CitiusTech

OAuth 2.0

Version 1.0
July 2020

Agenda

- What is OAuth 2.0?
- OAuth 2.0 Roles
- OAuth 2.0 Client Types
- OAuth Access Tokens and Refresh Token
- OAuth 2.0 and OpenID Connect (OIDC)
- OAuth 2.0 Grant Types and Authorization Flow
- What is a Keycloak?
- Installation and setting up Keycloak server
- Demo 1: Basic Demo using Keycloak as an Authorization Server.
- Demo 2: OAuth 2.0 with social login

What is Spring Security?

- Spring Security is a framework that focuses on providing both authentication and authorization (or “access-control”) to Java web application and SOAP/RESTful web services.
- Spring Security currently supports integration with all of the following technologies:
  - HTTP basic access authentication
  - LDAP system
  - OpenID identity providers
  - JAAS API
  - CAS Server
  - ESB Platform
  - Your own authentication systems
- It is built on top of Spring Framework.

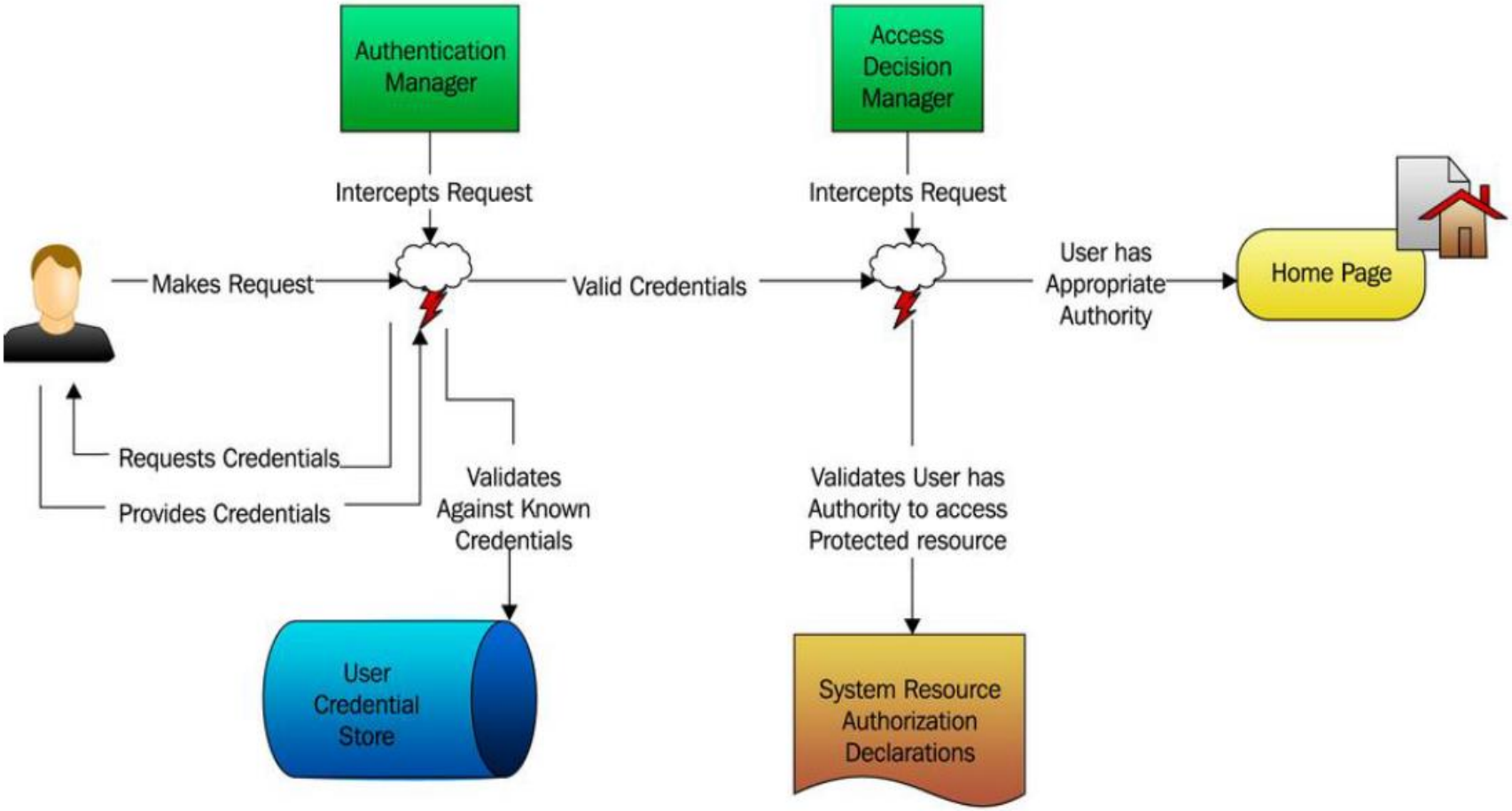
Authentication vs Authorization

- Principal
  - User that performs the action
- Authentication
  - Confirming truth of credentials
- Authorization
  - Define access policy for principal
- GrantedAuthority
  - Application permission granted to a principal
- SecurityContext
  - Hold the authentication and other security information
- SecurityContextHolder
  - Provides access to SecurityContext
- AuthenticationManager
  - Controller in the authentication process
- AuthenticationProvider
  - Interface that maps to a data store which stores your user data.

Authentication vs Authorization

- Authentication Object
  - Object is created upon authentication, which holds the login credentials.
- UserDetails
  - Data object which contains the user credentials, but also the Roles of the user.
- UserDetailsService
  - Collects the user credentials, authorities(roles) and build an UserDetails object.

Authentication vs Authorization



Spring Security Configuration

- Create a Spring Boot Project (https://start.spring.io)
- Add Spring Web and Spring Security dependencies.
- When the application starts, autoconfiguration will detect that Spring Security is in the classpath and will set up some basic security configuration.
- If you want to try it out, fire up the application and try to visit the homepage (or any page for that matter). You'll be prompted for authentication with a login form.
- Default username is **user** and password is randomly generated and written to the application log file. The log entry will look something like this:

  Using default security password: 087cfc6a-027d-44bc-95d7-cbb3a798a1ea
- Spring Security Configuration: Web Security Authentication (@EnableWebSecurity)
  - In-memory Authentication
  - JDBC Authentication
  - LDAP Authentication
  - UserDetailsService

Spring Security Configuration

- Spring Security Java-Based Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web
                          .configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web
                     .configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

- You have to override configure() method defined in WebSecurityConfigurerAdapter class.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

   ...

}
```

In-memory Authentication

- In-memory Authentication, user information / credentials is stored in memory.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .inMemoryAuthentication()
      .withUser("buzz")
        .password("infinity")
        .authorities("ROLE_USER")
      .and()
      .withUser("woody")
        .password("bullseye")
        .authorities("ROLE_USER");

}
```

- The in-memory user store is convenient for testing purposes or for very simple applications, but it doesn't allow for easy editing of users.
- If you need to add, remove, or change a user, you'll have to make the necessary changes and then rebuild and redeploy the application.

JDBC Authentication

- With JDBC-based Authentication, the user's authentication and authorization information are stored in the database.
- The standard JDBC implementation of UserDetailsService requires tables to load the password, account status (enabled or disabled) and list of authorities (roles) for the user. You have to use the schema defined.

```
@Autowired
DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource);

}
```

JDBC Authentication

- Although this minimal configuration will work, it makes some assumptions about your database schema. It expects that certain tables exist where user data will be kept.
- More specifically, the following snippet of code from Spring Security's internals shows the SQL queries that will be performed when looking up user details:

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
        "select username,password,enabled " +
        "from users " +
        "where username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
        "select username,authority " +
        "from authorities " +
        "where username = ?";
public static final String DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY =
        "select g.id, g.group_name, ga.authority " +
        "from groups g, group_members gm, group_authorities ga " +
        "where gm.username = ? " +
        "and g.id = ga.group_id " +
        "and g.id = gm.group_id";
```

- The first query retrieves a user's username, password, and whether or not they're enabled. This information is used to authenticate the user.
- The next query looks up the user's granted authorities for authorization purposes.
- The final query looks up authorities granted to a user as a member of a group.

JDBC Authentication

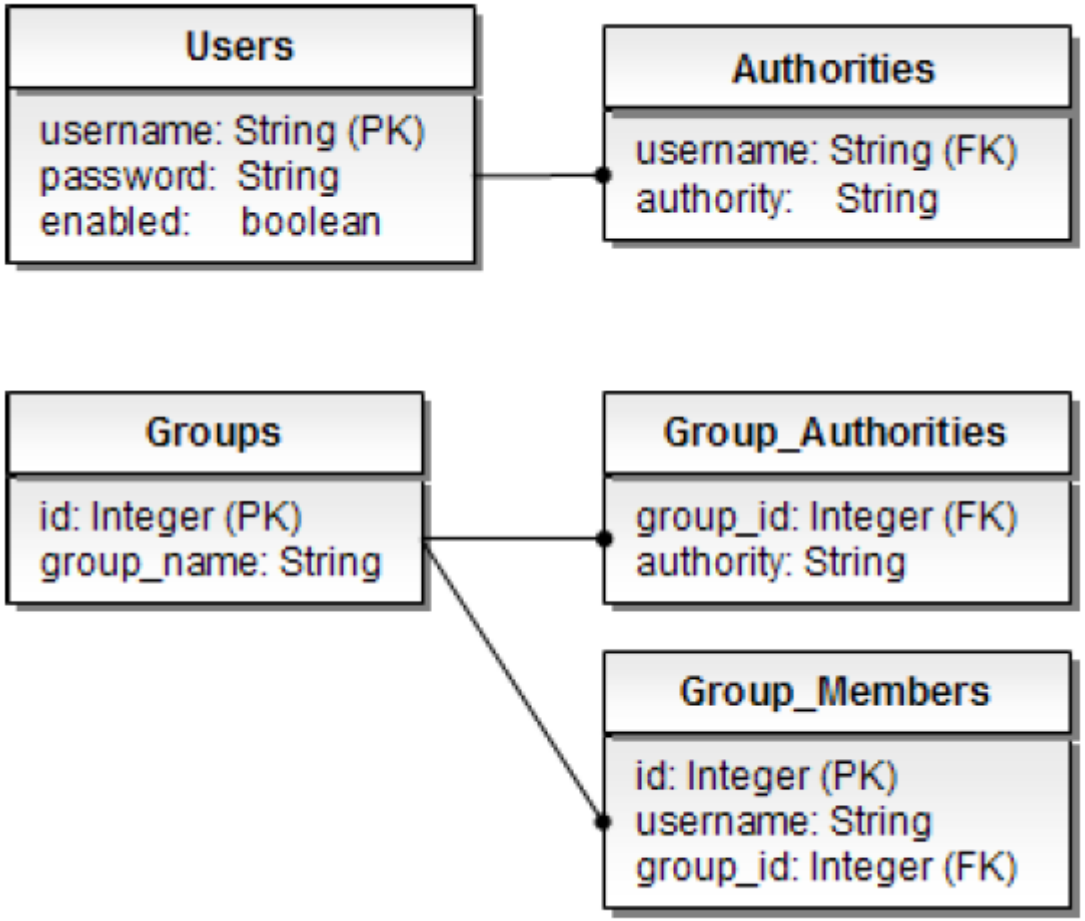
- If you're OK with defining and populating tables in your database that satisfy those queries, there's not much else for you to do. But chances are your database doesn't look anything like this, and you'll want more control over the queries.
- In that case, you can configure your own queries.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
          "select username, password, enabled from Users " +
          "where username=?")
      .authoritiesByUsernameQuery(
          "select username, authority from UserAuthorities " +
          "where username=?");

}
```

- In this case, you only override the authentication and basic authorization queries.
- But you can also override the group authorities query by calling groupAuthoritiesByUsername() with a custom query.

JDBC Authentication

- Authentication DB Schema



JDBC Authentication

- When replacing the default SQL queries with those of your own design, it's important to adhere to the basic contract of the queries. All of them take the username as their only parameter.
- The authentication query selects the username, password, and enabled status.
- The authorities query selects zero or more rows containing the username and a granted authority.
- The group authorities query selects zero or more rows, each with a group ID, a group name, and an authority.
- Focusing on the authentication query, you can see that user passwords are expected to be stored in the database.
- The only problem with this is that if the passwords are stored in plain text, they're subject to the prying eyes of a hacker.
- But if you encode the passwords in the database, authentication will fail because it won't match the plaintext password submitted by the user.
- To remedy this problem, you need to specify a password encoder by calling the passwordEncoder() method

JDBC Authentication

- The passwordEncoder() method accepts any implementation of Spring Security's PasswordEncoder interface.
- Spring Security's cryptography module includes several such implementations:
  - BCryptPasswordEncoder - Applies bcrypt strong hashing encryption
  - NoOpPasswordEncoder - Applies no encoding
  - bkdf2PasswordEncoder - Applies PBKDF2 encryption
  - SCryptPasswordEncoder - Applies scrypt hashing encryption
  - StandardPasswordEncoder - Applies SHA-256 hashing encryption
- The preceding code uses StandardPasswordEncoder. But you can choose any of the other implementations or even provide your own custom implementation if none of the out-of-the-box implementations meet your needs.

```
public interface PasswordEncoder {
  String encode(CharSequence rawPassword);
  boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

- Spring Security 5 recommends the use of BCrypt, a salt that helps prevent pre-computed dictionary attacks.

JDBC Authentication

- Most of the other encryption mechanisms, such as the MD5PasswordEncoder and ShaPasswordEncoder use weaker algorithms and are now deprecated.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
          "select username, password, enabled from Users " +
          "where username=?")
      .authoritiesByUsernameQuery(
          "select username, authority from UserAuthorities " +
          "where username=?")
      .passwordEncoder(new StandardPasswordEncoder("53cr3t");

}
```

JDBC Authentication

- Custom PasswordEncoder

```
@Service
public class PasswordEncoderImpl implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        String hashed = BCrypt.hashpw(rawPassword.toString(), BCrypt.gensalt(12));
        return hashed;
    }
    @Override
    public boolean matches(CharSequence rawPassword, String encodedPassword) {
        return BCrypt.checkpw(rawPassword.toString(), encodedPassword);
    }
}
```

Customizing user Authentication - UserDetailsService

- The standard scenario uses a simple UserDetailsService where I usually store the password in database and spring security framework performs a check on the password.
- What happens when you are using a different authentication system and the password is not provided in your own database / data model?
- Implement UserDetailsService interface

```
/*Core interface which loads user-specific data.*/
public interface UserDetailsService {
  /* Locates the user based on the username.
   * @param username the username identifying the user
   * @return a fully populated user record (never null)
   * @throws UsernameNotFoundException if the user could not be
     found or the user has no GrantedAuthority
   * @throws DataAccessException if user could not be found for a
     repository-specific reason*/
  UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException, DataAccessException;
}
```

Customizing user Authentication - UserDetailsService

- Implement UserDetails interface

```
/* Provides core user information.*/
public interface UserDetails extends Serializable {

  Collection<GrantedAuthority> getAuthorities();
  String getPassword();
  String getUsername();

  boolean isAccountNonExpired();
  boolean isAccountNonLocked();
  boolean isCredentialsNonExpired();
  boolean isEnabled();
}
```

Customizing user Authentication - UserDetailsService

- This UserDetailsService only has access to the username in order to retrieve the full user entity - and in a large number of scenarios, this is enough.

```
@Service
public class UserRepositoryUserDetailsService
        implements UserDetailsService {

  private UserRepository userRepo;

  @Autowired
  public UserRepositoryUserDetailsService(UserRepository userRepo) {
    this.userRepo = userRepo;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {
    User user = userRepo.findByUsername(username);
    if (user != null) {
      return user;
    }
    throw new UsernameNotFoundException(
                    "User '" + username + "' not found");
  }

}
```

Customizing user Authentication - UserDetailsService

```
@Autowired
private UserDetailsService userDetailsService;

@Bean
public PasswordEncoder encoder() {
  return new StandardPasswordEncoder("53cr3t");
}

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .userDetailsService(userDetailsService)
    .passwordEncoder(encoder());

}
```

Securing Web Requests

- In our application, it should require that a user must be authenticated before accessing certain resources / pages. But certain resources / pages like homepage, login page and registration page should be available to unauthenticated users.
- To configure these security rules, you have to override WebSecurityConfigurerAdapter's configure() method:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  ...
}
```

- This configure() method accepts an HttpSecurity object, which can be used to configure how security is handled at the web level.
- Among the many things you can configure with HttpSecurity are these:
  - Requiring that certain security conditions be met before allowing a request to be served
  - Configuring a custom login page
  - Enabling users to log out of the application
  - Configuring cross-site request forgery protection
- Intercepting requests to ensure that the user has proper authority is one of the most common things you'll configure HttpSecurity to do.

Securing Web Requests

- URL-Based Authorization - Very common especially with role-based authorization.
- Map URL structure to authorities.
- Optionally including HTTP methods.

