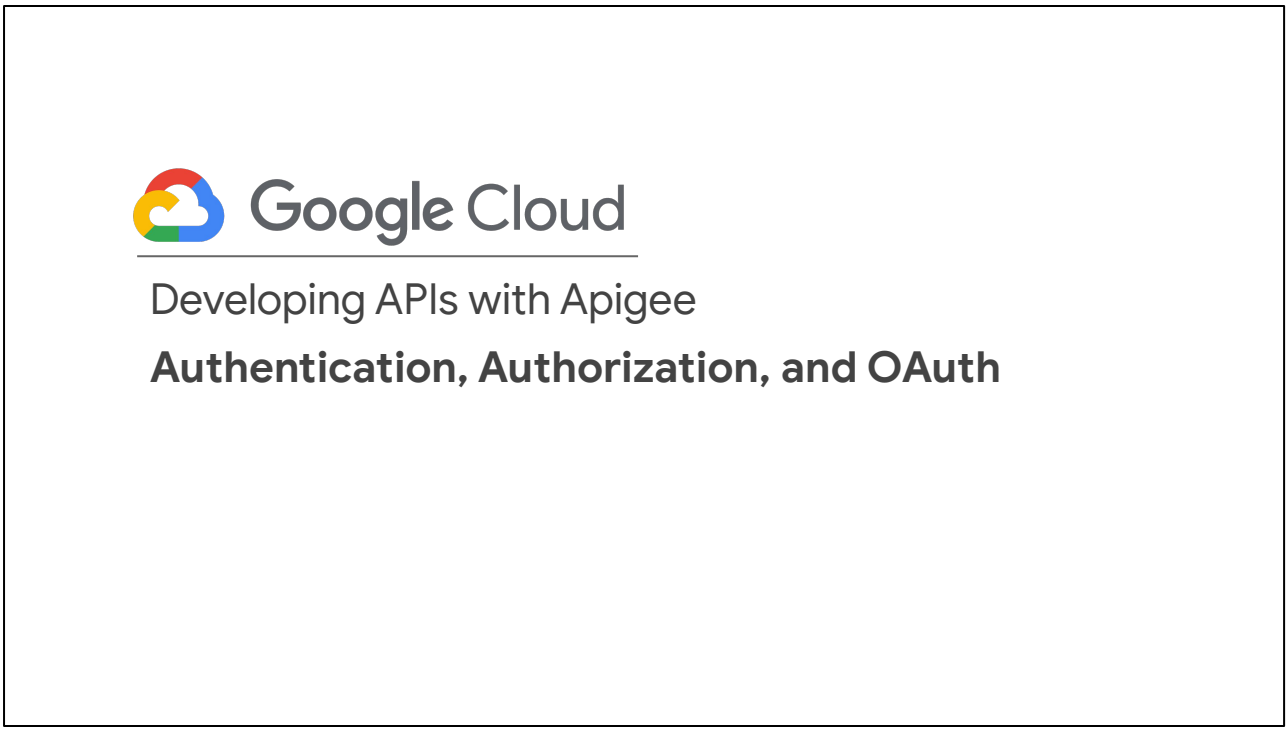


In this module, you'll learn about API security concerns.

You will also learn about OAuth, an authorization framework for REST APIs. We'll discuss the OAuth grant types, the different scenarios that OAuth was designed to handle, and learn the authorization flows of the grant types and how Apigee can implement those flows.

You will complete a lab where you will add OAuth to your retail API proxy.

Finally, you will learn about JSON Web Tokens, and the federated security standards, SAML and OpenID Connect.

Agenda

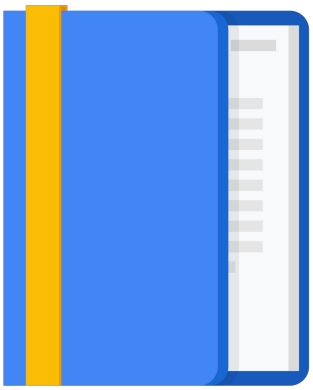
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

Quiz



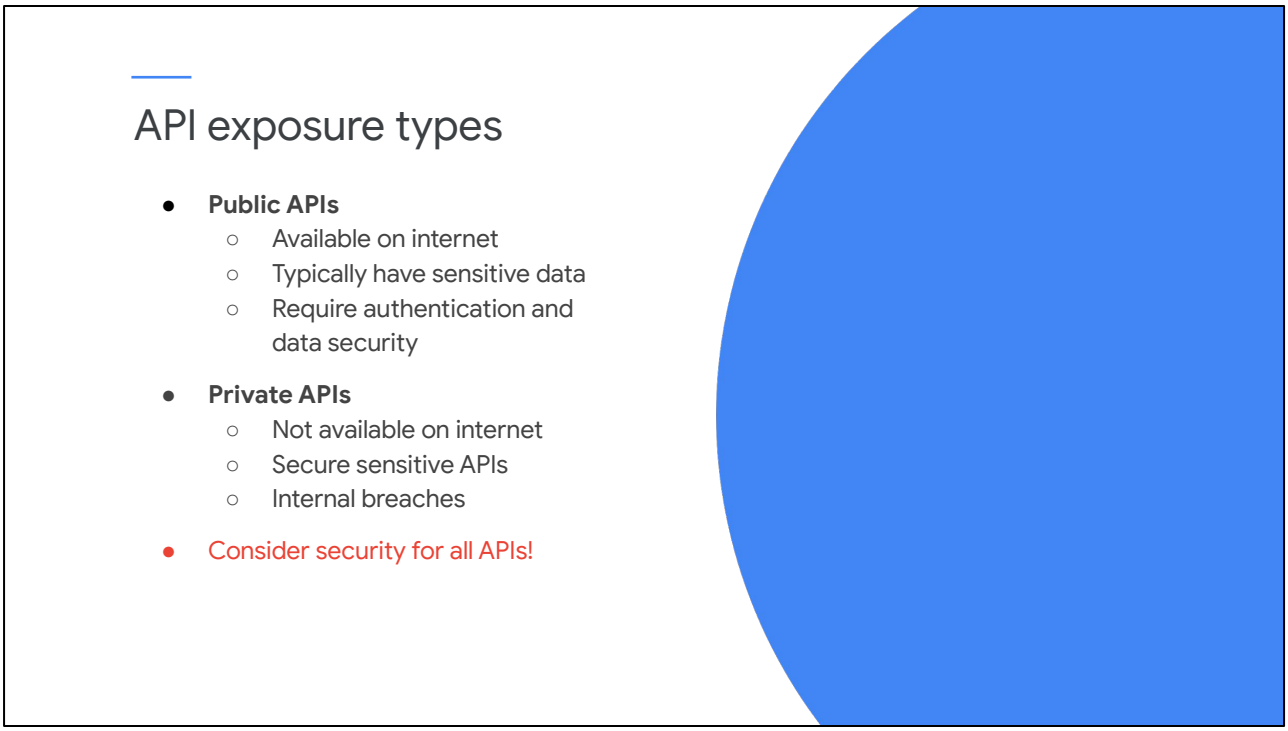
In this section you'll learn about API security concerns.

API security is vital to ensuring that your apps and backend services are not compromised. There are many different aspects to API security, and this lecture will introduce you to many of them.

We will be diving deeper into most of these topics in future sections.

If you read or watch the news, it seems like there are constant reports of data breaches. Many of those data breaches involve apps and APIs that were not properly secured.

As APIs become more and more important to companies, securing those APIs is equally important. You want to make sure you don't end up in the news for the wrong reason.



Before we look at the different types of API security, let's categorize API exposure types.

We'll call the first type "public APIs." Public APIs are available and discoverable on the internet. This public exposure means easy access for bad actors and generally makes these APIs more prone to attack.

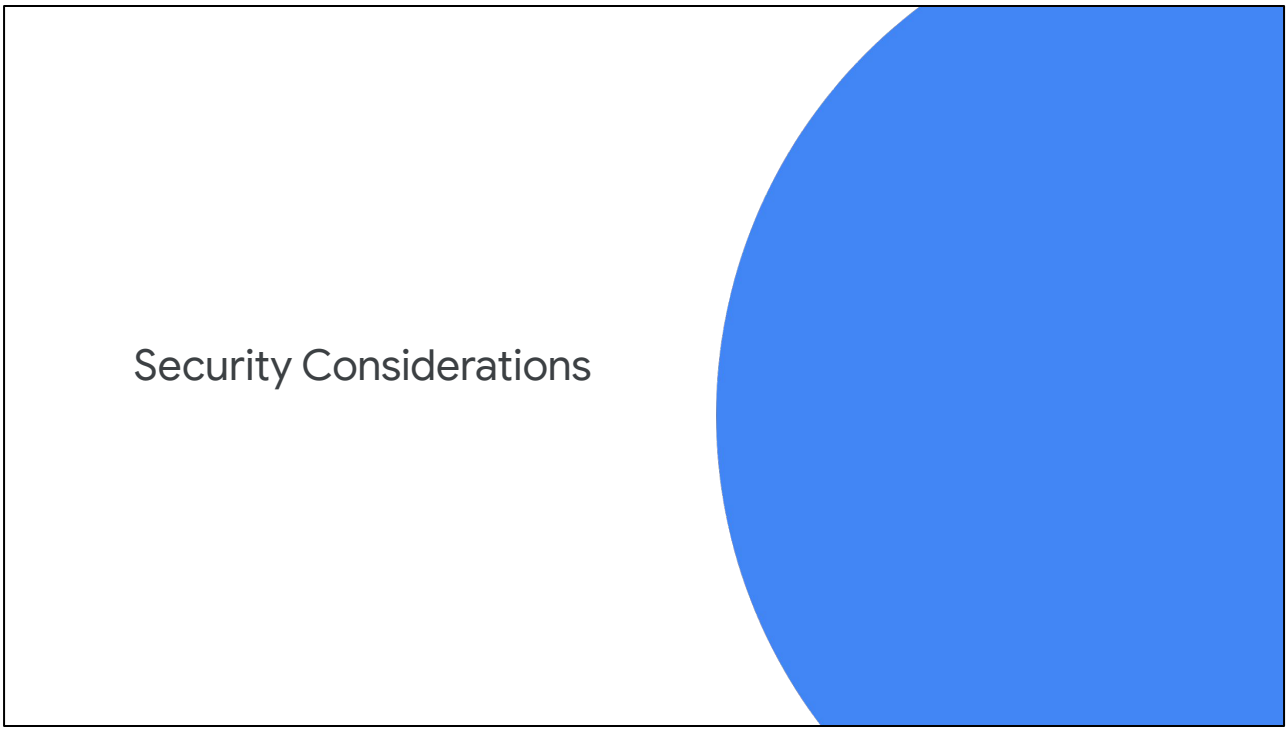
Some public APIs might not have any sensitive data, and it is possible that you won't have to provide significant security for these. This is the exception, though, and not the rule. Most public APIs still deal with sensitive data, so you'll definitely need to authenticate callers and secure the APIs.

The second type are "private APIs." We often think of these as "internal APIs." These APIs are not exposed on the internet.

We tend to worry less about securing these APIs, and, yes, the attack vector is smaller. However, any sensitive APIs should still be secured. User data should always be protected, even for APIs you think will never be made public. Some security breaches happen using internal access.

Note also that any sensitive APIs that are accessible outside your firewall should be fully authenticated and encrypted, even if the API's existence and details are not publicized.

The most important point to remember is that all APIs require thoughtful consideration of security. Even if you decide not to secure a particular API, which should be rare, you should always think about the security requirements when creating a new API.



Now let's discuss some security considerations for your APIs.

You'll learn about the different types of security required, and how Apigee can help you provide this security for your APIs.

Infrastructure and network security

- An API cannot be secure unless the underlying infrastructure is secure.
- Google Cloud and the Apigee API Platform are secure by design.
- Customers can leverage Google Cloud security tools, including Cloud Armor and ReCAPTCHA.

One important security consideration is the infrastructure and network security of the API management platform. APIs cannot be secure unless the platform upon which they are built is secure.

Google has years of experience designing data centers, infrastructure, and networks that are secure by design, and Google Cloud leverages this infrastructure to provide a secure platform for APIs you build on Apigee.

In addition, Google Cloud also provides powerful security tools that you can use to protect your APIs. Cloud Armor is a web application firewall that can protect your APIs against distributed denial of service and other types of attacks. ReCAPTCHA uses algorithms that apply continuous machine learning to protect against bots and other automated attacks.

Traffic management: Rate limiting

- Limit allowed rate of requests
- Spike Arrest policy
  - Smooths traffic using a maximum rate
  - Rate limit by app, user, IP address, or any other variable
- Quota policy
  - Limit allowed requests over specified period

You often need to limit the rate of requests coming into your APIs, especially for public APIs.

APIs can be overwhelmed when traffic is allowed to exceed backend service capacity. Rate limiting your APIs can reduce traffic to a reasonable level and allow requests to be serviced successfully. This can help ensure that app performance is maintained by not allowing certain apps or users to consume the majority of the bandwidth.

Apigee has two rate-limiting policies available for your API proxies.

The first is the SpikeArrest policy, which smooths traffic by allowing only a specific rate of traffic through. Overall traffic rates can be set for the API, or traffic can be rate limited by app, user, IP address, or any other variable.

The second policy is the Quota policy. This policy sets a limit on the allowed number of requests over a specified period of time. The Quota policy does not enforce a rate at which those requests can be made.

You'll learn more about the SpikeArrest and Quota policies in a later lecture.

Traffic management: Source IP

- Allow traffic only from specific IP addresses or ranges
- Access Control policy
  - Allows or denies by IP address or IP range

Sometimes you know that requests should only come from specific IP addresses or ranges, or you want to block traffic for specific regions or IP addresses.

The AccessControl policy can be used to allow or deny traffic based on incoming IP address. A series of rules can be set up to allow or deny access based on IP ranges or specific IP addresses.

Application identity

- Track API traffic by app
- Create API key for each app
  - Revoke an API key to remove access
- Verify API Key policy
  - Rejects requests without a valid API key

When you provide an API that is to be used by multiple apps, you want to be able to track which app is making a request, and you want to make sure that only apps registered for the API can make a request.

Apigee allows the creation of a unique API key for each app. API keys are associated with API products, which are used to grant access to specific APIs.

API keys can also be revoked by the API creator when it is necessary to remove access for these apps.

The VerifyAPIKey policy will block any traffic that does not provide an API key that allows access to the requested API.

User authentication and authorization

- App users must be authenticated and authorized.
- Apigee provides several policies for user authentication and authorization:
  - Basic Authentication
  - OAuth 2.0
  - JSON Web Tokens and Signatures (JWT/JWS)
  - Security Assertion Markup Language (SAML)
- Apigee proxies can be integrated with other security providers.

Whenever your APIs manage user data, it is vitally important to protect that data by authenticating and authorizing app users.

Apigee provides policies to help with authentication and authorization.

The BasicAuthentication, OAuth 2.0, JSON Web Token, and SAML policies can be used to provide authentication and authorization for your API proxies.

We will discuss all of these policies in later lectures.

Apigee proxies can also be used to integrate with other security providers or integrate with your own token services.

Data in transit

- Use Transport Layer Security (TLS) for all API traffic.
- Enforce TLS from client to Apigee and from Apigee to backend services.
  - Incoming requests can use TLS certificates hosted on load balancers in your Google Cloud project.
  - Requests to backend services can be secured using one-way or two-way TLS configured on Apigee.

Transport Layer Security, or TLS, should be used for all your APIs. TLS encrypts API requests and responses across the internet, keeping your API data private.

TLS should be enforced for requests from the client to your API proxies and for requests from your API proxies to your backend or third-party services.

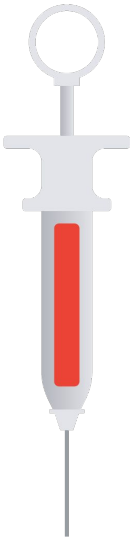
Incoming requests should be secured using a domain validation certificate on a load balancer hosted in your Google Cloud project.

Requests to backend services can be secured using TLS configured on Apigee. Both one-way and two-way TLS are supported.

We will discuss this in a later lecture.

Injection and data-based attacks

- API proxy provides security layer where inspection of incoming data can be performed.
- Regular Expression Protection policy
  - Configure regular expressions to detect dangerous content in any part of a request.
  - Rejects the request before sending to the backend.



APIs typically accept payloads of data to be saved in backend databases.

A well-known category of data attack is called injection attacks. The point of an injection attack is to send untrusted data to a service, causing the execution of the service to be altered. Your backend services may not be protected against all of these attacks, especially if they have previously only been used inside the company network.

The Apigee API proxy provides an opportunity to inspect incoming data before passing it on to your backends.

The RegularExpressionProtection policy allows you to craft regular expressions to detect dangerous content in the request. This policy can be used to check for patterns in the payload, headers, query parameters, or any other variables in your proxy. When a pattern match is found, the request is rejected, and the backend is not compromised.

Application-level denial of service

- Parsing malformed XML or JSON payloads can cause services to fail.

- The JSON Threat Protection policy detects malformed or unnecessarily large JSON payloads without loading them into a JSON parser.
- The XML Threat Protection policy does the same for XML.

Another type of data attack is an application-level denial-of-service attack, which tries to overwhelm a service so that it can't handle traffic.

One form of this attack uses large or malformed JSON or XML structures to cause issues when they are being parsed. The malformed JSON or XML can cause a parser to use excessive memory, which may greatly reduce API capacity or stop the operation of the service.

Apigee provides policies to protect against these large or malformed payloads. The JSONThreatProtection and XMLThreatProtection policies detect these dangerous payloads without parsing them, protecting both the proxy and the backend from adverse effects.

We will discuss these policies in a later lecture.

Data confidentiality

- Regulations and privacy concerns force API providers to limit access to user data.
- Role-based access control can limit internal access to production, including tracing of live production traffic.
- Data masking and private variables can be used to keep sensitive data from showing up in trace sessions.

It is very important to make sure you protect user data from being accessed by entities that should not be allowed access.

In addition to locking down your APIs with authentication and authorization, you also need to consider access by your internal users, those logging in to the Apigee management UI. Apigee provides some features to help you lock down internal access.

You can block most users from being able to access production and trace production traffic by using role-based access control. Role-based access control allows you to set up roles and assign them to your users. Permissions can be enabled and disabled for a role, providing only the access that is required for the role.

Support teams often need to be able to trace production traffic to help debug issues. You can keep users from seeing sensitive data when tracing production traffic by using the data masking and private variables features.

We will talk about all of these features in a later lecture.

Agenda

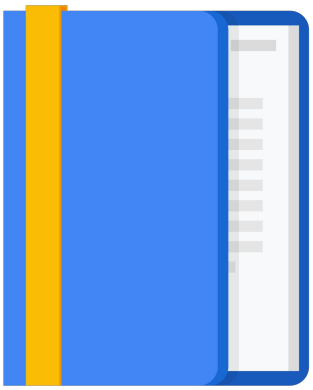
API Security Concerns

Identity, AuthN, and AuthZ

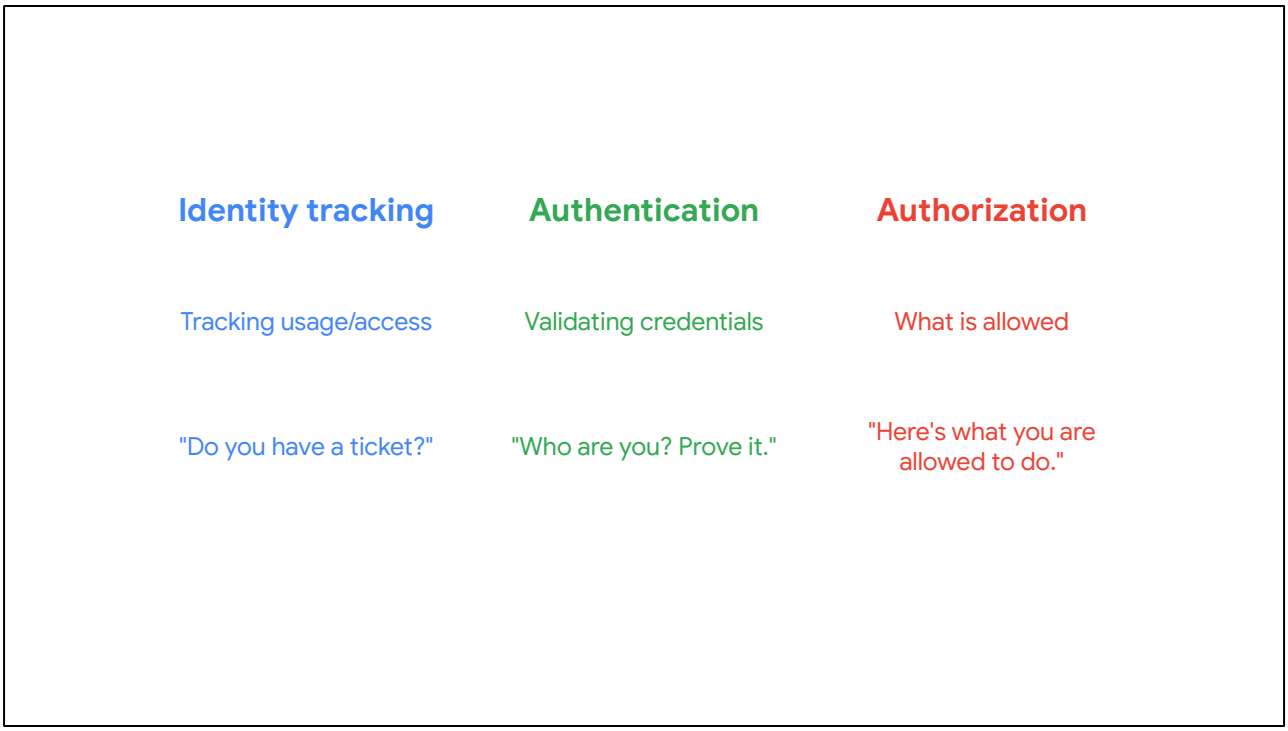
OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

Quiz



This section introduces the concepts of identity, authentication, and authorization with regard to APIs.



Let's explore the differences between identity tracking, authentication, and authorization.

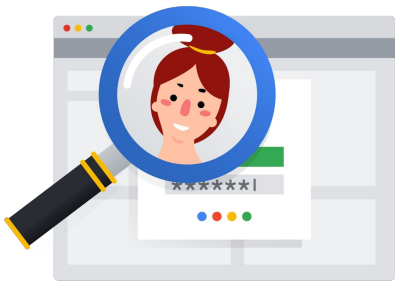
Identity tracking is the act of determining the identity of the app or the user for a request. Some part of the request is used to identify the caller or app. This is like asking someone for a ticket. A ticket can be required for access and may be associated with a particular person, but the ticket isn't proof that the person is who they say they are.

Authentication is similar to identity tracking, except that some confidential information is used to prove that someone is are who they say they are. Authentication is the process of validating credentials. Once the identity of the caller is authenticated, you can trust it.

Authorization is determining what the app or user is allowed to do. Authorization can only take place after the app or user has been authenticated, so that you are sure that you are granting access to the correct person or entity.

Identity tracking

- Apps are generally identified using API keys.
- Users are identified by username or account number.



Apps are generally identified using API keys. API keys are passed in with each request. This provides access to protected APIs and also allows you to track API usage by app.

Users can be identified by information like username or account number.

When you need proof of identity, use authentication.

Authentication

- App credentials are consumer key and secret.
- User credentials are often username and password.
- TLS certificate can be used for machine credentials.
- Credentials must be protected with Transport Layer Security (TLS).
- Standards: AD/LDAP and OpenID Connect

Authentication is the act of validating credentials to prove identity.

For apps, the credentials are the app's consumer key and secret.

User credentials are often username and password

A client certificate can also be used as credentials.

Because you will use credentials to provide access and authorization for your APIs, protecting credentials is important. All authentication traffic must be sent using TLS.

Authentication standards include Active Directory, LDAP, and OpenID Connect.

Authorization

- Only makes sense when user/app has been authenticated
- Proof of authorization often via token
- Standards: OAuth 2.0, SAML 2.0

Authorization is determining what a user or app is allowed to do.

Authorization only makes sense if the user or app has been authenticated.

Tokens are often presented as proof of authorization.

API authorization standards include OAuth 2.0 and SAML 2.0.

Common patterns of usage

API key:
https://api.example.org/api/v1/resource?apikey=45c78ece5b77647854a
ApiKey: 45c78ece5b77647854a *(header — preferred)*

Basic authentication:
Authorization: Basic YmFkOnBhc3N3b3Jk *(basic auth header)*

Access token:
Authorization: Bearer 4WCAchNNtVyK8JsACl1HP7ml *(bearer token header)*

Here are examples of how identity, authentication, and authorization are specified in an API request.

An API key is often passed using a query parameter. It is recommended, though, that the API key be passed in a header, because a header is less likely to be logged in access logs.

