Introduction to Application Security

🔐🔐🔐

Terminology

Principal

A principal in computer security is an entity e.g. a user that can be authenticated by a computer system or network.

Authority

- A permission that is granted to a user to perform some action.
- It can take the form of a privilege e.g. `repo:create` or `REPO_CREATE`.
- Or a role e.g. `ROLE_ADMIN`, `ROLE_USER`.

Authentication

Are you who you are you say you are?

- Password
- Fingerprint

Authorization

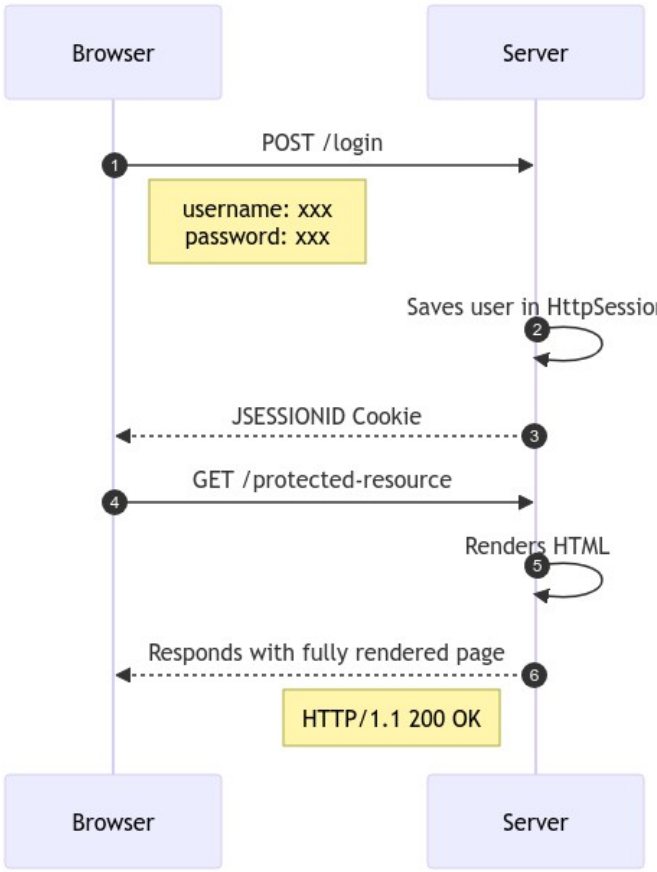
Do you have the *authority* to perform an action in the system?

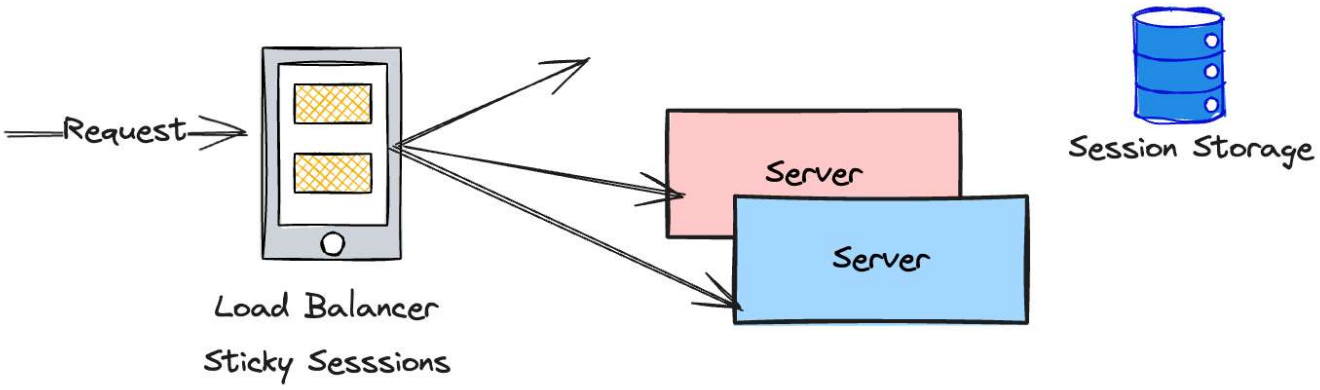
- RBAC
  - Authority / Privilege → e.g. `CAN_CREATE_PR` or `pr:write`
  - Role → e.g. `ADMIN`
- ABAC → Working hours
- ReBAC → did this user create this PR? If so they can edit it

Stateful vs Stateless applications

Server-side rendered apps

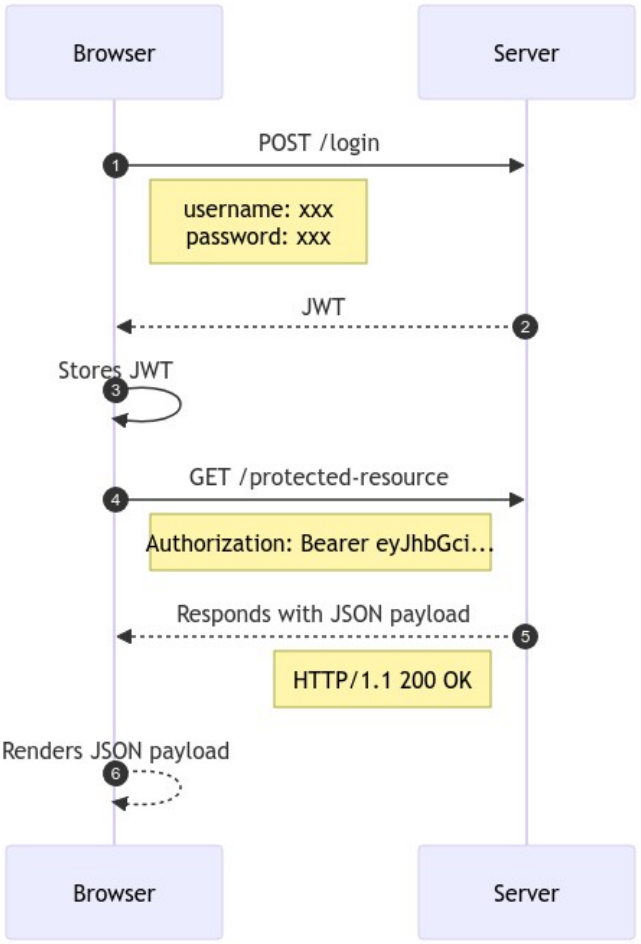
- Stateful (HttpSession)
- Needs sticky sessions or a distributed session store e.g. Redis (See `Spring Session`) for multiple instances

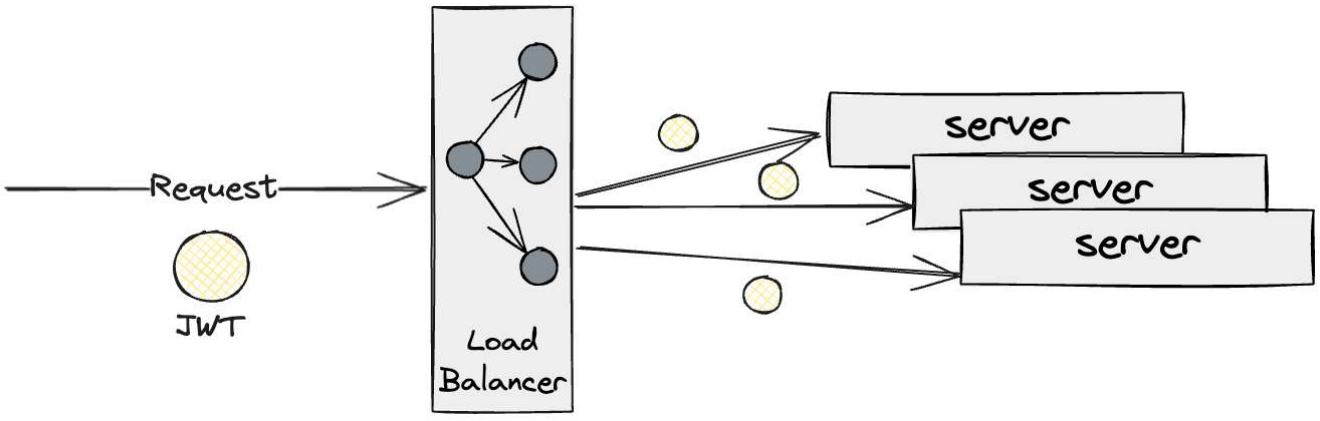




SPA + REST API

- Stateless
- Scalable without needing a distributed session store e.g. Redis


Browser
Server
POST /login
1
username: xxx
password: xxx
JWT
2
Stores JWT
3
GET /protected-resource
4
Authorization: Bearer eyJhbGci...
Responds with JSON payload
5
HTTP/1.1 200 OK
Renders JSON payload
6
Browser
Server


Request
JWT
Load Balancer
server
server
server

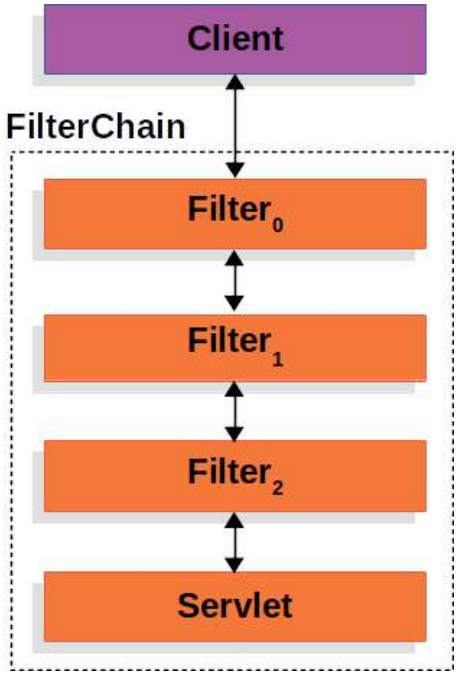
Kahoot x2

Spring Security

🌿🌿🌿

Spring Security Architecture

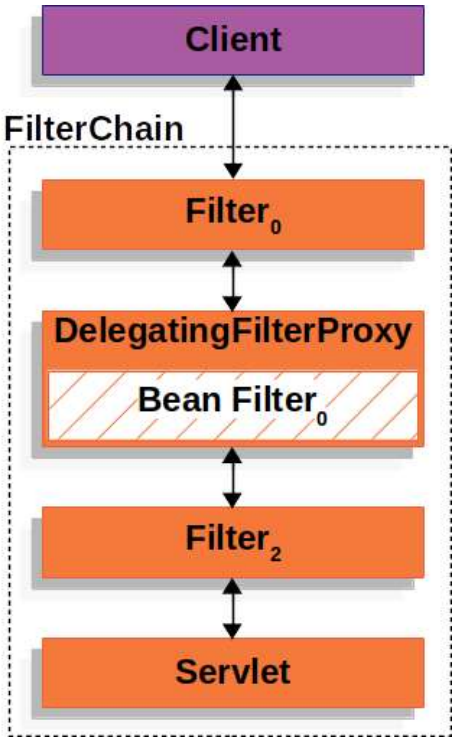
- 🔗 Spring Security Architecture
- Normal filter chain



- `DelegatingFilterProxy` is a filter that binds the lifecycle of a `jakarta.servlet.Filter` `@Bean` in the `ApplicationContext` with the lifecycle of the servlet container
- The bean is fetched lazily whenever the filter is invoked
- **Delegating**: delegates to a Spring bean `jakarta.servlet.Filter`
- **Filter**: it implements the `Filter` interface
- **Proxy**: Implementation of the Virtual proxy design pattern

> ⓘ **Virtual proxy**
>
> In place of a complex or heavy object, a skeleton representation may be advantageous in some cases. When an underlying image is huge in size, it may be represented using a virtual proxy object, loading the real object on demand.
> Proxy pattern - Wikipedia

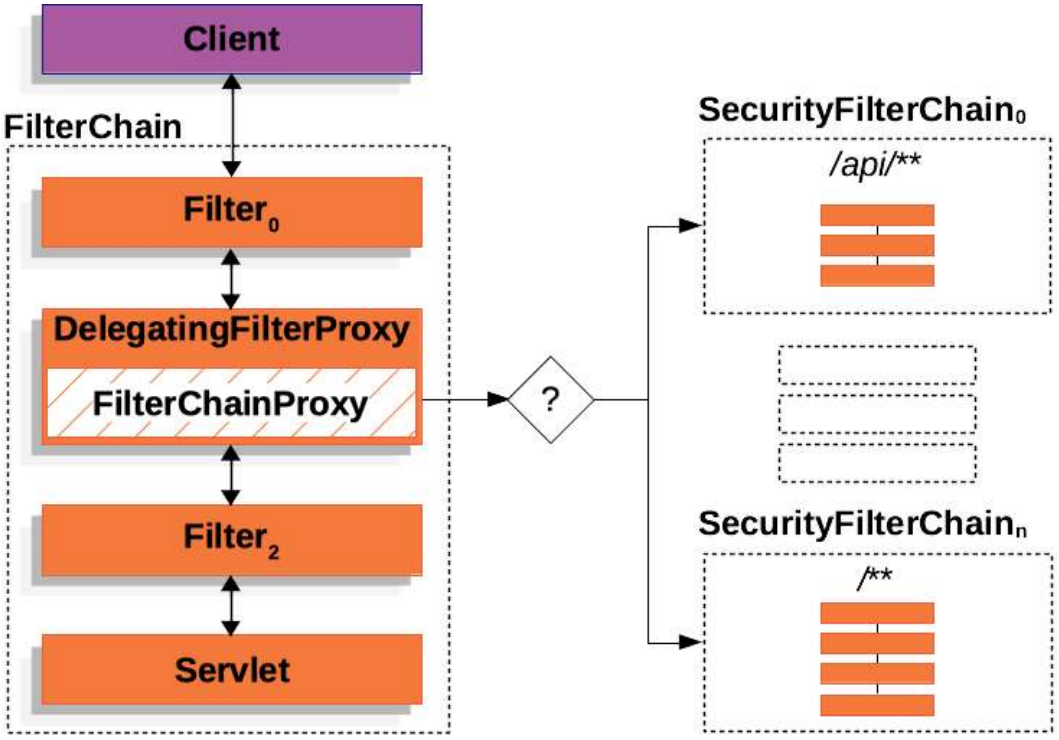
```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
{
        Filter delegate = getFilterBean(someBeanName);
        delegate.doFilter(request, response);
}
```



- Spring Security's `Filter` is `FilterChainProxy`
- It's a whole `FilterChain` inside a single `Filter`, namely the `SecurityFilterChain`

```
public interface SecurityFilterChain {
        boolean matches(HttpServletRequest request);
        List<Filter> getFilters();
}
```

```
public class FilterChainProxy extends GenericFilterBean {
        private List<SecurityFilterChain> filterChains;
        // ...
}
```



- Why `FilterChainProxy?`
  - `FilterChainProxy` performs tasks that are not optional, it is the start of Spring Security's Servlet support
  - Decides which `SecurityFilterChain` should be invoked based on `jakarta.servlet.HttpServletRequest`, not just its URI like a servlet container
- Want to see which Filters of the SecurityFilterChain are invoked?

```
# Filters are printed at application started at INFO level
2023-06-14T08:55:22.321-03:00  INFO 76975 --- [           main]
o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [
org.springframework.security.web.session.DisableEncodeUrlFilter@404db674,
org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@50
f097b5,
org.springframework.security.web.context.SecurityContextHolderFilter@6fc6deb7,
org.springframework.security.web.header.HeaderWriterFilter@6f76c2cc,
org.springframework.security.web.csrf.CsrfFilter@c29fe36,
org.springframework.security.web.authentication.logout.LogoutFilter@ef60710,
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@7c2df
a2,
org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@4397a6
39,
org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter@7add8
38c,
org.springframework.security.web.authentication.www.BasicAuthenticationFilter@5cc9d3d0,
org.springframework.security.web.savedrequest.RequestCacheAwareFilter@7da39774,
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@32b087
6c,
org.springframework.security.web.authentication.AnonymousAuthenticationFilter@3662bdff,
org.springframework.security.web.access.ExceptionTranslationFilter@77681ce4,
org.springframework.security.web.access.intercept.AuthorizationFilter@169268a7]
```

```
logging.level.org.springframework.security=TRACE
```

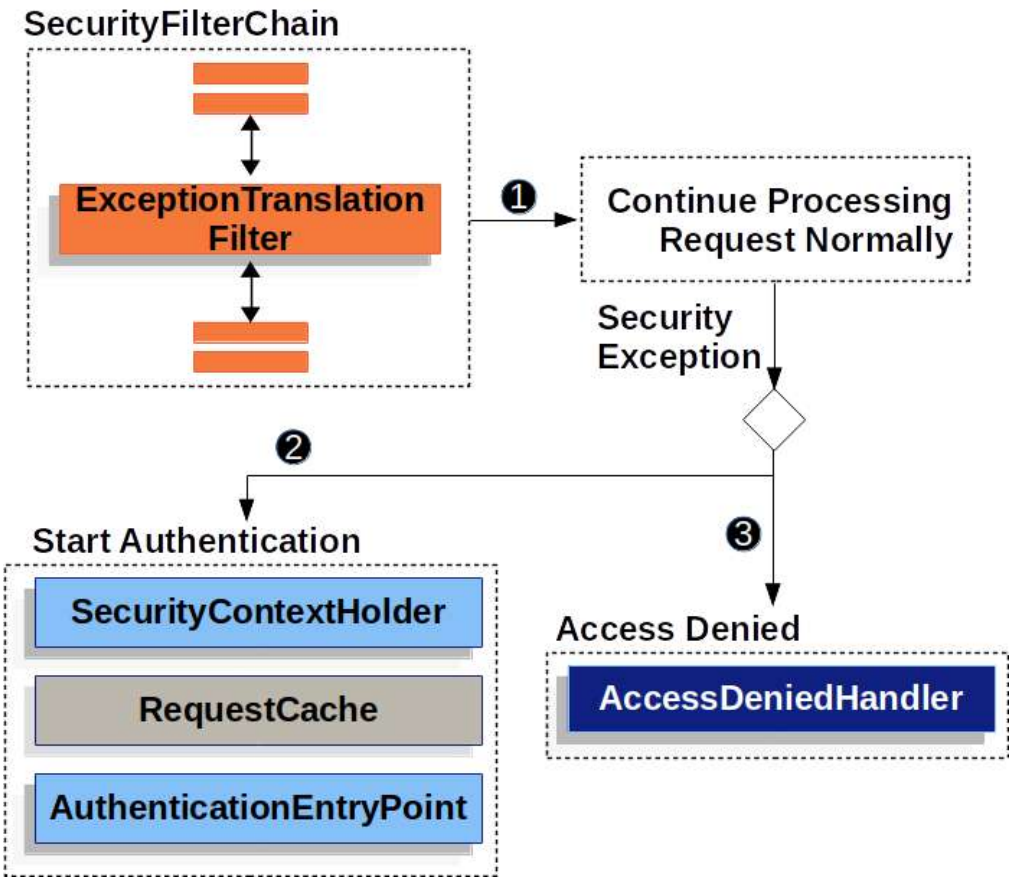
```
2023-06-14T09:44:25.797-03:00 DEBUG 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Securing POST /hello
2023-06-14T09:44:25.797-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking DisableEncodeUrlFilter (1/15)
2023-06-14T09:44:25.798-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking WebAsyncManagerIntegrationFilter
(2/15)
2023-06-14T09:44:25.800-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking SecurityContextHolderFilter (3/15)
2023-06-14T09:44:25.801-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking HeaderWriterFilter (4/15)
2023-06-14T09:44:25.802-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.security.web.FilterChainProxy        : Invoking CsrfFilter (5/15)
2023-06-14T09:44:25.814-03:00 DEBUG 76975 --- [nio-8080-exec-1]
o.s.security.web.csrf.CsrfFilter         : Invalid CSRF token found for
http://localhost:8080/hello
2023-06-14T09:44:25.814-03:00 DEBUG 76975 --- [nio-8080-exec-1]
o.s.s.w.access.AccessDeniedHandlerImpl   : Responding with 403 status code
2023-06-14T09:44:25.814-03:00 TRACE 76975 --- [nio-8080-exec-1]
o.s.s.w.header.writers.HstsHeaderWriter  : Not injecting HSTS header since it did not
match request to [Is Secure]
```

ExceptionTranslationFilter

- Handles `AuthenticationException` and `AccessDeniedException` that happen after it.

```
# Handles its own AuthenticationException
BasicAuthenticationFilter
#...
ExceptionTranslationFilter
AuthorizationFilter
```

```
// ExceptionTranslationFilter.java
try {
        filterChain.doFilter(request, response);
} catch (AccessDeniedException | AuthenticationException ex) {
        if (!authenticated || ex instanceof AuthenticationException) {
                startAuthentication();
        } else {
                accessDenied();
        }
}
```



- 1 First, the `ExceptionTranslationFilter` invokes `FilterChain.doFilter(request, response)` to invoke the rest of the application.
- 2 If the user is not authenticated or it is an `AuthenticationException`, then *Start Authentication*.
  - The SecurityContextHolder is cleared out.
  - The `HttpServletRequest` is saved so that it can be used to replay the original request once authentication is successful.

