eazy bytes

Zero To Master

Spring Security along with JWT, OAUTH2

COURSE AGENDA

Welcome to the world of Spring Security

Securing a web app using Spring Security

Important Interfaces, Classes, Annotations of Spring Security

Configuring Authentication & Authorization for a web app

COURSE AGENDA

Implementing role based access using ROLES, AUTHORITIES

Different strategies that Spring security provides when coming to passwords

Method level security using Spring Security

How to handle most common attacks like CORS, CSRF with Spring Security

COURSE AGENDA

Deep dive on JWT & it's role in Authentication & Authorization

Deep dive on OAUTH2, OpenID & securing a web application using the same

Exploring Authorization servers available like Keycloak, Spring Authorization Server

Important topics of Security like Hashing, Tokens & many more

Most Common Security Questions?

eazy bytes

SECURITY

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

PASSWORDS

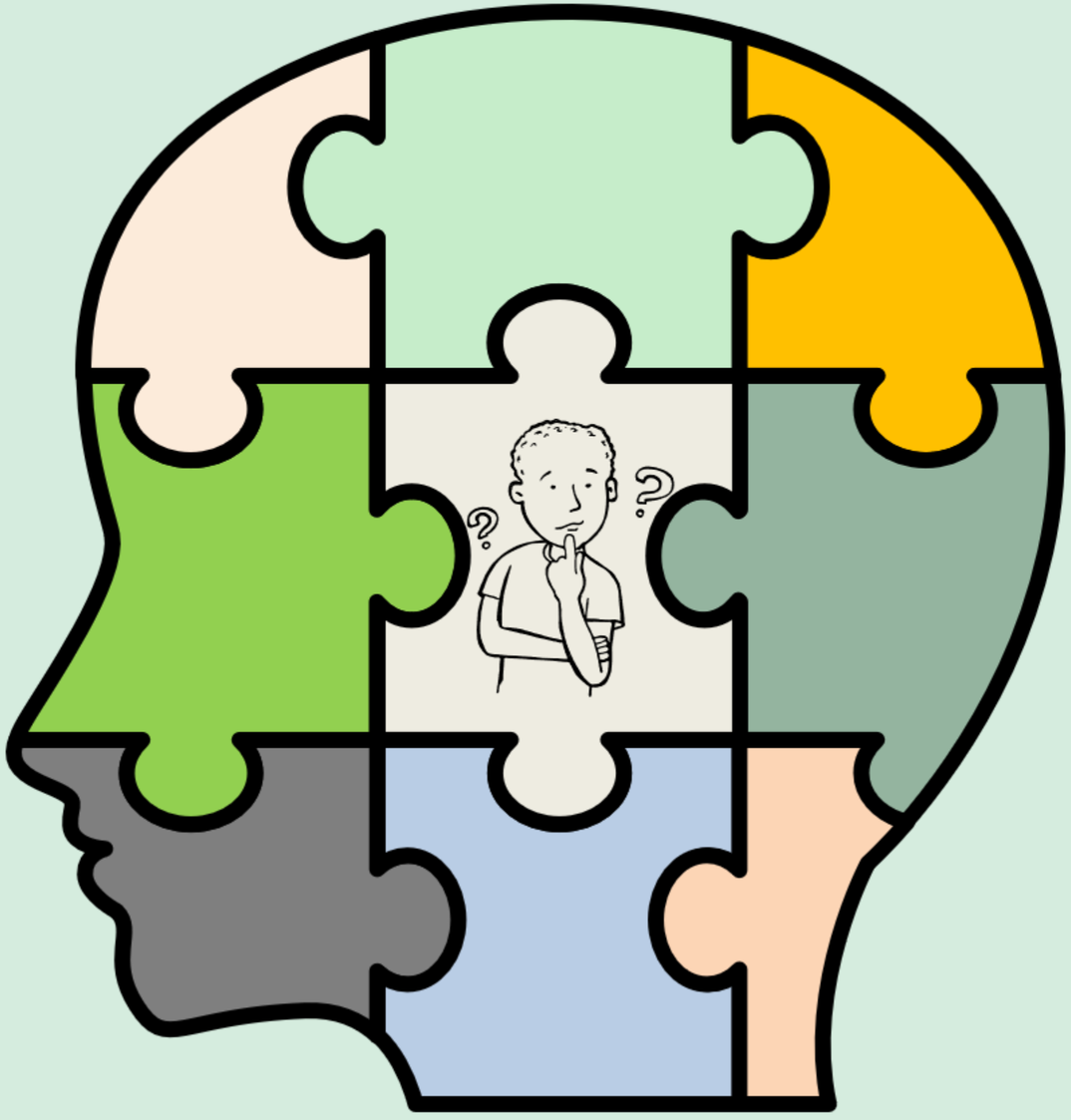
How to store passwords, validate them, encode, decode them using industry standard hashing algorithms ?

AUTHENTICATION

How should users authenticate into the application and what are the different methods available?

AUTHORIZATION

How can role-based or attribute-based access control be implemented?



METHOD LEVEL SECURITY

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

CSRF & CORS

What is CSRF attacks and CORS restrictions. How to handle them inside web applications ?

JWT, OAUTH2, OpenID

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

PREVENTING ATTACKS

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

Project Roadmap

eazy bytes

MILESTONE 1

Firstly we will create a simple spring security project and enable basic security in it

MILESTONE 2

Authentication using login screen will be implemented along with the URLs security, CSRF and CORS protection

MILESTONE 3

Authorization will be implemented using ROLES, AUTHORITIES

MILESTONE 4

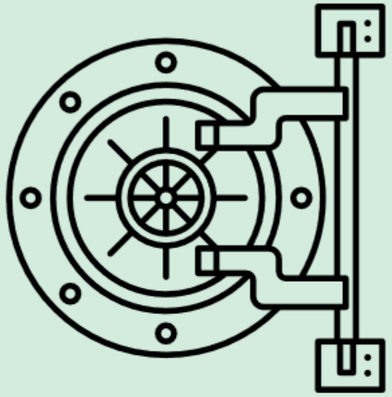
Method level security will be implemented along with custom filters in the authentication and authorization process

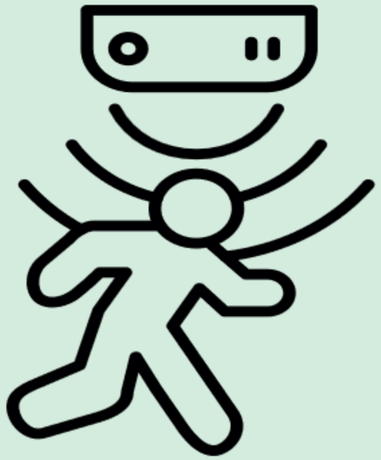
MILESTONE 5

Finally the application will be enhanced with OAUTH2/OpenID using JWT which is the most common security approach in the industry

What is Security & Why we need it ?

eazy bytes



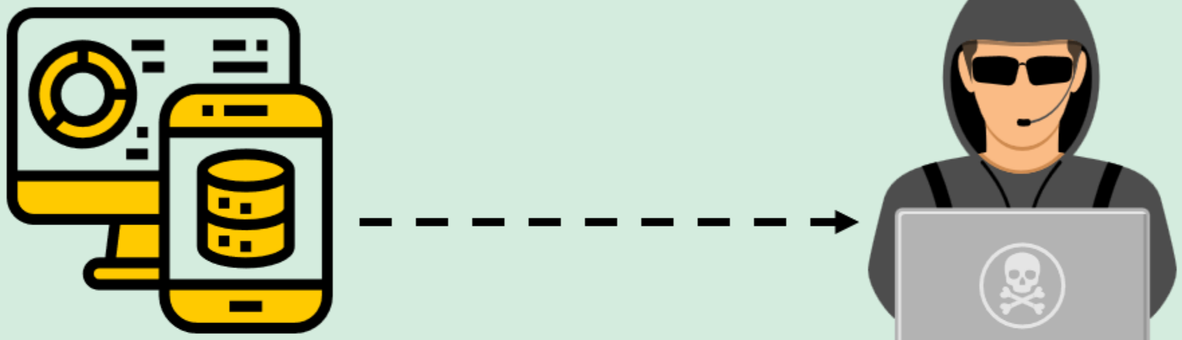


Have you ever observed how well Banks are protected?

Why Banks are well secured ? Because they hold valuable assets inside it

What is Security & Why we need it ?

eazy bytes



Similar to Banks, our web apps also hold valuable data. Then why not secure them?

Don’t you think all hackers will steal the data if web app is not properly secured ?

What is Security & Why we need it ?

eazy bytes

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, Authentication, Authorization etc.

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

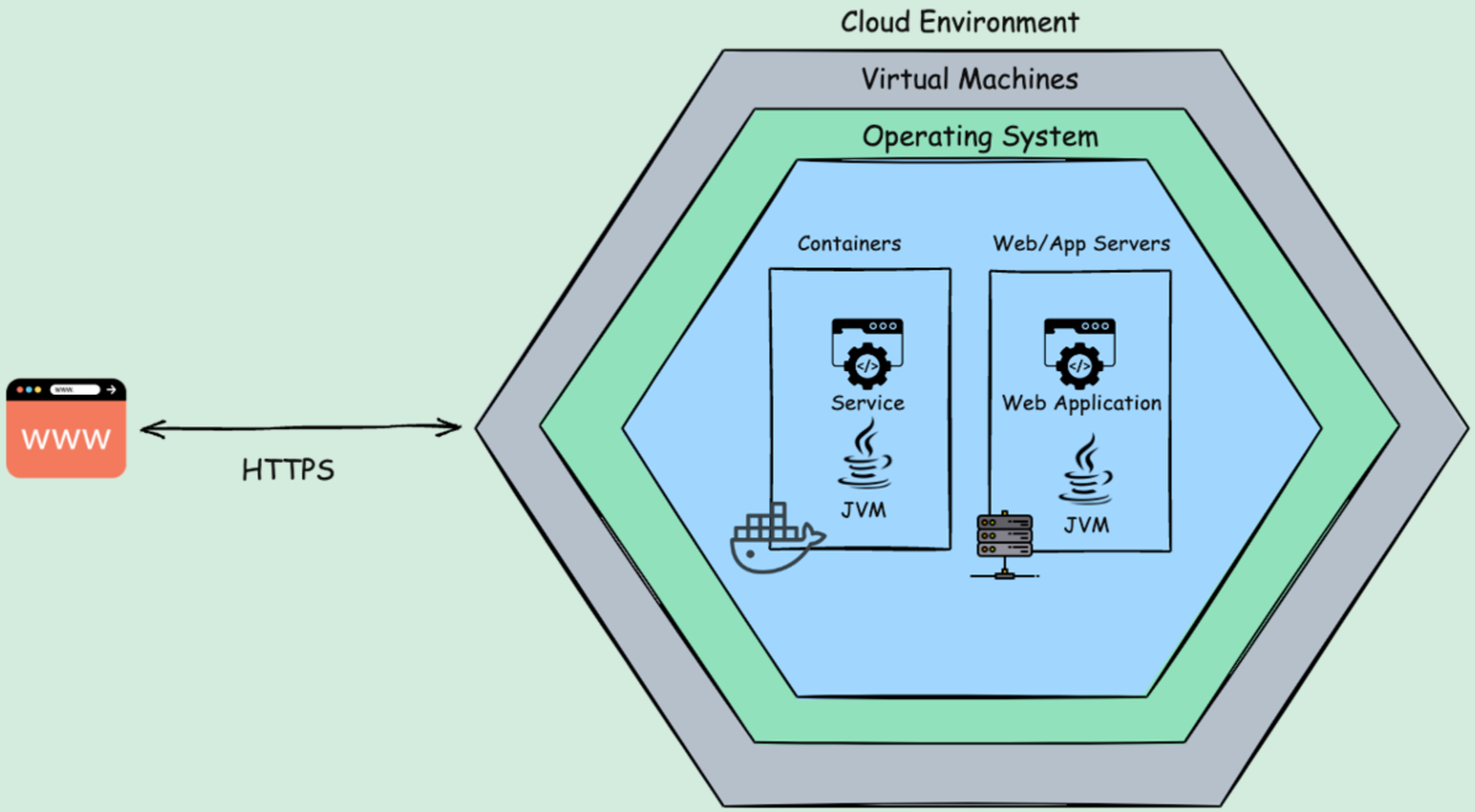
Security should be considered right from development phase itself along with business logic

AVOIDING MOST COMMON ATTACKS

Using Security we should also avoid most common security attacks like CSRF, Session Fixation, XSS, CORS etc. inside our application.

Typical security measures for an app

eazy bytes



Cloud Environment

Handle DDoS Attacks

Firewalls

Virtual Machines

Secure Access

Network Security

Operating System

Regular Updates and Patching

Antivirus & Malware protection

Containers

Use Trusted Images

Minimize Attack Surface

Implement Container Isolation

Web/App Server

Regular Security Updates

Access Control

Web Application Firewall (WAF)

Web Apps/Services

Authentication

Authorization

Protection from exploits like CSRF, CORS etc.

Why Spring Security ?

eazy bytes

Application security is not fun and challenging to implement with our custom code/framework.

Spring Security built by a team at Spring who are good at security by considering all the security scenarios. Using Spring Security, we can secure web apps with minimum configurations. So there is no need to re-invent the wheel here.

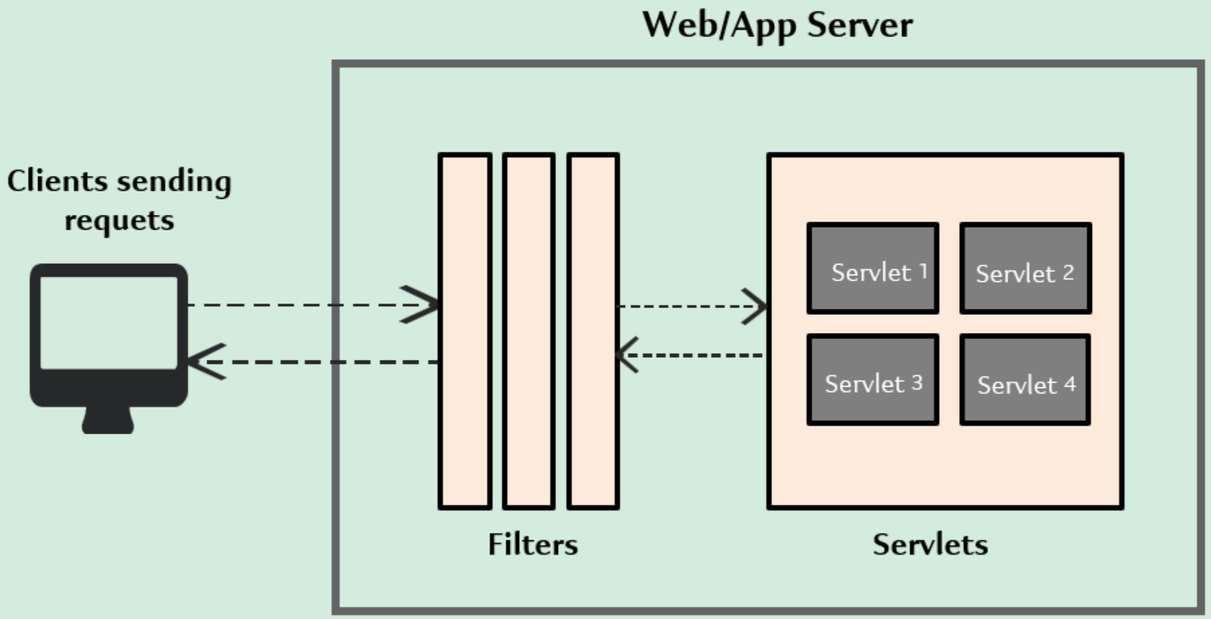
Spring Security handles the common security vulnerabilities like CSRF, CORs etc. For any security vulnerabilities identified, the framework will be patched immediately as it is being used by many organizations.

Using Spring Security we can secure our pages/API paths, enforce roles, method level security etc. with minimum configurations easily.

Spring Security supports various standards of security to implement authentication, like using username/password authentication, JWT tokens, OAuth2, OpenID etc.

Servlets & Filters

eazy bytes



★ Typical Scenario inside a web application

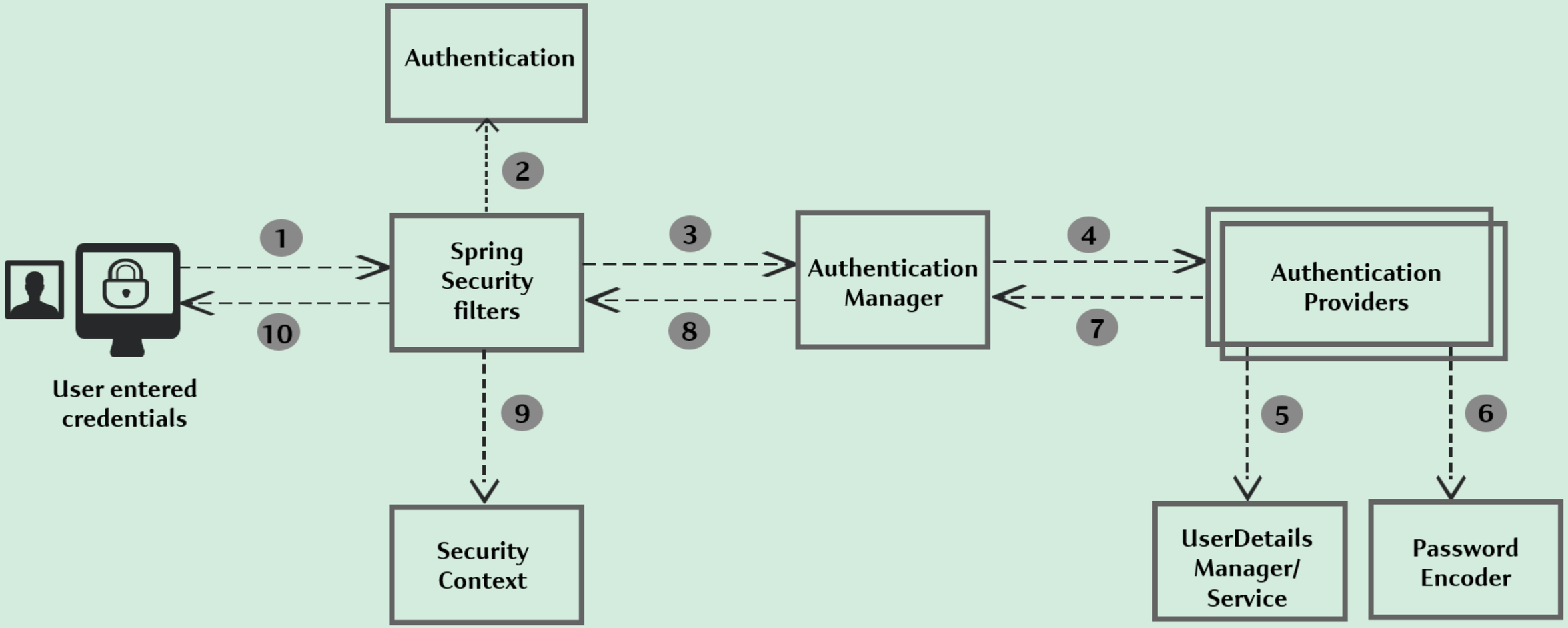
In Java web apps, Servlet Container (Web Server) takes care of translating the HTTP messages for Java code to understand. One of the mostly used servlet container is Apache Tomcat. Servlet Container converts the HTTP messages into ServletRequest and hand over to Servlet method as a parameter. Similarly, ServletResponse returns as an output to Servlet Container from Servlet. So everything we write inside Java web apps are driven by Servlets

★ Role of Filters

Filters inside Java web applications can be used to intercept each request/response and do some pre work before our business logic. So using the same filters, Spring Security enforce security based on our configurations inside a web application.

Spring Security Internal flow

eazy bytes



Spring Security Internal flow

eazy bytes

Spring Security Filters

A series of Spring Security filters intercept each request & work together to identify if Authentication is required or not. If authentication is required, accordingly navigate the user to login page or use the existing details stored during initial authentication.

Authentication

Filters like UsernamePasswordAuthenticationFilter will extract username/password from HTTP request & prepare Authentication type object. Because Authentication is the core standard of storing authenticated user details insdie Spring Security framework.

AuthenticationManager

Once received request from filter, it delegates the validating of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the AuthenticationManager to manage all the authentication providers available. In simple words, the authentication manager takes the responsibility for authentication.

AuthenticationProvider

AuthenticationProviders has all the core logic of validating user details for authentication.

UserDetailsManager/UserDetailsService

UserDetailsManager/UserDetailsService helps in retrieving, creating, updating, deleting the User Details from the DB/storage systems.

PasswordEncoder

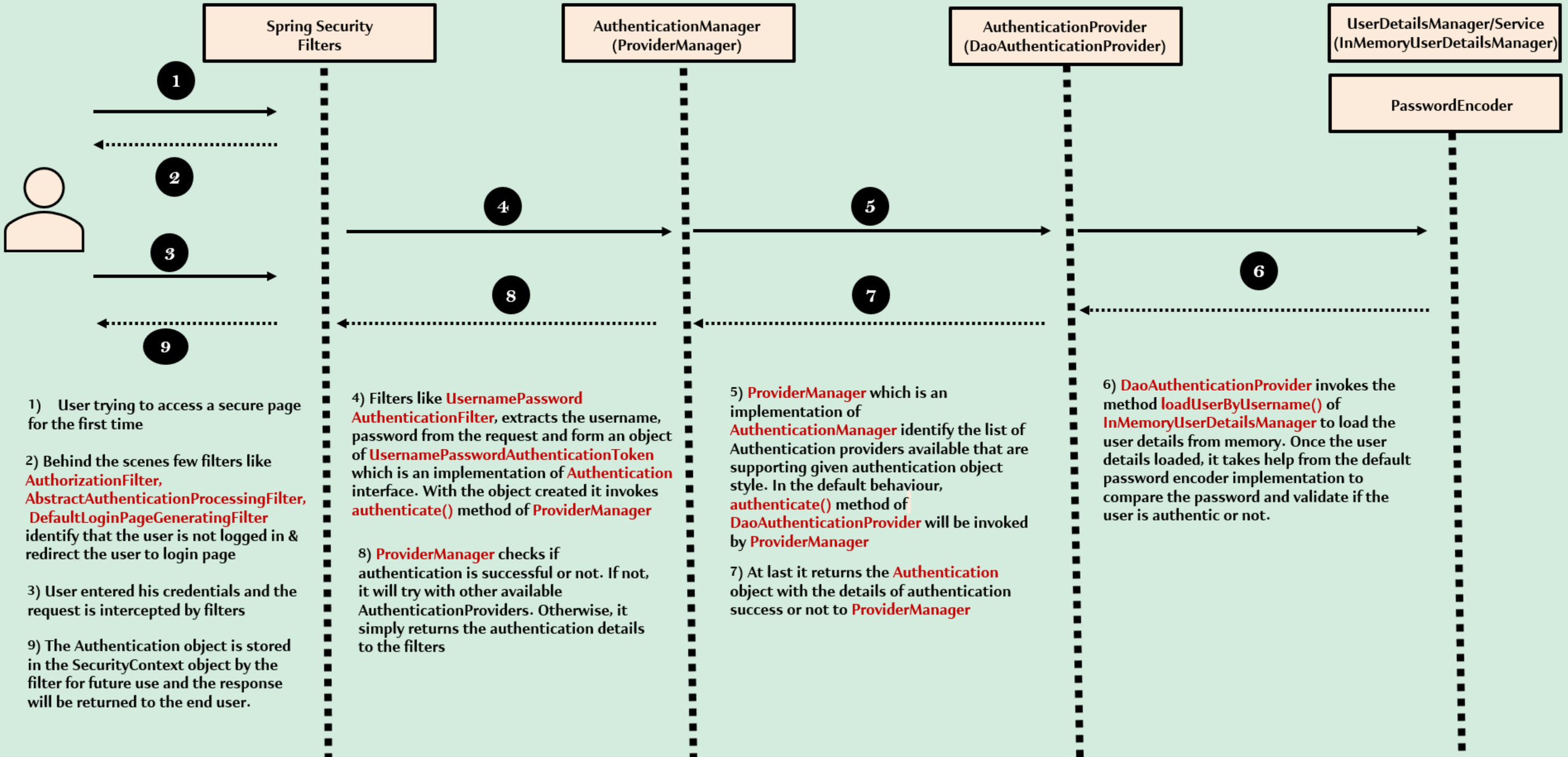
Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords ☹

SecurityContext

Once the request has been authenticated, the Authentication will usually be stored in a thread-local SecurityContext managed by the SecurityContextHolder. This helps during the upcoming requests from the same user.

Spring Security Internal flow for Default behaviour

eazy bytes



1) User trying to access a secure page for the first time

2) Behind the scenes few filters like AuthorizationFilter, AbstractAuthenticationProcessingFilter, DefaultLoginPageGeneratingFilter identify that the user is not logged in & redirect the user to login page

3) User entered his credentials and the request is intercepted by filters

4) Filters like UsernamePassword AuthenticationFilter, extracts the username, password from the request and form an object of UsernamePasswordAuthenticationToken which is an implementation of Authentication interface. With the object created it invokes authenticate() method of ProviderManager

5) ProviderManager which is an implementation of AuthenticationManager identify the list of Authentication providers available that are supporting given authentication object style. In the default behaviour, authenticate() method of DaoAuthenticationProvider will be invoked by ProviderManager

6) DaoAuthenticationProvider invokes the method loadUserByUsername() of InMemoryUserDetailsManager to load the user details from memory. Once the user details loaded, it takes help from the default password encoder implementation to compare the password and validate if the user is authentic or not.