The authentication for a user or app is usually put in a basic authentication header. The Basic Auth header is used when passing some form of a username and password in an Authorization header. For a user, it might be a username and password, and for an app, it is generally the consumer key and secret.

An access token, which is used for authorization, is generally passed in the Authorization header as a bearer token.

Agenda

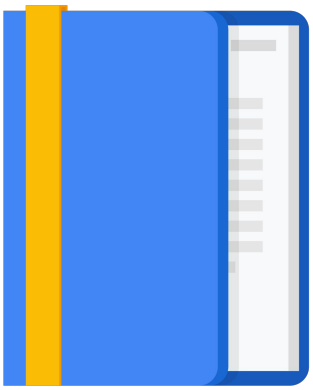
API Security Concerns

Identity, AuthN, and AuthZ

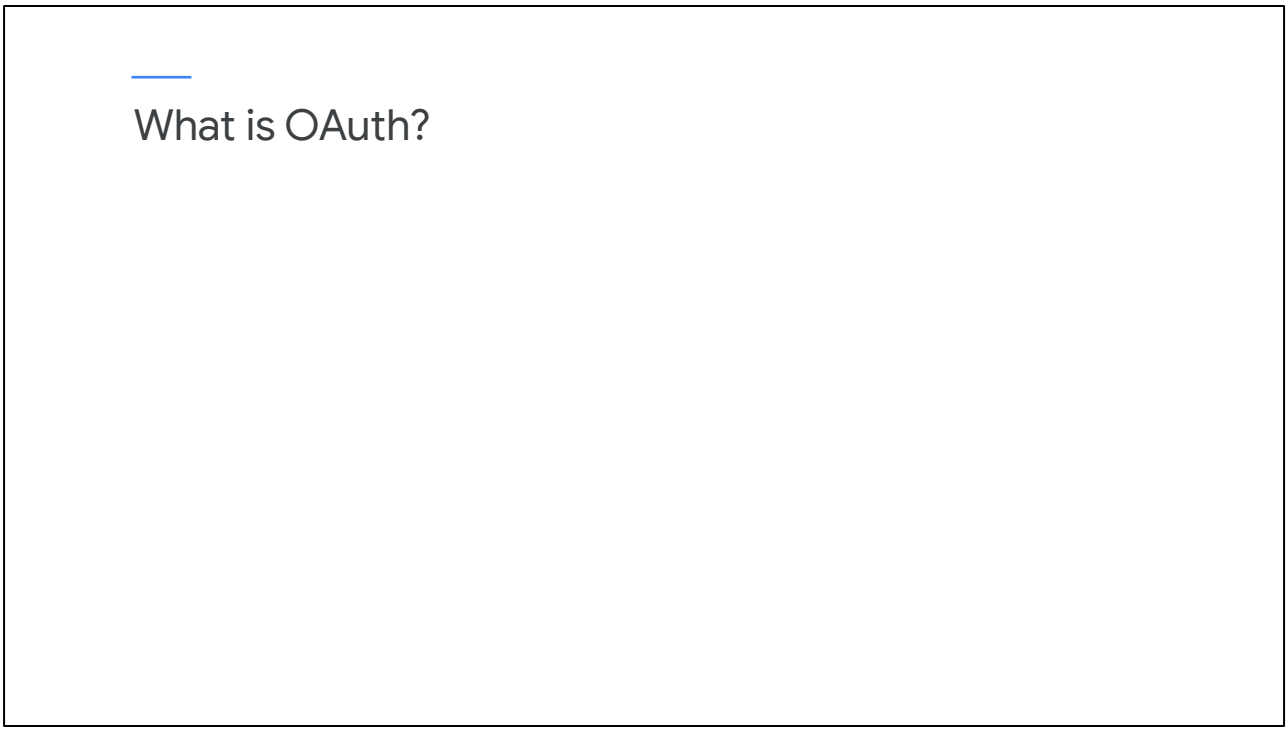
OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

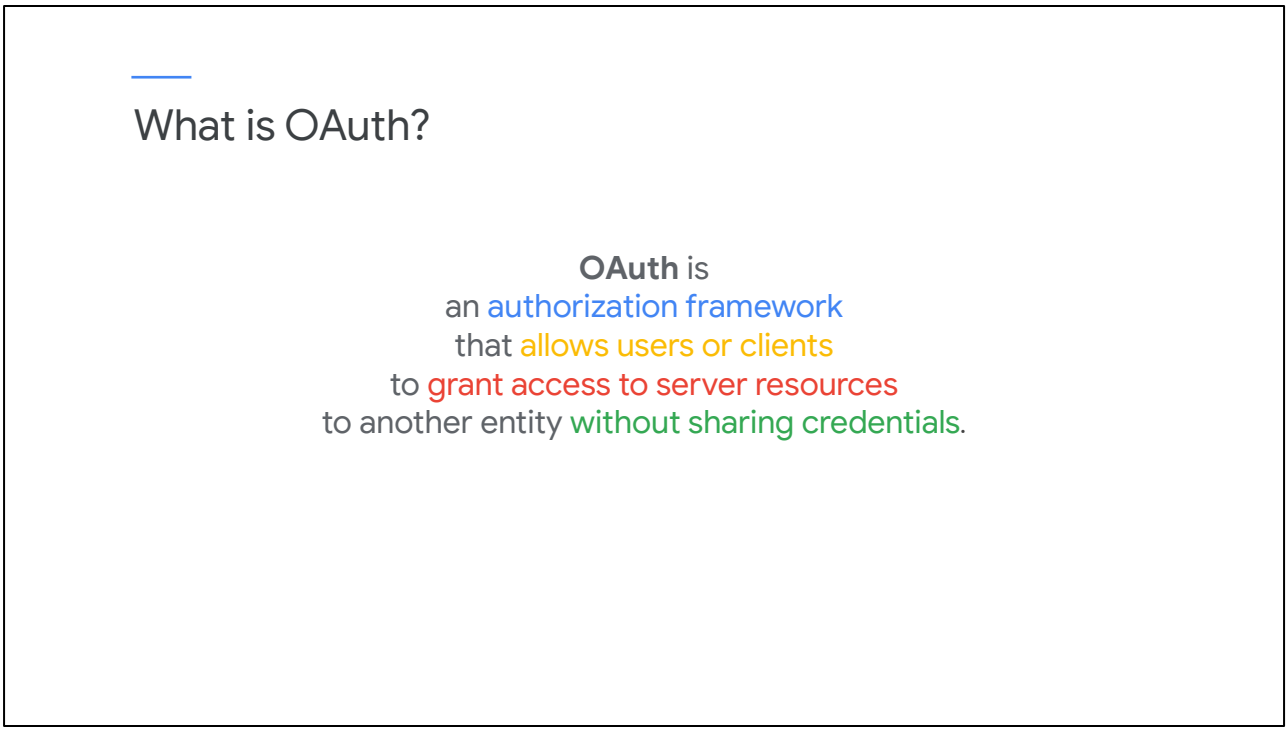
Quiz



Let's discuss OAuth, an authorization framework for APIs.



The first obvious question is, what is OAuth?



OAuth is an authorization framework that allows users or clients to grant access to server resources to another entity, without sharing credentials.

Let's parse this statement.

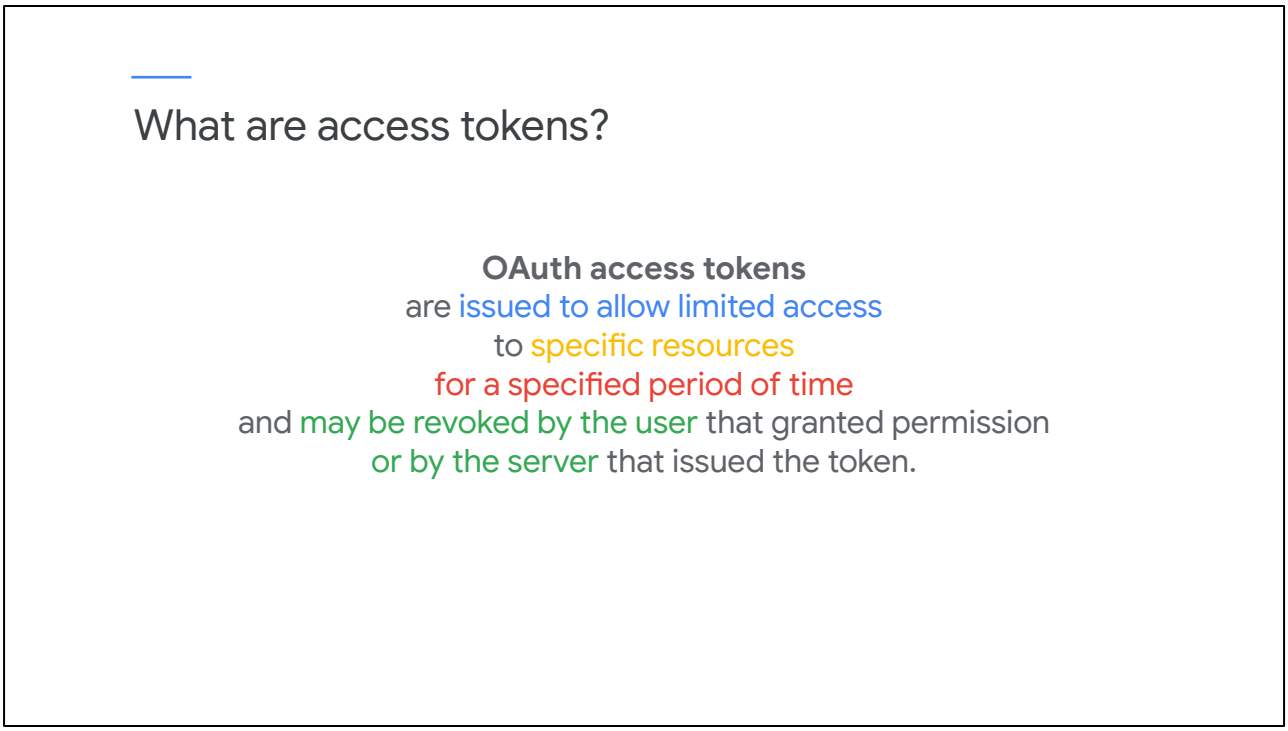
OAuth is an authorization framework for APIs, which can be used to control permissions for accessing APIs.

There are currently two versions of OAuth: 1.0a and 2.0. Version 1.0a is deprecated. Version 2.0 is the current industry standard, and this is supported by Apigee. When we discuss OAuth during the rest of this course, we are referring to OAuth 2.0.

OAuth allows users or clients to grant access to server resources to another entity without sharing credentials.

This is a key feature of OAuth; you can grant access to resources without sharing the user or client credentials that would typically be used to access the resources on the resource server.

You grant access by using an access token.



OAuth access tokens are issued to allow limited access to specific resources for a specified period of time and may be revoked by the user that granted permission or by the server that issued the token. We'll parse this statement too.

Access tokens are issued to allow limited access to specific resources.

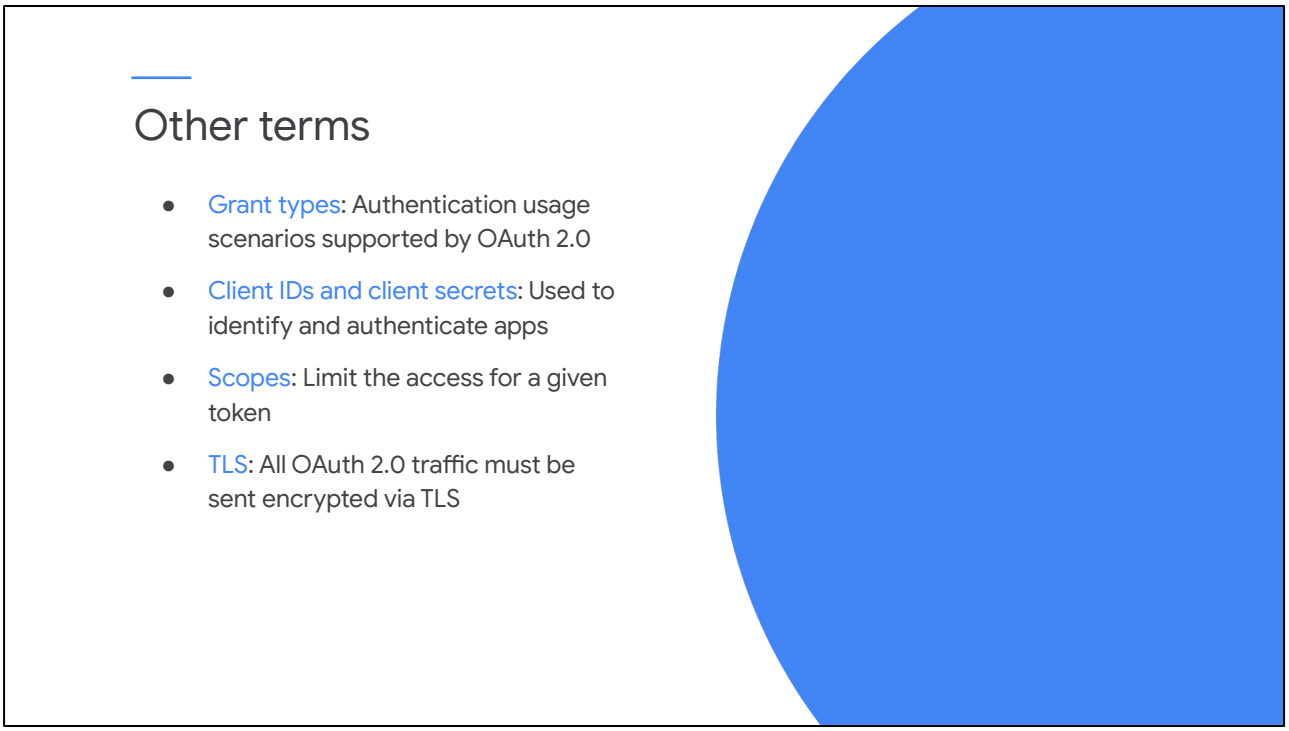
An authorization server issues OAuth tokens. In the examples we'll be seeing in this course, Apigee is acting as the authorization server. You may, however, want another authorization server to issue the tokens, maybe one in your backend data center, and Apigee can use that model too.

Limited access to specific resources means that the token does not need to provide full access to the resources of the user

Access tokens also are valid for a limited time. Limiting the time for an access token limits the exposure in case the token is compromised.

Access tokens can also be revoked, making them no longer valid. A revoked token has no access. Revocation can be initiated by either the user that originally granted permission to access the user resources or by the issuing server. If either the user or issuing server believes the access token has been compromised, the token should be revoked.

We will learn more about access tokens during the next several lectures.



Let's introduce some other terms you'll need to know.

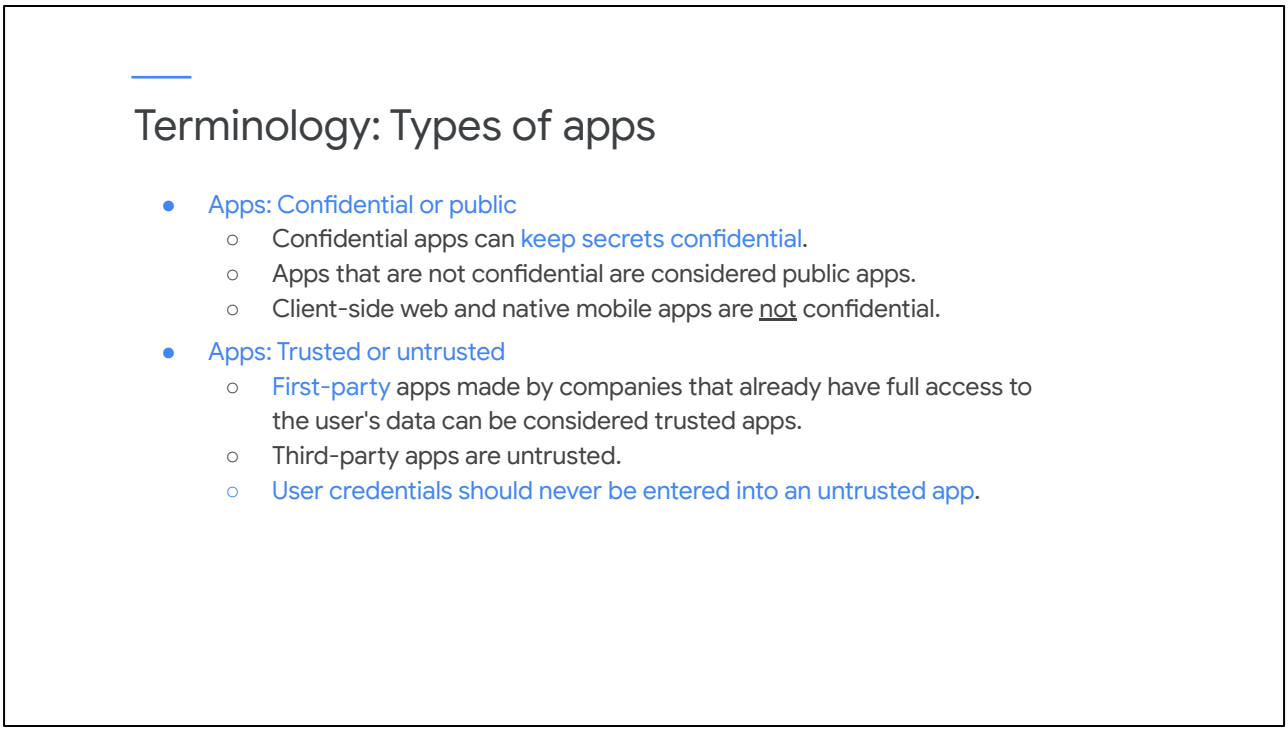
Grant types specify the different kinds of authorization scenarios that OAuth has been designed to handle. There are four grant types specified by the OAuth 2.0 specification.

OAuth client IDs and secrets are used to identify and authenticate apps. For Apigee apps, the consumer key and consumer secret are used as the client ID and client secret respectively.

Scopes can be used to limit the access for a specific access token, granting permission only for necessary operations and access.

I've already mentioned TLS, or Transport Layer Security. You may also know it as SSL, or Secure Sockets Layer, or you may know it as https. TLS is the successor to SSL, and SSL should no longer be used.

TLS is how API requests and responses are encrypted between the client and service. OAuth has a strict requirement that all OAuth traffic be encrypted, including any API request containing a token. This will protect entities like tokens, client IDs, and client secrets from being compromised.



Here is some terminology we'll use regarding types of apps.

First, apps can be considered confidential or public.

Confidential apps can keep secrets confidential. This means that tokens and client secrets are stored in a way that they cannot be accessed.

Public apps are those that are not confidential. You can't store secret information in public apps because they cannot be secured.

A couple of examples of public apps are client-side web apps and native mobile apps. The storage for code running locally in a browser or in a mobile app on a phone or tablet can be accessed by a bad actor.

Whether an app is confidential or public affects how OAuth is used for the app.

Second, apps can be considered trusted or untrusted.

Trusted or untrusted indicates whether the app could be trusted with seeing your password.

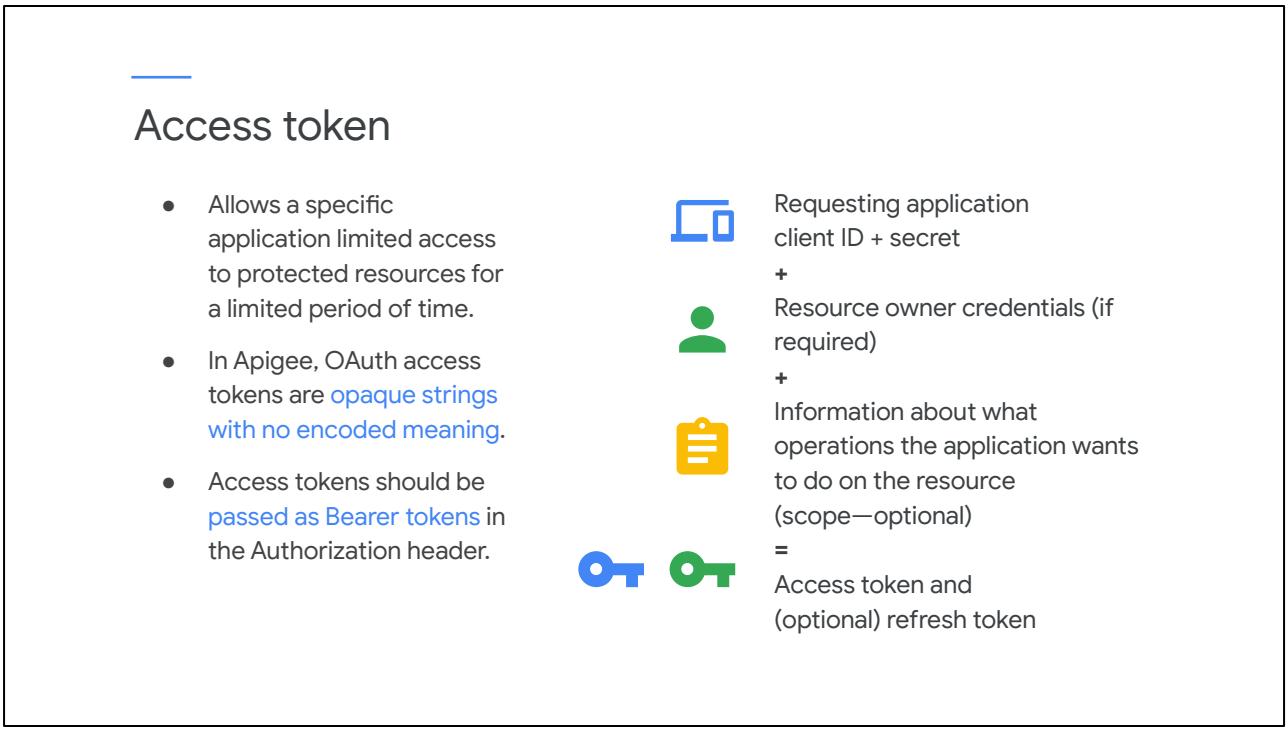
First-party apps made by a company that is also storing the data being accessed may be considered trusted apps.

Third-party apps would always be untrusted. User credentials should never be entered into untrusted apps. For example, if you are logging into a game app using your Google account to confirm your identity, you would definitely not want the game developer to have access to your Google password. This game app would be an untrusted app.

OAuth can be used to allow that app to get specific information about you from your Google account without ever providing it with your password or giving it full access to your account.

We will learn more about how this works in a later lecture.

You should generally classify your app as public or confidential, and trusted or untrusted. This will help you choose the OAuth grant type to use.



Let's learn a bit more about access tokens.

As we said before, an access token allows a specific application limited access to protected resources for a limited period of time.

Apigee OAuth access tokens are opaque strings. There is no encoded or encrypted information in the token: an access token is just a random set of characters. This means that all of the interesting information about the token is stored in the authorization server.

When you use an access token in a request to a protected resource, put the access token into the Authorization header, and pass it as a bearer token. We'll see how to do that later.

How do you get an access token?

First, the application needs to authenticate with the authorization server. In OAuth, this is done using the client ID and client secret of the app. In apps created on Apigee, they are called the consumer key and consumer secret.

Second, the owner of the resource or resources, if it is the user of the app, needs to authenticate with the authorization server using their credentials. This confirms that the user of the app is who they say they are, and that they want to grant the application access to the resource or resources.

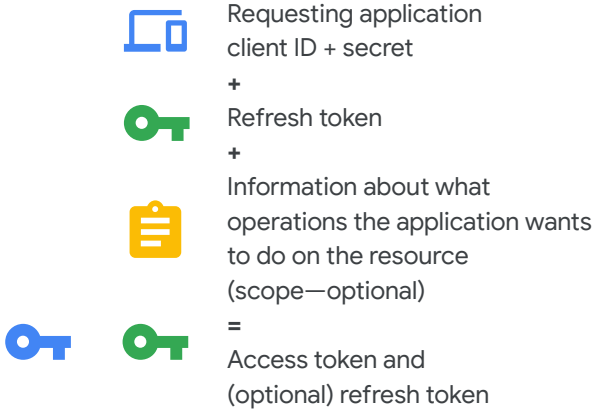
OAuth can also be used to access non-user resources like application data. In this case, there would be no resource owner credentials, just the application credentials in the form of client ID and secret.

Third, a scope can be provided, which is optional. This would indicate the level of access that should be allowed for the token.

The app credentials, resource owner credentials, and scope can be used to get an access token, and optionally a refresh token.

Refresh token

- Provides a limited right to reauthorize the granted access by obtaining new access tokens without user credentials.
- In Apigee, OAuth refresh tokens are also opaque strings with no encoded meaning.



A refresh token is used to reauthorize the granted access by allowing the app to obtain an access token without the user credentials. The refresh token would be used when the previous access token expires and is no longer valid.

Like the access token, a refresh is an opaque string.

How do you use a refresh token to get a new access token?