    - The `AuthenticationEntryPoint` is used to request credentials from the client. For example, it might redirect to a log in page or send a `WWW-Authenticate` header.
- 3️⃣ Otherwise, if it is an `AccessDeniedException`, then *Access Denied*. The `AccessDeniedHandler` is invoked to handle access denied.

`RequestCache`

- The `HttpServletRequest` is saved in the `RequestCache`. When the user successfully authenticates, the `RequestCache` is used to replay the original request. The `RequestCacheAwareFilter` is what uses the `RequestCache` to save the `HttpServletRequest`.
- By default, an `HttpSessionRequestCache` is used. The code below demonstrates how to customize the `RequestCache` implementation that is used to check the `HttpSession` for a saved request if the parameter named `continue` is present.
- There is also a `CookieRequestCache` suitable for stateless applications.

Spring Security `AutoConfiguration`

- 👆 Hello Spring Security

Spring Security default `SecurityFilterChain`

```
# Disables saving the HTTP session by url re-writing
session.DisableEncodeUrlFilter@3894d1b1,
# Manages saving of authentication object when using async servlet APIs
context.request.async.WebAsyncManagerIntegrationFilter@5f6b1f19,
# Initializes SecurityContextHolder with SecurityContext object
# if the authentication is persisted already e.g. in the HTTP session
context.SecurityContextHolderFilter@65a01901,
# Responsible for writing security HTTP headers e.g. X-Frame-Options
header.HeaderWriterFilter@5319efc0,
# Responsible for handling CORS
org.springframework.web.filter.CorsFilter@253c7189,
# Responsible for validating CSRF tokens to protect against CSRF
csrf.CsrfFilter@648bfebe,
# Responsible for calling LogoutHandlers when a user logs out
authentication.logout.LogoutFilter@15e748b5,
```

```
# For allowing login through a form using username and password
authentication.UsernamePasswordAuthenticationFilter@15096b0e,
# Generates default login page
authentication.ui.DefaultLoginPageGeneratingFilter@17810908,
# Generates default logout page
authentication.ui.DefaultLogoutPageGeneratingFilter@cfe72b0,
# Allows basic HTTP authentication
# using Authorization: Basic username:password
authentication.www.BasicAuthenticationFilter@3e3d8e6c,
# Handles retrieving the original request you intended after successful login
# e.g. /admin → login → back to /admin
savedrequest.RequestCacheAwareFilter@342a5b57,
```

```
# Wraps HttpServletRequest with authentication-related
# method implementing decorator
```

```
servletapi.SecurityContextHolderAwareRequestFilter@7305191e,
# Creates an anonymous SecurityContext if user is not authenticated
authentication.AnonymousAuthenticationFilter@12f933c7,
# Handles AuthenticationException and AccessDeniedException
access.ExceptionTranslationFilter@518ed9b4,
# Restricts access to urls based on role, is the user authenticated, etc,
# by delegating to AuthorizationManager
access.intercept.AuthorizationFilter@bbf361a
```

> 🕵️ Go through each in Spring Security source code

Default Headers provided by Spring Security in responses

```
# Default headers

# If a user authenticates to view sensitive information and then logs out, we do not want
a malicious user to be able to click the back button to view the sensitive information
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0

# Don't sniff content type, to prevent XSS
X-Content-Type-Options: nosniff

# Don't redirect http to https, but treat this site as https always by browser to protect
against MITM attacks
Strict-Transport-Security: max-age=31536000 ; includeSubDomains

# Prevent Clickjacking attacks
X-Frame-Options: DENY

# Disable browser XSS protection, as it deprecated and causes vulnerabilities to other XSS
attacks
X-XSS-Protection: 0
```

- 🔗 Headers
- 🔗 Customizing security headers
  - Spring Security allows you to customize each of those default headers and add your own

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

        @Bean
        public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
                http
                        // ...
                        .headers(headers → headers
                                .frameOptions(frameOptions → frameOptions
                                        .sameOrigin()
                                )
                        );
                return http.build();
```

```
        }
}
```

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
        @Bean
        public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
                http
                        // ...
                        .headers(headers -> headers
                                .addHeaderWriter(new
XFrameOptionsHeaderWriter(XFrameOptionsMode.SAMEORIGIN))
                        );
                return http.build();
        }
}
```

- 👆 Running behind a proxy

Password Storage

- 🗒️ Plaintext
- #️⃣ Hash e.g. `SHA-256`
  - 🏳️‍🌈 Rainbow Tables
- 🧂 Random salt in plaintext + hashed password
  - Salt makes rainbow tables useless
  - Cryptographic hashes are unsafe → billions of hash calculations / second
- Salt + Adaptive hash functions
  - Intentionally resource (CPU + Memory intensive) ~ 1 second to verify a password - makes brute force attacks even more impractical
  - Tuned "work factor"
- `PasswordEncoder` interface >>
- `DelegatingPasswordEncoder` is the default password encoder for good reason
  - Allows you to migrate to more secure hash algorithms as they get implemented
  - Allows you to support legacy hashed passwords
  - `{algorithmId}password`
- 🔗 Password Storage in Spring Security

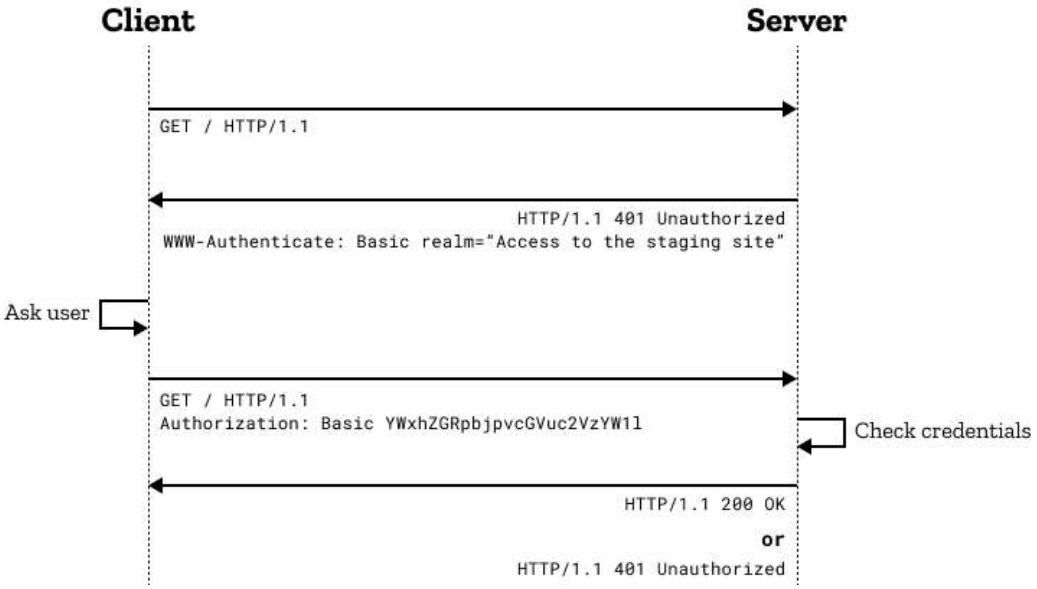
Spring Security Authentication

Contracts used in Authentication Architecture

- 🔗 Authentication Architecture
- We will explain the Authentication Architecture using a Basic Authentication example.

Basic Authentication Primer

- 🔗 What is Basic Authentication?



Danger

The "Basic" authentication scheme used in the diagram above sends the credentials encoded but not encrypted. This would be completely insecure unless the exchange was over a secure connection (HTTPS/TLS).

`SecurityContextHolder` and `SecurityContext`

- At the heart of Spring Security's authentication model is the `SecurityContextHolder`. It contains the SecurityContext.
- The SecurityContextHolder is where Spring Security stores the details of who is authenticated.
- Spring Security *does not care* how the SecurityContextHolder is populated. *If it contains a value, it is used as the currently authenticated user.

```
SecurityContext context = SecurityContextHolder.createEmptyContext();
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER");
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);
```

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

```
public class SecurityContextHolder {

        public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";

        public static final String MODE_INHERITABLETHREADLOCAL =
"MODE_INHERITABLETHREADLOCAL";

        public static final String MODE_GLOBAL = "MODE_GLOBAL";

        private static final String MODE_PRE_INITIALIZED = "MODE_PRE_INITIALIZED";

        public static final String SYSTEM_PROPERTY = "spring.security.strategy";

        private static String strategyName = System.getProperty(SYSTEM_PROPERTY);

        private static SecurityContextHolderStrategy strategy;

        // ...
}
```

`Authentication`

The `Authentication` interface serves two main purposes within Spring Security:

- An input to `AuthenticationManager` to provide the credentials a user has provided to authenticate. When used in this scenario, `isAuthenticated()` returns `false`.
- Represent the currently authenticated user. You can obtain the current `Authentication` from the `SecurityContext`.

The `Authentication` contains:

- `principal`: Identifies the user. When authenticating with a username/password this is often an instance of `UserDetails`.
- `credentials`: Often a password. In many cases, this is cleared after the user is authenticated, to ensure that it is not leaked.
- `authorities`: The `GrantedAuthority` instances are high-level permissions the user is granted. Two examples are roles and scopes.

`GrantedAuthority`

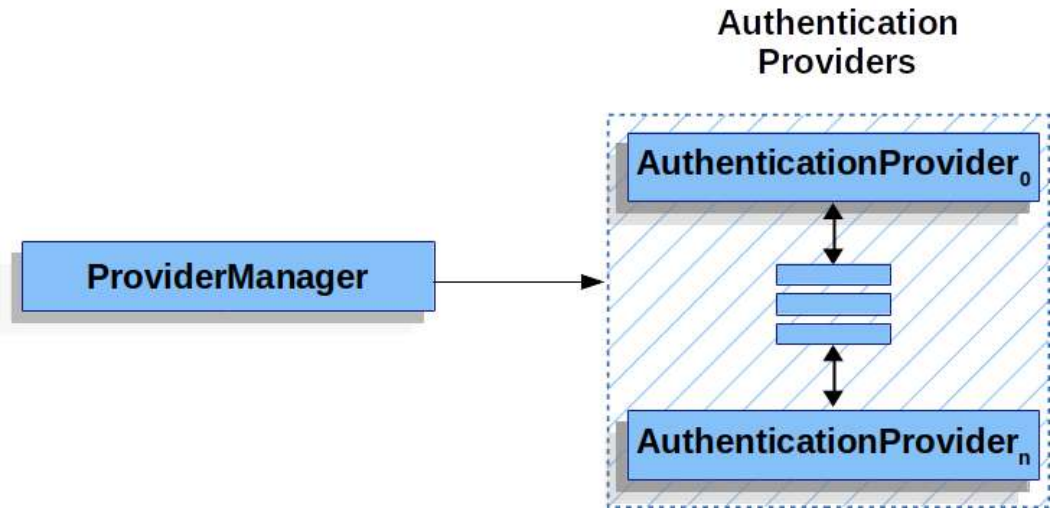
- `GrantedAuthority` instances are high-level permissions that the user is granted. Two examples are roles and scopes.

`AuthenticationManager`

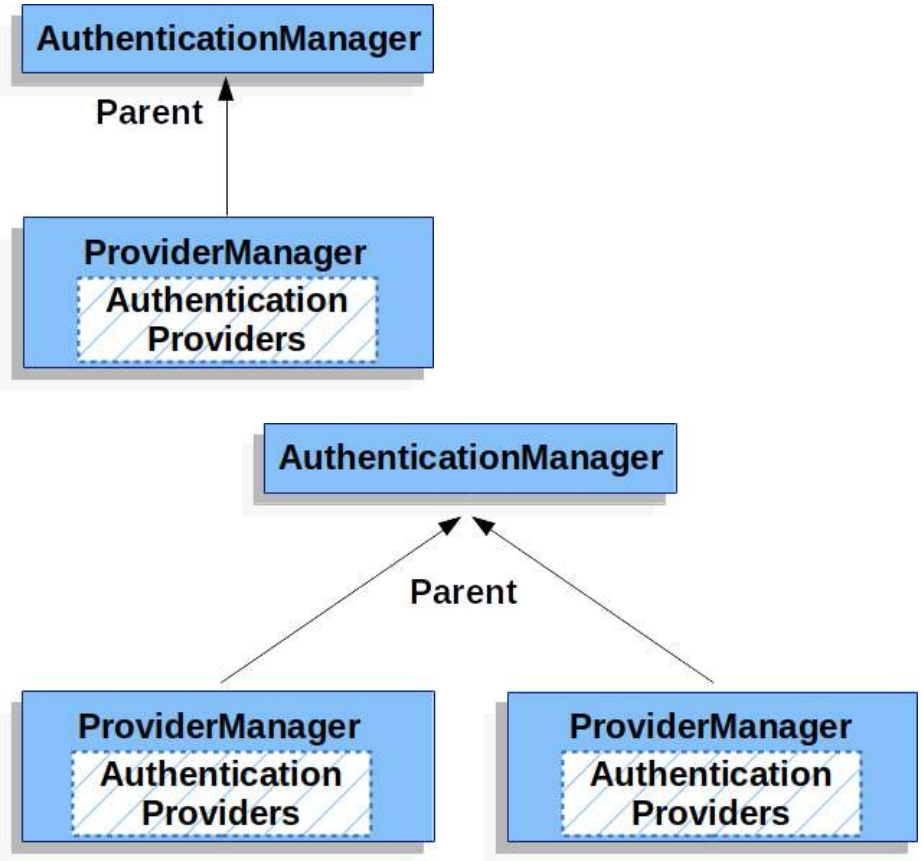
- `AuthenticationManager` is the API that defines how Spring Security's Filters perform `authentication`. The `Authentication` that is returned is then set on the `SecurityContextHolder` by the controller (that is, by `Spring Security's Filters instances`) that invoked the `AuthenticationManager`.

- While the implementation of `AuthenticationManager` could be anything, the most common implementation is `ProviderManager`.

`ProviderManager`



- `ProviderManager` is the most commonly used implementation of `AuthenticationManager`.
- `ProviderManager` delegates to a `List` of `AuthenticationProvider` instances.
- Each `AuthenticationProvider` has an opportunity to indicate that authentication should be successful, fail, or indicate it cannot make a decision and allow a downstream `AuthenticationProvider` to decide.
- If none of the configured `AuthenticationProvider` instances can authenticate, authentication fails with a `ProviderNotFoundException`, which is a special `AuthenticationException` that indicates that the `ProviderManager` was not configured to support the type of `Authentication` that was passed into it.



- `ProviderManager` also allows configuring an optional parent `AuthenticationManager`, which is consulted in the event that no `AuthenticationProvider` can perform authentication. The parent can be any type of `AuthenticationManager`, but it is often an instance of `ProviderManager`.
- This is somewhat common in scenarios where there are multiple `SecurityFilterChain` instances that have some authentication in common (the shared parent `AuthenticationManager`), but also different authentication mechanisms (the different `ProviderManager` instances).
- By default, `ProviderManager` tries to clear any sensitive credentials information from the `Authentication` object that is returned by a successful authentication request. This prevents information, such as passwords, being retained longer than necessary in the `HttpSession`.
- This may cause issues when you use a cache of user objects, for example, to improve performance in a stateless application.

```
// ProviderManager.java
if (this.eraseCredentialsAfterAuthentication && (result instanceof CredentialsContainer))
{
    // Authentication is complete. Remove credentials and other secret data
    // from authentication
    ((CredentialsContainer) result).eraseCredentials();
}
```

`AuthenticationProvider`

- You can inject multiple `AuthenticationProvider`s instances into `ProviderManager`. Each `AuthenticationProvider` performs a specific type of authentication. For example, `DaoAuthenticationProvider` supports username/password-based authentication, while `JwtAuthenticationProvider` supports authenticating a JWT token.

