Prepared by:

**Poplin Working Group**

**MITA Reference Architecture**

Service Definition: Security

Draft

Version 0.3

March 30, 2018

Record of Changes

| Version | Date | Author / Owner | Description of Change |
| --- | --- | --- | --- |
| 0.1 | 03/09/2018 | Dave Hill | Initial draft |
| 0.2 | 03/20/2018 | Dave Hill | Integrated first round feedback |
| 0.3 | 03/30/2018 | Dave Hill | Integrated feedback from Technical Review Board |

Table of Contents

[1. Introduction 1](#_Toc509412293)

[1.1 Purpose 1](#_Toc509412294)

[1.2 Audience 2](#_Toc509412295)

[1.3 Document Organization 2](#_Toc509412296)

[1.4 Reference Material 2](#_Toc509412297)

[2. Process Model 4](#_Toc509412298)

[2.1 OAuth 2.0 Overview 4](#_Toc509412299)

[2.1.1 Actors and Roles 4](#_Toc509412300)

[2.1.2 OAuth 2.0 Authorization Flow 4](#_Toc509412301)

[2.1.3 Proof Key for Code Exchange 5](#_Toc509412302)

[2.2 OpenID Connect Overview 5](#_Toc509412303)

[2.2.1 Actors and Roles 5](#_Toc509412304)

[2.2.2 OpenID Connect Authentication Flow 6](#_Toc509412305)

[2.3 Example Use Case 7](#_Toc509412306)

[2.4 OAuth 2.0, OpenID Connect Authentication / Authorization Sequence 7](#_Toc509412307)

[2.4.1 Initial Request 8](#_Toc509412308)

[2.4.2 Another Request With Valid Access Token 9](#_Toc509412309)

[2.4.3 Another Request After Access Token Expires 9](#_Toc509412310)

[2.4.4 Subsequent Requests 9](#_Toc509412311)

[2.5 Client Registration 9](#_Toc509412312)

[2.6 Dynamic Client Registration 10](#_Toc509412313)

[2.7 Web Portal Use Case 11](#_Toc509412314)

[2.8 Delegation Use Case 13](#_Toc509412315)

[2.9 Service-to-Service Use Case 15](#_Toc509412316)

[2.10 Refresh Tokens 16](#_Toc509412317)

[2.11 Revoking Tokens 17](#_Toc509412318)

[2.12 Caching Opportunities 18](#_Toc509412319)

[2.12.1 Resource Server: Access Token Cache 18](#_Toc509412320)

[2.12.2 Clients and Resource Server: Discovery Information Cache 19](#_Toc509412321)

[2.12.3 Resource Server: Introspection Cache 19](#_Toc509412322)

[3. Object Model 20](#_Toc509412323)

[3.1 JSON Web Token 20](#_Toc509412324)

[3.1.1 Standard JWT Claims 20](#_Toc509412325)

[4. Resource Model 22](#_Toc509412326)

[4.1 ID Token 22](#_Toc509412327)

[4.1.1 ID Token Claims 22](#_Toc509412328)

[4.2 Authorization Grant 23](#_Toc509412329)

[4.3 Access Token 23](#_Toc509412330)

[4.3.1 Access Token Claims 24](#_Toc509412331)

[4.4 JSON Web Key (JWK) 24](#_Toc509412332)

[5. API Specification 26](#_Toc509412333)

[5.1 Authentication Service (OpenID Connect) 26](#_Toc509412334)

[5.1.1 Discovery Endpoint 26](#_Toc509412335)

[5.1.2 Token Endpoint 28](#_Toc509412336)

[5.1.3 UserInfo Endpoint 30](#_Toc509412337)

[5.2 Authorization Service (OAuth 2.0) 32](#_Toc509412338)

[5.2.1 Discovery Endpoint 32](#_Toc509412339)

[5.2.2 Registration Endpoint 33](#_Toc509412340)

[5.2.3 Authorization Endpoint 36](#_Toc509412341)

[5.2.4 Token Endpoint 39](#_Toc509412342)

[5.2.5 Introspection Endpoint 44](#_Toc509412343)

[5.2.6 Revocation Endpoint 46](#_Toc509412344)

[5.2.7 Callbacks 47](#_Toc509412345)

[6. Message Specification 49](#_Toc509412346)

[6.1 Unauthorized Access Message 49](#_Toc509412347)

[6.1.1 Queue Type 49](#_Toc509412348)

[6.1.2 Message Payload 49](#_Toc509412349)

[Appendix A: Use Case Example 50](#_Toc509412350)

**Figures**

**No table of figures entries found.**

**Tables**

[Table 1. Document Organization 2](#_Toc509316097)

# Introduction

The Poplin project is a Medicaid reference architecture being developed by key states, vendors, and other stakeholders, under the Medicaid Information Technology Architecture (MITA) Governance Board and sponsored by the Centers for Medicare and Medicaid Services (CMS). Poplin defines a standard, extensible set of business processes, object definitions, and application programming interfaces (APIs) for State Medicaid Agencies. The Poplin [web site](http://projectpoplin.org/) and [GitHub site](https://github.com/MITA-Governance-Board/Poplin) provides further detail about the Poplin reference architecture and implementation. This paper defines a *Security* shared service for Poplin.

## Purpose

The security shared service is responsible for ensuring that resources and data transfers between services, both within the enterprise and across organizational boundaries are protected against unauthorized access.

The security shared service uses a variety of RESTful open security standards, including Transport Layer Security (TLS) v1.2 X.509 v3, OAuth 2.0 for authorization, OpenID Connect for authentication and identity federation, and JSON Object Signature and Encryption (JOSE, pronounced ‘ho-ZAY’) keys, signatures, algorithms, and encryption strategies to generate secure JSON Web Tokens (JWT, pronounced ‘jawt’) for access. User Managed Access (UMA) is another technology built on top of this stack to provide patient privacy and consent capabilities and is beyond the scope of the security shared service.

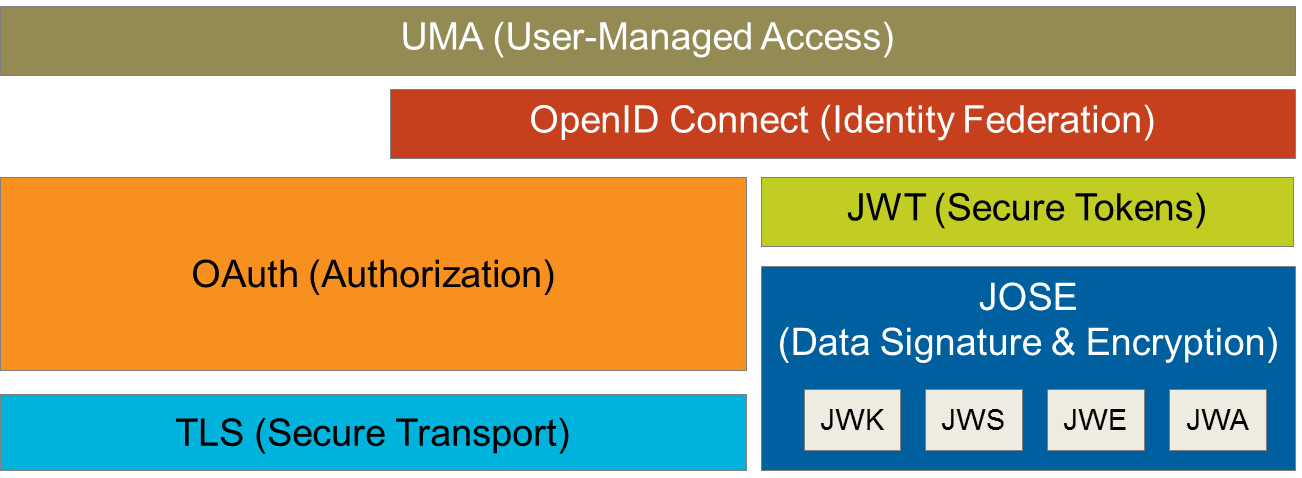


Figure 1‑1: Open Security Standards for RESTful APIs

The technology stack outlined in Figure 1-1 provides the foundation for a flexible, scalable security framework already in wide-spread use across the internet. The stack can provide federated access and single sign-on capabilities across organizations, the ability for a user to delegate access to a desktop or mobile application (e.g. for health tracking), and the ability to dynamically add clients to securely access owned resources.

This service definition leverages the work of the HEART Working Group, Blue Button 2.0, and the ONC Trusted Exchange Framework in using these technologies to provide a nationwide environment of trust for health information exchange, both within and potentially beyond the realm of Medicaid.

## Audience

The entire Poplin reference architecture and implementation are open source and available to the public ([Poplin GitHub site](https://github.com/MITA-Governance-Board/Poplin)). The primary audience of the Poplin service definitions are State Medicaid Agencies (SMAs) and their vendors, specifically system architects, developers, and IT operations personnel supporting their state’s move to a modular, microservices-based enterprise architecture. The Poplin shared services are not specific to the Medicaid domain and could be leveraged by any enterprise.

## Document Organization

This document is organized in the following sections:

Table 1. Document Organization

| Section | Purpose |
| --- | --- |
| Section 2: Process Model | Describes the dynamic aspects of the service area |
| Section 3: Object Model | Describes the logical aspects of the service area |
| Section 4: Resource Model | Describes the important elements or key abstractions for the service area |
| Section 5: API Specification | Describes how the API should look structurally for the service area |
| Section 6: Message Specification | Describes asynchronous messages generated by the service area |

## Reference Material

The following reference material may be useful to better understand the goals and objectives of OAuth2 and OpenID Connect protocols and how they operate:

* [“Draft Trusted Exchange Framework”](https://www.healthit.gov/sites/default/files/draft-trusted-exchange-framework.pdf) – ONC, January 5, 2018
* [“Blue Button 2.0 Authorization”](https://bluebutton.cms.gov/developers/#authorization) – CMS, visited March 27, 2018
* [“OAuth2 in Action”](https://www.manning.com/books/oauth-2-in-action), Justin Richer, Antonio Sanso – Manning Publications, 2017
* [“Use Cases for Argonaut Project”](http://argonautwiki.hl7.org/images/4/4c/Argonaut_UseCasesV1.pdf) – Version 1.0, HL7 Argonaut Project, May 26, 2015
* [“Health Relationship Trust Profile for OAuth2”](http://openid.net/specs/openid-heart-oauth2-1_0-ID2.html) – Version 1.0, HEART Working Group, April 25, 2017
* “Health Relationship Trust Profile for Fast Healthcare Interoperability Resources (FHIR) OAuth 2.0 Scopes” – HEART Working Group, May 25, 2017
* [“Health Relationship Trust Profile for OpenID Connect”](http://openid.net/specs/openid-heart-openid-connect-1_0-ID2.html) – Version 1.0, HEART Working Group , October 3, 2016
* [“Argonaut App-Authorization Profile Alignment with HEART Profiles”](http://argonautwiki.hl7.org/images/b/b7/%282015Dec30%29HEARTcomparisonV3.pdf) – HL7 Argonaut Project, December 30, 2015
* [“The OAuth 2.0 Authorization Framework”](https://tools.ietf.org/html/rfc6749) – RFC 6749, Internet Engineering Task Force (IETF), October 2012
* [“The OAuth 2.0 Authorization Framework: Bearer Token Usage”](https://tools.ietf.org/html/rfc6750) – RFC 6750, IETF, October 2012
* [“An IETF URN Sub-Namespace for OAuth”](https://tools.ietf.org/html/rfc6755) – RFC 6755, IETF, October 2012
* [“OAuth 2.0 Threat Model and Security Considerations”](https://tools.ietf.org/html/rfc6819) – RFC 6819, IETF, January 2013
* [“OAuth 2.0 Token Revocation”](https://tools.ietf.org/html/rfc7009) – RFC 7009, IETF, August 2013
* [“OpenID Connect Core 1.0”](http://openid.net/specs/openid-connect-core-1_0.html) – OpenID Foundation, November 8, 2014
* [“JSON Web Key (JWK)”](https://tools.ietf.org/html/rfc7517) – RFC 7517, IETF, May 2015
* [“JSON Web Token (JWT)”](https://tools.ietf.org/html/rfc7519.html) – RFC 7519, IETF, May 2015
* [“Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants”](https://tools.ietf.org/html/rfc7521) – RFC 7521, IETF, May 2015
* [“JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants”](https://tools.ietf.org/html/rfc7523) – RFC 7523, IETF, May 2015
* [“OAuth 2.0 Dynamic Client Registration Protocol”](https://tools.ietf.org/html/rfc7591) – RFC 7591, IETF, July 2015
* [“OAuth 2.0 Dynamic Client Registration Management Protocol”](https://tools.ietf.org/html/rfc7592) – RFC 7592, IETF, July 2015
* [“Proof Key for Code Exchange by OAuth Public Clients”](https://tools.ietf.org/html/rfc7636) – RFC 7636, IETF, September 2015
* [“OAuth 2.0 Token Introspection”](https://tools.ietf.org/html/rfc7662) – RFC 7662, IETF, October 2015
* [“Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)”](https://tools.ietf.org/html/rfc7800) – RFC 7800, IETF, April 2016

# Process Model

The authorization process through OAuth 2.0 and authentication process through OpenID Connect involve several actors and roles in order to provide secured access to protected resources.

## OAuth 2.0 Overview

The following is a basic overview of the OAuth 2.0 protocol. For more detailed information about OAuth2 and how it works, please see the reference material in Section 1.4.

### Actors and Roles

#### Resource Owner

The resource owner is a user with the authority to authorize clients to access protected resources on his/her behalf, such as a mobile application.

#### Client Application

The client application is a software application that requests OAuth access tokens from an authorization server to access protected resources on the behalf of the resource owner. The client application could be a web application accessed through a browser, an application on a mobile device, or a separate client service.

#### Protected Resource

The protected resource is an API, application, or data for which authorization must be granted in order to access the resource. Generally, a “resource server” manages access to the protected resources.

#### Authorization Server

The authorization server is a service that implements the OAuth 2.0 protocols and endpoints, and issues authorization grants and access tokens to clients.

### OAuth 2.0 Authorization Flow

The OAuth 2.0 Authorization Flow is shown in Figure 2‑1 and starts with the Resource Owner accessing a client application to retrieve a protected resource that they own. The client can be a web portal, mobile application, or client service. If the client does not yet have authorization, it redirects the resource owner to the authorization endpoint (AE on the diagram) of the authorization server. The resource owner authenticates with the authorization server and, if valid, the authorization server returns an authorization grant code and redirects the resource owner back to the client.

The client uses the authorization grant code when calling the token endpoint (TE on the diagram) to get an access token required to retrieve the protected resource. The authorization grant code ties the client request to the original authentication by the resource owner. The authorization server takes that information and, if valid, generates an access token and returns it to the client. The client then provides the access token the protected resource server to retrieve the resource. If the access token is valid for the request, the requested resource is returned to the client.

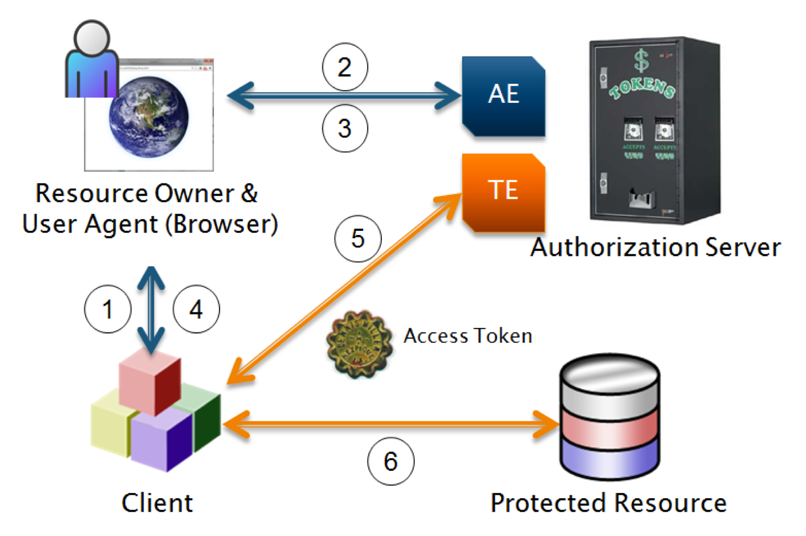


Figure 2‑1: OAuth 2.0 Control Flow[[1]](#footnote-1)

## OpenID Connect Overview

The following is a basic overview of the OpenID Connect protocol, which is based on OAuth 2.0 For more detailed information about OpenID Connect and how it works, please see the reference material in Section 1.4.