Just like the original access token request, the app provides its client ID and secret to authenticate with the authorization server.

In this case, the refresh token is used instead of the resource owner credentials. This means that the user would not need to re-enter their username and password.

An access token generally has a fairly short lifespan, to limit the damage that could be done if it were compromised. A common expiration time for access tokens is five minutes. Can you imagine the user experience for an app if the user had to enter their username and password every five minutes?

This is why we use refresh tokens.

The scope is again optional. You cannot use a scope that has more access than was originally used when creating the first access token.

If the app provides its client ID and secret, the refresh token, and optionally a scope, a new access token can be generated. A new refresh token can also be created.

In a later lecture we will see why it is recommended to create a new refresh token every time you use a refresh token.

Scope

Scope 1: "READ"
- GET /photos
- GET /photos/{id}

Scope 2: "UPDATE"
- GET /photos
- GET /photos/{id}
- POST /photos
- PUT /photos/{id}

- Identify what an app can do with the requested resources.
- Scope names are defined by the authorization server.
- Blocking of requests based on scope needs to be implemented within the proxy.

A scope identifies what an app can do with the requested resources when using the access token. The authorization server defines both the scope names and the access allowed for a particular scope. Requests containing tokens without the correct scope can be blocked by an API proxy using the verify access token policy.

Let's look at the example on the left.

For this photo editing service, there are two scopes. The first scope is called READ, and it allows read-only operations on the user's photos. The second scope, UPDATE, allows the read operations, but also allows creation of new photos using POST and updates to existing photos using PUT.

OAuth grant types

Grant Type	User Resources?	Use Case
Client Credentials	NO	System-to-system interactions Resources owned by partner, not a specific user
Resource Owner Password	YES	Requesting app is trusted Resources owned by user
Authorization Code	YES	Requesting app is NOT trusted Resources owned by user
Implicit	*YES*	*Designed for public browser-based or mobile apps* ***IMPLICIT NOT RECOMMENDED; USE AUTH CODE***

Remember that OAuth grant types specify the different authorization scenarios that OAuth is designed to handle. We will review the details of the different grant types in the next lectures, but let's discuss the different grant types and when you should use them.

First is the client credentials grant type. It is designed for system-to-system interactions. For example, this may be a partner service calling into your API. It is used only when there are no user resources involved.

The second is called the resource owner password grant type, often just called the password grant type. This grant type and the following grant types can be used when user resources are involved.

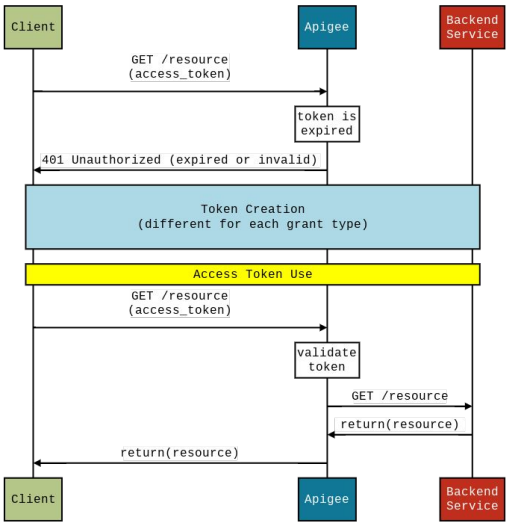
The password grant type can only be used when an app is trusted. For this grant type, the user's credentials are passed through the app to the authorization server. Because the app has access to the user's credentials, it makes sense to use this grant type only if the app was created by the same company that hosts the resources being accessed.

The third is the authorization code grant type, also called the auth code grant type. This grant type can be used when the app is untrusted. We'll see later how the auth code grant type allows the user to pass credentials to the authorization server without the app gaining access to them.

The fourth is the implicit grant type. This was originally designed for client-side browser or mobile apps. Remember, we called these public apps, as opposed to confidential apps.

The implicit grant type is no longer recommended, even when using a public app. Instead, there is a way to use the auth code grant type with a public app, and you'll learn how to do that when we discuss the auth code grant type in a later lecture.

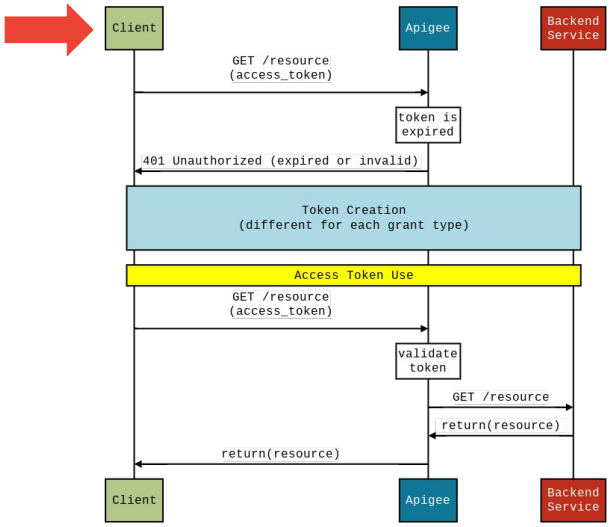
Common pattern for all grant types



All of the grant types follow a common pattern.

The app determines a need for a token, requests a new token, and finally uses the token to gain access to the protected resource.

Common pattern for all grant types



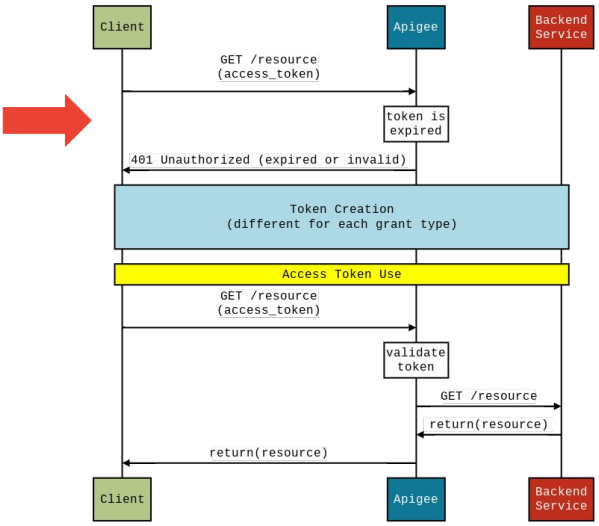
Note the three entities shown at the top of the interaction diagram.

The client is the app making the requests.

Apigee is the API gateway. Tokens are issued and validated by proxies running on Apigee.

The backend service provides access to protected resources.

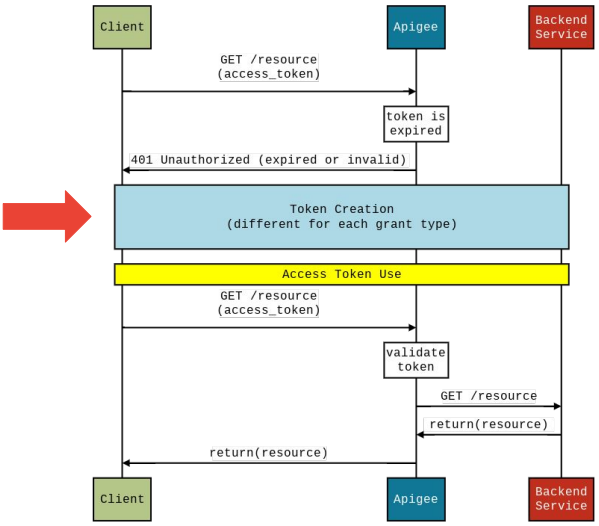
Common pattern for all grant types



First the client determines the need for a new access token. This often happens when apps try to access a resource using an access token that has expired. The Apigee proxy detects that the token is no longer valid and rejects the request, returning 401 Unauthorized.

The request to access the resource is not sent to the backend.

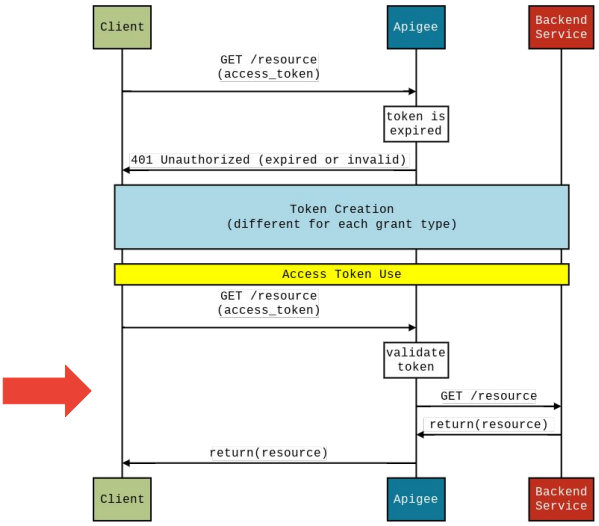
Common pattern for all grant types



The client then requests creation of a new token.

This process is different for each grant type, and we'll cover each grant type's flow during the next lectures.

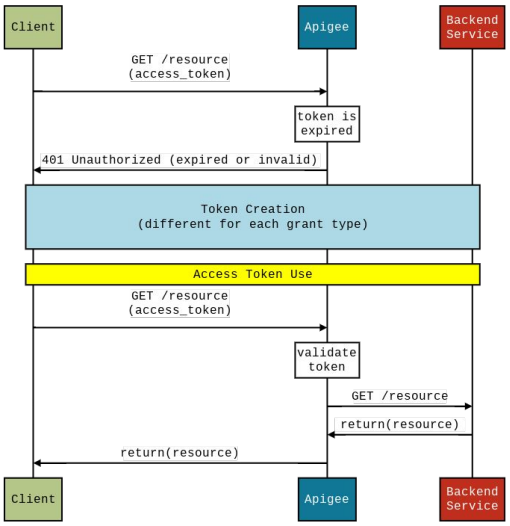
Common pattern for all grant types



After getting a new access token, the client retries the request.

Apigee validates the token, and because it is valid, proxies the request to the backend service and retrieves the resource.

Common pattern for all grant types



Notice that once a token is created, it is generally used in the same way regardless of the grant type being used.

In the next lectures you will learn much more about the specific grant types, focusing on use cases and how tokens are created.

Agenda

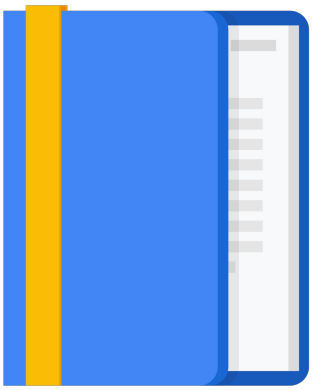
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

Quiz



Let's talk about the client credentials grant type.

OAuth grant types

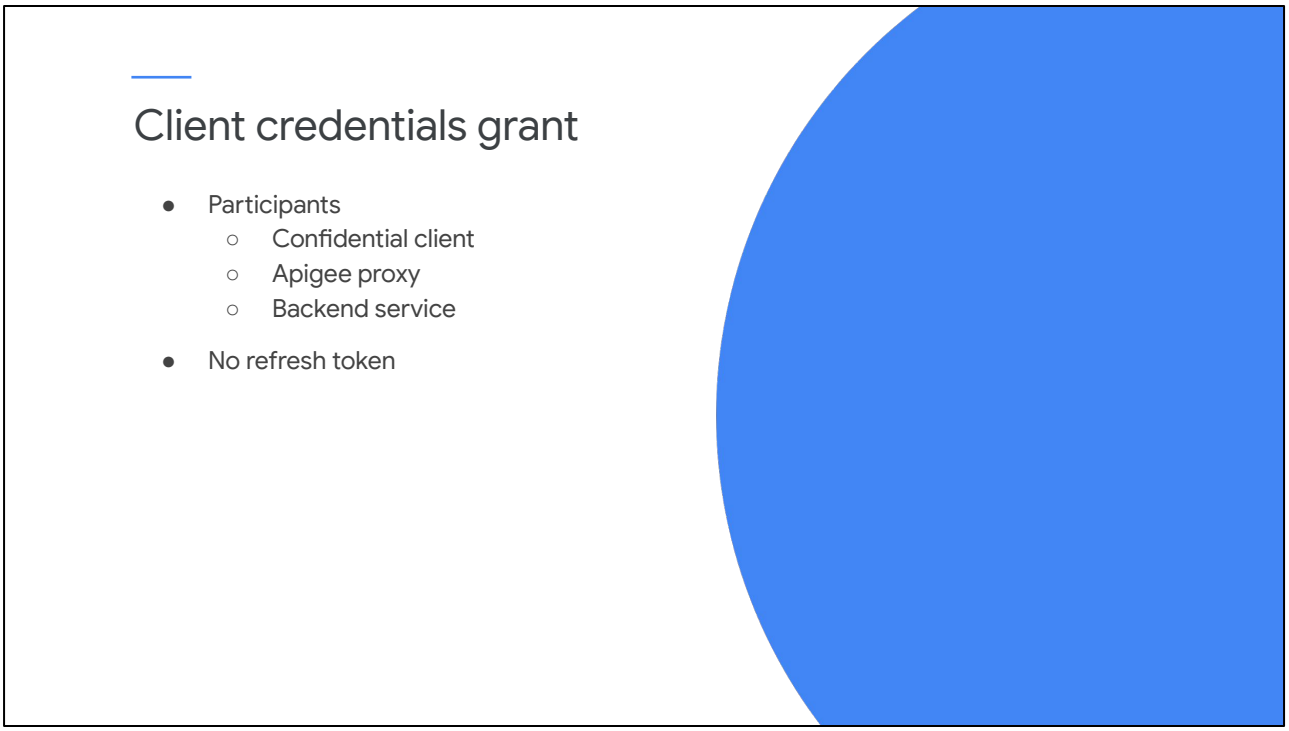
Grant Type	User Resources?	Use Case
Client Credentials	NO	Business system interactions Resources owned by partner, not a specific user
Resource Owner Password	YES	Requesting app is trusted Resources owned by user
Authorization Code	YES	Requesting app is NOT trusted Resources owned by user
Implicit	*YES*	*Designed for client-side browser-based or mobile apps* ***IMPLICIT IS NO LONGER RECOMMENDED***

We will review the use cases for the client credentials grant type, learn the details and interactions for the grant type, and see which policies should be used and how they should be configured.

The client credentials grant type is generally used for business system, server-to-server interactions. This is often used for providing access to internal or partner data.

Another use case is authorization for publicly available APIs. These services may be paid or may have usage limitations. Using the OAuth client credentials grant type is an excellent way to control access to these services.

The number one rule to remember for the client credentials grant type is to never use it when protected user data is being accessed. This grant type does not collect any user credentials, so the user has no chance to authenticate or consent to an app gaining access to the data.



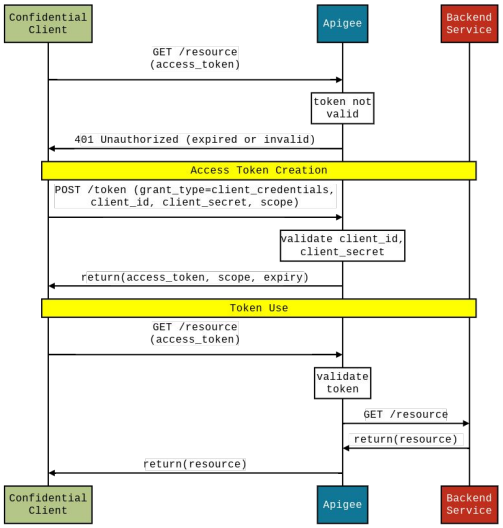
The first participant in the client credentials grant type interaction is the confidential client consuming the API. The client must be confidential, because the client credentials grant requires a client secret.

The Apigee proxy generates tokens. It also rejects requests to the backend if the provided token is not valid.

The last participant is the backend service, which serves the protected resources.

Unlike the other grant types, the client credentials grant type does not return a refresh token. A refresh token is used to avoid sending in user credentials when refreshing an access token, and because there are no user credentials involved when using the client credentials grant type, the refresh token is unnecessary.

Client credentials



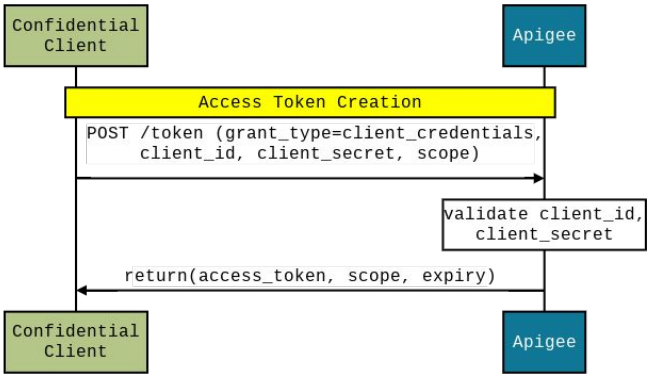
This is the overall flow for the client credentials grant type.

Access tokens are used in the same way for all grant types, but the access token creation process is different for each type.

The overall flow diagram will be shown for each grant type, and the intention is to show you the relative complexity of each grant type.

For each grant type we will look at the detailed flow for token creation.

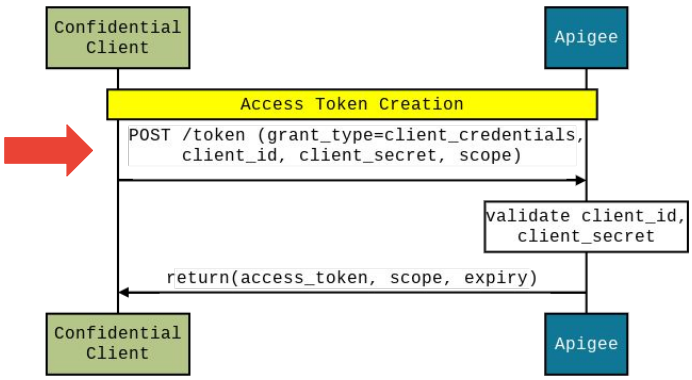
Client credentials: Token creation



Let's look at how tokens are created for the client credentials grant type.

The first thing to note is that the backend service has no part in token creation for any of the grant types.

Client credentials: Token creation



The client makes the client credentials request by sending a POST request to a token endpoint. In this case we show a POST to /token, but your API path can be different. It is important to have the request be a POST, because GET requests can be bookmarked or cached, which is not appropriate for token requests. Typically the OAuth token creation functionality is implemented in a proxy that is separate from the proxies that use your backend services.

The POST parameters are sent using form parameters in the request payload. The grant type is specified as client credentials.

The client ID and client secret for the app are also sent, which allows the app to be authenticated by the OAuth proxy. For all grant types, a token should only be used by the app that created it.

An optional scope is passed in as well.

The OAuth proxy validates the client ID and secret, and, if valid, returns an access token and information about the scope of the token and the expiration time.

Remember that there is no refresh token generated, because there are no user credentials for this grant type.

OAuthV2 policy (GenerateAccessToken)

```
<OAuthV2 continueOnError="false" enabled="true" name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>86400000</ExpiresIn>
  <SupportedGrantTypes>
    <GrantType>client_credentials</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
</OAuthV2>
```

The OAuth proxy will use an OAuthV2 policy to create the token.

The OAuthV2 policy is used in many different ways in our proxies. Its behavior is controlled by the operation element. In this case, the operation is GenerateAccessToken.

OAuthV2 policy (GenerateAccessToken)

```
<OAuthV2 continueOnError="false" enabled="true" name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>86400000</ExpiresIn>
  <SupportedGrantTypes>
    <GrantType>client_credentials</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
</OAuthV2>
```

The ExpiresIn element specifies the time-to-live for the token in milliseconds.

OAuthV2 policy (GenerateAccessToken)

```
<OAuthV2 continueOnError="false" enabled="true" name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>86400000</ExpiresIn>
  <SupportedGrantTypes>
    <GrantType>client_credentials</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
</OAuthV2>
```

The GrantType element specifies where the grant type should be found in the request.

This example shows the grant type being passed in the form parameter grant_type. This form parameter is the default location; if the GrantType element is not included, the grant_type form parameter is the expected location.

OAuthV2 policy (GenerateAccessToken)

```
<OAuthV2 continueOnError="false" enabled="true" name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>86400000</ExpiresIn>
  <SupportedGrantTypes>
    <GrantType>client_credentials</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
</OAuthV2>
```

The GenerateResponse element specifies whether the policy should automatically generate the API response payload. When the policy runs, it creates lots of variables for all of the information about the token.

You can create your own API response format by setting GenerateResponse's enabled field to false, and then using the variables set by the policy to craft your own response.

API call: Get token

- The client ID and secret are passed in the Authorization header.

Request

```
POST https://api.example.org/oauth/token

Headers:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=client_credentials
```

Response

```
{
  "issued_at": "1565729213000",
  "application_name": "030fdcea-cf97-12084aea513c",
  "scope": "READ",
  "status": "approved",
  "api_product_list": "[gold]",
  "expires_in": "86400",
  "developer.email": "bob@example.com",
  "organization_id": "0",
  "token_type": "BearerToken",
  "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
  "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
  "organization_name": "apigee",
  "refresh_token_expires_in": "0",
  "refresh_count": "0"
}
```

Let's look at an example request and response when getting a new token using the client credentials grant type.

We POST our request to the token endpoint.

API call: Get token

- Client ID and secret passed in Basic Auth header.

Request

```
POST https://api.example.org/oauth/token

Headers:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=client_credentials
```

Response

```
{
  "issued_at": "1565729213000",
  "application_name": "030fdcea-cf97-12084aea513c",
  "scope": "READ",
  "status": "approved",
  "api_product_list": "[gold]",
  "expires_in": "86400",
  "developer.email": "bob@example.com",
  "organization_id": "0",
  "token_type": "BearerToken",
  "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
  "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
  "organization_name": "apigee",
  "refresh_token_expires_in": "0",
  "refresh_count": "0"
}
```

The client ID and client secret are encoded in the Authorization header. We use basic authentication, often shortened to basic auth.

The credentials are the app's consumer key and consumer secret, separated by a colon, with that entire string being base64 encoded. The header value is the capitalized word "Basic" and then a single space, followed by the base64 encoded credentials.

It is important to note that base64 encoding is not encryption. If you have base64 encoded credentials, you can decode the string and retrieve the original credentials.

We count on TLS to encrypt the request and response, protecting the data while it travels across the network.

API call: Get token

- Client ID and secret passed in Basic Auth header.

Request

```
POST https://api.example.org/oauth/token

Headers:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=client_credentials
```

Response


```
{
  "issued_at": "1565729213000",
  "application_name": "030fdcea-cf97-12084aea513c",
  "scope": "READ",
  "status": "approved",
  "api_product_list": "[gold]",
  "expires_in": "86400",
  "developer.email": "bob@example.com",
  "organization_id": "0",
  "token_type": "BearerToken",
  "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
  "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
  "organization_name": "apigee",
  "refresh_token_expires_in": "0",
  "refresh_count": "0"
}
```


The payload contains URL-encoded form parameters. Specifying the content type header as application slash x-www-form-urlencoded indicates that the payload contains form parameters.

For the client credentials grant type, the only required form parameter is grant_type, although you may pass in a scope if necessary.

The response shows the default response format returned by Apigee.

API call: Get token

- Client ID and secret passed in Basic Auth header.

Request

```
POST https://api.example.org/oauth/token

Headers:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=client_credentials
```

Response

```
{
  "issued_at": "1565729213000",
  "application_name": "030fdcea-cf97-12084aea513c",
  "scope": "READ",
  "status": "approved",
  "api_product_list": "[gold]",
  "expires_in": "86400",
  "developer.email": "bob@example.com",
  "organization_id": "0",
  "token_type": "BearerToken",
  "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
  "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
  "organization_name": "apigee",
  "refresh_token_expires_in": "0",
  "refresh_count": "0"
}
```

The access token and expiration time are typically important for the app.

The expiration time returned here is in seconds. The policy configuration uses milliseconds.

Many of the other fields have default values or are unnecessary, so you may choose to return a simpler response format.

API call: Use token

- The token is passed as Bearer token in the Authorization header.
-  OAuthV2 policy (VerifyAccessToken) will validate the access token for all grant types.

```
Request:
GET https://api.example.org/orders/v1/items/14

Header:
Authorization: Bearer 4WCAchNNtVyK8JsACl1HP7
```

```
<OAuthV2 name="O2-VerifyAccessToken">
  <Operation>VerifyAccessToken</Operation>
</OAuthV2>
```

Once you have a token, always use it in the same way. When you make a request to a protected resource, include the token in the authorization header as a bearer token.

The Apigee proxy for the backend can verify the access token using an OAuthV2 policy with the operation VerifyAccessToken.

You generally don't need to include any other information in the policy, including the location to find the token. As long as you pass a bearer token using the Authorization header, which is the default location for OAuth, you don't need to specify the location.

The VerifyAccessToken operation will ensure that the token is valid and not expired. It will then verify that the products associated with the app that created and presented the token are allowed access to the proxy and URL being called. If the token is invalid or expired or does not grant access to the called proxy and URL, the policy will return a 401 Unauthorized.