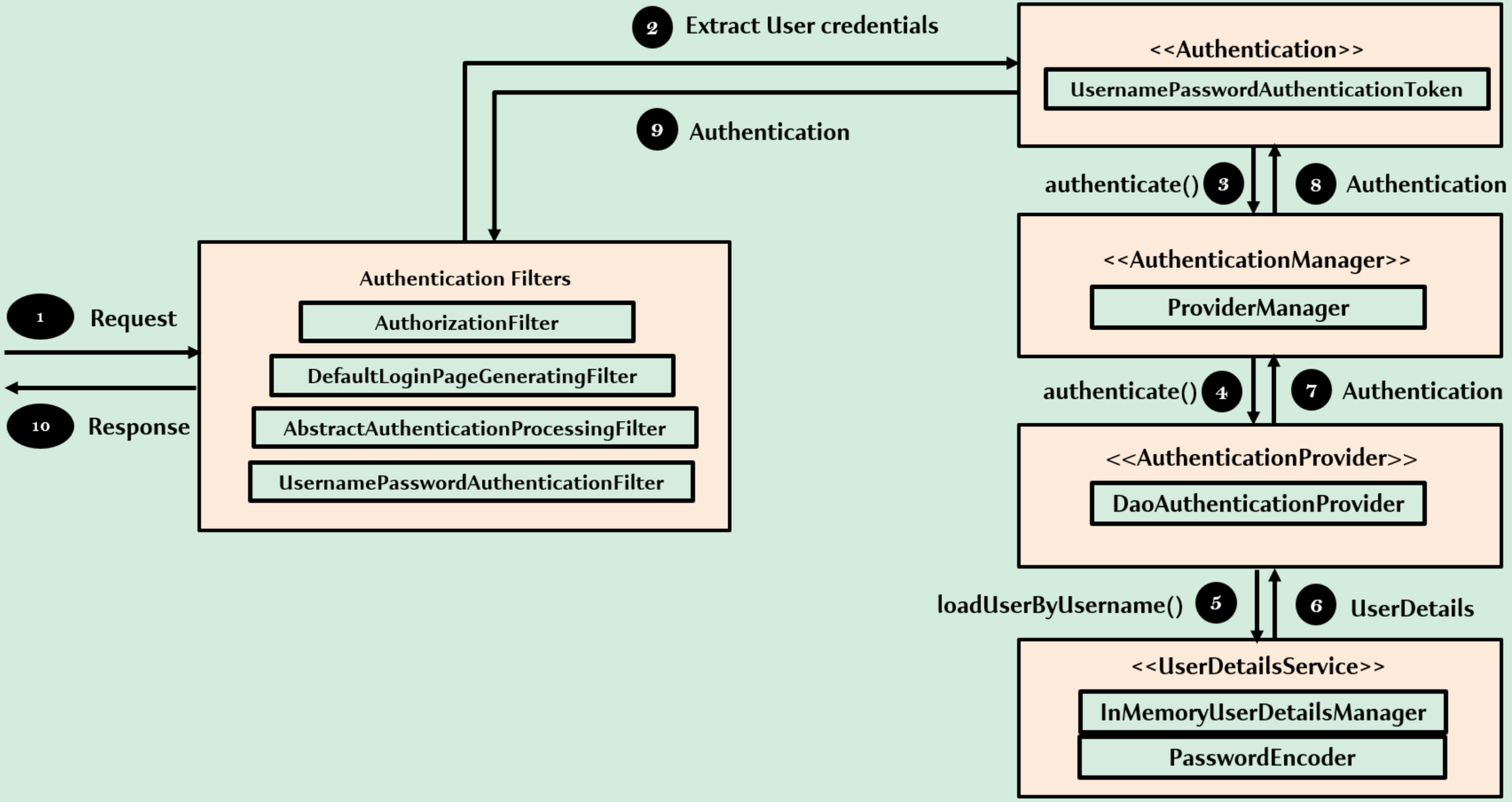
7) At last it returns the Authentication object with the details of authentication success or not to ProviderManager

8) ProviderManager checks if authentication is successful or not. If not, it will try with other available AuthenticationProviders. Otherwise, it simply returns the authentication details to the filters

9) The Authentication object is stored in the SecurityContext object by the filter for future use and the response will be returned to the end user.

Spring Security Internal flow for Default behaviour

eazy bytes



Backend REST services for EazyBank application

eazy bytes

Services with out any security

/contact – This service should accept the details from the Contact Us page in the UI and save to the DB.

/notices – This service should send the notice details from the DB to the 'NOTICES' page in the UI

Services with security

/myAccount – This service should send the account details of the logged in user from the DB to the UI

/myBalance – This service should send the balance and transaction details of the logged in user from the DB to the UI

/myLoans – This service should send the loan details of the logged in user from the DB to the UI

/myCards – This service should send the card details of the logged in user from the DB to the UI

DEFAULT SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

eazy bytes

By default, Spring Security framework protects all the paths present inside the web application. This behaviour is due to the code present inside the method defaultSecurityFilterChain(HttpSecurity http) of class SpringBootWebSecurityConfiguration

```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests((requests) -> requests.anyRequest().authenticated());
   http.formLogin(withDefaults());
   http.httpBasic(withDefaults());
   return http.build();
}
```

PERMIT ALL SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

eazy bytes

We can permit all the requests coming towards our web application APIs, Paths using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests((requests) -> requests.anyRequest().permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();

  }
```

DENY ALL SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

eazy bytes

We can deny all the requests coming towards our web application APIs, Paths using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests((requests) -> requests.anyRequest().denyAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();

  }
```

CUSTOM SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

eazy bytes

We can secure the web application APIs, Paths as per our custom requirements using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests((requests) -> requests
                .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards").authenticated()
                .requestMatchers("/notices", "/contact").permitAll())
          .formLogin(Customizer.withDefaults())
          .httpBasic(Customizer.withDefaults());
    return http.build();

  }
```

DISABLE FORM LOGIN

INSIDE SPRING SECURITY FRAMEWORK

eazy bytes

If your Spring Security-protected endpoints are only accessed programmatically (using APIs) by clients (such as mobile apps or other services) and there's no need for a web-based login form, disabling form login can be appropriate. This can be done in the following way,

```
http.authorizeHttpRequests((requests) -> requests
        .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards").authenticated()
        .requestMatchers("/notices","/contact", "/error").permitAll())
        .formLogin(flc -> flc.disable())
        .httpBasic(Customizer.withDefaults());
```

DISABLE HTTP BASIC LOGIN

INSIDE SPRING SECURITY FRAMEWORK

eazy bytes

If your application primarily serves web pages and requires users to interact via a browser, then HTTP Basic authentication may not be required. HTTP Basic style logic can be disabled in the following way,

```
http.authorizeHttpRequests((requests) -> requests
        .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards").authenticated()
        .requestMatchers("/notices","/contact", "/error").permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(hbc -> hbc.disable());
```

CONFIGURE USERS

Using InMemoryUserDetailsManager

NOT RECOMMENDED FOR PRODUCTION

eazy bytes

Instead of defining a single user inside application.properties, as a next step we can define multiple users along with their authorities with the help of InMemoryUserDetailsManager & UserDetails

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}

@Bean
public UserDetailsService userDetailsService() {
    UserDetails user = User.withUsername("user")
            .password("{noop}12345")
            .authorities("read")
            .build();
    UserDetails admin = User.withUsername("admin")
            .password("{bcrypt}$2a$12$v/X62C2bs.UrdI41uqHS/e1KgYR9i0KzMsNUKLrZok00QRwUWS1VW")
            .authorities("admin")
            .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

CompromisedPasswordChecker

eazy bytes

Using CompromisedPasswordChecker implementation, we can check if a password has been compromised. This feature is released as part of Spring Security 6.3 version.

```
@Bean
public CompromisedPasswordChecker compromisedPasswordChecker() {
   return new HaveIBeenPwnedRestApiPasswordChecker();
}
```

USER MANAGEMENT

IMPORTANT CLASSES & INTERFACES

eazy bytes

Core interface which loads user-specific data.

UserDetailsService (Interface)

- ✓ loadUserByUsername(String username)

An extension of the UserDetailsService which provides the ability to create new users and update existing ones.

UserDetailsManager (Interface)

- ✓ createUser(UserDetails user)
- ✓ updateUser(UserDetails user)
- ✓ deleteUser(String username)
- ✓ changePassword(String oldPwd, String newPwd)
- ✓ userExists(String username)

Sample implementation classes provided by the Spring Security team

- InMemoryUserDetailsManager
- JdbcUserDetailsManager
- LdapUserDetailsManager

UserDetails

All the above interfaces & classes uses an interface UserDetails & its implementation which provides core user information.

USERDETAILS & AUTHENTICATION

RELATION BETWEEN THEM

eazy bytes

Principal (Interface)

Authentication (Interface)

UsernamePasswordAuthenticationToken (Class)

- ✓ getName()
- ✓ getPrincipal()
- ✓ getAuthorities()
- ✓ getCredentials()
- ✓ getDetails()
- ✓ isAuthenticated()
- ✓ setAuthenticated()
- ✓ eraseCredentials()

Authentication is the return type in all the scenarios where we are trying to determine if the authentication is successful or not. Like inside the AuthenticationProvider & AuthenticationManager

Why do we have 2 separate ways to store login user details?

UserDetails (Interface)

User (Class)

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

UserDetails is the return type in all the scenarios where we try to load the user info from the storage systems. Like inside the UserDetailsService & UserDetailsManager

AUTHENTICATION

USING JdbcUserDetailsManager

eazy bytes

Instead of creating users inside the memory of web server, we can store them inside a DB and with the help of JdbcUserDetailsManager, we can perform authentication.

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    return new JdbcUserDetailsManager(dataSource);
}
```

Please note to create table as per the JdbcUserDetailsManager class & insert user records inside them.

USERDETAILSSERVICE IMPLEMENTATION

FOR CUSTOM USER FETCHING LOGIC

eazy bytes

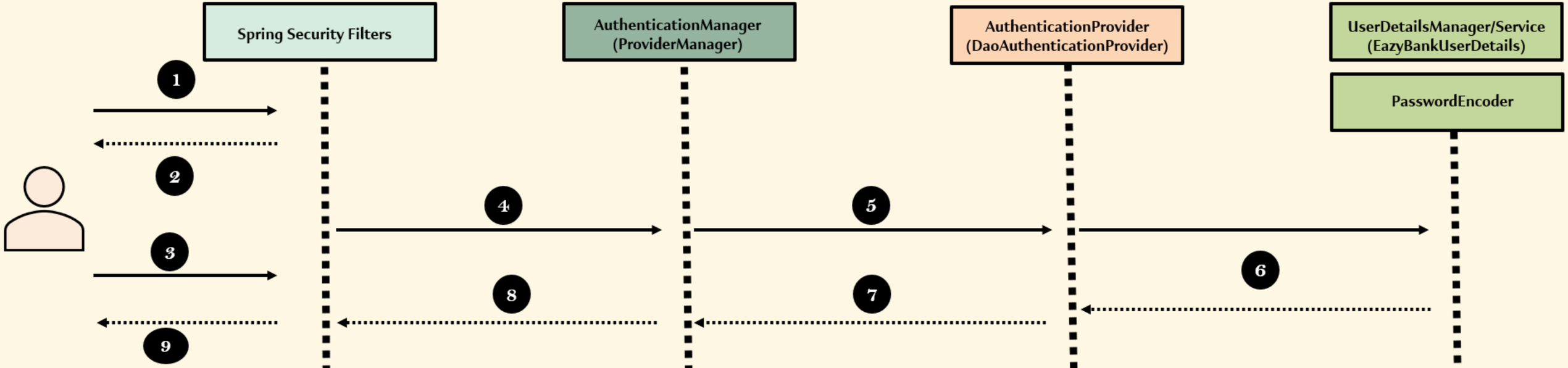
When we want to load the user details based on our own tables, columns, custom logic, then we need to create a bean that implements UserDetailsService and overrides the method loadUserByUsername()

```java
@Service
@RequiredArgsConstructor
public class EazyBankUserDetailsService implements UserDetailsService {

    private final CustomerRepository customerRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Customer customer = customerRepository.findByEmail(username).orElseThrow(() -> new
                UsernameNotFoundException("User details not found for the user: " + username));
        List<GrantedAuthority> authorities = List.of(new SimpleGrantedAuthority(customer.getRole()));
        return new User(customer.getEmail(), customer.getPwd(), authorities);
    }
}
```

SEQUENCE FLOW

WITH OUR OWN USERDETAILSSERVICE IMPLEMENTATION

eazy bytes



1) User trying to access a secure page for the first time

2) Behind the scenes few filters like AuthorizationFilter, AbstractAuthenticationProcessingFilter, DefaultLoginPageGeneratingFilter identify that the user is not logged in & redirect the user to login page

3) User entered his credentials and the request is intercepted by filters

9) The Authentication object is stored in the SecurityContext object by the filter for future use and the response will be returned to the end user.

4) Filters like UsernamePassword AuthenticationFilter, extracts the username, password from the request and form an object of UsernamePasswordAuthenticationToken which is an implementation of Authentication interface. With the object created it invokes authenticate() method of ProviderManager

8) ProviderManager checks if authentication is successful or not. If not, it will try with other available AuthenticationProviders. Otherwise, it simply returns the authentication details to the filters

5) ProviderManager which is an implementation of AuthenticationManager identify the list of Authentication providers available that are supporting given authentication object style. In the default behaviour, authenticate() method of DaoAuthenticationProvider will be invoked by ProviderManager

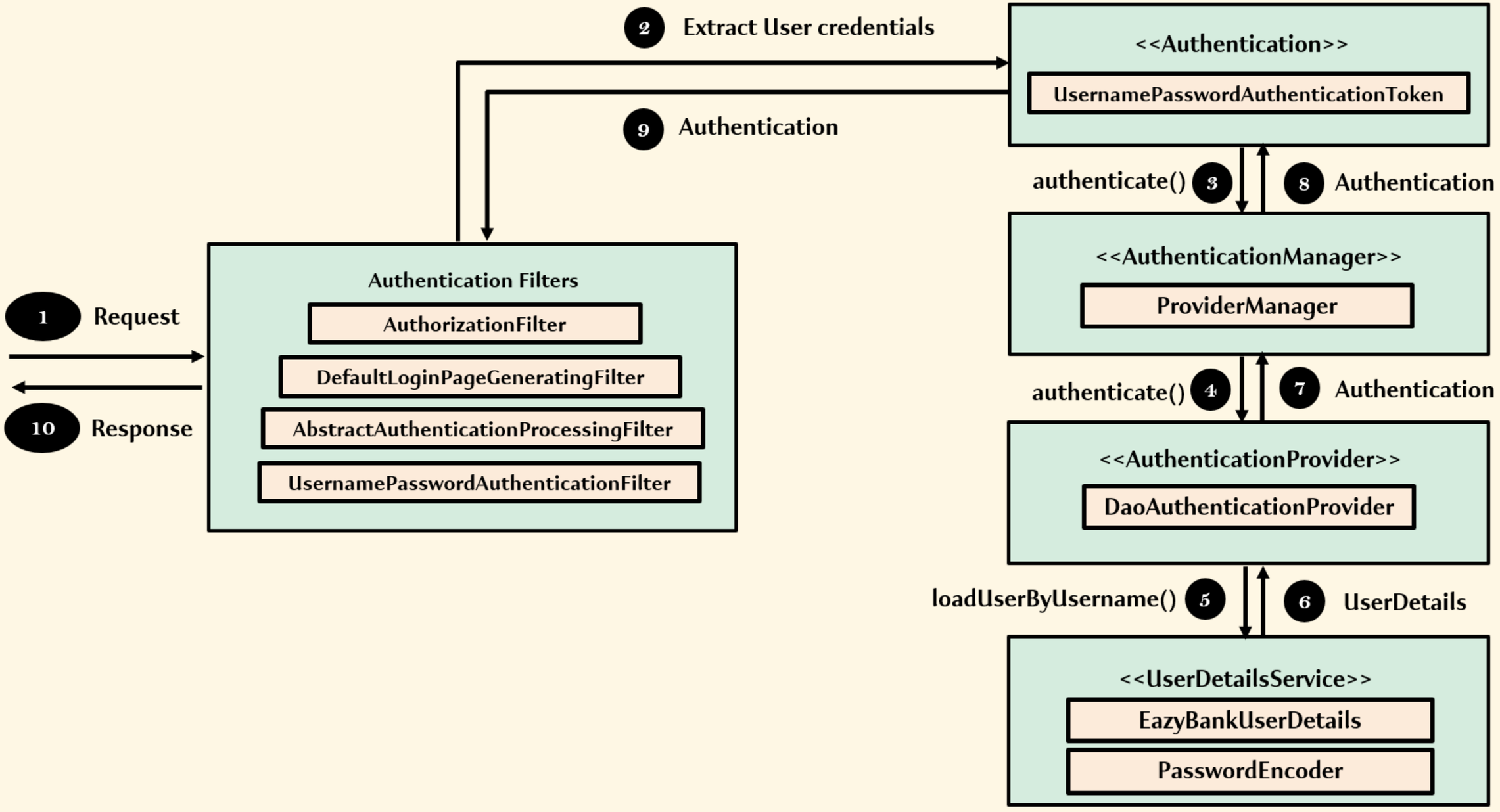
7) At last it returns the Authentication object with the details of authentication success or not to ProviderManager

6) DaoAuthenticationProvider invokes the method loadUserByUsername() of EazyBankUserDetails to load the user details. Once the user details loaded, it takes help from the default password encoder implementation to compare the password and validate if the user is authentic or not.

SEQUENCE FLOW

WITH OUR OWN USERDETAILSSERVICE IMPLEMENTATION

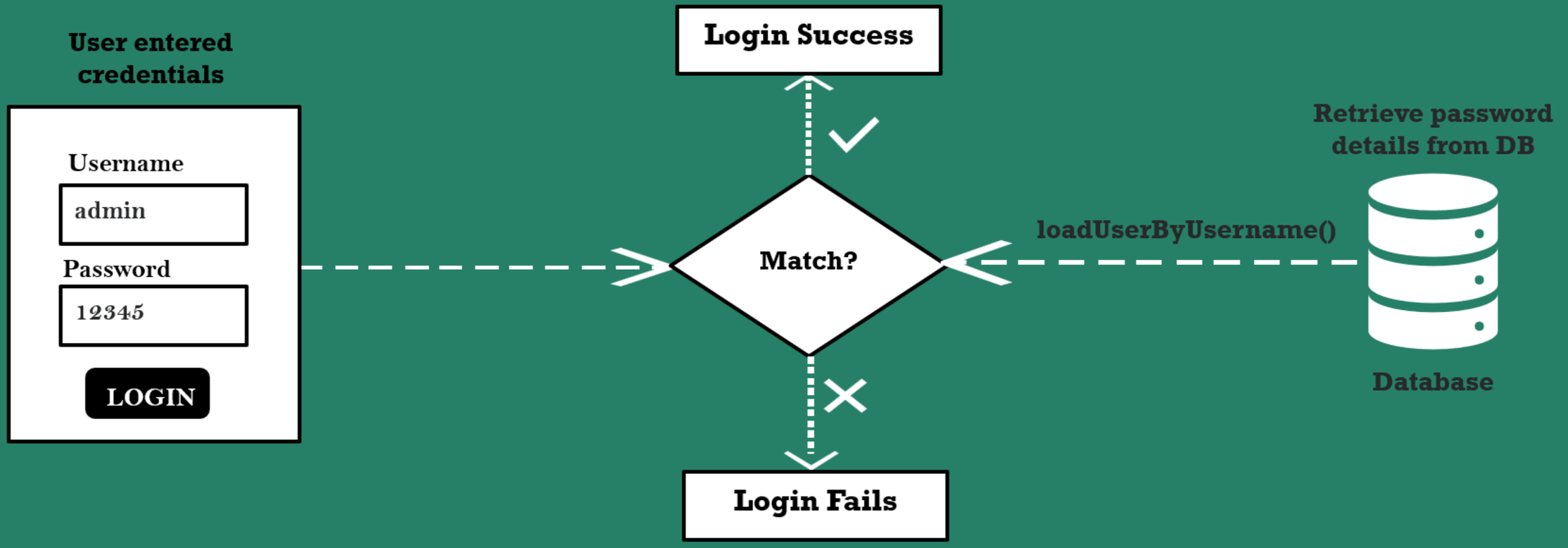
eazy bytes



HOW PASSWORDS VALIDATED

With NoOpPasswordEncoder

eazy bytes



Storing the passwords in a plain text inside a storage system like DB will have Integrity & Confidentiality issues. So this is not a recommended approach for Production applications.

Encoding Vs Encryption Vs Hashing

eazy bytes

Different ways for Data privacy

Encoding

- ✓ Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography.
- ✓ It involves no secret and completely reversible.
- ✓ Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding.

Ex: ASCII, BASE64, UNICODE

Encryption

- ✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.
- ✓ To achieve confidentiality, encryption requires the use of a secret which, in cryptographic terms, we call a "key".
- ✓ Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured.

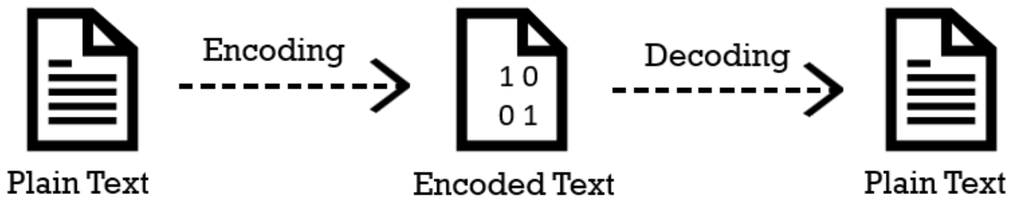
Hashing

- ✓ In hashing, data is converted to the hash value using some hashing function.
- ✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
- ✓ 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

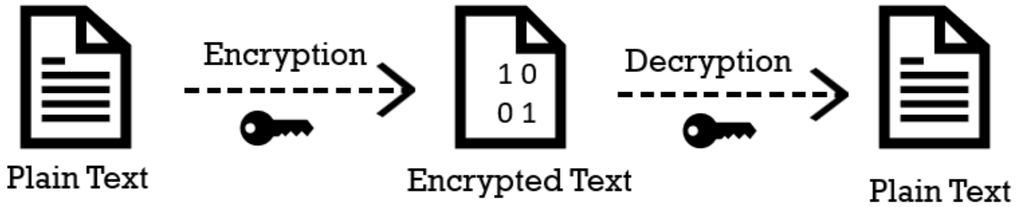
Encoding Vs Encryption Vs Hashing

eazy bytes

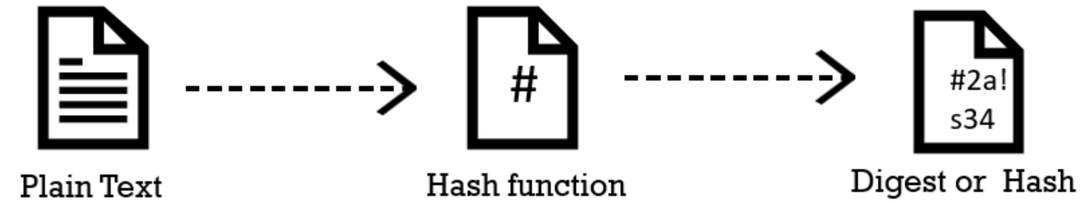
Encoding & Decoding



Encryption & Decryption



Hashing (Only 1-way)



DEEP DIVE OF ENCRYPTION

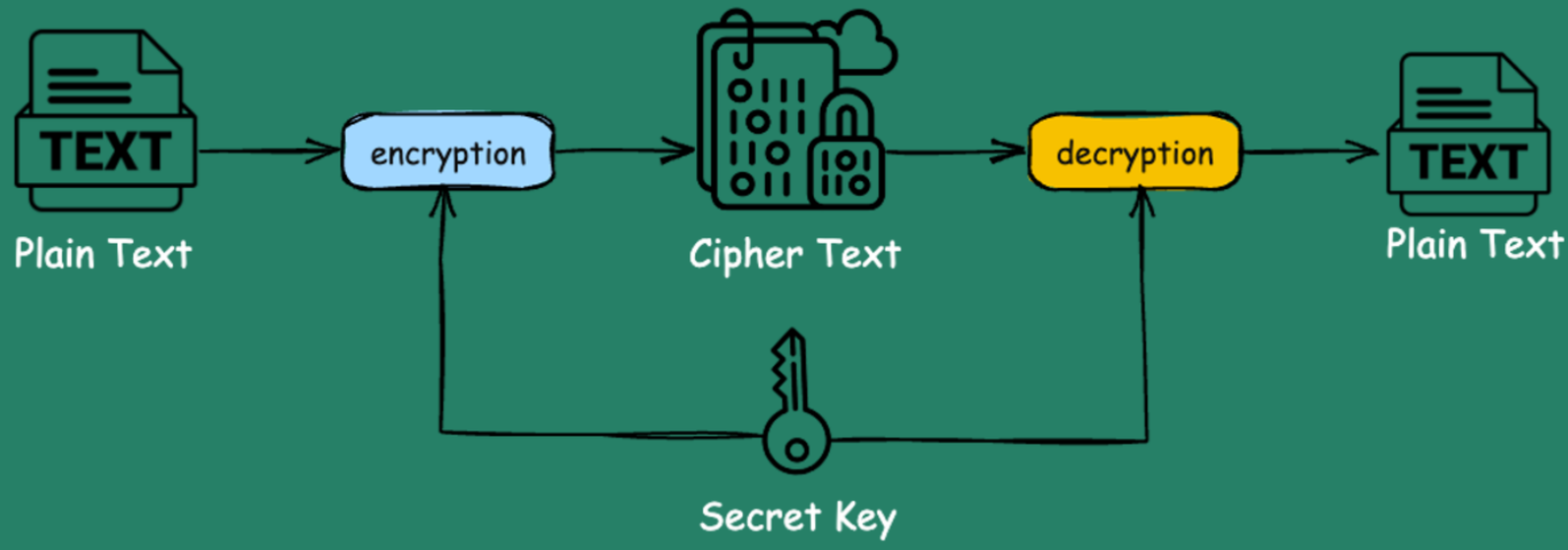
eazy bytes

Encryption is essential in our daily routines, whether you're messaging friends on WhatsApp, ensuring a website's authenticity through your browser, or entering your credit card details while shopping online. It safeguards your data from potential threats and unauthorized access, ensuring privacy and security.

Central to any encryption process are the **encryption algorithm** and the **key**. While numerous encryption algorithms exist, there are generally two types of keys: **symmetric** and **asymmetric**.

