Introduction to Spring Cloud

Microservices Security

LXFT LISTED NYSE

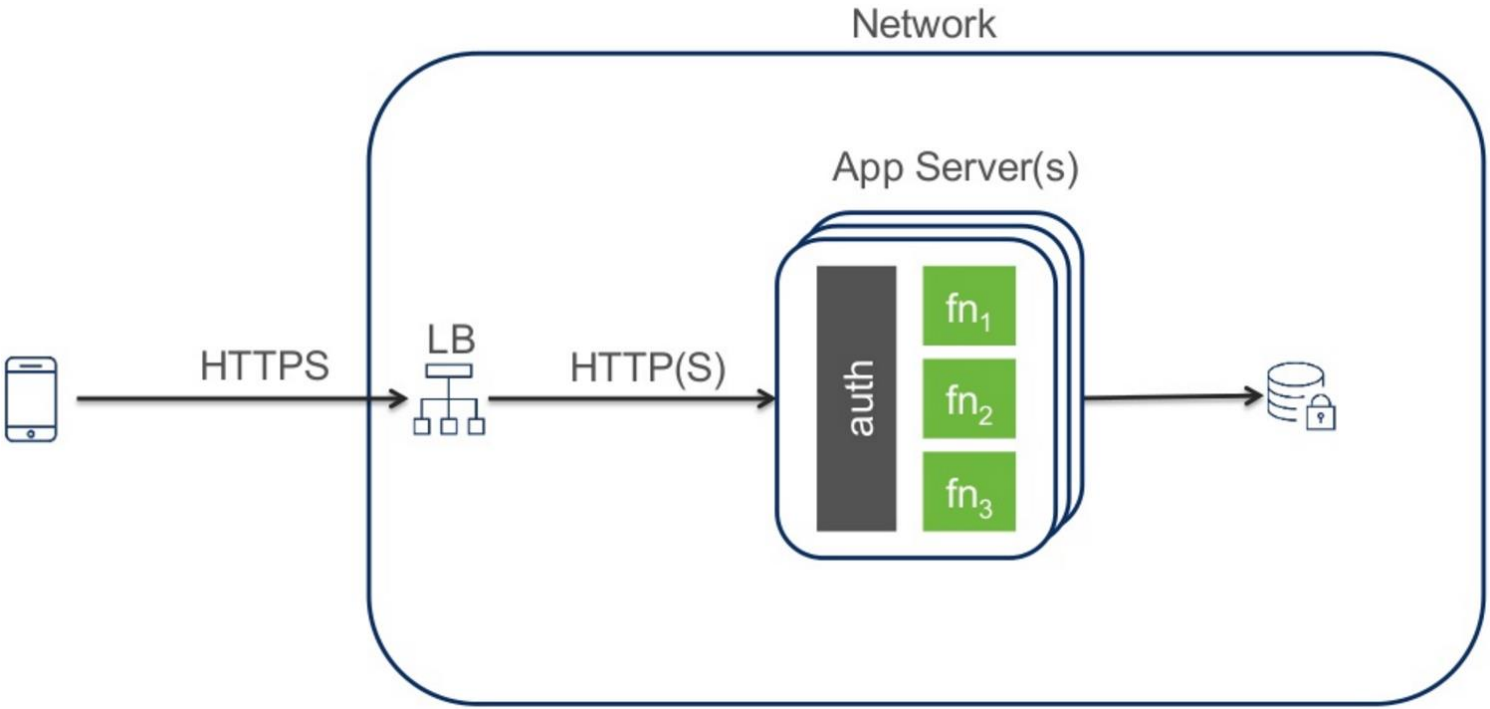
Prepared by Orkhan Gasimov

Feb. 2017

LUXOFT

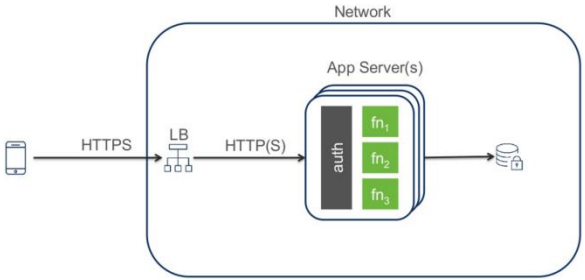
Securing The Monolith

Securing The Monolith



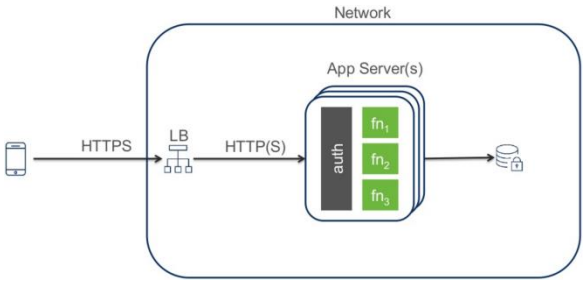
Securing The Monolith

- Securing the monolith is easier, you only need to authorize once per user request.
  - No session:
    - Verify user credentials, get the user roles, start a new user session;
  - Have session:
    - Verify it is not expired;
  - You can trust method calls.
    - Request and response are handled in a single process.



Securing The Monolith

- Pros:
  - Limited attack surface;
- Cons:
  - The application has all the credentials it needs, to do anything it wants to the database or any other available resources;
  - Break a single process and you get it all;



Microservices Security Implementations

Microservices Security Implementations

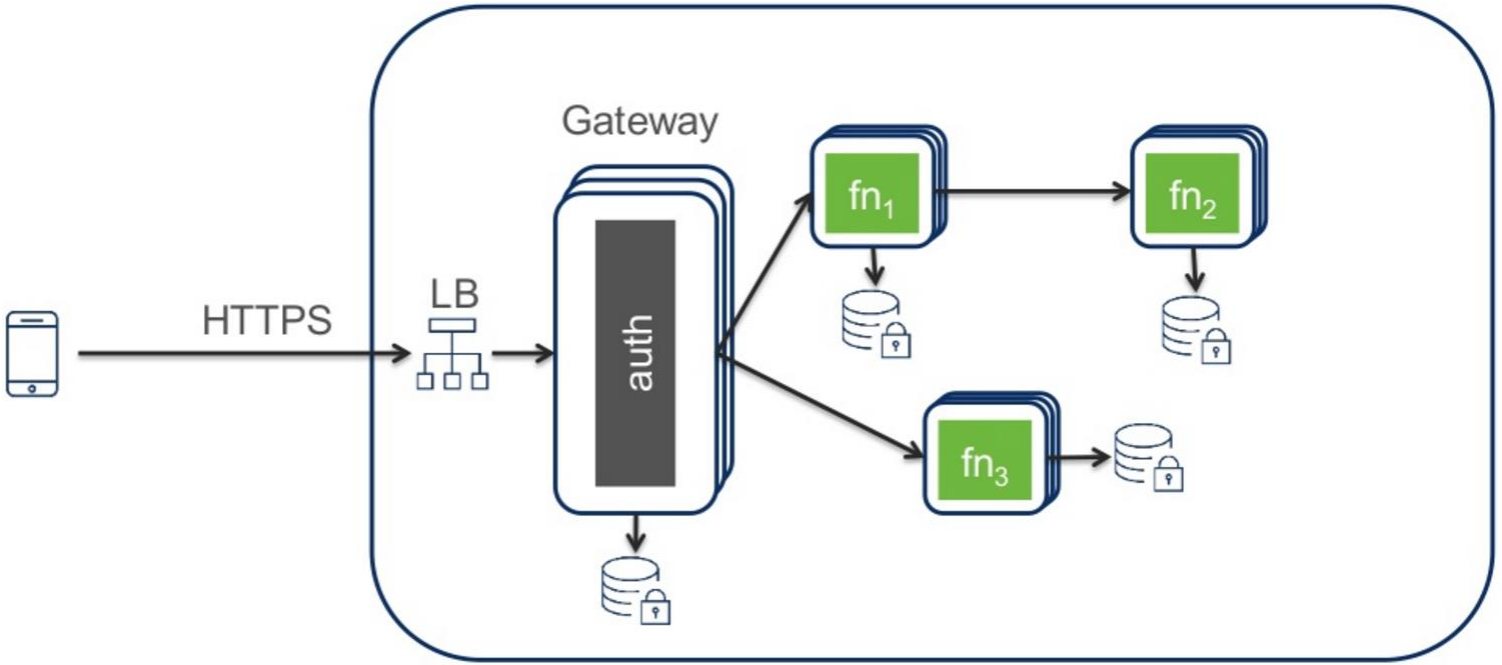
- Microservice security is harder
  - You win having principal of least privilege.
    - Every component only has access to what it needs to perform its function.
  - But lose in:
    - Much larger attack surface, especially for internal threats.
    - How do other services know who's accessing them?
    - How can other services trust each other?

Microservices Security Implementations

- API Gateway / Perimeter Security
- Everybody Can Auth (with HTTP Basic)
- Basic + Central Auth DB
- Sessions Everywhere
- API Tokens
- SAML

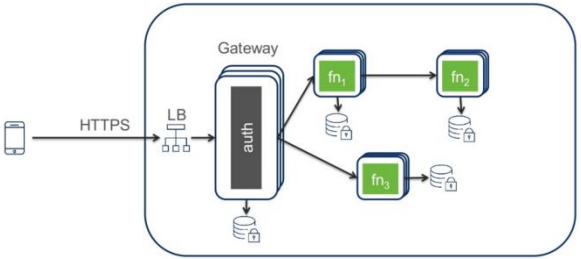
API Gateway / Perimeter Security

API Gateway / Perimeter Security



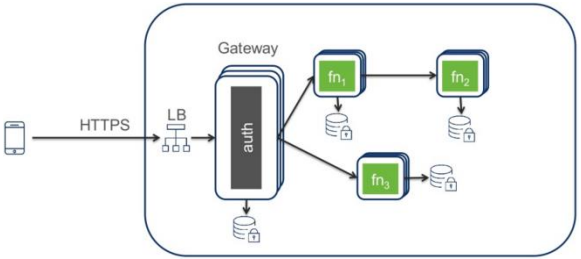
API Gateway / Perimeter Security

- We could secure the API gateway, assuming all the access to our data will go through it.
  - This is nearly the same as you would do with classic Spring MVC security.
- But,
  - There is no way of method (microservice) security;
  - Your data is insecure from inside attacks;
    - Microservices are not secured inside.



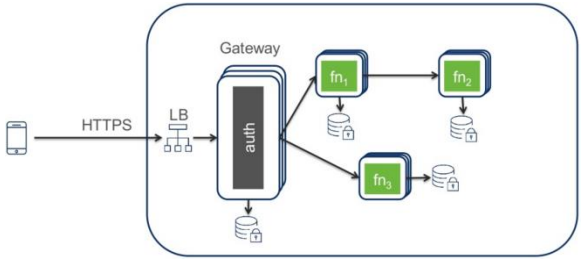
API Gateway / Perimeter Security

- Requests are authenticated and authorized by the gateway.
  - The public load balancer cannot send requests to applications directly.
- Applications trust all traffic they receive by assumption that it is authorized.



