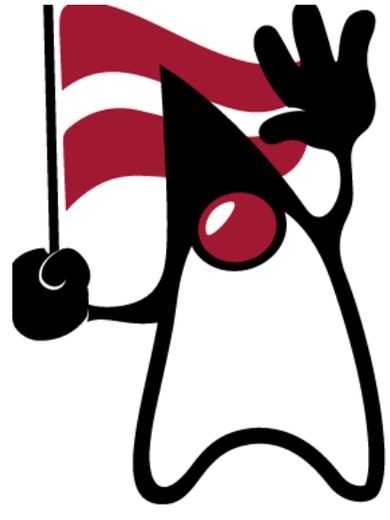
MODERN SECURITY WITH OAUTH 2.0 AND JWT AND SPRING

Dmitry Buzdin

03.11.2016



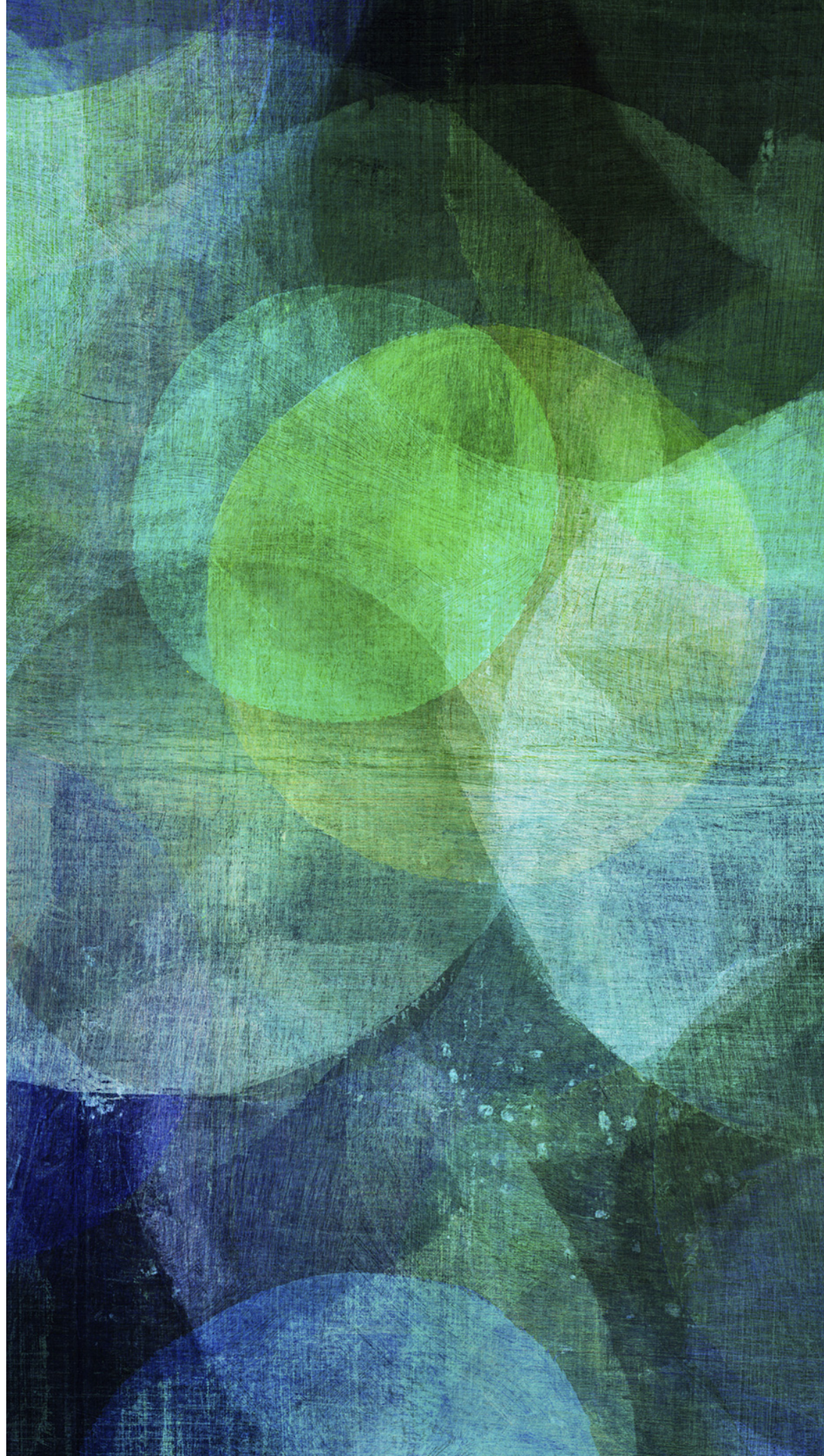
JUG
Java User Group
Latvia

AGENDA

- Single-sign on
- OAuth 2.0
- JSON Web Tokens
- Some Spring examples
- You will learn what is it and why you need that

OAUTH 2.0

Explained



SECURITY MATTERS

- Every app needs security
- Basic security knowledge is a must
- Developers are ignoring security sometimes
- Security is based on standards - do not invent stuff!

SINGLE SIGN-ON

- Accessing multiple systems with single id and password
- Centralised control of access rights
- Well known protocols
  - LDAP
  - Kerberos
  - SAML 2.0
  - OpenID
  - **OAuth 2.0**

WHY YOU NEED SSO?

- Internal applications with one corporate login
- Integration with platform as a service
- Web sites with business affiliates
  - Partner sites
  - Mobile apps
  - Third-party plugins