```
/* configure URL-based authorization: */

@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/admin/**").hasRole("ADMIN")
      .antMatchers(HttpMethod.POST,
                   "/projects/**").hasRole("PROJECT_MGR")
      .anyRequest().authenticated();
  // additional configuration not shown…
  }
}
```

Spring Security Configuration

@EnableWebSecurity and HttpSecurity

```java
@RequestMapping("/")
public String root() {
   return "redirect:/index";
}

@RequestMapping("/index")
public String index() {
   return "index";
}

@RequestMapping("/user/index")
public String userIndex() { return "user/index"; }

@RequestMapping(value = "/login")
public String login() {
   return "login";
}

@RequestMapping(value = "/forbidden")
public String forbidden() {
   return "forbidden";
}
```

```java
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   protected void configure(HttpSecurity http) throws   Exception {

      http
            .authorizeRequests()
            .antMatchers("/", "/index").permitAll()
            .anyRequest()
            .hasAnyRole("Consultant“, “Teamlead”, “Hr”)
            .and()
            .formLogin()
            .loginPage("/login").failureUrl("/login-error")
            .and()
            .exceptionHandling()
            .accessDeniedPage("/forbidden");

...
```

You have to adapt it for your own application

- Adding more configuration options:
  - Form Login
  - Authorize request
  - Handling logout, exception, custom filters…

Securing Web Requests

Method	What it does
`access(String)`	Allows access if the given SpEL expression evaluates to `true`
`anonymous()`	Allows access to anonymous users
`authenticated()`	Allows access to authenticated users
`denyAll()`	Denies access unconditionally
`fullyAuthenticated()`	Allows access if the user is fully authenticated (not remembered)
`hasAnyAuthority(String...)`	Allows access if the user has any of the given authorities
`hasAnyRole(String...)`	Allows access if the user has any of the given roles
`hasAuthority(String)`	Allows access if the user has the given authority
`hasIpAddress(String)`	Allows access if the request comes from the given IP address

Securing Web Requests

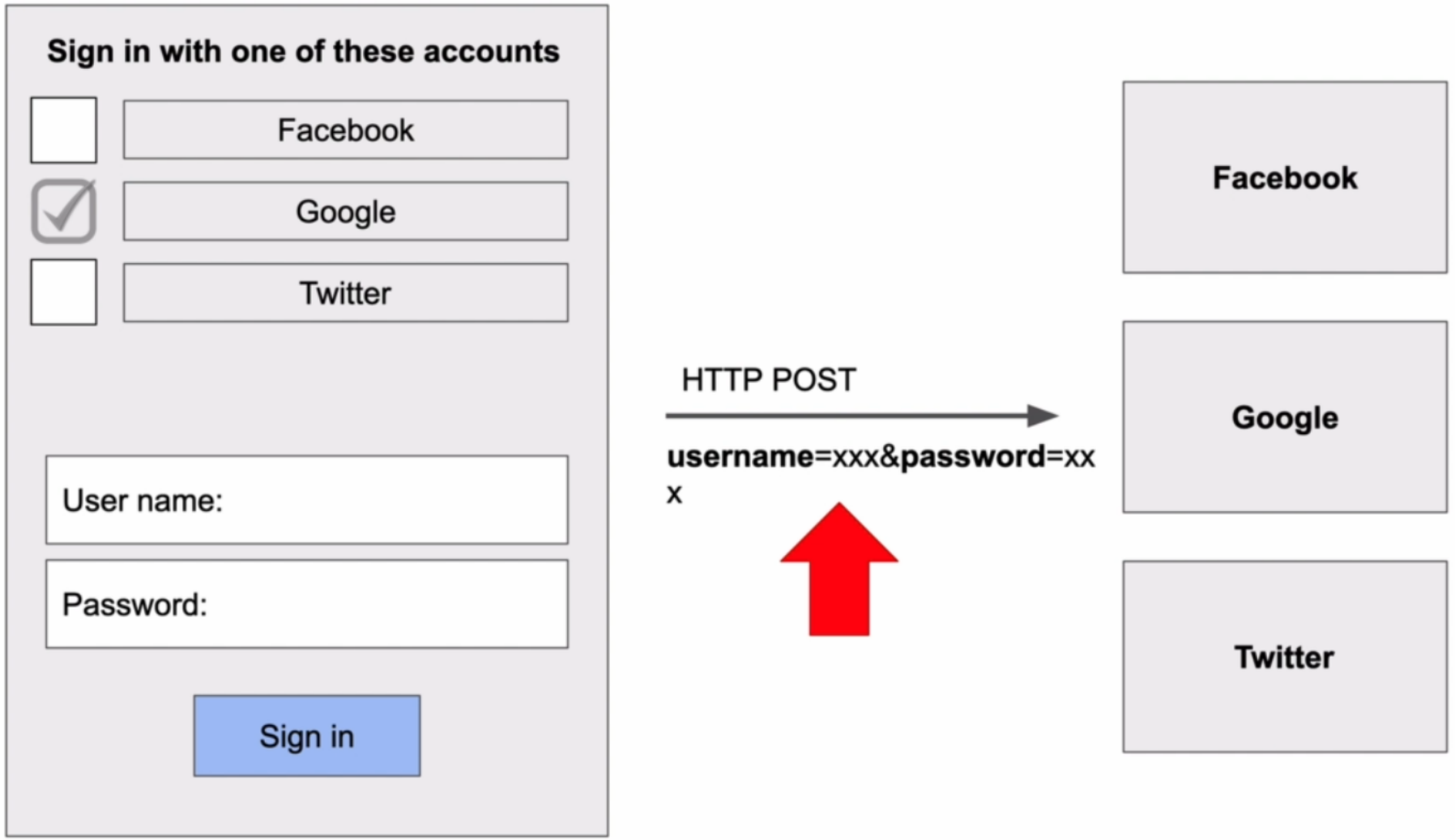
Method	What it does
`hasRole(String)`	Allows access if the user has the given role
`not()`	Negates the effect of any of the other access methods
`permitAll()`	Allows access unconditionally
`rememberMe()`	Allows access for users who are authenticated via remember-me

Securing Web Requests

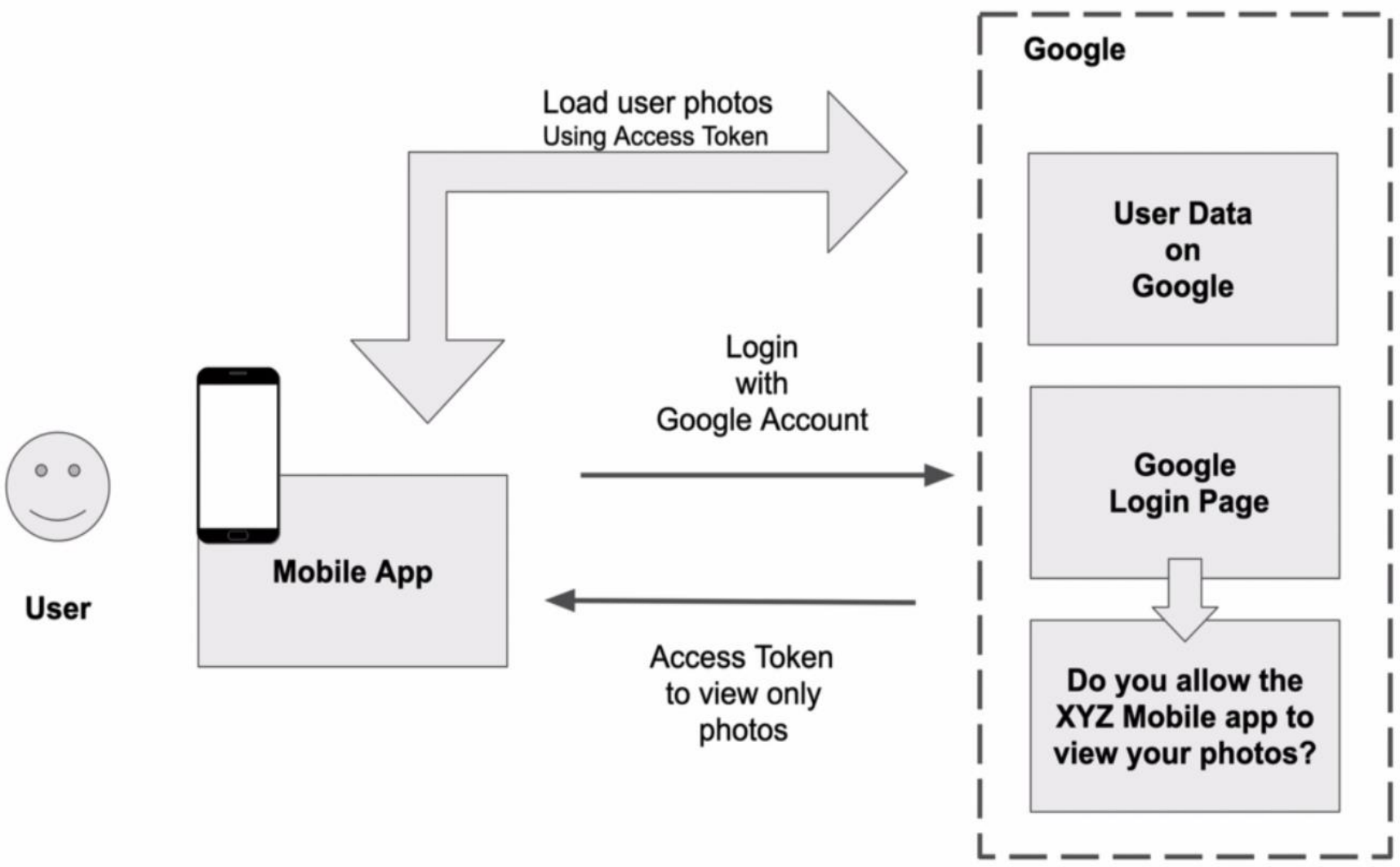
Security expression	What it evaluates to
`authentication`	The user's authentication object
`denyAll`	Always evaluates to `false`
`hasAnyRole(list of roles)`	`true` if the user has any of the given roles
`hasRole(role)`	`true` if the user has the given role
`hasIpAddress(IP address)`	`true` if the request comes from the given IP address
`isAnonymous()`	`true` if the user is anonymous
`isAuthenticated()`	`true` if the user is authenticated
`isFullyAuthenticated()`	`true` if the user is fully authenticated (not authenticated with remember-me)
`isRememberMe()`	`true` if the user was authenticated via remember-me
`permitAll`	Always evaluates to `true`
`principal`	The user's principal object

What is OAuth 2.0?

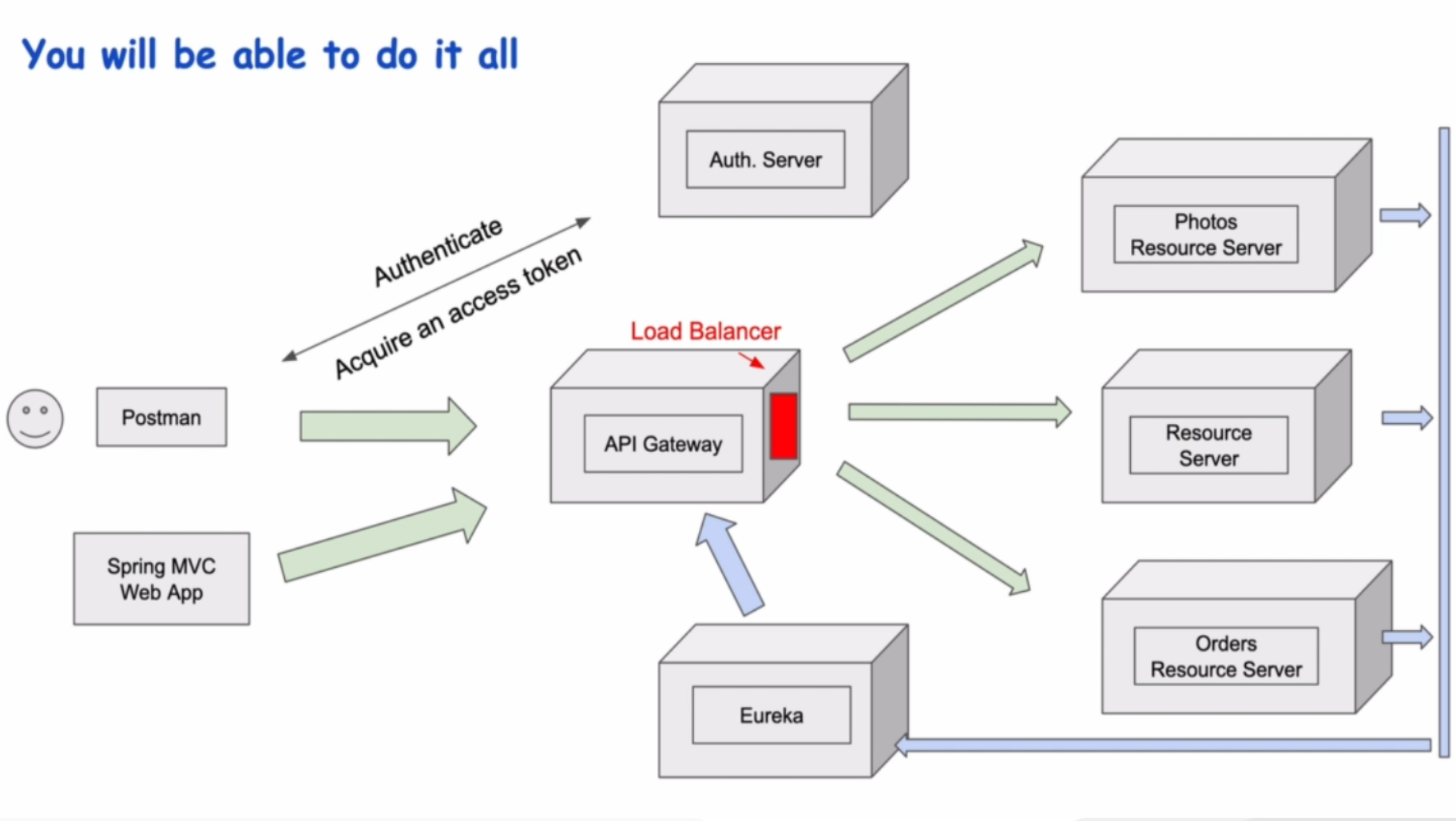
- OAuth = Open + Authorization
- OAuth 2.0 – is an **Authorization Framework.**
- OAuth – is a **delegated** authorization Framework.


Sign in with one of these accounts
Facebook
Google
Twitter
User name:
Password:
Sign in
HTTP POST
username=xxx&password=xx
x
Facebook
Google
Twitter

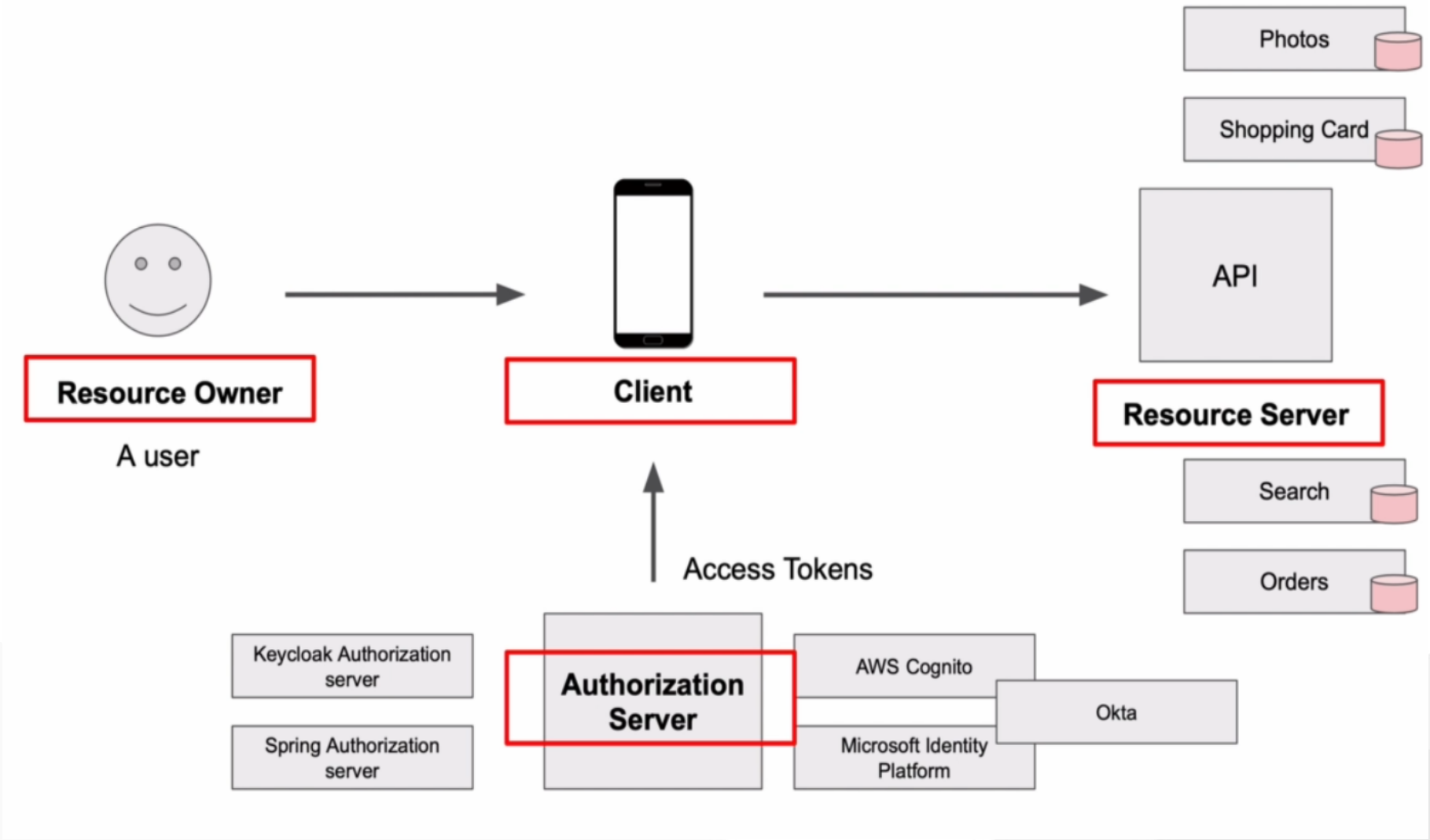
What is OAuth 2.0?



What is OAuth 2.0?



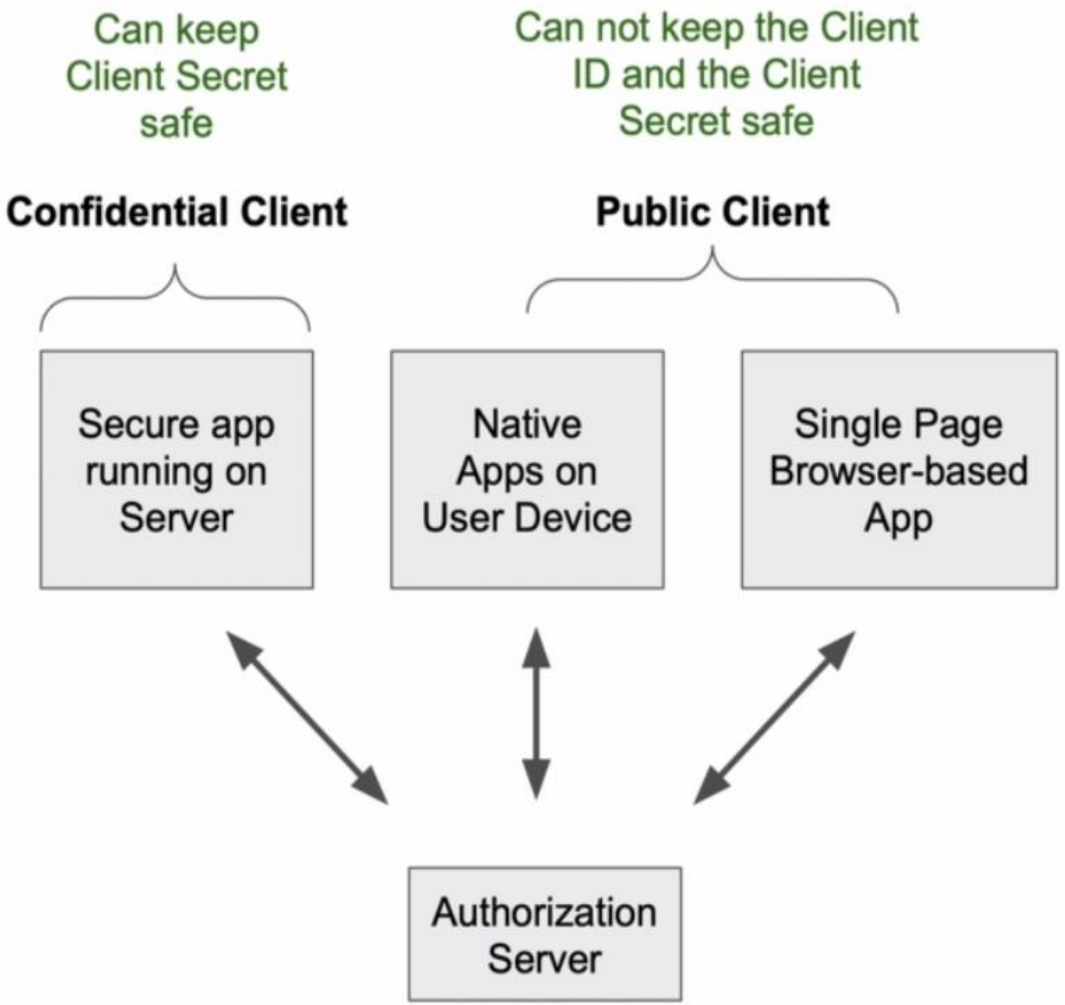
OAuth Roles


Photos
Shopping Card
API
Resource Owner
A user
Client
Resource Server
Search
Orders
Access Tokens
Keycloak Authorization server
Spring Authorization server
Authorization Server
AWS Cognito
Okta
Microsoft Identity Platform

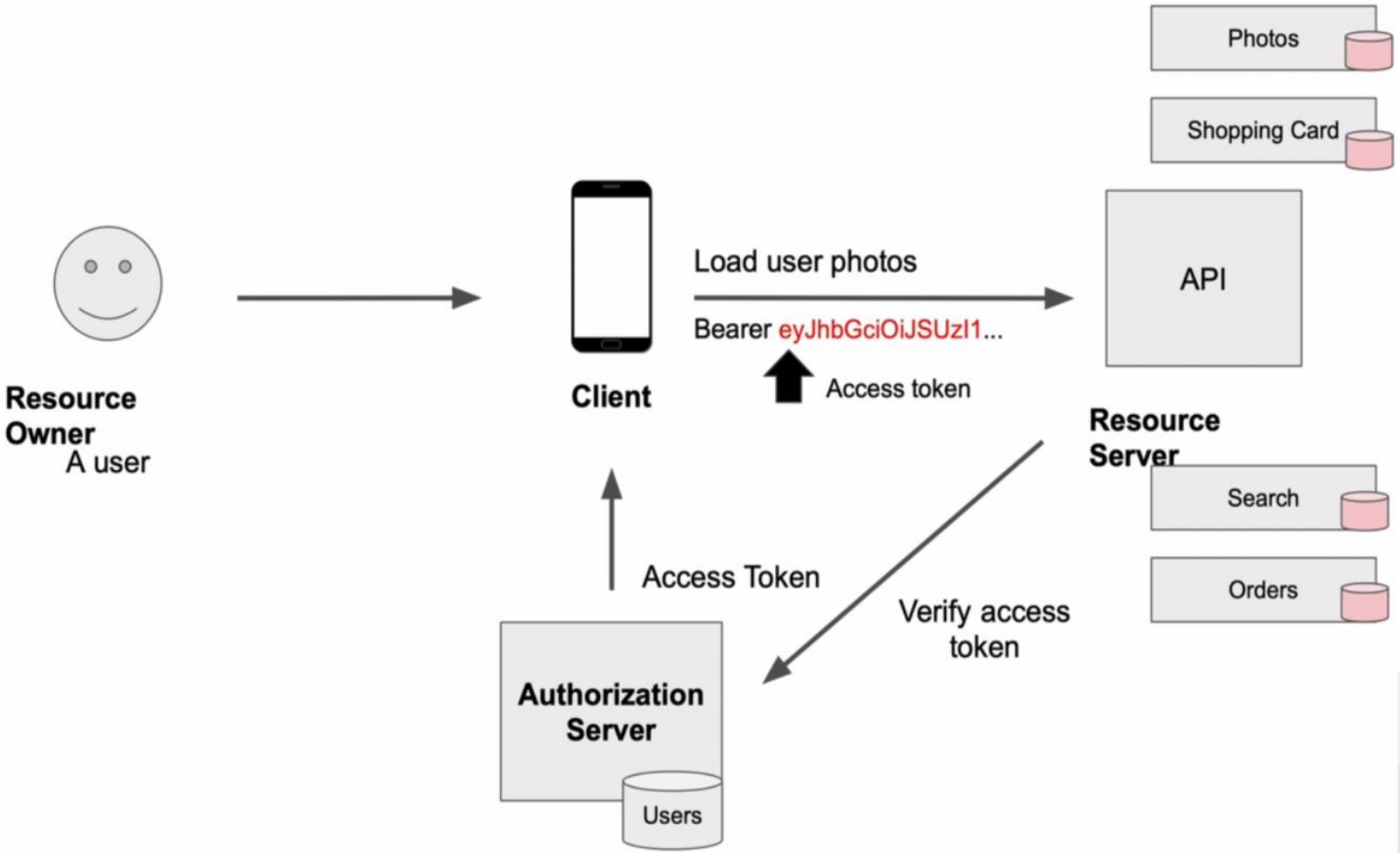
OAuth Roles

