Spring  
Security

short line

Your Name  
4th September, 20XX

# SHA256

**What SHA-256 Is and Its Output:**

SHA-256 is a hash function that takes an **input of any size** and produces **a fixed-size output**, which is a **256-bit string** of letters and numbers. This output is often referred to as a "fingerprint" or "digest."

**Key Properties of SHA-256 (and Hash Functions in General):**

* **Deterministic Scrambling:** For the same input, SHA-256 will always produce the exact same output. The process scrambles the data in a predictable manner.
* **Quote:** "given the same input it will always produce exactly the same output"
* **Fixed Size Output:** Regardless of the input size (a single letter or the entire Lord of the Rings series), the output will always be 256 bits long.
* **Quote:** "no matter what the size of the input the output will always be of a fixed length"
* **Irreversibility:** Given only the SHA-256 hash (the fingerprint), it should be computationally infeasible to determine the original input. Even a tiny change in the input results in a drastically different output, making reverse engineering extremely difficult.
* **Quote:** "if you only have access to the output the input should be unguessable" and "if just one letter changes in the input the output changes dramatically right it didn't just change by one character it changed like everything changed it seemingly changed in a random and complete way"

**Why SHA-256 is Special:**

1. **256-bit Output Length:** This specific length is a defining characteristic of SHA-256.
2. **Security:** SHA-256 is currently considered secure, meaning no significant exploitable vulnerabilities have been found. Unlike older hash functions like SHA-1 and MD5, no practical methods for easily creating collisions (different inputs producing the same hash) have been discovered.
3. **Speed:** SHA-256 is relatively fast, making it efficient for hashing large amounts of data quickly. However, this speed makes it less suitable for password hashing, where slower functions are preferred to hinder brute-force attacks.

**Real-World Use Cases of SHA-256:**

* **Anti-Virus Software:** When software is published (e.g., word.exe), a SHA-256 fingerprint is often provided alongside it. When a user downloads the software, their computer can calculate the SHA-256 hash of the downloaded file and compare it to the published fingerprint. If they match, it confirms the integrity of the file and assures the user it hasn't been tampered with or infected with malware.
* **Authentication (Auth):** SHA-256 plays a role in authentication processes. Websites often use it to generate authentication tokens. For example, upon successful login, a server might take a user's identifier (like a username) and append a secret key (known only to the server). This combined data is then hashed using SHA-256 to create a unique token. This token is then used to verify the user's identity in subsequent requests without needing to send the password repeatedly. The irreversibility of SHA-256 ensures that even if an attacker intercepts the token, they cannot easily derive the secret key or other sensitive information.
* **Blockchains (e.g., Bitcoin):** Bitcoin utilizes SHA-256 as its hashing algorithm. In a blockchain, each block of transactions contains a hash of its own data, as well as the hash of the previous block. This chaining of hashes ensures the integrity and immutability of the blockchain. If any data in a previous block is altered, its hash will change, which in turn will affect the hash of all subsequent blocks, making tampering easily detectable. The computational cost of generating new blocks makes it extremely difficult for malicious actors to retroactively change transaction history.

**Conclusion:**

SHA-256 is a fundamental cryptographic hash function with crucial properties like deterministic scrambling, fixed output size, and irreversibility. These properties make it invaluable for various real-world applications, including verifying file integrity, securing authentication processes, and underpinning the security of blockchain technologies like Bitcoin. While fast and efficient for many purposes, its speed makes it less suitable for password hashing, where slower, more computationally intensive functions are preferred. The continued security and widespread adoption of SHA-256 highlight its importance in modern digital security.

# Basic Web App Security

**Servlets in Java**

* A **Servlet Container** (like Tomcat or JBoss) handles HTTP requests and responses.
* When a browser sends a request (HTTP/HTTPS), the container converts it into a ServletRequest or HttpServletRequest object.
* This object is passed to a **Servlet**, which interacts with frameworks like Spring Boot and Spring Security.
* The servlet invokes your **business logic** (controllers, services).
* The response is processed similarly using a ServletResponse, which is then converted back to HTTP before sending to the client.

**Why Frameworks Help**

* Before frameworks like Spring Boot, developers had to write all web logic manually using servlets.
* Now, frameworks handle the boilerplate, letting developers focus on writing business logic with minimal configuration.

**Filters in Java**

* **Filters** intercept every HTTP request **before** it reaches the servlet.
* They're used to inspect, modify, or block requests and responses.
* In Spring Security, filters handle **authentication, authorization, and redirection** to login pages.

**Role of Filters in Spring Security**

* Spring Security uses a **chain of filters** to process every request.
* These filters check if the user is authenticated, authorized, or should be redirected.
* This mechanism ensures **security logic runs before** any business logic.

# Spring Security internal flow

## Filter Interface

**What is a Servlet Filter?** A Servlet Filter is part of the Java EE Servlet specification. It allows preprocessing and postprocessing of HTTP requests and responses in a web application.

*public interface* Filter {  
 *void* init(FilterConfig filterConfig) *throws* ServletException;  
 *void* doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
 *throws* IOException, ServletException;  
 *void* destroy();  
}

**Lifecycle Methods:**

* **init(FilterConfig config)**
  + Called once when the web application starts.
  + Used for initializing resources, configs, logging, etc.
* **doFilter(request, response, chain)**
  + Called for each matching HTTP request.
  + Core logic goes here.
  + Filters can:
    - Modify the request/response.
    - Pass control to the next filter using chain.doFilter().
    - Short-circuit the chain (e.g., by sending a response directly).
* **destroy()**
  + Called once during app shutdown.
  + Used to release resources initialized in init().

**How the Servlet Container Uses Filters**

Filter Registration

1. Via web.xml:

<filter>  
 <filter-name>authFilter</filter-name>  
 <filter-class>com.example.AuthFilter</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>authFilter</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

1. Via Annotation:

@WebFilter("/\*")  
*public class* AuthFilter *implements* Filter { ... }

**Execution Flow:**

1. Container starts → Instantiates all filters.
2. Calls init() on each filter.
3. For every incoming HTTP request:
   * Executes filters **in the order** defined.
   * Each filter decides whether to:
     + Pass the request to the next filter (chain.doFilter()).
     + Or handle it and stop the chain.

**How Spring Security Uses Filters**

Spring Security integrates into the servlet container’s filter system using two key components:

**A. DelegatingFilterProxy**

* A standard javax.servlet.Filter registered with the servlet container.
* Delegates filtering logic to a **Spring-managed bean**.
* Acts as a bridge between the servlet world and the Spring application context.

@Bean  
*public* FilterRegistrationBean<DelegatingFilterProxy> securityFilterChain() {  
 FilterRegistrationBean<DelegatingFilterProxy> registration = *new* FilterRegistrationBean<>();  
 registration.setFilter(*new* DelegatingFilterProxy("springSecurityFilterChain"));  
 registration.addUrlPatterns("/\*");  
 *return* registration;  
}

* It delegates to the bean named springSecurityFilterChain, which is typically an instance of FilterChainProxy.

**FilterChainProxy (Spring Security’s Filter Chain Manager)**

* + The **main filter** that manages multiple security filter chains.
  + Each SecurityFilterChain contains filters like:
    - SecurityContextPersistenceFilter
    - UsernamePasswordAuthenticationFilter
    - ExceptionTranslationFilter
    - CsrfFilter
    - And others...

**Execution:** When FilterChainProxy#doFilter() is called, it:

1. **Matches the incoming request** with an appropriate SecurityFilterChain.
2. **Creates a VirtualFilterChain** containing matched Spring Security filters.
3. Executes each filter in order.

**What is VirtualFilterChain?**

* A Spring-internal class that simulates the Servlet's FilterChain.
* Manages Spring Security’s internal filter sequence and invokes them one by one.

Example Flow : Assume: filters = [FilterA, FilterB, FilterC]

The flow:

* FilterChainProxy#doFilter() → Creates VirtualFilterChain
* FilterA#doFilter() is called → calls chain.doFilter() (i.e., VirtualFilterChain)
* VirtualFilterChain → Calls FilterB#doFilter()
* And so on…
* After the last filter, control returns to the servlet (e.g., your controller).

*@Override  
public void* doFilter(ServletRequest request, ServletResponse response) *throws* IOException, ServletException {  
 *if* (*this*.currentPosition == *this*.size) {  
 *this*.originalChain.doFilter(request, response);  
 *return*;  
 }  
 *this*.currentPosition++;  
 Filter nextFilter = *this*.additionalFilters.get(*this*.currentPosition - 1);  
 *if* (logger.isTraceEnabled()) {  
 String name = nextFilter.getClass().getSimpleName();  
 logger.trace(LogMessage.format("Invoking %s (%d/%d)", name, *this*.currentPosition, *this*.size));  
 }  
 nextFilter.doFilter(request, response, *this*);  
}

**Filters vs. Interceptors**:

* Filters: Servlet-level, sit outside Spring MVC, work on all requests.
* Interceptors: Spring MVC-specific, can access handler/controller info.

| **Order** | **Filter** | **Purpose** |
| --- | --- | --- |
| 1 | WebAsyncManagerIntegrationFilter | Integrates SecurityContext with async web requests. |
| 2 | SecurityContextPersistenceFilter | Loads/Saves the SecurityContext (e.g., from session). |
| 3 | HeaderWriterFilter | Writes HTTP security headers (HSTS, CSP, etc.). |
| 4 | CorsFilter *(if defined)* | Handles CORS preflight and headers. |
| 5 | CsrfFilter | Validates CSRF tokens on state-changing HTTP methods. |
| 6 | LogoutFilter | Processes logout URLs. |
| 7 | OAuth2AuthorizationRequestRedirectFilter | Redirects to OAuth2 provider login (if configured). |
| 8 | Saml2WebSsoAuthenticationRequestFilter | Starts SAML 2.0 login flow (if configured). |
| 9 | X509AuthenticationFilter | Authenticates based on client certificates. |
| 10 | AbstractPreAuthenticatedProcessingFilter | Handles requests already authenticated by an external system. |
| 11 | CasAuthenticationFilter | CAS login filter (if CAS is used). |
| 12 | UsernamePasswordAuthenticationFilter | Authenticates login form credentials. |
| 13 | OpenIDAuthenticationFilter | Legacy OpenID login (rarely used now). |
| 14 | DefaultLoginPageGeneratingFilter | Generates login page if none provided. |
| 15 | DefaultLogoutPageGeneratingFilter | Generates logout page if none provided. |
| 16 | BearerTokenAuthenticationFilter | Authenticates using Bearer (JWT) token in header. |
| 17 | BasicAuthenticationFilter | Authenticates using HTTP Basic Auth header. |
| 18 | RequestCacheAwareFilter | Saves the original request and restores it post-login. |
| 19 | SecurityContextHolderAwareRequestFilter | Adds security-specific methods to the request (e.g., isUserInRole). |
| 20 | AnonymousAuthenticationFilter | Provides anonymous auth when no user is authenticated. |
| 21 | SessionManagementFilter | Handles invalid or expired sessions. |
| 22 | ExceptionTranslationFilter | Handles exceptions (like 403 or redirect to login). |
| 23 | FilterSecurityInterceptor | Performs authorization (access control check). |

## Spring Security flow

A diagram of a security management system

AI-generated content may be incorrect.

**Spring Security components flow:**

1. **Client Request**: A browser, mobile app, or Postman sends a request to access a protected REST API or MVC path.
2. **Spring Security Filters**:
   1. Intercepts every request (over 20 filters exist). Checks if the user is authenticated.
   2. If unauthenticated i.e no *Anthentication* object is present in requet and accessing a protected resource, spring redirects to the login. By default *DefaultLoginPageGeneratingFilter* is used to generate the page
3. **Authentication Object Creation**:
   1. Filters extract credentials from the request and create an *Authentication* object (with username, password, isAuthenticated=false).
   2. By default it uses *UsernamePasswordAuthenticationFilter*
      1. This is a concrete implementation of *AbstractAuthenticationProcessingFilter*.
      2. It reads the username and password from the HttpServletRequest in the *attemptAuthentication* method
      3. It creates a *UsernamePasswordAuthenticationToken*(implementation of *Authentication*) Token with these credentials.
      4. This token is unauthenticated at this point (setAuthenticated(false)).
   3. This object is passed to Authentication manager via authenticate() method of Authentication manager

return this.getAuthenticationManager().authenticate(authRequest);

1. **Authentication Manager**: 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
   1. Receives the *Authentication* object and delegates actual authentication to available AuthenticationProviders.
   2. The default implementation is *ProviderManager*.
   3. The *ProviderManager* iterates through a list of AuthenticationProviders and selects the appropriate one.
   4. For username/password-based login, this is typically the *DaoAuthenticationProvider*.
2. **Authentication Providers**:
   1. *DaoAuthenticationProvider* calls retrieveUser(), which delegates to a *UserDetailsService* (like *InMemoryUserDetailsManager* or *JdbcUserDetailsManager*) to fetch user details.
   2. Use UserDetailsService or UserDetailsManager to load user details by loadUserByUsername.
3. **Use PasswordEncoder** to compare user-provided and stored passwords securely.
4. **Authentication Success/Failure**:
   1. On success: isAuthenticated=true is set in the Authentication object.
5. Updated object is returned to filters
6. **Security Context**:
   1. Stores the authenticated object against a session ID.(Need to check where it is stored)
   2. For future requests with the same session, authentication isn’t re-processed data is reused from the context.
7. **Response to Client**:
   1. If authenticated, the client gets the API response.
   2. If authentication fails, a 401 or 403 error is returned.

