eazy
bytes

Zero To Master

Spring Security along with JWT, OAUTH2

SPRING SECURITY

Most Common Questions?

SECURITY

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

PASSWORDS

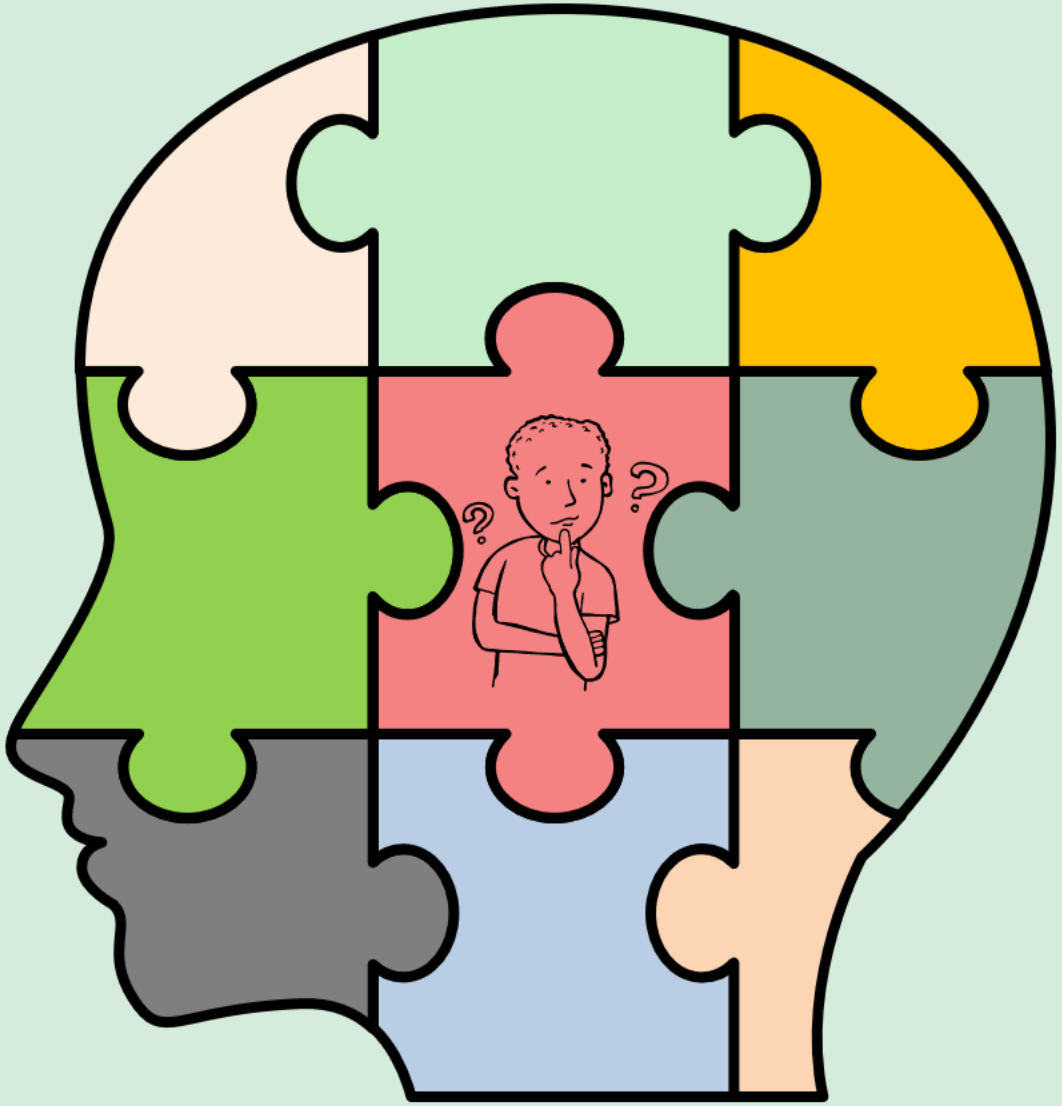
How to store passwords, validate them, encode, decode them using industry standard encryption algorithms ?

USERS & ROLES

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

MULTIPLE LOGINS

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



FINE GRAINED SECURITY

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

CSRF & CORS

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

JWT & OAUTH2

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

PREVENTING ATTACKS

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

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

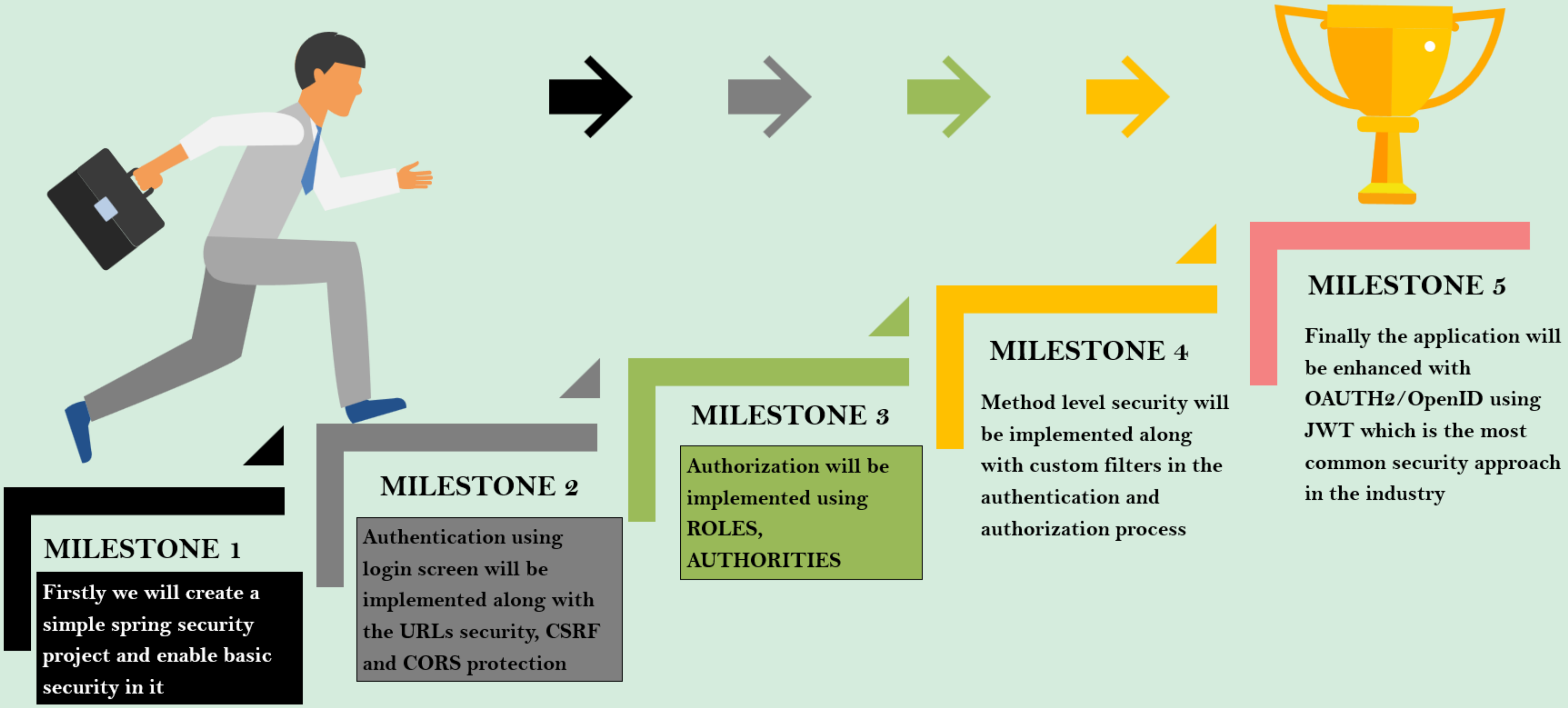
Exploring Authorization servers available like Keycloak

Important topics of Security like Hashing, Tokens & many more

SPRING SECURITY

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

INTRODUCTION TO SECURITY

WHAT & WHY

Have you ever observed how well Banks are protected?

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

INTRODUCTION TO SECURITY

WHAT & WHY

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 ?

INTRODUCTION TO SECURITY

WHAT & WHY

WHAT IS SECURITY?

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

DIFFERENT TYPES OF SECURITY

Security for a web application will be implemented in different way like using firewalls, HTTPS, SSL, 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, Broken Authentication inside our application.

WHY SPRING SECURITY ?

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

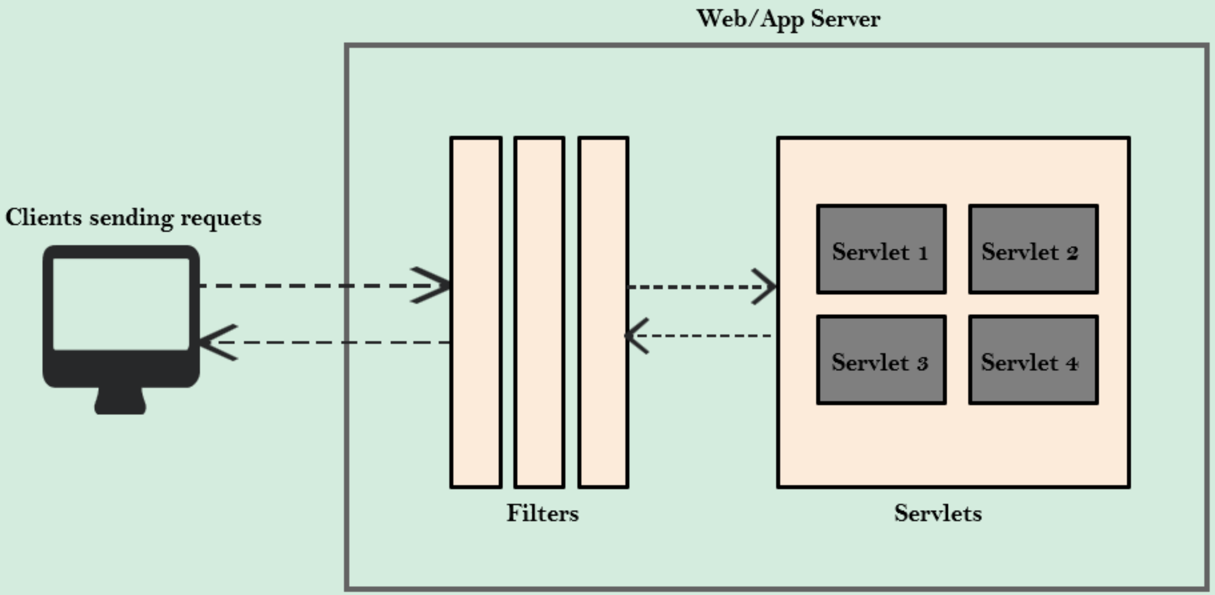
Spring Security built by a team at Spring who are good at security by considering all the security scenarios. Using Spring Security, we can secure web apps with minimum configurations. So there is no need to re-invent the wheel here.

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



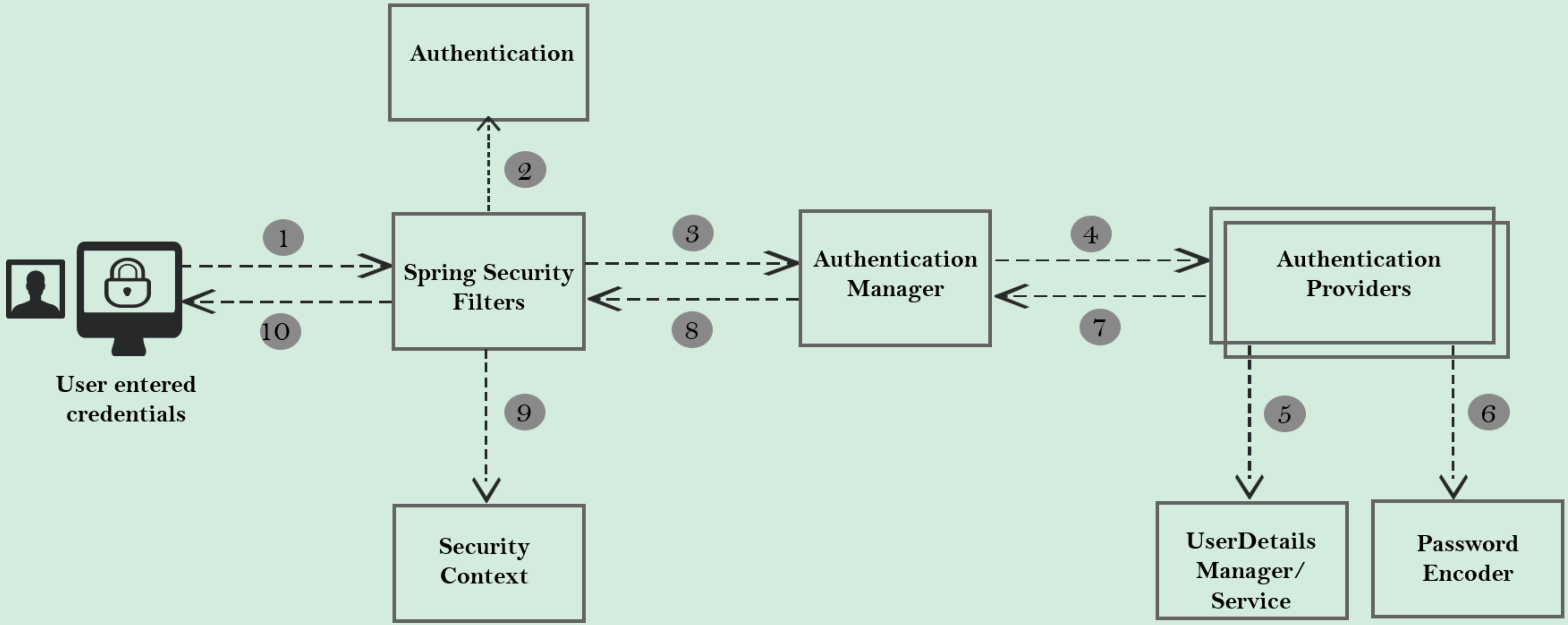
★ Typical Scenario inside a web application

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



SPRING SECURITY INTERNAL FLOW

★ **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.

★ **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**

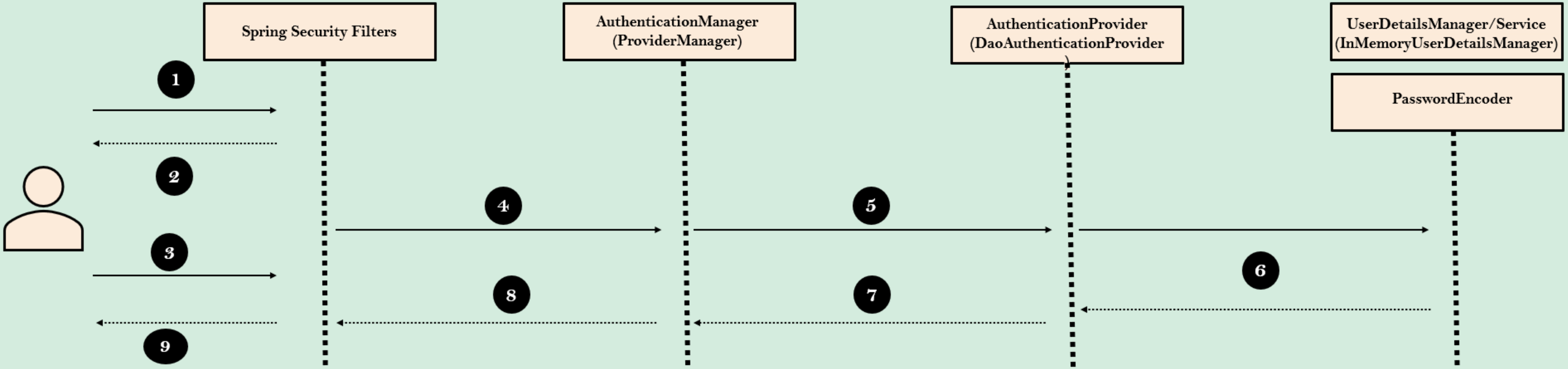
Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords ☹

★ **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.

SEQUENCE FLOW

SPRING SECURITY DEFAULT BEHAVIOUR


Spring Security Filters
AuthenticationManager (ProviderManager)
AuthenticationProvider (DaoAuthenticationProvider)
UserDetailsManager/Service (InMemoryUserDetailsManager)
PasswordEncoder
1
2
3
4
5
6
7
8
9

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

2) Behind the scenes few filters like AuthorizationFilter, 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

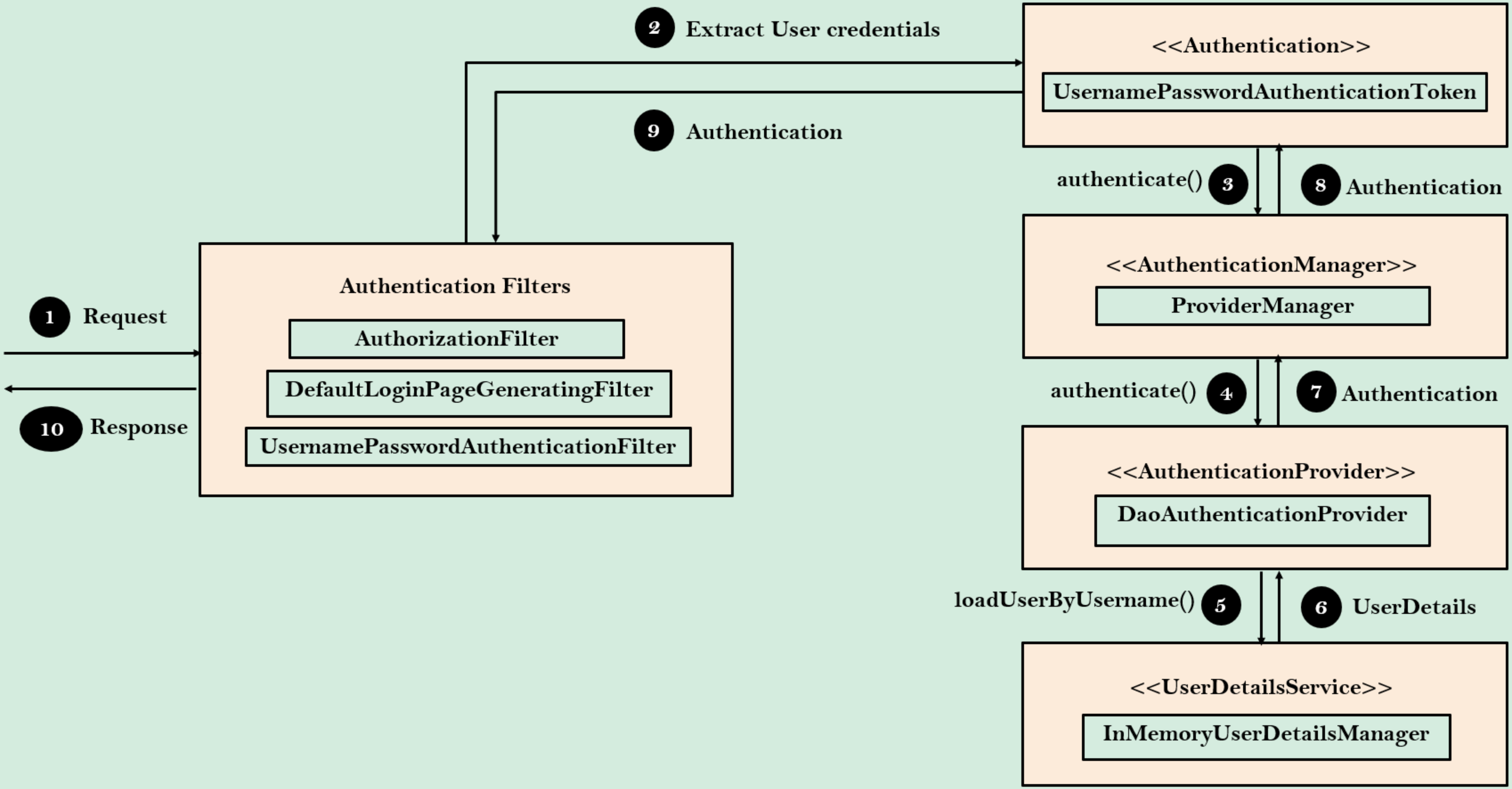
5) ProviderManager which is an implementation of AuthenticationManager identify the list of Authentication providers available that are supporting given authentication object style. In the default behaviour, authenticate() method of DaoAuthenticationProvider will be invoked by ProviderManager

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 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.

SEQUENCE FLOW

SPRING SECURITY DEFAULT BEHAVIOUR



1 Request

Authentication Filters

AuthorizationFilter

DefaultLoginPageGeneratingFilter

UsernamePasswordAuthenticationFilter

2 Extract User credentials

<<Authentication>>

UsernamePasswordAuthenticationToken

authenticate() 3

<<AuthenticationManager>>

ProviderManager

authenticate() 4

<<AuthenticationProvider>>

DaoAuthenticationProvider

loadUserByUsername() 5

<<UserDetailsService>>

InMemoryUserDetailsManager

6 UserDetails

7 Authentication

8 Authentication

9 Authentication

10 Response

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

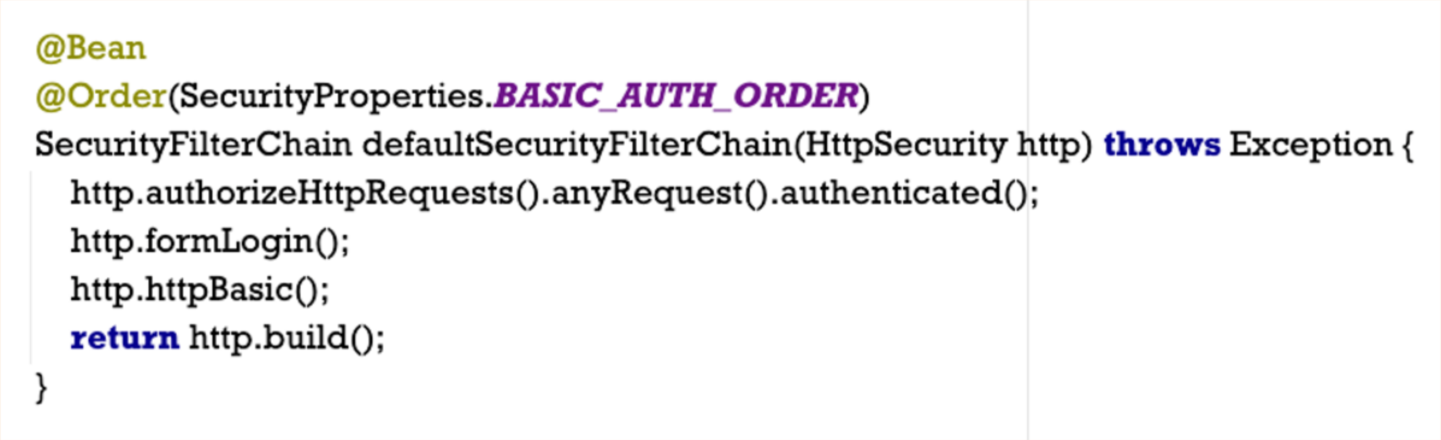
__/myLoans__ – This service should send the loan details of the logged in user from the DB to the UI

__/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

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

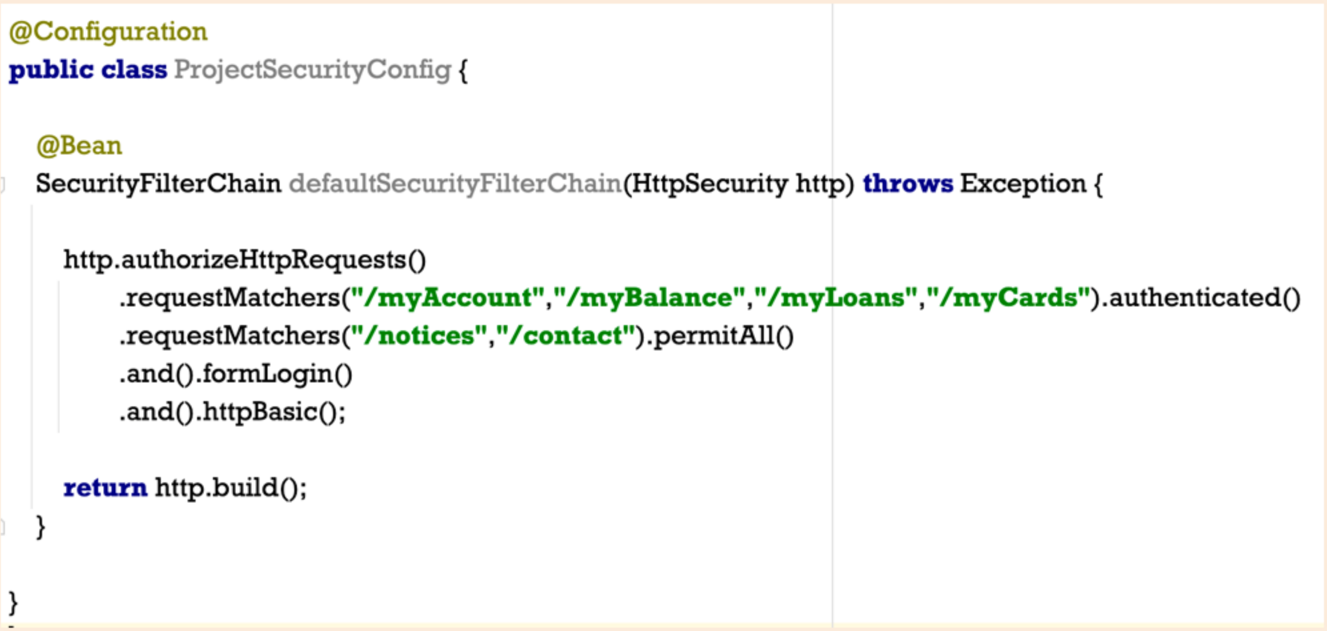


```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests().anyRequest().authenticated();
    http.formLogin();
    http.httpBasic();
    return http.build();
}
```

