Pivotal Cloud Foundry®

OAuth Primer

Pivotal.

Topics

- **OAuth and OpenID Connect Introduction**
- OAuth Application Types

Why an OAuth Primer?

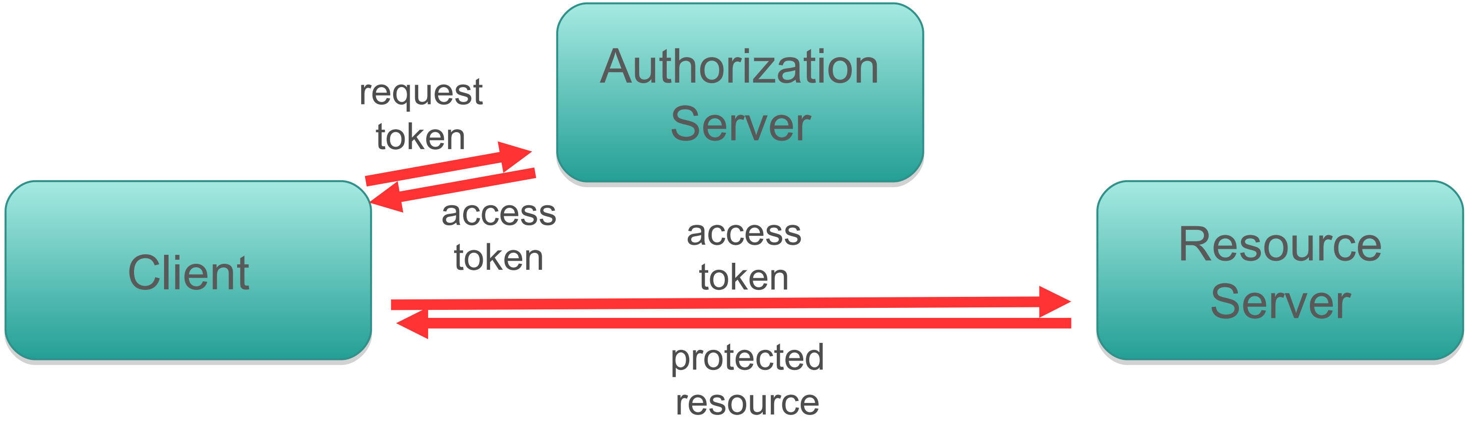
- Securing the Cloud Foundry platform and applications includes OAuth
- OAuth demonstrates a modern approach to securing highly distributed cloud-native applications

Authentication vs. Authorization

- Authentication- Verifies the credentials of a user
- Authorization- Verifies that connection from an authenticated user to a resource is allowed

OAuth 2.0

- Open standard framework for delegated resource authorization
  - http://oauth.net
  - Flexible enough to handle many implementations and authorization flows
- Designed for distributed systems
- Allows third-party delegated access to secure resources via access tokens, not by sharing user credentials

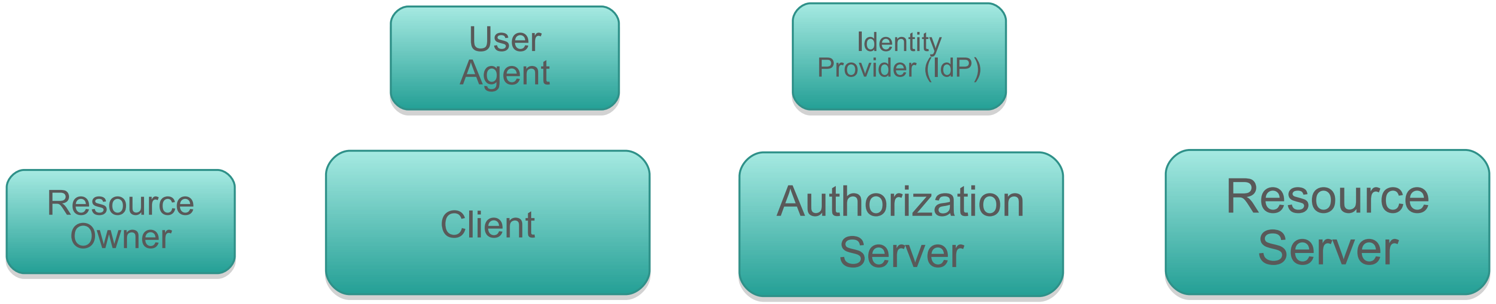


OpenID Connect (OIDC)