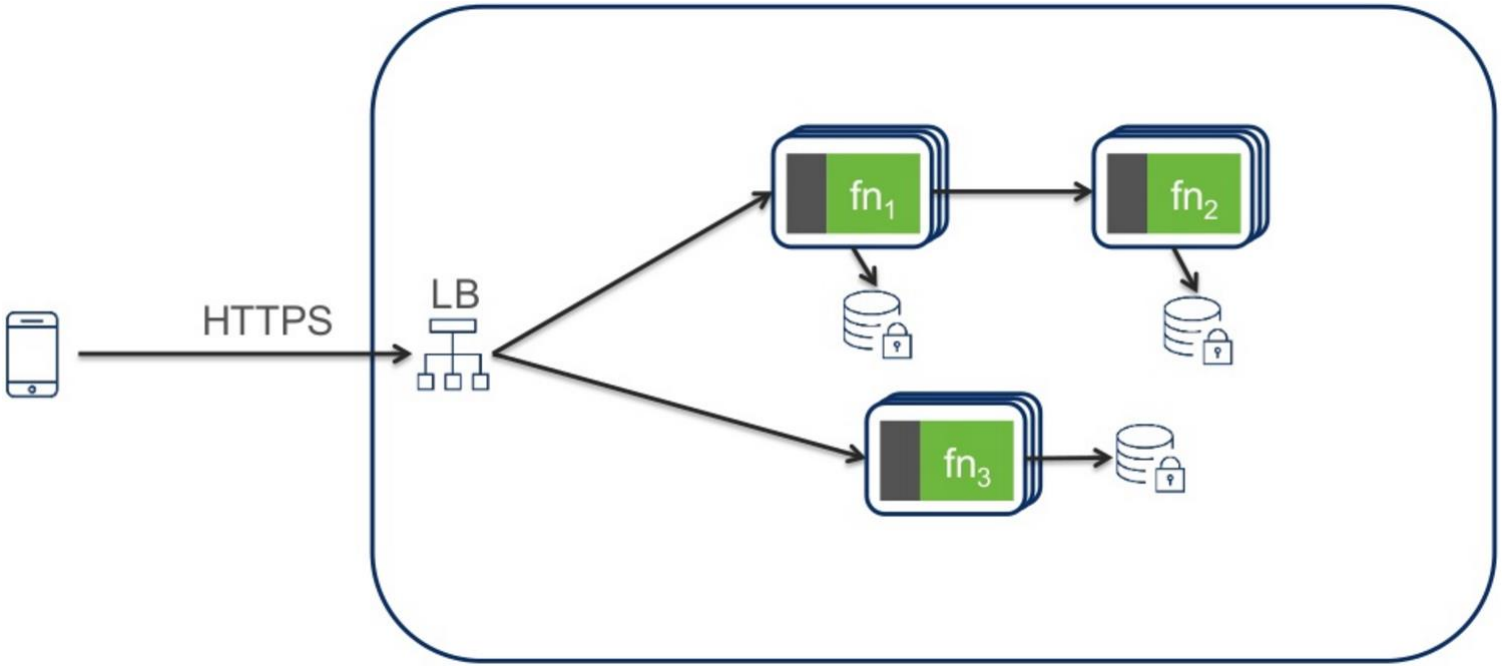
API Gateway / Perimeter Security

- Pros:
  - Network setup can virtually guarantee assumptions.
  - Applications have stateless security.
- Cons:
  - Does nothing for internal threats.



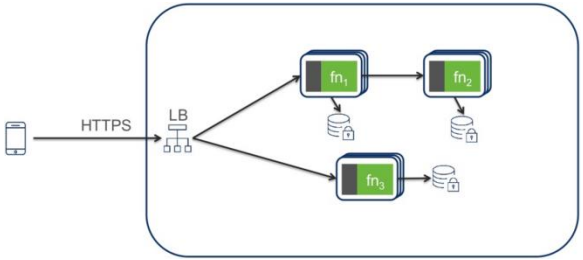
Everybody Can Auth (with HTTP Basic)

Everybody Can Auth (with HTTP Basic)



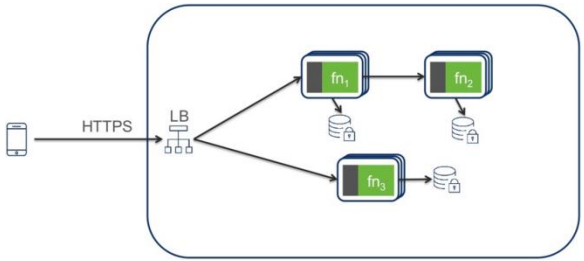
Everybody Can Auth (with HTTP Basic)

- All applications have to do authentication and authorization themselves.
- Basic credentials are passed along in every request.



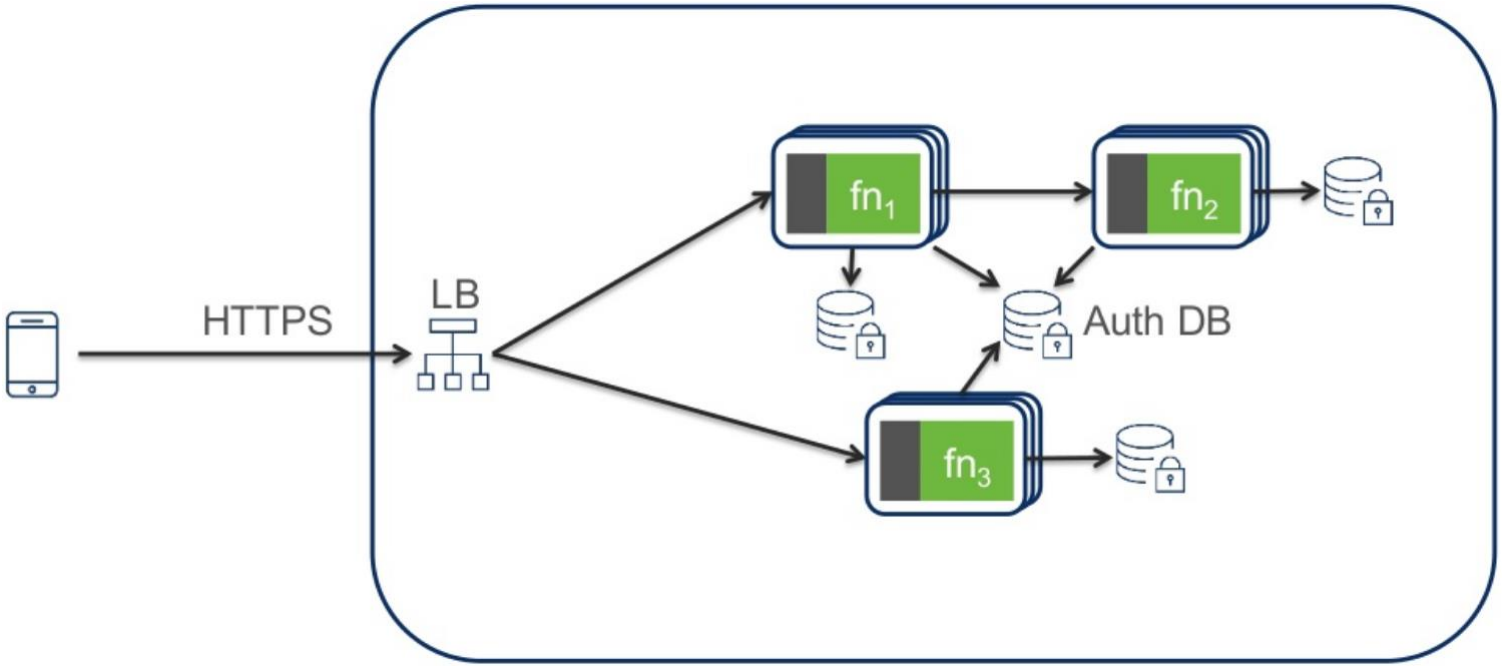
Everybody Can Auth (with HTTP Basic)

- Pros:
  - Stateless (authenticate every time).
  - Easy.
- Cons:
  - How do you store and lookup the credentials?
  - How do you manage authorization?
  - User’s credentials can unlock all functionality (until user updates password).



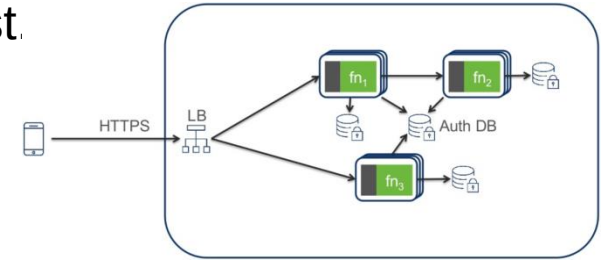
Basic + Central Auth Database

Basic + Central Auth Database

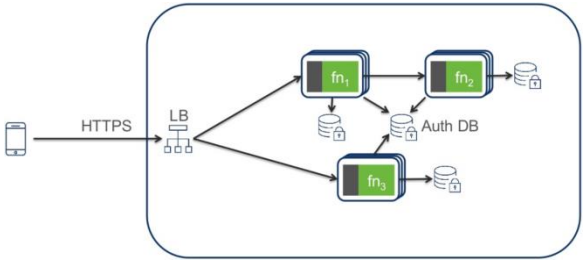


Basic + Central Auth Database

- We share the user credential database between all applications and authenticate the user on each application before access.
- Sounds stupid, but it is actually a working approach with all the Spring Security features available.
  - All applications have to do authentication and authorization themselves.
  - Basic credentials are passed along in every request.
  - Credentials are verified against a central database.



Basic + Central Auth Database

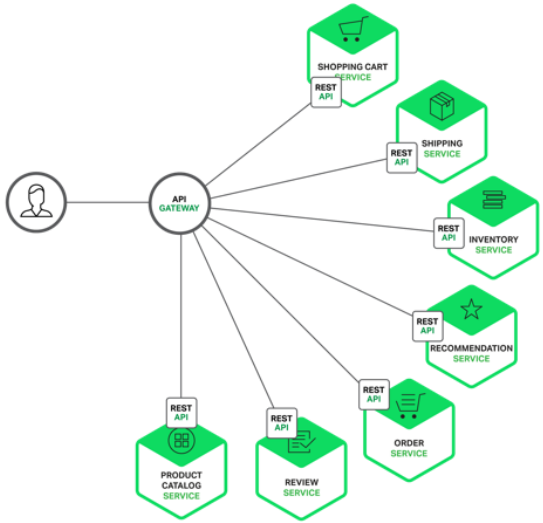


- Pros:
  - Central user store;
  - Stateless (authenticate every time);
- Cons:
  - Auth DB is hit every request;
  - Database lookup logic needs to be implemented everywhere;
  - User’s credentials can unlock all functionality;

Sessions Everywhere

Sessions Everywhere

- Same as previous, but each application gets to maintain a session with the client device.
  - Pros:
    - Auth DB is hit once per session.
  - Cons:
    - Hard to manage all the sessions.
    - No single sign on.
    - Database lookup logic needs to be implemented everywhere.
    - User’s credentials can unlock all functionality.



API Tokens

API Tokens

- The user authenticates on a authorization service, which maps the user session to a token.
  - Any further API calls to the resource services must provide this token.
- Applications validate the token for each request by hitting the authorization service.
  - The services are able to recognize the provided token and ask the authorization service, which authorities this token grants, and who is the owner of this token.

API Tokens

- Pros:
  - Applications do not receive user credentials.
- Cons:
  - Authorization server bottleneck.
  - Token provides all or nothing access.