Agenda

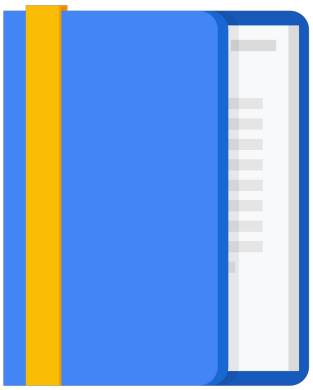
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

Quiz



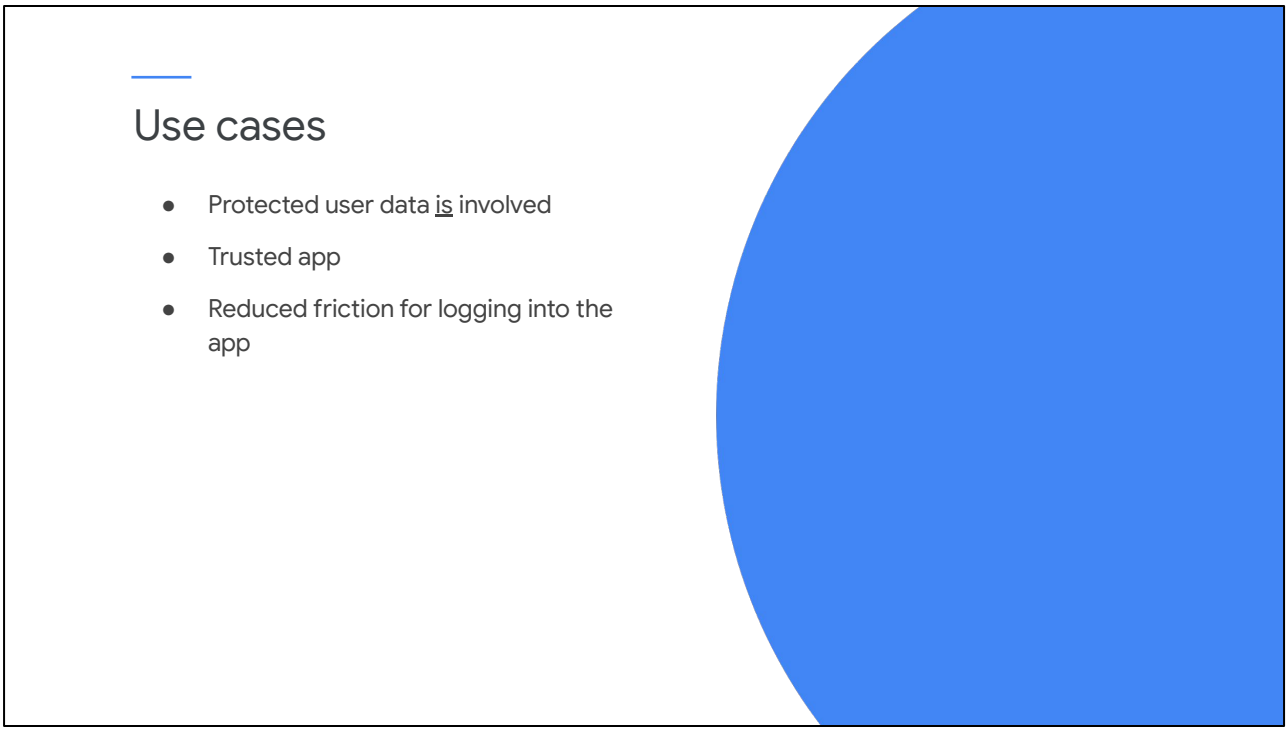
The password grant type is the second OAuth grant type.

OAuth grant types

Grant Type	User Resources?	Use Case
Client Credentials	NO	Business system interactions Resources owned by partner, not a specific user
Resource Owner Password	YES	Requesting app is trusted Resources owned by user
Authorization Code	YES	Requesting app is NOT trusted Resources owned by user
Implicit	*YES*	*Designed for client-side browser-based or mobile apps* ***IMPLICIT IS NO LONGER RECOMMENDED***

We will reiterate the use cases for the resource owner password grant type, review the interactions, and see how policies can be used to implement this grant type.

The resource owner password grant type is usually called the password grant type.

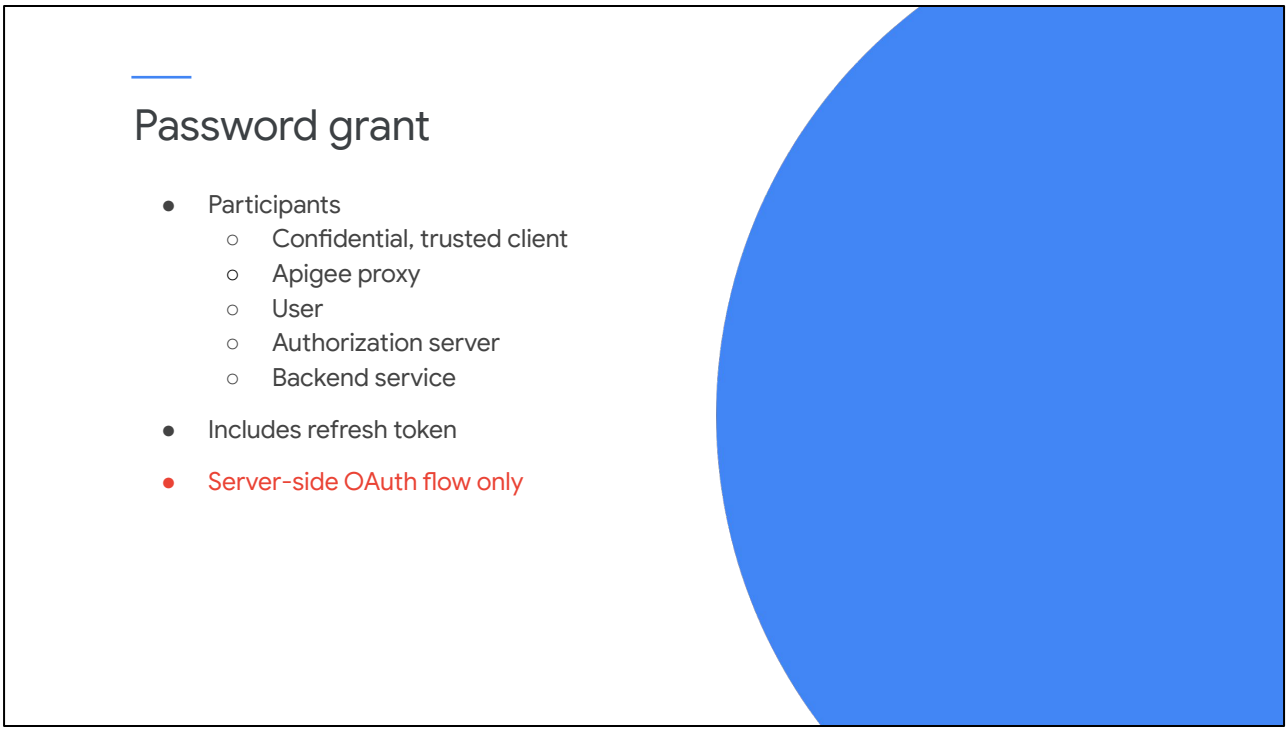


The password grant can be used when protected user data is being accessed.

The password grant is appropriate only for a trusted app provided by a company that already has access to the data. If there is any reason for a user to be uncomfortable about providing credentials to the app, and therefore to the app developer, the password grant type should not be used.

The primary benefit of the password grant is that there is reduced friction when logging into the app.

We will see in the next lecture that the Authorization Code grant involves leaving the app to enter credentials and consent in a web browser. However, the Auth Code grant is significantly more secure, and many companies use the Auth Code grant instead of the Password grant, even for first-party apps.



The first participant for the password grant is the confidential and trusted app that consumes the API. The client must be confidential so it can protect the client secret, and it must be trusted, because the user's password will be passed through the app.

Like the client credentials grant type, the Apigee proxy generates tokens and rejects requests to the backend if the provided token is not valid.

There is now an app user involved who is granting permissions to their protected resources.

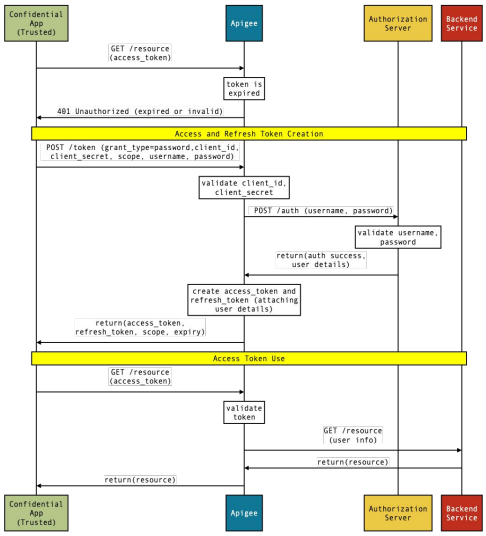
Another new participant is the authorization server, responsible for validating user credentials.

The last participant is again the backend service, which serves the protected resources.

Unlike the client credentials grant, the password grant generates a refresh token along with the access token.

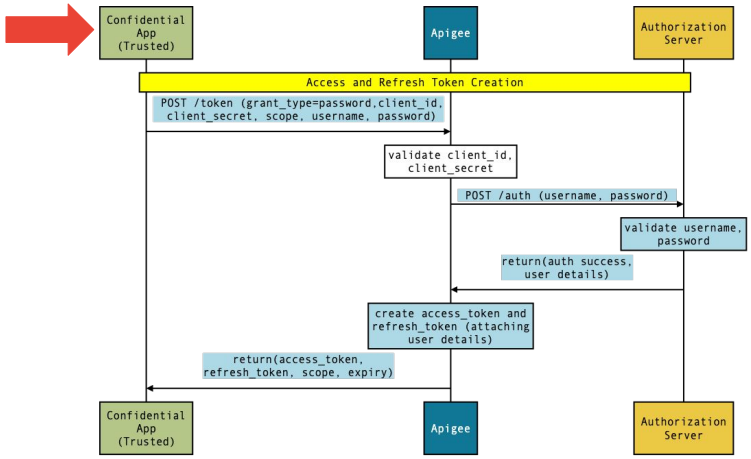
One key item to remember when implementing the password grant is to never run the password grant OAuth flow in client-side code when running on a device. App code and storage on a device are susceptible to hacking, so the client secret could be compromised if stored locally.

Password



This is the overall flow for the password grant type.

Password: Token creation

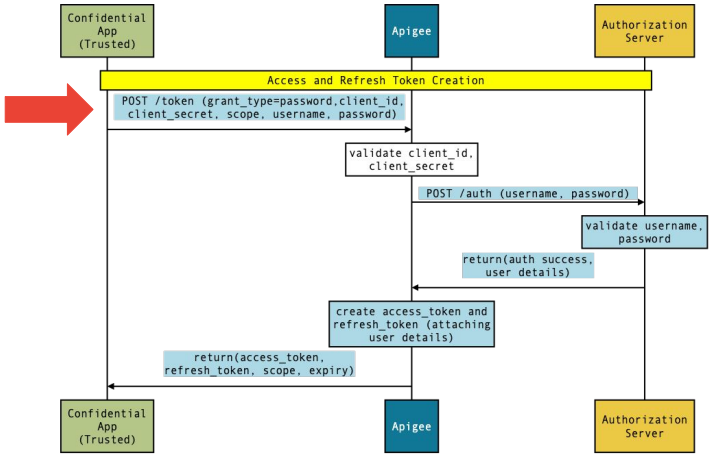


Highlighted in light blue are the parts of the flow that differ from the client credentials grant type flow.

The app must be a trusted and confidential app. The user must trust the app, because the user's username and password will be provided to the app.

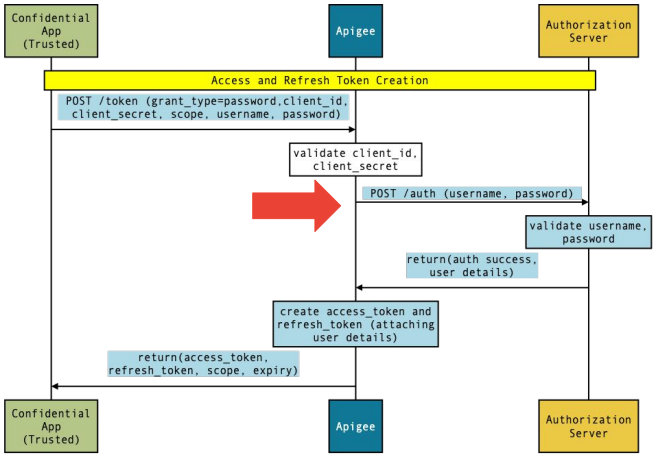
There is now an authorization server involved. The authorization server maintains the usernames and passwords for the users.

Password: Token creation



In the POST request to the OAuth proxy, in addition to the grant type, client ID and secret, and optional scope, we also pass the username and password of the user that is granting access to the protected resources.

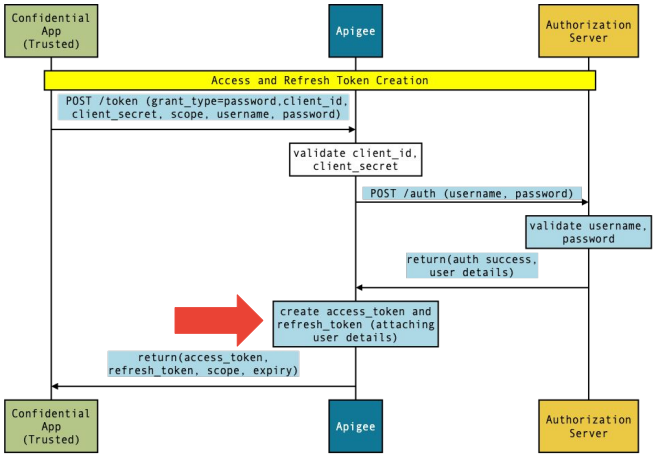
Password: Token creation



After the proxy validates the client ID and secret, the user credentials must be authenticated.

Apigee calls the authorization server's authorization endpoint with the username and password of the user. The authorization server validates the username and password and returns details about the user if the user credentials are valid. If the user credentials are not valid, the authorization server returns an error, and the OAuth proxy returns an error back to the app.

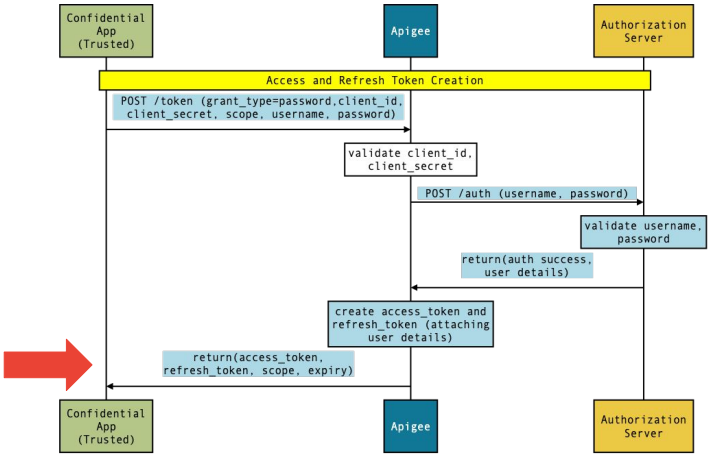
Password: Token creation



If authentication of the user is successful, the OAuth proxy creates an access token and refresh token for the app that provides access to the protected resources. Typically the proxy also attaches custom attributes to the access and refresh tokens.

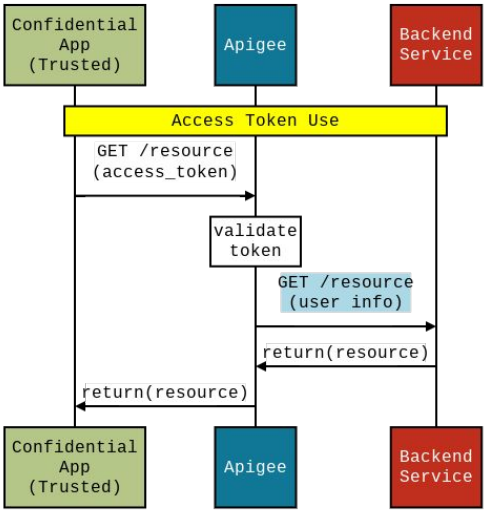
These attributes contain any user details returned from the authorization server that the backend service will need. These custom attributes are available when a token is validated.

Password: Token creation



The returned information is the same as for the client credentials grant type, except that the response includes a refresh token.

Password: Token use



The token is used by the app in the same way as the token is used for the client credentials grant.

Upon validating the token, the Apigee proxy has access to the user details that were attached as custom attributes. The user details can be sent to the backend service if required.

OAuthV2 policy (GenerateAccessToken)

```
<OAuthV2 continueOnError="false" enabled="true"
name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>1800000</ExpiresIn>
  <RefreshTokenExpiresIn>28800000</RefreshTokenExpiresIn>
  <SupportedGrantTypes>
    <GrantType>password</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <UserName>request.formparam.username</UserName>
  <PassWord>request.formparam.password</PassWord>
  <GenerateResponse enabled="true"/>
  <AppEndUser>backend.endUser</AppEndUser>
  <Attributes>
    <Attribute name="backend.data" ref="receivedData"
      display="false">NONE</Attribute>
  </Attributes>
</OAuthV2>
```

The configuration of the GenerateAccessToken OAuthV2 policy for the password grant type is similar to the configuration for the client credentials grant type, with just a few differences.

The RefreshTokenExpiresIn element contains the refresh token expiration in milliseconds.

OAuthV2 policy (GenerateAccessToken)

```
<OAuthV2 continueOnError="false" enabled="true"
name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>1800000</ExpiresIn>
  <RefreshTokenExpiresIn>28800000</RefreshTokenExpiresIn>
  <SupportedGrantTypes>
    <GrantType>password</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <UserName>request.formparam.username</UserName>
  <PassWord>request.formparam.password</PassWord>
  <GenerateResponse enabled="true"/>
  <AppEndUser>backend.endUser</AppEndUser>
  <Attributes>
    <Attribute name="backend.data" ref="receivedData"
      display="false">NONE</Attribute>
  </Attributes>
</OAuthV2>
```

The UserName and PassWord elements tell the policy where to find the user's username and password.

The form parameter values shown are the default locations as defined by the OAuth spec.

OAuthV2 policy (GenerateAccessToken)

- The policy does not validate username and password; it only checks that the values exist.
- The proxy developer is responsible for calling the authorization server to validate user credentials.

```
<OAuthV2 continueOnError="false" enabled="true"
name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>1800000</ExpiresIn>
  <RefreshTokenExpiresIn>28800000</RefreshTokenExpiresIn>
  <SupportedGrantTypes>
    <GrantType>password</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <UserName>request.formparam.username</UserName>
  <PassWord>request.formparam.password</PassWord>
  <GenerateResponse enabled="true"/>
  <AppEndUser>backend.endUser</AppEndUser>
  <Attributes>
    <Attribute name="backend.data" ref="receivedData"
      display="false">NONE</Attribute>
  </Attributes>
</OAuthV2>
```

Note that the policy does not, and cannot, validate the values of the username and password, because the credentials are not stored on Apigee. The policy only verifies that the locations specified have non-empty values.

The proxy developer must make the proxy call the authorization server to validate the values in those locations.

OAuthV2 policy (GenerateAccessToken)

- The policy does not validate username and password; it only checks that the values exist.
- The proxy developer is responsible for calling the authorization server to validate user credentials.

```
<OAuthV2 continueOnError="false" enabled="true"
name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>1800000</ExpiresIn>
  <RefreshTokenExpiresIn>28800000</RefreshTokenExpiresIn>
  <SupportedGrantTypes>
    <GrantType>password</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <UserName>request.formparam.username</UserName>
  <PassWord>request.formparam.password</PassWord>
  <GenerateResponse enabled="true"/>
  <AppEndUser>backend.endUser</AppEndUser>
  <Attributes>
    <Attribute name="backend.data" ref="receivedData"
      display="false">NONE</Attribute>
  </Attributes>
</OAuthV2>
```

The AppEndUser element specifies a variable containing the ID of the end user. This is typically a username or email address.

OAuthV2 policy (GenerateAccessToken)

- The policy does not validate username and password; it only checks that the values exist.
- The proxy developer is responsible for calling the authorization server to validate user credentials.

```
<OAuthV2 continueOnError="false" enabled="true"
name="O2-GenerateToken">
  <Operation>GenerateAccessToken</Operation>
  <ExpiresIn>1800000</ExpiresIn>
  <RefreshTokenExpiresIn>28800000</RefreshTokenExpiresIn>
  <SupportedGrantTypes>
    <GrantType>password</GrantType>
  </SupportedGrantTypes>
  <GrantType>request.formparam.grant_type</GrantType>
  <UserName>request.formparam.username</UserName>
  <PassWord>request.formparam.password</PassWord>
  <GenerateResponse enabled="true"/>
  <AppEndUser>backend.endUser</AppEndUser>
  <Attributes>
    <Attribute name="backend.data" ref="receivedData"
      display="false">NONE</Attribute>
  </Attributes>
</OAuthV2>
```

The example configuration also specifies that an attribute, backend.data, be populated using the received data variable and attached to the access and refresh tokens.

If the received data variable is empty or doesn't exist, the attribute element's value, NONE, is used instead.

This attribute is automatically populated as a variable in the proxy when the token is verified.

API call: Get token

- User credentials are passed in form parameters.

Request

```
POST https://api.example.org/oauth/token

Header:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=password&username=bob&
password=opensesame
```

Response


```
{
    "issued_at": "1565729213000",
    "scope": "READ",
    "application_name": "030fdcea-cf97-12084aea513c",
    "app_enduser": "bob",
    "refresh_token_issued_at": "1565729213000",
    "status": "approved",
    "refresh_token_status": "approved",
    "api_product_list": "[gold]",
    "expires_in": ”119",
    "developer.email": "bob@example.com",
    "organization_id": "0",
    "token_type": "Bearer",
    "refresh_token": "IFl7jlijYuexu6XVSSjLMJq8SVXGOAAq",
    "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
    "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
    "organization_name": "apigee",
    "refresh_token_expires_in": "28799",
    "refresh_count": "0"
}
```


To request a token using the password grant flow, the API request is identical to the client credentials request except that the username and password are passed as form parameters.

For the response, the access token and expires in fields are the same as with the client credentials grant type.

API call: Get token

- User credentials are passed in form parameters.

Request

```
POST https://api.example.org/oauth/token

Header:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=password&username=bob&
password=opensesame
```

Response

```
{
    "issued_at": "1565729213000",
    "scope": "READ",
    "application_name": "030fdcea-cf97-12084aea513c",
    "app_enduser": "bob",
    "refresh_token_issued_at": "1565729213000",
    "status": "approved",
    "refresh_token_status": "approved",
    "api_product_list": "[gold]",
    "expires_in": ”119",
    "developer.email": "bob@example.com",
    "organization_id": "0",
    "token_type": "Bearer",
    "refresh_token": "IFl7jlijYuexu6XVSSjLMJq8SVXGOAAq",
    "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
    "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
    "organization_name": "apigee",
    "refresh_token_expires_in": "28799",
    "refresh_count": "0"
}
```

The password grant type response has additional fields for the refresh token and its expiration. If the AppEndUser is specified, it will also be returned in the response.

API call: Use token

- Token is passed as Bearer token in Authorization header.
-  OAuthV2 policy (Verify Access Token) will validate the access token for all grant types.

```
Request:
GET https://api.example.org/orders/v1/items/14

Header:
Authorization: Bearer 4WCAchNNtVyK8JsACl1HP7
```

```
<OAuthV2 name="O2-VerifyAccessToken">
  <Operation>VerifyAccessToken</Operation>
</OAuthV2>
```

The call to the protected resource is identical for all grant types: the token is sent using a bearer token in the Authorization header.