- An OAuth client is a piece of software that attempts to access the protected resource on behalf of the resource owner, and it uses OAuth to obtain that access.
- An OAuth protected resource is available through an HTTP server and it requires an OAuth token to be accessed. The protected resource also has the responsibility to validate the token
- A resource owner is the entity that has the authority to delegate access to the client. In most cases, it's a user accessing web browser.
- An OAuth authorization server is an HTTP server that acts as the central component to an OAuth system. The authorization server authenticates the resource owner and client, provides mechanisms for allowing resource owners to authorize clients, and issues tokens to the client.

OAuth Client Types



OAuth Access Tokens

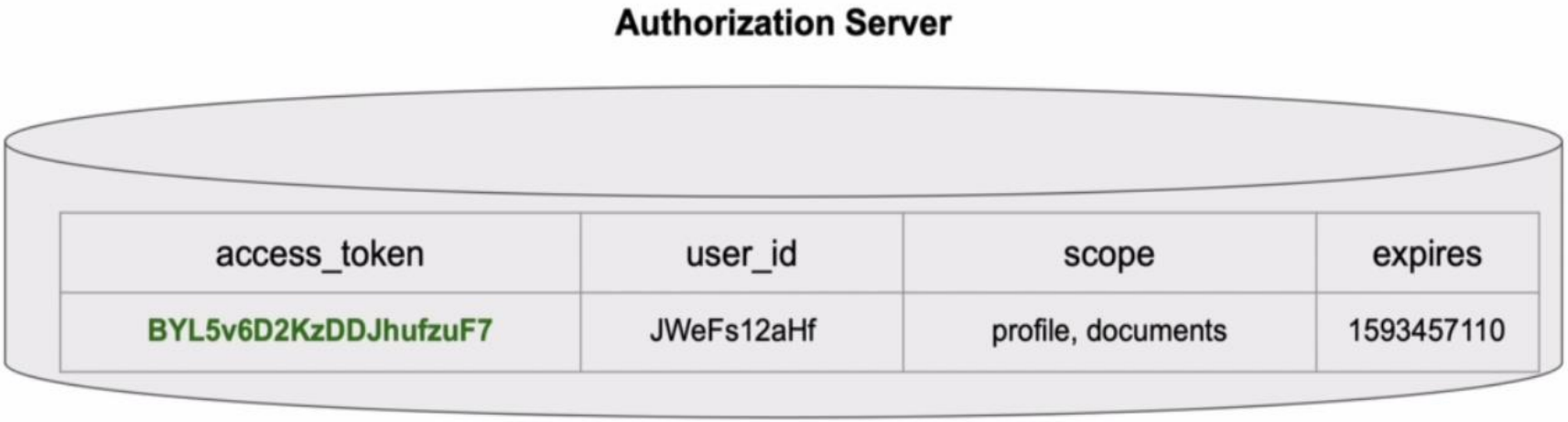


OAuth Access Tokens

- The token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in verifiable manner (i.e. a token string consisting of some data and a signature).
- There are 2 types of tokens:
  - Identifier type
  - Self-contain the authorization information
- OAuth does not define a format or content for the token itself, but it always represents the combination of the client's requested access, the resource owner that authorized the client, and the rights conferred during that authorization
- OAuth tokens are opaque to the client, which means that the client has no need (and often no ability) to look at the token itself.
- The client's job is to carry the token, requesting it from the authorization server and presenting it to the protected resource

Identifier Type Access Token

- Does not contain information about authorized user.



Self-contained Type Access Token (JWT)

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret or a public/private key pair using RSA.
- **Compact:** Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.
  - E.g. eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
- **Self-contained:** The payload contains all the required information about the user, avoiding the need to query the database more than once.

Self-contained Type Access Token (JWT)

- JSON Web Tokens consist of three parts separated by dots (.), which are:
  - Header
  - Payload
  - Signature
- Therefore, a JWT typically looks like the following.
  - xxxxx.yyyyy.zzzzz
- **Header**
  - The header typically consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA.
- For Example

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

- Then, this JSON is Base64Url encoded to form the first part of the JWT.

Self-contained Type Access Token (JWT)

- **Payload:** The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims:
  - Reserved
  - Public
  - Private
- Reserved claims
  - These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub**(subject), **aud** (audience), and others.
  - Notice that the claim names are only three characters long as JWT is meant to be compact.
- Public claims
  - These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
- Private claims
  - These are the custom claims created to share information between parties that agree on using them.

Self-contained Type Access Token (JWT)

- **Payload Example:**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

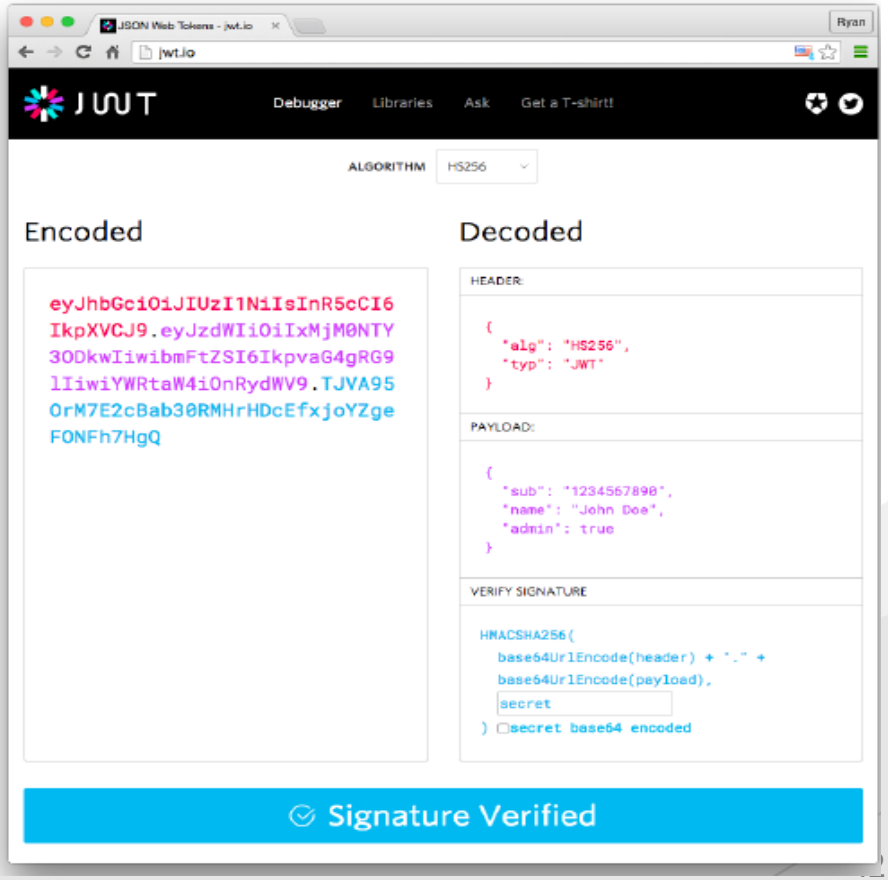
- The payload is then Base64Url encoded to form the second part of the JSON Web Token.
- **Signature:**
  - To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
  - The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
  - For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

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

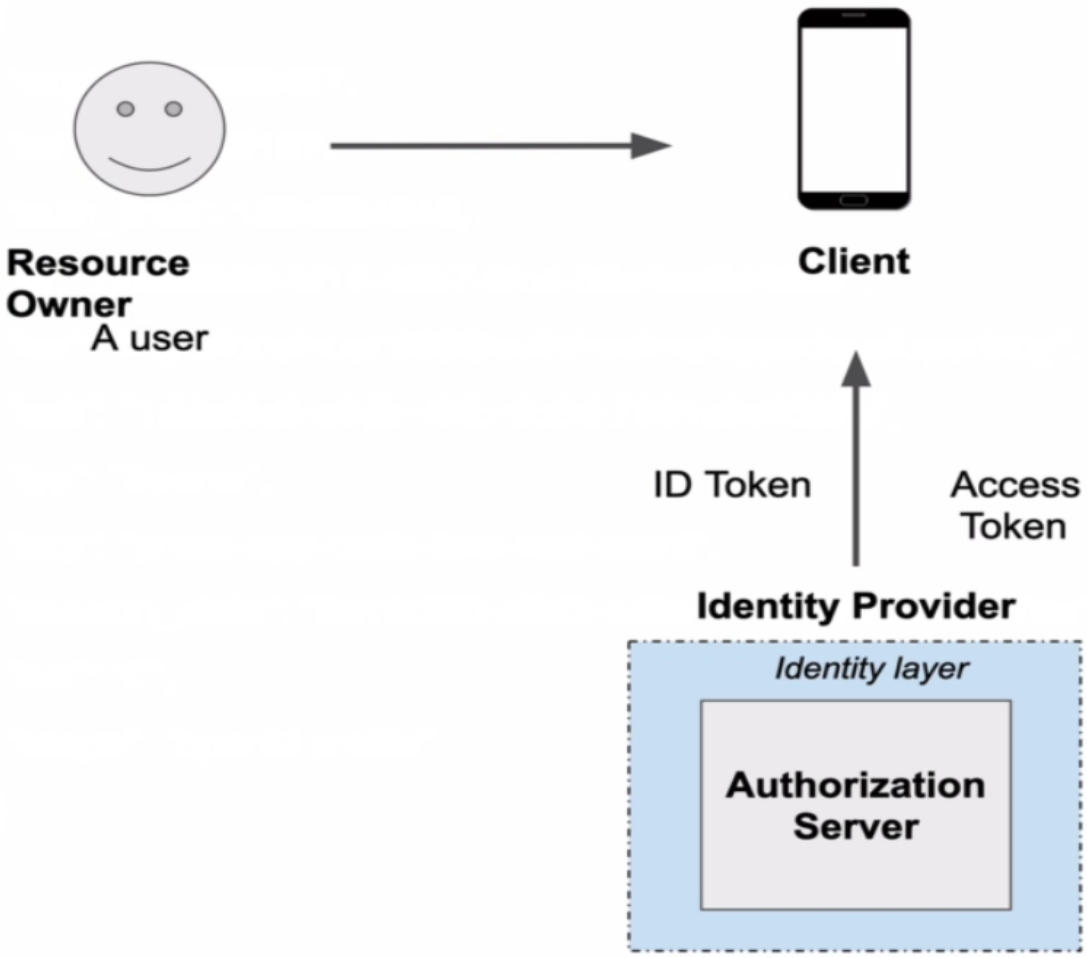
Self-contained Type Access Token (JWT)

- **Putting all together**
- The output is three Base64 strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.
- The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```



OAuth 2.0 and OpenID Connect (OIDC)



OAuth 2.0 and OpenID Connect (OIDC)