### Actors and Roles

OpenID Connect is based on the OAuth 2.0 protocol, so the actors are similar, but play slightly different roles.

#### End User

The end user is the entity that is looking to authenticate with the identity provider. In comparison to the OAuth 2.0 roles, the end user acts as the resource owner (owning their credentials).

#### Identity Provider

The identity provider is an OpenID Connect service that authenticates the resource owner and generates a signed identity token for subsequent system accesses. The identity token is used for subsequent system accesses, replacing the need for each system to ask the resource owner for credentials.

In comparison to the OAuth 2.0 roles, the identity provider acts as both the authorization server and the protected resource server (user information).

#### Relying Party

The relying party is a system or application that consumes authentication information from an identity provider (e.g., an OpenID Provider). In other words, the relying party relies on the identity provider to authenticate the user.

In comparison to the OAuth 2.0 roles, the relying party acts as the client application seeking to gain access to the protected resource (user information).

### OpenID Connect Authentication Flow

OpenID Connect is based on OAuth 2.0, so the OpenID Connect authentication flow is almost identical to the OAuth 2.0 authorization flow, with some notable exceptions, as shown in Figure 2‑2. As before, when the end user accesses the relying party without a token, the call is redirected to the authorization endpoint of the OpenID provider. The OpenID provider authenticates the end user by asking for their credentials. If the credentials are valid, an authorization grant is returned to the relying party. The relying party can then call the token endpoint of the OpenID provider with the authorization grant to get tokens.

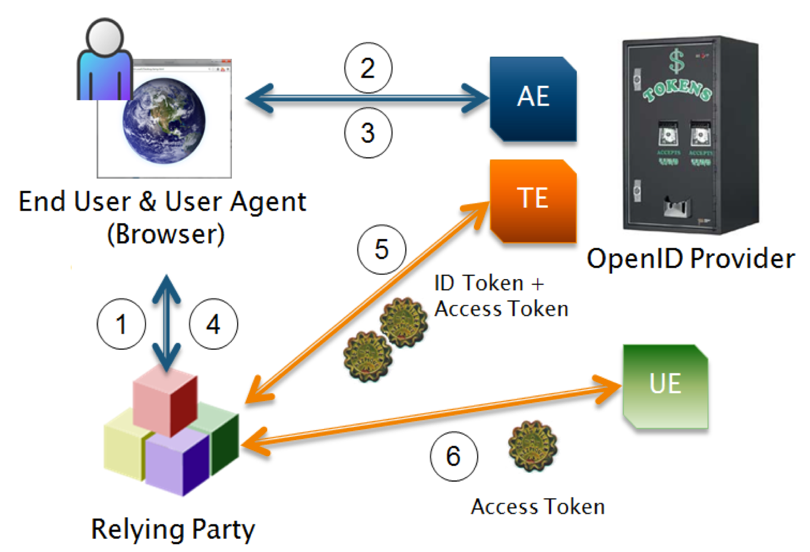


Figure 2‑2: OpenID Connect Control Flow[[2]](#footnote-2)

When tokens are returned to the relying party, two tokens are returned instead of just one – an ID token and the access token. Two tokens are returned because each token is intended for a different audience. The access token is an opaque token that the OpenID provider and OAuth authorization server use to determine access capabilities. The intended audience for the ID Token is the relying party.

The relying party can call the UserInfo endpoint (UE in the diagram) to retrieve more information about the user. The amount of information returned can vary across identity providers, but includes the name of the user and how and when they authenticated.

## Example Use Case

A use case example can help illustrate OAuth 2.0 and OpenID Connect capabilities and how they can be used to authorization and authenticate accesses across multiple security domains. A sample use case is described in Appendix A.

## OAuth 2.0, OpenID Connect Authentication / Authorization Sequence

The OAuth 2.0 authorization and OpenID Connect authentication sequence can involve multiple interactions and coordination between all of the actors defined in the previous section. Figure 2‑3 shows the interactions for several color-coded requests that are covered individually below.

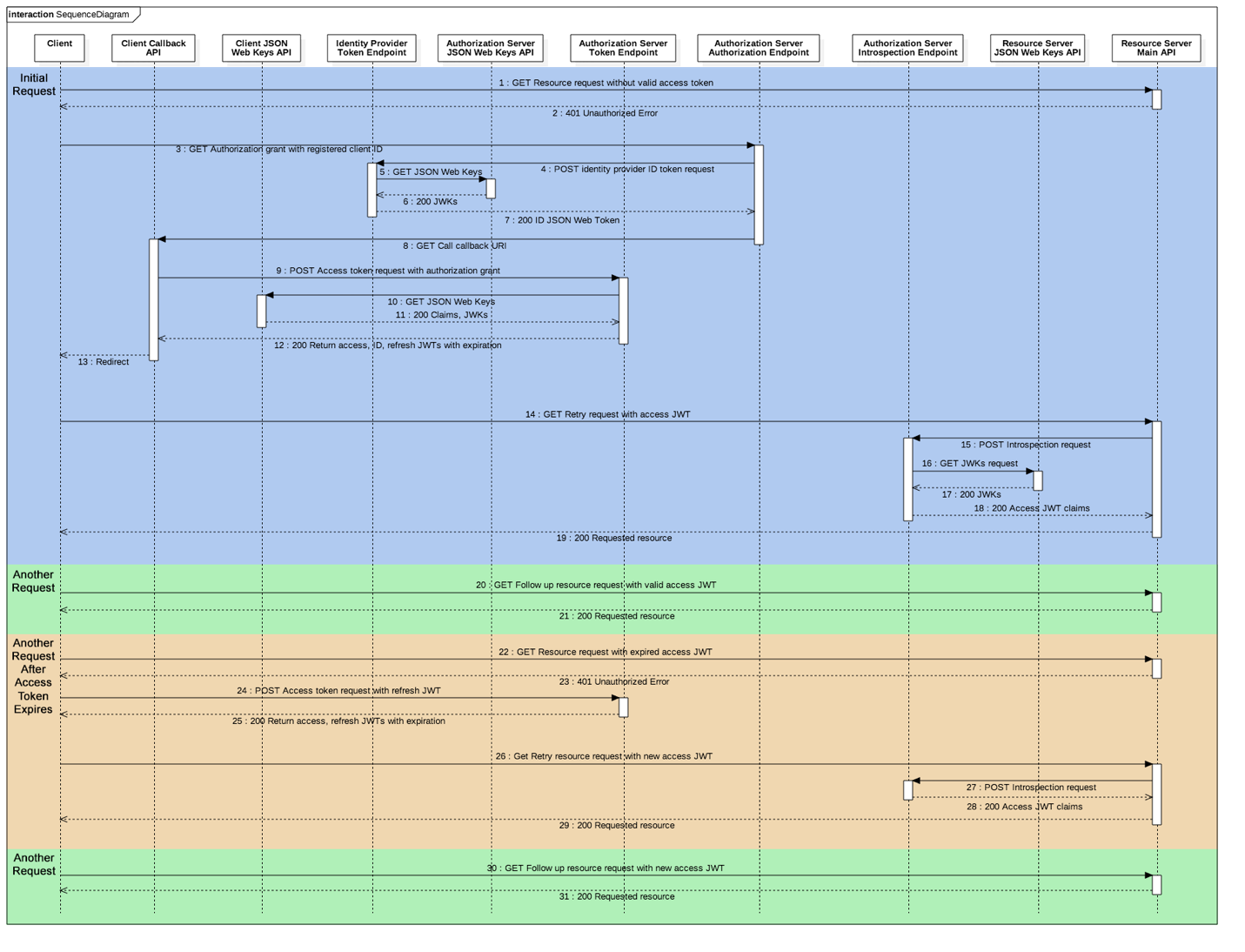


Figure 2‑3: OAuth2/OpenID Connect Sequence Diagram

### Initial Request

The sequence in blue in Figure 2‑3 begins with the client accessing the resource server for a protected resource without a valid access token (step #1). The resource responds with an unauthorized error (step #2). The client responds to the error by calling the authorization server to get an authorization grant code (step #3) with a callback Uniform Resource Identifier (URI) that the authorization server can call when the authorization grant code has been generated.

When the authorization server receives the authorization grant code request from the client, it calls the identity provider to request an ID token for the resource owner (step #4). The identity provider can use any method it wants to authenticate the user, from requesting a simple username and password to two-factor authentication or applying more rigorous processes (e.g. biometric scans). Once the identity provider is satisfied by the identity of the resource owner, the identity provider calls the authorization server to retrieve its public key set as JSON Web Keys (steps #5, #6), if it hasn’t already been cached. The identity provider generates an ID token in JSON Web Token (JWT) format is generated, signs it with the identity provider’s private key, encrypts the token using the authorization server’s public key, and returns the result to the authorization server as evidence of that authentication (step #7).

With authentication complete, the authorization server uses its private key to decrypt the ID token and verifies that the ID token is valid. The authorization server can also request from the resource owner a list of permissions they want to grant to the client on their behalf. From that information, the authorization generates an authorization grant code which is included in a parameter to the callback URI that the client provided with the request (step #8).

The callback code provided by the client takes the authorization grant code and calls the authorization server to request an access token with permission claims encoded in a JSON Web Token, signed by the client’s private key (step #9). When that request is received by the authorization server, it calls the clients JSON Web Key endpoint (steps #10, #11) to retrieve the client’s public keys to decrypt the JSON Web Token. If the authorization grant code and the claims are valid, the authorization server generates an access token (JWT) incorporating the grant information from the resource owner, signed with the authorization server’s private key, and returns the access token and ID token to the client. A refresh token (JWT) is also generated and signed by the authorization server and returned to allow the client to request new access tokens later without involving the resource owner (step #12).

Now that the client has an access token, it retries the request to the resource server with that access token (step #14). The resource server receives the request and posts an introspection request (step #15), including the access token, to the authorization server. The authorization server deciphers the access token and returns the permission claims, state, expiration time, and other information about the token to the resource server (step #18). If the access token is valid and the permission claims cover the type of request, the resource server returns the protected resource to the client (step #19).

### Another Request With Valid Access Token

After the initial request, the client may make other requests for protected resources with the access token, highlighted in green in Figure 2‑3 (step #20). The resource server validates the token and has cached the claims received from the introspection call during the initial request, so it does not need to call the authorization server again and can directly validate the request against the cached claims of the access token. If the request is valid, it returns the requested resource to the client (step #21).

### Another Request After Access Token Expires

If the client makes a request to the resource server after the access token has expired, the client can use the refresh token to get a new access token without involving the resource owner. This sequence is highlighted in orange in Figure 2‑3.

To request a new access token, the client calls the token endpoint of the authorization server with the refresh token. The authorization server looks up and validates the refresh token. If the refresh token is valid, the authorization server generates a new access token with the same claims as the original access token and returns it to the client along with the refresh token.

The client uses the new access token to retry the request. The resource server receives the access token, discovers it is new, and calls the introspection endpoint of the authorization server to see if the access token is valid for the request. If the access token is valid, it caches the introspection data for the token and returns the resource to the client. This time, the authorization server does not need to call the JWK endpoint of the resource server since it was cached during the initial (blue) request.

### Subsequent Requests

Subsequent requests with the new access token follow the (green) process described in Section 2.4.2 until the new access token expires, which would require another refresh as described in Section 2.4.3. When the refresh token expires, the initial access process described in Section 2.4.1 needs to be repeated.

### Proof Key for Code Exchange

An additional provision has been added to the OAuth 2.0 control flow for mobile applications. When a mobile application requests an authorization grant code, the request is accomplished through a secure API with TLS encryption, making it difficult for bad actors to intercept the authorization grant code. When the authorization server generates the authorization grant code, the code is returned through the redirection URI provided by the mobile application, again encrypted through TLS. However, within the smartphone, a malicious mobile application can register to handle the incoming redirection along with the legitimate application. Since interapplication communication within the smartphone operating system is not protected through TLS, the malicious application can gain access to the authorization grant code and subsequently obtain an access token for the protected resource.

To prevent a malicious mobile application from using the authorization grant code, the legitimate application includes a Proof Key for Code Exchange (PKCE – pronounced ‘pixy’) with the authorization grant code request to the authorization server. The authorization server saves the PKCE code and does not include it in the call to the redirection URI. When a subsequent request is made for an access token based on that authorization grant code, the authorization server looks for a matching PKCE code in the request for the access token. If the PKCE code is missing or incorrect, the authorization server refuses the request.

Since the PKCE code is generated and sent only by the legitimate mobile application and not returned by the authorization server in the redirection URI call, the malicious application does not have the opportunity to intercept the PKCE code. Without a valid PKCE code, the malicious application cannot get an access token for the protected resource.

## Client Registration

Clients of the authorization server are identified by client IDs which, in the case of static client registration, is setup ahead of time between the client and the server. The server provides the client an ID as part of that relationship, which needs to be included by the client when calls to the authorization server are made.

Static client registration works well when the relationship between clients and authorization servers do not change very much. The relationship is established by the user signing up for the authorization service or having an administrator setup the account.

## Dynamic Client Registration

As the number of clients increases along with resource servers and authorization servers across multiple organizations, static client registration can become burdensome and inflexible. Under the Poplin reference architecture, to provide nationwide interoperability, a client may need to access several similar services from a large number of separate state and federal agencies. Each of those agencies will likely have its own authorization server, which makes setting up static accounts ahead of time challenging. Each authorization server will have its own client ID for a given client, further adding complexity as clients access systems across authorization boundaries. Adding new clients to the ecosystem would require effort within all of those organizations to setup prior static relationships. In these situations, manual registration does not scale well.

Consider this extreme example based on email:

*“Would it be reasonable for a developer to register each copy of an email client with each potential email service provider before shipping the software? After all, every single domain and host on the internet could have its own separate mail server, not to mention intranet mail services. It’s clear that this is not at all reasonable, but this is the assumption made with manual registration in OAuth.”*[[3]](#footnote-3)

Many other security mechanisms designed for single enterprises make similar assumptions.

Note that dynamic registration only allows the client to begin a relationship with the authorization server. Ultimately, it is still up to the authorization server to determine what resources that client can access and subject to resource owner credentials. The activity diagram for dynamic registration is shown in Figure 2-4.

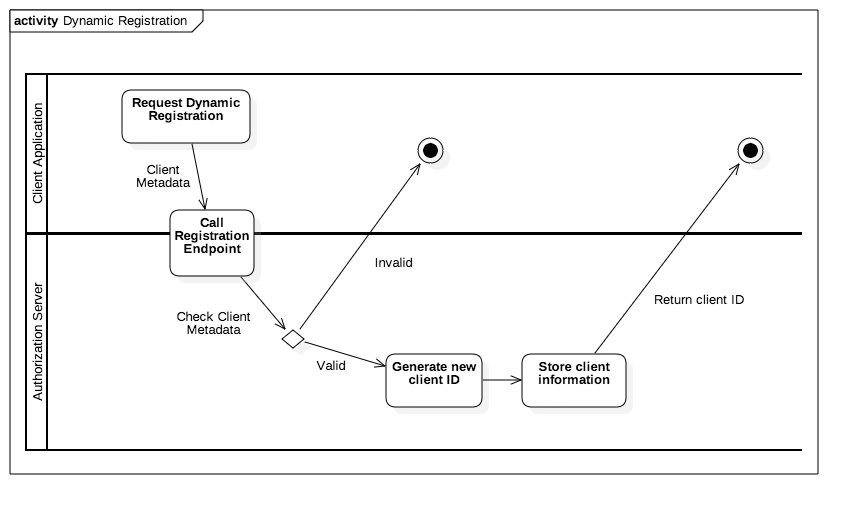


Figure 2‑4: Web Portal Use Case Activity Diagram

## Web Portal Use Case

In this use case, the resource owner accesses resources directly through a web browser, as shown in Figure 2‑5.

