OAuth + OIDC for Rust Developers

•••

Standards and Best Practices

Thomas Darimont

About me

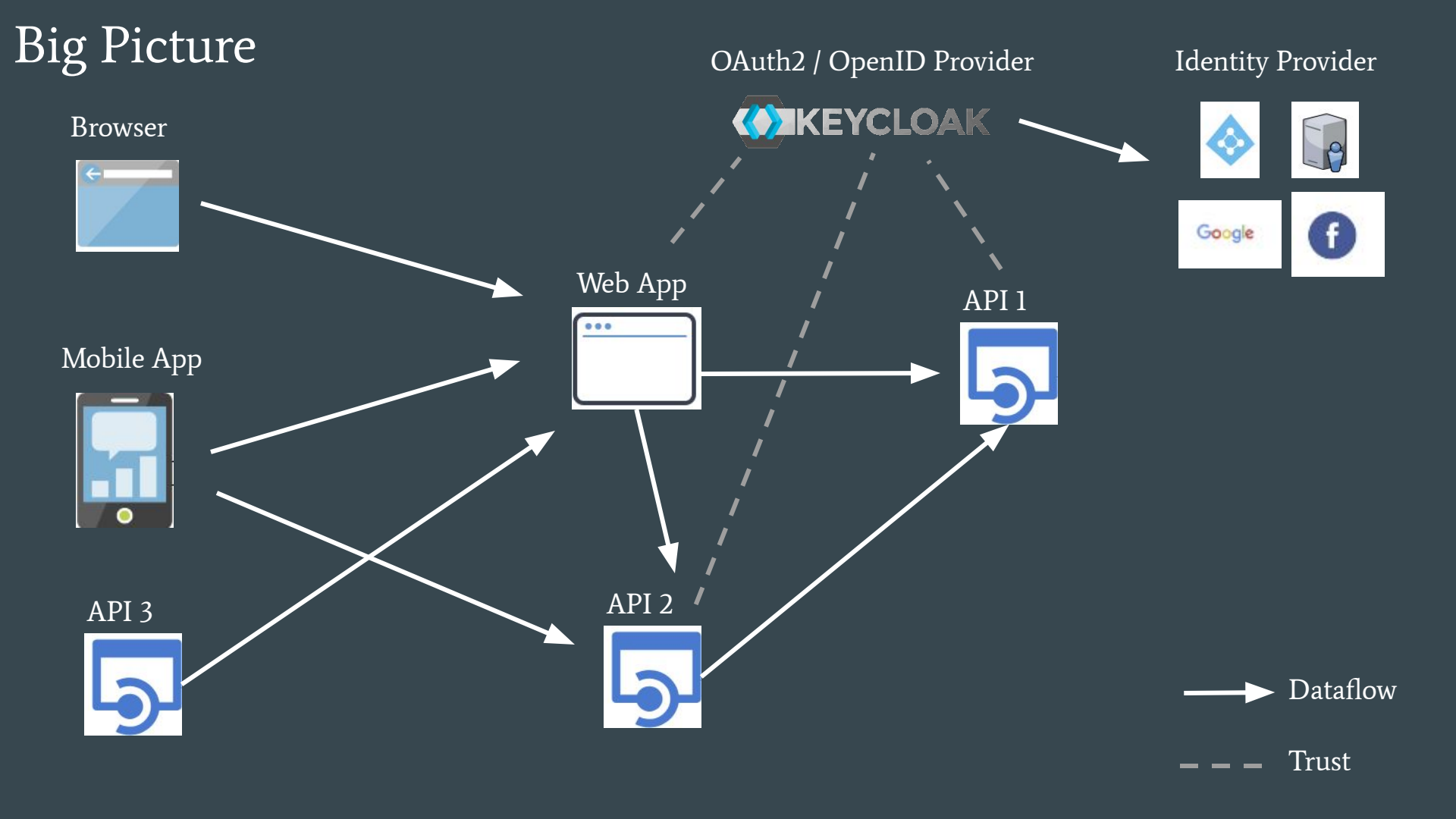
- Thomas Darimont
- Principal Consultant codecentric
- Java Head and Rust Novice
- Open Source Enthusiast
- Official Keycloak Maintainer
- Extism WASM Java SDK Maintainer
- Java User Group Saarland Organizer

KEYCLOAK

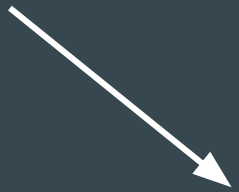
@thomasdarimont
@jugsaar
t.darimont@codecentric.de

Problem

Big Picture


OAuth2 / OpenID Provider
KEYCLOAK
Identity Provider
Browser
Google
Web App
API 1
Mobile App
API 3
API 2
Dataflow
Trust

Authentication & Authorization?



Who is the User?

What can the User do?