Agenda

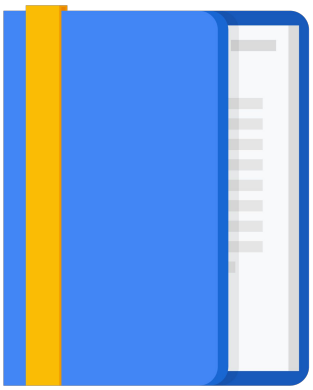
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

Quiz



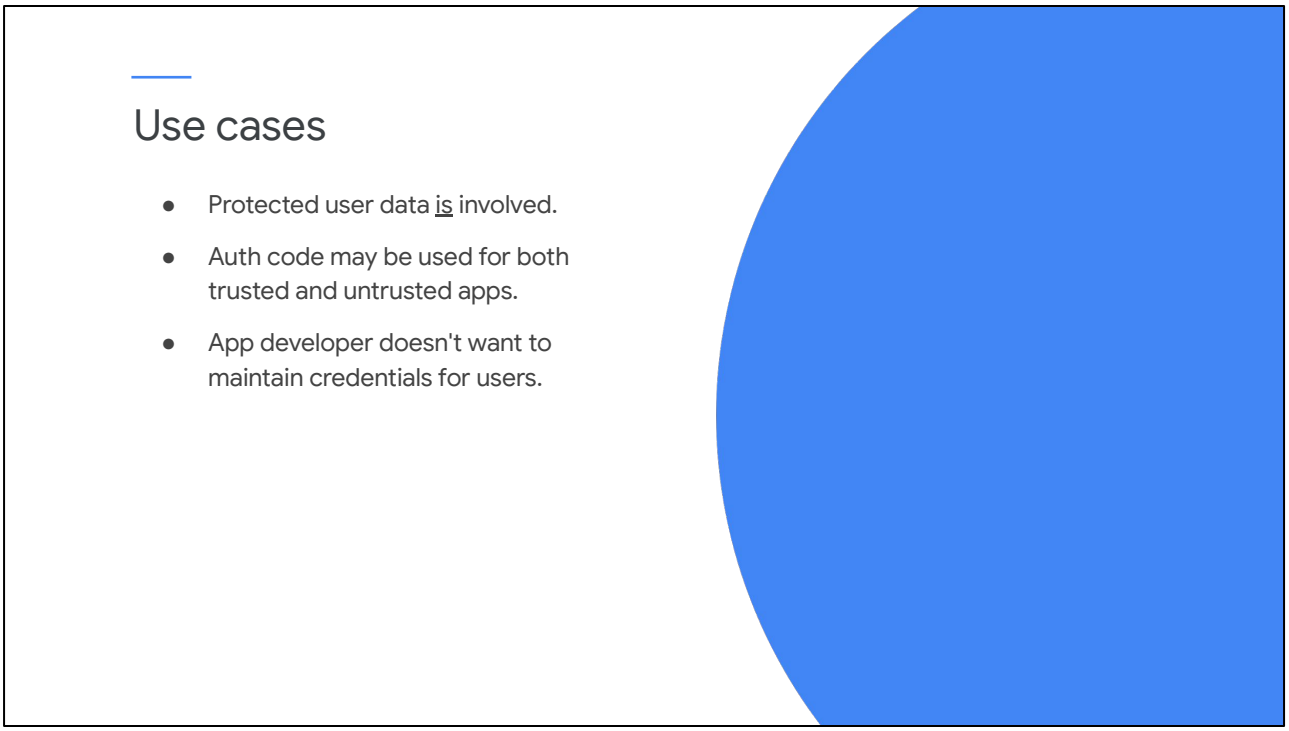
The authorization code grant type, also called auth code grant type, is the third OAuth grant type.

OAuth grant types

Grant Type	User Resources?	Use Case
Client Credentials	NO	Business system interactions Resources owned by partner, not a specific user
Resource Owner Password	YES	Requesting app is trusted Resources owned by user
Authorization Code	YES	Requesting app is NOT trusted Resources owned by user
Implicit	*YES*	*Designed for untrusted browser-based or mobile apps* ***IMPLICIT IS NO LONGER RECOMMENDED***

We will reiterate the use cases for the auth code grant type, review the interactions, and see how policies can be used to implement this grant type.

We will also see how to use the auth code grant in place of the implicit grant.

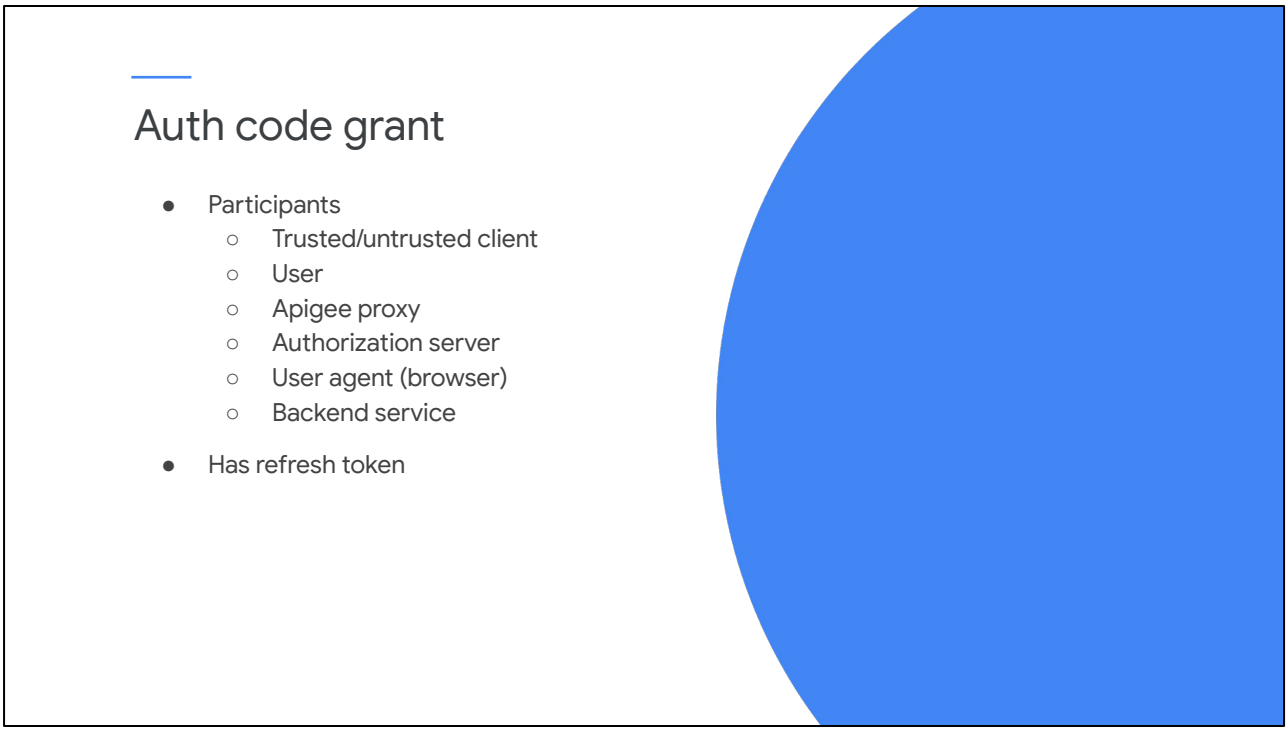


Like the password grant type, the auth code grant can be used when protected user data is being accessed.

But unlike the password grant type, the auth code grant can be used for both trusted and untrusted apps.

The auth code grant is designed to allow a user to authenticate without exposing their credentials to the app. In addition, the user is given full visibility and control over the app access using scopes.

Finally, the auth code grant is the best choice when the app developer doesn't want to maintain passwords for its users. An app can use the auth code grant to use accounts with common providers like Google, Twitter, and Facebook to identify the app's users. This can also be a benefit for users, because they don't need to maintain separate accounts and passwords for all of the apps and services they use.



The first participant for the auth code grant is the client that consumes the API. The client may be trusted or untrusted. In addition, the client can be confidential or public.

For public apps, you'll learn how to add an OAuth extension, called Proof Key for Code Exchange (PKCE), or "pixie," to the auth code grant for public clients that cannot protect a client secret.

There is a user involved who is granting permissions to their protected resources.

The Apigee OAuth proxy generates tokens, and the API proxies to the backend services will reject requests to the backend when the provided token is not valid.

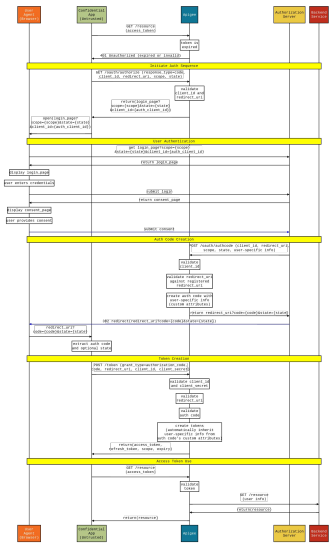
The authorization server is responsible for validating user credentials and maintaining the list of scopes that can be used for protected resources.

A new participant is a user agent, or browser, that is used to communicate with the authorization server.

The last participant is again the backend service, which serves the protected resources.

And like the password grant, the auth code grant generates a refresh token along with the access token to avoid having to reenter or save the user's password too often.

Authorization code (confidential app)

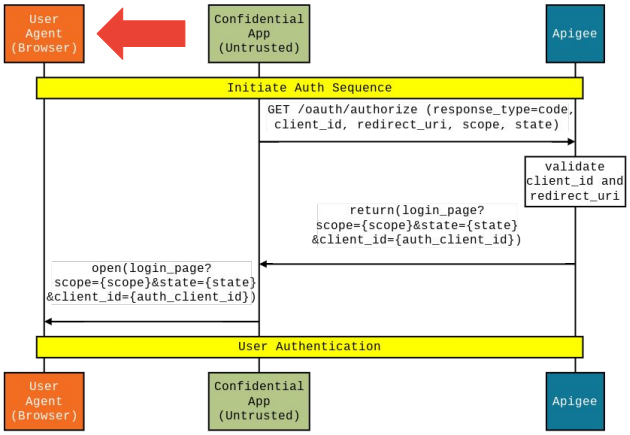


Here is the flow for the authorization code grant type for confidential apps. I don't expect you to be able to read the diagram on this page -- we will go through the details in the next slides.

You can see that this flow is significantly more complex than the password grant type flow. With that added complexity, we get better security.

Let's get started.

Auth code (confidential app): Initiate auth

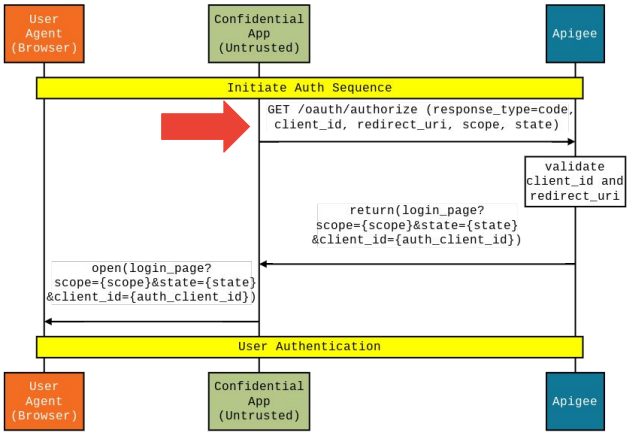


Let's first look at the new participant in the interaction.

A user agent, typically a browser, allows the user to interact directly with the authorization server and not through the application. This prevents the user's credentials from being exposed to the app.

You have probably encountered the OAuth authorization code flow before. When you try to log in to an app or website with your Google credentials, and you are directed away from your app to a direct browser connection with Google, you are using the auth code flow. This flow allows the user to log in and confirm the level of access requested directly with Google, and not through the app.

Auth code (confidential app): Initiate auth



The authorization process is initiated by the application, with a call to an authorization endpoint in the Apigee OAuth proxy.

The response type of code indicates that we want to get an authorization code, typically shortened to auth code, from the authorization server. The auth code will eventually be returned to the app, and the app will exchange the auth code for an access token and refresh token. We will see this entire process shortly.

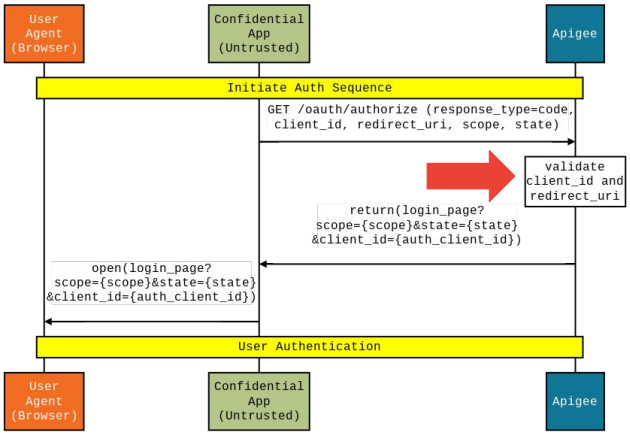
In this initial request, the app sends its client ID, but not its client secret; the client secret will be sent later when the app exchanges the auth code for a token.

The app also sends the requested scope and a redirection URI.

The redirection URI will be used by the authorization server to redirect the auth code into the app after the user has authenticated and consented to the scope. More about the redirection URI later.

The app may also send an optional state parameter that will be returned when returning from the user authentication process, allowing the app to resume processing the request. The state is exposed to the OAuth proxy and the authorization server, so it is best not to send sensitive information in the state.

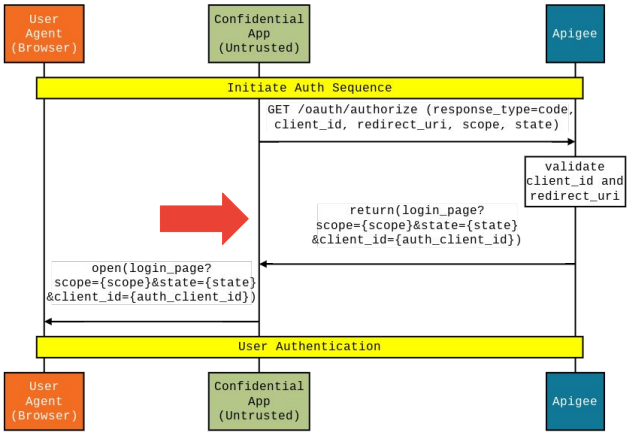
Auth code (confidential app): Initiate auth



The proxy validates the client ID and verifies that the redirect URI passed by the app matches the redirect URI associated with the Apigee app.

An app's redirect URI is configured by the app developer when registering the app on Apigee.

Auth code (confidential app): Initiate auth

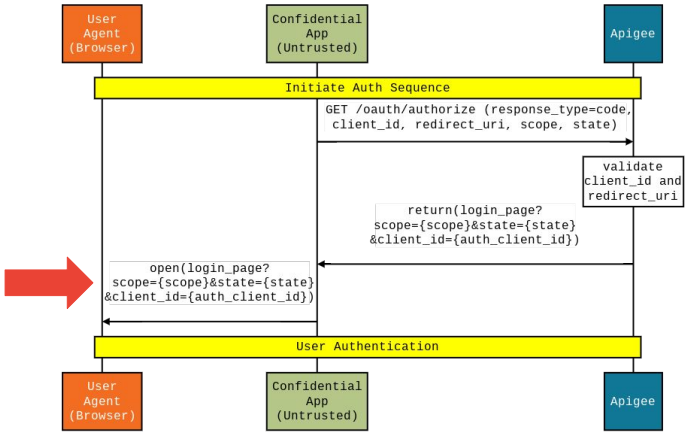


Assuming that the client ID and redirect URI are valid, the proxy redirects the app to the authorization server's login page, with the scope, state, and client_id as query parameters.

Note that the returned client_id value is shown as the auth_client_id. This client ID will be used to identify the app for the login page.

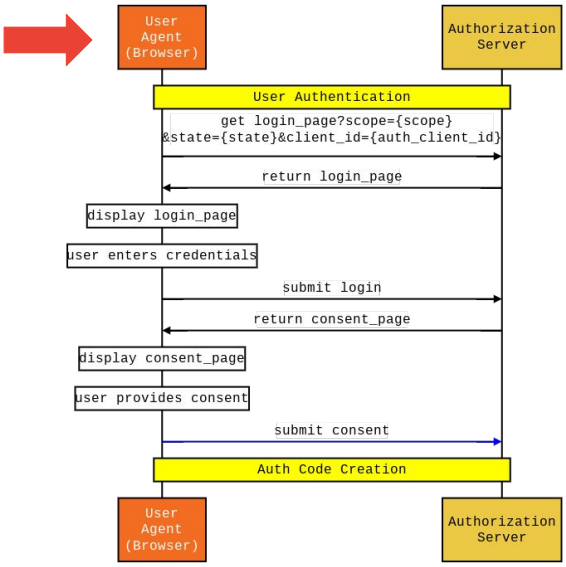
The app is registered with both Apigee and the authorization server. That authorization server registration may use a separate client ID for the app. If so, the OAuth proxy needs to map the Apigee client ID to the separate client ID used by the authorization server and include the authorization server client ID in the login page URL.

Auth code (confidential app): Initiate auth



The app redirects to the login page within the browser, outside of the app.

Auth code (confidential app): User authentication

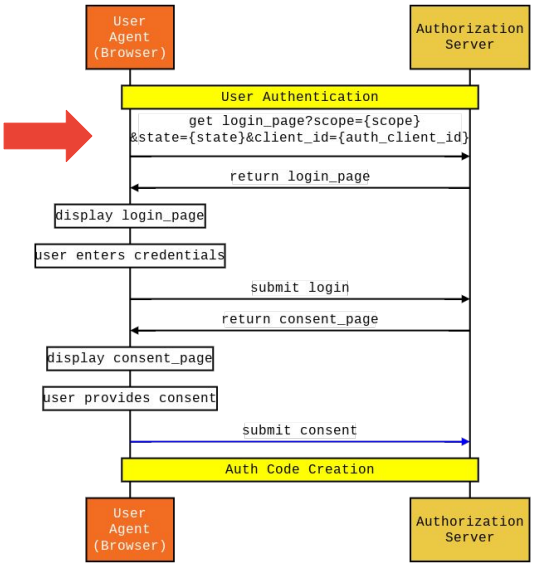


During this next section of the flow, the user interacts with the browser and not the app. You'll notice that neither Apigee nor the app is part of the interaction.

This direct interaction between the user and authorization server ensures that the user can safely authenticate and provide consent for the level of access being requested by the app. This interaction provides a level of security that the password grant type clearly does not.

Let's see how this interaction works.

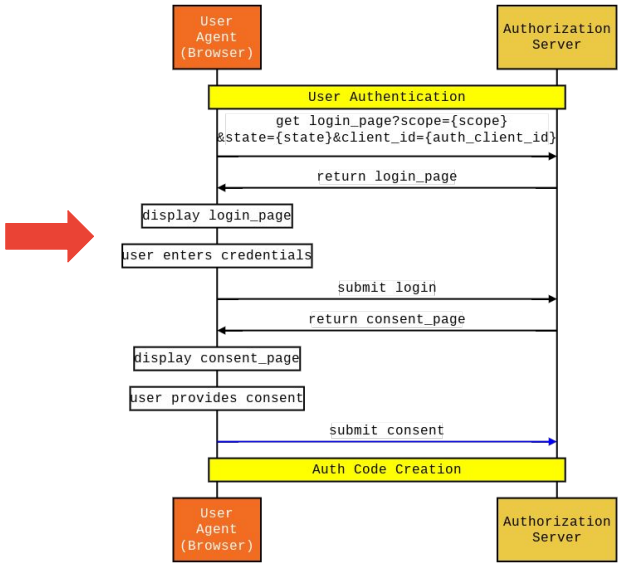
Auth code (confidential app): User authentication



The browser opens the login page URL, which includes the scope, state, and client_id query parameters, and is typically a web page on the authorization server.

The login page is returned to the browser...

Auth code (confidential app): User authentication

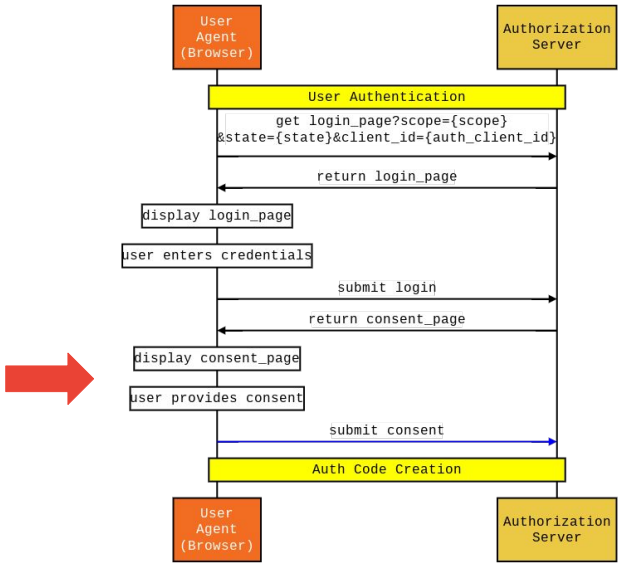


...which displays the page.

The user enters their credentials to authenticate, and the login page is submitted.

After the authorization server confirms the user's identity, the consent page is returned...

Auth code (confidential app): User authentication



...and displayed in the browser.

The consent page specifies the application requesting access and provides a human-readable description of the scope or scopes that have been requested.

You have probably noticed this type of page before. For example, if you use your Google account to log in to a third-party app, the consent page might say that the app is requesting access to your email address and name. This is a reasonable level of access for an app that is only using Google for a login account.

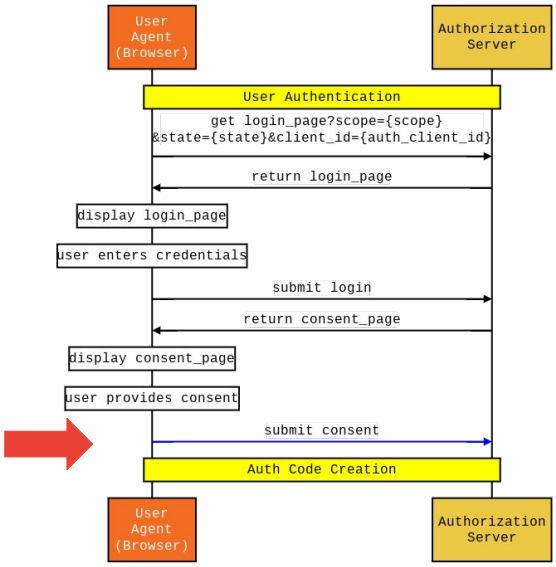
Another app might legitimately need the ability to read or write documents in your Google Drive, and that access might be requested.

A malicious app might request full access to your account, which you don't want to provide.

The consent page is where Google, or whichever authorization service is being used, can confirm that the user is informed of the requested level of access for the app, and capture consent. Because the app is not involved in this process at all, the app cannot request access without the user knowing, and the app cannot get access without the consent of the user.

This is a key strength of the auth code grant type.

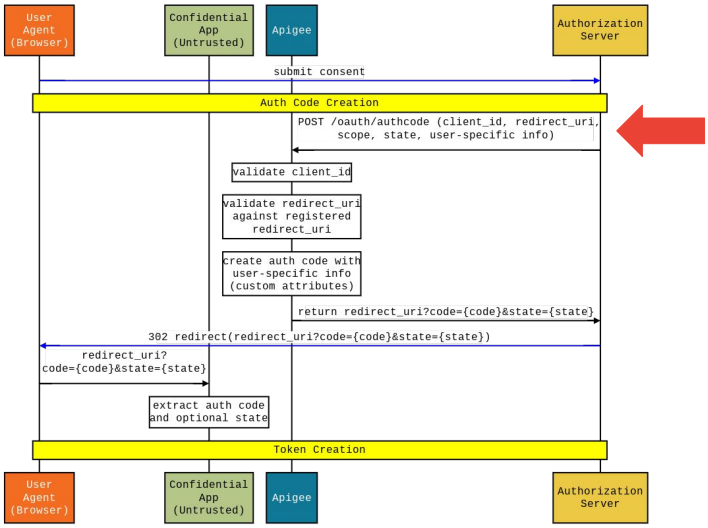
Auth code (confidential app): User authentication



When the consent page is submitted to the authorization server, the authorization server begins the process of creating the auth code.

The response to the browser will only happen after the auth code has been created.

Auth code (confidential app): Code creation



The process for creating the auth code is not specified by the OAuth specification. A solution that uses a call from the authorization server directly to an OAuth proxy is shown here.

The authorization server POSTs a request to an OAuth proxy to create a new auth code.

The client ID required by the proxy is the Apigee client ID for the app. If the authorization server does not know the Apigee client ID, the proxy must map the authorization server client ID to the Apigee id.

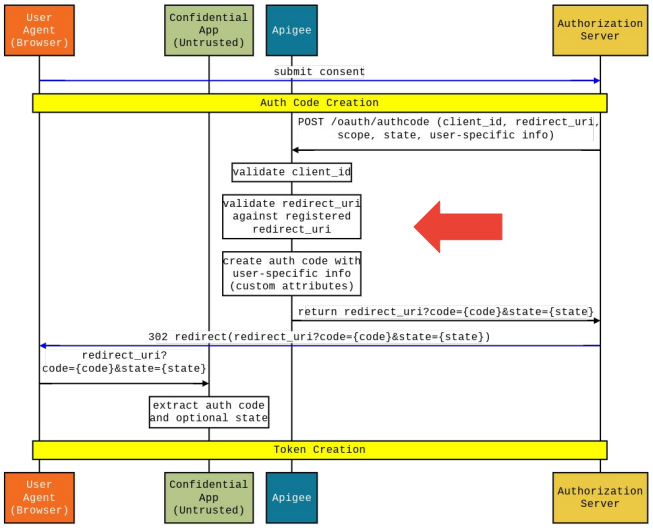
The redirect URI sent by the authorization server must match the redirect URI configured for the app on Apigee.