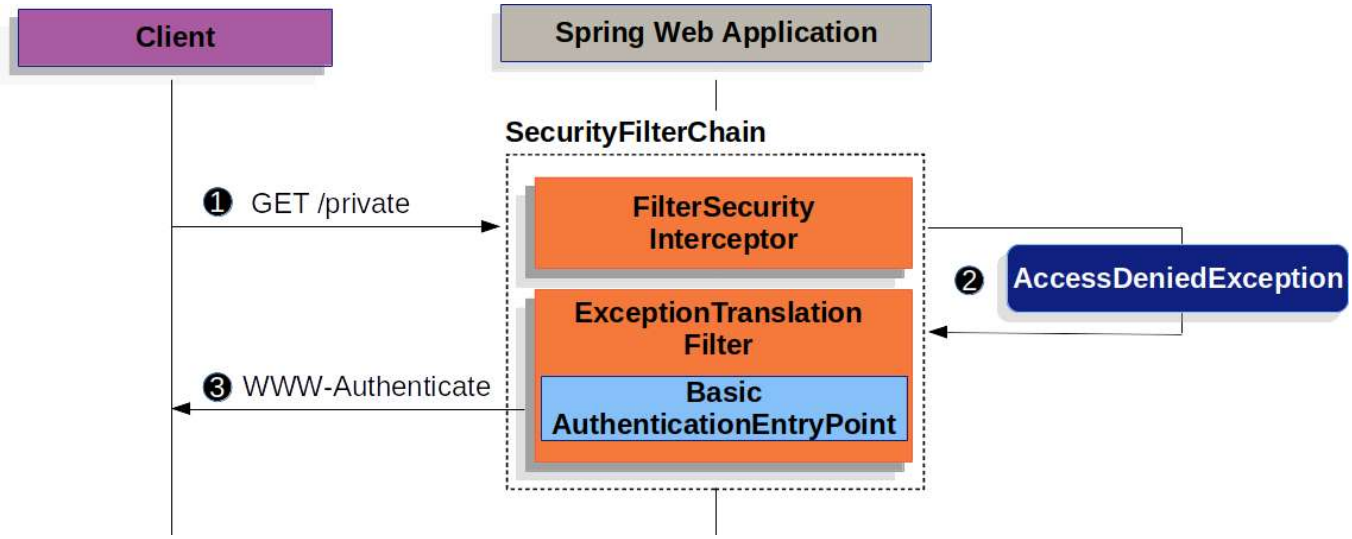
Requesting Credentials with `AuthenticationEntryPoint`

- `AuthenticationEntryPoint` is used to send an HTTP response that requests credentials from a client.
- If a client makes an unauthenticated request to a resource that they are not authorized to access, an implementation of `AuthenticationEntryPoint` is used to request credentials from the client.
- The `AuthenticationEntryPoint` implementation might perform a redirect to a log in page, respond with an `WWW-Authenticate` header, or take other action.

```
public class BasicAuthenticationEntryPoint implements AuthenticationEntryPoint {

        @Override
        public void commence(HttpServletRequest request, HttpServletResponse response,
                        AuthenticationException authException) throws IOException {
                response.setHeader("WWW-Authenticate", "Basic realm=\"" + this.realmName +
"\"");
                response.sendError(HttpStatus.UNAUTHORIZED.value(),
HttpStatus.UNAUTHORIZED.getReasonPhrase());
        }
}
```

```
// BasicAuthenticationFilter.java
                catch (AuthenticationException ex) {
                        this.securityContextHolderStrategy.clearContext();
                        this.logger.debug("Failed to process authentication request", ex);
                                this.authenticationEntryPoint.commence(request, response,
ex);
                        }
                        return;
                }
```

Basic Authentication

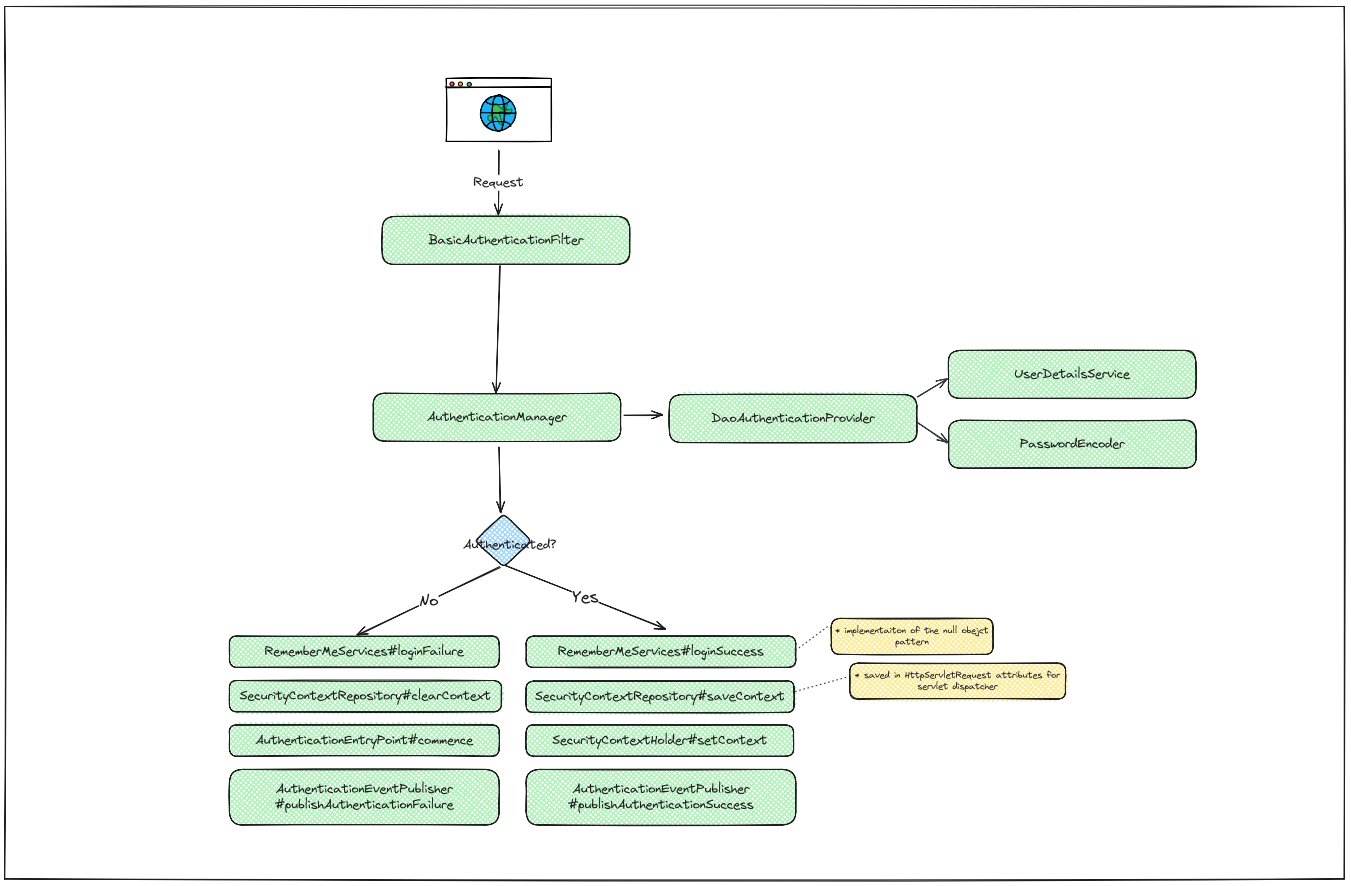


❶ First, a user makes an unauthenticated request to the resource `/private` for which it is not authorized.

❷ Spring Security's `AuthorizationFilter` indicates that the unauthenticated request is *Denied* by throwing an `AccessDeniedException`.

❸ Since the user is not authenticated, `ExceptionTranslationFilter` initiates *Start Authentication*. The configured `AuthenticationEntryPoint` is an instance of `BasicAuthenticationEntryPoint`, which sends a WWW-Authenticate header. The `RequestCache` is typically a `NullRequestCache` that does not save the request since the client is capable of replaying the requests it originally requested.

When a client receives the `WWW-Authenticate` header, it knows it should retry with a username and password. The following image shows the flow for the username and password being processed:



- 🔗 Basic Authentication in Spring Security

☰👨🏻‍💻

`SecurityConfig`

UserDetailsService

```
public interface UserDetailsService {
        UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

🔗 UserDetailsService implementations

- `InMemoryUserDetailsManager`
- `JdbcUserDetailsManager`

☰👨🏻‍💻

`InMemoryUserDetailsServiceConfig`
`PasswordEncoderConfig`
`JpaUserDetailsServiceConfig`
`RoleHierarchyConfig`

Other types of authentication

- 🔗 LDAP authentication

    - LDAP (Lightweight Directory Access Protocol) is often used by organizations as a central repository for user information and as an authentication service. It can also be used to store the role information for application users.
- Form authentication (for SSR applications)
- Digest authentication (not recommended)

Authentication Persistence and Session Management

- 🔗 Authentication Persistence and Session Management

Persistence Mechanism

```
public class SecurityContextHolderFilter extends GenericFilterBean {

private final SecurityContextRepository securityContextRepository;

private SecurityContextHolderStrategy securityContextHolderStrategy =
SecurityContextHolder
    .getContextHolderStrategy();

        private void doFilter(HttpServletRequest request, HttpServletResponse response,
FilterChain chain)
                        throws ServletException, IOException {
                Supplier<SecurityContext> deferredContext =
this.securityContextRepository.loadDeferredContext(request);
                try {

this.securityContextHolderStrategy.setDeferredContext(deferredContext);
                        chain.doFilter(request, response);
                }
                finally {
                        this.securityContextHolderStrategy.clearContext();
                        request.removeAttribute(FILTER_APPLIED);
                }
        }
}
```

Stateless applicaitons

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) {
                return httpSecurity
                                .sessionManagement(s -> s
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                                .build();
        }
```

Authentication Events

- 🔗 Authentication Events

```
@Component
public class AuthenticationEvents {
        @EventListener
    public void onSuccess(AuthenticationSuccessEvent success) {
                // ...
    }

    @EventListener
    public void onFailure(AbstractAuthenticationFailureEvent failures) {
                // ...
    }
}
```

Exception	Event
`BadCredentialsException`	`AuthenticationFailureBadCredentialsEvent`
`UsernameNotFoundException`	`AuthenticationFailureBadCredentialsEvent`
`AccountExpiredException`	`AuthenticationFailureExpiredEvent`
`ProviderNotFoundException`	`AuthenticationFailureProviderNotFoundEvent`
`DisabledException`	`AuthenticationFailureDisabledEvent`
`LockedException`	`AuthenticationFailureLockedEvent`
`AuthenticationServiceException`	`AuthenticationFailureServiceExceptionEvent`
`CredentialsExpiredException`	`AuthenticationFailureCredentialsExpiredEvent`
`InvalidBearerTokenException`	`AuthenticationFailureBadCredentialsEvent`

```
@Bean
public AuthenticationEventPublisher authenticationEventPublisher
        (ApplicationEventPublisher applicationEventPublisher) {
    AuthenticationEventPublisher authenticationEventPublisher =
        new DefaultAuthenticationEventPublisher(applicationEventPublisher);
    authenticationEventPublisher.setDefaultAuthenticationFailureEvent
        (GenericAuthenticationFailureEvent.class);
    return authenticationEventPublisher;
}
```

Logout

- 🔗 Handling Logouts - read if interested
- As we're dealing mainly with stateless applicaitons, there is no such thing as "logout".

Custom Authentication

📝🥷 Implementing API Key authentication

❓🦉 Kahoot x3

Spring Security Authorization

Authorization Architecture

- 🔗 Authorization Architecture

Authorities

- The `GrantedAuthority` interface has only one method:

```
String getAuthority();
```

- This method is used by an `AuthorizationManager` instance to obtain a precise `String` representation of the `GrantedAuthority`.
- By returning a representation as a `String`, a `GrantedAuthority` can be easily "read" by most `AuthorizationManager` implementations.
- If a `GrantedAuthority` cannot be precisely represented as a `String`, the `GrantedAuthority` is considered "complex" and `getAuthority()` must return `null`.
- This indicates to any `AuthorizationManager` that it needs to support the specific `GrantedAuthority` implementation to understand its contents.
- Spring Security includes one concrete `GrantedAuthority` implementation: `SimpleGrantedAuthority`. This implementation lets any user-specified `String` be converted into a `GrantedAuthority`.
- All `AuthenticationProvider` instances included with the security architecture use `SimpleGrantedAuthority` to populate the `Authentication` object.
- By default, role-based authorization rules include `ROLE_` as a prefix. This means that if there is an authorization rule that requires a security context to have a role of "USER", Spring Security will by default look for a `GrantedAuthority#getAuthority` that returns "ROLE_USER".

```
// You expose GrantedAuthorityDefaults using a static method to ensure that Spring publishes it before it initializes Spring Security's method security @Configuration classes
@Bean
static GrantedAuthorityDefaults grantedAuthorityDefaults() {
        return new GrantedAuthorityDefaults("MYPREFIX_");
}
```

Invocation Handling

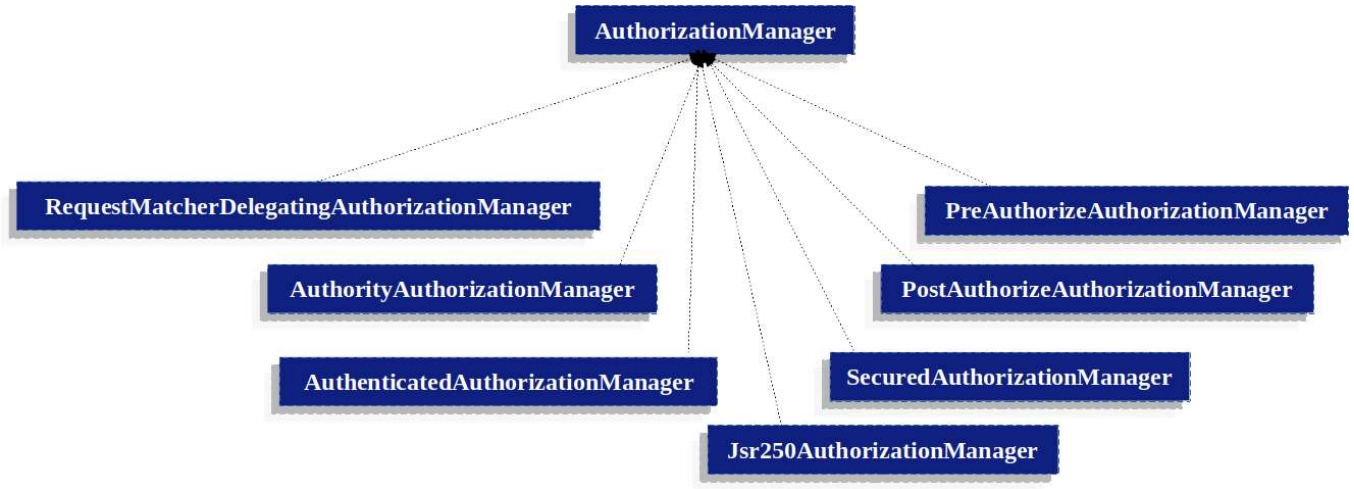
- Spring Security provides interceptors that control access to secure objects, such as method invocations or web requests.
- A pre-invocation decision on whether the invocation is allowed to proceed is made by `AuthorizationManager` instances.
- Also post-invocation decisions on whether a given value may be returned is made by `AuthorizationManager` instances.

AuthorizationManager

- `AuthorizationManager` supersedes both `AccessDecisionManager` and `AccessDecisionVoter`.
- `AuthorizationManager`s are called by Spring Security's request-based, method-based, and message-based authorization components and are responsible for making final access control decisions.

- The `AuthorizationManager` interface contains two methods:

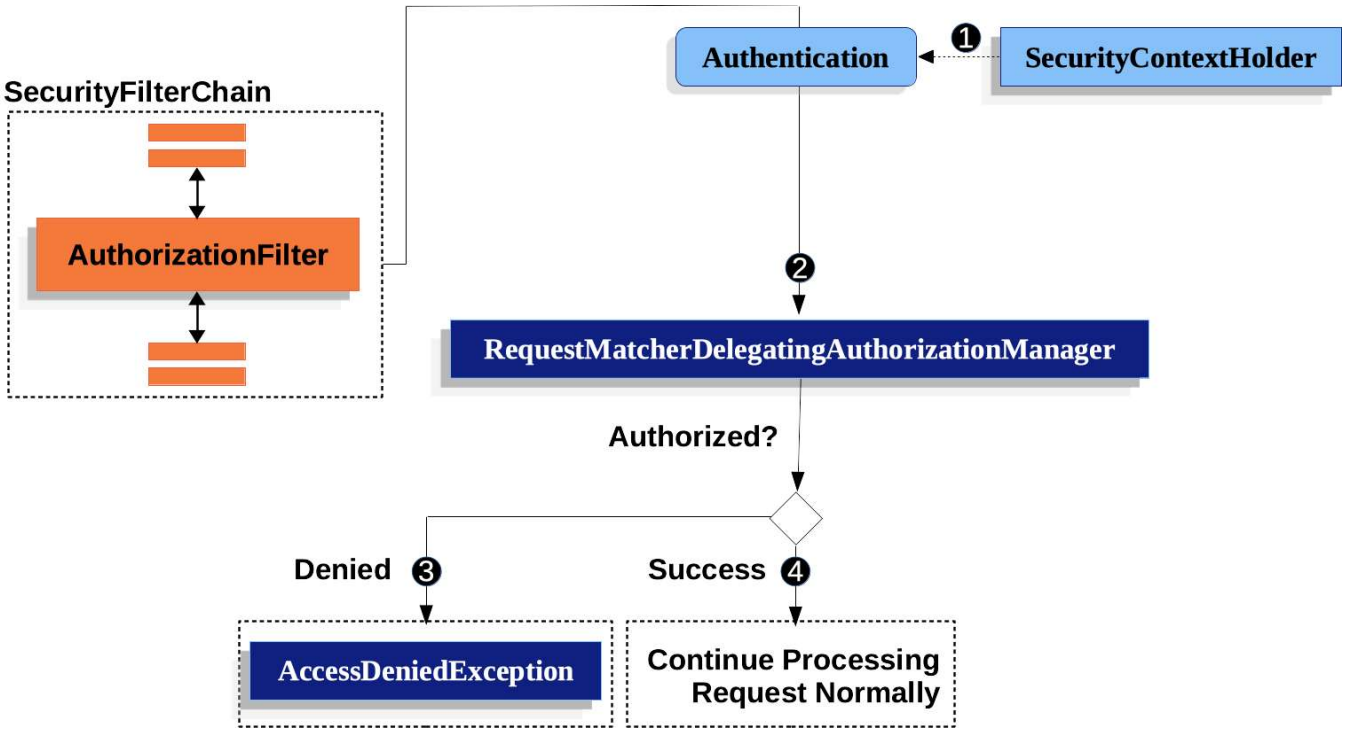
```
@FunctionalInterface
public interface AuthorizationManager<T> {
        // T can be HttpServletRequest, MethodInvocation, etc..

        // Clever usage of TemplateMethod to enforce an exception-based contract
        default void verify(Supplier<Authentication> authentication, T object) {
                AuthorizationDecision decision = check(authentication, object);
                // Null means this AuthorizationManager refrains from making a decision
                if (decision != null && !decision.isGranted()) {
                        throw new AccessDeniedException("Access Denied");
                }
        }

        @Nullable
        AuthorizationDecision check(Supplier<Authentication> authentication, T object);

}
```



- `RequestMatcherDelegatingAuthorizationManager` will match the request with the most appropriate delegate `AuthorizationManager` based on `RequestMatcher` e.g. matching against path, headers, or others.
- For method security, you can use `AuthorizationManagerBeforeMethodInterceptor` and `AuthorizationManagerAfterMethodInterceptor` which are aspects registered automatically by `@EnableMethodSecurity`.

Authorizing HttpServletRequest

- 🔗 Authorize HttpServletRequests



- `AuthorizationFilter` Is Last By Default
- All Dispatches Are Authorized
  - The `AuthorizationFilter` runs not just on every request, but on every dispatch. This means that the `REQUEST` dispatch needs authorization, but also `FORWARD`s, `ERROR`s, and `INCLUDE`s.

```
http
    .authorizeHttpRequests((authorize) → authorize
        .dispatcherTypeMatchers(DispatcherType.FORWARD, DispatcherType.ERROR).permitAll()
        .requestMatchers("/endpoint").permitAll()
        .anyRequest().denyAll()
    )
```

- `Authentication` Lookup is Deferred
  - This matters with `authorizeHttpRequests` when requests are always permitted or always denied. In those cases, the `Authentication` is not queried, making for a faster request.