Different Componenets of Spring Security

1. **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.
2. **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.
3. **AuthenticationManager** Once received request from filter, it delegates the validating of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the AuthenticationManager to manage all the authentication providers available. In simple words, the authentication manager takes the responsibility for authentication.
4. **AuthenticationProvider** AuthenticationProviders has all the core logic of validating user details for authentication.
5. **UserDetailsManager/UserDetailsService** UserDetailsManager/UserDetailsService helps in retrieving, creating, updating, deleting the User Details from the DB/storage systems.
6. **PasswordEncoder** Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords ®
7. **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.

Below will set the security logging level to Trace by default. It can be overwritten using the environment variable also

logging.level.org.springframework.security=${SPRING\_SECURITY\_LOG\_LEVEL:TRACE}

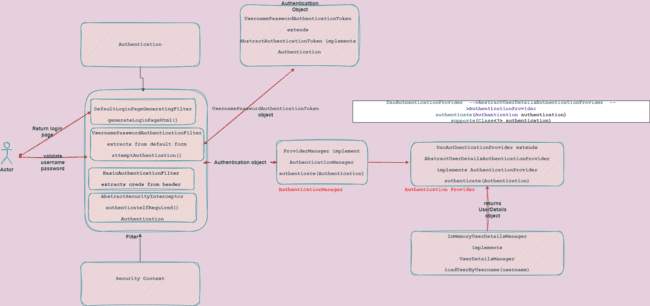
When a user interacts with a secured Spring Boot application (e.g., tries to log in), Spring Security processes the request using a **filter chain** composed of various filters, each responsible for part of the security logic.

**Different classes in Spring Security**

1. **Filter Chain Execution (DelegatingFilterProxy & SecurityFilterChain)**
2. These filters are invoked in a defined order (can be seen via TRACE logging) and include:
   1. DefaultLoginPageGeneratingFilter
   2. UsernamePasswordAuthenticationFilter
   3. AuthorizationFilter
   4. ExceptionTranslationFilter, etc.
3. **User Initiates Login (via /login form)**  
   When the login form is submitted, it hits the endpoint intercepted by the UsernamePasswordAuthenticationFilter.
4. **Filter: UsernamePasswordAuthenticationFilter**
   1. This is a concrete implementation of AbstractAuthenticationProcessingFilter.
   2. It reads the username and password from the HttpServletRequest.
   3. It creates a UsernamePasswordAuthenticationToken(implementation of Authentication) with these credentials.
   4. This token is *unauthenticated* at this point (setAuthenticated(false)).
5. **Authentication Manager Delegation**
   1. The filter passes the token to an AuthenticationManager via its authenticate() method.
   2. The default implementation is ProviderManager.
6. **Provider Selection: DaoAuthenticationProvider**
   1. The ProviderManager iterates through a list of AuthenticationProviders and selects the appropriate one.
   2. For username/password-based login, this is typically the DaoAuthenticationProvider.
7. **User Lookup: UserDetailsService**
   1. DaoAuthenticationProvider calls retrieveUser(), which delegates to a UserDetailsService (like InMemoryUserDetailsManager or JdbcUserDetailsManager) to fetch user details.
8. **Password Verification**
   1. Password matching is performed using a PasswordEncoder (like BCryptPasswordEncoder).
   2. It compares the raw password entered with the hashed password stored.
9. **Successful Authentication**
   1. If credentials are valid, Spring creates a fully authenticated Authentication object (e.g., UsernamePasswordAuthenticationToken with setAuthenticated(true)).
   2. This object is stored in the SecurityContext, which is kept in the SecurityContextHolder.
10. **Security Context Persistence**
    1. The SecurityContext is stored in the HTTP session so that subsequent requests don’t need to go through the authentication flow again.
11. **Authorization Flow**
    1. For each subsequent request, the filters (like AuthorizationFilter) inspect the security context to validate the user’s roles and access rights for the requested resource.
    2. If the user lacks access, an AccessDeniedException is thrown and handled appropriately (e.g., redirect to error page).

On subsequent requests:

* When a user first accesses a **protected page**, they are **redirected to the login page** and must enter credentials.
* After **successful authentication**, Spring Security **doesn't ask for credentials again** for future protected requests.
* This is possible because Spring Security sets a **JSESSIONID cookie**, which maintains the user's **authenticated session**.
* As long as this cookie is valid, the user remains authenticated, and Spring Security uses the **stored security context** tied to that session.
* If the cookie is **tampered with or altered**, the session becomes invalid, and the user is **redirected to login again**.
* Once logged in again, a **new JSESSIONID** is issued and linked to the authenticated session.



Important Classes

1. **org.springframework.security.core.userdetails.User** This Class will contain the User details fetched from DB/Memory/Application properties like username, password, authorities(Roles), accountExpired. accountLocked etc.
2. org.springframework.security.core.userdetails.UserDetails Interface implemented by user
3. org.springframework.security.core.userdetail.UserDetailsService Interface which exposes loadUserByUsername method to fetch User
4. **org.springframework.security.provisioning.UserDetailsManager** Interface which provides a lot of other Methods to create User, update user, change password etc.
5. **Authentication** This Class on other side of authentication flow which contains the username password details entered by user after comparison isAuthenticated() is set as true and password removed using eraseCredentials()

# Overriding Security Configuration

Default Security Configuration:

Spring Security uses a class internally (e.g., SpringBootWebSecurityConfiguration) to apply default security settings.

*@Configuration*(proxyBeanMethods = *false*)  
*@ConditionalOnDefaultWebSecurity  
static class* SecurityFilterChainConfiguration {  
  
 *@Bean  
 @Order*(SecurityProperties.BASIC\_AUTH\_ORDER)  
 SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) *throws* Exception {  
 http.authorizeHttpRequests((requests) -> requests.anyRequest().authenticated());  
 http.formLogin(withDefaults());  
 http.httpBasic(withDefaults());  
 *return* http.build();  
 }

This includes a method defaultSecurityFilterChain(HttpSecurity http) with below cofigurations

* **@Order(SecurityProperties.BASIC\_AUTH\_ORDER)** - This sets the priority of this security filter chain. SecurityProperties.BASIC\_AUTH\_ORDER is a predefined constant in Spring Security (typically with value 100) that determines when this filter chain should be applied relative to other security configurations.
* **SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception** - This method declares a security filter chain with the HttpSecurity object injected by Spring.
* **http.authorizeHttpRequests((requests) -> requests.anyRequest().authenticated())** - any HTTP request to any endpoint requires authentication
* **http.formLogin(withDefaults())** - This enables form-based authentication with default settings. It creates a login page at /login and handles login form submissions.
* **http.httpBasic(withDefaults())** - This enables HTTP Basic Authentication as a fallback authentication method, which is useful for API access or non-browser clients.
* **return http.build()** - This builds and returns the complete security filter chain with all the configurations applied.

What It Means:

* Any request must be authenticated.
* Users are redirected to a login page for UI-based apps or use HTTP Basic Auth for APIs.
* This explains the behavior we see in the browser: all endpoints are protected by default.

Comment out the @Bean on the customer SecurityFilterChain defined in SecurityConfig.java and access any endpoint like <http://localhost:8080/private/balance> or <http://localhost:8080/public/home> . As all are protected you will get Spring default login page. Since we have UserDetailsService bean configured we need to login using the credentials mentioned in the bean. Since there are a lot of Additional filters you will not get the page but login can be succcessful.

## Permit Deny all

* **permitAll() with anyRequest()** :Makes all endpoints publicly accessible — **no security at all**.
* **denyAll() with anyRequest()**: Blocks all endpoints — returns **403 Forbidden** even after authentication. Shows login page first, then denies access.

We can override these configuration by selectively allowing or endpoints or making them authenticated

http.authorizeHttpRequests(authorizationManagerRequestMatcherRegistry -> authorizationManagerRequestMatcherRegistry  
 .requestMatchers("/favicon.ico", "/public/style.css", "/public/main.js", "/images/\*\*","/public/home.html", "/h2-console/\*\*").permitAll()  
 .requestMatchers(HttpMethod.POST, "/v1/api/login", "/v2/api/login", "/v3/api/login").permitAll()  
 .requestMatchers("/private/balance", "/private/message", "/admin/announcement", "/admin/loan").authenticated()  
 .requestMatchers("/public/home", "/public/contact", "/error", "/public/myLogin","/public/me").permitAll()  
 )

When you configure a Spring Security filter chain with multiple rules, the order matters significantly and URL patterns are matched in the order they're defined. The first matching rule applies.

http.authorizeHttpRequests(authorize -> authorize  
 .requestMatchers("/public/\*\*").permitAll() *// This applies first for /public/\*\* paths* .requestMatchers("/\*\*").authenticated() *// This applies to everything else* );

In this example:

* Requests to /public/\*\* will be permitted without authentication
* All other requests (/\*\*) will require authentication

If we keep like below then everything would require authentication because the /\*\* pattern (which matches everything) is evaluated first and the second rule would never be reached. Spring Security follows an "first-match-wins" approach, so properly ordering your rules is crucial for implementing the security policy you want.

http.authorizeHttpRequests(authorize -> authorize  
 .requestMatchers("/\*\*").authenticated() *// This would match everything first* .requestMatchers("/public/\*\*").permitAll() *// This would never be reached!* );

## Form Login and HTTP basic

By default, both formLogin() and httpBasic() are enabled in a Spring Boot web app.

**httpBasic()**

1. **Using a Browser**: When you access a protected endpoint, the browser will show a basic login dialog. Enter your username and password. <http://localhost:8080/private/balance>
2. **Using cURL** curl -u admin:qwerty <http://localhost:8080/private/balance>
3. **Using HTTP Headers**: Send an "Authorization" header with the value "Basic" followed by the base64-encoded string of "username

echo -n "admin:qwerty" | base64

curl -H "Authorization: Basic YWRtaW46cXdlcnR5" <http://localhost:8080/private/balance>

To access the API with Postman using HTTP Basic Authentication, follow these steps:

1. Open Postman and create a new request
2. Enter your URL: http://localhost:8080/private/balance
3. Go to the "Authorization" tab
4. From the Type dropdown, select "Basic Auth"
5. Enter your credentials:
   1. Username: admin
   2. Password: qwerty

Postman will automatically handle the base64 encoding and add the appropriate Authorization header to your request.

Alternatively, you can manually set the header:

1. Go to the "Headers" tab
2. Add a new header:
   * Key: Authorization
   * Value: Basic YWRtaW46cXdlcnR5

Note that HTTP Basic Authentication:

* Sends credentials with every request
* Base64 encoding is NOT encryption (it's easily decoded)
* Should be used with HTTPS to secure the credentials in transit
* Doesn't provide logout functionality (credentials are cached by the browser)
* Shouldbe disabled if you are using UI based login

**formLogin()**

* When this is enabled and user accesses an authenticated page. Spring will generate a default login page using DefaultLoginPageGeneratingFilter
* In REST API-based applications (especially consumed by other services or mobile apps), formLogin() is unnecessary. Hence, it is **recommended to disable it**.
* To disable formLogin(), avoid using deprecated methods like .withDefaults() or .formLogin(). Instead, use a lambda with .formLogin(form -> form.disable()).
* Disabling formLogin() ensures that the login page is not shown. Authentication will then be handled by BasicAuthenticationFilter, not UsernamePasswordAuthenticationFilter.

**Internals:**

* **Form Login**: Extracts credentials using UsernamePasswordAuthenticationFilter from HttpServletRequest.
* **HTTP Basic**: Uses BasicAuthenticationFilter, which looks for the Authorization header (e.g., Basic base64(username:password)).
* The credentials are Base64-decoded and parsed into a UsernamePasswordAuthenticationToken.

**Testing Behavior:**

* After disabling formLogin(), browser triggers httpBasic() login with a popup.
* Disabling both formLogin() and httpBasic() leads to **403 Forbidden**, as no authentication method is available.

