin Resource Sharing (CORS) policy when accessing resources from external servers. CORS is a mechanism that web browsers implement to prevent JavaScript code from accessing data, that originates from a different origin than the page where the code runs. For example, SDKs handle errors and display the error message through error logs for debugging, which are essential for a developer using the SDK. When the SDK script is loaded from the external site, the browser shows a *Script Error* message in the logs instead of the proper error message. This happens because the same-origin policy check within the browser prevents sharing of the error

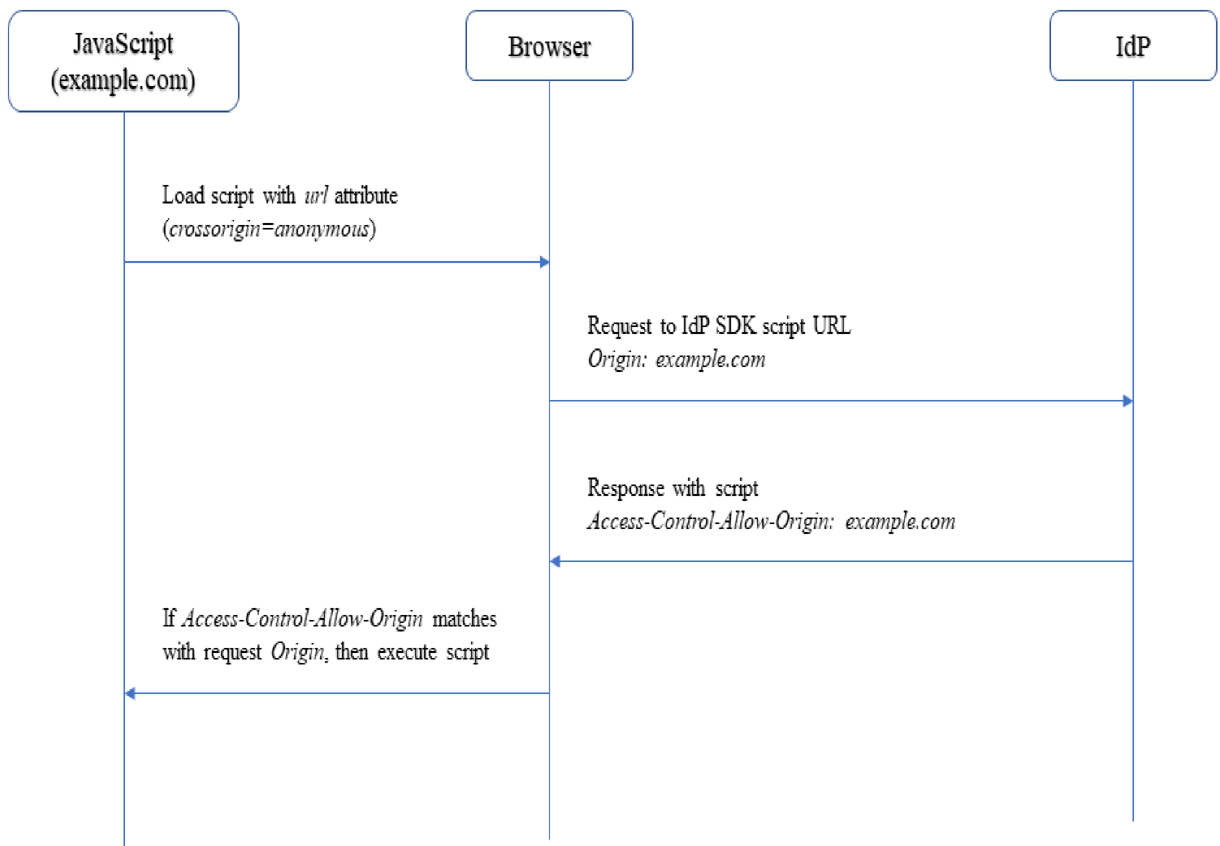

Figure 12: Script loading with CORS

log data across different domains. The same-origin check is not done by browsers when a script is loaded from an external server using the *script* tag. This can be circumvented with two steps. First, the *crossorigin* attribute on the *script* tag is set to *anonymous*, which tells the browser to enable Same Origin policy on the request [29]. Second, The external IdP server which hosts the SDK script responds to the request by including *Access-Control-Allow-Origin* header with the *Origin* value of the request. When both values match, the web browser allows data sharing from the external script. Apart from the *crossorigin* attribute, the *script* tag can also include the *defer* and *async* attributes [29]. *defer* ensures that the script is executed after the whole page has finished parsing, and the *async* attribute ensures that the script is run in the background. These two attributes ensure that the download and execution of the script does not block the parsing and rendering of the main HTML page.

Every SDK requires an initial configuration after loading the script. The configuration usually takes the client ID, token scopes and other IdP related fields. Most SDKs require an API key in the configuration, like Google's JavaScript SDK.

Although it is insecure for user authentication, Google uses API keys to restrict the number of requests that can be made to its API and is not related to user authentication. A different web application can be prevented from using the API key by using the HTTP *referrer* header field. The *referrer* field contains the information of the last web page that the user visited. For API requests, it would contain the value of the proper web application domain that called the API. The *referrer* header field is a retricted field which cannot be modified programmatically through JavaScript. The developer also needs to configure the accepted *referrer* values in the SDK's configuration portal.

Using the APIs is straightforward. They use asynchronous programming either by using callbacks or promises. JavaScript callbacks or promises are functions that are passed as a parameter to an SDK function. The SDK function calls the callback function with the defined parameters after it has executed its own code. Every SDK defines which parameters need to be passed with the callback function when calling the APIs. For example, Facebook's JavaScript SDK passes the result of a *login* call in a response object. The SDK manages the token storage, revocation and renewal, if supported. For every authentication and authorization related call, the SDK automatically adds the tokens to the API calls.

A common functionality that all SDKs provide is authenticating a user and providing an ID token to the calling web application. Most SDKs come with a pre-built login button that is styled according to the IdP's unique design. They offer a plug-and-play approach, enabling web applications to support the IdP's identity services with minimal settings. Developers can also use their *login* API for achieving the same result. A common trend for displaying the IdP's login page is by opening a pop-up window, instead of a redirect. In these cases, the pop-up window and the original application window communicate via the *postMessage* API. The original application listens for a *message* event that is sent by the pop-up window. The message contains the tokens and information about the login process. For security reasons, the original application should always check the origin of every request received through the *message* event. It should only accept and process messages that are sent from a known and trusted origin.

2.12 Credentials Manager

Credentials manager is built into to user-agent and helps manage user credentials. The World Wide Web (W3C) consortium defines the Credentials Management API that user-agents should provide, through which web applications can interact with the Credentials Manager [30]. Through the Credentials Manager, front-end developers can streamline authentication in their applications while providing an enhanced user experience by reducing the frequency of user interaction.

Credentials Management API provides a number of benefits which are detailed below:

- Users do not need to remember their username and password, as it is saved in the Credentials Manager. This allows users to set a secure password consisting

