ONE Authentication / Authorization Framework

SPRING SECURITY Enabler Document

Objective

This document aims to capture various features of Spring security and its relevance to our One Authentication framework. We will list down the capabilities of Spring security on various context for reference.

The Spring Security features will be described from the solution architecture for enabling Spring security. The document also captures key tenants of spring security and will provide additional information that helps to build a One Authentication Architecture.

The purpose of collating key information is to provide a perspective from a spring security and our approach to build Authentication framework using these features.

We would also capture scenarios and use cases for OAUTH2.0 Implementation using Spring Security for web application in another document.
Our ideology is to illustrate the spring security with context of authentication and derive a value relevant to our implementation.

We would also customize the content below and provide a comprehensive solution methodology with detailed components to build our One Authentication framework.

SPRING SECURITY

SECURITY

How can I implement security to my web/mobile applications so that there won't be any security breaches in my application ?

PASSWORDS

How to store passwords, validate them, encode, decode them using industry standard encryption algorithms ?

USERS & ROLES

How to maintain the user level security based on their roles and grants associated to them ?

MULTIPLE LOGINS

How can I implement a mechanism where the user will login only use and start using my application ?

FINE GRAINED SECURITY

How can I implement security at each level of my application using authorization rules ?

CSRF & CORS

What is CSRF attacks and CORS restrictions. How to overcome them?

JWT & OAUTH2

What is JWT and OAUTH2. How I can protect my web application using them?

PREVENTING ATTACKS

How to prevent security attacks like Brute force, stealing of data, session fixation

Table of Contents

INTRODUCTION TO SECURITY

WHAT & WHY

WHAT IS SECURITY?

Security is for protecting your data and business logic inside your web applications.

DIFFERENT TYPES OF SECURITY

Security for a web application will be implemented in different way like using firewalls, HTTPS, SSL,

SECURITY IS AN NON FUN REQ

Security is very important similar to scalability, performance and availability. No client will specifically asks that I need security.

WHY SECURITY IMPORTANT?

Security doesn't mean only loosing data or money but also the brand and trust from your users which you have built over years.

SECURITY FROM DEV PHASE

Security should be considered right from development phase itself along with business logic

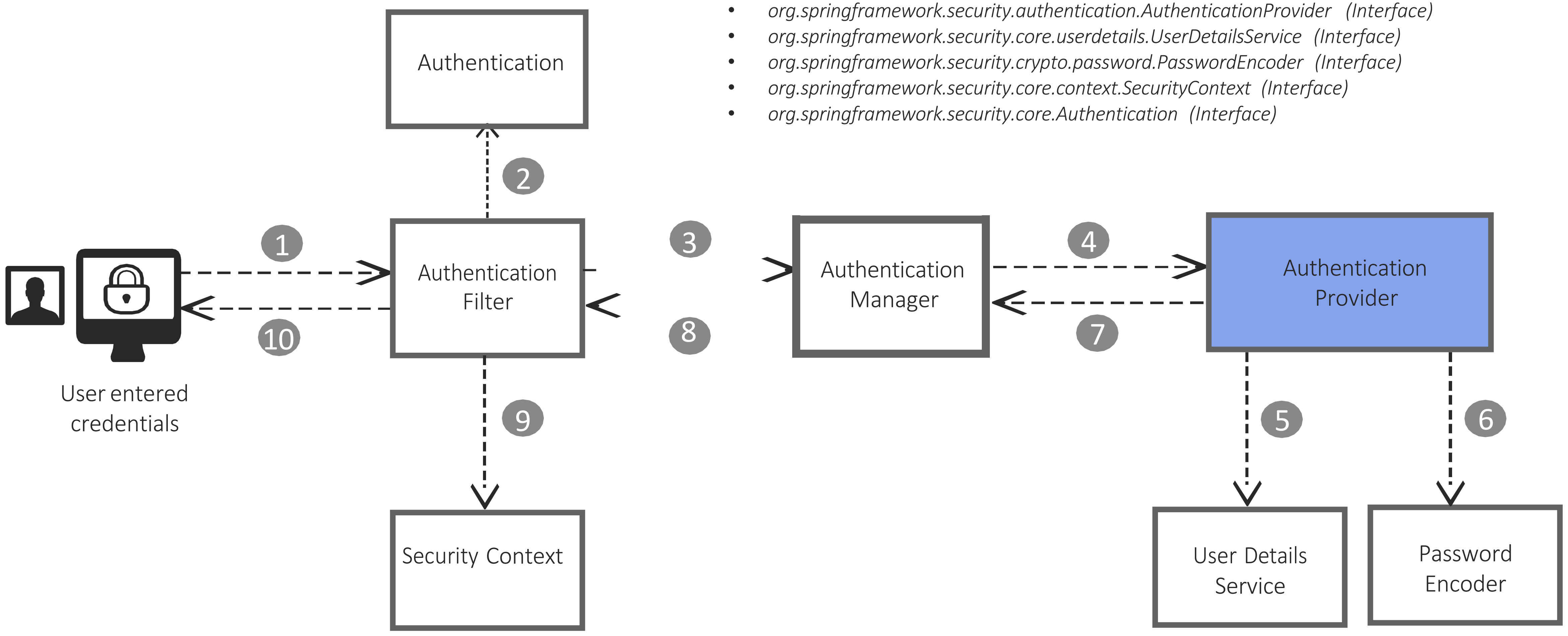
AVOIDING MOST COMMON ATTACKS

Using Security we should also avoid most common security attacks like CSRF, Broken Authentication inside our application.

SPRING SECURITY FLOW

INTERNAL FLOW

- *org.springframework.security.web.authentication.AuthenticationFilter (Class)*
- *org.springframework.security.authentication.AuthenticationManager (Interface)*
- *org.springframework.security.authentication.AuthenticationProvider (Interface)*
- *org.springframework.security.core.userdetails.UserDetailsService (Interface)*
- *org.springframework.security.crypto.password.PasswordEncoder (Interface)*
- *org.springframework.security.core.context.SecurityContext (Interface)*
- *org.springframework.security.core.Authentication (Interface)*



SPRING SECURITY FLOW

INTERNAL FLOW API DETAILS

1. AuthenticationFilter: A filter that intercepts and performs authentication of a particular request by delegating it to the authentication manager. If authentication is successful, the authentication details is set into SecurityContext.

2. Authentication: Using the supplied values from the user like username and password, the authentication object will be formed which will be given as an input to the AuthenticationManager interface.

3. AuthenticationManager: Once received request from filter it delegates the validating of the user details to the authentication provider.

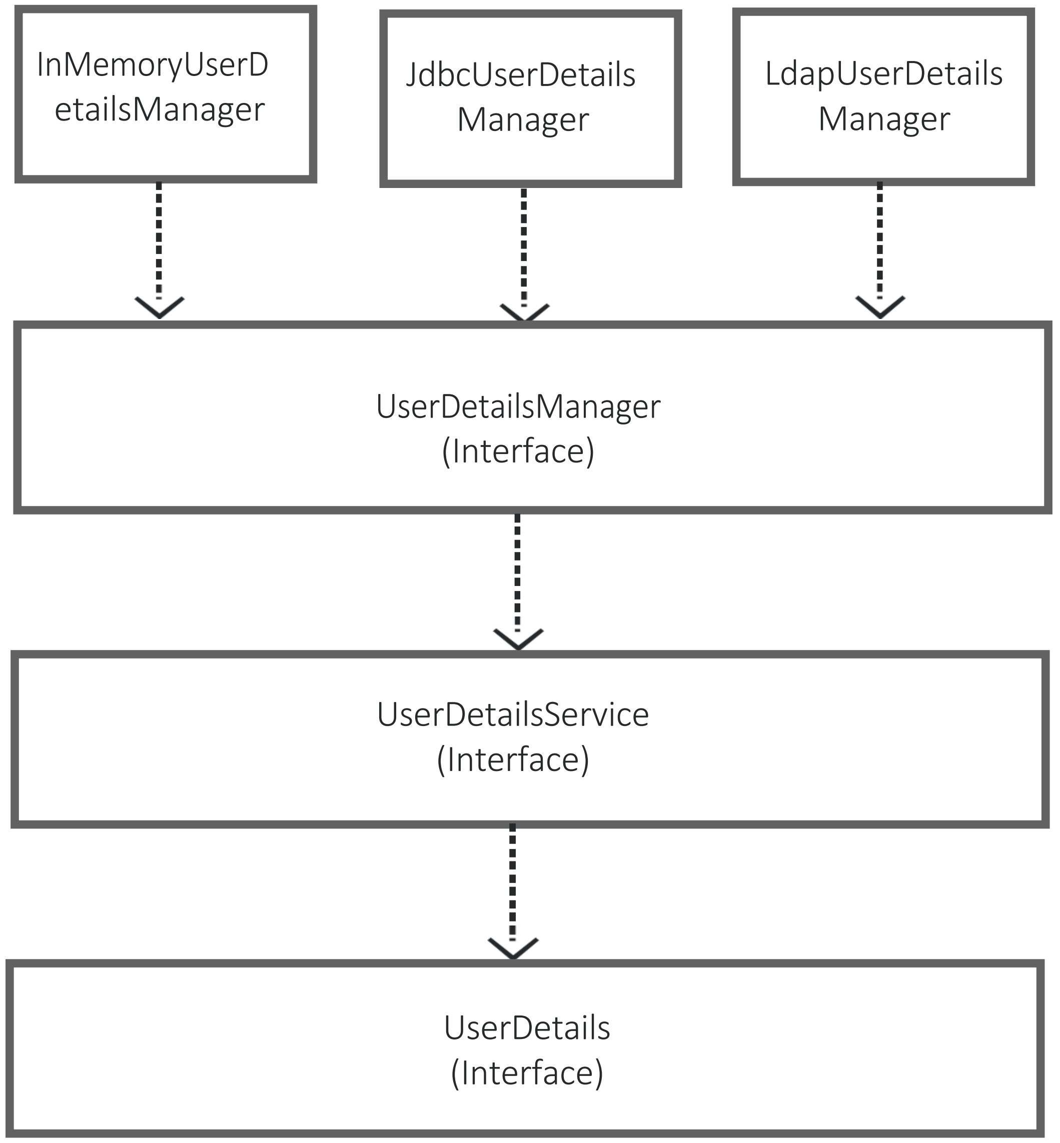
4. AuthenticationProvider: It has all the logic of validating user details using UserDetailsService and PasswordEncoder.

5. UserDetailsService: UserDetailsService retrieves UserDetails and implements the User interface using the supplied username.

6. PasswordEncoder: Service interface for encoding passwords.

7. SecurityContext: Interface defining the minimum security information associated with the current thread of execution. It holds the authentication data post successful authentication

USER MANAGEMENT



InMemoryUserDetailsManager
JdbcUserDetailsManager
LdapUserDetailsManager
UserDetailsManager (Interface)
UserDetailsService (Interface)
UserDetails (Interface)

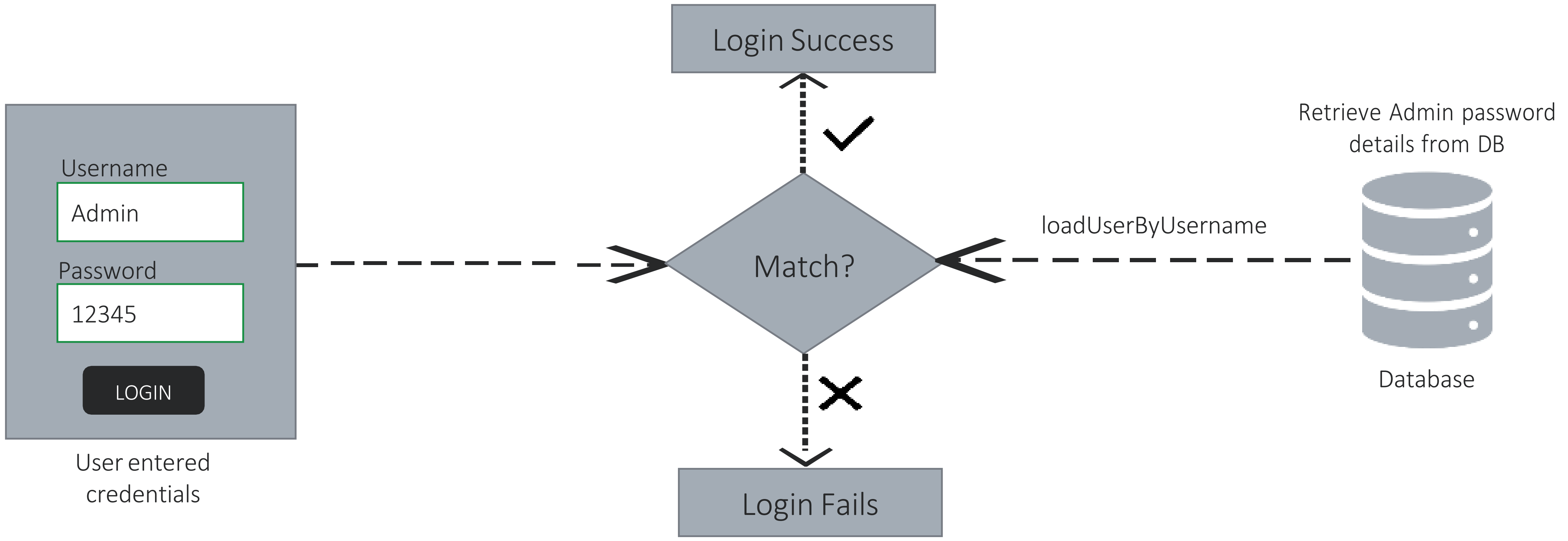
- createUser(UserDetails user)
- updateUser(UserDetails user)
- deleteUser(String username)
- changePassword(String oldPassword, String newPassword)
- userExists(String username)

- loadUserByUsername(String username)

- getPassword()
- getUsername()
- getAuthorities()
- isAccountNonExpired()
- isAccountNonLocked()
- isCredentialsNonExpired()
- isEnabled()

HOW OUR PASSWORDS VALIDATED

BY DEFAULT IN SPRING SECURITY



This approach has below issues,

- Integrity Issues
- Confidentiality

ENCODING

DETAILS

- Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography**.** It guarantees none of the 3 cryptographic properties of confidentiality**,** integrity, and authenticity because it involves no secret and is completely reversible**.**

- Encoding can be used for reducing the size of audio and video files. Each audio and video file format has a corresponding coder-decoder (codec) program that is used to code it into the appropriate format and then decodes for playback.

- It can’t be used for securing data, various publicly available algorithms are used for encoding.

Example**:** ASCII, BASE64, UNICODE

ENCRYPTION

DETAILS

- Encryption is defined as the process of transforming data in such a way that guarantees confidentiality**.** To achieve that**,** encryption requires the use of a secret which**,** in cryptographic terms**,** we call a **"**key**".**

- Encryption is divided into two categories: symmetric and asymmetric, where the major difference is the number of keys needed.

- In symmetric encryption algorithms, a single secret (key) is used to both encrypt and decrypt data. Only those who are authorized to access the data should have the single shared key in their possession.

Example: file system encryption, database encryption e.g. credit card details

- On the other hand, in asymmetric encryption algorithms, there are two keys in use: one public and one private. As their names suggest, the private key must be kept secret, whereas the public can be known to everyone. When applying encryption, the public key is used, whereas decrypting requires the private key. Anyone should be able to send us encrypted data, but only we should be able to decrypt and read it!

Example: TLS, VPN, SSH

HASHING

DETAILS

- In hashing, data is converted to the hash using some hashing function, which can be any number generated from string or text. Various hashing algorithms are MD5, SHA256. Data once hashed is non-reversible.

- One cannot determine the original data given only the output of a hashing algorithm.

- Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data

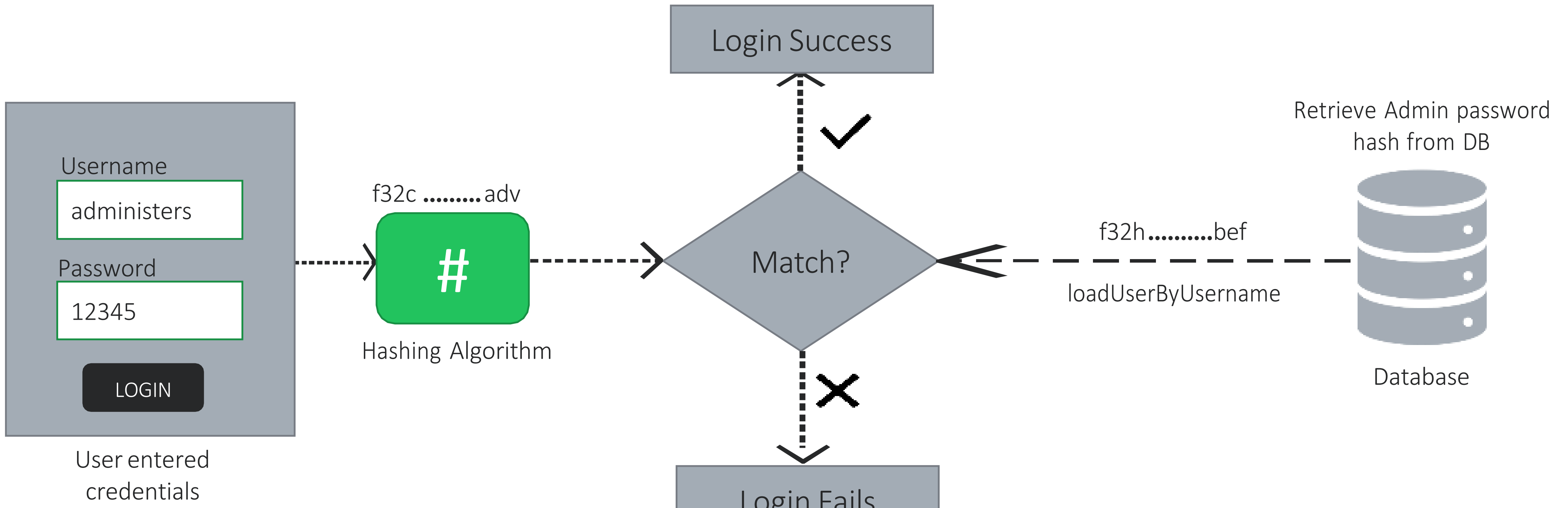
- You may have heard of hashing used in the context of passwords. Among many uses of hashing algorithms, this is one of the most well-known. When you sign up on a web app using a password, rather than storing your actual password, which would not only be a violation of your privacy but also a big risk for the web app owner, the web app hashes the password and stores only the hash.