**Best Practice:**

* Enable/disable authentication mechanisms based on project needs.

## Session Management

When authentication succeeds:

1. **Session Creation**: If a session doesn't exist yet, Spring creates an HTTP session.
2. **JSESSIONID Cookie**: The server sends a JSESSIONID cookie to the client, which is the session identifier.
3. **SecurityContext Storage**: The SecurityContextHolder stores the authenticated principal in the SecurityContext.
4. **SecurityContext Persistence**: The SecurityContextPersistenceFilter saves the SecurityContext to the session.

**Subsequent Requests**

For each subsequent request:

1. **Cookie Transmission**: The browser automatically sends the JSESSIONID cookie.
2. **Session Retrieval**: The server identifies the session using the JSESSIONID.
3. **SecurityContext Restoration**: The SecurityContextPersistenceFilter retrieves the SecurityContext from the session and places it in the SecurityContextHolder.
4. **Authorization**: The FilterSecurityInterceptor uses the authenticated principal from the SecurityContext to make authorization decisions.
5. **Cleanup**: After request processing, the SecurityContextHolder is cleared.

If we tamper with JSESSIONID the request will fail

**JSESSIONID Generation and Storage**

1. **Random Generation**:
   * 1. The JSESSIONID is a cryptographically secure random value
     2. It's typically a long string like 3A1G6E23ABC15679131415E3E28106A8
     3. The randomness is important to prevent session hijacking attacks
2. **Server-Side Storage**:
   * 1. When the server generates a JSESSIONID, it creates an entry in the session store
     2. This session store is usually an in-memory map maintained by the servlet container (Tomcat, Jetty, etc.)
     3. The structure is essentially: Map<String, HttpSession> where the key is the JSESSIONID
3. **Spring Security Context**:
   * 1. Spring Security stores the SecurityContext object in this HTTP session
     2. Specifically, it's stored as an attribute within the session with a predetermined key

In a distributed system with multiple servers, the standard JSESSIONID and SecurityContext approach faces significant challenges. Here's how session management typically works in such environments:

**Challenges in Distributed Systems**

1. **Session Affinity Problem**:
   * 1. The JSESSIONID stored in one server's memory isn't accessible to other servers
     2. If a user's request goes to Server A first and then Server B, authentication will be lost
2. **Server Failure Issues**:
   * 1. If the server holding a session fails, all users authenticated on that server lose their sessions
     2. This creates poor user experience and potential security issues

**Common Solutions for Distributed Systems**

**1. Sticky Sessions (Session Affinity)**

* Configure the load balancer to route all requests from the same user to the same server
* Uses the JSESSIONID to determine routing
* **Drawbacks**: Creates uneven load distribution and fails if the specific server goes down

User -> Load Balancer (reads JSESSIONID) -> Always routes to Server A

**2. Centralized Session Store**

* Store sessions in a shared external system accessible by all application servers
* Common implementations:
  + Redis session store
  + Database-backed sessions
  + Memcached

**3. JWT (JSON Web Tokens)**

* Stateless approach where authentication data is encoded in a signed token sent to the client
* No server-side session storage required
* Each request contains all the necessary authentication information

**4. Spring Session**

* Spring Session provides a standardized way to address distributed sessions. Spring Session transparently replaces the container's HttpSession implementation, allowing session data to be stored in:
  + Redis
  + JDBC (database)
  + MongoDB
  + Hazelcast
  + Other distributed stores

Typical Production Setup : A typical enterprise setup might look like:

Client -> Load Balancer -> Multiple App Servers -> Shared Redis Session Store

1. When a user authenticates, the session is stored in Redis
2. The JSESSIONID cookie is sent to the browser
3. On subsequent requests, any server can retrieve the session from Redis using the JSESSIONID
4. Each server can then reconstruct the SecurityContext for that user

This approach provides both high availability and scalability while maintaining session integrity.

**Additional Considerations**

* **Session Replication**: Some app servers support direct session replication between nodes
* **Serialization**: Security contexts must be serializable for distributed storage
* **Token Expiration**: Proper timeout policies must be implemented
* **Cookie Settings**: Use secure, HTTP-only cookies with appropriate domain settings
* **Session Invalidation**: Proper logout mechanisms must invalidate sessions across all servers

## Setting Custom Login

So far we have been using login provided by Spring. But we can’t use that for production Grade app we can use our own custom login page

http.authorizeHttpRequests(authorizationManagerRequestMatcherRegistry -> authorizationManagerRequestMatcherRegistry  
 .requestMatchers("/favicon.ico", "/public/style.css", "/public/main.js", "/images/\*\*","/public/home.html", "/h2-console/\*\*").permitAll()  
 .requestMatchers(HttpMethod.POST, "/v1/api/login", "/v2/api/login", "/v3/api/login").permitAll()  
 .requestMatchers("/private/balance", "/private/message", "/admin/announcement", "/admin/loan").authenticated()  
 .requestMatchers("/public/home", "/public/contact", "/error", "/public/myLogin","/public/me").permitAll()  
 )  
 .formLogin(form -> form  
 .loginPage("/public/home.html")  
 .loginProcessingUrl("/login")  
 .defaultSuccessUrl("/public/home.html", *false*)  
 .failureUrl("/public/home.html?error")  
 .permitAll()  
 )

1. **.requestMatchers("/favicon.ico", "/public/style.css", "/public/main.js", "/images/\*\*","/public/home.html", "/h2-console/\*\*").permitAll()**  Allowing all the html resources
2. **loginPage("/public/home.html")** This tells Spring Security where to show the custom login page. When an unauthenticated user tries to access a protected resource, they'll be redirected to /public/home.html. You must ensure this URL is publicly accessible (which you’ve already permitted in your authorizeHttpRequests).
3. **loginProcessingUrl("/login")** This defines the endpoint that Spring Security will watch for login form submissions. Your HTML login form should POST to this URL:

        <div class="form-wrapper">

            <h3>Form Login</h3>

            <form method="POST" action="/login">

                <label>Username:</label>

                <input type="text" name="username" required><br>

                <label>Password:</label>

                <input type="password" name="password" required><br><br>

                <button type="submit">Login</button>

            </form>

        </div>

Spring intercepts this POST and performs authentication automatically (you don’t need a controller for this).

1. **defaultSuccessUrl("/public/home.html", true)** This sets the page to redirect to after a successful login.

* If false (or omitted), Spring would: Redirect back to the originally requested protected URL (if any), or fall back to this one.
* If true Spring always redirects to this URL after login URL mentioned , regardless of what page the user was trying to access.
* It is useful if we want to redirect to some other page. In our case we want to redirect to requester page /public/home.html So true false is same
* If entire configuration is removed it will redirect to localhost:8080

1. **failureUrl("/public/home.html?error")** This sets the URL to redirect to when login fails. If the user enters wrong credentials, Spring Security redirects them to this page. You can check for the presence of ?error in your page to show an appropriate message:

*// Handle error and logout messages*

    const urlParams = new URLSearchParams(window.location.search);

    if (urlParams.has('error')) {

        document.getElementById('status').innerText = '❌ Login failed. Please try again.';

        document.getElementById('status').style.color = 'red';

    }

    if (urlParams.has('logout')) {

        document.getElementById('status').innerText = '✅ You have been logged out.';

        document.getElementById('status').style.color = 'green';

    }

1. **permitAll()** This tells Spring Security that all users (even unauthenticated ones) are allowed to access the login page and submit the login form. Without this, Spring might block access to your login form itself, creating a redirect loop.

## Custom JSON Based Login

We may not want to do form post at /login in that case we have to define our own Authentication mechanism. Which will replace the Basic login. We can do this using our own filter Which will creaate the Authentication Object and pas to Spring Security filter chain. This will read the JSON posted at /v1/api/login in the form {"username": "admin", "password": "qwerty"} create Authentication object and do validation.

The JsonUsernamePasswordAuthenticationFilter provides a custom authentication mechanism that handles JSON-based login requests instead of the traditional form-based login. Here's how it works:

1. **Filter Registration in the Security Chain**

The filter is added to Spring Security's filter chain using:

.addFilterAt(jsonAuthFilter, UsernamePasswordAuthenticationFilter.*class*)

This places it at the same position as the standard form-login filter in the chain, allowing it to intercept JSON login requests while the traditional form login still remains active for form submissions.

1. **Custom Endpoint Configuration** The filter is configured to process authentication at a custom endpoint:

*public* JsonUsernamePasswordAuthenticationFilter(AuthenticationManager authenticationManager) {  
 *super*.setAuthenticationManager(authenticationManager);  
 setFilterProcessesUrl("/v1/api/login"); *// your custom login endpoint*}

Unlike the default /login endpoint used by form login, this filter triggers only when POST requests are made to /v1/api/login.

1. **JSON Request Processing**

When a login request arrives:

* It first checks if the content type is application/json:
* Then it deserializes the request body into a LoginRequest object:
* This expects a JSON payload like: {"username": "admin", "password": "qwerty"}
* Creates an Authentication token with the credentials:
* Passes the token to the authentication manager for validation:

*@Override  
public* Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)  
 *throws* AuthenticationException {  
  
 *if* (!MediaType.APPLICATION\_JSON\_VALUE.equals(request.getContentType())) {  
 *throw new* AuthenticationServiceException("Unsupported Content Type: " + request.getContentType());  
 }  
  
 *try* {  
 LoginRequest loginRequest = objectMapper.readValue(request.getInputStream(), LoginRequest.*class*);  
  
 UsernamePasswordAuthenticationToken authRequest =  
 *new* UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword());  
  
 *return this*.getAuthenticationManager().authenticate(authRequest);  
  
 } *catch* (IOException e) {  
 *throw new* AuthenticationServiceException("Failed to parse login request", e);  
 }  
}

1. **Authentication Success Handling**

When authentication succeeds:

* A security context is created and populated with the authenticated user:
* The security context is stored in the HTTP session (just like with form login):
* Two cookies are added to the response:
  1. A secure, HttpOnly cookie (SECURE\_TOKEN) that's not accessible to JavaScript
  2. A plain cookie (PLAIN\_SESSION\_ID) that is JavaScript-accessible
* A success response is sent with a JSON message:

*@Override  
protected void* successfulAuthentication(HttpServletRequest request, HttpServletResponse response,  
 FilterChain chain, Authentication authResult) *throws* IOException, ServletException {  
 *// Set SecurityContext and persist in session* SecurityContext context = SecurityContextHolder.createEmptyContext();  
 context.setAuthentication(authResult);  
 SecurityContextHolder.setContext(context);  
  
 request.getSession(*true*).setAttribute(  
 HttpSessionSecurityContextRepository.SPRING\_SECURITY\_CONTEXT\_KEY,  
 context  
 );  
 *// --- Add cookies ---  
 // Secure, HttpOnly cookie (server-only)* Cookie secureCookie = *new* Cookie("SECURE\_TOKEN", "secure-value-12345");  
 secureCookie.setHttpOnly(*true*); *// JS can't access* secureCookie.setSecure(*true*); *// Only sent over HTTPS* secureCookie.setPath("/"); *// Available to entire app* secureCookie.setMaxAge(3600); *// 1 hour* response.addCookie(secureCookie);  
  
 *// Plain, JS-readable cookie* Cookie plainCookie = *new* Cookie("PLAIN\_SESSION\_ID", "visible-value-abcde");  
 plainCookie.setHttpOnly(*false*); *// JS can access* plainCookie.setSecure(*false*); *// OK for HTTP (dev env)* plainCookie.setPath("/");  
 plainCookie.setMaxAge(3600);  
 response.addCookie(plainCookie);  
  
 *// Let the success handler respond (configured in SecurityConfig)* getSuccessHandler().onAuthenticationSuccess(request, response, authResult);  
}

1. **Authentication Failure Handling** When authentication fails, an error response is sent:

## User Management

Authentication Flow Context (Quick Recap)

* UsernamePasswordAuthenticationFilter
* ↓
* AuthenticationManager
* ↓
* AuthenticationProvider(s)
* ↓
* UserDetailsService.loadUserByUsername(username)
* ↓
* UserDetails

InMemoryUserDetailsManager, an implementation of UserDetailsService, to store multiple user credentials in memory.

1. **UserDetailsService** An interface used to **load user data** from some source (memory, database, etc.).
2. **UserDetailsManager** An extension of UserDetailsService that adds **create**, **update**, **delete**, and **check** methods for users.
3. **InMemoryUserDetailsManager** An **implementation** of UserDetailsManager that stores users in **memory** using a HashMap. Used here.
4. **User** A builder class from Spring Security to easily create UserDetails objects.
5. **{noop}** A password encoder prefix that tells Spring **not to encode the password**. Useful for dev/demo, but **not safe for production**.

