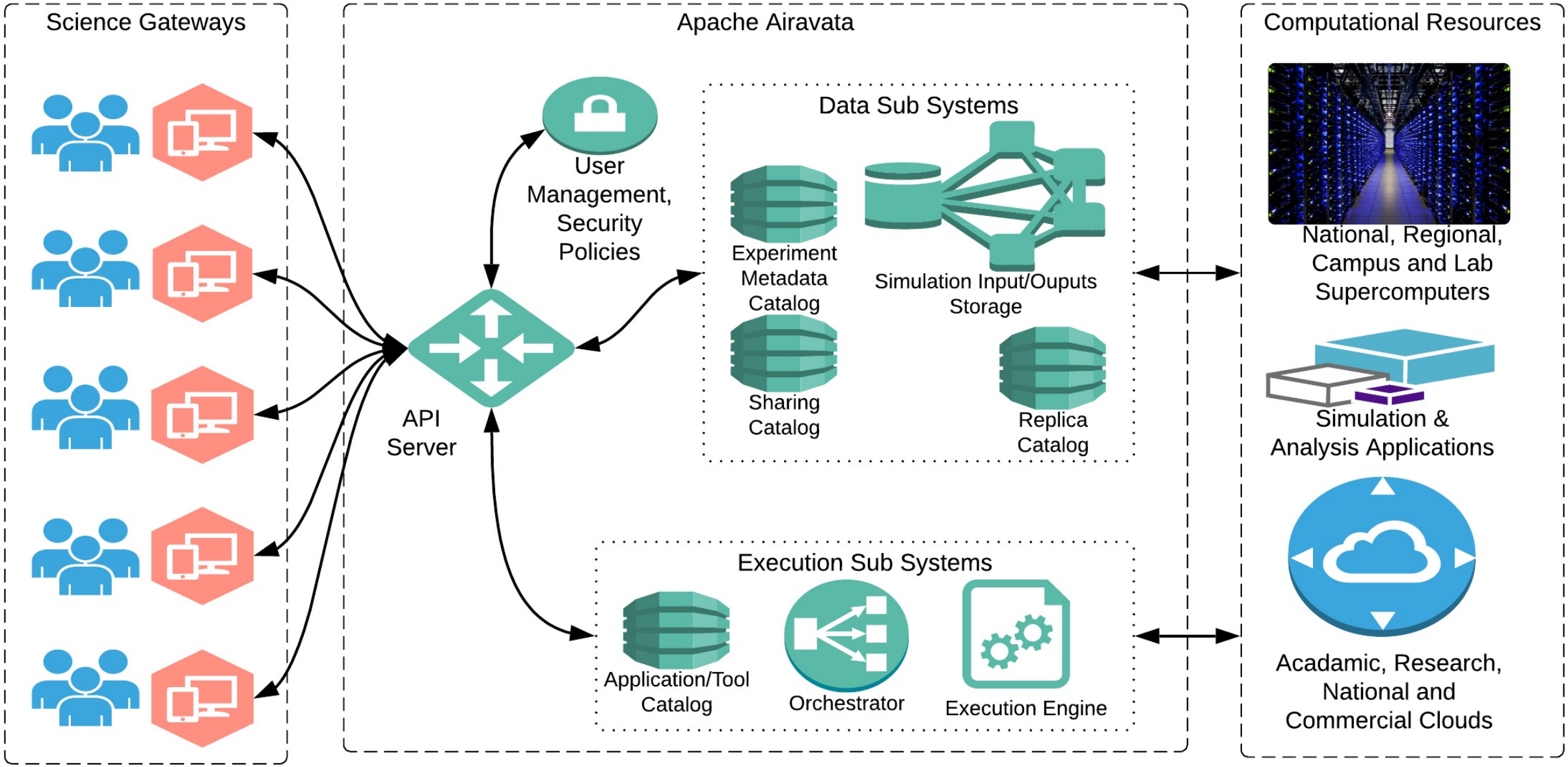
# Security Considerations for Science Gateways

Applications of OAuth2 and OpenID Connect

https://tools.ietf.org/html/rfc6749


Science Gateways
Apache Airavata
Computational Resources
User Management, Security Policies
API Server
Data Sub Systems
Experiment Metadata Catalog
Simulation Input/Ouputs Storage
Sharing Catalog
Replica Catalog
Execution Sub Systems
Application/Tool Catalog
Orchestrator
Execution Engine
National, Regional, Campus and Lab Supercomputers
Simulation & Analysis Applications
Acadamic, Research, National and Commercial Clouds

# Three Major Divisions of a Science Gateway Architecture: Three Security Issues

**Science gateway tenants**

- Domain specific: SEAGrid.org, SimVascular, etc
- Maintain their own user bases

**Science gateway middleware**

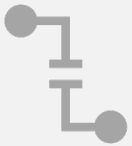
- Provides general purpose services that are used by gateway tenants.
- One middleware instance can support multiple science gateway tenants and multiple resources

**Science gateway resources**

- A gateway provider “rents” space on a resource
- Could be a supercomputer, Google Drive, AWS, etc

# Three Levels of Security Considerations

Security Between Tenants and Middleware



Security within Middleware (Byzantine Faults)

Security Between Middleware and Resource Layer: Direct and Brokered

# Custos: Services that Provide Three Types of Security

Authenticate users and manage their profiles.

Manage the “secrets” needed to access remote resources

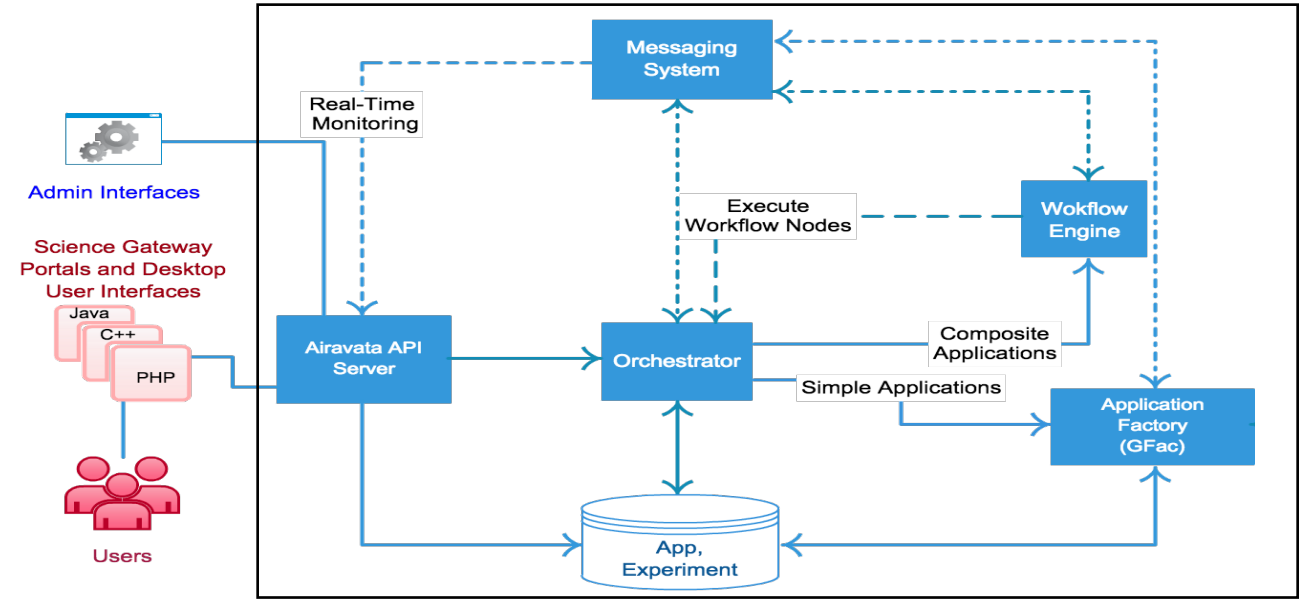
Manage access to session metadata created and managed by the gateway