The scope parameter contains the scope to which the user consented.

The state is the original state passed with the first request.

The app configuration on the authorization server might be configured to send extra information about the user, like the name or email address. This information could then be attached to the auth code and then the access and refresh tokens, so that the user data would be available for calls using the access token. This requires a custom implementation, but we'll show it here.

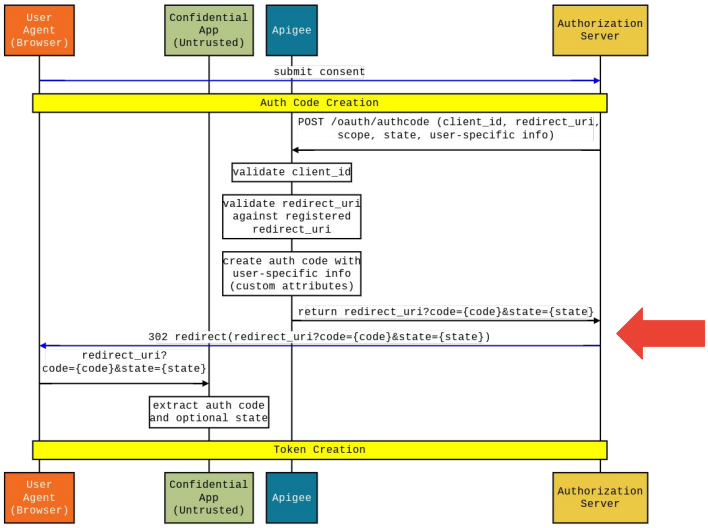
Auth code (confidential app): Code creation



The OAuth proxy validates the client ID and matches the provided redirect URI with the redirect URI registered for the app on Apigee. If these validate successfully, the OAuth proxy creates an auth code. This auth code is associated with the Apigee app client ID and the user-consented scope. The user-specific information can also be attached to the auth code as custom attributes.

Just like the access and refresh tokens, an auth code is an opaque string with no encoded information. All of the configured information for the auth code is stored on Apigee.

Auth code (confidential app): Code creation

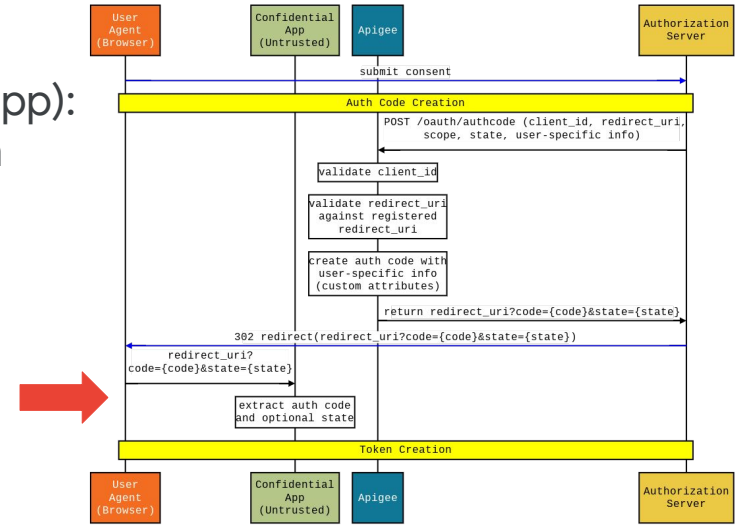


The OAuth proxy returns the redirect URI with the auth code and state as query parameters. This URI is used by the authorization server to redirect the user's browser back into the app.

This redirect_uri generally uses a specially formatted URI with a custom scheme. For example, instead of "https://", the URI for the foo app might start with "foo://".

This app's scheme name could be registered for Android and iOS, and opening a link with this scheme on the device would open the foo app instead of the browser.

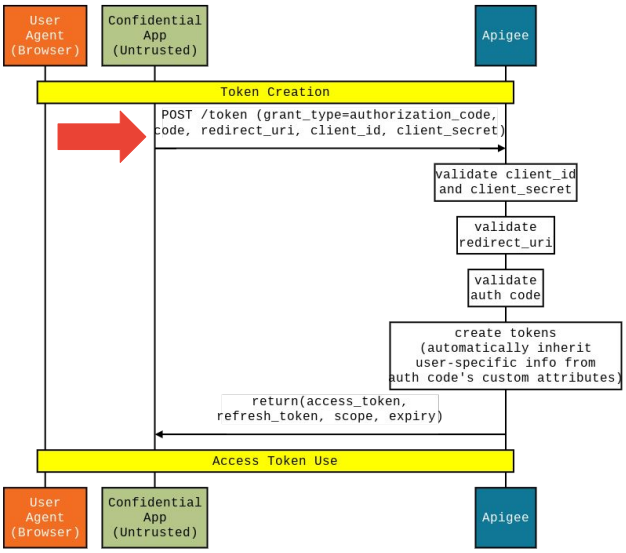
Auth code (confidential app): Code creation



The browser then redirects back into the app, and the app can now extract the auth code and the original state from the URL.

Next, the app will exchange the auth code for an access token and refresh token.

Auth code (confidential app): Token creation



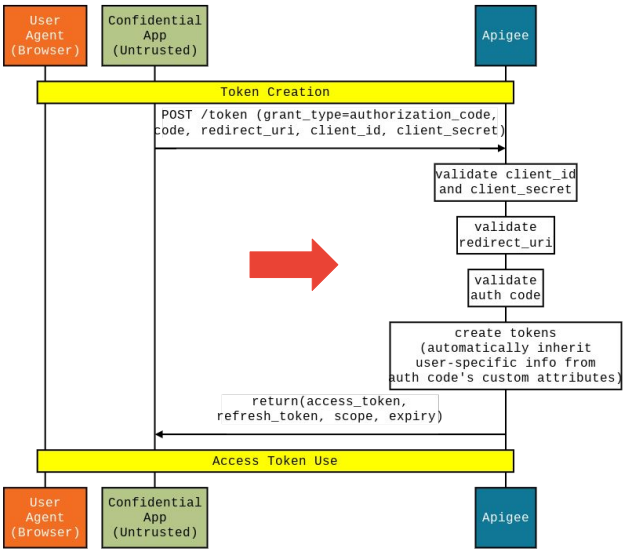
The app POSTs a request to the OAuth proxy to exchange an auth code for an access and refresh token.

The grant type is specified as authorization code.

The authorization code and redirect URI are passed in.

The app has not yet been authenticated, so both the client ID and client secret will be sent via the Basic Authentication header.

Auth code (confidential app): Token creation

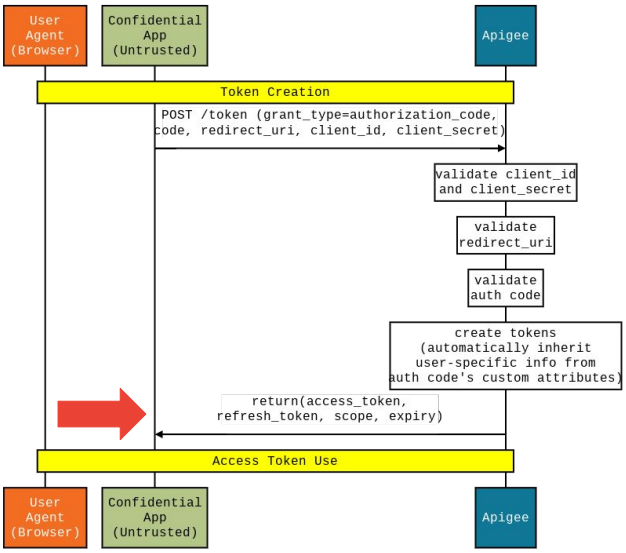


The client ID, secret, and redirect URI are all validated.

The auth code is checked to make sure it is valid and that it is associated with the app.

The auth code is then used to create the access and refresh tokens. The tokens are provided with the user-consented scope that was specified for the auth code. The tokens automatically inherit any user-specific info that was attached to the auth code as custom attributes.

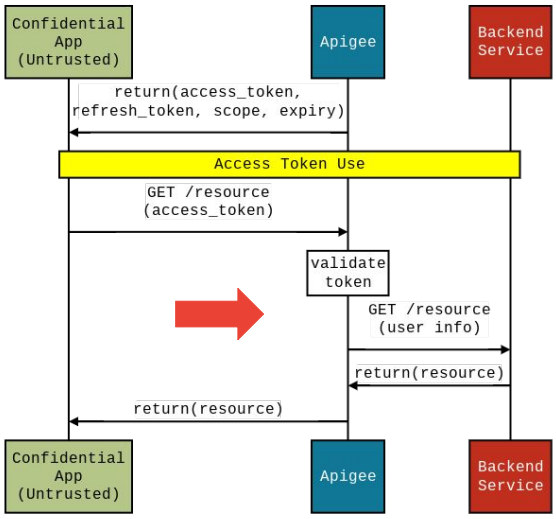
Auth code (confidential app): Token creation



Finally, the access token and refresh token are returned to the app.

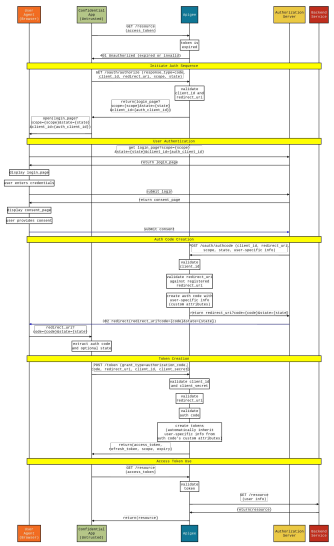
At this point, the app uses the access and refresh tokens as they are used for the other grant types.

Auth code (confidential app): Token use



Token use is the same as for the password grant.

Authorization code (confidential app)



If you were able to understand all of that during the first pass, congratulations.

Most people do not understand the auth code flow without going through the flow a few times, so go ahead and review the flow again if necessary.

Try to understand the information being passed between the participants, and why the information is being passed the way it is.

Another recommendation is to read RFC 6749, which is the OAuth 2.0 Authorization Framework specification.

> “Clients SHOULD NOT use the implicit grant...Clients SHOULD instead use the...authorization code grant type.”
>
> Internet Engineering Task Force (IETF),
> OAuth 2.0 Security Best Current Practice

Now let's look at the auth code solution for public clients.

When the OAuth spec was originally released, the implicit grant type was the recommended solution, but there are security issues with the implicit grant type.

The current recommendation is to use the authorization code grant type with PKCE instead.

Use auth code for public clients

- Public clients cannot protect secrets.
- Auth code over https is safe, but mobile redirect with auth code may be compromised.
- How do we keep a compromised auth code from being exchanged for an access token?
- Proof Key for Code Exchange (PKCE):
  - Specified in RFC 7636
  - Uses cryptography to guarantee that the client exchanging the auth code for tokens also initiated the auth request

The problem with public clients is that they cannot safely store secrets. Mobile apps and client side JavaScript apps are examples of public clients.

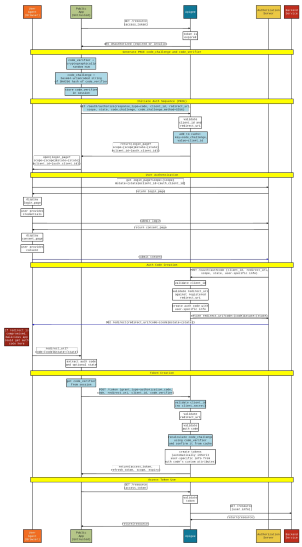
The TLS communication used for OAuth is secure over the network, but the redirect on the mobile device may not be secure.

The redirect URL contains the auth code. If the auth code is compromised, how do we keep the bad actor from exchanging the auth code for an access token if we can't require a client secret?

The answer is that we use the OAuth extension called Proof Key for Code Exchange, known as PKCE. PKCE is specified in RFC 7636.

PKCE uses cryptography to guarantee that the client exchanging an auth code for tokens is the same client that started the original auth request.

Auth code with PKCE (public app)

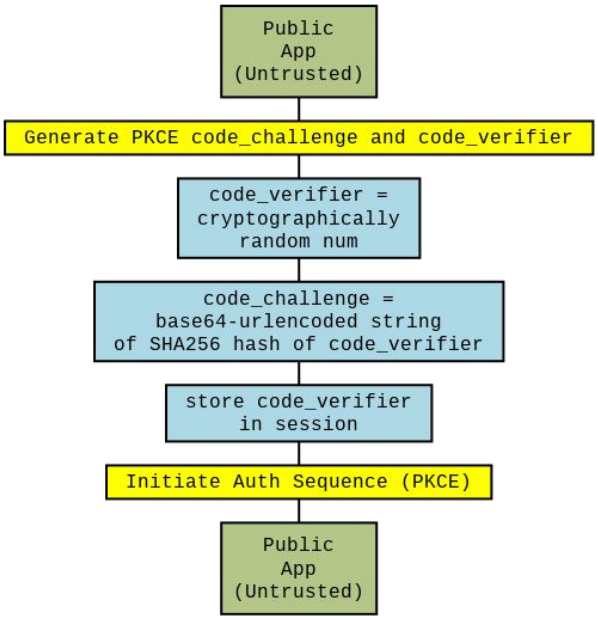


This is the flow for auth code with PKCE. Most of the flow is the same as for auth code without PKCE.

Let's look at the changes to the auth code flow when using PKCE. The changes are shown in light blue.

Auth code with PKCE (public app): Generate PKCE “password”

- SHA256 one-way hash is easy to calculate, but effectively impossible to reverse.
- Same way passwords are protected (stored as hashes).



Before each request for a token, the app generates a PKCE code challenge and code verifier.

The code verifier is a cryptographically random number. Cryptographically random numbers are unguessable.

The code challenge is a base64 URL encoded string of the SHA256 hash of the code verifier.

A hash is a one-way operation: it is easy to calculate a hash on a value, but effectively impossible to determine the original value from its hash. It is also effectively impossible to build another value that has the same hash value.

Passwords are stored in plaintext files using hashes. The hash is stored in the password file. Then when a password is entered, the system hashes the entry and compares the calculated hash against the hash in the password file. If they match, it means that the password is correct. It also means that compromising the password hash file does not allow retrieval of the passwords. PKCE takes advantage of this property of hashes.

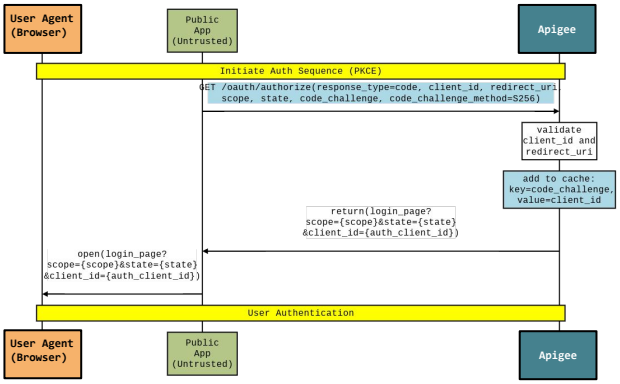
The app sends the code challenge, the hash, in the first communication.

Later, when exchanging the auth code for tokens, the app sends the code verifier to prove that it is the same client that initiated the original communication. The proxy can

calculate the hash for the code verifier and confirm that it matches the code challenge that was originally sent.

Even if a bad actor is able to hijack the code challenge, they won't be able to provide the code verifier.

Auth code with PKCE (public app): Initiate auth



Two additional parameters are sent when initiating the auth code flow.

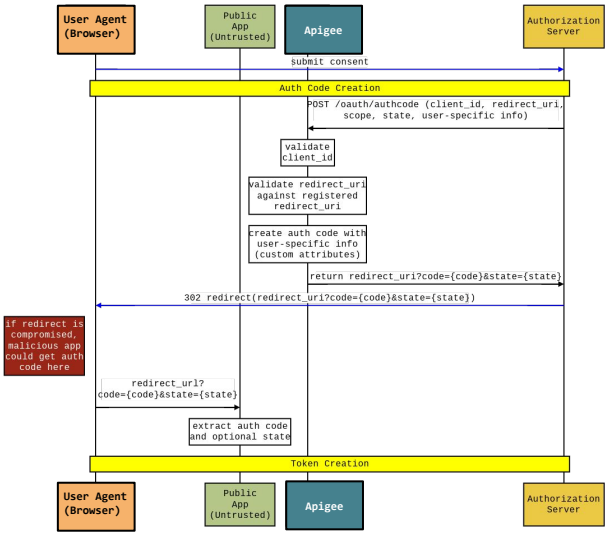
Code challenge is the hashed value. The code challenge method parameter specifies the type of code challenge being sent, which is SHA256.

Apigee should cache the code challenge to be used later when the auth code is exchanged for tokens. The key for the cache entry is the code challenge, and the value is the client ID.

The user authentication section is unchanged for PKCE.

Auth code with PKCE (public app): Code creation

- Redirect URI from Authorization Server, including code in query parameter, may be vulnerable on compromised phone.
- Compromised auth code cannot be exchanged for a token without code_verifier.

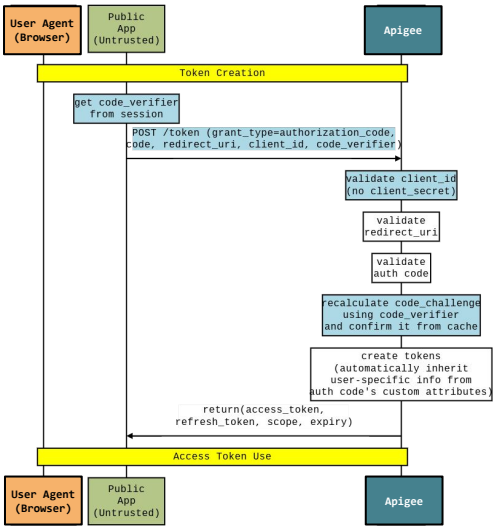


The redirect URI from the authorization server, including the auth code in the query parameter, may be vulnerable on a compromised device.

A bad actor may be able to get the auth code, but, with PKCE, the auth code can't be exchanged for tokens without the code verifier.

Auth code with PKCE (public app): Token creation

- Recalculation of code challenge from code_verifier is custom code in the OAuth proxy.
- Once code_verifier is verified, the rest of the policies and flows are exactly the same as auth code without PKCE.



The app has the original code verifier, which has never been sent across the network.

Instead of the client secret, which the public app cannot secure, the code verifier is sent in the token request.

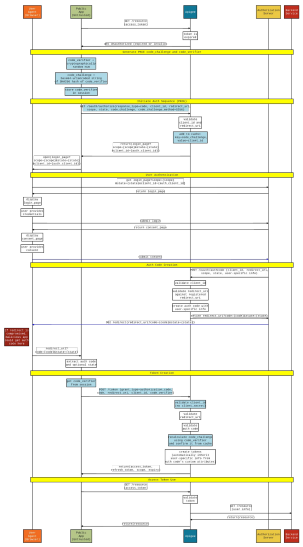
The OAuth proxy can now validate the client ID, redirect URL, and auth code as before.

The code challenge hash is calculated for the provided code verifier. That calculated hash can be looked up in the API proxy cache, and as long as the entry is in the cache, and the client ID value from the cache matches the client ID in the request, the OAuth proxy can be confident that this request came from the original requestor and not a bad actor.

If the entry is not in the cache, or the client ID does not match, the OAuth proxy should reject the request.

The access tokens and refresh tokens are created as before and returned to the caller. Usage of the tokens is unchanged.

Auth code with PKCE (public app)



So now you understand how PKCE can be used for public apps, and how it affects the auth code flow.

Effectively, we are just creating a new password, or client ID, for every auth code request.

Remember, for both the password and auth code grant types, after you have an access token and refresh token, you can easily keep retrieving new access tokens until the refresh token expires.

OAuthV2 policy (GenerateAuthorizationCode)

- Custom attributes attached to an auth code will automatically be transferred to access and refresh tokens created using the auth code.

```
<OAuthV2 continueOnError="false" enabled="true" name="O2-GenerateAuthCode">
  <Operation>GenerateAuthorizationCode</Operation>
  <ExpiresIn>120000</ExpiresIn>
  <ResponseType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
  <ClientId>request.formparam.client_id</ClientId>
  <RedirectUri>request.formparam.redirect_uri</RedirectUri>
  <Scope>request.formparam.scope</Scope>
  <AppEndUser>backend.endUser</AppEndUser>
  <Attributes>
    <Attribute name="backend.token" ref="request.formparam.token" display="false">NONE</Attribute>
  </Attributes>
</OAuthV2>
```

Let's see how to implement the policies for the auth code flow on Apigee.

The first step uses the client ID, redirect URI, and scope with the OAuthV2 policy GenerateAuthorizationCode operation.

You can also attach the app end user and custom attributes to an auth code, and they will automatically be attached to the access and refresh token created from the auth code.

This policy is the same whether or not PKCE is being used.

The GenerateAccessToken for the auth code grant type takes an auth code instead of the username and password used for the password grant type.

Note that the GenerateAccessToken operation expects the request to have a basic auth header with the client ID and secret. If you are using PKCE, the app won't send the basic auth header with the request. The OAuth proxy must create the basic auth header before the policy will generate the access token.

The VerifyAPIKey policy is used to verify the client ID provided by the app. The policy creates a variable containing the client secret. Then the OAuth proxy can build the basic auth header using the client ID and client secret and the BasicAuthentication policy.

We haven't looked at the refresh flow yet, but you'd also use the same process when using PKCE, since the client secret is not passed when refreshing a token.

API call: Get token

Request without PKCE

```
POST https://api.example.org/oauth/token

Header:
Authorization: Basic YmFkOnBhc3N3b3Jk...
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=authorization_code&
scope=READ&code=I4IdkE3&
redirect_uri=https://api.example.org/resume
```

Response (SAME AS PASSWORD GRANT TYPE)


```
{
    "issued_at": "1565729213000",
    "scope": "READ",
    "application_name": "030fdcea-cf97-12084aea513c",
    "app_enduser": "bob",
    "refresh_token_issued_at": "1565729213000",
    "status": "approved",
    "refresh_token_status": "approved",
    "api_product_list": "[gold]",
    "expires_in": ”119",
    "developer.email": "bob@example.com",
    "organization_id": "0",
    "token_type": "Bearer",
    "refresh_token": "IFl7jlijYuexu6XVSSjLMJq8SVXGOAAq",
    "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
    "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
    "organization_name": "apigee",
    "refresh_token_expires_in": "28799",
    "refresh_count": "0"
}
```


When getting a token using the auth code grant type without PKCE, the auth code form parameter replaces the username and password parameters.

The default response is identical to the password grant type response.

API call: Get token (PKCE)

Request with PKCE

```
POST https://api.example.org/oauth/token

Header:
(no auth header)
Content-Type:
  application/x-www-form-urlencoded

Payload (form parameters):
grant_type=authorization_code&
client_id=RqBca4HG...&
scope=READ&code=I4IdkE3&
redirect_uri=apigee://authcode&
code_verifier=CxUCHEkm...
```

Response (SAME AS PASSWORD GRANT TYPE)


```
{
    "issued_at": "1565729213000",
    "scope": "READ",
    "application_name": "030fdcea-cf97-12084aea513c",
    "app_enduser": "bob",
    "refresh_token_issued_at": "1565729213000",
    "status": "approved",
    "refresh_token_status": "approved",
    "api_product_list": "[gold]",
    "expires_in": ”119",
    "developer.email": "bob@example.com",
    "organization_id": "0",
    "token_type": "Bearer",
    "refresh_token": "IFl7jlijYuexu6XVSSjLMJq8SVXGOAAq",
    "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
    "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
    "organization_name": "apigee",
    "refresh_token_expires_in": "28799",
    "refresh_count": "0"
}
```


For PKCE, we can't pass the client secret, so there is no basic auth header.

The client ID and code verifier are passed using form parameters.

API call: Use token

- Token is passed as Bearer token in Authorization header.
-  OAuthV2 policy (Verify Access Token) will validate the access token for all grant types.

```
Request:
GET https://api.example.org/orders/v1/items/14

Header:
Authorization: Bearer 4WCAchNNtVyK8JsACl1HP7
```

```
<OAuthV2 name="O2-VerifyAccessToken">
  <Operation>VerifyAccessToken</Operation>
</OAuthV2>
```

Once you have an access token, it is used the same way regardless of grant type.