01 Symmetric encryption -

In symmetric key encryption, the same key used to encrypt the data is used to decrypt the data. Some famous symmetric encryption algorithms include AES, known for its security and used globally; Triple DES, an enhancement of DES providing better security; and Blowfish, notable for its speed and variable key lengths. Each offers distinct advantages for protecting data. symmetric key encryption is generally used for encrypting **data at rest**

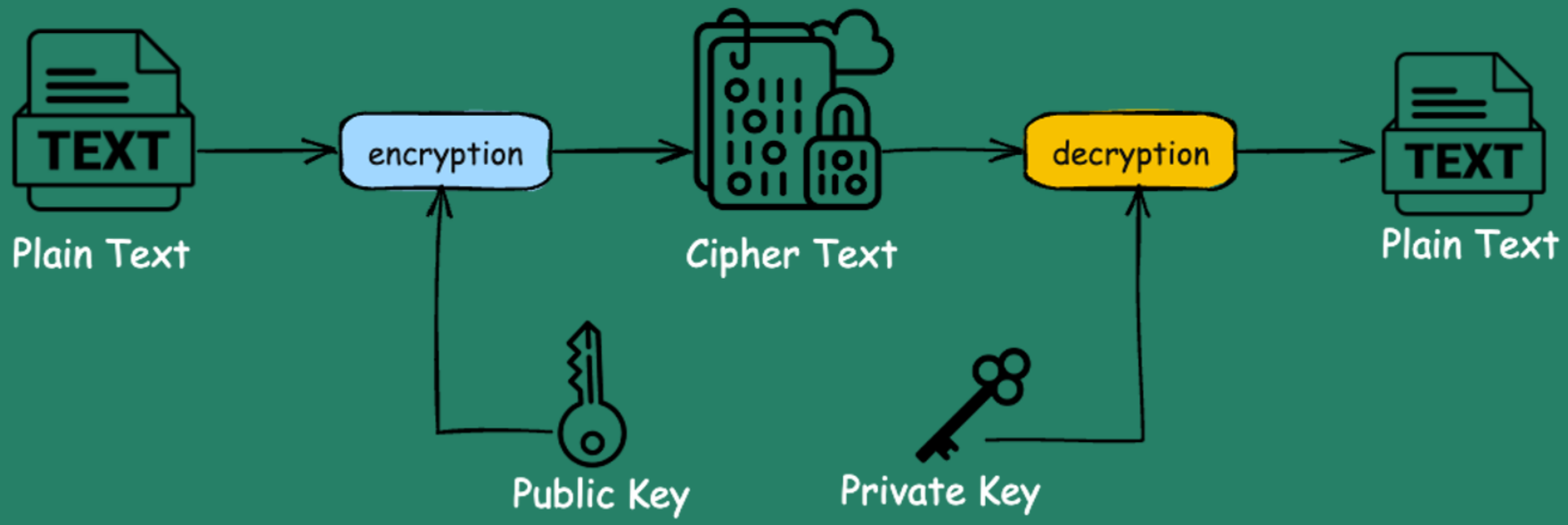


DEEP DIVE OF ENCRYPTION

eazy bytes

02 Asymmetric encryption -

Asymmetric encryption functions like a unique lock with two distinct keys: one for locking (encrypting) and one for unlocking (decrypting). You keep the private key, which unlocks the message, secure with you, while freely distributing the public key to anyone who wants to send you an encrypted message. They use this public key to encrypt their message, and only you can decrypt it using your private key. This system ensures secure communication, as it relies on a pair of keys: a public key for encryption and a private key for decryption



Asymmetric key encryption is ideal for encrypting data in transit, where you need to share the key with another system. The most commonly used asymmetric algorithms include Rivest-Shamir-Adleman (RSA), Diffie-Hellman, Elliptic Curve Cryptography (ECC) and Pretty Good Privacy (PGP).

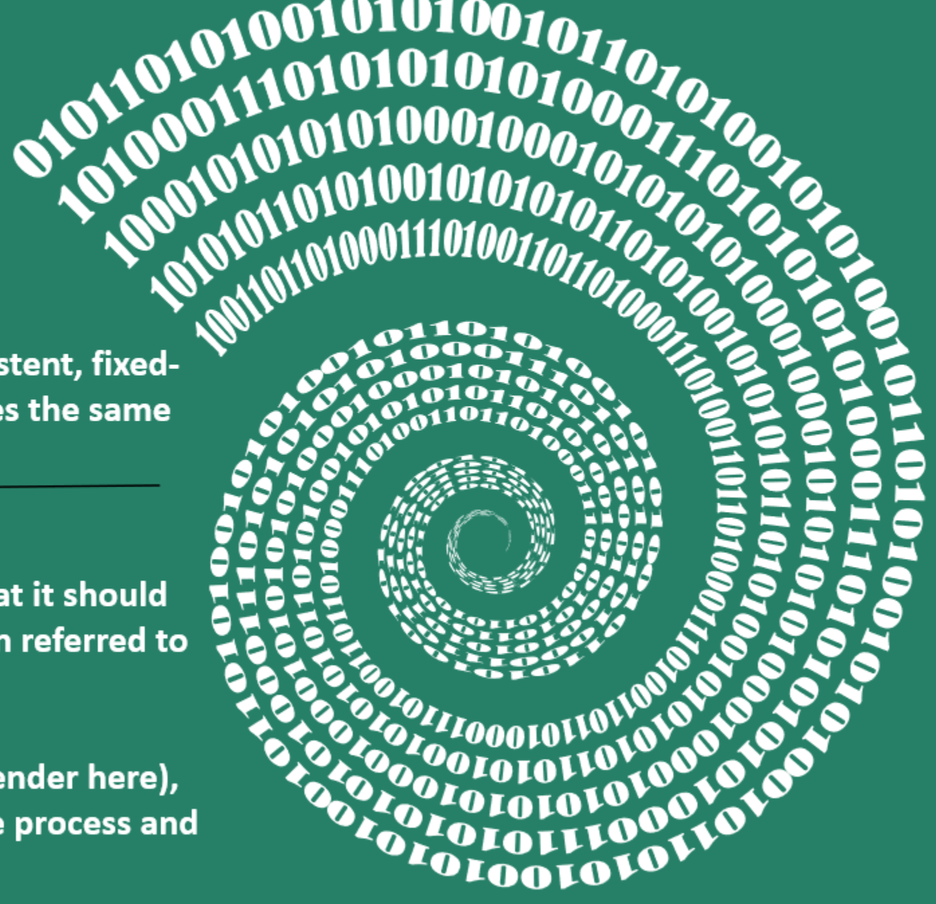
DEEP DIVE OF HASHING

eazy bytes

01

Hash functions take any data as input and produce a unique string of bytes in return. Given the same input, the hash function always reproduces the same string of bytes. The output of a hash function is often called a digest or a HASH.

12345 → Hashing → 5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5



A hash function transforms input of any length (such as a file, message, or video) into a consistent, fixed-length output (e.g., 256 bits for SHA-256). When the same input is hashed, it always generates the same resulting digest or hash.

02

A fundamental characteristic of a hash function is its irreversibility, meaning that it should not be possible to deduce the input from its output alone. This property is often referred to as being "one-way."

In other words, given the output produced by a hash function (depicted as a blender here), it should be computationally infeasible, or practically impossible, to reverse the process and uncover the original input.

HASHING PASSWORDS

eazy bytes

The typical approach for storing passwords within an organization involves hashing user passwords and retaining only the resulting digests. When a user attempts to log in to your website, the process typically follows these steps:

1. You receive the user's password during the registration process.
2. You hash the provided password and discard the original. Store it in a storage system like Database
3. During login operation, you compare the resulting digest with the stored version; if they match, the user gains access.

While this may look like a perfect solution as no one can get to know the plain plassword from the hash or digest value even if they manage to steal the user data from DB. It has some loop holes. Let's imagine few users they used simple passwords and there is also a good chance that multiple user choosed to have the same password like **123456** or **password** etc. Below is how, it is going to look,

User ID	Username	Plain password	Hash password
1	johndoe	12345	5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
2	mike123	password	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
3	pavan189	12345	5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
4	saanvi180	password	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

HASHING PASSWORDS

eazy bytes

There will be couple of problems if we use simple hashing mechanisms for passwords management.

1. If an attacker gains access to hashed passwords in a DB, they could potentially launch a brute force attack or an exhaustive search (on Rainbow table) by trying all possible passwords. This method tests each attempt against the entire database of hashed passwords. Ideally, we aim to limit attackers to targeting only one hashed password at a time.

2. Hash functions are designed to be fast, which attackers can exploit to try numerous passwords per second during a brute force attack. Ideally, we should implement mechanisms to impede such attacks, slowing down the process significantly.

What is a Brute force attack ?

A brute force attack is a trial-and-error method used by hackers to decode encrypted data, such as passwords or encryption keys. In this type of attack, the attacker systematically tries all possible combinations of characters until the correct one is found. It's essentially like trying every possible key in a lock until the right one opens it.

What is a Rainbow table or Dictionary table attack ?

A rainbow table or Dictionary table attacks are a type of cryptographic attack used to crack password hashes. It involves precomputing and storing a large number of password hashes along with their corresponding plaintext passwords in a table known as a "rainbow table" or a "dictionary table"

When a hashed password is obtained, instead of computing hashes for all possible passwords as in a brute force attack, the attacker can simply look up the hash in the rainbow or dictionary table to find the corresponding plaintext password. This significantly reduces the time and computational resources required to crack the password.

HASHING PASSWORDS

eazy bytes

Let’s try to understand how to overcome the challenges of Brute force attacks and Rainbow table attacks,

1. To prevent rainbow table attacks, **salts** are commonly used. Salts are random values that are unique for each user and are made public. The salt value is stored as part of the resulting hash itself. Specifically, the salt is included in the hash string along with the actual hashed password. The salt is typically stored at the beginning of the hash string and is used during the password verification process to generate the same hash again for comparison. Since each user has a unique hash function, attackers can't precompute large tables of passwords (rainbow tables) to test against the entire database of stolen password hashes.

> Plain text password + Random salt value + Hashing Algo = Protection from Rainbow attacks

2. Brute force attacks issue can be solved with password hashes, which are designed to be slow. The normal hashing process is quite fast, due to which attackers can do the trail and error by trying any number of plain password to check against hashed password. We can make attacker life tough by using password hashing algorithms like **PBKDF2, bcrypt, scrypt and Argon2**. These algorithms slows the hashing process and demands lots of CPU cost, memory cost.

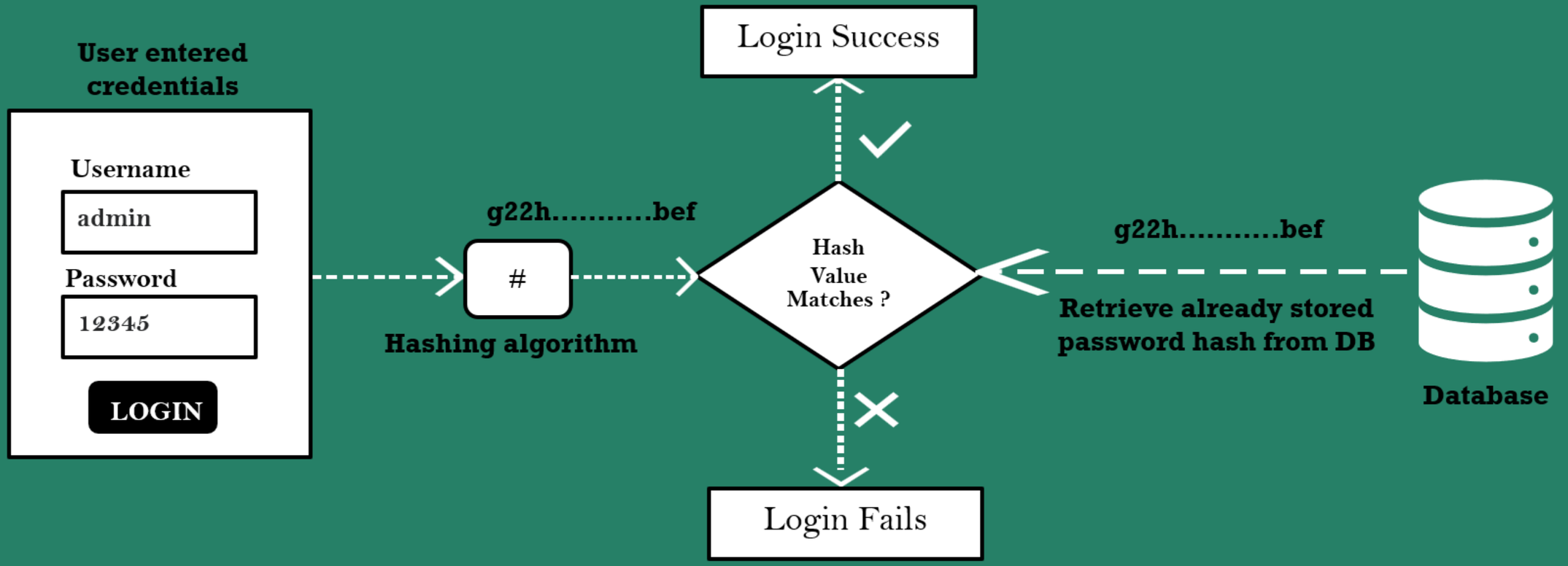
These algorithms are specifically designed to slow down the hashing process, thereby increasing the time and resources required for an attacker to guess passwords. They achieve this by introducing additional computational costs, such as CPU and memory, into the hashing process. By making it more computationally expensive to hash passwords, these algorithms make brute force attacks less practical and deter attackers from attempting them.

> Spring Security provides industry recommended Password Encoders that are capable of generating random salt and leverage password hashing algorithms like bcrypt

HOW PASSWORDS VALIDATED

With Hashing & PasswordEncoders

eazy bytes



Storing & managing the passwords with hashing is the recommended approach for Production applications. With various PasswordEncoders available inside Spring Security, it makes our life easy.

DETAILS OF PASSWORDENCODER

eazy bytes

Methods inside PasswordEncoder Interface

```
public interface PasswordEncoder {

        String encode(CharSequence rawPassword);

        boolean matches(CharSequence rawPassword, String encodedPassword);

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

Different implementations of PasswordEncoder inside Spring Security

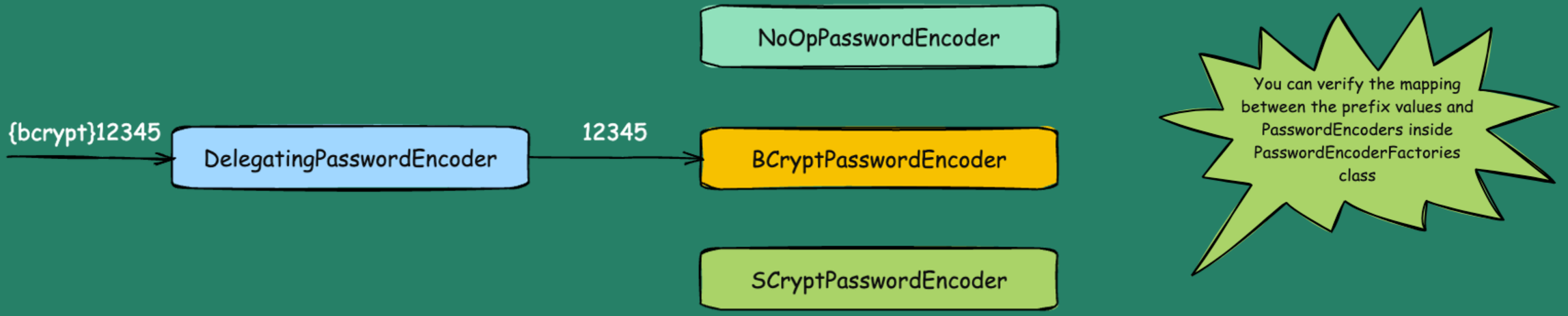
- NoOpPasswordEncoder (Not recommended for Prod apps)
- StandardPasswordEncoder (Not recommended for Prod apps)
- Pbkdf2PasswordEncoder
- BCryptPasswordEncoder
- SCryptPasswordEncoder
- Argon2PasswordEncoder

MULTIPLE ENCODING STRATEGIES WITH DELEGATINGPASSWORDENCODER

eazy bytes

When needing to switch password hashing algorithms in an application, such as due to a vulnerability, the DelegatingPasswordEncoder is handy. It allows managing multiple hash types simultaneously. For instance, it enables using a new algorithm for new users while maintaining the old one for existing credentials.

The DelegatingPasswordEncoder delegates to the correct implementation of the PasswordEncoder based on the prefix of the password.



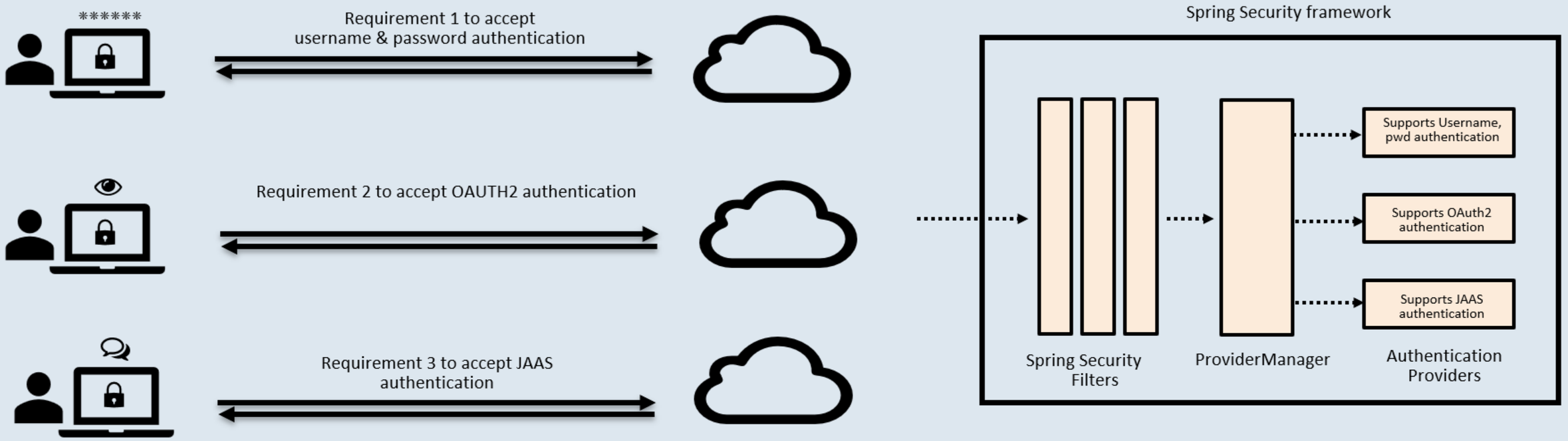
During the user registration process, the DelegatingPasswordEncoder uses the default password encoder defined in the framework and appends the corresponding prefix to the encoded password.

During the login operation, when we call the matches() method with a password that has prefix {bcrypt}, the call is delegated to the BCryptPasswordEncoder. Similarly based on the other prefix values, the corresponding password encoder will be selected.

AUTHENTICATION PROVIDER

WHY DO WE NEE IT?

eazy bytes



- ✓ The AuthenticationProvider in Spring Security takes care of the authentication logic. The default implementation of the AuthenticationProvider is to delegate the responsibility of finding the user in the system to a UserDetailsService implementation & 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.

- ✓ It is the responsibility of the ProviderManager which is an implementation of AuthenticationManager, to check with all the implementations of Authentication Providers and try to authenticate the user.

DETAILS OF AUTHENTICATION PROVIDER

eazy bytes

Methods inside AuthenticationProvider Interface

```
public interface AuthenticationProvider {

        Authentication authenticate(Authentication authentication)
                throws AuthenticationException;

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

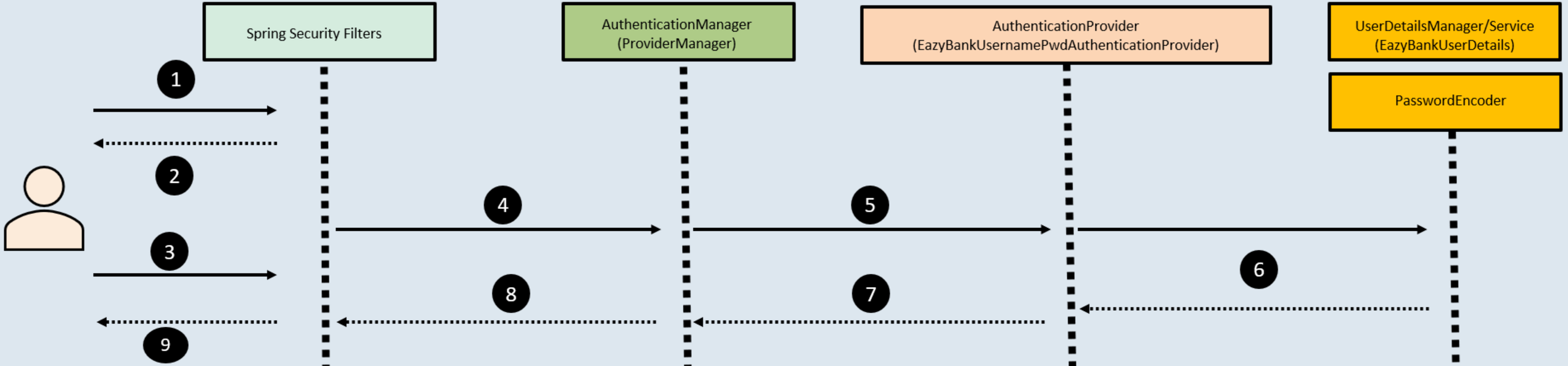
- The authenticate() method receives and returns authentication object. We can implement all our custom authentication logic inside authenticate() method.

- 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 of the Authentication object provided.

SEQUENCE FLOW

WITH OUR OWN AUTHENTICATIONPROVIDER IMPLEMENTATION

eazy bytes



1) User trying to access a secure page for the first time

2) Behind the scenes few filters like AuthorizationFilter, AbstractAuthenticationProcessingFilter, DefaultLoginPageGeneratingFilter identify that the user is not logged in & redirect the user to login page

3) User entered his credentials and the request is intercepted by filters

9) The Authentication object is stored in the SecurityContext object by the filter for future use and the response will be returned to the end user.

4) Filters like UsernamePassword AuthenticationFilter, extracts the username, password from the request and form an object of UsernamePasswordAuthenticationToken which is an implementation of Authentication interface. With the object created it invokes authenticate() method of ProviderManager

8) ProviderManager checks if authentication is successful or not. If not, it will try with other available AuthenticationProviders. Otherwise, it simply returns the authentication details to the filters

5) ProviderManager which is an implementation of AuthenticationManager identify the list of Authentication providers available that are supporting given authentication object style. In this scenario, the authenticate() method of our custom AuthenticationProvider will be invoked by ProviderManager

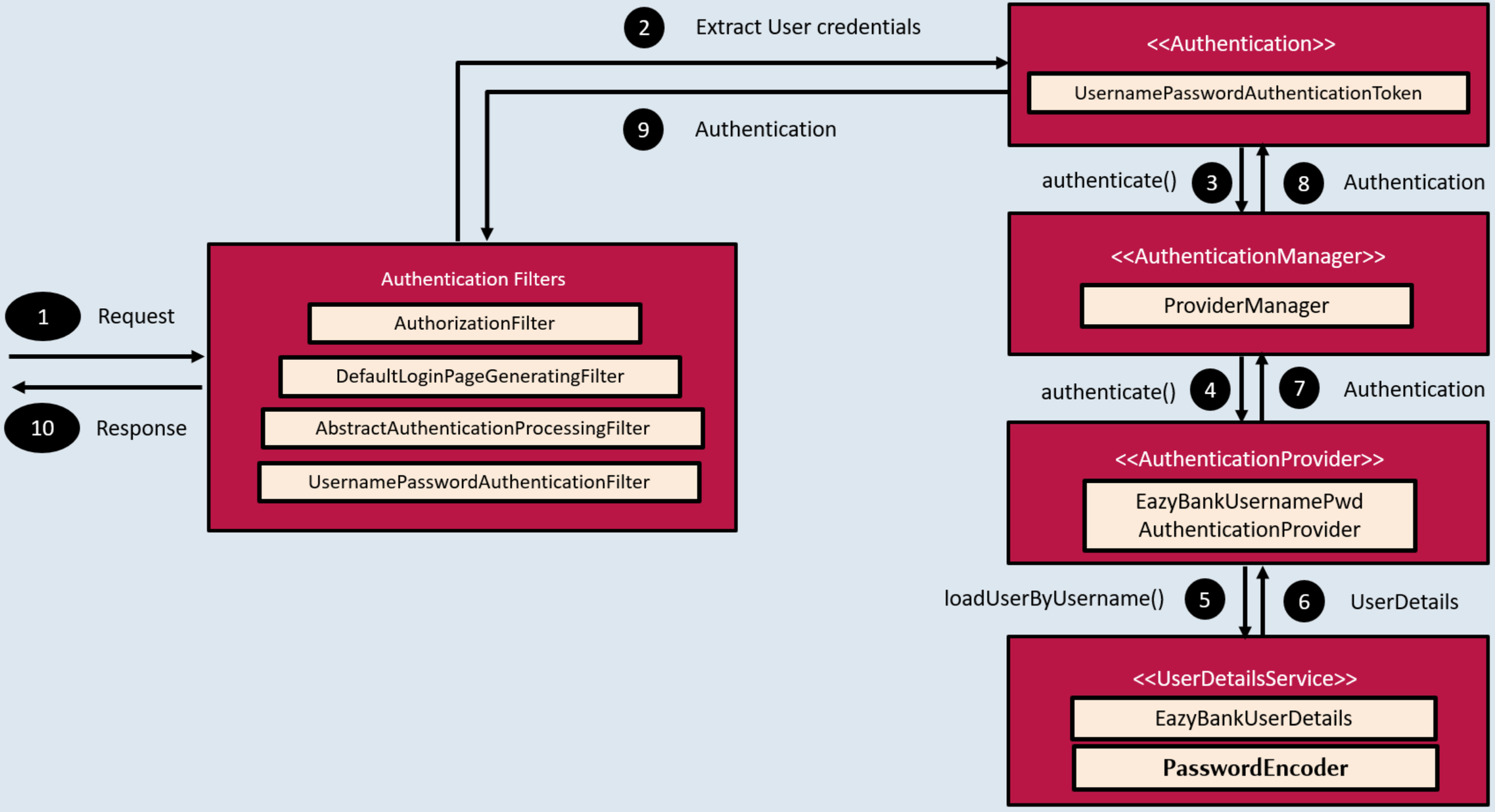
7) At last it returns the Authentication object with the details of authentication success or not to ProviderManager

6)EazyBankUsernamePwdAuthenticationProvider invokes the method loadUserByUsername() of EazyBankUserDetails to load the user details. Once the user details loaded, it takes help from the password encoder implementation to compare the password and validate if the user is authentic or not.

SEQUENCE FLOW

WITH OUR OWN AUTHENTICATIONPROVIDER IMPLEMENTATION

eazy bytes



Environment specific Security configurations using Profiles

eazy bytes

We can activate Spring Secuirty configuration or beans conditionally using Spring Boot profiles concept. Below is the sample snippet on how we can activate a Bean or configuration based on a profile,

```
@Configuration
@Profile("!prod")
public class ProjectSecurityConfig {


}
```

```
@Configuration
@Profile("prod")
public class ProjectSecurityProdConfig {

}
```

```
@Component
@Profile("!prod")
public class EazyBankUsernamePwdAuthenticationProvider
        implements AuthenticationProvider {

}
```

```
@Component
@Profile("prod")
public class EazyBankProdUsernamePwdAuthenticationProvider
        implements AuthenticationProvider {

}
```

Accepting only HTTPS Traffic

Inside spring security framework

eazy bytes

Enabling HTTPS in your application is crucial for safeguarding data confidentiality and integrity during communication between clients and servers. By encrypting the data transmitted, HTTPS prevents eavesdropping and man-in-the-middle attacks. Spring Security offers effortless integration to ensure only HTTPS traffic is accepted in your application.