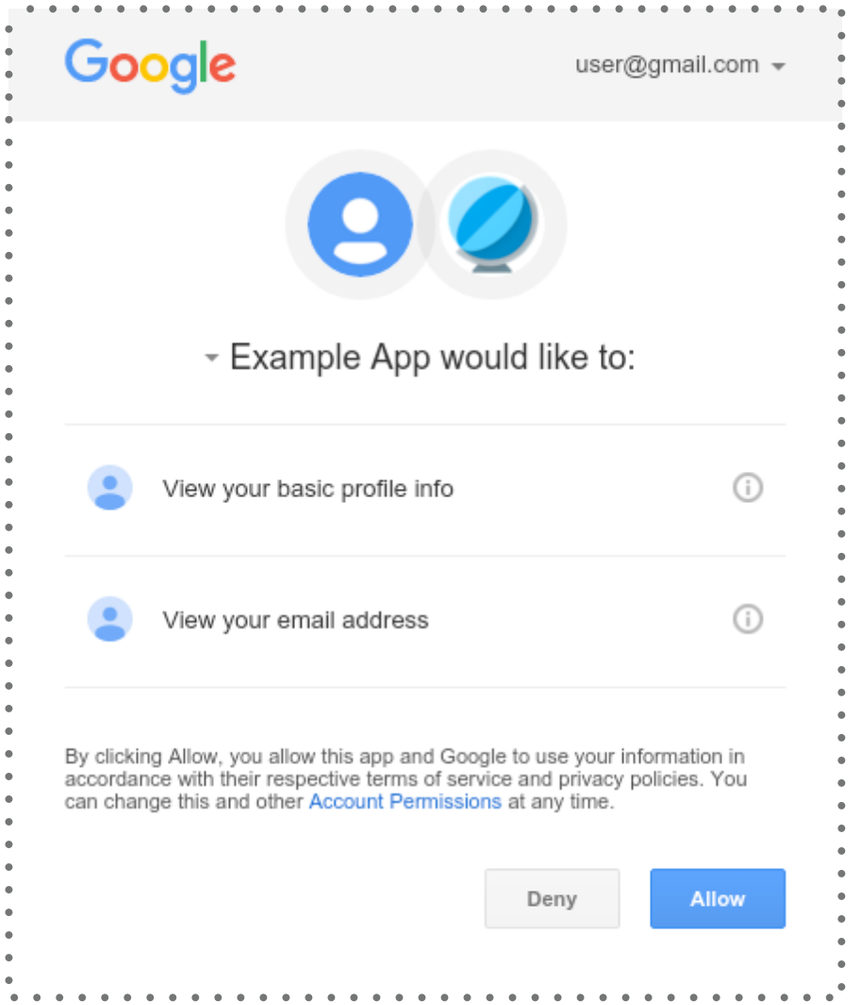
OAUTH 2.0

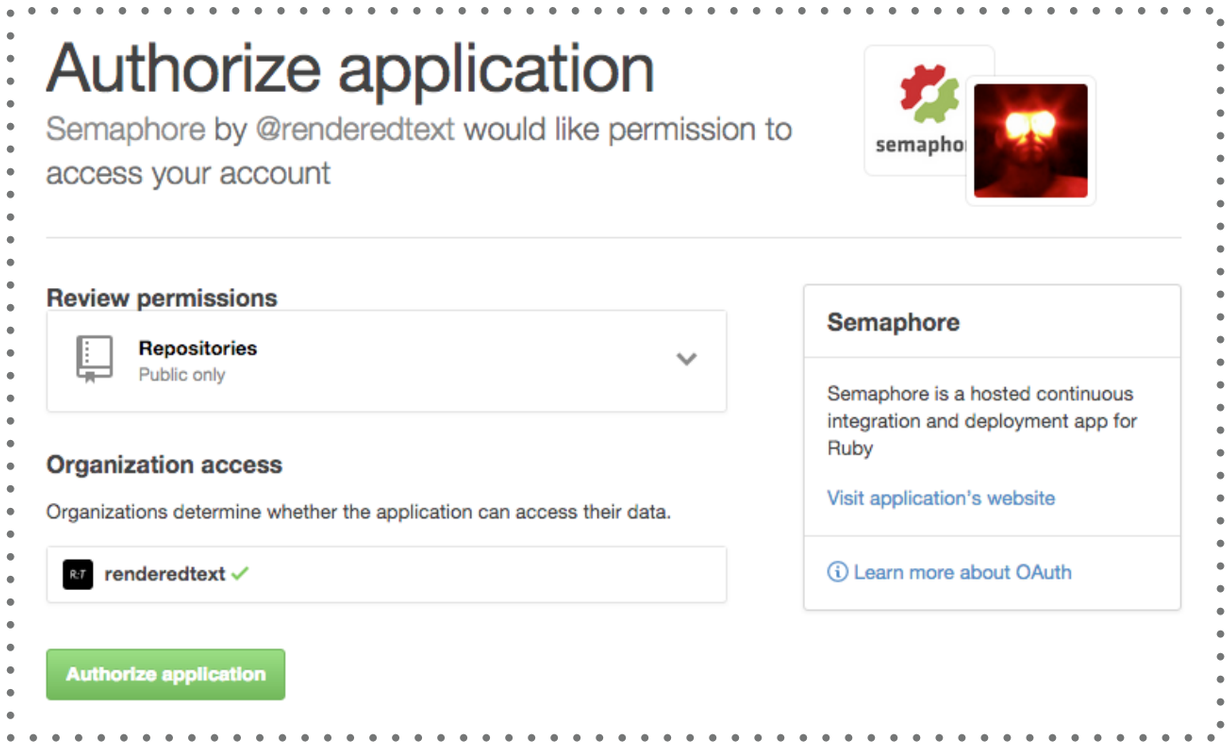
- OAuth is an open standard for authorization, commonly used as a way for Internet users to authorize websites or applications to access their information on other websites but without giving them the passwords
- Standard published in October 2012
- Open and cross-platform

WHO USES OAUTH 2.0

- GitHub
- Google
- Facebook
- DigitalOcean
- etc.

HAVE YOU SEEN THESE PAGES?





OAUTH 2.0 OPEN STANDARD

Internet Engineering Task Force (IETF)
Request for Comments: 6749
Obsoletes: 5849
Category: Standards Track
ISSN: 2070-1721

D. Hardt, Ed.
Microsoft

October 2012

The OAuth 2.0 Authorization Framework

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

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

OAUTH 2.0 COMPONENTS

Resource Owner

Client

Authorisation Server

Resource Server

RESOURCE OWNER

> resource owner
>
> An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

- Basically a user
- Could be technical user as well
- Owns resources on the resource server

CLIENT

```
client
   An application making protected resource requests on behalf of the
   resource owner and with its authorization.  The term "client" does
   not imply any particular implementation characteristics (e.g.,
   whether the application executes on a server, a desktop, or other
   devices).
```

- Third-party application
- Could be trusted or not-trusted
- Wants to access resources on Resource Server

AUTHORIZATION SERVER

```
authorization server
   The server issuing access tokens to the client after successfully
   authenticating the resource owner and obtaining authorization.
```

- Centralised security gateway
- Issues access tokens
- Knows user credentials

RESOURCE SERVER

```
resource server
   The server hosting the protected resources, capable of accepting
   and responding to protected resource requests using access tokens.
```

- Application expecting requests with authorised tokens
- There could be many resource servers