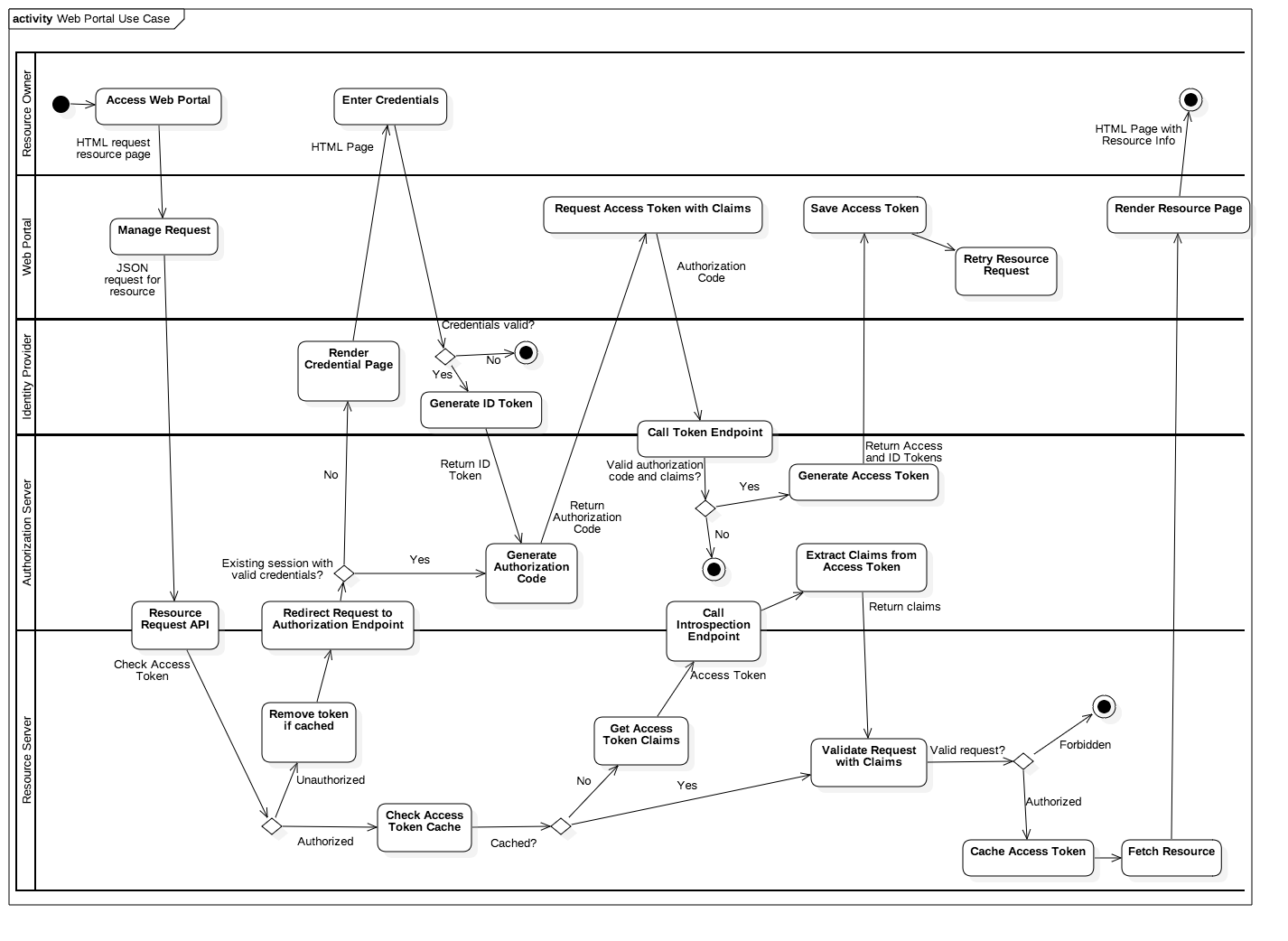


Figure 2‑5: Web Portal Use Case Activity Diagram

The web browser makes a request to the web application to provide information about a resource of interest. The web application then attempts to retrieve the necessary information to satisfy the request from the resource server. When receiving the request, resource server checks to see if a valid access token associated with the request.

If the access token has expired or is not present, the resource server redirects the call to the authentication endpoint of the authorization server to retrieve an authorization grant code. The authorization grant code ties the client request to the original authentication by the resource owner.

When the authorization server receives the redirected call to its authorization endpoint, the authorization server presents a form to the resource owner to enter their credentials. OAuth 2 does not specify what credentials are required. It could be as simple as user name and password or through a more thorough multi-factor authentication process.

Once the resource owner submits their credentials, the authorization server verifies the credentials and if authorized, returns an authorization code to the web application. The web application uses that authorization code to call the token endpoint of the authorization server to request an access token for retrieving the resource.

When the authorization server receives the access token request on the token endpoint, it verifies that the authorization code. If the authorization code is valid, it will generate an access token and return it to the web application, which should save that token for later accesses to the resource.

Now that the web application has an access token, it retries the request to the resource server with that access token. The resource server verifies the access token, which should be valid this time, and tries to find the token in its token cache. Since this is a new token, it would not appear in the cache, so the resource server calls the introspection endpoint of the authorization server to determine what resources can be accessed through that token.

When the authorization server receives the access token through the introspection endpoint, after verifying the token, it extracts the claim information encrypted into the token. The claim information indicates what information that token is authorized to access. The authorization server returns the claim information to the resource server.

Once the resource server receives the claim information for the access token, it verifies that the resource the request is looking to retrieve falls within the scope of the claim. If it does, the token and its claims are cached to speed future processing and the resource is returned. If not, the resource server responds that the request is forbidden.

## Delegation Use Case

The delegation use case allows the resource owner to delegate access to a client acting on their behalf. Such a client could be a mobile application that needs to access information to perform an analysis or provide an easily digested dashboard. This use case is shown in Figure 2‑6.

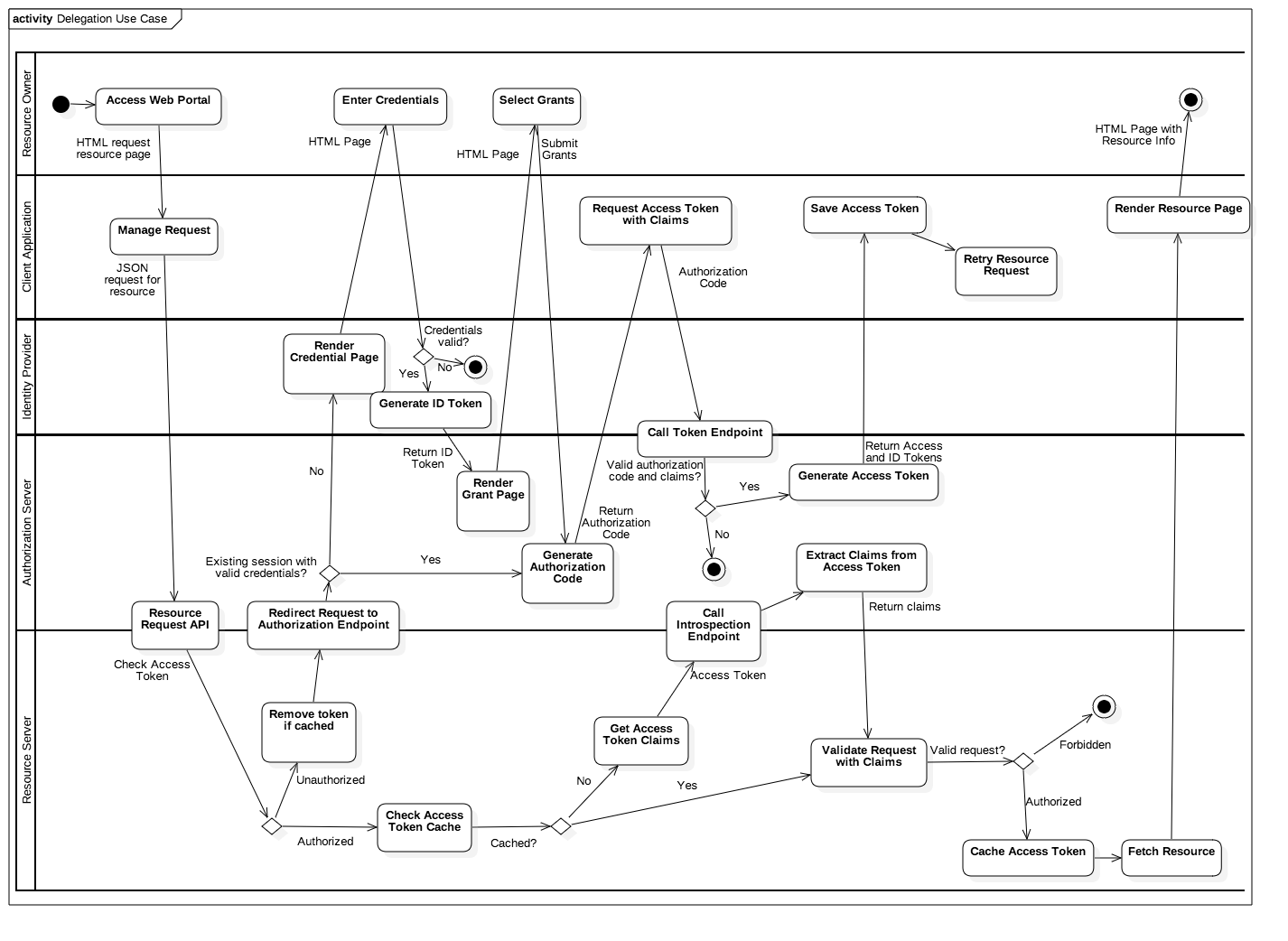


Figure 2‑6: Delegation Use Case Activity Diagram

The progression through the use case is similar to the web portal use case discussed in the last session, but with the additional step of the authorization server asking the resource owner to select a series of “grants” indicating what types of information the client is able to access.

The screen provided by the authorization server is similar to what you might see if one uses Facebook or Google to sign in to other web sites instead of entering a user name and password, as shown in Figure 2‑7.

In the Facebook example, the resource owner is presented with an inflexible decision, either to “Allow” or “Don’t Allow” access to all of the listed resources or to reject client access to those resources altogether. Other grant authorization screens allow the resource owner to specifically select what the client can access and what it cannot. It is preferable to provide the resource owner with flexible choices when possible.

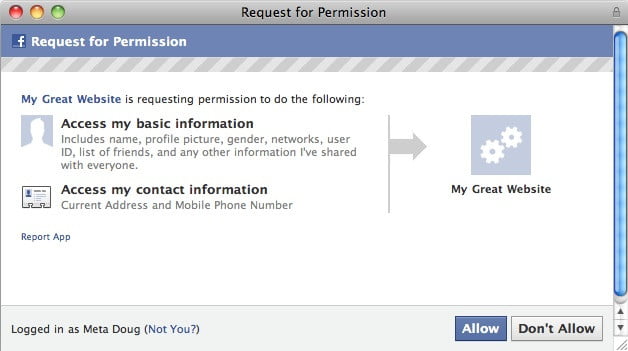


Figure 2‑7: Example Facebook OAuth 2.0 Authorization Grant Dialog[[4]](#footnote-4)

## Service-to-Service Use Case

The service-to-service use case differs from the use cases discussed so far in that there is no user interface involved and the service needs to be provide its credentials to access the information it needs. Otherwise, the activities are similar to the previous use cases.

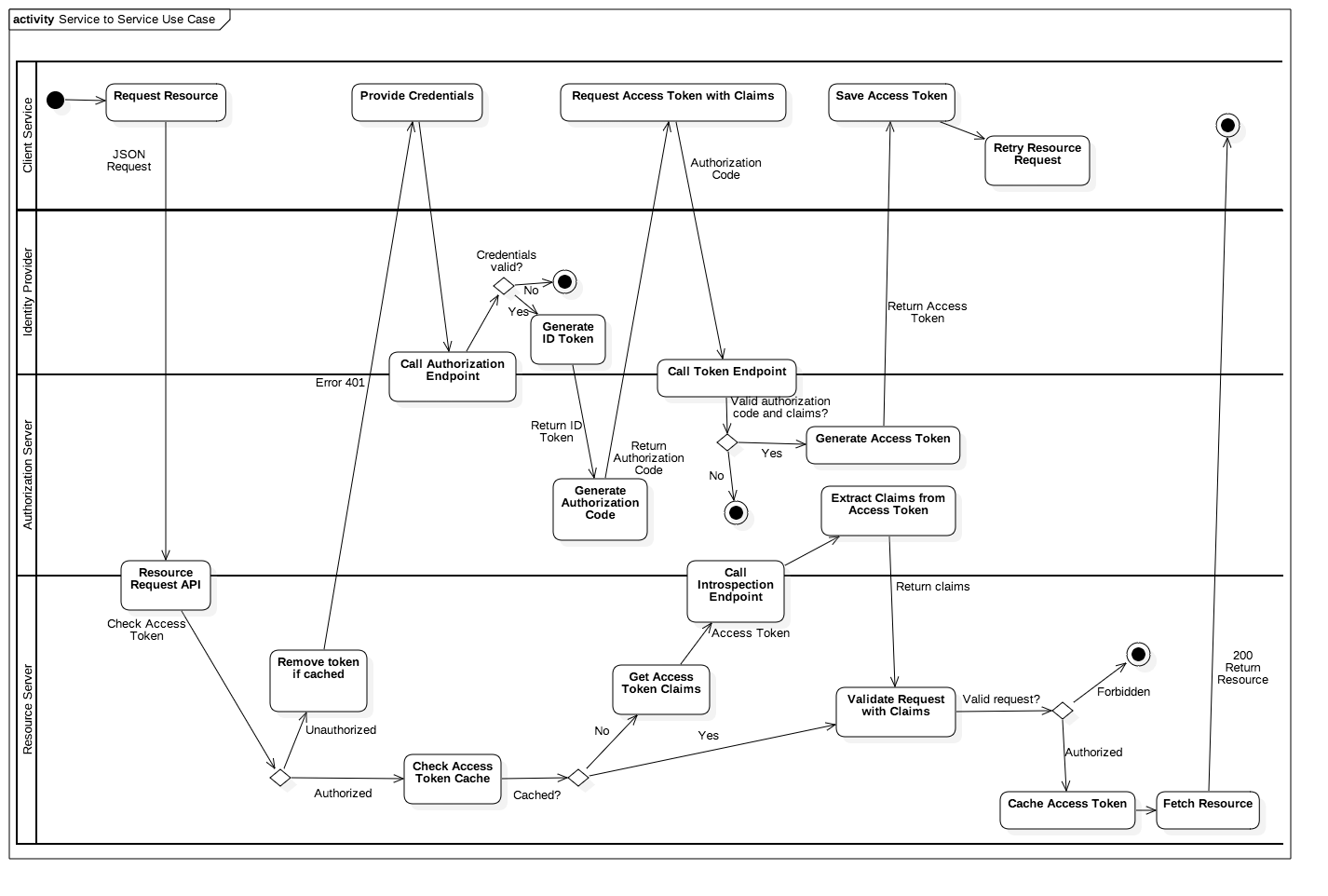


Figure 2‑8: Service-to-Service Use Case Activity Diagram

## Refresh Tokens

Refresh tokens provide a way for clients to replace expired access tokens without having to ask the resource owner to re-enter their credentials. As previously outlined, if a request is made with an expired access token, the call is redirected by the resource server to the authorization server to re-authorize the request. Through refresh tokens, the client can directly obtain a fresh access token by calling the token endpoint of the authorization server. The refresh use case is shown in Figure 2‑9.