- Delegated identity protocol built on top of OAuth 2.0
  - http://openid.net/connect
- Enables the client to use the authorization server to authenticate the user
  - Complementary to resource authorization in OAuth 2.0
- Provides the client basic profile information about the user
- Enables a single set of user credentials to access multiple sites
- Standard means of obtaining and verifying identity (ID) tokens
  - ID tokens convey identity and authorization information

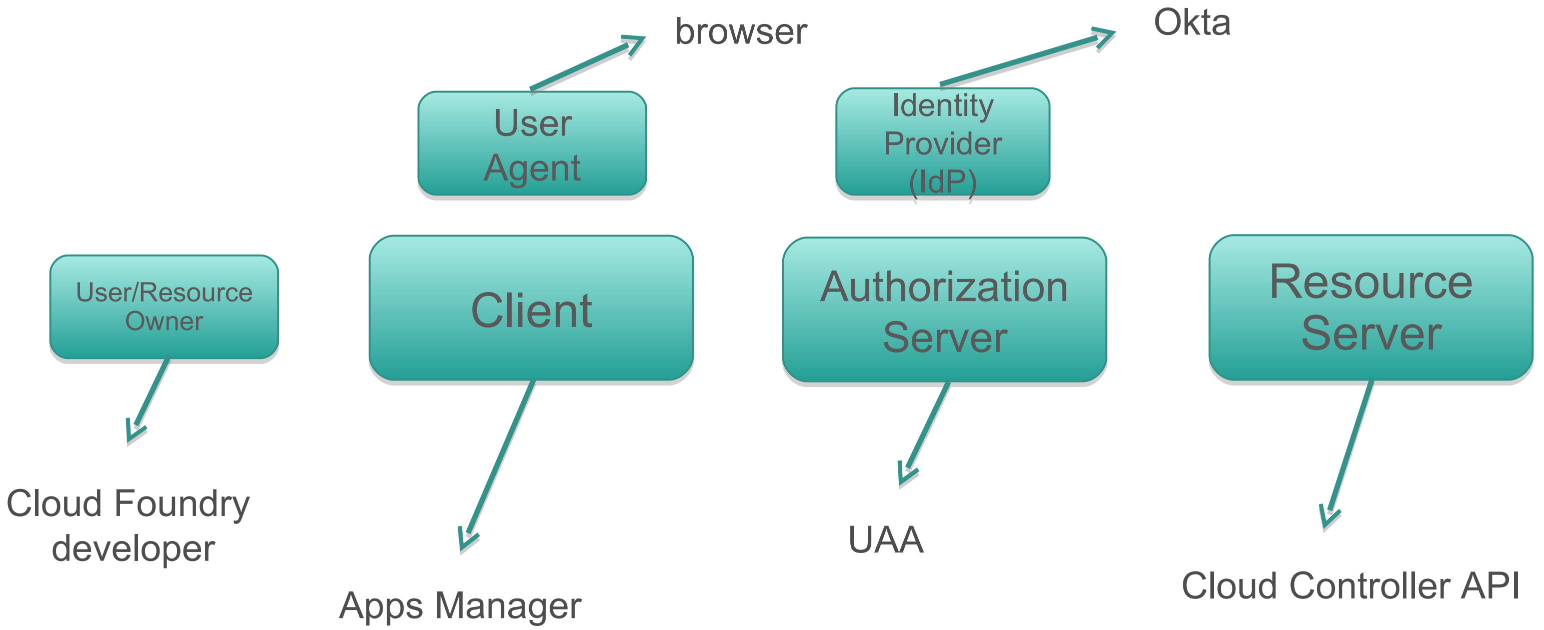
The UAA issues OAuth 2.0 or OpenID Connect ID tokens

OAuth / OpenID Connect Players



- Resource Owner / User- The person or system wanting to access resources (provides credentials)
- Client- Application wanting to access resources on the resource server (e.g. an app on a web server)
- User Agent- The application used to access the client (e.g. a browser)
- Authorization Server / OAuth Provider / OAuth Server / Token Server- Acts as a security intermediary- issues tokens
- Identity Provider / User Store- Provides user information to the authorization server (Ping Identity, CA SSO, Azure ADFS, Okta, …)
- Resource Server / Resource Provider- Contains the token-protected resources (services, APIs, etc.)