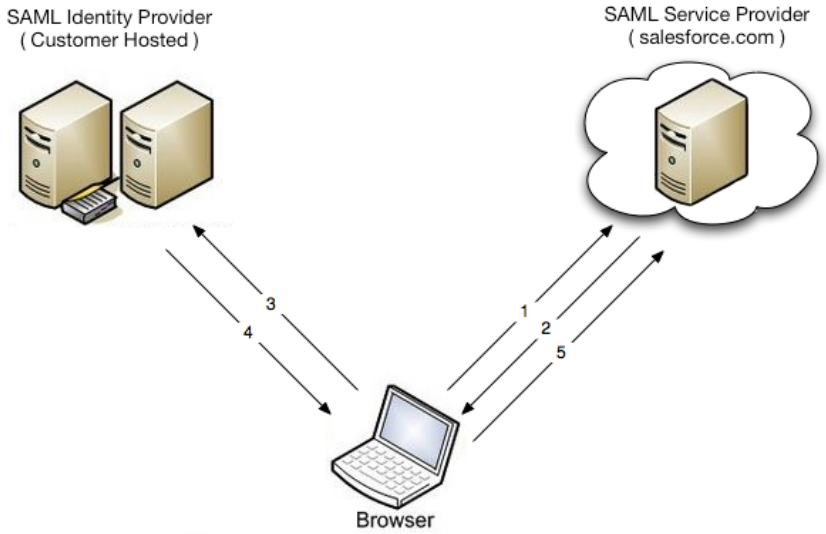
SAML

SAML

- The SAML specification defines three roles:
  - the Principal (user), the Identity Provider, and the Service Provider.
- In the use case addressed by SAML,
  - the Principal requests a service from the Service Provider.
  - The Service Provider requests and obtains an identity assertion from the Identity Provider.
  - On the basis of this assertion, the service provider can make an access control decision
    - in other words it can decide whether to perform some service for the connected Principal.

SAML

- Pros:
  - Standard trust model.
  - Self verification of assertions.
- Cons:
  - XML is big and ugly.
  - Difficult for non-browser clients.
    - e.g. mobile



Common Concerns

Common Concerns

- All implementations discussed above have a few concerns that are common to some of them and should be addressed:
  - Central user store bottleneck;
  - Lack of single sign on;
  - Statelessness;
  - Exposure of user credentials;
  - Lack of fine grained authorization;
  - Interoperability with non browser clients;

OAuth2

OAuth2

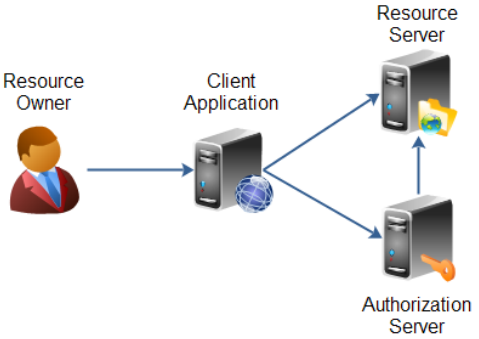
- OAuth2 is delegated authorization that delivers:
  - A protocol for conveying authorization decisions via token;
  - Standard means of obtaining a token with 4 grant types;
  - Users and Clients are separate entities;
    - *“I am authorizing this app to perform these actions on my behalf”*

OAuth2

- OAuth2 is NOT Authentication.
  - The user must be authenticated to obtain a token;
  - How the user is authenticated is outside of the spec;
  - How the token is validated is outside of the spec;
  - What the token contains is outside of the spec;

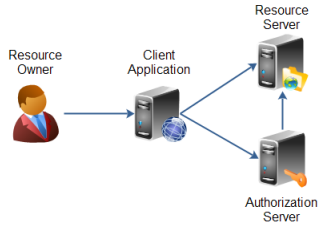
OAuth2

- OAuth2 defines four roles in this process:
  - Resource Owner
    - a user, application or service.
  - Resource Server
    - an application or service.
  - Authorization Server
    - the service which knows resource owner, its session and data.
  - Client
    - the application that requests access to resource.



OAuth2

- The four OAuth2 roles depend of the direction in which data is requested.
  - For asking protected business data from resource server, the authorization server is what it is, the resource servers also, the application is the client and the service holding the permissions (often the same as authorization server), is the owner.
  - When asking the users data, the authorization service becomes a resource server, and resource server the client.

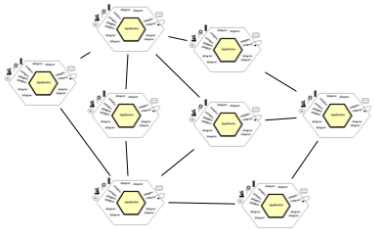


OAuth2

- With OAuth2 you can define, which applications (web, mobile, desktop, additional website) can access which resources.
- There is one dimension, the scope, where we have to decide which user can access which data, or which application or service, can access which resource.
  - In other words: scopes are access controlling which endpoints are visible to clients, and authorities filter the data to the user based on his permissions.

OAuth2

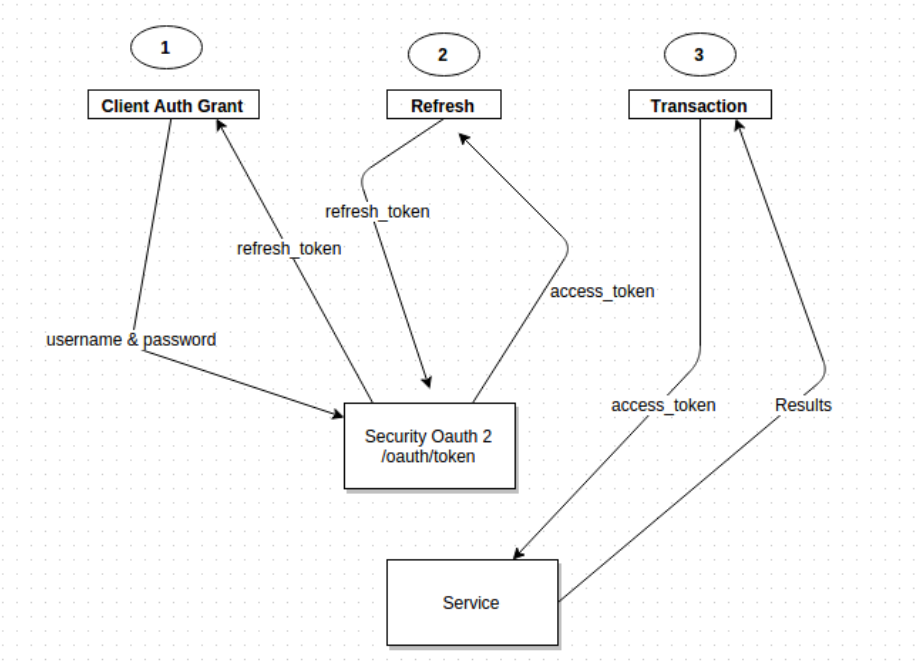
- In a web shop, the frontend may act as an client, having access to products, orders and customers, but the backend also about logistics, contracts and more, independent of the users authorities.
- In the other way, a user may have potential access to a service but no access to all its data, because he is using a web application, where other users are permitted to access while he is not.
  - This service-to-service access, is our other dimension.



OAuth 2
Tokens

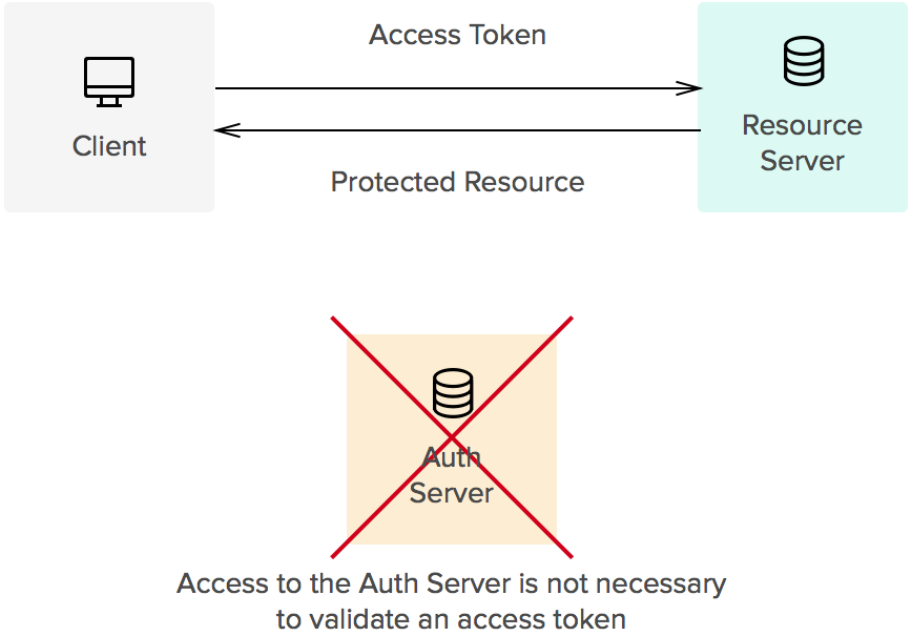
OAuth2 – Tokens

- With OAuth2 we may highlight 2 kinds of tokens:
  - Access tokens
  - Refresh tokens



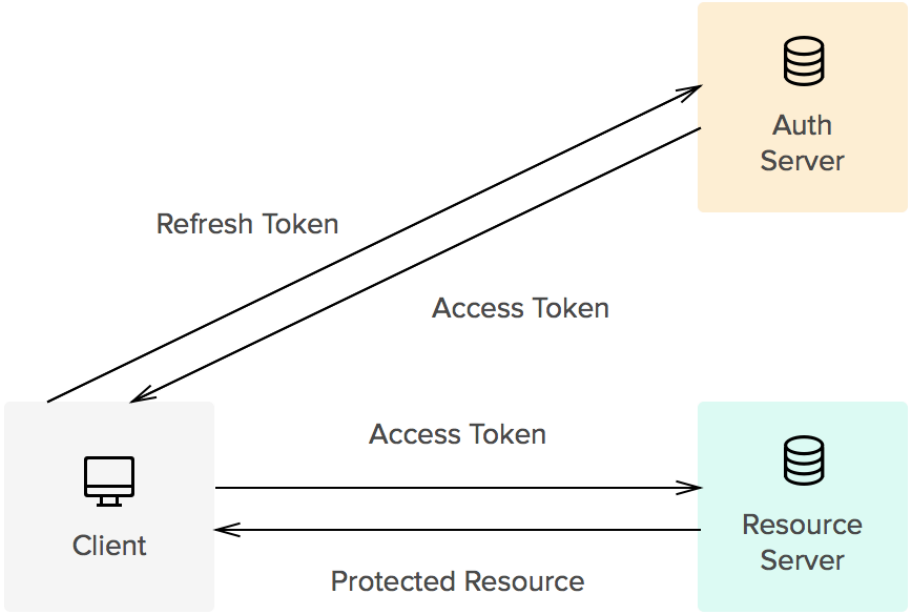
OAuth2 – Tokens

- Access tokens carry the necessary information to access a resource directly.
  - In other words, when a client passes an access token to a server managing a resource, that server can use the information contained in the token to decide whether the client is authorized or not.
- Access tokens usually have an expiration date and are short-lived.