Authorizing Endpoints

```
// MVC matchers
@Bean
SecurityFilterChain web(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests((authorize) → authorize
                        .requestMatchers("/endpoint").hasAuthority("USER")
                        .anyRequest().authenticated()
                )
        // ...

        return http.build();
}

// Regex
```

```
http
    .authorizeHttpRequests((authorize) → authorize
        .requestMatchers(RegexRequestMatcher.regexMatcher("/resource/[A-Za-z0-
9]+")).hasAuthority("USER")
        .anyRequest().denyAll()
    )

// Matching by HTTP method
http
    .authorizeHttpRequests((authorize) → authorize
        .requestMatchers(HttpMethod.GET).hasAuthority("read")
        .requestMatchers(HttpMethod.POST).hasAuthority("write")
        .anyRequest().denyAll()
    )

// Complex example
@Bean
SecurityFilterChain web(HttpSecurity http) throws Exception {
        http
                // ...
                .authorizeHttpRequests(authorize → authorize
            .dispatcherTypeMatchers(FORWARD, ERROR).permitAll()
                        .requestMatchers("/static/**", "/signup", "/about").permitAll()
                        .requestMatchers("/admin/**").hasRole("ADMIN")
                        .requestMatchers("/db/**").access(allOf(hasAuthority("db"),
hasRole("ADMIN")))
                        .anyRequest().denyAll()
                );

        return http.build();
}
```

- Favor `permitAll` over `ignoring`. When you have static resources it can be tempting to configure the filter chain to ignore these values. A more secure approach is to permit them using `permitAll` like so:

```
http
    .authorizeHttpRequests((authorize) → authorize
        .requestMatchers("/css/**").permitAll()
        .anyRequest().authenticated()
    )
```

- In this past, this came with a performance tradeoff since the session was consulted by Spring Security on every request.
- As of Spring Security 6, however, the session is no longer pinged unless required by the authorization rule. Because the performance impact is now addressed, Spring Security recommends using at least `permitAll` for all requests.
- Security matchers decide if this `SecurityFilterChain` should be invoked for this request

```
http
        .securityMatcher("/api/**")
```

MethodSecurity

- 🔗 Method Security

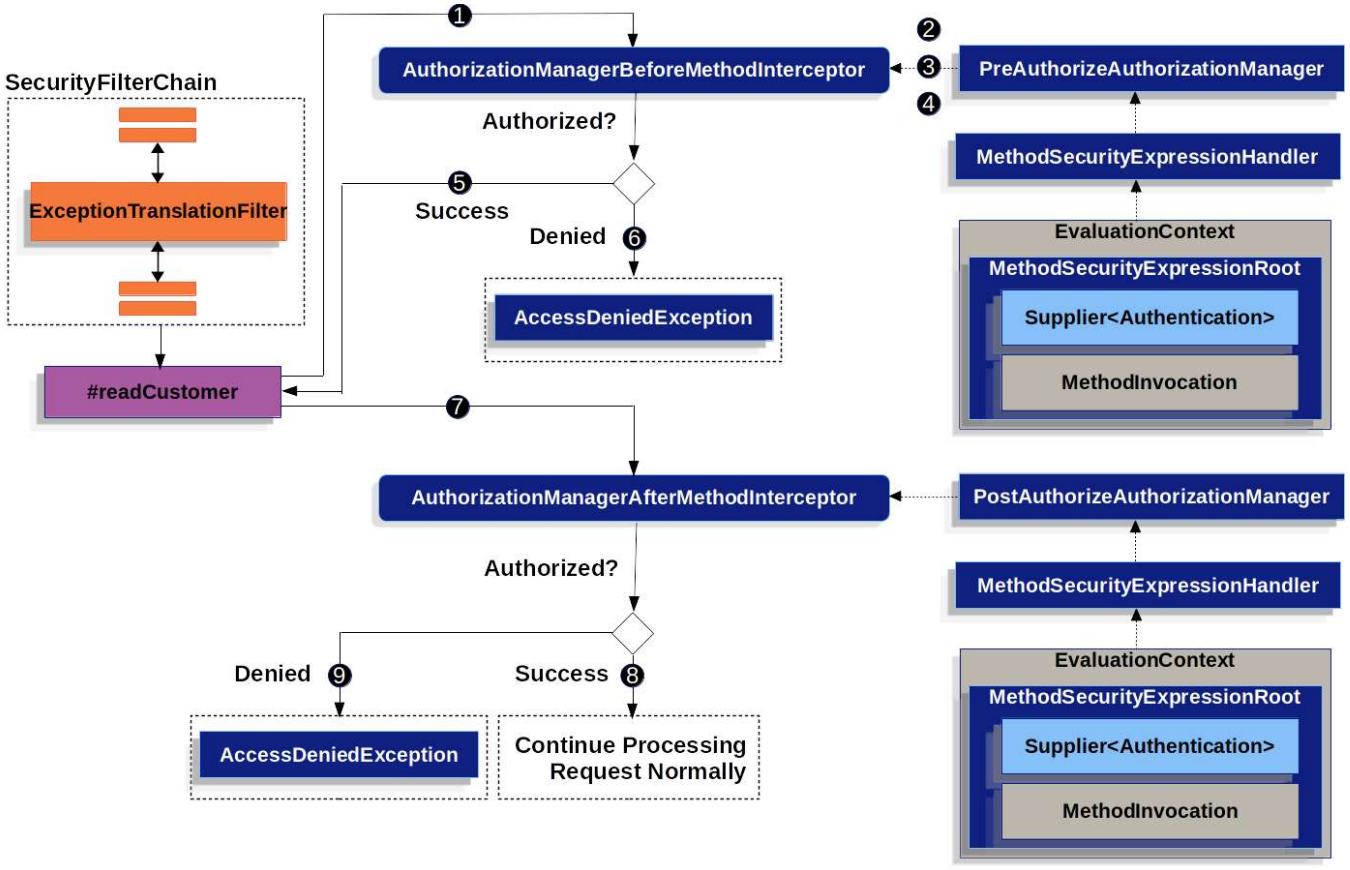
How Method Security Works

Spring Security's method authorization support is handy for:

- Extracting **fine-grained authorization logic**; for example, when the method parameters and return values contribute to the authorization decision.
- Enforcing security at the **service layer**
- **Stylistically** favoring annotation-based over `HttpSecurity`-based configuration

> **Tip**
>
> Always enforce security as close to the data / in the lowest layer, if possible. e.g. on the `Repository` classes.

```
@Service
public class MyCustomerService {
    @PreAuthorize("hasAuthority('permission:read')")
    @PreAuthorize("hasAuthority('permission:bla')") // Not supported, use "and" in SpEL
    // ANDed with
    @PostAuthorize("returnObject.owner == authentication.name")
    public Customer readCustomer(String id) { ... }
}
```



Favor Granting Authorities Over Complicated SpEL Expressions

- Quite often it can be tempting to introduce a complicated SpEL expression like the following:

```
@PreAuthorize("hasAuthority('permission:read') || hasRole('ADMIN')")
```

- However, you could instead grant `permission:read` to those with `ROLE_ADMIN`. One way to do this is with a `RoleHierarchy` like so:

```
@Bean
static RoleHierarchy roleHierarchy() {
    return new RoleHierarchyImpl("ROLE_ADMIN > permission:read");
}

@PreAuthorize("hasAuthority('permission:read')")
```

- Or, preferably centralize all authorization logic in a bean with this responsibility

```
@Component
public class Authorizer {

    public boolean canReadCourses(MethodSecurityExpressionOperations ops) {
        return ops.hasAuthority("course:read");
    }

    public boolean canCreateCourses(MethodSecurityExpressionOperations ops) {
        return ops.hasAuthority("course:write");
    }

    public boolean canDeleteCourses(MethodSecurityExpressionOperations ops) {
        return ops.hasAuthority("course:delete");
    }
}

@PreAuthorize("@authorizer.canCreateCourses(#root)")
```

Comparing Request-level vs Method-level Authorization

	request-level	method-level
authorization type	coarse-grained	fine-grained
configuration location	declared in a config class	local to method declaration
configuration style	DSL	Annotations
authorization definitions	programmatic	SpEL

> **Danger**
>
> It's important to remember that when you use annotation-based Method Security, then unannotated methods are not secured. To protect against this, declare a catch-all authorization rulein your `HttpSecurity` instance.

Authorizing with Annotations

```
@Component
public class BankService {
        @PreAuthorize("hasRole('ADMIN')")
```

```
        public Account readAccount(Long id) {
        // ... is only invoked if the `Authentication` has the `ROLE_ADMIN` authority
        }
}

@Component
public class BankService {
        @PostAuthorize("returnObject.owner == authentication.name")
        public Account readAccount(Long id) {
        // ... is only returned if the `Account` belongs to the logged in user
        }
}
```

```
@Component
public class BankService {
        @PreFilter("filterObject.owner == authentication.name")
        public Collection<Account> updateAccounts(Account... accounts) {
        // ... `accounts` will only contain the accounts owned by the logged-in user
        return updated;
        }
}

@Component
public class BankService {
        @PostFilter("filterObject.owner == authentication.name")
        public Collection<Account> readAccounts(String... ids) {
        // ... the return value will be filtered to only contain the accounts owned by the
logged-in user
        return accounts;
        }
}
```

Danger

You should not pass accounts that aren't owned by the user in the first place, this kind of filtering should happen at the data layer!

In-memory filtering can obviously be expensive, and so be considerate of whether it is better to filter the data in the data layer instead.

Specifiying Order

- As already noted, there is a Spring AOP method interceptor for each annotation, and each of these has a location in the Spring AOP advisor chain.
- Namely, the `@PreFilter` method interceptor's order is 100, `@PreAuthorize`'s is 200, and so on.
- The reason this is important to note is that there are other AOP-based annotations like `@EnableTransactionManagement` that have an order of `Integer.MAX_VALUE`. In other words, they are located at the end of the advisor chain by default.
- At times, it can be valuable to have other advice execute before Spring Security. For example, if you have a method annotated with `@Transactional` and `@PostAuthorize`, you might want the transaction to still

be open when `@PostAuthorize` runs so that an `AccessDeniedException` will cause a rollback.

- To get `@EnableTransactionManagement` to open a transaction before method authorization advice runs, you can set `@EnableTransactionManagement`'s order like so:

```
@EnableTransactionManagement(order = 0)
```

SpEL expressions

- `#` refers to a variable inside the SpEL evaluation context, this includes method params and return objects in case of `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, and `@PostFilter`
- `@` refers to any bean in your application context
- e.g. `@myBean.authorize(#param)`
- Spring Security encapsulates all of its authorization fields and methods in a set of root objects. The most generic root object is called `SecurityExpressionRoot` and it forms the basis for `MethodSecurityExpressionRoot`. Spring Security supplies this root object to `MethodSecurityEvaluationContext` when preparing to evaluate an authorization expression.
- The first thing this provides is an enhanced set of authorization fields and methods to your SpEL expressions. What follows is a quick overview of the most common methods:
- `permitAll` - The method requires no authorization to be invoked; note that in this case, the `Authentication` is never retrieved from the session
- `denyAll` - The method is not allowed under any circumstances; note that in this case, the `Authentication` is never retrieved from the session
- `hasAuthority` - The method requires that the `Authentication` have a `GrantedAuthority` that matches the given value
- `hasRole` - A shortcut for `hasAuthority` that prefixes `ROLE_` or whatever is configured as the default prefix
- `hasAnyAuthority` - The method requires that the `Authentication` have a `GrantedAuthority` that matches any of the given values
- `hasAnyRole` - A shortcut for `hasAnyAuthority` that prefixes `ROLE_` or whatever is configured as the default prefix
- `hasPermission` - A hook into your `PermissionEvaluator` instance for doing object-level authorization
- And here is a brief look at the most common fields:
- `authentication` - The `Authentication` instance associated with this method invocation
- `principal` - The `Authentication#getPrincipal` associated with this method invocation

Using a Custom Bean in SpEL

- First, declare a bean that has a method that takes a `MethodSecurityExpressionOperations` instance like the following:

```
@Component("authz")
public class AuthorizationLogic {
```

```
    public boolean decide(MethodSecurityExpressionOperations operations) {
        // ... authorization logic
    }
}
```

- Then, reference that bean in your annotations in the following way:

```
@Controller
public class MyController {
    @PreAuthorize("@authz.decide(#root)")
    @GetMapping("/endpoint")
    public String endpoint() {
        // ...
    }
}
```

- Spring Security will invoke the given method on that bean for each method invocation.
- What's nice about this is all your authorization logic is in a separate class that can be independently unit tested and verified for correctness. It also has access to the full Java language.

Authorization Events

- 🔗 Authorization Events
- For each authorization that is denied, an `AuthorizationDeniedEvent` is fired. Also, it's possible to fire an `AuthorizationGrantedEvent` for authorizations that are granted.
- To listen for these events, you must first publish an `AuthorizationEventPublisher`.
- Because `AuthorizationGrantedEvent`s have the potential to be quite noisy, they are not published by default.
- In fact, publishing these events will likely require some business logic on your part to ensure that your application is not inundated with noisy authorization events.

```
@Component
public class MyAuthorizationEventPublisher implements AuthorizationEventPublisher {
    private final ApplicationEventPublisher publisher;
    private final AuthorizationEventPublisher delegate;

    public MyAuthorizationEventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
        this.delegate = new SpringAuthorizationEventPublisher(publisher);
    }

    @Override
    public <T> void publishAuthorizationEvent(Supplier<Authentication> authentication,
            T object, AuthorizationDecision decision) {
        if (decision == null) {
            return;
        }
        if (!decision.isGranted()) {
        // ...
```

```
        }
}
```

```
@Component
public class AuthenticationEvents {

    @EventListener
    public void onFailure(AuthorizationDeniedEvent failure) {
                // ...
    }
}
```

Other features and integrations

Java Configuration

- 🔗 Custom DSLs
- 👆 Post Processing Configured Objects

Spring Security Crypto module

- 🔗 Spring Security Crypto module
  - Symmetric encryption
  - Key generation
  - Password encoding

Spring Security concurrency APIs

- 👆 Spring Security concurrency APIs
- `DelegatingSecurityContextRunnable`

Jackson Support

- 🔗 Jackson Support Spring Security

```
ClassLoader loader = getClass().getClassLoader();
List<Module> modules = SecurityJackson2Modules.getModules(loader);
mapper.registerModules(modules);

// ... use ObjectMapper as normally ...
SecurityContext context = new SecurityContextImpl();
// ...
String json = mapper.writeValueAsString(context);
```

Spring Security 5 and Older

- `WebSecurityConfigurerAdapter` in Spring Security 5 and older
  🔗 Spring Security without the WebSecurityConfigurerAdapter
- [🔗Legacy Authorization Components](https://docs.spring.io/spring-security/reference/servlet/authorization/architecture.html#authz-legacy-note)

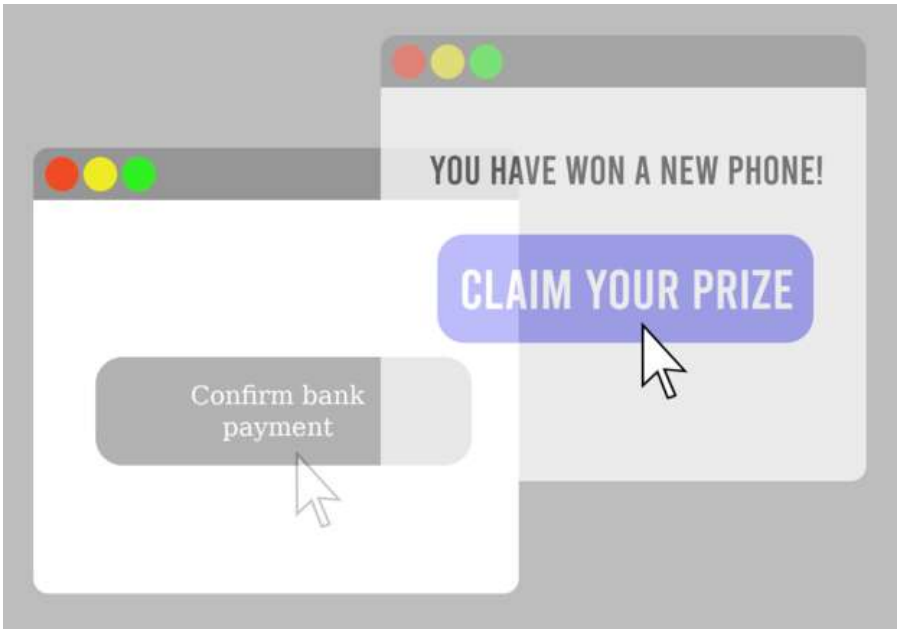
Common attacks

🏴‍☠️🏴‍☠️🏴‍☠️

OWASP Top 10

- [ ] 🔗 guide - hacksplaining.com/owasp

Clickjacking



- `X-Frame-Options` header prevents click jacking of your website
- Clickjacking | OWASP Foundation

Cross-Site Request Forgery (CSRF)

- 🔗 What is CSRF?

```
// your site
<form method="post"
        action="/transfer">
<input type="text"
        name="amount"/>
<input type="text"
        name="routingNumber"/>
<input type="text"
        name="account"/>
<input type="submit"
        value="Transfer"/>
</form>

// Transfer HTTP request
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

```
// Now pretend you authenticate to your bank's website and then, without logging out,
visit an evil website. The evil website contains an HTML page with the following form:

<form method="post"
        action="https://bank.example.com/transfer">
<input type="hidden"
        name="amount"
        value="100.00"/>
<input type="hidden"
        name="routingNumber"
        value="evilsRoutingNumber"/>
<input type="hidden"
        name="account"
        value="evilsAccountNumber"/>
<input type="submit"
        value="Win Money!"/>
</form>

// You like to win money, so you click on the submit button. In the process, you have
unintentionally transferred $100 to a malicious user. This happens because, while the evil
website cannot see your cookies, the cookies associated with your bank are still sent
along with the request.
```

- Happens only if you use cookies as an authentcation mechanism: relevant in SSR apps and Backend For Frontend (BFF) pattern.
- Methods like `GET`, `HEAD`, `OPTIONS`, and `TRACE` should be READ ONLY.

`SameSite` property

- Origin = `http://blog.example.com:80` = everything is considered, exact match
- Site = TLD+1 e.g. the site of `blog.example.org` is `example.org`, so a request from `docs.example.org` is considered same-site, but cross-origin
- `Lax` send cookie from cross-site read-only requests
- `Strict` send cookie from only same-site requests
- Spring Security does not create the HTTP session so you must customize session cookie behavior in SSR applications

Protecting against CSRF

SSR apps

- `SameSite` cookies
  - Doesn't protect against Cross-Origin Request Forgery (CORF)
- Synchronizer token pattern (must be used for SSR apps)
  - Session-based synchronizer token

```
<form method="post"
        action="/transfer">

// CSRF token
<input type="hidden"
        name="_csrf"
        value="4bfd1575-3ad1-4d21-96c7-4ef2d9f86721"/>