# Simplifying Assumption: the Middleware Perimeter

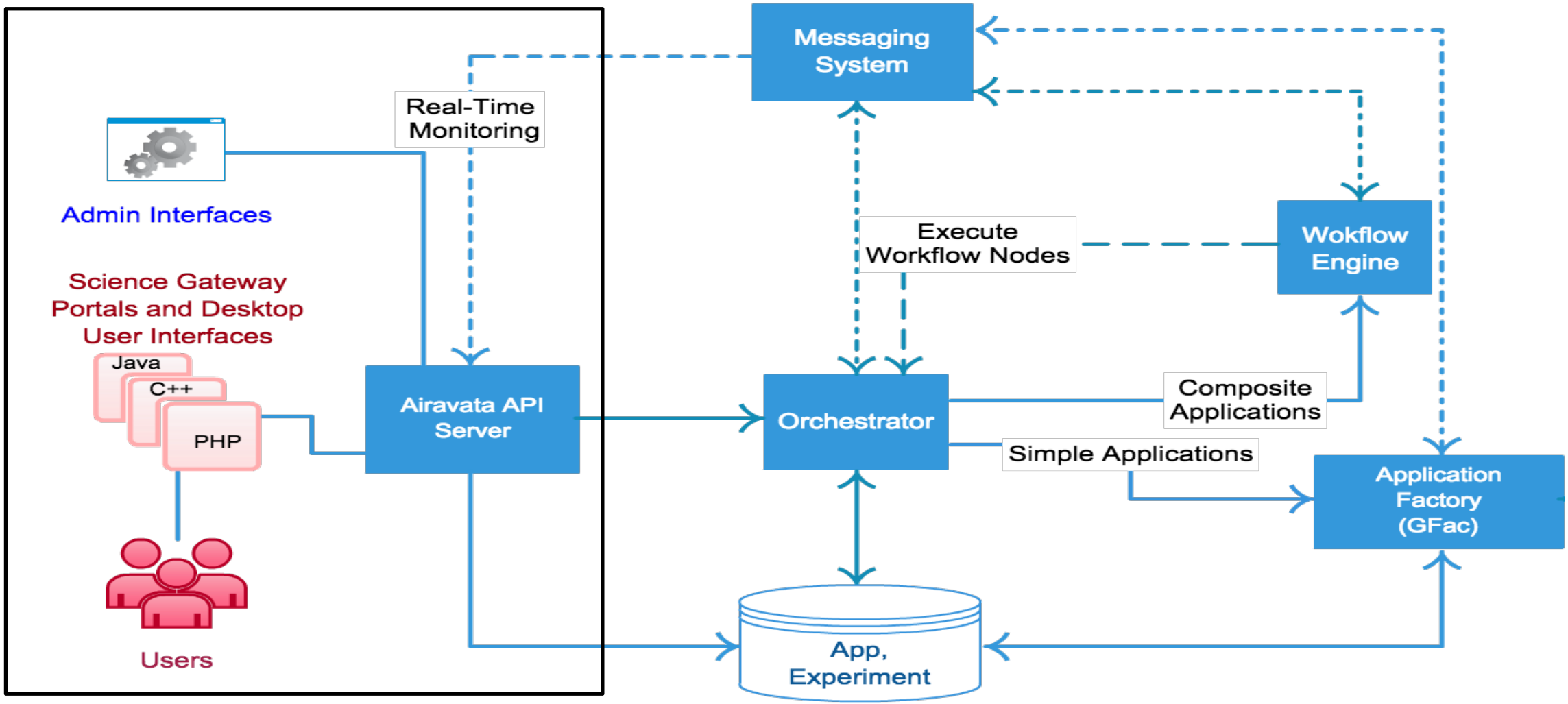
- Assume all the microservices run under a single administrative domain.
- Can use “operational security” rather than “architectural security”
  - Firewalls, closed networks and similar approaches to limit access to services to trusted entities.
  - Logging and intrusion detection
- Service Mesh solutions use TLS and mutual authentication within the perimeter

Some interesting further Microservice security considerations:

- Rogue services, Byzantine Fault Tolerance: RAFT
- Inter-Service Mesh Security
- Integrating trusted third party services like Box.



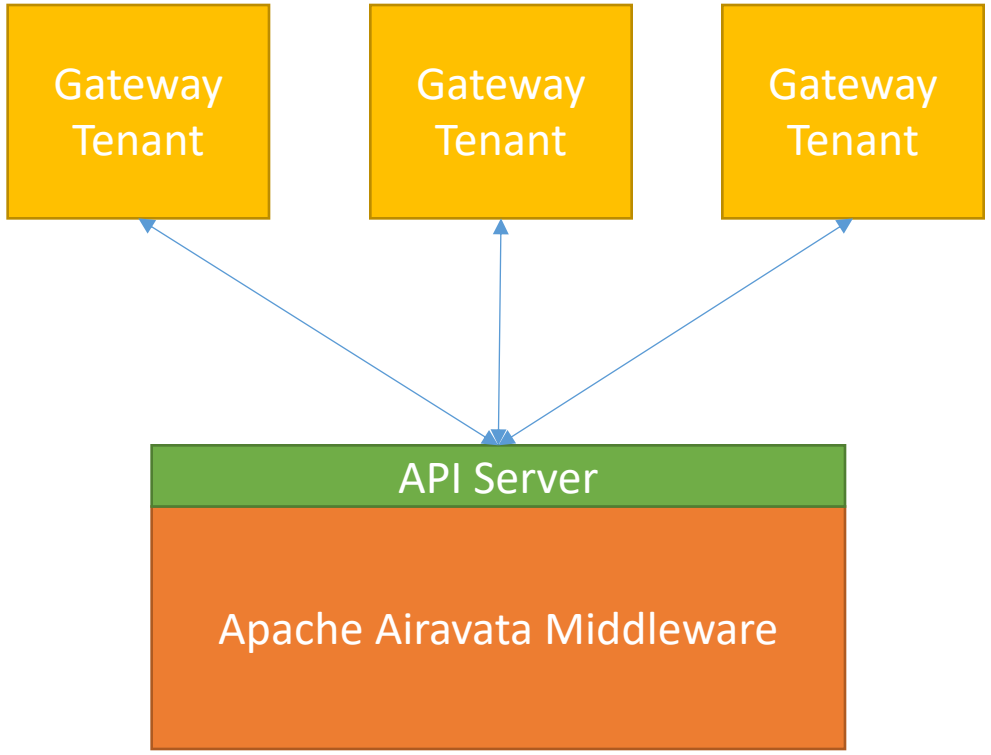
# Let's Look at the Tenant-Middleware Security




APACHE AIRAVATA

# Zoom in on the User Interface and API Servers

- UI: this is the gateway tenant
- The API Server can communicate with multiple tenants.
- Tenants can be Web servers, mobile applications, native browser JavaScript apps, or desktop applications.
- Tenants and the API server communicate over network connections (TCP or HTTPS)