*@Bean  
//public UserDetailsManager userDetailsService() { This has additional API for create user, reset pwd etc.  
public* UserDetailsService userDetailsService() {  
 UserDetails admin = User.withUsername("admin").password("{noop}erQu773\_\_").roles("ADMIN").build();  
 *//https://bcrypt-generator.com/* UserDetails user = User.withUsername("user").password("{bcrypt}$2a$12$HknEwKGJto6O4zTn0pSA6.L9OX2wDEa3beQpN3W5XKrbNCipR0eTm")  
 .roles("USER").build();  
 *return new* InMemoryUserDetailsManager(admin, user);  
}

**UserDetailsService**

* **Purpose**: Loads user-specific data from a source (DB, memory, etc.
* Core Method: UserDetails loadUserByUsername(String username)
* Use Case:
  + Read-only scenarios (e.g., authentication only).
  + No need to allow registration, updates, or deletion.

**UserDetailsManager**

* **Extends**: UserDetailsService
* **Purpose**: Full user CRUD management.
* **Key Methods**:
  + void createUser(UserDetails user)
  + void updateUser(UserDetails user)
  + void deleteUser(String username)
  + void changePassword(String oldPassword, String newPassword)
  + boolean userExists(String username)
* Use Case:
  + Full lifecycle management (Register, Update, Delete, Change password).
* Common Implementations Provided by Spring Security

| **Class** | **Interface** | **Storage Type** | **Notes** |
| --- | --- | --- | --- |
| InMemoryUserDetailsManager | UserDetailsManager | In-memory | Ideal for testing/dev |
| JdbcUserDetailsManager | UserDetailsManager | JDBC (DB) | Works with relational DBs |
| LdapUserDetailsManager | UserDetailsManager | LDAP | Less common, used in legacy enterprise setups |

**UserDetails Interface**

* **Purpose**: Represents the authenticated user.
* Core Methods:
  + String getUsername()
  + String getPassword()
  + Collection<? extends GrantedAuthority> getAuthorities()
  + boolean isAccountNonExpired()
  + boolean isAccountNonLocked()
  + boolean isCredentialsNonExpired()
  + boolean isEnabled()

**User Implementation (User class)** The User class in Spring Security is a concrete implementation of the UserDetails interface. It's a helper class provided by Spring Security to represent the authenticated user's details like username, password, and authorities (roles/permissions).

It doesn't have setter methods, making it immutable and safe to use during authentication

When you implement a custom UserDetailsService, Spring expects a return of type UserDetails. You can either return a User object or your own class that implements UserDetails.

*@Override  
public* UserDetails loadUserByUsername(String username) *throws* UsernameNotFoundException {  
 *// usually fetched from DB  
 return new* User("john", "{noop}password", List.of(*new* SimpleGrantedAuthority("ROLE\_USER")));  
}

* **Implements**: UserDetails
* **Immutable**: No setters; use constructor or User.withUsername()...build().
* **Common Builder Method**:

User.withUsername("amit")

.password("encodedPassword")

.roles("USER")

.build();

**Internal Fields of the User class**

| **Field** | **Purpose** |
| --- | --- |
| username | Unique name to identify the user |
| password | Hashed (or raw) password |
| authorities | Roles or permissions granted |
| accountNonExpired | If false, account is expired |
| accountNonLocked | If false, account is locked |
| credentialsNonExpired | If false, credentials are expired |
| enabled | If false, account is disabled |

**Summary**

* The User class is a **ready-to-use** implementation of UserDetails.
* It’s **suitable for in-memory authentication** or quick testing.
* You can also use it in custom UserDetailsService implementations when fetching user data from a database.

**Authentication Interface**

* Represents the **authentication request and result**.
* Is the **input and output** of the AuthenticationManager.authenticate() method.
* Common implementation: UsernamePasswordAuthenticationToken

Key Methods:

| **Method** | **Purpose** |
| --- | --- |
| getPrincipal() | Usually returns the UserDetails |
| getCredentials() | Returns raw credentials (e.g., password) |
| getAuthorities() | Returns authorities after successful auth |
| isAuthenticated() | Boolean indicating success/failure |
| setAuthenticated(boolean) | Sets the status of auth |
| getDetails() | Extra details (e.g., IP, device info) |
| getName() | Returns the username (from Principal interface) |

* **UserDetails** is about **who the user is**.
* **Authentication** is about **whether the user is authenticated**.

## Custom JDBCUserDetailsManager

**The Problem Being Solved**

Spring Security, by default, loads user data from:

1. InMemoryUserDetailsManager – for in-memory users.
2. JdbcUserDetailsManager – for users stored in a predefined table structure in a database.

But in most **real-world applications**, you:

* Use **custom tables** with your own schema (like a Customer table).
* Need **full control** over how to load user data from the database.

To do that, you implement your **own class** that implements UserDetailsService.

This method is where you:

* Fetch the user from your database.
* Convert the user to a Spring Security UserDetails object.
* Throw an exception if the user isn’t found.

*@Component  
public class* EazyBankUserDetailsService *implements* UserDetailsService {  
  
 *private final* CustomerRepository customerRepository;  
  
 *@Autowired  
 public* EazyBankUserDetailsService(CustomerRepository customerRepository) {  
 *this*.customerRepository = customerRepository;  
 }  
  
 *@Override  
 public* UserDetails loadUserByUsername(String username) *throws* UsernameNotFoundException {  
 Customer customer = customerRepository.findByEmail(username).orElseThrow(() -> *new* UsernameNotFoundException("User details not found for the user: " + username));  
 List<GrantedAuthority> authorities = List.of(*new* SimpleGrantedAuthority(customer.getRole()));  
 *return new* User(customer.getEmail(), customer.getPwd(), authorities);  
 }  
}

## Passwords

**Problem 1: Hardcoding Passwords in Source Code**

* The password is written directly in your Java code or application.properties
* This is risky because:
  + If the code is committed to GitHub or shared, **anyone can see your password**.
  + Even for simple demo apps, this can lead to **unauthorized access** if someone uses the password to hit your secured APIs.

**Problem 2: Password Stored in Plain Text in Memory**

* Even after hardcoding, Spring stores that password **as-is** (plain text) in memory.
* If a hacker somehow gets a **heap dump** of your app (basically a memory snapshot), they can easily extract the password.

**Solution: Use a PasswordEncoder**

Spring Security provides the PasswordEncoder interface to handle passwords **securely** using hashing.

You can create a bean using:

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

* This gives you flexibility to use **multiple encoders**.
* By default, it uses **BCryptPasswordEncoder**, which is secure and recommended for production.
* The prefix {bcrypt} is used internally or explicitly like:

**Supported Encoders & Prefixes**

Spring supports many encoders via prefixes, for example:

| **Prefix** | **Encoder** | **Secure?** | **Use Case** |
| --- | --- | --- | --- |
| {noop} | NoOpPasswordEncoder | No | Plain text (NOT secure) |
| {bcrypt} | BCryptPasswordEncoder | Yes | Recommended for production |
| {scrypt} | SCryptPasswordEncoder | Yes | Secure alternative |
| {sha256} | StandardPasswordEncoder | No(deprecated) | Legacy support |

So your user credentials might look like:

*public* UserDetailsService userDetailsService() {  
 UserDetails admin = User.withUsername("admin").password("{noop}erQu773\_\_").roles("ADMIN").build();  
 *//https://bcrypt-generator.com/* UserDetails user = User.withUsername("user").password("{bcrypt}$2a$12$HknEwKGJto6O4zTn0pSA6.L9OX2wDEa3beQpN3W5XKrbNCipR0eTm")  
 .roles("USER").build();  
 *return new* InMemoryUserDetailsManager(admin, user);  
}

**Why Does Spring Security Use Bcrypt by Default?**

* You might not have **explicitly specified** to use bcrypt, yet it is still being used.
* That’s because Spring Security provides a **default implementation** using: PasswordEncoderFactories.createDelegatingPasswordEncoder()
* This method returns a DelegatingPasswordEncoder.
* It uses bcrypt as the default encoder, with a prefix like {bcrypt}.
* This prefix is stored along with the hashed password to help Spring know which algorithm was used.
* But using createDelegatingPasswordEncoder() is future-proof. If standards change or you need to support multiple formats (e.g., legacy {MD5}), Spring handles it automatically by checking the prefix.

**Why use PasswordEncoderFactories.createDelegatingPasswordEncoder()?**

Because:

* It supports **multiple encoders** via prefix.
* You can **migrate legacy** systems (e.g., {md4}, {noop}) while still supporting modern ones.
* Future Spring Security updates may change the recommended encoder; using the factory lets you adapt without rewriting code.

**How login works with PasswordEncoder**

* User enters plain password (e.g., 54321).
* Spring fetches the hashed password from memory or DB (e.g., a {bcrypt} hash).
* It uses the appropriate PasswordEncoder (based on the prefix) to compare:
  + matches(rawPassword, hashedPassword)
* If it matches, login is successful.

### Encoding

**Encoding** is:

* A process of converting data from one form to another.
* **Not** meant for security.
* **Reversible** — anyone can decode it, without needing a password or secret key.
* Example: If you **BASE64 encode** a password like EazyBytes@12345, you get something like: RWF6eUJ5dGVzQDEyMzQ1
* But this isn’t secure! Anyone who knows it's base64 can decode it back easily.

**Why Encoding is NOT good for password protection**

* **No secrets involved**: Anyone can reverse it with a simple tool or script.
* **Fully reversible**: The data (like passwords) can be brought back to original form.
* If a hacker gets access to your database, they can **easily decode** the passwords.

**When *is* encoding used?**

It’s commonly used when systems **can’t handle binary data**, like:

* Email systems (Outlook, Gmail): Attachments like images and videos are **base64 encoded** before sending.
* JSON APIs: Binary data (like PDFs) are base64 encoded to send as a string.
* So it’s mostly for **data transmission**, not data protection.

openssl base64 -in plain.txt -out encode.txt

openssl base64 -d -in encode.txt -out decode.txt

### Encryption

**Encryption: What It Is**

* **Encryption** is the process of transforming readable data (**plaintext**) into an unreadable format (**ciphertext**) to protect its confidentiality.
* It uses an **algorithm** and a **key** to do this.
* Decryption requires the **same key (symmetric)** or a **corresponding key (asymmetric)**.

**Two Types of Encryption**

**1. Symmetric Encryption**

* Uses the **same secret key** for both encryption and decryption.
* Example: AES (Advanced Encryption Standard).
* **Risk**: If someone gains access to the key, they can decrypt everything.
* **Use case**: Good for encrypting **data at rest** (e.g., files on disk, S3 buckets).

**2. Asymmetric Encryption**

* Uses a **pair of keys**: public key (for encryption) and private key (for decryption).
* Example: RSA, ECC.
* **Public key** can be shared with anyone; only the **private key** holder can decrypt.
* **Use case**: Ideal for **data in transit** (e.g., HTTPS, API calls).

Example openssl enc -aes-256-cbc -pass pass:12345 -pbkdf2 -in plain.txt -out encrypt.txt -base64

|  |  |
| --- | --- |
| openssl enc | Use OpenSSL's encryption tool |
| -aes-256-cbc | AES encryption algorithm with 256-bit key and CBC mode |
| -pass pass:12345 | Secret key (password) used for encryption is 12345 |
| -pbkdf2 | Strengthens the password-derived key using a modern key derivation function (makes it harder to brute-force) |
| -in plain.txt | Input file containing the original password |
| -out encrypt.txt | Output file to store the encrypted text |
| -base64 | Encode the encrypted binary data to base64 (makes it human-readable & safe to store/send) |

**Decryption Process:** openssl enc -aes-256-cbc -base64 -pass pass:12345 -d -pbkdf2 -in encrypt.txt -out decrypt.txt

| **Part** | **Meaning** |
| --- | --- |
| -d | This flag tells OpenSSL to **decrypt** |
| Rest of the flags | Must **match** the encryption process (same algorithm, password, etc.) |
| encrypt.txt | Input file containing encrypted data |
| decrypt.txt | Output file where decrypted data (original password) is written |

**Why Not Encryption for Passwords?**

The key reason : **Encryption can be reversed** if someone gets the key. Passwords should never be something that can be reversed.

**Problems with Using Encryption for Passwords:**

* You must **store the encryption key** securely.
* If the **key is compromised**, all passwords are compromised.
* Internal threats (even within the org) can be dangerous.
* Encryption ensures **confidentiality**, but not **integrity**.

So What Should Be Used for Passwords?