<input type="text"
```

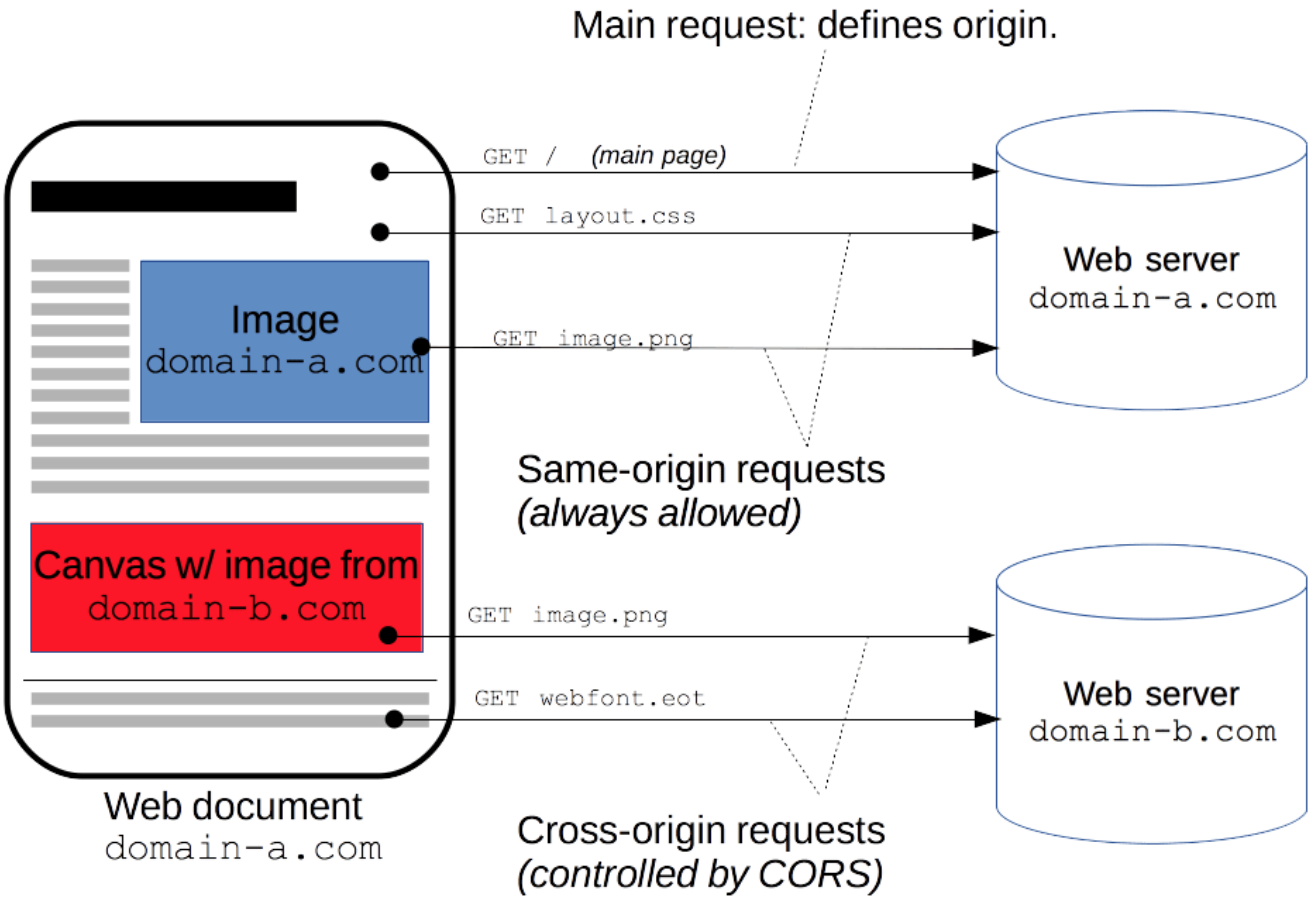
```
        name="amount"/>
<input type="text"
        name="routingNumber"/>
<input type="hidden"
        name="account"/>
<input type="submit"
        value="Transfer"/>
</form>
```

SPA + REST API apps

- `SameSite` cookies
- `CORS` configuration
  - Protects against CORF
- Cookie-based synchronizer token for SPA apps
  - This is irrelevant if you use SameSite cookies and CORS and allow only your SPA's specific origin to prevent CORF

CORS

`cors.html`
`CorsConfig`



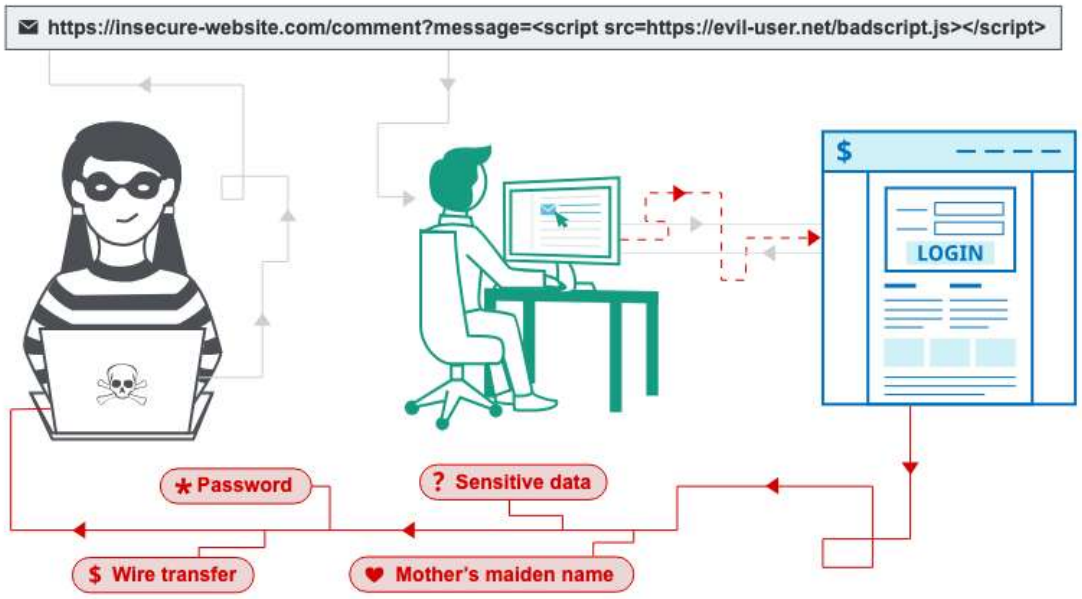
- CORS on MDN
- Simple vs Preflighted requests

- Spring Security CORS support (if you don't use Spring MVC)
- Spring MVC CORS support (preferred)

Cross-Site Scripting (XSS)

xss.html

- Happens due to treating data as code.
- Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users.
- When the malicious code executes inside a victim's browser, the attacker can do any action that user can do, and steal their data, etc.



- What is cross-site scripting (XSS) and how to prevent it?

Reflected XSS

- The simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

```
GET https://insecure-website.com/status?message=All+is+well.
<p>Status: All is well.</p>

GET https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>
<p>Status: <script>/* Bad stuff here... */</script></p>
```

Stored XSS

- Arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Protecting against XSS

- Sanitize input (must do if you want to dynamically render user-generated HTML), use a library like DOMPurify to clear it of malicious patterns

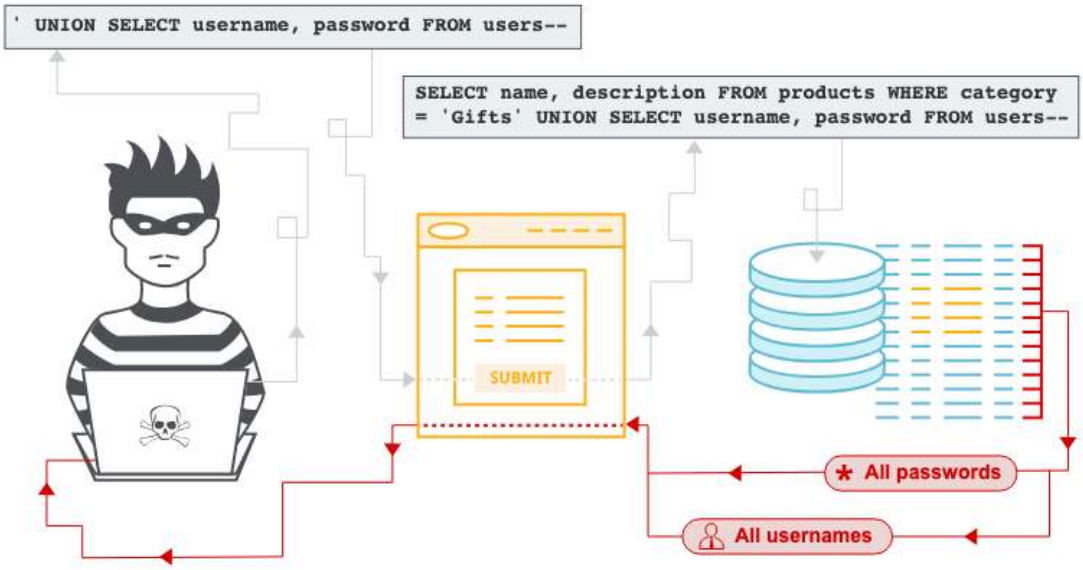
- Encode output (preferred)
- Content Security Policy (CSP) - HTTP | MDN - specifies

```
// Front end frameworks e.g. React, Angular, Vue automatically escape user-generated
content so it is rendered as a harmless string to the user
// React.js
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

SQL injection

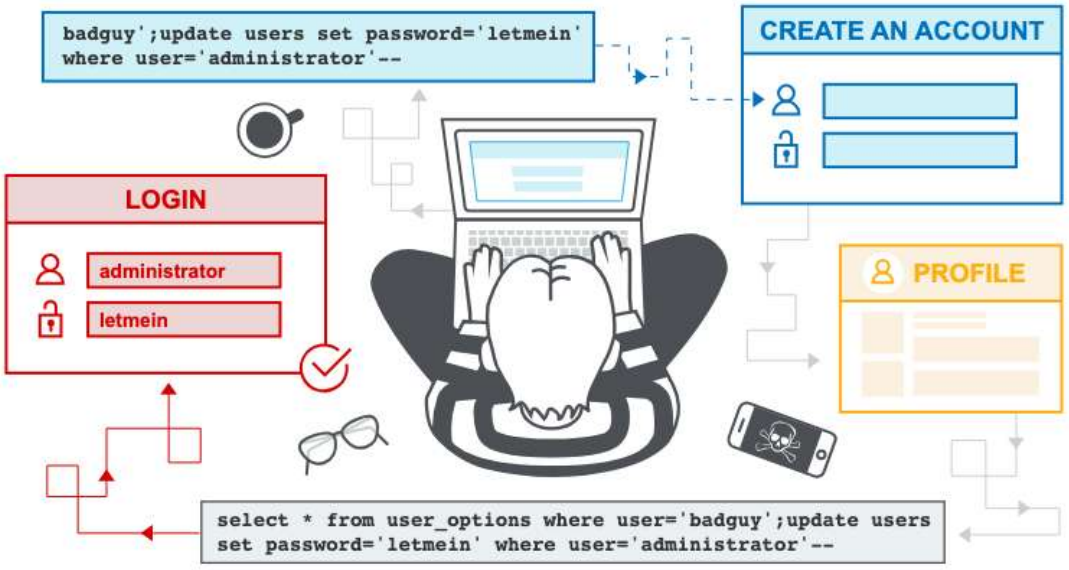
Attack

- Happens due to treating data as code.



- SQL Injection

Stored SQL Injection



Protecting against SQL Injection

- Use prepared statements.
- A prepared statement is first parsed by the database and its parameters `?` are only treated as data when they are supplied, never as code.
- Never create SQL queries by concatenating SQL code and parameters together.

```
// The following code is vulnerable to SQL injection because the user input is
concatenated directly into the query:

String query = "SELECT * FROM products WHERE category = '"+ input + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);

// You can rewrite this code in a way that prevents the user input from interfering with
the query structure:
PreparedStatement statement = connection.prepareStatement("SELECT * FROM products WHERE
category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

Lab

🔬🔬🔬

- Re-implement API Key authentication using `org.springframework.security.web.authentication.AuthenticaitonFilter`.
- This is a generic authenticaiton filter created by the Spring Security team, parameterized with strategies for correct handling of authentication.

JWT

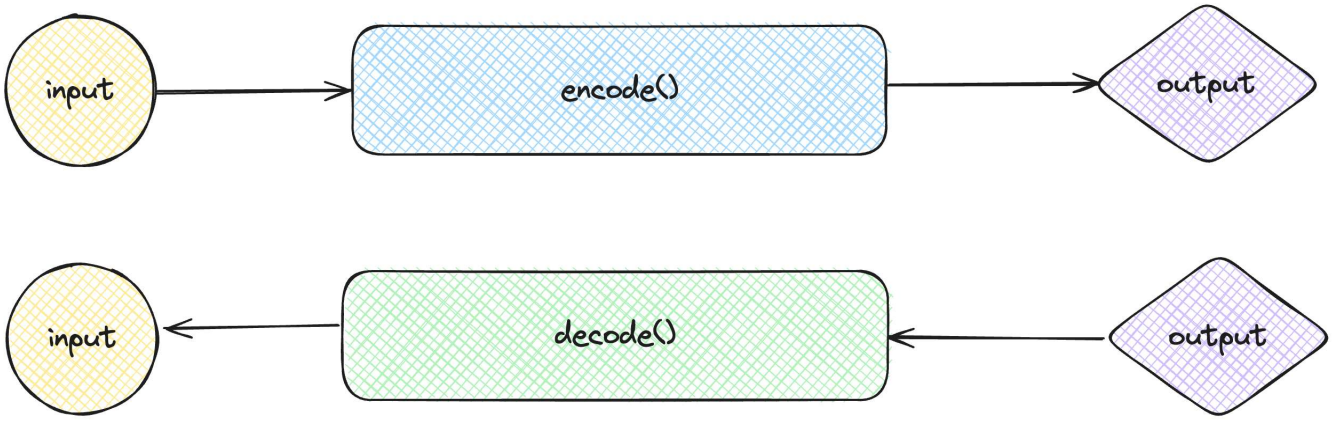
Cryptography

A mathematical method to guarantee:

- **Confidentiality**: only the intended receiver can read the message
- **Authenticity**: we are sure of who sent the message
- **Integrity**: the message was not modified along the way
- **Non-repudiation**: sender cannot deny having sent the message

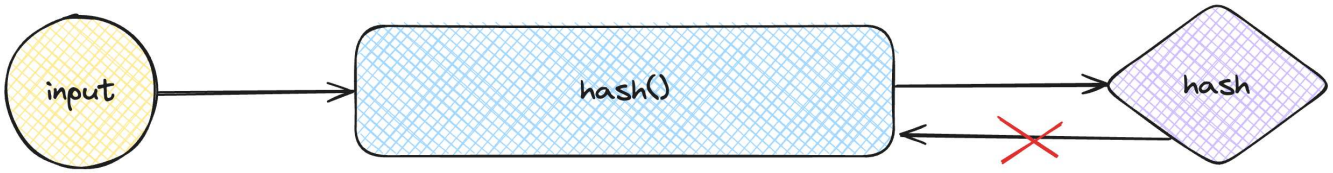
Encoding

- A two-way function that takes input in one format e.g. binary and outputs another format e.g. text.
- Example is Base64 encoding of binary to text.



Hashing

- One-way function
- Fixed length output
- Same input → same output

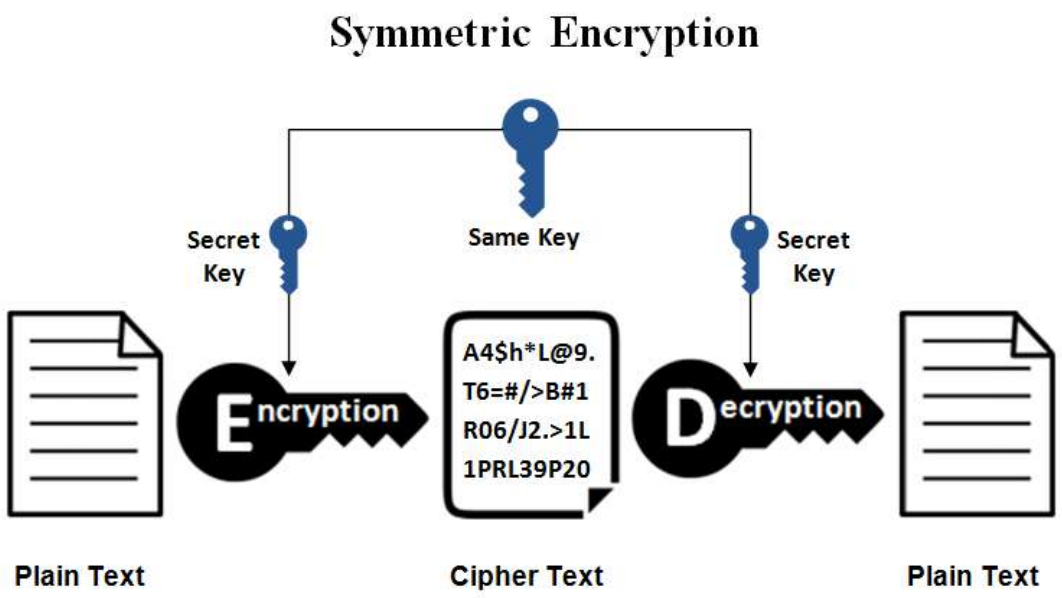


Encryption

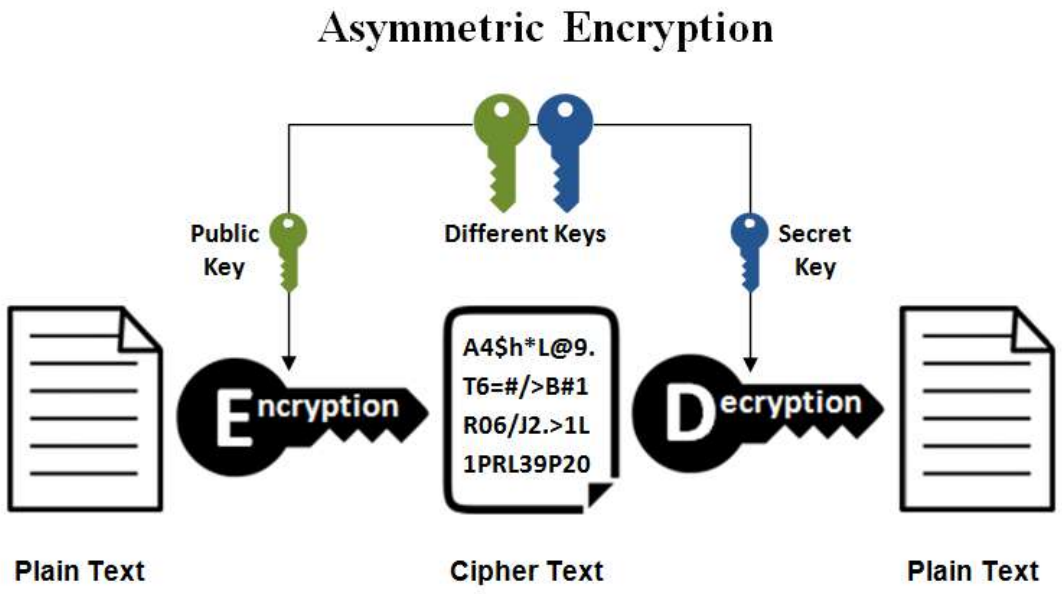
- Guarantees **Confidentiality**

Symmetric encryption

- Encrypt with key
- Decrypt with same key
- e.g. Advanced Encryption Standard (AES)



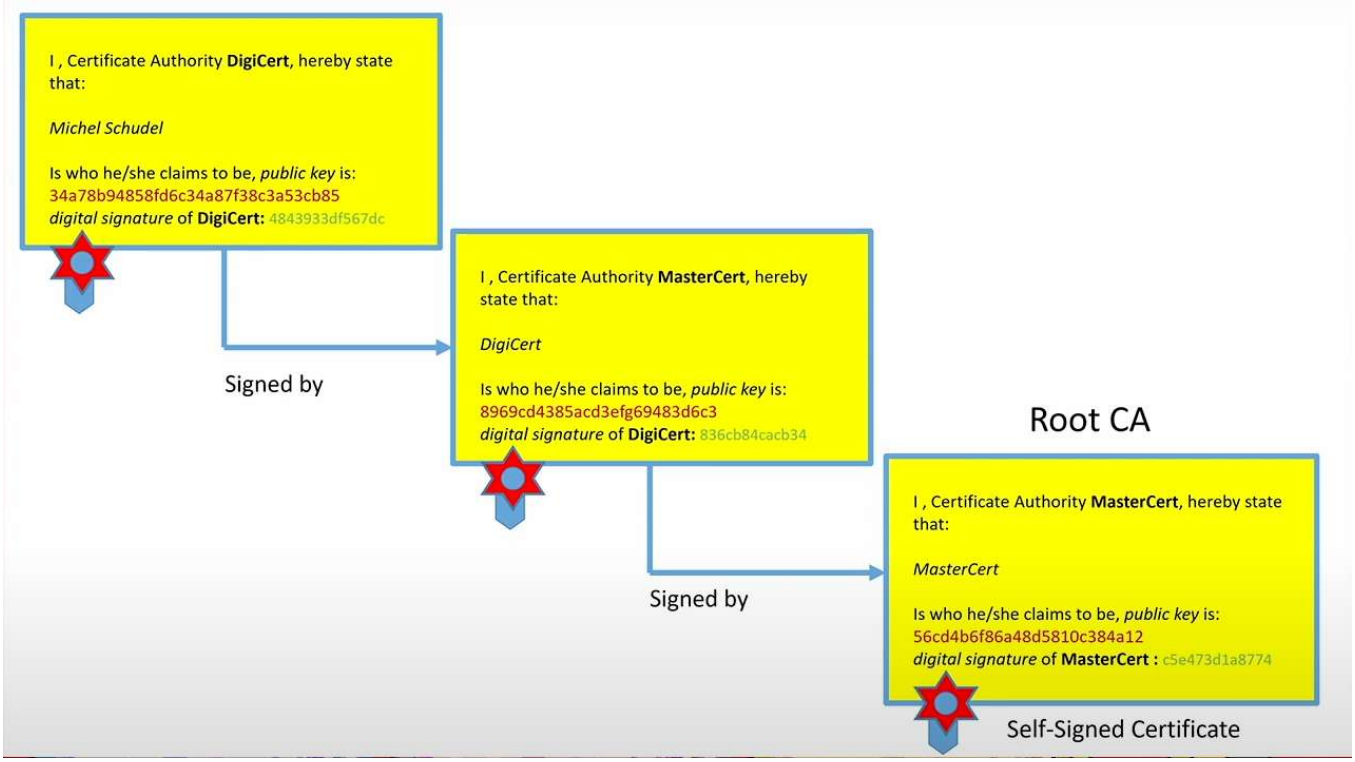
Asymmetric encryption



- Encrypt with either key
- Decrypt with the other key
- e.g. RSA (Rivest–Shamir–Adleman)

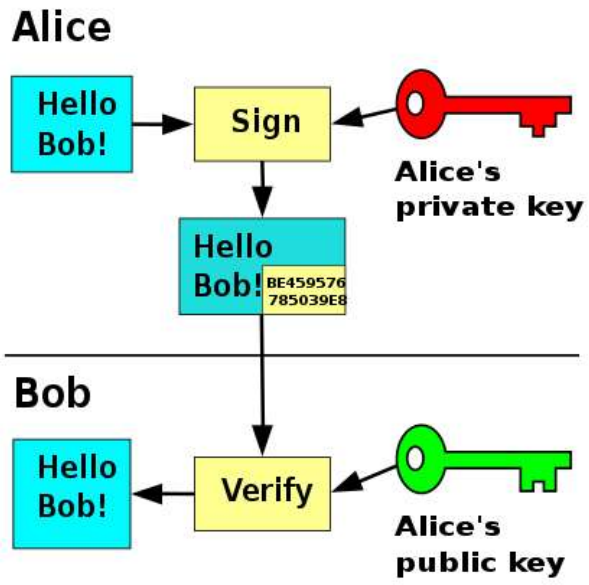
Public Key Certificate

- Also known as a **digital certificate** or **identity certificate**
- An electronic document used to prove the validity of a public key.
- The certificate includes the public key and information about it, information about the identity of its owner (called the subject), and the digital signature of an entity that has verified the certificate's contents (called the issuer).
- The digital signature is the hash of the body of the certificate, encrypted using the issuer's private key.



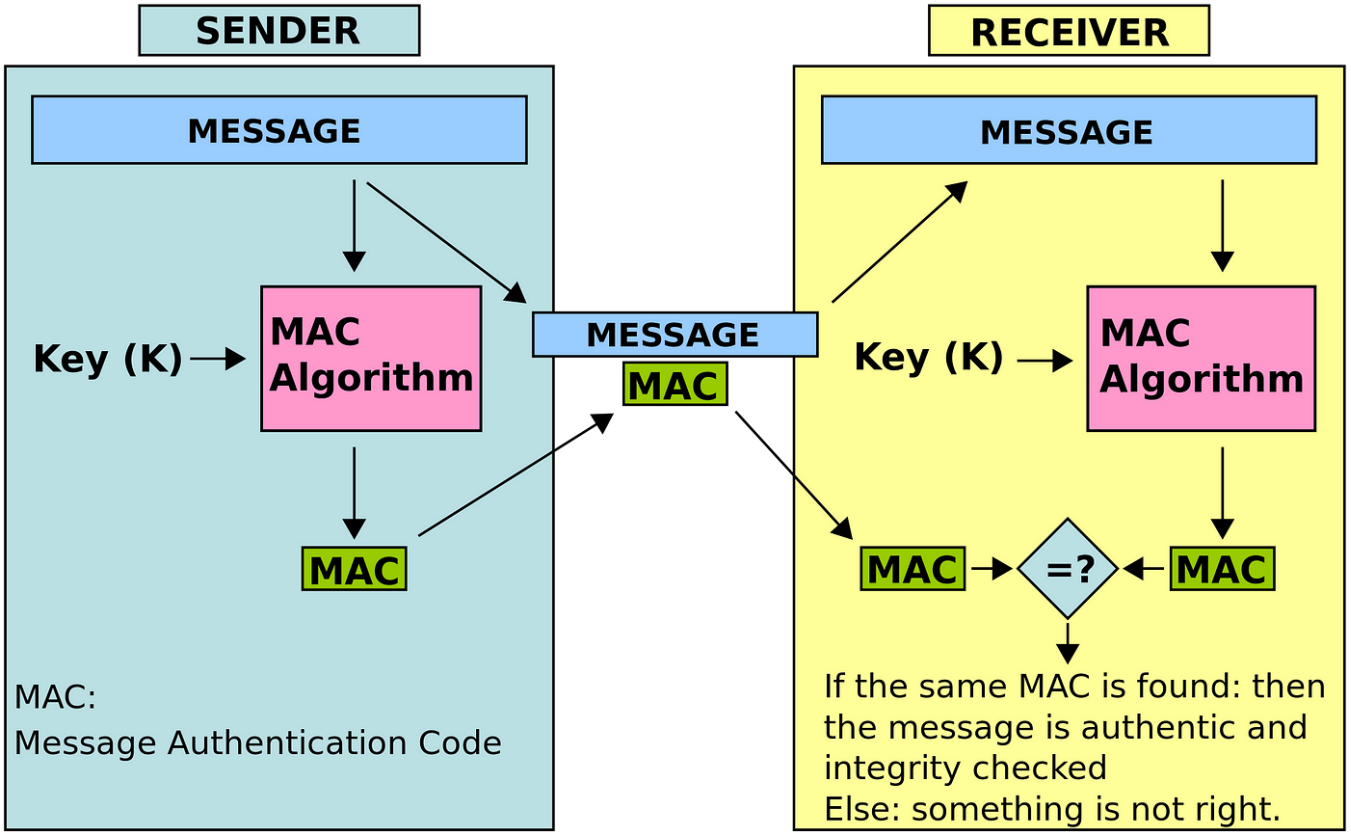
Digital Signature

- Guarantees Integrity, Authenticity, Non-repudiation
- Uses assymetric encryption



Message Authentication Code (MAC)

- Guarantees: Integrity, Authenticity
- Uses symmetric encryption



JOSE (JSON Object Signing and Encryption) Framework

- It is a collection of standards.
- **JWT**: JSON Web Token, a compact and standardized way of securely transmitting information between parties using JSON data structures.
- **JWS**: JSON Web Signature, a specification for digitally signing JSON data.
- **JWE**: JSON Web Encryption, a specification for encrypting JSON data.
- **JWK**: JSON Web Key, a specification for representing cryptographic keys in JSON format.