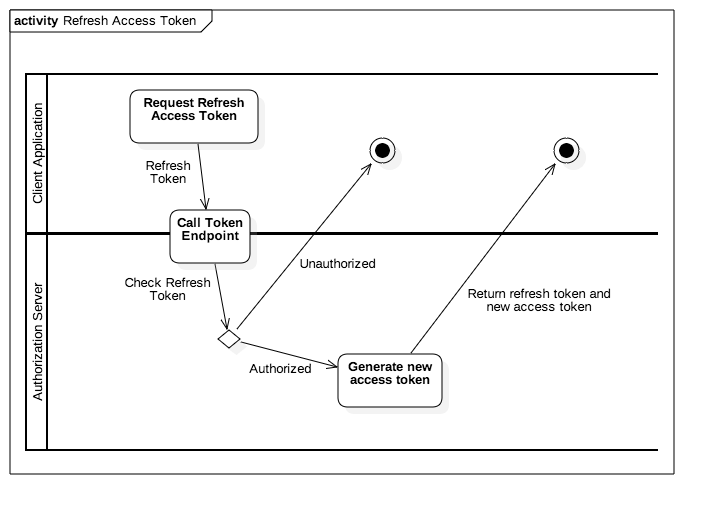


Figure 2‑9: Activity Diagram for Refreshing an Access Token

When servicing an access token request, the authorization server may also include a refresh token in the response that the client can use to retrieve a new access token when the current access token has expired.

## Revoking Tokens

When a client is done using an access token, it should call the token revocation endpoint of the authorization service to remove the access token from future use. The authorization service always responds with a successful response so that information about other tokens are not exposed. The revocation use case is shown in Figure 2‑10.

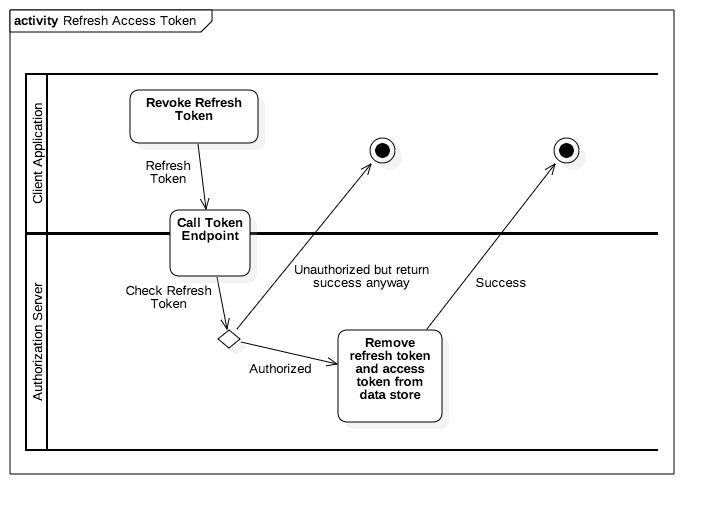


Figure 2‑10: Activity Diagram for Revoking a Refresh Token

## Caching Opportunities

Numerous caching opportunities are available to improve performance when using the security shared service, for both client applications and services that ultimately provide the requested resources.

### Resource Server: Access Token Cache

By leveraging caching and appropriately short access token expirations, subsequent requests with valid access tokens can be fulfilled by the resource server directly with minimal overhead. The resource server can cache valid access tokens with associated claims so that calls to the authorization server are not necessary, speeding performance.

Expiration times for access tokens must be carefully managed to limit gaps when an access token is invalidated due to an adverse event, but still cached in the resource server. The resource server must check the expiration time for the token and remove expired tokens from its cache.

An example expiration time for an access token could be 60 seconds, which would allow quick follow-up requests to resources while keeping exposure during adverse events small.

The green pathway through the Service-to-Service use case diagram in Figure 2‑11 demonstrates how the authorization process around a request can be streamlined through the path highlighted in green.

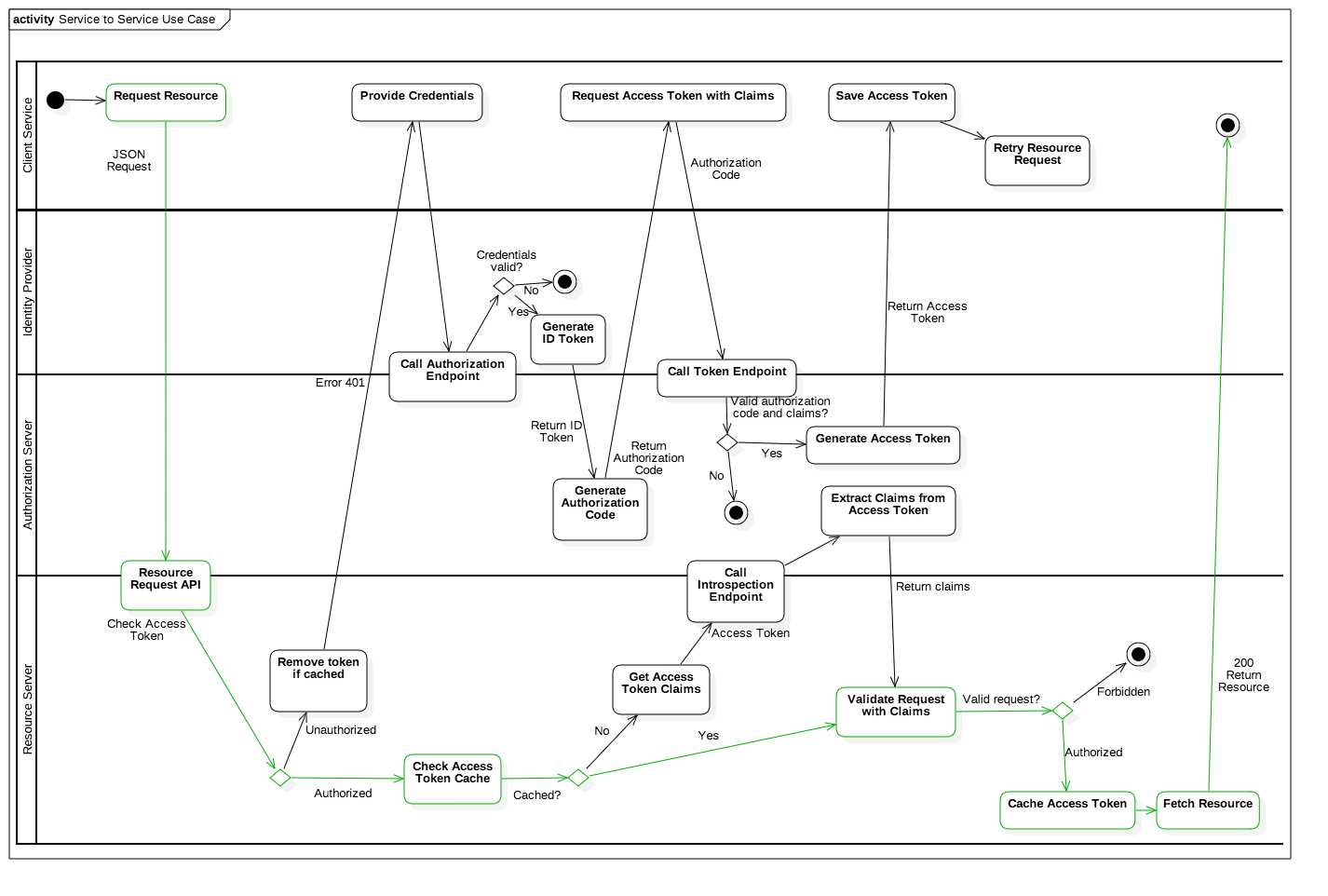


Figure 2‑11: How Caching Can Streamline Authorization

### Clients and Resource Server: Discovery Information Cache

The authorization server and identity provider provide a Discovery Endpoint at a standard URL to inform clients what type of interactions are possible and the URLs of the various endpoints (e.g. authorization endpoints, token endpoints, JWK sets, etc.) This information does not change and should be cached by the clients of these services to reduce the number of HTTP roundtrips required during future requests.

### Resource Server: Introspection Cache

Introspection information for a particular access token remains the same throughout the life of the access token, so the resource server should cache that information so that future requests with that access token can be evaluated without an extra HTTP call to the authorization server.

# Object Model

## JSON Web Token

JSON Web Tokens (JWT) are comprised of three pieces: (1) a base set of fields that identifies it as JSON web token, (2) the data payload associated with the token, and (3) a signature. Each of these pieces is separated by a period (‘.’), which gives the JWT the following encoded format:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ

zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4

gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHM

I6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o

When decoded, the red letters form the header for the JWT:

{

“alg”: “HS256”, The type of algorithm used to encrypt the JWT

“typ”: “JWT” JSON web token

}

The green letters and numbers form the payload of the token:

{

“sub”: “1234567890”,

“name”: “John Doe”,

“iat”: 1516239022

}

Finally, the blue letters and numbers are the verify signature:

HMACSHA256(

base64UrlEncode(header) + “.” +

base64UrlEncode(payload),

secret

)

### Standard JWT Claims

Within the payload, a JSON web token can contain a variety of claims that can change with the intended use of the token. For example, OpenID Connect tokens contain different claims than standard OAuth2 tokens. The following table outlines the standard claims found in JWTs used in OAuth2 and OpenID Connect.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| iss | string | Issuer - URL of entity that issued the token. |
| sub | string | Token subject, machine-readable string unique for that subject by that issuer. |
| aud | string | Token audience, for whom the token is intended. |
| exp | number | Expiration time of token in epoch time. |
| nbf | number | Not before – time that token starts to be accepted in epoch time. |
| iat | number | Issued at - Time token was issued in epoch time. |
| jti | string | JWT ID – case-sensitive unique identifier of token |

For OAuth2 and OpenID Connect, the most important claim fields in the JWT are the **iss** and **sub** fields. Since the **iss** field is a URL, it is by definition unique for that issuer. The **sub** field is unique for the token subject within the set of subjects for that particular issuer. As a result, when the **iss** and **sub** fields are put together, the resulting value is globally unique across a virtually unlimited set of issuers and subjects.

Another important field within the JWT claims is the expiration time. By limiting the lifespan of the token to a relatively short period of time (e.g. 60 seconds), the time window for a bad actor to acquire the token, break the signature, and access protected resources with it is very short and much more difficult to achieve. All JWTs are signed by the issuer’s private key.

**Example:**

{

"iss": "https://server.example.com",

"sub": "24400320",

"aud": "s6BhdRkqt3",

"exp": 1311281970,

"nbf": 1311280980,

"iat": 1311280970,

"jti": “r584ksi3e38R7”

}

# Resource Model

## ID Token

The ID Token returned by the identity provider is a JSON web token. The following is an example of an ID token, based on the color-coded structure of the JSON Web Token described in Section 3.1.

**Example:**

eyJhbGciOiJSUzI1NiJ9.eyJhdXRoX3RpbWUiOjE0MTg2OTg3ODIsImV4cCI6MTQxODY5OTQxMiwic3ViIjoiNldaUVBwblF4ViIsIm5vbmNlIjoiMTg4NjM3YjNhZjE0YSIsImF1ZCI6WyJjMWJjODRlNC00N2VlLTRiNjQtYmI1Mi01Y2RhNmM4MWY3ODgiXSwiaXNzIjoiaHR0cHM6XC9cL2lkcC1wLm1pdHJlLm9yZ1wvIiwiaWF0IjoxNDE4Njk4ODEyfQ.mQc0rtL56dnJ7\_zO\_fx8-qObsQhXcn-qN-FC3JID BuNmP8i11LRA\_sgh\_omRRfQAUhZD5qTRPAKbLuCD451lf7ALAUwoGg8zAASI5QNGXoBVVn7buxPd2SElbSnHxu0o8ZsUZZwNpircWNUlYLje6APJf0kre9ztTj-5J1hRKFbbHodR2I1m5q8zQR0ql-FoFl OfPhvfurXxCRGqP1xpvLLBUi0JAw3F8hZt\_i1RUYWMqLQZV4VU3eVNeIPAD38qD1fxTXGVEd2XDJpmlcxjrWxzJ8fGfJrbsiHCzmCjflhv34O22zb0lJpC0d0VScqxXjNTa2-ULyCoehLcezmssg

### ID Token Claims

The ID Token payload contains several fields which indicate the claims associated with the token.

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Required? | Description |
| iss | URL | required | Issuer - URL of identity provider that issued the token. |
| sub | string | required | Token subject, a unique machine-readable string. |
| aud | string | required | Token audience, client ID of relying party. |
| exp | number | required | Expiration time of token in epoch time, should expire quickly. |
| iat | number | required | Issued at - Time token was issued in epoch time. |
| auth\_time | number | optional | Time that user authenticated with identity provider. |
| nonce | string | optional\* | Number or bit string that can only be used once. Sent by relying party during authentication, used to prevent replay attacks. \*Must be included if relying party includes it. |
| acr | string | optional | Authentication context reference – how the user authenticated with the identity provider. |
| amr | string | optional | Authentication method reference – how the user authenticated with the identity provider. |

**Example:**

{

"iss": "https://identityprovider.example.com",

"sub": "24400320",

"aud": "s6BhdRkqt3",

"nonce": "n-0S6\_WzA2Mj",

"exp": 1311281970,

"iat": 1311280970,

"auth\_time": 1311280969,

"acr": "urn:mace:incommon:iap:silver",

"amr": ["pwd"]

}

## Authorization Grant

The authorization grant is an opaque string that is returned by the authorization server, indicating what level of access has been granted by the resource owner. Once the client has received the authorization grant, it can use that grant to request an access token to retrieve the protected resource.

**Example:**

SplxlOBeZQQYbYS6WxSbIA

## Access Token

The access token returned by the authorization server is a JSON web token.

**Example:**

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o

### Access Token Claims

The Access Token payload contains several fields which indicate the claims associated with the token.

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Required? | Description |
| iss | URL | required | Issuer - URL of identity provider that issued the token. |
| sub | string | required | Token subject, a unique machine-readable string. |
| aud | string | required | Token audience, client ID of relying party. |
| exp | number | required | Expiration time of token in epoch time, should expire quickly. |
| iat | number | required | Issued at - Time token was issued in epoch time. |
| auth\_time | number | optional | Time that user authenticated with identity provider. |
| nonce | string | optional\* | Number or bit string that can only be used once. Sent by relying party during authentication, used to prevent replay attacks. \*Must be included if relying party includes it. |
| acr | string | optional | Authentication context reference – how the user authenticated with the identity provider. |
| amr | string | optional | Authentication method reference – how the user authenticated with the identity provider. |

## JSON Web Key (JWK)

The Access Token payload contains several fields which indicate type of the key, how it was generated, and information about the X.509 certificate associated with the key.

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Required? | Description |
| kty | string | required | Key type |
| n | string | required | Modulus for the standard X.509 certificate |
| e | string | required | Exponent for the standard X.509 certificate |
| alg | string | required | Algorithm for the key |
| kid | string | required | Unique identifier for the key |

**Example:**

{

"kty":"RSA",

"n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx

4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc\_BJECPebWKRXjBZCiFV

4n3oknjhMstn64tZ\_2W-5JsGY4Hc5n9yBXArwl93lqt7\_RN5w6Cf

0h4QyQ5v-65YGjQR0\_FDW2QvzqY368QQMicAtaSqzs8KJZgnYb

9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbISD08qNLyrdkt-

bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G\_xBniIqb

w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",

"e":"AQAB",

"alg":"RS256",

"kid":"2011-04-29"

}

# API Specification

The Poplin security mechanism is broken up into two services, one to implement the OpenID Connect identity provider (authentication) and another to implement the OAuth2-based authorization service (authorization).