Below are the Spring Security configurations to accept HTTPS traffic only,

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.requiresChannel(rcf -> rcf.anyRequest().requiresSecure())
            .authorizeHttpRequests((requests) -> requests
            .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards").authenticated()
            .requestMatchers("/notices","/contact", "/error").permitAll());
    return http.build();
}
```

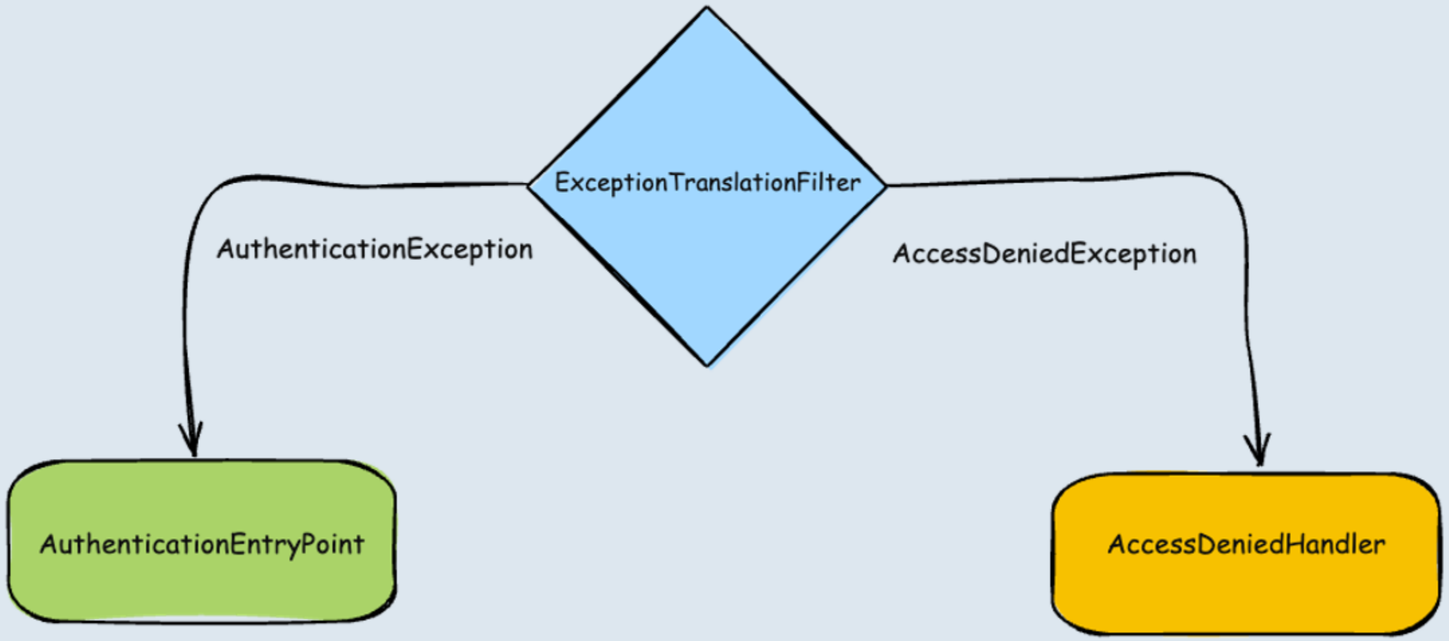
If you are looking for an option to accept only HTTP traffic, then you can invoke requiresInsecure() Method instead of requiresSecure() method

Exception Handling in Spring Security framework

eazy bytes

If a client attempts to access a resource without authentication or proper authorization, they should be prompted to provide credentials to access the resource. This can be achieved by either redirecting the client to a login page in the user interface or by informing them about the situation through an API response.

Inside Spring Security authentication & authorization related exceptions are handled by `ExceptionTranslationFilter`



AuthenticationEntryPoint

Inside spring security framework

eazy bytes

An AuthenticationEntryPoint in your Spring application serves as a handler for unauthenticated access attempts, determining how to respond when someone tries to access a part of your site or API without the authentication. Think of it as a gatekeeper in a party who decided what needs to be done if there is no invite. Similarly AuthenticationEntryPoint might redirect users to a login page or return an error message indicating that authentication is necessary.

Below is the definition of AuthenticationEntryPoint interface,

```
public interface AuthenticationEntryPoint {

    void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException)
            throws IOException, ServletException;

}
```

LoginUrlAuthenticationEntryPoint, BasicAuthenticationEntryPoint are the few of the Spring Security provider implementations of AuthenticationEntryPoint interface that gets executed by default in case of AuthenticationException.

Defining Custom AuthenticationEntryPoint

Inside spring security framework

eazy bytes

Below are the snippets on how to define a Custom AuthenticationEntryPoint for the HTTP Basic login flow,

1) We need to create a implementation class using AuthenticationEntryPoint interface & by overriding the commence() method. Inside the commence() method, we can build logic on what response to send back to clients based on our requirements,

```
public class CustomBasicAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
                         AuthenticationException authException) throws IOException {
        response.setHeader("eazybank-error-reason", "Authentication failed");
        response.sendError(HttpStatus.UNAUTHORIZED.value(), authException.getMessage());
    }
}
```

2) The CustomBasicAuthenticationEntryPoint created in the above step can be configured for the httpBasic flow like shown below,

```
http.httpBasic(hbc -> hbc.authenticationEntryPoint(
    new CustomBasicAuthenticationEntryPoint()));
```

Defining Custom AuthenticationEntryPoint

Inside spring security framework

eazy bytes

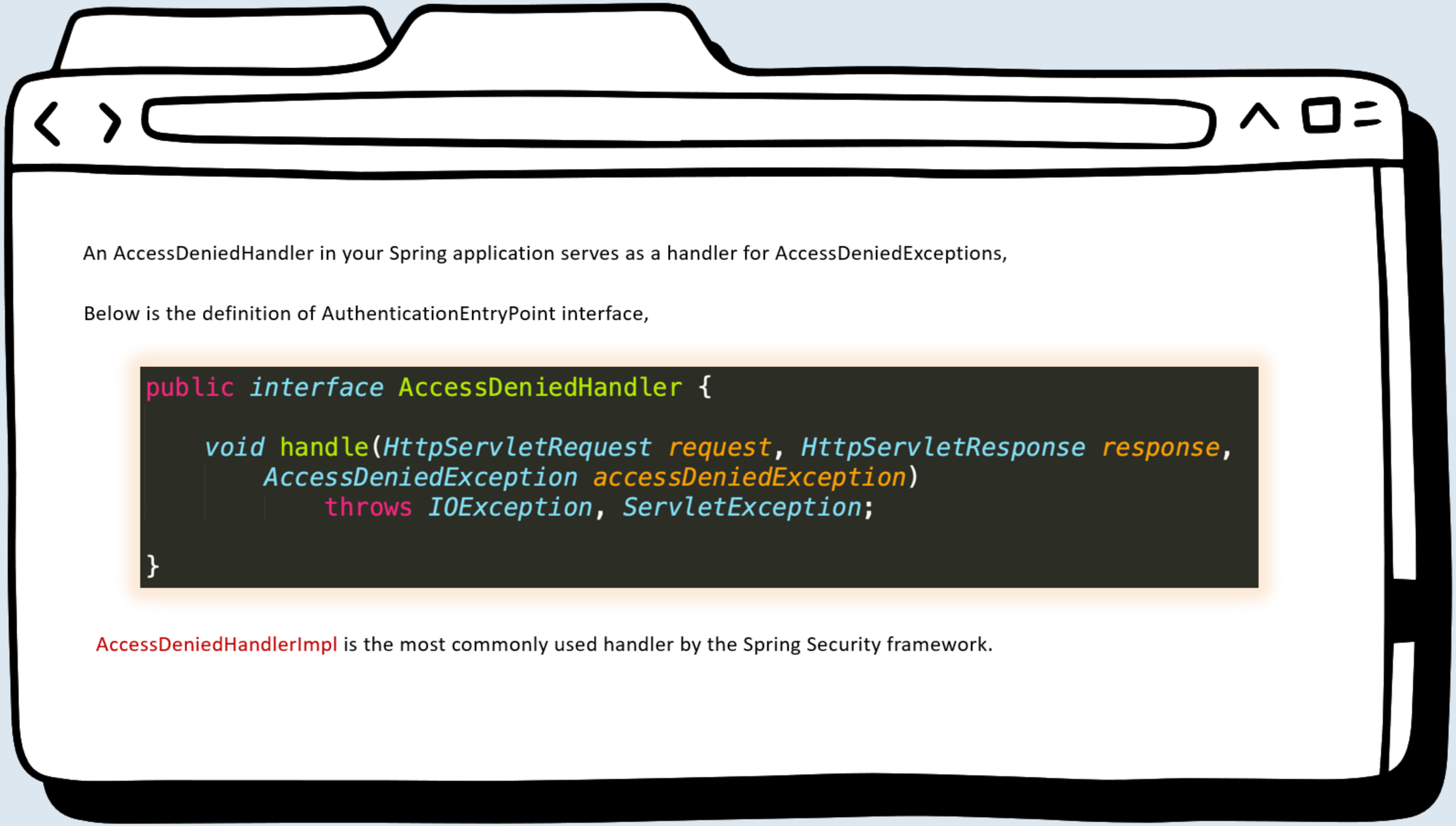
We can also configure custom AuthenticationEntryPoint globally that is common for all the login types using exceptionHandling configurations. Below is the snippet of the same,

```
http.exceptionHandling(ehc -> ehc.authenticationEntryPoint(
    new CustomBasicAuthenticationEntryPoint())); // It is an Global Config
```

AccessDeniedHandler

Inside spring security framework

eazy bytes



An AccessDeniedHandler in your Spring application serves as a handler for AccessDeniedExceptions,

Below is the definition of AuthenticationEntryPoint interface,

```
public interface AccessDeniedHandler {

    void handle(HttpServletRequest request, HttpServletResponse response,
        AccessDeniedException accessDeniedException)
            throws IOException, ServletException;

}
```

AccessDeniedHandlerImpl is the most commonly used handler by the Spring Security framework.

Defining Custom AccessDeniedHandler

Inside spring security framework

eazy bytes

Below are the snippets on how to define a Custom AccessDeniedHandler,

1) We need to create a implementation class using AccessDeniedHandler interface & by overriding the handle() method. Inside the handle() method, we can build logic on what response to send back to clients based on our requirements,

```
public class CustomAccessDeniedHandler implements AccessDeniedHandler {

    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
            AccessDeniedException accessDeniedException)
                throws IOException, ServletException {
        response.setHeader("eazybank-denied-reason", "Authorization failed");
        response.setStatus(HttpStatus.FORBIDDEN.value());
    }

}
```

2) The CustomAccessDeniedHandler created in the above step can be configured using exceptionHandling() like shown below,

```
http.exceptionHandling(ehc -> ehc.accessDeniedHandler
        (new CustomAccessDeniedHandler()));
```

Session Management in Spring Security

Handling Invalid Session

Using the below config, the user can be redirected to the given URL when an invalid session is detected,

```
http.sessionManagement(sessionConfig ->
        sessionConfig.invalidSessionUrl("/invalidSession"))
```

Session Timeout

With the help of the below property, we can configure what is the idle time after which the Spring Security created session should be expired. 20m indicates 20 minutes

```
server.servlet.session.timeout=20m
```

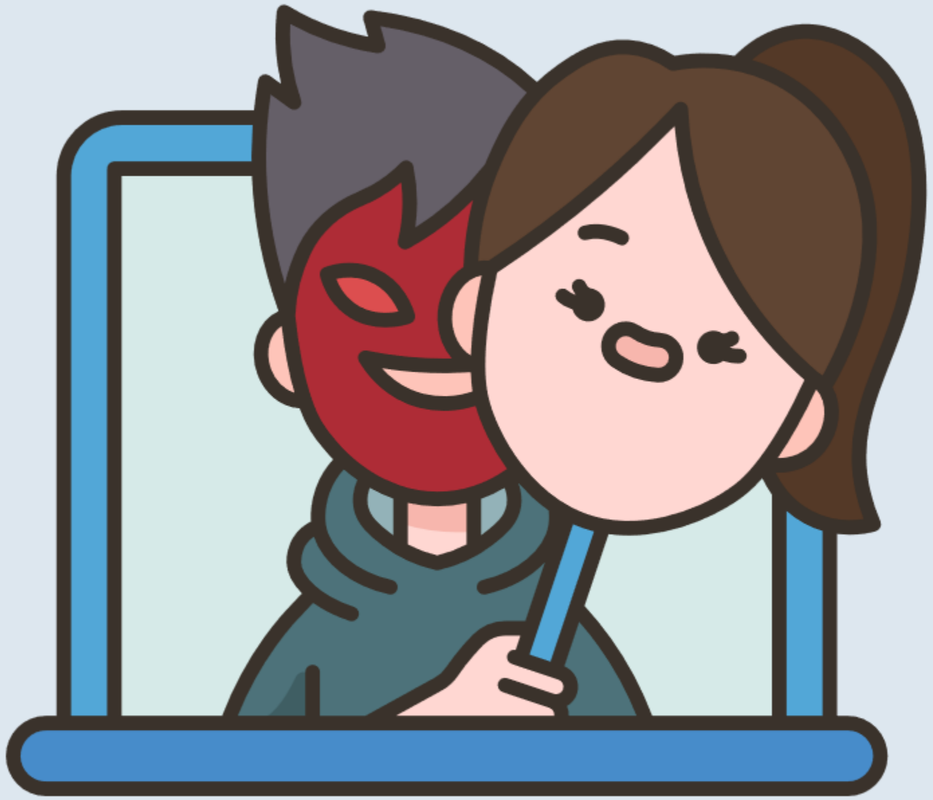
Concurrent Session Control

We can control the maximum sessions allowed for a user and what should happen in the case of invalid session due to too many sessions for the current user,

```
http.sessionManagement(sessionConfig ->
        sessionConfig.maximumSessions(1)
            .maxSessionsPreventsLogin(true)
            .expiredUrl("/expiredSession"))
```

Session Fixation, Session Hijacking Attacks

eazy bytes

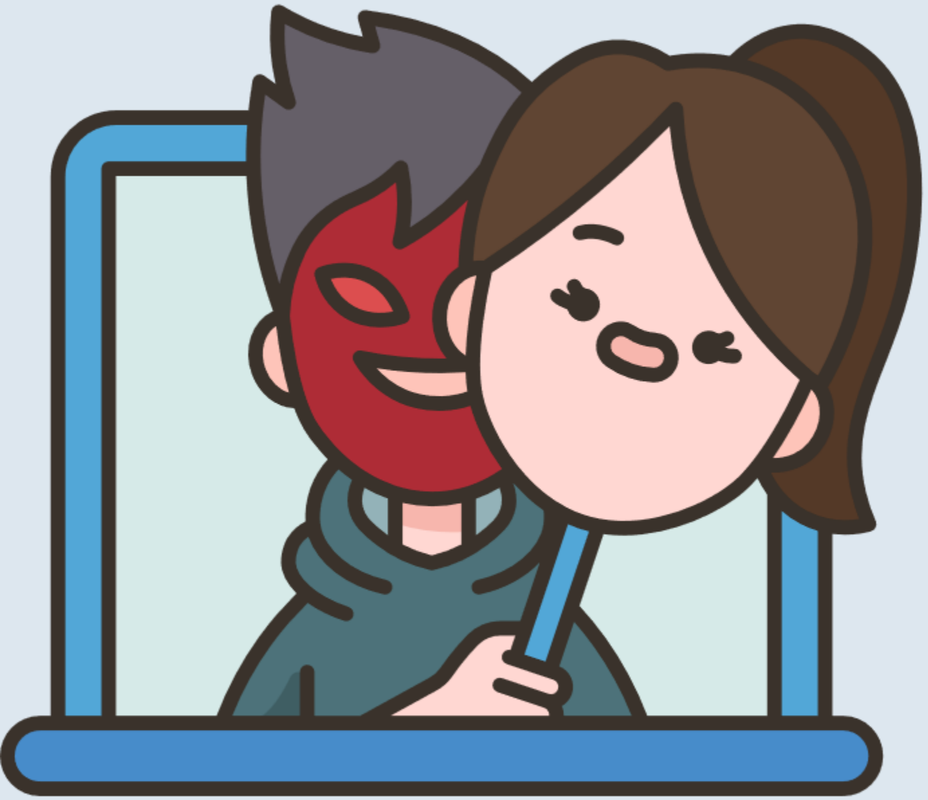


Session fixation attacks are a potential risk where it is possible for a malicious attacker to create a session by accessing a site, then persuade another user to log in with the same session (by sending them a link containing the session identifier as a parameter, for example). Spring Security protects against this automatically by creating a new session or otherwise changing the session ID when a user logs in.

In a session hijacking attack, the attacker steals the session IDs for a web application by eavesdropping on the communication between a user and an application or by gaining access to the user's computer or web browser data.

Session Fixation Attack flow

eazy bytes



Below are the steps that happens during the Session Fixation attack,

1) Emily often shops online at http://fashionmart.com/ where she stores her payment information for quick purchases.

2) Eva, a hacker, discovers a vulnerability in http://fashionmart.com/ that allows her to manipulate session identifiers. So Eva logged into the App with her credentials which will provide a valid session identifier like 34hjhj454h8efop43

3) Eva sends Emily an email pretending to be from the fashion website, saying: "Exciting news! We're offering exclusive discounts to our loyal customers. Click here to access your personalized discount: http://fashionmart.com/?session= 34hjhj454h8efop43

4) Intrigued by the promise of discounts, Emily clicks the link and unknowingly logs into her shoppin account with the session ID " 34hjhj454h8efop43 " provided by Eva.

5) With this fixed session ID, Eva gains unauthorized access to Emily's shopping account and can view her personal information, make purchases using her stored payment details, and potentially steal her identity.

Session Fixation Attack protection with Spring Security

eazy bytes

By default, Spring Security framework takes care of handling the Session Fixation Attacks. We have three options to control the strategy for Session Fixation Protection by following the configurations mentioned in the code snippet,

changeSessionId: This option utilizes the session fixation protection provided by the Servlet container. It does not create a new session but instead changes the session ID. Note that this option is only available in Servlet 3.1 (Java EE 7) and newer containers.

newSession: With this option, a new "clean" session is created, without copying the existing session data. However, Spring Security-related attributes will still be copied.

migrateSession: This option creates a new session and copies all existing session attributes to the new session. It is the default in Servlet 3.0 or older containers.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    http
        .sessionManagement((session) -> session
            .sessionFixation((sessionFixation) -> sessionFixation
                .newSession()
            )
        );
    return http.build();
}
```

We can also set the session fixation protection to none to disable it, but this is not recommended as it leaves your application vulnerable.

Authentication Events

eazy bytes

For each authentication that succeeds or fails, a AuthenticationSuccessEvent or AbstractAuthenticationFailureEvent, respectively, is fired by Spring Security framework. Spring Security provided DefaultAuthenticationEventPublisher takes care of publishing these events to the listeners. We as developers just need to build listeners to listen to these events and execute business logic based on our requirements.

By using Spring's @EventListener support, we can listen to the Authentication success event and failure event. Below is the sample code to log these events,

```
@Component
@Slf4j
public class AuthenticationEvents {

    @EventListener
    public void onSuccess(AuthenticationSuccessEvent success) {
        log.info("Login successful for the user : {}",
            success.getAuthentication().getName());
    }

    @EventListener
    public void onFailure(AbstractAuthenticationFailureEvent failures) {
        log.error("Login failed for the user : {} due to : {}",
            failures.getAuthentication().getName()
            ,failures.getException().getMessage());
    }

}
```

Form Login Configurations for MVC or monolithic apps

eazy bytes

For form based logins, Spring Security already provides a simple login page. But this behaviour can be changed with the following configurations,

```
.formLogin(formLoginConfig -> formLoginConfig
                .loginPage("/login")
                .usernameParameter("userid")
                .passwordParameter("secretPwd")
               .defaultSuccessUrl("/dashboard")
               .failureUrl("/login?error=true").permitAll())
```

loginPage -> Specifies the URL to send users to if login is required. The default is /login

usernameParameter – The HTTP parameter to look for the username when performing authentication. Default is "username"

passwordParameter - The HTTP parameter to look for the password when performing authentication. Default is "password"

defaultSuccessUrl - Specifies where users will be redirected after authenticating successfully if they have not visited a secured page prior to authenticating. This is a shortcut for calling successHandler(AuthenticationSuccessHandler).

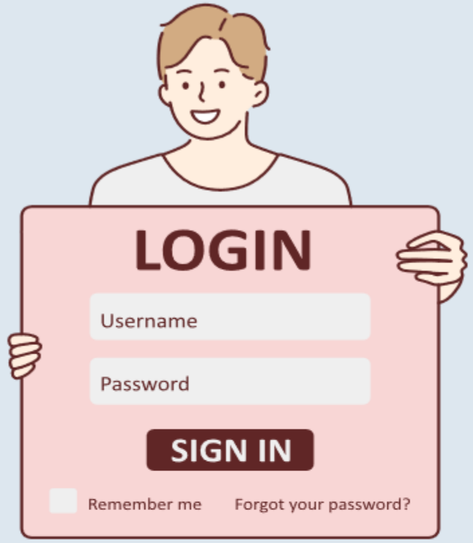
failureUrl - The URL to send users if authentication fails. This is a shortcut for invoking failureHandler(AuthenticationFailureHandler). The default is "/ login?error".

Form Login Configurations for MVC or monolithic apps

eazy bytes

If we want more control on the successful authentication and failure authentication of form login, we can configure the successHandler and failureHandler like shown below,

```
.formLogin(formLoginConfig -> formLoginConfig
                .loginPage("/login")
                .usernameParameter("userid")
                .passwordParameter("secretPwd")
                .successHandler(authenticationSuccessHandler)
                 .failureHandler(authenticationFailureHandler))
```



Here the authenticationSuccessHandler and authenticationFailureHandler are the beans of the implementation classes of AuthenticationSuccessHandler and AuthenticationFailureHandler interfaces

```
public class CustomAuthenticationFailureHandler implements
          AuthenticationFailureHandler {

    @Override
    public void onAuthenticationFailure(HttpServletRequest
          request,HttpServletResponse response,
         AuthenticationException exception) throws IOException {
          ....
    }
}
```

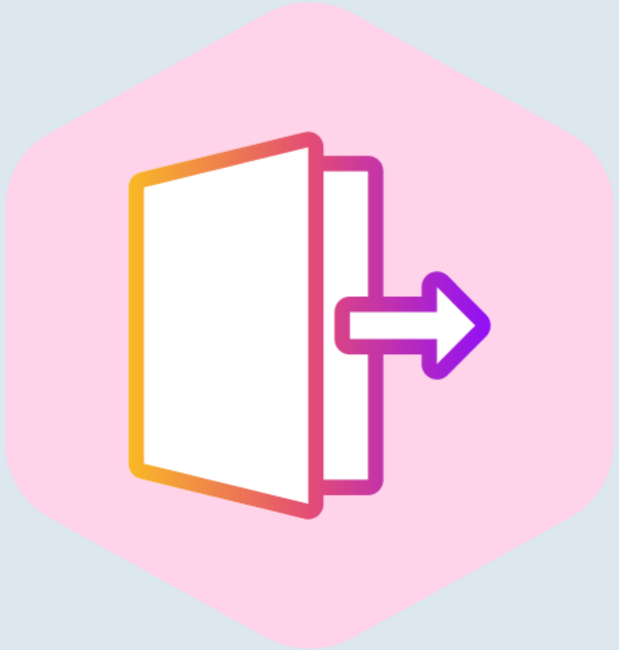
```
public class CustomAuthenticationSuccessHandler implements
          AuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest
          request, HttpServletResponse response,
          Authentication authentication) throws IOException {
        ...
    }
}
```

Logout Configurations for MVC or monolithic apps

eazy bytes

With the below configurations, we can configure what should happen during the logout operation of the user,

```
.logout(logoutConfigurer -> logoutConfigurer
            .logoutSuccessUrl("/login?logout=true").permitAll()
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .deleteCookies("JSESSIONID"))
```



logoutSuccessUrl -> The URL to redirect to after logout has occurred. The default is "/ login?logout".
This is a shortcut for invoking logoutSuccessHandler(LogoutSuccessHandler) with a SimpleUrlLogoutSuccessHandler.

invalidateHttpSession – Configures SecurityContextLogoutHandler to invalidate the HttpSession at the time of logout.

clearAuthentication - Specifies if SecurityContextLogoutHandler should clear the Authentication at the time of logout.

deleteCookies - Allows specifying the names of cookies to be removed on logout success. This is a shortcut to easily invoke addLogoutHandler(LogoutHandler) with a CookieClearingLogoutHandler.

Spring Security Thymeleaf integration

eazy bytes

We can get the currently authenticated user details in Thymeleaf pages by following the below steps,

First, we need to add the below dependency inside pom.xml:

```
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity6</artifactId>
        </dependency>
```

Next, we can load the username like shown below:

```
<html xmlns:th="https://www.thymeleaf.org"
 xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity6">
<body>
    <div sec:authorize="isAuthenticated()">
      Authenticated as <span sec:authentication="name"></span></div>
</body>
</html>
```

Spring Security JSP integration

eazy bytes

We can get the currently authenticated user details in JSP pages by leveraging the Spring Security Taglib support.