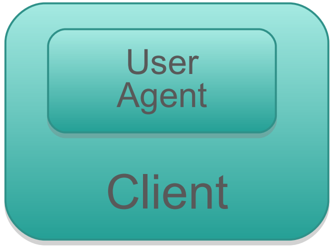
OAuth / OpenID Connect- Example Players



Note: Depending on the circumstances, some of the players are not used or are combined

Types of Clients

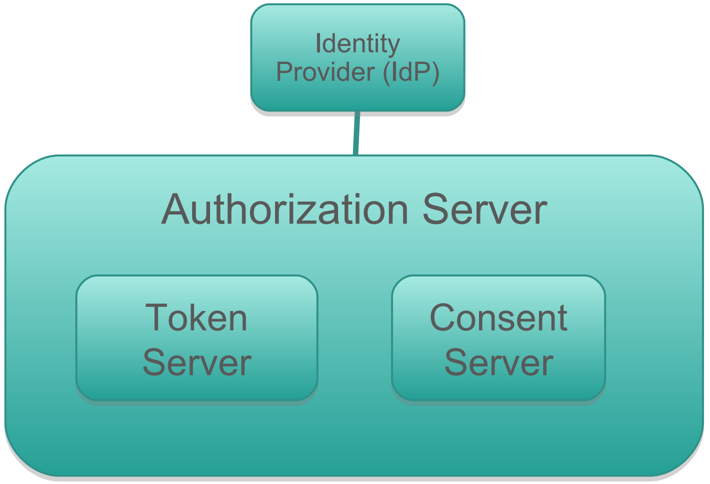
1. Confidential- can keep client authentication information and tokens secret
   - Web application- client running on a web server
     - No client credentials or tokens available to the resource owner
2. Public- client can not be trusted to hold client authentication information- client code is downloaded to the user agent
   - Browser- single page JavaScript app
   - Mobile app



Note: A client can be distributed, e.g. consisting of a browser and a web app

•*Each component can have a different client type and security context*

Authorization Server



- Token server- Creates and manages authentication codes and tokens
- Consent server- Requests consent from an authenticated user for access rights for the client

OAuth Access Tokens and OpenID Connect ID Tokens

- OAuth access tokens and OpenID Connect ID tokens are a confirmation of the delegation of access rights
  – Avoids the password anti-pattern
- Tokens have access rights associated with them
  - Secure them with SSL/TLS (transport layer security) communications
  - They usually are signed/self validating JSON Web Tokens (JWT)

Pivotal

Types of Tokens

- Access token and ID token
  - Sent by the client to the resource server to access protected resources
  - May be valid for a short period (implementation specific)
  - Can be verified by the authorization server or the resource server
- Refresh token
  - Valid for a longer period
  - Used to request a new access or identity token when the current one expires
  - The process is transparent for the resource owner
  - When a refresh token expires, the client must re-authenticate
  - Usually involves a database lookup, so authorization can be revoked