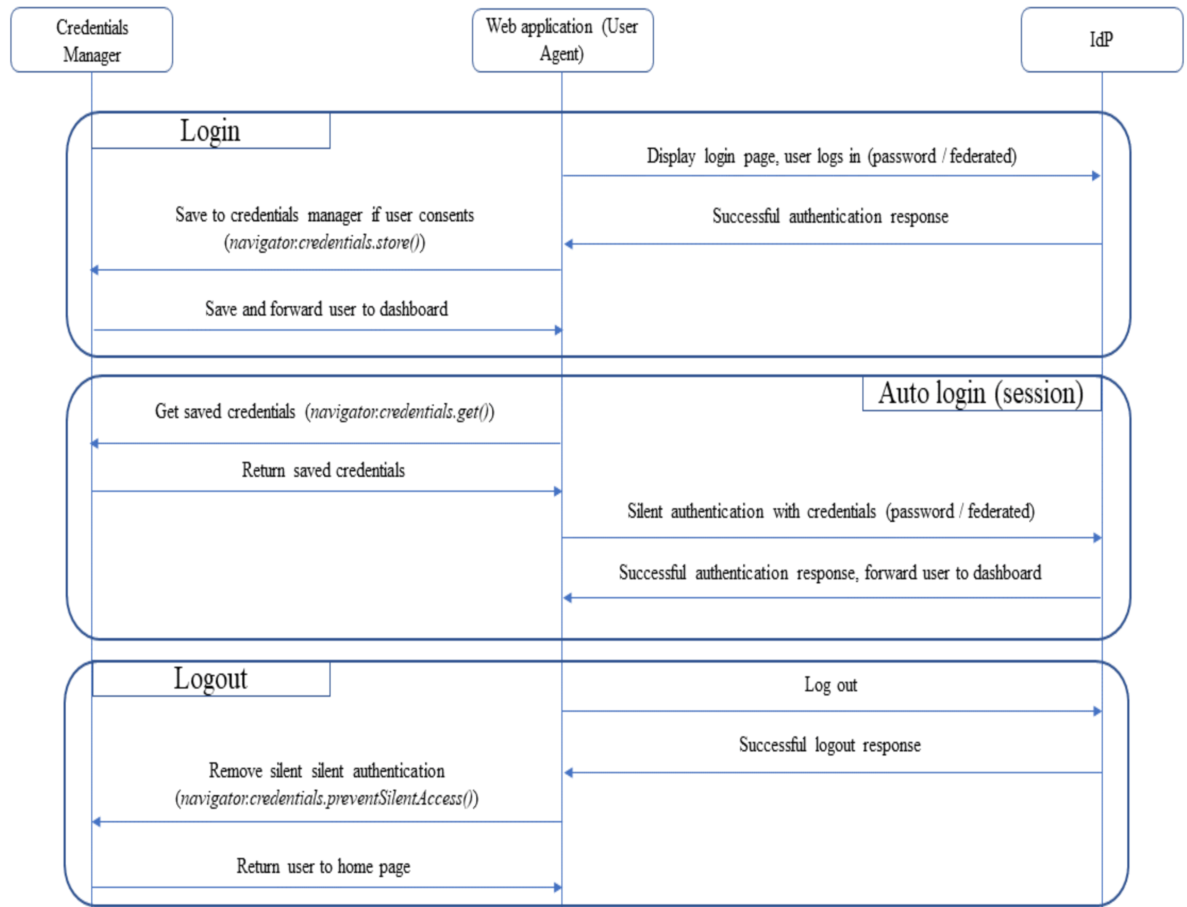

Figure 13: Credentials Management API usage

of random strings. Furthermore, this eliminates the need for a third-party password manager.

- Web applications can perform auto-sign in with the saved credentials. Since sessions usually expire after a day, users are not greeted with a login form every time they visit the web page. Considering that the credentials of the user do not change, this allows for simulating a virtually permanent session.
- Credentials Manager supports saving federated login credentials alongside password-based credentials. This allows web applications to integrate federated identity managers into their application while using the Credentials Manager.
- Web Authentication API, built on top of Credentials Management API, allows using physical objects, such as a USB fob or a biometric reader, as identification entities. This eliminates text-based passwords, thus raising the level of security.

2.12.1 Credentials Management API

Credentials Management API exposes interfaces to interact with the Credentials Manager. At this moment, it is supported by 83% of the browsers in use. Web applications can use these API through the *navigator.credentials* object. The API offers four main functions [30]:

- *navigator.credentials.create* — This API creates a *Credential* object which contains the user credentials. It can either be *PasswordCredential* or *FederatedCredential*. *PasswordCredential* takes a user ID and password as required values. *FederatedCredential* takes user ID and the federated identity provider as required values. The username and password is automatically detected from *HTML input* elements through the *autocomplete* attribute. HTML specifications define the *autocomplete* attribute values *username* for the username or user ID, *new-password* for a new password, which is used in a register form where ther user is creating a new password, and *current-password* for the user's current password.

- *navigator.credentials.store* — This API stores a credential into the Credentials Manager store. This is called after creating a *Credential* using the *create* API. The user agent always notifies the user before storing a password. It can store the *Credentials* only if the user consents.

- *navigator.credentials.get* — This API gets a credential stored in the Credentials Manager store.

- *navigator.credentials.preventSilentAccess* — This API prevents auto sign-in using the user's credentials. This is usually called after the user logs out of the application, such that the auto sign-in flow is not performed when the user visits the web page again.

The above API runs in a secure context of the user-agent. The secure context of the web application is calculated by the *origin* of the top-most active document of the web-page. This prevents iframes from accessing any data through the credentials manager. Every *Credential* object is saved against its secure context, which is checked every time its access is requested through the *get* API. Credentials Manager allows storing multiple credentials for the same web application. If multiple credentials exist, the user is always prompted to select the account they wish to sign in with when performing an auto sign-in.

2.12.2 Web Authentication API

Web Authentication (WebAuthn) API is built on top of Credential Management API that bring Public-Key based credentials to web applications. It uses *authenticators* to created strongly attested and scoped Public-Key credentials and allows storage and usage of these credentials with IdPs supporting them. An authenticator is an abstract entity that provides and confirms the identity of the user. W3C defines the *WebAuthn Authenticator Model* that lays down the protocols that an authenticator

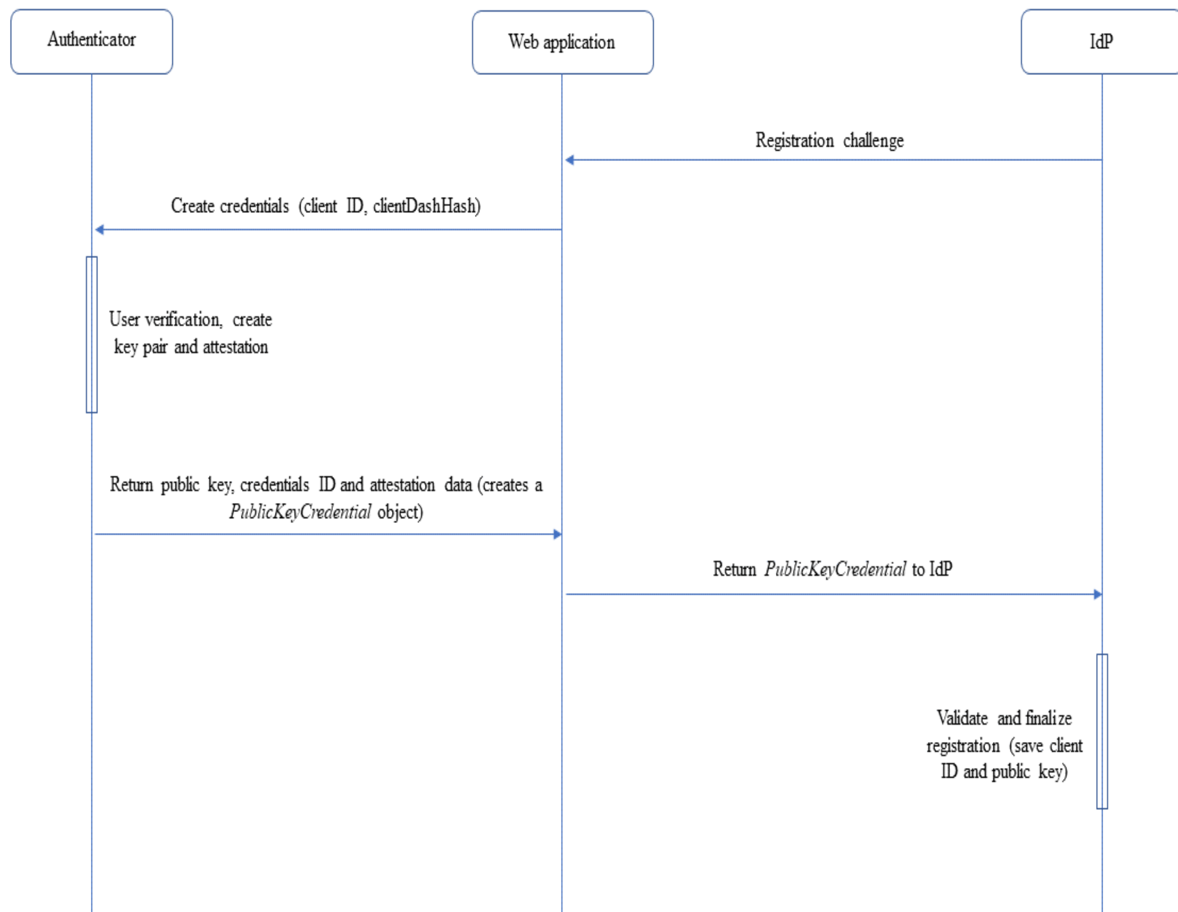

Figure 14: Registration using Web Authentication API

should follow so that it can be used by a user-agent with the WebAuthn API [31]. The authenticator can be a software entity that makes use of a platforms Trusted Security Module (TPM) or a hardware entity such as Windows Hello, or fingerprint sensors of a mobile device. It can also be an independant FIDO-CTAP compliant hardware module that can be used with Bluetooth or Near Field Communications (NFC). [32]

The WebAuthn API uses the follow two APIs, which are described as below:

- *navigator.credentials.create* — This prompts the authenticator to create and store a new set of public key credentials for the user.
- *navigator.credentials.get* — This fetches the existing public key credentials from the store.