# Security Challenges for Gateway Tenants

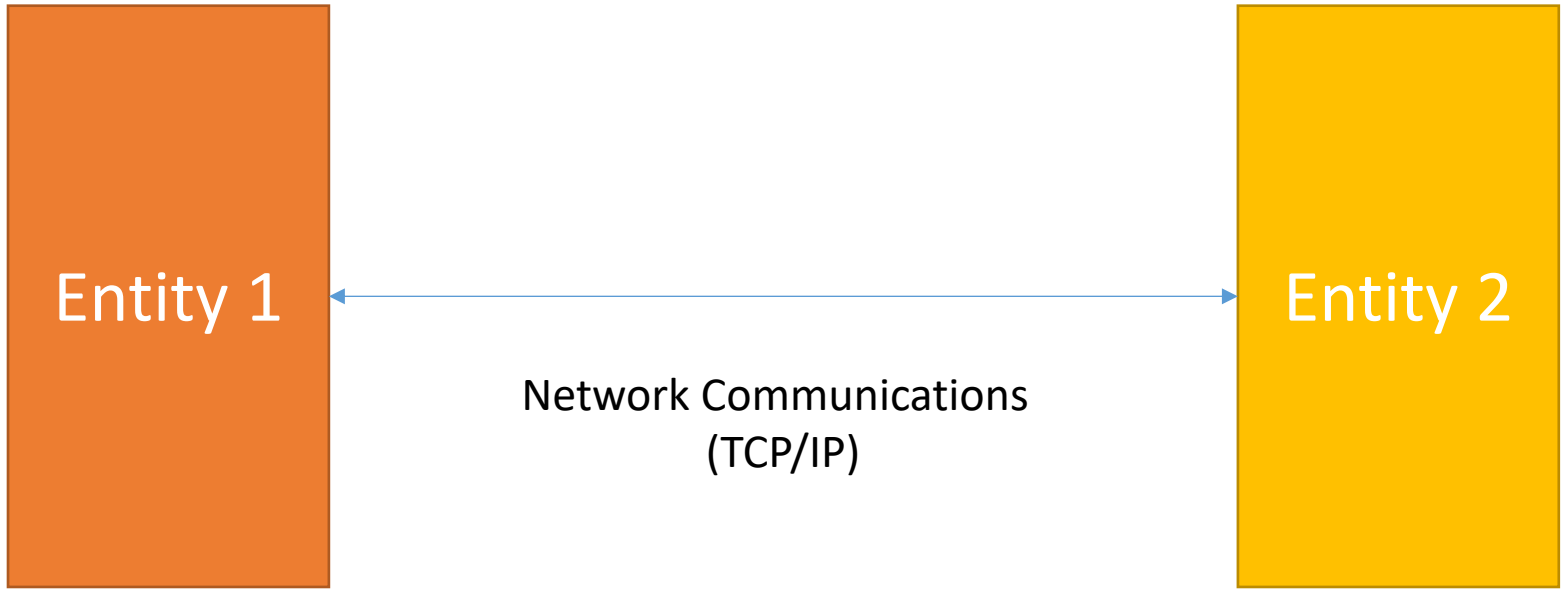
- Establishing trust between a gateway tenant and the API server.
- The gateway tenant may manage its own user base, but these must be communicated to the API server.
- A gateway tenant may be a single web server for an entire community (server-side PHP, Python, Java, etc)
- A gateway tenant also may be a desktop application, scripting tool, or in-browser application that get distributed to every user.

OAuth2 can address many of these issues.

# Network Security and OAuth2

A basic introduction

# Entities on a Network



# Security Concept 1: Entities

Entities have unique identities

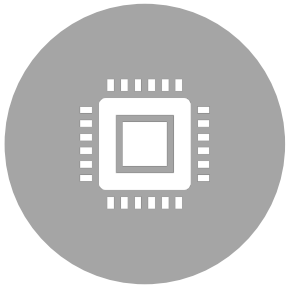
Entities can prove their identity: authentication

Entities can limit access of other entities based on identity: authorization

# Security Concept 2: Messages

Entities can verify that messages came from a specific authenticated entity. **Implemented with public/private keys**

Communications between entities can only be read by those entities. **Implemented with encryption, shared secret keys**



Detecting if the network message between entities has been altered. **Implemented with message digests (hashes).**

Each message between entities is unique. Avoids accidental or malicious replays. **Uses nonces, timestamps, etc.**

# The Authorization Problem

Trust Boundary

Client

Resource Owner

Resource Service


The **Resource Owner** wants to authorize the **Client** to act on **Resource Service** on the Resource Owner's behalf. How do you do delegate this authority?

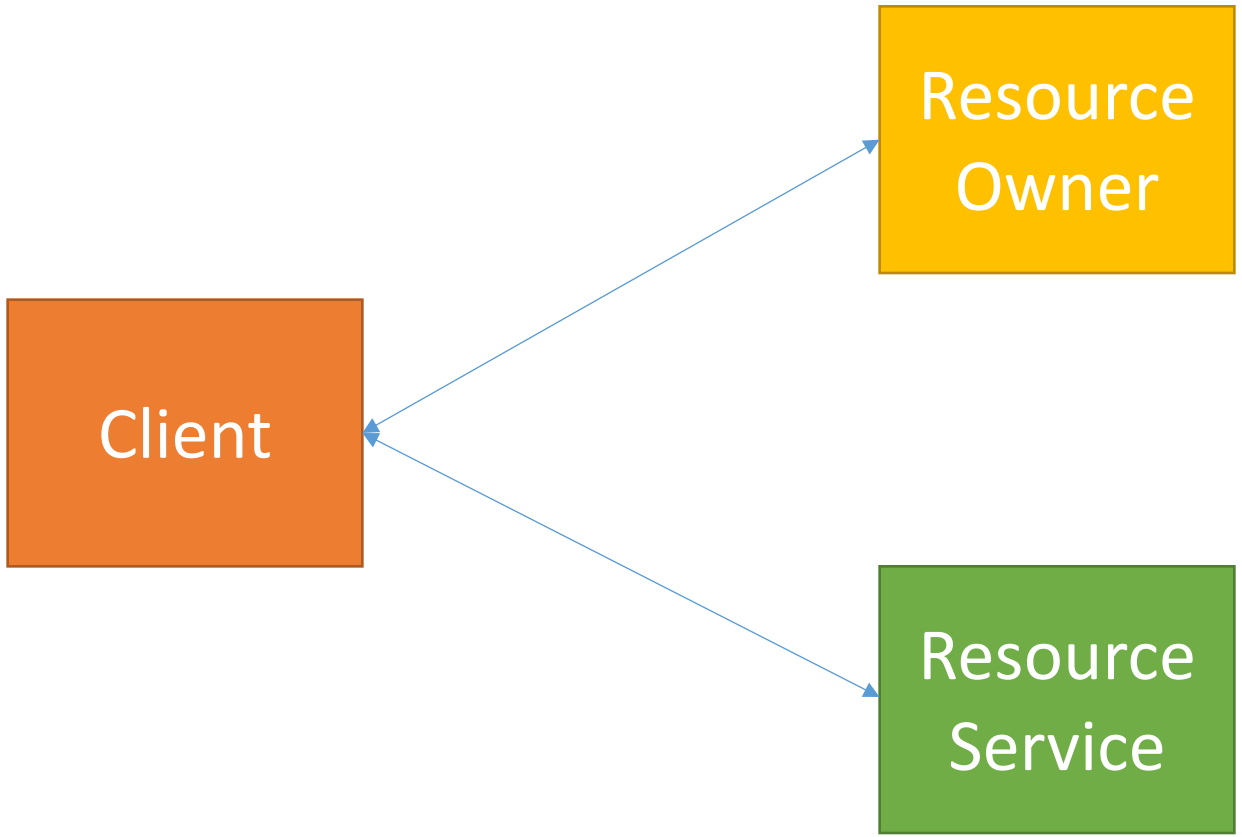
# Authorization and 3rd Party Services

- This is a pervasive problem
- Platforms (Facebook, Google) and devices (phones) hold your personal data (resources).
- Third party applications need to access some of this data.
- You decide which applications to authorize
  - “Facebook, it is ok for this application to access the names of my Facebook friends and other personal information.”
  - “IPhone, it is OK for this app to know my location”

I am the Resource Owner. My list of friends, personal information, and location are accessible through a Resource Service. Facebook and IPhone apps are Clients.