All requests are made and encrypted using TLS v1.2 X.509 v3 certificates through the HTTPS protocol.

## Authentication Service (OpenID Connect)

### Discovery Endpoint

#### Get Endpoints Supported by Authentication (OpenID Connect) Service

**GET /.well-known/openid-configuration**

**Purpose:** This call retrieves information about the capabilities of the authentication service, including what token types and encryption algorithms are supported. It also provided URLs for each of the available endpoints.

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of identity provider |
| Accept | MIME Type | Type of data accepted in return |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| issuer | string | URL of identity provider |
| authorization\_endpoint | string | URL for the authorization endpoint |
| token\_endpoint | string | URL for the token endpoint |
| userinfo\_endpoint | string | URL for the user information endpoint |
| revocation\_endpoint | string | URL for the token revocation endpoint |
| jwks\_uri | string | URI for public JSON Web Key |
| response\_types\_supported | array[string] | List of types can be included in responses |
| subject\_types\_supported | array[string] | Subject identifier types supported by the identity provider. |
| id\_token\_signing\_alg\_values\_supported | array[string] | List of signing algorithms used to sign ID tokens |
| scopes\_supported | array[string] | List of scope values that the authentication service supports |
| token\_endpoint\_auth\_methods\_supported | array[string] | List of authorization methods supported on token endpoint |
| claims\_supported | array[string] | List of user identity claims supported by authentication service |

**Example call:**

GET /.well-known/openid-configuration HTTP/1.1

Host: identityprovider.example.com

Accept: text/plain, application/json, application/\*+json, \*/\*

**Example response:**

{

"issuer": "https://identityprovider.example.com",

"authorization\_endpoint": "https:// identityprovider.example.com/auth",

"token\_endpoint": "https:// identityprovider.example.com/token",

"userinfo\_endpoint": "https:// identityprovider.example.com/userinfo",

"revocation\_endpoint": "https:// identityprovider.example.com/revoke",

"jwks\_uri": "https:// identityprovider.example.com/jwks\_uri",

"response\_types\_supported": [

"code",

"token",

"id\_token",

"code token",

"code id\_token",

"token id\_token",

"code token id\_token",

"none"

],

"subject\_types\_supported": [

"public"

],

"id\_token\_signing\_alg\_values\_supported": [

"RS256"

],

"scopes\_supported": [

"openid",

"email",

"profile"

],

"token\_endpoint\_auth\_methods\_supported": [

"client\_secret\_post",

"client\_secret\_basic"

],

"claims\_supported": [

"aud",

"email",

"email\_verified",

"exp",

"family\_name",

"given\_name",

"iat",

"iss",

"locale",

"name",

"picture",

"sub"

]

}

### Token Endpoint

#### Get ID Token

**POST /token**

**Purpose:** This call generates an ID token if the user credentials are valid.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Authorization | JSON Web Token | Bearer token |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| grant\_type | string | “id\_token” to get an ID token |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| access\_token | string | Access token to get user information |
| token\_type | string | Type of ID token |
| id\_token | JSON Web Token | ID token |

**Example call:**

GET /token HTTP/1.1

Host: identityprovider.example.com

Accept: text/plain, application/json, application/\*+json, \*/\*

Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJleHAiOjE0MTg3MDI0MTIsImF1ZCI6WyJjMWJjODRlNC00N2VlLTRiNjQtYmI1Mi01Y2RhNmM4MWY3ODgiXSwiaXNzIjoiaHR0cHM6XC9cL2lkcC1wLmV4YW1wbGUuY29tXC8iLCJqdGkiOiJkM2Y3YjQ4Zi1iYzgxLTQwZWMtYTE0MC05NzRhZjc0YzRkZTMiLCJpYXQiOjE0MTg2OTg4MTJ9i.HMz\_tzZ90\_b0QZSAXtQtvclZ7M4uDAs1WxCFxpgBfBanolW37X8h1ECrUJexbXMD6rrj\_uuWEqPD738oWRo0rOnoKJAgbF1GhXPAYnN5pZRygWSD1a6RcmN85SxUig0H0e7drmdmRkPQgbl2wMhu-6h2Oqw-ize4dKmykN9UX\_2drXrooSxpRZqFVYX8PkCvCCBuFy2O-HPRov\_SwtJMk5qjUWM yn2I4Nu2s-R20aCA-7T5dunr0iWCkLQnVnaXMfA22RlRiU87nl21zappYb1\_EHF9ePyq3Q353cDUY 7vje8m2kKXYTgc\_bUAYuWW3SMSw5UlKaHtSZ6PQICoA

grant\_type=id\_token

**Example response:**

HTTP 200 OK

Content-Type: application/json

Cache-Control: no-store

Pragma: no-cache

{

“access\_token”: “eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3O

DkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPf

bIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o”,

“token\_type”: “Bearer”,

“id\_token”: “eyJhbGciOiJSUzI1NiJ9.eyJhdXRoX3RpbWUiOjE0MTg2OTg3ODIsImV4cCI6M

TQxODY5OTQxMiwic3ViIjoiNldaUVBwblF4ViIsIm5vbmNlIjoiMTg4NjM3YjN

hZjE0YSIsImF1ZCI6WyJjMWJjODRlNC00N2VlLTRiNjQtYmI1Mi01Y2RhNmM4M

WY3ODgiXSwiaXNzIjoiaHR0cHM6XC9cL2lkcC1wLmV4YW1wbGUuY29tXC8iL.C

JpYXQiOjE0MTg2OTg4MTJ9mQc0rtL56dnJ7\_zO\_fx8-qObsQhXcn-qN-FC3JID

BuNmP8i11LRA\_sgh\_omRRfQAUhZD5qTRPAKbLuCD451lf7ALAUwoGg8zAASI5Q

NGXoBVVn7buxPd2SElbSnHxu0o8ZsUZZwNpircWNUlYLje6APJf0kre9ztTj-5

J1hRKFbbHodR2I1m5q8zQR0ql-FoFlOfPhvfurXxCRGqP1xpvLLBUi0JAw3F8h

Zt\_i1RUYWMqLQZV4VU3eVNeIPAD38qD1fxTXGVEd2XDJpmlcxjrWxzJ8fGfJrb

siHCzmCjflhv34O22 zb0lJpC0d0VScqxXjNTa2-ULyCoehLcezmssg”

}

### UserInfo Endpoint

The UserInfo endpoint can be used by clients to get information about an authenticated user.

#### Get Information about the User

**GET /userinfo**

**Purpose:** This call allows a client with an access token to retrieve information about the user associated with that token from the identity provider.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Authorization | JSON Web Token | Token type, access token |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| sub | string | Token subject, a unique machine-readable string. |
| name | string | Name of the user |
| preferred\_username | string | Username for user |
| given\_name | string | First name |
| family\_name | string | Last name |
| nickname | string | Nickname that user prefers |
| gender | string | Gender of the user |
| updated\_time | number | Last time that the user info has been updated |
| birthdate | string | Birthdate of the user (yyyy-mm-dd format) |
| email | string | Email address for the user |
| email\_verified | boolean | TRUE if the email address has been verified |
| phone\_number | string | Phone number for the user |
| phone\_number\_verified | boolean | TRUE if the phone number has been verified |

**Example call:**

GET /userinfo HTTP/1.1

Host: identityprovider.example.com

Accept: text/plain, application/json, application/\*+json, \*/\*

Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJleHAiOjE0MTg3MDI0MTIsImF1ZCI6WyJjMWJjODRlNC00N2VlLTRiNjQtYmI1Mi01Y2RhNmM4MWY3ODgiXSwiaXNzIjoiaHR0cHM6XC9cL2lkcC1wLmV4YW1wbGUuY29tXC8iLCJqdGkiOiJkM2Y3YjQ4Zi1iYzgxLTQwZWMtYTE0MC05NzRhZjc0YzRkZTMiLCJpYXQiOjE0MTg2OTg4MTJ9i.HMz\_tzZ90\_b0QZSAXtQtvclZ7M4uDAs1WxCFxpgBfBanolW37X8h1ECrUJexbXMD6rrj\_uuWEqPD738oWRo0rOnoKJAgbF1GhXPAYnN5pZRygWSD1a6RcmN85SxUig0H0e7drmdmRkPQgbl2wMhu-6h2Oqw-ize4dKmykN9UX\_2drXrooSxpRZqFVYX8PkCvCCBuFy2O-HPRov\_SwtJMk5qjUWM yn2I4Nu2s-R20aCA-7T5dunr0iWCkLQnVnaXMfA22RlRiU87nl21zappYb1\_EHF9ePyq3Q353cDUY 7vje8m2kKXYTgc\_bUAYuWW3SMSw5UlKaHtSZ6PQICoA

**Example response:**

HTTP 200 OK

Date: Tue, 16 Dec 2014 03:00:12 GMT

Access-Control-Allow-Origin: \*

Content-Type: application/json;charset=ISO-8859-1

Content-Language: en-US

Content-Length: 333

Connection: close

{

"sub": "6WZQPpnQxV",

"name": "Steve Emeritus",

"preferred\_username": "steve",

"given\_name": "Stephen",

"family\_name": "Emeritus",

"nickname": "Steve",

"gender": "M",

"updated\_time": "2014-09-24 14:27:43.701000",

"birthdate": "1980-01-01",

"email": "steve.e@example.com",

"email\_verified": true,

"phone\_number": "857-555-1234",

"phone\_number\_verified": true

}

**Common errors:**

401: Bearer token for authorization is not valid

## Authorization Service (OAuth 2.0)

### Discovery Endpoint

The authorization service supports a “discovery” call to retrieve the endpoints for the service.

#### Get Endpoints Supported by Authorization (OAuth2) Service

**GET /**

**Purpose:** This call retrieves the URLs for each of the available endpoints.

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| authorizationEndpoint | string | URL for the authorization endpoint |
| tokenEndpoint | string | URL for the token endpoint |
| introspectionEndpoint | string | URL for the token introspection endpoint |
| revocationEndpoint | string | URL for the token revocation endpoint |
| registrationEndpoint | string | URL for the dynamic registration endpoint |

**Example call:**

GET / HTTP/1.1

Host: authserver.example.com

Accept: application/json

**Example response:**

HTTP 200 OK

{

“authorizationEndpoint”: “https://authserver.example.com/authorize”,

“tokenEndpoint”: “https://authserver.example.com/token”,

“introspectionEndpoint”: “https://authserver.example.com/introspect”,

“revocationEndpoint”: “https://authserver.example.com/revoke”,

“registrationEndpoint”: “https://authserver.example.com/register”

}

**Common errors:**

None

### Registration Endpoint

The registration endpoint of the authorization service is called when a client wishes to dynamically register with the authorization service.

**POST /register**

**Purpose:** To register a client with the authorization service on demand.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Content-type | MIME Type | Type of content included with call |
| Authorization | JSON Web Token | Bearer token |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| client\_name | string | Human-readable string name of client to be presented to end user during authorization. |
| redirect\_uris | array[string] | Callback URI at client to use during redirection flows (e.g. authorization grant code requests). |
| client\_uri | string | URL string of web page providing information about client. |
| grant\_types | array[string] | OAuth 2.0 grant type strings that client desires at token endpoint. |
| scope | string | Space-separated list of scope values that client desires when requesting access tokens. |
| jwks\_uri | string | URI to access client’s public JSON web key set. These keys can be used by higher-level protocols that use signing or encryption. |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| client\_id | string | Client identifier that is unique at the identity provider. |
| client\_id\_issued\_at | number | Time client identifier was issued in epoch time (seconds). |
| token\_endpoint\_auth\_method | string | Requested authentication method for token endpoint. |
| client\_name | string | Human-readable string name of client to be presented to end user during authorization. |
| redirect\_uris | array[string] | Callback URI at client to use during redirection flows (e.g. authorization grant code requests). |
| client\_uri | string | URL string of web page providing information about client. |
| grant\_types | array[string] | OAuth 2.0 grant type strings that client can use at token endpoint. |
| scope | string | Space-separated list of scope values that client can use when requesting access tokens. |
| jwks\_uri | string | URI to access client’s public JSON web key set. These keys can be used by higher-level protocols that use signing or encryption. |

**Example call:**

POST /register HTTP/1.1

Host: authserver.example.com

Content-Type: application/json

Accept: application/json

Authorization: Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

{

“client\_name”: “OAuth Client”,

“redirect\_uris”: [“https://client.example.com/callback”],

“client\_uri”: “https://client.example.com”,

“grant\_types”: [“authorization\_code”],

“scope”: “conditions allergies medications”

}

**Example response:**

HTTP 201 Created

Content-Type: application/json

{

“client\_id”: “474djee3i83”,

“client\_id\_issued\_at”: 1519946470,

“token\_endpoint\_auth\_method”: “client\_secret\_post”,

“client\_name”: “OAuth Client”,

“redirect\_uris”: [“https://client.example.com/callback”],

“client\_uri”: “https://client.example.com”,

“grant\_types”: [“authorization\_code”],

“response\_types”: [“code”],

“scope”: “conditions allergies medications”

}

**Common errors:**

*400: Grant types didn’t match response types*

* Client is requesting grants that not reflected in the response types

*400: Redirect URI is invalid*

* Client either didn’t specify a redirect URI or it wasn’t in the right format

*401: Bearer token for authorization is not valid*

* Client must get a valid bearer token

### Authorization Endpoint

The authorization endpoint of the authorization service is called when a client wants to obtain an authorization grant code from the authorization service. The authorization grant code is used for subsequent calls to the authorization service to get access tokens and refresh tokens, which are ultimately used to access protected resources. When servicing an authorization grant code request, the authorization service authenticates the resource owner and permission claims before generating the authorization grant code.

#### Get Authorization Code

**GET /authorize**

**Purpose:** To authenticate the resource owner, collect permissions allowed by the resource owner, and generate an authorization grant code that can be used to retrieve tokens.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Content-type | MIME Type | Type of content included with call |
| Authorization | JSON Web Token | Bearer token |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| response\_type | string | Must be “code”. |
| client\_id | string | Client identifier provided at registration with authentication service. |
| redirect\_uri | string | Callback URI at client to use when generation of authorization grant code is completed. |
| scope | string | Space-separated list of scope values that client desires when requesting access tokens. |
| state | string | Unique string value that will be passed to the client redirect URI. |

**Example authorization grant code request:**

GET /authorize HTTP/1.1

Host: authserver.example.com

Accept: application/json

Content-type: application/x-www-form-encoded

Authorization: Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

{

“response\_type”: “code”,

“client\_id”: “474djee3i83”,

“redirect\_uri”: “https://client.example.com/callback”,

“scope”: “conditions allergies medications”,

“state”: “12345678”

}

**Common errors:**

*400: Unknown client*

* Client needs to register with the authorization service

*400: Redirect URI doesn’t match*

* Redirect URI in the request doesn’t match the redirect URI provided during registration

**Redirect callback:**

The authorization service responds to the client in the form of a callback to the redirect URI provided in the authorization grant code request. The redirect URI must refer to code that is part of the client and match the URI value the client used when registering with the authorization service.

**GET {redirect\_uri}**

**Purpose:** The purpose of the call to the client’s redirect URI by the authorization service is to pass the authorization code only to the URI registered by the client during the registration process. A state variable is also included which was set to a unique value by the client during the request. The value of the state variable it the callback must match the callback value provided in the request.

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Content-type | MIME Type | Type of content included with call |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| code | string | Generated authorization grant code. |
| state | string | String value that must match state value client used in the authorization grant code request. |

**Example redirect call:**

GET https://client.example.com/callback

Content-Type: application/json

{

“code”: “SplxlOBeZQQYbYS6WxSbIA”,

“state”: “12345678”

}

GET https://client.example.com/callback

Content-Type: application/json

{

“error”: “Unsupported response type”

}

### Token Endpoint

The token endpoint of the authorization service is called when an access token is needed to retrieve a resource. The endpoint can also be used to refresh an existing access token.