```
{
 "exp": 1594066487,
 "iat": 1594066187,
 "auth_time": 1594064948,
 "jti": "bc8e8b98-bf10-43b7-9fcd-30a4ed3d495e",
 "iss": "http://localhost:8080/auth/realms/appsdeveloperblog",
 "sub": "1dde3fc3-c6db-49fb-9b3d-7964c5c0687a",
 "typ": "Bearer",
 "azp": "photo-app-code-flow-rest-client",
 "session_state": "9e1fa365-e9d3-4d3a-8753-870120b537ce",
 "acr": "0",
 "scope": "openid profile"
}
```

OAuth 2.0 and OpenID Connect (OIDC)


```
{
  "exp": 1594066487,
  "iat": 1594066187,
  "auth_time": 1594064948,
  "jti": "b0885896-79ef-42b0-8181-53582f58411d",
  "iss": "http://localhost:8080/auth/realms/appsdeveloperblog",
  "aud": "photo-app-code-flow-rest-client",
  "sub": "1dde3fc3-c6db-49fb-9b3d-7964c5c0687a",
  "typ": "ID",
  "azp": "photo-app-code-flow-rest-client",
  "session_state": "9e1fa365-e9d3-4d3a-8753-870120b537ce",
  "acr": "0",
  "name": "Kargopolov",
  "preferred_username": "sergey",
  "family_name": "Kargopolov"
}
```

OAuth 2.0 and OpenID Connect (OIDC)

Standard Claims

sub
name
given_name
family_name
middle_name
nickname
preferred_username
profile
picture
website
email
email_verified
gender
birthdate
zoneinfo
locale
phone_number
phone_number_verified
address
updated_at

Address Claims

formatted
street_address
locality
region
postal_code
country

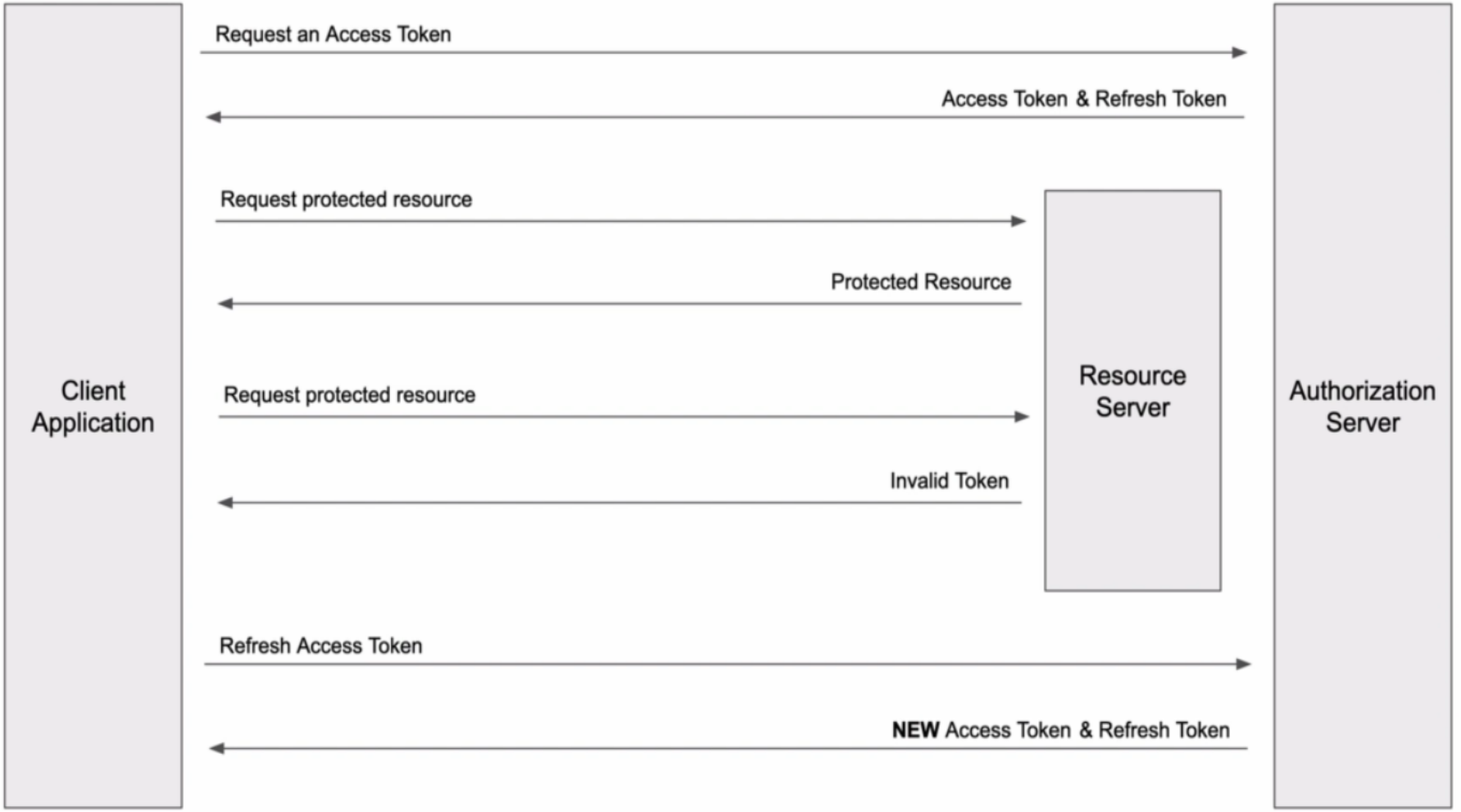
Scopes

openid
profile
email
address
phone

Refresh Token

- An OAuth refresh token is similar in concept to the access token, in that it's issued to the client by the authorization server and the client doesn't know or care what's inside the token.
- What's different, though, is that the token is never sent to the protected resource.
- Instead, the client uses the refresh token to request new access tokens from Authorization Server without involving the resource owner

Refresh Token



Refresh Token


```
{
  "access_token": "eyJhbGciOiJSUz...",
  "expires_in": 300,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUzI1NiIsIn...",
  "token_type": "bearer",
  "not-before-policy": 1593436696,
  "session_state": "4ace79d4-29e9-458f-8253-57e1c194f74b",
  "scope": "profile"
}
```


Expires in 30 minutes