# Problems Delegating Authority

- Straightforward Approach: Client requests an access-restricted resource by authenticating **using the resource owner's credentials**, like passwords
  - The Resource Owner shares its credentials with the third party Client.
  - The Client impersonates the Resource Owner.
- This is a really bad solution
  - What are some problems with this approach?



# Some Problems with Credential Sharing

Third-party applications gain full, permanent access to the Resource Owner's protected resources.

Resource Owners cannot revoke access to specific clients without revoking access to all clients (must change passwords)

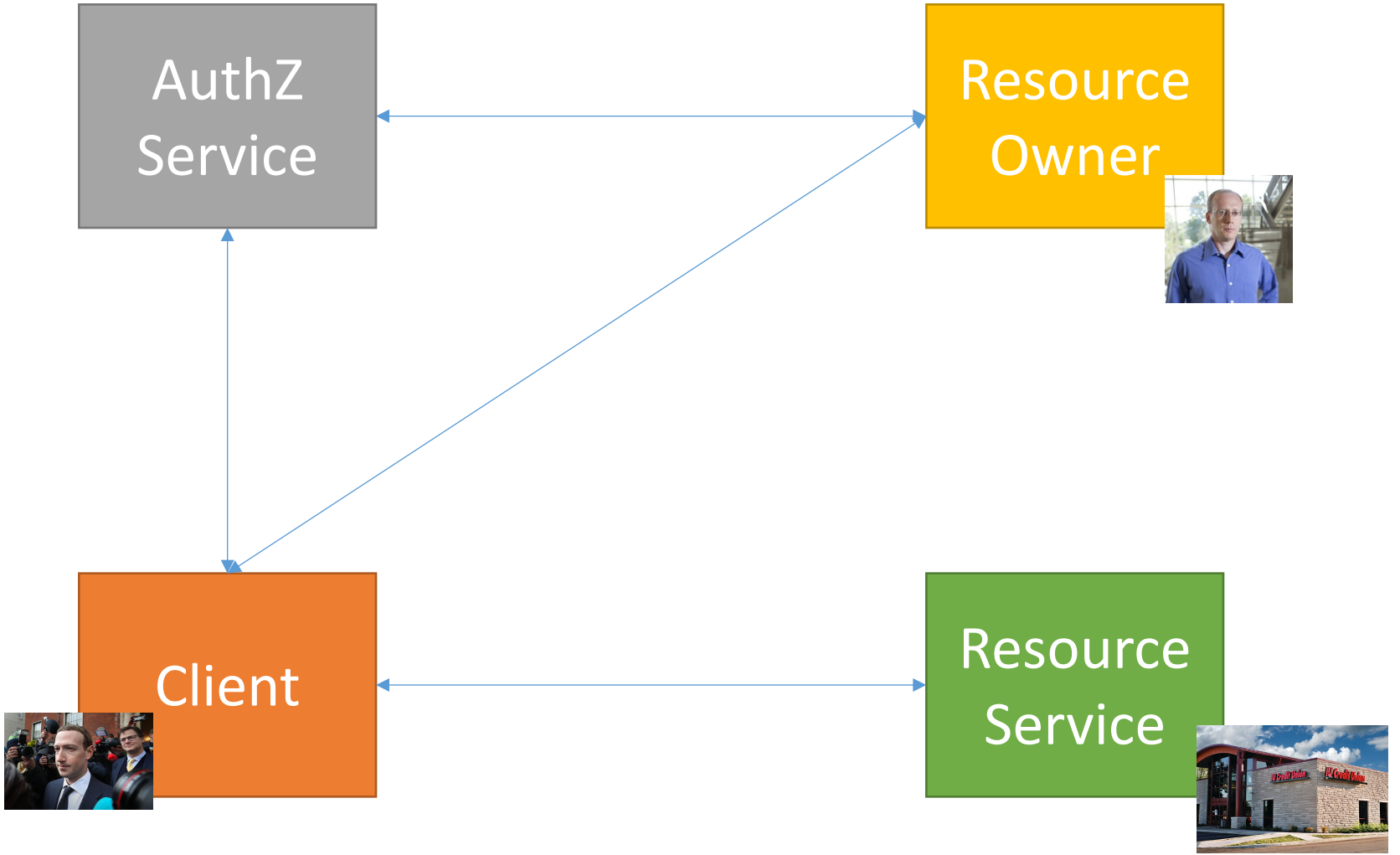
Compromise of the client results in compromise of the end-user's long-term credentials and all the data protected by that password.

Compromise of one client compromises all the clients and all the Resource Services.

# Introducing OAuth2

OAuth2 solves this problem by introducing a **mutually trusted*** Authorization Service

*There are rigorous ways, like key exchanges, for establishing mutual trust.



# OAuth2 Main Concepts

- Introduces an authorization layer
- Separates the role of the client from that of the resource owner.
- Client is issued a different set of credentials (OAuth2 access tokens) than those of the resource owner (passwords)
- An OAuth2 access token has a specific scope, lifetime, and other access attributes.
  - These limit what the Client can do and how long the Client's requests are valid
- Access tokens are issued to third-party clients by an Authorization Server with the approval of the Resource Owner.
- The Client uses the access token to access the protected resources hosted by the Resource Server.

# Credentials vs. Tokens

**Resource Owner Credentials**

- Can be used by the client to do anything the user can do
- Don’t expire
- Resource Owner must manually change them
- All Clients use the same credentials for a specific Resource Owner

**Access Tokens**

- Can be associated with specific, limited operations
  - Read but not write, for example
- Have a specific lifetime
- Generated by the Authorization Server, not a human
- Each Client has a different token

# Types of OAuth2 Clients

| Client Type | Description |
| --- | --- |
| Web Application | Confidential client that runs on a Web server. Client credentials and access tokens are stored on a Web server. |
| Native Applications | Public client that runs on a device used by the Resource Owner. Client credentials and access tokens are stored on the device. Examples: mobile devices and desktop clients |
| User Agent Applications | Public client code is downloaded from a server and runs on the user's Web browser. Client credentials and access tokens are stored in the user's browser. |

These clients have different security implications

# Client Registration: Trusting the Client

- Clients register with the Authorization Server
  - This is a one-time operation.
- The Client can be either *confidential* or *public*
  - Confidential: a web server-based Client, for example
  - Public: Browser, desktop, or mobile clients
- The Authorization Server issues a client identifier to the Client
  - Unique string representing the information provided by the client.
- Confidential Clients authenticate to the Authorization Server
  - Passwords, key pairs, secrets, etc.