CUSTOM SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

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



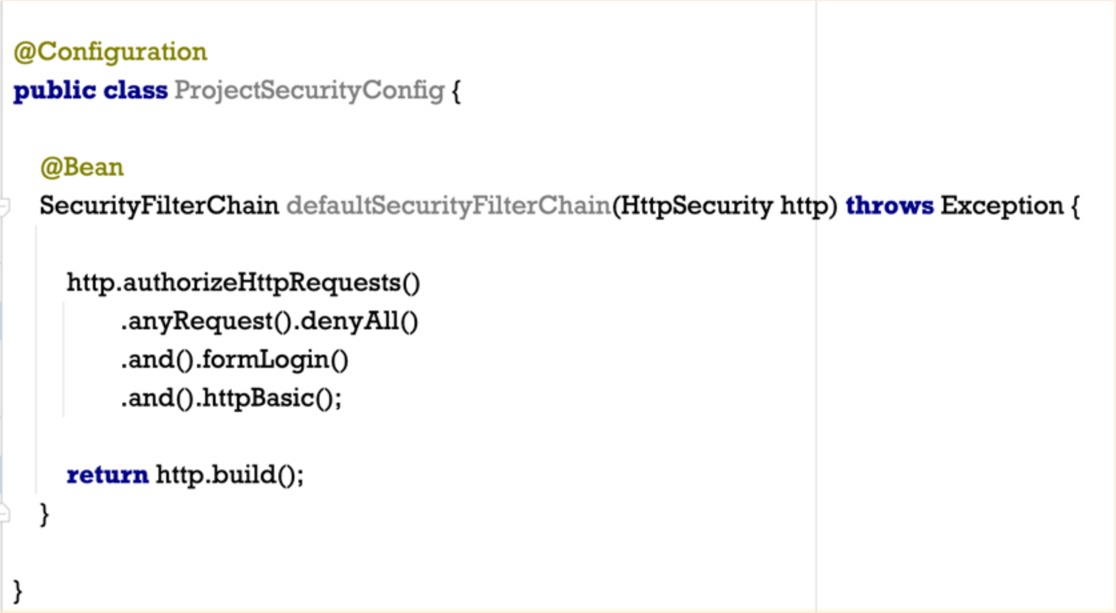
```
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests()
        .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards").authenticated()
        .requestMatchers("/notices","/contact").permitAll()
        .and().formLogin()
        .and().httpBasic();

    return http.build();
  }

}
```

NOT RECOMMENDED
FOR PRODUCTION

DENY ALL SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

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



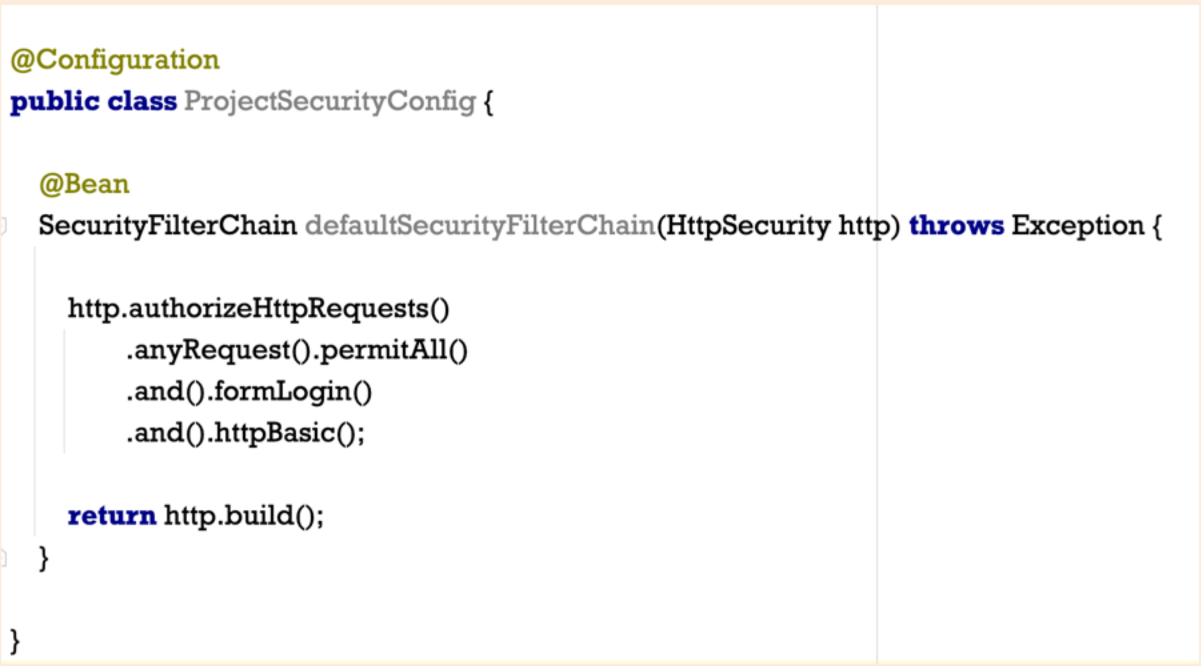
```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests()
                .anyRequest().denyAll()
                .and().formLogin()
                .and().httpBasic();

        return http.build();
    }

}
```

NOT RECOMMENDED FOR PRODUCTION

PERMIT ALL SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

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()
                .anyRequest().permitAll()
                .and().formLogin()
                .and().httpBasic();

        return http.build();
    }

}
```

NOT RECOMMENDED FOR PRODUCTION

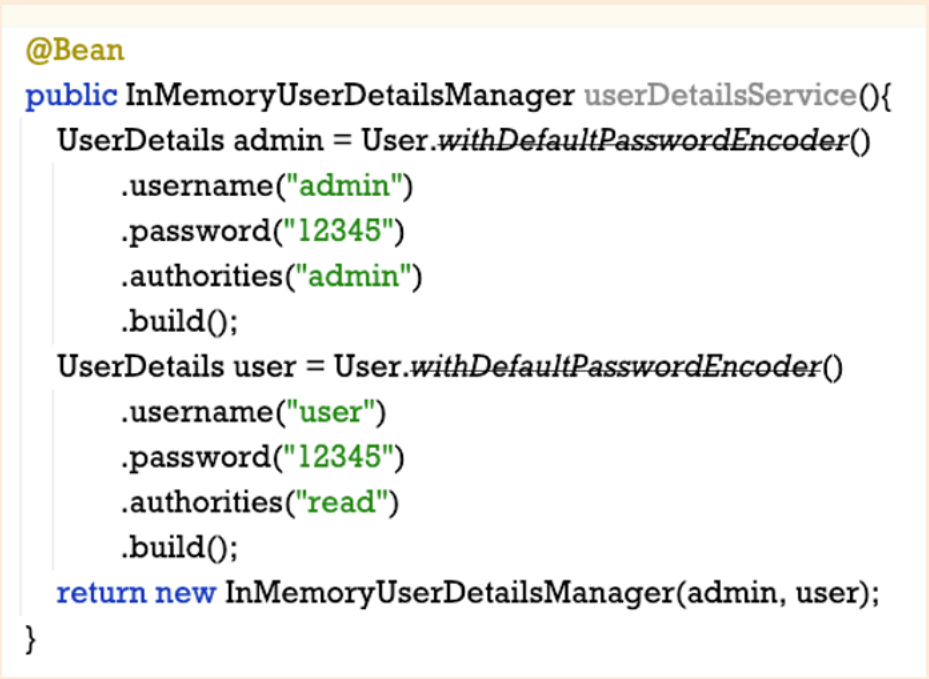
CONFIGURE USERS

USING *InMemoryUserDetailsManager*

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

Approach 1 where we use withDefaultPasswordEncoder() method while creating the user details



```
@Bean
public InMemoryUserDetailsManager userDetailsService(){
    UserDetails admin = User.~~withDefaultPasswordEncoder~~()
            .username("admin")
            .password("12345")
            .authorities("admin")
            .build();
    UserDetails user = User.~~withDefaultPasswordEncoder~~()
            .username("user")
            .password("12345")
            .authorities("read")
            .build();
    return new InMemoryUserDetailsManager(admin, user);
}
```

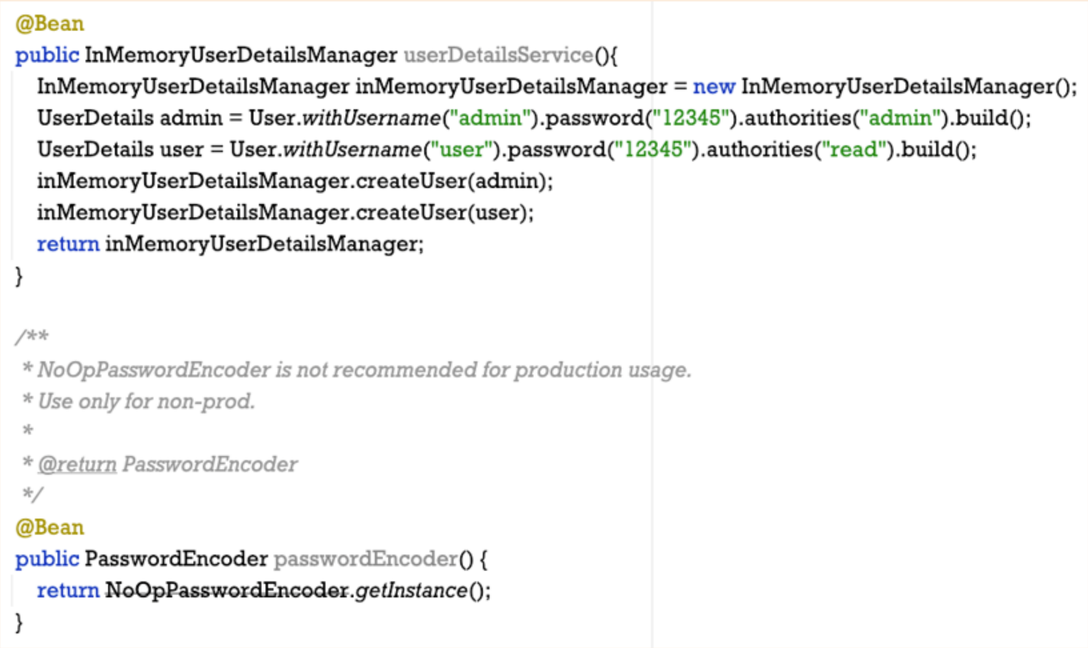
NOT RECOMMENDED FOR PRODUCTION

CONFIGURE USERS

USING *InMemoryUserDetailsManager*

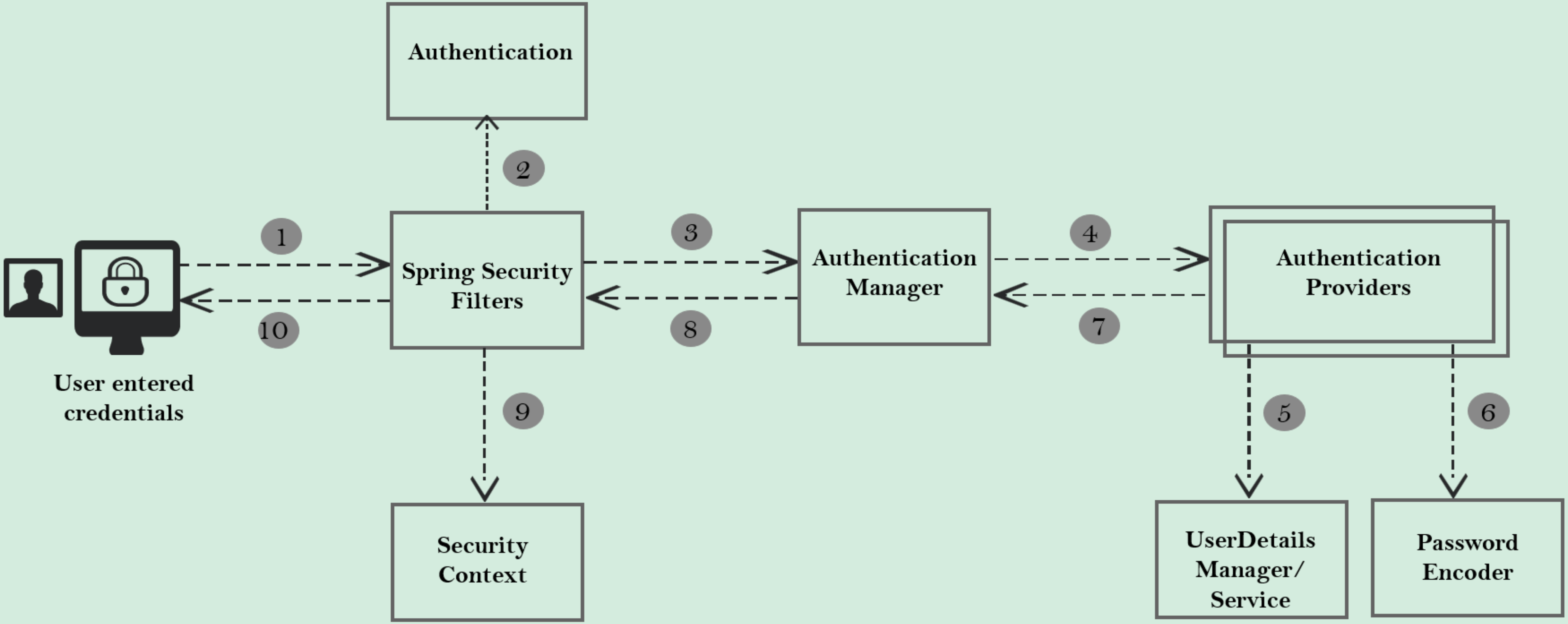
Instead of defining a single user inside application.properties, as a next step we can define multiple users along with their authorities with the help of InMemoryUserDetailsManager & UserDetails

Approach 2 where we create a bean of PasswordEncoder separately



```
@Bean
public InMemoryUserDetailsManager userDetailsService(){
   InMemoryUserDetailsManager inMemoryUserDetailsManager = new InMemoryUserDetailsManager();
   UserDetails admin = User.withUsername("admin").password("12345").authorities("admin").build();
   UserDetails user = User.withUsername("user").password("12345").authorities("read").build();
   inMemoryUserDetailsManager.createUser(admin);
   inMemoryUserDetailsManager.createUser(user);
   return inMemoryUserDetailsManager;
}

/**
 * NoOpPasswordEncoder is not recommended for production usage.
 * Use only for non-prod.
 *
 * @return PasswordEncoder
 */
@Bean
public PasswordEncoder passwordEncoder() {
   return NoOpPasswordEncoder.getInstance();
}
```

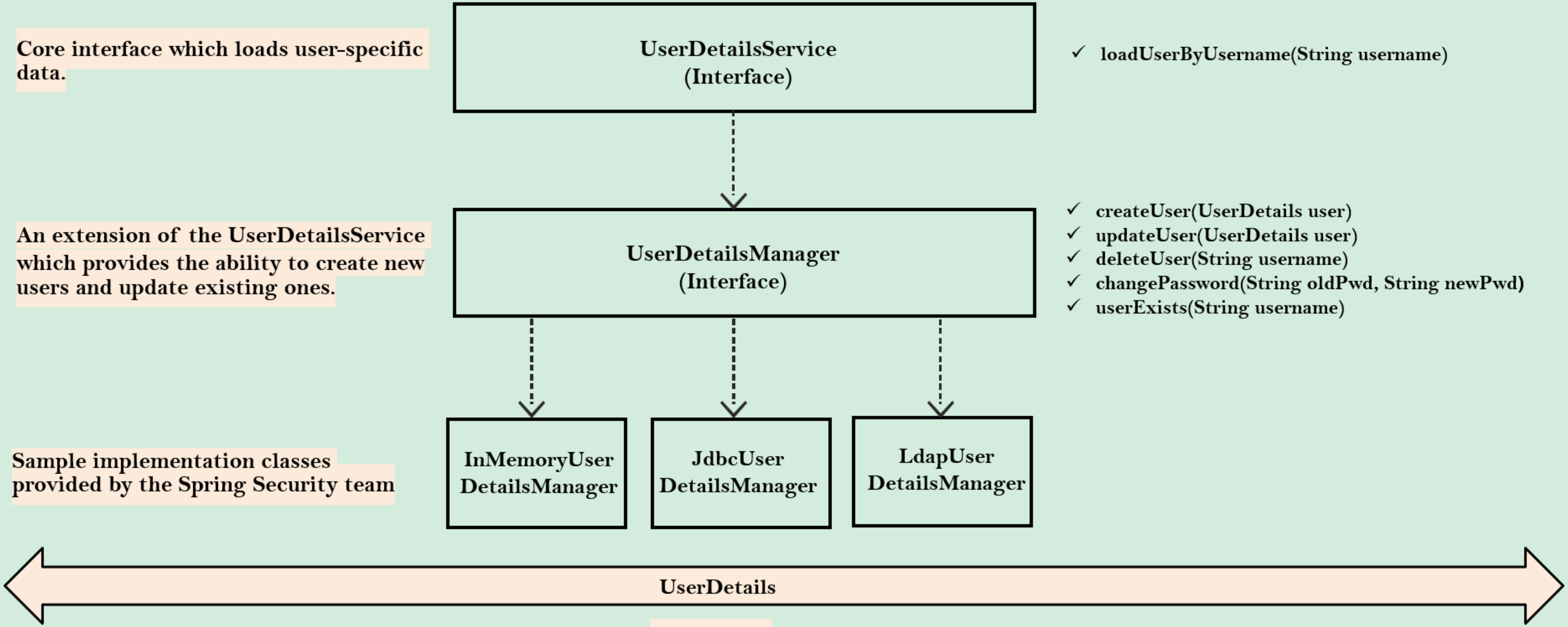
SPRING SECURITY INTERNAL FLOW



USER MANAGEMENT

IMPORTANT CLASSES & INTERFACES

eazy bytes



Core interface which loads user-specific data.

- ✓ loadUserByUsername(String username)

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

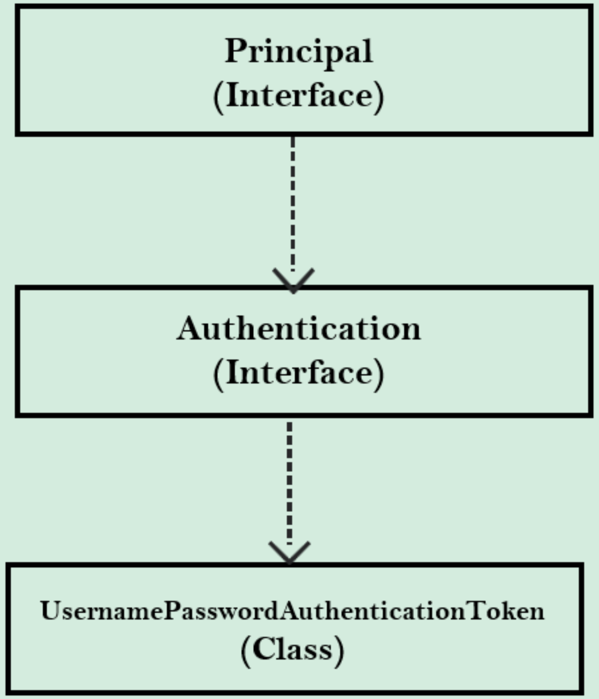
- ✓ createUser(UserDetails user)
- ✓ updateUser(UserDetails user)
- ✓ deleteUser(String username)
- ✓ changePassword(String oldPwd, String newPwd)
- ✓ userExists(String username)

Sample implementation classes provided by the Spring Security team

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

USERDETAILS & AUTHENTICATION

RELATION BETWEEN THEM

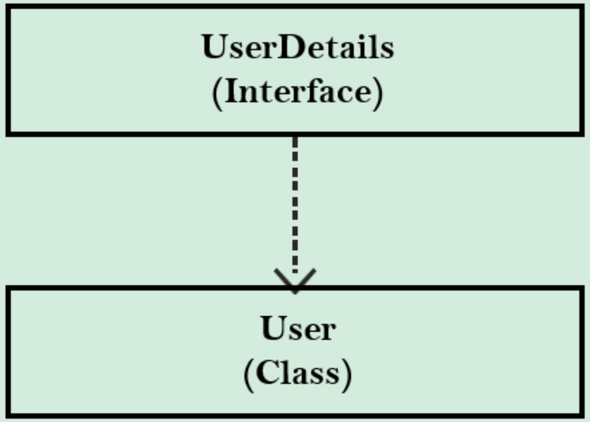

Principal
(Interface)

Authentication
(Interface)

UsernamePasswordAuthenticationToken
(Class)

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

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


UserDetails
(Interface)

User
(Class)

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

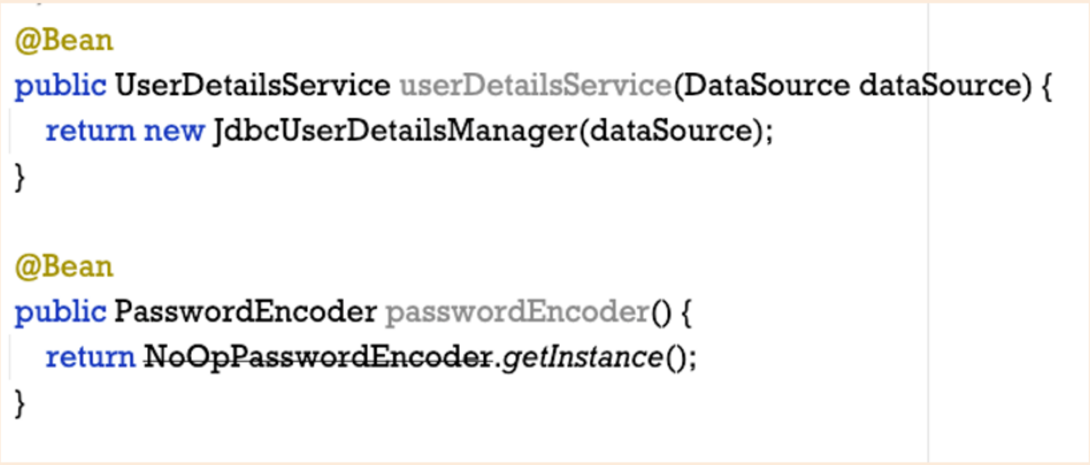
UserDetails is the return type in all the scenarios where we try to load the user info from the storage systems. Like inside the UserDetailsService & UserDetailsManager

AUTHENTICATION

USING *JdbcUserDetailsManager*

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.

Please note to create table as per the JdbcUserDetailsManager class & insert user records inside them. NoOpPasswordEncoder is not recommended for prod apps.

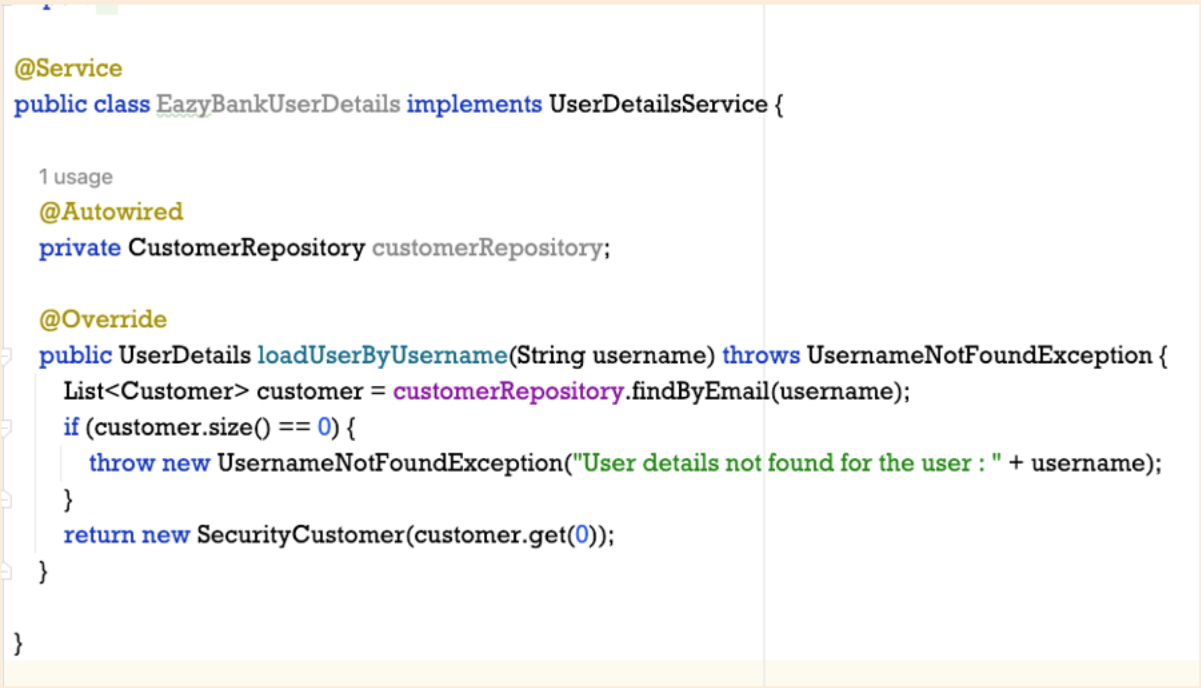