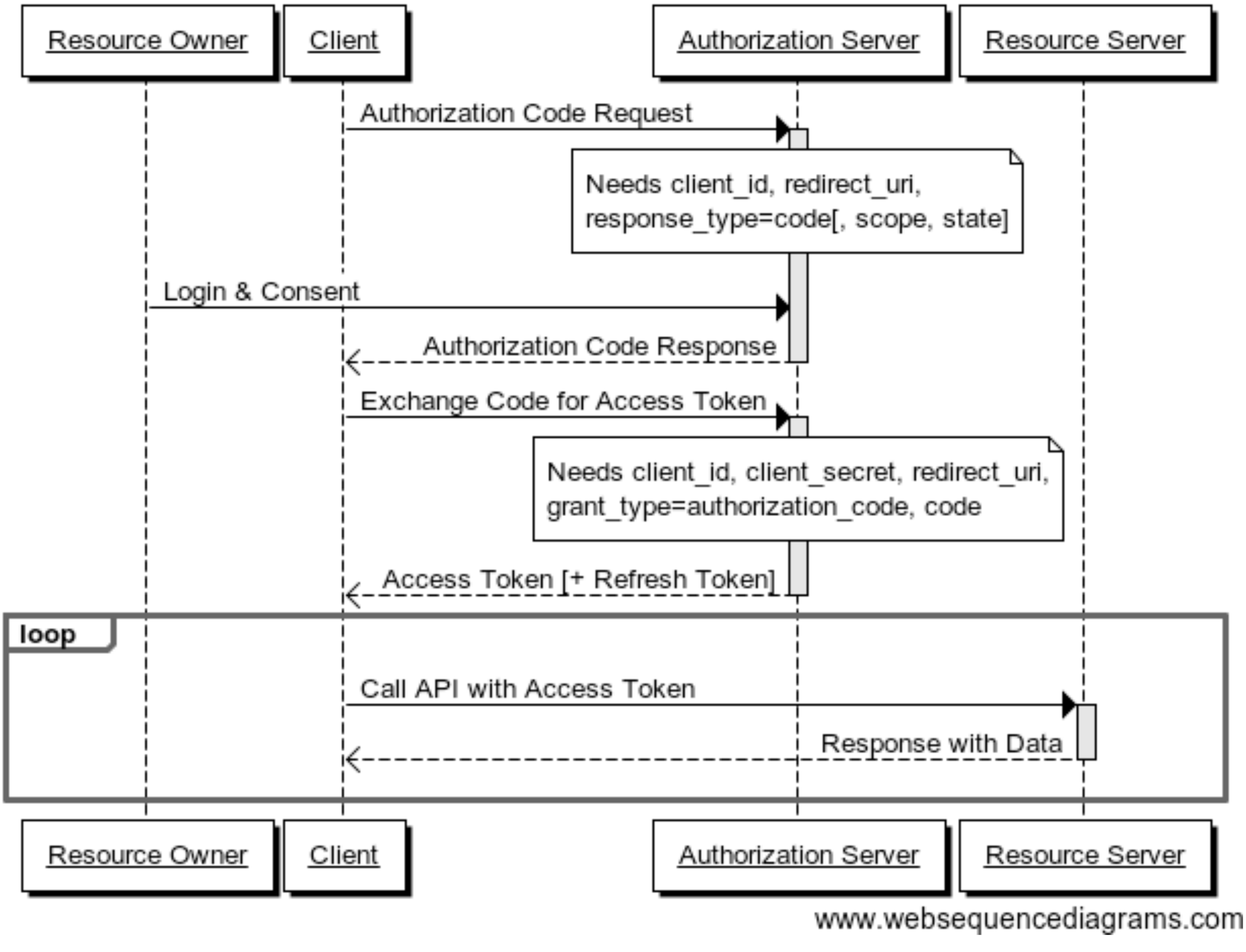
CLIENT REQUIRES ACCESS TOKEN TO RETRIEVE RESOURCES

AUTHORIZATION GRANT TYPES

- Access token is granted upon authorization
- There are following standard grant types:
  - Authorization Code Grant
  - Resource Owner Password Credentials
  - Client Credentials
  - Implicit Grant

http://bshaffer.github.io/oauth2-server-php-docs/overview/grant-types/

Authorization Code Grant Flow



AUTHORIZATION CODE GRANT

- User is not entering credentials in client app, but in auth server authorisation page
- Auth server redirects back to with auth code
- Auth code is exchanged for access token
- Auth code is short-lived
- Access token is used for requests to resource server