The correct approach is:

### Hashing

* Hashing is a **one-way function**: once hashed, it **cannot be reversed**. Its is a one-way process that turns your password into a fixed-length "digest" using a mathematical function (like SHA-256).
* Common hashing algorithms: **bcrypt**, **PBKDF2**, **Argon2**.
* Even if someone steals the hashed passwords, **they can't convert them back** to the original passwords easily (due to salting and computational cost).
* It ensures **integrity**, **non-reversibility**, and **better security**.

echo -n "password" | openssl dgst -sha256

**Hashing Has Drawbacks Too!**

1. Same Input = Same Hash (Deterministic Nature)
   1. If two users (U1 and U2) use the same password 12345, both will have the same hash (aef) in the database.
   2. That means if a hacker knows one password's hash, they can identify all users using the same password.
2. Hashing Is Too Fast
   1. Hash functions are designed to be very fast.
   2. This is great for performance, but bad for security — because it helps hackers guess millions of passwords very quickly.

**How Hackers Exploit These Weaknesses**

1. Brute Force Attack
   1. The hacker tries every possible common password and hashes them.
   2. Example: try 123, 1234, 12345, password, etc., one by one and hash them.
   3. If the hash matches a known value (e.g., aef), they’ve cracked the password.
2. Dictionary or Rainbow Table Attack
   1. Instead of calculating hashes every time, hackers pre-compute and store them.
   2. They create a giant dictionary
   3. During a breach, they just look up the hash in their prebuilt table — no computation needed.
3. **Rainbow Tables**
   1. These are **space-efficient, optimized** versions of dictionary tables.
   2. Designed to reduce storage needs and speed up the lookup process.

**Solution 1: Add a Salt to the Password**

**What is a Salt?** A **salt** is a **random string** added to the user's password **before hashing** it.

**Why is Salt Useful?** If two users choose the **same password** (e.g., "12345"), hashing alone would give the **same hash**. That’s bad because it makes it easier for hackers to recognize reused passwords.

By adding a **unique salt** per user:

* Each hash becomes **unique**, even if the passwords are the same.
* It defeats **rainbow table** and **dictionary attacks**.

**How Does It Work?**

**During Registration:**

* User enters password → e.g., "12345".
* Backend generates a **random salt** → e.g., "aex2fdac".
* Concatenate salt + password → "aex2fdac12345".
* Hash that combination → e.g., "af4c".
* Store both:
  + Salt: "aex2fdac"
  + Hash: "af4c"

**During Login:**

* User enters password "12345".
* System fetches stored salt "aex2fdac".
* Concatenates again → "aex2fdac12345".
* Hashes it → gets "af4c".
* Compares with stored hash. If it matches → login success.

**Why Salt Helps:**

Even if a hacker steals the hashed password:

* They **can’t use precomputed hash tables** (rainbow tables) because they **don’t know the salt in advance**.
* Each user's hash is unique, so **same password ≠ same hash**.

**Solution 2: Slow Down Hashing with Specialized Algorithms**

**Why?**

* Hash functions (like SHA-256) are **very fast**, which helps hackers.
* Hackers can try **millions of guesses per second** using brute force.

**Solution: Use "slow" password hashing algorithms**

These are designed to make brute force expensive in time and resources.

Examples:

* **BCrypt**
* **Scrypt**
* **Argon2**

These:

* Take **~1 second** per hash (instead of microseconds).
* Use **CPU and memory** heavily.
* Can be **configured** to be even slower.

**Real-World Impact:**

* A user logging in waits **1 second**—not a big deal.
* But a hacker trying **millions of guesses** needs **years of compute time and a huge budget**.

**Real Implementations (a.k.a. the ones that aren’t rotting in the deprecated bin):**

1. **BCryptPasswordEncoder (Popular Kid)**
   1. Hashes password using BCrypt algorithm.
   2. Built-in salting.
   3. Very secure. Very recommended.
   4. Slows down hashing enough to make brute-force attempts painful.
2. **Argon2PasswordEncoder (New Kid on the Block)**
   1. Uses Argon2, winner of the Password Hashing Competition (yes, that’s a real thing).
   2. More tunable and future-ready than BCrypt.
   3. Slower = better defense.
   4. Requires native libs sometimes, so can be annoying to set up.
3. **Pbkdf2PasswordEncoder (Solid, Respectable)**
   1. Uses PBKDF2 with configurable:
      1. salt length (default: 16 bytes)
      2. iteration count (default: 310,000 = slow)
      3. secret (for extra cryptographic spice)
   2. Generates salt, hashes the password using it, and verifies by re-generating the same salted hash and comparing it.
   3. Good for environments where you control all the knobs.
   4. Not deprecated, but easily misconfigured. A bit of a DIY experience.
4. **SCryptPasswordEncoder (Cryptographically Buff)**
   1. Strong key derivation function.
   2. Even more secure and memory intensive.
   3. Not as commonly used as BCrypt but very solid.
5. **DelegatingPasswordEncoder (The Switchboard Operator)**
   1. Lets you support *multiple encoders* based on prefix.
   2. Like: {bcrypt}hashedPasswordHere or {noop}plaintextPassword.
   3. Essential for migrating from old schemes to new ones.

**Deprecated or Joke-Level Encoders:**

1. **NoOpPasswordEncoder (a.k.a. "Please Rob Me")**
   1. Literally just returns the plain text. Still implements PasswordEncoder so you can plug it into Spring Security… but it **doesn’t encode** anything. Only use in demos , tests etc.
2. **StandardPasswordEncoder**
   1. Used SHA-256 with 1024 iterations and 8-byte salt.
   2. Sounds decent until you realize CPUs laugh at it now.
   3. Deprecated for good reason: *too easy to brute-force* in modern times.
3. **MessageDigestPasswordEncoder, Md4PasswordEncoder, LdapShaPasswordEncoder, etc.**
   1. Cryptographically embarrassing.
   2. Only exist for *backward compatibility*.
   3. Using these in production is basically consenting to being hacked.

## Authentication Provider

**What is AuthenticationProvider?**

It’s an **interface** in Spring Security that is responsible for performing authentication. This interface has **only two methods**:

**1. authenticate(Authentication authentication)**

* This is where **actual authentication logic** goes.
  + It receives an authentication object, typically with: **Username, Password,** authenticated flag set to **false**
* Your job is to:
  + Load user details (like from a DB)
  + Match credentials (e.g., passwords)
  + Do additional checks (if needed)
  + Finally return a new Authentication object with authenticated = true
* Spring’s default AuthenticationProvider is DaoAuthenticationProvider, which:
* Extracts username
* Loads user from database
* Uses PasswordEncoder to check passwords
* Returns a successful authentication if all is well

**2. supports(Class<?> authentication)**

* This method tells **what kind of Authentication tokens this provider can handle.**
* For example:
  + DaoAuthenticationProvider supports UsernamePasswordAuthenticationToken
* If the supports() method returns true for a token, Spring will **use this provider to call authenticate()**

By default, Spring Security uses:

* **DaoAuthenticationProvider**, which works with:
  + **UserDetailsService** (to load user info from DB)
  + **PasswordEncoder** (to match passwords)

This setup works well for common use-cases where users log in with **username + password**, and credentials are stored in a database. But what if your app has complex or unusual authentication requirements?

Some examples:

1. Username/password for one app
2. OAuth2 for another (e.g., login via Google/Facebook)
3. JAAS for legacy systems
4. Age-based restrictions (only allow users above 18)
5. Country-based login checks (only allow users from specific regions)

These scenarios go **beyond the default logic** of just checking username and password. So, you’ll need to write **your own AuthenticationProvider** to handle them.

**How does Spring Security decide which AuthenticationProvider to use?**

When a request comes in:

1. The **filter chain** intercepts it.
2. The request is converted into an Authentication object (like UsernamePasswordAuthenticationToken).
3. The AuthenticationManager (usually a ProviderManager) takes over.
4. Based on the **type of Authentication object**, it decides which AuthenticationProvider to call.

For example:

* If it’s UsernamePasswordAuthenticationToken → use DaoAuthenticationProvider
* If it’s a custom OAuth2Token → use your custom provider
* If it’s JaasAuthenticationToken → use JAAS provider

**How does Spring use these?**

* Spring Security has a class called **ProviderManager**, which manages multiple AuthenticationProviders.
* When an authentication request comes in:
  + It loops through all registered AuthenticationProviders.
  + For each provider, it checks: supports(authentication.getClass())
  + If true, it calls the provider’s authenticate() method.
  + It stops on the first successful result (or continues if not successful).

**TestingAuthenticationToken?**

* This is a **special token for unit testing**.
* There's a corresponding TestingAuthenticationProvider.
* It doesn’t actually check anything — just returns the input Authentication object.
* Very useful to **mock authentication** in tests without doing real logins.

| **Concept** | **Meaning** |
| --- | --- |
| **DaoAuthenticationProvider** | Default provider that checks username/password using UserDetailsService |
| **AuthenticationProvider** | Interface you implement when you want **custom login logic** |
| **ProviderManager** | Delegates to one of the available providers, based on the Authentication object type |
| **When to write custom provider?** | When you need logic like age checks, external systems, multiple login types (OAuth2, JAAS, etc.) |
| **Where to put extra logic?** | Inside your custom AuthenticationProvider, **not** in UserDetailsService |

*@Component  
public class* EazyBankUsernamePwdAuthenticationProvider *implements* AuthenticationProvider {  
  
 *@Autowired  
 private* CustomerRepository customerRepository;  
  
 *@Autowired  
 private* PasswordEncoder passwordEncoder;  
  
 *@Override  
 public* Authentication authenticate(Authentication authentication) *throws* AuthenticationException {  
 String username = authentication.getName();  
 String pwd = authentication.getCredentials().toString();  
 Optional<Customer> customer = customerRepository.findByEmail(username);  
 *if* (customer.isPresent()) {  
 *if* (passwordEncoder.matches(pwd, customer.get().getPwd())) {  
 List<GrantedAuthority> authorities = *new* ArrayList<>();  
 authorities.add(*new* SimpleGrantedAuthority(customer.get().getRole()));  
 *return new* UsernamePasswordAuthenticationToken(username, pwd, authorities);  
 } *else* {  
 *throw new* BadCredentialsException("Invalid password!");  
 }  
 }*else* {  
 *throw new* BadCredentialsException("No user registered with this details!");  
 }  
 }  
  
 *@Override  
 public boolean* supports(Class<?> authentication) {  
 *return* (UsernamePasswordAuthenticationToken.*class*.isAssignableFrom(authentication));  
 }  
  
}

# Spring Security Features

## HTTP/HTTPS

Spring Boot apps by default **accept both HTTP and HTTPS**. We can make it accept https only using below

.requiresChannel(rcc -> rcc.anyRequest().requiresSecure()) *// Only HTTPS*

## Error Handling

**Secured API Access and Basic Auth**

* When you try to access a secured API (a protected URL or endpoint), Spring Security checks whether the request is **authenticated** and **authorized**.
* If you're using **Postman**, you need to provide the **username and password** using the **Basic Auth** type to authenticate.
* If the credentials are correct → you get a **successful response**.
* If you're accessing it via a browser without credentials → you'll be redirected to a **login page**.

**Testing Error Scenarios in Postman**

* If you give an **invalid username** (e.g., admin1 instead of admin) and try again:
  + You’ll get a **401 Unauthorized** response with a JSON payload like:

{

"timestamp": "...",

"status": 401,

"error": "Unauthorized",

"message": "...",

"path": "/api/..."

}

Spring Security **only handles security-related exceptions**, not all runtime or business logic exceptions.

Two Main Exception Types:

1. AuthenticationException → Results in 401 Unauthorized
   * E.g., BadCredentialsException, UsernameNotFoundException
   * Means the user is not authenticated.
2. AccessDeniedException → Results in 403 Forbidden
   * Means the user is authenticated but does **not have the required permissions**.

The Core Component: ExceptionTranslationFilter

This filter sits in the Spring Security filter chain and is responsible for catching and handling these two types of exceptions.

Flow:

* **If no error**: it simply passes control to the next filter.
* If an error occurs:
  + It checks the type of exception.
  + If it's AuthenticationException, it calls an implementation of AuthenticationEntryPoint.
  + If it's AccessDeniedException, it calls an implementation of AccessDeniedHandler.

Interfaces to Customize Responses

1. AuthenticationEntryPoint (for 401)

* Method to implement: commence()
* Responsible for handling unauthenticated access attempts.
* Default: BasicAuthenticationEntryPoint (adds WWW-Authenticate header with realm info)

Customize:

Create a new class that implements AuthenticationEntryPoint, override commence(), and define your **own response body and headers**.