```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    return new JdbcUserDetailsManager(dataSource);
}

@Bean
public PasswordEncoder passwordEncoder() {
    return ~~NoOpPasswordEncoder~~.getInstance();
}
```

USERDETAILSSERVICE IMPLEMENTATION

FOR CUSTOM USER FETCHING LOGIC

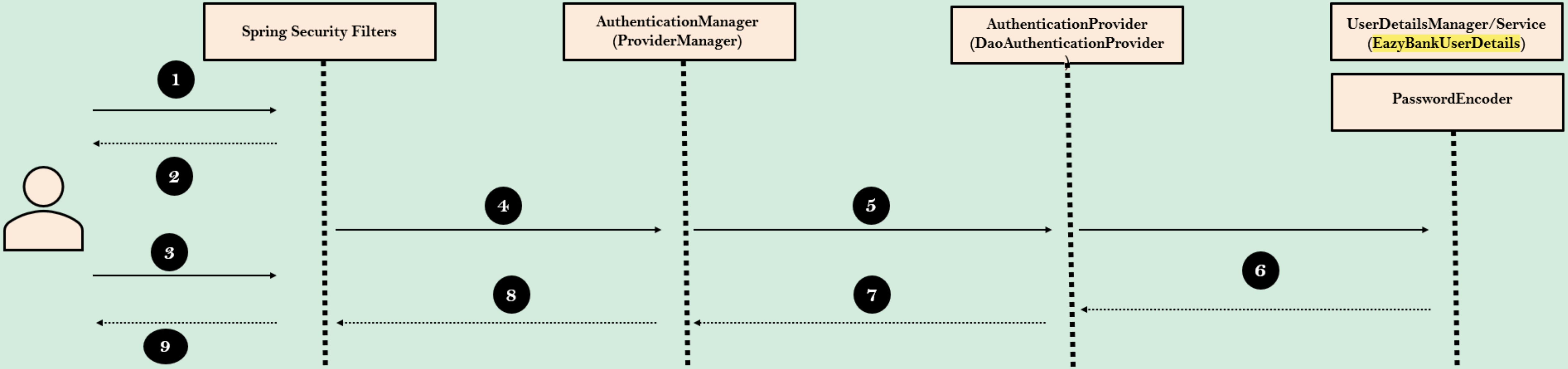
When we want to load the user details based on our own tables, columns, custom logic, then we need to create a bean that implements UserDetailsService and overrides the method loadUserByUsername()



```
@Service
public class EazyBankUserDetails implements UserDetailsService {

    1 usage
    @Autowired
    private CustomerRepository customerRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        List<Customer> customer = customerRepository.findByEmail(username);
        if (customer.size() == 0) {
            throw new UsernameNotFoundException("User details not found for the user : " + username);
        }
        return new SecurityCustomer(customer.get(0));
    }

}
```

SEQUENCE FLOW

WITH OUR OWN USERDETAILSSERVICE IMPLEMENTATION



Spring Security Filters
AuthenticationManager (ProviderManager)
AuthenticationProvider (DaoAuthenticationProvider)
UserDetailsManager/Service (EazyBankUserDetails)
PasswordEncoder
1
2
3
4
5
6
7
8
9

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

2) Behind the scenes few filters like AuthorizationFilter, 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

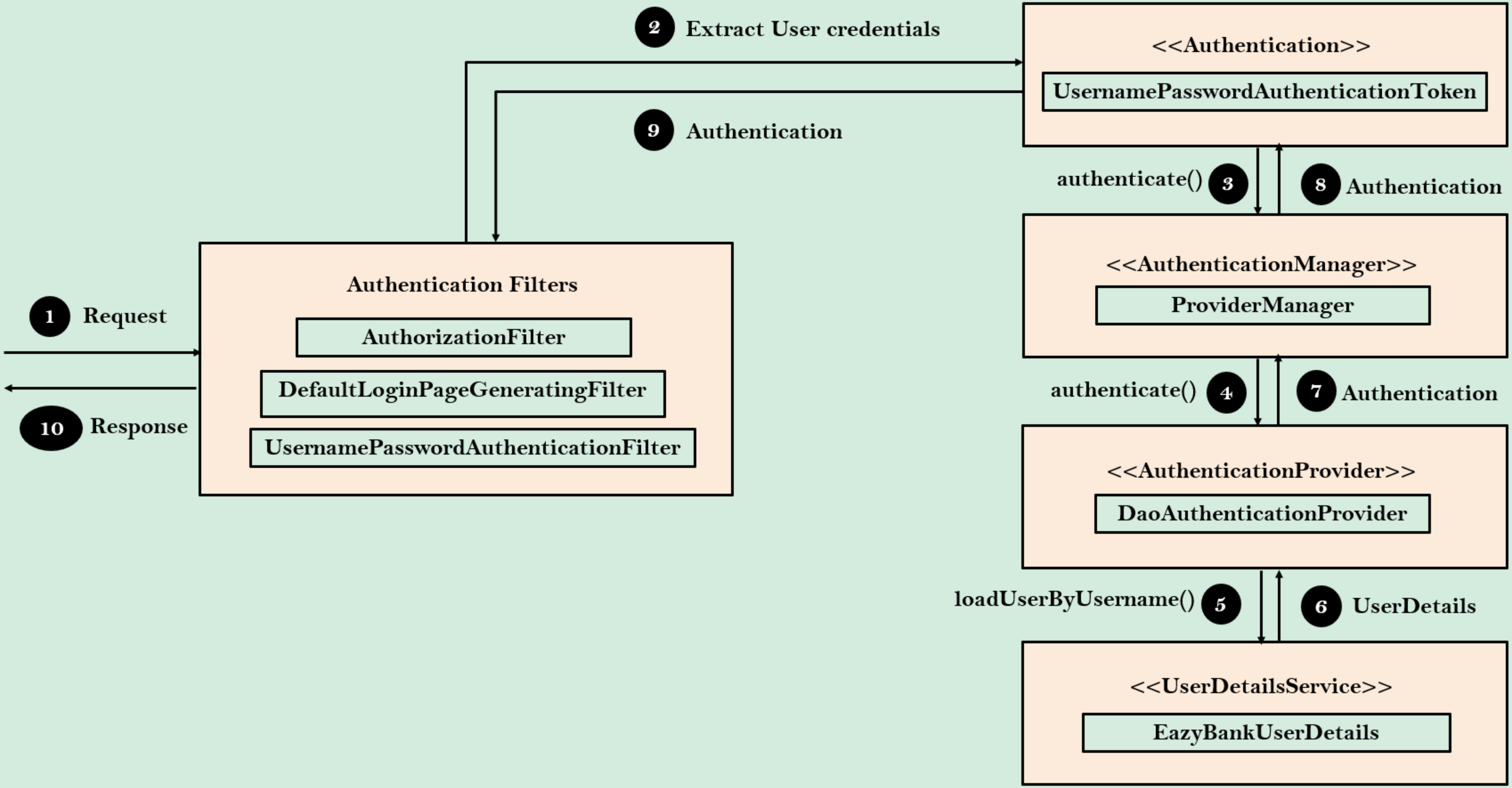
7) At last it returns the Authentication object with the details of authentication success or not to ProviderManager

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

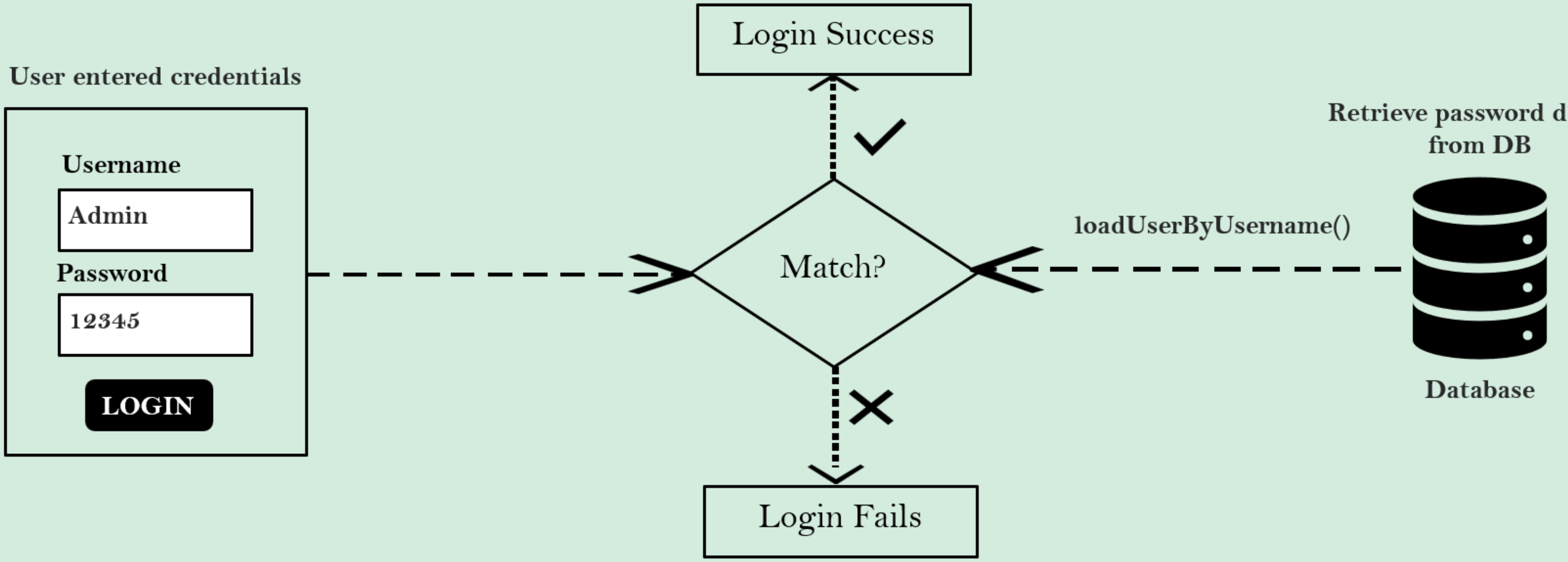


HOW PASSWORDS VALIDATED

With default PasswordEncoder

eazy bytes

NOT RECOMMENDED FOR PRODUCTION



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.

Different ways of Pwd management

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

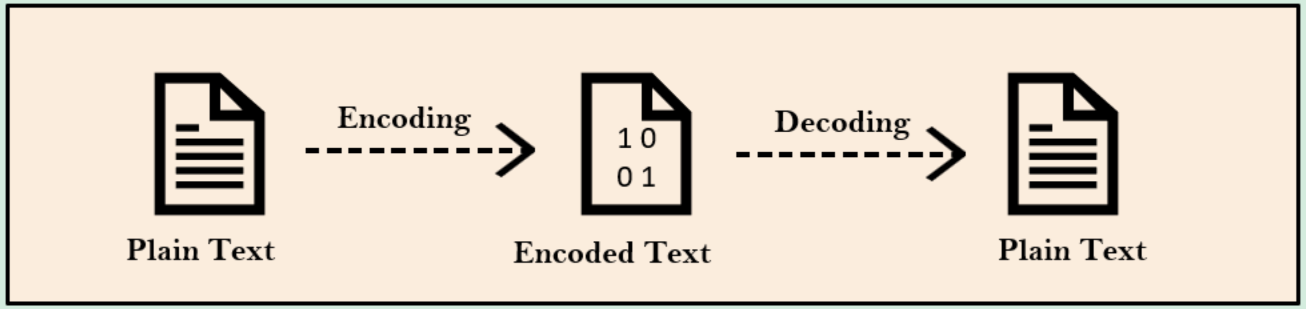
- ✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.
- ✓ To achieve confidentiality,encryption requires the use of a secret which, in cryptographic terms, we call a "key".
- ✓ Encryption can be reversible by using decryption with the help of the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

Hashing

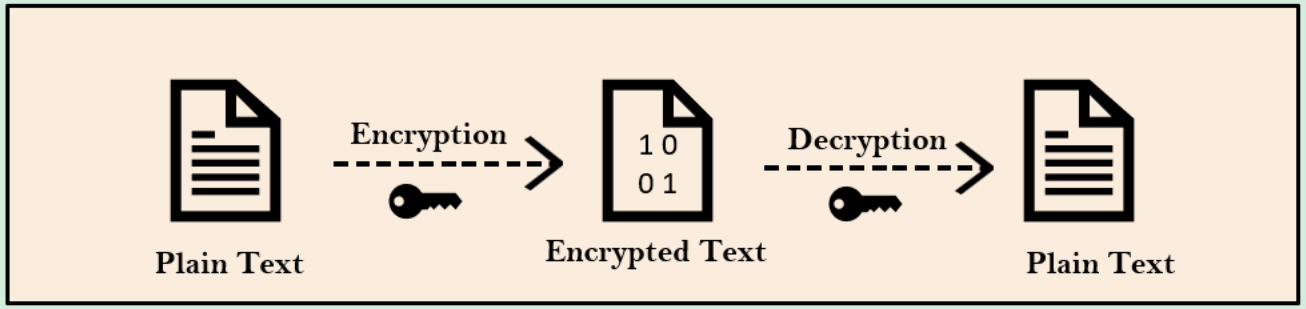
- ✓ In hashing, data is converted to the hash value using some hashing function.
- ✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
- ✓ Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data

Encoding Vs Encryption Vs Hashing

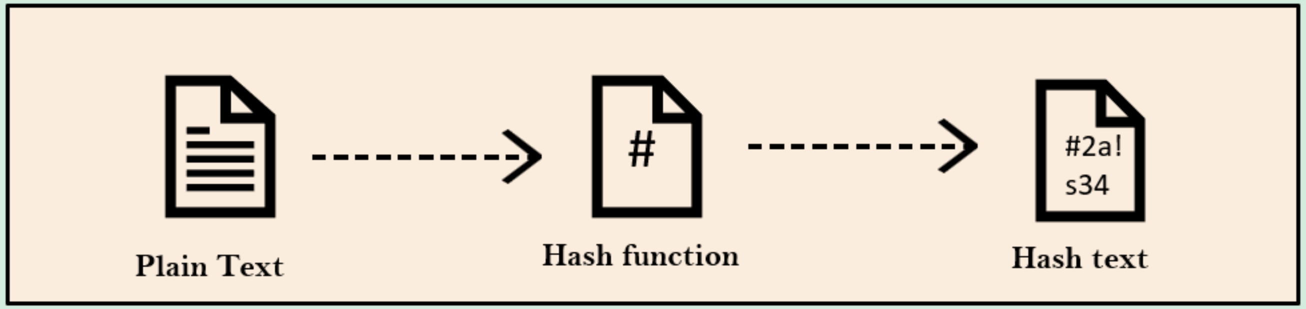
Encoding & Decoding



Encryption & Decryption

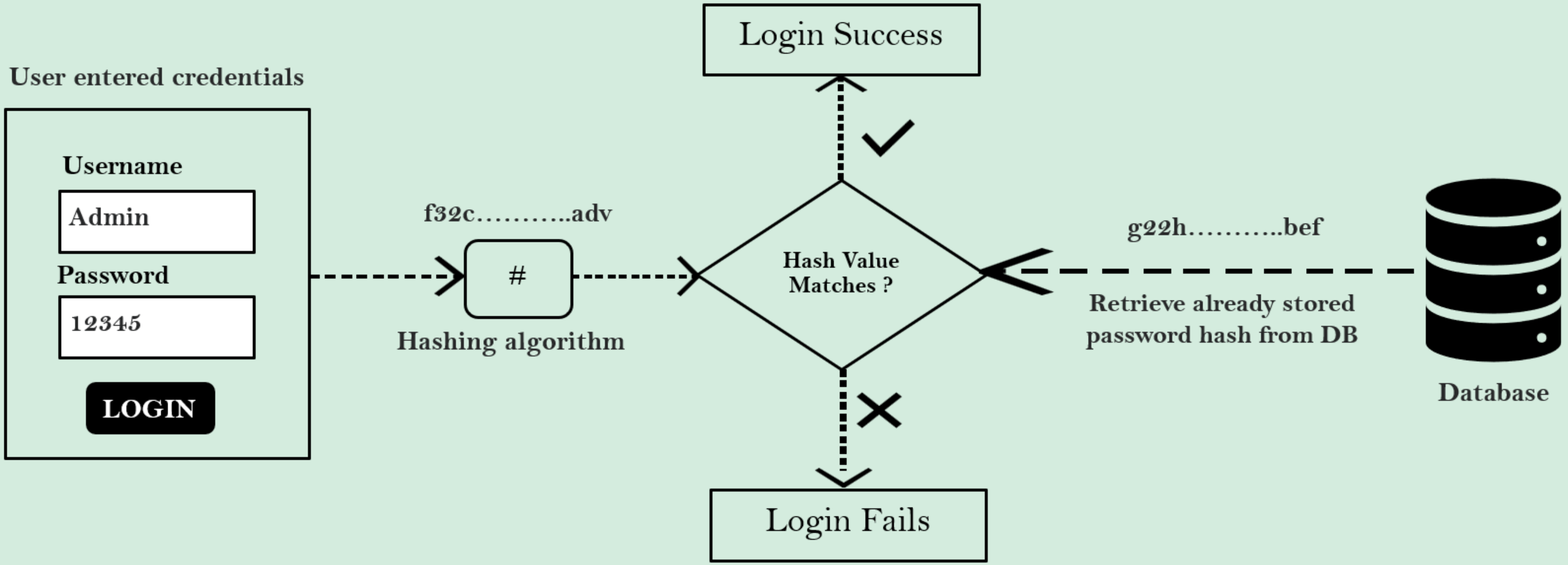


Hashing (Only 1-way)



HOW PASSWORDS VALIDATED

With Hashing & PasswordEncoders



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

Methods inside PasswordEncoder Interface

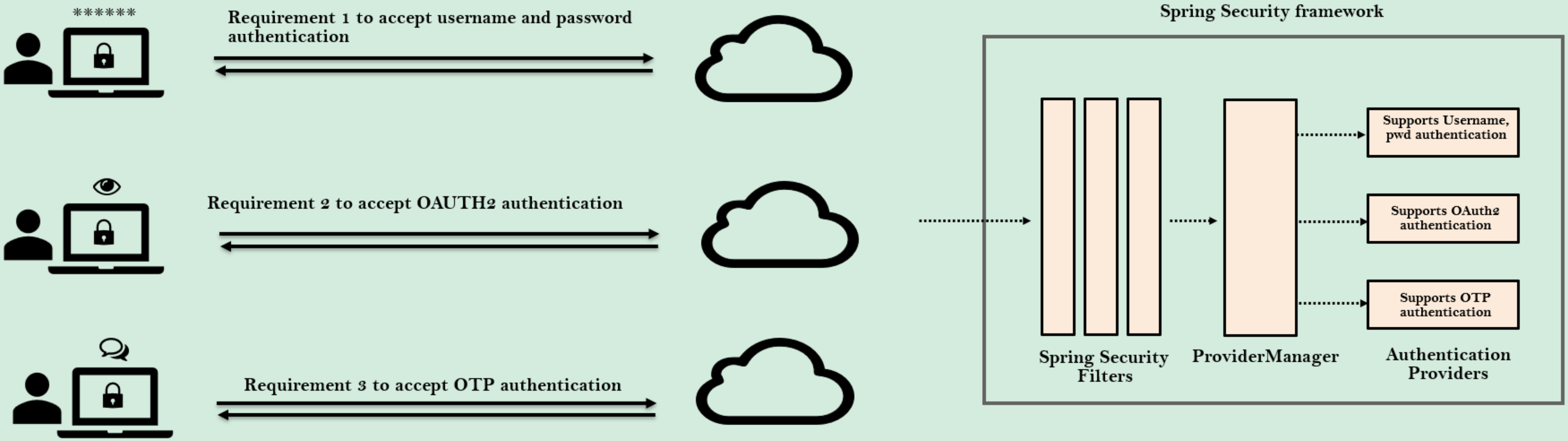
```
public interface PasswordEncoder {

        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**

AUTHENTICATION PROVIDER

WHY DO WE NEE IT?