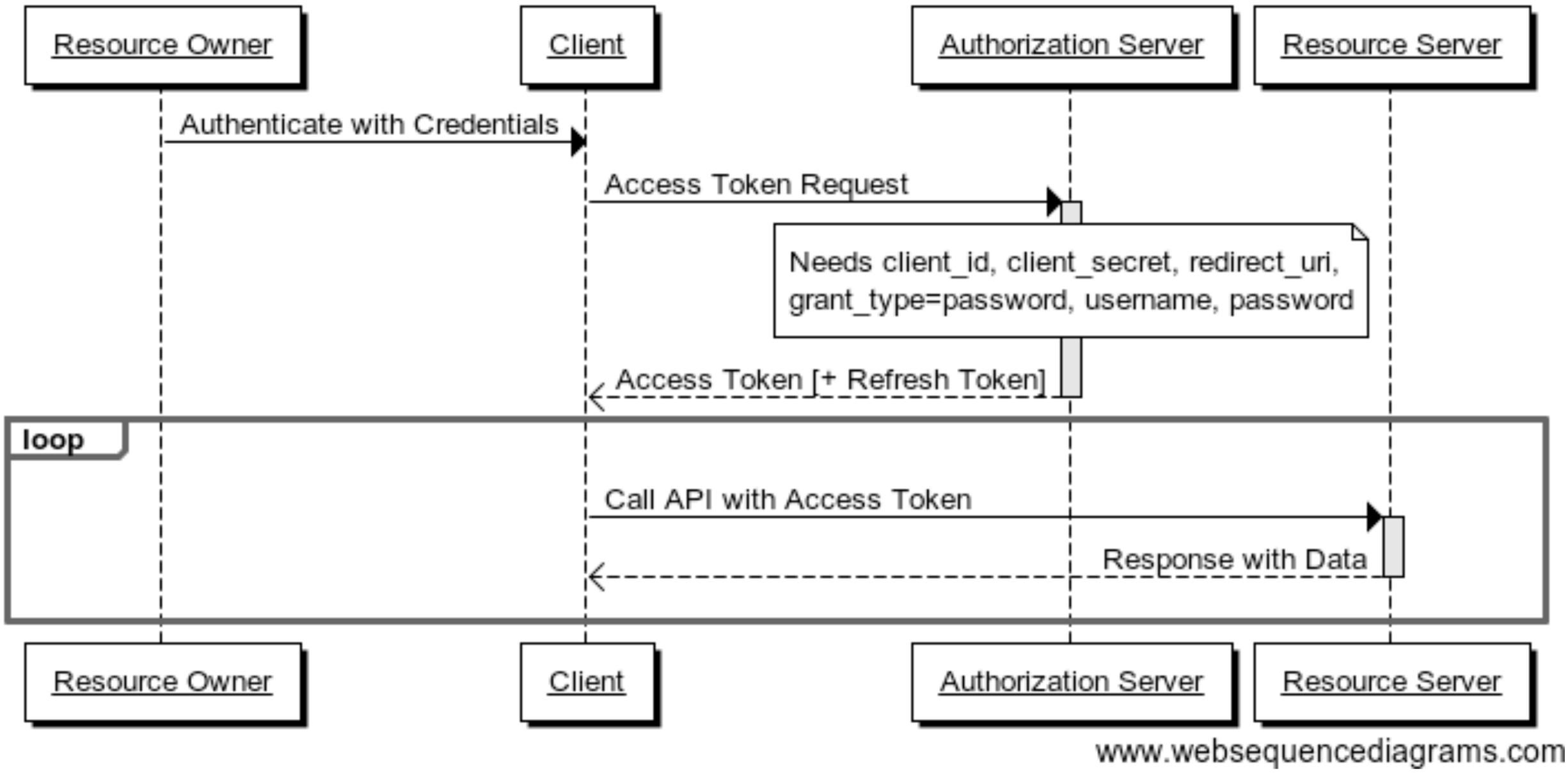
AUTHORISATION CODE GRANT HTTP

```
GET /authorize?response_type=code
        &client_id=123
        &scope=view_profile
       &redirect_uri=https://partner.com/oauth

302 REDIRECT https://partner.com/oauth
  &code=9srN6sqmjrvG5bWvNB42PCGju0TFVV

POST /token?code=9srN6sqmjrvG5bWvNB42PCGju0TFVV
        &grant_type=authorization_code
        &client_id=123
       &redirect_uri=https://partner.com/oauth
```

Resource Owner Password Credentials Grant Flow

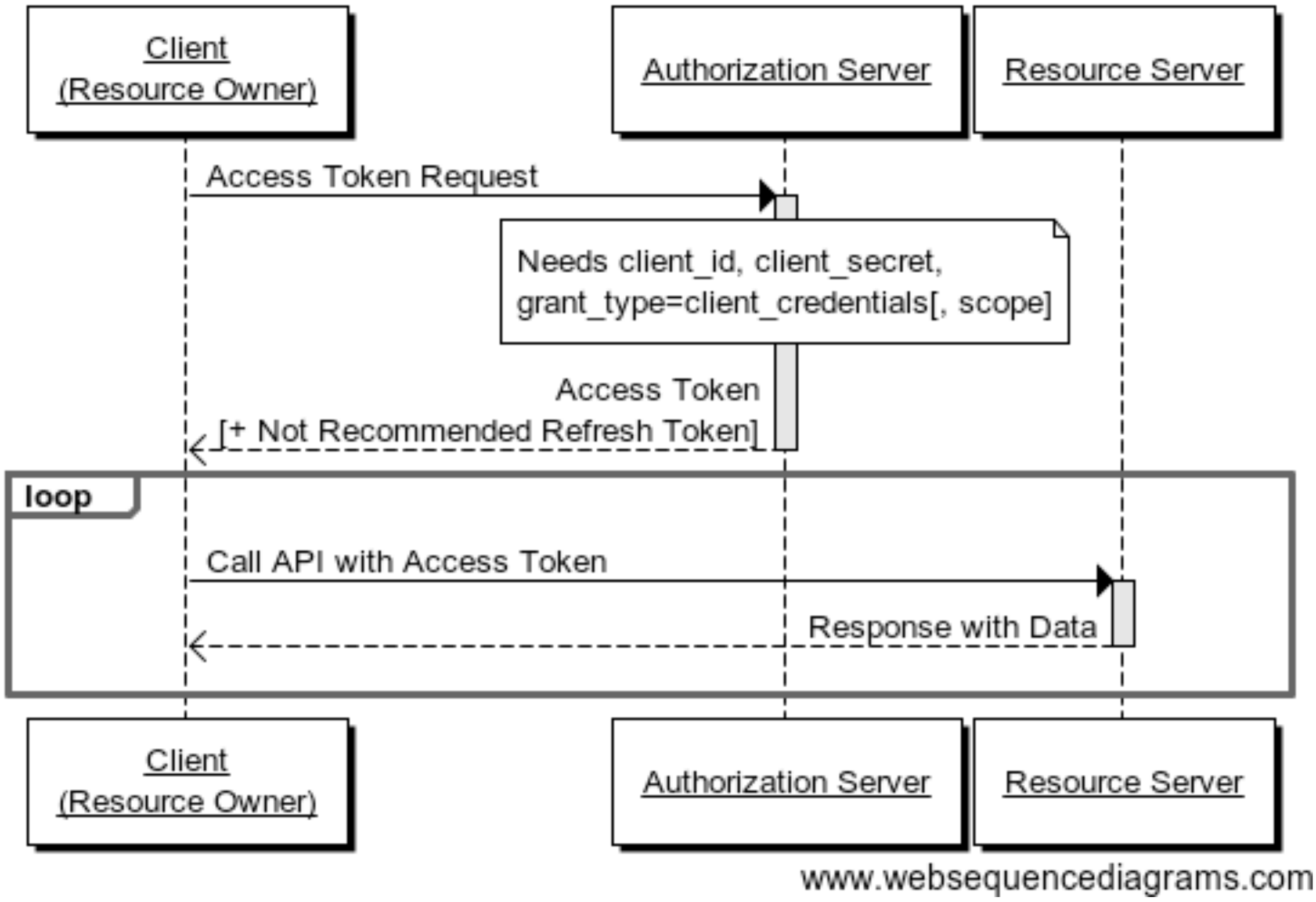


RESOURCE OWNER PASSWORD GRANT

- Trusted client, has access to resource owner credentials
- Less secure as there is a “middleman”
- Could be used for subdomains in one organization

```
POST /authorize?grant_type=password
        &username=code
        &password=password
        &client_id=123
        &client_secret=secret
```