*public class* CustomBasicAuthenticationEntryPoint *implements* AuthenticationEntryPoint {  
 *@Override  
 public void* commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)  
 *throws* IOException, ServletException {  
 *// Populate dynamic values* LocalDateTime currentTimeStamp = LocalDateTime.now();  
 String message = (authException != *null* && authException.getMessage() != *null*) ? authException.getMessage()  
 : "Unauthorized";  
 String path = request.getRequestURI();  
 response.setHeader("eazybank-error-reason", "Authentication failed");  
 response.setStatus(HttpStatus.UNAUTHORIZED.value());  
 response.setContentType("application/json;charset=UTF-8");  
 *// Construct the JSON response* String jsonResponse =  
 String.format("{\"timestamp\": \"%s\", \"status\": %d, \"error\": \"%s\", \"message\": \"%s\", \"path\": \"%s\"}",  
 currentTimeStamp, HttpStatus.UNAUTHORIZED.value(), HttpStatus.UNAUTHORIZED.getReasonPhrase(),  
 message, path);  
 response.getWriter().write(jsonResponse);  
 }  
}

2. AccessDeniedHandler (for 403)

* Method to implement: handle()
* Handles cases where the user is authenticated but forbidden.
* Default: AccessDeniedHandlerImpl

Customize:

Create a class that implements AccessDeniedHandler, override handle(), and return your custom JSON + headers.

*public class* CustomAccessDeniedHandler *implements* AccessDeniedHandler {  
 *@Override  
 public void* handle(HttpServletRequest request, HttpServletResponse response,  
 AccessDeniedException accessDeniedException) *throws* IOException, ServletException {  
 *// Populate dynamic values* LocalDateTime currentTimeStamp = LocalDateTime.now();  
 String message = (accessDeniedException != *null* && accessDeniedException.getMessage() != *null*) ?  
 accessDeniedException.getMessage() : "Authorization failed";  
 String path = request.getRequestURI();  
 response.setHeader("eazybank-denied-reason", "Authorization failed");  
 response.setStatus(HttpStatus.FORBIDDEN.value());  
 response.setContentType("application/json;charset=UTF-8");  
 *// Construct the JSON response* String jsonResponse =  
 String.format("{\"timestamp\": \"%s\", \"status\": %d, \"error\": \"%s\", \"message\": \"%s\", \"path\": \"%s\"}",  
 currentTimeStamp, HttpStatus.FORBIDDEN.value(), HttpStatus.FORBIDDEN.getReasonPhrase(),  
 message, path);  
 response.getWriter().write(jsonResponse);  
 }  
}

Hooked it into Spring Security config

Two Ways to Configure

* + - 1. Local Configuration with httpBasic()
      2. 2. Global Configuration with exceptionHandling()

.httpBasic(hbc -> hbc.authenticationEntryPoint(*new* CustomBasicAuthenticationEntryPoint()))  
.exceptionHandling(ehc -> ehc.accessDeniedHandler(*new* CustomAccessDeniedHandler())); *// Global exceptional handling*

What About UI Login Flows?

* For UI login (e.g., form-based login), Spring uses a different entry point: LoginUrlAuthenticationEntryPoint.
* You don’t usually override that.
* If you want to redirect users to a **custom login page**, Spring provides other ways to do that.

So, AuthenticationEntryPoint customization is mostly relevant for **API-based authentication (like Basic Auth)**.

Debugging Tip

You can put breakpoints in:

* BasicAuthenticationEntryPoint#commence() → for 401 errors
* AccessDeniedHandlerImpl#handle() → for 403 errors

This helps you confirm how Spring is invoking the default logic.

Why you can’t use authenticationEntryPoint() with form login

* The instructor tries to show that **formLogin** doesn’t allow you to set authenticationEntryPoint() directly.
* Why? Because form login is meant for **UI-based flows**, not API-based ones.
* With UI, users expect to be **redirected to a login page**, not to receive a JSON error or a raw HTTP 401.
* So instead, formLogin provides other methods like:
  + loginPage()
  + defaultSuccessUrl()
  + failureHandler()
  + loginProcessingUrl()
  + failureUrl()

So why bother with global exceptionHandling()?

* Good question, and the instructor answers it well.
* If you only configure authenticationEntryPoint() for httpBasic(), it will only be triggered **during login flow**.
* But **what if other parts of your app** — maybe REST endpoints or internal Spring mechanisms — also throw AuthenticationException?
* Those won’t be handled unless you also configure exceptionHandling() globally.

| Error Code | Meaning | When it Happens |
| --- | --- | --- |
| 401 | Unauthorized | The user is **not authenticated** (not logged in) |
| 403 | Forbidden | The user is **authenticated but not authorized** to access a resource |

## Session management

**What is the default session timeout?**

By **default**, the session timeout is **30 minutes**. So:

* If the user is inactive for 30 minutes (doesn’t send any request),
* The session becomes **invalid**,
* And if the user tries to interact with the app afterward, they are redirected to the **login page**.

**How to customize the session timeout**

You can set your own timeout in the application.properties file: server.servlet.session.timeout=20mz

server.servlet.session.timeout=20m

* 20m = 20 minutes (you can also use s for seconds, h for hours)
* The **minimum allowed** value is **2 minutes** (Spring Boot enforces this).

**Better user experience with** invalidSessionUrl()

Instead of redirecting to the login page, we can redirect users to a **custom URL** (e.g., /invalid-session) that explains why they are being redirected:

http.sessionManagement(smc ->

smc.invalidSessionUrl("/invalid-session")

);

You also need to **permit** access to this path:

.requestMatchers("/invalid-session").permitAll()

**Other scenarios that trigger invalid sessions**

1. **Timeout (idle)** → session becomes invalid.
2. **Tampered JSESSIONID** (e.g., user manually modifies the cookie) → also treated as invalid session.

In **both cases**, Spring Security will redirect the user to the **same URL** (/invalid-session) instead of the login page, which makes your app more **user-friendly and informative**.

**Summary**

* Sessions in Spring Boot expire after 30 mins by default.
* You can customize this using server.servlet.session.timeout.
* Minimum allowed timeout is 2 minutes.
* Use invalidSessionUrl() to handle expired/tampered sessions gracefully.
* Always permitAll() access to that URL.
* Build a user-friendly page for /invalid-session.

By **default**, Spring Security **does not limit** the number of sessions a user can have.  
So the same user (smith@example.com) could:

* Login in **multiple browsers**
* Use **Postman**, **browser tabs**, or different **devices**
* Have **unlimited active sessions** concurrently

For security or business reasons (e.g., online banking, exams, subscriptions), you might want to:

* Allow **only 1 session per user**
* Or **2, 3, or 5 sessions max**, depending on the use case

Configure sessionManagement() in Security Config

http

.sessionManagement(session -> session

.invalidSessionUrl("/session-expired")

.maximumSessions(1) // Only 1 session allowed per user

);

Prevent Login if Maximum Session Reached

If you don’t want the old session to be invalidated, but instead want to **block** new logins:

.maximumSessions(1)

.maxSessionsPreventsLogin(true)

You can configure Spring Security like:

http

.csrf().disable()

.authorizeHttpRequests(auth -> auth

.requestMatchers("/login", "/session-expired").permitAll()

.anyRequest().authenticated()

)

.sessionManagement(session -> session

.invalidSessionUrl("/session-expired")

.maximumSessions(1)

.maxSessionsPreventsLogin(true)

.expiredUrl("/session-expired")

)

.formLogin()

.httpBasic()

.exceptionHandling();

return http.build();

## Session Hijacking Session Fixation

What is Session Hijacking?

Session Hijacking is a security attack where an attacker **steals a valid session ID** and uses it to impersonate a user.

How it happens:

* The session ID can be found in:
  + **URLs** (e.g., example.com/page?sessionId=xyz)
  + **Cookies** stored in the browser.
* A hacker can steal this session ID by:
  + Looking at **browser history** (e.g., public computers).
  + **Intercepting traffic** if the app doesn’t use HTTPS.

Prevention Techniques:

1. Use HTTPS → Encrypts data in transit.
2. Short session timeouts → e.g., 10-15 minutes.
3. Public computer mode → Websites may avoid storing session cookies.
4. Avoid putting session IDs in URLs.

Spring Security **doesn't directly prevent** Session Hijacking. It's mostly **up to the developers and infrastructure** to apply safe practices.

What is Session Fixation?

Definition:

Session Fixation is a type of attack where the attacker fixes (sets) a known session ID for a user before the user logs in, and later uses it to hijack their session.

Example scenario:

1. Eva (hacker) logs into fashionmart.com → gets session ID abc123.
2. Eva sends Emily (victim) a fake promo email with a link containing session ID abc123.
3. Emily clicks and logs in → Now her account is associated with the same session ID abc123.
4. Eva reuses abc123 → Now she sees Emily’s data and payment info.

Prevention:

The solution lies in ensuring that once the user logs in, the application changes the session ID, making the one the hacker fixed invalid.

How Spring Security Prevents Session Fixation

By default, Spring Security protects against Session Fixation using one of these strategies:

A. changeSessionId() (Default)

* Keeps session data, but generates a new session ID after login.
* Hacker doesn’t know this new ID → attack fails.

B. newSession()

* Creates a brand-new session and does NOT copy any old data except for Spring Security data (like auth info).

C. migrateSession()

* Creates a new session, but copies all data (like user attributes) from the old session.

D. none()

* Disables protection. Not recommended unless you implement custom protection.

http

.sessionManagement(session -> session

.sessionFixation(sfc -> sfc

.migrateSession() // or .newSession(), .none()

)

);

## Handling authentication events

Spring Security automatically **publishes events** when authentication happens:

* **AuthenticationSuccessEvent** — when login is successful.
* **AuthenticationFailureEvent** — when login fails (wrong password, etc).

These are published by a class called:

* DefaultAuthenticationEventPublisher

**What should you do as a developer?**

You can **listen** for these events using standard Spring mechanisms:

1. Create a Spring **bean** (a normal @Component class).
2. Inside it, create **methods** like onSuccess() or onFailure() to handle events.
3. Annotate those methods with **@EventListener**.

*@Component  
@Slf4j  
public class* AuthorizationEvents {  
  
 *@EventListener  
 public void* onFailure(AuthorizationDeniedEvent deniedEvent) {  
 log.error("Authorization failed for the user : {} due to : {}", deniedEvent.getAuthentication().get().getName(),  
 deniedEvent.getAuthorizationDecision().toString());  
 }  
  
}

*@Component  
@Slf4j  
public class* AuthenticationEvents {  
  
 *@EventListener  
 public void* onSuccess(AuthenticationSuccessEvent successEvent) {  
 log.info("Login successful for the user : {}", successEvent.getAuthentication().getName());  
 }  
  
 *@EventListener  
 public void* onFailure(AbstractAuthenticationFailureEvent failureEvent) {  
 log.error("Login failed for the user : {} due to : {}", failureEvent.getAuthentication().getName(),  
 failureEvent.getException().getMessage());  
 }  
  
}

## UI

**Login Flow**

Here's how the login works:

1. **User fills login form** (LoginComponent.html)
2. **Form submits → validateUser()** in LoginComponent.ts
3. Calls LoginService.validateLoginDetails(user)
4. Stores user in session, sends GET to backend
5. Backend authenticates and responds with JSESSIONID
6. For future requests, Angular sends this cookie (if withCredentials: true)

* **Login API** is the only call made directly from the login.service.ts file.
* It's invoked from login.component.ts → validateUser() method.
* Inside this method:
  + A call is made to the backend for login.
  + The response is handled using .subscribe(), which waits for the response asynchronously.
  + If successful, the response (user info) is saved into a model of type User.
  + It also sets a special variable (e.g. authStatus = "AUTH") to indicate the user is authenticated.
  + User details are stored in sessionStorage under the key "userdetails".
  + Then, the app navigates to the /dashboard.

**DashboardService**

This service fetches different types of data (accounts, cards, loans, etc.):

* Uses functions like getAccountDetails(), getCardDetails()
* Calls backend endpoints with user ID as query param
* Secure APIs use withCredentials: true
* Public APIs (e.g. notices, contact form) **don’t** send cookies
* For protected routes like /dashboard, a **guard class** like AuthActivateRouteGuard is used.
* This class implements canActivate(), which checks:
  + Is userdetails present in sessionStorage?
  + Does it have a valid email?
* If **true**, allow navigation.
* If **false**, redirect to login.

**Intercepting HTTP Requests (HttpInterceptor)**

* Angular uses an **interceptor** class to inspect and modify all outgoing HTTP requests.
* It checks if the user is logged in (email & password available).
* If yes (usually only during login):
  + Adds an Authorization header with a base64 encoded value: Basic base64(email:password).
  + This lets Spring Security authenticate via **Basic Auth**.
* If no password (i.e., post-login requests), no credentials are sent.
  + But cookies like JSESSIONID are sent automatically **because** of withCredentials: true.