Requirement 1 to accept username and password authentication

Requirement 2 to accept OAUTH2 authentication

Requirement 3 to accept OTP authentication

Spring Security framework

Spring Security Filters

ProviderManager

Authentication Providers

Supports Username, pwd authentication

Supports OAuth2 authentication

Supports OTP authentication

- ✓ 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

Methods inside AuthenticationProvider Interface

```
public interface AuthenticationProvider {

        Authentication authenticate(Authentication authentication)
                        throws AuthenticationException;

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

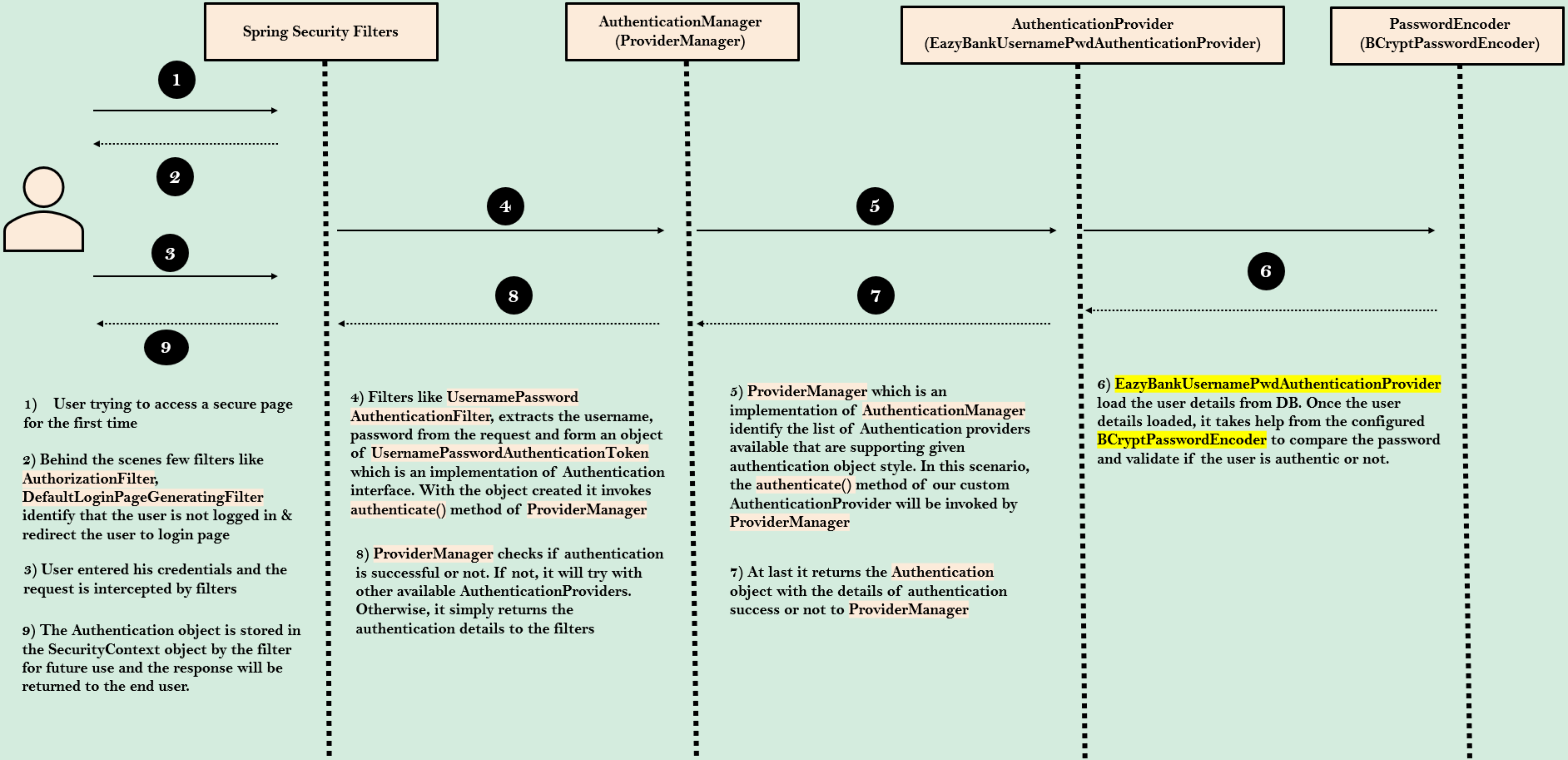
- ★ The authenticate() method receives and returns authentication object. We can implement all our custom authentication logic inside authenticate() method.

- ★ 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



Spring Security Filters

AuthenticationManager (ProviderManager)

AuthenticationProvider (EazyBankUsernamePwdAuthenticationProvider)

PasswordEncoder (BCryptPasswordEncoder)

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

2) Behind the scenes few filters like AuthorizationFilter, 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

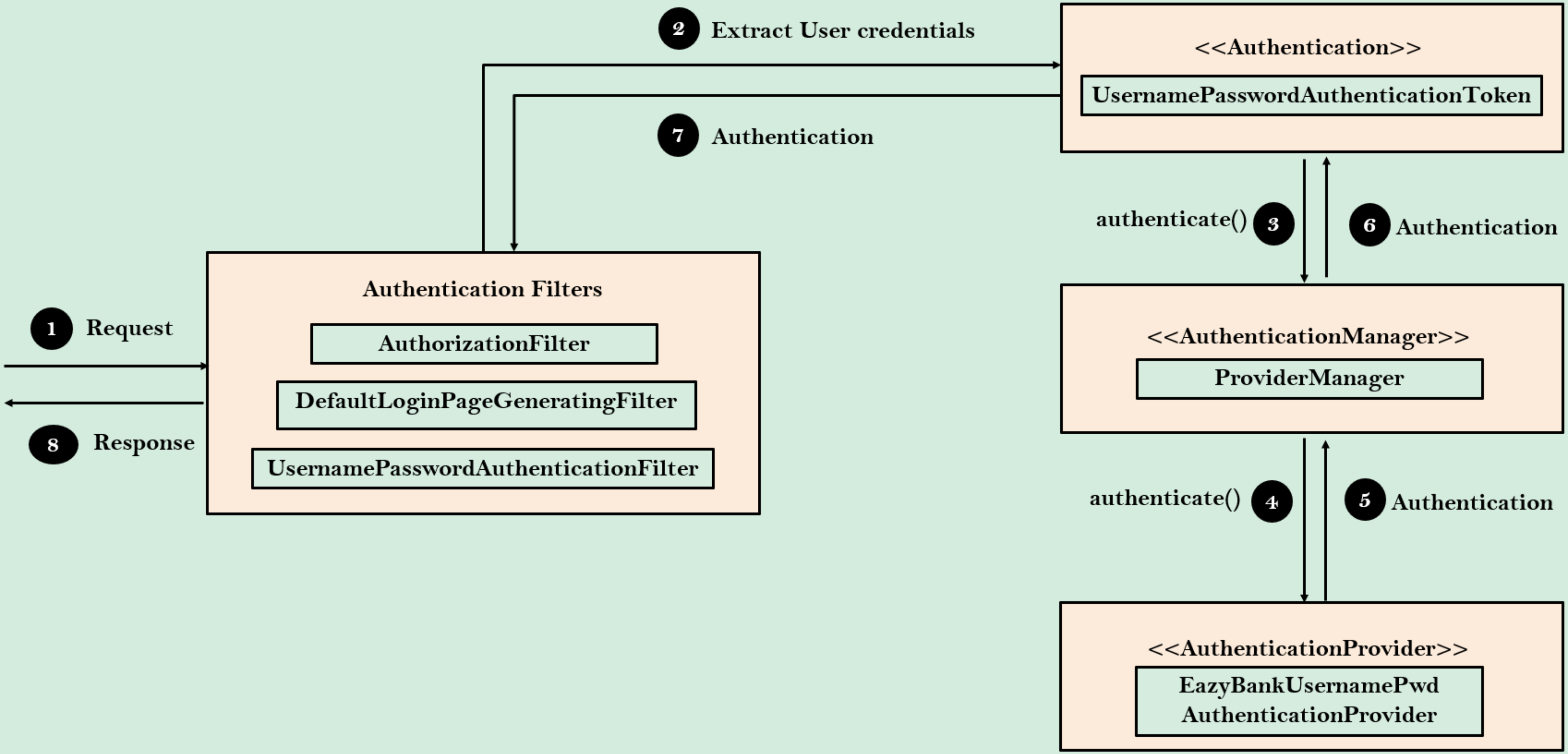
5) ProviderManager which is an implementation of AuthenticationManager identify the list of Authentication providers available that are supporting given authentication object style. In this scenario, the authenticate() method of our custom AuthenticationProvider will be invoked by ProviderManager

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

6) EazyBankUsernamePwdAuthenticationProvider load the user details from DB. Once the user details loaded, it takes help from the configured BCryptPasswordEncoder to compare the password and validate if the user is authentic or not.

SEQUENCE FLOW

WITH OUR OWN AUTHENTICATIONPROVIDER IMPLEMENTATION



CORS & CSRF

SPRING SECURITY APPROACH

CORS

CROSS-ORIGIN RESOURCE SHARING

CSRF

CROSS-SITE REQUEST FORGERY

SPRING SECURITY

HOW TO HANDLE THEM USING THE SPRING SECURITY FRAMEWORK?

CROSS-ORIGIN RESOURCE SHARING (CORS)

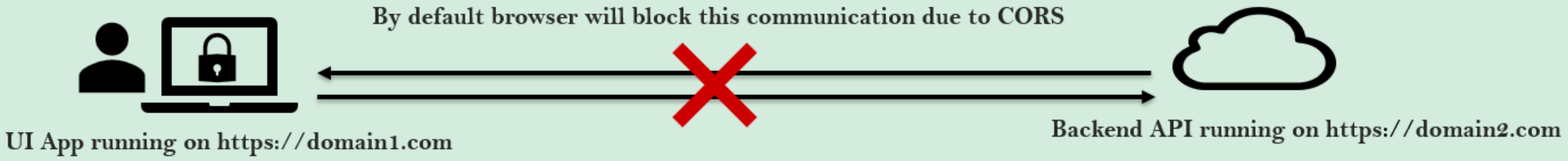
eazy bytes

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

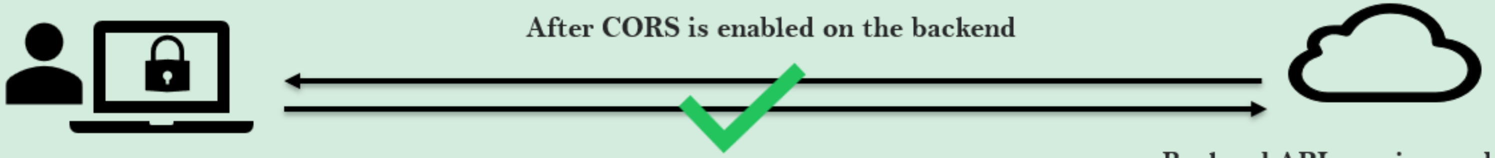

By default browser will block this communication due to CORS
UI App running on https://domain1.com
Backend API running on https://domain2.com

SOLUTION TO HANDLE CORS

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
```

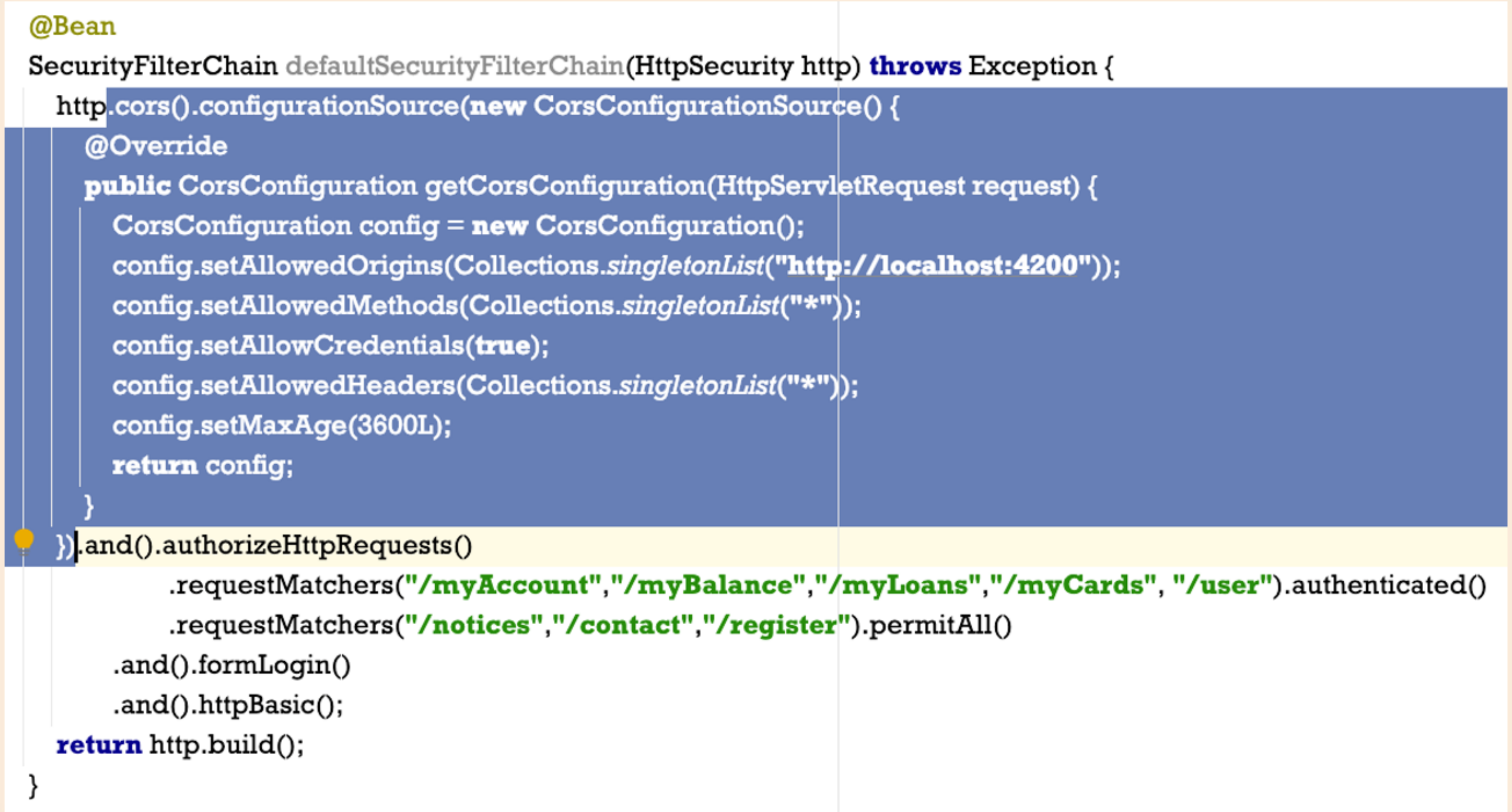

After CORS is enabled on the backend

UI App running on https://domain1.com

Backend API running on https://domain2.com

SOLUTION TO HANDLE CORS

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,

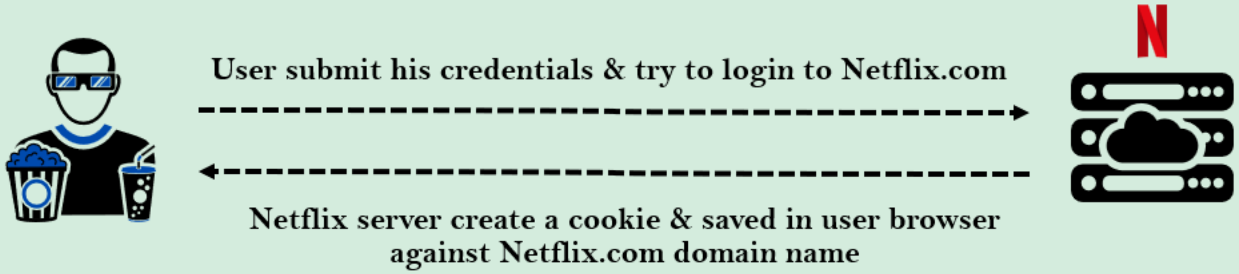


```java
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.cors().configurationSource(new CorsConfigurationSource() {
        @Override
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
            CorsConfiguration config = new CorsConfiguration();
            config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
            config.setAllowedMethods(Collections.singletonList("*"));
            config.setAllowCredentials(true);
            config.setAllowedHeaders(Collections.singletonList("*"));
            config.setMaxAge(3600L);
            return config;
        }
    }).and().authorizeHttpRequests()
                .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards", "/user").authenticated()
                .requestMatchers("/notices","/contact","/register").permitAll()
            .and().formLogin()
            .and().httpBasic();
    return http.build();
}
```

CROSS-SITE REQUEST FORGERY (CSRF)

- 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.

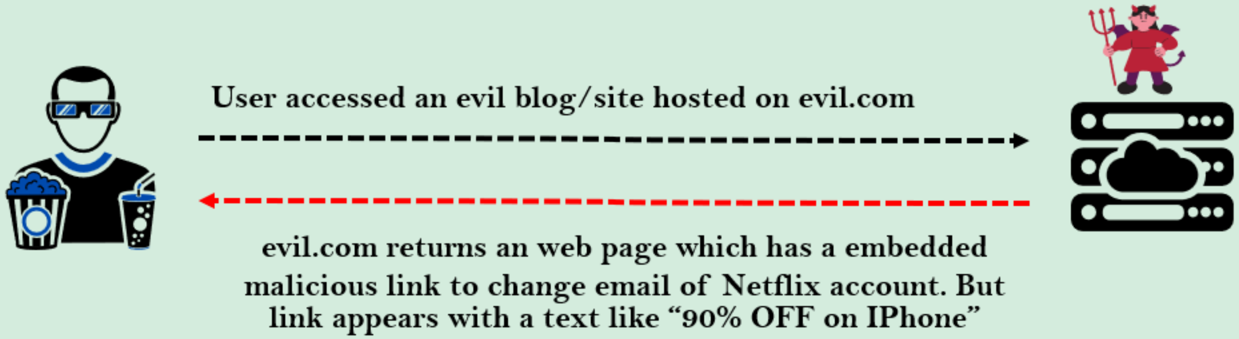
Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com



User submit his credentials & try to login to Netflix.com

Netflix server create a cookie & saved in user browser against Netflix.com domain name

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

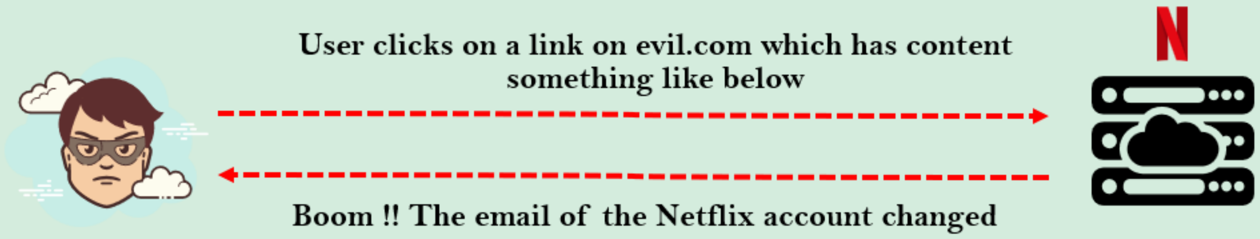


User accessed an evil blog/site hosted on evil.com

evil.com returns an web page which has a embedded malicious link to change email of Netflix account. But link appears with a text like "90% OFF on IPhone"

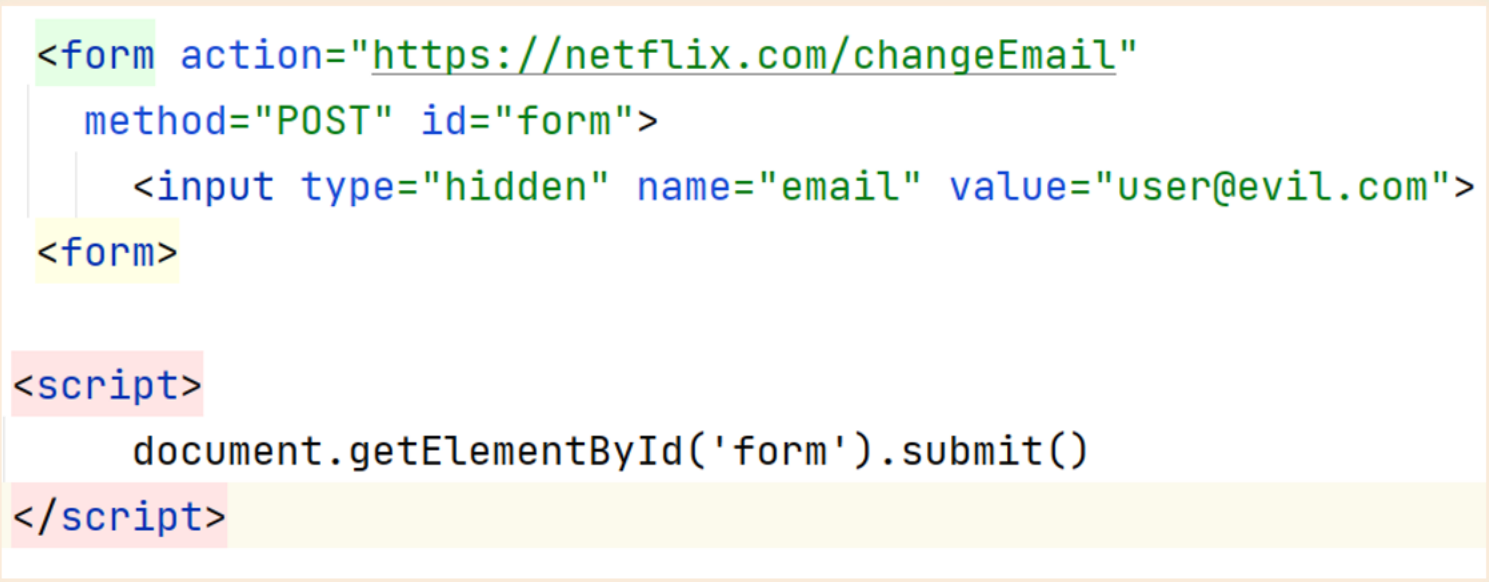
CROSS-SITE REQUEST FORGERY (CSRF)

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.



User clicks on a link on evil.com which has content something like below

Boom !! The email of the Netflix account changed

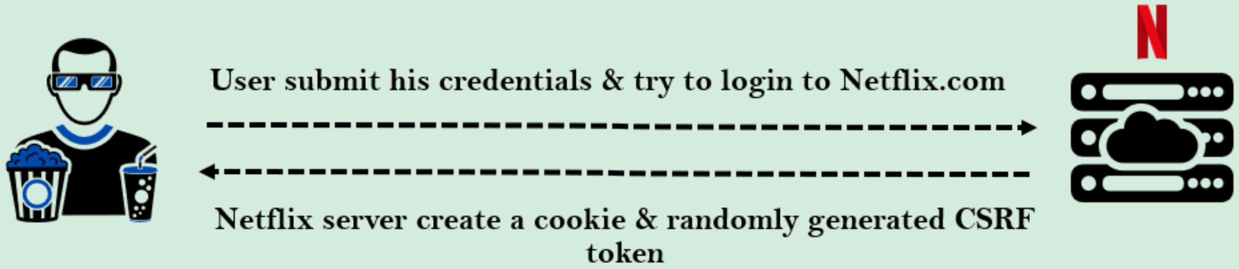


```
<form action="https://netflix.com/changeEmail"
  method="POST" id="form">
    <input type="hidden" name="email" value="user@evil.com">
<form>

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

SOLUTION TO CSRF ATTACK

- 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,

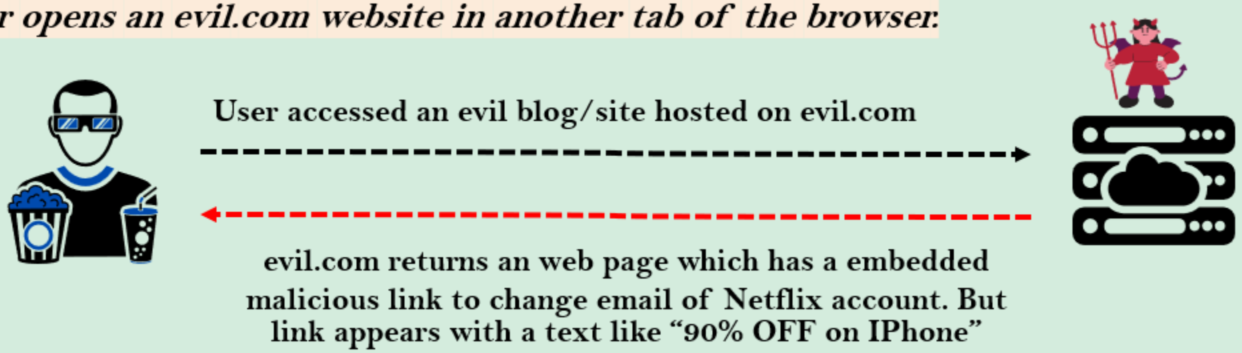
Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session. CSRF token is inserted within hidden parameters of HTML forms to avoid exposure to session cookies.



User submit his credentials & try to login to Netflix.com

Netflix server create a cookie & randomly generated CSRF token

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

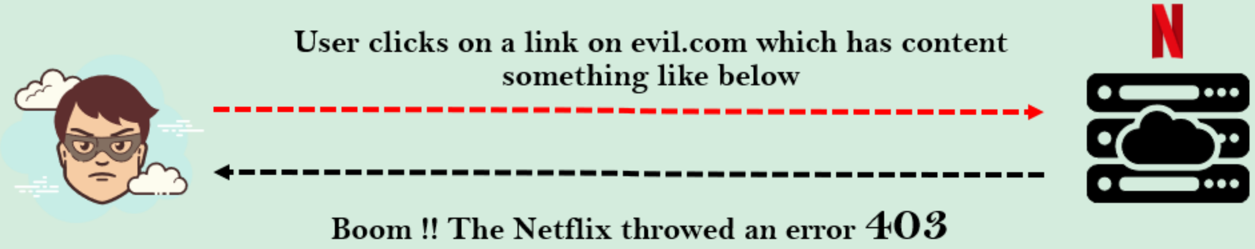


User accessed an evil blog/site hosted on evil.com

evil.com returns an web page which has a embedded malicious link to change email of Netflix account. But link appears with a text like "90% OFF on IPhone"

SOLUTION TO CSRF ATTACK

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



User clicks on a link on evil.com which has content something like below

Boom !! The Netflix throwed an error 403

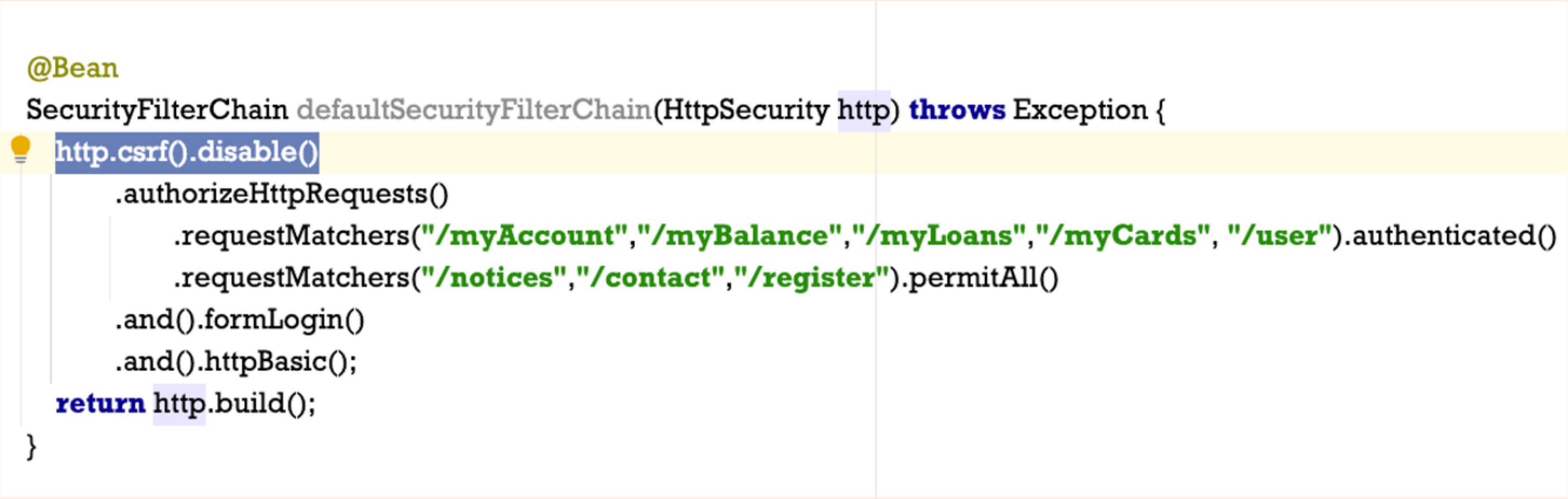
The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.

NOT RECOMMENDED FOR PRODUCTION

DISABLE CSRF PROTECTION

INSIDE SPRING SECURITY

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.

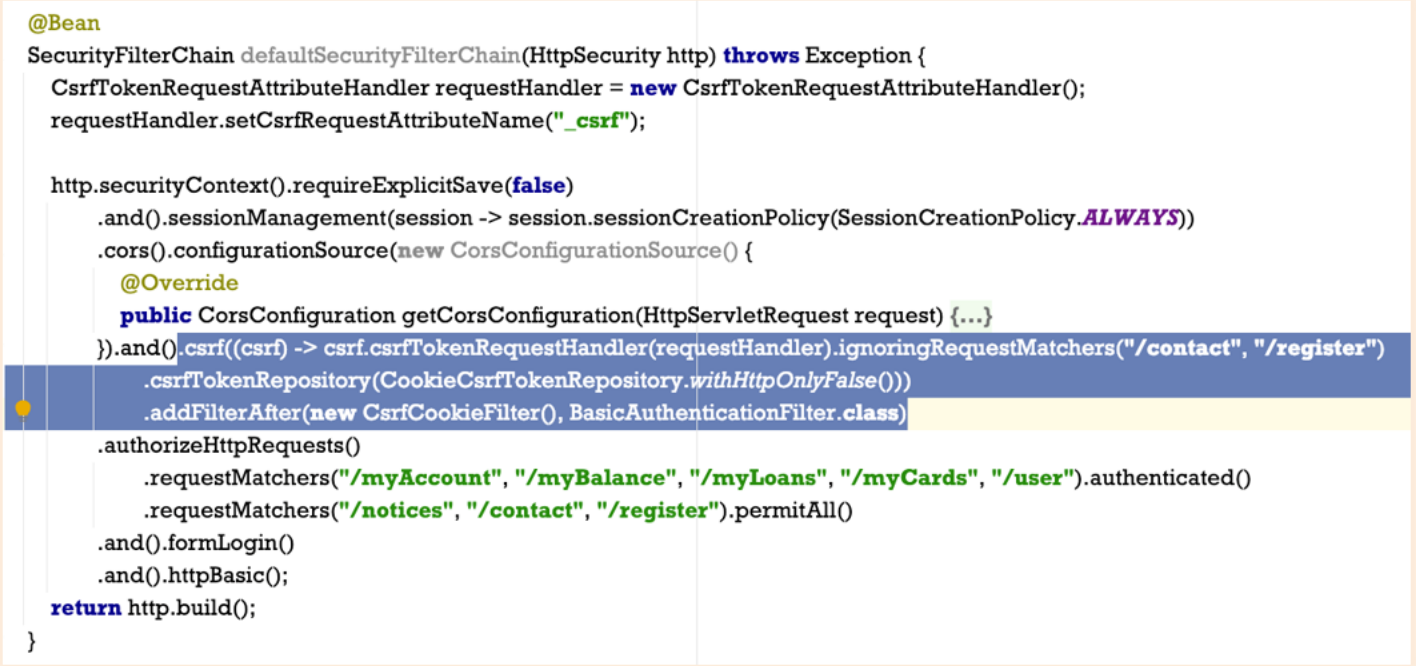


```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
            .authorizeHttpRequests()
                .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards", "/user").authenticated()
                .requestMatchers("/notices","/contact","/register").permitAll()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

CSRF ATTACK SOLUTION

INSIDE SPRING SECURITY

With the below 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.