Agenda

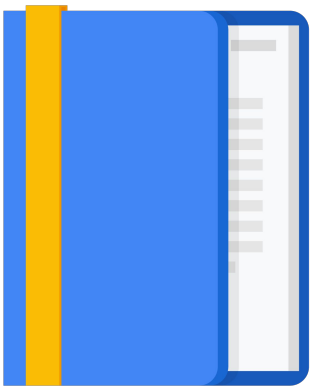
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

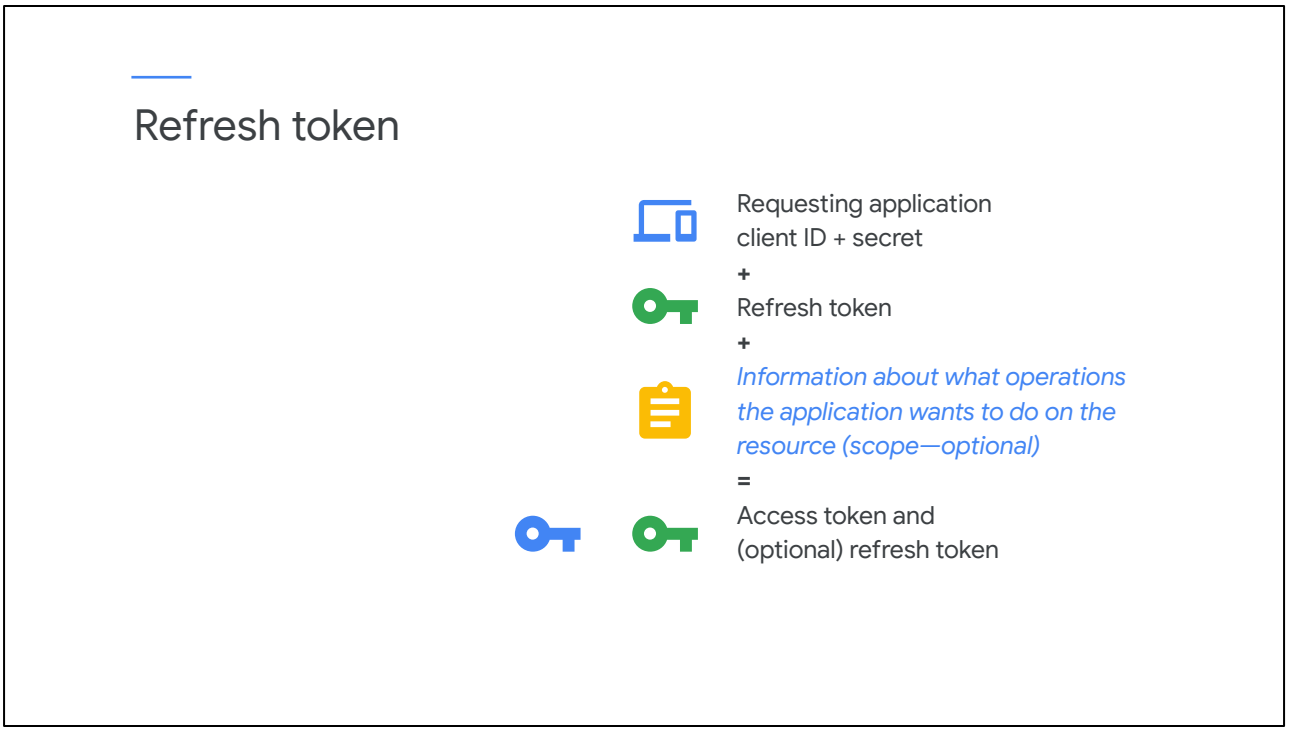
Quiz



We have covered OAuth 2.0 access and refresh tokens and the basic grant type use cases and flows.

In this OAuth wrap-up section, we'll learn about the refresh grant flow and a few more Apigee policies that will help with your OAuth proxies.

We'll end this section with an OAuth lab.



Now that we've discussed the basic OAuth grant type use cases, let's revisit the refresh token.

When a refresh token is used, there is no user interaction.

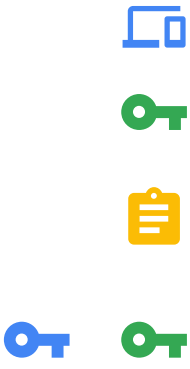
The app exchanges the refresh token for a new token and optionally a new refresh token.

By default, new access and refresh tokens generated using a refresh token keep the scope associated with the refresh token.

The app can choose to get reduced scope for the access token, but this is unusual.

Refresh token

- New refresh token retains same scope as original refresh token.



As discussed before, a new refresh token can be created when using a refresh token, and we will see why this is a best practice.

A new refresh token will always keep the same scope as the previous refresh token.

No new scopes can be added for the new refresh token, because the user does not have the opportunity to provide consent.

Refresh token

- New refresh token retains same scope as original refresh token.
- Client secret not used when using Authorization Code with PKCE.


Requesting application client ID + secret

+


Refresh token

+


Information about what operations the application wants to do on the resource (scope—optional)

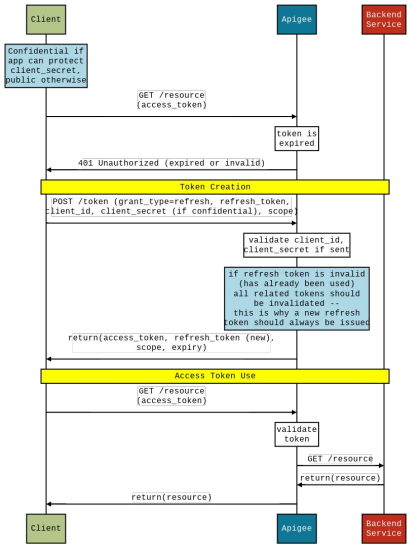
=



Access token and (optional) refresh token

When an app uses the auth code flow with PKCE, the client secret is not sent in the refresh token request.

Refresh

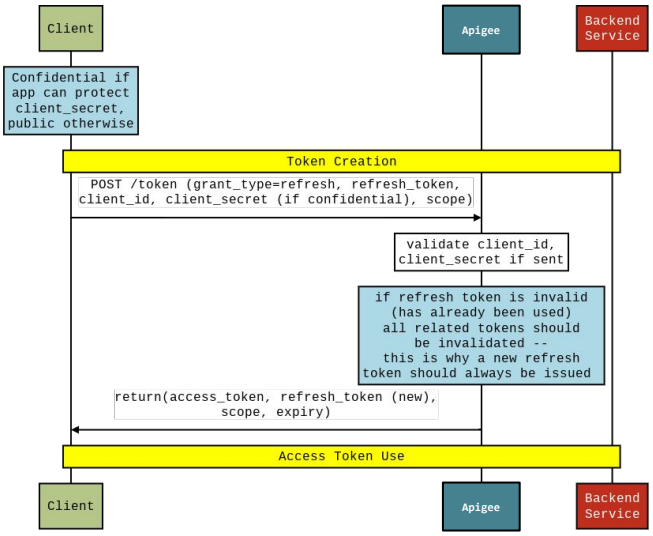


This is the refresh token flow.

Only a single request is required to refresh tokens.

Refresh: Token creation

- If app attempts to use a refresh token to create a new access token, and the refresh token is invalid, the refresh token may have been used already.

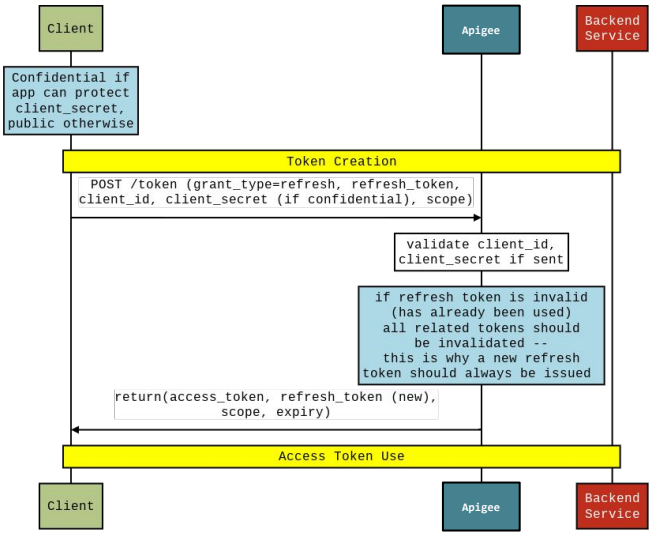


As mentioned before, it is a best practice to always get a new refresh token when using a refresh token. When a new refresh token is created during the refresh grant flow, the original refresh token becomes invalid.

The advantage to getting a new refresh token each time is that compromised refresh tokens can be detected. If an app goes to use a refresh token, and it is invalid, it means that the refresh token might have been used.

Refresh: Token creation

- If app attempts to use a refresh token to create a new access token, and the refresh token is invalid, the refresh token may have been used already.
- Best practice: Revoke all related tokens in case refresh token was compromised.



The best practice is to invalidate all tokens related to the refresh token, in case a bad actor has gained access to a refresh token. The app will need to collect the user's credentials again anyway to get a new access token.

OAuthV2 policy (RefreshAccessToken)

- Custom attributes will automatically be transferred to created tokens.

```
<OAuthV2 name="O2-RefreshToken">
  <Operation>RefreshAccessToken</Operation>
  <ExpiresIn>120000</ExpiresIn>
  <GrantType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
  <RefreshToken>request.formparam.refresh_token</RefreshToken>
  <Scope>request.formparam.scope</Scope>
  <ReuseRefreshToken>false</ReuseRefreshToken>
</OAuthV2>
```

The RefreshAccess token operation for the OAuthV2 policy will create new tokens from a refresh token.

Any custom attributes associated with the refresh token will automatically be inherited by tokens created during this process.

OAuthV2 policy (RefreshAccessToken)

- Custom attributes will automatically be transferred to created tokens.
- Set ReuseRefreshToken to true to keep existing refresh token.
  - Default and best practice is false (create new refresh token each time).

```
<OAuthV2 name="O2-RefreshToken">
  <Operation>RefreshAccessToken</Operation>
  <ExpiresIn>120000</ExpiresIn>
  <GrantType>request.formparam.grant_type</GrantType>
  <GenerateResponse enabled="true"/>
  <RefreshToken>request.formparam.refresh_token</RefreshToken>
  <Scope>request.formparam.scope</Scope>
  <ReuseRefreshToken>false</ReuseRefreshToken>
</OAuthV2>
```

When set to true, ReuseRefreshToken will not create a new refresh token.

By default, ReuseRefreshToken is false, causing a new refresh token to be created each time a refresh token is used. The previous refresh token will be invalid.

Creating a new refresh token each time is highly recommended.

API call: Refresh token

Request without PKCE

```
POST https://api.example.org/oauth/token

Header:
Authorization: Basic YmFkOnBhc3N3b3Jk
Content-Type:
  application/x-www-form-urlencoded

Payload:
grant_type=refresh_token&
refresh_token=l7jlijYuexu6XVSdSFMJq8SVXGOAAq
```

Response:


```
{
  "issued_at": "1565729213000",
  "scope": "READ",
  "application_name": "030fdcea-cf97-12084aea513c",
  "refresh_token_issued_at": "1565729213000",
  "status": "approved",
  "refresh_token_status": "approved",
  "api_product_list": "[gold]",
  "expires_in": ”1799",
  "developer.email": "bob@example.com",
  "organization_id": "0",
  "token_type": "BearerToken",
  "refresh_token": "IFl7jlijYuexu6XVSSjLMJq8SVXGOAAq",
  "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
  "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
  "organization_name": "apigee",
  "refresh_token_expires_in": "28799",
  "refresh_count": "0"
}
```


The refresh token flow requires that a valid and unexpired refresh token be supplied as a form parameter.

For a request without PKCE, the basic auth header should be created using the app's client ID and secret.

The response for the refresh token flow is identical to the response for the original token creation.

API call: Refresh token (PKCE)

Request with PKCE

```
POST https://api.example.org/oauth/token

Header:
(no auth header)
Content-Type:
  application/x-www-form-urlencoded

Payload:
grant_type=refresh_token&
refresh_token=l7jlijYuexu6XVSdSFMJq8SVXGOAAq&
client_id=RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf
```

Response:


```
{
  "issued_at": "1565729213000",
  "scope": "READ",
  "application_name": "030fdcea-cf97-12084aea513c",
  "refresh_token_issued_at": "1565729213000",
  "status": "approved",
  "refresh_token_status": "approved",
  "api_product_list": "[gold]",
  "expires_in": ”1799",
  "developer.email": "bob@example.com",
  "organization_id": "0",
  "token_type": "BearerToken",
  "refresh_token": "IFl7jlijYuexu6XVSSjLMJq8SVXGOAAq",
  "client_id": "RqBca4HGxdyaDM6AAPIHfQ53kLLIGFMf",
  "access_token": "4WCAchNNtVyK8JsACl1HP7mlWW1X",
  "organization_name": "apigee",
  "refresh_token_expires_in": "28799",
  "refresh_count": "0"
}
```


When using PKCE, you can't send in a client secret. The client ID should be passed in as a form parameter along with the refresh token.

The OAuth proxy will need to create the basic auth header before using the OAuthV2 policy to create the new access and refresh token.

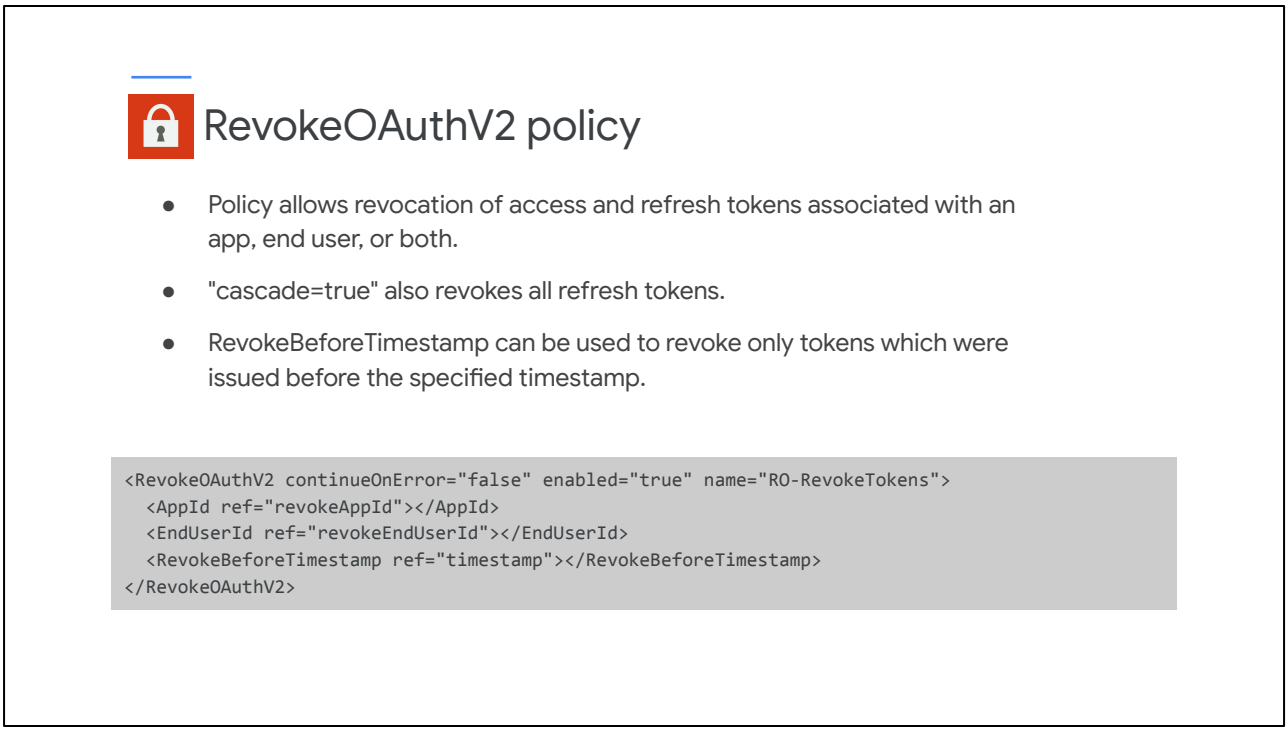
OAuthV2 policy (InvalidateToken)

- Both access and refresh tokens may be revoked.
- "cascade=true" propagates invalidation to all related access and refresh tokens.

```
<OAuthV2 continueOnError="false" enabled="true" name="O2-InvalidateTokens">
  <Operation>InvalidateToken</Operation>
  <Tokens>
    <Token type="accesstoken" cascade="true">request.formparam.access_token</Token>
    <Token type="refreshtoken" cascade="true">request.formparam.refresh_token</Token>
  </Tokens>
</OAuthV2>
```

There is an InvalidateToken operation for the OAuthV2 policy. This can be used to revoke specified access or refresh tokens.

The cascade attribute, when set to true, will cause all related access and refresh tokens to be invalidated. Cascade=true should be used when you think a refresh token or access token may have been compromised.



The InvalidateToken operation of the OAuthV2 policy requires a token. The RevokeOauthV2 policy can invalidate all access tokens for an app or end user ID.

The AppId element specifies the ID for a developer app. The EndUserId element specifies the ID of the end user that was provided when generating the token.

Specifying the app ID without an end user ID will invalidate all access tokens for an app. Specifying both will only invalidate the access tokens that contain that end user ID.

The cascade attribute, when set to true, will cause all refresh tokens for the app and end user to be revoked as well.

If RevokeBeforeTimestamp is specified, only the tokens that were issued before the specified timestamp will be revoked.

GetOAuthV2Info policy

- The GetOAuthV2Info policy gets information about access tokens, refresh tokens, auth codes, or client apps.
- The policy is generally not needed, because the OAuthV2 policy with VerifyAccessToken operation populates similar variables.

```
<GetOAuthV2Info continueOnError="false" enabled="true" name="G2I-GetTokenAttributes">
  <AccessToken ref="request.formparam.access_token"/>
</GetOAuthV2Info>
```

```
Example variables:
oauthv2accesstoken.{policy_name}.scope
oauthv2accesstoken.{policy_name}.expires_in
oauthv2accesstoken.{policy_name}.status
oauthv2accesstoken.{policy_name}.accesstoken.{custom_attribute_name}
oauthv2accesstoken.{policy_name}.refresh_token
```

The GetOAuthV2Info policy will get information about a token, auth code, or client app and populate variables with the details.

This policy is rarely needed, because the VerifyAccessToken operation of the OAuthV2 policy will populate similar variables for use in your proxy.

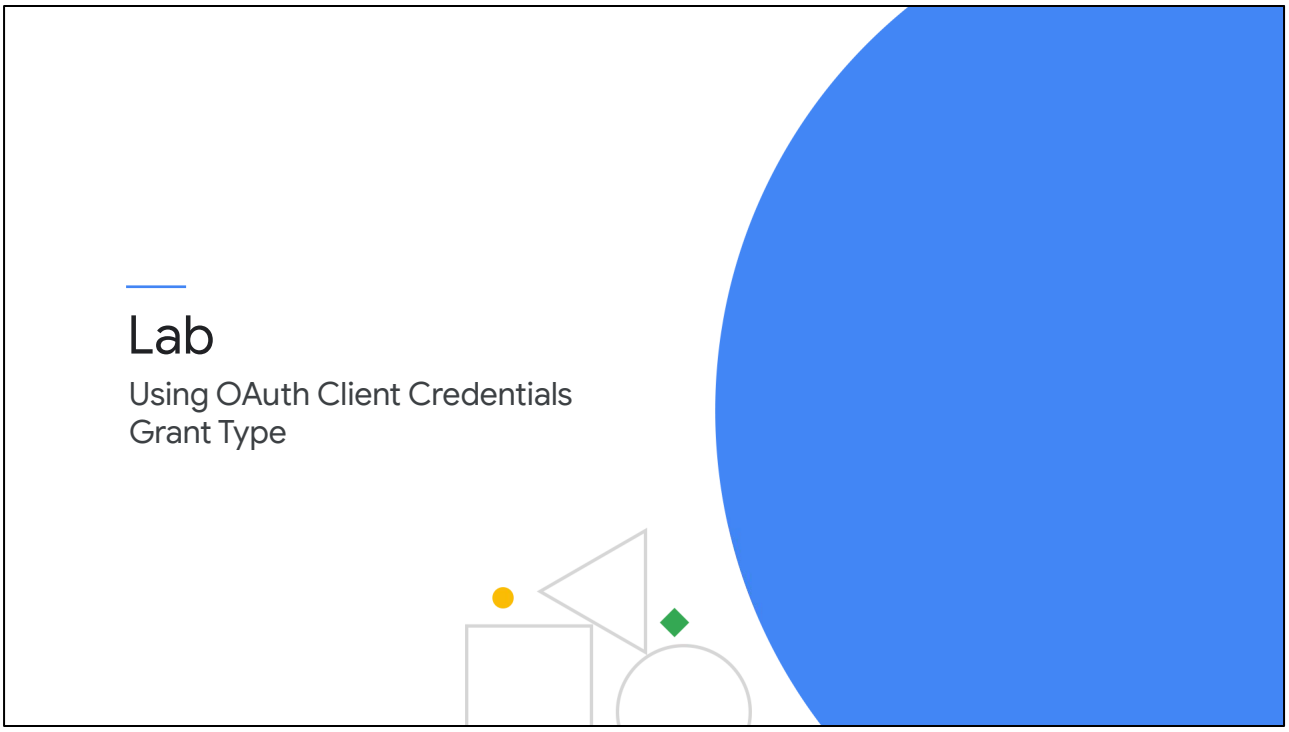
SetOAuthV2Info policy

- The SetOAuthV2Info policy sets custom attributes for access tokens, refresh tokens, auth codes, or client apps.
- Custom attributes are available as variables when access token is verified.

```
<SetOAuthV2Info continueOnError="false" enabled="true" name="S2I-SetBackendToken">
  <AccessToken ref="request.formparam.access_token"/>
  <Attributes>
    <Attribute name="backend.data" ref="backendData" display="false">NONE</Attribute>
  </Attributes>
</GetOAuthV2Info>
```

The SetOAuthV2Info policy is used to set custom attributes for tokens, auth codes, or apps.

This policy can be used to update attributes associated with a token or add new attributes. These attributes will be populated as variables when the access token is verified.



In this lab you add OAuth to your retail API proxy. First, you replace the Verify API Key policy with an OAuthV2 policy that uses the VerifyAccessToken operation. You create an access token for your app using a separate OAuth proxy, and then you test your retail proxy.

When you finish testing your proxy with OAuth tokens, you will replace the OAuthV2 policy with the VerifyAPIKey policy.

Agenda

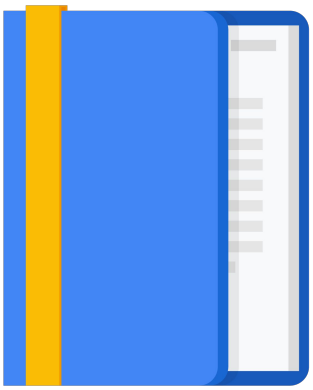
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up

JWT, JWS, SAML, OpenID Connect

Quiz



We've covered quite a bit about OAuth 2.0, the authorization framework used by many APIs.

In this section, we'll discuss JSON web tokens and signatures. We'll also cover federated identity and two protocols that can be used for federated identity: SAML and OpenID Connect.

JSON Web Token (JWT)

- JWT (pronounced "jot") is a JSON-based access token
  - Asserts a set of claims.
- JWTs are digitally signed, enabling claim verification without sending to a token server.
  - Valuable for microservices because verification is quick.
- JWTs are not verified by a token server on every call, so they generally cannot be revoked.
  - Therefore the best practice is to keep the TTL fairly short.

A JSON web token (JWT), or "jot," is a JSON-based access token. The JWT is used to assert claims that can be trusted by the recipient.

JWTs are digitally signed. This means that the JWT creator uses encryption to create a signature. By decrypting the digital signature, the recipient of the JWT can verify that the token was signed by the token service and that the token has not been modified since it was created. JWT tokens can therefore be verified by a service without requiring a network hop.

This is in contrast to an opaque OAuth token, which cannot be verified without sending it to a token service because there is no encoded value in the token itself.

This makes JWTs very useful for microservices, where there might be several microservices for a single API call, and it would cause significant latency for each microservice to make a network call to verify the token.

The downside to JWTs is that they typically cannot be revoked. Opaque OAuth tokens require validation by a token service for each call, so a token can be invalidated at that service so that it is not usable in the future. JWT tokens are validated by the service locally, so it generally isn't practical to create a revocation list at each service.

This leads to a best practice for JWTs: keep the time-to-live short, because the JWT will be valid as long as it lives.

JWT format

- JWTs have three sections, each Base64URL encoded, separated by dots.

The JWT format includes three sections.

These sections are base64 URL encoded separately, and then separated by dots.

JWT format

- JWTs have three sections, each Base64URL encoded, separated by dots.
  - **Header**: algorithm for signing

```
Unencoded Header:
  {"alg":"HS256","typ":"JWT"}
```

The first section is the header, a JSON structure with claims about the token itself.

The header is primarily used to specify the algorithm used to generate the signature.