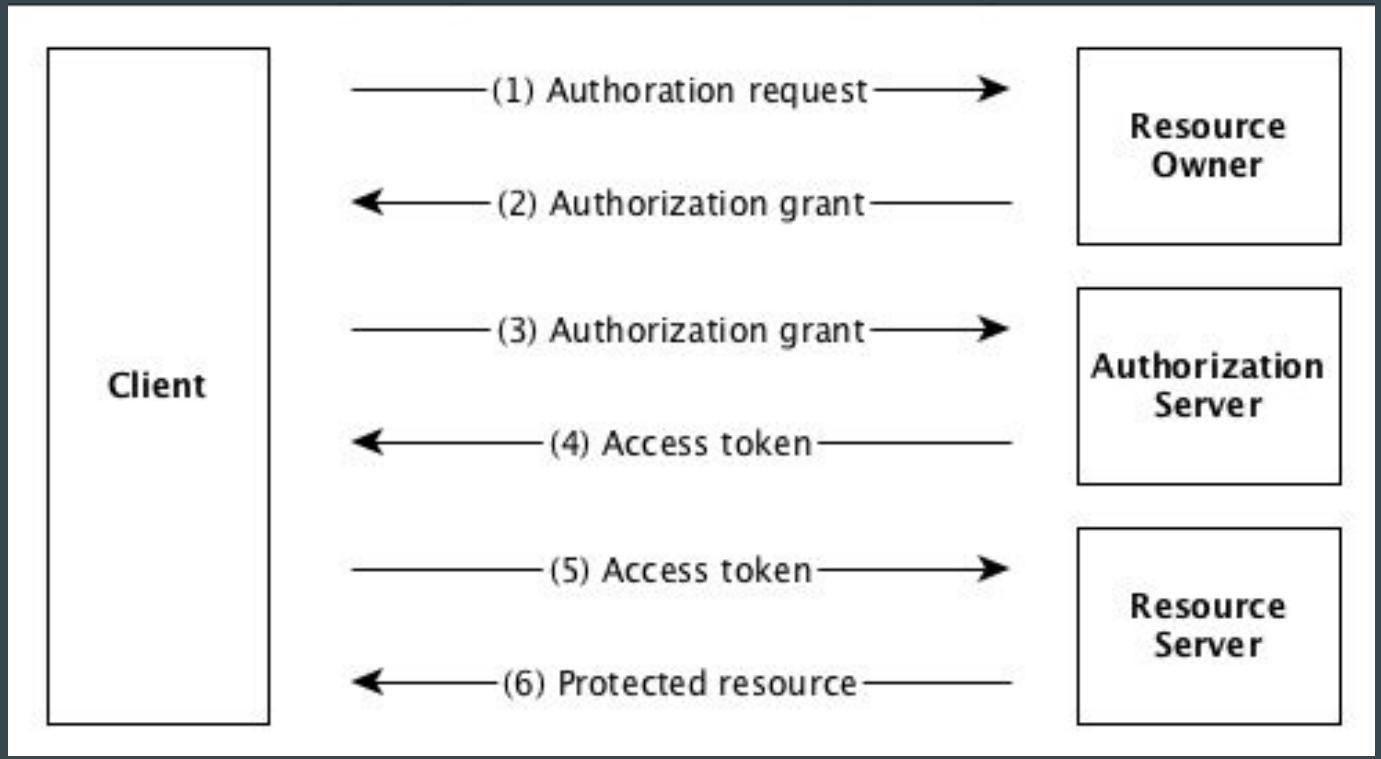
OAuth

OAuth 2.0

- **Framework** for building **Delegated Authorization Granting** Protocols
- Widespread **industry standard governed** by **IETF** OAuth WG
- **OAuth 2.0** collection of specifications and extensions
- **OAuth 2.1** in-progress effort to **consolidate & simplify OAuth 2.0** features
  - Omitted “deprecated” flows
  - Simplified terms
  - Secure recommendations
- **Flows** define **Protocol interactions** between involved parties / **Roles**
- **Threat Model** and **Security Best Current Practice**

Roles in OAuth 2.0

APP



USER

STS
(Secure Token Service)

API

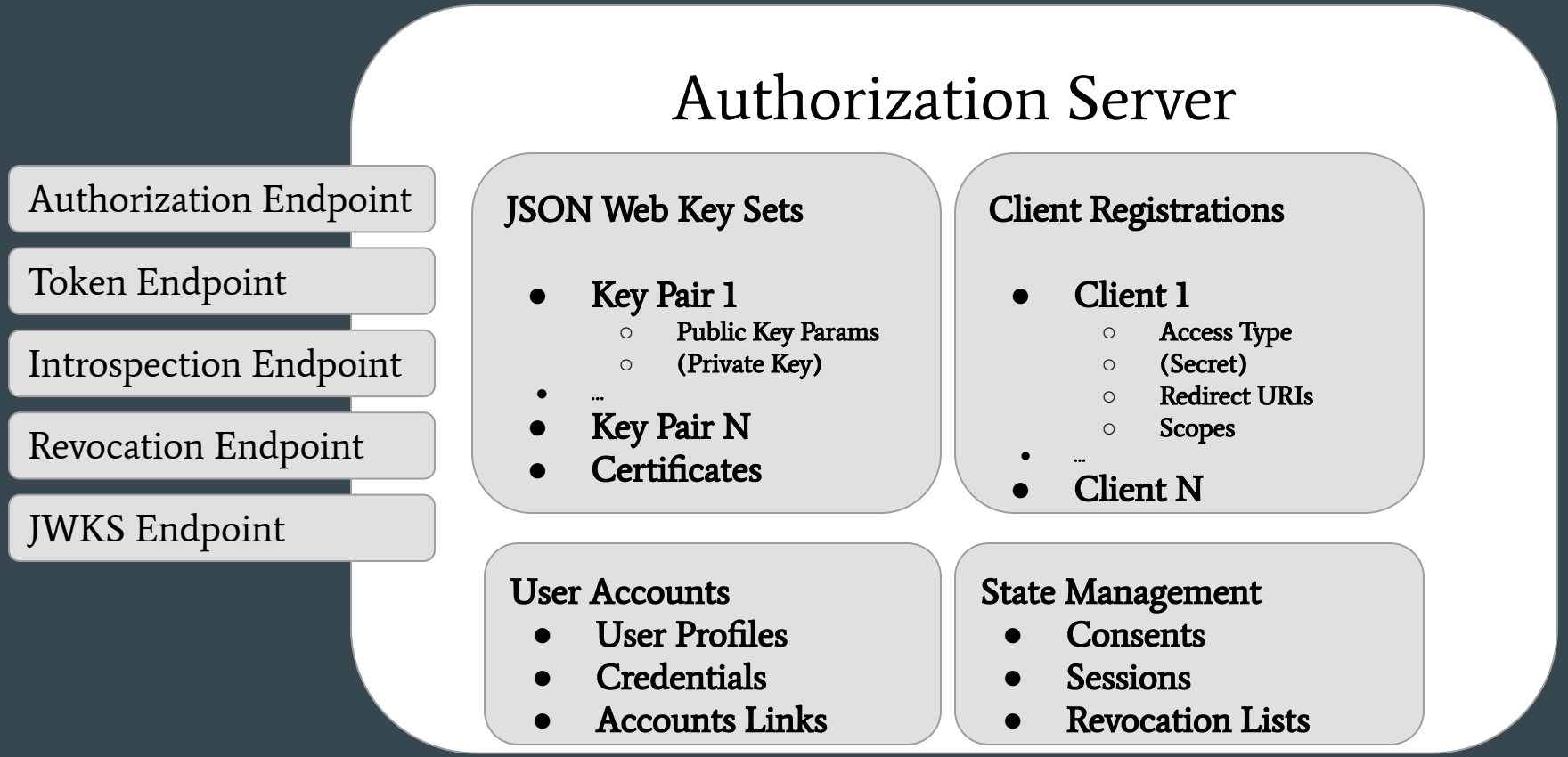
Roles in OAuth 2.0 contd.