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

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

AUTHENTICATION & AUTHORIZATION

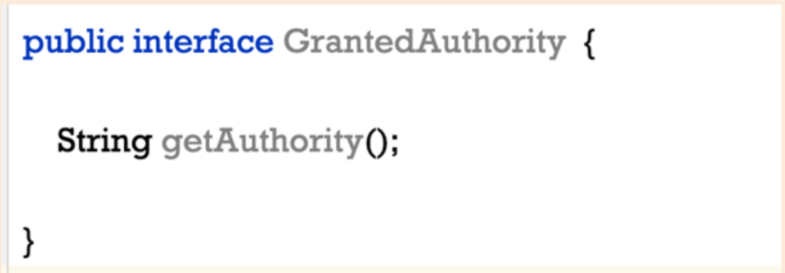
DETAILS & COMPARISION

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

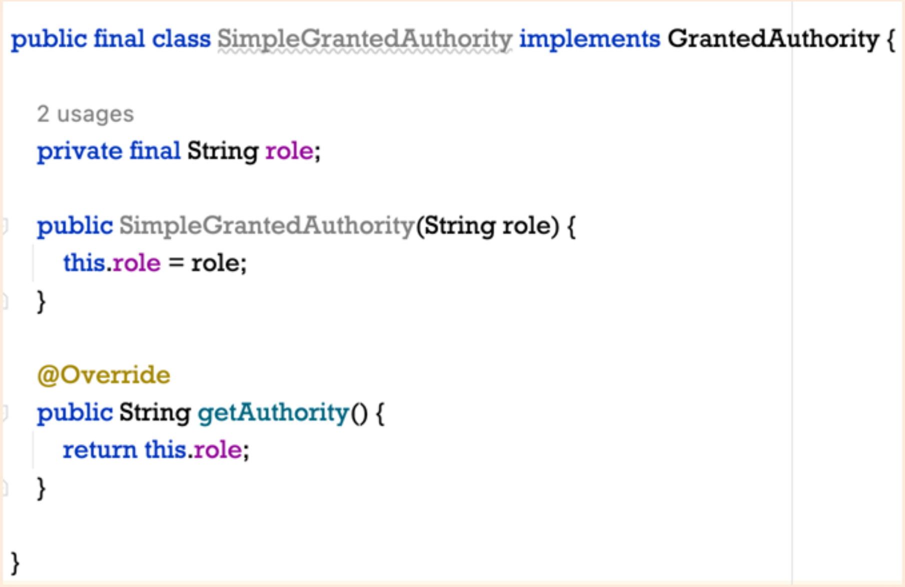
HOW AUTHORITIES STORED ?

INSIDE SPRING SECURITY

- 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();

}
```

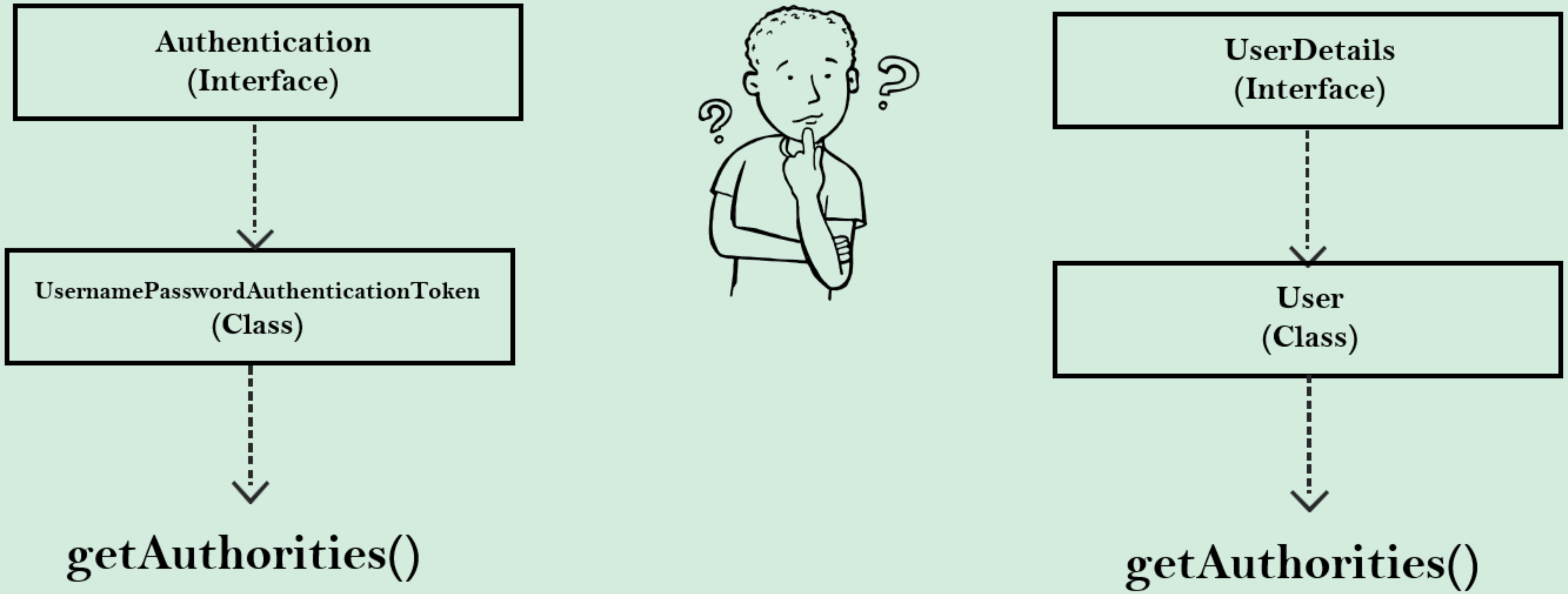


```
public final class SimpleGrantedAuthority implements GrantedAuthority {

   2 usages
   private final String role;

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

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

}
```

HOW AUTHORITIES STORED ?

INSIDE SPRING SECURITY

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



CONFIGURING AUTHORITIES

INSIDE SPRING SECURITY

In Spring Security the authorities requirements can be configured using the following ways,

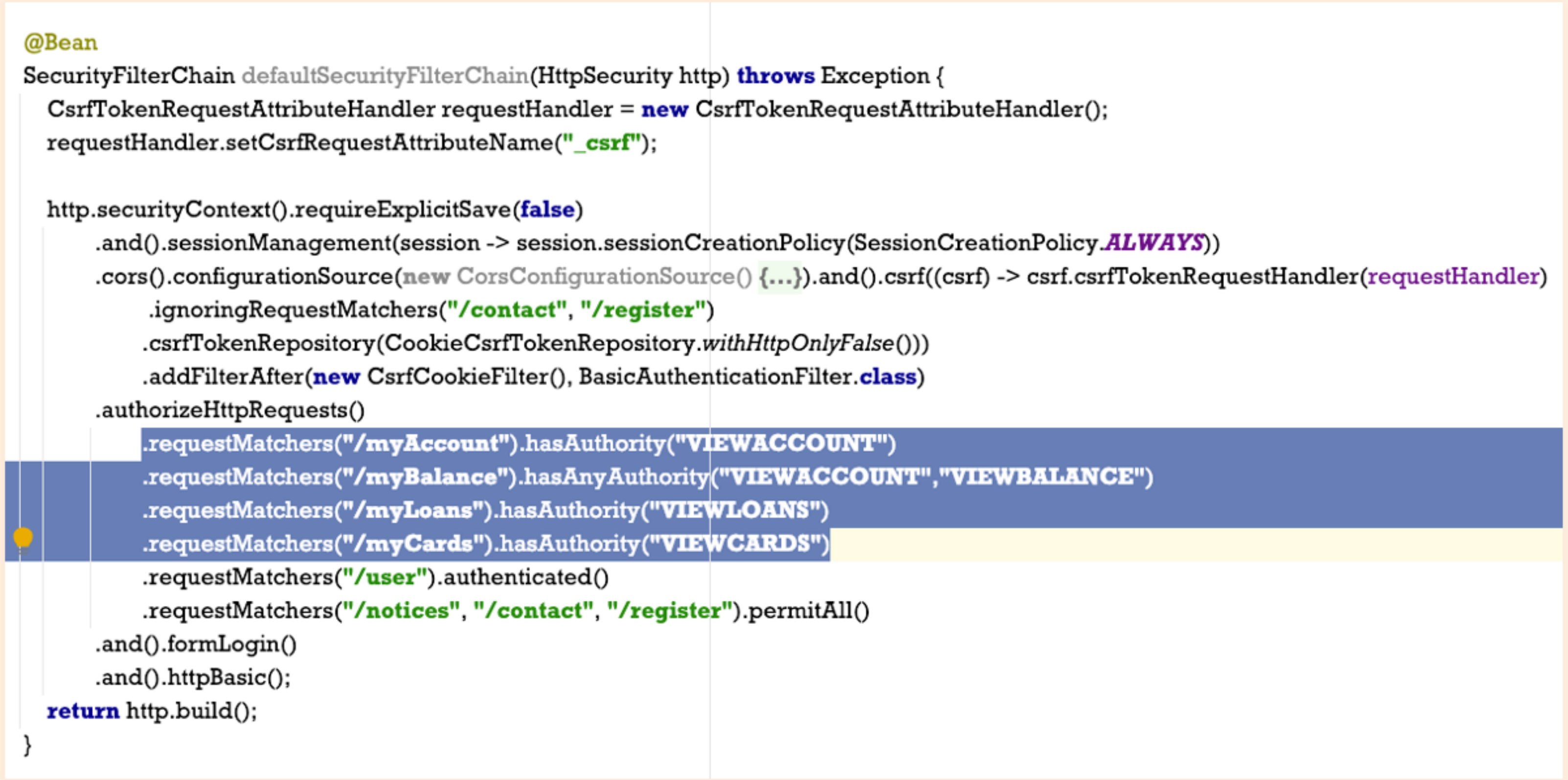
hasAuthority() — Accepts a single authority for which the endpoint will be configured and user will be validated against the single authority mentioned. Only users having the same authority configured can invoke the endpoint.

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

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().requireExplicitSave(false)
            .and().sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
            .cors().configurationSource(new CorsConfigurationSource() {...}).and().csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler)
                    .ignoringRequestMatchers("/contact", "/register")
                    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
                    .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
            .authorizeHttpRequests()
                    .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()
            .and().formLogin()
            .and().httpBasic();
    return http.build();
}
```

AUTHORITY vs ROLE

INSIDE SPRING SECURITY

AUTHORITY

- Authority is like an individual privilege or an action
- Restricting access in a fine-grained manner
- Ex: VIEWACCOUNT,
  VIEWCARDS etc.

ROLE

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

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

CONFIGURING AUTHORITIES

INSIDE SPRING SECURITY

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.

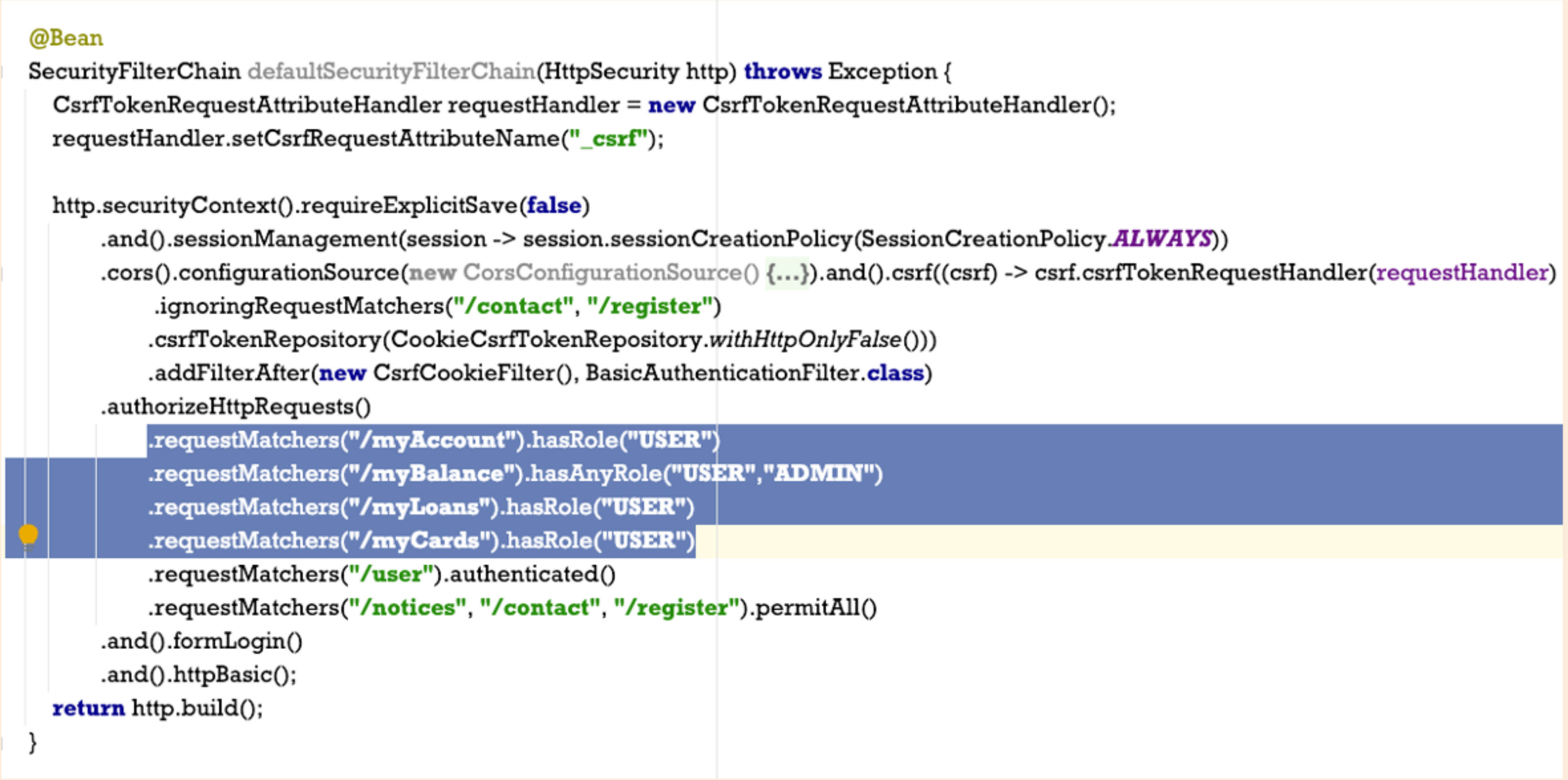
Note :

- 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.
- access() method can be used not only for configuring authorization based on authority or role but also with any special requirements that we have. For example we can configure access based on the country of the user or current time/date.

CONFIGURING ROLES

INSIDE SPRING SECURITY

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().requireExplicitSave(false)
            .and().sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
            .cors().configurationSource(new CorsConfigurationSource() {...}).and().csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler)
                    .ignoringRequestMatchers("/contact", "/register")
                    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
                    .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
            .authorizeHttpRequests()
                    .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()
            .and().formLogin()
            .and().httpBasic();
    return http.build();
}
```

FILTERS IN SPRING SECURITY

✓ Lot of times we will have situations where we need to perform some house keeping activities during the authentication and authorization flow. Few such examples are,

- Input validation
- Tracing, Auditing and reporting
- Logging of input like IP Address etc.
- Encryption and Decryption
- Multi factor authentication using OTP

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

✓ We already saw some in built filters of Spring security framework like 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.

NOT RECOMMENDED FOR PRODUCTION

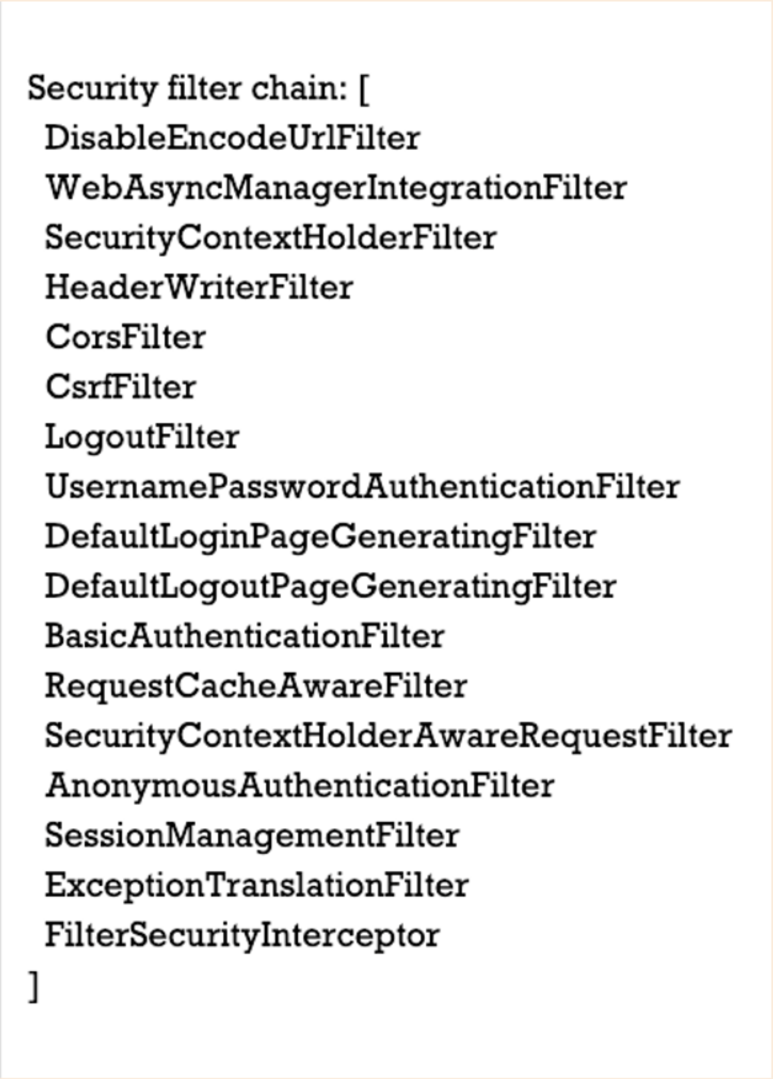
FILTERS IN SPRING SECURITY

✓ 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
  WebAsyncManagerIntegrationFilter
  SecurityContextHolderFilter
  HeaderWriterFilter
  CorsFilter
  CsrfFilter
  LogoutFilter
  UsernamePasswordAuthenticationFilter
  DefaultLoginPageGeneratingFilter
  DefaultLogoutPageGeneratingFilter
  BasicAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  FilterSecurityInterceptor
]
```

IMPLEMENTING CUSTOM FILTERS

INSIDE SPRING SECURITY

✓ 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,

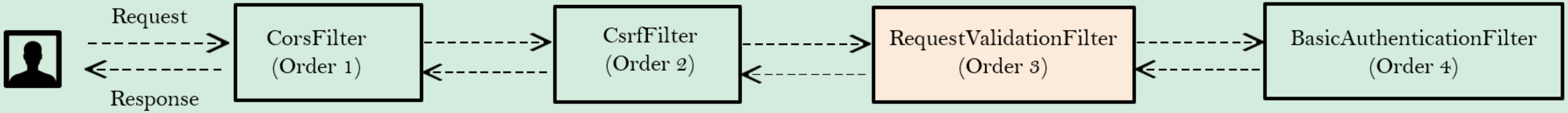
- **addFilterBefore(filter, class)** – adds a filter before the position of the specified filter class
- **addFilterAfter(filter, class)** – adds a filter after the position of the specified filter class
- **addFilterAt(filter, class)** – adds a filter at the location of the specified filter class

ADD FILTER BEFORE

IN SPRING SECURITY

eazy bytes

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

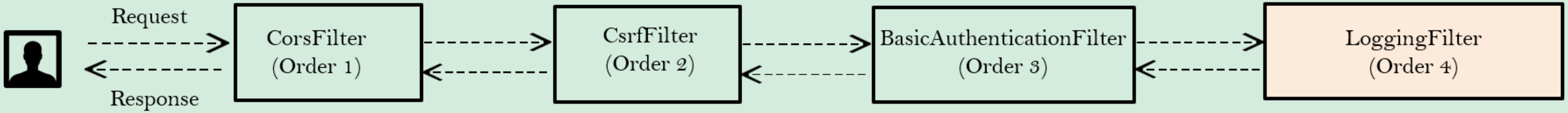

Request
Response
CorsFilter (Order 1)
CsrfFilter (Order 2)
RequestValidationFilter (Order 3)
BasicAuthenticationFilter (Order 4)

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

ADD FILTER AFTER

IN SPRING SECURITY

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

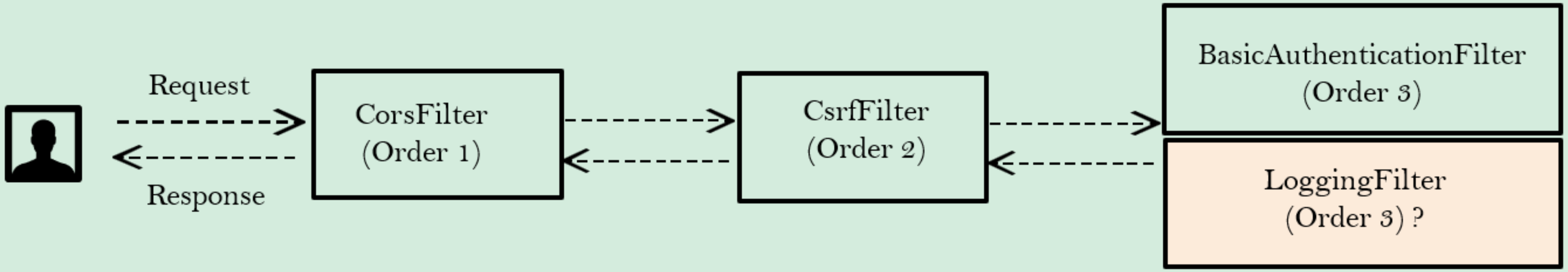


Here we add a filter just after authentication to write a logger about successful authentication and authorities details of the logged in users.

ADD FILTER AT

IN SPRING SECURITY

addFilterAt(filter, class) – Adds a filter at the location of the specified filter class. But the order of the execution can't be guaranteed. This will not replace the filters already present at the same order.



Since we will not have control on the order of the filters and it is random in nature we should avoid providing the filters at same order.