Client Credentials Grant Flow

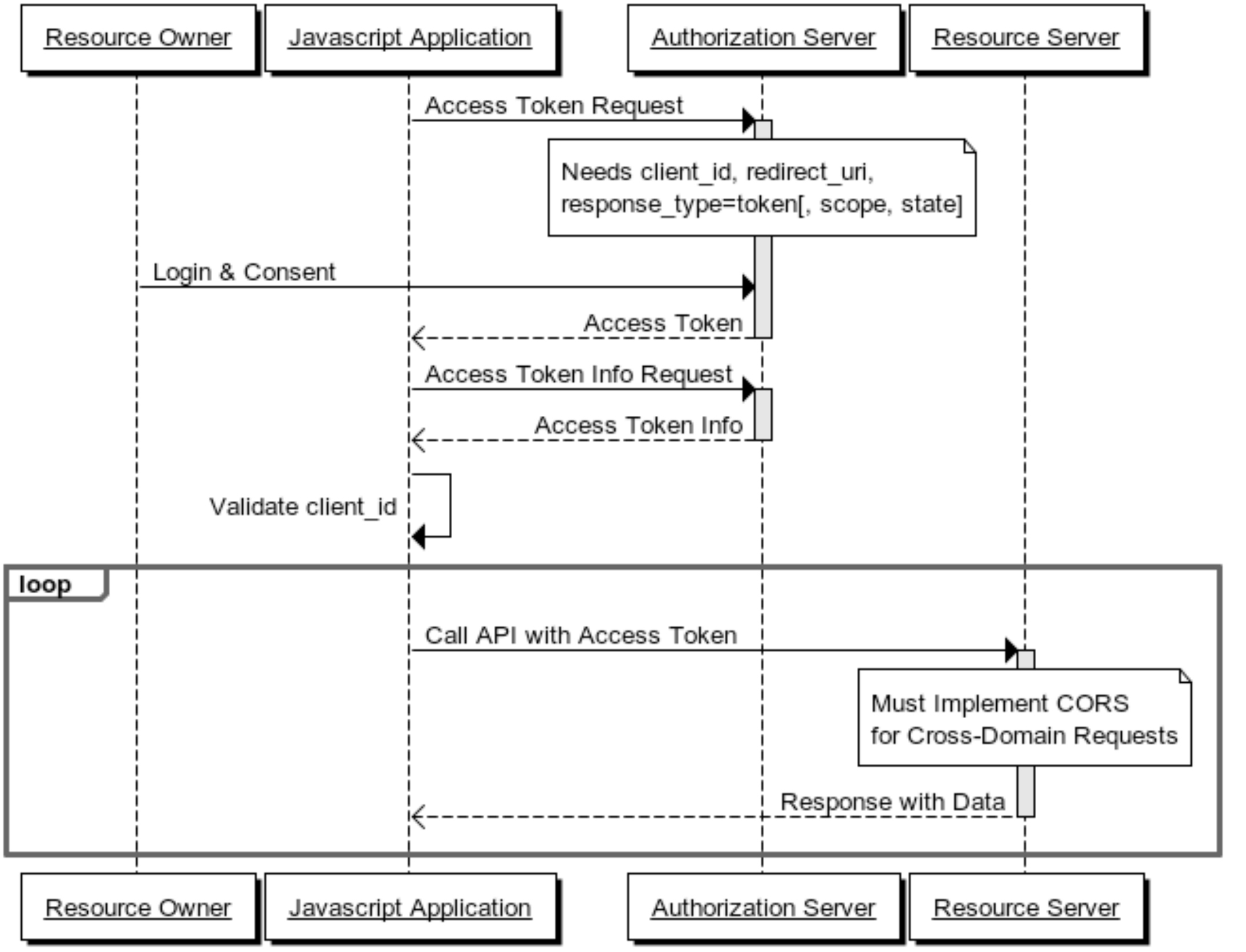


CLIENT CREDENTIALS GRANT

- Client is sending its own password directly
- Used in a situation when the client is the resource owner
- Again, less secure option

```
POST /authorize?grant_type=client_credentials
        &client_id=123
        &client_secret=secret
```

Implicit Grant Flow



www.websequencediagrams.com

IMPLICIT GRANT

- Used in JavaScript front-ends
- Does not allow the issuance of a refresh token
- Requires Cross-Origin Resource Sharing (CORS)
- Least secure, access token is available in the client
- Exposure to Cross-site Request Forgery (XSRF) attack

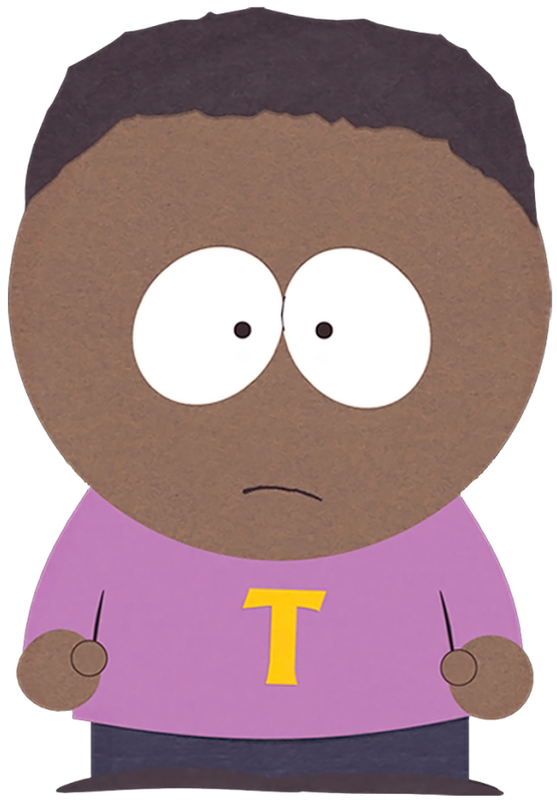
IMPLICIT GRANT HTTP

```
GET /authorize?response_type=token
        &client_id=123
       &redirect_uri=https://partner.com/oauth
```

```
302 REDIRECT https://partner.com/
oauth#access_token=19437jhj2781FQd44AzqT3Zg
 &token_type=Bearer&expires_in=3600
```

AUTHORIZATION TOKEN

- What is a token?
- Anything you like, really…
- Its important that OAuth 2.0 server can validate the token



OPEN STANDARD

Internet Engineering Task Force (IETF)
Request for Comments: 6750
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
D. Hardt
Independent