Two primary entities in this operation are the IdP server and the authenticator. The server needs to support registration and authentication via asymmetric key cryptography. The authenticator stores the identity information for the user. In a password-based authentication, the identity information is stored in the user's brain.

The registration of a web client with a compliant IdP takes place in a series of steps as detailed below [31]:

- Server initially sends a registration challenge to a request made by the web client. The challenge is randomly generated by the server and it is used as an identifier of the operation. Additionally, the server sends the user information and the client ID.

- The client calls *navigator.credentials.create* method and passed the challenge parameters received by the server as a *PublicKeyCredentialCreationOptions*. This object contains the client ID, user information, challenge sent by the server, and a *pubKeyCredParams* object that contains the algorithms that should be used for the credentials.

- The browser internally calls the *authenticatorMakeCredential* method on the authenticator after verifying the client data, and adding some additional data on its own together into a *clientDataJSON* object. The *origin* of the request is also added to this object which the server uses for verifying same-origin requests. The *SHA-256* hash of the *clientDataJSON*, which is known as *clientDataHash* is passed to the authenticator method call.

- The authenticator prompts the user for identifying themselves with available options, such as Windows Hello, or a finger print reader. After the verification, the authenticator creates a new asymmetric key pair and attests the public key by signing it with its own private key. The private key on authenticators are embedded into them during the manufacturing process. It can be validated from a root of trust cerfificate through a certificate chain.

- The browser receives the *PublicKeyCredential* object, which contains an authenticator generated *rawId*, the *clientDataJSON*, the attestation data created by the authenticator and the public key. The object is sent back to the IdP server to complete the registration process.

- The server verifies the information in the *PublicKeyCredential* object. Most importantly, it verifies the original challenge, the client ID, and the origin. Finally, it verifies the attestation over the *clientDataHash* and saves the credentials information for future use if validation succeeds.

The authentication of a user from the web client follows similar steps to the registration steps with a few alterations as detailed below:

- The server sends a challenge to an authentication request by the web client. No other information is passed in this step.

- The web client calls the *navigator.credentials.create* method of the Credentials Manager API. It passes the challenge in a *PublicKeyCredentialRequestOptions* object to the *create* method, which contains the challenge sent by the server along with some additional optional information such as the client ID.

- Internally, the browser calls the *authenticatorGetCredential* API of the authenticator, passing the client ID and a *SHA-256* hash (*clientDataHash*) of the *clientDataJSON* object which contains the data that was provided in the *create* call. The browser will carry out some initial verification of the data before creating the *clientDataJSON* object.

- The authenticator uses the client ID to search for a matching credential, and prompt the user for verification and consent. Once verified and consented, the authenticator creates an attestation by signing the *clientDataHash* with the user's private key generated in the registration step. The authenticator sends this assertion and the authenticator data back to the browser in a *PublicKeyCredential* object.

- The browser sends this assertion, the *clientDataHash* and the authenticator data back to the server. The server uses the user's stored public key to verify the assertion. Furthermore, it verifies the challenge that it sent in the first step, the origin of the request, and the client ID. Once verified, the server logs the user in and send the authentication tokens to the web client.

2.13 Authentication and Authorization in Native applications

Native (mobile) application development traditionally uses their own development environments and programming languages. Two of the most popular mobile Operating Systems in the world at this current moment are Android and iOS. Both provide their own SDKs for application developers that enable them access to the whole ecosystem. However, there has been a steep rise in the use of OS-agnostic tools for creating native applications. This allows developers to target both platforms while saving significant development time by using the same set of tools and development environment. Many businesses have adopted this approach to provide their applications to their customers irrespective of the OS they use. Among the libraries that exist today, React Native and Ionic are the most popular for native application development. Developers use web technologies like HTML, JavaScript and CSS to create the application, increasing the adaptability and familiarity of developing native applications. This allows teams or organisations to use their existing front-end developers in a native application development project, instead of hiring native application development specialist for Android or iOS. In that regard, it is important to touch upon token-based authentication and authorization mechanisms that are followed and recommended for native applications. [33]

OAuth 2.0 lays down some ground rules that should be followed for native applications [34]. Native applications should use a web browser for performing the authentication with the IdP. This is done by using platform specific APIs that allow one app to communicate with another. Using a browser application is advantageous because it allows applications to leverage SSO. Most IdPs save session state about a user in the browser cookies. That way, the user can be logged in automatically on

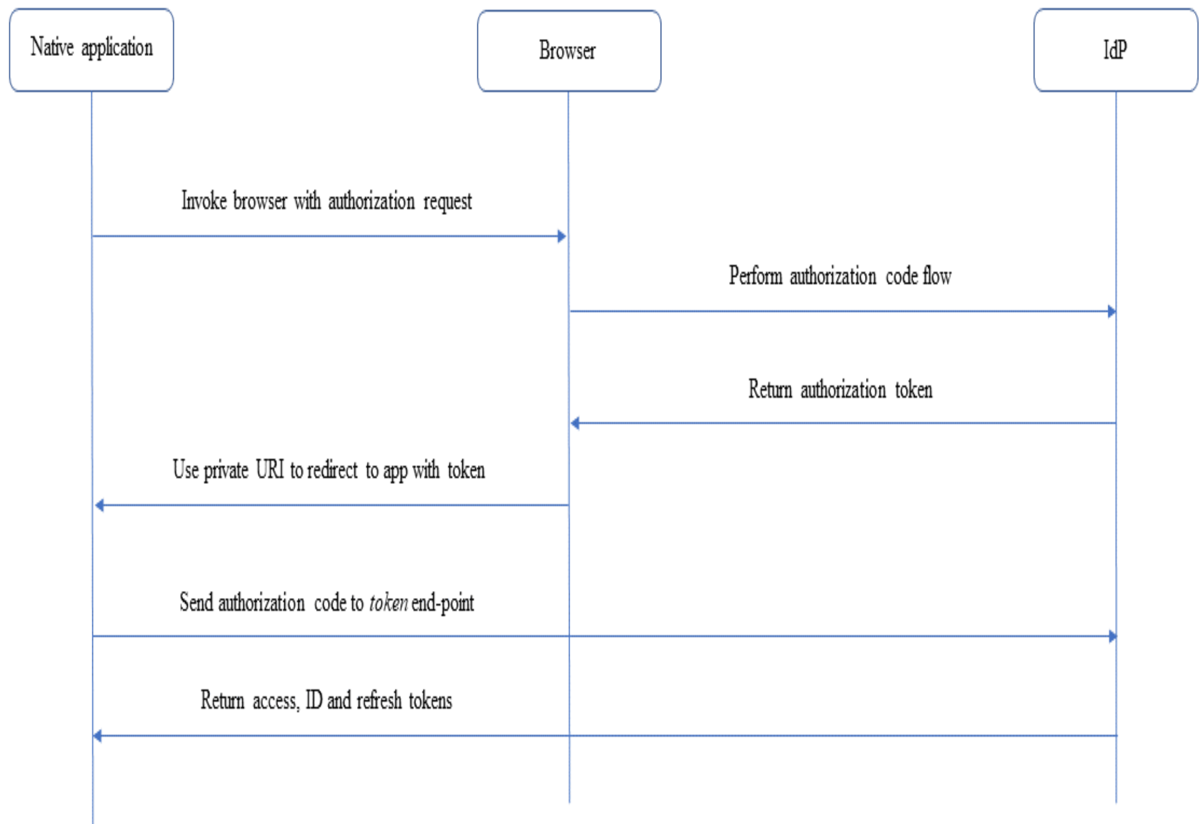

Figure 15: Native app, browser and IdP interaction

every other app which uses the same IdP. On the other hand, if a native application chooses to perform the authentication steps on its own without a browser, it will face constraints when communicating about the session state of a user with other applications.

The OSes allow native applications to use web-views, which are instances of a browser that can be spawned inside an application's context. Use of web-views are strongly discouraged because the native application has full access to the web-view's context. It can record every activity happening inside the web-view and can also read the cookies. Instead of web-views, native applications should use external trusted browsers.