OTHER IMPORTANT FILTERS

GenericFilterBean

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

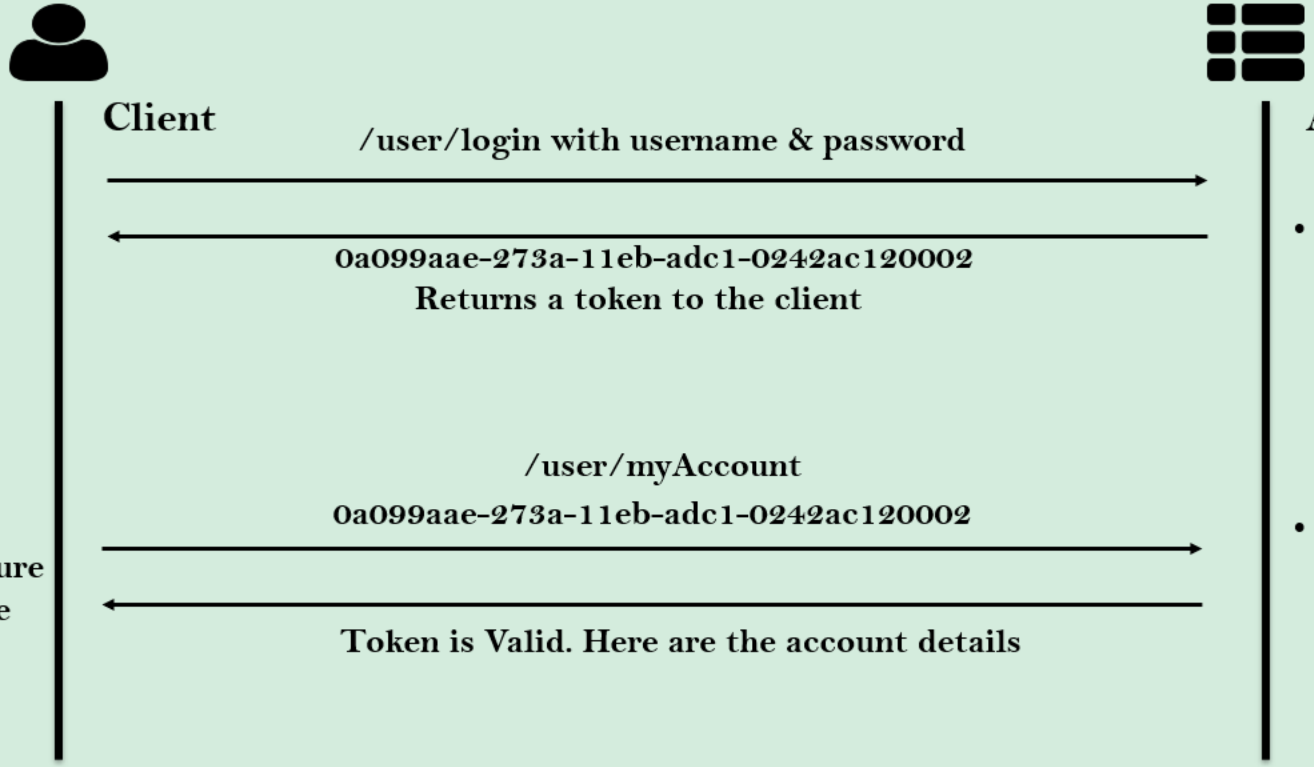
OncePerRequestFilter

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.

ROLE OF TOKENS

IN AUTHN & AUTHZ

- ✓ A Token can be a plain string of format universally unique identifier (UUID) or it can be of type JSON Web Token (JWT) usually that get generated when the user authenticated for the first time during login.
- ✓ On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it's valid, returns the secure resource to the client.



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

- **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.**

ADVANTAGES OF TOKENS

eazy bytes

★ **Token helps us not to share the credentials for every request. It is a security risk to send credentials over the network frequently.**

★ **Tokens can be invalidated during any suspicious activities without invalidating the user credentials.**

★ **Tokens can be created with a short life span.**

★ **Tokens can be used to store the user related information like roles/authorities etc.**

★ **Reusability - We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user.**

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

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

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

JWT TOKENS

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

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

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

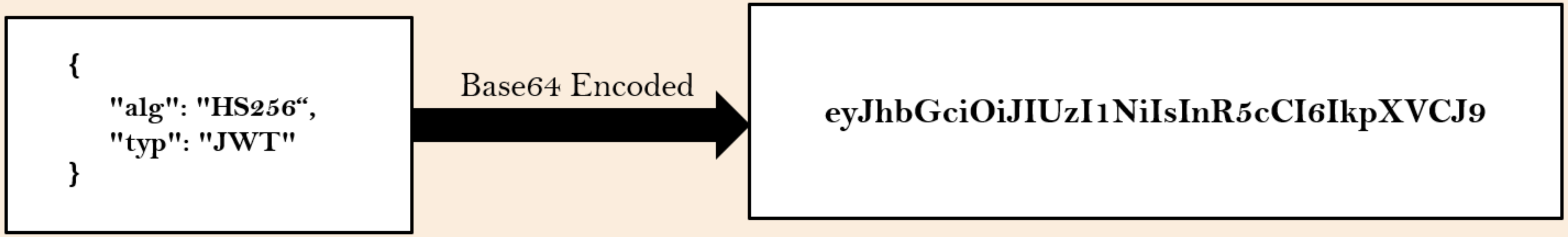
A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

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

JWT TOKENS

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

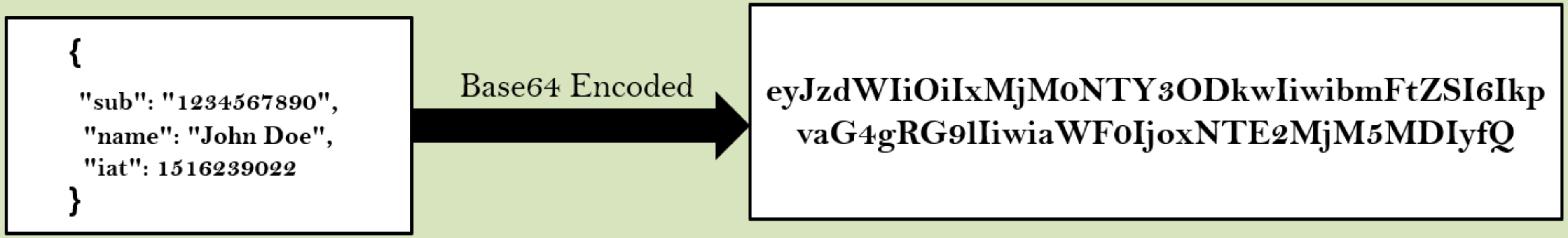


```
{
    "alg": "HS256“,
    "typ": "JWT"
}
```

Base64 Encoded

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

- ✓ 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.



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

Base64 Encoded

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp
vaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

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.

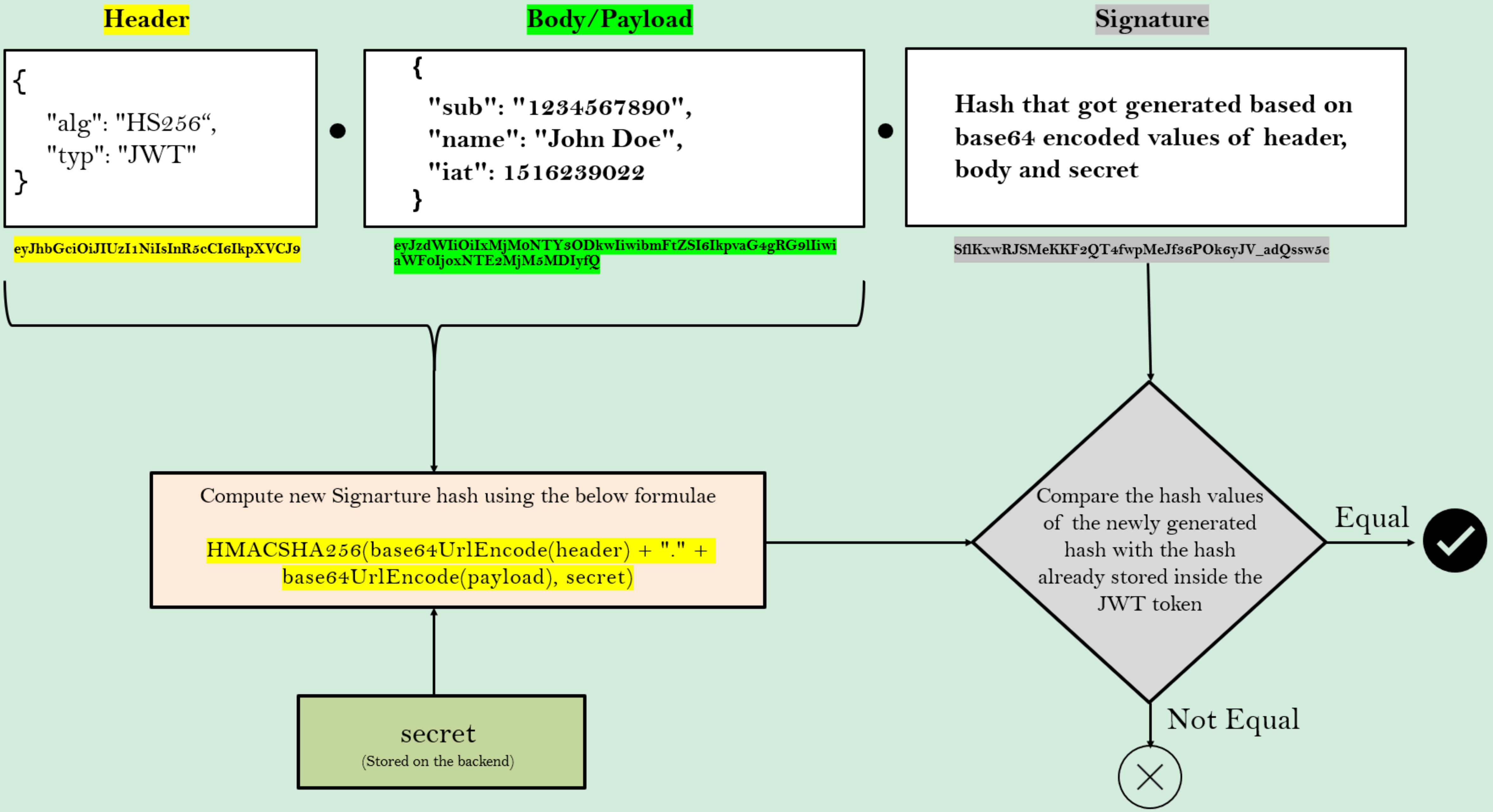
- ✓ For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
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.

VALIDATION OF JWT TOKENS

eazy bytes



Header

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

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Body/Payload

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

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

Signature

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

SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Compute new Signarture hash using the below formulae

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

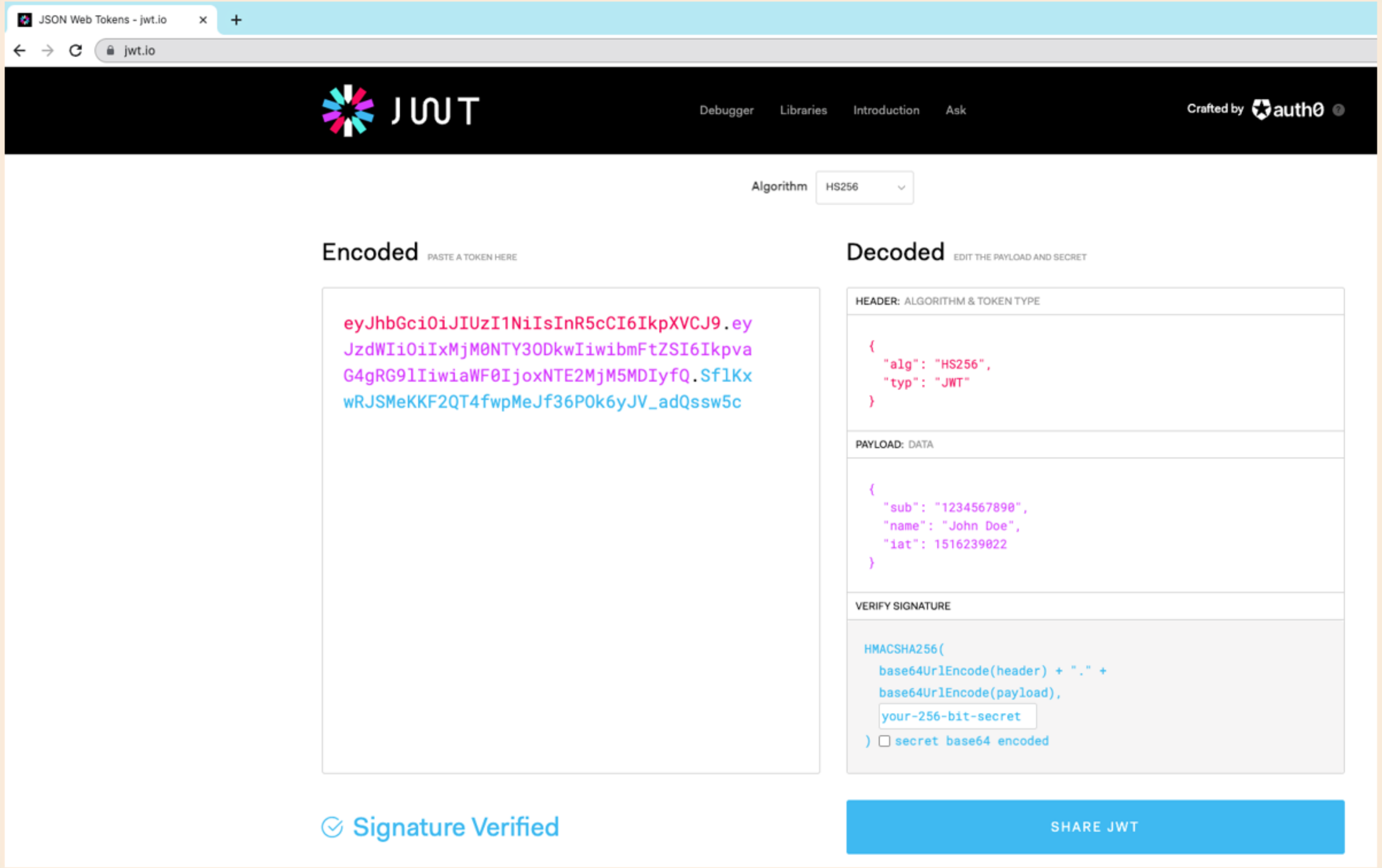
secret

(Stored on the backend)

Compare the hash values of the newly generated hash with the hash already stored inside the JWT token

JWT TOKENS

- ✓ 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.



Algorithm HS256

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG
4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

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

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

✓ Signature Verified

SHARE JWT

METHOD LEVEL SECURITY

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

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

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

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

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

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

METHOD LEVEL SECURITY

- **Using invocation authorization we can decide if a user is authorized to invoke a method before the method executes (preauthorization) or after the method execution is completed (postauthorization). For filtering the parameters before calling the method we can use Prefiltering,**

```
@Service
public class 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

- **For applying postauthorization rules below is the sample configuration,**

```
@Service
public class LoanService {

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

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

METHOD LEVEL SECURITY

- 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

- **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.**

OAUTH2

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

Twitter App

Twitter user

TweetAnalyzer website that analyzes user tweets data and generates metrics from it

✓ **Scenario :** The twitter user want to use an third party website called **TweetAnalyzer,** to get some insights about his tweets data present inside Twitter App.

- **With Out OAUTH2 :** Twitter user has to share his twitter account credentials to the TweetAnalyzer website. Using user credentials, the TweetAnalyzer website will invoke the APIs of Twitter app to fetch the tweet details and post that generates a report for the end user.

  But it has a bigger disadvantage, the **TweetAnalyzer** can go fraud and make another operations on your behalf like change password, change email, make a rouge tweet etc.

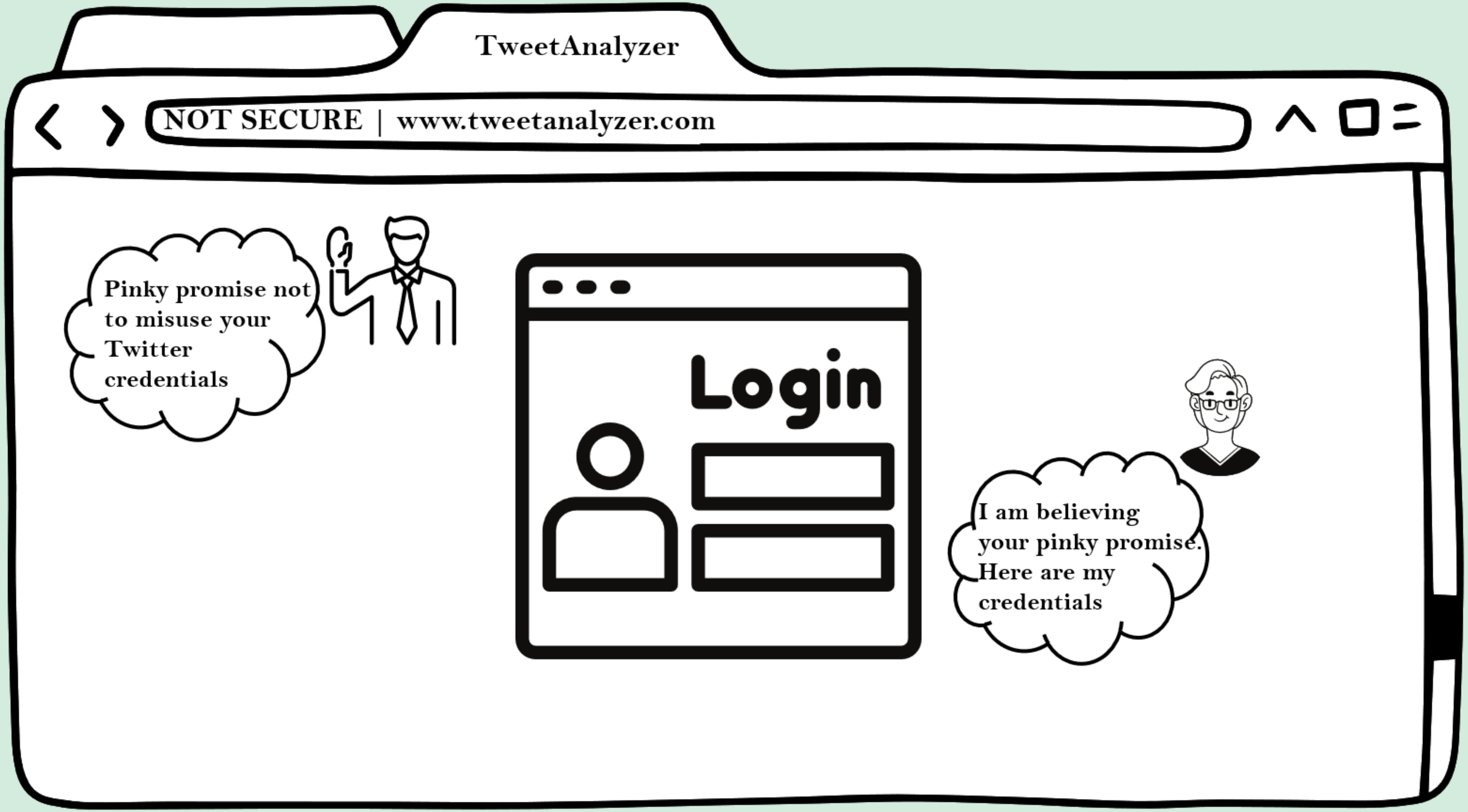
- **With OAUTH2 :** Twitter user doesn't have to share his twitter account credentials to the TweetAnalyzer website. Instead he will let Twitter App to give a temporary access token to TweetAnalyzer with limited access like it can only read the tweets data.

  With this approach, the TweetAnalyzer can only read the tweets data and it can't perform any other operation.

STONE AGE APPROACH

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES


TweetAnalyzer
NOT SECURE | www.tweetanalyzer.com
Pinky promise not to misuse your Twitter credentials
Login
I am believing your pinky promise. Here are my credentials

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

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

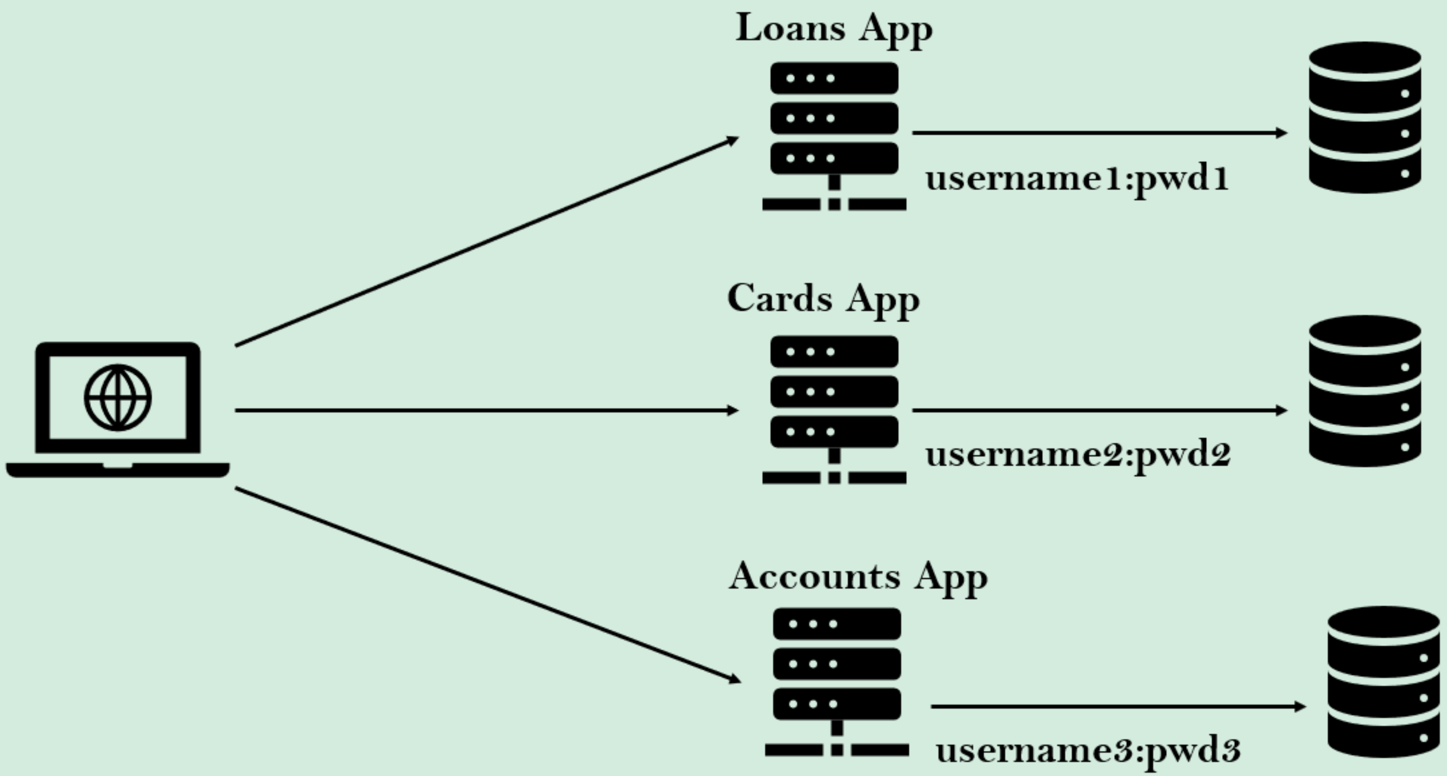
Well the answer is with the help of OAuth2. OAuth2 recommend to use a **separate Auth server** for Authentication & Authorization

INTRO TO OAUTH2

PROBLEM THAT OAUTH2 SOLVES

WITH OUT OAUTH2

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

INTRODUCTION TO OAUTH2

- ✓ **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

Resource owner – It is you the end user. In the scenario of **TweetAnalyzer**, the end user who want to use the **TweetAnalyzer** website to get insights about this tweets. In other words, the end user owns the resources (Tweets), that why we call him as Resource owner

Client – The TweetAnalyzer website is the client here as it is the one which interacts with Twitter 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 **TweetAnalyzer,** the Twitter server which has authorization logic acts as Authorization server.

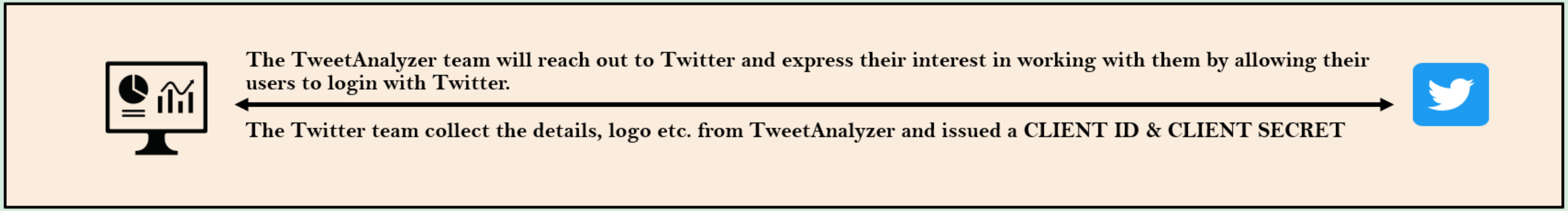
Resource Server – This is the server where the APIs, services that client want to consume are hosted. In the scenario of **TweetAnalyzer,** the Twitter server which has APIs like **/getTweets** etc. logic implemented. In smaller organizations, a single server can acts as both resource server and auth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

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

OAUTH2 SAMPLE FLOW

IN TWEETANALYZER SCENARIO

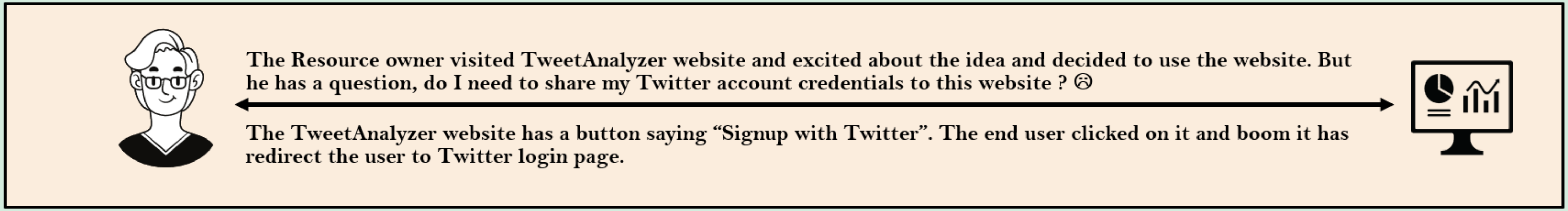
1



The TweetAnalyzer team will reach out to Twitter and express their interest in working with them by allowing their users to login with Twitter.