```
{
  "access_token": "eyJhbGciOiJSUz...",
  "expires_in": 300,
  "refresh_expires_in": 0,
  "refresh_token": "eyJhbGciOiJIUzI1NiIsI...",
  "scope": "offline_access email openid",
  ...
}
```


Never expires

Scope

- An OAuth scope is a representation of a set of rights at a protected resource.
- Scopes are represented by strings in the OAuth protocol, and they can be combined into a set by using a space-separated list.
- Scopes are defined by the protected resource, based on the API that it's offering.
- Clients can request certain scopes, and the authorization server can allow the resource owner to grant or deny particular scopes to a given client during its request.

OAuth 2.0 Grant Types

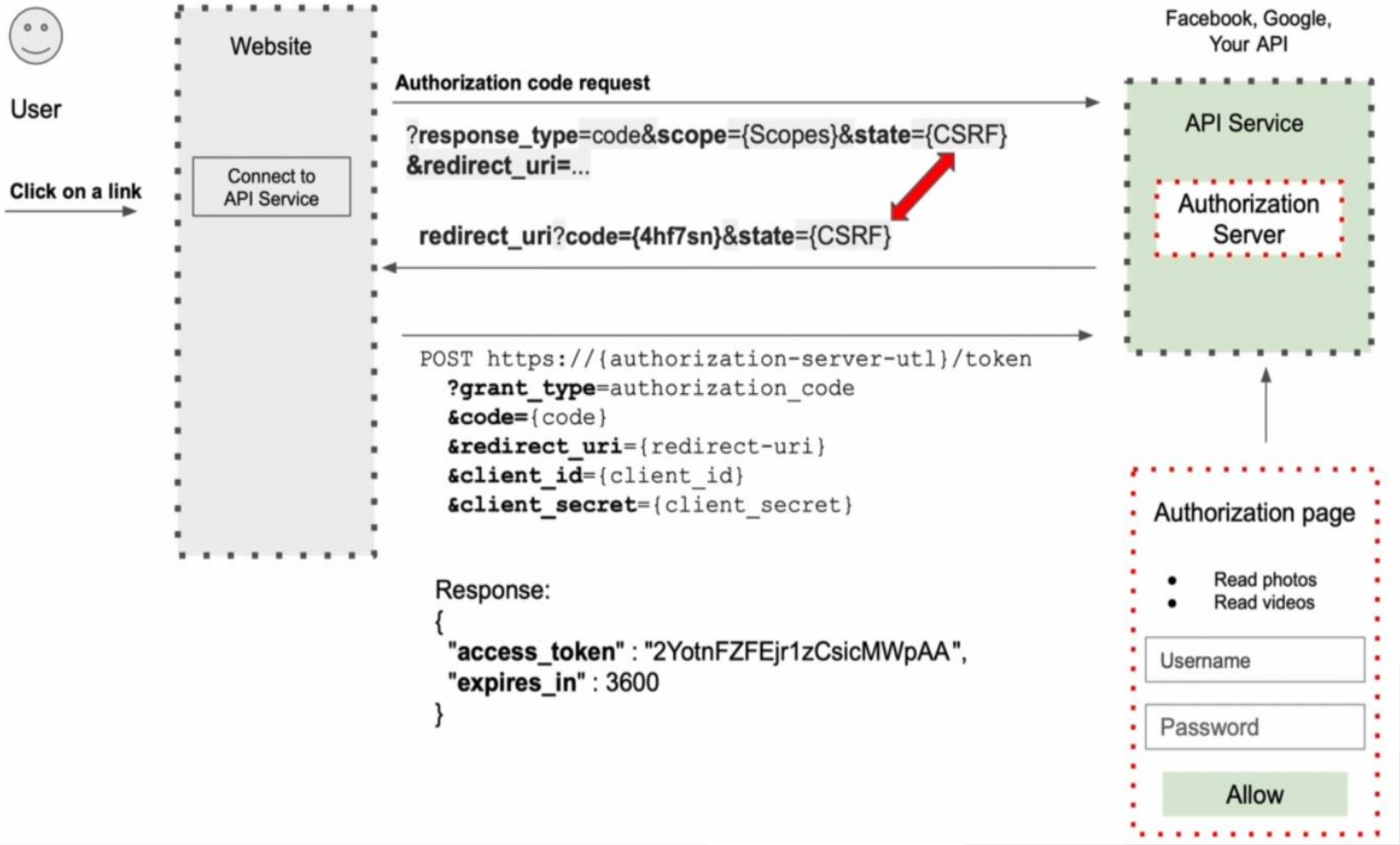
- Grant type is a way an application gets an access token.

Server Side Web App	Server Side Script with no UI	JavaScript Single Page App	Mobile Native App	Device
Authorization Code	Client Credentials	PKCE Enhanced Authorization Code	Authorization Code	Device Code
Password Grant		Implicit Flow	PKCE Enhanced Authorization Code	
		Password Grant	Implicit Flow	
			Password Grant	

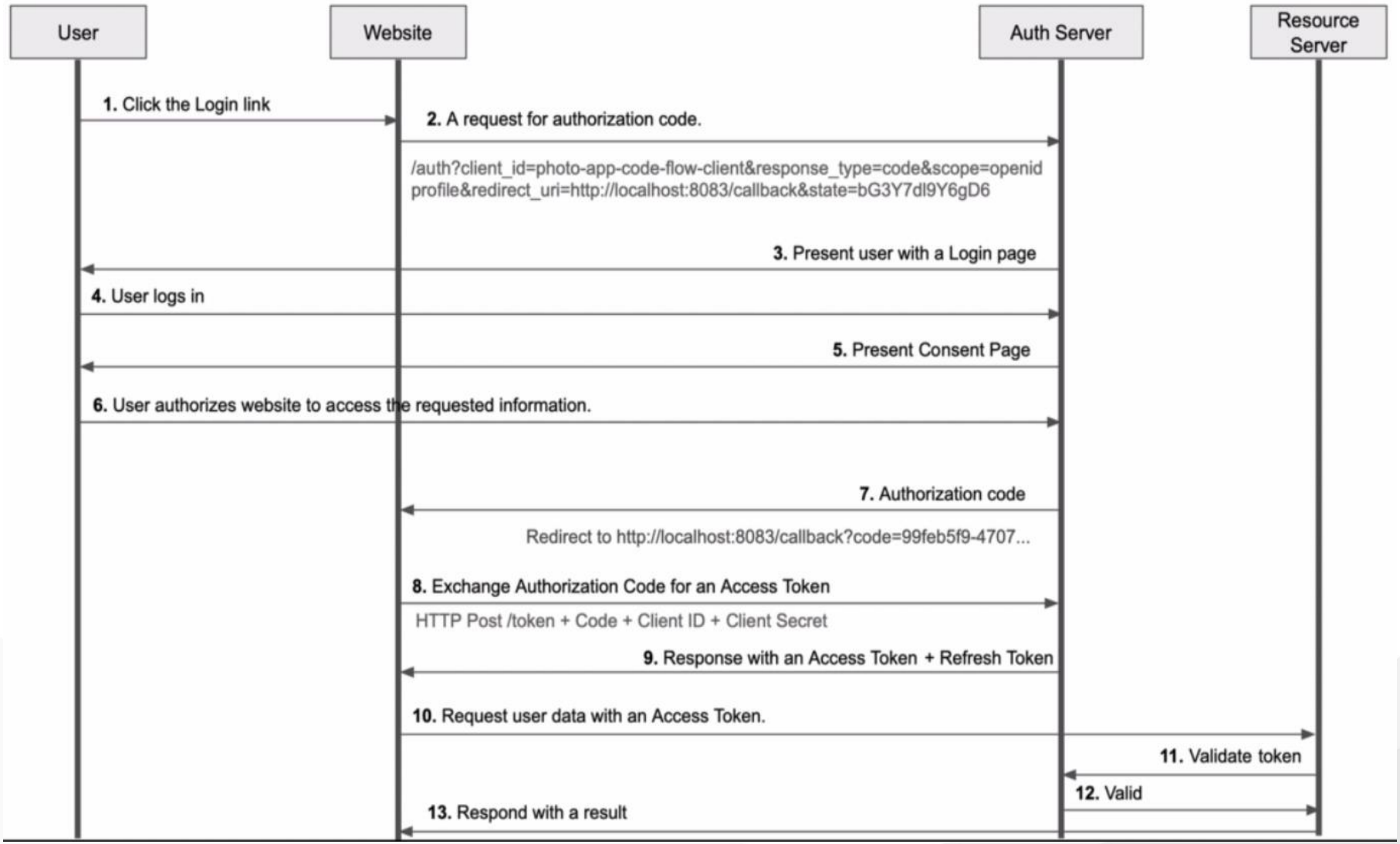
- Refresh Token Grant Type is used to exchange a refresh token for an access token

Authorization Code Grant

Authorization Code Grant



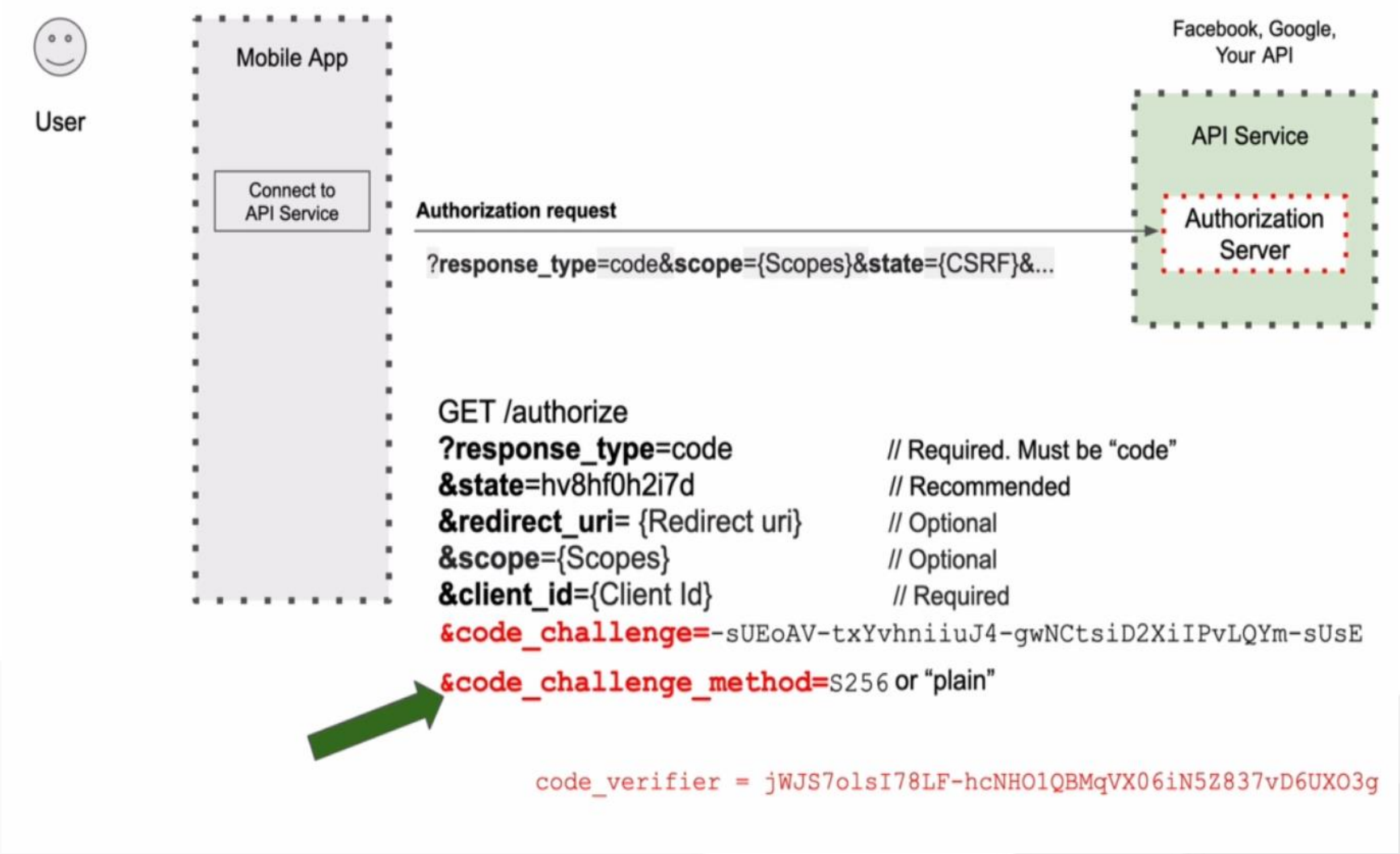
Authorization Code Grant



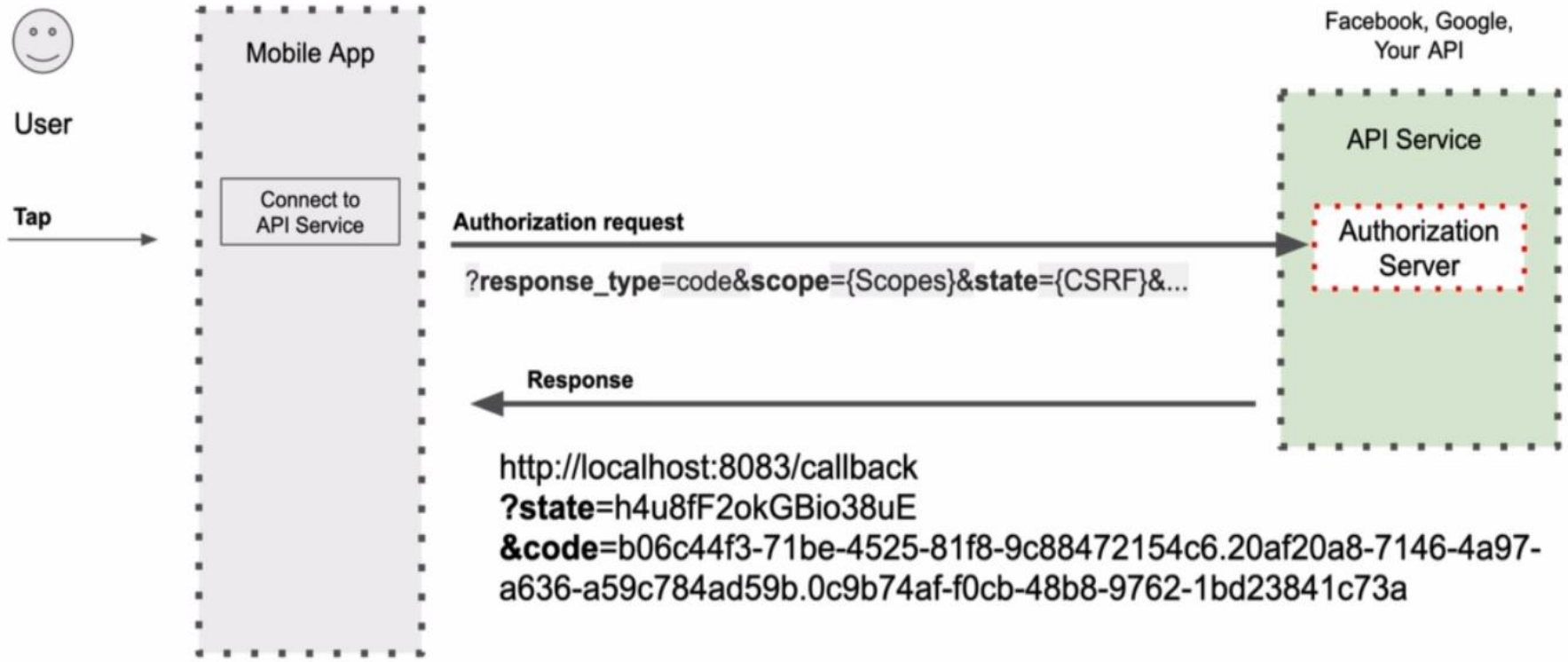
Client Credentials

Password Grant

PKCE Enhanced Authorization Code



PKCE Enhanced Authorization Code



PKCE Enhanced Authorization Code

Request for an OAuth Code

```
curl --location --request GET
'http://localhost:8080/auth/realms/appsdeveloperblog/protocol/openid-connect/auth
?client_id=photo-app-pkce
&response_type=code
&scope=openid
&redirect_uri=http://localhost:8083/callback
&state=h4u8fF2okGBio38uE
&code_challenge=NDEyYjM0YzhkZTZhNWVlMzE3YWVjYmJkZWJiYTg4ZDFhMTIxNjQyMGQwZTU0NjE1NjlmZjMzNTg0NzkwODVlYQ
&code_challenge_method=S256'
```

PKCE Enhanced Authorization Code

Exchange OAuth Code for an Access Token