- Then, the next time you log in, the web app again hashes your password and compares this hash with the hash stored earlier. If the hashes match, the web app can be confident that you know your password even though the web app doesn't have your actual password in storage.

Example: Password management, verify the integrity of the downloaded file

HOW OUR PASSWORDS VALIDATED

IF WE USE HASHING



DEFINITION OF THE PASSWORDENCODER

DETAILS

```
public interface PassswordEncoder {

        String encode(CharSequence rawPassword);

        boolean matches(CharSequence rawPassword, String encodedPassword);

        default boolean upgradeEncoding(String encodedPassword) {
                return false;
        }
}
```

Different Implementations of PasswordEncoders provided by Spring Security

- *NoOpPasswordEncoder*
- *StandardPasswordEncoder*
- *Pbkdf2PasswordEncoder*
- *BCryptPasswordEncoder*
- *SCryptPasswordEncoder*

Pbkdf2PasswordEncoder

DETAILS

- Password-Based Key Derivation Function 2 (PBKDF2) is a pretty easy slow-hashing function that performs an HMAC (Hashed Message Authentication Code) as many times as specified by an iteration's argument.
- The three parameters received by the last call are the value of a key used for the encoding process, the number of iterations used to encode the password, and the size of the hash. The second and third parameters can influence the strength of the result.
- You can choose more or fewer iterations as well as the length of the result. The longer the hash, the more powerful the password is.

```
PasswordEncoder p = new Pbkdf2PasswordEncoder();
PasswordEncoder p = new Pbkdf2PasswordEncoder("secret");
PasswordEncoder p = new Pbkdf2PasswordEncoder("secret", 185000, 256);
```

Bcrypt & Scrypt PasswordEncoder

DETAILS

- BCryptPasswordEncoder uses a BCrypt strong hashing function to encode the password. You could instantiate the BCryptPasswordEncoder by calling the no-arguments constructor. But you also have the option to specify a strength coefficient representing the log rounds used in the encoding process. Moreover, you can as well alter the SecureRandom instance used for encoding.

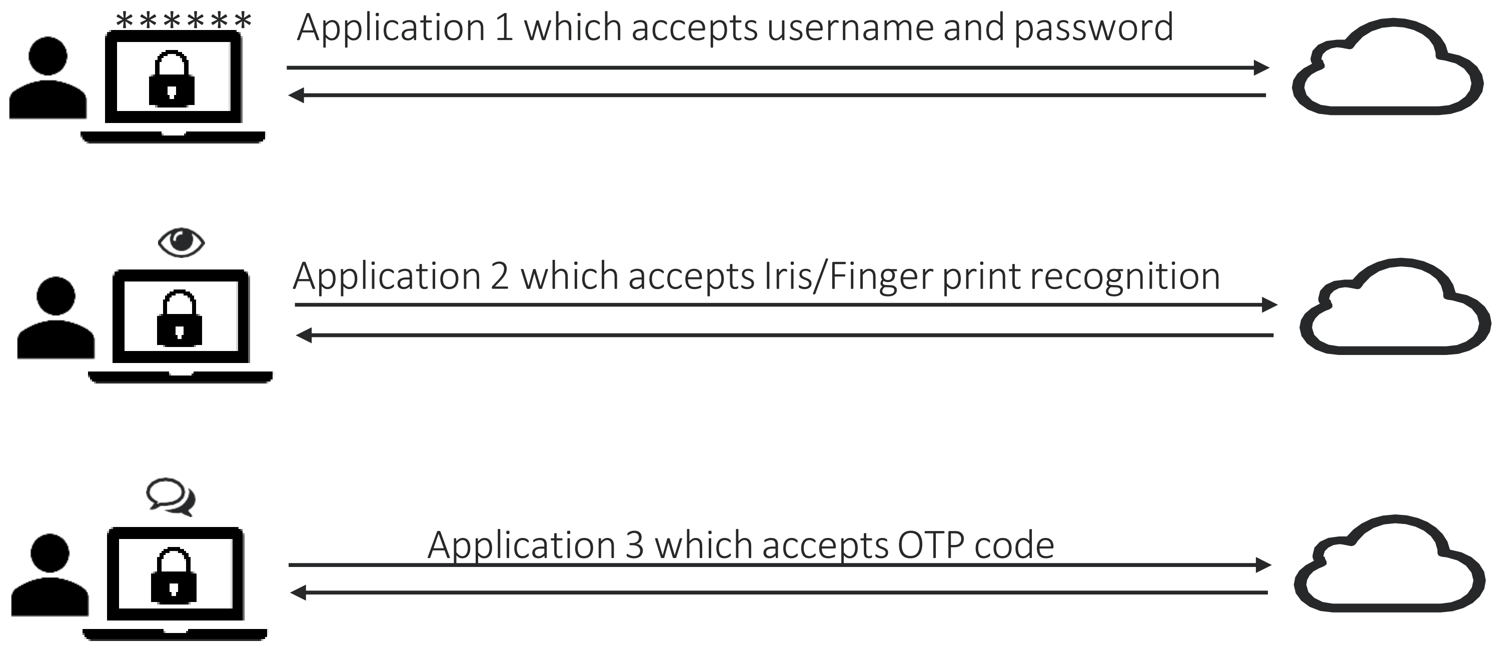
```
PasswordEncoder p = new BCryptPasswordEncoder();
PasswordEncoder p = new BCryptPasswordEncoder(4);
SecureRandom s = SecureRandom.getInstanceStrong();
PasswordEncoder p = new BCryptPasswordEncoder(4, s);
```

- SCryptPasswordEncoder uses a SCrypt hashing function to encode the password. For the SCryptPasswordEncoder, you have two options to create its instances:

```
PasswordEncoder p = new SCryptPasswordEncoder();
PasswordEncoder p = new SCryptPasswordEncoder(16384, 8, 1, 32, 64);
```

AUTHENTICATION PROVIDER

WHY DO WE NEED IT?


******
Application 1 which accepts username and password
Application 2 which accepts Iris/Finger print recognition
Application 3 which accepts OTP code

The AuthenticationProvider in Spring Security takes care of the authentication logic.

The default implementation of the AuthenticationProvider delegates the responsibility of finding the user in the system to a UserDetailsService and PasswordEncoder for password validation.

But if we have a custom authentication requirement that is not fulfilled by Spring Security framework then we can build our own authentication logic by implementing the AuthenticationProvider interface.

AUTHENTICATION PROVIDER DEFINITION

DETAILS

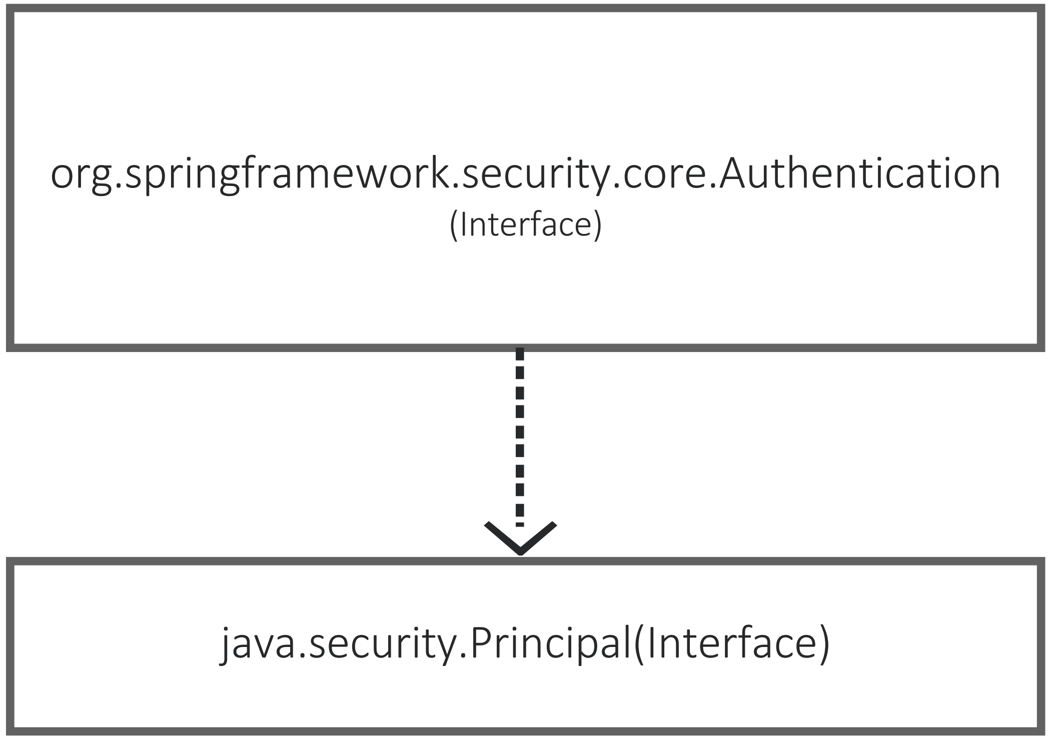
```
public interface AuthenticationProvider {

        Authentication authenticate(Authentication authentication)
                        throws AuthenticationException;

        boolean supports(Class<?> authentication);
}
```

- The authenticate() method receives an Authentication object as a parameter and returns an Authentication object as well. We implement the authenticate() method to define the authentication logic.

- The second method in the AuthenticationProvider interface is supports(Class<?> authentication). You'll implement this method to return true if the current AuthenticationProvider supports the type provided as the Authentication object.

AUTHENTICATION & PRINCIPAL

DETAILS


org.springframework.security.core.Authentication
(Interface)
java.security.Principal(Interface)

- getAuthorities()
- getCredentials()
- getDetails()
- getPrincipal()
- isAuthenticated()
- setAuthenticated(boolean isAuthenticated)

- getName**()**

CORS & CSRF

SPRING SECURITY APPROACH

CORS

CROSS-ORIGIN RESOURCE SHARING

CSRF

CROSS-SITE REQUEST FORGERY

SPRING SECURITY

HOW TO HANDLE THEM USING THE SPRING SECURITY FRAMEWORK?

CROSS-ORIGIN RESOURCE SHARING (CORS)

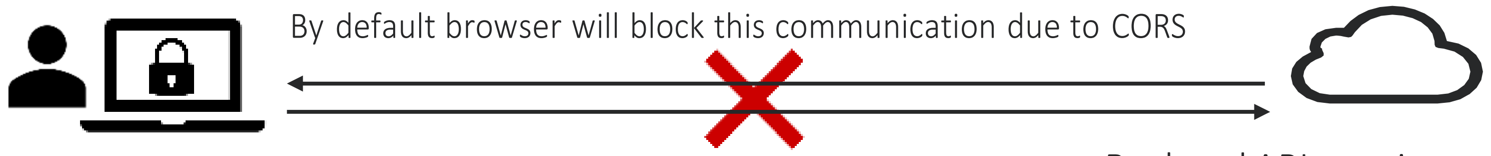
HOW TO HANDLE IT USING SPRING SECURITY

1. CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin.

2. For example, if a UI app wishes to make an API call running on a different domain, it would be blocked from doing so by default due to CORS. So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

3. "other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:

    - a different scheme (HTTP or HTTPS)
    - a different domain
    - a different port


By default browser will block this communication due to CORS

UI App running on https://domain1.com

Backend API running on https://domain2.com

CROSS-ORIGIN RESOURCE SHARING (CORS)

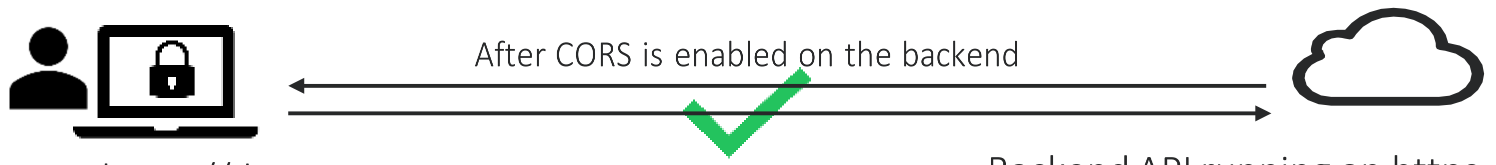
HOW TO HANDLE IT USING SPRING SECURITY

4. But what if there is a legitimate scenario where cross-origin access is desirable or even necessary. For example, in our sample application where the UI and backend are hosted on two different ports.

5. When a server has been configured correctly to allow cross-origin resource sharing, some special headers will be included. Their presence can be used to determine that a request supports CORS. Web browsers can use these headers to determine whether a request should continue or fail.

6. First the browser sends a pre-flight request to the backend server to determine whether it supports CORS or not. The server can then respond to the pre-flight request with a collection of headers:

   - *Access-Control-Allow-Origin: Defines which origins may have access to the resource. A '*' represents any origin*
   - *Access-Control-Allow-Methods: Indicates the allowed HTTP methods for cross-origin requests*
   - *Access-Control-Allow-Headers: Indicates the allowed request headers for cross-origin requests*
   - *Access-Control-Allow-Credentials : Indicates whether or not the response to the request can be exposed when the credentials flag is true.*
   - *Access-Control-Max-Age: Defines the expiration time of the result of the cached preflight request*



UI App running on https://domain1.com

Backend API running on https://domain2.com

CROSS-SITE REQUEST FORGERY (CSRF)

HOW TO HANDLE IT USING SPRING SECURITY

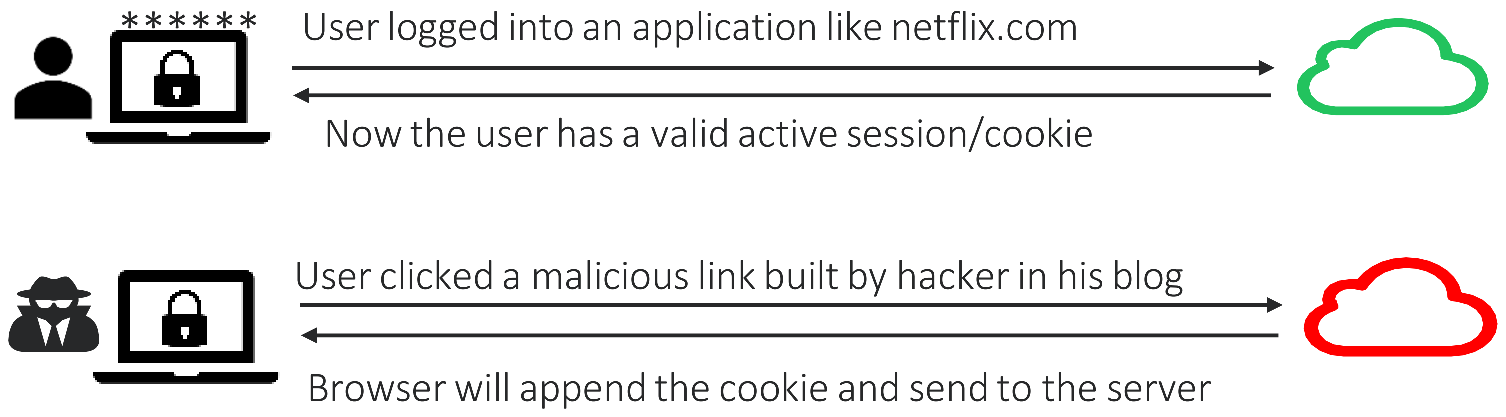
1. A typical Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.

2. Consider a website netflix.com and the attacker’s website travelblog.com. Also assume that the victim is logged in and his session is being maintained by cookies. The attacker will:

    - Find out what action he needs to perform on behalf of the victim and find out its endpoint (for example, to change password on netflix.com a POST request is made to the website that contains new password as the parameter)
    - Place HTML code on his website travelblog.com that will imitate a legal request to netflix.com (for example, a form with method as post and a hidden input field that contains the new password).
    - Make sure that the form is submitted by either using “autosubmit” or luring the victim to click on a submit button.