The Twitter team collect the details, logo etc. from TweetAnalyzer and issued a CLIENT ID & CLIENT SECRET

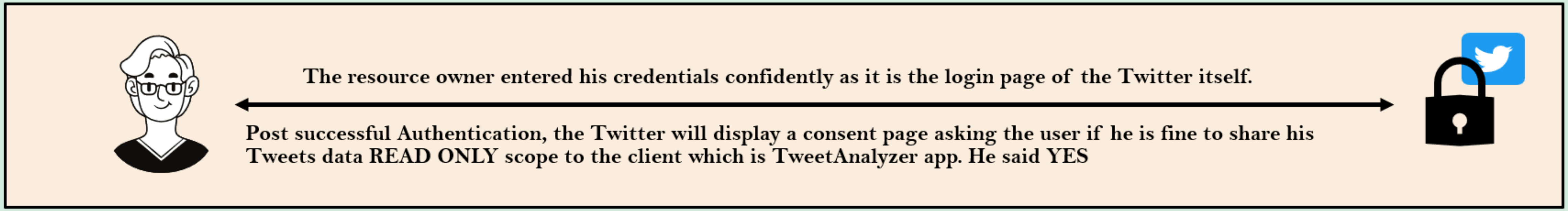
2



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

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

3



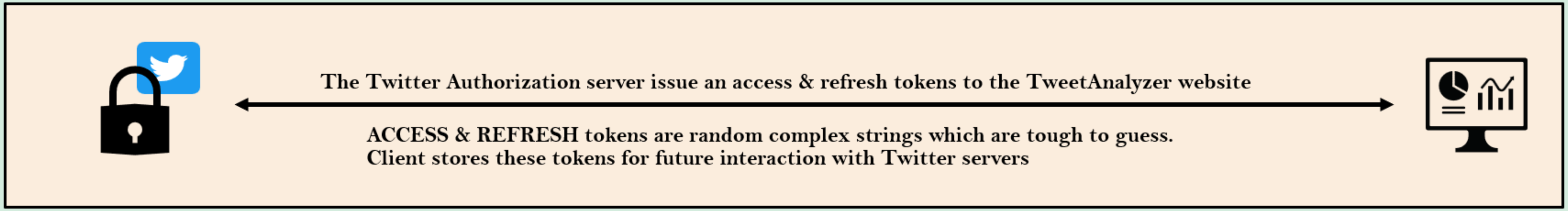
The resource owner entered his credentials confidently as it is the login page of the Twitter itself.

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

OAUTH2 SAMPLE FLOW

IN TWEETANALYZER SCENARIO

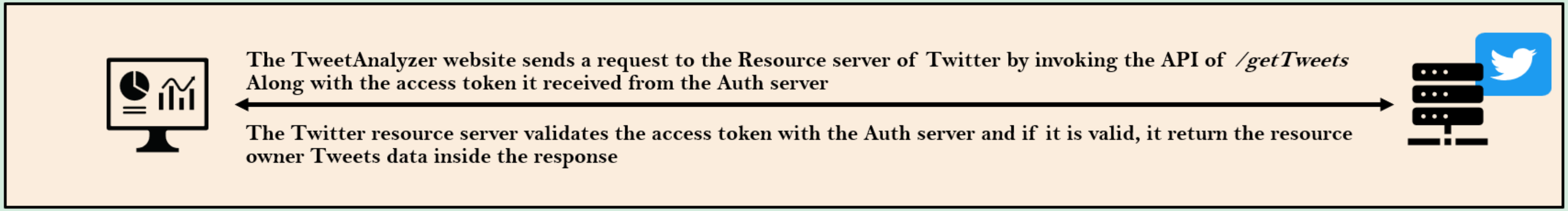
4



The Twitter Authorization server issue an access & refresh tokens to the TweetAnalyzer website

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

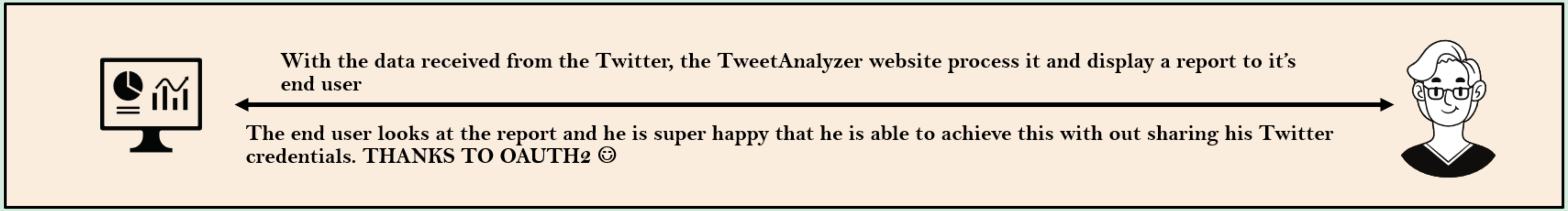
5



The TweetAnalyzer website sends a request to the Resource server of Twitter by invoking the API of */getTweets*
Along with the access token it received from the Auth server

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

6



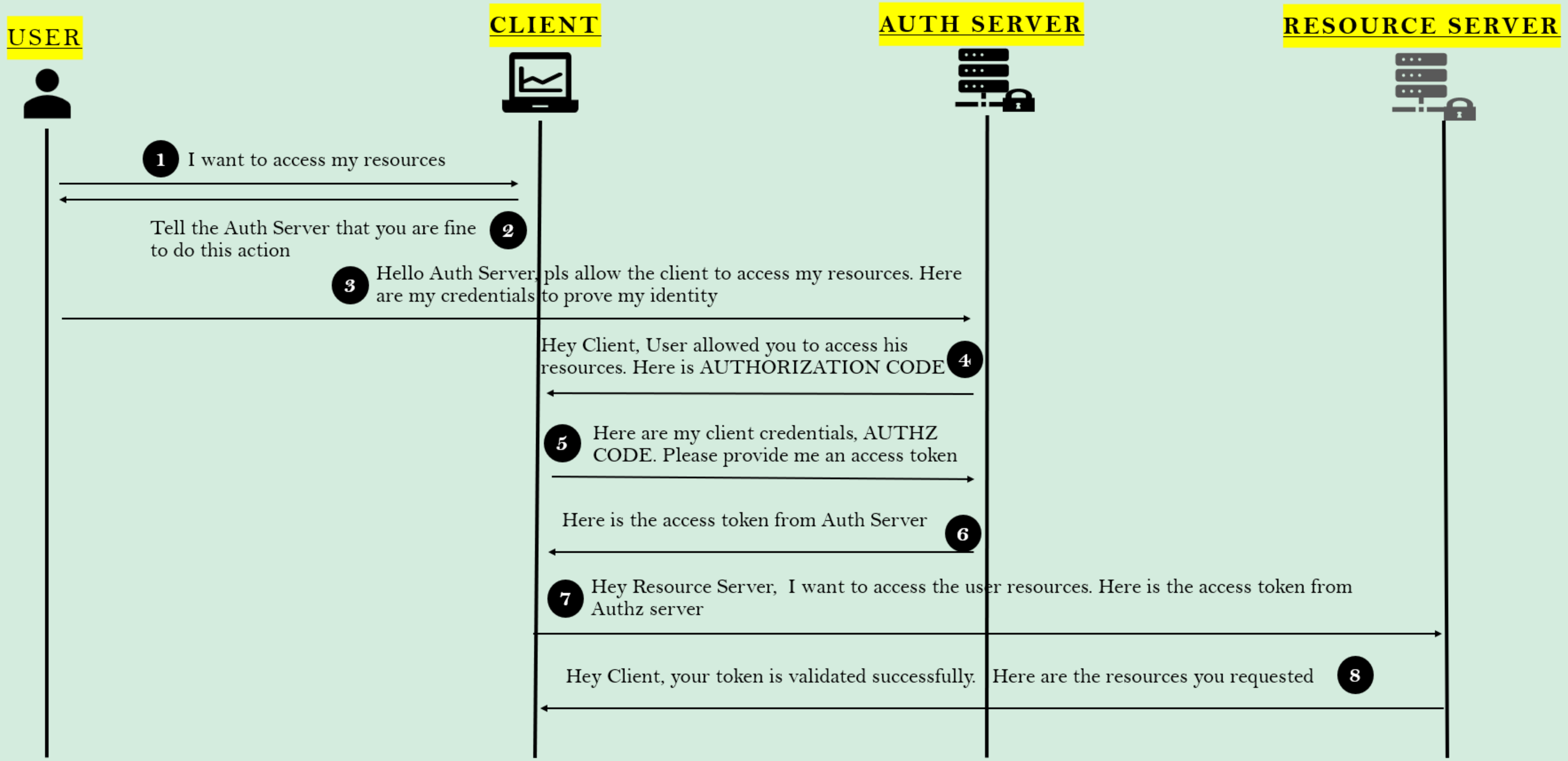
With the data received from the Twitter, the TweetAnalyzer website process it and display a report to it's end user

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

OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

eazy bytes



USER

CLIENT

AUTH SERVER

RESOURCE SERVER

1 I want to access my resources

2 Tell the Auth Server that you are fine to do this action

3 Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity

4 Hey Client, User allowed you to access his resources. Here is AUTHORIZATION CODE

5 Here are my client credentials, AUTHZ CODE. Please provide me an access token

6 Here is the access token from Auth Server

7 Hey Resource Server, I want to access the user resources. Here is the access token from Authz server

8 Hey Client, your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

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

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

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

- **code** – the authorization code received from the above steps
- **client_id** & **client_secret** – the client credentials which are registered with the auth server. Please note that these are not user credentials
- **grant_type** – With the value 'authorization_code' which identifies the kind of grant type is used
- **redirect_uri**

OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

✓ We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.

- In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, auth server send the authorization code to client
- Once it receives the authorization code, in this step client has to prove it's identity along with the authorization code & client credentials to get the access token.

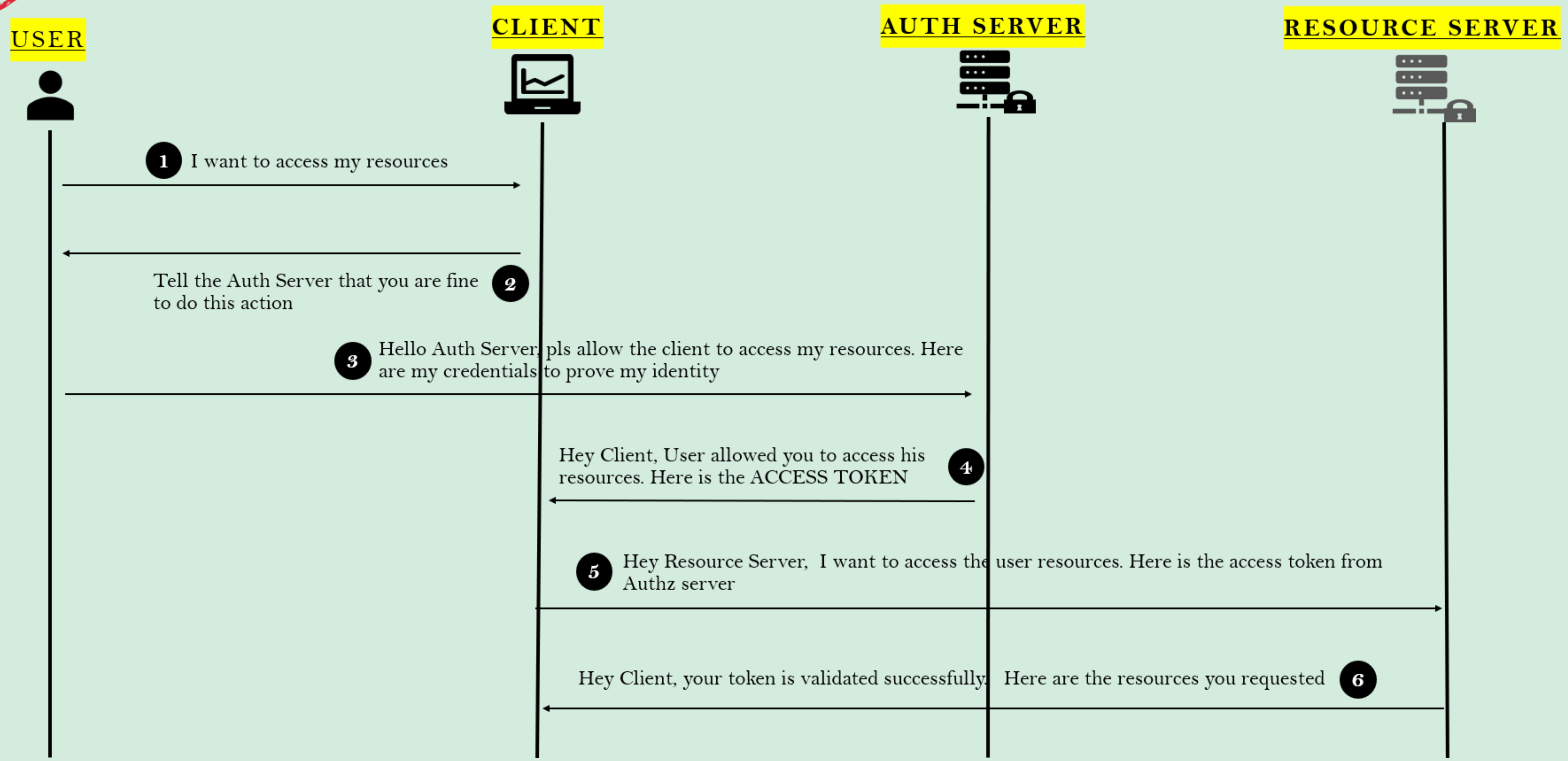
✓ Well you may ask why can't Auth server directly club both the steps together and provide the token in a single step. The answer is that we used to have that grant type as well which is called as **'implicit grant type'**. But this grant type is not recommended to use due to it's less secure.

NOT RECOMMENDED FOR PRODUCTION

OAUTH2 FLOW

IN THE IMPLICIT GRANT FLOW

eazy bytes



USER

CLIENT

AUTH SERVER

RESOURCE SERVER

1 I want to access my resources

2 Tell the Auth Server that you are fine to do this action

3 Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity

4 Hey Client, User allowed you to access his resources. Here is the ACCESS TOKEN

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

6 Hey Client, your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE IMPLICIT GRANT FLOW

✓ 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.

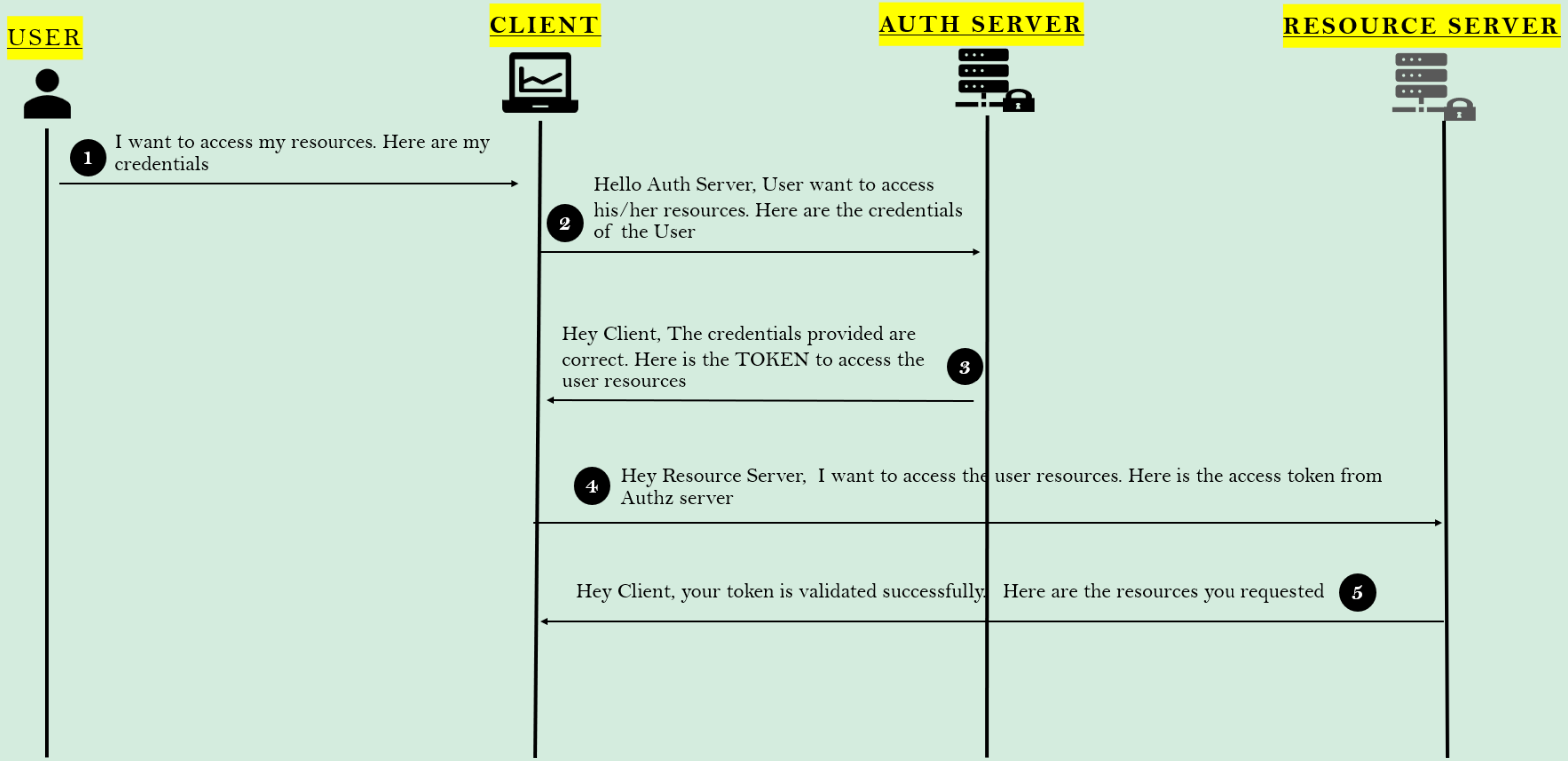
✓ Implicit Grant flow is deprecated and is not recommended to use in production applications. Always use the Authorization code grant flow instead of implicit grant flow.

NOT RECOMMENDED FOR PRODUCTION

OAUTH2 FLOW

IN THE PASSWORD GRANT/RESOURCE OWNER CREDENTIALS GRANT TYPE

eazy bytes



USER

CLIENT

AUTH SERVER

RESOURCE SERVER

1 I want to access my resources. Here are my credentials

2 Hello Auth Server, User want to access his/her resources. Here are the credentials of the User

3 Hey Client, The credentials provided are correct. Here is the TOKEN to access the user resources

4 Hey Resource Server, I want to access the user resources. Here is the access token from Authz server

5 Hey Client, your token is validated successfully. Here are the resources you requested

OAUTH2 FLOW

IN THE RESOURCE OWNER CREDENTIALS GRANT TYPE

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

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

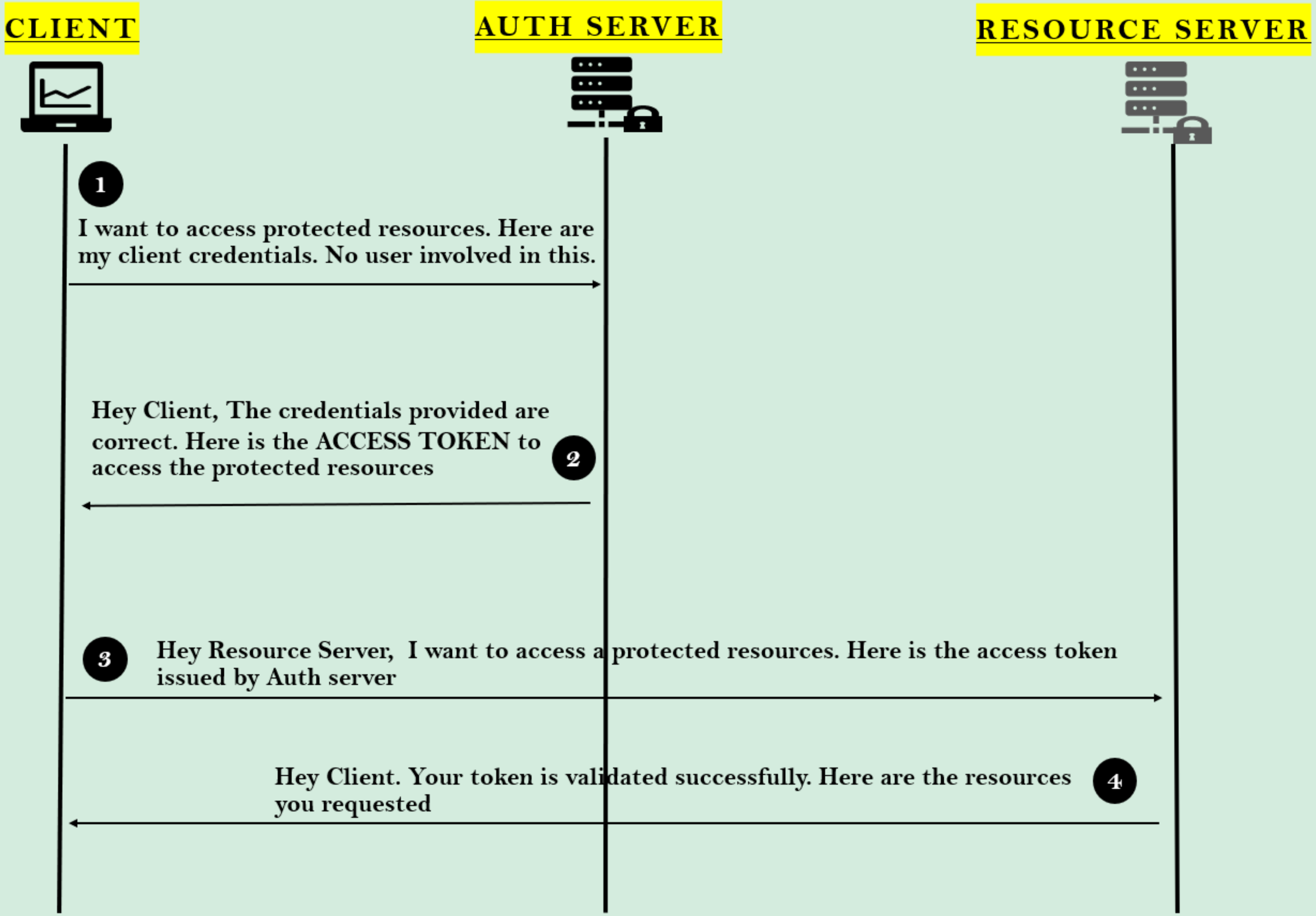
✓ We use this authentication flow only if the client, authorization server and resource servers are maintained by the same organization.