```
curl --location --request POST
'http://localhost:8080/auth/realms/appsdeveloperblog/protocol
/openid-connect/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=authorization_code' \
--data-urlencode 'client_id=photo-app-pkce' \
--data-urlencode
'code=b06c44f3-71be-4525-81f8-9c88472154c6.20af20a8-7146-4a97
-a636-a59c784ad59b.0c9b74af-f0cb-48b8-9762-1bd23841c73a' \
--data-urlencode
'redirect_uri=http://localhost:8083/callback' \
--data-urlencode
'code_verifier=c3cxd2UzNHJmZGUzNHJneWh1NzhpazFxd2U0cmZkZXI1Nn
l1N3lnZnJ0NmpraW85NHJkc3dlcg'
```

Generating Code Verifier

4. Protocol

4.1. Client Creates a Code Verifier

The client first creates a code verifier, "code_verifier", for each OAuth 2.0 [RFC6749] Authorization Request, in the following manner:

code_verifier = high-entropy cryptographic random STRING using the unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~" from Section 2.3 of [RFC3986], with a minimum length of 43 characters and a maximum length of 128 characters.

ABNF for "code_verifier" is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

NOTE: The code verifier SHOULD have enough entropy to make it impractical to guess the value. It is RECOMMENDED that the output of a suitable random number generator be used to create a 32-octet sequence. The octet sequence is then base64url-encoded to produce a 43-octet URL safe string to use as the code verifier.

Generating Code Verifier

```
package com.appsdeveloperblog.pkce;

import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Base64;

public class PkceUtil {

    String generateCodeVerifier() throws UnsupportedEncodingException {
        SecureRandom secureRandom = new SecureRandom();
        byte[] codeVerifier = new byte[32];
        secureRandom.nextBytes(codeVerifier);
        return
Base64.getUrlEncoder().withoutPadding().encodeToString(codeVerifier);
    }

}
```

5GluDRih4mQPRoG4C4WylsHp0l--aBbOcwGO1MPEfLA

Generating Code Challenge

4.2. Client Creates the Code Challenge

The client then creates a code challenge derived from the code verifier by using one of the following transformations on the code verifier:

plain
   code_challenge = code_verifier

S256 (4 3 2 1)
   code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))

If the client is capable of using "S256", it MUST use "S256", as "S256" is Mandatory To Implement (MTI) on the server. Clients are permitted to use "plain" only if they cannot support "S256" for some technical reason and know via out-of-band configuration that the server supports "plain".

The plain transformation is for compatibility with existing deployments and for constrained environments that can't use the S256 transformation.

ABNF for "code_challenge" is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

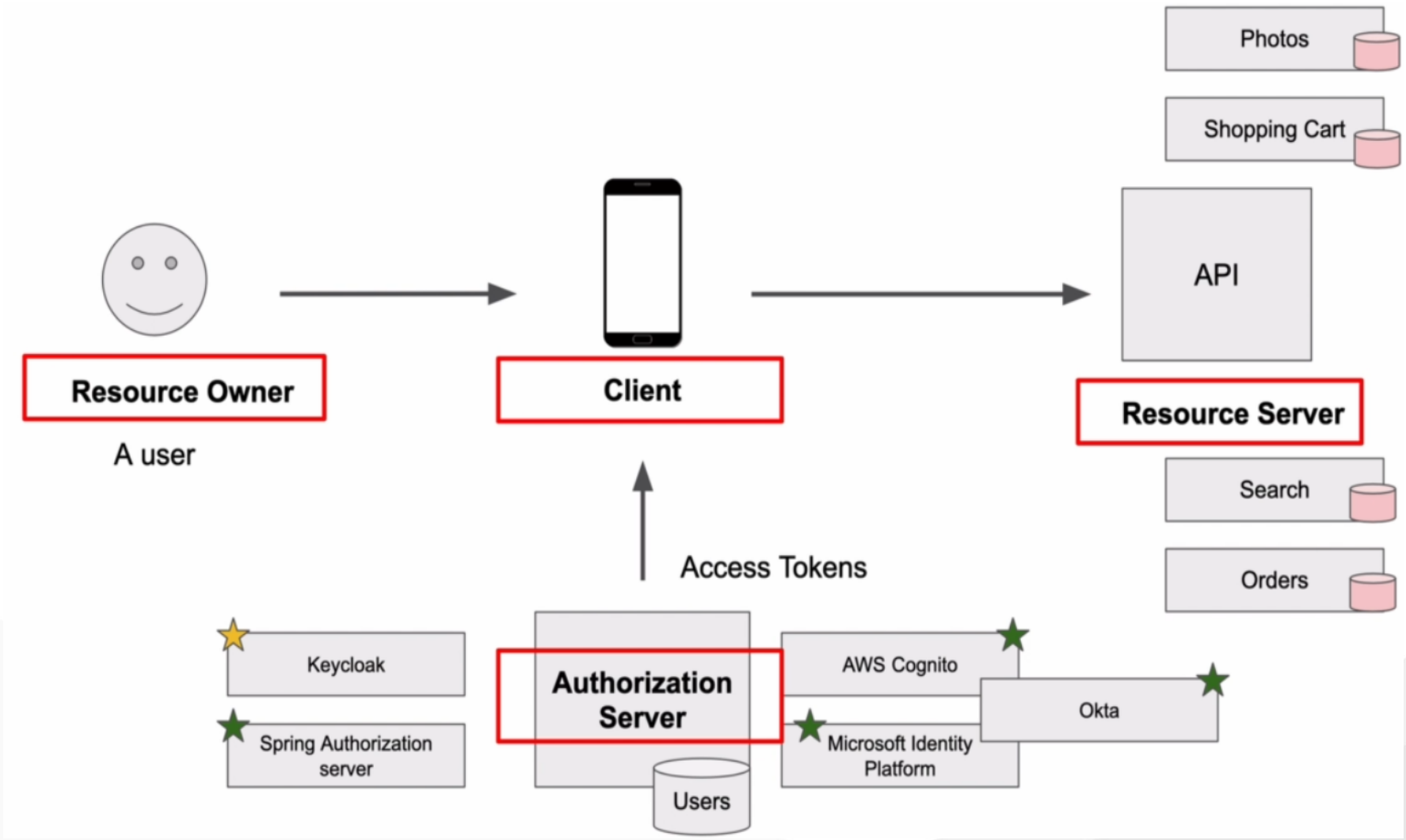
Generating Code Challenge

```
    String generateCodeChallange(String codeVerifier) throws
UnsupportedEncodingException, NoSuchAlgorithmException {

        byte[] bytes = codeVerifier.getBytes("US-ASCII");
        MessageDigest messageDigest =
MessageDigest.getInstance("SHA-256");
        messageDigest.update(bytes, 0, bytes.length);
        byte[] digest = messageDigest.digest();

        return
Base64.getUrlEncoder().withoutPadding().encodeToString(digest);

    }
```

What is Keycloak?

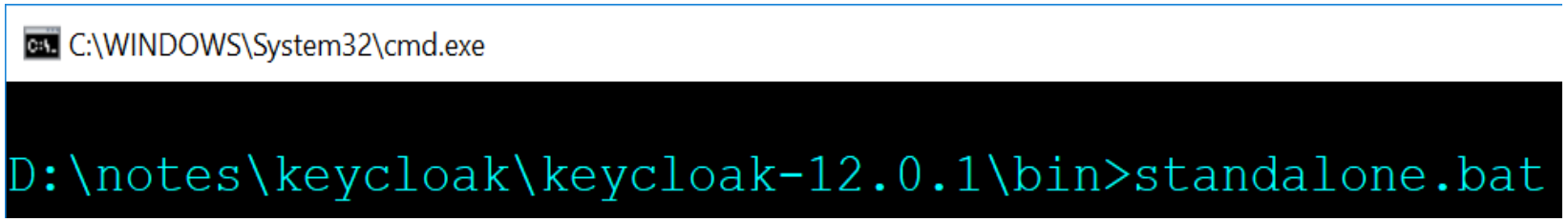
- Keycloak is an open source Identity and Access Management server that supports OpenID Connect and OAuth 2.0 authorization flows.
- Supports Single-Sign On (SSO) and Sign out.
- Social Login like facebook, google, github or twitter.
- Keycload can also authenticate users with existing OpenID Connect or SAML identity provider.
- User Federation – If your organization has an existing user database, then it can also be integrated with keycloak. Keycloak supports LDAP, Active Directory servers. We can also implement our own provider and integrate keycloak with existing MySQL database, for example.
- Provides web interface to manage users of applications as well as with REST APIs that you can use to add new and manage existing users.
- Keycloak has very user-friendly user interface that allows you to manage client applications, users, their credentials and roles. You can configure what grant type supported by your application, generate new client secret key, create client scopes and define fine-grained authorization policies and so on.

What is Keycloak?

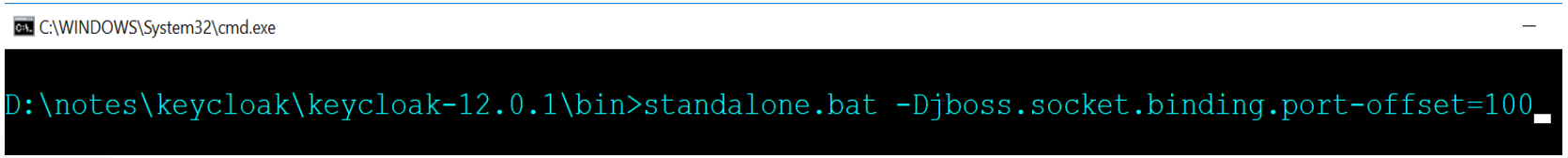


Installation and setting up Keycloak server

- Go to https://www.keycloak.org/downloads
- Download **keycloak** for respective OS.
- Go to bin folder and open **CMD.** Execute below command (Windows OS)



- For Linux and MacOS, execute **standalone.sh** file.
- You can access keycloak server using **http://<IP_ADDRESS>:8080/**
- By default, keycloak server runs on port 8080 but you can change it.



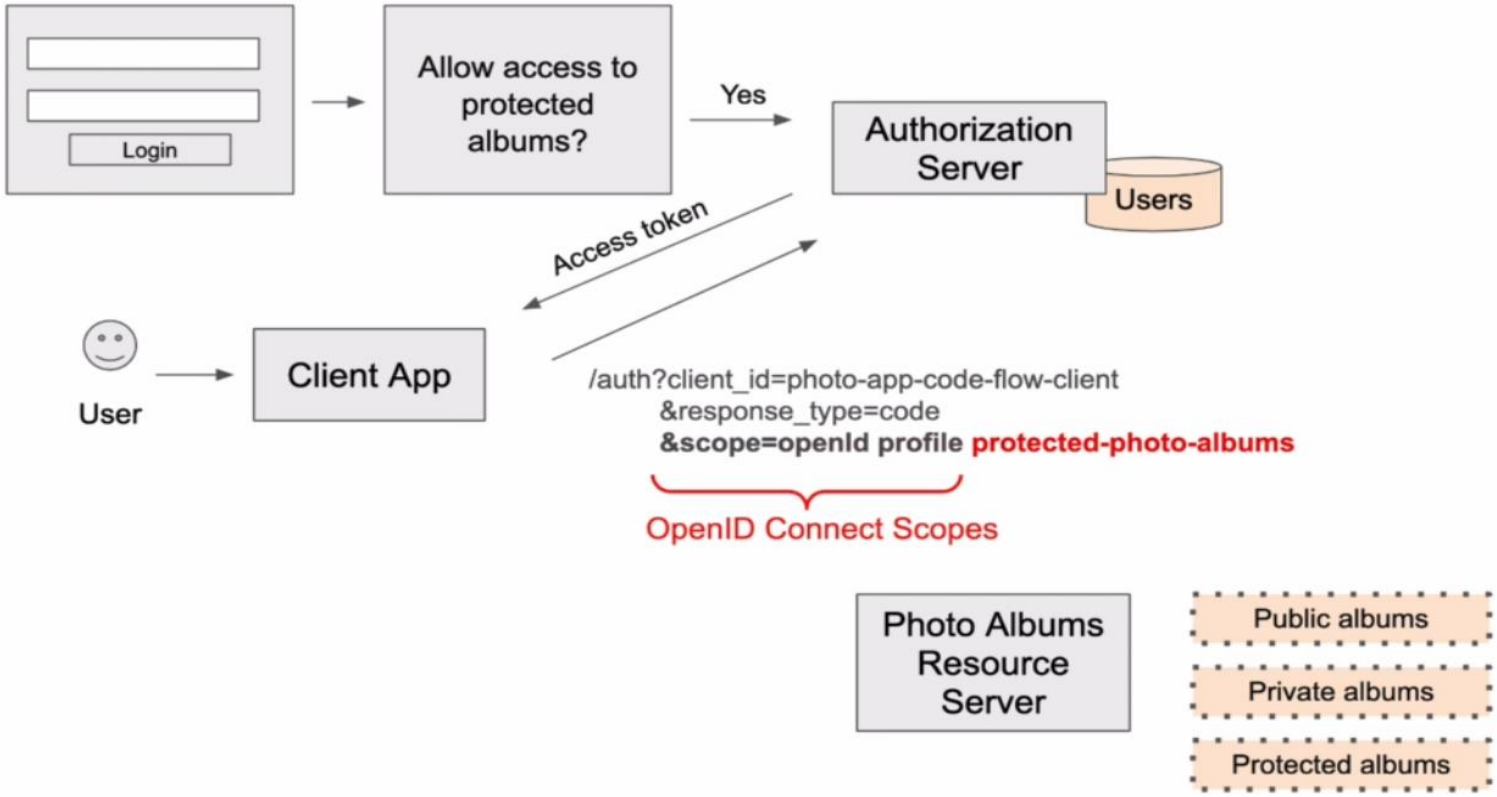
- Now, you can access keycloak server on port **8180** (8080 + 100 offset).
- We can also run keycloak as docker container.

What OAuth 2.0 isn’t

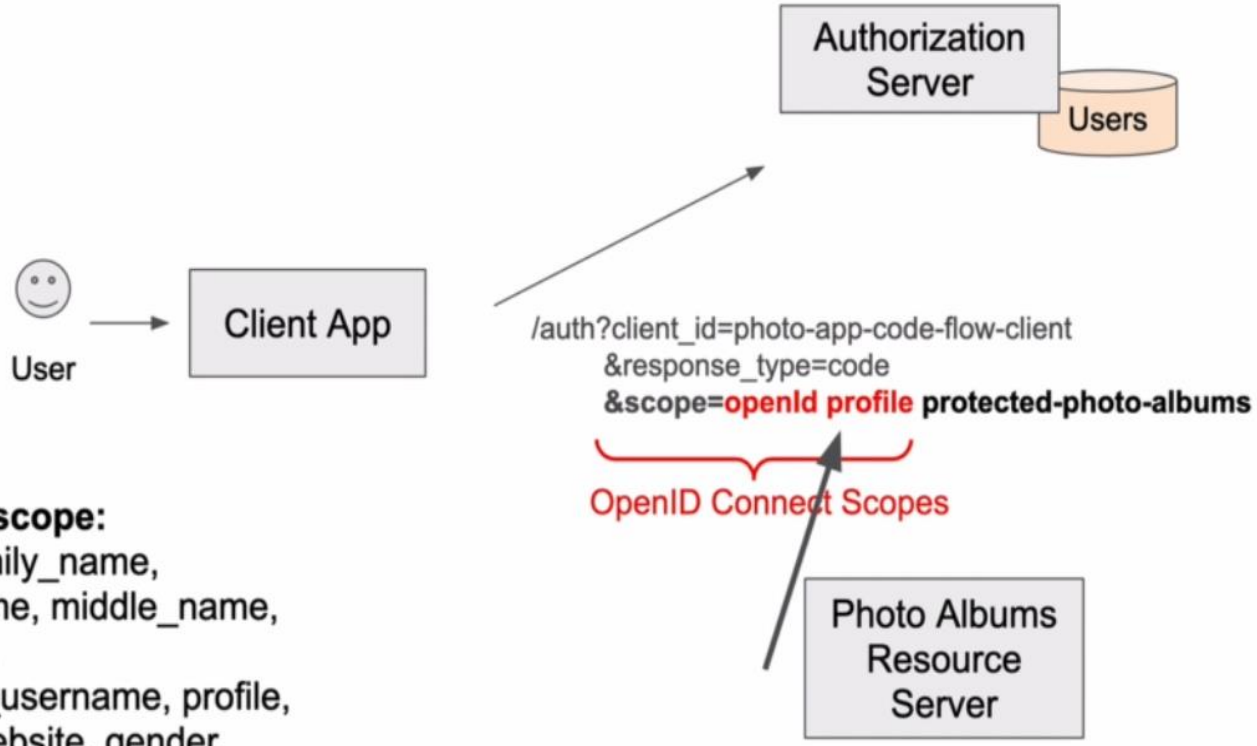
- OAuth isn’t defined outside of the HTTP protocol. HTTPS is a must.
- OAuth isn’t an authentication protocol.
- OAuth doesn’t define authorization-processing mechanism.
- OAuth doesn’t define a Token format but the content of the token is completely opaque to the client application
- OAuth doesn’t define cryptographic method.

Resource Server - Scope Based Access Control

- Scope is a mechanism in OAuth 2.0 to limit application's access to a user's account.
- An application can request one or more scopes, this information is then presented to the user in the consent screen, and the access token issued to the application will be limited to the scopes granted.



Resource Server - Scope Based Access Control



“profile” scope:
name, family_name,
given_name, middle_name,
nickname,
preferred_username, profile,
picture, website, gender,
birthdate, zoneinfo, locale, and
updated_at

Resource Server - Scope Based Access Control