CROSS-SITE REQUEST FORGERY (CSRF)

HOW TO HANDLE IT USING SPRING SECURITY


******
User logged into an application like netflix.com
Now the user has a valid active session/cookie
User clicked a malicious link built by hacker in his blog
Browser will append the cookie and send to the server

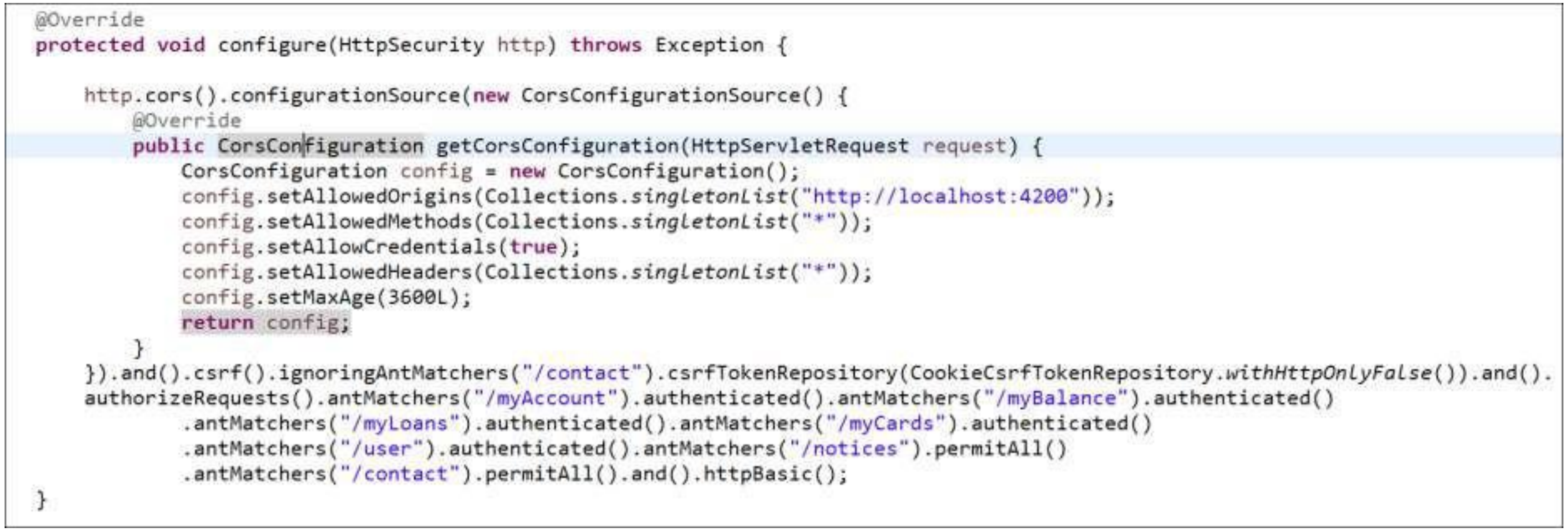
- When the victim visits travelblog.com and that form is submitted, the victim's browser makes a request to netflix.com for a password change. Also the browser appends the cookies with the request. The server treats it as a genuine request and resets the victim's password to the attacker's supplied value. This way the victim's account gets taken over by the attacker.

- There are many proposed ways to implement CSRF protection on server side, among which the use of CSRF tokens is most popular. A CSRF token is a string that is tied to a user's session but is not submitted automatically. A website proceeds only when it receives a valid CSRF token along with the cookies, since there is no way for an attacker to know a user specific token, the attacker can not perform actions on user's behalf.

Code Sample

CORS & CSRF

CORS & CSRF, how to resolve it using Spring Security



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

    http.cors().configurationSource(new CorsConfigurationSource() {
        @Override
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
            CorsConfiguration config = new CorsConfiguration();
            config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
            config.setAllowedMethods(Collections.singletonList("*"));
            config.setAllowCredentials(true);
            config.setAllowedHeaders(Collections.singletonList("*"));
            config.setMaxAge(3600L);
            return config;
        }
    }).and().csrf().ignoringAntMatchers("/contact").csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()).and().
    authorizeRequests().antMatchers("/myAccount").authenticated().antMatchers("/myBalance").authenticated()
            .antMatchers("/myLoans").authenticated().antMatchers("/myCards").authenticated()
            .antMatchers("/user").authenticated().antMatchers("/notices").permitAll()
            .antMatchers("/contact").permitAll().and().httpBasic();
}
```

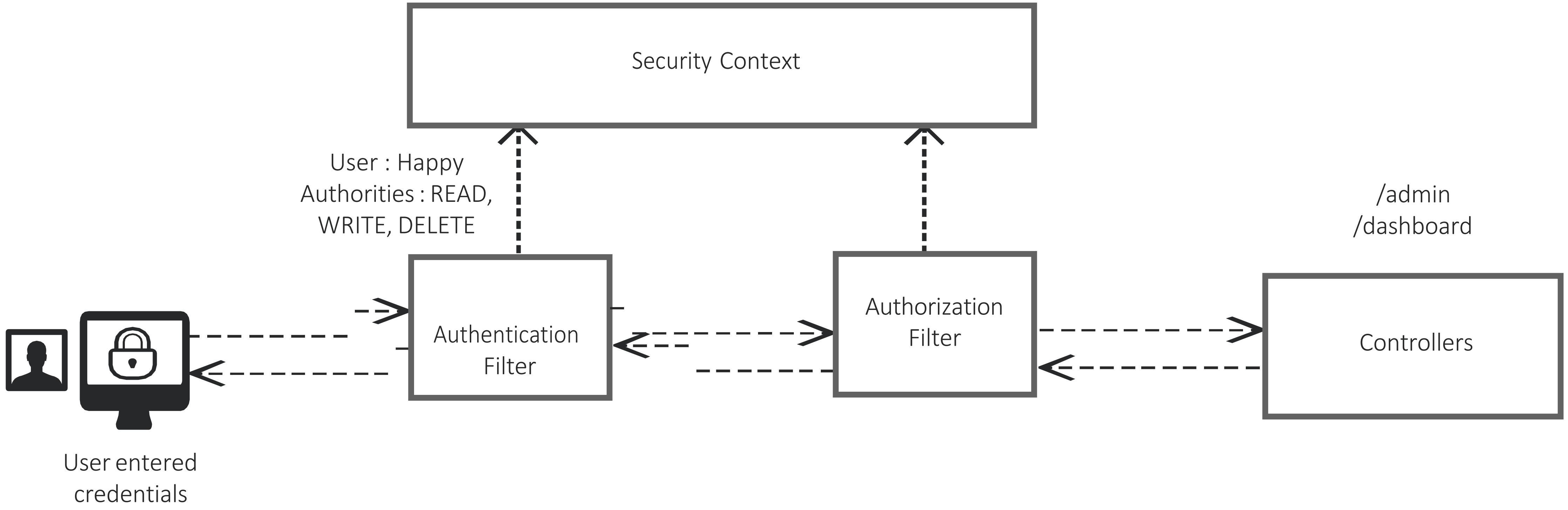
AUTHENTICATION & AUTHORIZATION

DETAILS & COMPARISION

AUTHENTICATION	AUTHORIZATION
In authentication, the identity of users are checked for providing the access to the system.	In authorization, person's or user's authorities are checked for accessing the resources.
Authentication done before authorization	Authorization always happens after authentication.
It needs usually user's login details	it needs user's privilege or roles
If authentication fails usually we will get 401 error response	If authentication fails usually we will get 403 error response
For example as a Bank customer/employee in order to perform actions in the app, we need to prove our identity	Once logged into the application, my roles, authorities will decide what kind of actions I can do

AUTHN & AUTHZ

INTERNAL FLOW



When the Client makes a request with the credentials, the authentication filter will intercept the request and validate if the person is valid and is he/she the same person whom they are claiming. Post authentication the filter stores the UserDetails in the SecurityContext.

The UserDetails will have his username, authorities etc. Now the authorization filter will intercept and decide whether the person has access to the given path based on this authorities stored in the SecurityContext. If authorized the request will be forwarded to the applicable controllers.

HOW AUTHORITIES STORED?

IN SPRING SECURITY

Inside UserDetails which is a contract of the User inside the Spring Security, the authorities will be stored in the form of Collection of GrantedAuthority. These authorities can be fetched using the method getAuthorities()

```
public interface UserDetails {
		Collection<? extends GrantedAuthority> getAuthorities();
}
```

Inside GrantedAuthority interface we have a getAuthority() method which will return the authority/role name in the form of a string. Using this value the framework will try to validate the authorities of the user with the implementation of the application.

```
public interface GrantedAuthority {
		String getAuthority();
}
```

CONFIGURING AUTHORITIES

IN SPRING SECURITY

In Spring Security the authorities of the user can be configured and validated using the following ways,

- hasAuthority() —Accepts a single authority for which the endpoint will be configured and user will be validated against the single authority mentioned. Only users having the same authority configured can call the endpoint.

- hasAnyAuthority() — Accepts multiple authorities for which the endpoint will be configured and user will be validated against the authorities mentioned. Only users having any of the authority configured can call the endpoint.

- access() — Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring authorities which are not possible with the above methods. We can use operators like OR, AND inside access() method.

AUTHORITY & ROLE

IN SPRING SECURITY

GRANTED AUTHORITY

- Authority is like an individual privilege.
- Restricting access in a fine-grained manner
- Ex: READ, UPDATE, DELETE

ROLE

- ROLE is a group of privileges
- Restricting access in a coarse-grained manner
- Ex: ROLE_ADMIN, ROLE_USER

- The names of the authorities/roles are arbitrary in nature and these names can be customized as per the business requirement
- Roles are also represented using the same contract GrantedAuthority in Spring Security.
- When defining a role, its name should start with the ROLE_ prefix. This prefix specifies the difference between a role and an authority.

CONFIGURING ROLES

IN SPRING SECURITY

In Spring Security the roles of the user can be configured and validated using the following ways,

- hasRole() —Accepts a single role name for which the endpoint will be configured and user will be validated against the single role mentioned. Only users having the same role configured can call the endpoint.

- hasAnyRole() — Accepts multiple roles for which the endpoint will be configured and user will be validated against the roles mentioned. Only users having any of the role configured can call the endpoint.

- access() — Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring roles which are not possible with the above methods. We can use operators like OR, AND inside access() method.

Note :

- *ROLE_ prefix only to be used while configuring the role in DB. But when we configure the roles, we do it only by its name.*
- *access() method can be used not only for configuring authorization based on authority or role but also with any special requirements that we have. For example we can configure access based on the country of the user or current time/date.*

MATCHERS METHODS

IN SPRING SECURITY

Spring Security offers three types of matchers methods to configure endpoints security,

1) MVC matchers, 2) Ant matchers, 3) Regex matchers

- MVC matchers

MvcMatcher() uses Spring MVC's HandlerMappingIntrospector to match the path and extract variables.

- *mvcMatchers(HttpMethod method, String... patterns)— We can specify both the HTTP method and path pattern to configure restrictions*

```
http.authorizeRequests().mvcMatchers(HttpMethod.POST,  "/example").authenticated()
.mvcMatchers(HttpMethod.GE , "/example").permitAll()
.anyRequest().denyAll();
```

- *mvcMatchers(String... patterns)—We can specify only path pattern to configure restrictions and all the HTTP methods will be allowed.*

```
http.authorizeRequests().mvcMatchers( "/profile/edit/**").authenticated()
.anyRequest().permitAll();
```

Note :

- *** indicates any number of paths. For example, /x/**/z will match both /x/y/z and /x/y/abc/z*
- *Single * indicates single path. For example /x/*/z will match /x/y/z, /x/abc/z but not /x/y/abc/z*

MATCHERS METHODS

IN SPRING SECURITY

- ANT matchers is an implementation for Ant-style path patterns. Part of this mapping code has been kindly borrowed from Apache Ant.

  - *antMatchers(HttpMethod method, String... patterns)— We can specify both the HTTP method and path pattern to configure restrictions*

    ```
    http.authorizeRequests().antMatchers(HttpMethod.POST,  "/example").authenticated()
    .antMatchers(HttpMethod.GE , "/example").permitAll()
    .anyRequest().denyAll();
    ```

  - *antMatchers(String... patterns)—We can specify only path pattern to configure restrictions and all the HTTP methods will be allowed.*

    ```
    http.authorizeRequests().antMatchers( "/profile/edit/**").authenticated()
    .anyRequest().permitAll();
    ```

  - *antMatchers(HttpMethod method)—We can specify only HTTP method ignoring path pattern to configure restrictions. This is same as antMatchers(httpMethod, "/**")*

    ```
    http.authorizeRequests().antMatchers(HttpMethod.POST).authenticated()
    .anyRequest().permitAll();
    ```

Note : Generally mvcMatcher is more secure than an antMatcher. As an example

- *antMatchers("/secured") matches only the exact /secured URL*
- *mvcMatchers("/secured") matches /secured as well as /secured/, /secured.html, /secured.xyz*

MATCHERS METHODS

IN SPRING SECURITY

- REGEX matchers

Regexes can be used to represent any format of a string, so they offer unlimited possibilities for this matter

- *regexMatchers(HttpMethod method, String regex)— We can specify both the HTTP method and path regex to configure restrictions*
  ```
  http.authorizeRequests().regexMatchers(HttpMethod.GET, ".*/(en|es|zh) ").authenticated()
  .anyRequest(). denyAll();
  ```
- *regexMatchers(String regex)—We can specify only path regex to configure restrictions and all the HTTP methods will be allowed.*
  ```
  http.authorizeRequests().regexMatchers(".*/(en|es|zh)  ").authenticated()
  .anyRequest(). denyAll();
  ```

FILTERS IN SPRING SECURITY

IN AUTHN & AUTHZ FLOW

- Lot of times we will have situations where we need to perform some house keeping activities during the authentication and authorization flow. Few such examples are,
  - Input validation
  - Tracing, Auditing and reporting
  - Logging of input like IP Address etc.
  - Encryption and Decryption
  - Multi factor authentication using OTP

- All such requirements can be handled using HTTP Filters inside Spring Security. Filters are servlet concepts which are leveraged in Spring Security as well.

- We already saw some in built filters of Spring security framework like Authentication filter, Authorization filter, CSRF filter, CORS filter in the previous sections.

- A filter is a component which receives requests, process its logic and handover to the next filter in the chain.

- Spring Security is based on a chain of servlet filters. Each filter has a specific responsibility and depending on the configuration, filters are added or removed. We can add our custom filters as well based on the need.

FILTERS IN SPRING SECURITY

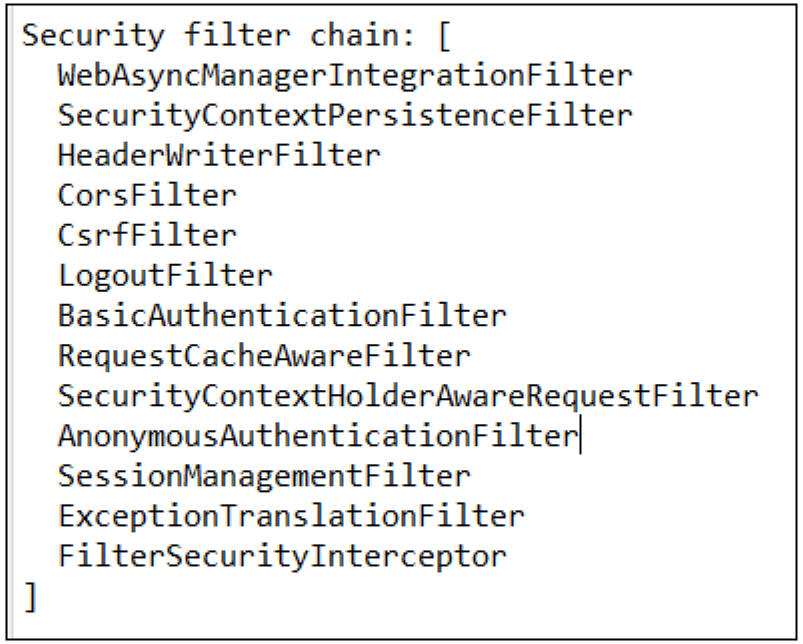
IN AUTHN & AUTHZ FLOW

- We can always check the registered filters inside Spring Security with the below configurations,

  1. @EnableWebSecurity(debug = true) – We need to enable the debugging of the security details
  2. Enable logging of the details by adding the below property in application.properties

     logging.level.org.springframework.security.web.FilterChainProxy=DEBUG

- Below are the some of the internal filters of Spring Security that gets executed in the authentication flow,