Redirecting from the browser to the native application with the authorization code is done using Private-use URI schemes. Every application is assigned a URI scheme by the OS. They are typically in a reverse order of web domain names. It is encouraged that native applications use a similar scheme, for example, an app with a domain *app.example.com* should use a URI scheme as *com.example.app*. OSes allow applications to communicate using these private-use URI schemes. The redirect URI

after authenticating with the IdP should be the private-use URI. When the browser attempts to redirect using such URI, the operating system handles the URI and opens the application which is registered to the URI.

Native applications should use Implicit flow with PKCE. The secret verifier that the client app generates in this approach, helps protect the authorization code re-use if it is intercepted by another application which is registered to the same Private-use URI. Only the app in possession of the secret verifier will be granted the tokens by the IdP.

3 Use Cases

3.1 Industrial Control System

3.1.1 Architecture

The control system provides users an interface to monitor and analyze the data from an industrial. The core services are provided by the middleware, which is responsible for various operations such as device discovery, device management, data analysis and API management. This middleware is based on a microservices architecture, which allows independent development, deployment and maintenance. The API management layer exposes various interfaces that are used by the client software to display and interact with the data.

The IdP is a directory-based OIDC-compliant OpenID Connect provider (OP). The company employs Identity and Access Management (IAM) tools for managing the user data within the company. An intermediate API management layer provides the bridge between the microservices middleware and the client application. The API management layer provides APIs that the client calls for different services that it wants to access. Internally, the API management layer is responsible for translating the client-based API calls to microservices-based API calls. This allows the advantage for the client applications to not worry about the discovery and accessibility of the microservices directly. Apart from request proxying, the API gateway also serves as the security layer between the IdP and client applications. It provides services such as auditing, session management with client applications and client key storage.

There are two types of web applications in this system. The first type is a Single-Page Application (SPA) which acts as the Relying Party (RP) itself. The architecture is depicted in Figure 16. The RP registers itself with the IdP, which provides the RP with a Client ID. The RP also provides a login callback URL and logout callback URL during registration, which is required by the IdP. The IdP redirects the browser to these callback URLs during login and logout respectively. The SPA is hosted by a back-end server which serves the SPA when requested by the user agent. All interactions with the IdP and API gateway happens directly from the SPA. The SPA is responsible for storing the tokens and using them for each request made to the APIs. The API gateway verifies the authorization tokens received with each request. It communicates with the IdP to check the validity of the token and incorporates access control checks against the API that is being accessed.

The RP uses Implicit grant with PKCE flow of the OIDC specification for getting the tokens. During the authentication phase, the front-end application sends an HTTP GET request to the authentication end-point of the IdP. First, the RP generates a code challenge and code verifier. Then, it redirects the user-agent to the IdP authentication end-point, and sends its client ID and code challenge as redirect parameters. The IdP presents the user agent with the login form, where the user provides their registered credentials. Once authenticated, the IdP redirects the user agent to the login callback that was registered for that RP and includes an authorization code. The RP calls the token endpoint of the IdP in an HTTP POST request with the authorization code and code verifier. The IdP verifies the

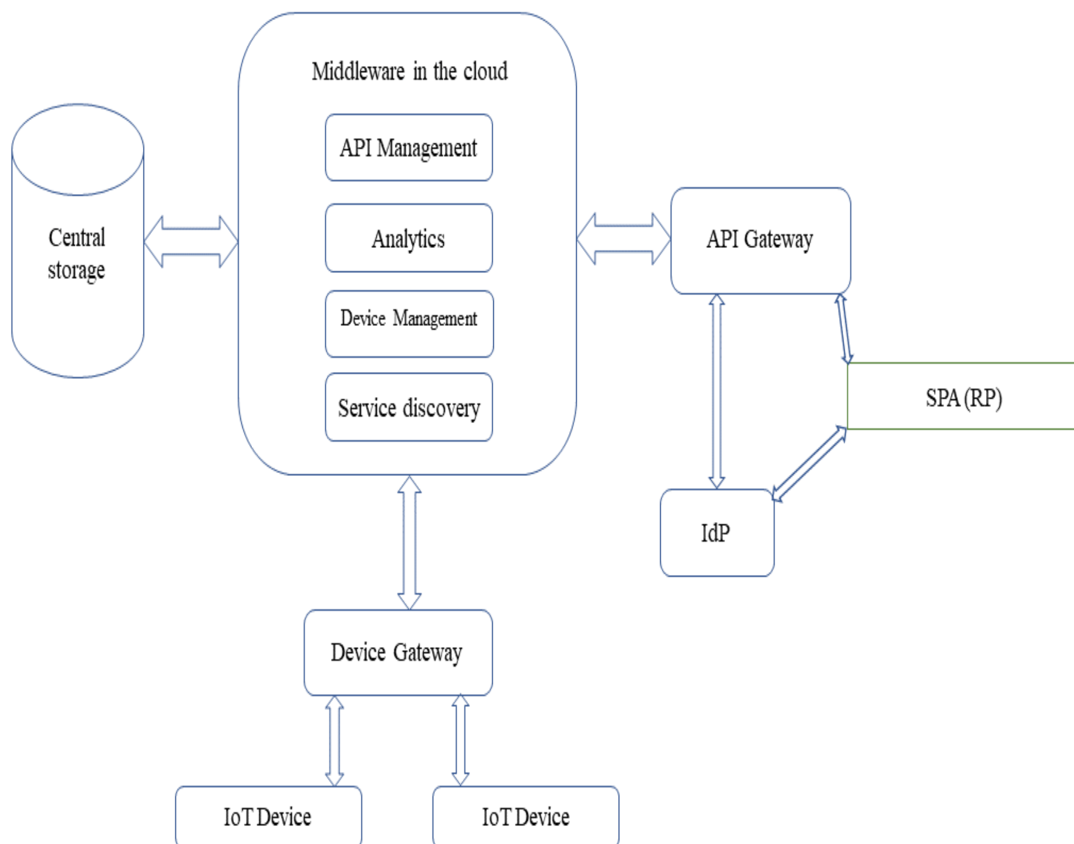

Figure 16: IoT architecture.

previously sent code challege with the code verifier and replies back to the RP with the access token and the ID token. The RP parses the JWT tokens and creates a new session for the user and redirects the user agent to a customized home page. The RP uses information provided with the ID token to create a new session.

The RP saves the tokens in the browser localstorage. It sends the authorization token with each request to an API. This is done by implementing an HTTP interceptor in the application. The HTTP interceptor intercepts every outbound request to the API server and injects the authorization token in it as a *Bearer* token. The SPA also implements route guards for the protected routes. Using the HTTP interceptor, the SPA checks its localstorage for a valid token before granting access to the route. If a valid token is not found, the SPA redirects the user to the login page.

For session management, the SPA uses iframes to preform session validation with the IdP. After receiving an ID token, it creates two iframes. The first iframe is

loaded using its own internal page, which contains session management operations. The second iframe is loaded using a URL provided by the IdP. The IdP stores session information in the browser cookies for the current user session. The SPA iframe periodically communicates with the IdP iframe to get the session status. If the IdP sends a changed status, the SPA tries a silent re-authentication with the IdP. If the IdP returns an error status, the SPA treats it as a logout and initiates logout operations.