- **Resource Owner (RO)**
  - **User** or **system** that **owns** the protected **resources** and can **grant access**
- **Client**
  - **Application / System** that **requires access** to the protected **resources**
  - **To access resources**, the Client must hold the appropriate **Access Token**
  - **Access Type**: **Public** or **Confidential** (can use a secret for additional authentication)
- **Authorization Server (AS)**
  - **Receives Access Tokens requests** from Client
  - **Issues Access Tokens** upon successful **authentication** and **consent** by the **Resource Owner**
- **Resource Server (RS)**
  - **Protects** the **user's resources** and **receives access requests** from Client
  - **Accepts** and **validates** an **Access Token** from Client and **returns appropriate resources**

OAuth 2 Endpoints

- **Authorization Endpoint**
  - Used by resource owner for obtaining an authorization grant
- **Token Endpoint**
  - Issues Tokens, like Access-Tokens, Refresh-Tokens etc.
- **Token Introspection Endpoint**
  - Allows to validate tokens sent in request
- **Token Revocation Endpoint**
  - Allows to invalidate tokens

Anatomy of a typical OAuth Authorization Server



OAuth 2.0 Flows and Grant Types

- Defines **Protocol interactions** between **involved parties**
  - OAuth 2.0 defines a set of generic flows
  - Flows can be controlled via parameters, e.g. grant_type / response_type
- **Grant types** define **means** to **obtain Tokens**
  - Authorization Code
    - Extension PKCE (Proof Key for Code Exchange)
  - Client Credentials
  - Device Code
  - Refresh Token

OAuth 2 Flows

- **Authorization Code Grant Flow**
  - “Standard Flow” for browser based authorization
- **Client Credentials Grant Flow**
  - Used for service to service communication
- **Implicit Grant Flow**
  - Simplified Version of Authorization Code Grant Flow
- **Resource Owner Password Grant Flow**
  - Used for integration with legacy applications

Omitted in **OAuth 2.1** and **no longer recommended**

OAuth 2.1 Consolidated Flows

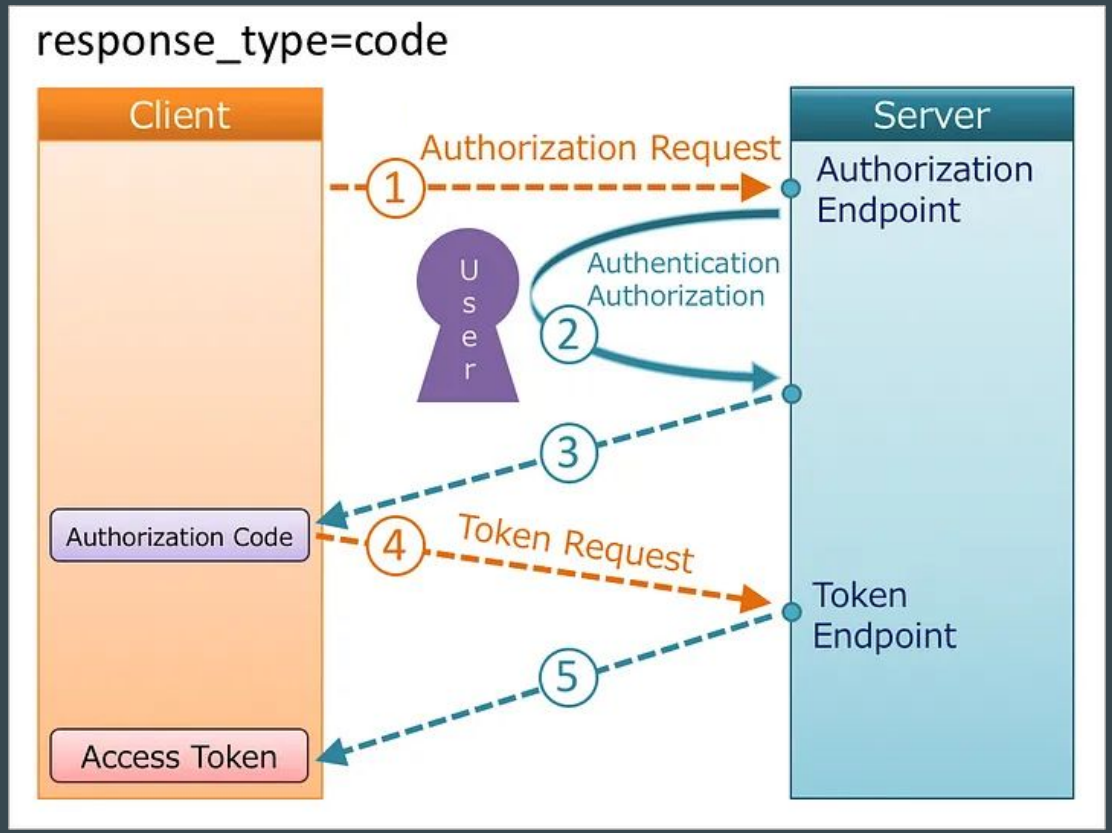
- **Authorization Code Grant Flow**
  - “Standard Flow” for browser based authorization
  - + Proof Key for Code Exchange (PKCE) aka “Pixi”
  - PKCE mitigates Authorization Code interception attack
- **Client Credentials Grant Flow**
  - Used for service to service communication