```
Security filter chain: [
  WebAsyncManagerIntegrationFilter
  SecurityContextPersistenceFilter
  HeaderWriterFilter
  CorsFilter
  CsrfFilter
  LogoutFilter
  BasicAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  FilterSecurityInterceptor
]
```

IMPLEMENTING FILTERS

IN SPRING SECURITY

- We can create our own filters by implementing the Filter interface from the javax.servlet package. Post that we need to override the doFilter() method to have our own custom logic. This method receives as parameters the ServletRequest, ServletResponse and FilterChain.

  - *ServletRequest—It represents the HTTP request. We use the ServletRequest object to retrieve details about the request from the client.*
  - *ServletResponse—It represents the HTTP response. We use the ServletResponse object to modify the response before sending it back to the client or further along the filter chain.*
  - *FilterChain—The filter chain represents a collection of filters with a defined order in which they act. We use the FilterChain object to forward the request to the next filter in the chain.*

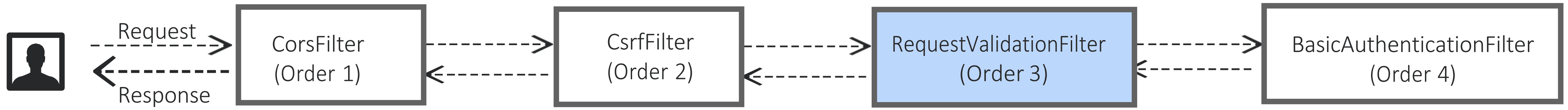
- You can add a new filter to the spring security chain either before, after, or at the position of a known one. Each position of the filter is an index (a number), and you might find it also referred to as “the order.”

- Below are the methods available to configure a custom filter in the spring security flow,
  - *addFilterBefore(filter, class) – adds a filter before the position of the specified filter class*
  - *addFilterAfter(filter, class) – adds a filter after the position of the specified filter class*
  - *addFilterAt(filter, class) – adds a filter at the location of the specified filter class*

ADD FILTER BEFORE

IN SPRING SECURITY

addFilterBefore(filter, class) – It will add a filter before the position of the specified filter class.

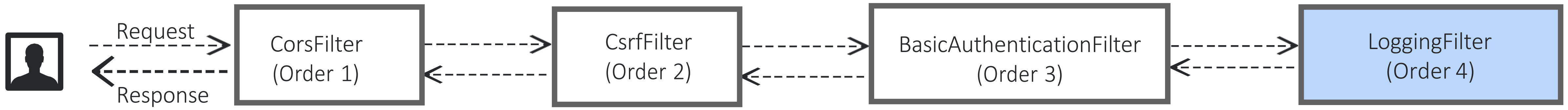

Request
Response
CorsFilter (Order 1)
CsrfFilter (Order 2)
RequestValidationFilter (Order 3)
BasicAuthenticationFilter (Order 4)

Here we add a filter just before authentication to write our own custom validation where the input email provided should not have the string ‘test’ inside it.

ADD FILTER AFTER

IN SPRING SECURITY

addFilterAfter(filter, class) – It will add a filter after the position of the specified filter class

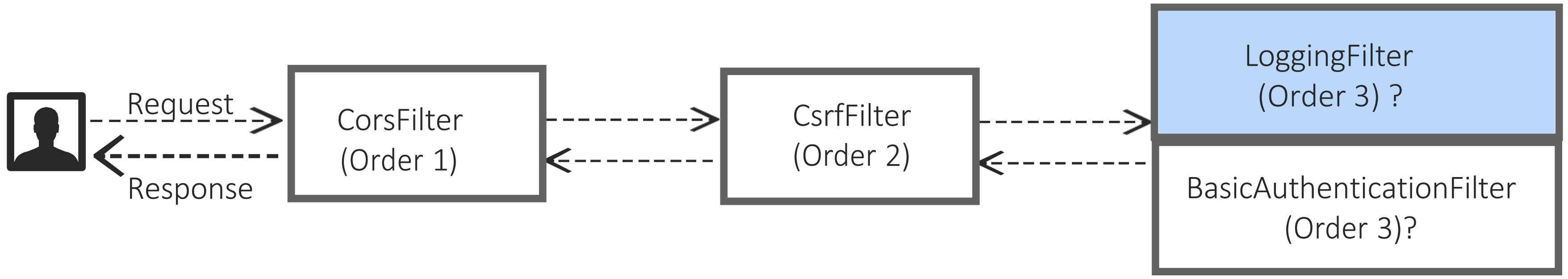


Here we add a filter just after authentication to write a logger about successful authentication and authorities details of the logged in users.

ADD FILTER AT

IN SPRING SECURITY

- addFilterAt(filter, class) – Adds a filter at the location of the specified filter class. But the order of the execution can’t be guaranteed. This will not replace the filters already present at the same order. Since we will not have control on the order of the filters and it is random in nature we should avoid providing the filters at same order.



INTERNAL FILTERS

IN SPRING SECURITY

GenericFilterBean

- This is an abstract class filter bean which allows you to use the initialization parameters and configurations done

inside the web.xml

OncePerRequestFilter

- Spring doesn’t guarantee that your filter will be called only once.But if we have a scenario where we need to make sure to execute our filter only once then we can use this.

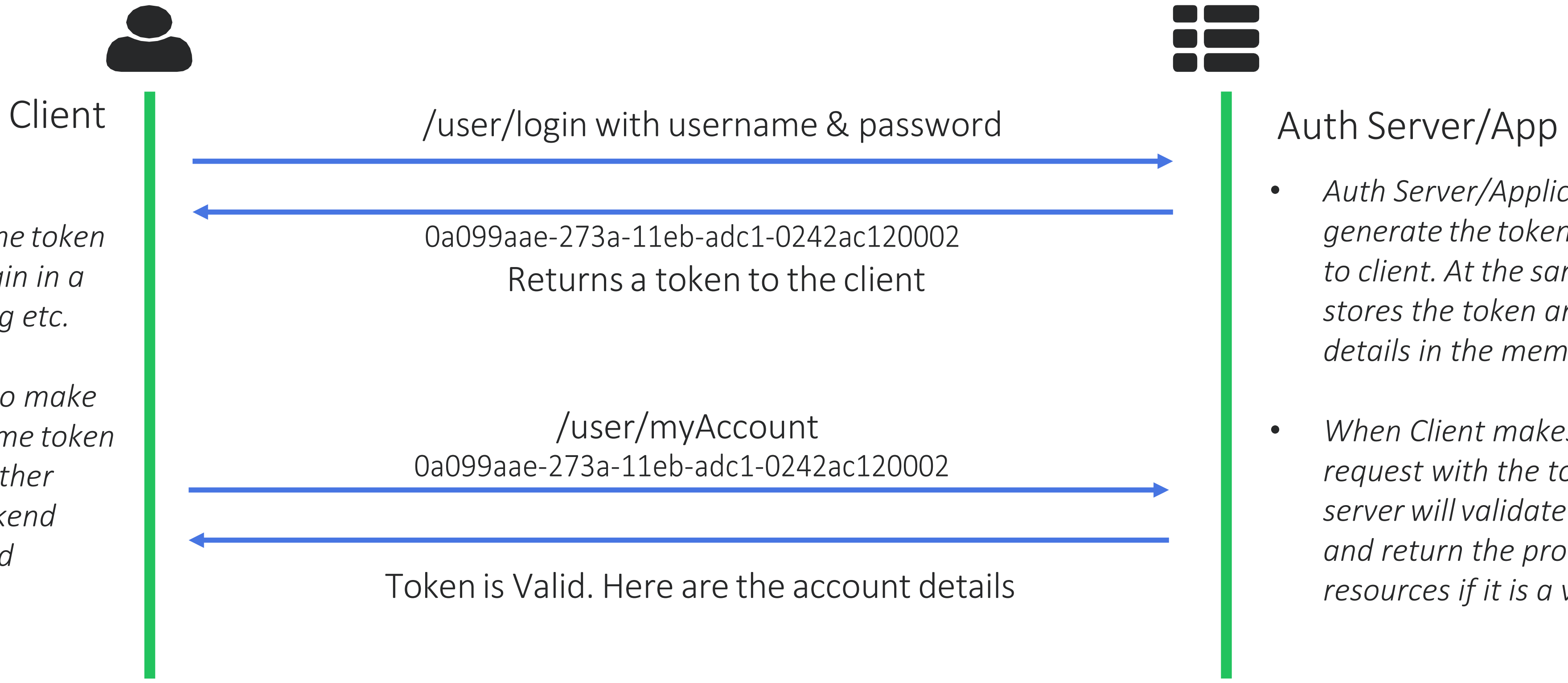
Blank

TOKENS

IN AUTHN & AUTHZ

- A Token can be a plain string of format universally unique identifier (UUID) or it can be of type JSON Web Token (JWT) usually that get generated when the user authenticated for the first time during login.

- On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it's valid, returns the secure resource to the client.


Client
/user/login with username & password
0a099aae-273a-11eb-adc1-0242ac120002
Returns a token to the client
/user/myAccount
0a099aae-273a-11eb-adc1-0242ac120002
Token is Valid. Here are the account details
Auth Server/App

- *Client will receive the token after successful login in a header/query string etc.*

- *Client system has to make sure to send the same token value on all the further request to the backend server for protected resources*

- *Auth Server/Application will generate the token and send to client. At the same time it stores the token and client details in the memory/DB.*

- *When Client makes a request with the token, the server will validate the token and return the protected resources if it is a valid.*

TOKENS

IN AUTHN & AUTHZ

Advantages of Token based Authentication

- Token helps us not to share the credentials for every request which is a security risk to make credentials send over the network frequently.
- Tokens can be invalidated during any suspicious activities without invalidating the user credentials.
- Tokens can be created with a short life span.
- Tokens can be used to store the user related information like roles/authorities etc.
- Reusability - We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user.

TOKENS

IN AUTHN & AUTHZ

Advantages of Token based Authentication

- Security - Since we are not using cookies, we don’t have to protect against cross-site request forgery (CSRF) attacks.

- Stateless, easier to scale. The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.

- We already used tokens in the previous sections in the form of CSRF and JSESSIONID tokens.

  1. CSRF Token protected our application from CSRF attacks.
  2. JSESSIONID is the default token generated by the Spring Security which helped us not to share the credentials to the backend every time.

JWT

TOKEN DETAILS

- JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.

- JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.

- JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the server side.

- A JWT token has 3 parts each separated by a dot(.). Below is a sample JWT token,

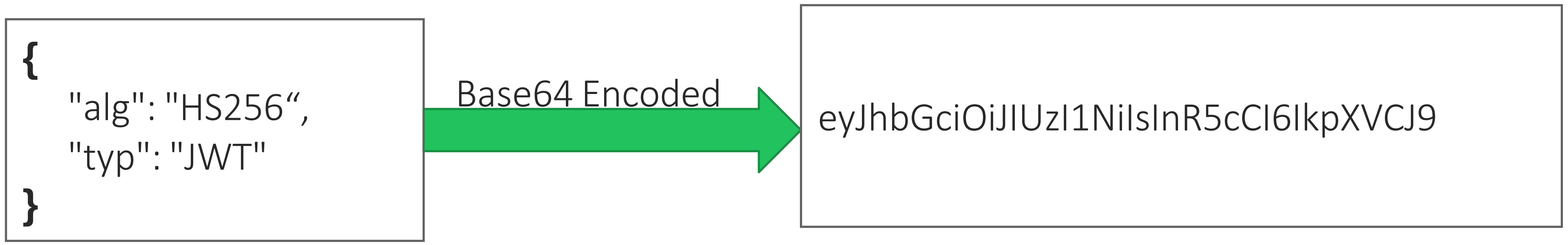
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM
5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

1. Header
2. Payload
3. Signature (Optional)

JWT

TOKEN DETAILS

- JWTs have three parts: a header, a payload, and a signature.
- In the header, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.



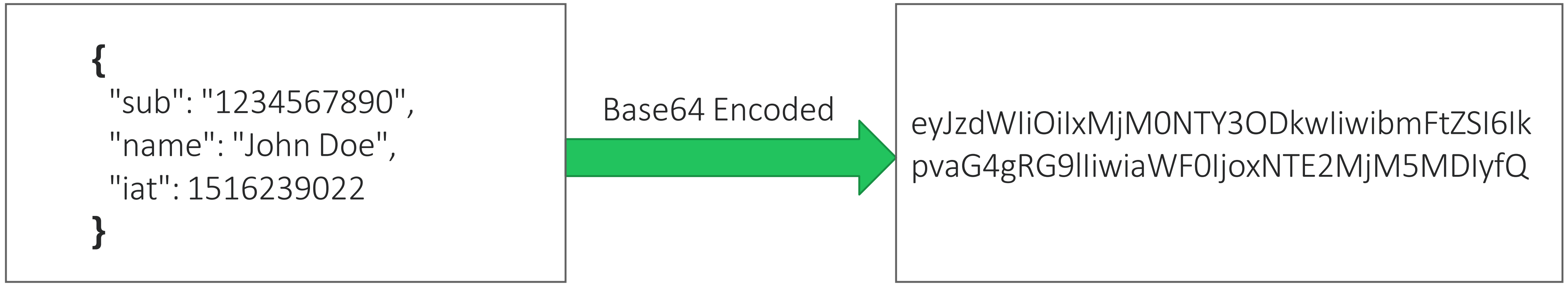
```
{
    "alg": "HS256“,
    "typ": "JWT"
}
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

JWT

TOKEN DETAILS

- In the body, we can store details related to user, roles etc. which can be used later for AuthN and AuthZ. Though there is no such limitation what we can send and how we can send in the body, but we should put our best efforts to keep it as light as possible.



```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik
pvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

JWT

TOKEN DETAILS

- The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.

- But if you are sharing this token to the client applications which will be used by all the users in the open web then we need to make sure that no one changed the header and body values like Authorities, username etc.

- To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. 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.

- 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)

- The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

JWT

TOKEN DETAILS

- Putting all together the JWT token is three Base64-URL 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.

Header

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

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

●

Body/Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNT
E2MjM5MDIyfQ

●

Signature

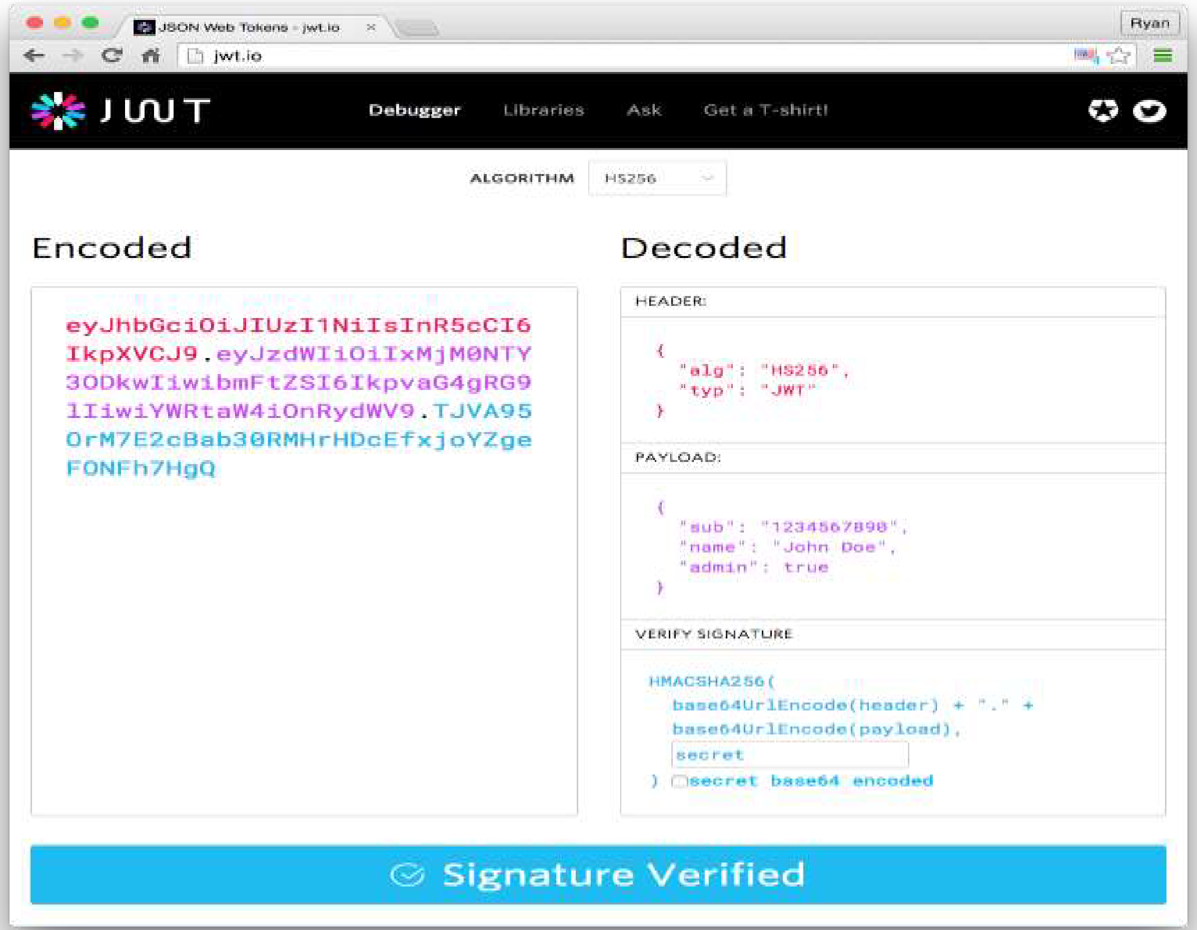
Hash that got generated based on base64 encoded values of header and body

SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

JWT

TOKEN DETAILS

- If you want to play with JWT and put these concepts into practice, you can use [jwt.io](jwt.io) Debugger to decode, verify, and generate JWTs.



ALGORITHM HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.TJVA95
OrM7E2cBab30RMHrHDcEfxjoYZge
FONFh7HgQ
```

Decoded

HEADER:

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

PAYLOAD:

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

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) ☐secret base64 encoded
```

Signature Verified

Blank

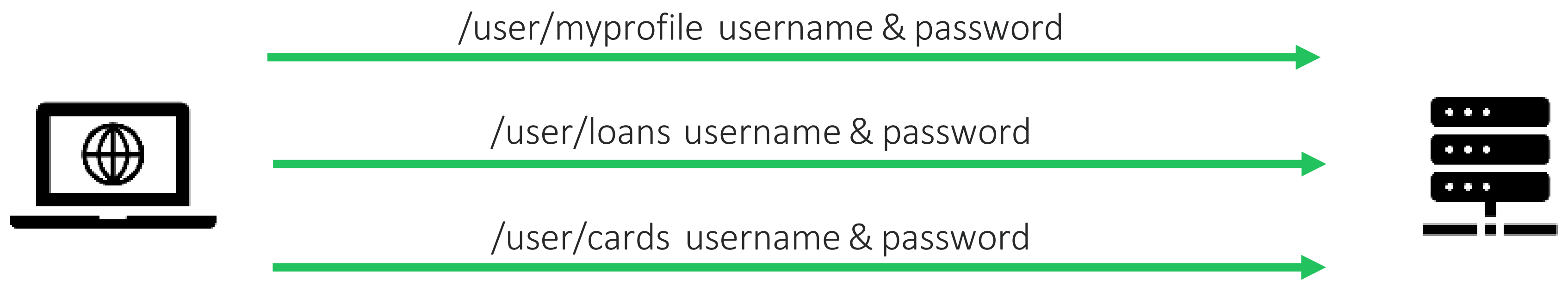
Security Challenges & OAUTH

We would like to narrate the problems faced by application without adoption of Strong level security measure.

PROBLEMS WITH OUT OAUTH2

SCENARIO 1 -HTTP BASIC AUTHENTICATION

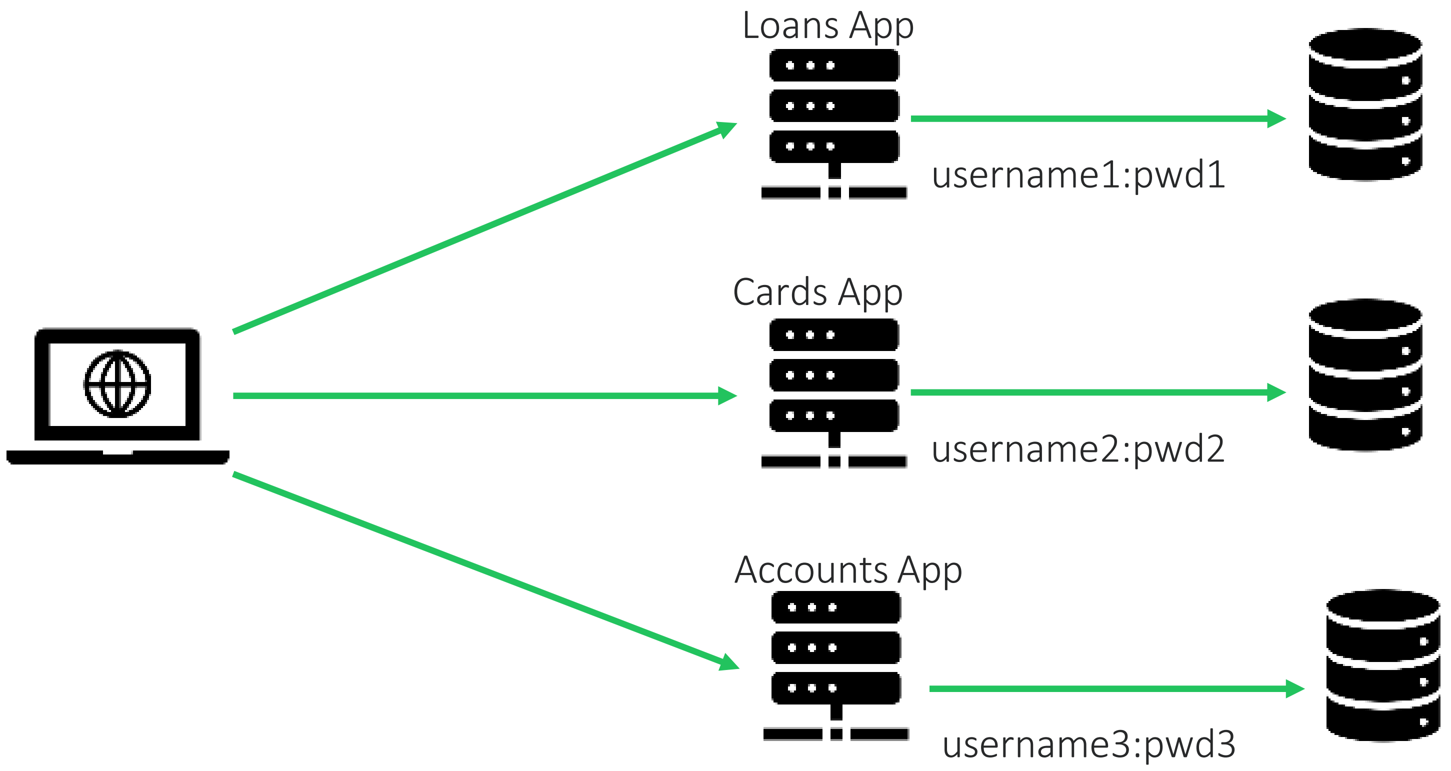
We would like to narrate the problems by application without adoption of Strong level security measure.


/user/myprofile username & password
/user/loans username & password
/user/cards username & password

With HTTP Basic authentication, the client need to send the user credentials every time and authentication logic has to be executed every time with all the requests. With this approach we ended up sharing the credentials often over the network.

PROBLEMS WITH OUT OAUTH2

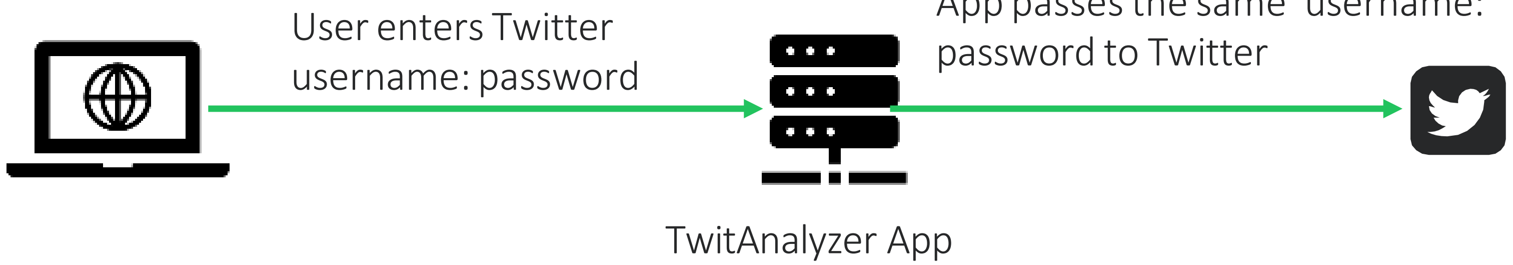
SCENARIO 2 –MULTIPE APPS INSIDE A ORGANIZATION



- The Bank system maintain separate applications for 3 departments like Loans, Cards and Accounts.
- The users has to register and maintain different credentials/same credentials but will be stored in 3 different DBs.
- Even the AuthN & AuthZ logic, security standards will be duplicated in all the 3 apps.

PROBLEMS WITH OUT OAUTH2

SCENARIO 3 –INTERACTION WITH THIRD PARTY APPS


User enters Twitter username: password
App passes the same username: password to Twitter
TwitAnalyzer App

- I want to use an application 'TwitAnalyzer' which will analyze my tweets on Twitter
- For the same, 'TwitAnalyzer' ask my credentials of Twitter during the login page
- 'TwitAnalyzer' used my credentials to interact with the Twitter on my behalf.
- But there is a serious security breach here if the app misuses user credentials.

OAUTH2

INTRODUCTION

- OAuth stands for Open Authorization. It’s a free and open protocol, built on IETF standards and licenses from the Open Web Foundation.

- OAuth 2.0 is a delegation protocol, which means letting someone who controls a resource allow a software application to access that resource on their behalf without impersonating them.

  - *For example in our EazyBank application instead of maintain both Auth and Business logic inside a same application/server, it will allow other application to handle authorization before allowing the client to access protected resources. This will happen mostly with the help of tokens.*

  - *The other example is, I have an application ‘TwitAnalyzer’ where it can analyze the tweets that somebody made on the Twitter. But in order to work we should allow this application to pull the tweets from the Twitter. So here Twitter exposes a Authorization server to the ‘TwitAnalyzer’. So this application now will have a login page where it will redirect the user to the Twitter login and his credentials will be validated by Twitter. Post that Twitter provides a token to the ‘TwitAnalyzer’ which will be used to pull the tweets of the user.*

OAUTH2

INTRODUCTION

- According to the OAuth 2.0 specification it 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.

- In the OAuth world, a client application wants to gain access to a protected resource on behalf of a resource owner (usually an end user). For this client application will interact with the Authorization server to obtain the token.

- In many ways, you can think of the OAuth token as a “access card” at any office/hotel. These tokens provides limited access to someone, without handing over full control in the form of the master key.

OAUTH2

INTRODUCTION

- OAuth 2.0 has the following components,

✓ The Resource Server – where the protected resources owned by user is present like photos, personal information, transactions etc.

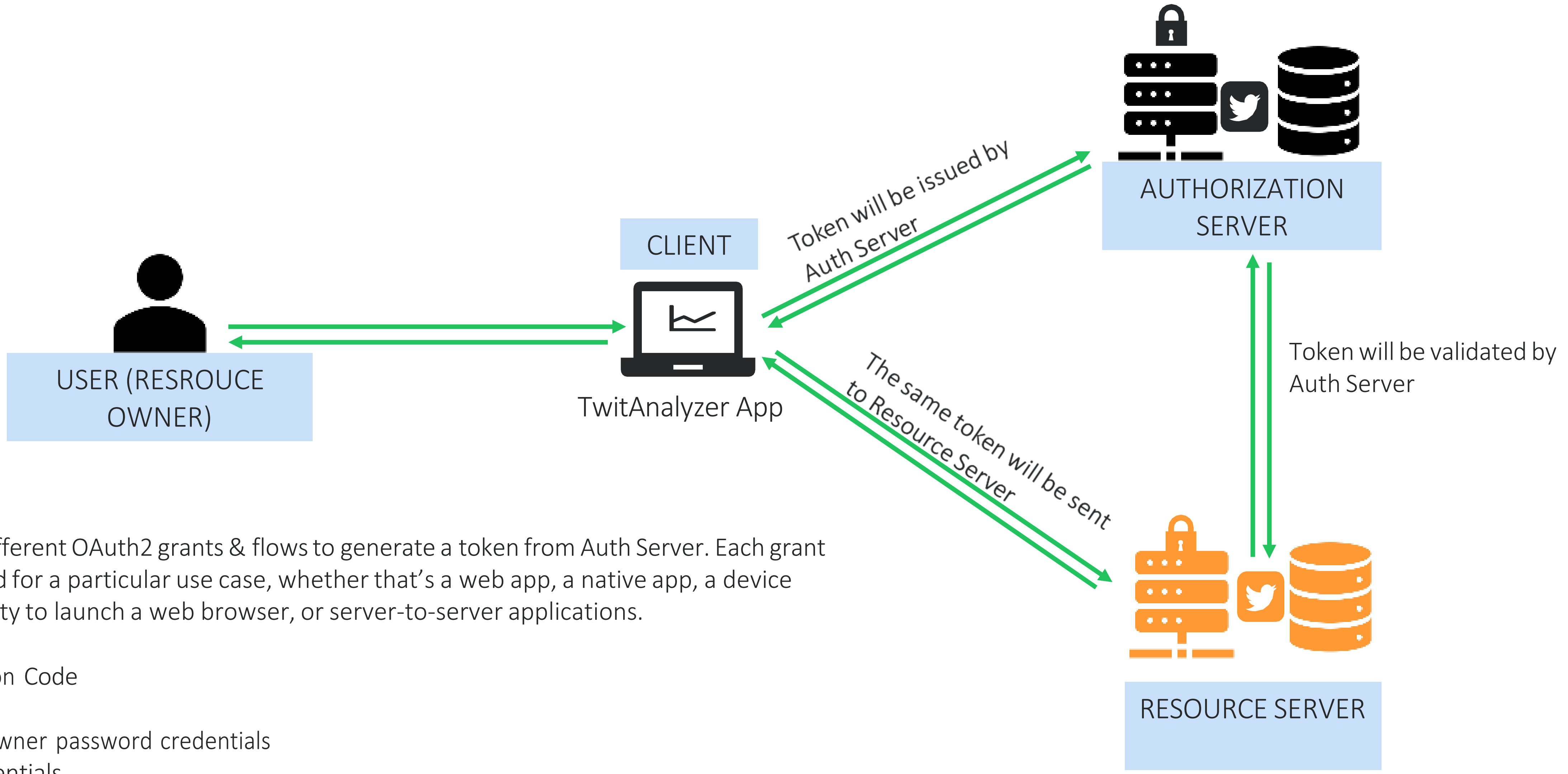
✓ The user (also known as resource owner) – The person who owns resources exposed by the resource server. Usually the user will prove his identity with the help of username and password.

✓ The client – The application that want to access the resources owned by the user on their behalf. The client uses a client id and secret to identify itself. But these are not same as user credentials.

✓ The authorization server - The server that authorizes the client to access the user resources in the resource server. When the authorization server identifies that a client is authorized to access a resource on behalf of the user, it issues a token. The client application uses this token to prove to the resource server that it was authorized by the authorization server. The resource server allows the client to access the resource it requested if it has a valid token after validating the same with Auth server.

SAMPLE OAUTH2 FLOW

IN THE TWITANALYZER APP


CLIENT
USER (RESROUCE OWNER)
TwitAnalyzer App
Token will be issued by Auth Server
AUTHORIZATION SERVER
Token will be validated by Auth Server
The same token will be sent to Resource Server
RESOURCE SERVER

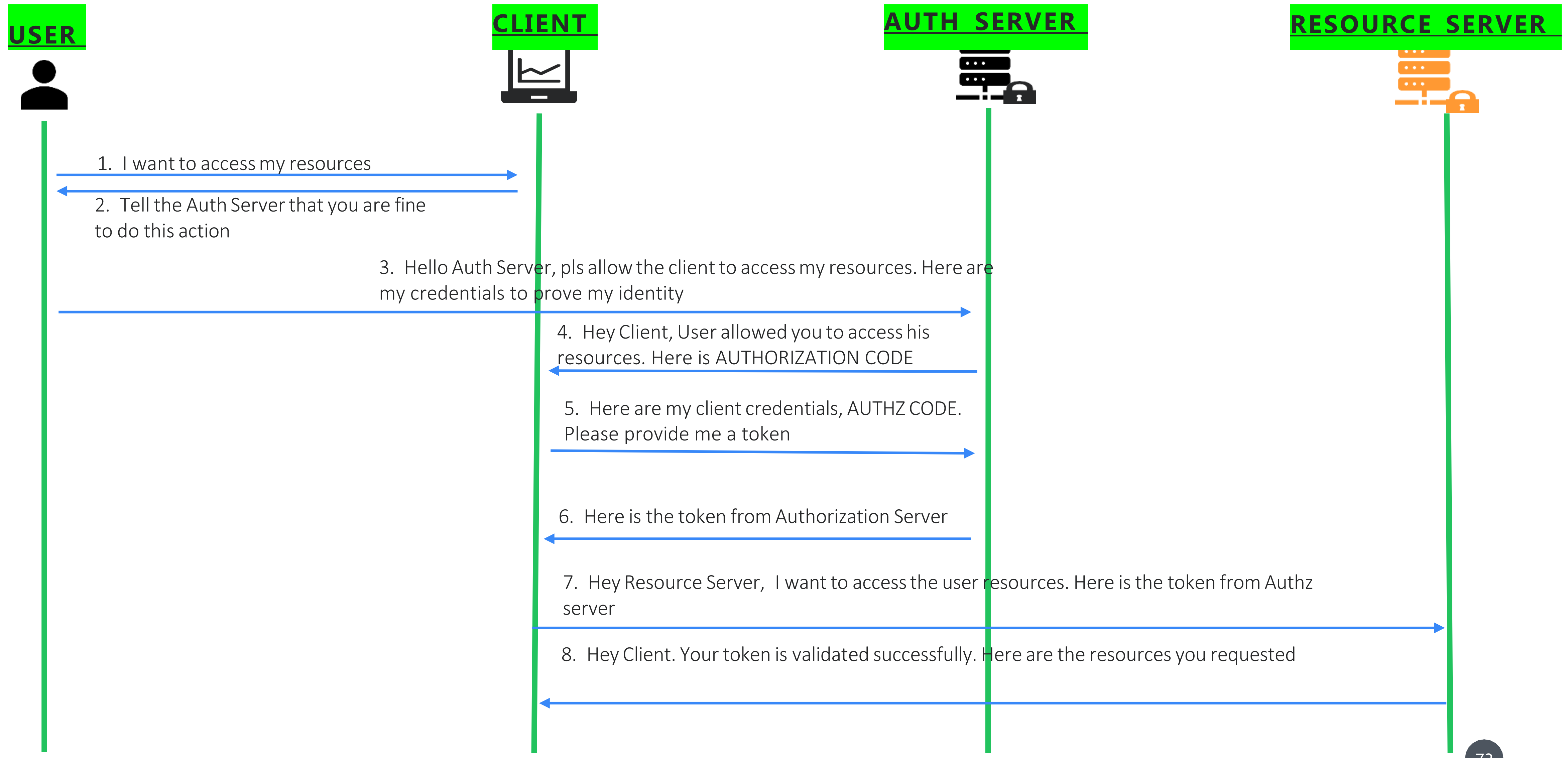
Below are the different OAuth2 grants & flows to generate a token from Auth Server. Each grant type is optimized for a particular use case, whether that's a web app, a native app, a device without the ability to launch a web browser, or server-to-server applications.

1. Authorization Code
2. Implicit
3. Resource Owner password credentials
4. Client Credentials
5. Refresh Token

\

OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE


USER
CLIENT
AUTH SERVER
RESOURCE SERVER
1. I want to access my resources
2. Tell the Auth Server that you are fine to do this action
3. Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity
4. Hey Client, User allowed you to access his resources. Here is AUTHORIZATION CODE
5. Here are my client credentials, AUTHZ CODE. Please provide me a token
6. Here is the token from Authorization Server
7. Hey Resource Server, I want to access the user resources. Here is the token from Authz server
8. Hey Client. Your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

- In the steps 2 & 3, where client is making a request to Auth Server endpoint have to send the below important details,

  - ✓ client_id – the id which identifies the client application by the Auth Server. This will be granted when the client register first time with the Auth server.
  - ✓ redirect_uri – the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration then this value is optional
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ state – CSRF token value to protect from CSRF attacks
  - ✓ response_type – With the value 'code' which indicates that we want to follow authorization code grant

- In the step 5 where client after received a authorization code from Auth server, it will again make a request to Auth server for a token with the below values,
  - ✓ code – the authorization code received from the above steps
  - ✓ client_id & client_secret – the client credentials which are registered with the auth server. Please note that these are not user credentials
  - ✓ grant_type – With the value 'authorization_code' which identifies the kind of grant type is used
  - ✓ redirect_uri

OAUTH2 FLOW

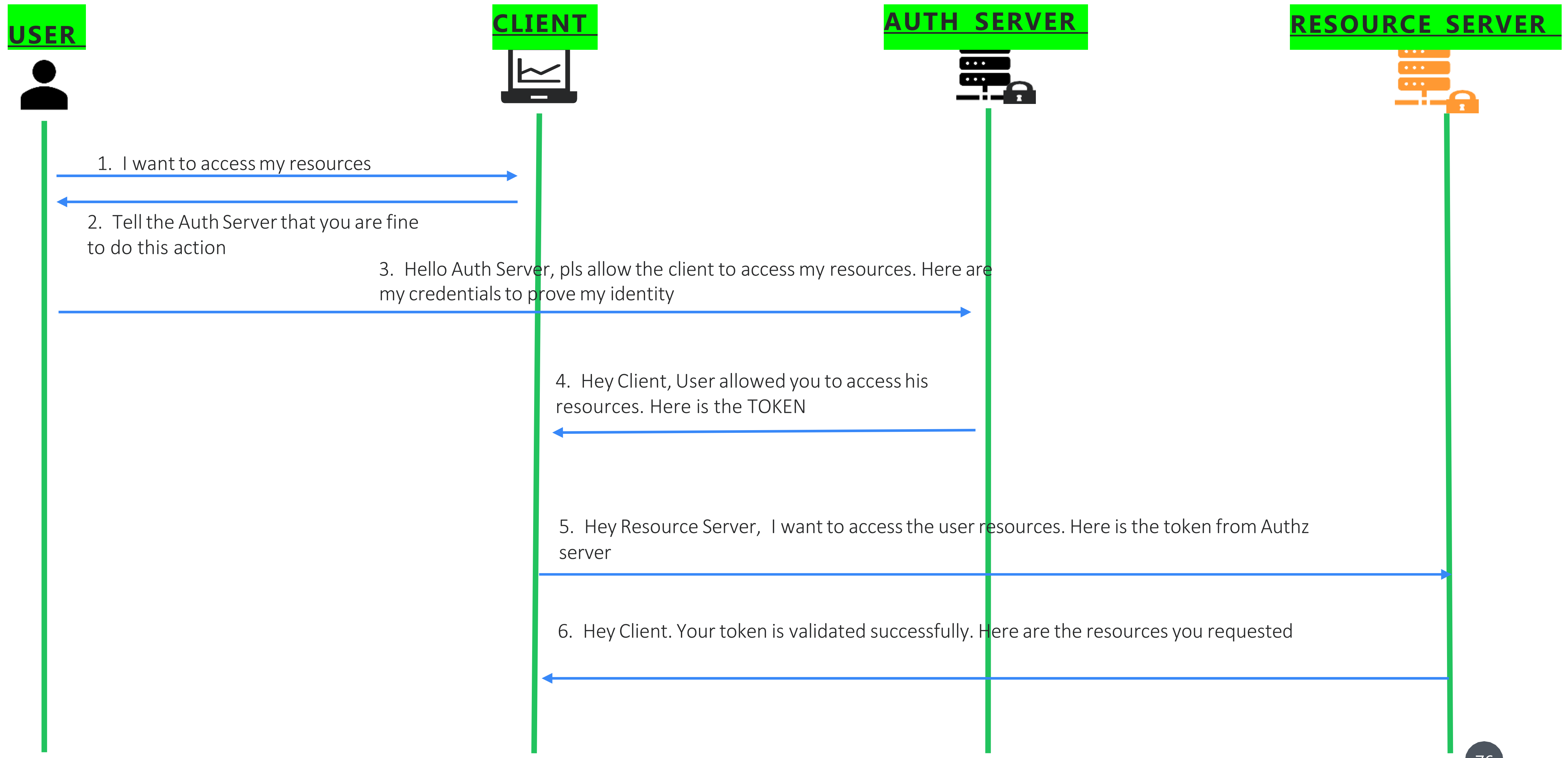
IN THE AUTHORIZATION CODE GRANT TYPE

- We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.
  - ✓ In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, auth server send the authorization code to client
  - ✓ Once it receives the authorization code, in this step client has to prove it’s identity along with the authorization code, client credentials to get the access token.

- Well you may ask why can’t Auth server directly club both the steps together and provide the token in a single step. The answer we used to have that grant type as well which is called as ‘implicit grant type’. But this grant type is not recommended to use due to it’s less secure.