Topics

- OAuth and OpenID Connect Introduction
- **OAuth Application Types**

Application Types

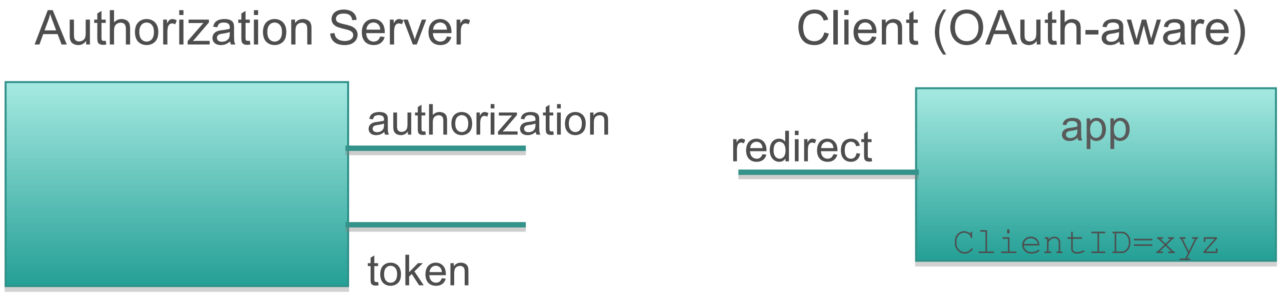
- There are four application types defined in the OAuth 2.0 spec:
  - Web App (grant_type=authorization_code)
  - Native Mobile App (grant_type=password)
  - Single Page JavaScript App (grant_type=implicit)
  - Service-to-Service App (grant_type=client_credentials)
- Each application type represents a different authorization flow
- Some application types are more secure than others- the spec is flexible

Stages of OAuth Security

1. Registration
   - Client, authorization server and resource server establish trust by exchanging metadata and secrets
   - This is usually a one-time, manual setup to enable token exchange
   - This is when the ClientID and ClientSecret are created
2. Authorization
   - The client requests permission to access protected resources for a particular user
3. Token Exchange
   - Tokens provided to the resource server when accessing protected resources

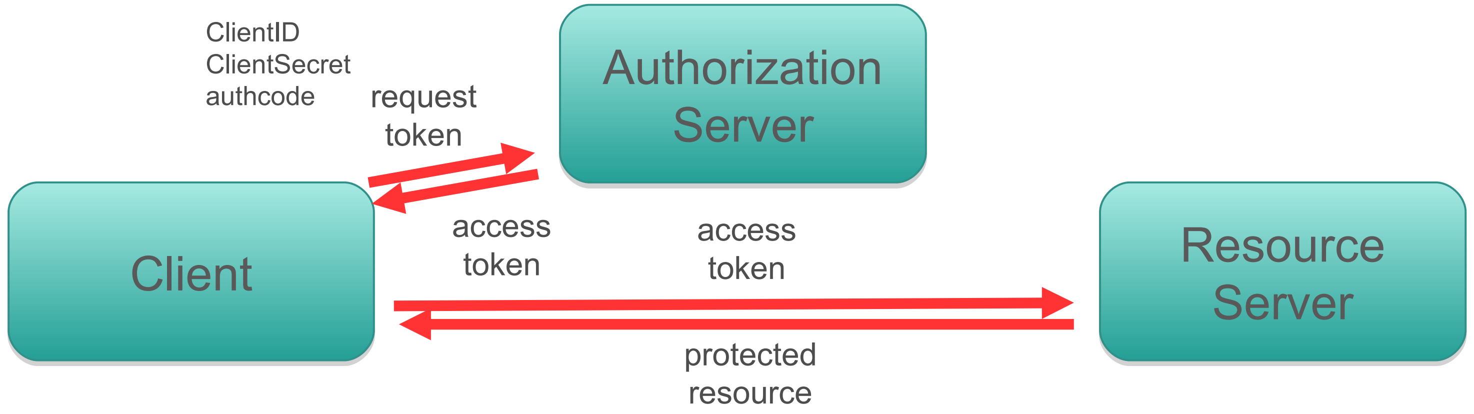
OAuth RESTful Endpoints/Services used with the Application Types

- On the authorization server:
  - authorization- provides a client with an authorization code
  - token- accepts an authorization code or refresh token from the client and returns refresh and access tokens
    - Protected using HTTP Basic authentication with ClientID:ClientSecret (base64 encoded)
- On your OAuth 2.0-aware application (client):
  - redirect- callback entry point on the client- receives the authorization code from the authorization endpoint
    - The redirect_uri provided by the client during registration must match the redirect_uri sent during an authorization request