```
{
  "kty": "RSA",
  "kid": "rsa-key-1",
  "use": "sig",
  "n":
"0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJ
ECPebWKRXjBZCiFVHUvkiDw9iFOjK2kcp--BIKOntVrF9s_mXbpu",
  "e": "AQAB",
  "d": "X4cTteJYGVBA_ECx6FzSX6Uvtv6QbT6KjczuLZSInQgY1YypA24tcI4i4y-l1XXMvLgHF-
pG8n2lnDg6f0GuxH-QI6A8PhgG4YoX9-
mJGKlDOZzK8sZJfLztWK3ikvMskKjV8RyGx6X9RB2PRhvTfTjKFrRwBYRSFicWSJ2EEY"
}
```

- **JWA**: JSON Web Algorithms, a specification for defining cryptographic algorithms used in JWE and JWS.

> **Important**
>
> What is most often used for authentication is a JWT signed with JWS using a JWA. The signature can be verified with a public key distributed using JWE.

Structure

- 👆 JSON Web Tokens - jwt.io
- They're either *digitally signed* (99% of cases) or have an *HMAC* signature

OAuth 2.0 and OIDC

🆔🆔🆔

OAuth 2.0 Authorization Framework

- A standard for delegated access control.
- e.g. I have an application and I want to access your Google Drive on your behalf to save the output of my application to your Google Drive.
- How do we do this safely without you have to give me your Google email and password?

Open ID Connect (OIDC)

- A standard for user authentication and identification.
- It is built on top of the OAuth 2.0 standard. The changes are very minimal.
- e.g. I have an application and I want you to login with your Google account, instead of creating an account with me.

> **Tip**
>
> If I want to do both authenticate you and access APIs on your behalf, then I will be using both OAuth 2.0 and OIDC.

Actors

Actor	OAuth 2.0	OIDC
User	Resource Owner (RO)	End-User
API	Resource Server (RS)	-
Authorization Server	Authorization Server (AS)	OpenID Provider
Client	Client	Relying Party

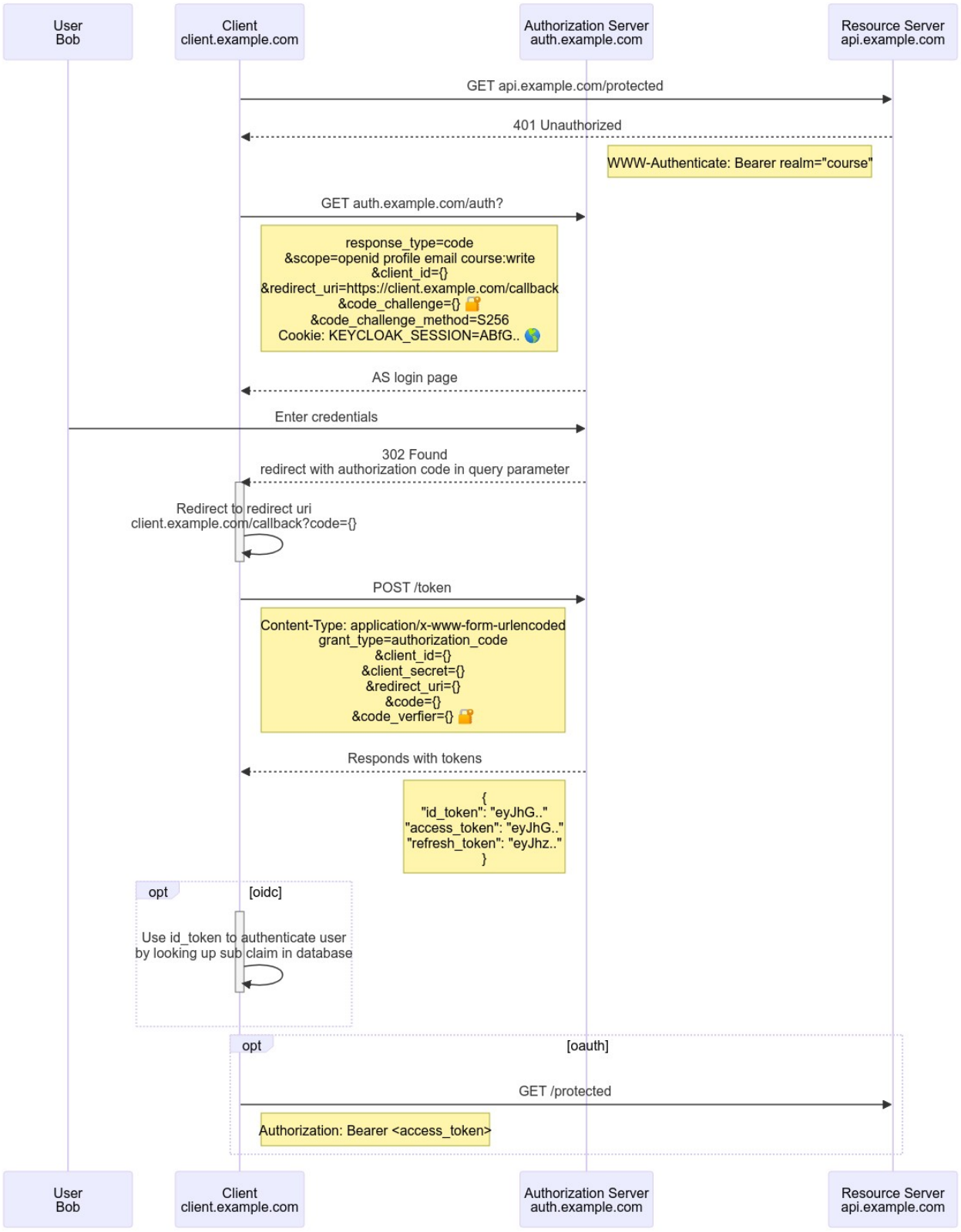
Why use these standards?

- Authentication and Authorization are supporting domains, not core domains. Do not waste developer effort in re-inventing the wheel. Just use Keycloak or a similar provider, which implements both of these standards.

OAuth 2.0 Grants / OIDC Flows

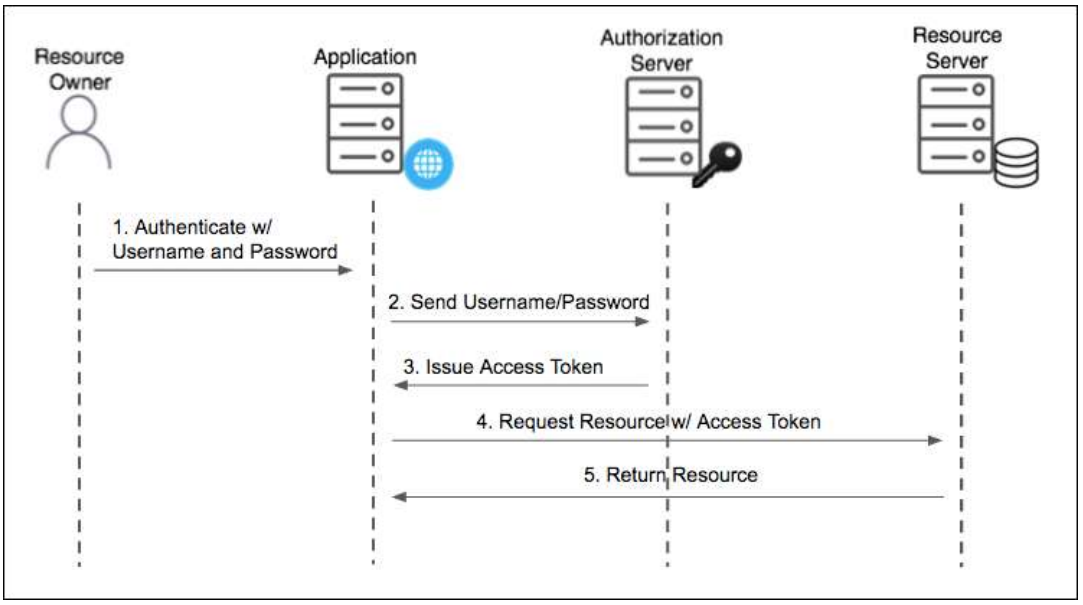
- They are both the exact same thing
- They're the way the different actors communicate with each other in order to arrive at the endgoal: authentication or delegated access control (authorization).

Authorization Code Flow



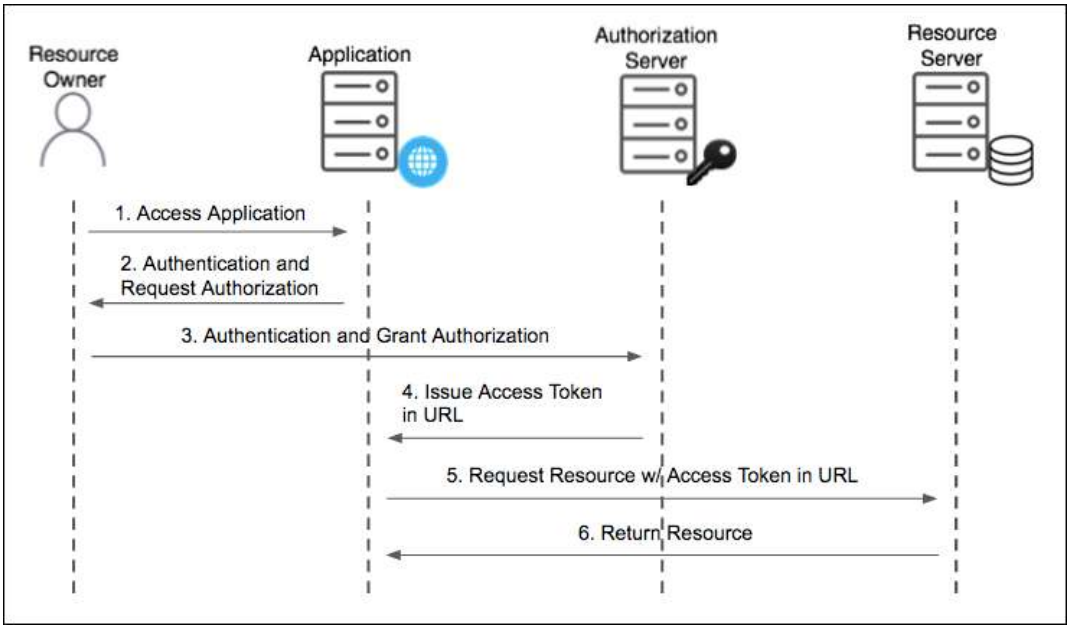
- 🔐 Proof-Key for Code Exchange (PKCE)
- 🌎 Single Sign-On (SSO)

Resource Owner Password Flow



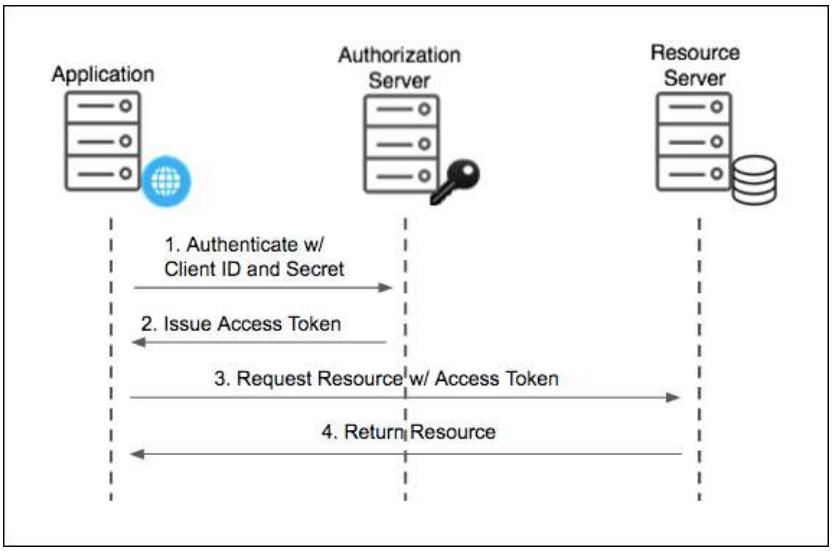
- Deprecated as it is insecure.
- Applications other than the Authorization Server handle the user's credentials.

Implicit Flow



- Deprecated as it is insecure.
- Token is passed in plain text in redirect URL, can be logged by servers, many attack vectors.

Client Credentials Flow



- Only in OAuth 2.0, why?
- For servers (confidential clients) talking to other servers.
- Clients must be registered with the Authorization Server.
- Clients can either be confidential (have a password) or public (can't have a password as there is no safe place to store it i.e. SPAs)

Tokens

ID Token

- Used for authenticating the user on the Client.
- e.g. Login with Google
- e.g. obtaining information about the user in your enterprise app frontend and backend.