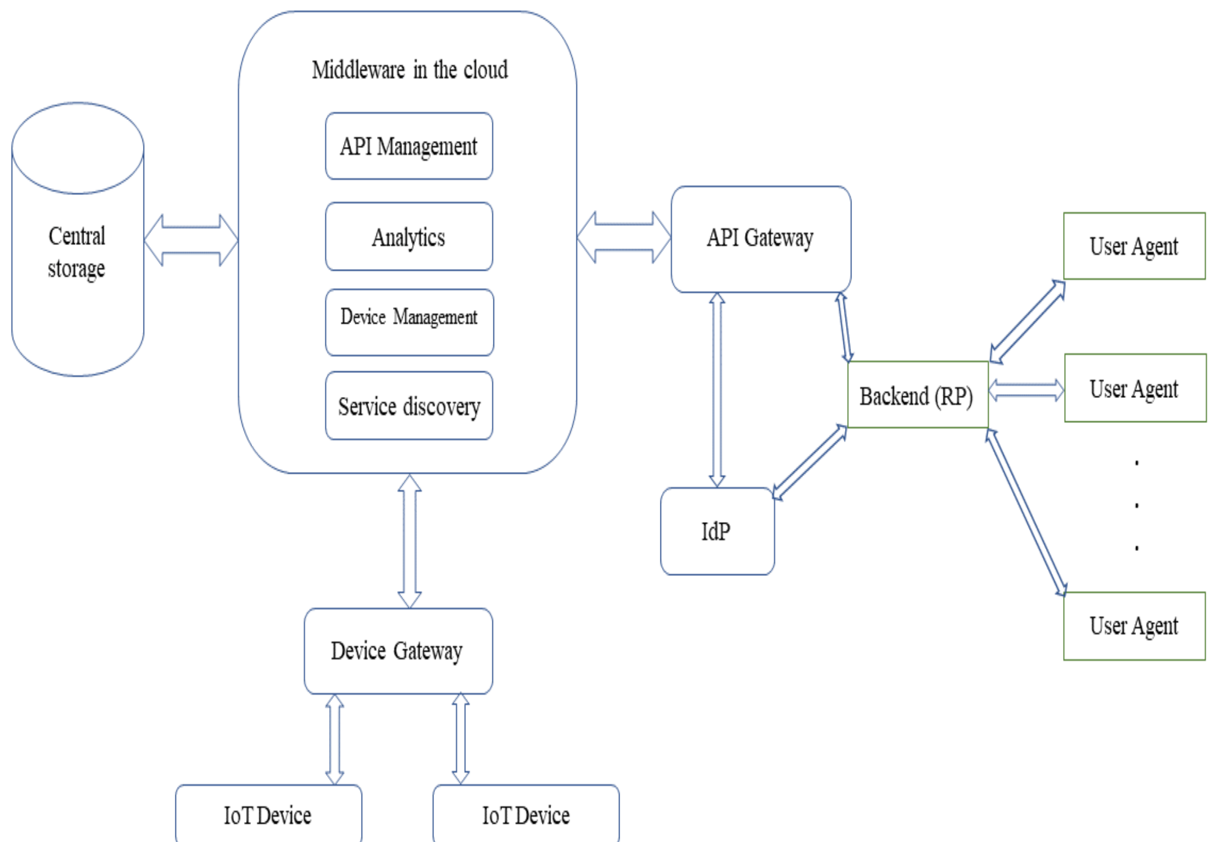

Figure 17: IoT architecture.

The second type is an SPA supported by a back-end server which acts as the RP. The architecture is depicted in Figure 17. The RP uses Authorization code flow for getting the tokens. It uses the *client_secret_basic* method for the */token* end-point authentication. The developer saves the *client_secret* that it receives from the IdP during the registration and securely stores it in the back-end.

The back-end sends the ID and access tokens to the SPA in a redirect. The SPA stores these tokens in the browser localStorage. Token storage and session

management follows the same principle as the SPA without the back-end server interaction.

3.2 Social Media Center

The Social Media Center is a web application that groups together all the posts made by an organisation to various social media platforms and lists them in a chronological order. It is developed and hosted by a district council in a city that wants to provide a unified experience to its users. The organisation posts various messages throughout the day to Twitter, YouTube, Instagram, Facebook and its own blog page. A survey carried out by the organisation showed that most of the district's inhabitants do not check all the platforms. Due to the different capabilities and functionalities offered by the aforementioned platforms, a post created in one of the platforms may not be possible to create in the other. For example, uploading a high quality video on YouTube or creating an event on Facebook are not supported by the other platforms.

The Social Media Center fetches all posts made by the organisation in its own accounts across the different platforms and groups them together chronologically in one place. Users are able to browse through the list and view each post without leaving the application. Furthermore, the application allows the users to *like* or *comment* with their own accounts in Twitter and YouTube. Instagram does not allow commenting or liking on any post through the APIs. It allows only business accounts to be managed by a third-party application. The application also allows the users to filter the posts based on platform, so that the users can choose which platform's feeds they want to see.

The web application is an SPA that is hosted by a back-end server. It uses OIDC compliant access tokens with all the platforms. The application needs two different authentication and authorization levels. The first level is to fetch all the feeds from the organisation's own accounts. This operation includes registering the application with a developer account, getting the API access tokens and using the access tokens to make calls to the APIs. The second level is to allow the users to interact with the posts using their own account in the respective platforms.

For Twitter, the developers need to create and register their app on Twitter developer dashboard. Once registered, Twitter provides the *consumer key* and *consumer secret*. These allow the application to identify itself with Twitter and access its APIs. The developer also gets the option to link an existing public account. If done, then the developers gets an *access token* and *access token secret*. The access token and secret are used to fetch the feeds from the council's public Twitter feed. To allow a user to post on their behalf, the application needs user access tokens. Twitter uses an operation which they call *3-legged OAuth flow* [35]. As a first step, the application requests Twitter to grant it a request token. It sends its *consumer_key* and a registered callback URL in this request. Twitter replies with a *token* and *token verifier*. In the second step, the application redirects the user to Twitter's */authorize* endpoint along with the *token* received in the previous step. The user uses their credentials to authenticate themselves, unless already logged in. The user also needs to grant access for the permissions the application requests. If granted,

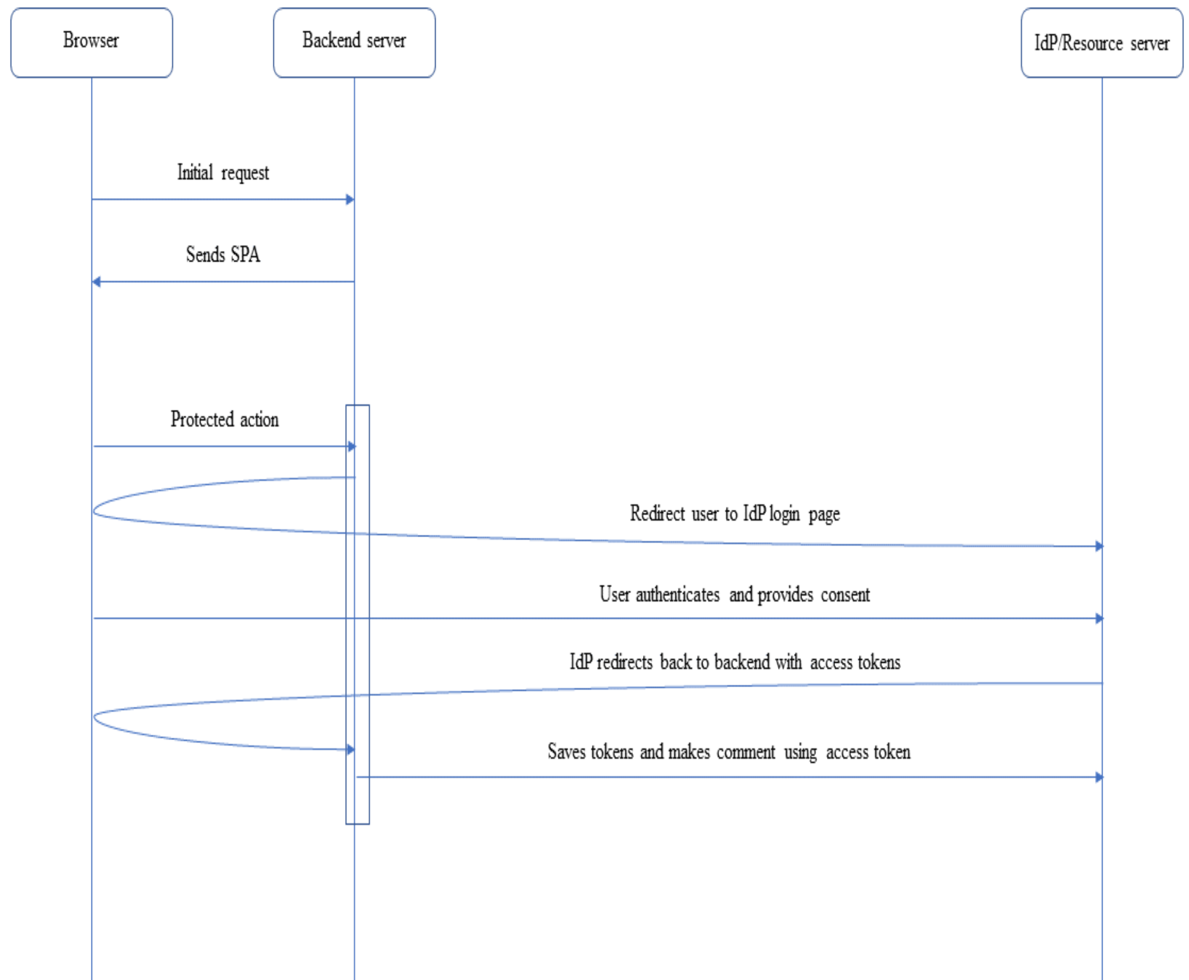

Figure 18: Access token flow.