Application Types and Recommended Flows in OAuth 2.1

Application Type	Flow
Machine-to-Machine	Client Credentials Flow
Server-side Web Application	Authorization Code Flow + PKCE ~~OpenID Connect Hybrid Flow *)~~ ~~Password Grant *)~~
Native / Desktop / Mobile Application / CLI	Authorization Code Flow + PKCE ~~Implicit Flow *)~~ ~~OpenID Connect Hybrid Flow *)~~ ~~Password Grant *)~~
Single Page Application (SPA)	Authorization Code Flow + PKCE ~~Implicit Flow *)~~ ~~Password Grant *)~~

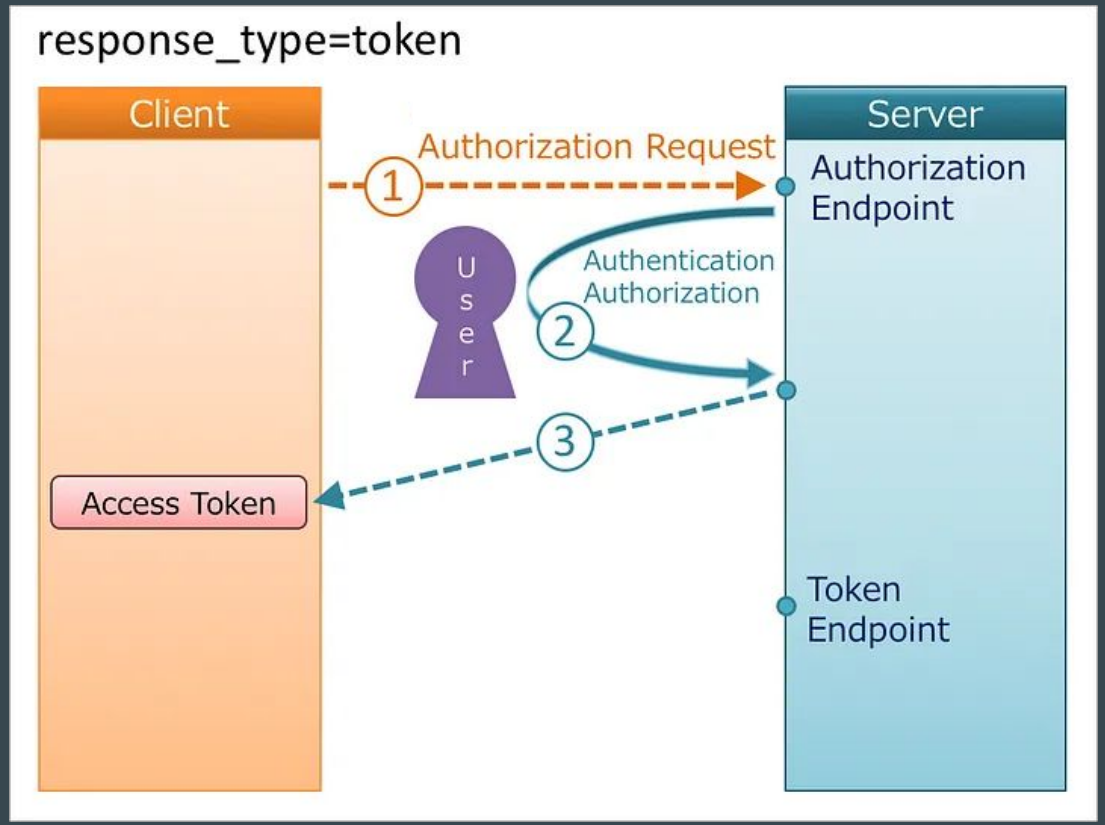
*) Omitted / Deprecated in OAuth 2.1 Standard