#### Get Access Token

**POST /token**

**Purpose:** If the authorization grant code is valid, an access code with the proper scope matching the permissions provided in the authorization grants is generated.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Content-type | MIME Type | Type of content included with call |
| Authorization | JSON Web Token | Bearer token |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| grant\_type | string | “authorization\_code” |
| code | string | Authorization grant code |
| redirect\_uri | string | Callback URI at client to use when generation of authorization grant code is completed. |
| client\_id | string | Client identifier provided at registration with authentication service. |
| client\_assertion\_type | string | “urn:ietf:params:oauth:client-assertion-type:  jwt-bearer” |
| client\_assertion | JSON Web Token | Claims client is requesting for access token |

**JSON Web Token for Client Assertion Claims:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| iss | string | “authorization\_code” |
| sub | string | Authorization grant code |
| aud | string | Authorization service URL |
| iat | number | Time token was issued by client in epoch time (seconds) |
| exp | number | Expiration time of token in epoch time (seconds) |
| jti | string | Unique string for grant request |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| access\_token | JSON Web Token | Generated access token |
| token\_type | string | Type of token generated |
| expires in | number | Number of seconds until expiration |
| refresh\_token | JSON Web Token | Generated refresh token |
| scope | string | Scope that the resource owner granted |
| state | string | Code to prevent cross-site request forgery (CSRF) attacks |

**Example call:**

POST /token HTTP/1.1

Host: authserver.example.com

Accept: application/json

Content-type: application/x-www-form-encoded

Authorization: Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

{

“grant\_type”: “authorization\_code”,

“code”: “SplxlOBeZQQYbYS6WxSbIA”,

“redirect\_uri”: “https://client.example.com/callback”,

“client\_id”: 474djee3i83,

“client\_assertion\_type”: “urn:ietf:params:oauth:client-assertion-type:

jwt-bearer”

“client\_assertion”: “eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiI0NzRka

mVlM2k4MyIsInN1YiI6IjQ3NGRqZWUzaTgzIiwiYXVkIjoiaHR0cHM6Ly9hdXRoc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwiaWF0IjoxNTIxNjYzMDc1LCJleHAiOjE1MjE2NjMxMzUsImp0aSI6IjQ3NHk2ZGU4ZTdmZSJ9.4YA1aWU9x1LoA79GESwMV88lKoIcvvIt1CK5TNupeiQ”

}

**Example response:**

HTTP 200 OK

Content-Type: application/json

Cache-Control: no-store

Pragma: no-cache

{

“access token”: “eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3O

DkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPf

bIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o”,

“token\_type”: “bearer”,

“expires\_in”: 60,

“refresh\_token”: “eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhm

ODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHV

TCMA9vdoHrcZxH-x5mb11y1537t3rGzcM”,

“scope”: “create”,

“state”: “12345678”

}

**Common errors:**

*400: Invalid grant*

* Authorization code was not found. Client needs to get a new authorization code.

*400: Unsupported grant type*

* Grant type must be “authorization\_code”.

*401: Bearer token for authorization is not valid*

* Client must get a valid bearer token

*401: Invalid client*

* Client needs to register with the authorization service

#### Refresh Access Token

**POST /token**

**Purpose:** If the current access token has expired and the client has a refresh token provided by the authorization service, the client can use this call with the refresh token to get a new access token without having to ask for the resource owner’s credentials.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Content-type | MIME Type | Type of content included with call |
| Authorization | JSON Web Token | Bearer token |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| grant\_type | string | “refresh\_token” to refresh the access token |
| refresh\_token | JSON Web Token | Refresh token previously provided by authorization service |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| access\_token | JSON Web Token | New access token |
| refresh\_token | JSON Web Token | Refresh token |

**Example call:**

POST /token HTTP/1.1

Host: authserver.example.com

Accept: application/json

Content-type: application/x-www-form-encoded

Authorization: Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

grant\_type=refresh\_token

refresh\_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537 t3rGzcM

**Example response:**

HTTP 200 OK

{

“access\_token”: “eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3O

DkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjQ3OTY4fQ.0hgo

dQJ546rfWpeWFO5xRfvpFgHdJza9a06fW\_NAJ3k”

“refresh\_token”: “eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhm

ODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHV

TCMA9vdoHrcZxH-x5mb11y1537t3rGzcM”

}"

**Common errors:**

*400: Invalid grant* – matching refresh token not found or token has the wrong client ID

* Client can’t refresh the current access token and must get a new access token from scratch

*401: Bearer token for authorization is not valid*

* Client must get a valid bearer token

### Introspection Endpoint

The introspection endpoint of the authorization service is called by the resource server to get more information about the token, including whether it is currently active, its scope, and its expiration time. The resource server can use this information to determine whether to provide a resource to the requesting client.

#### Perform Introspection on Access Token

**POST /introspect**

**Purpose:** This call extracts information from the issued access token and returns it to the resource server.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Content-type | MIME Type | Type of content included with call |
| Authorization | JSON Web Token | Bearer token |

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| Token | JSON Web Token | Access token |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| active | boolean | True if token is active, otherwise false. Resource server must accept only active tokens. |
| scope | string | Space delimited string of items that can be accessed through this token. |
| client\_id | string | Client ID of the client requesting authorization. |
| username | string | User name requesting authorization. |
| iss | string | URL of the authorization server that issued the token. |
| sub | string | Token subject, a unique machine-readable string. |
| iat | number | Time in UNIX Epoch seconds that token was issued. |
| exp | number | Expiration time of token in UNIX Epoch time. |

**Example call:**

POST /introspect HTTP/1.1

Host: authserver.example.com

Accept: application/json

Content-type: application/x-www-form-encoded

Authorization: Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM

**Example response:**

HTTP 200 OK

{

“active”: true,

“scope”: “conditions medications allergies”,

“client\_id” : “oauth-client-1”,

“username”: “fred”,

“iss”: “authserver.example.com”,

“sub”: “fred”,

“aud”: “rsrcserver.example.com”,

“iat”: 1519946470, (seconds since midnight 1 January 1970 UTC)

“exp”: 1519946530 (Note: token expires 60 seconds after issuance)

}

**Common errors:**

### Revocation Endpoint

#### Revoke a Token

**POST /revoke**

**Purpose:** This call allows the client to indicate that it no longer needs the access token and that the authorization server should remove the token from further use.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |
| Content-type | MIME Type | Type of content included with call |
| Authorization | JSON Web Token | Bearer token |

**Example call:**

POST /revoke HTTP/1.1

Host: authserver.example.com

Accept: application/json

Content-type: application/x-www-form-encoded

Authorization: Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM

**Example response:**

HTTP 201 No Content

**Common errors:**

The authorization server always responds with a successful response so that information about other tokens is not exposed.

### Callbacks

Clients to the security service need to provide the following API calls to allow the authorization server to retrieve public keys for clients.

#### JSON Web Key

**GET /jwk**

**Purpose:** Services or applications that call the authorization server must provide a jwk endpoint so that the authorization server can retrieve the public JSON web key for that service or application. The endpoint only needs to provide a single GET call to retrieve the public key.

**Format:** JSON

**Headers:**

|  |  |  |
| --- | --- | --- |
| Header | Type | Description |
| Host | URL | URL of authorization server |
| Accept | MIME Type | Type of data accepted in return |

**Response:**

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| keys | array[JWK] | List of public JSON Web Keys |

**Example call:**

GET /jwk HTTP/1.1

Host: client.example.com

Accept: application/json

**Example response:**

HTTP 200 OK

{

"keys":

[

{

"kty":"RSA",

"n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx4cbbfAAtVT86

zwu1RK7aPFFxuhDR1L6tSoc\_BJECPebWKRXjBZCiFV4n3oknjhMstn64tZ\_2W-5J

sGY4Hc5n9yBXArwl93lqt7\_RN5w6Cf0h4QyQ5v-65YGjQR0\_FDW2QvzqY368QQMi

cAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbISD08qNLyrdkt

-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G\_xBniIqbw0Ls1jF44-cs

FCur-kEgU8awapJzKnqDKgw",

"e":"AQAB",

"alg":"RS256",

"kid":"2011-04-29"  
 }

]

}

# Message Specification

The security services can generate messages to notify other services within the Medicaid enterprise system of important security events.

## Unauthorized Access Message

### Queue Type

This message uses the publish and subscribe message queue. Authorized services that want to be notified of unauthorized access events can subscribe to these messages using the APIs described in the messaging service definition.

### Message Payload

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| auth\_server | string | List of public JSON Web Keys |
| request | JSON object | Type of request and parameters |
| authorization\_grant | string | Authorization grant provided to client |
| client\_id | string | Client identifier provided by authorization server |
| client\_public\_key | JSON Web Token | Public key provided by client |
| client\_ip\_addr | string | IPv4 or IPv6 address of client |
| access\_token | JSON Web Token | Access token provided by client |
| access\_token\_claims | JSON object | Access token claims from authorization server |

# Use Case Example

In this use case example, Steve is a Medicaid member who regularly visits his Medicaid provider. While away on vacation, Steve gets into a car accident and needs to go to the local emergency room for treatment. In this example, there are three security domains: one for Steve, one for the Medicaid provider, and a third for the local emergency room, as shown in Figure A‑1. In order for Steve to get the best care, information needs to flow between these three security domains.

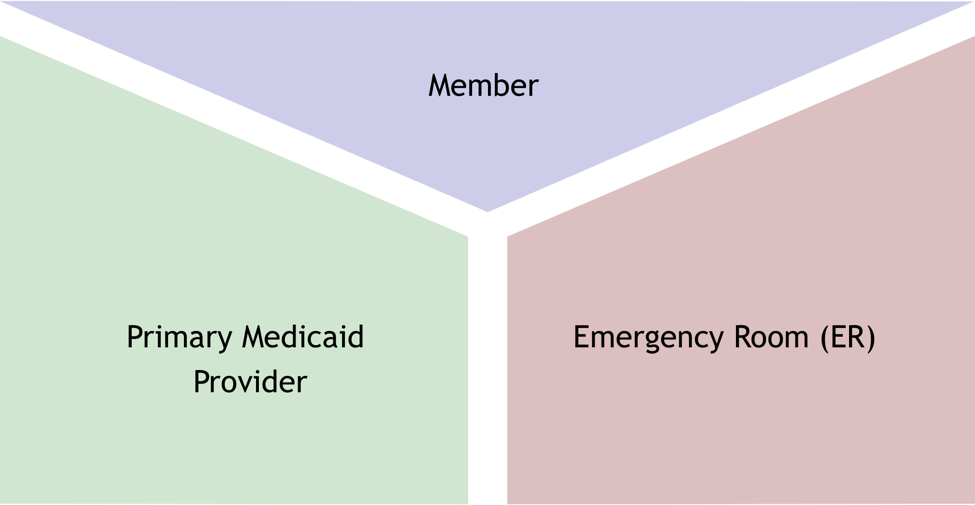


Figure A‑1: Use Case - Three Separate Security Domains

Steve sees his Medicaid provider regularly for checkups and treatment and so he routinely accesses his medical records, either through the web portal of the provider or through his mobile health tracking application. When Steve goes to the emergency room after his car accident, the emergency room physicians need access to Steve’s medical records to provide the best care. The results of the emergency visit should also be shared with the Medicaid provider for any follow-up care that might be necessary. The high-level information flow for this use case is shown in Figure A‑2.

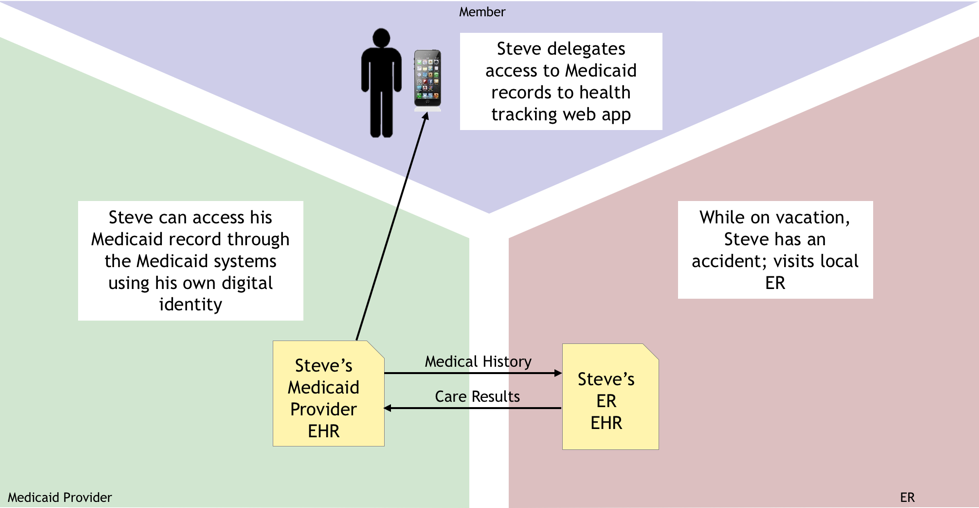


Figure A‑2: Use Case Overview

When Steve uses his health tracking application to access his Medicaid provider records, he delegates access to the health tracking application by providing his credentials to his identity provider at example.com. As shown in Figure A‑3, Steve’s identity information is provided to the Medicaid provider to verify that Steve, through his health tracking application, can access his records.

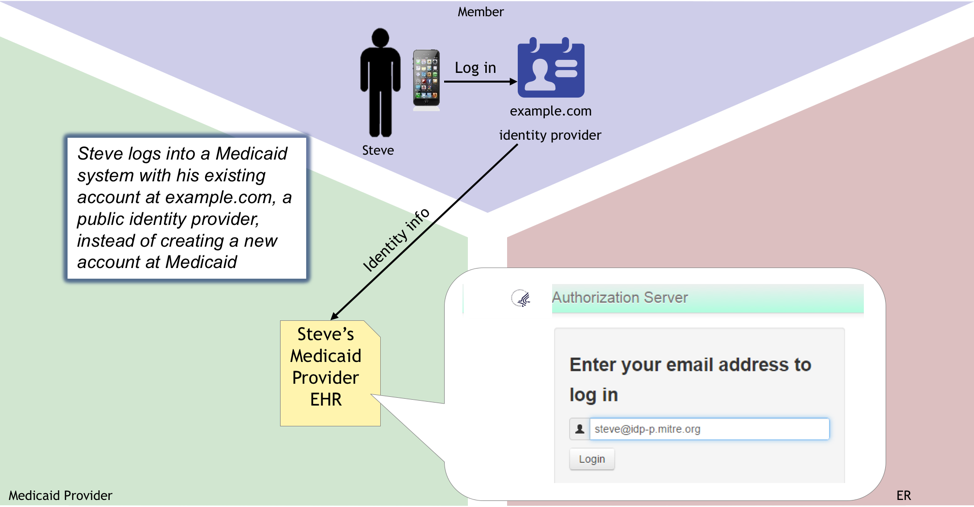


Figure A‑3: Storyboard – Steve views his Medicaid provider records

The authorization server at the Medicaid provider redirects the health tracking application to a page where Steve can confirm permission to the health tracking application to access certain elements of his medical record. This process is shown in Figure A‑4.