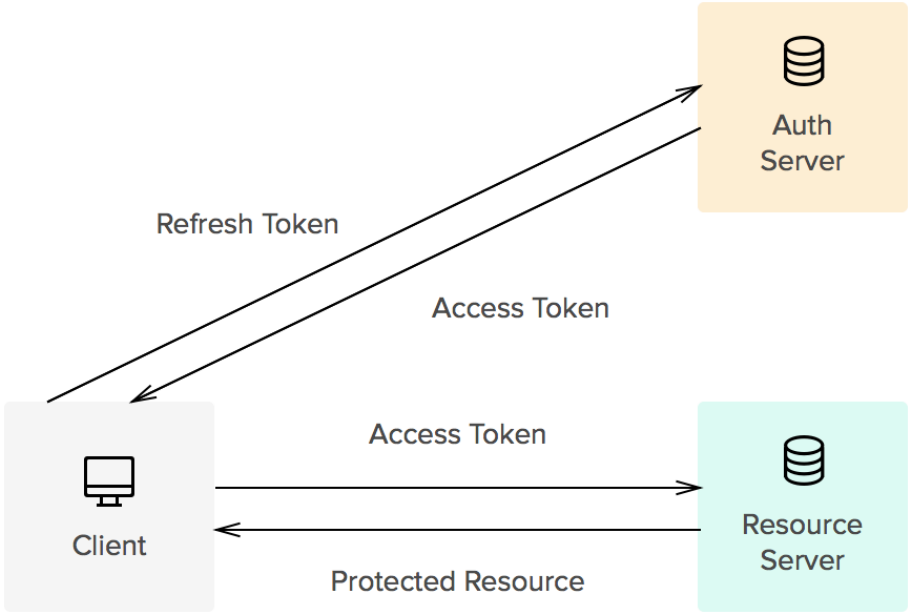
OAuth2 – Tokens

- Refresh tokens carry the information necessary to get a new access token.
  - In other words, whenever an access token is required to access a specific resource, a client may use a refresh token to get a new access token issued by the authentication server.



OAuth2 – Tokens

- Common use cases include getting new access tokens after old ones have expired, or getting access to a new resource for the first time.
  - Refresh tokens can also expire but are rather long-lived.
- Refresh tokens are usually subject to strict storage requirements to ensure they are not leaked.
  - They can also be blacklisted by the authorization server



OAuth 2
Grant Types

OAuth2 – Grant Types

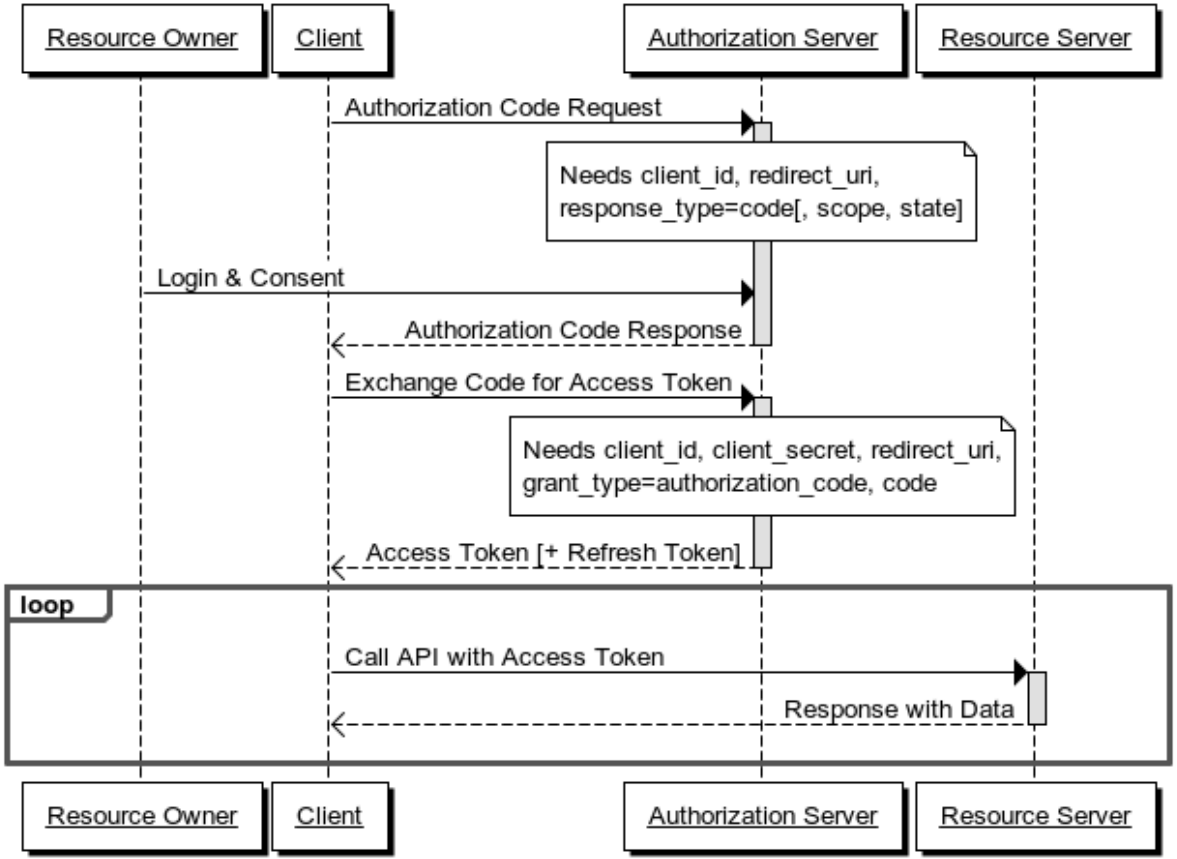
- OAuth2 defines 4 grant types depending on the location and the nature of the client involved in obtaining an access token.
  - Authorization Code
  - Password
  - Client Credentials
  - Implicit

OAuth 2
Grant Type: Authorization Code

OAuth2 – Grant Type: Authorization Code

- Authorization Code
  - It should be used as soon as the client is a web server.
  - It allows you to obtain a long-lived access token since it can be renewed with a refresh token (if the authorization server enables it).
- For applications running on a web server, browser-based and mobile applications.

OAuth2 – Grant Type: Authorization Code



OAuth2 – Grant Type: Authorization Code

- Example:
  - Resource Owner: you
  - Resource Server: a Google server
  - Client: any website
  - Authorization Server: a Google server

OAuth2 – Grant Type: Authorization Code

- Scenario:
  - A website wants to obtain information about your Google profile.
  - You are redirected by the client (the website) to the authorization server (Google).
  - If you authorize access, the authorization server sends an authorization code to the client (the website) in the callback response.
  - Then, this code is exchanged against an access token between the client and the authorization server.
  - The website is now able to use this access token to query the resource server (Google again) and retrieve your profile data.

OAuth2 – Grant Type: Authorization Code

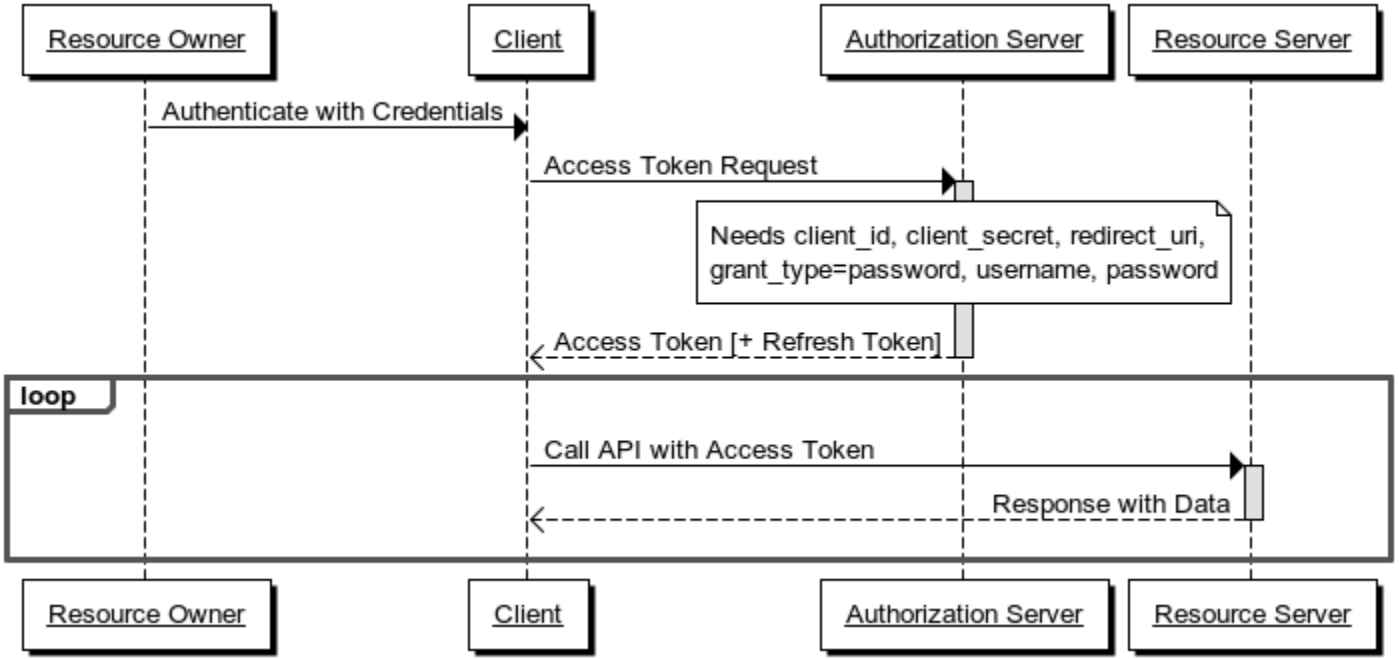
- You never see the access token, it will be stored by the website (in session for example).
  - Google also sends other information with the access token, such as the token lifetime and eventually a refresh token.
- This is the ideal scenario and the safer one because the access token is not passed on the client side (web browser in our example).

OAuth 2
Grant Type: Password

OAuth2 – Grant Type: Password

- Password
  - With this type of authorization, the credentials (and thus the password) are sent to the client and then to the authorization server.
  - It is therefore imperative that there is absolute trust between these two entities.
  - It is mainly used when the client has been developed by the same authority as the authorization server.
  - For example, we could imagine a website named example.com seeking access to protected resources of its own subdomain api.example.com (the user would not be surprised to type his login/password on the site example.com since his account was created on it).
- For logging in with a username and password.