First, we need to define the tag on the JSP page like shown below:

```
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
```

Next, we can load the username like shown below:

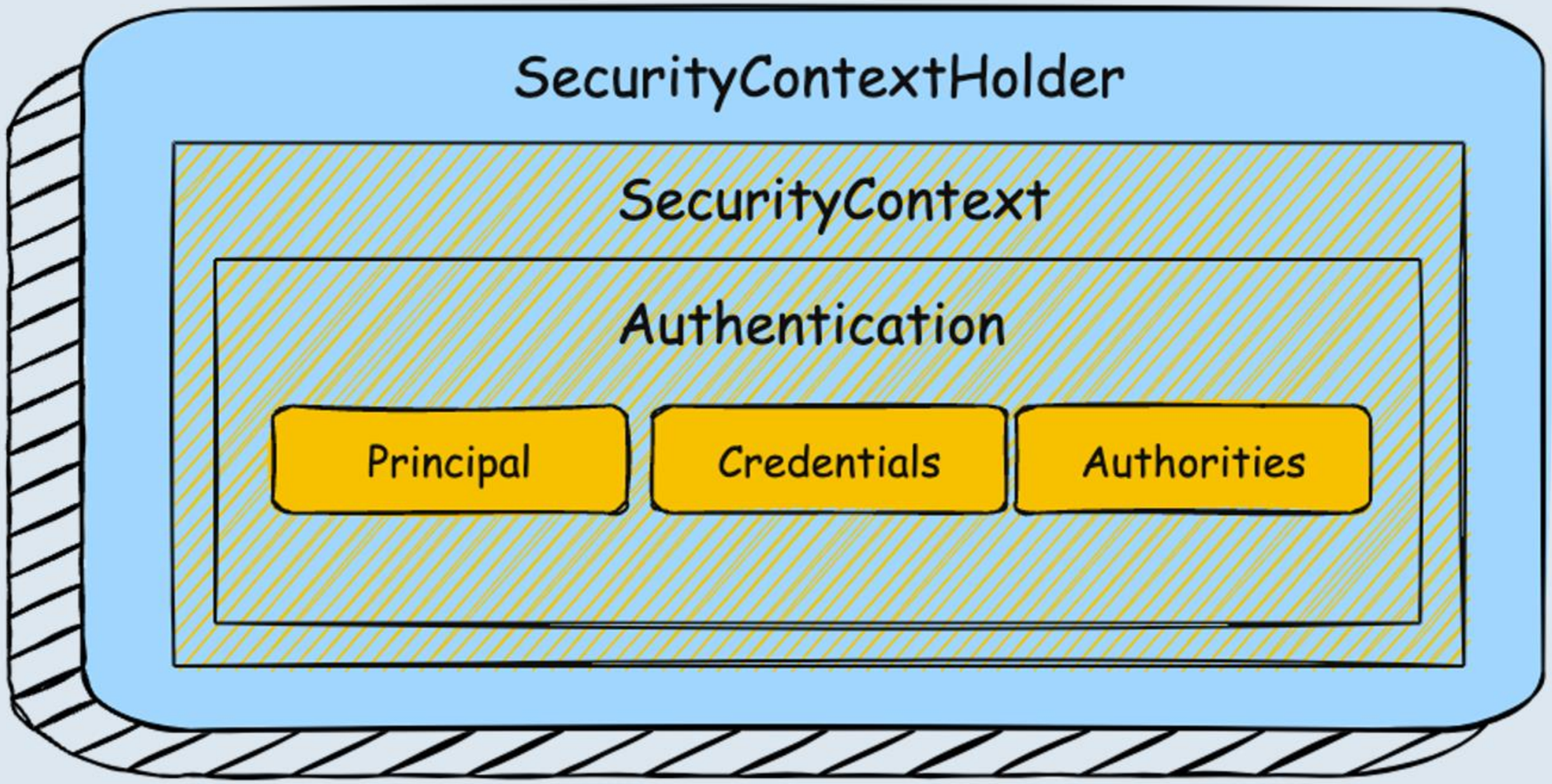
```
<security:authorize access="isAuthenticated()">
    authenticated as <security:authentication property="principal.username" />
</security:authorize>
```

Role of SecurityContext & SecurityContextHolder

eazy bytes

At the heart of Spring Security's authentication model is the SecurityContextHolder. It contains the SecurityContext.

The SecurityContextHolder is where Spring Security stores the details of who is authenticated. Spring Security does not care how the SecurityContextHolder is populated. If it contains a value, it is used as the currently authenticated user.



Role of SecurityContext & SecurityContextHolder

eazy bytes

After the authentication process, you may need details about the authenticated entity, such as the username or authorities of the current user. This information remains accessible even after authentication is complete. Once the AuthenticationManager successfully completes the authentication process, it stores the Authentication instance for the rest of the request. The component responsible for storing this object is called the **SecurityContext**.

SecurityContext is a interface defining the minimum security information associated with the current thread of execution.

```
public interface SecurityContext {

    Authentication getAuthentication();

    void setAuthentication(Authentication authentication);

}
```

The security context is stored in a **SecurityContextHolder**.

SecurityContextHolder is a helper class that provides access to the security context. By default, it uses a ThreadLocal object to store the security context, ensuring it is always available to methods in the same thread, even if you don't explicitly pass it around. You don't need to worry about ThreadLocal memory leaks in web applications, as Spring Security handles the cleanup.

Additionally, SecurityContextHolder can be configured with different strategies for storing the security context.

Holding strategies for the security context

eazy bytes

To specify which strategy should be used, you must provide a mode setting. Below are the three valid mode settings,

MODE_THREADLOCAL—Allows each thread to store its own details in the security context. In a thread-per-request web application, this is a common approach, as each request has an individual thread. This is the default mode

MODE_INHERITABLETHREADLOCAL—Similar to MODE_THREADLOCAL, this strategy also instructs Spring Security to copy the security context to the next thread in the case of an asynchronous method. This ensures that the new thread running the @Async method inherits the security context.
The @Async annotation is used on methods to instruct Spring to execute the annotated method on a separate thread.

MODE_GLOBAL—Makes all the threads of the application see the same security context instance.

As shown in the following code snippet, you can change the strategy to MODE_GLOBAL. You can use the method SecurityContextHolder.setStrategyName() or the system property spring.security.strategy:

```
@Bean
public InitializingBean initializingBean() {
        return () -> SecurityContextHolder.setStrategyName(
                SecurityContextHolder.MODE_GLOBAL);
}
```

Load login user details in Spring Security

eazy bytes

Approach 1 — The simplest way to load the currently authenticated user details is via a static call to the SecurityContextHolder:

```
@RestController
public class SomeController {

    @GetMapping(value = "/username")
    public String currentUserName() {
       Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
       String currentPrincipalName = authentication.getName();
       return currentPrincipalName;
    }
}
```

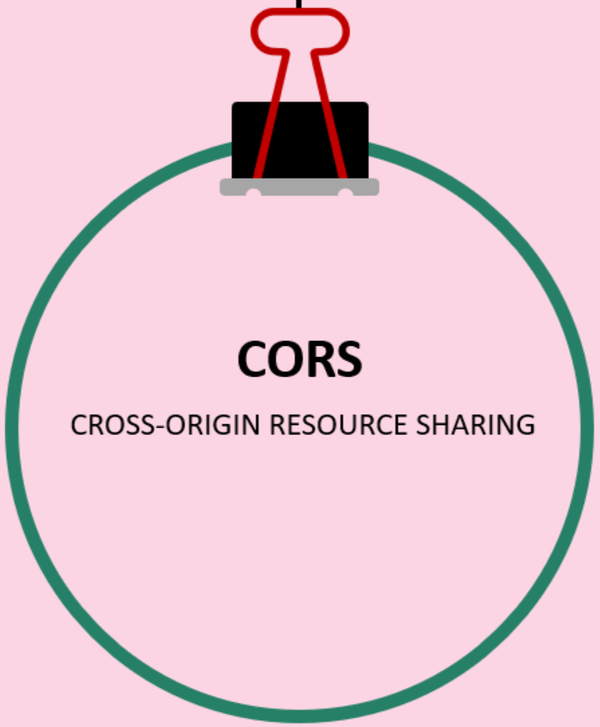
Approach 2 — We can mention the Authentication directly as a method argument, and the framework will correctly resolve it by extracting from the SecurityContextHolder. Instead of Authentication, we can also mention Principal.

```
@RestController
public class SomeController {

    @GetMapping(value = "/username")
    public String currentUserName(Authentication authentication) {
       return authentication.getName();
    }
}
```

CORS & CSRF

SPRING SECURITY APPROACH

eazy bytes



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)

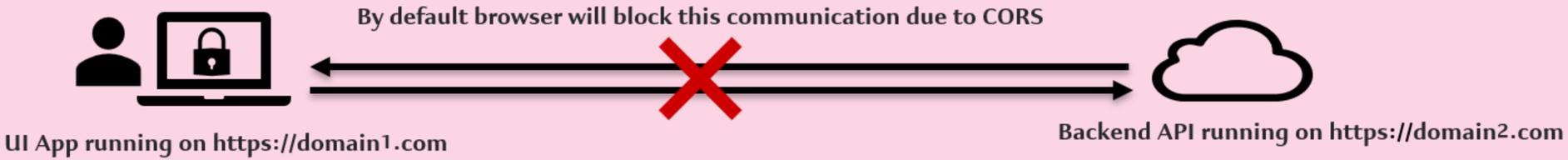
eazy bytes

CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin. 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. It is a specification from W3C implemented by most browsers.

So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

"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



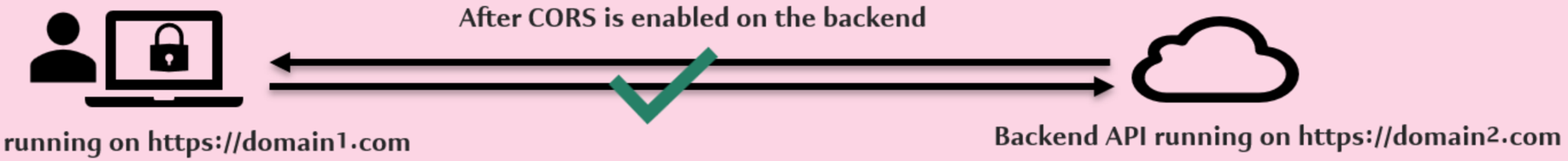
SOLUTION TO HANDLE CORS

eazy bytes

If we have a valid scenario, where a Web APP UI deployed on a server is trying to communicate with a REST service deployed on another server, then these kind of communications we can allow with the help of @CrossOrigin annotation. @CrossOrigin allows clients from any domain to consume the API.

@CrossOrigin annotation can be mentioned on top of a class or method like mentioned below,

```
@CrossOrigin(origins = "http://localhost:4200") // Will allow on specified domain

@CrossOrigin(origins = "*") // Will allow any domain
```



SOLUTION TO HANDLE CORS

eazy bytes

Instead of mentioning @CrossOrigin annotation on all the controllers inside our web app, we can define CORS related configurations globally using Spring Security like shown below,

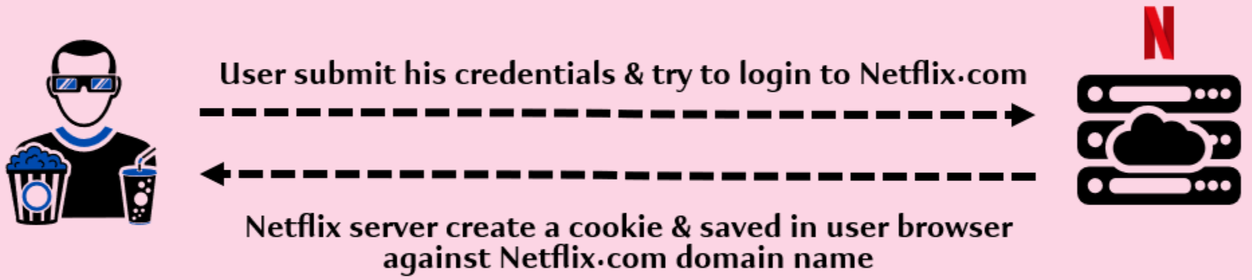
```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.cors(corsCustomizer -> corsCustomizer.configurationSource(new CorsConfigurationSource() {
            @Override
            public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
                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;
            }
        })).authorizeHttpRequests((requests)->requests
                .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
                .requestMatchers("/notices", "/contact", "/register").permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();
}
```

CROSS-SITE REQUEST FORGERY (CSRF)

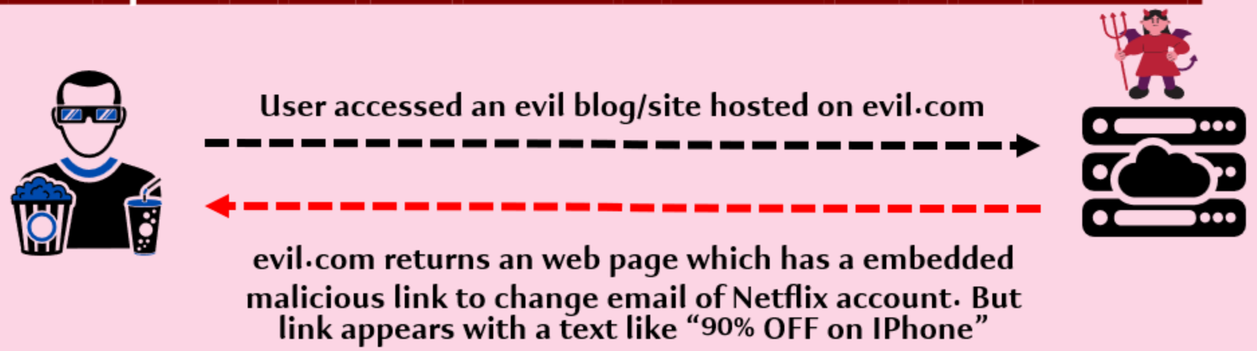
eazy bytes

- 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.
- Consider you are using a website netflix.com and the attacker's website evil.com

Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com



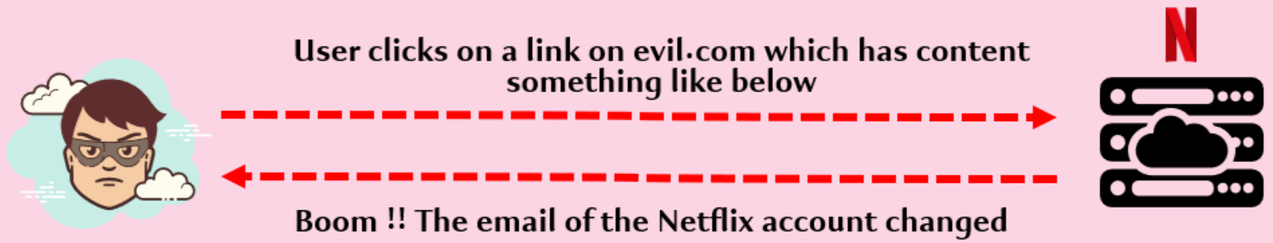
Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.



CROSS-SITE REQUEST FORGERY (CSRF)

eazy bytes

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com, the backend server of Netflix.com can't differentiate from where the request came. So here the evil.com forged the request as if it is coming from a Netflix.com UI page.




```
<form action="https://netflix.com/changeEmail"
  method="POST" id="form">
    <input type="hidden" name="email" value="user@evil.com">
<form>


<script>
    document.getElementById('form').submit()
</script>
```

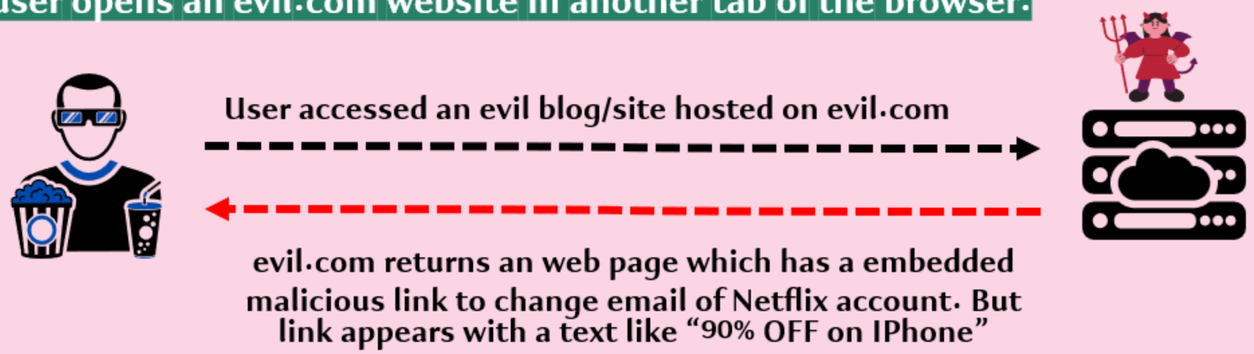
SOLUTION TO CSRF ATTACK

eazy bytes

- To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. The best way to achieve this is through a CSRF token. A CSRF token is a secure random token that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.
- Let's see how this solve CSRF attack by taking the previous Netflix example again,

Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session.

Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.



SOLUTION TO CSRF ATTACK

eazy bytes

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com. This time the Netflix.com backend server expects CSRF token along with the cookie. The CSRF token must be same as initial value generated during login operation

The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.

DISABLE CSRF PROTECTION

INSIDE SPRING SECURITY

eazy bytes

By default Spring Security block all HTTP POST, PUT, DELETE, PATCH operations with an error of 403, if there is no CSRF solution implemented insdie a web application. We can change this default behaviour by disabling the CSRF protection provided by Spring Security.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf((csrf) -> csrf.disable())
            .authorizeHttpRequests((requests)->requests
                    .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
                    .requestMatchers("/notices", "/contact", "/register").permitAll())
            .formLogin(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults());
    return http.build();
}
```

CSRF ATTACK SOLUTION

INSIDE SPRING SECURITY

eazy bytes

With the given configuration of Spring Security, we can let the framework to generate a random CSRF token which can be sent to UI after successful login. The same taken need to be sent by UI for every subsequent requests it is making to backend. For certain paths, we can disable CSRF with the help of ignoringRequestMatchers.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
    requestHandler.setCsrfRequestAttributeName("_csrf");

    http.securityContext((context) -> context
                .requireExplicitSave(false))
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
        .cors(corsCustomizer -> corsCustomizer.configurationSource(new CorsConfigurationSource() {...}))
        .csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler).ignoringRequestMatchers("/contact", "/register")
                .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
                .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
        .authorizeHttpRequests((requests)->requests
                .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
                .requestMatchers("/notices", "/contact", "/register").permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();
}
```

CSRF ATTACK SOLUTION

INSIDE SPRING SECURITY

eazy bytes

Below are the important components that handles CSRF attacks in Spring Security,

CsrfToken - Provides the information/contract about an expected CSRF token.

CsrfTokenRepository – Provides the contract to create, store, and load CSRF tokens. HttpSessionCsrfTokenRepository(stores the CsrfToken in the HttpSession), CookieCsrfTokenRepository (persists the CSRF token in a cookie named "XSRF-TOKEN" and reads from the header "X-XSRF-TOKEN" following the conventions of Angular) are the typically used implementation classes.

CsrfTokenRequestHandler – A callback interface that is used to make the CsrfToken created by the CsrfTokenRepository available as a request attribute. Implementations of this interface may choose to perform additional tasks or customize how the token is made available to the application through request attributes. CsrfTokenRequestAttributeHandler is the typically used implementation classes.

CsrfFilter – Spring Security in built filter responsible to validate the CSRF token received by client and to decide whether to throw an exception or not.

CsrfCookieFilter – Custom filter written by developer to send the generated CSRF token back to client inside the response header

AUTHENTICATION & AUTHORIZATION

DETAILS & COMPARISION

eazy bytes

AUTHENTICATION

- In authentication, the identity of users are checked for providing the access to the system.
- Authentication(AuthN) done before authorization
- It needs usually user's login details
- If authentication fails usually we will get 401 error response
- For example as a Bank customer/employee in order to perform actions in the app, we need to first prove our identity

AUTHORIZATION

- In authorization, person's or user's authorities are checked for accessing the resources.
- Authorization(AuthZ) always happens after authentication.
- It needs user's privilege or roles
- If authorization fails usually we will get 403 error response
- Once logged into the application, my roles, authorities will decide what kind of actions I can do

HOW AUTHORITIES STORED ?

INSIDE SPRING SECURITY

eazy bytes

- Authorities/Roles information in Spring Security is stored inside GrantedAuthority. There is only one method inside GrantedAuthority which return the name of the authority or role.
- SimpleGrantedAuthority is the default implementation class of GrantedAuthority interface inside Spring Security framework.

```
public interface GrantedAuthority {

    String getAuthority();

}
```

```
public final class SimpleGrantedAuthority implements GrantedAuthority {

    private final String role;

    public SimpleGrantedAuthority(String role) {
        this.role = role;
    }

    @Override
    public String getAuthority() {
        return this.role;
    }

}
```

HOW AUTHORITIES STORED ?

INSIDE SPRING SECURITY

eazy bytes

How does Authorities information stored inside the objects of UserDetails & Authentication interfaces which plays a vital role during authentication of the user ?

Authentication
(Interface)

UsernamePasswordAuthenticationToken
(Class)

getAuthorities()

UserDetails
(Interface)

User
(Class)

getAuthorities()

CONFIGURING AUTHORITIES

INSIDE SPRING SECURITY

eazy bytes

In Spring Security the authorities requirements can be configured 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 invoke 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 invoke 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.

CONFIGURING AUTHORITIES

INSIDE SPRING SECURITY

eazy bytes

You can extract path values from the request, as seen below by using access():

```
http.authorizeHttpRequests((authorize) -> authorize
        .requestMatchers("/cards/{name}").access(new
    WebExpressionAuthorizationManager("#name == authentication.name")))
```

You can check if the user has multiple roles, as seen below by using access():

```
.requestMatchers("/admin/**").access(allOf(hasAuthority(”admin"),
hasAuthority(”manager"))
```

You can check if the user has multiple roles, as seen below by using access():

```
.requestMatchers("/admin/**").access(new WebExpressionAuthorizationManager("
hasAuthority('admin') && hasAuthority('manager')"))
```

CONFIGURING AUTHORITIES
INSIDE SPRING SECURITY

eazy bytes

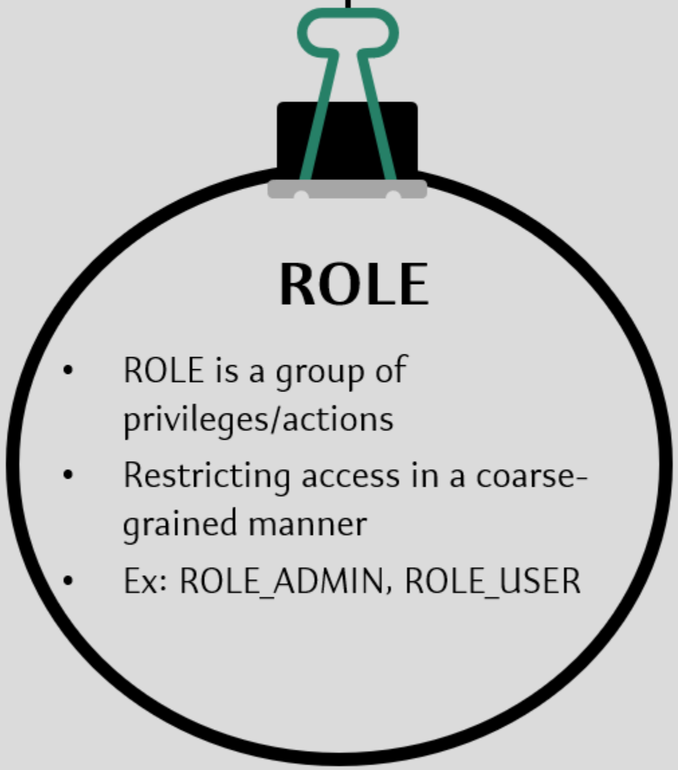
Like shown below, we can configure authority requirements for the APIs/Paths.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
  CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
  requestHandler.setCsrfRequestAttributeName("_csrf");
  http.securityContext((context) -> context.requireExplicitSave(false))
      .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
      .cors(corsCustomizer -> corsCustomizer.configurationSource(new CorsConfigurationSource() {...}))
      .csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler).ignoringRequestMatchers("/contact","/register")
       .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
      .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
      .authorizeHttpRequests((requests)->requests
          .requestMatchers("/myAccount").hasAuthority("VIEWACCOUNT")
          .requestMatchers("/myBalance").hasAnyAuthority("VIEWACCOUNT","VIEWBALANCE")
          .requestMatchers("/myLoans").hasAuthority("VIEWLOANS")
          .requestMatchers("/myCards").hasAuthority("VIEWCARDS")
          .requestMatchers("/user").authenticated()
          .requestMatchers("/notices","/contact","/register").permitAll())
      .formLogin(Customizer.withDefaults())
      .httpBasic(Customizer.withDefaults());
  return http.build();
}
```

AUTHORITY vs ROLE

INSIDE SPRING SECURITY

eazy bytes



- **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.**

AUTHORITY vs ROLE

INSIDE SPRING SECURITY

eazy bytes

You can configure the authorization rules to use a different prefix by providing a GrantedAuthorityDefaults bean, as shown below:

```
@Bean
static GrantedAuthorityDefaults grantedAuthorityDefaults() {

    return new GrantedAuthorityDefaults("MYPREFIX_");

}
```

You need to expose GrantedAuthorityDefaults using a static method to ensure that Spring publishes it before initializing Spring Security's method security @Configuration classes

CONFIGURING ROLES

INSIDE SPRING SECURITY

eazy bytes

In Spring Security the ROLES requirements can be configured 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 invoke 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.**

CONFIGURING ROLES

INSIDE SPRING SECURITY

eazy bytes

Like shown below, we can configure ROLES requirements for the APIs/Paths.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
    requestHandler.setCsrfRequestAttributeName("_csrf");
    http.securityContext((context) -> context.requireExplicitSave(false))
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
            .cors(corsCustomizer -> corsCustomizer.configurationSource(new CorsConfigurationSource() {...}))
            .csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler).ignoringRequestMatchers("/contact","/register")
             .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
            .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
            .authorizeHttpRequests((requests)->requests
                    .requestMatchers("/myAccount").hasRole("USER")
                    .requestMatchers("/myBalance").hasAnyRole("USER","ADMIN")
                    .requestMatchers("/myLoans").hasRole("USER")
                    .requestMatchers("/myCards").hasRole("USER")
                    .requestMatchers("/user").authenticated()
                    .requestMatchers("/notices","/contact","/register").permitAll())
            .formLogin(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults());
    return http.build();
}
```

AUTHORIZATION EVENTS

INSIDE SPRING SECURITY

eazy bytes

For each denied authorization, an `AuthorizationDeniedEvent` is triggered. You can listen these events, using the below kind of logic,

```
@EventListener
public void onFailure(AuthorizationDeniedEvent deniedEvent) {
    log.error("Authorization failed for the user : {} due to : {}",
       deniedEvent.getAuthentication().get().getName(),
             deniedEvent.getAuthorizationDecision().toString());
}
```

Due to the potential for AuthorizationGrantedEvents to generate excessive noise, they are not published by default.