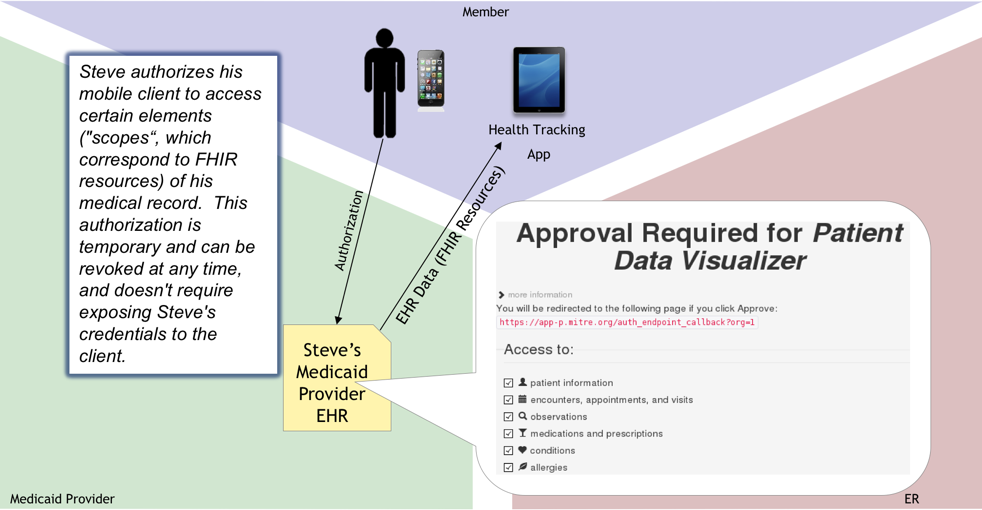


Figure A‑4: Storyboard – Steve connects his mobile app to his data

After Steve’s car accident, he is transported to the local emergency room. To provide the best care possible, the emergency room physicians would like Steve’s permission to access his medical records at his Medicaid provider. Steve grants this permission and the emergency room physicians are able to gain access to his medication and prescription history, as shown in Figure A‑5.

Note that this example assumes that Steve is conscious and coherent. Obviously, a real-time grant from Steve would not be possible if Steve were unconscious. To solve this problem, a system of patient consent and privacy management is needed to allow Steve to make consent decisions ahead of time and also allow physicians to gain exceptional access under HIPAA requirements. User Managed Access (UMA) adds this functionality by leveraging OAuth 2.0 and OpenID Connect, however patient consent and privacy is beyond the scope of this security definition.

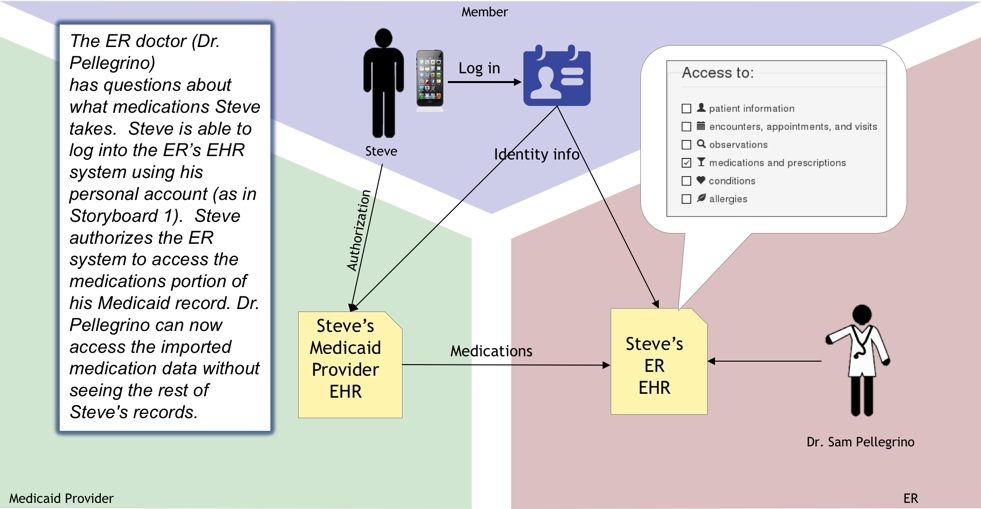


Figure A‑5: Storyboard – Steve links his new ER records to his Medicaid records

Once Steve returns from his vacation, he has a follow-up exam with his Medicaid provider, who can access the emergency room record about Steve’s injuries and update Steve’s Medicaid records. This process is shown in Figure A‑6.

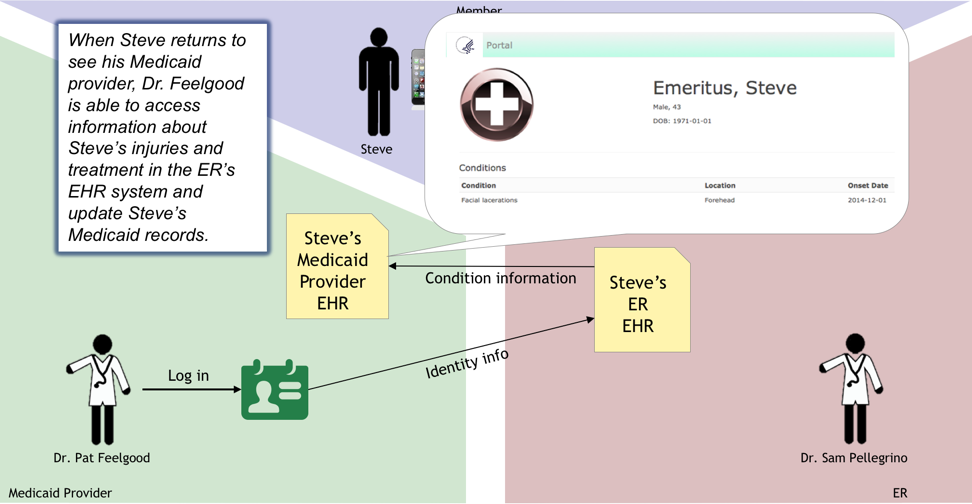


Figure A‑6: Storyboard – Medicaid provider accesses ER records

After reviewing the high-level data flow that is occurring between the three security domains in this use case, a review of the components and control flows are needed to see how OAuth 2.0 and OpenID Connect provide the necessary security to only allow authorized actors to gain access to protected data. In this use case example, there are ten components across the three security domains, as shown in Figure A‑7.

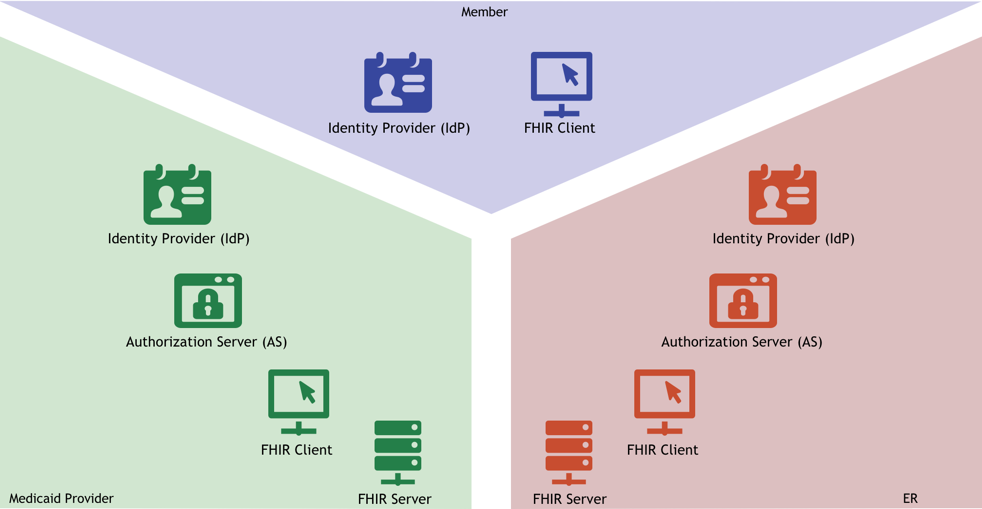


Figure A‑7: Use Case Security Components

As shown in Figure A‑8, there are two components within the “member” domain: an identity provider (idp-p) and health-tracking application (app-p). Both components are trusted by the member. The member can either run the services themselves, purchase them from a provider, or these services can be provided to the member. The health tracking application is a mobile application that acts as a FHIR client and can retrieve and analyze data from multiple providers. The identity provider provides the identity for the member when visiting other systems and security domains. The other security domains also need to trust the member’s identity provider.

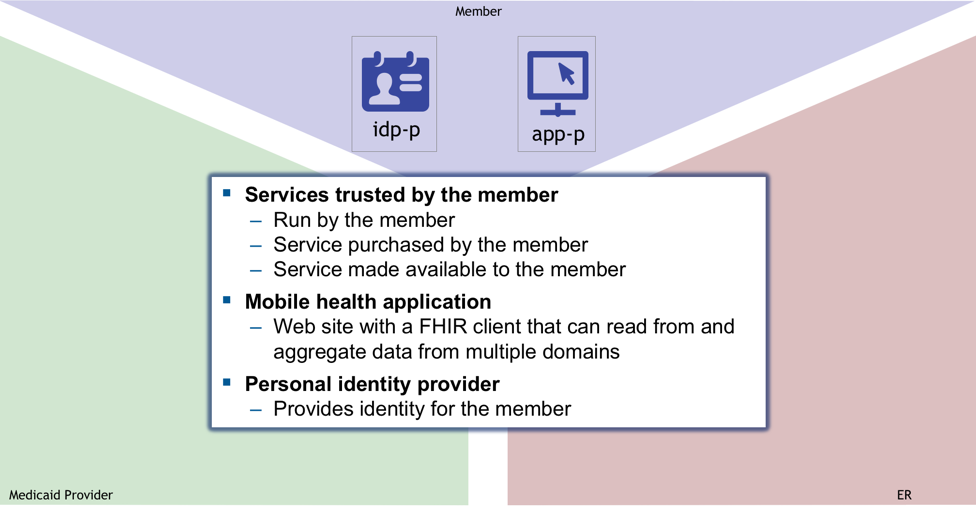


Figure A‑8: Member Domain Components

The Medicaid provider and emergency room security domains each have four components, as shown in Figure A‑9. Like the member domain, each of these security domains also has an identity provider to manage identities within those organizations. Each of these healthcare domains also has an authorization server to protect the APIs and resources within its own respective security domain. Lastly, each of the healthcare organizations has a FHIR server, which acts as the resource server containing member records as well as a FHIR-based web portal or client that allows members to sign in and view their records.

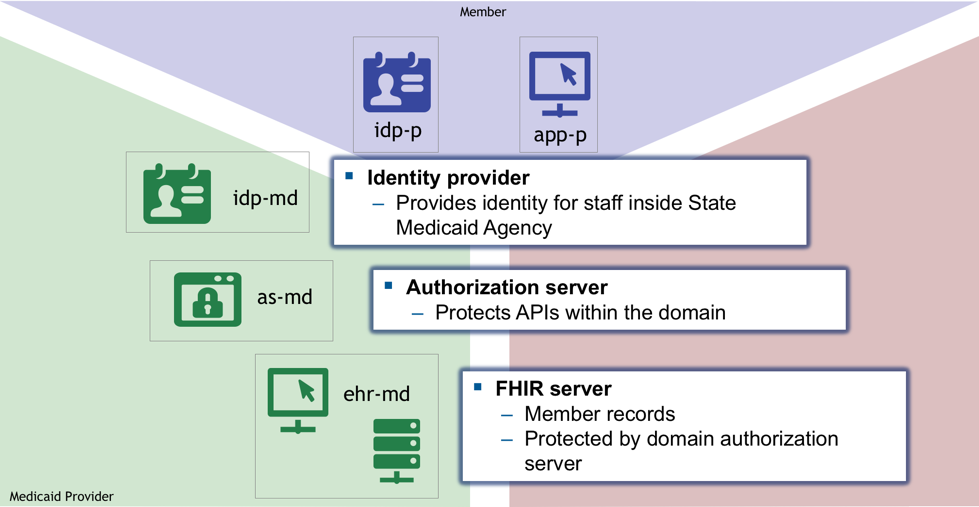


Figure A‑9: Healthcare Provider Components

Within the use case, there are three FHIR-based clients, one for each security domain that provide a user-facing display to the users within each of those security domains, as shown in Figure A‑10. The FHIR clients can connect to multiple FHIR records across security domains and authenticate users through OpenID Connect.

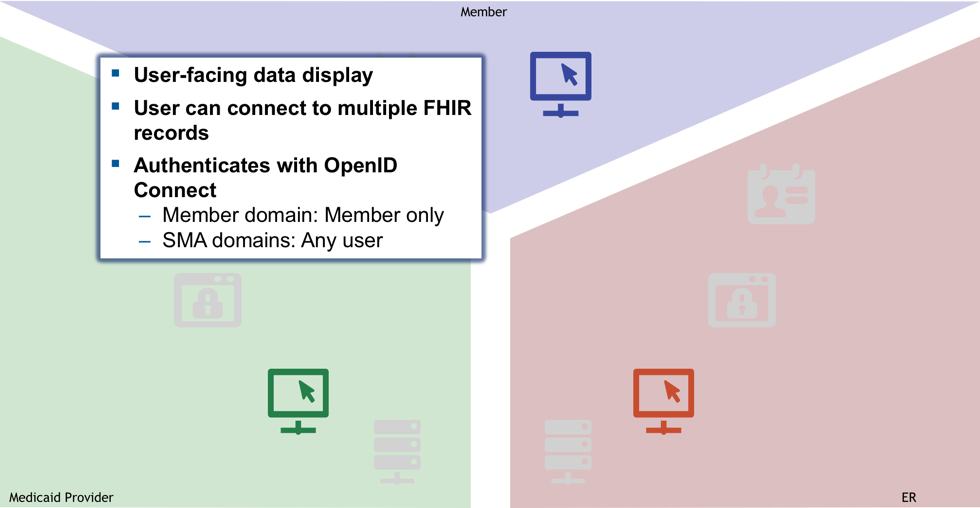


Figure A‑10: FHIR Clients

The Medicaid provider and emergency room each have FHIR servers to maintain the medical records for their respective organizations. As shown in Figure A‑11, the FHIR servers determine, based on the access tokens sent with the request, which users have access to what records. The FHIR servers only accept tokens from the authorization server within their own security domain and serve only records local their domain.

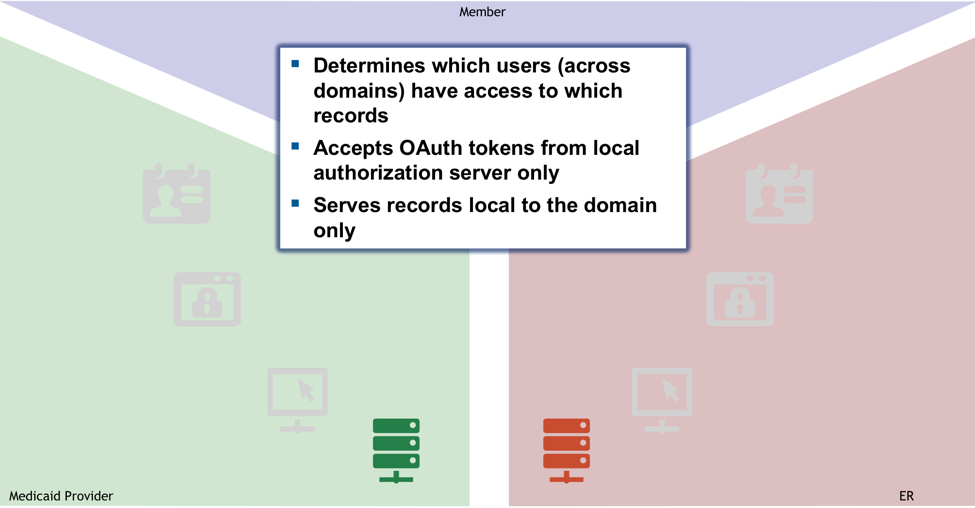


Figure A‑11: FHIR Servers

Figure A‑12 highlights that security domain has an identity server to manage the identities and direct logins for local accounts within their respective domains. The identity servers do not provide API authorization – that is done by the authorization servers.