OAuth 2.0 Authorization Code Grant Flow



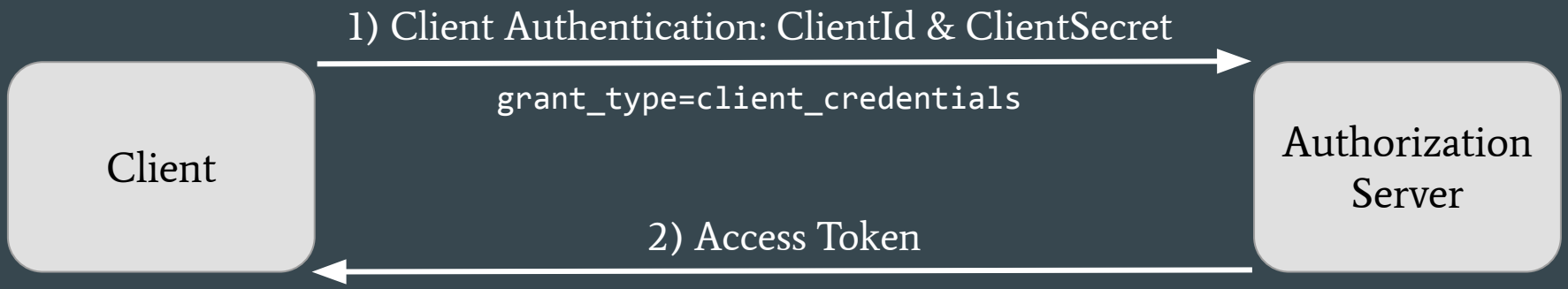
Source

OAuth 2.0 Implicit Flow

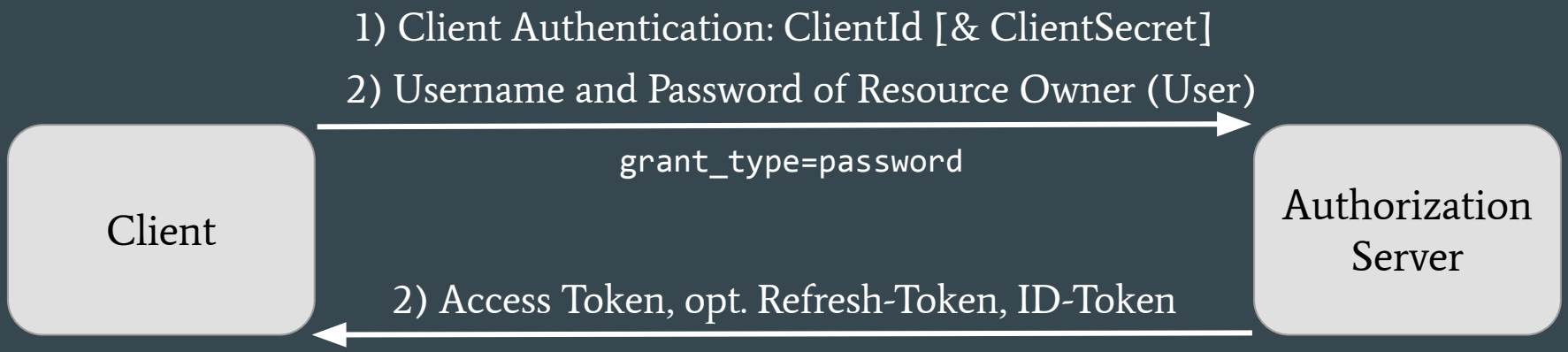


Source

Client Credentials Grant Flow



Resource Owner Password Credentials Grant Flow



OAuth 2.0 Tokens

- **Bearer Tokens**
  - Holder of token can access resource on behalf of the user
- **Access Token (AT)**
  - Used for Accessing Resources
  - Usually short-lived (minutes)
  - Random string or structured
- **Refresh Token (RT)**
  - Used for Obtaining new tokens (Access & Refresh Token) from STS
  - Usually longer-lived (hours, days)
  - Random string or structured

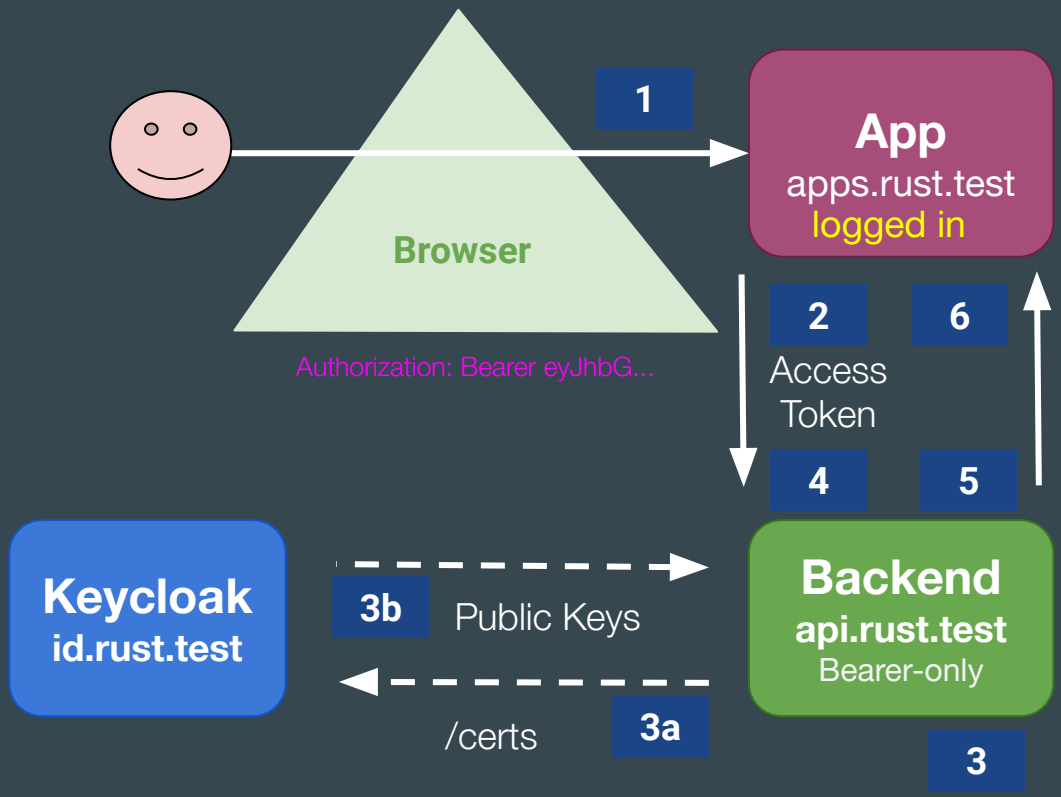
OAuth 2.0 Scope

- A **scope** is a **mechanism** to **limit** app **access** to an **user account**
  - **App** can **request** 1..n **scopes** to **access** resources **on behalf of** a **user**
  - **App asks** the user for **consent** via the **authorization server**
  - Generated **Access Token** will be **limited** to **scopes granted**
- **Scopes often** used to **denote privileges** or **data access contexts**
  - Defines **framework** for **scope definition - no standard scopes**
  - Example scopes: `documents:read, contracts, profile`
- **Scopes** can denote a **collection** of **Claims**
  - e.g. address → Street, City, Country

JSON Identity Suite

- JOSE: JSON Object Signing and Encryption
- JSON Web Token (JWT)
- JSON Web Signature (JWS) → Signed JWTs
- JSON Web Encryption (JWE) → Encrypted JWTs (nested JWS)
- JSON Web Algorithms (JWA) → Common Algorithm names / params
- JSON Web Key (JWK) → Public- / Private-Key representations

Calling Backend Services with Access-Token



1. Authenticated User **accesses** App
2. App **uses** *Access-Token* in *HTTP Header* to access backend
3. Backend **looks-up** *Realm Public Key* in cache with in *Kid* from *JWT*
3a. *If not found*, **fetch** *Public Keys* from AS JWKS endpoint
3b. Keycloak **returns** *Realm Public Keys*
4. Backend **verifies** signature of *Access-Token* with *Realm Public Key*
5. Backend Service **grants** *access* and **returns** *user data*
6. App can now display user data