# OAuth2's Abstract Protocol Flow

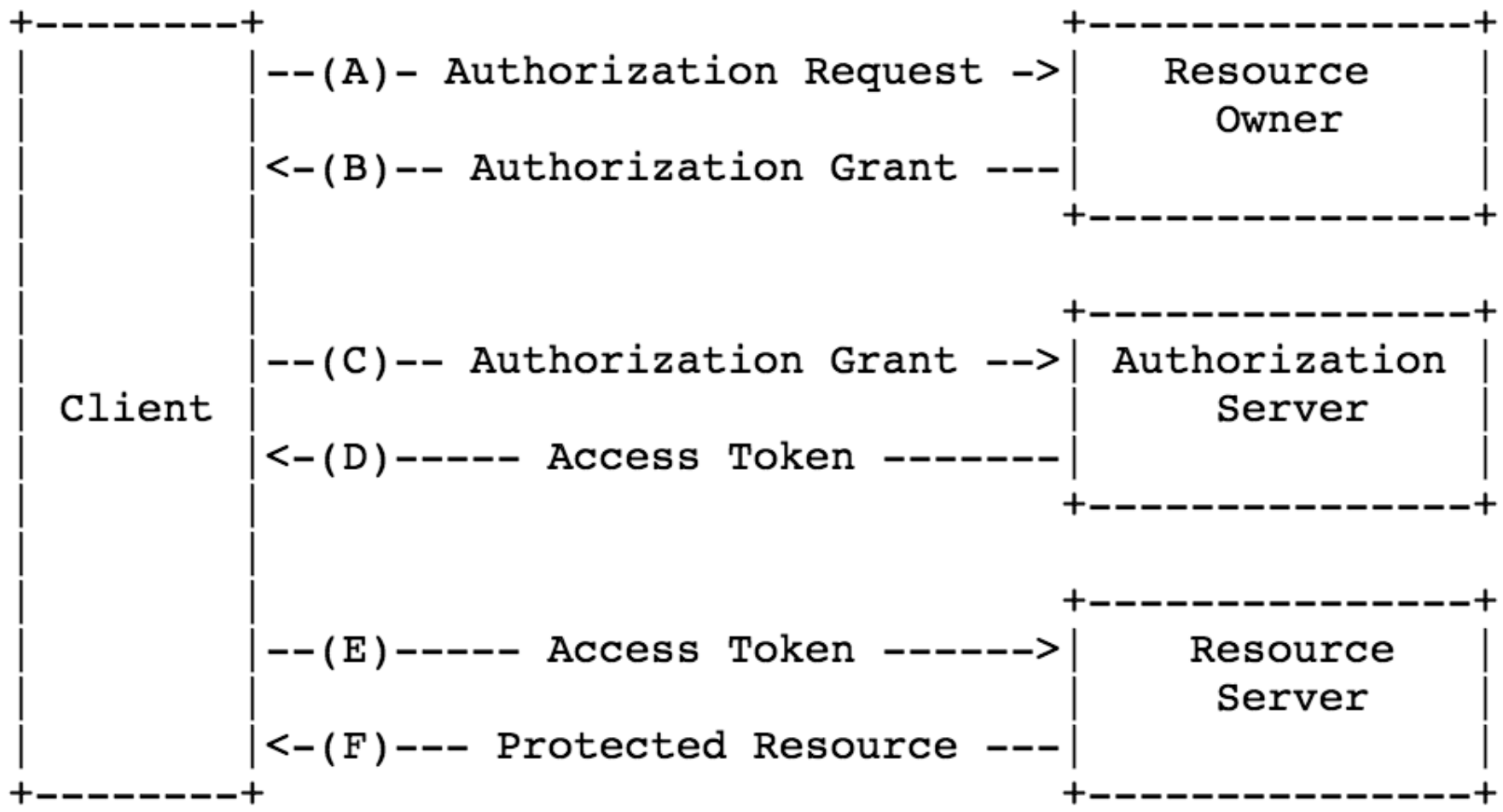



Figure 1: Abstract Protocol Flow

# OAuth2 In Brief...

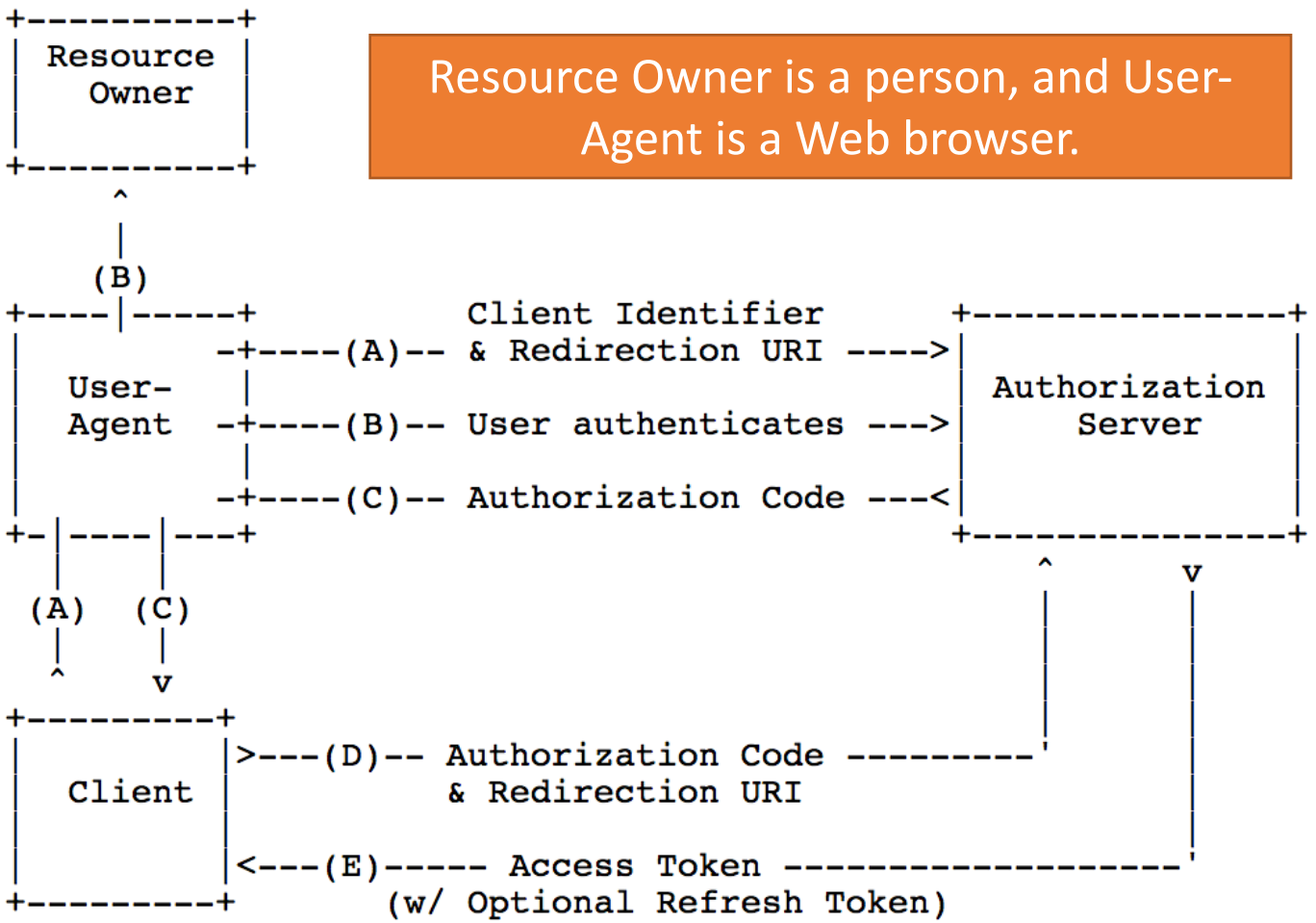
- The Resource Owner issues a **grant** to the client.
  - The grant usually comes from the Authorization Service
- The Client uses the grant to get an **access token** from Authorization Service.
- The Client uses the access token to make requests from the Resource Service until the access token expires.

OAuth2 has several **grant types** that are appropriate for different scenarios.

# Authorization Code Grant Type for Private Clients

- The Client is a server-side application
- The Resource Owner attempts to use the Client to access a Resource Server (not shown in figure)
- When complete, the Client can use the Access Token to access the Resource Server.