OAuth2 – Grant Type: Password



OAuth2 – Grant Type: Password

- Example:
  - Resource Owner: you having an account on acme.com website of the Acme company
  - Resource Server: Acme company exposing its API at api.acme.com
  - Client: acme.com website from Acme company
  - Authorization Server: an Acme server

OAuth2 – Grant Type: Password

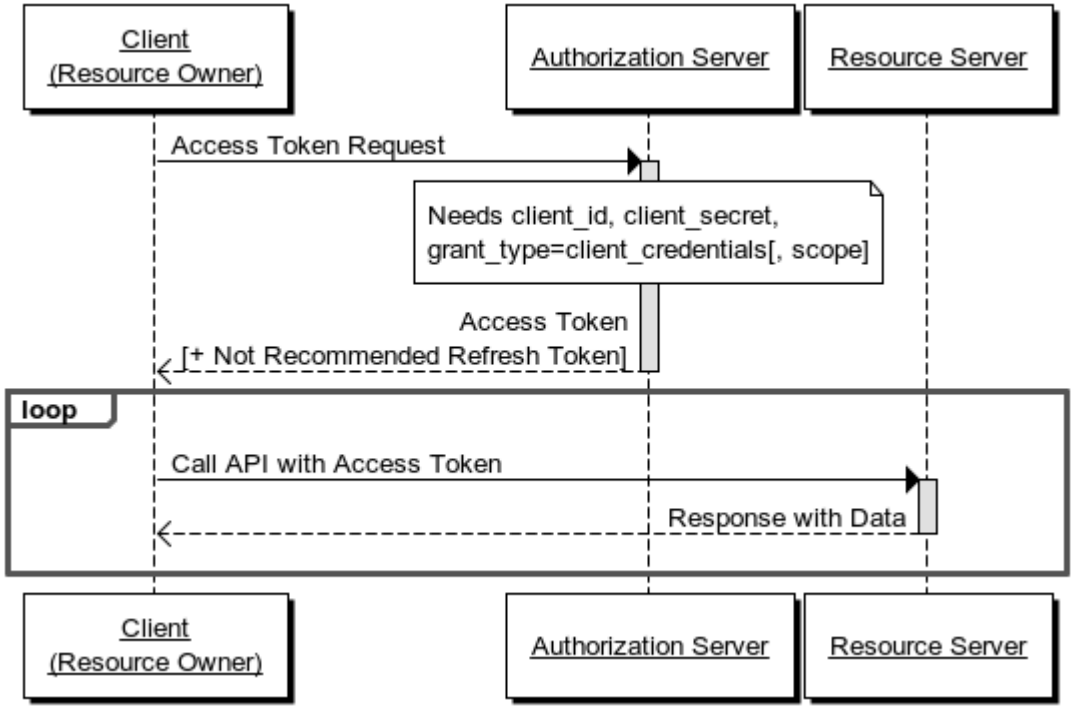
- Scenario:
  - Acme company, doing things well, thought to make available a RESTful API to third-party applications.
  - This company thinks it would be convenient to use its own API to avoid reinventing the wheel.
  - Company needs an access token to call the methods of its own API.
  - For this, company asks you to enter your login credentials via a standard HTML form as you normally would.
  - The server-side application (website acme.com) will exchange your credentials against an access token from the authorization server (if your credentials are valid, of course).
  - This application can now use the access token to query its own resource server (api.acme.com).

OAuth 2
Grant Type: Client Credentials

OAuth2 – Grant Type: Client Credentials

- Client Credentials
  - For application access.
- This type of authorization is used when the client is himself the resource owner.
  - There is no authorization to obtain from the end-user.

OAuth2 – Grant Type: Client Credentials



OAuth2 – Grant Type: Client Credentials

- Example:
  - Resource Owner: any website
  - Resource Server: Google Cloud Storage
  - Client: the resource owner
  - Authorization Server: a Google server

OAuth2 – Grant Type: Client Credentials

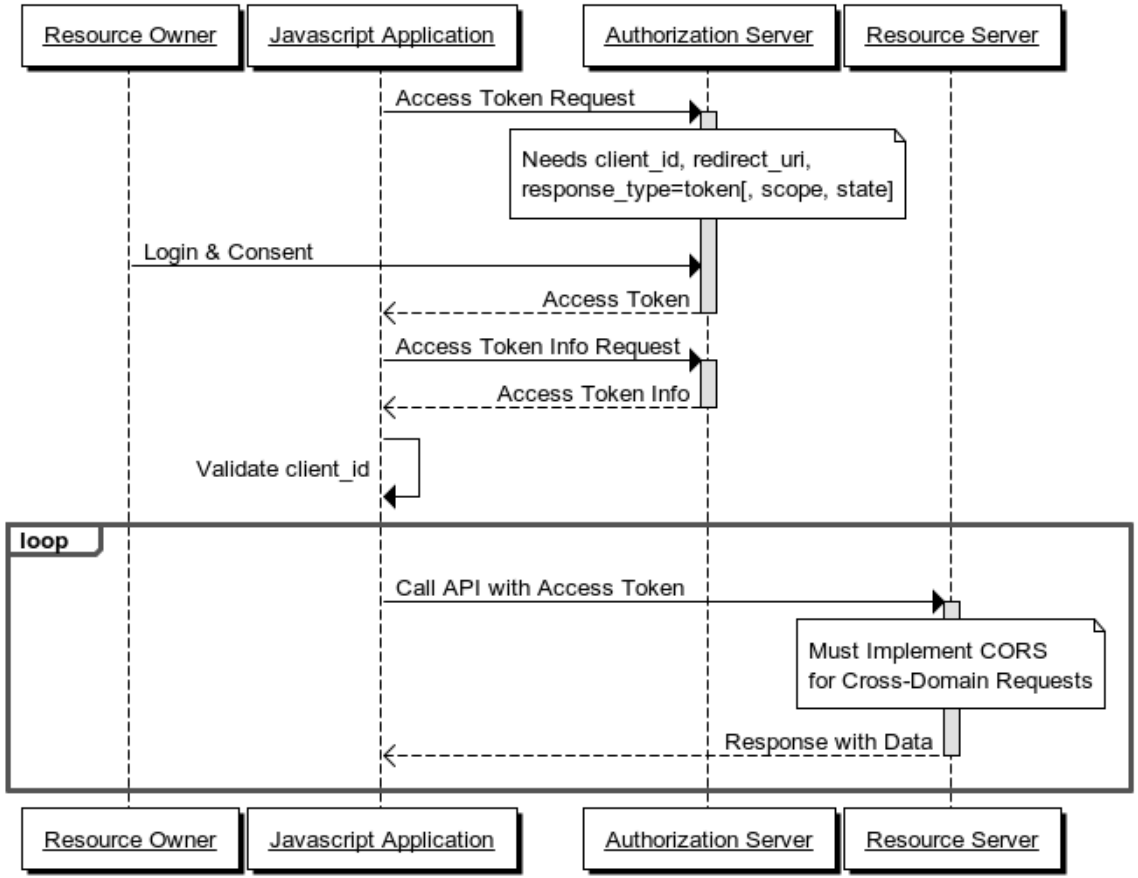
- Scenario:
  - A website stores its files of any kind on Google Cloud Storage.
  - The website must go through the Google API to retrieve or modify files and must authenticate with the authorization server.
  - Once authenticated, the website obtains an access token that can now be used for querying the resource server (Google Cloud Storage).
- Here, the end-user does not have to give its authorization for accessing the resource server.

OAuth 2
Grant Type: Implicit

OAuth2 – Grant Type: Implicit

- Implicit
  - It is typically used when the client is running in a browser using a scripting language such as JavaScript.
  - This grant type does not allow the issuance of a refresh token.
- Was previously recommended for clients without a secret, but has been superseded by using the Authorization Code grant with no secret.

OAuth2 – Grant Type: Implicit



OAuth2 – Grant Type: Implicit

- Example:
  - Resource Owner: you
  - Resource Server: a Facebook server
  - Client: a website using AngularJS for example
  - Authorization Server: a Facebook server

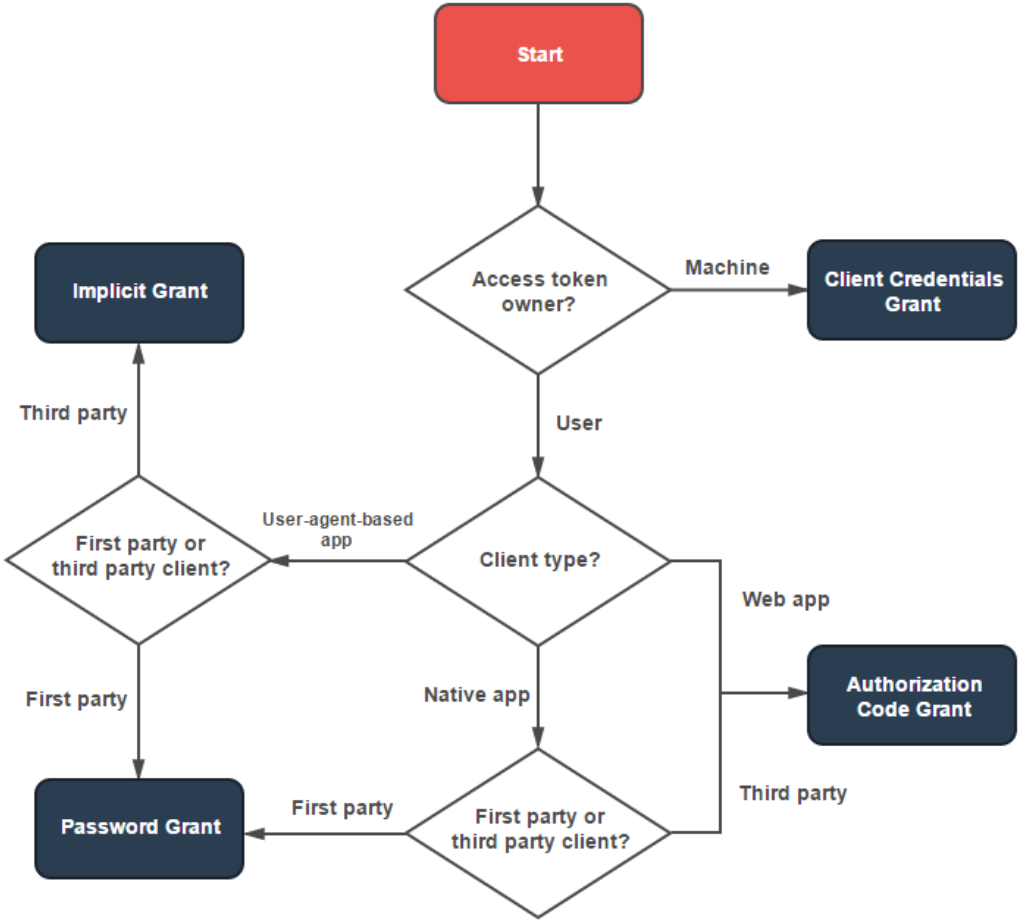
OAuth2 – Grant Type: Implicit

- Scenario:
  - The client (AngularJS) wants to obtain information about your Facebook profile.
  - You are redirected by the browser to the authorization server (Facebook).
  - If you authorize access, the authorization server redirects you to the website with the access token in the URI fragment (not sent to the web server). Example of callback: http://example.com/oauthcallback#access_token=MzJmNDc3M2VjMmQzN.
  - This access token can now be retrieved and used by the client (AngularJS) to query the resource server (Facebook). Example of query: https://graph.facebook.com/me?access_token=MzJmNDc3M2VjMmQzN.

OAuth2 – Grant Type: Implicit

- Maybe you wonder how the client can make a call to the Facebook API with JavaScript without being blocked because of the Same Origin Policy?
- Well, this cross-domain request is possible because Facebook authorizes it thanks to a header called Access-Control-Allow-Origin present in the response.

OAuth2 – Grant Types



JWT

JWT

- OAuth2 is not about, how the token is looking like and where it is stored.
  - One approach is, to generate random strings and save token related data to these strings in a store.
  - Over a token endpoint, other services may ask something “is this token valid, and which permissions does it grant?”.
  - This is the “user info URL” approach, where the authorization server is turning into a resource server for the user info endpoint.