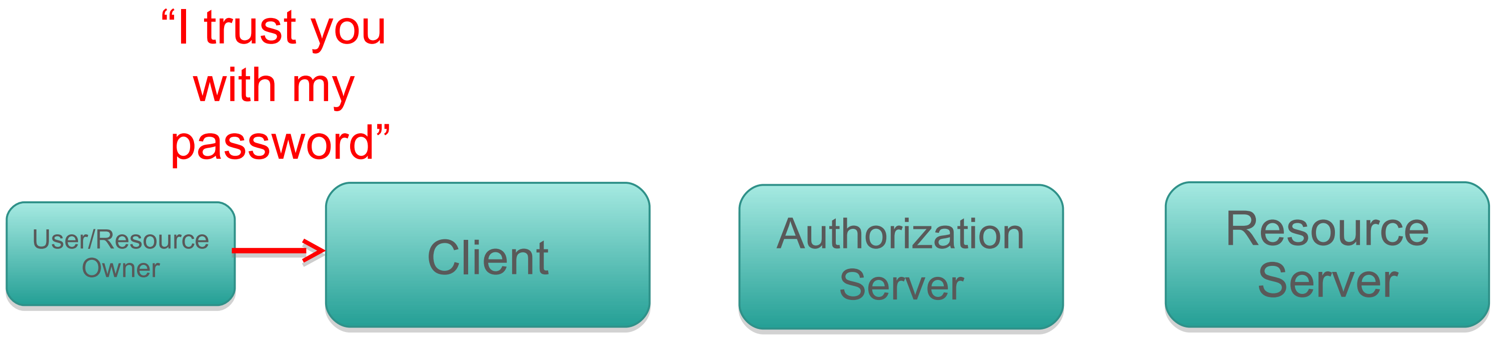
Authorization Code Flow (grant_type=authorization_code)

- Default and most secure flow- no access token on the browser, no user/password sent to client
- Optimized for confidential clients
  - The client must securely be able to hold the ClientID, ClientSecret and tokens
- Called 3 legged OAuth because the identity of the three main actors is checked (authorization server, resource owner, client)



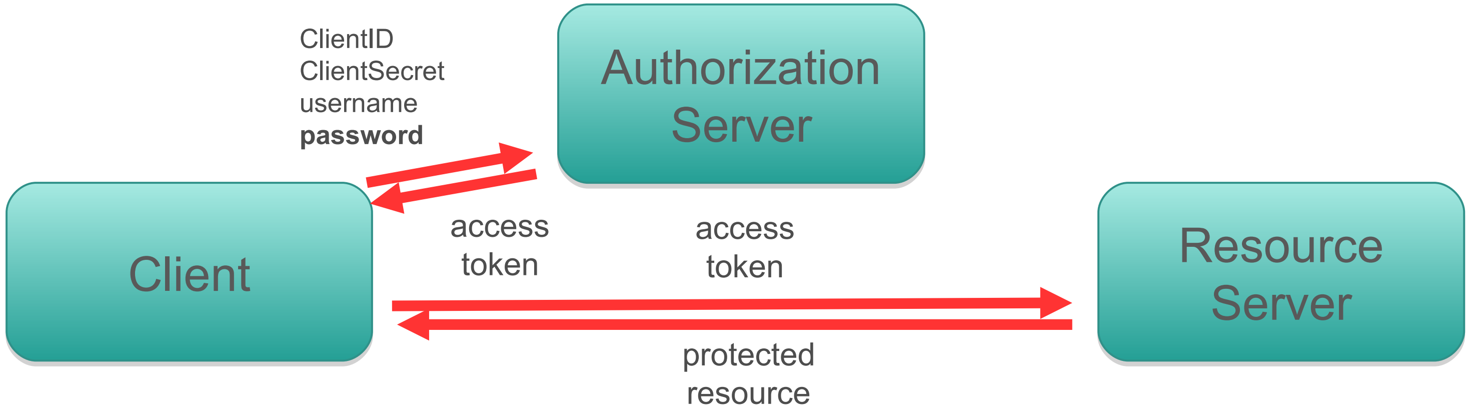
Password Flow (grant_type=password)

- Sometime called “native mobile app” or “resource owner password” flow
- For cases when the resource owner trusts giving the password to the client (e.g. same company)
  - Usually the resource owner only trusts the Authorization Server login component
- The client uses the username and password when requesting tokens, but does not store them after that
  - Only the access and refresh tokens should be stored by the client
- Uses credentials as authorization- no authorization endpoint is used



Password Flow (grant_type=password)