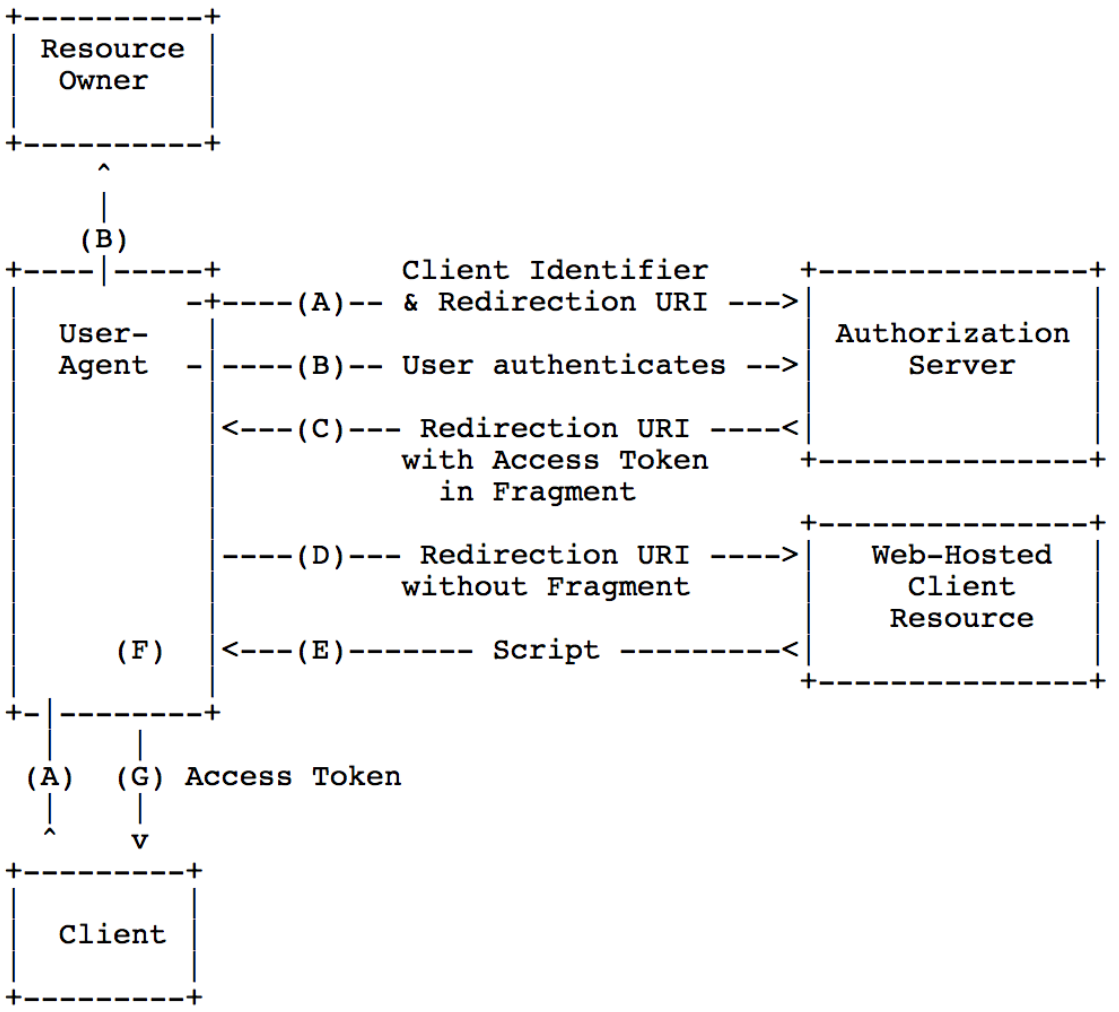
This is the most common grant type.




Resource Owner is a person, and User-Agent is a Web browser.

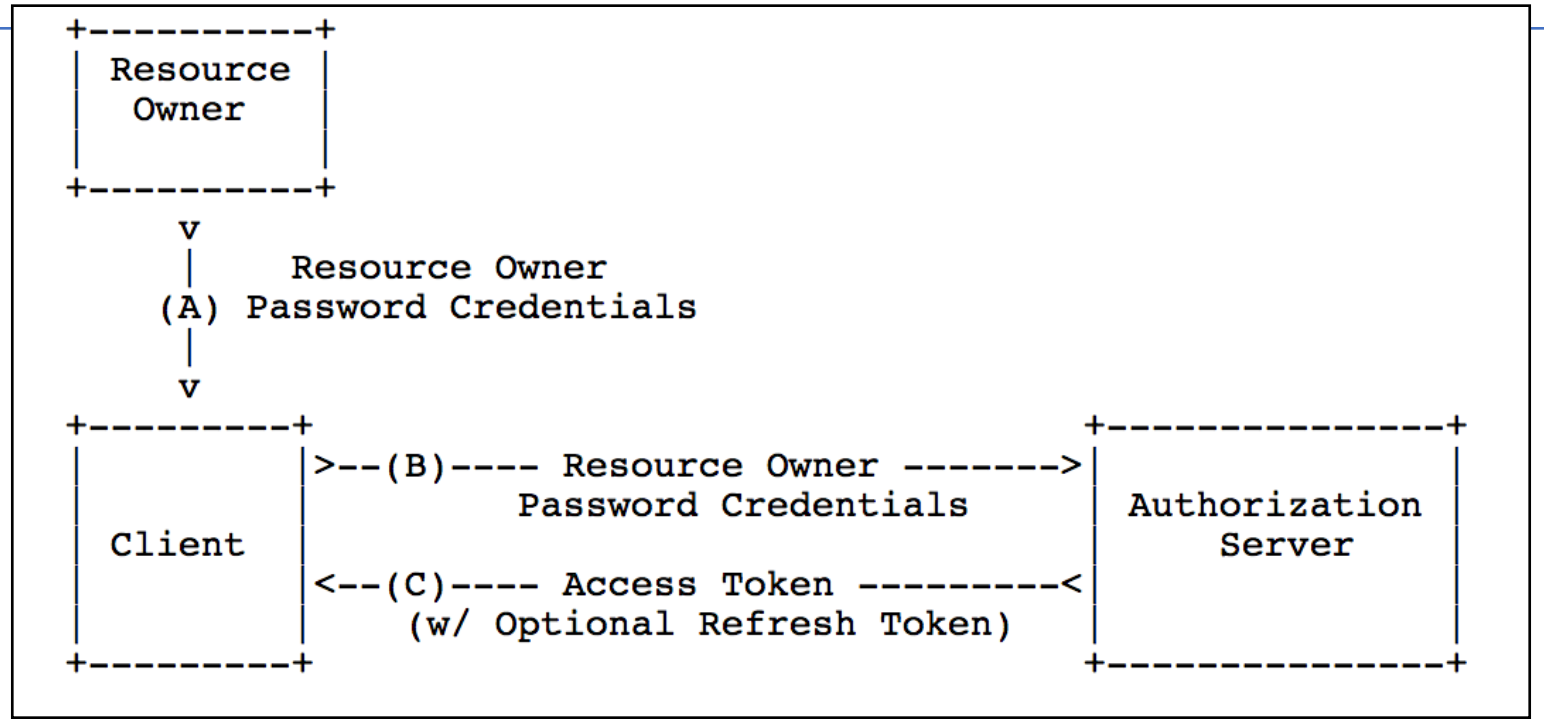
# Implicit Grant Type for Public Clients

- Authorization flow suitable for Clients that run as JavaScript applications in the user's browser.
- Client gets the access token directly in a redirect URL, skipping the authorization code step.
- Convenient but less secure.