```
/auth?client_id=photo-app-code-flow-client
      &response_type=code
      &scope=openId profile email address phone offline_access
```

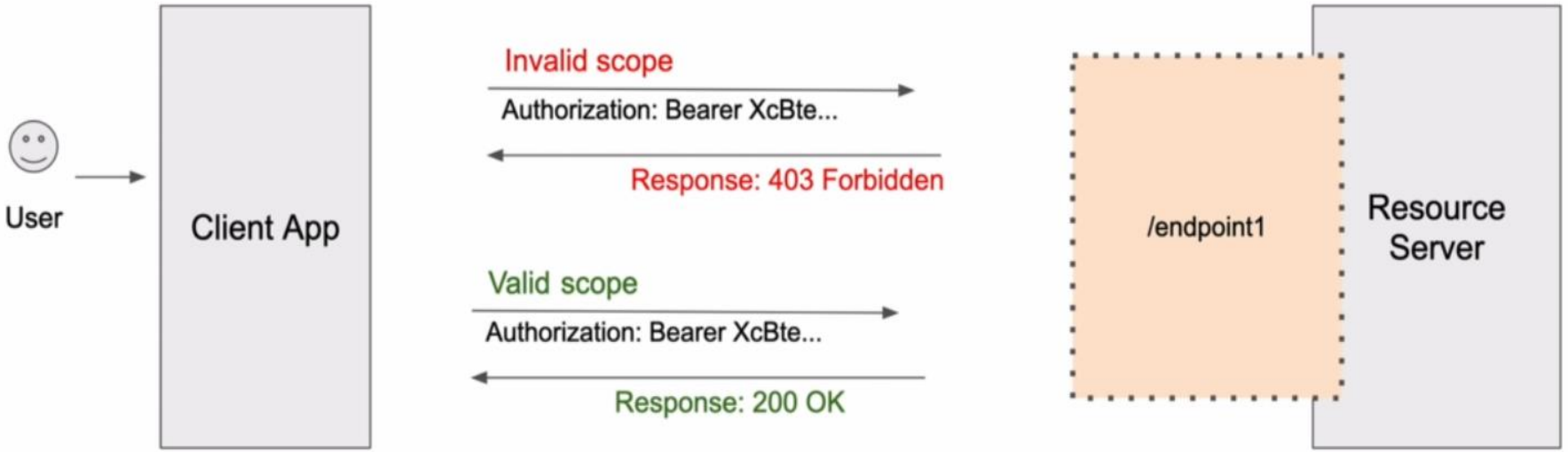
“profile” scope:
name, family_name,
given_name,
middle_name,
nickname,
preferred_username,
profile,
picture,
website,
gender,
birthdate,
zoneinfo,
locale, and
updated_at

“email” scope:
email
email_verified

“address” scope:
formatted,
street_address
locality
region
postal_code
country

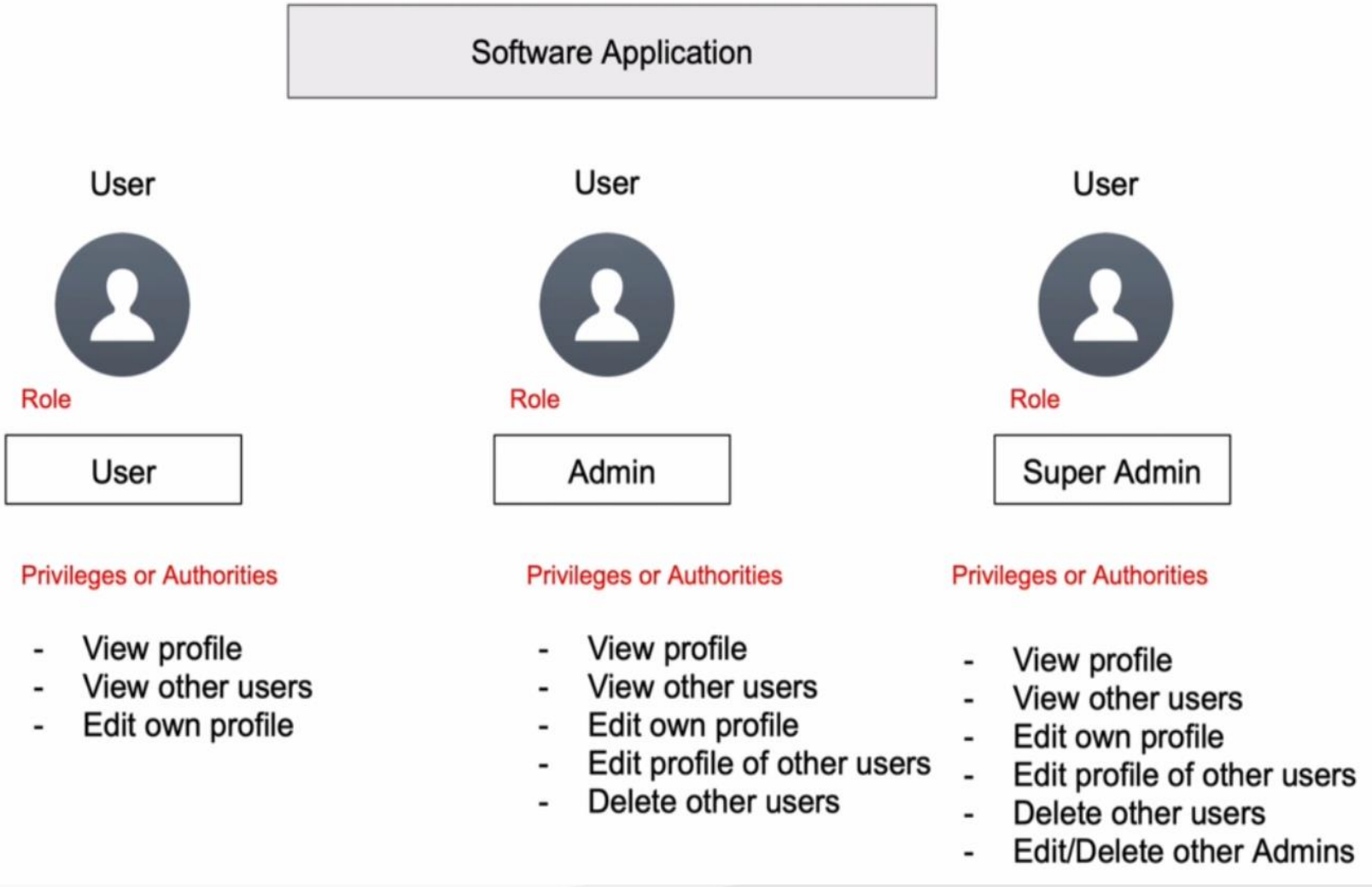
“phone” scope:
phone,
phone_number_verified

Resource Server - Scope Based Access Control



Role Based Access Control using Keycloak

- **Roles vs Authority**



Role Based Access Control using Keycloak

- **Roles vs Authority**

Authority name = Role Name = ROLE_ADMIN

hasRole("ADMIN")

hasAuthority("ROLE_ADMIN")

Collection<GrantedAuthority>

ROLES:

ROLE_USER,
ROLE_ADMIN,
ROLE_DBADMIN.

AUTHORITIES:

READ,
WRITE,
DELETE.

Method Level Security

```
@Secured("DELETE_AUTHORITY")
@Transactional
public ResponseEntity deleteUser(@PathVariable String id) {

    // some code here

}
```

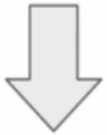
Method Level Security

```
@PreAuthorize("hasAuthority('DELETE_AUTHORITY') or #id == principal.userId")
@DeleteMapping(path = "/{id}")
public ResponseEntity deleteUser(@PathVariable String id) {
        ...
}
```

Method Level Security

```
@PreAuthorize("hasAuthority('DELETE_AUTHORITY') or #id == principal.userId")
@DeleteMapping(path = "/{id}")
public ResponseEntity deleteUser(@PathVariable String id) {
        ...
}
```

Method Level Security

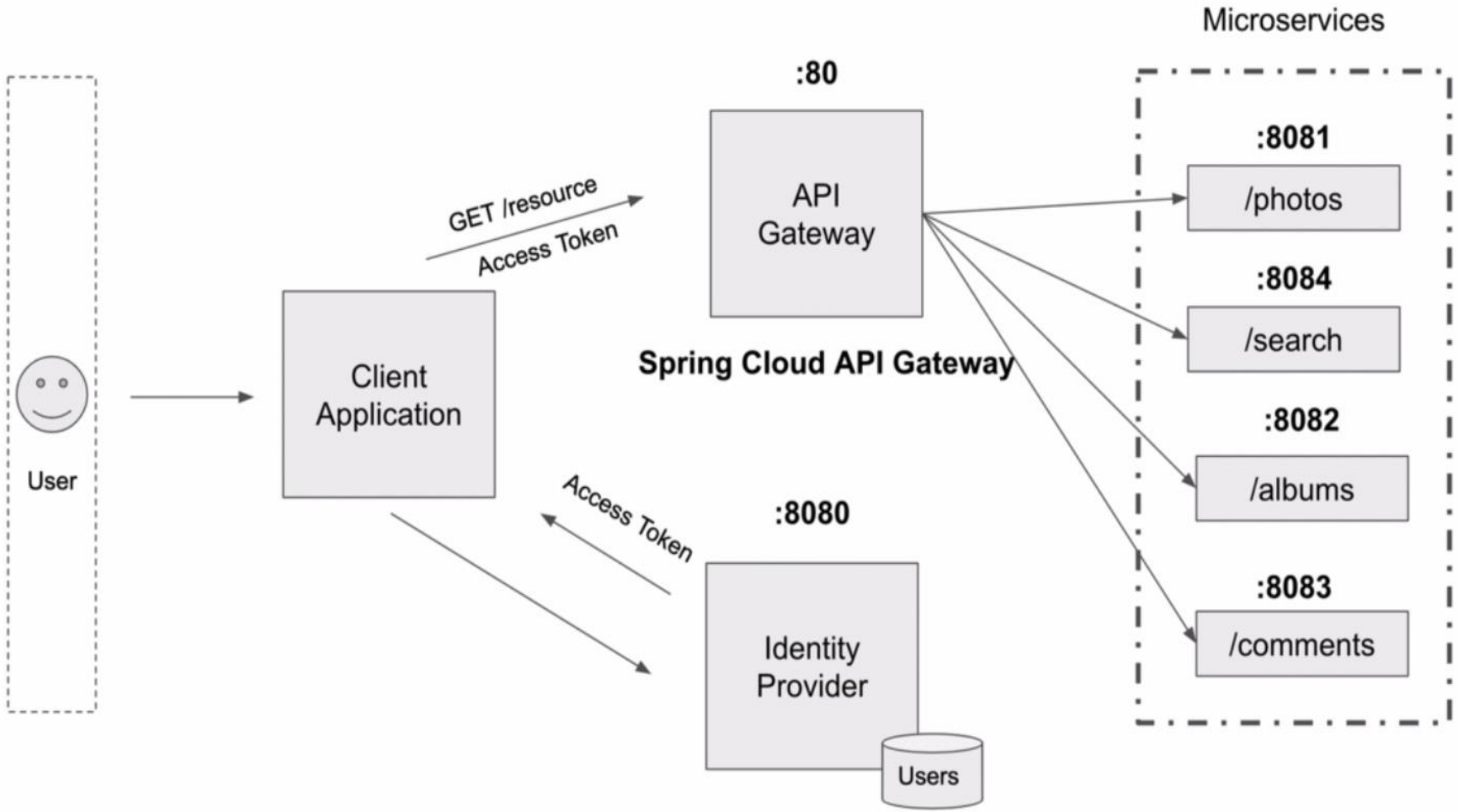


```
@PostAuthorize("hasRole('ADMIN') or returnObject.userId == principal.userId")
@GetMapping(path = "/{id}")
public UserRest getUser(@PathVariable String id) {
        UserRest returnValue = new UserRest();