```
// Header
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "g_qzRLodrg7B1Z_m3WmD5ZFJGiw38OAU3BIqjlNIGGI" // JWK ID
}
// Payload
{
  "exp": 1714779317, // When the token expires
  "iat": 1714779017, // When the token was issued at
  "auth_time": 1714779001, // When did the user authenticate
  "jti": "292528a3-40f4-4847-90a3-d2884f41bfbd", // Token unique id
  "iss": "http://localhost:8088/realms/course-practice", // Issuer URI
  "aud": "oidc-playground", // Audience, who should accept this token, must contain the
Relying party that this token was issued to
  "sub": "195ac444-6165-4cfe-bebd-ec03b3d92b1c", // The unique identity of the user
  "typ": "ID", // Type of the token
  "azp": "oidc-playground", // The relying party this token was issued to
  "session_state": "ac28705f-f6c5-4071-9598-d71b8c74e585",
  "at_hash": "T5UfiFU4T0Rrq5YGxdDYpw",
  "acr": "0",
  "sid": "ac28705f-f6c5-4071-9598-d71b8c74e585",
  "email_verified": true,
  "name": "Bob Mortimer",
  "preferred_username": "bob@example.com",
```

```
  "given_name": "Bob",
  "family_name": "Mortimer",
  "email": "bob@example.com"
}
```

- The claims returned in the ID Token depend on the scope requested in the OIDC authentication request.
- The claims of an ID token can also be obtained from an access token by `POST`ing to the Authorization Server's OIDC `userinfo` endpoint.

Scope	Claims
openid	(required) Returns the **sub** claim, which uniquely identifies the user. In an ID Token, **iss**, **aud**, **exp**, **iat**, and **at_hash** claims will also be present. To learn more about the ID Token claims, read ID Token Structure.
profile	Returns claims that represent basic profile information, including **name**, **family_name**, **given_name**, **middle_name**, **nickname**, **picture**, and **updated_at**.
email	Returns the **email** claim, which contains the user's email address, and **email_verified**, which is a boolean indicating whether the email address was verified by the user.

- OpenID Connect Scopes

Access Token

Self-contained tokens i.e. JWT

- 👆 Example Access Token

```
// Header
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "g_qzRLodrg7B1Z_m3WmD5ZFJGiw38OAU3BIqjlNIGGI"
}
// Payload
{
  "exp": 1714779317,
  "iat": 1714779017,
  "auth_time": 1714779001,
  "jti": "f6f64105-9961-4e9f-907f-86e5ce115fda",
  "iss": "http://localhost:8088/realms/course-practice",
  "aud": "account",
  "sub": "195ac444-6165-4cfe-bebd-ec03b3d92b1c",
  "typ": "Bearer",
  "azp": "oidc-playground",
  "session_state": "ac28705f-f6c5-4071-9598-d71b8c74e585",
  "acr": "0",
  "allowed-origins": [
        // Can be used for preflight requests in Resource Servers
    "http://localhost:8000"
  ],
  // This contains a list of global realm roles. It is the intersection between the roles granted to the user, and the roles the client has access to.
  "realm_access": {
    "roles": [
      "offline_access",
```

```
      "superadmin",
      "uma_authorization",
      "default-roles-course-practice"
    ]
  },
  // This contains a list of client roles.
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  // Scopes can be used both to decide what fields (or claims) to include in the token and
by backends to decide what APIs the token can access.
  "scope": "openid email profile",
  "sid": "ac28705f-f6c5-4071-9598-d71b8c74e585",
  "email_verified": true,
  "name": "Bob Mortimer",
  "preferred_username": "bob@example.com",
  "given_name": "Bob",
  "family_name": "Mortimer",
  "email": "bob@example.com"
}
```

Opaque / Reference tokens

- They are just an identifier that the resource server must call the token introspection endpoint with to receive the token details.
- Have better revocation properties at the cost of performance.
- The improved security properties of opaque access tokens are only useful if you have an automatic fraud and anomaly detection process, otherwise it's pointless if it will take longer to revoke the token than its actual expiration time (e.g. 5 minutes).

Refresh Token

- 👆 Example Refresh Token

Why not just a long-lived access token?

- Access token revocation would be a pain. The Authorization Server would need to keep track of every token it has issued and its status.
- Reduces damage from access token getting leaked.
- Allow for re-authenticating the user periodically without having to send credentials again using refresh tokens.
- Refresh tokens have better security properties through refresh token rotation.
- Refresh tokens allow updates to a user's information without having to logout and login again.

Danger

> If an attacker gains access to your refresh tokens, you're screwed anyway! 💣

Where to store refresh tokens?

An attacker gaining a refresh token gives them long term access to a user's account. They must be stored as securely as possible

Confidential Client

- Preferably stored *encrypted*
- `client_secret` is required to be able to use the token

Public Client

- Stored in-memory for SPAs e.g. KeycloakJS
- Preferably use BFF pattern
- For mobile clients stored in storage secured by OS protected by user's fingerprint or other credentials

How long should refresh tokens live?

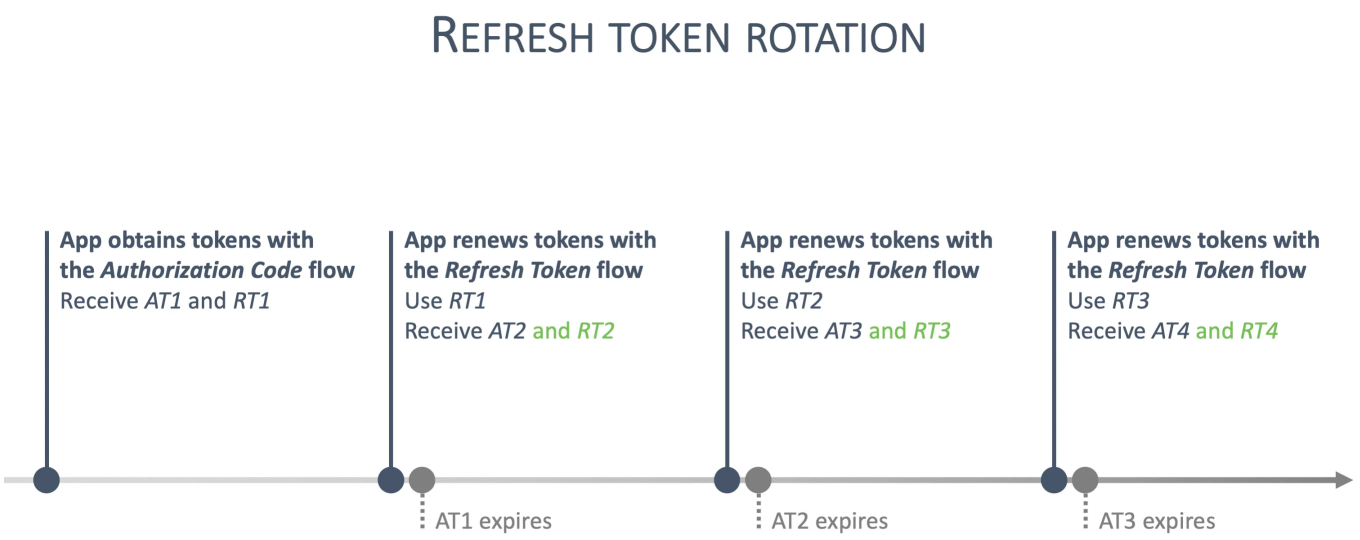
- LinkedIn: Months.
- Enterprise : 12 hours.

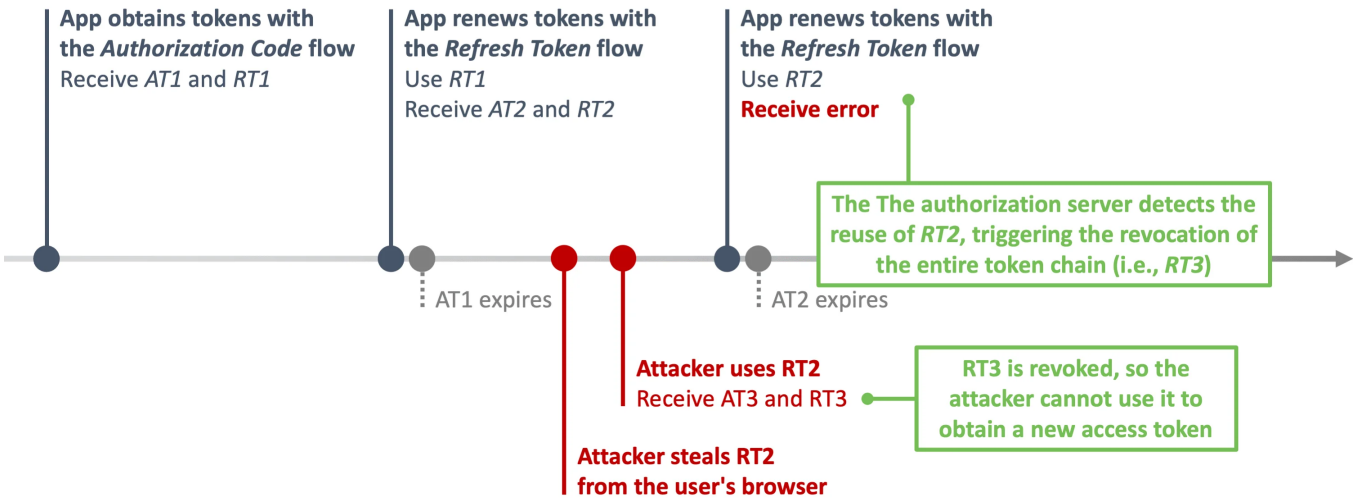
Whatever you think is a good time.

- Clients can revoke refresh tokens when they no longer need them ("Disconnect FB from X App")
- Users can revoke refresh tokens, obviously, through "Connected Apps" for example in FB.

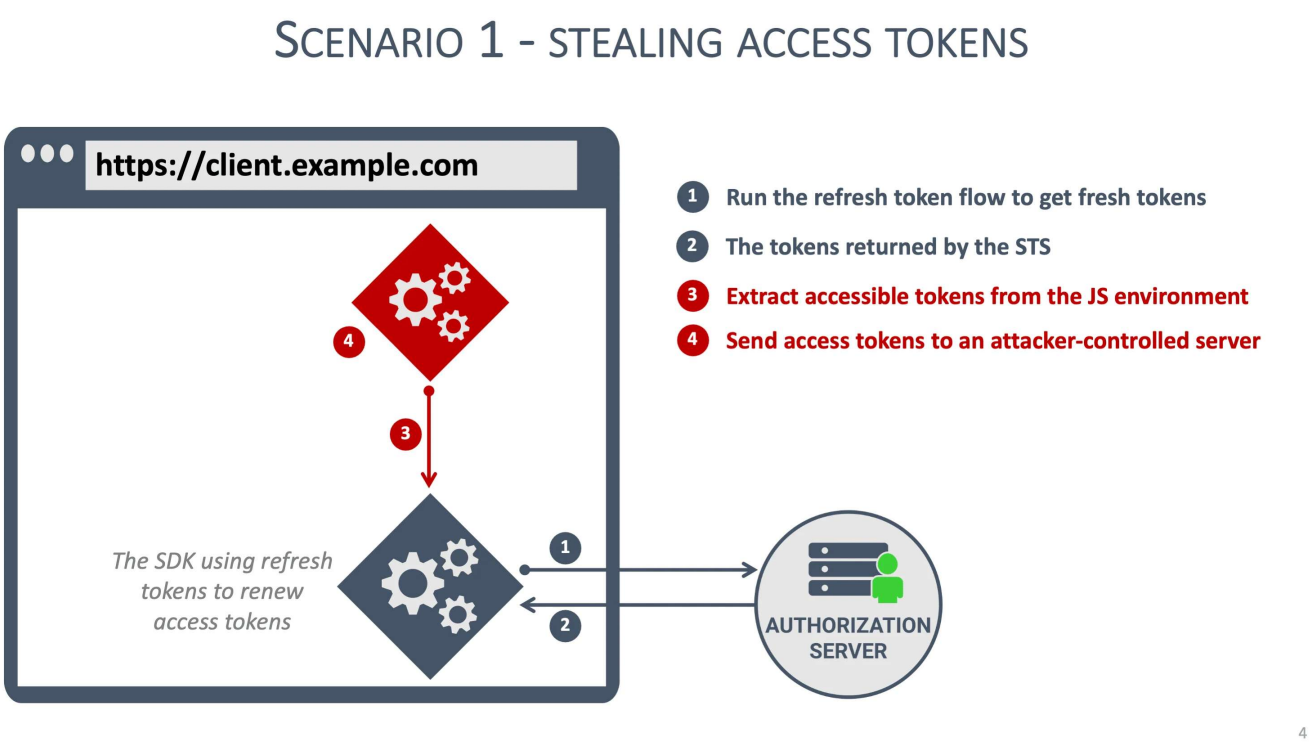
Refresh Token Rotation

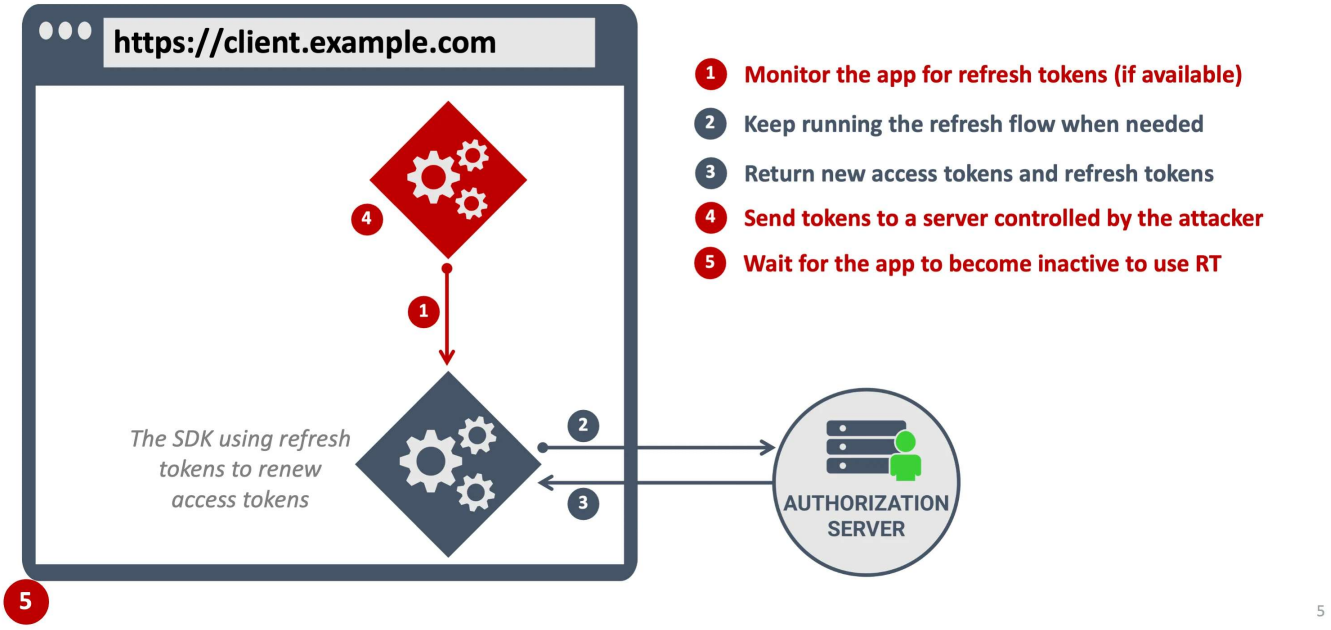
- Whenever you receive new tokens you must use the latest refresh token, not just for **refresh token rotation** but also for **key rotation**.



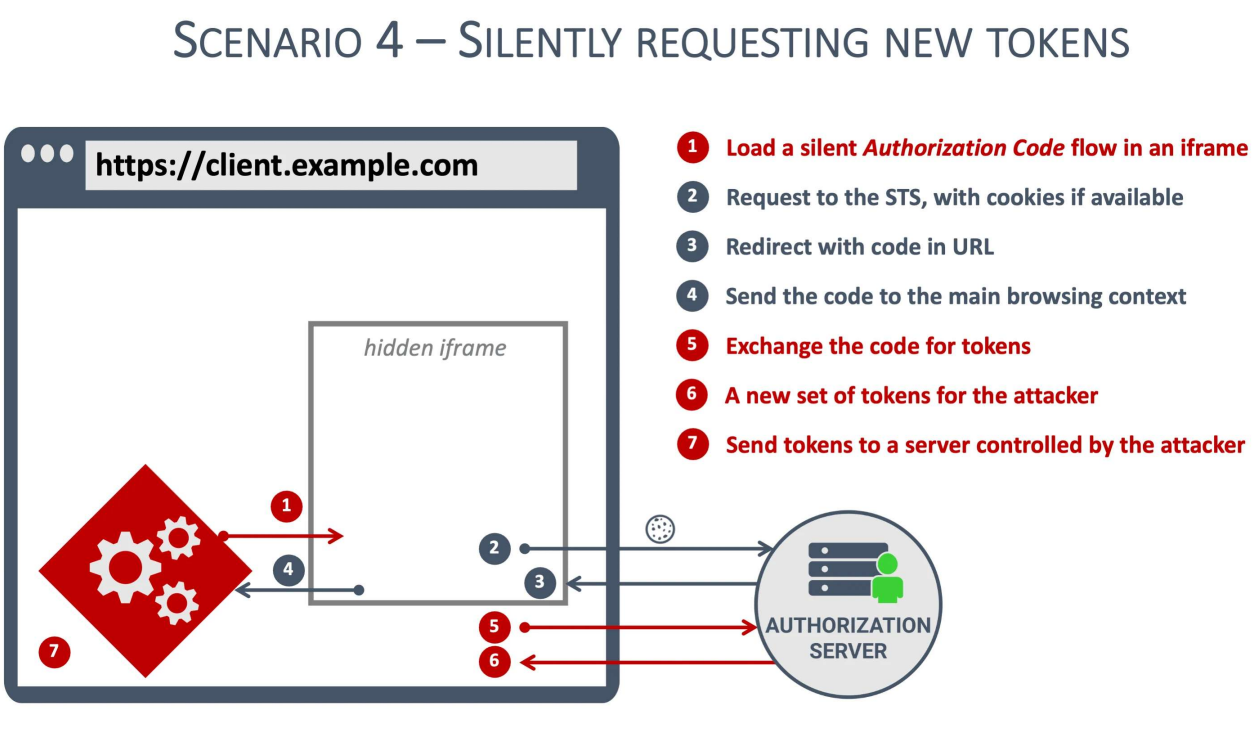


Bypassing Refresh Token Rotation





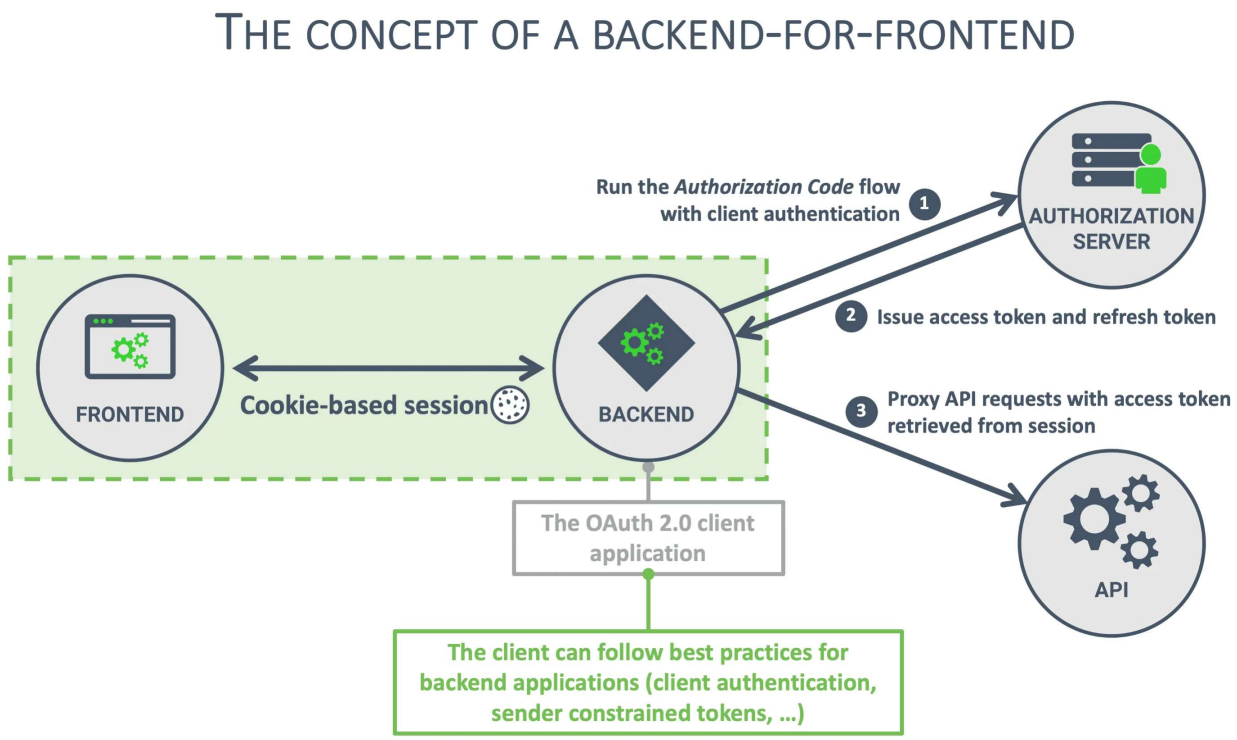
`Scenario 3: Impersonating the legitimate client`



6

- `/auth?scope=opendi&..&prompt=none` parameter in authorization request allows for clients to just get an access token if the user has an SSO session
- [A Critical Analysis of Refresh Token Rotation in Single-page Applications | Ping Identity]

Backend For Frontend



7

OAuth 2.0 Scopes

`scope=openid email profile course:write`

- A space-delimited string with scope values.

- A mechanism defined by OAuth 2.0 to define the scope of an access token: what does the descretionary access control of the token allow the bearer of the token to do?
- Scopes can be used both to decide what fields (or claims) to include in the token and by backends to decide what APIs the token can access.
- OAuth 2.0 does not define any scopes, but OIDC has some reserved scopes e.g. `openid`, `email`, `profile`.
- 👆 OAuth 2.0 Scopes for Google APIs | Authorization | Google for Developers
- 👆 Scopes for OAuth apps - GitHub Docs

OIDC Discovery Endpoint

- A standard endpoint for loading provider details to configure clients and resource servers.
- `/.well-known/openid-configuration`