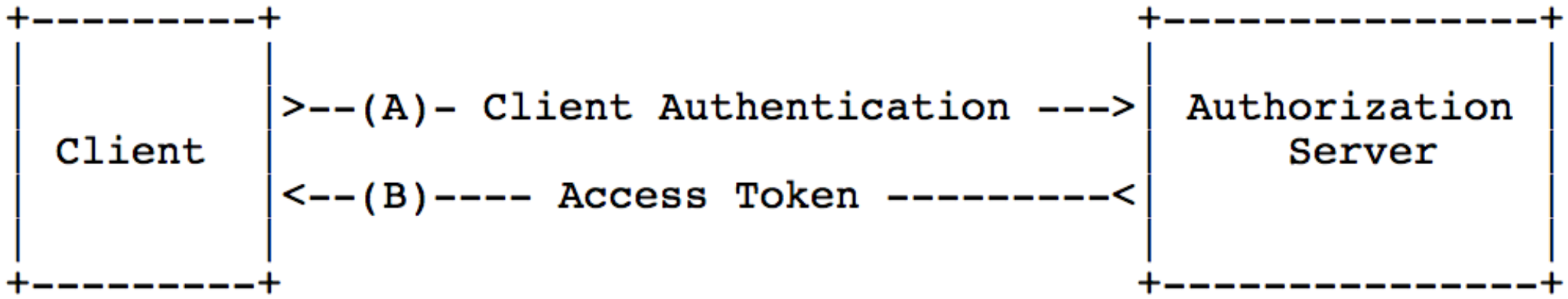
# Resource Owner Password Credentials

- Resource Owner gives the Client its full credentials.
- Client uses these to obtain an access token and possibly refresh tokens.
- Owner must trust the Client, and Client can use the credentials only once per access token.
- Best way to authorize desktop applications?



# Client Credentials Grant Type

- Client and Resource Server are owned by the same entity, or Client and Resource Owner are the same.
- Ex:  Facebook's internal services only access your personal data if you authorize them.
- Machine-to-machine, no human in the loop
- You could use this between microservices within your perimeter



# What Are Access Tokens?

- These may be identifiers (“kdjk-111-dkjfkljd-0kdkj-kwjlej”) meaningful only to the Resource Server
- Or they may be structured and meaningful
  - JSON Web Tokens
  - OpenID Connect Tokens (shown)
  - SAML
- The Client may not understand or even decrypt the token

```
{
 "iss": "https://server.example.com",
 "sub": "24400320",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "auth_time": 1311280969,
 "acr": "urn:mace:incommon:iap:silver"
}
```

# Refresh Tokens

Access tokens should expire in order to limit their potential misuse.

Refresh tokens are used to obtain new access tokens after the access token has expired.

Issued to the Client by the Authorization Server when the Access Token is issued.

Refresh tokens are optional

# OpenID Connect: A Summary

An OAuth2-Based Authentication Protocol

http://openid.net/connect/

# What Is OpenID Connect?

- Authentication as a Service
  - Don’t run your own authentication service
  - Use a trusted service instead
  - Authentication mechanisms and details handled by the service.

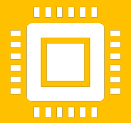
# Why Use OpenID Connect?

- Trusted Identity Providers (IdPs) absorb lots of headaches
  - Follow best practices and implementations for securing user accounts and information.
  - Avoid the need to provide separate identity management for every application
  - Handle federated identities.
  - Handle advanced authentication mechanisms such as two-factor authentication

# Examples

CAS: not OpenID Connect based, but similar

CILogon: a service from University of Illinois that provides federated identity



Keycloak: Open source software for running your own IdP.

Google, Microsoft, Amazon, Auth0, and others run OpenID Connect services for you.

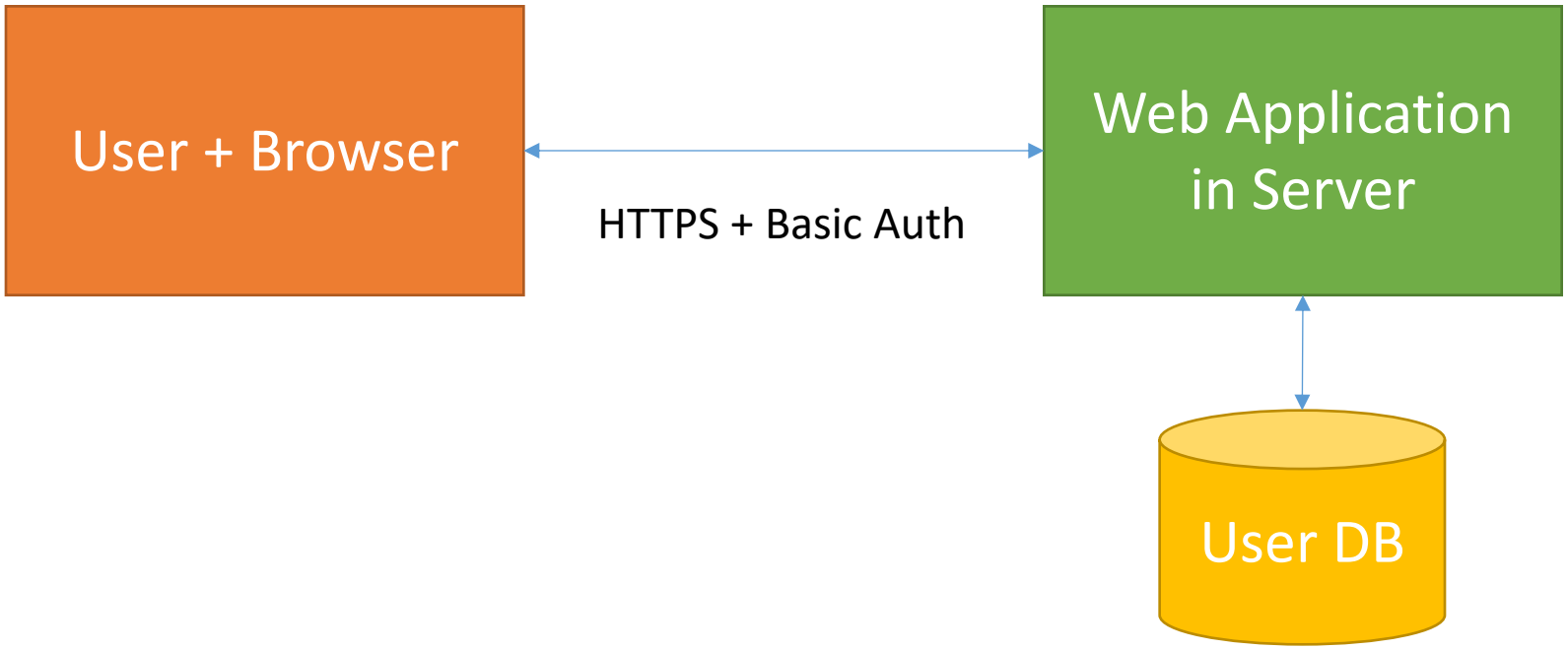
# OAuth2 and OpenID Connect

OAuth2 is used to authorize clients to access resources using access tokens.