OAUTH2 FLOW

IN THE IMPLICIT GRANT TYPE


USER
CLIENT
AUTH SERVER
RESOURCE SERVER
1. I want to access my resources
2. Tell the Auth Server that you are fine to do this action
3. Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity
4. Hey Client, User allowed you to access his resources. Here is the TOKEN
5. Hey Resource Server, I want to access the user resources. Here is the token from Authz server
6. Hey Client. Your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE IMPLICIT GRANT TYPE

- In the step 3, where client is making a request to Auth Server endpoint have to send the below important details,

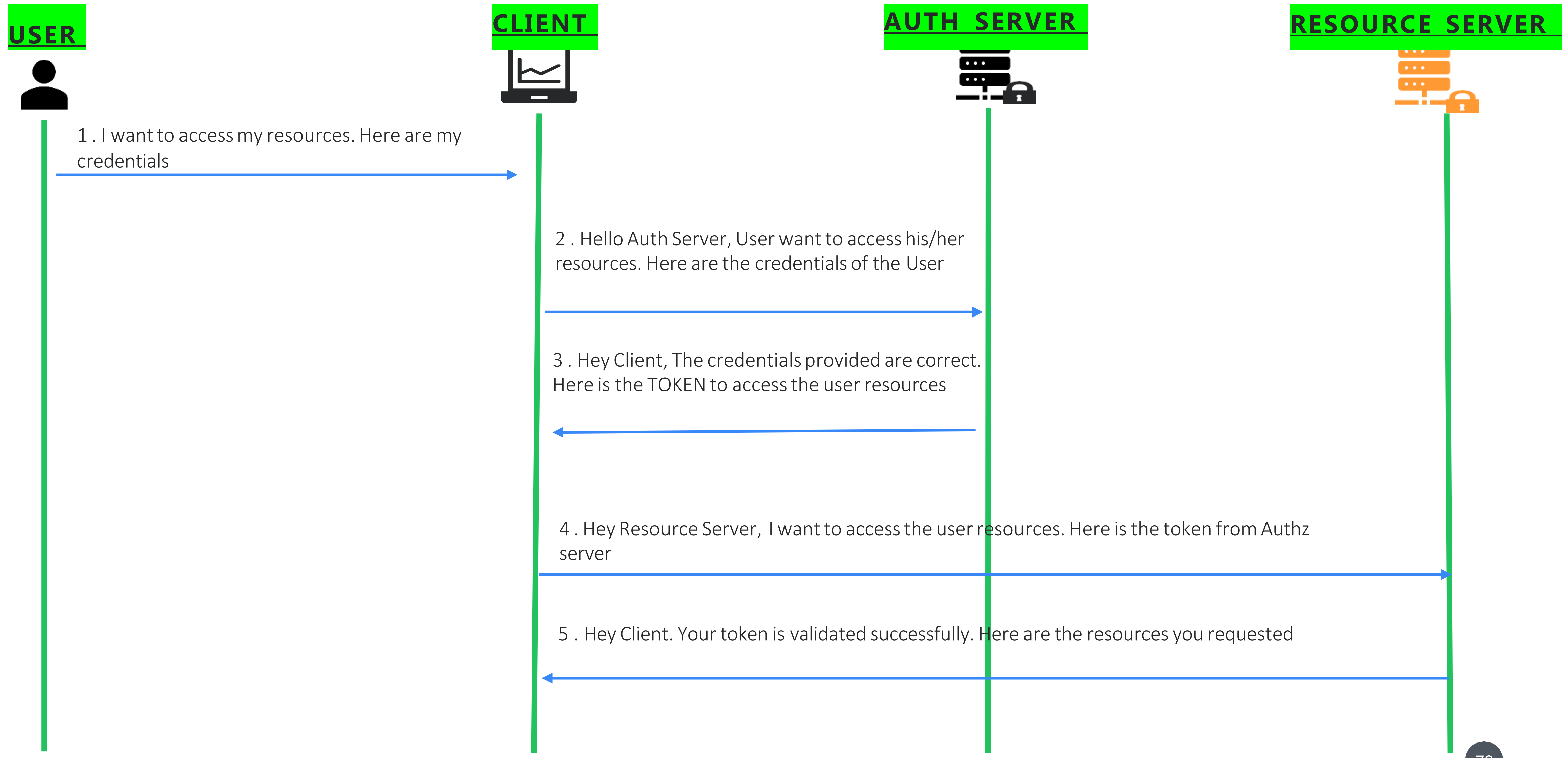
  - ✓ client_id – the id which identifies the client application by the Auth Server. This will be granted when the client register first time with the Auth server.
  - ✓ redirect_uri – the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration then this value is optional
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ state – CSRF token value to protect from CSRF attacks
  - ✓ response_type – With the value 'token' which indicates that we want to follow implicit grant type

- If the user approves the request, the authorization server will redirect the browser back to the redirect_uri specified by the application, adding a token and state to the fragment part of the URL. For example, the user will be redirected back to a URL such as

  https://example-app.com/redirect
  #access_token=xMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiY&token_type=Bearer&expires_in=600
  &state=80bvIn4pUfdSxr6UTjtay8Yzg9ZiAkCzKNwy

OAUTH2 FLOW

IN THE RESOURCE OWNER CREDENTIALS GRANT TYPE


USER
CLIENT
AUTH SERVER
RESOURCE SERVER
1 . I want to access my resources. Here are my credentials
2 . Hello Auth Server, User want to access his/her resources. Here are the credentials of the User
3 . Hey Client, The credentials provided are correct. Here is the TOKEN to access the user resources
4 . Hey Resource Server, I want to access the user resources. Here is the token from Authz server
5 . Hey Client. Your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE RESOURCE OWNER CREDENTIALS GRANT TYPE

- In the step 2, where client is making a request to Auth Server endpoint have to send the below important details,

  - ✓ client_id & client_secret – the credentials of the client to authenticate itself.
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ username & password – Credentials provided by the user in the login flow
  - ✓ grant_type – With the value 'password' which indicates that we want to follow password grant type

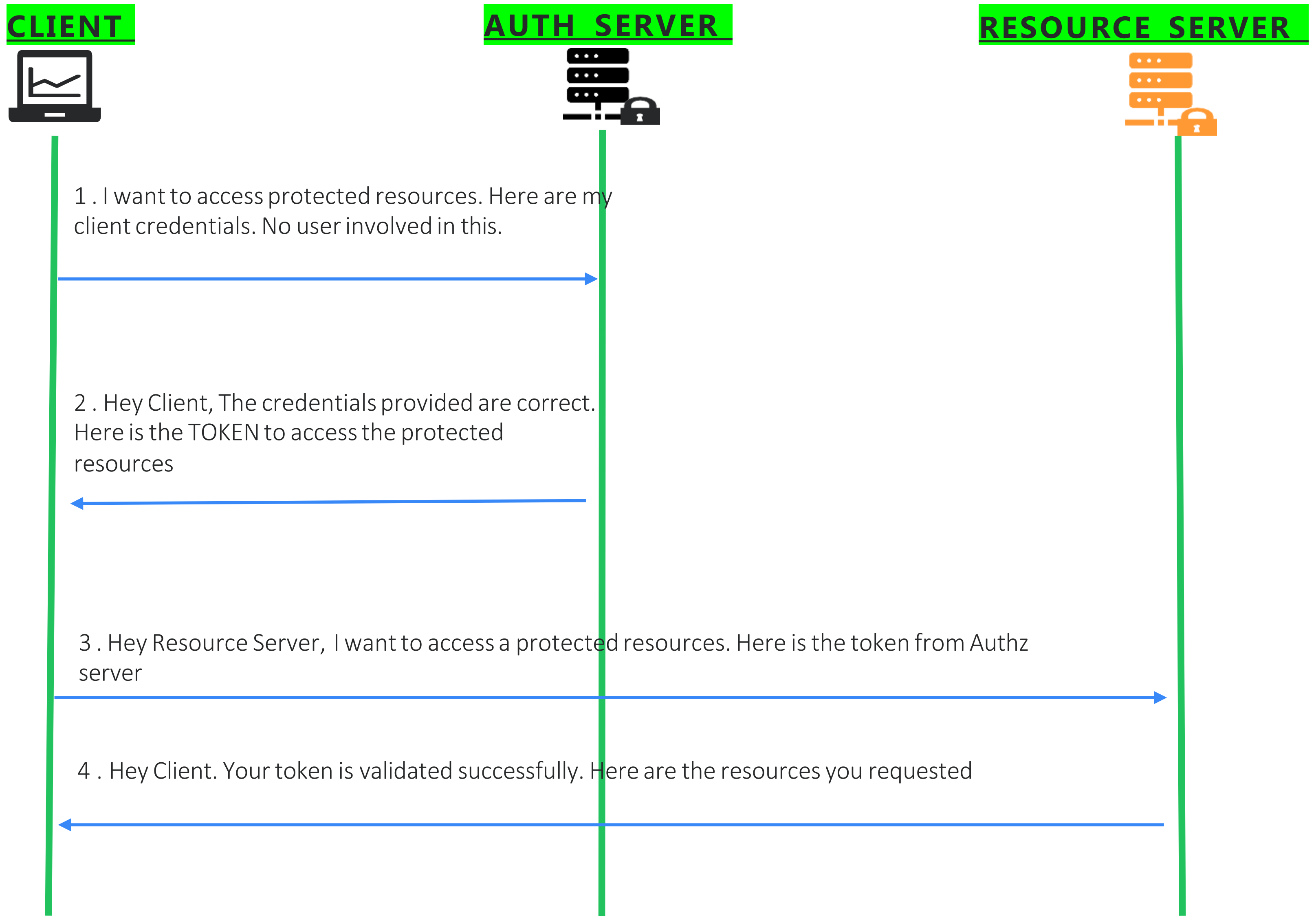
- We use this authentication flow only if the client, authorization server and resource servers are maintained by the same organization.

- This flow will be usually followed by the enterprise applications who want to separate the Auth flow and business flow. Once the Auth flow is separated different applications in the same organization can leverage it.

- We can't use the Authorization code grant type since it won't look nice for the user to redirect multiple pages inside your organization for authentication.

OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE


CLIENT
AUTH SERVER
RESOURCE SERVER
1 . I want to access protected resources. Here are my client credentials. No user involved in this.
2 . Hey Client, The credentials provided are correct. Here is the TOKEN to access the protected resources
3 . Hey Resource Server, I want to access a protected resources. Here is the token from Authz server
4 . Hey Client. Your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE

- In the step 1, where client is making a request to Auth Server endpoint have to send the below important details,

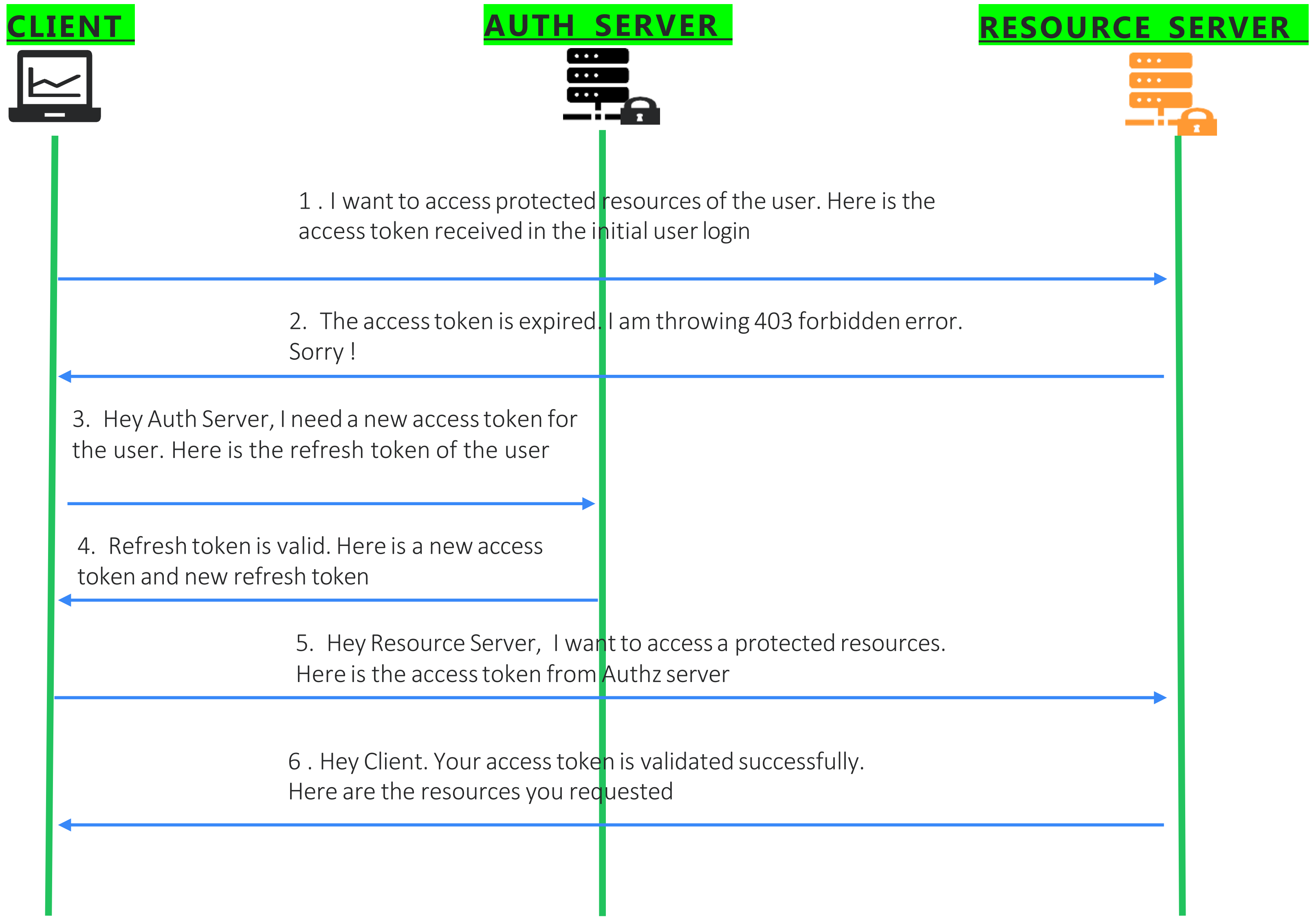
  - ✓ client_id & client_secret – the credentials of the client to authenticate itself.
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ grant_type – With the value 'client_credentials' which indicates that we want to follow client credentials grant type

- This is the most simplest grant type flow in OAUTH2.

- We use this authentication flow only if there is no user and UI involved. Like in the scenarios where 2 different applications want to share data between them using backend APIs.

OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE



1 . I want to access protected resources of the user. Here is the access token received in the initial user login

2. The access token is expired. I am throwing 403 forbidden error. Sorry !

3. Hey Auth Server, I need a new access token for the user. Here is the refresh token of the user

4. Refresh token is valid. Here is a new access token and new refresh token

5. Hey Resource Server, I want to access a protected resources. Here is the access token from Authz server

6 . Hey Client. Your access token is validated successfully. Here are the resources you requested

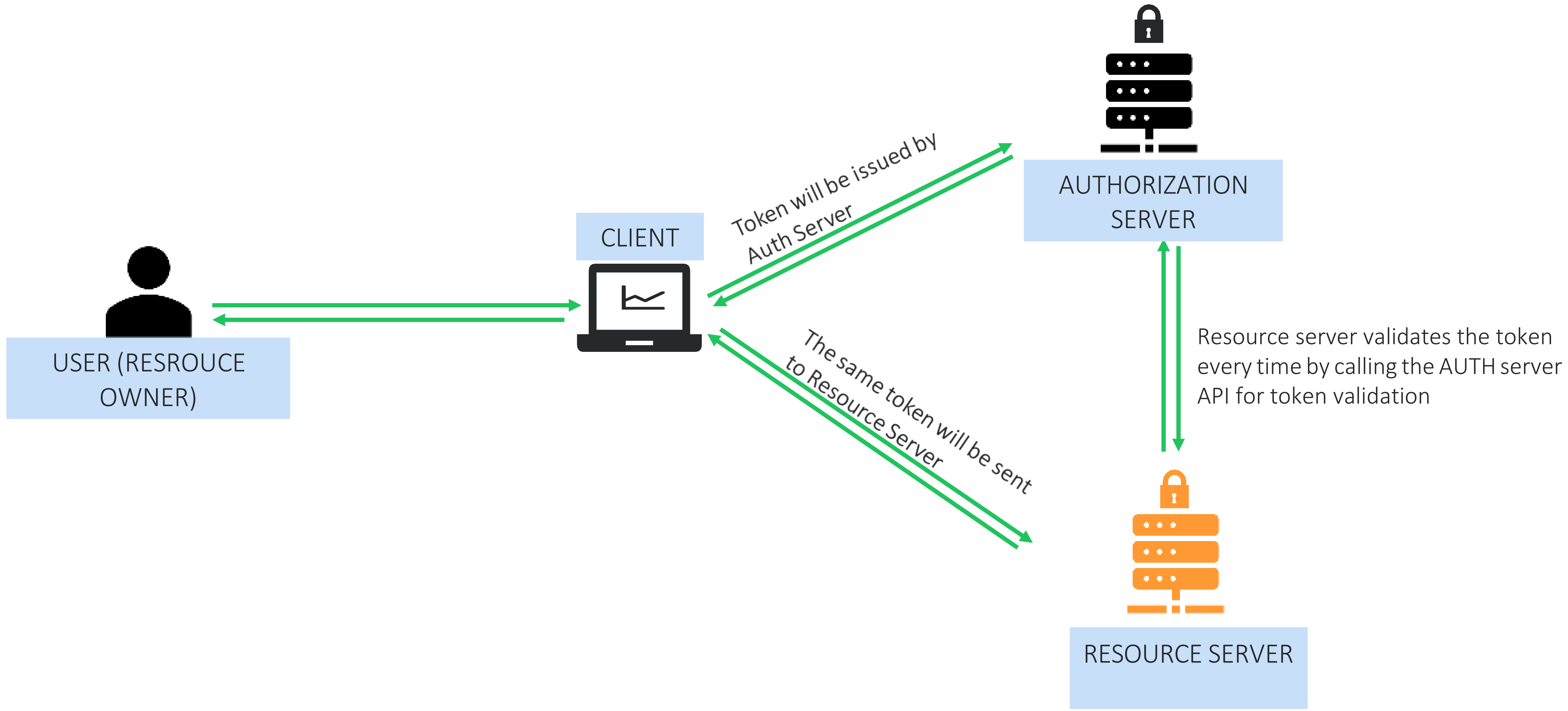
OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE

- In the step 3, where client is making a request to Auth Server endpoint have to send the below important details,
  - ✓ client_id & client_secret – the credentials of the client to authenticate itself.
  - ✓ refresh_token – the value of the refresh token received initially
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ grant_type – With the value 'refresh_token' which indicates that we want to follow refresh token grant type
- This flow will be used in the scenarios where the access token of the user is expired. Instead of asking the user to login again and again, we can use the refresh token which originally provided by the Authz server to reauthenticate the user.
- Though we can make our access tokens to never expire but it is not recommended considering scenarios where the tokens can be stole if we always use the same token
- Even in the resource owner credentials grant types we should not store the user credentials for reauthentication purpose instead we should reply on the refresh tokens.

RESOURCE SERVER TOKEN VALIDATION

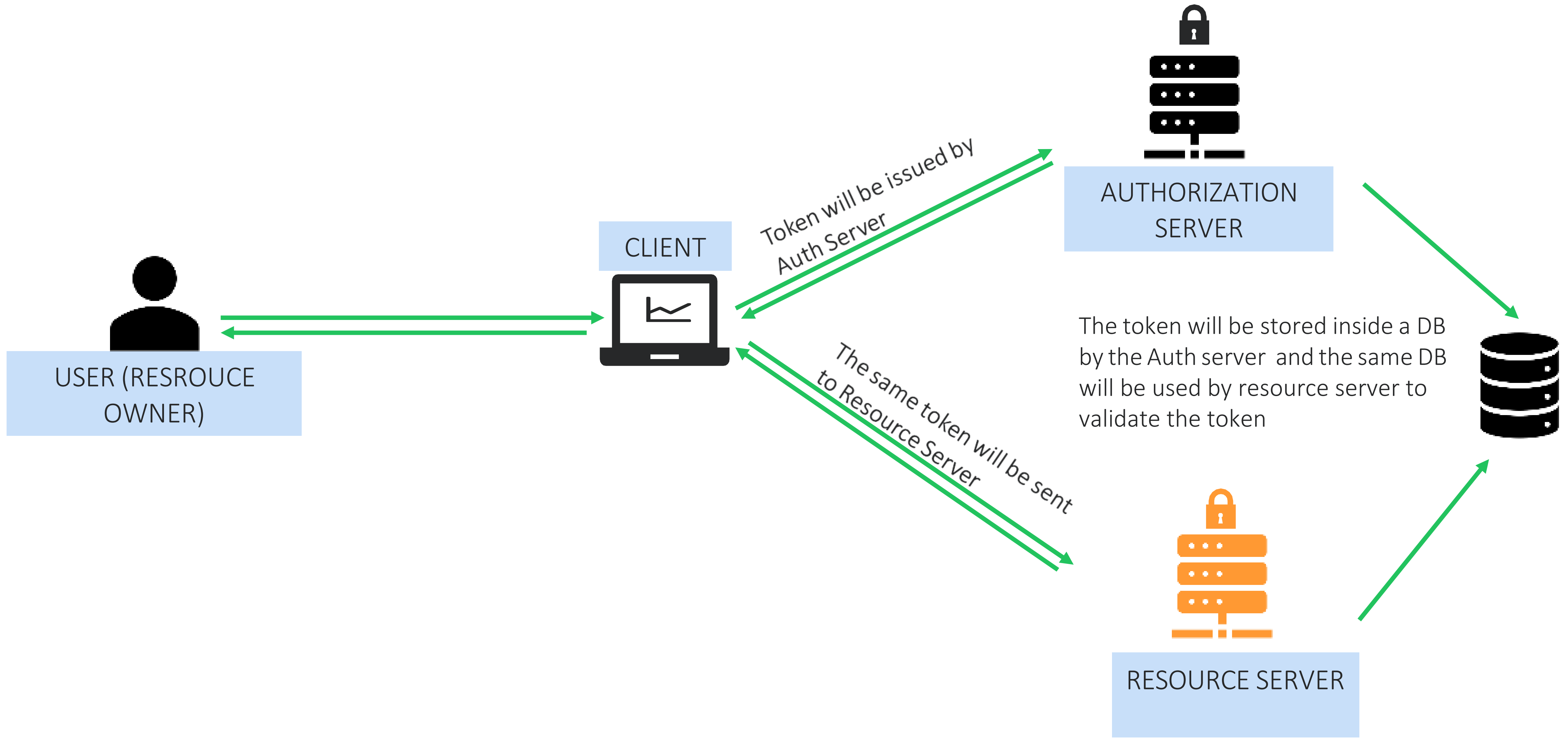
IN THE OAUTH2 FLOW USING DIRECT API CALL



AUTHORIZATION SERVER
CLIENT
Token will be issued by Auth Server
USER (RESROUCE OWNER)
Resource server validates the token every time by calling the AUTH server API for token validation
The same token will be sent to Resource Server
RESOURCE SERVER

RESOURCE SERVER TOKEN VALIDATION

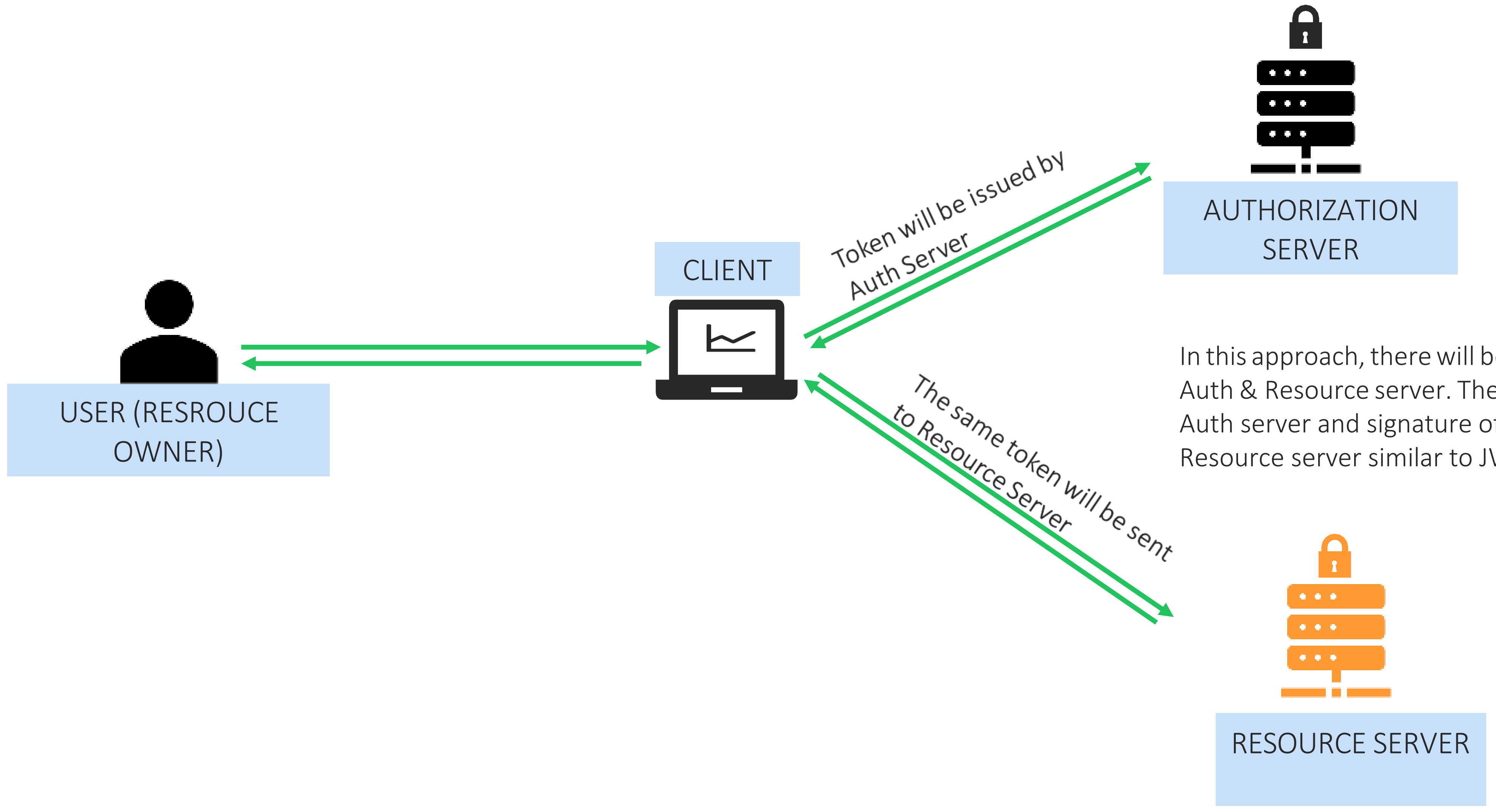
IN THE OAUTH2 FLOW USING COMMON DB


AUTHORIZATION SERVER
CLIENT
Token will be issued by Auth Server
USER (RESROUCE OWNER)
The same token will be sent to Resource Server
RESOURCE SERVER

The token will be stored inside a DB by the Auth server and the same DB will be used by resource server to validate the token

RESOURCE SERVER TOKEN VALIDATION

IN THE OAUTH2 FLOW USING TOKEN SIGNATURE


CLIENT
Token will be issued by Auth Server
AUTHORIZATION SERVER
USER (RESROUCE OWNER)
The same token will be sent to Resource Server
RESOURCE SERVER

In this approach, there will be no interaction b/w the Auth & Resource server. The token will be signed by Auth server and signature of it will be verified by Resource server similar to JWT tokens.

SUMMARY

OAUTH2

1. OAUTH2 and what kind of problems it solves in the AuthN & AuthZ flow

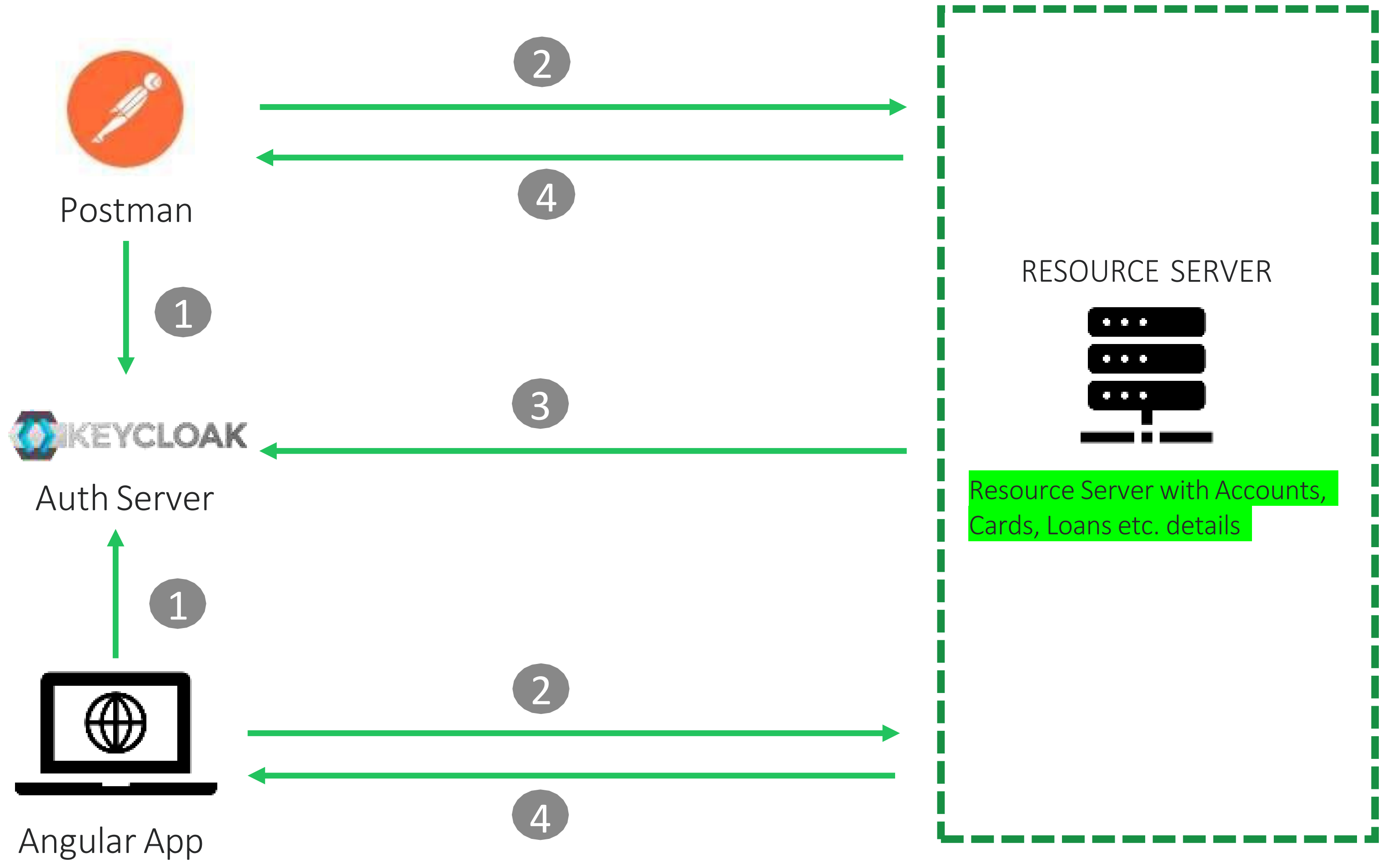
2. Different components in the OAUTH2 flow like
   - *Authorization Server*
   - *Client*
   - *User/Resource Owner*
   - *Resource server*

3. Different OAUTH2 grants and flows using the below approaches
   - *Authorization Code*
   - *Implicit*
   - *Resource Owner password credentials*
   - *Client Credentials*