Twitter replies back with the same *token* and a *token verifier* to the callback URL. As a third step, the application calls the Twitter */access_token* endpoint with the *consumer key*, *token* and *token verifier*. If everything checks out, Twitter replies back with the user's access token.

Youtube requires an API key per application to make calls to its APIs. This API key is available to the developers once they register the app in the YouTube developer dashboard. To get information about a particular channel, the app makes an HTTP GET request to the *channel* end-point of the YouTube API URL [36]. It sends the API key in a *key* parameter and the channel ID in a *channelId* parameter. Since the requested channel is public and does not have any private videos, an access token is not required. The videos in the channel are fetched from the *videos* end-point, using the same API key and the Channel ID [37]. For posting comments by the user, the web application needs to get an access token. YouTube supports the OAuth 2.0 Authorization grant flow for granting access tokens. The web application constructs

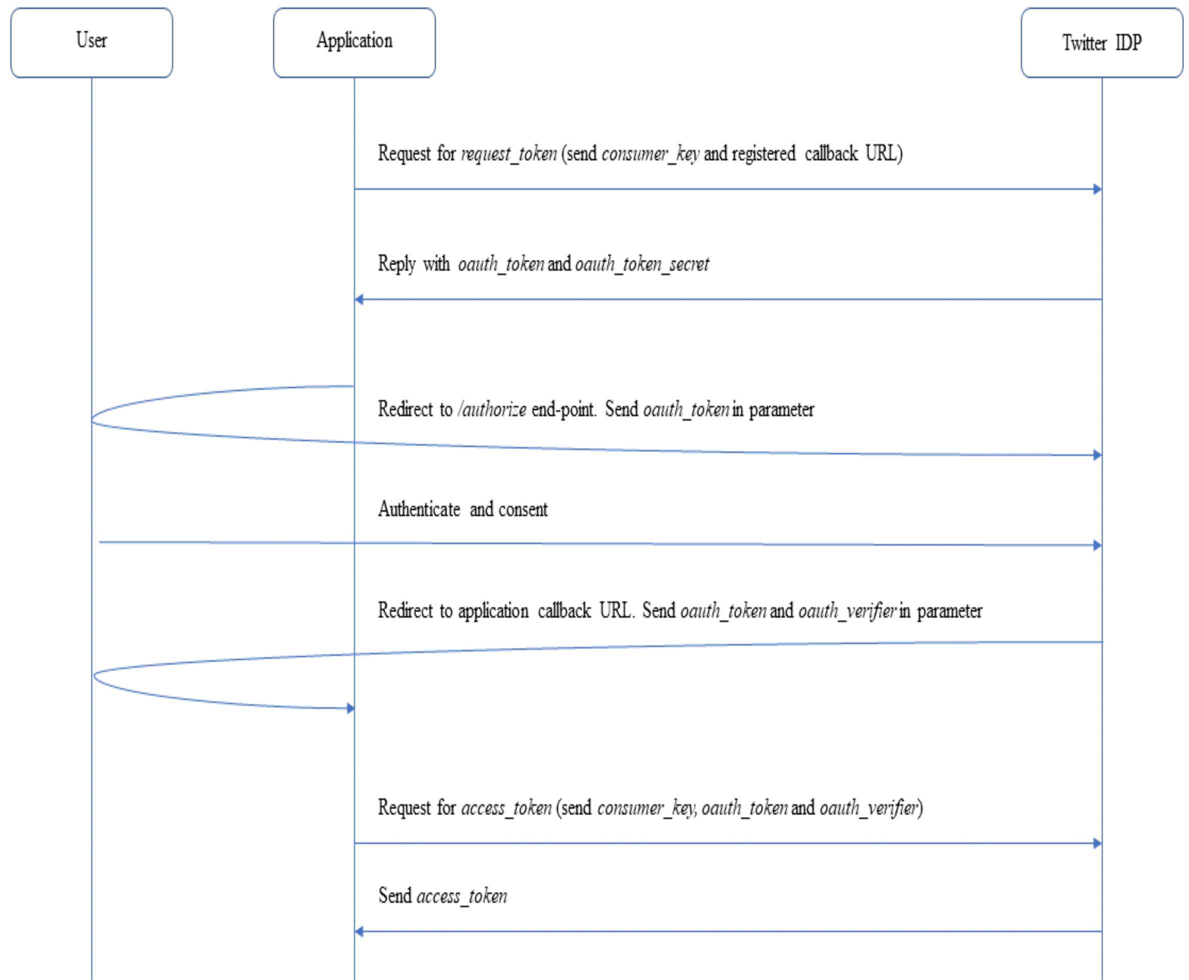

Figure 19: Twitter 3-legged authorization OAuth flow.

a URL to YouTube's *https://accounts.google.com/o/oauth2/v2/auth* end-point along with the client ID, redirect URI, response type and scope parameters. Then, it redirects the user's browser to the URL where the user authenticates themselves in the Google IdP. After authentication, the user is asked to grant permission to the web application for the operation that was requested. If the user grants the permission, the IdP redirects the user's browser to the redirect URL specified in the earlier redirect with an authorization code. The web application sends a HTTP POST request to *https://oauth2.googleapis.com/token* with the client ID, client secret, the authorization code and a redirect URI. The IdP verifies the parameters and replies back with an access token. The reply also contains an *expires_in* field that contains the expiration time of the access token. The web application sends this access token as a *Bearer* token in the *Authorization* header of every HTTP POST request for making a comment. The comments are made through an HTTP POST request to the *comments* end-point of the YouTube API URL [38].

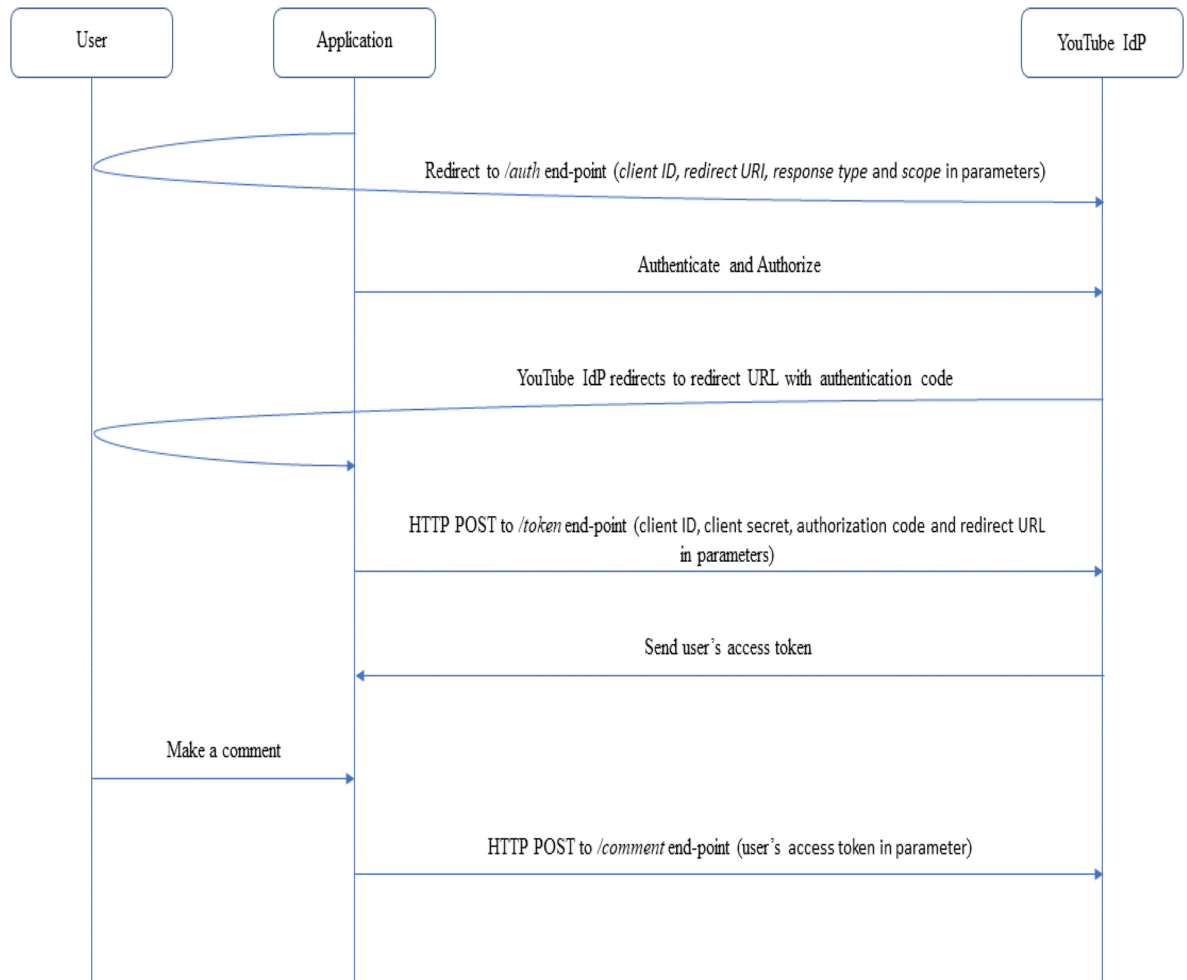

Figure 20: YouTube Authorization code flow for user's access token

Facebook allows access to its APIs through a suite which they call as the Graph API platform [39]. Facebook requires applications to have a *Page token* to be able to access a page. Furthermore, the token should have the *MODERATE* permission granted by the page owner. Getting a page access token involves the following steps:

- Developer creates a developer account on Facebook and registers an app.
- Developer selects permission required by the app and chooses the permission *pages_read_engagement* and *pages_read_user_content*
- Page owner grants the permission to the app to access the page.
- Developer gets short-lived user access token from the developer dashboard.
- Developer exchanges the short-lived user access token for a long-lived user access token. This is done using an HTTP GET call to */access_token* end-point along with the client ID, client secret, and the short-lived user access

token. The developer gets the client ID and the client secret from the developer dashboard.

```
GET https://graph.facebook.com/{graph-api-version}
    /oauth/access_token
    ?grant_type=fb_exchange_token
    &client_id={app-id}
    &client_secret={app-secret}
    &fb_exchange_token={short-lived-user-access-token}
```

- The call returns a long-lived user access token. The developer exchanges it with a long-lived page access token using a HTTP GET call to the */accounts* end-point along with the long-lived user access token. The *user-id* is accessed from the user's Facebook account.

```
GET https://graph.facebook.com/{graph-api-version}/{user-
    id}/accounts/
    ?access_token={long-lived-user-access-token}
```

- The call returns a long-lived page access token that contains the permission that was granted in the earlier steps. This access token has no expiration date.

The web application uses the page access token to make requests to the */page-id/posts* end-point *feeds* API to get a list of all published posts in the page. The data contains links to like and comment for each post. These redirect the user to the Facebook post where the user can comment or like on the post.

Instagram moved its API to Facebook's Graph API from 2018 onwards. This use case, however, uses the old Instagram authentication APIs. It supports the OAuth 2.0 protocol for the access token exchange. To get an access token, the developer creates a developer's account by logging in. The credentials used for logging in are the same as the ones used for the Instagram page of the district council. In the developer's dashboard, the developer creates and registers a new application. The developer needs to provide a redirect URL, which the Instagram IdP will use for redirects. Once successfully registered, the developer gets a client ID and client secret. Instagram supports two methods for receiving the access token: *Server side flow* and *Implicit Flow*. Since the relying party for this use case is the back-end server, hence the developer uses *Server side flow*. First, the developer redirects the web browser to the */authorize* URL.

```
redirect https://api.instagram.com/oauth/authorize/
    ?client_id=CLIENT-ID
    &redirect_uri=REDIRECT-URI
    &response_type=code
```

The *client-id* is the same as the one they received from the dashboard, and the *redirect-uri* is the same as the one registered in the dashboard. Instagram displays a

login page, where the developer authenticates using the Instagram page's credentials. Once the developer has logged in and approved the permissions, Instagram redirects to the registered redirect URL with an authorization code. The developer exchanges this authorization code with an access token by making an HTTP POST request to the *access_token* end-point.

```
POST https://api.instagram.com/oauth/access_token/
    ?code=<authorization_code>
    &client_id=<client ID>
    &client_secret=<client secret>
    &redirect_uri=REDIRECT-URI
    &grant_type=authorization_code
```

The request contains the authorization code received in the previous step, the client ID, the client secret, the grant type which is set to *authorization_code* and the registered redirect URL. The Instagram IdP replies back with a JSON object that contains the access token. The JSON object does not contain any expiry date. In case the access token expires, the developer will get an error message of type *OAuthAccessTokenException*. In this case, the developer has to perform the aforementioned steps again to get a new access token. With the access token, the developer can make API calls to fetch the posts from the page.

```
GET https://api.instagram.com/v1/self/media/recent
    ?access_token=ACCESS_TOKEN
```

The session is managed by the back-end server by creating a session identifier at the start of each session and storing it in its own database. The back-end server creates a session cookie which contains the identifier value. It saves the Twitter and YouTube user tokens along with the session identifier. It displays the active token information to the user in the app settings. The user is given the option to revoke any active tokens if they so choose. Alternatively, they can revoke access to the web application from their own Twitter or YouTube pages. During logout, the web application clears the session cookies and removes the user tokens that it has saved.

3.3 Component Library

Component Library is a product of a design system that provides User Interface (UI) elements. These UI elements are used to compose complex user interfaces for a web application. A design system specifies and defines a set of design rules that ensures uniformity and coherence. It closely follows industry standards and brand guidelines connected to a company, and produces a single source of information on UI design. The component library complements the design system and provides users a set of pre-defined components which allow them to build a product adhering to the design system guidelines.

The component library is published as an NPM package for React and Angular, which are the two most popular front-end frameworks to create single-page Applications. The library provides a set of atomic components, as well as composite

components. For example, a checkbox is one of the atomic components and a dialog modal is one of the composite components. The composite components are composed of atomic components.

The library publishes a Login component as part of its composite components. Users use this component in their web applications for displaying a Login UI to their users. Being a presentational component, it only provides the text input and password input elements, as well as APIs that pass the raw text value of these inputs to the parent component.

3.3.1 Proposal

A common architecture paradigm for SPAs is to segregate the components into presentational and container components. Presentational components are stateless and are primarily concerned with how the UI looks. They are responsible for changing the UI state in response to an event or a change in state. Container components, on the other hand, are stateful and contain the business logic of the application. They are concerned with performing a specific task in response to an event passed down by the presentational components, and pass the result of the operation back to the presentational components. In its current implementation, the Login component is a presentational component. In this section, we discuss how it can be turned into a library of its own that provides authentication and authorization for every web application within the company.

The UI for the Login component and associated functionality for authentication and authorization can be provided as a Software Development Kit (SDK). The purpose of this SDK is to abstact the authentication and authorization steps with the company's IdP. The SDK will follow OIDC specifications to factilitate token exchange with the IdP. It will expose a *login* API that opens up the login page in the IdP domain. The login page will be opened via *window.open* browser API. To facilitate cross-browser communication, *window.postMessage* API will be used. The popup will allow the user of the web application to authenticate themselves with the IdP, following which, the IdP will return the access token back to the web application.

The SDK will handle the token storage in the web browser and expose APIs through which the web application will access the tokens. The SDK will handle session management by creating hidden iframes. It will handle silent re-authentication before session expiration. It will also handle logout using OIDC front-channel logout specification and provide options for logging out of the IdP if the user chooses.

The developers of the SDK should consider the consistency and coherency of the user's experience when integrating the SDK with their applications. A few principles can be laid down that should be followed in regards to that.

- The SDK should clearly identify the platforms that it can be used with. For example, a web application runs in a web browser and its context. Thus, the supported protocols should adhere to the specifications laid down for browser-based applications. Currently, the advancements in technologies allow the creation of Windows-based or mobile-based applications using web technologies.

It allows users to use the same code to create applications for browsers and operating system. Even if they are created using the same technologies, an application running in the operating system runs in a different context than an application running in the browser. Thus, the security requirements differ vastly between the two types of applications.

- The SDK should identify the identity providers that it supports. In the context of an organisation, it is usually a single private IdP that is maintained by the company. The implementation of the authentication and authorization steps should be followed according to the specifications laid down by the IdP. It should support all the necessary configuration profiles that the users can configure with the IdP.
- The SDK should handle all errors and provide the necessary error logs and reports back to the developer. Furthermore, the error reports should comply with the security standards. It should allow the authentication or authorization steps to fail gracefully.
- Token properties such as token expiration times should follow the standards laid down by the IdP for the application. For example, most IdPs allow the users to set a custom expiration date for the ID tokens for certain type of applications.