        // some code here

        return returnValue;
}
```

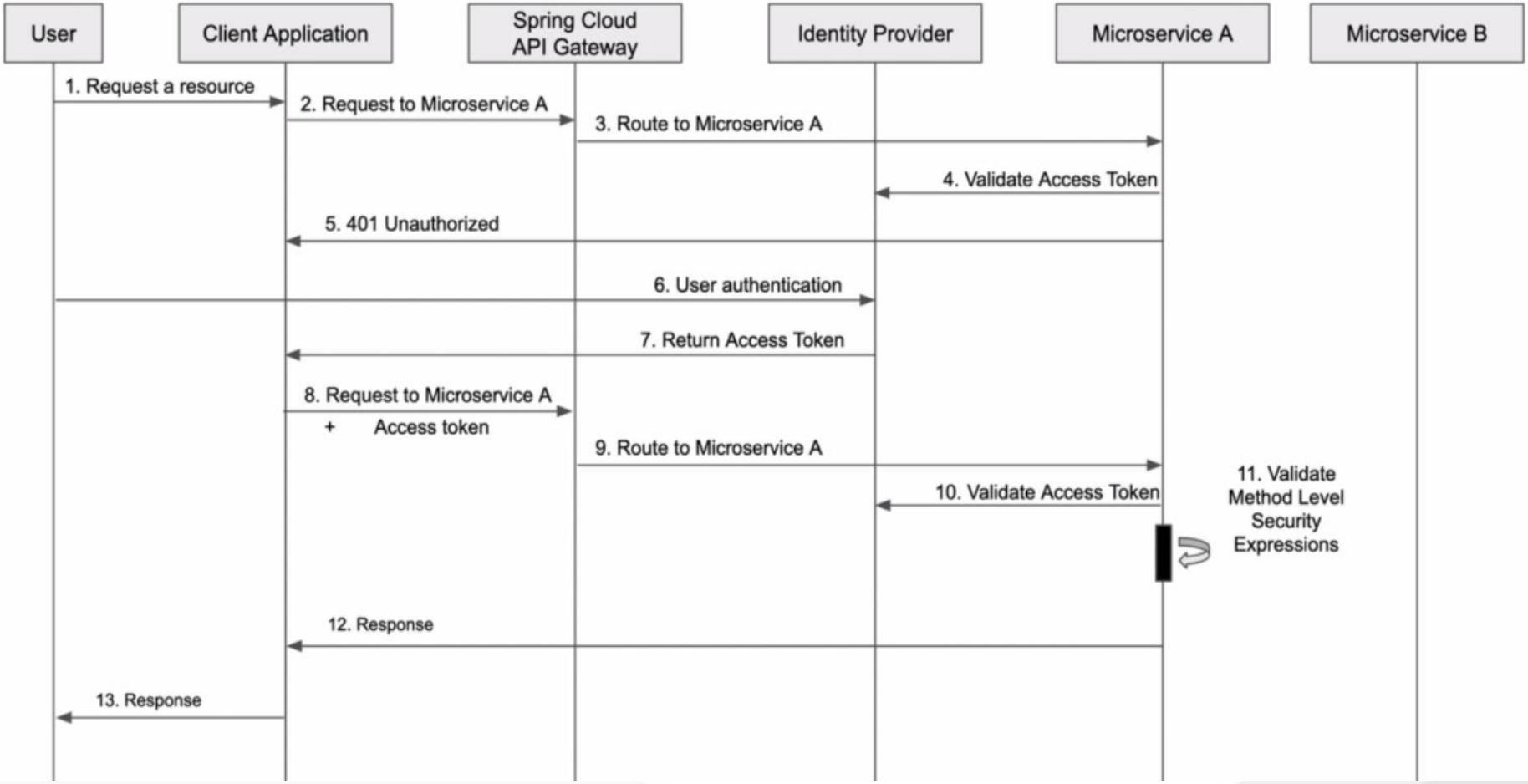
Method Level Security

```
@RestController
@Secured("ROLE_ADMIN")
public class UsersController {

   @PreAuthorize("permitAll")
   @GetMapping("/users/status/check")
   public String usersStatusCheck() {
      return "Working for users";
   }

   @GetMapping("/managers/status/check")
   public String managersStatusCheck() {
      return "Working for managers";
   }
}
```

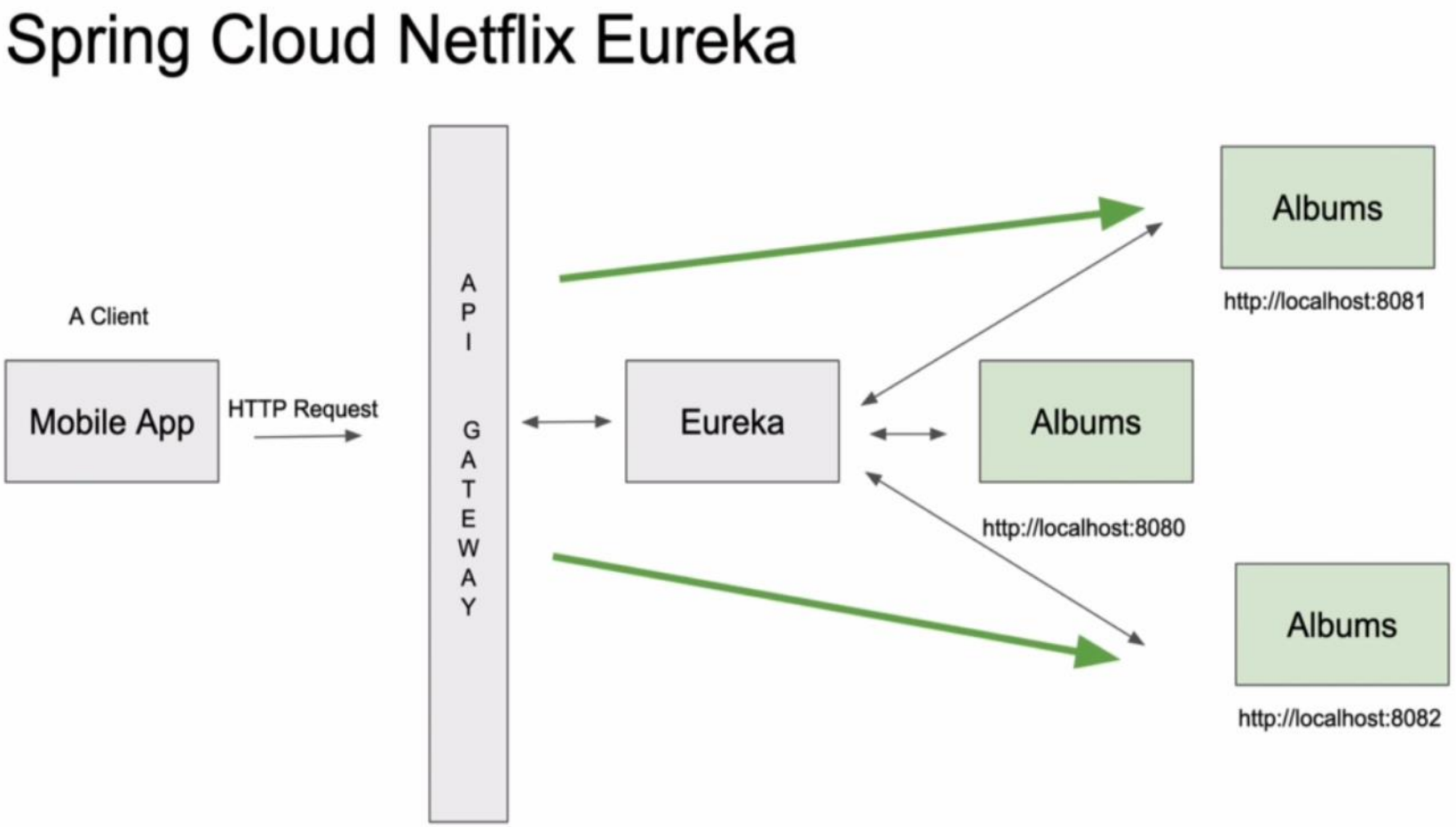
API Gateway


Microservices
:80
:8081
/photos
GET /resource
Access Token
API
Gateway
:8084
/search
Spring Cloud API Gateway
User
Client
Application
:8082
/albums
Access Token
:8080
:8083
/comments
Identity
Provider
Users

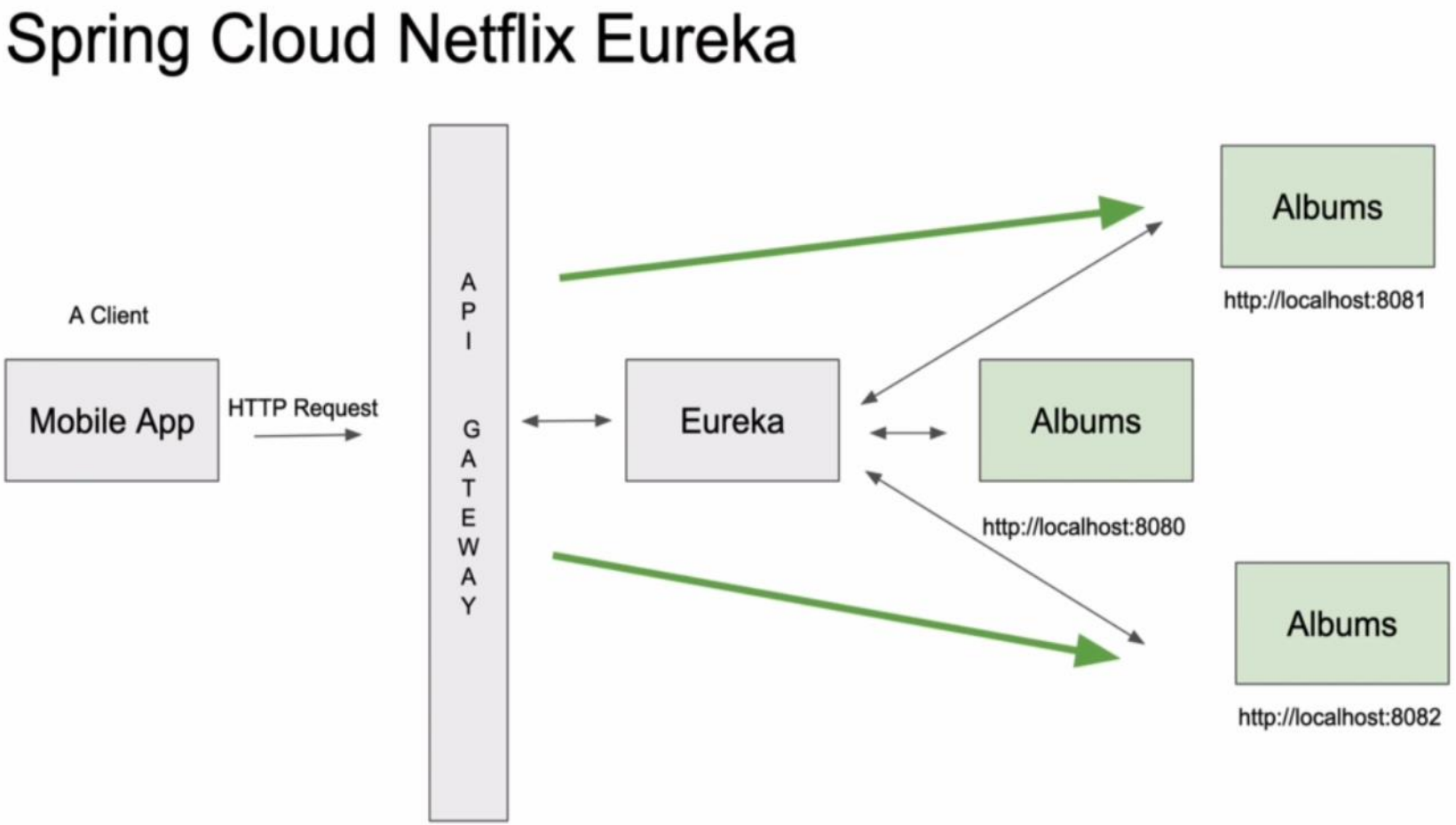
API Gateway


User
Client Application
Spring Cloud API Gateway
Identity Provider
Microservice A
Microservice B
1. Request a resource
2. Request to Microservice A
3. Route to Microservice A
4. Validate Access Token
5. 401 Unauthorized
6. User authentication
7. Return Access Token
8. Request to Microservice A
+ Access token
9. Route to Microservice A
10. Validate Access Token
11. Validate Method Level Security Expressions
12. Response
13. Response

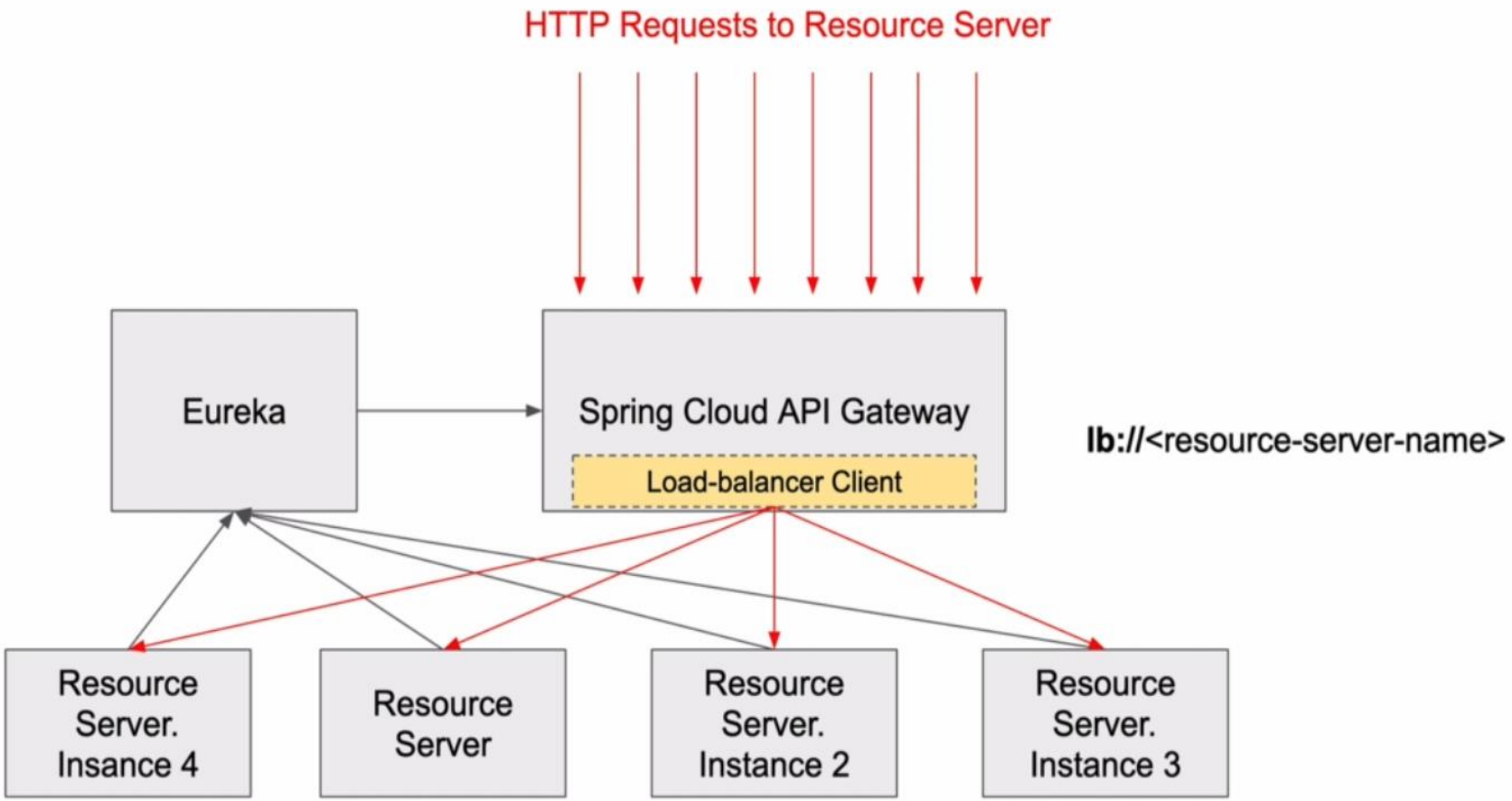
Eureka Discovery Server



Eureka Discovery Server



Load Balancing



Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email ct-univerct@citiustech.com

www.citiustech.com