October 2012

The OAuth 2.0 Authorization Framework: Bearer Token Usage

Abstract

This specification describes how to use bearer tokens in HTTP requests to access OAuth 2.0 protected resources. Any party in possession of a bearer token (a "bearer") can use it to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens need to be protected from disclosure in storage and in transport.

https://tools.ietf.org/html/rfc6750

TOKEN RESPONSE

```
HTTP/1.1 200 OK
        Content-Type: application/json;charset=UTF-8
        Cache-Control: no-store
        Pragma: no-cache

        {
          "access_token":"mF_9.B5f-4.1JqM",
          "token_type":"Bearer",
          "expires_in":3600,
          “refresh_token”:”********************”
        }
```

TOKEN INSIDE REQUEST

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer ****************
```

REFRESH TOKEN

- Tokens should be refreshed after they have expired
- Optional feature
- Allows easier implementation of OAuth 2.0 providers

```
POST /token?grant_type=refresh_token
        &refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

SPRING IMPLEMENTATION

@EnableAuthorizationServer

@EnableResourceServer

Authorization and Resource servers could be same or separate applications

org.springframework.security.oauth:spring-security-oauth2

SPRING: AUTHORISATION SERVER

```
@Configuration

@EnableAuthorizationServer

class AuthorizationServerConfiguration extends
AuthorizationServerConfigurerAdapter {


public void configure(ClientDetailsServiceConfigurer clients) {
    clients.inMemory()
        .withClient(“client-id")
        .authorizedGrantTypes("password", "refresh_token", "authorization_code")
        .authorities("USER")
        .scopes(“view_profile", “view_email")
        .resourceIds(“user_profile”)
        .secret("secret");
}

void configure(AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints
        .tokenStore(tokenStore())
        .accessTokenConverter(accessTokenConverter())
        .authenticationManager(authenticationManager)
        .userDetailsService(userDetailsService);
}
```

CLIENT CONFIGURATION

- `clientId` : (required) the client id.
- `secret` : (required for trusted clients) the client secret, if any.
- `scope` : The scope to which the client is limited. If scope is undefined or empty (the default) the client is not limited by scope.
- `authorizedGrantTypes` : Grant types that are authorized for the client to use. Default value is empty.
- `authorities` : Authorities that are granted to the client (regular Spring Security authorities).

Client configuration could be in memory, jdbc based or any other configuration

User credentials configuration could be anywhere as well

SPRING: RESOURCE SERVER

```
@Configuration

@EnableResourceServer
public class ResourceServerConfiguration extends
ResourceServerConfigurerAdapter {


public void configure(ResourceServerSecurityConfigurer config) {
    config
        .resourceId(“user_profile)
        .tokenServices(tokenServices());
}

public void configure(HttpSecurity http) {
    http
        .authorizeRequests()

        .anyRequest().hasRole("USER")
}
```

RESTRICTING FUNCTIONALITY BY SCOPE

```
@Service
public class SecureResourceServer {
    @PreAuthorize("#oauth2.hasScope('write')")
    public void create(Contact contact) {
        …
    }
}
```