This thesis leaves a couple of areas for future work which are listed as follows:

- Implementing the Web Authentication API and analyzing its benefits and drawbacks in terms of user experience and security.
- Implementing the proposed SDK and documenting the developer's feedback on its usage.

4 Summary

In this thesis, we examined the methods related to authentication and authorization that can be securely used in a front-end application. We documented the token based authentication and authorization methods in detail. The OIDC specifications, which lay down the operation principles, provide detailed steps that should be implemented for different application scenarios. The use of JWT as the token format simplifies its adoption across web applications. Since JWTs can be easily passed through the HTTP protocol, it allows integration of identity and access management without compromises in the user experience.

Different strategies can be employed based on the application architecture. For a web application that has a back-end, Authorization Code flow can be used. Meanwhile, a web application, such as an SPA, with no back-end, can employ Implicit flow with PKCE. The use of PKCE is particularly important to protect a stolen authorization code from being mis-used. Session management is done through iframes which communicate the session status between each other through existing Web APIs. Furthermore, the browser has robust storage facilities that allow storage of the tokens securely.

Usage of token based identity and access control methods enable Single Sign On capabilities for applications. This is possible because of the structured and consistent format of the tokens. The IdP use tokens for verifying and communicating the identity and session status of its users. It allows a stateless architecture where a session connection does not need to be established between the client and the IdP. However, the application should perform all the necessary verifications and validations on a JWT to verify its authenticity before trusting and using the data that it provides about the user. The verification steps are laid down in the OIDC specifications.

Along with Single Sign On capabilities, OIDC also defines Single Logout methods. Depending upon the architecture, applications can use either a front-channel logout method or a back-channel logout method. Front-channel logout is useful for front-end applications, as the IdP performs logout redirects through the browser. For back-end applications, a back-channel logout is better suited, where the IdP directly opens a logout request with the application.

We also documented some experimental browser-based technologies. The Credentials Management API brings a number of improvements to user experience related to password management. Applications can store the user's passwords in the credential store and can reuse them for subsequent page visits. Furthermore, applications can create a virtually permanent session by silently authenticating with the stored password. The API extends its offerings by supporting federated login credentials. Applications can use it to support for federated login through IdPs such as Facebook and Google. Web Authentication API, which is an extension of Credentials Management API, allows usage of asymmetric key cryptography for user identity. Supported authenticators create a public-private key pair after user consent and identification. The private key is stored in the authenticator, which is used to provide attestations. Servers save the public key and identify the user by verifying

the attestation with it.

Finally, we went through three use cases that were implemented by the user in different professional projects. The Industrial Control Systems case study used token-based access management for granting access to IoT sensor data to its users. The Social Media Center case study followed the authorization principles that are laid down by the social media platforms, namely Twitter, Instagram, Facebook and Google. Finally, we propose the idea of an SDK built around the login component that is provided as a presentational component of a design system.

References

[1] I. Iliev and G. P. Dimitrov. Front end optimization methods and their effect. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 467–473, 2014.

[2] J. Oh, W. H. Ahn, S. Jeong, J. Lim, and T. Kim. Automated transformation of template-based web applications into single-page applications. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 292–302, 2013.

[3] Julian Reschke. The 'Basic' HTTP Authentication Scheme. RFC 7617, September 2015. https://rfc-editor.org/rfc/rfc7617.txt.

[4] Rifaat Shekh-Yusef, David Ahrens, and Sophie Bremer. HTTP Digest Access Authentication. RFC 7616, September 2015. https://rfc-editor.org/rfc/rfc7616.txt.

[5] S. Farrell. Api keys to the kingdom. *IEEE Internet Computing*, 13(5):91–93, 2009.

[6] H. K. Lu. Keeping your api keys in a safe. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 962–965, 2014.

[7] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 1, November 2014. https://openid.net/specs/openid-connect-core-1_0.html.

[8] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. https://rfc-editor.org/rfc/rfc6749.txt.

[9] D. Fett, R. Küsters, and G. Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202, 2017.

[10] K. Dodanduwa and I. Kaluthanthri. Role of trust in oauth 2.0 and openid connect. In *2018 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, pages 1–4, 2018.

[11] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, September 2015. https://rfc-editor.org/rfc/rfc7636.txt.

[12] B. de Medeiros, N. Agarwal, N. Sakimura, J. Bradley, and M. Jones. OpenID Connect Session Management 1.0 - draft 29, July 2020. https://openid.net/specs/openid-connect-session-1_0.html.

[13] Z. Triartono, R. M. Negara, and Sussi. Implementation of role-based access control on oauth 2.0 as authentication and authorization system. In *2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 259–263, 2019.

[14] Justin Richer. OAuth 2.0 Token Introspection. RFC 7662, October 2015. https://rfc-editor.org/rfc/rfc7662.txt.

[15] O. Ethelbert, F. F. Moghaddam, P. Wieder, and R. Yahyapour. 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)*, pages 47–53, 2017.

[16] Steve Mansfield-Devine. Single sign-on: matching convenience with security. *Biometric Technology Today*, 2011:7–11, 07 2011.

[17] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. https://rfc-editor.org/rfc/rfc7519.txt.

[18] N. Sakimura, J. Bradley, M. Jones, and E. Jay. OpenID Connect Discovery 1.0 incorporating errata set 1, November 2014. https://openid.net/specs/openid-connect-discovery-1_0.html.

[19] Eran Hammer-Lahav and Mark Nottingham. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785, April 2010. https://rfc-editor.org/rfc/rfc5785.txt.

[20] Michael Jones. JSON Web Key (JWK). RFC 7517, May 2015. https://rfc-editor.org/rfc/rfc7517.txt.

[21] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. pages 378–390, 10 2012.

[22] M. Jones. OpenID Connect Front-Channel Logout 1.0 - draft 03, July 2020. https://openid.net/specs/openid-connect-frontchannel-1_0.html.

[23] J. Bradley and M. Jones. OpenID Connect Back-Channel Logout 1.0 - draft 05, July 2020. https://openid.net/specs/openid-connect-backchannel-1_0.html.

[24] V. Rastogi and A. Agrawal. All your google and facebook logins are belong to us: A case for single sign-off. In *2015 Eighth International Conference on Contemporary Computing (IC3)*, pages 416–421, 2015.

[25] M. D. Karunanithi and B. Kiruthika. Single sign-on and single log out in identity. In *International Conference on Nanoscience, Engineering and Technology (ICONSET 2011)*, pages 607–611, 2011.

[26] Phil Hunt, Michael Jones, William Denniss, and Morteza Ansari. Security Event Token (SET). RFC 8417, July 2018. https://rfc-editor.org/rfc/rfc8417.txt.

[27] M. Jemel and A. Serhrouchni. Security enhancement of html5 local data storage. In *2014 International Conference and Workshop on the Network of the Future (NOF)*, pages 1–2, 2014.

[28] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23:1–0, 09 2015.

[29] The Web Hypertext Application Technology Working Group (WHATWG) community. HTML Living Standard , July 2020. https://html.spec.whatwg.org/.

[30] The World Wide Web Consortium (W3C). Credential Management Level 1 , January 2019. https://www.w3.org/TR/credential-management-1/.

[31] The World Wide Web Consortium (W3C). Web Authentication: An API for accessing Public Key Credentials Level 1, March 2019. https://www.w3.org/TR/webauthn/.

[32] F. Alqubaisi, A. S. Wazan, L. Ahmad, and D. W. Chadwick. Should we rush to implement password-less single factor fido2 based authentication? In *2020 12th Annual Undergraduate Research Conference on Applied Computing (URC)*, pages 1–6, 2020.

[33] M. Shehab and F. Mohsen. Towards enhancing the security of oauth implementations in smart phones. In *2014 IEEE International Conference on Mobile Services*, pages 39–46, 2014.

[34] William Denniss and John Bradley. OAuth 2.0 for Native Apps. RFC 8252, October 2017. https://rfc-editor.org/rfc/rfc8252.txt.

[35] Twitter Inc. Obtaining user access tokens using 3-legged OAuth. https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a/obtaining-user-access-tokens.

[36] Youtube. YouTube Data API: Channels. https://developers.google.com/youtube/v3/docs/channels.

[37] Youtube. YouTube Data API: Videos. https://developers.google.com/youtube/v3/docs/videos.

[38] Youtube. YouTube Data API: Comments. https://developers.google.com/youtube/v3/docs/comments.

[39] Facebook Inc. Facebook Graph API. https://developers.facebook.com/docs/graph-api/.