1. Client first registers with the OAuth provider- obtains ClientID and ClientSecret
2. Client requests (via HTTP POST) tokens from the token endpoint of the OAuth provider
   - The client uses base64-encoded value of ClientID:ClientSecret in the Authorization HTTP header
   - Credentials are passed as form parameters: username, password and grant_type=password
   - Response is JSON containing access and refresh tokens, and expiration



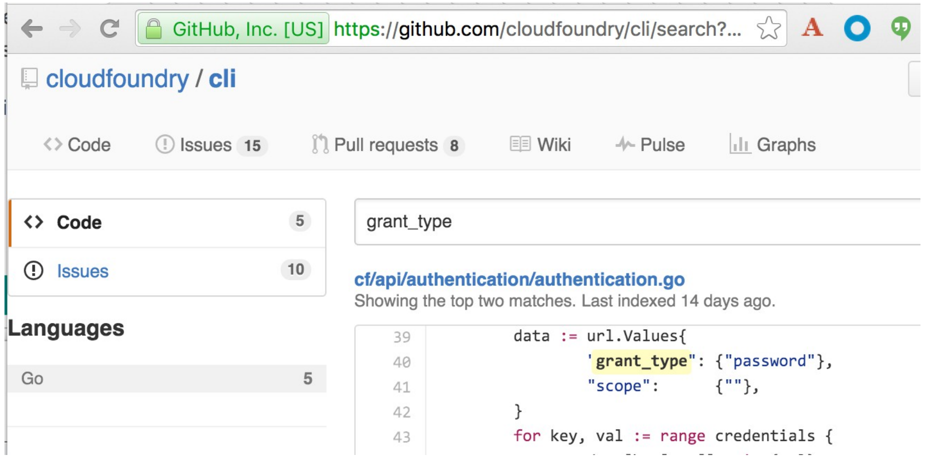
Password Flow Example- UAA API

- To obtain an access token, use UAA's /oauth/token endpoint
- This token can then be used to make Cloud Controller API calls

```
curl -k -H 'AUTHORIZATION: Basic Y2Y6'
-d 'username=[username]&
password=[password]&grant_type=password'
https://uaa.[path to system domain]/oauth/token
```

Password Flow Use- cf CLI

- The cf CLI is an open source client that makes Cloud Controller API calls
- You can search the source code of the cf CLI for grant_type:



GitHub, Inc. [US] https://github.com/cloudfoundry/cli/search?...

cloudfoundry / cli

Code | Issues 15 | Pull requests 8 | Wiki | Pulse | Graphs

Code 5
Issues 10

Languages

Go 5

grant_type

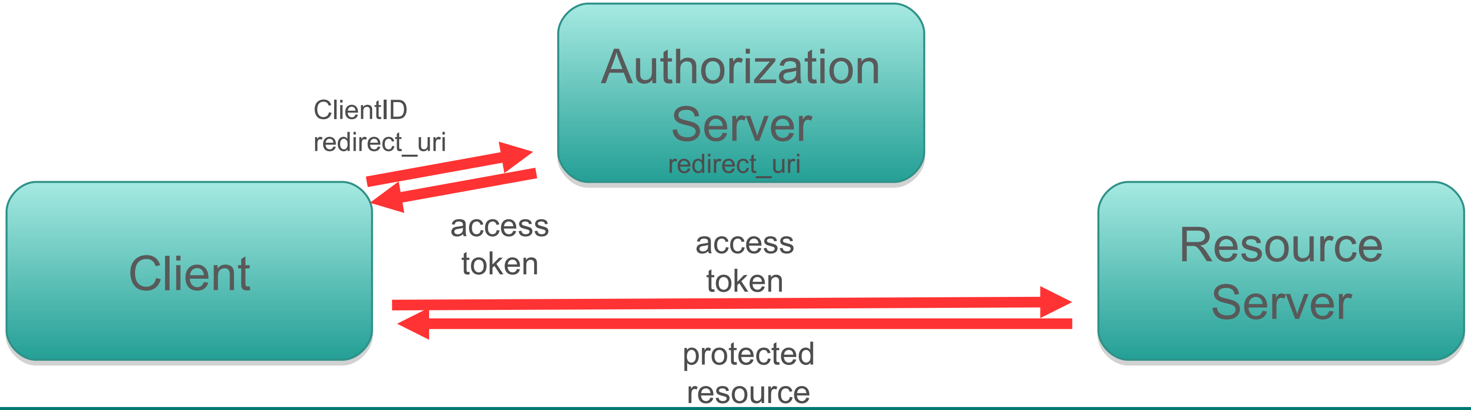
cf/api/authentication/authentication.go
Showing the top two matches. Last indexed 14 days ago.

```
	data := url.Values{
		"grant_type": {"password"},
		"scope":      {""},
	}
	for key, val := range credentials {
```