OpenID Connect

OpenID Connect 1.0

- **Authentication Protocol** extends OAuth2 with Identity Layer
- Governed by OpenID Foundation
- **Simple, widespread and flexible**
- **Collection of specifications**
- Features
  - **ID Token**: **claims** with **User-Information** → **Notion of Identity**
  - **Session Management** for **Single Sign-On** (SSO)
  - **User Info Endpoint** provides **API** for **accessing user information**
  - Endpoints for Discovery automatic configuration of Apps
  - Client Self-Registration
  - Front- / Backchannel Logout

Roles in OpenID Connect

- **OpenID Provider (OP)**
  - Authenticates the user and issues ID tokens after getting proper authorization
  - Authorization Server can be leveraged as OP
- **Relying Party (RP)**
  - App that wants to authenticate users with the OP
  - Can be client, application, service or another OP
- **Authentication**
  - The process of verifying the identity of a user
- **Identity Token (ID Token, IDT)**
  - Token containing claims from authentication of an user by OP when using a client app
  - Must be a JSON Web Token (JWT)

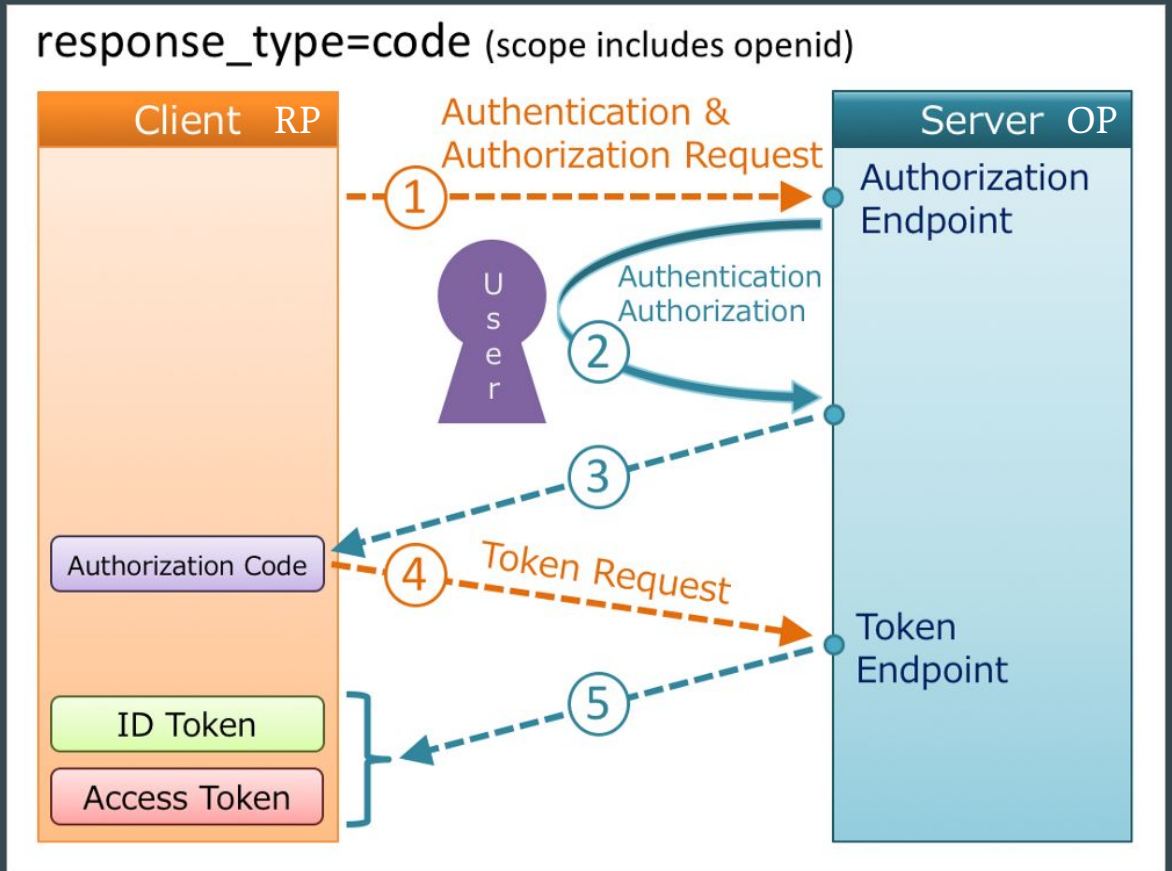
OpenID Connect Flows

- **All flows from OAuth 2.0**
  - Extended with support for ID Token generation
  - ID Token can be returned at different steps in response
- **Behaviour** can be **controlled** via the **response_type** parameter
  - Combination of values `code`,`token`,`id_token`
  - Enables up to 8 different flows
  - **Auth Code Flow** with `response_type=code` most commonly used