FILTERS IN SPRING SECURITY

eazy bytes

- ✓ 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
- ✓ 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 UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, DefaultLoginPageGeneratingFilter etc. 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

eazy bytes

- ✓ 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

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

```
Security filter chain: [
  DisableEncodeUrlFilter
  ForceEagerSessionCreationFilter
  ChannelProcessingFilter
  WebAsyncManagerIntegrationFilter
  SecurityContextPersistenceFilter
  HeaderWriterFilter
  CorsFilter
  CsrfFilter
  LogoutFilter
  UsernamePasswordAuthenticationFilter
  DefaultLoginPageGeneratingFilter
  DefaultLogoutPageGeneratingFilter
  BasicAuthenticationFilter
  CsrfCookieFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  AuthorizationFilter
]
```

IMPLEMENTING CUSTOM FILTERS

INSIDE SPRING SECURITY

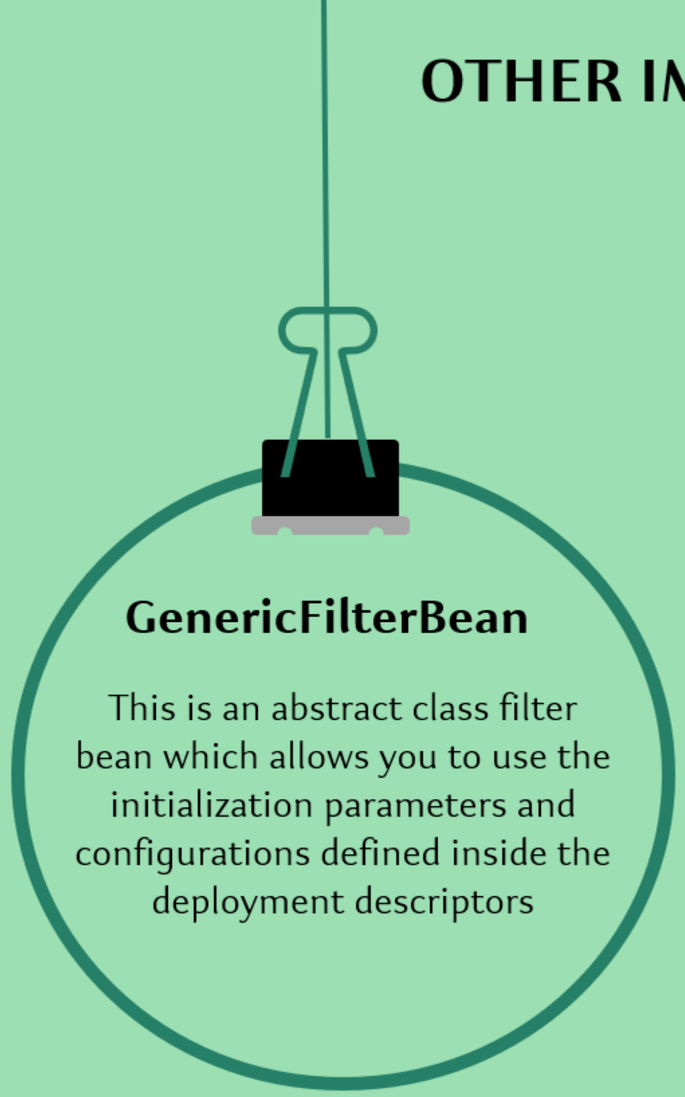
eazy bytes

- ✓ We can create our own filters by implementing the Filter interface from the jakarta.servlet package. Post that we need to override the doFilter() method to have our own custom logic. This method accepts 3 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

OTHER IMPORTANT FILTERS

eazy bytes



GenericFilterBean

This is an abstract class filter bean which allows you to use the initialization parameters and configurations defined inside the deployment descriptors

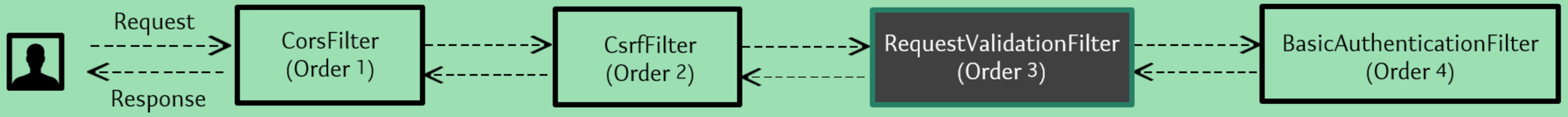
OncePerRequestFilter

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.

ADD FILTER BEFORE
IN SPRING SECURITY

eazy bytes

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

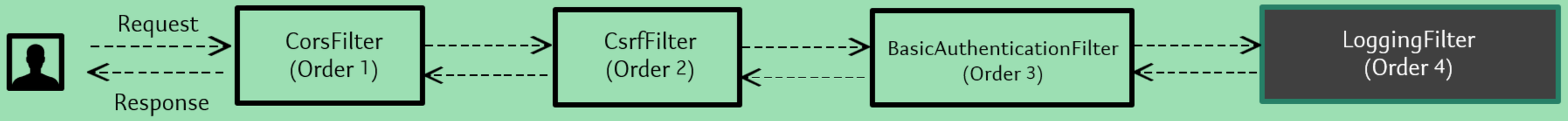


Here we add a filter just before authentication to write our own custom validation where the input email provided should not have the string 'test' inside it.

ADD FILTER AFTER
IN SPRING SECURITY

eazy bytes

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

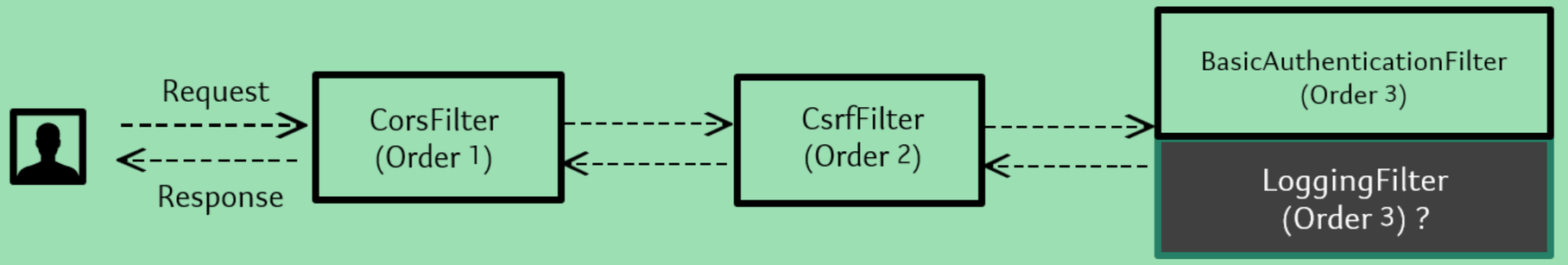


Here we add a filter just after authentication to write a logger about successful authentication and authorities details of the logged in users.

ADD FILTER AT
IN SPRING SECURITY

eazy bytes

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.

TOKEN FORMATS

eazy bytes

Token formats

In general, there are two token formats.

1) Opaque Tokens
2) JSON Web Tokens (JWT)

Opaque Tokens

Opaque tokens are typically random strings with no inherent meaning.

Used to reference authentication information stored on the server-side.

Requires a call to the authorization server or a dedicated introspection endpoint to validate and obtain user information.

JWT Tokens

JWTs are self-contained tokens that consist of three parts: a header, a payload, and a signature, encoded in Base64 URL.

Encodes user information and claims directly within the token.

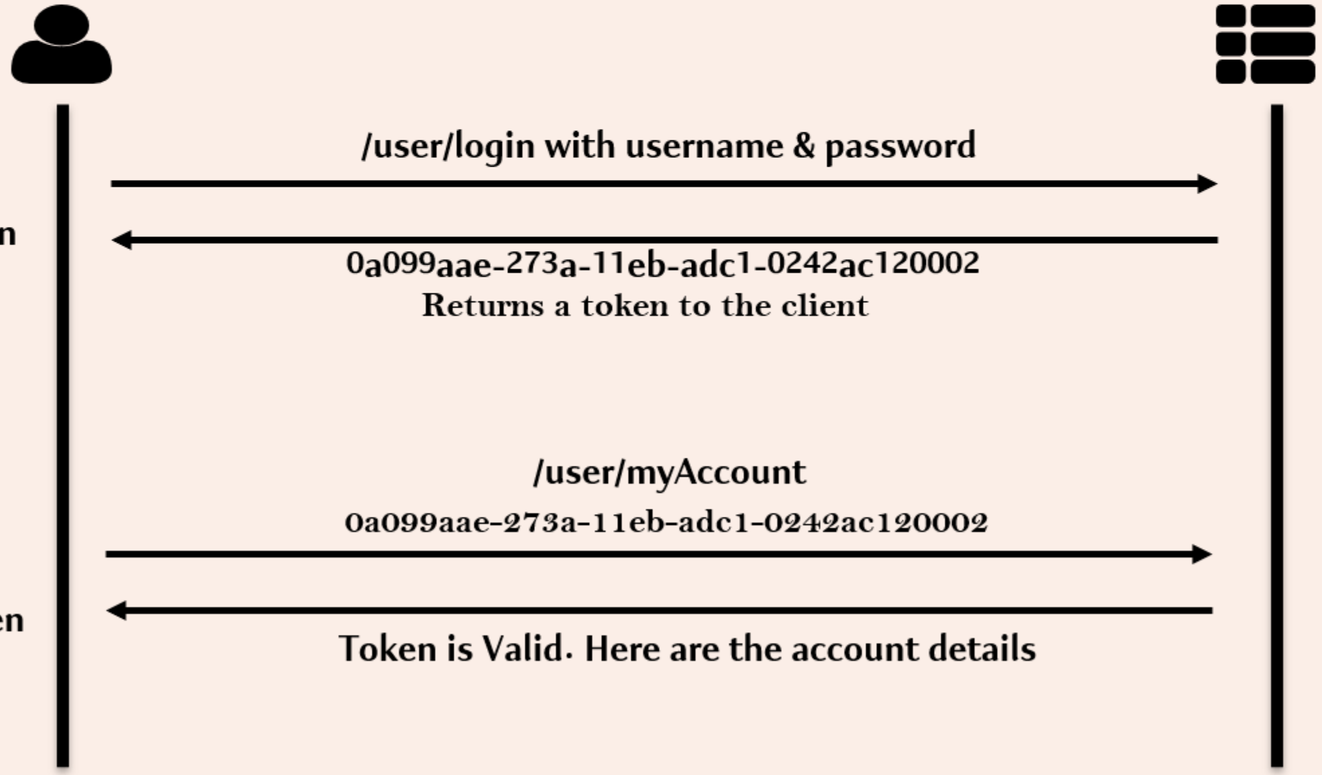
Can be validated locally by verifying the token's signature using a public key, without needing a server call.

Opaque tokens are suitable for scenarios where token validation by a central server is feasible, such as within a secure internal network where as JWT tokens are ideal for stateless, distributed systems where quick token validation is needed without frequent server calls.

ROLE OF TOKENS
IN AUTHN & AUTHZ

eazy bytes

- ✓ 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 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

- 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.

eazy bytes

Token Advantages

Security

- Limited expose of the user credentials inside the network
- Tokens can be revoked during any suspicious activities without invalidating the user credentials.

Reusability

- Tokens can be used across different domains and services, making them suitable for single sign-on (SSO) systems.

Expiration

- Tokens can have specific expiration times set, ensuring tokens are valid only for a predefined duration.

Cross-Platform compatibility

- Tokens can be used across various platforms and devices, including web applications, mobile apps, and IoT devices.

Self-contained

- Tokens are self-contained and carry all the necessary information about the user, roles/authorities etc.

Statelessness

- 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.

JWT TOKENS

eazy bytes

- ✓ 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 client/server side.

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

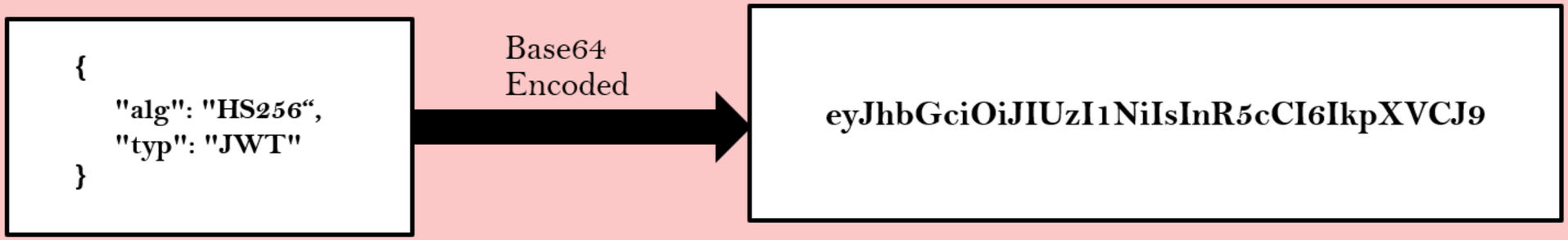
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

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

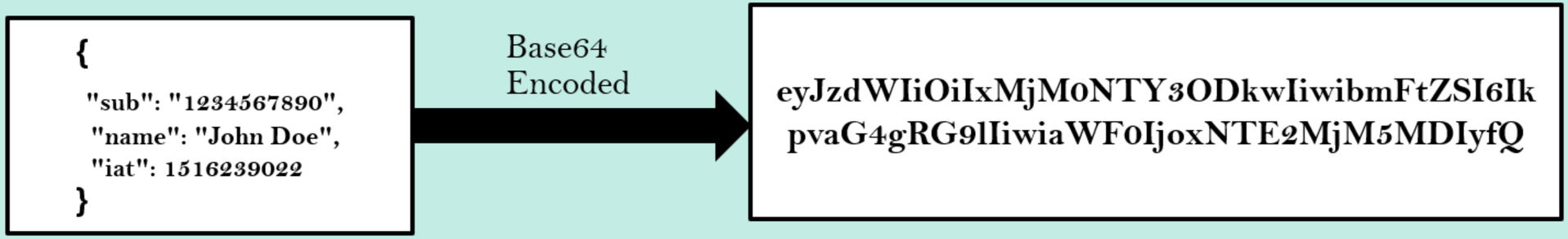
JWT TOKENS

eazy bytes

- ✓ Inside the JWT header, we store metadata/info related to the token. If we chose to sign the token, the header contains the name of the algorithm that generates the signature.



- ✓ 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 much we can send in the body, but we should put our best efforts to keep it as light as possible.



JWT TOKENS

eazy bytes

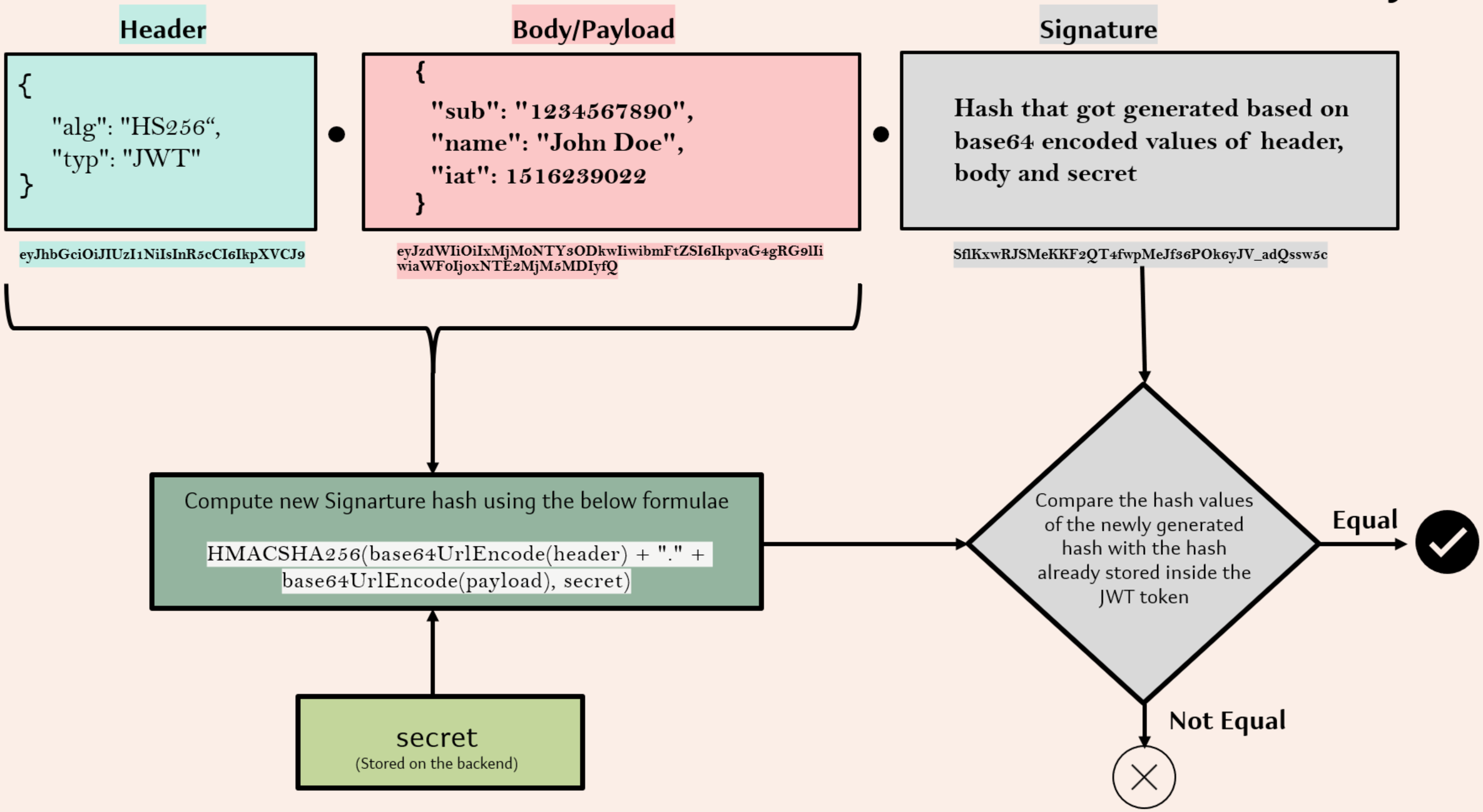
- ✓ The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.
- ✓ But if you are sharing this token to the client applications which will be used by all the users in 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.

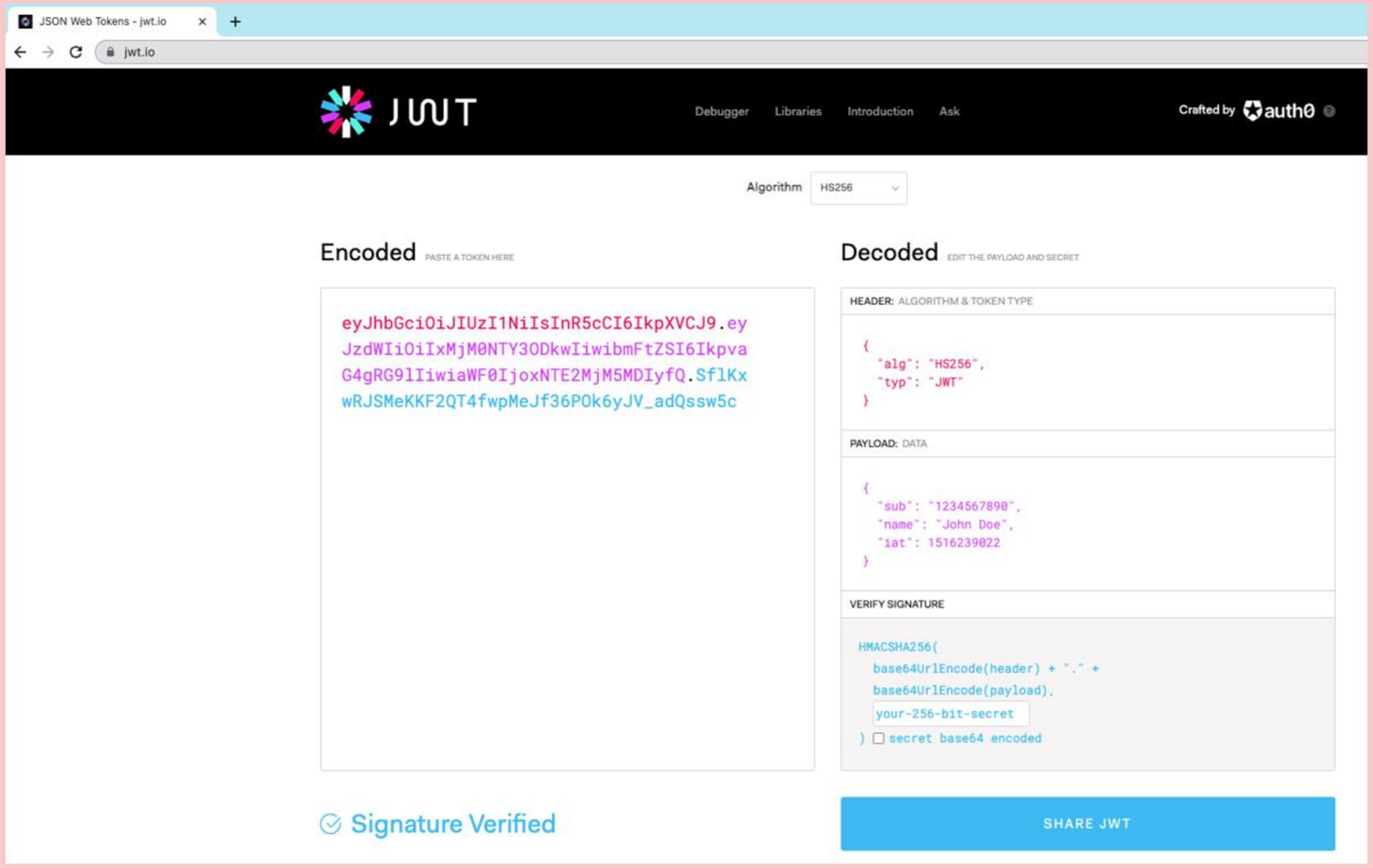
eazy bytes


Header
{
"alg": "HS256“,
"typ": "JWT"
}
Body/Payload
{
"sub": "1234567890",
"name": "John Doe",
"iat": 1516239022
}
Signature
Hash that got generated based on base64 encoded values of header, body and secret
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
Compute new Signarture hash using the below formulae
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
secret
(Stored on the backend)
Compare the hash values of the newly generated hash with the hash already stored inside the JWT token
Equal
Not Equal

JWT TOKENS

eazy bytes

- ✓ If you want to play with JWT tokens and put these concepts into practice, you can use jwt.io debugger to decode, verify, and generate JWTs.



Publish an AuthenticationManager bean for custom authentication

eazy bytes

A common requirement is to publish an `AuthenticationManager` bean to enable custom authentication, such as within a `@Service` or a Spring MVC `@Controller`. For instance, you might want to authenticate users via a REST API instead of using form login. You can publish an `AuthenticationManager` for custom authentication scenarios using the following configuration:

Inside ProjectSecurityConfig.java class create a bean of AuthenticationManager like shown below

```
@Bean
public AuthenticationManager authenticationManager(
        UserDetailsService userDetailsService,
        PasswordEncoder passwordEncoder) {
    EazyBankUsernamePwdAuthenticationProvider authenticationProvider =
            new EazyBankUsernamePwdAuthenticationProvider(userDetailsService, passwordEncoder);
    ProviderManager providerManager = new ProviderManager(authenticationProvider);
    providerManager.setEraseCredentialsAfterAuthentication(false);
    return providerManager;
}
```

Request and Response related record files,

```
public record LoginRequest(String username, String password) {

}

public record LoginResponse(String status, String jwtToken) {

}
```

Publish an AuthenticationManager bean for custom authentication

eazy bytes

Inside controller class add the logic to manually authenticate and generate a token if the authentication is successful. Also make sure to permitAll and add CSRF ignoringRequestMatchers for the /apiLlogin API inside ProjectSecurityConfig class,

```
@PostMapping("/apiLogin")
public ResponseEntity<LoginResponse> apiLogin(@RequestBody LoginRequest loginRequest) {
    String jwt = null;
    Authentication authenticationRequest =
            UsernamePasswordAuthenticationToken.unauthenticated(
                    loginRequest.username(), loginRequest.password());
    Authentication authentication =
            this.authenticationManager.authenticate(authenticationRequest);
    if (null != authentication && authentication.isAuthenticated()) {
        SecretKey key = Keys.hmacShaKeyFor(SecurityConstants.JWT_KEY.getBytes(StandardCharsets.UTF
         jwt = Jwts.builder().issuer("Eazy Bank").subject("JWT Token")
                .claim("username", authentication.getName())
                .claim("authorities", populateAuthorities(authentication.getAuthorities()))
                .issuedAt(new java.util.Date())
                .expiration(new java.util.Date((new java.util.Date()).getTime() + 30000000))
                .signWith(key).compact();
    }
    return ResponseEntity.status(HttpStatus.OK).header(SecurityConstants.JWT_HEADER,jwt).
            body(new LoginResponse(HttpStatus.OK.getReasonPhrase(),jwt));
}
```

METHOD LEVEL SECURITY

eazy bytes

- ✓ 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 @EnableMethodSecurity 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 can receive back from the method post business logic execution.

METHOD LEVEL SECURITY

eazy bytes

- ✓ 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 which enabled by default enables Spring Security **@PreAuthorize** & **@PostAuthorize** annotations
  - The **securedEnabled** property enables **@Secured** annotation
  - The **jsr250Enabled** property enables **@RoleAllowed** annotation

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

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

METHOD LEVEL SECURITY

eazy bytes

- 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 LoansService {

    @PreAuthorize("hasAuthority('VIEWLOANS')")
    @PreAuthorize("hasRole('ADMIN')")
    @PreAuthorize("hasAnyRole('ADMIN','USER')")
    @PreAuthorize("#username == authentication.principal.username")
    public Loan getLoanDetails(String username) {
        return loansRepository.loadLoanDetailsByUserName(username);
    }

}
```

METHOD LEVEL SECURITY

eazy bytes

- For applying postauthorization rules below is the sample configuration,

```
@Service
public class LoanService {

 @PostAuthorize("hasAuthority('VIEWLOANS')")
 @PostAuthorize("hasRole('ADMIN')")
 @PostAuthorize("hasAnyRole('ADMIN','USER')")
  @PostAuthorize("returnObject.username == authentication.principal.username")
  public Loan getLoanDetails(String username) {
      return loanRepository.loadLoanByUserName(username);
  }
}
```

METHOD LEVEL SECURITY

eazy bytes

- 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 or filtering criteria, then we can consider filtering.

- For filtering the parameters before calling the method we can use PreFilter annotation. But please note that the filterObject should be of type Collection interface.

```
@RestController
public class ContactController {

    @PreFilter("filterObject.contactName != 'Test'")
    public List<Contact> saveContactInquiryDetails(@RequestBody
            List<Contact> contacts)
    {
        // business logic
        return contacts;
    }
}
```

METHOD LEVEL SECURITY

eazy bytes

- For filtering the parameters after executing the method we can use PostFilter annotation. But please note that the filterObject should be of type Collection interface.