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

eazy bytes

OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE



CLIENT

AUTH SERVER

RESOURCE SERVER

1 I want to access protected resources. Here are my client credentials. No user involved in this.

2 Hey Client, The credentials provided are correct. Here is the ACCESS TOKEN to access the protected resources

3 Hey Resource Server, I want to access a protected resources. Here is the access token issued by Auth server

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

OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE

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

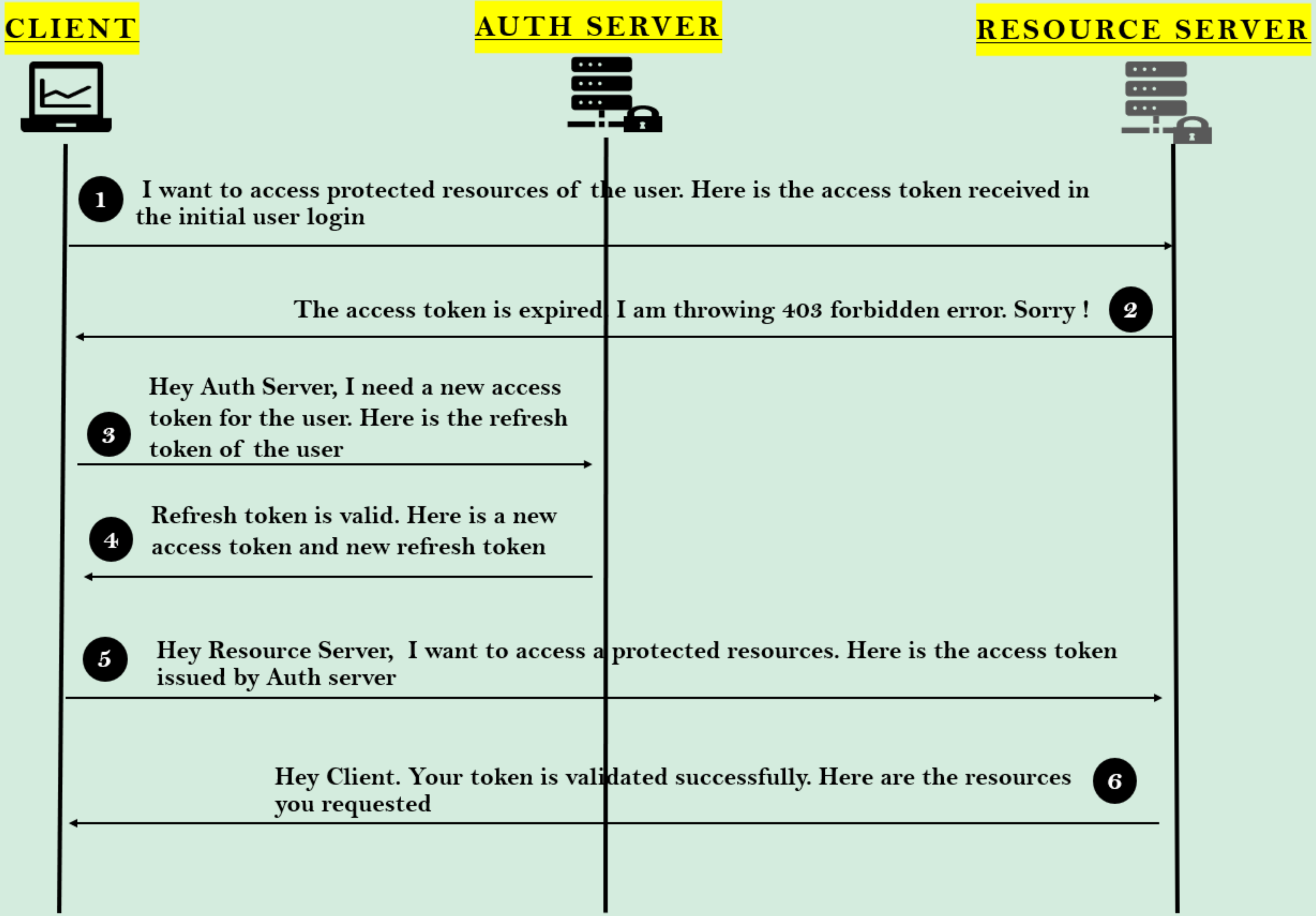
- **client_id** & **client_secret** – the credentials of the client to authenticate itself.
- **scope** – similar to authorities. Specifies level of access that client is requesting like READ
- **grant_type** – With the value **'client_credentials'** which indicates that we want to follow client credentials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

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

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

OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE



CLIENT

AUTH SERVER

RESOURCE SERVER

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

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

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

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

5 Hey Resource Server, I want to access a protected resources. Here is the access token issued by Auth server

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

OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE

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

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

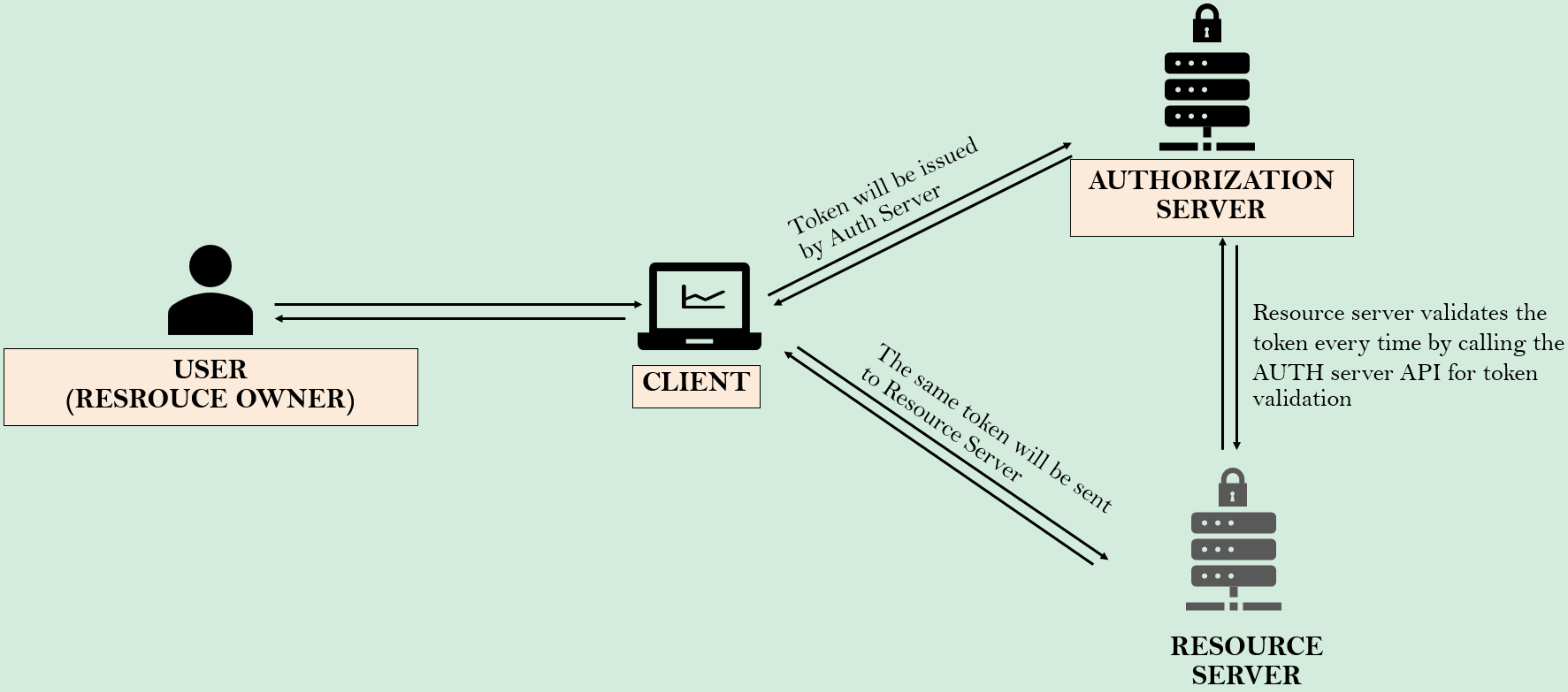
- This flow will be used in the scenarios where the access token of the user is expired. Instead of asking the user to login again and again, we can use the refresh token which originally provided by the Authz server to reauthenticate the user.

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

- Even in the resource owner credentials grant types we should not store the user credentials for reauthentication purpose instead we should reply on the refresh tokens.

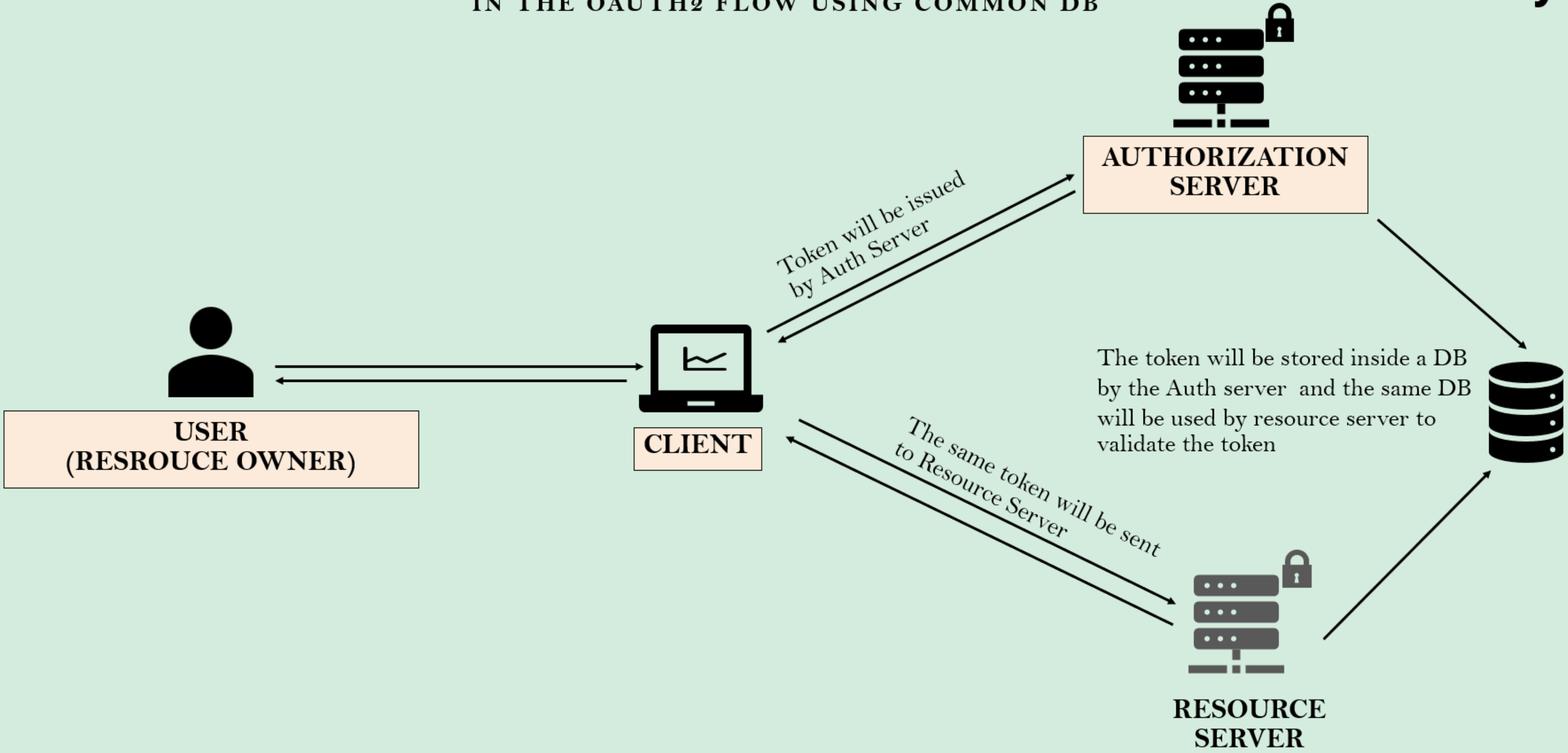
RESOURCE SERVER TOKEN VALIDATION

IN THE OAUTH2 FLOW USING DIRECT API CALL


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

RESOURCE SERVER TOKEN VALIDATION

IN THE OAUTH2 FLOW USING COMMON DB

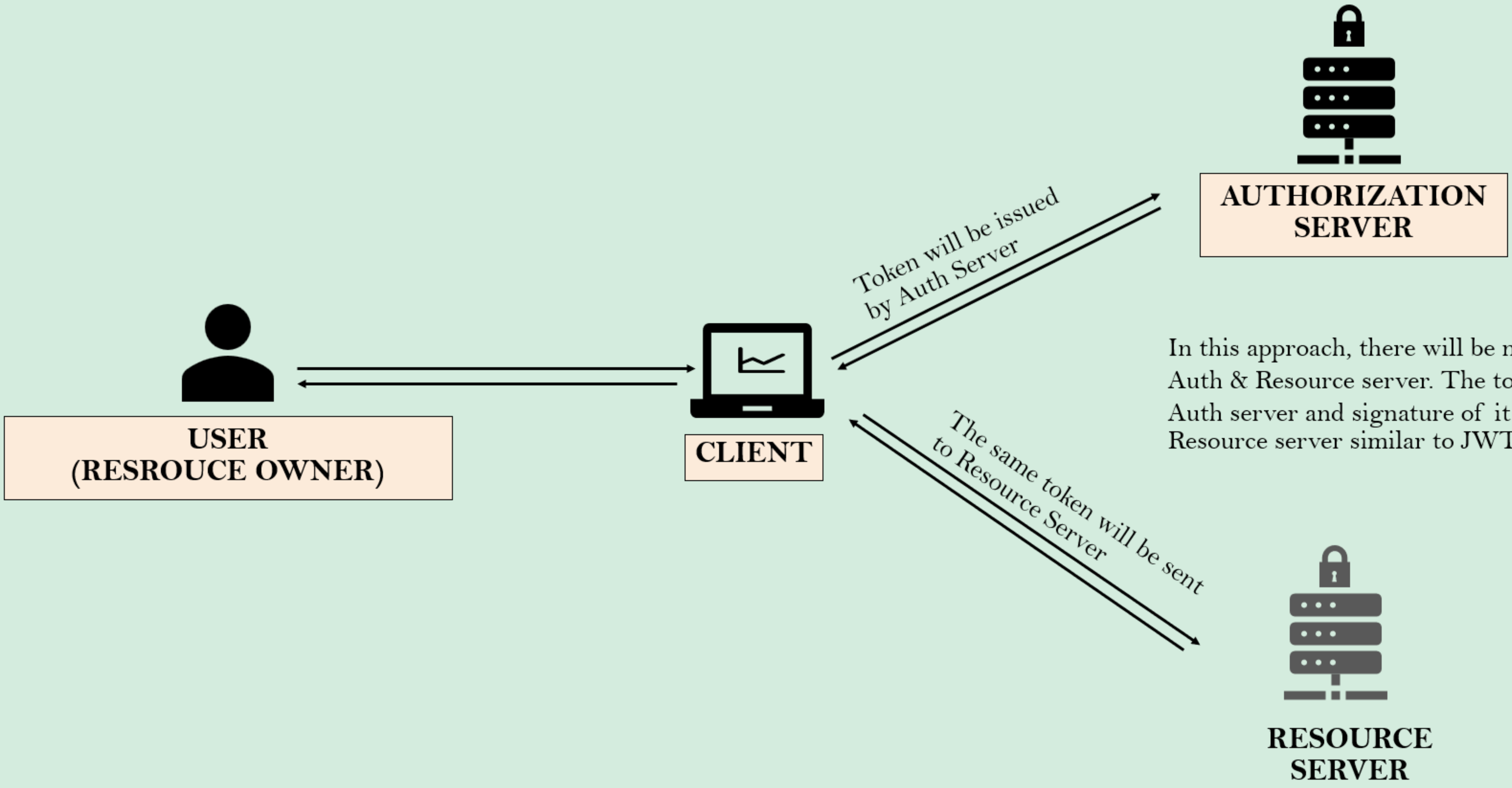

AUTHORIZATION SERVER
Token will be issued by Auth Server
USER (RESROUCE OWNER)
CLIENT
The same token will be sent to Resource Server
RESOURCE SERVER

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

RESOURCE SERVER TOKEN VALIDATION

IN THE OAUTH2 FLOW USING CERTIFICATES

eazy bytes


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

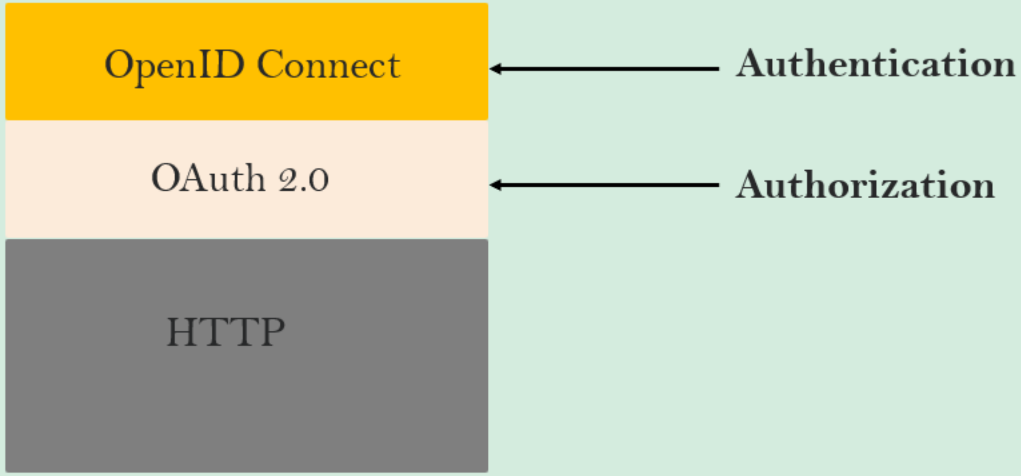
In this approach, there will be no interaction b/w the Auth & Resource server. The token will be signed by Auth server and signature of it will be verified by Resource server similar to JWT tokens.

OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

What is OpenID Connect?

- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of 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

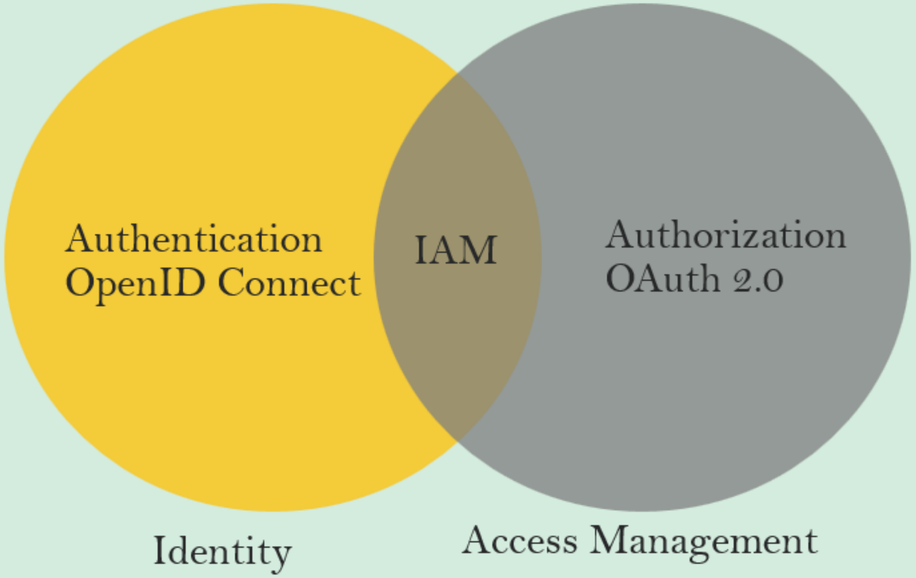
WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

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.

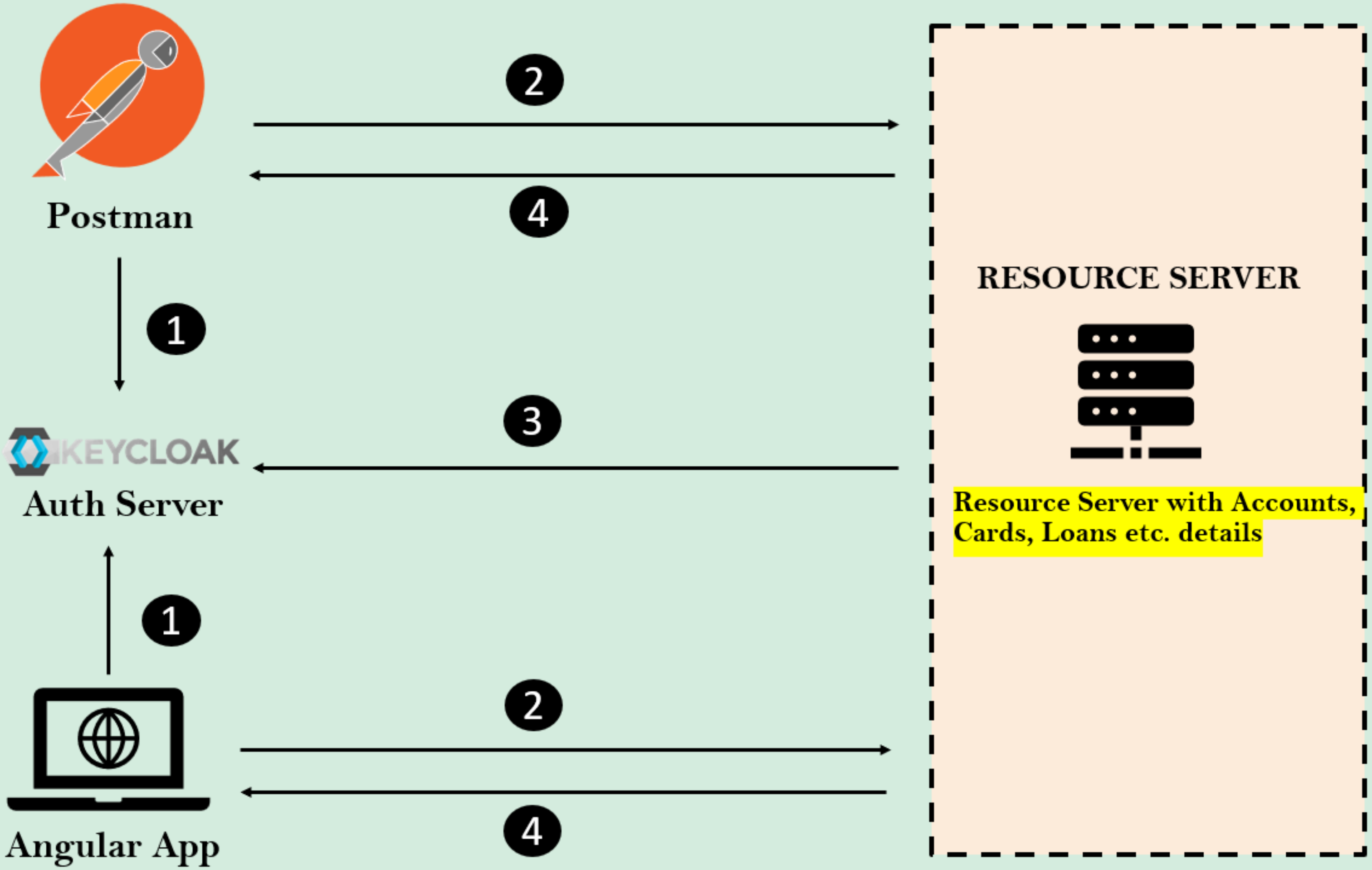


OpenID Connect add below details to OAuth 2.0

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



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.

OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

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

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

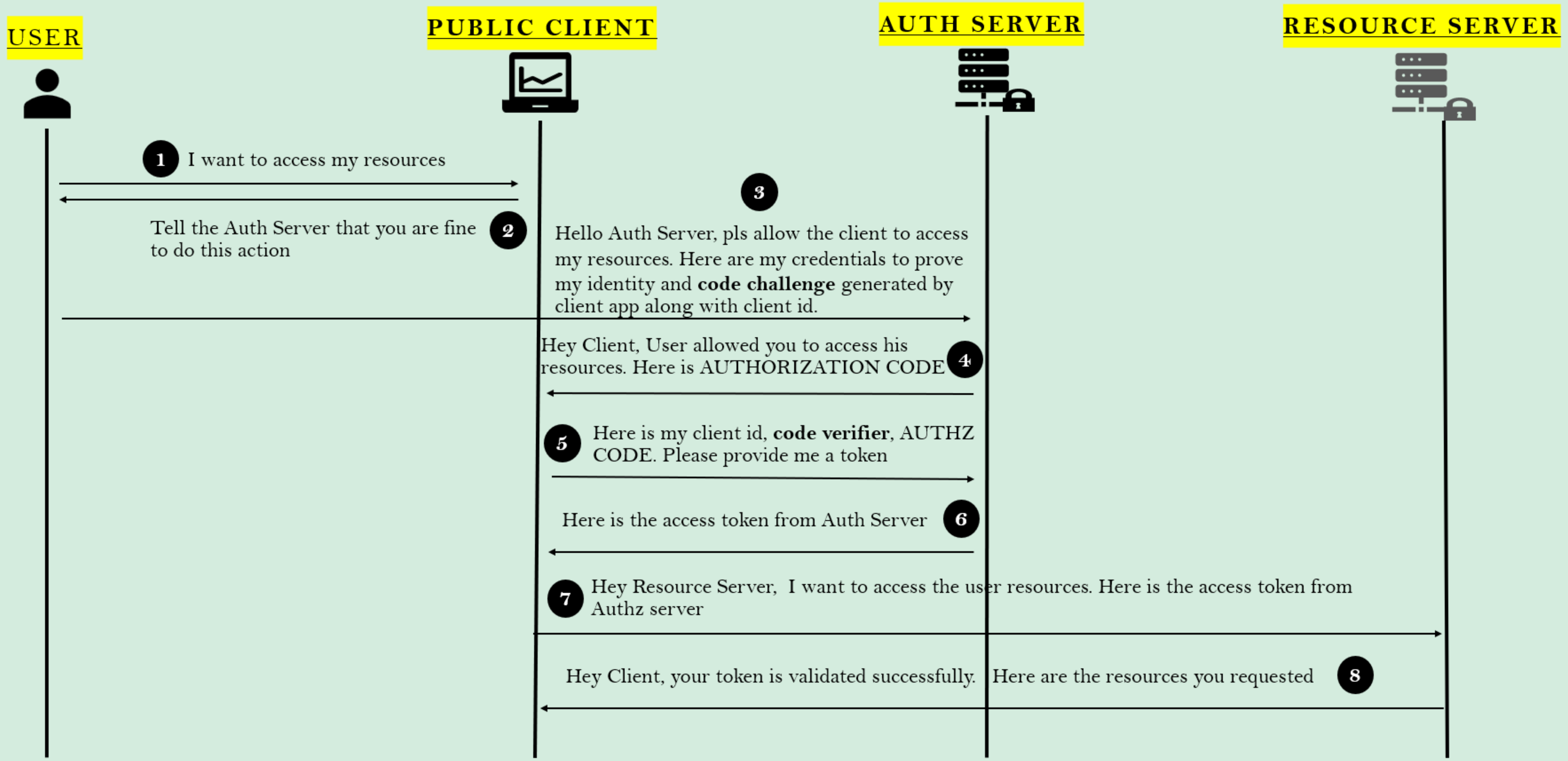
✓ The PKCE-enhanced Authorization Code Flow follows below steps,

- Once user clicks login, client app creates a cryptographically-random **code_verifier** and from this generates a **code_challenge**.
- 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



USER

PUBLIC CLIENT

AUTH SERVER

RESOURCE SERVER

1 I want to access my resources

2 Tell the Auth Server that you are fine to do this action

3 Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity and **code challenge** generated by client app along with client id.

4 Hey Client, User allowed you to access his resources. Here is AUTHORIZATION CODE

5 Here is my client id, **code verifier**, AUTHZ CODE. Please provide me a token

6 Here is the access token from Auth Server

7 Hey Resource Server, I want to access the user resources. Here is the access token from Authz server

8 Hey Client, your token is validated successfully. Here are the resources you requested

OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

✓ 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.

THANK YOU