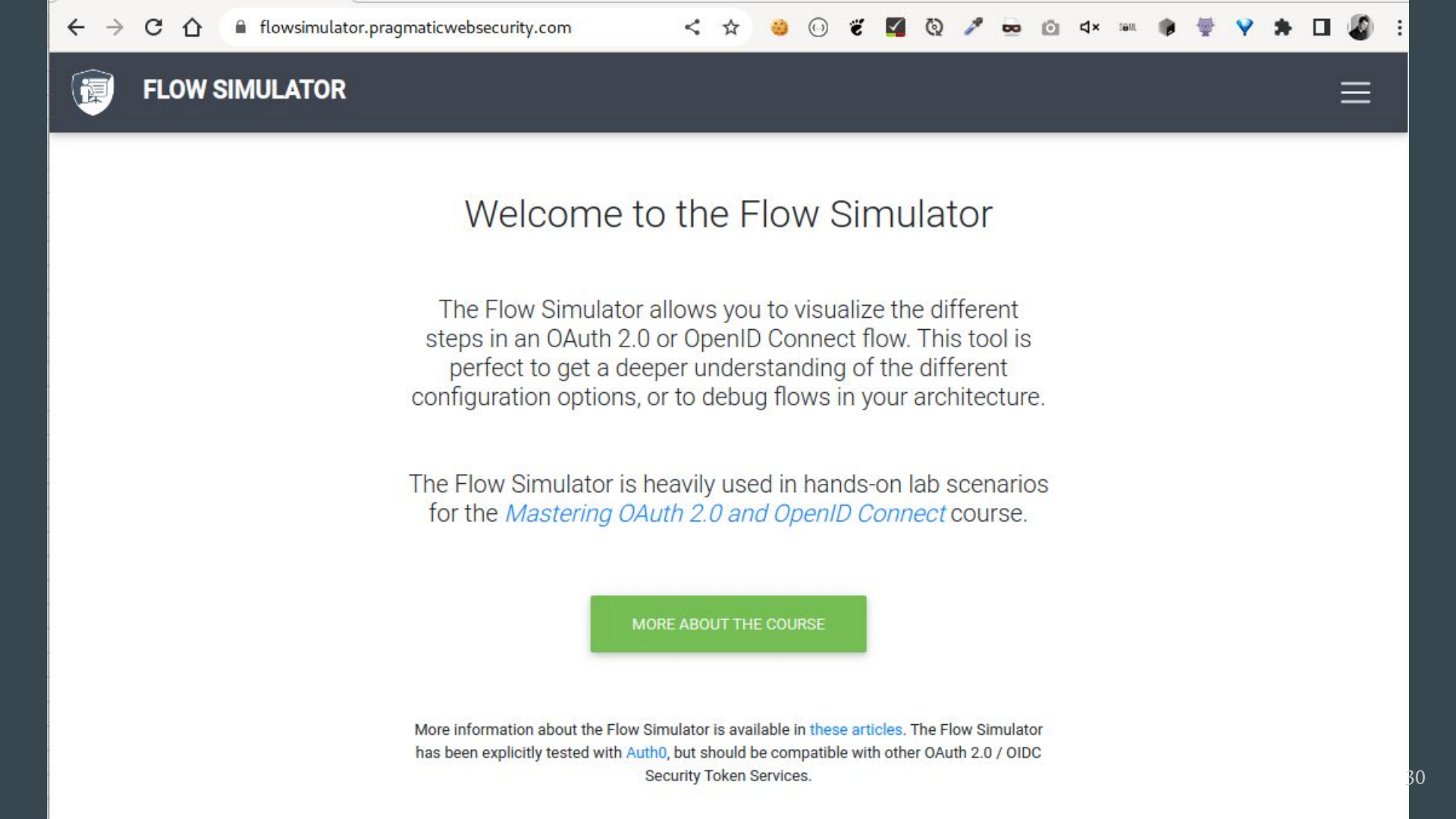
OpenID Connect Scopes

- **Scope** “**openid**” is **mandatory**
  - Enables OpenID Connect protocol and ID Token generation
- **Defines** set of **standardized scopes** with associated claims
  - openid
  - profile
  - email
  - address
  - phone
  - offline_access

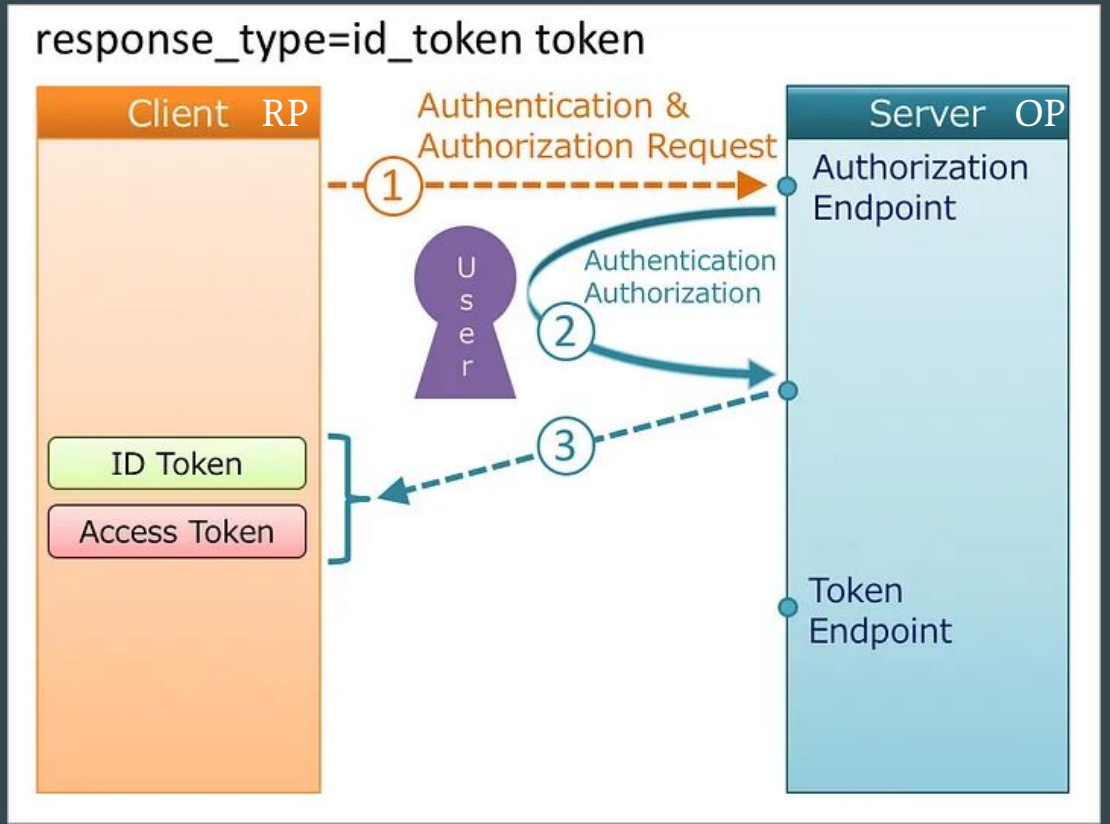
OpenID Connect: Authorization Code Grant Flow