```
@RestController
public class ContactController {

	@PostFilter("filterObject.contactName != 'Test'")
	public List<Contact> saveContactInquiryDetails(@RequestBody List<Contact>
		contacts)
	{
		// business logic
		return contacts;
	}
}
```

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

eazy
bytes

OAUTH2

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

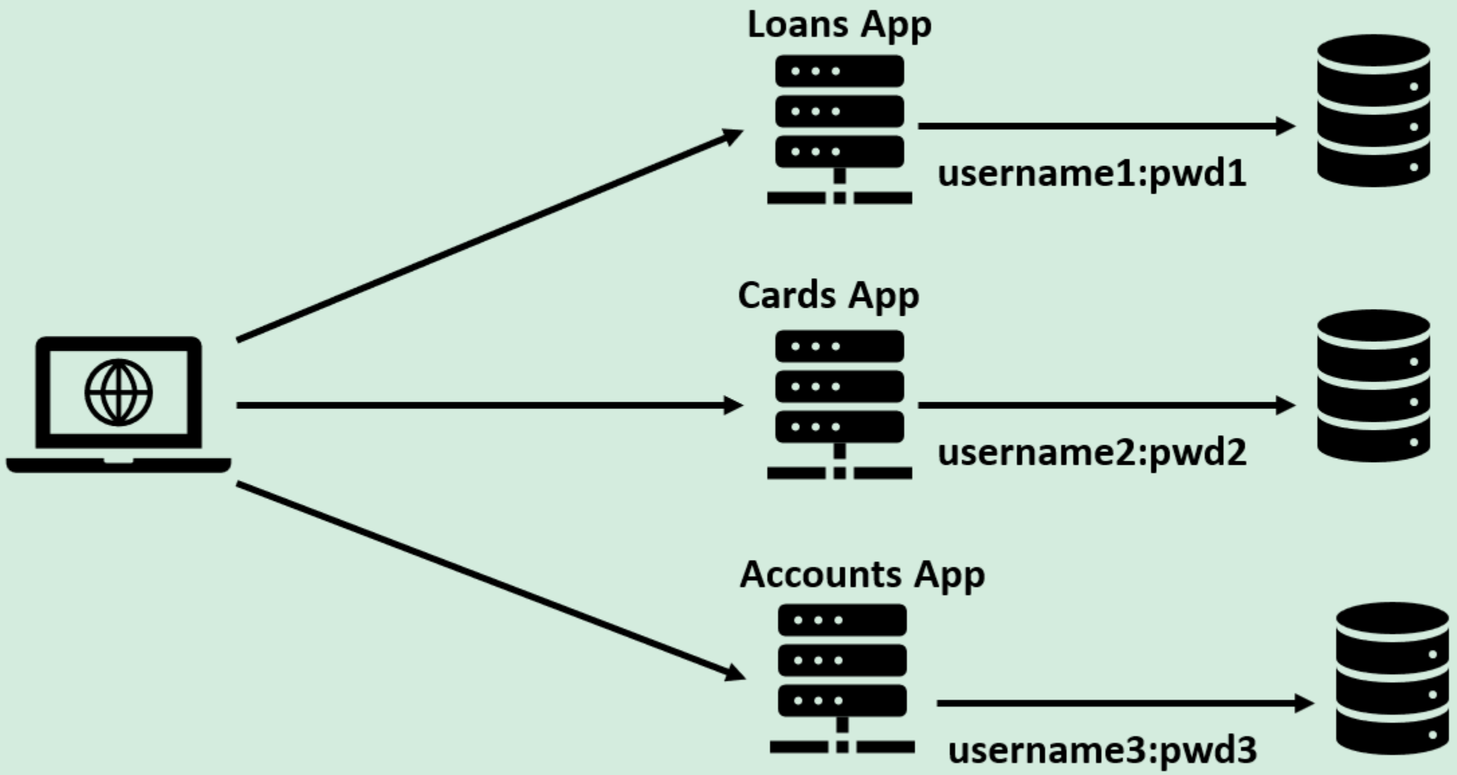
How come, Google let me use the same account in all it's products ? Though they are different websites/ Apps ?

Well the answer is with the help of OAuth2. OAuth2 recommend to use a separate Auth server for Authentication & Authorization

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

eazy bytes



- If a Bank has multiple websites supporting accounts, loans, cards etc. With out OAuth2, the Bank customers has to register and maintain different user profiles all the 3 systems
- Even the AuthN & AuthZ logic, security standards will be duplicated in all the 3 websites.
- Any future changes or enhancements around security, authentication etc. need to done in all the places

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

photos.google.com

Google Photos
End user

PhotoEditor website that allows to edit your phtos, make albums, collage etc.

✓ **Scenario :** The end user wants to use a third-party website called PhotoEditor to edit the photos stored in their Google Photos account.

- **Without OAuth2:** The end user must share their Google account credentials with the PhotoEditor website. Using these credentials, the PhotoEditor website will invoke the Google Photos API to fetch photo details and generate albums, collages, etc.

  However, this approach has a significant disadvantage: PhotoEditor could act fraudulently and perform other operations on the user's behalf, such as changing the password, changing the email address, deleting photos, and more.

- **With OAuth2:** The end user doesn't have to share their Google account credentials with the PhotoEditor website. Instead, they authorize Google to give a temporary access token to PhotoEditor with limited access permissions, such as only reading the image data.

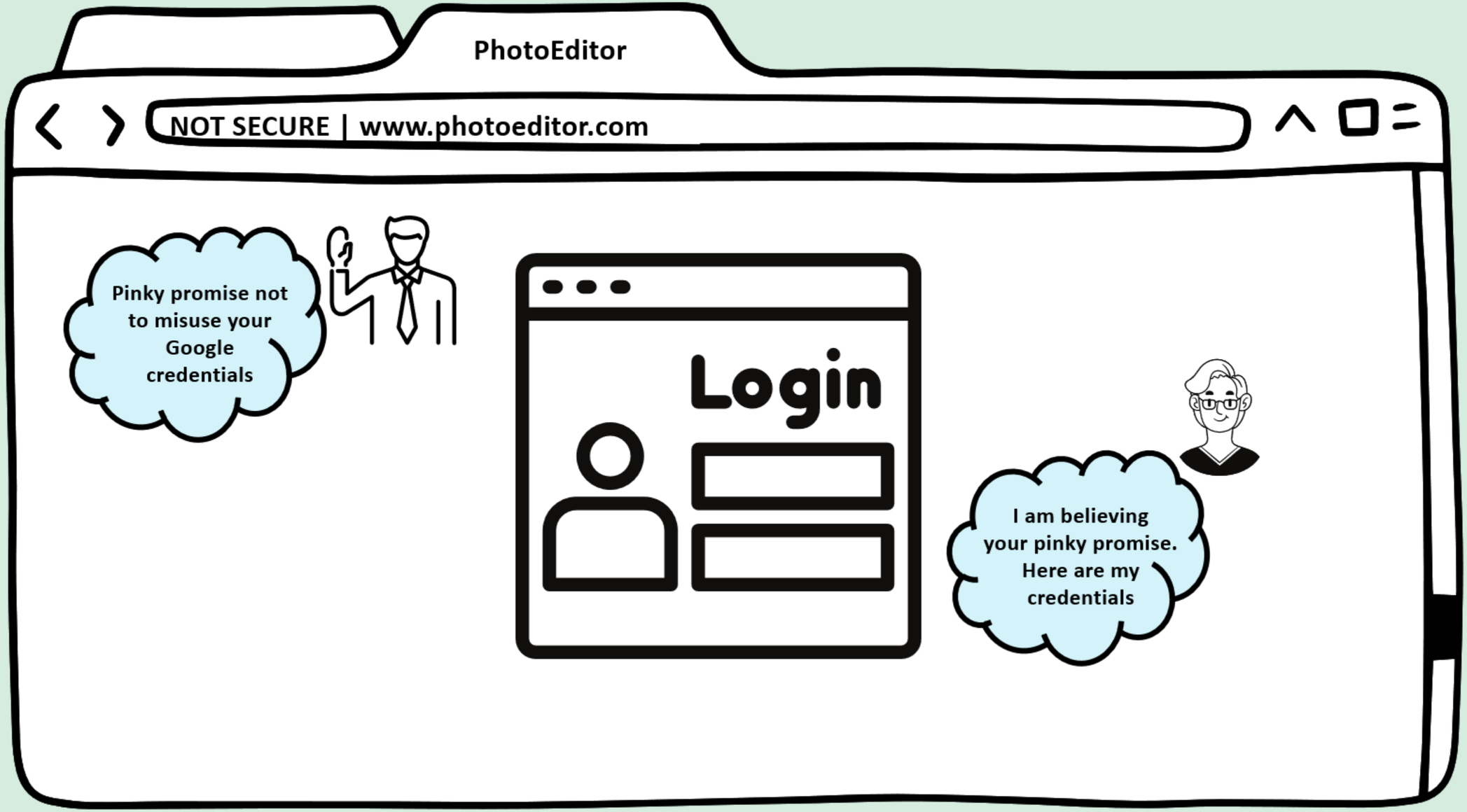
  With this approach, PhotoEditor can only read the image data and cannot perform any other operations on the user's behalf, ensuring a more secure interaction.

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

STONE AGE APPROACH



INTRODUCTION TO OAUTH2

eazy bytes

- ✓ **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 security standard where you give one application permission to access your data in another application. The steps to grant permission, or consent, are often referred to as authorization or even delegated authorization. You authorize one application to access your data, or use features in another application on your behalf, without giving them your password.

- ✓ 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.

- ✓ The OAuth framework specifies several grant types for different use cases, as well as a framework for creating new grant types.

  - **Authorization Code**
  - **PKCE**
  - **Client Credentials**
  - **Device Code**
  - **Refresh Token**
  - **Implicit Flow (Legacy)**
  - **Password Grant (Legacy)**

OAUTH2 TERMINOLOGY

eazy bytes

Resource owner – It is you the end user. In the scenario of **PhotoEditor**, the end user who want to use the **PhotoEditor** website to edit his photos. In other words, the end user owns the resources (photos), that why we call him as Resource owner

Client – The **PhotoEditor** website is the client here as it is the one which interacts with Google after taking permission from the resource owner/end user.

Authorization Server – This is the server which knows about resource owner. In other words, resource owner should have an account in this server. In the scenario of **PhotoEditor,** the Google Auth server which has authorization logic acts as Authorization server.

Resource Server – This is the server where the APIs, services that client want to consume are hosted. In the scenario of **PhotoEditor,** the Google Photos server which has APIs like **/getImages** etc. logic implemented. In smaller organizations, a single server can acts as both resource server and auth server.

Scopes – These are the granular permissions the Client wants, such as access to data or to perform certain actions. In the scenario of **PhotoEditor,** the Auth server can issue an access token to client with the scope of only READ IMAGES.

OAUTH2 SAMPLE FLOW

IN PHOTOEDITOR SCENARIO

eazy bytes

1

The PhotoEditor team will reach out to Google Photos and express their interest in working with them by allowing their users to login with Google credentials.

The Google Photos team collect the details, logo etc. from PhotoEditor and issued a CLIENT ID & CLIENT SECRET

2

The Resource owner visited PhotoEditor website and excited about the idea and decided to use the website. But he has a question, do I need to share my Google account credentials to this website ? ☹

The PhotoEditor website has a button saying “Signup with Google”. The end user clicked on it and boom it has redirect the user to Google login page.

3

The resource owner entered his credentials confidently as it is the login page of the Google itself.

Post successful Authentication, the Google will display a consent page asking the user if he is fine to share his Images data READ ONLY scope to the client which is PhotoEditor app. He said YES

OAUTH2 SAMPLE FLOW

IN PHOTOEDITOR SCENARIO

eazy bytes

4

The Google Authorization server issue an access & refresh tokens to the PhotoEditor website

ACCESS & REFRESH tokens are random complex strings which are tough to guess. Client stores these tokens for future interaction with Google servers

5

The PhotoEditor website sends a request to the Resource server of Google Photos by invoking the API of /getImages along with the access token it received from the Auth server

The Google Photos resource server validates the access token with the Auth server and if it is valid, it return the resource owner images data inside the response

6

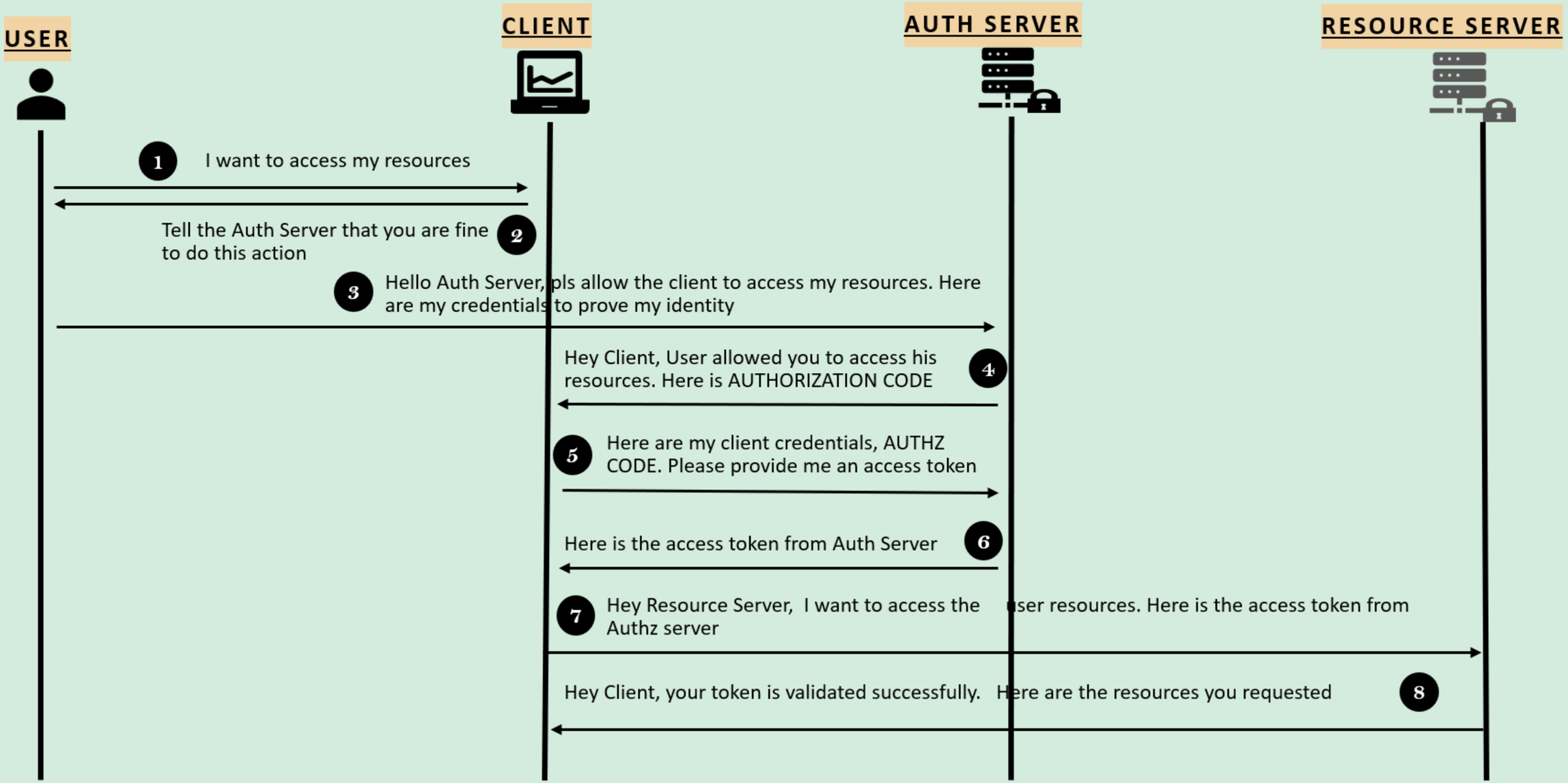
With the data received from the Google Photos, the PhotoEditor website process it and display a album/collage to it's end user

The end user looks at the results and he is super happy that he is able to achieve this with out sharing his Google credentials. THANKS TO OAUTH2 ☺

OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

eazy bytes



OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

eazy bytes

✓ 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

eazy bytes

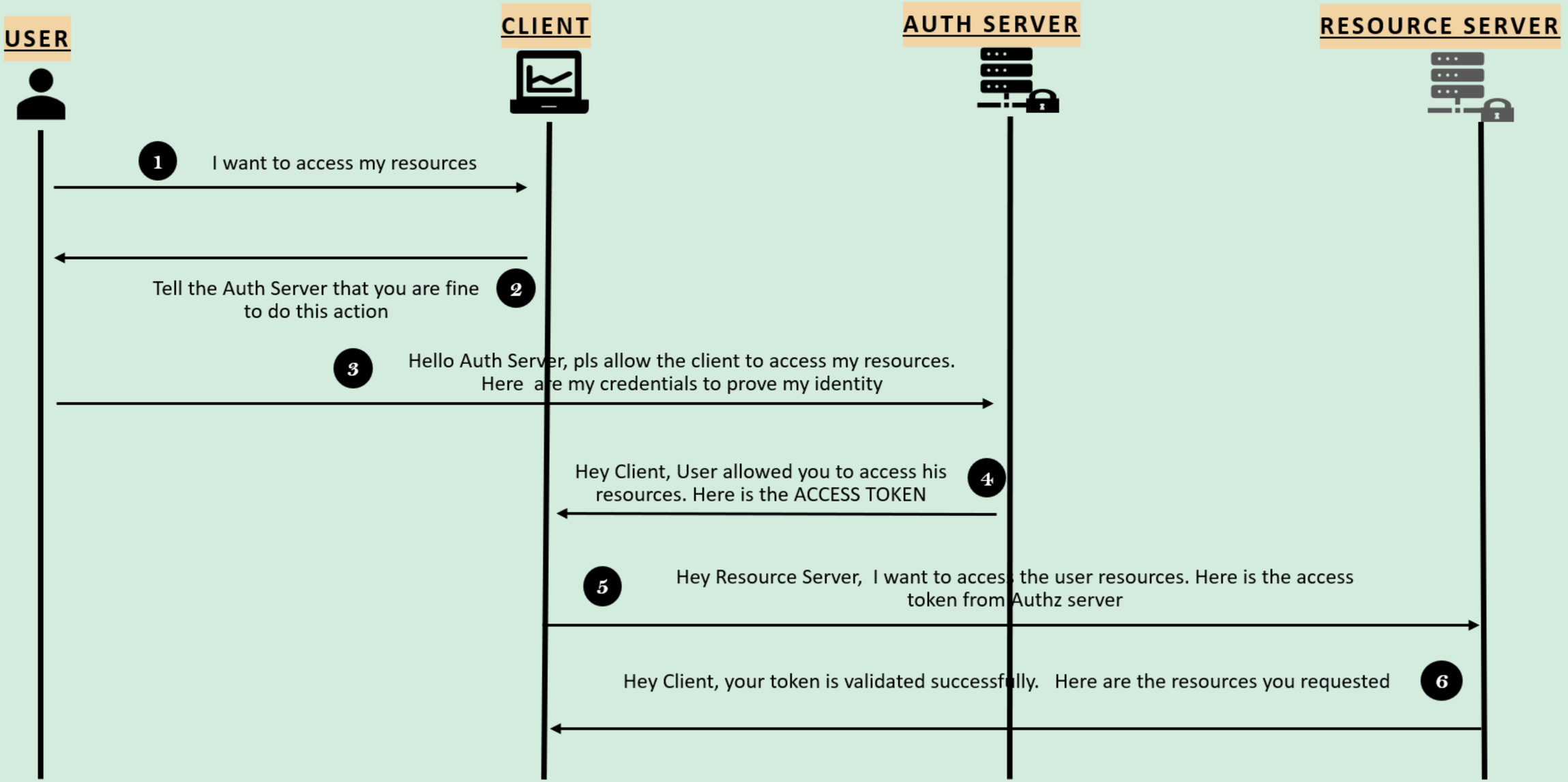
- ✓ We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.