JWT

- As we talking about microservices, we need a way to replicate such a token store, to make the authorization server scalable.
- Despite this is a tricky task, there is one more issue with user info URL.
  - For each request against a resource microservice containing a token in the header, the service will perform another request to authorization server to check the token.
  - Without caching tricks, we got twice more request we really need, only for security.

JWT

- Both, scaling token store and token request are affecting the scalability of our architecture heavily.
- This is where JWT come into play.
  - In short, the answer of that user info URL request, containing info about the OAuth client, optionally the user with its authorities and the granted scope, is serialized into JSON first, encoded with base64 and finally signed using a token.
  - The result is a so called JSON Web Token, which we use as an access token directly.

JWT

- When these JWTs are signed using RSA, the authorization server first signs it with the RSA private key, assuming every resource server will have a public key.
  - When this token is passed via HTTP header, the resource servers just have to take this JWT, verify it was really signed by the proper private key (meaning this token is coming from authorization server), and instead of asking user info, deserializing the JSON content into a OAuth2Authentication, establishing a SecurityContext based on this.

JWT

- Using JWT provides a simple way of transmitting hard to fake tokens, containing the permissions and user data in the access token string.
  - Since all the data is already inside, there neither is a need to maintain a token store, nor the resource servers must ask authorization for token checks.
  - So, using JWT makes OAuth2 available to microservices, without affecting the architectures scalability.

OpenID Connect

OpenID Connect

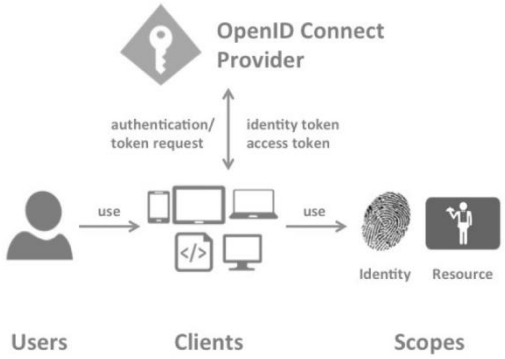
- OpenID Connect was developed to add secure authentication to OAuth 2.0.
- Large providers i.e. Google, Facebook, Yahoo, etc. began using OAuth 2.0 as a way to authenticate users with "login with" services so users could use their credentials to authenticate to a variety of third-party services.

OpenID Connect

- Standard OAuth 2.0 cannot securely satisfy this requirement because of deficiencies within the protocol.
- OpenID Connect solves these deficiencies and allows providers to securely use OAuth 2.0 as an authentication framework. OAuth 2.0 was originally developed as an authorization framework which allows a user to grant a third party service access to their data stored on the provider.

OpenID Connect

- OpenID Connect is delegated authentication
  - A protocol for conveying user identity (via a signed JWT)
  - Built on top of OAuth2
  - Standard means of obtaining an ID token
    - The same 4 OAuth2 grant types are supported
  - Standard means of verifying ID tokens
  - *“Will is authorizing this app to perform these actions on his behalf”*
    - *“And here’s his email address in case you need it”*

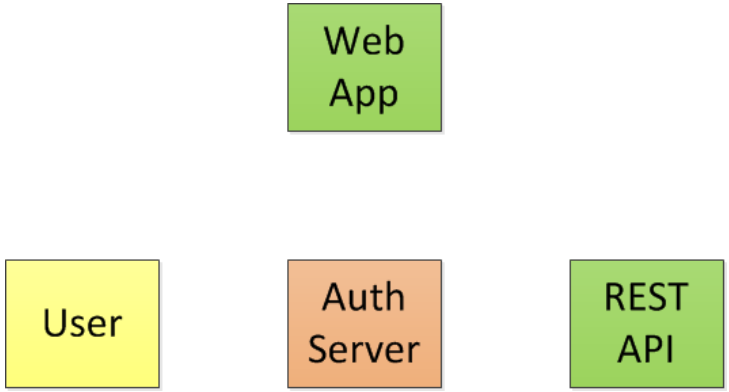


OpenID Connect

- OpenID Connect is NOT Authentication
  - Still doesn’t say how users are to be authenticated
  - This is good: there are a lot of ways to authenticate users
    - Internal DB
    - Another Identity Provider
      - SAML
      - LDAP
    - Multi-factor

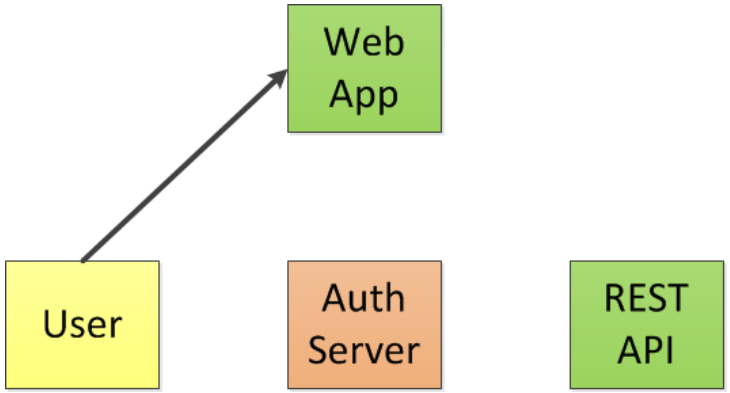
OpenID Connect – How It Works

- Actors:
  - User – resource owner;
  - Web App – client application;
  - REST API – resource server;
  - Auth Server – OpenID Connect provider;
- Setup:
  - User has no session with the Auth Server or web app
- Use case:
  - User wants to place an order on the REST API using the web app.



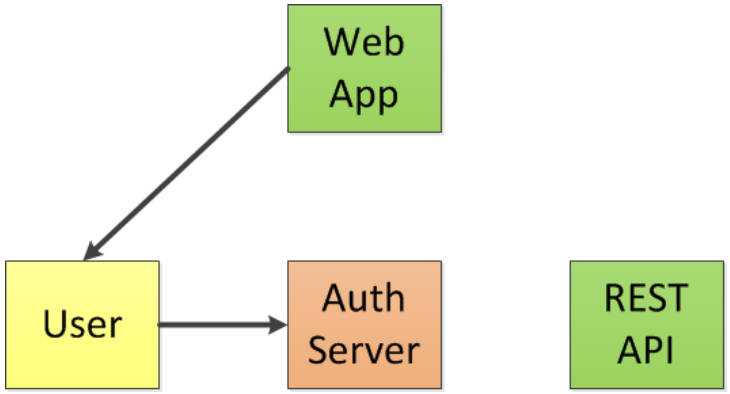
OpenID Connect – How To Get Tokens

- Step 1
  - User accesses web app and does not have a session with it.



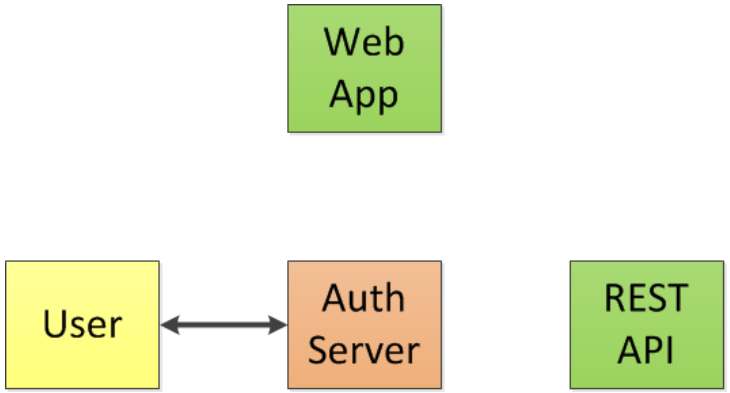
OpenID Connect – How To Get Tokens

- Step 2
  - Web app redirects user to the /authorize endpoint on the auth. server. The redirect URL containt the scopes openid and order.me
  - This means that the app is requesting a token that allows apps to view the users identity (openid) and place order on the user's behalf (order.me).



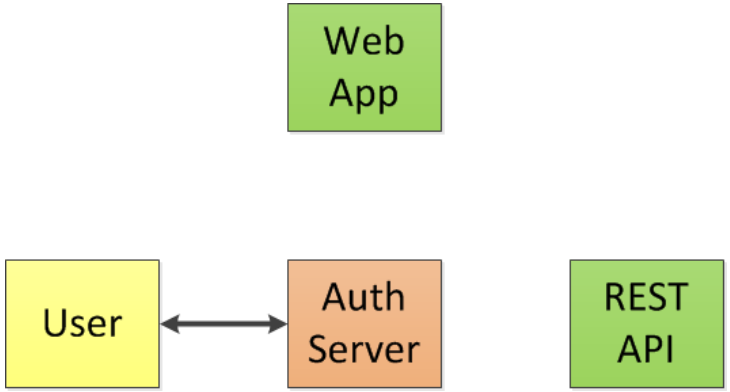
OpenID Connect – How To Get Tokens

- Step 3
  - Auth server redirects user to its login page because the user is not logged in.



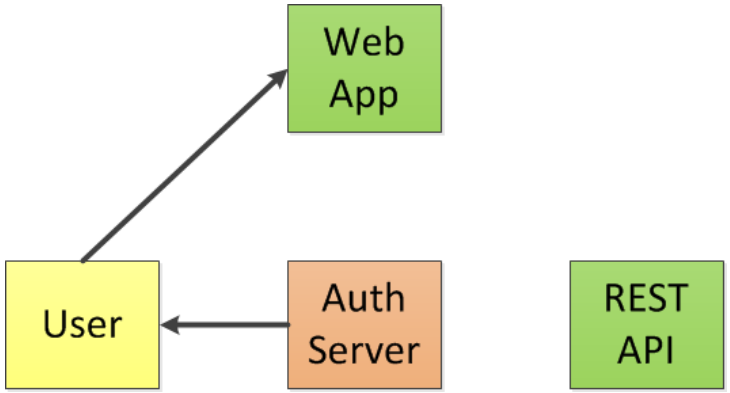
OpenID Connect – How To Get Tokens

- Step 4
  - User logs in, starts a session with the auth server, and is redirected back to the /authorize endpoint.
  - Control is given back to the user, who sees a page asking if the user permits the web app to access user's identity and manage user's orders on user's behalf.



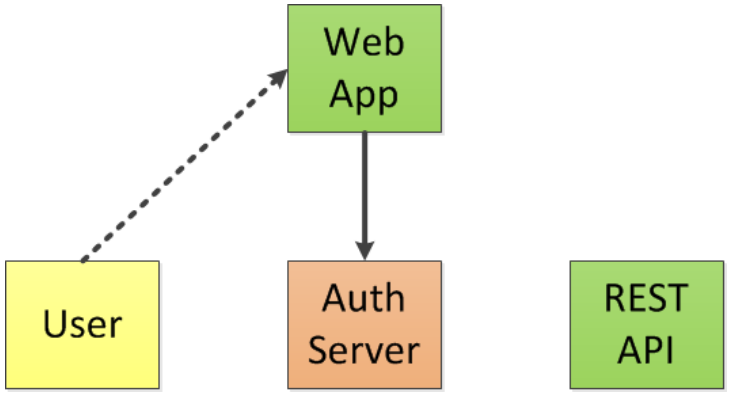
OpenID Connect – How To Get Tokens

- Step 5
  - User authorizes access. Auth server redirects the user back to the web app with a one time code in the query params of the redirect.