   - *Refresh Token*

4. How Resource server validate the token issued by Authorization server in the OAUTH2 flow.

IMPLEMENT OAUTH2 INSIDE AN APP

USING KEYCLOAK AUTH SERVER



1. We may have either Angular like Client App or REST API clients to get the resource details from resource server. In both kinds we need to get access token from Auth Servers like KeyCloak.

2. Once the access token received from Auth Server, client Apps will connect with Resource server along with the access token to get the details around Accounts, Cards, Loans etc.

3. Resource server will connect with Auth Server to know the validity of the access token.

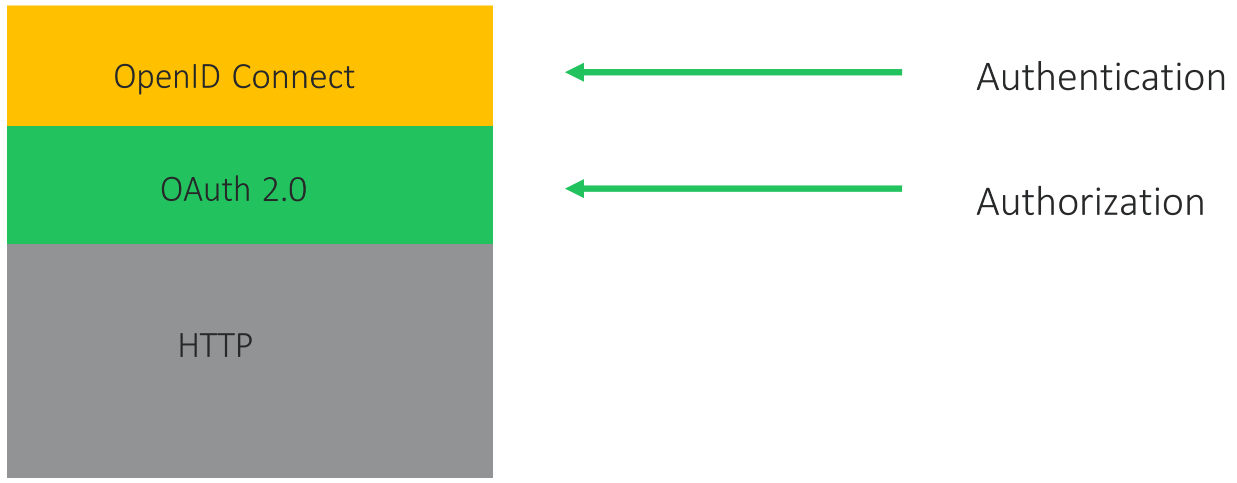
4. If the access token is valid, Resource server will respond with the details to client Apps.

OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

What is OpenID Connect?

- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of scopes and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.

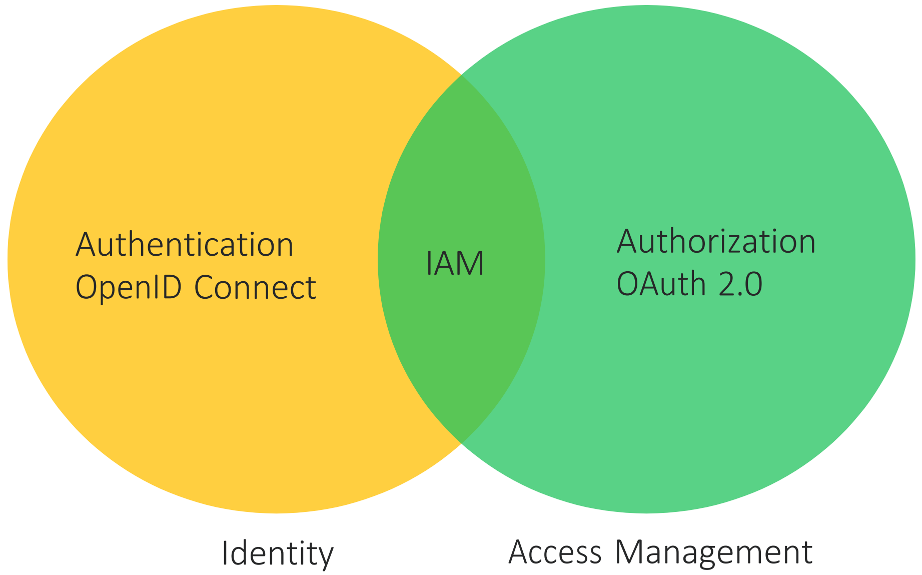


OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

Why is OpenID Connect important?

- Identity is the key to any application. At the core of modern authorization is OAuth 2.0, but OAuth 2.0 lacks an authentication component. Implementing OpenID Connect on top of OAuth 2.0 completes an IAM (Identity & Access Management) strategy.
- As more and more applications need to connect with each other and more identities are being populated on the internet, the demand to be able to share these identities is also increased. With OpenID connect, applications can share the identities easily and standard way.



OpenID Connect add below details to OAuth 2.0

1. OIDC standardizes the scopes to openid, profile, email, and address.
2. ID Token using JWT standard
3. OIDC exposes the standardized "/userinfo" endpoint.

OAUTH 2.0 AUTHORIZATION CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

- When public clients (e.g., native and single-page applications) request Access Tokens, some additional security concerns are posed that are not mitigated by the Authorization Code Flow alone. This is because public clients cannot securely store a Client Secret.

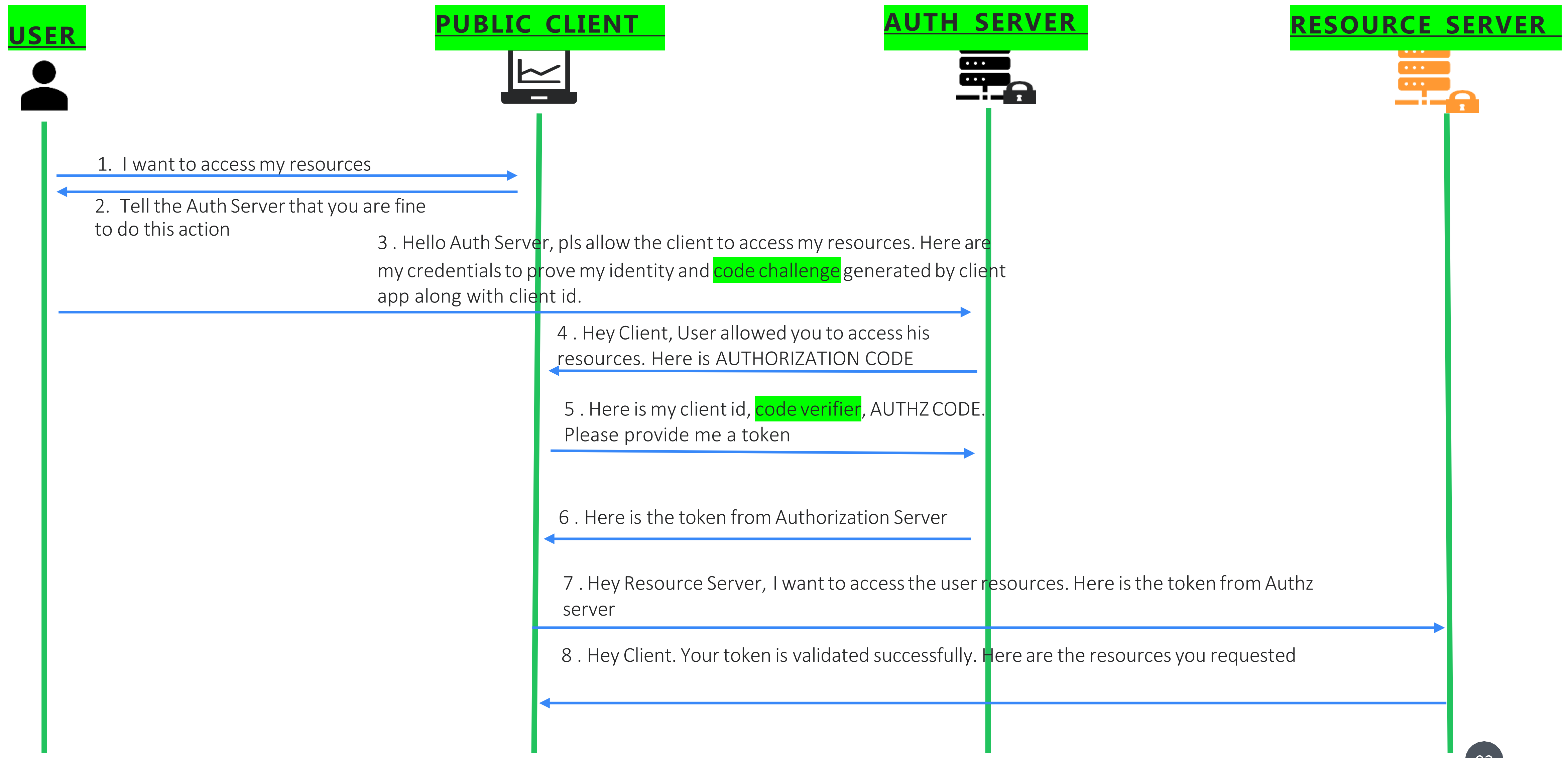
- Given these situations, OAuth 2.0 provides a version of the Authorization Code Flow for public client applications which makes use of a Proof Key for Code Exchange (PKCE).

- The PKCE-enhanced Authorization Code Flow follows below steps,

  ✓ Once user clicks login, client app creates a cryptographically-random code_verifier and from this generates a code_challenge.
  ✓ Redirects the user to the Authorization Server along with the code_challenge.
  ✓ Authorization Server stores the code_challenge and redirects the user back to the application with an authorization code, which is good for one use.
  ✓ Client App sends the authorization code and the code_verifier(created in step 1) to the Authorization Server.
  ✓ Authorization Server verifies the code_challenge and code_verifier. If they are valid it respond with ID Token and Access Token (and optionally, a Refresh Token).

OAUTH2 FLOW

IN THE AUTH CODE FLOW + PKCE


USER
PUBLIC CLIENT
AUTH SERVER
RESOURCE SERVER
1. I want to access my resources
2. Tell the Auth Server that you are fine to do this action
3 . Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity and code challenge generated by client app along with client id.
4 . Hey Client, User allowed you to access his resources. Here is AUTHORIZATION CODE
5 . Here is my client id, code verifier, AUTHZ CODE. Please provide me a token
6 . Here is the token from Authorization Server
7 . Hey Resource Server, I want to access the user resources. Here is the token from Authz server
8 . Hey Client. Your token is validated successfully. Here are the resources you requested

Summary

1. OAUTH2 implementation inside web applications

    - *OpenID connect*
    - *KeyCloak*
    - *KeyCloak setup, realms creation, client creation, users creation and roles inside it*

2. Resource server that exposes secured APIs by integrating with Keycloak

3. OAUTH2 grant types inside an app,
    - *Client Credentials Grant type*
    - *Authorization code Grant type*
    - *Authorization code Grant type with PKCE*

Next Steps

Our next course of action will be to work and deliver comprehensive Conceptual Solution Architecture document covering detailed use cases for One Authentication Framework.

Case 1 :

- Build a user-based authentication framework using ID & Password
- Enable Oauth2 Authorization

Case 2 :

- Build a Social Login authentication framework.
- Enable Oauth2 Authorization & integrate with Open ID connect

Case 3 :

- Build a OAUTH2 authentication framework.
- Enable Security from a client application using Token and Other Security Principal

More set of use cased to be identified and corresponding solution framework

Appendix – A : Provider Solution for OAUTH2.0 – Auth0

- Provide an implementation approach to enable Authorization/Authentication using Auth0 Provider
- Illustrate the dependencies and cost matrix for implementation support
- Create a Step-by-Step guide to configure Auth0 implementation for application security

Appendix – B - OAUTH2.0 Detailed implementation Guide

Prepare OAUTH2.0 detailed implementation guide to enable One Authentication/Authorization framework

- **OAuth 2.0 - Fundamentals**
- **Spring Security & OAuth 2.0: Overview**
- **Spring Security & OAuth 2.0 Authorization Servers**
- **Spring Security & OAuth 2.0 Resource Servers**
- **Spring Security & OAuth 2.0 Clients**

  - OAuth 2.0
  - OAuth 2 Authorization Flows
  - The New OAuth 2.0 stack in Spring Security 5
  - Use OAuth 2.0 in Spring Boot Applications
  - Configure OAuth 2.0 Resource Server
  - Keycloak Identity and Access Management Solution
  - Resource Servers behind API Gateway
  - New Spring Authorization Server
  - OAuth 2.0 in MVC Web App
  - OAuth 2 - Social Login
  - OAuth2 + PKCE in JavaScript Application
  - Register Resource Servers with Eureka Service Registry

Appendix – C - Method level Security

METHOD LEVEL SECURITY

IN SPRING SECURITY

- As of now we have applied authorization rules on the API paths/URLs using spring security but method level security allows to apply the authorization rules at any layer of an application like in service layer or repository layer etc. Method level security can be enabled using the annotation @EnableGlobalMethodSecurity on the configuration class.

- Method level security will also helps authorization rules even in the non-web applications where we will not have any endpoints.

- Method level security provides the below approaches to apply the authorization rules and executing your business logic,

  ✓ Invocation authorization – Validates if someone can invoke a method or not based on their roles/authorities.

  ✓ Filtering authorization – Validates what a method can receive through its parameters and what the invoker

METHOD LEVEL SECURITY

IN SPRING SECURITY

can receive back from the method post business logic execution.

METHOD LEVEL SECURITY

IN SPRING SECURITY

- Spring security will use the aspects from the AOP module and have the interceptors in between the method invocation to apply the authorization rules configured.

- Method level security offers below 3 different styles for configuring the authorization rules on top of the methods,

  ✓ The prePostEnabled property enables Spring Security @PreAuthorize & @PostAuthorize annotations
  ✓ The securedEnabled property enables @Secured annotation
  ✓ The jsr250Enabled property enables @RoleAllowed annotation

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true,  securedEnabled = true,  jsr250Enabled = true)
public class MethodSecurityConfig {
                ...
}
```