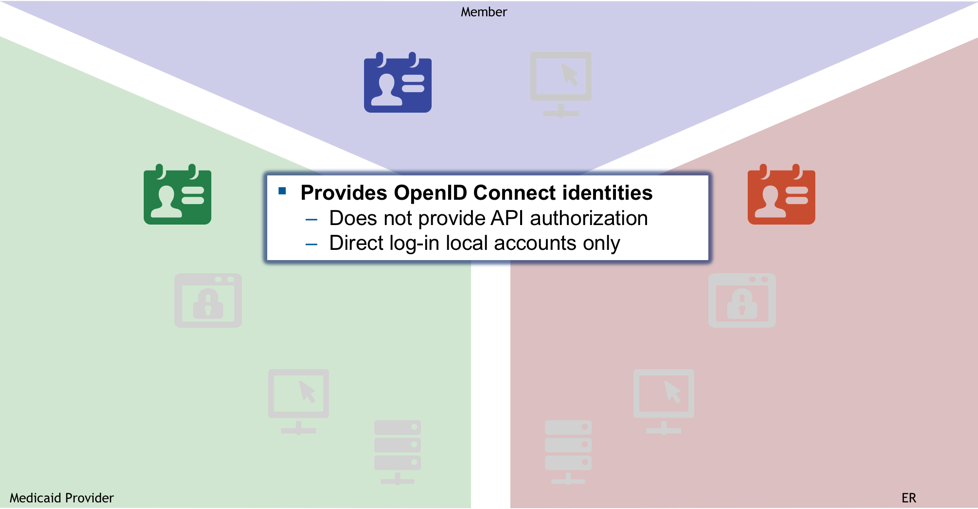


Figure A‑12: Identity Providers

The Medicaid provider and emergency room have sensitive information that they need to protect, so each domain has an authorization server, as shown in Figure A‑13. The authorization server provides OAuth 2.0 authorization capabilities, allowing logins both from within its security domain from other domains. Both clients and protected resources use the authorization server to determine whether the requested information can be retrieved.

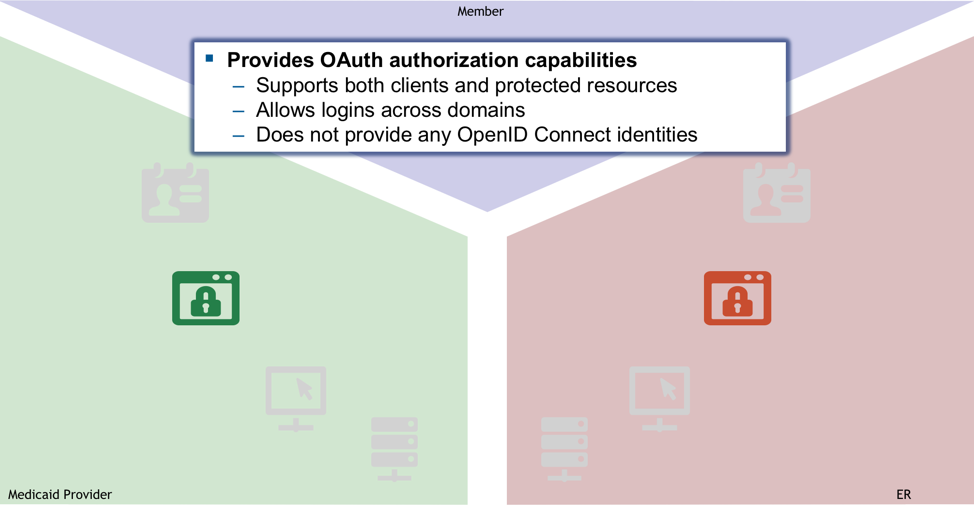


Figure A‑13: Authorization Servers

Now that the components of the use case have been covered, both by function and by security domain, the control flows between those components must be described. The control flows for each of the steps in the use case will be covered. In all cases, it is important to note that the FHIR server and the authorization server always reside together in the same security domain. In each scenario, the control flow is the same, but different components are involved. In other words, it is the same dance, just with different dance partners.

Figure A‑14 shows the control flows for when Steve logs into the web portal of his Medicaid provider to view his medical records.

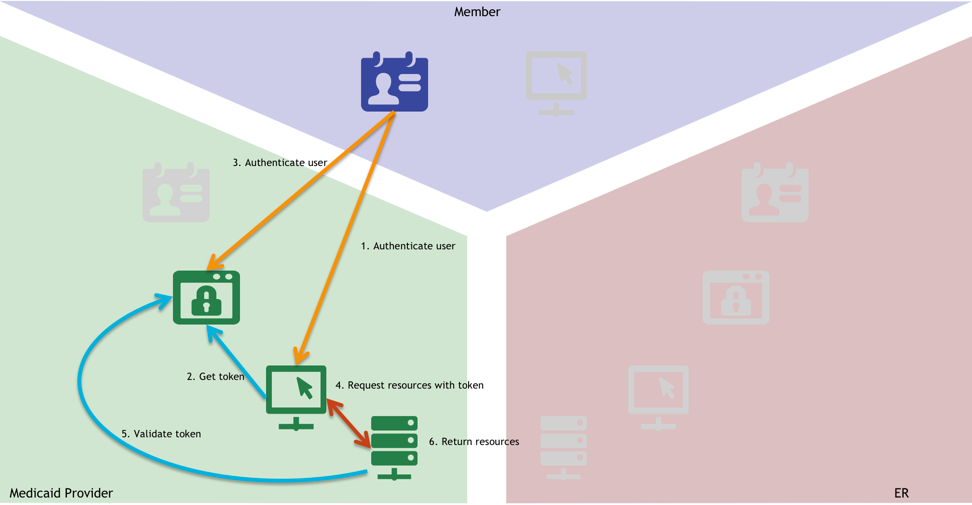


Figure A‑14: Steve logs in to view his record

The control flow for when Steve delegates access of his Medicaid information to his mobile health tracking application is shown in Figure A‑15. This is the same as the last scenario except that Steve is using the mobile application instead of the web portal for his FHIR client. The interactions with the FHIR client in both examples are the same, as are the transactions between the other components.

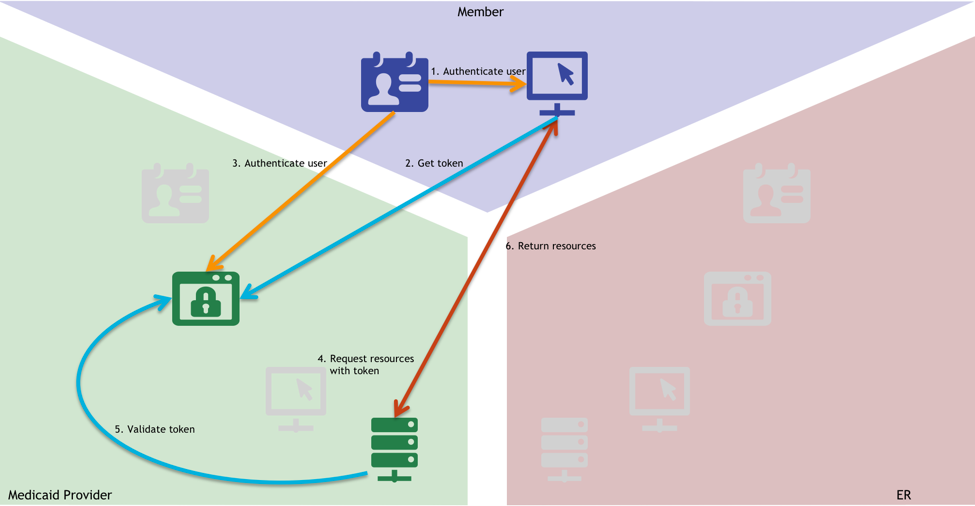


Figure A‑15: Steve delegates access to his client, such as a mobile app

The control flow in Figure A‑16, where Steve logs into the emergency room web portal to view his medical record, is a mirror copy of the control flow used in Figure 16, with emergency room components replacing the same Medicaid provider components.

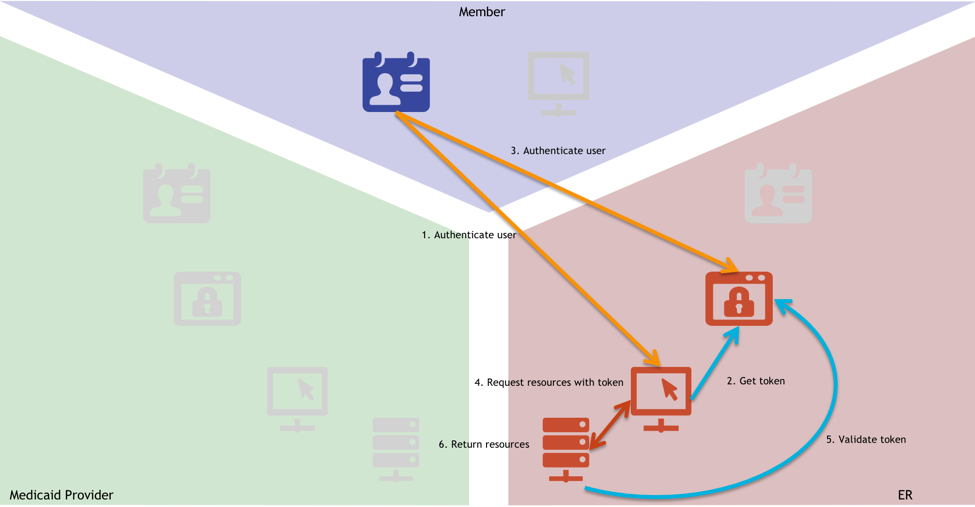


Figure A‑16: Steve logs in to view his ER record

So far, the scenarios have only involved two security domains, the member domain and a single healthcare provider domain. The scenario in Figure A‑17 involves all three security domains, but is actually similar to the scenario in Figure A‑15, where Steve delegated authority to access his records to his mobile health tracking application. In Figure A‑17, Steve is instead delegating authority to the FHIR client in the emergency room to access his Medicaid provider records on his behalf. Otherwise, the control flow between the components in both scenarios is the same.

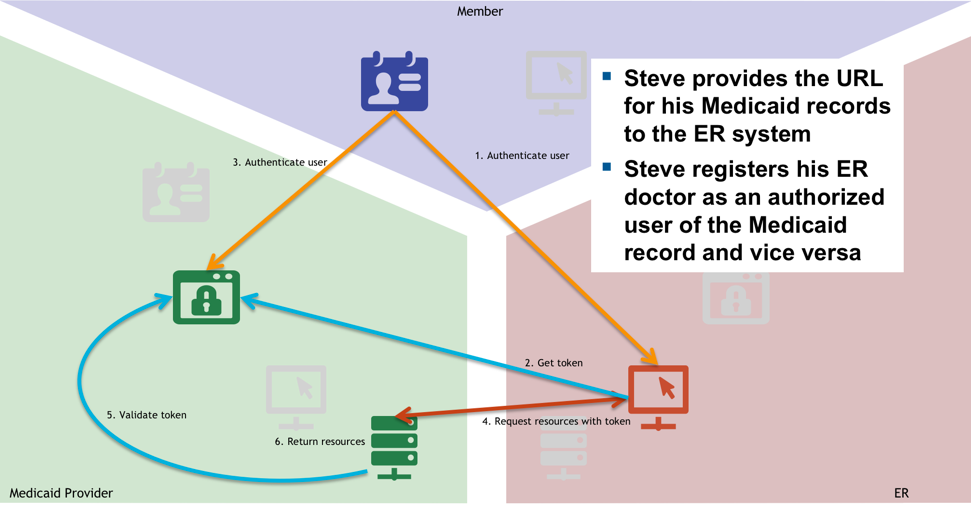


Figure A‑17: Steve allows the ER to access his Medicaid provider's records

The scenario in Figure A‑18, where Steve’s Medicaid provider accesses Steve’s medical record at the emergency room is the same as in Figure A‑16. In Figure A‑18, the Medicaid provider performs the same activity as Steve did in Figure A‑16. Otherwise, all of the other components in the two scenarios behave the same way.

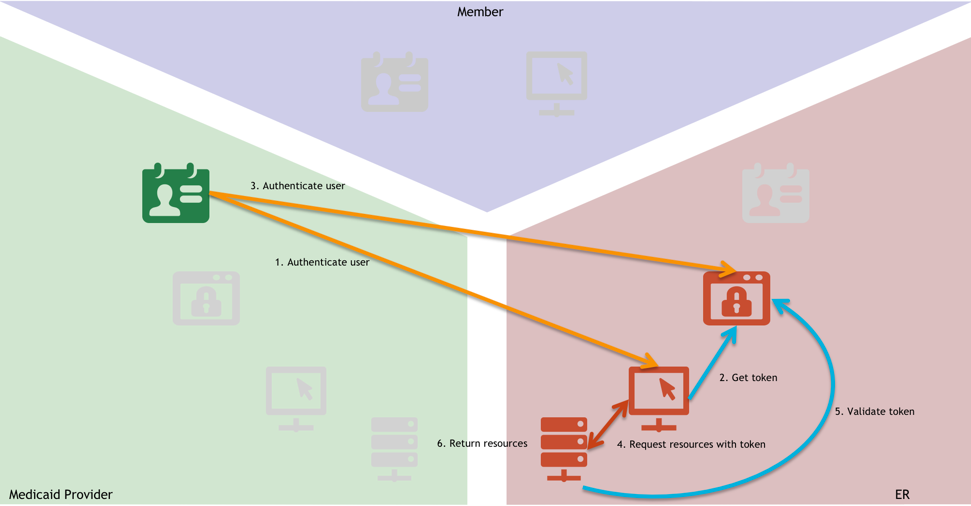


Figure A‑18: Steve's Medicaid provider logs in to access Steve's ER record

The last scenario, shown in Figure A‑19, shows the control flow when Steve’s Medicaid provider pulls Steve’s emergency room records back into the Medicaid system so that the his Medicaid provider has complete information about Steve’s medical history.

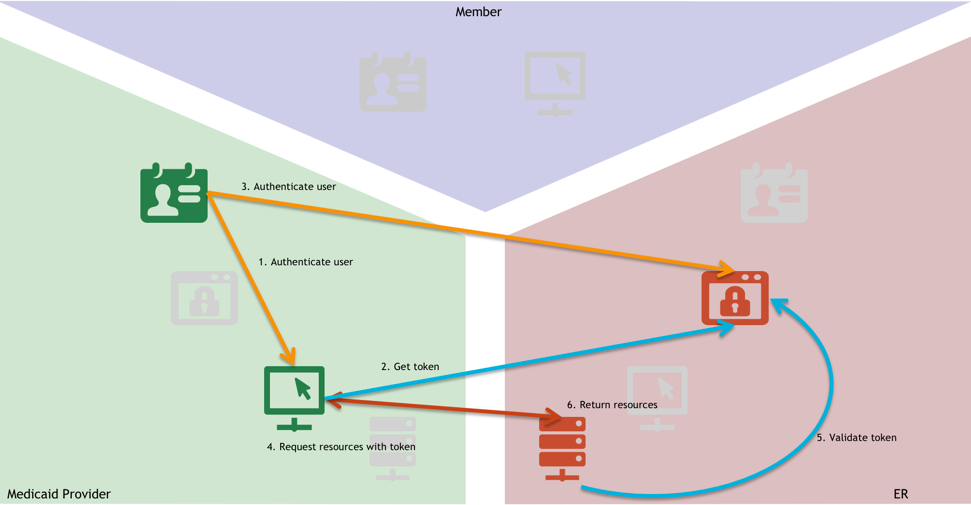


Figure A‑19: Steve's Medicaid provider pulls ER record into their system

1. Secure RESTful Interface Profile Security Analysis and Guidance – M. Russell, MITRE, July, 2014 [↑](#footnote-ref-1)
2. Secure RESTful Interface Profile Security Analysis and Guidance – M. Russell, MITRE, July, 2014 [↑](#footnote-ref-2)
3. “OAuth2 in Action” – J. Richer, A. Sanso, page 210. [↑](#footnote-ref-3)
4. <https://www.webpronews.com/faceboook-developers-get-access-to-user-phone-numbers-and-addresses/> [↑](#footnote-ref-4)