In this case, the algorithm is HMAC SHA256.

JWT format

- JWTs have three sections, each Base64URL encoded, separated by dots
  - **Header**: algorithm for signing
  - **Payload**: claims about the user

```
Unencoded Header:
  {"alg":"HS256","typ":"JWT"}

Unencoded Payload:
  {"sub":"18373426","iss":"example.com","aud":"idsvc",
   "iat":1570819380,"exp":1570819680,
   "name":"Joe Apigeek","locale":"en",
   "email":"apigeek@example.com"}
```

The second section is the payload, which contains claims about the user.

There are some predefined claim names, for standard types of claims.

In this example, the standard claims are a subject, an issuer, an audience, an issued-at time, and an expiration time.

JWT format

- JWTs have three sections, each Base64URL encoded, separated by dots
  - **Header**: algorithm for signing
  - **Payload**: claims about the user

```
Unencoded Header:
  {"alg":"HS256","typ":"JWT"}

Unencoded Payload:
  {"sub":"18373426","iss":"example.com","aud":"idsvc",
   "iat":1570819380,"exp":1570819680,
   "name":"Joe Apigeek","locale":"en",
   "email":"apigeek@example.com"}
```

Custom claims can also be created.

Here we have a name, locale, and email address for the user.

JWT format

- JWTs have three sections, each Base64URL encoded, separated by dots
  - **Header**: algorithm for signing
  - **Payload**: claims about the user
  - **Signature**: guarantees integrity of the token

```
Unencoded Header:
  {"alg":"HS256","typ":"JWT"}

Unencoded Payload:
  {"sub":"18373426","iss":"example.com","aud":"idsvc",
   "iat":1570819380,"exp":1570819680,
   "name":"Joe Apigeek","locale":"en",
   "email":"apigeek@example.com"}

Signature Calculation:
   HMACSHA256(
     base64UrlEncode(header) + "." + base64UrlEncode(payload),
     secret)
```

The third section is the signature, which is calculated by encoding and cryptographically signing the header and payload. The signature guarantees the integrity of the token.

The header shows HMAC SHA256 as the hashing function, so the signature will be an SHA256 hash of the base64 URL encoded header and payload, separated by a dot.

JWT format

- JWTs have three sections, each Base64URL encoded, separated by dots
  - **Header**: algorithm for signing
  - **Payload**: claims about the user
  - **Signature**: guarantees integrity of the token

```
Unencoded Header:
  {"alg":"HS256","typ":"JWT"}

Unencoded Payload:
  {"sub":"18373426","iss":"example.com","aud":"idsvc",
   "iat":1570819380,"exp":1570819680,
   "name":"Joe Apigeek","locale":"en",
   "email":"apigeek@example.com"}

Signature Calculation:
   HMACSHA256(
     base64UrlEncode(header) + "." + base64UrlEncode(payload),
     secret)

Token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxODM3MzQyNiIsIm
lzcyI6ImV4YW1wbGUuY29tIiwiYXVkIjoiaWRzdmMiLCJpYXQiOjE1NzA4MTkzO
DAsImV4cCI6MTU3MDgxOTY4MCwibmFtZSI6IkpvZSBBcGlnZWVrIiwibG9jYWxl
IjoiZW4iLCJlbWFpbCI6ImFwaWdlZWtAZXhhbXBsZS5jb20ifQ.b__ZOVZ0yHlL
yFifHZHGxI73S3bSXqQuE27gkeBp4bs
```

The token shown is the base64 URL encoded sections, concatenated and separated by periods.

Note that this token is not encrypted. The first two sections are base64 URL encoded, and they are easily reversed to reveal the JSON.

The JWT will be encrypted while traveling across the network because it must always be sent via TLS.

Like an opaque OAuth token, a JWT should be stored securely.

GenerateJWT policy

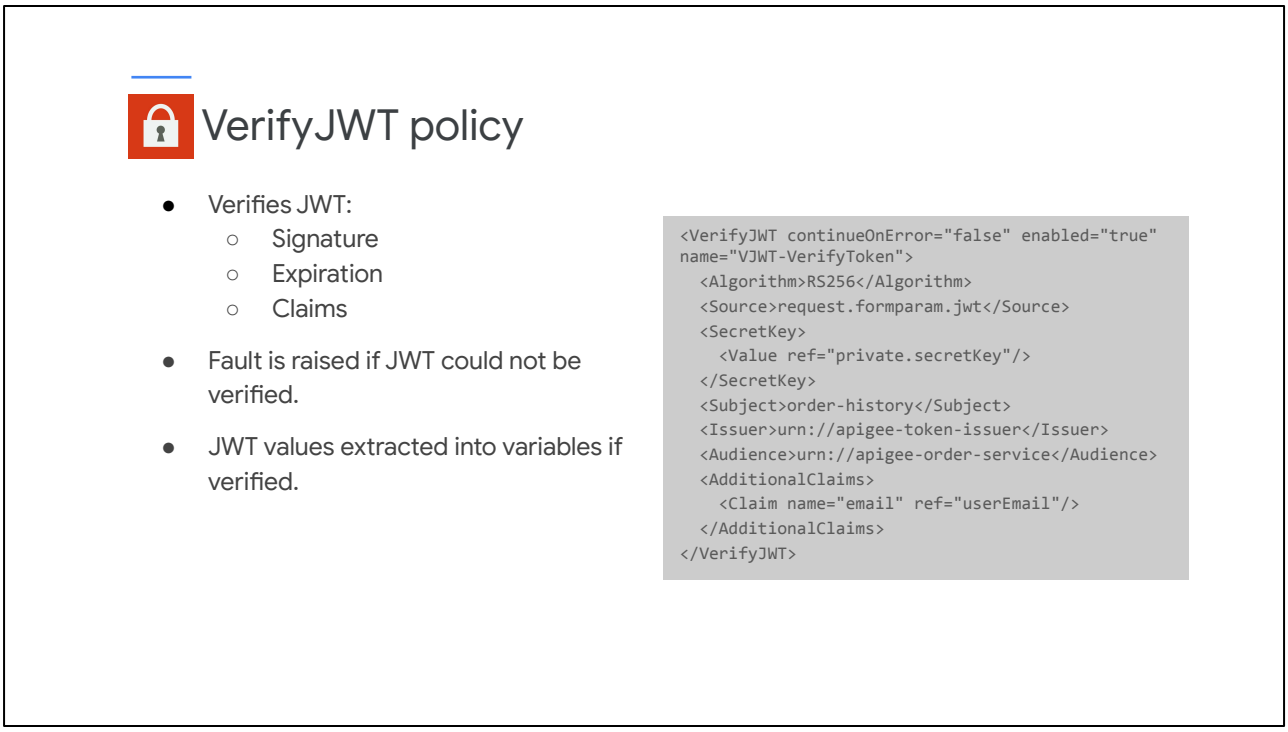
- Generates a signed JWT with configured set of claims.
- Token can be generated using key encryption algorithms supported by OpenSSL.

```
<GenerateJWT continueOnError="false" enabled="true"
name="GJWT-GenerateRS256">
  <Algorithm>RS256</Algorithm>
  <PrivateKey>
    <Value ref="private.privateKey"/>
    <Password ref="private.keyPassword"/>
    <Id ref="private.keyId"/>
  </PrivateKey>
  <Subject>order-history</Subject>
  <Issuer>urn://apigee-token-issuer</Issuer>
  <Audience>urn://apigee-order-service</Audience>
  <ExpiresIn>60m</ExpiresIn>
  <Id/>
  <AdditionalClaims>
    <Claim name="email" ref="userEmail"/>
  </AdditionalClaims>
  <OutputVariable>jwtString</OutputVariable>
</GenerateJWT>
```

Apigee has policies that generate, decode, and verify JWTs.

The GenerateJWT policy creates a signed JWT with a configured set of claims and stores the JWT string in an output variable.

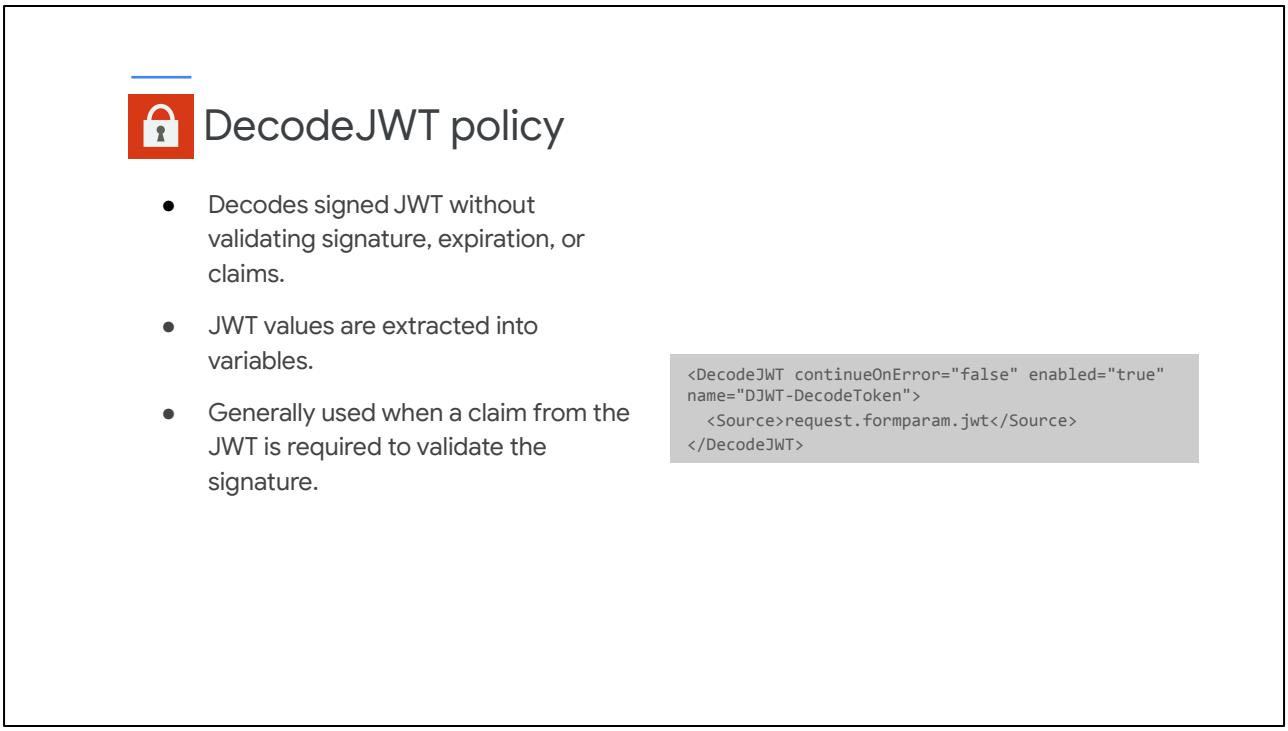
JWTs can be generated with key encryption algorithms that are supported by the OpenSSL library.



The VerifyJWT policy verifies the validity of a signed JWT.

If the signature is invalid, the JWT has expired, or claims set in the policy don't match the claims in the token, the policy will raise a fault.

If the JWT is valid, all of the JWT information will be populated into variables for use in the proxy.

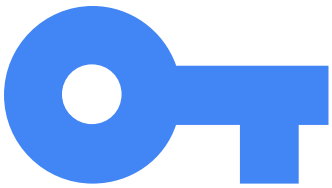


The DecodeJWT policy decodes the signed JWT without validating the signature, expiration, or any claims. This policy will just extract the values of the JWT into variables.

The DecodeJWT policy is generally used only when information from the JWT is required to verify the signature. For example, if you allow multiple signing algorithms, DecodeJWT might be used to determine the algorithm, and then VerifyJWT would be used to make sure the token is valid.

JSON Web Signature (JWS)

- A JWS can be used to digitally sign any payload.
- A JWS payload can have any format and does not require the payload to be attached to the JWS.
- Signed JWT is a JWS with attached payload containing set of claims in a JSON object.
- Policies similar to the JWT policies exist for JWS:
  - GenerateJWS
  - VerifyJWS
  - DecodeJWS



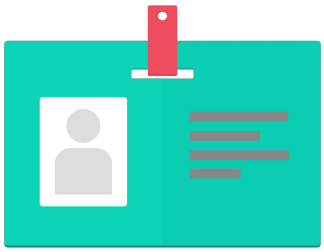
Apigee also provides policies for handling a JSON web signature, or JWS.

A JWS is more flexible than a JWT. JWS doesn't have format requirements for the payload, and the payload does not need to be attached to the JWS.

A signed JWT is just a JWS with an attached JSON payload that contains a set of claims.

There are three JWS policies that are similar to the three JWT policies: GenerateJWS, VerifyJWS, and DecodeJWS.

Federated identity



- Identity Provider (IdP):
  - Stores user profiles.
  - Is responsible for authenticating users.
- Service provider:
  - Can delegate identity management to trusted IdPs.
- Users benefit from:
  - Fewer accounts and passwords to manage.
  - Single sign-on (SSO) to multiple apps and services.
- Standards: SAML, OpenID Connect

Now let's talk about federated identity.

Management of users involves managing user profile information like names, email addresses, and passwords.

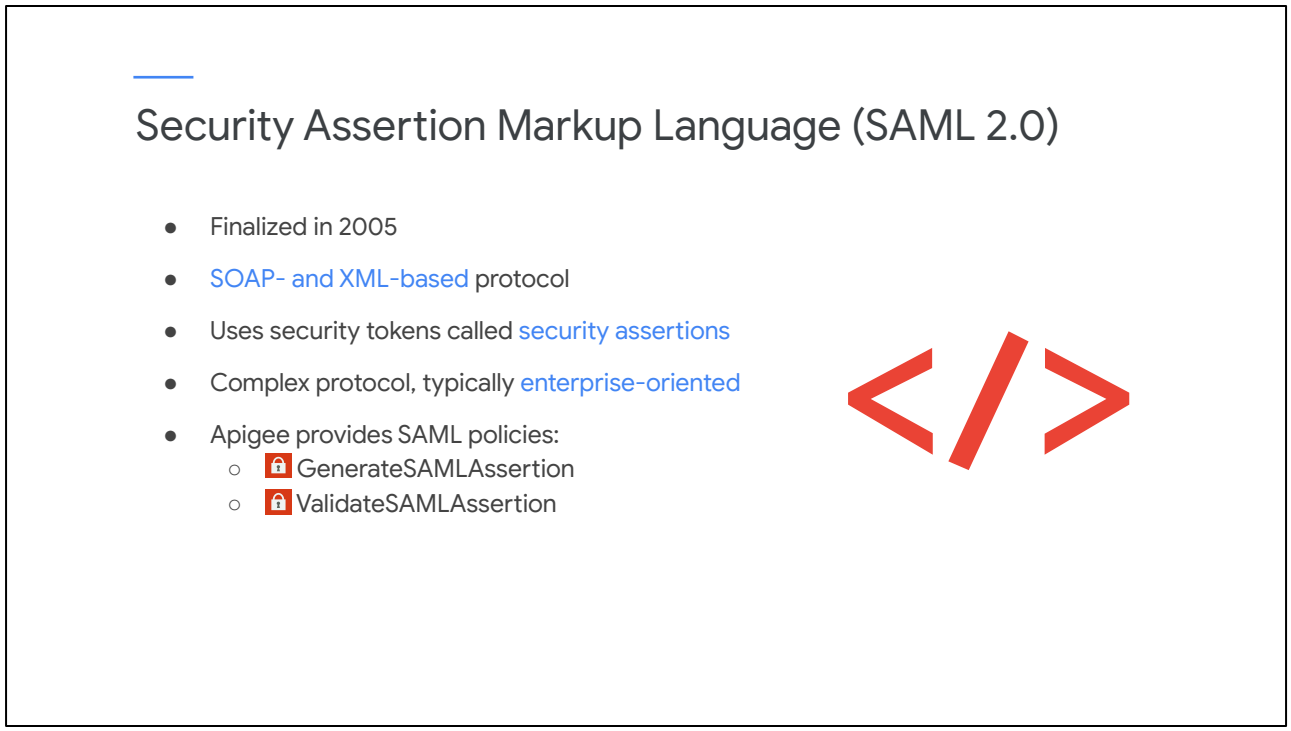
An identity provider, or IdP, manages and secures the user information stored in its database and authenticates those users.

A service provider needs to manage users of its service. Securing passwords and user data is of utmost importance.

Many service providers choose not to manage user data, but instead delegate the management to one or more trusted identity providers, or IDPs. For example, many apps and services allow you to use your Google account to log in. The service provider can choose to store the name and email of the user, but rely on Google to maintain the security of the rest of the profile information.

Users also benefit from federated identity. Users have fewer accounts and passwords to manage and can benefit from single sign-on to multiple apps and services.

Let's learn about two of the most popular standards that can be used to provide federated identity: SAML and OpenID Connect.



SAML stands for Security Assertion Markup Language.

SAML 2.0, the current version in use, was finalized in 2005.

It is a SOAP- and XML-based protocol that uses XML-formatted tokens, known as security assertions, to exchange authentication and authorization information between IdPs and service providers.

SAML is a fairly complex protocol, still used by many large enterprises.

Apigee provides two SAML policies you can use in your proxies: GenerateSAMLAssertion creates a security assertion, and ValidateSAMLAssertion validates a security assertion and sets variables containing the information from the assertion.

OpenID Connect (OIDC)

- Lightweight RESTful redesign of SAML was created in 2014.
- Layers user authentication and identity on OAuth 2.0.
- Adds ID token to OAuth 2.0:
  - ID token is secure JWT containing user claims.
- ID tokens can be used to verify the authentication of users and get user details without contacting the issuer.
- Apigee's JWT and OAuthV2 policies can be used to create OIDC providers and consumers.

OpenID Connect, or OIDC, is a lightweight, RESTful redesign of SAML. OIDC layers identity and authentication on top of the OAuth 2.0 authorization framework, adding an ID token to OAuth.

The ID token can be created by an OpenID provider when a user authenticates. The ID token will contain claims about the user and is represented as a secure JWT token.

An ID token can then be used to verify the identity of the bearer and get user details without contacting the IdP that issued the token.

OIDC is typically preferred over SAML for mobile applications or systems that are already using OAuth 2.0.

Apigee's OAuthV2 and JWT policies can be used to create OIDC providers and consumers.

Agenda

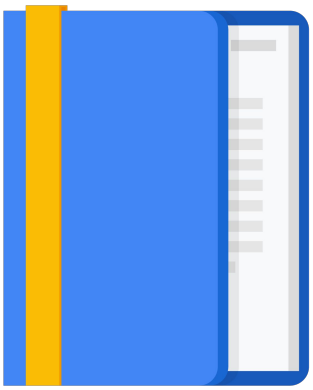
API Security Concerns

Identity, AuthN, and AuthZ

OAuth
- Introduction
- Client Credentials Grant
- Password Grant
- Authorization Code Grant
- Wrap-up (lab)

JWT, JWS, SAML, OpenID Connect

Quiz



Quiz

Which OAuth grant type should be used for untrusted apps that need access to user data?

A. Client credentials grant type

B. Password grant type

C. Authorization code grant type

D. Implicit grant type

Quiz

Which OAuth grant type should be used for untrusted apps that need access to user data?

A. Client credentials grant type

B. Password grant type

C. Authorization code grant type

D. Implicit grant type

Explanation:

A: Client credentials grant type - Sorry, this is not correct. The client credentials grant type should never be used to access user data.

B: Resource owner password grant type - Sorry, this is not correct. The password grant type should only be used for trusted apps.

*C: Authorization code grant type - Correct!

D: Implicit grant type - Sorry, this is not correct. The implicit grant type is never recommended.

Quiz

Why does the client credentials grant type **not** use refresh tokens?

A. The client credentials grant type flow cannot secure refresh tokens.

B. The authorization code grant type with PKCE should now be used instead of the client credentials grant type.

C. An app's credentials do not change, so a refresh token is unnecessary.

D. Refresh tokens are only used when user credentials are necessary.

Quiz

Why does the client credentials grant type **not** use refresh tokens?

A. The client credentials grant type flow cannot secure refresh tokens.

B. The authorization code grant type with PKCE should now be used instead of the client credentials grant type.

C. An app's credentials do not change, so a refresh token is unnecessary.

D. Refresh tokens are only used when user credentials are necessary.

Explanation:

A: The client credentials grant type flow cannot secure refresh tokens. - No, that's not correct.

B: The authorization code grant type with PKCE should now be used instead of the client credentials grant type. - No, that's not correct. The client credentials grant type is appropriate for use cases without user data.

C: An app's credentials do not change, so a refresh token is unnecessary. - No, that's not correct. Refresh tokens are never used to replace app credentials.

*D: Refresh tokens are only used when user credentials are necessary. - Correct!

Quiz

Which of the following are reasons that the authorization code flow uses a user agent? Select two.

A. The user does not need to expose their password to the app.

B. User authentication should not be done in a mobile app, so the app provides a web interface for validating the user credentials.

C. The user is given the chance to consent to the level of access being requested by the app.

D. One part of the authorization code flow uses cookies, so a web request is necessary.

Quiz

Which of the following are reasons that the authorization code flow uses a user agent? Select two.

A. The user does not need to expose their password to the app.

B. User authentication should not be done in a mobile app, so the app provides a web interface for validating the user credentials.

C. The user is given the chance to consent to the level of access being requested by the app.

D. One part of the authorization code flow uses cookies, so a web request is necessary.

Explanation:

*A: The user does not need to expose their password to the app. - Correct!

B: User authentication should not be done in a mobile app, so the app provides a web interface for validating the user credentials. - Incorrect. The web user interface used in the authorization code flow is not provided by the app.

*C: The user is given the chance to consent to the level of access being requested by the app. - Correct!

D: One part of the authorization code flow uses cookies, so a web request is necessary. - Incorrect. OAuth does not use cookies.

Quiz

Which of the following statements about the Proof Key for Code Exchange (PKCE) extension are true? Select three.

A. PKCE adds an extra call to the authorization code flow that is used to validate the code verifier.

B. PKCE is necessary because an authorization code may be intercepted by a bad actor.

C. The authorization code grant type with PKCE should be used instead of the implicit grant type.

D. PKCE uses a one-way hash to prove the identity of the app.

E. PKCE can be used to secure OAuth when TLS is not being used.

Quiz

Which of the following statements about the Proof Key for Code Exchange (PKCE) extension are true? Select three.

A. PKCE adds an extra call to the authorization code flow that is used to validate the code verifier.

B. PKCE is necessary because an authorization code may be intercepted by a bad actor.

C. The authorization code grant type with PKCE should be used instead of the implicit grant type.

D. PKCE uses a one-way hash to prove the identity of the app.

E. PKCE can be used to secure OAuth when TLS is not being used.

Explanation:

A: PKCE adds an extra call to the authorization code flow that is used to validate the code verifier. - Incorrect. PKCE is included in the existing calls.

*B: PKCE is necessary because an authorization code may be intercepted by a bad actor. - Correct!

*C: The authorization code grant type with PKCE should be used instead of the implicit grant type. - Correct!

*D: PKCE uses a one-way hash to prove the identity of the app. - Correct!

E: PKCE can be used to secure OAuth when TLS is not being used. - No, TLS must always be used for any OAuth request.

Quiz

True or false:

It is best to generate a new refresh token every time a refresh token is used.

Quiz

True or false:

It is best to generate a new refresh token every time a refresh token is used.

True! Generating new refresh tokens makes it possible to detect when a refresh token has been compromised.

Quiz

Which of the following statements about JWTs are true? Select two.

A. JWTs are encrypted, so TLS is not required.

B. JWTs can be validated without sending them to a token server.

C. SAML uses JWTs for passing authentication and authorization information between IdPs and service providers.

D. The JWT payload is used to guarantee the integrity of the token.

E. Apigee provides policies for generating and verifying JWTs.

Quiz

Which of the following statements about JWTs are true? Select two.

A. JWTs are encrypted, so TLS is not required.

B. JWTs can be validated without sending them to a token server.

C. SAML uses JWTs for passing authentication and authorization information between IdPs and service providers.

D. The JWT payload is used to guarantee the integrity of the token.

E. Apigee provides policies for generating and verifying JWTs.

Explanation:

A: JWTs are encrypted, so TLS is not required. - Incorrect. JWTs are often not encrypted.

*B: JWTs can be validated without sending them to a token server. - Correct!

C: SAML uses JWTs for passing authentication and authorization information between IdPs and service providers. - No, SAML uses XML-based security assertions.

D: The JWT payload is used to guarantee the integrity of the token. - No, the JWT signature guarantees the token's integrity.

*E: Apigee provides policies for generating and verifying JWTs. - Correct!

Review

Developing APIs with Apigee

Authentication, Authorization, and OAuth

In this module you were introduced to API security concerns.

You learned about OAuth, the most popular authorization framework for REST APIs, and added verification of OAuth tokens to your retail API proxy.

We also learned about JWT tokens and API federated identity.