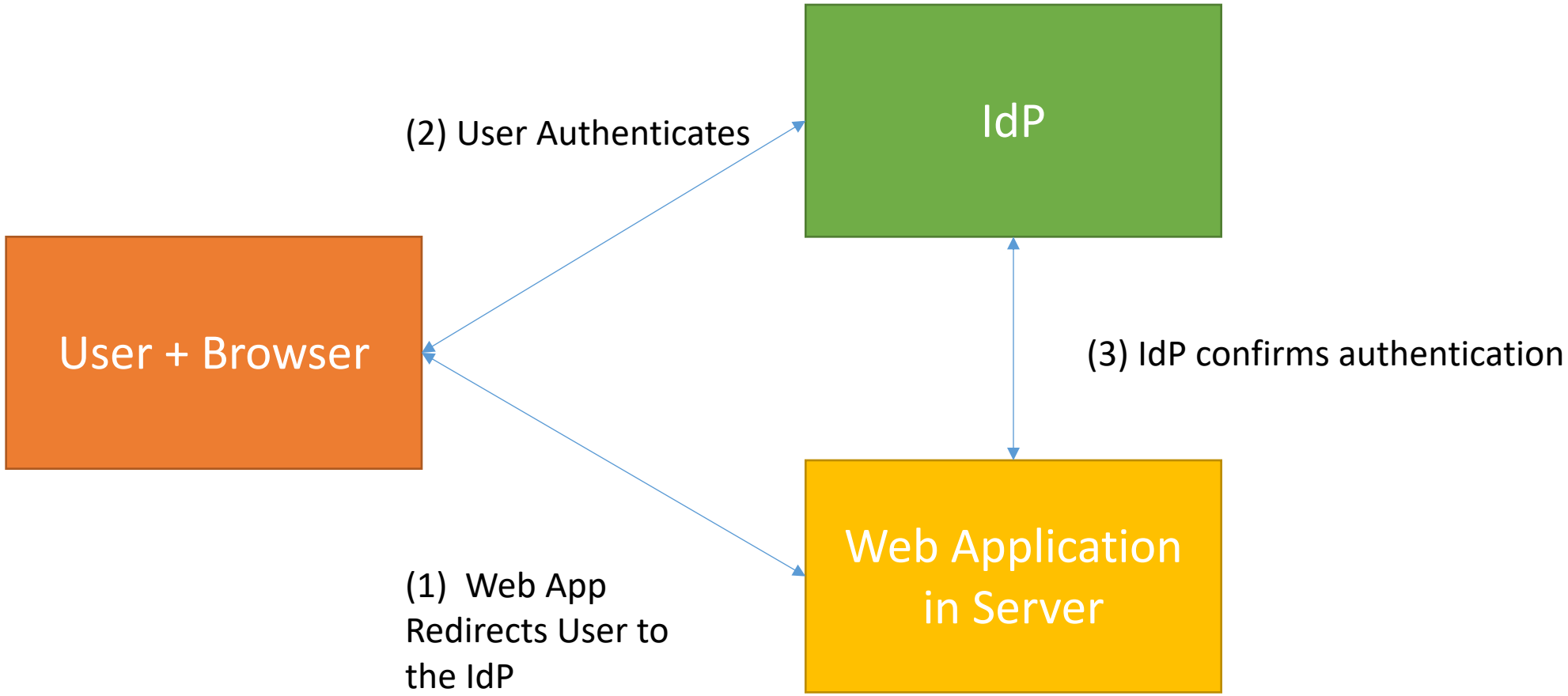
OpenID Connect uses the same ideas to authenticate users before they can access services.

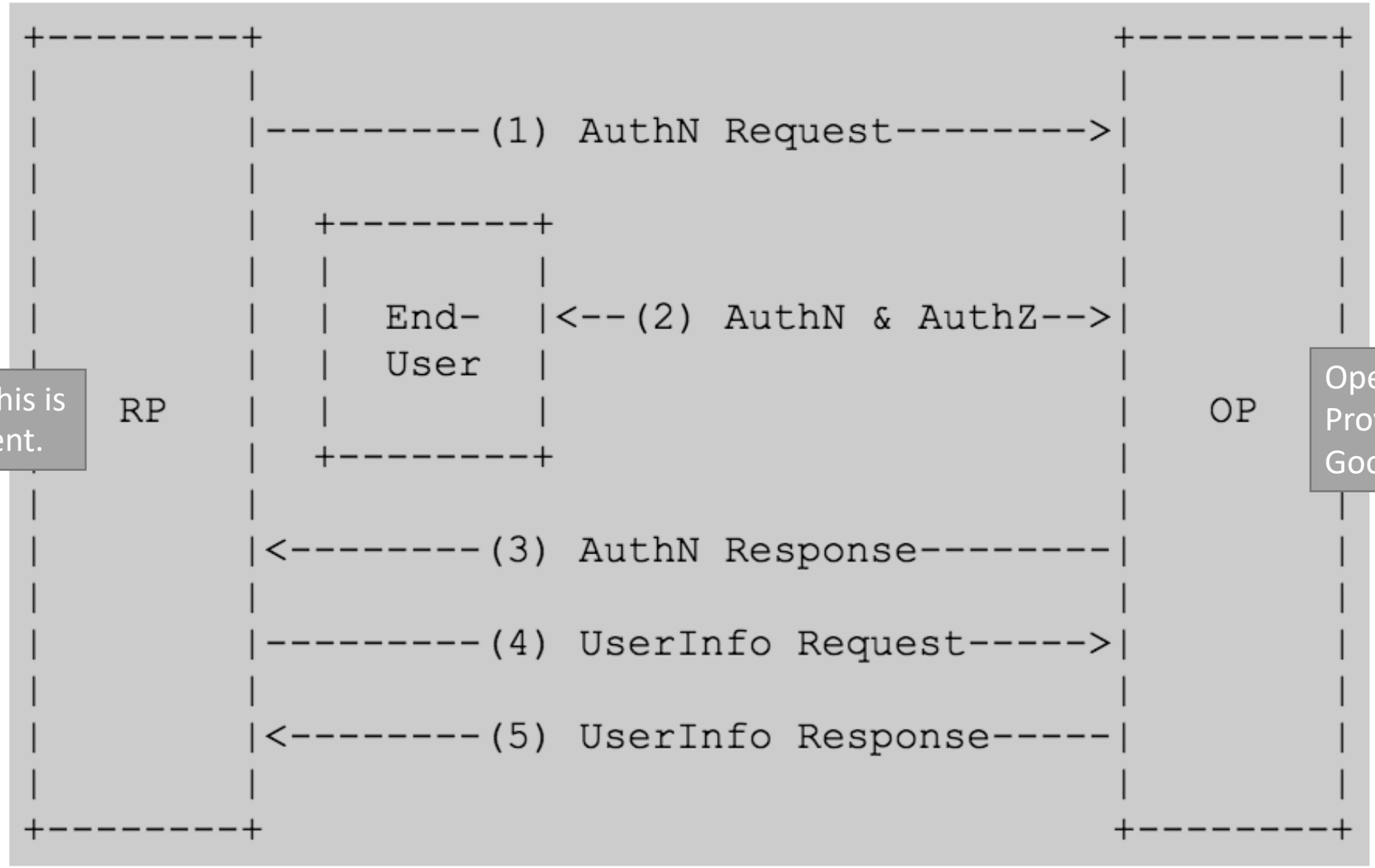
Clients can also obtain basic profile information about the user in an interoperable and REST-like manner.

# Direct Authentication



# Authentication as a Service






Relying Party. This is the OAuth2 Client.

OpenID Connect Provider (i.e., Google)

# Basic OIDC Flow

# Basic OIDC Steps

- The Relying Party (RP) sends a request to the OpenID Provider (OP).
  - This is the science gateway
- The OP authenticates the End-User and obtains authorization.
- The OP responds with an ID Token and usually an Access Token.
  - Verifies to the client that the user authenticated correctly.
  - The ID Token is specific to OIDC and is its primary extension of OAuth2
- The RP can send a request with the Access Token to the UserInfo Endpoint.
- The UserInfo Endpoint returns Claims about the End-User.

We can make use of the Access Tokens for other authorization decisions.

# OAuth2, OIDC, and Science Gateways

- We treat science gateway tenants as OAuth2 clients
- Tenants thus need to authenticate users and obtain OAuth2 access tokens.
- Tenants present access tokens to the API server for basic access levels.
- Finer grained decisions can be made using access control mechanisms, groups, etc.
- Custos represents our current implementation, originating in 2014-2015 GSOC projects

Nakandala, S., Gunasinghe, H., Marru, S. and Pierce, M., 2016, October. Apache Airavata security manager: Authentication and authorization implementations for a multi-tenant escience framework. In *2016 IEEE 12th International Conference on e-Science (e-Science)* (pp. 287-292). IEEE.