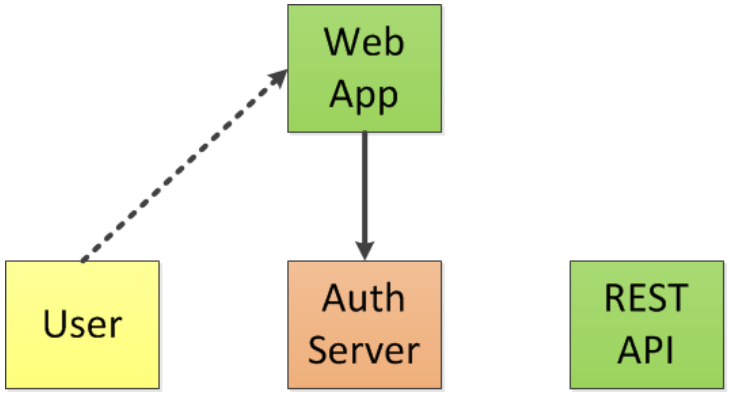
OpenID Connect – How To Get Tokens

- Step 6
  - Web App hits the /token endpoint with the one time code in the query params.
  - Auth server validates the code.



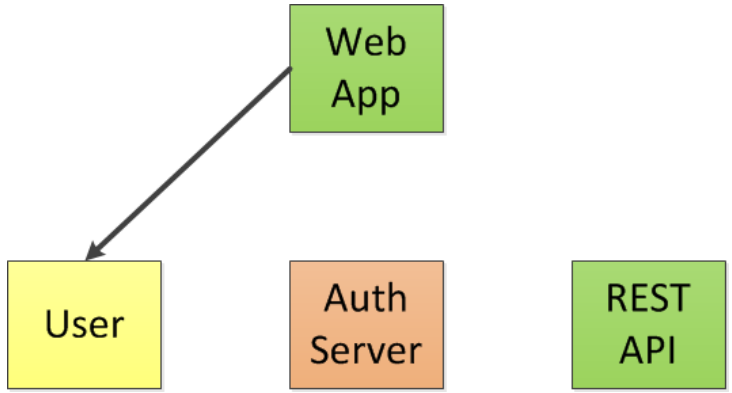
OpenID Connect – How To Get Tokens

- Step 7
  - Auth server responds with an access token (random string), and ID token (signed JWT).
  - Web app verifies the ID token, consumes its contents, and starts an authenticated session, and saves the access token in session.



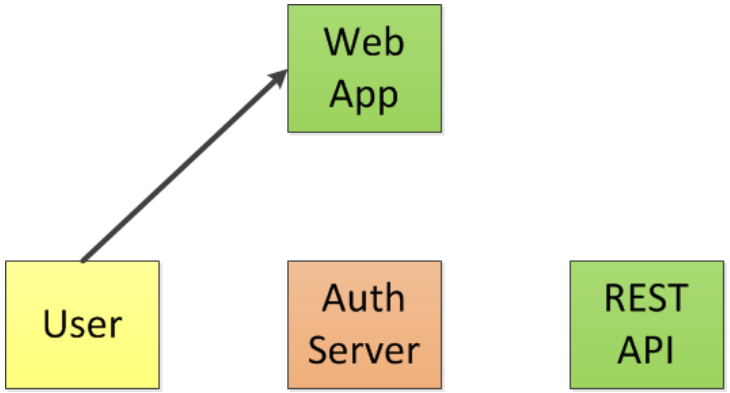
OpenID Connect – How To Use Tokens

- Step 8
  - Web app now gives control back to the user and responds with an order form.



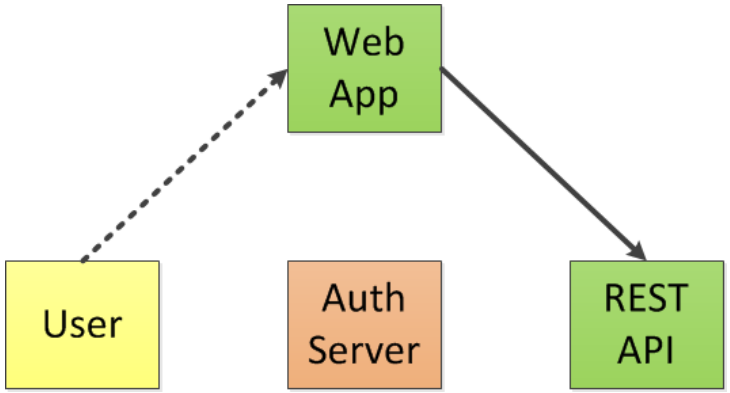
OpenID Connect – How To Use Tokens

- Step 9
  - User fills out and submits the order form.



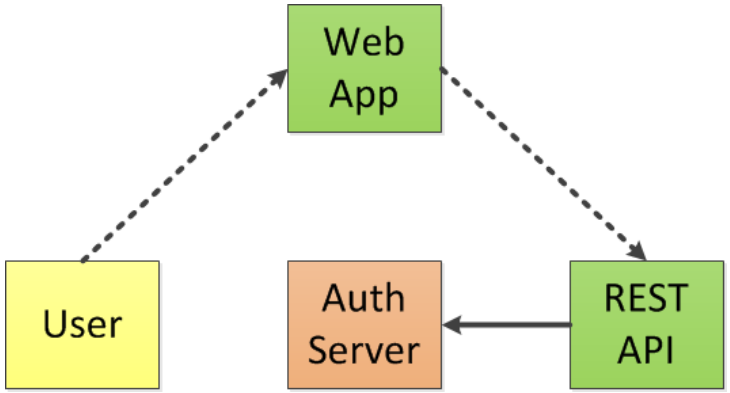
OpenID Connect – How To Use Tokens

- Step 10
  - The web app submits the order to the REST API with the access token that was stored in session.



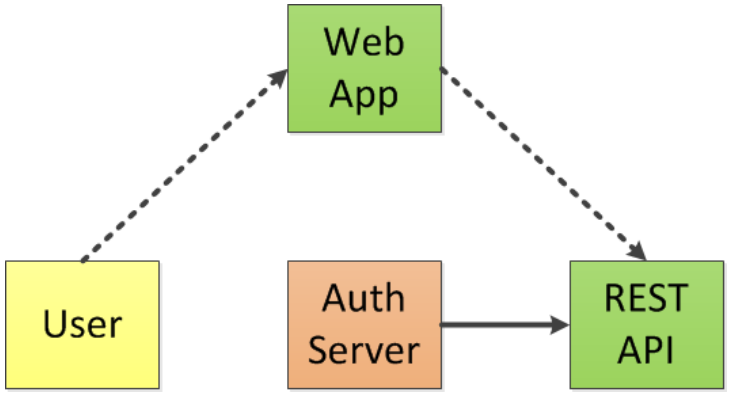
OpenID Connect – How To Use Tokens

- Step 11
  - The REST API needs to validate the token. It sends the token to the Auth server's token verification endpoint.



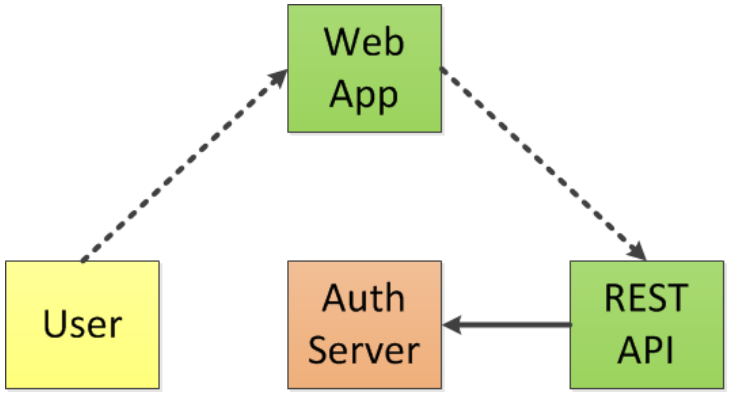
OpenID Connect – How To Use Tokens

- Step 12
  - The Auth server responds with the permissions (scopes) that the token grants. The REST API now knows that the request is authorized.



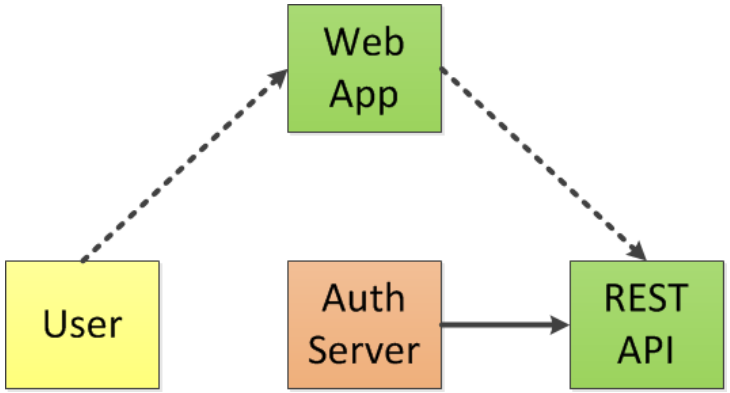
OpenID Connect – How To Use Tokens

- Step 13
  - But wait! Before saving the order, the REST API wants to populate it with other user information not contained in the request, e.g. address, phone number. The REST API make a request with that same token to the /userinfo endpoint.



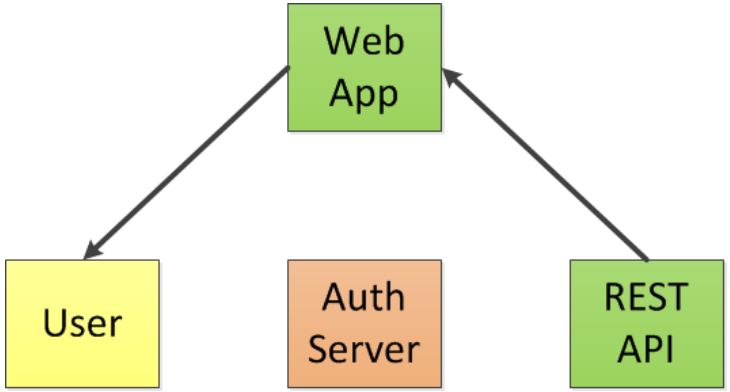
OpenID Connect – How To Use Tokens

- Step 14
  - The Auth server responds with the user's information. The REST API can now save the order.



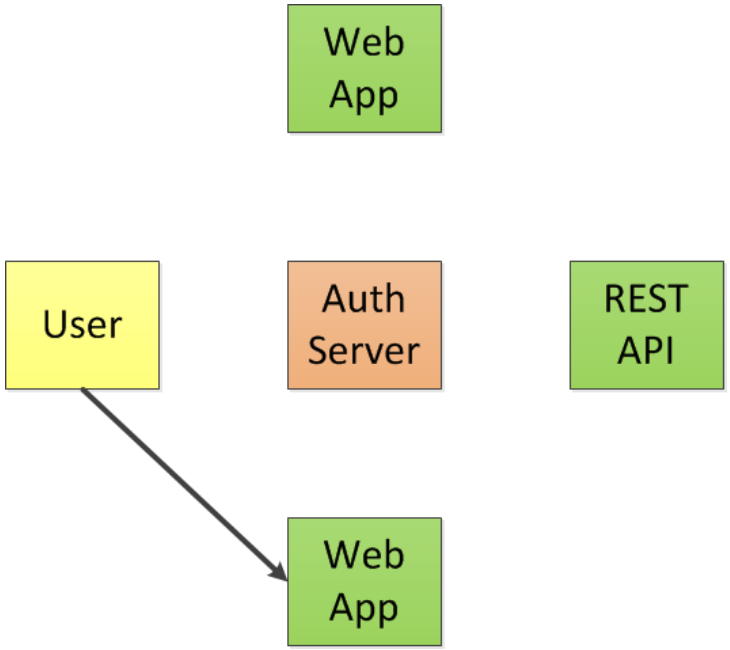
OpenID Connect – How To Use Tokens

- Step 15
  - Control is now given back to the user.