The cf CLI uses the password grant type- users provide their password directly to the application rather than using a login component

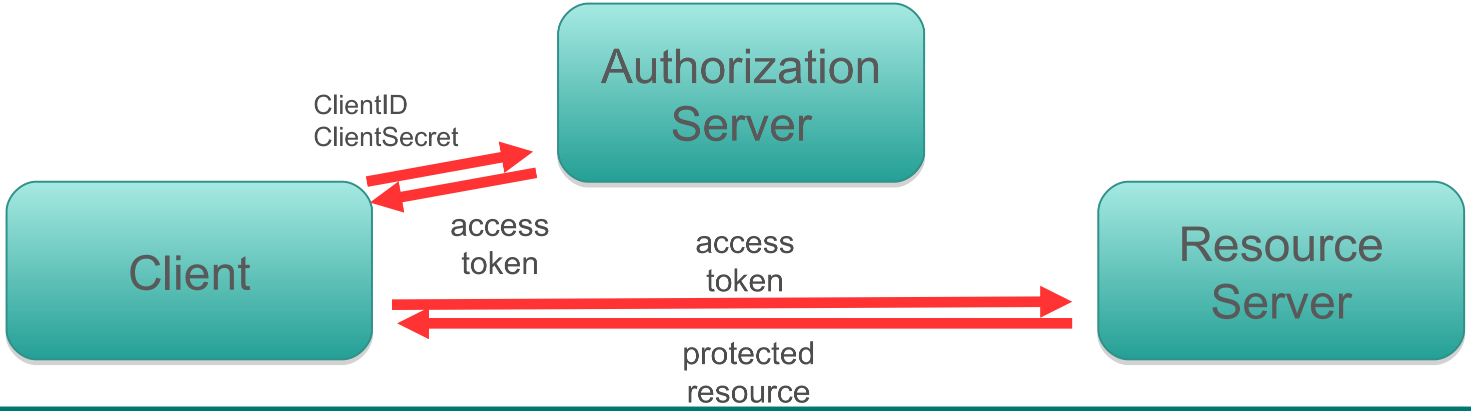
Single Page JavaScript App Flow (grant_type=implicit)

- For cases where the client can’t securely hold the refresh token and ClientSecret (e.g. JavaScript in the browser)
  - No refresh tokens or ClientSecrets are used
  - Access tokens are issued directly from the authorization endpoint- no authorization code is used
  - Only the ClientID and redirect_uri are sent when requesting tokens
  - The redirect_uri sent must match the redirect_uri sent when originally registering the client
- Simple but less secure- only one call to the authorization endpoint is necessary



Service-to-Service App Flow (grant_type=client_credentials)

- Also known as 2 legged OAuth
- For cases where the client is also the resource owner, or there is no resource owner
  - **User credentials not necessary**- client credentials are used for the authorization grant
- Client must register with the OAuth server, and receives a ClientID and a ClientSecret
- Client then sends the ClientID and ClientSecret to the token endpoint
- Client must securely keep the ClientID, ClientSecret, access token and refresh token
- No authorization endpoint or redirect_uri is used



Summary

- OAuth is an open standard framework for delegated resource authorization
- An authorization server acts as security intermediary, which is especially useful for distributed applications
- Tokens are used instead of credentials
- There are four application types commonly used, depending on the situation
  - Web App (grant_type=authorization_code)
  - Native Mobile App (grant_type=password)
  - Single Page JavaScript App (grant_type=implicit)
  - Service-to-Service App (grant_type=client_credentials)