**Session Continuity After Login**

* Even though only the login API sends credentials,
  + Subsequent secure calls work because of the JSESSIONID cookie (managed by the browser).
  + Angular automatically sends this with requests if withCredentials: true is set.

## CORS

What is CORS?

CORS stands for Cross-Origin Resource Sharing. It’s a security feature implemented by browsers to control how resources are shared across different origins.

**What is an "Origin"?**

An **origin** is defined by three components:

1. **Protocol** (HTTP or HTTPS)
2. **Domain/Hostname** (like google.com)
3. **Port** (like :8080)

Two URLs are of different origin if **any of these differ**.

Examples:

* http://example.com:8080 vs http://example.com:8081 → **different origin**
* https://google.com vs https://amazon.com → **different origin**

Why CORS Exists

Browsers block requests from different origins by default to prevent malicious sites from stealing data from your app/server (a form of attack called Cross-Site Request Forgery or data theft via unauthorized scripts).

So CORS is not a bug or a security threat, but rather a security measure implemented by browsers.

In Spring Boot, this is typically done by:

* Adding a **CORS configuration** using @CrossOrigin on controllers or globally via a WebMvcConfigurer bean.

@CrossOrigin(origins = "http://localhost:4200")

@RestController

public class NoticeController {

// your endpoints

}

* Or configure CORS globally like:

http

.sessionManagement() // already existing config

.and()

.cors(corsConfig -> corsConfig

.configurationSource(new CorsConfigurationSource() {

@Override

public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {

CorsConfiguration config = new CorsConfiguration();

// Allow only this origin (Angular dev server)

config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));

// Allow all HTTP methods like GET, POST, PUT, DELETE

config.setAllowedMethods(Collections.singletonList("\*"));

// Allow credentials like cookies or HTTP basic auth

config.setAllowCredentials(true);

// Allow all headers from frontend

config.setAllowedHeaders(Collections.singletonList("\*"));

// Cache preflight response for 1 hour

config.setMaxAge(3600L);

return config;

}

})

);

## CSRF

**What is CSRF?**

CSRF (Cross-Site Request Forgery) is a **legitimate security attack**—unlike CORS, which is a browser-enforced policy. In a CSRF attack, a malicious site tricks a logged-in user’s browser into sending **unauthorized requests** to a vulnerable web application on behalf of the user. If the app doesn't verify the legitimacy of these requests, sensitive operations (like money transfers, account changes, etc.) may be performed **without the user's consent**.

**Spring Security & CSRF (Default Behavior)**

By **default**, Spring Security **enables CSRF protection** for any HTTP methods that can modify data — like POST, PUT, DELETE, and PATCH.

However, **CSRF protection is not applied** to **safe methods** like GET, HEAD, OPTIONS, and TRACE—because they are not intended to change data and are generally considered safe from CSRF.

**Technical Flow of a CSRF Attack**

Let’s use the Netflix + evil.com analogy again, with a clearer structure:

**Step 1: User logs in to Netflix**

* The user logs in to netflix.com using valid credentials.
* The server authenticates and sets a **cookie** (JSESSIONID=abc123) for the domain netflix.com.
* The browser stores this cookie and will send it **automatically** with any future requests to netflix.com.

**Step 2: User visits a malicious site (evil.com)**

* In the **same browser session**, the user opens evil.com.
* evil.com is controlled by a hacker and renders a page with **hidden malicious form** or **JavaScript**.
* For example, a fake "90% off iPhone" ad tempts the user to click.

**Step 3: Hidden form submits a request to Netflix**

<form action="https://netflix.com/change-email" method="POST">

<input type="hidden" name="email" value="hacker@evil.com" />

</form>

<script>

document.forms[0].submit();

</script>

* When the form is submitted:
  + The request goes to netflix.com
  + The browser automatically attaches the Netflix cookie (abc123)
* Result: The Netflix server thinks this is a legitimate request from the user, because the cookie is valid.
* The user's email gets changed without their knowledge.

**Disabling CSRF (Why it works in your local setup)**

In the demo setup, **CSRF protection is disabled**, which is why POST requests like registration or contact form submissions work **without a CSRF token**. Disabling CSRF is fine in:

* Stateless APIs (e.g., using tokens or JWT)
* Non-browser clients (e.g., Postman)
* Internal microservices talking over trusted networks

But it is **not recommended in production** for web applications that use **cookie-based authentication**, because that’s exactly where CSRF thrives.

**What happens when CSRF protection is enabled?**

If you **remove** .csrf().disable() or explicitly **enable CSRF**, Spring Security expects every modifying HTTP request (POST/PUT/DELETE) to include a **valid CSRF token**. If the token is **missing**, **invalid**, or **null**, Spring will respond with:

403 Forbidden – Invalid CSRF Token

**How Does Spring Security Help?**

By default, **Spring Security enables CSRF protection** for state-changing HTTP methods (POST, PUT, DELETE, PATCH).

When enabled:

1. Spring expects a **CSRF token** to be present on every modifying request.
2. This token is:
   * Sent by the server to the frontend (via form or meta tag)
   * Then sent back with the request (via header or hidden field)
3. If the token is missing or invalid → Spring throws **403 Forbidden**.

**CSRF Protection in Action (Spring Security Example)**

1. If CSRF protection is **disabled**, requests like POST/DELETE succeed without tokens.
2. If **enabled**:
   * GET requests work (because they are "safe").
   * POST/PUT/DELETE requests **fail with 403** unless a valid token is included.

**How to Prevent CSRF (Solutions)**

1. **Enable CSRF protection** in Spring (default behavior).
2. **Use SameSite cookies**: Modern browsers support SameSite=Lax or Strict, reducing CSRF risks.
3. **Require CSRF tokens**:
   * Add CSRF token as a hidden field in HTML forms
   * For APIs, send it via HTTP header (X-CSRF-TOKEN)
   * Expected to be sent **back in every state-changing request** (POST, PUT, DELETE) in the **header or body**

**How it Works Step-by-Step**

1. Step 1: User Logs into netflix.com
2. Server authenticates the user.
3. Sends:
   * JSESSIONID cookie (for session)
   * XSRF-TOKEN cookie (for CSRF protection)

Now the browser has both cookies stored.

**Step 2: A Malicious Site (evil.com) Tries an Attack**

1. The user visits evil.com (perhaps tricked by a fake offer).
2. evil.com tries to send a POST request to netflix.com to change the user’s email or password.
3. The browser automatically includes:
   * JSESSIONID cookie
   * XSRF-TOKEN cookie

So how does the server detect it's a forgery?

**Step 3: Backend Double-Checks the CSRF Token**

The server expects the CSRF token to appear in two places:

1. In the cookie (automatically sent by browser)
2. In the request header or body, manually added by JavaScript code

Now here’s the key:

1. evil.com can’t read the CSRF token from the cookie.
2. Why? Because JavaScript on evil.com can’t access cookies from netflix.com (same-origin policy).
3. So evil.com can’t include the CSRF token in the header or body.

So the server sees:

1. Cookie -Present
2. Header/body -Missing

And responds with 403 Forbidden.

**Why Manual Token Loading Is Needed?**

1. **Spring generates the CSRF token lazily**, i.e., **only when someone accesses it**.
2. For GET requests, no CSRF is needed → token not generated.
3. But frontend apps often expect to **receive CSRF token on first load** (even on GET).
4. Solution: **read the token manually** so Spring generates it.

**Why This Matters for Frontends Like Angular**

1. Angular expects a CSRF token in a cookie (XSRF-TOKEN) and sends it in a header (X-XSRF-TOKEN).
2. Spring matches this convention with CookieCsrfTokenRepository.
3. If you don't manually load the token (using a filter), Angular won’t get it → CSRF validation will fail on POST/PUT/DELETE.

Spring support for CSRF

**CsrfToken Interface (in package org.springframework.security.web.csrf)**

* Represents the **CSRF token object**.
* Has methods like:
  + getToken() – gets the actual token value.
  + getHeaderName() – name of the header used for the token (like X-XSRF-TOKEN).
  + getParameterName() – used for form submissions.

Usually, Spring uses the implementation class: **DefaultCsrfToken**.

**CsrfTokenRepository Interface**

* Manages the **creation, storage, and retrieval** of CSRF tokens.
* Two main implementations:
  + **HttpSessionCsrfTokenRepository**
    - Stores CSRF token in the **session**.
    - **Not ideal** for REST APIs (stateless).
  + **CookieCsrfTokenRepository**
    - Stores token in a **cookie**.
    - Recommended for SPAs (Angular, React, etc).

**CSRF Token Generation After Login**

* **What happens initially?**
  + After a successful login using HTTP Basic Auth, **Spring Security generates a CSRF token** and **sends it to the client as a cookie** (e.g., XSRF-TOKEN).
  + The session (JSESSIONID) is also created **only if** we explicitly tell Spring to do so (more on that later).
* This first step works if you’ve configured:
  + A CSRF token repository (like CookieCsrfTokenRepository)
  + Basic CSRF settings in the SecurityFilterChain

**2. CSRF Token Validation on Subsequent Requests**

* After login, for any **POST/PUT/DELETE** requests, Spring’s **CSRF filter** kicks in.
* This filter must know **where to look for the CSRF token**:
  + It expects the token value in the **request header** (e.g., X-XSRF-TOKEN).
  + Spring will also check the **cookie** for comparison.
* Spring uses a class called CsrfTokenRequestAttributeHandler:
  + This handler reads the token from the header or parameter.
  + It sets the token as a **request attribute** under the key "\_csrf".

**3. What You Must Configure in Spring Security**

Inside your SecurityFilterChain method:

**a. Set up CsrfTokenRequestAttributeHandler**

CsrfTokenRequestAttributeHandler handler = new CsrfTokenRequestAttributeHandler();

http.csrf(csrf -> csrf

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())

.csrfTokenRequestHandler(handler)

);

**b. Configure session creation**

Spring doesn't automatically create a session for HTTP Basic login with a separate frontend.

To ensure a JSESSIONID is created:

http.sessionManagement(session ->

session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS)

);

**c. Configure security context storage**

Tell Spring to handle saving the authentication to SecurityContextHolder:

http.securityContext(context ->

context.requireExplicitSave(false)

);

**What Happens on Login (UI Side)?**

1. When the user logs in, the **Spring Boot backend**:
   1. Generates a **CSRF token**.
   2. Sends it as a cookie (XSRF-TOKEN).
2. In the **Angular frontend**:
   1. After login succeeds (validateLoginDetails().subscribe(...)), the code:
      1. **Reads the CSRF token from the cookie** using a helper function like getCookie("XSRF-TOKEN").
      2. **Stores it in sessionStorage** using:

window.sessionStorage.setItem('XSRF-TOKEN', xsrf!);

The ! tells TypeScript you are sure xsrf is not null or undefined.

**How Does the CSRF Token Get Sent with Future Requests?**

* Angular has an **HTTP interceptor**, which intercepts every outgoing request.
* In that interceptor:
  + It reads the CSRF token from sessionStorage.
  + If the token exists, it sets a header X-XSRF-TOKEN on the request:

headers = headers.set('X-XSRF-TOKEN', xsrf);

This is crucial: the CSRF token must be **in both** the cookie and the header for Spring Security to accept the request.

**What If User Directly Visits a Public Page (e.g., Contact Us)?**

* The CONTACT US page is a **public API**.
* If a POST request is sent to this page **without first logging in**, the CSRF token will not be available.
* Spring will respond with a **403 Forbidden** error because the required CSRF token is missing.

**How to Disable CSRF for Specific APIs:**

1. Use the ignoringRequestMatchers() method:

http.csrf(csrf -> csrf

.ignoringRequestMatchers("/contact", "/register")

);

1. This tells Spring Security: “Don’t apply CSRF protection to these endpoints.”
2. After setting this up, you can test it by sending POST requests to these endpoints without logging in and without a CSRF token, and they’ll still work (return 200 OK, not 403 Forbidden).

# JWT

In general, there are two types of tokens

1. **Opaque Tokens** are typically random strings with no inherent meaning. Used to reference authentication information stored on the server-side.
2. **JSON Web Tokens (JWT)** are self-contained tokens that consist of three parts: a header, a payload, and a signature, encoded in Base64 URL. Requires a call to the authorization server or a dedicated introspection endpoint to validate and obtain user information. Encodes user information and claim directly within the token. Can be validated locally by verifying the token's signature using a public key, without needing a server call.