SPRING OAUTH 2.0 ENDPOINTS

/oauth/authorize - requests for authorisation

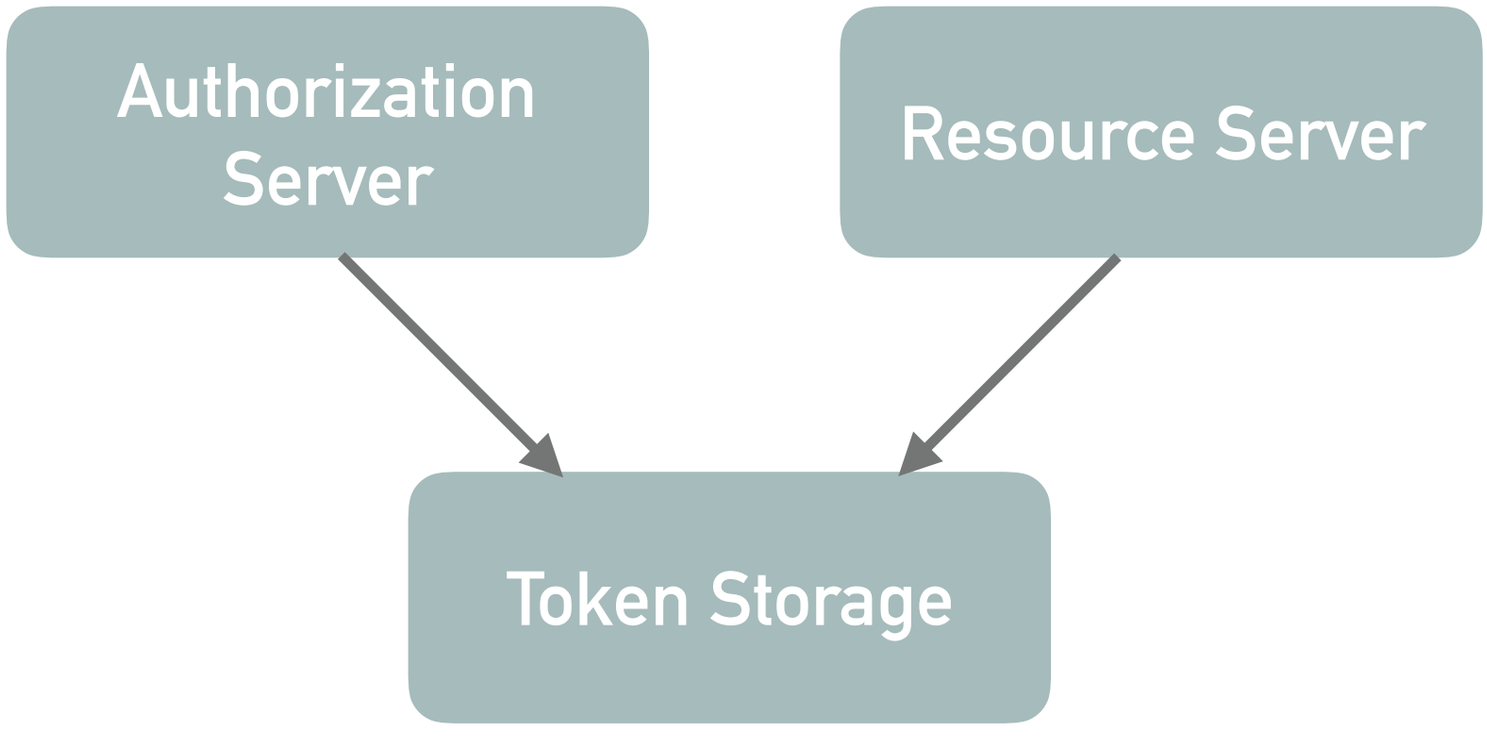
/oauth/token - requests for token

contains default Spring MVC authentication page, which could be customised

http://projects.spring.io/spring-security-oauth/docs/oauth2.html

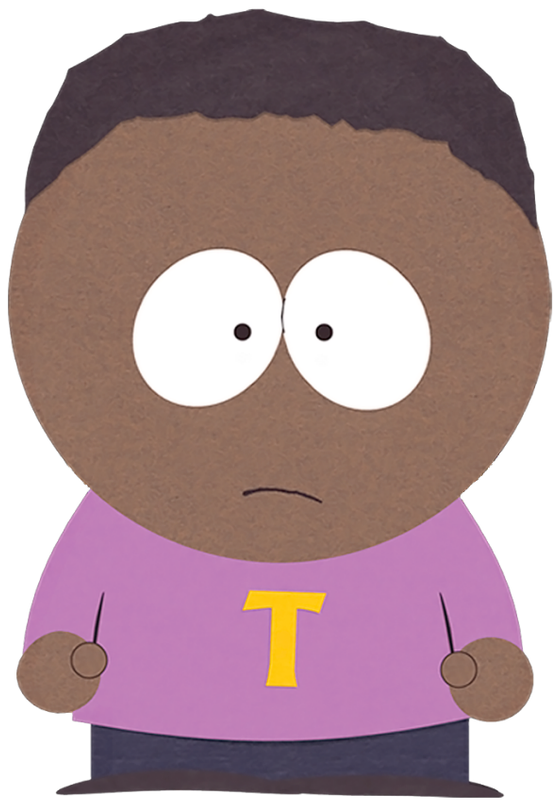
TOKEN STORAGE

- Shared token service is required
- Could be in-memory or persisted



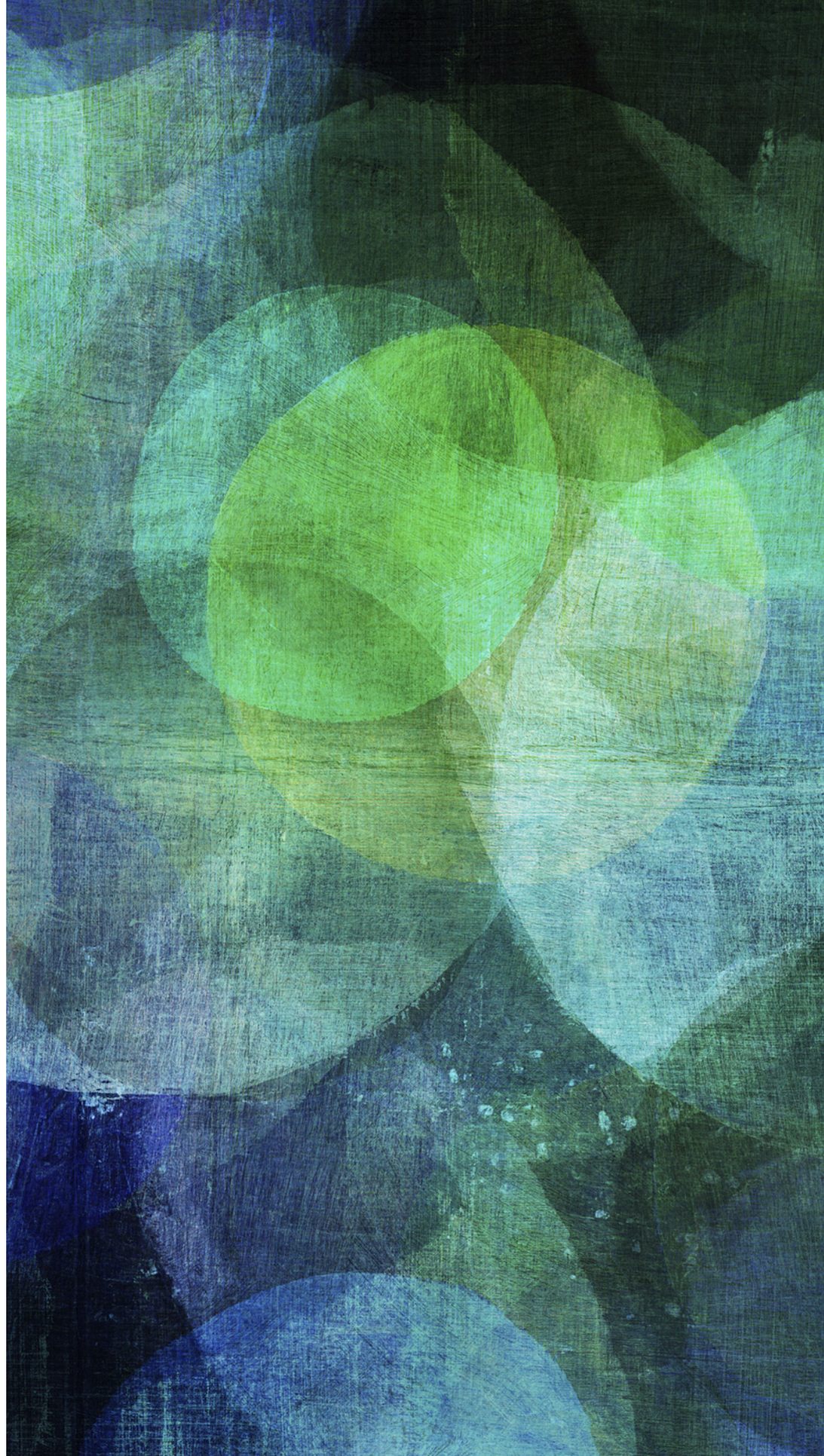
WHAT TOKENS TO USE?

- AtomicLong - predictable?
- Random numbers - clashes possible?
- Hash - from what?
- Is there any existing approach?