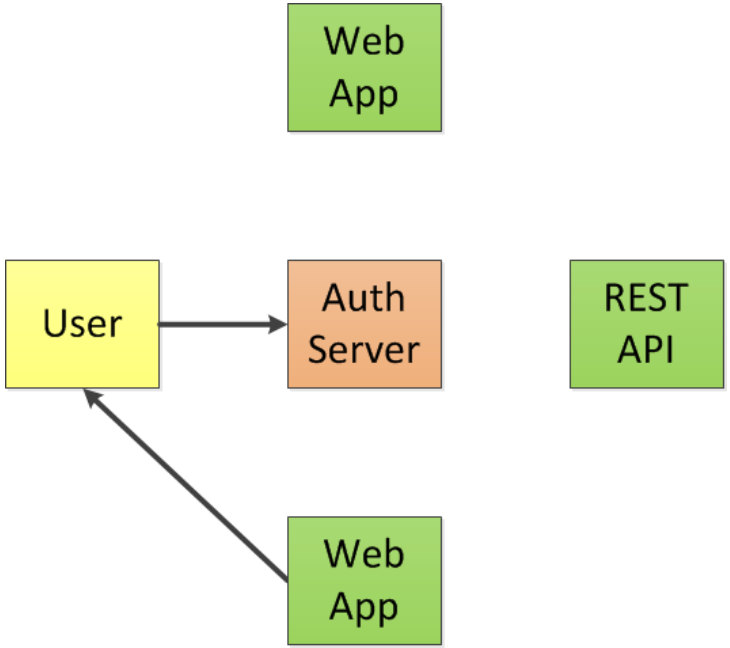
OpenID Connect – How To SSO

- Step 16
  - User wants to use Web App 2 to track their order and is not authenticated with it.



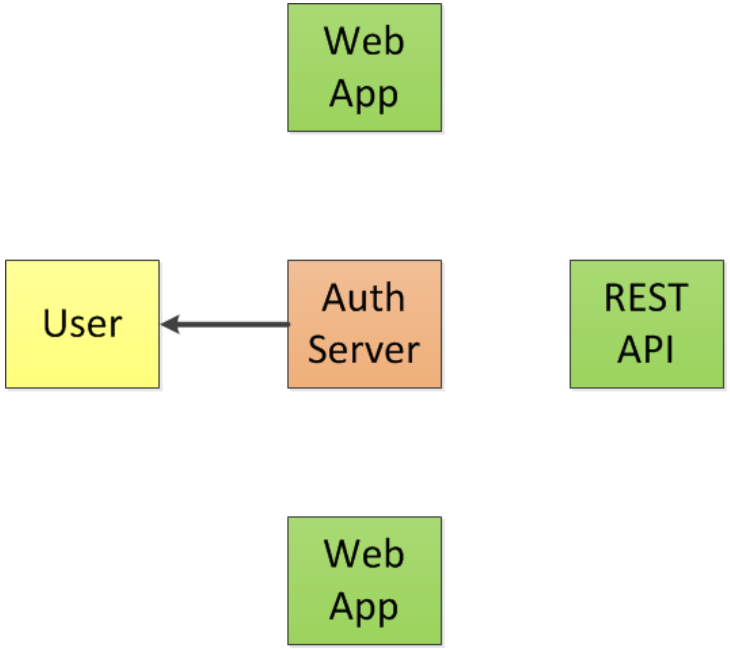
OpenID Connect – How To SSO

- Step 17
  - Web App 2 redirects the user to the Auth server’s /authorize endpoint



OpenID Connect – How To SSO

- Step 18
  - The user already has an authenticated session with the Auth server, so the server responds to the /authorize request with a page that asks if the user permits the web app to access user's identity and manage user's order on user's behalf and the flow continues as before.



Spring Cloud Security

Spring Cloud Security

- Spring Cloud Security offers a set of primitives for building secure applications and services with minimum fuss.
  - A declarative model which can be heavily configured externally (or centrally) lends itself to the implementation of large systems of co-operating, remote components, usually with a central identity management service.

Spring Cloud Security

- Building on Spring Boot and Spring Security OAuth2 we can quickly create systems that implement common patterns like single sign on, token relay and token exchange.
- Most features will be available with a single annotation and some configuration.

Spring Cloud Security

- Spring Cloud Security features:
  - SSO with OAuth2 and OpenID Connect servers;
  - Secure Resource Servers with tokens;
  - Relay SSO tokens from a front end to a back end service in a Zuul proxy;
  - Relay tokens between resource servers;
  - An auto-configured OAuth2RestTemplate;
  - An interceptor to make a Feign client behave like OAuth2RestTemplate (fetching tokens etc.)
  - Configure downstream authentication in a Zuul proxy;

Spring Cloud Security

- Caveats:
  - OpenID Connect ID tokens aren't directly consumed, but you can use /userinfo instead.
  - If the access token is a JWT containing identity claims you have everything you need.
- Dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-security</artifactId>
</dependency>
```

Spring Cloud Security

- If Access Tokens carry information
  - With scopes in the token
    - You can authorize the request yourself
  - With identity claims in the token
    - You know who the originator of the request is
  - With a signed token
    - You can validate the token's authenticity yourself
    - Your auth server won't become the bottleneck

Lab 9

Lab 9 – Microservices Security

- In this lab we will
  - Create the User Authentication & Authorization (UAA) server;
  - Add security layer between microservices;
  - Enable token exchange and relay;
  - Allow single sign on at API gateway;

Lab 9 – UAA

- UAA server will support OAuth2 with JWT tokens. For this purpose we will use Spring Security OAuth2 and Spring Security JWT dependencies with **`@EnableAuthorizationServer`** annotation.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

- Security configuration is handled by Spring Security.

Lab 9
Auth Server

Lab 9 – UAA Context Configuration

- The context path has to be explicit if you are running both the client and the auth server on localhost.
  - otherwise the cookie paths clash and the two apps cannot agree on a session identifier.
- For this reason we will use `/uaa` context for auth server.

```
spring.application.name: AuthService
server.port: 8500
server.contextPath: /uaa
```

Lab 9 – UAA OAuth2 Configuration

```
@Configuration
@EnableAuthorizationServer
public class OAuth2Configuration extends AuthorizationServerConfigurerAdapter {
    @Autowired
    @Qualifier("authenticationManagerBean")
    private AuthenticationManager authenticationManager;

    @Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(jwtTokenEnhancer());
    }

    @Bean
    protected JwtAccessTokenConverter jwtTokenEnhancer() {
        KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(
            new ClassPathResource("jwt.jks"), "mySecretKey".toCharArray()
        );
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        converter.setKeyPair(keyStoreKeyFactory.getKeyPair("jwt"));
        return converter;
    }
```

Lab 9 – UAA OAuth2 Configuration

```
    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
        .withClient("client")
        .secret("secret")
        .scopes("openid")
        .autoApprove(true)
        .authorizedGrantTypes("implicit","refresh_token", "password", "authorization_code");
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints.tokenStore(tokenStore()).tokenEnhancer(jwtTokenEnhancer())
        .authenticationManager(authenticationManager);
    }

    @Override
    public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {
        oauthServer.tokenKeyAccess("permitAll()").checkTokenAccess("permitAll()");
    }
}
```

Lab 9 – UAA Web Security Configuration

```
@Configuration
class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
                .antMatchers("/**").authenticated()
            .and().httpBasic()
            .and().formLogin().permitAll()
            .and().logout();
    }
```

Lab 9 – UAA Web Security Configuration

```
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("account")
                .password("account")
                .authorities("ACCOUNT_READ", "ACCOUNT_WRITE", "ACCOUNT_PROCESS")
                .and()
                .withUser("card")
                .password("card")
                .authorities("CARD_WRITE", "ACCOUNT_READ")
                .and()
                .withUser("client")
                .password("client")
                .authorities("CLIENT_READ", "CLIENT_WRITE", "ACCOUNT_READ", "CARD_READ")
                .and()
                .withUser("processing")
                .password("processing")
                .authorities("PROCESSING", "ACCOUNT_PROCESS");
    }
}
```

Lab 9
Resource Server

Lab 9 – Resource Server

Microservices now become resource servers, using same OAuth2 and JWT dependencies with **`@EnableResourceServer`** annotation.

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {
    @Bean
    @LoadBalanced
    public OAuth2RestTemplate oAuth2RestTemplate(OAuth2ProtectedResourceDetails details) {
        return new OAuth2RestTemplate(details);
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("/**").authenticated()
            .antMatchers(HttpMethod.GET, "/test").hasAuthority("PROCESSING");
    }
}
```

Lab 9 – Resource Server

- We also need to add security configuration to application.yml (in config-repo) as follows:

```
security:
  oauth2:
    client:
      clientId: client
      clientSecret: secret
      scope: openid
      accessTokenUri: http://localhost:8500/uaa/oauth/token
      userAuthorizationUri: http://localhost:8500/uaa/oauth/authorize
    resource:
      jwt:
        key-uri: http://localhost:8500/uaa/oauth/token_key
```

Lab 9
Rest Calls

Lab 9 – Rest Calls

- OAuth2 ships with OAuth2FeignRequestInterceptor to relay token with Feign Rest Calls.
- However, this will not work if you use Feign with Hystrix Thread isolation strategy.
  - The issue with Hystrix is that the security context is not propagated to threads.
  - It works with isolation strategy Semaphore.
- For testing purposes Hystrix can be disabled for Feign temporarily with
  - `feign.hystrix.enabled: false`
- Spring Security enables propagation of security context:
  - `hystrix.shareSecurityContext: true`
  - But, you have to code manually to support OAuth2, as default security context does not include OAuth2 specific properties.

Lab 9 – Rest Calls

- Use provided `OAuth2RestTemplate` to make OAuth2 authenticated REST requests with credentials of provided `OAuth2ProtectedResourceDetails`.
- An `OAuth2ClientContext` with `AccessTokenRequest` can be used to obtain a new token.
- Grant type with according properties is defined by resource details
  - `AuthorizationCodeResourceDetails`
  - `ResourceOwnerPasswordResourceDetails`
  - `ClientCredentialsResourceDetails`
  - `ImplicitResourceDetails`

Lab 9
Zuul SSO

Lab 9 – Zuul SSO

- To enable SSO at Zuul we still need OAuth2 and JWT dependencies.
- We also need Spring Cloud Security dependency.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-security</artifactId>
</dependency>
```

- And `@EnableOAuth2Sso` annotation.

Thank You!

LXFT
LISTED
NYSE

LUXOFT