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

**ROLE OF TOKENS** IN AUTHN & AUTHZ

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

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

|  |  |  |
| --- | --- | --- |
| Client will receive the token after successful login in a header/query string etc.  Client system must make sure to send the same token value on all the further request to the backend |  | Server will send token and send to client and stores client details and token in memory  Sever will validate the token and return result |

**Key Advantages of Token-Based Authentication**

**Security**

* Limited exposure of user credentials within the network
* Tokens can be revoked during suspicious activities without invalidating the user credentials

**Reusability**

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

**Cross-Platform Compatibility**

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

**Expiration**

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

**Self-contained**

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

**Statelessness**

* The token contains all the information to identify the user, eliminating the need for the session state
* If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on
* JWT 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 favourite token type that many systems use these days due to its special features and advantages.
* JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the client/server side.

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

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjQ3Mjg0MiIsIm5hbWUiOiJBbWl0IFNhaGEiLCJlbWFpbCI6ImFtaXQ4Ny5zQHRjcy5jb20ifQ.OWyFwQWMwXie7FNZa1mPp-BOZeRllUWwfigwmrllg8s

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

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

{

"alg": "HS256",

"typ": "JWT"

}

**Payload** In the body, we can store details related to user, roles etc. which can be used later for Authentication and Authorization. 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.

{

"id": "472842",

"name": "Amit Saha",

"email": "amit87.s@tcs.com"

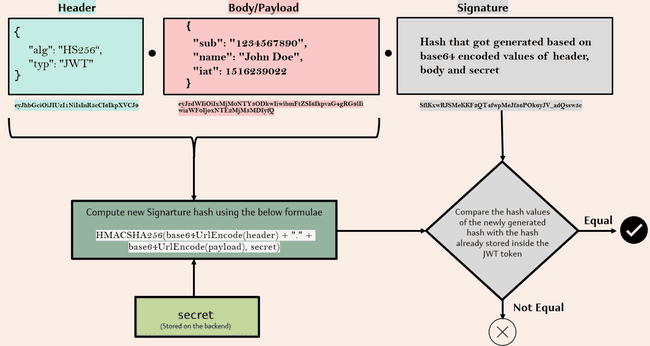
}

Signature

* 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 must 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.



# OAUTH2

[OAuth 2.0](https://tools.ietf.org/html/rfc6749), which stands for “Open Authorization”, is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.

OAuth 2.0 is an authorization protocol and NOT an authentication protocol

OAuth 2.0 uses Access Tokens. An **Access Token** is a piece of data that represents the authorization to access resources on behalf of the end-user. OAuth 2.0 doesn’t define a specific format for Access Tokens. However, in some contexts, the JSON Web Token (JWT) format is often used. This enables token issuers to include data in the token itself. Also, for security reasons, Access Tokens may have an expiration date.

**OAuth 2.0 Terminology - Standard Definitions**

**Resource Owner** The entity that can grant access to a protected resource. Typically, the end-user who owns the data (such as their email, profile, etc.). In OAuth flows, this is the person who authorizes an application to access their account.

**Client** The application requesting access to protected resources on behalf of the Resource Owner. This can be a website, mobile app, desktop application, or any other software that needs to access protected resources but cannot directly use the Resource Owner's credentials.

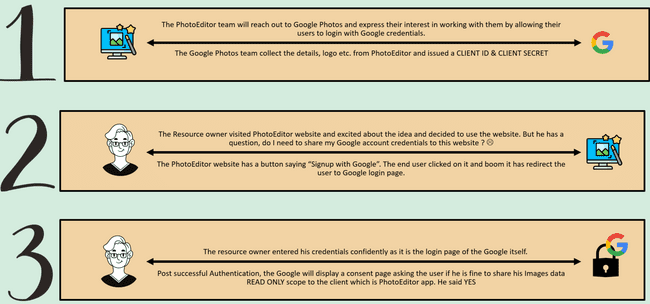
**Authorization Server** The server that authenticates the Resource Owner, obtains authorization decisions, and issues access tokens. It's responsible for validating user identity and handling the OAuth authorization process. It maintains information about clients and their allowed access scopes.

**Resource Server** The server that hosts the protected resources the Client wants to access. It accepts and validates access tokens from the Client and serves the requested resources if the token grants appropriate permissions. Often, this may be the same physical server as the Authorization Server but with a distinct logical role.

**Scopes** Granular permissions that define the extent of access granted to the Client. Scopes limit what the Client can do with the access token. Examples include read-only access to emails, write access to calendar events, or access to profile information. The Authorization Server enforces these scope limitations when issuing tokens.

These definitions establish the core components and roles within the OAuth 2.0 framework as formally specified in RFC 6749. Each plays a crucial part in enabling secure delegated access without sharing passwords across applications.

Sample OAUTH flow

A close-up of a text

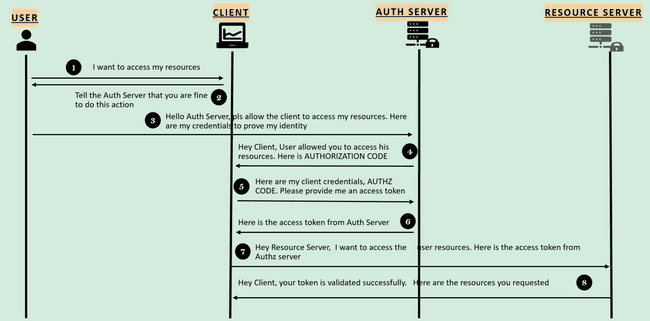
AI-generated content may be incorrect.

OAuth Grant Types

1. Authorization Code – When end user is involved
2. PKCE
3. Client Credentials
4. Device Code
5. Refresh Token
6. Legacy: Implicit Flow
7. Legacy: Password Grant

## Authorization Code

The Authorization Code grant is the most common OAuth 2.0 flow and is designed for applications that can securely store a client secret. It's optimized for security, particularly for web applications where the client can maintain confidentiality.



**Steps**

1. **Authorization Request**: The client redirects the user to the authorization server with its client ID, requested scope, a local state parameter, and a redirect URI.
2. **User Authentication & Consent**: The user logs in to the authorization server and approves the requested permissions.
3. **Authorization Code Response**: The authorization server redirects back to the client's redirect URI with a temporary authorization code and the original state parameter.
4. **Token Request**: The client sends a server-to-server request to the authorization server with the authorization code, client ID, client secret, and redirect URI.
5. **Token Response**: The authorization server validates the request and returns an access token, typically with a refresh token.
6. **Resource Access**: The client uses the access token to request protected resources from the resource server.

In the step 3, where client is making a request to Auth Server endpoint, must 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 registers 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** - like 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

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

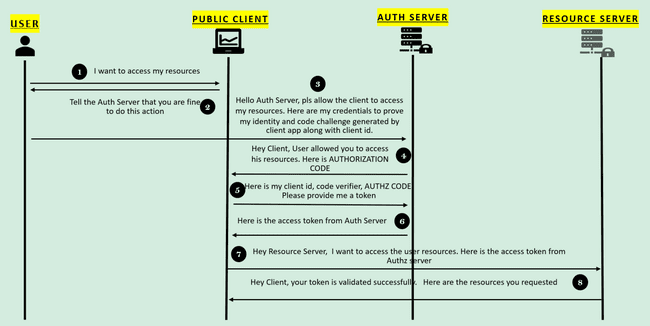
* code - the authorization code received from the above steps
* client\_id & client\_secret — the client credentials which are registered with the auth server. Please note that these are not user credentials
* grant\_type — With the value ‘authorization\_code’ which identifies the kind of grant type is used
* redirect\_uri

**Security Features**

* Authorization code is short-lived and single-use
* Full authentication flow happens server-to-server, not in the browser
* State parameter protects against CSRF attacks
* PKCE extension available for additional security with public clients
* Separates authentication from token issuance

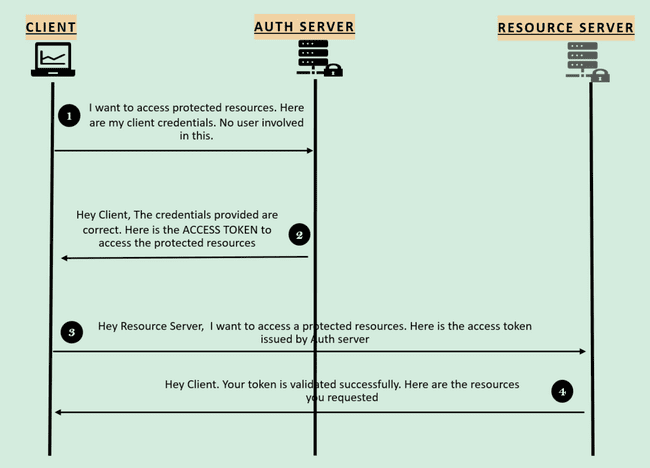
## PKCE

* 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 responds with ID Token and Access Token (and optionally, a Refresh Token).



## Client Credentials

The Client Credentials grant flow is one of the OAuth 2.0 authorization flows specifically designed for server-to-server authentication when there's no user context involved. It's the simplest OAuth 2.0 grant type and is used when the client itself is the resource owner.



**Steps**

1. The client (usually a server/service) authenticates with the authorization server using its client ID and client secret.
2. The authorization server validates these credentials.
3. If valid, the authorization server returns an access token to the client.
4. The client uses this access token to access protected resources on the resource server.

**Security Considerations**

* The client secret must be kept secure and never exposed to public clients
* Usually limited to confidential clients (servers that can securely store credentials)
* Scopes should be strictly limited to only what's necessary
* No refresh tokens are typically issued since the client can always request a new token

This flow is ideal for backend services that need to communicate with other APIs without user interaction, providing a secure authentication mechanism for service-to-service communication.

Authorization server examples

1. Key Cloak
2. Okta
3. Amazon Cognito
4. Forgerock
5. Fusion auth

# Key cloak

Local set up

docker run -p 8080:8080 -e KC\_BOOTSTRAP\_ADMIN\_USERNAME=admin -e KC\_BOOTSTRAP\_ADMIN\_PASSWORD=admin quay.io/keycloak/keycloak:26.1.4 start-dev

Set up on ec2

**SET UP SSL**

* **Step 1:** Install docker

*sudo yum update -y*

*sudo yum install -y docker*

*sudo service docker start*

*sudo usermod -a -G docker ec2-user*

* **Step 2:** Install OpenSSL

First, ensure OpenSSL is installed on your EC2 instance:

*sudo yum install -y openssl*

* **Step 3:** Generate a Self-Signed Certificate

1. Navigate to a directory where you want to store the certificate

*mkdir -p ~/keycloak-cert && cd ~/keycloak-cert*

1. Generate a private key:

*openssl genpkey -algorithm RSA -out keycloak.key*

1. Create a certificate signing request (CSR):

*openssl req -new -key keycloak.key -out keycloak.csr*

Fill in details as prompted (Common Name should be your EC2 public IP or domain).

1. Generate a self-signed certificate valid for 1 year:

*openssl x509 -req -days 365 -in keycloak.csr -signkey keycloak.key -out keycloak.crt*

* **Step 4:** Convert Certificate to PKCS12 Format

openssl pkcs12 -export -inkey keycloak.key -in keycloak.crt -out keycloak.p12 -name keycloak -password pass:changeit

* **Step 5:** Convert to Java Keystore (JKS)

1. Install keytool (part of OpenJDK):

*sudo yum install -y java-17-amazon-corretto*

1. Convert the PKCS12 file to a Java Keystore:

* **Step 6:** Convert Certificate to PKCS12 Format

*keytool -importkeystore -srckeystore keycloak.p12 -srcstoretype PKCS12 -destkeystore keycloak.jks -deststoretype JKS -deststorepass changeit -srcstorepass changeit -alias keycloak*

* **Step 7:** Run Keycloak with HTTPS

Run Keycloak with HTTPS enabled using your generated keystore:

*docker run -d -p 8443:8443 \*

*-e KC\_HTTPS\_CERTIFICATE\_FILE=/opt/keycloak/conf/keycloak.crt \*

*-e KC\_HTTPS\_CERTIFICATE\_KEY\_FILE=/opt/keycloak/conf/keycloak.key \*

*-e KC\_HOSTNAME=$(curl -s ifconfig.me) \*

*-e KC\_BOOTSTRAP\_ADMIN\_USERNAME=admin \*

*-e KC\_BOOTSTRAP\_ADMIN\_PASSWORD=admin \*

*-v ~/keycloak-cert:/opt/keycloak/conf \*

*quay.io/keycloak/keycloak:26.1.4 start-dev --https-port=8443 --https-key-store-file=/opt/keycloak/conf/keycloak.jks --https-key-store-password=changeit*

* **Step 8:** Set up security Group
* **Step 9:** Test

# Authorization flow types

## Client credentials

Between 2 api