JSON WEB TOKEN

Explained



JWT OPEN STANDARD

Internet Engineering Task Force (IETF)
Request for Comments: 7519
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
May 2015

JSON Web Token (JWT)

Abstract

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

https://tools.ietf.org/html/rfc7519

JSON WEB TOKENS

- Send stuff between client and server securely
- Signed content
- Cross-platform
- Token storage is not necessary

https://jwt.io/

JWT TOKEN STRUCTURE

HEADER

PAYLOAD

SIGNATURE

HEADER

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

PAYLOAD

- Reserved claims
  - issuer
  - expiration time
  - subject
- Public claims (named according to registry)
- Private claims (custom)

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

https://www.iana.org/assignments/jwt/jwt.xhtml

SIGNATURE

```
1   HMACSHA256(
2     base64UrlEncode(header) + "." +
3     base64UrlEncode(payload),
4     secret)
```

- ✓ HS256
- ✓ HS384
- ✓ HS512
- ✓ RS256
- ✓ RS384
- ✓ RS512
- ✓ ES256
- ✓ ES384
- ✓ ES512

- ➤ JSON Web Token could be signed with
- ➤ Secure hash based on salt
- ➤ Public/private key using RSA

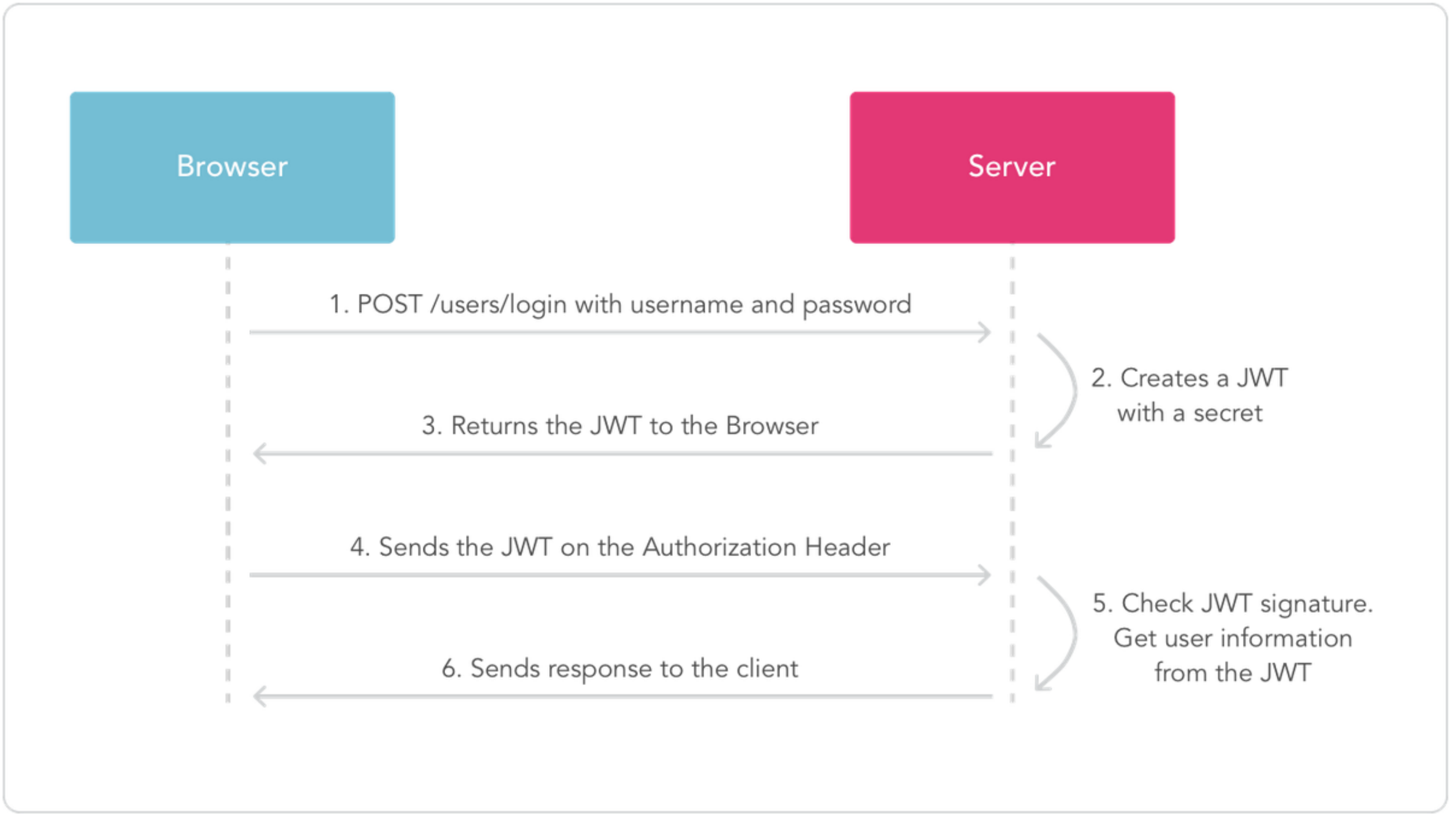
JWT EXAMPLE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMj
M0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRyd
WV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

BASE64 Encoded

Parts are separated by dots (.)

JWT SIMPLE FLOW



TOKEN INSIDE REQUEST

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer $JWT_TOKEN
```

JAVA IMPLEMENTATION

```
String token = Jwts.builder()
            .setSubject(user.getUsername())
            .setClaims([“scope” -> “user profile”])
            .setIssuedAt(new Date())
            .setExpiration(from(now().plus(3600)))
            .setId(random(1000000))
            .signWith(SignatureAlgorithm.HS512, secret)
            .compact();
```

io.jsonwebtoken:jjwt

JWT BENEFITS

- Standard approach
- Self-contained - no need for token/session storage
- Passed with each request to the server
- Plays nice with OAuth 2.0

SPRING OAUTH 2.0 INTEGRATION

```
@Bean public TokenStore tokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}
@Bean public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey(SIGNING_KEY);
    return converter;
}
@Bean @Primary public DefaultTokenServices tokenServices() {
    DefaultTokenServices tokenServices = new DefaultTokenServices();
    tokenServices.setTokenStore(tokenStore());
    tokenServices.setSupportRefreshToken(true);
    return tokenServices;
}
```

org.springframework.security:spring-security-jwt

JWT AND OAUTH 2.0

- JWT can be used as a token in OAuth 2.0 authorisation
- There is no need for token storage in this case
- Everything works out of the box

SUMMARY

- OAuth 2.0 is all about information flow
- Interpretation is possible
- Extensions are available (e.g. token revocation, additional grant types)
- Token could be arbitrary
- It is possible to use JWT tokens

REFERENCES

- https://oauth.net/2/
- http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/
- http://docs.oracle.com/cd/E39820_01/doc.11121/gateway_docs/content/oauth_flows.html
- https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2