Source


flowsimulator.pragmaticwebsecurity.com
FLOW SIMULATOR
Welcome to the Flow Simulator
The Flow Simulator allows you to visualize the different steps in an OAuth 2.0 or OpenID Connect flow. This tool is perfect to get a deeper understanding of the different configuration options, or to debug flows in your architecture.
The Flow Simulator is heavily used in hands-on lab scenarios for the Mastering OAuth 2.0 and OpenID Connect course.
MORE ABOUT THE COURSE
More information about the Flow Simulator is available in these articles. The Flow Simulator has been explicitly tested with Auth0, but should be compatible with other OAuth 2.0 / OIDC Security Token Services.

OpenID Connect: Implicit Grant Flow



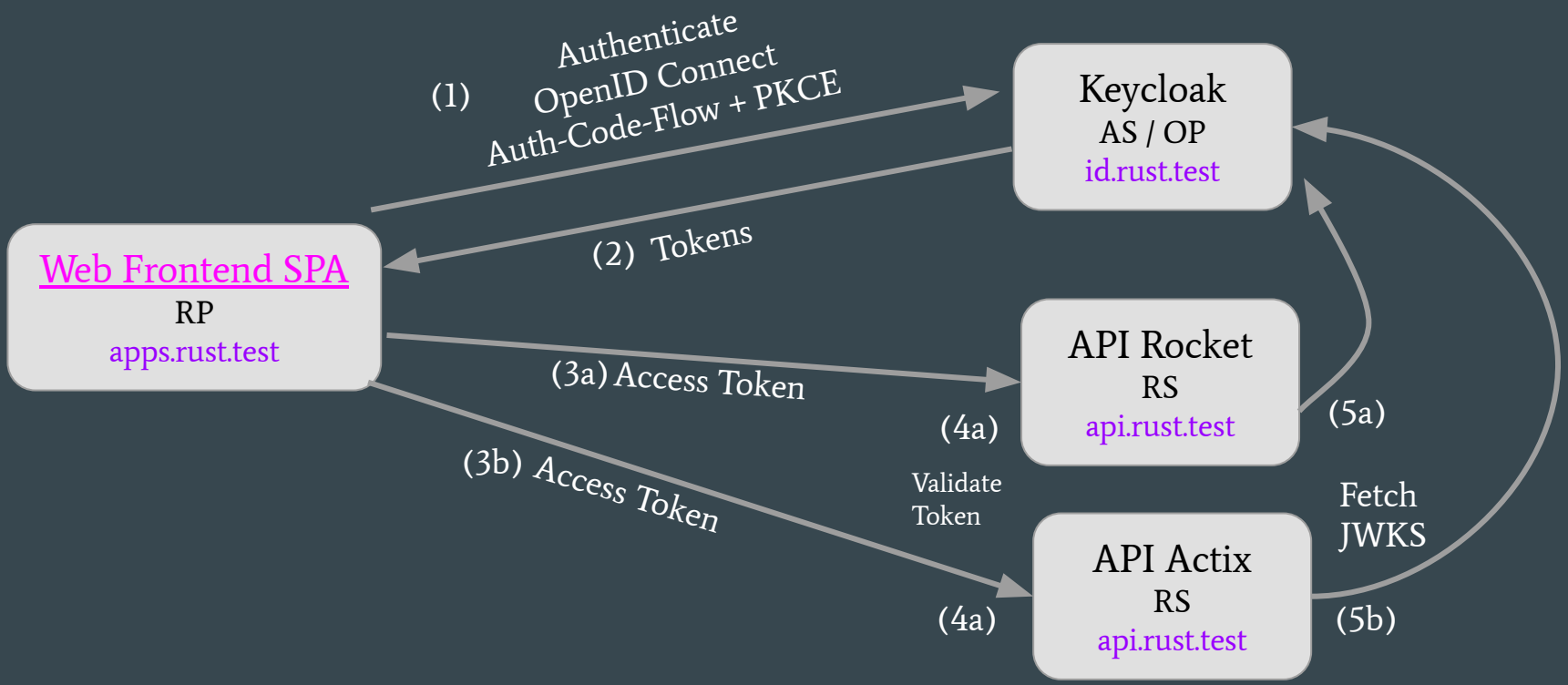
Source

OpenID Connect 1.0 Endpoints

- **Leverages OAuth 2.0 Endpoints** (Authorization, Token, Introspection)
- **User-Info Endpoint**
  - RP can obtain user information with a valid Access-Token
- **End Session Endpoint**
  - RP can request to logout a User on the OP
- **Discovery Endpoint**
  - Used to auto configure RPs
  - ${ISSUER_URL}/.well-known/openid-configuration

What about Rust?

Demo Environment



OAuth Support for Rust Developers

- **OAuth2**
  - `oauth2` ramosbugs/oauth2-rs
  - `pkce` GabrielRPrada/pkce-rs
- **OpenID Connect**
  - `openidconnect` ramosbugs/openidconnect-rs
  - `actix-4-jwt-auth` spectare/actix-4-jwt-auth
- **JWT**
  - `jsonwebtoken` Keats/jsonwebtoken

Summary

- OAuth2 → Authorization Protocol
- OpenID Connect → Authentication Protocol
- OpenID Connect builds upon OAuth
- OAuth 2.1 consolidates OAuth2 specifications, recommended flows:
  - Auth Code Flow + PKCE → Mobile Apps, Web Apps, SPA
  - Client Credentials Flow → Machine 2 Machine Communication
- OAuth2 / OpenID Connect support in Rust is quite usable
  - oauth2 and openidconnect + pkce libraries work well in practice
  - Hard to find examples beyond the basics

thomasdarimont/oauth2-oidc-for-rust-developers