  - In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, 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 is that 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 FLOW

eazy bytes



OAUTH2 FLOW
IN THE IMPLICIT GRANT FLOW

eazy bytes

✓ 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.

✓ Implicit Grant flow is deprecated and is not recommended to use in production applications. Always use the Authorization code grant flow instead of implicit grant flow.

OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

eazy bytes

- ✓ 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).

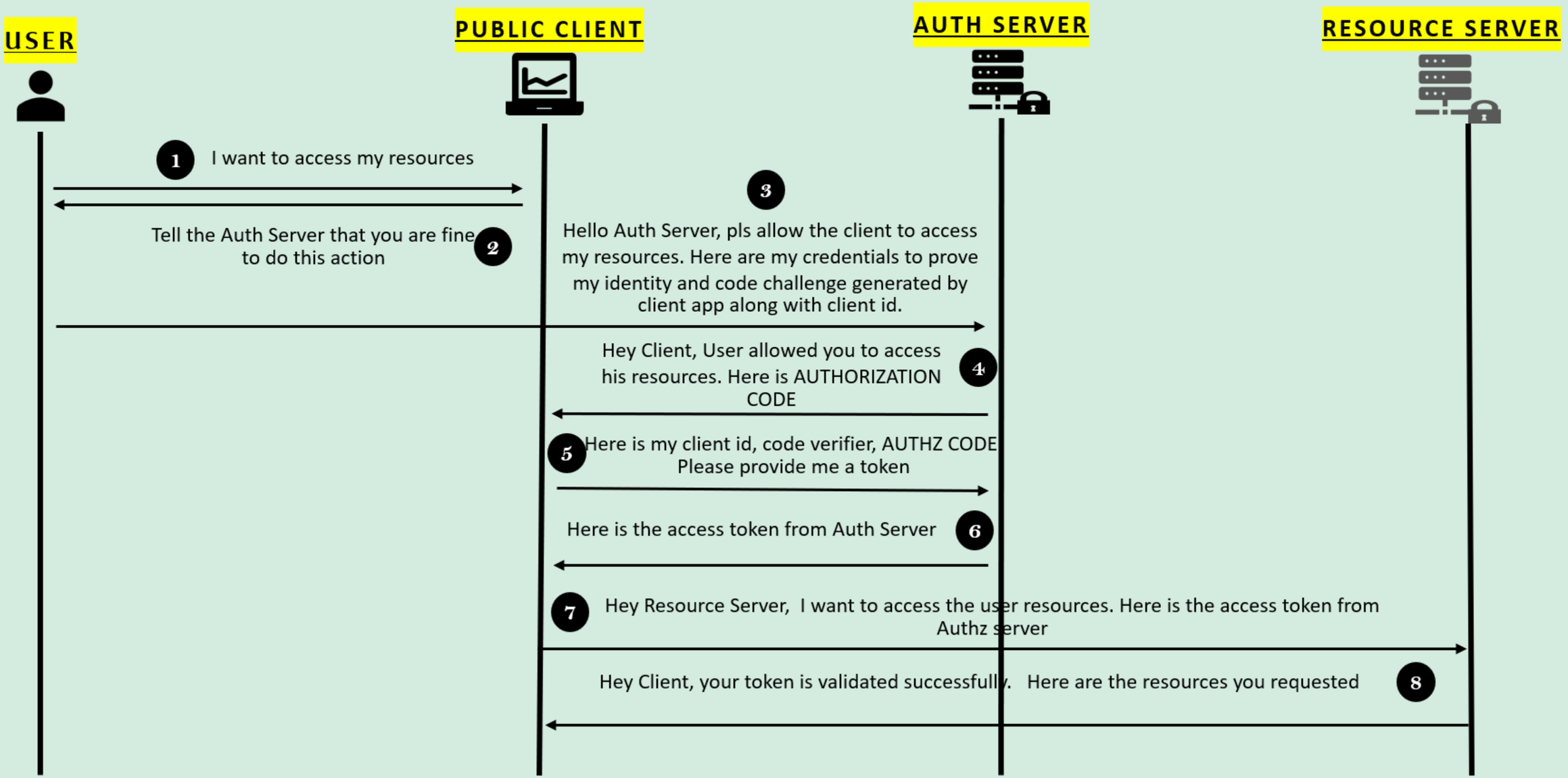
✓ The PKCE-enhanced Authorization Code Flow follows below steps,

- Once user clicks login, client app creates a cryptographically-random code_verifier and from this generates a code_challenge.
- code challenge is a Base64-URL-encoded string of the SHA256 hash of the code verifier.
- 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 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

eazy bytes



OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

eazy bytes

✓ 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
- code_challenge - XXXXXXXXX – The code challenge generated as previously described
- code_challenge_method - S256 (either plain or S256)

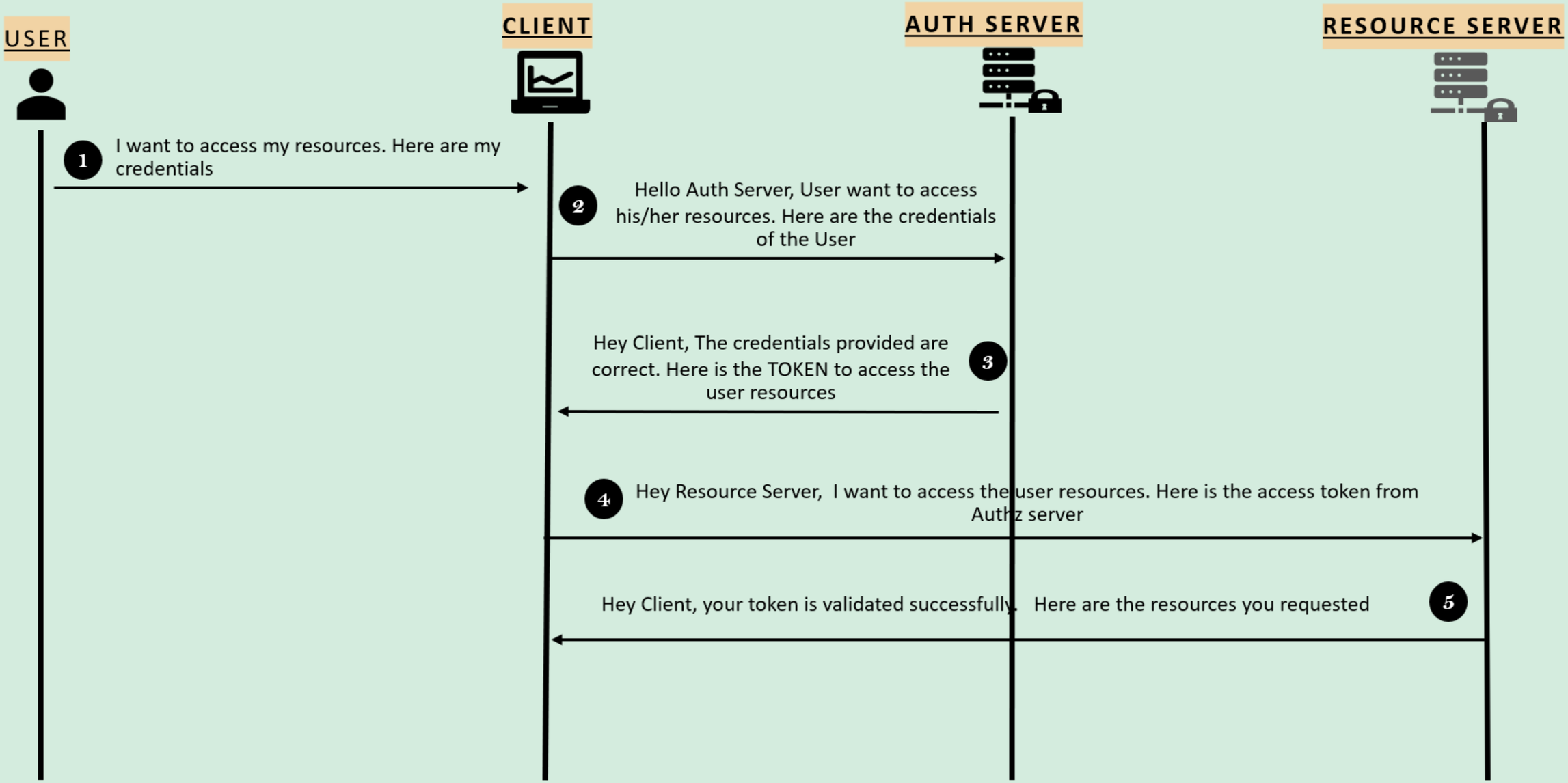
✓ In the step 5 where client after received a authorization code from Auth server, it will again make a request to Auth server for a token with the below values,

- code – the authorization code received from the above steps
- client_id & client_secret (optional) – 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
- code_verifier – The code verifier for the PKCE request, that the app originally generated before the authorization request.

OAUTH2 FLOW

IN THE PASSWORD GRANT/RESOURCE OWNER CREDENTIALS GRANT TYPE

eazy bytes



OAUTH2 FLOW

IN THE RESOURCE OWNER CREDENTIALS GRANT TYPE

eazy bytes

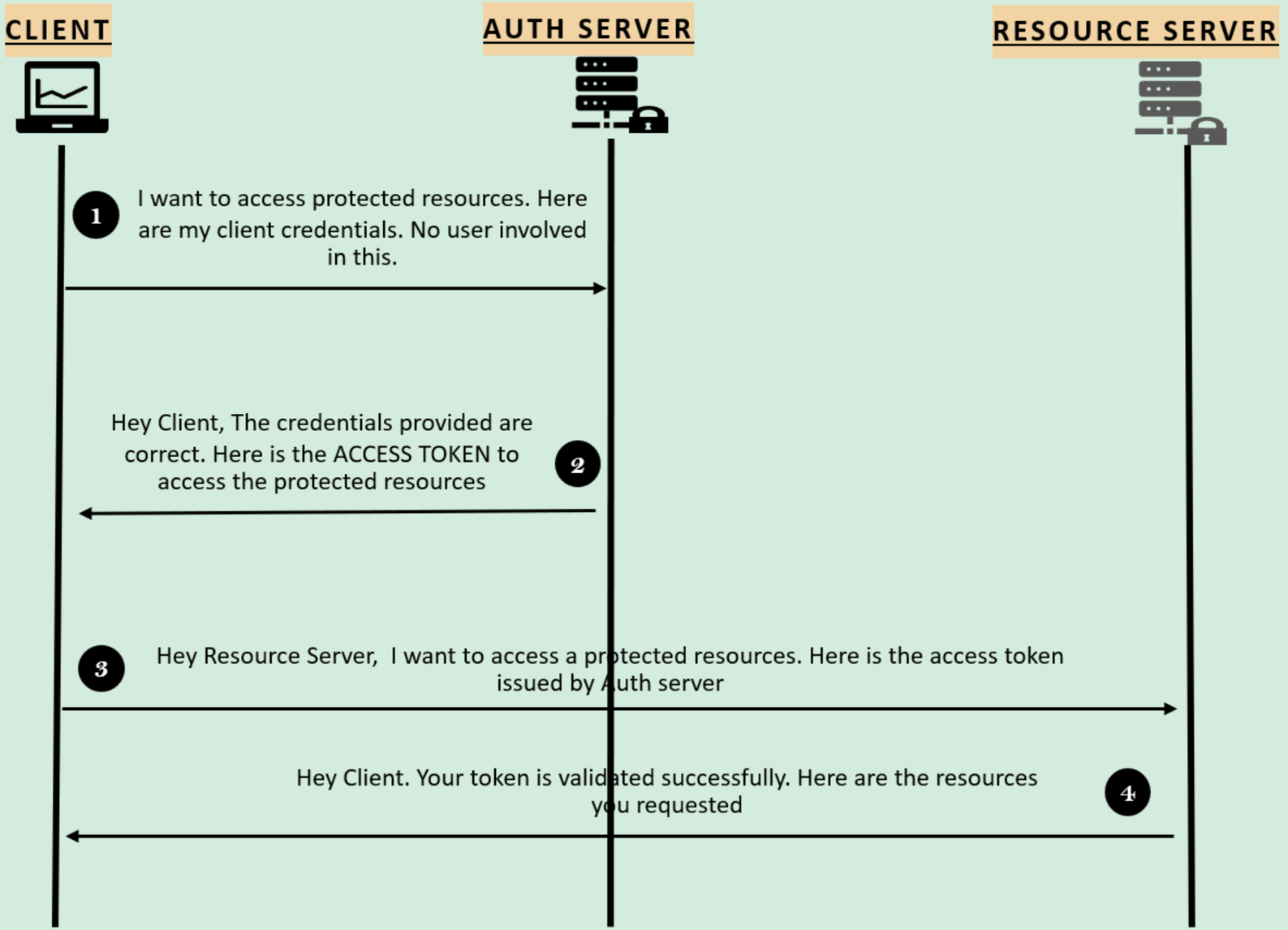
- ✓ In the step 2, where client is making a request to Auth Server endpoint have to send the below important details,
  - **client_id** & **client_secret** – the credentials of the client to authenticate itself.
  - **scope** – similar to authorities. Specifies level of access that client is requesting 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.

OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE

eazy bytes



OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE

eazy bytes

- ✓ 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

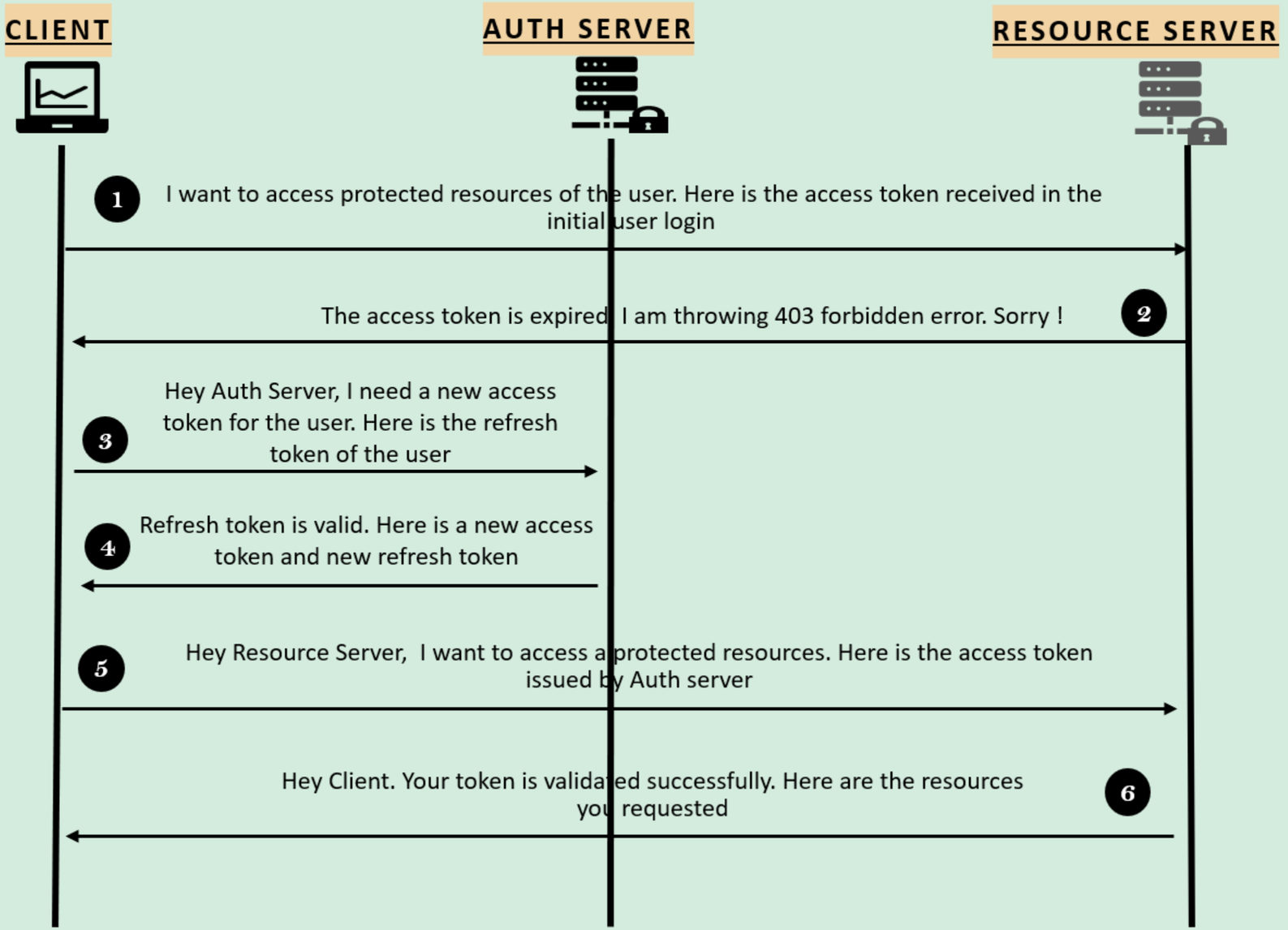
- ✓ This is the most simplest grant type flow in OAUTH2.

- ✓ 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

eazy bytes



OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE

eazy bytes

✓ 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.

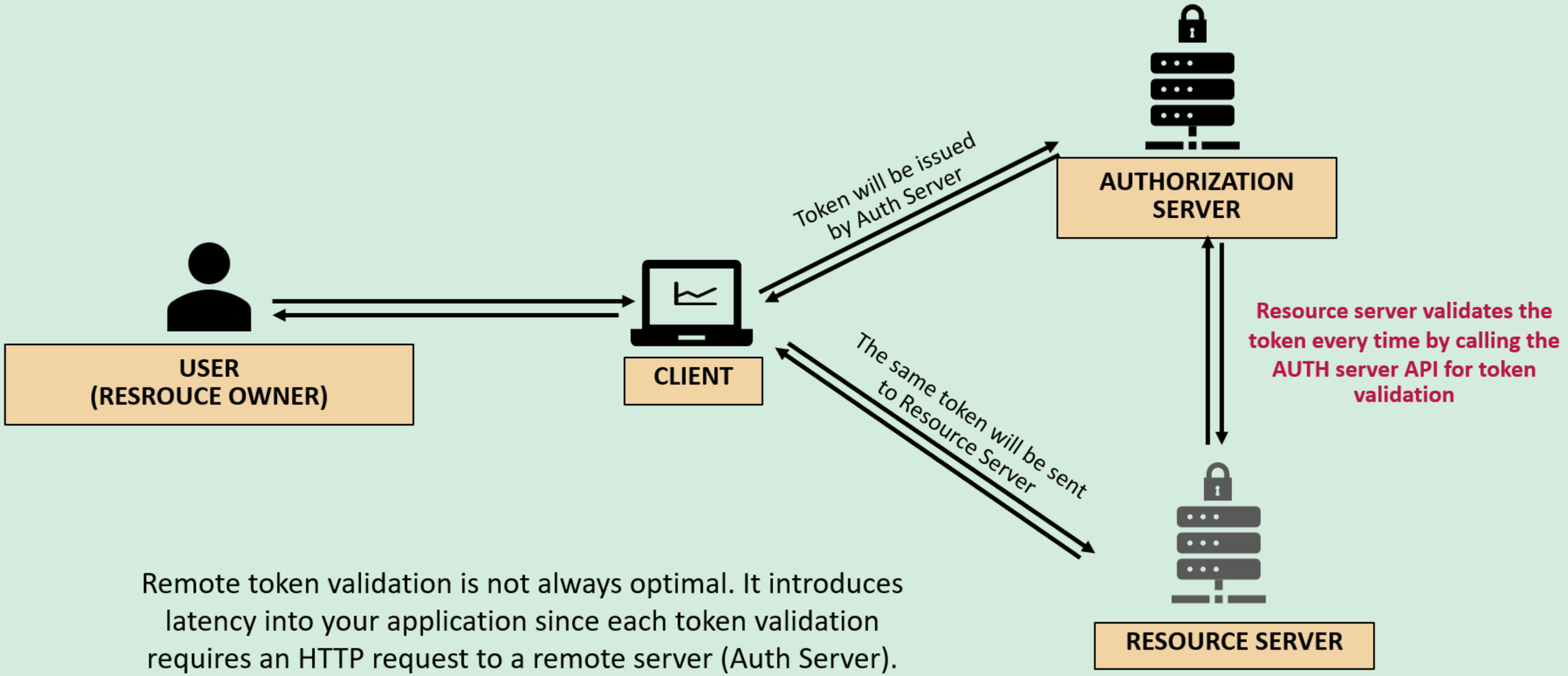
- Though we can make our refresh tokens to never expire but it is not recommended considering scenarios where the tokens can be stole if we always use the same token

- offline_access scope is used to get a refresh token that never expires.

RESOURCE SERVER TOKEN VALIDATION

VALIDATING ACCESS TOKENS REMOTELY

eazy bytes

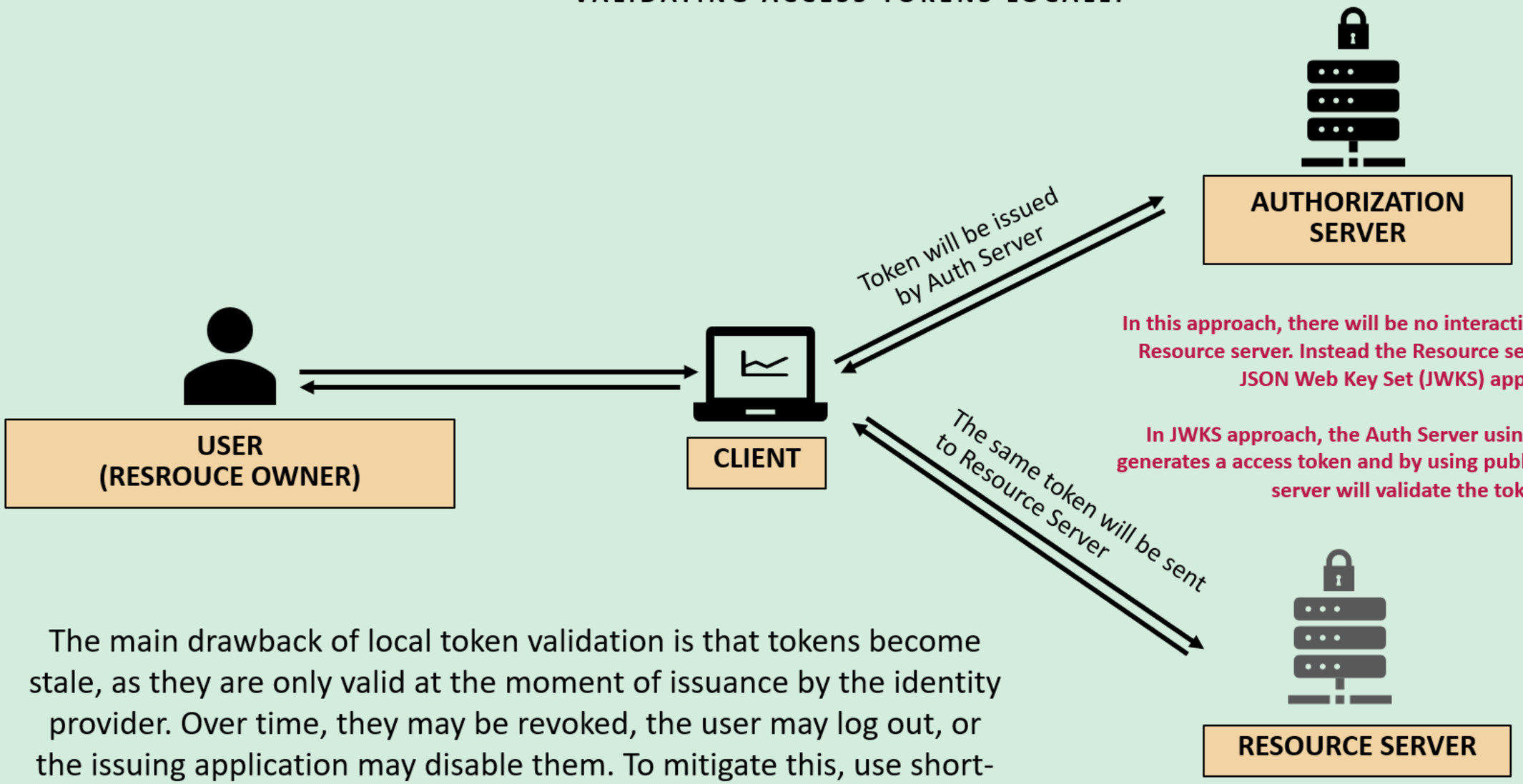


Remote token validation is not always optimal. It introduces latency into your application since each token validation requires an HTTP request to a remote server (Auth Server).

RESOURCE SERVER TOKEN VALIDATION

VALIDATING ACCESS TOKENS LOCALLY

eazy bytes



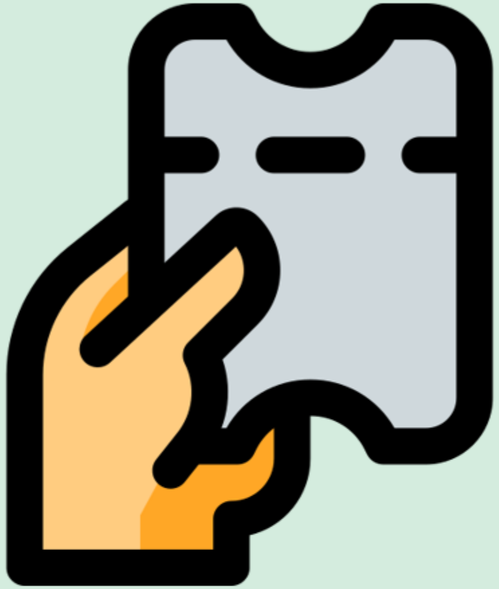
In this approach, there will be no interaction b/w the Auth & Resource server. Instead the Resource server relies on the JSON Web Key Set (JWKS) approach.

In JWKS approach, the Auth Server using a private key it generates a access token and by using public key, the resource server will validate the token.

The main drawback of local token validation is that tokens become stale, as they are only valid at the moment of issuance by the identity provider. Over time, they may be revoked, the user may log out, or the issuing application may disable them. To mitigate this, use short-lived access tokens (e.g., 5 minutes) to reduce the risk of using a revoked token.

JSON Web Token(JWT) vs Opaque Token

eazy bytes



There are multiple formats to generate tokens. But two of the most common are:

- ✓ JWT (JSON Web Tokens) - A JWT is a JSON string that contains all the claims and information it represents and is certified by a signature from the issuer. JWTs are self-contained, meaning they include all the information needed for verification within the token itself. This eliminates the need for a database lookup for each request.

- ✓ Opaque Tokens – Opaque tokens are typically long, random strings with no inherent meaning or structure that can be decoded. Since they contain no embedded information, opaque tokens must be validated by the issuing server or a dedicated token introspection endpoint. This is in contrast to JWTs where the holder of the token can't inspect it without contacting the issuer (ie the contents are opaque).

JSON Web Token(JWT) vs Opaque Token

eazy bytes

JWTs and opaque tokens serve different purposes and come with their own sets of advantages and trade-offs. The choice between them depends on the specific needs of your application, including considerations around security, performance, and ease of use.

JWT

Contains claims that can be read by anyone with the token (though sensitive data should be encrypted or omitted).

Can be validated locally using the signing key, without needing to contact the issuing server.

Ideal for stateless authentication and situations where quick, local validation is needed.

Opaque

Contains no readable information; only the server can decode and validate it.

Must be validated by the issuing server, typically requiring a network request.

Better for scenarios where token confidentiality and server-side control are prioritized, despite the potential latency from remote validation.

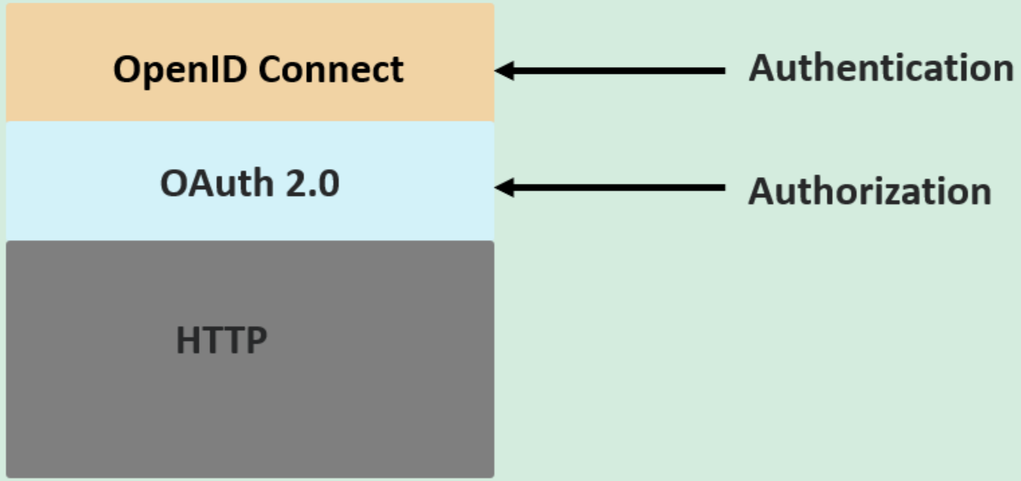
OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

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 information and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.



The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of **openid** is used, and in the final exchange the client receives both an **Access Token** and an **ID Token**.

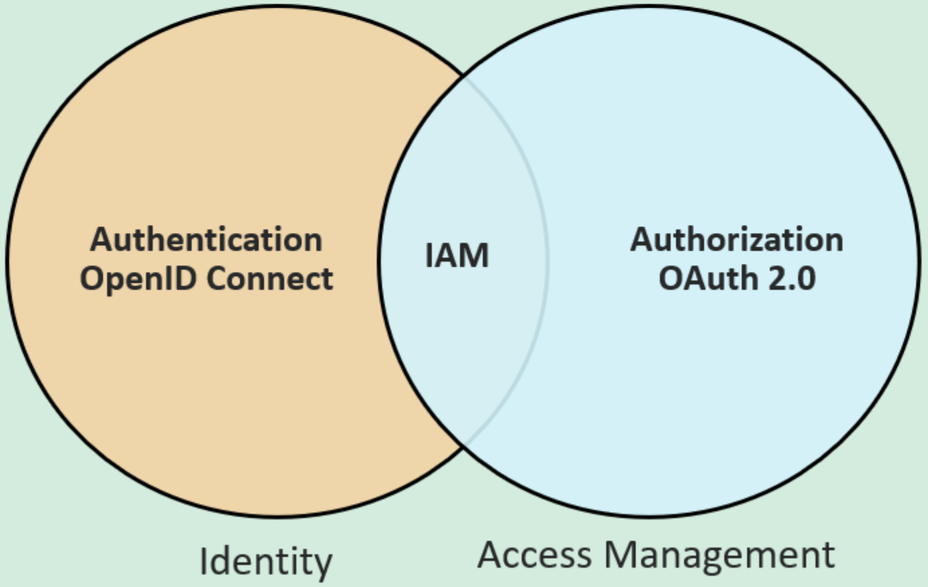
OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

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.



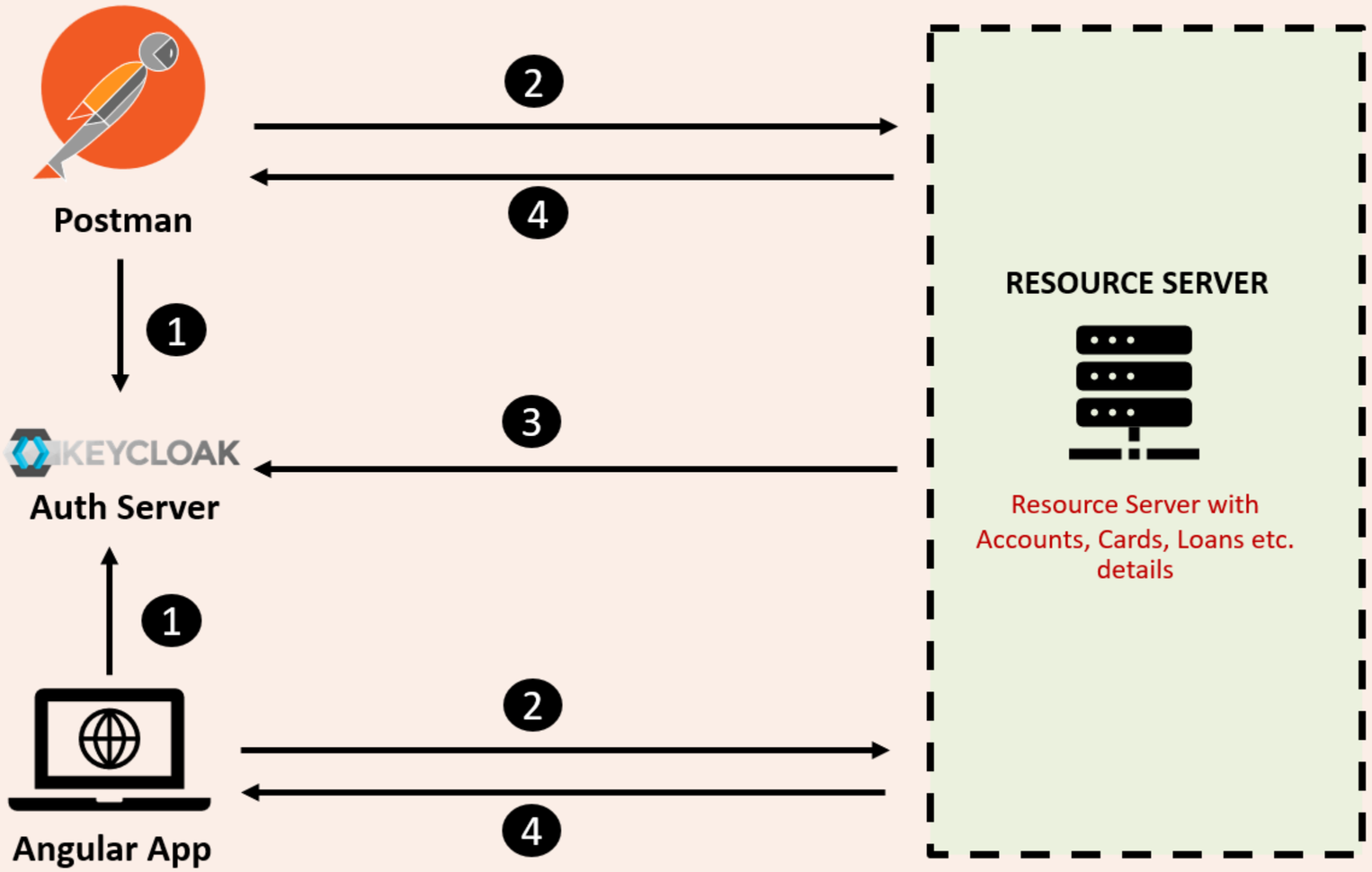
OpenID Connect add below details to OAuth 2.0

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

IMPLEMENT OAUTH2 INSIDE EAZYBANK APP

USING KEYCLOAK AUTH SERVER

eazy bytes



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.

eazy
bytes

THANK YOU