- @Secured and @RoleAllowed are less powerful compared to @PreAuthorize and @PostAuthorize

METHOD LEVEL SECURITY

USING INVOCATION AUTHORIZATION IN SPRING SECURITY

- Using invocation authorization we can decide if a user is authorized to invoke a method before the method executes (preauthorization) or after the method execution is completed (postauthorization).

- For filtering the parameters before calling the method we can use Prefiltering,

```
@Service
public class LoanService {

        @PreAuthorize("hasAuthority(‘admin’)”)
        @PreAuthorize("hasRole(‘admin’)”)
        @PreAuthorize("hasAnyRole(‘admin’)”)
        @PreAuthorize("# username == authentication.principal.username")
        @PreAuthorize("hasPermission(returnObject,  'admin')")
        public Loan getLoanDetails(String username) {
                return loanRepository.loadLoanByUserName(username);
        }
}
```

METHOD LEVEL SECURITY

USING INVOCATION AUTHORIZATION IN SPRING SECURITY

- For applying postauthorization rules below is the sample configuration,

```
@Service
public class LoanService {

            @PostAuthorize ("returnObject.username == authentication.principal.username”)
            @PostAuthorize("hasPermission(returnObject, 'admin')")
            public Loan getLoanDetails(String username) {
                return loanRepository.loadLoanByUserName(username);
            }
}
```

- When implementing complex authorization logic, we can separate the logic using a separate class that implements PermissionEvaluator and overwrite the method hasPermission() inside it which can be leveraged inside the hasPermission configurations.

METHOD LEVEL SECURITY

USING FILTERING AUTHORIZATION IN SPRING SECURITY

- If we have a scenario where we don't want to control the invocation of the method but we want to make sure that the parameters sent and received to/from the method need to follow authorization rules, then we can consider filtering.

- For filtering the parameters before calling the method we can use Prefiltering,

```
@Service
public class LoanService {

        @PreFilter("filterObject.username == authentication.principal.username")
        public Loan updateLoanDetails(Loan loan) {
                // business logic
                return loan;
        }
}
```

METHOD LEVEL SECURITY

USING FILTERING AUTHORIZATION IN SPRING SECURITY

- For filtering the parameters after executing the method we can use Postfiltering,

```
@Service
public class LoanService {

        @PostFilter("filterObject.username == authentication.principal.username")
        public Loan getLoanDetails() {
                // business logic
                return loans;
        }
}
```

- We can use the @PostFilter on the Spring Data repository methods as well to filter any unwanted data coming from the database.