```
// Example response
{
  "issuer": "http://localhost:8088/realms/course-practice",
  "authorization_endpoint": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/auth",
  "token_endpoint": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/token",
  "introspection_endpoint": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/token/introspect",
  "userinfo_endpoint": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/userinfo",
  "end_session_endpoint": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/logout",
  "frontchannel_logout_session_supported": true,
  "frontchannel_logout_supported": true,
  "jwks_uri": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/certs",
  "check_session_iframe": "http://localhost:8088/realms/course-practice/protocol/openid-
connect/login-status-iframe.html",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials",
    "urn:openid:params:grant-type:ciba",
    "urn:ietf:params:oauth:grant-type:device_code"
  ],
  "acr_values_supported": [
    "0",
    "1"
  ],
  "response_types_supported": [
    "code",
    "none",
    "id_token",
    "token",
    "id_token token",
    "code id_token",
```

```
    "code token",
    "code id_token token"
  ],
```

OIDC Logout

- OAuth 2.0 has no logout feature, but users can revoke an app's access to act on their behalf by revoking access and refresh tokens.
- OIDC has several standards for logout:

RP-initiated Logout

```
{
        "end_session_endpoint": "http://localhost:8088/realms/course-
practice/protocol/openid-connect/logout"
}
```

```
// This by default does Single Sign Out / Single Logout in Keycloak
POST http://localhost:8088/realms/course-practice/protocol/openid-connect/logout
Content-Type: application/x-www-form-urlencoded

id_token_hint=eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJnX3F6UkxvZHJnN0IxWl9tM1dt
RDVaRkpHaXczOE9BVTNCSXFqbE5JR0dJIn0.eyJleHAiOjE3MTQ3ODM3NjgsImlhdCI6MTcxNDc4MzQ2OCwiYXV0aF
90aW1lIjoxNzE0NzgwMTk3LCJqdGkiOiI1ODdjNjY1OC1lNDY5LTRiOTktODg0YS02NzQ1NDhiNTBlMzMiLCJpc3Mi
OiJodHRwOi8vbG9jYWxob3N0OjgwODgvcmVhbG1zL2NvdXJzZS1wcmFjdGljZSIsImF1ZCI6Im9pZGMtcGxheWdyb3
VuZCIsInN1YiI6IjE5NWFjNDQ0LTYxNjUtNGNmZS1iZWJkLWVjMDNiM2Q5MmIxYyIsInR5cCI6IklEIiwiYXpwIjoi
b2lkYy1wbGF5Z3JvdW5kIiwic2Vzc2lvbl9zdGF0ZSI6ImFjMjg3MDVmLWY2YzUtNDA3MS05NTk4LWQ3MWI4Yzc0ZT
U4NSIsImF0X2hhc2giOiJKRXp5dFdQbVBGVFQ2MVE4eUNmRF9nIiwiYWNyIjoiMCIsInNpZCI6ImFjMjg3MDVmLWY2
YzUtNDA3MS05NTk4LWQ3MWI4Yzc0ZTU4NSIsInByb2ZpbGVQaWN0dXJlIjoiaHR0cHM6Ly9pLm5hdGdlb2ZlLmNvbS
9uLzU0ODQ2N2Q4LWM1ZjEtNDU1MS05ZjU4LTY4MTdhOGQyYzQ1ZS9OYXRpb25hbEdlb2dyYXBoaWNfMjU3MjE4N19z
cXVhcmUuanBnIiwicmVzb3VyY2VfYWNjZXNzIjp7ImFjY291bnQiOnsicm9sZXMiOlsibWFuYWdlLWFjY291bnQiLC
JtYW5hZ2UtYWNjb3VudC1saW5rcyIsInZpZXctcHJvZmlsZSJdfSwiY291cnNlLWJhY2tlbmQiOnsicm9sZXMiOlsi
Y291cnNlOndyaXRlIl19fSwiZW1haWxfdmVyaWZpZWQiOnRydWUsInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJvZm
ZsaW5lX2FjY2VzcyIsInN1cGVyYWRtaW4iLCJ1bWFfYXV0aG9yaXphdGlvbiIsImRlZmF1bHQtcm9sZXMtY291cnNl
LXByYWN0aWNlIl19LCJuYW1lIjoiQm9iIE1vcnRpbWVyIiwicHJlZmVycmVkX3VzZXJuYW1lIjoiYm9iQGV4YW1wbG
UuY29tIiwiZ2l2ZW5fbmFtZSI6IkJvYiIsImZhbWlseV9uYW1lIjoiTW9ydGltZXIiLCJlbWFpbCI6ImJvYkBleGFt
cGxlLmNvbSJ9.Iw0Ak5CB1WcDd69tosAKVwdnrjWQ1I0kORTS40ULLpXsZLlrGp7Lup-
L5Robv74vNuXtZvulNcGhnaLFdQhPslE5xwl8z-dqQ6RvGsLKJYS3Sd5_sJamnUtmkxHoumVV1AycYsHxH-
DjtlD1QqH73J4ZIHQN5L5e6d25ZeyR7F5NQ97CCOz8FF60-hWu-
cLDopn2EcuCb2xWj1xr22U8GWA_cByuWhLmqmKaTwi05l7_BOWGo25xTLYD4WQgojWKxWCRw0zg4KlsAKOxz9hw-
pQij_QMa9n1MlSyIb4dMGWtKF3xVjquVH8Qhr4TBhQS6RrvifwToiolat3Xtv_GhA

&post_logout_redirect_uri=http://localhost:8000/

&state=hello
```

How does an application figure out that we're logged out?

- The simplest and perhaps most robust mechanism for an application to discover if a logout has taken place is simply to **leverage the fact that ID and access token usually have a short expiration**. As Authorizaiton Servers invalidate the session on logout, a refresh token can no longer be used to obtain new tokens.
- In cases where tokens have a **long validity**, it is still good practice to invoke the **Token Introspection endpoint to check token validity periodically**, which we

will look at in the next chapter.

Back-Channel Logout

- Through OIDC Back-Channel Logout, an application can register an endpoint to receive logout events.
- When a logout is initiated with AS, it will send a logout token to all applications in the session that have a back-channel logout endpoint registered.
- The logout token is similar to an ID token, so it is a signed JWT. On receiving the logout token, the application verifies the signature and can now log out of the application session associated with the AS session ID.

Keycloak

🔑😘🔑😘

Terminology

Realm

Think of a realm as a *tenant*. A realm is fully isolated from other realms; it has its own configuration, and its own set of applications and users. This allows a single installation of Keycloak to be used for multiple purposes. For example, you may want to have one realm for internal applications and employees, and another realm for external applications and customers.

User

A user in the realm.

Group

A group of users. They inherit *attributes* from the group e.g. all employees in an office have the office address. They also inherit *roles* from the group.

Realm Role

Global roles.

Composite Role

A role that contains other roles inside it. Like hierarchical roles in Spring Security. Composite roles can contain other composite roles. (Careful, bad performance if overused, difficult to manage).

Client

An OAuth 2.0 client / OIDC relying party. A client is configured using Client Scopes.

Client Scope

Scopes can be used both to decide what fields (or claims) to include in the token and by resource servers to decide what APIs the token can access.

Session

As part of the authentication process, Keycloak may create server-side sessions and correlate them with tokens. By relying on these sessions, Keycloak is able to:

- keep the state of the authentication context where sessions originated,
- track users' and clients' activity,
- check the validity of tokens, and
- decide when users and clients should re-authenticate.

SSO Session / User Session

Firstly, a user session is created to track the user activity regardless of the client. This first level is what is called the SSO session, also referred to as a user session.

The SSO session is like an HTTP session. Both are used to track and maintain the state across multiple requests from the same agent.

Setting	Description
`SSO Session Max`	How long a session should stay active e.g. 12 hours
`SSO Session Idle`	How long a user can stay idle before their session is prematurely expired e.g. 30 minutes without any activity on the session The idle timeout is bumped every time users interact with Keycloak, either directly through the authorization endpoint – when using a browser – or indi- rectly when tokens are refreshed by clients. If `Client Sesison Idle` is not set, this is the lifetime of refresh tokens.
`Client Session Idle`	Same but for clients. Can be overriden for a specific client. If set, this is the lifetime of refresh tokens.
`Client Session Max`	Same but for clients. Can be overriden for a specific client.
`Access Token Lifespan`	How long Access tokens and ID tokens live.

Client Session

At the second level, Keycloak creates a client session to track the user activity for each client the user is authenticated to in the user session. Client sessions are strictly related to the validity of tokens and how they are used by applications.

Keycloak Demo

📑💂🏻 Running Keycloak using Docker

```
# OIDC Playground
git clone https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition.git
cd ch4
npm install
npm start
```

```
docker run -p 8088:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/keycloak:24.0.3 start-dev
```

Introduction

- [ ] Creating a realm: course
- [ ] Creating a user: bob

- [ ] Adding custom attributes to the user: profile_picture
- [ ] Creating a realm role: superadmin
- [ ] Creating a group: instructors
- [ ] Adding a role to the group
- [ ] Adding custom attributes to the group

OIDC Playground

- [ ] Create a client

- Client ID: oidc-playground
- Client authentication: Off
- Authentication flow: Standard flow
- Valid Redirect URIs: http://localhost:8000/
- Web Origins: http://localhost:8000

- [ ] Go through OIDC flow
- [ ] Creating a client_scope, mapper, and adding it to a client to show the profilePicture attribute
- [ ] Showing roles in ID token
- [ ] Adding a course-backend client and adding a role for it in the instructors group
- [ ] Adding course-backend to audience (check access token) Via the oidc-playground client having a scope on the course-backend client role (audience resolve)
- [ ] Adding a mapper directly to a client without using a client scope
- [ ] UserInfo endpoint can only be invoked using an access token obtained from an OIDC flow
- [ ] Single Logout

```
###
@id_token_hint=eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJnX3F6UkxvZHJnN0IxWl9tM1d
tRDVaRkpHaXczOE9BVTNCSXFqbE5JR0dJIn0.eyJleHAiOjE3MTQ3ODQxODcsImlhdCI6MTcxNDc4Mzg4NywiYXV0a
F90aW1lIjoxNzE0NzgzODc0LCJqdGkiOiI5MmI3YWRjZS03MWEzLTRjZTYtOTIwZi1iZjZiNDM4NWVlZmMiLCJpc3M
iOiJodHRwOi8vbG9jYWxob3N0OjgwODgvcmVhbG1zL2NvdXJzZS1wcmFjdGljZSIsImF1ZCI6Im9pZGMtcGxheWdyb
3VuZCIsInN1YiI6IjE5NWFjNDQ0LTYxNjUtNGNmZS1iZWJkLWVjMDNiM2Q5MmIxYyIsInR5cCI6IklEIiwiYXpwIjo
ib2lkYy1wbGF5Z3JvdW5kIiwic2Vzc2lvbl9zdGF0ZSI6IjExZjA3ODQ2LTVlMmQtNDQyYi1hN2ZlLTU2YTUyNzIzY
TFjMiIsImF0X2hhc2giOiJmTHd4TkFPSFd3SmpsNmc3R0hUYmxBIiwiYWNyIjoiMCIsInNpZCI6IjExZjA3ODQ2LTV
lMmQtNDQyYi1hN2ZlLTU2YTUyNzIzYTFjMiIsInByb2ZpbGVQaWN0dXJlIjoiaHR0cHM6Ly9pLm5hdGdlb2ZlLmNvb
S9uLzU0ODQ2N2Q4LWM1ZjEtNDU1MS05ZjU4LTY4MTdhOGQyYzQ1ZS9OYXRpb25hbEdlb2dyYXBoaWNfMjU3MjE4N19
zcXVhcmUuanBnIiwicmVzb3VyY2VfYWNjZXNzIjp7ImFjY291bnQiOnsicm9sZXMiOlsibWFuYWdlLWFjY291bnQiL
CJtYW5hZ2UtYWNjb3VudC1saW5rcyIsInZpZXctcHJvZmlsZSJdfSwiY291cnNlLWJhY2tlbmQiOnsicm9sZXMiOls
iY291cnNlOndyaXRlIl19fSwiZW1haWxfdmVyaWZpZWQiOnRydWUsInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJvZ
mZsaW5lX2FjY2VzcyIsInN1cGVyYWRtaW4iLCJ1bWFfYXV0aG9yaXphdGlvbiIsImRlZmF1bHQtcm9sZXMtY291cnN
lLXByYWN0aWNlIl19LCJuYW1lIjoiQm9iIE1vcnRpbWVyIiwicHJlZmVycmVkX3VzZXJuYW1lIjoiYm9iQGV4YW1wb
GUuY29tIiwiZ2l2ZW5fbmFtZSI6IkJvYiIsImZhbWlseV9uYW1lIjoiTW9ydGltZXIiLCJlbWFpbCI6ImJvYkBleGF
tcGxlLmNvbSJ9.gLu5H5zyUoOTBTWgZ81q7779CuDWHP3R5wKqs4QOxqQndssqMVGPMvsU1yIAmA0JqjMQGbya66q2
FGBL-
nO3eX19_Xxv3kdKWOJ1EH75kMJ8MQqTPvk3_YFEagKZaVohc7zTX1kynUFPzLfRUIjvI9kUAZSunBjY_0NGZG5Ri7X
u2mNuKZicvfaoA32PIf4O4ZSWnIHwKFGVYWnJ2wQW96-BhTtLGCUsbaIsBkz-cDZ25gFZOhDnWsB_X-
QW0Uovn3MjI4BUk0CDlAgOf8agIVvjoV37vcR_ZgAbQzRuzVGIjbg8z39-Wrm4M-
eExwE52_uECuBUoWHGCS_pwCXfoQ
POST http://localhost:8088/realms/course-practice/protocol/openid-connect/logout
Content-Type: application/x-www-form-urlencoded

id_token_hint = {{id_token_hint}} &
```

```
post_logout_redirect_uri = http://localhost:8000/ &
state = {"copy": "me"}
```

- [ ] Enable refresh token rotation

Spring Security OAuth 2.0 Demo

- [ ] Create superadmin > course:read, course:write
- [ ] Create instructors and students group
- [ ] Remove unnecessary roles claims from ID token
- [ ] Use access token to invoke the API

👆 Admin Console
👆 User Console

Spring Security OAuth 2.0

- OAuth 2.0 Login
- The OAuth 2.0 Login feature lets an application have users log in to the application by using their existing account at an OAuth 2.0 Provider (such as GitHub) or OpenID Connect 1.0 Provider (such as Google). OAuth 2.0 Login implements two use cases: “Login with Google” or “Login with GitHub”.
- You want to implement a login with X provider feature.
- OAuth 2.0 Client
- The OAuth 2.0 Client features provide support for the Client role as defined in the OAuth 2.0 Authorization Framework.
- Your app is a client some API that is secured using OAuth 2.0 and you want to get a token on behalf of a user to access that API.
- Spring Authorization Server
- Spring Authorization Server is a framework that provides implementations of the OAuth 2.1 and OpenID Connect 1.0 specifications and other related specifications. It is built on top of Spring Security to provide a secure, light-weight, and customizable foundation for building OpenID Connect 1.0 Identity Providers and OAuth2 Authorization Server products.

Spring Security OAuth 2.0 Resource Server

- [ ] 👆 OAuth 2.0 Resource Server Architecture
- [ ] 👆 Token Propagation and Bearer Token Failure
- [ ] 👆 OAuth 2.0 Resource Server JWT
- [ ] 👆 OAuth 2.0 Resource Server Multi-tenancy Spring Security
    - Allows you to support multiple token types, multiple Authorization Servers, at the same time.

References

- [ ] ⭐ docs - Spring Security Docs
- [ ] ⭐ course - Spring Security Fundamentals 2022 - YouTube
- [ ] ⭐ Keycloak - book - Keycloak - Identity and Access Management for Modern Applications - Second Edition: Harness the power of Keycloak, OpenID Connect and OAuth 2.0 to secure applications: Thorgersen, Stian, Silva, Pedro Igor: 9781804616444: Amazon.com: Books
- [ ] ⭐ OAuth 2.0 and OIDC talk - Introduction to OAuth 2.0 and OpenID Connect By Philippe De Ryck - YouTube
- [ ] Legacy Spring Security Architecture (5 and older) talk - Architecture Deep Dive in Spring Security - Joe Grandja @ Spring I/O 2017 - YouTube
- [ ] Great workshop on new Spring Security 6 talk - Spring Security, demystified by Daniel Garnier Moiroux - YouTube
- [ ] Migrating from Spring Security 5 to 6 talk - Everything new in Spring Security 6 baked with a Spring Boot 3 recipe By Laur Spilca - YouTube
- [ ] Implementing `AuthorizationManager` with OPA example talk - Spring Security 6: The Next Generation - YouTube
- [ ] JWT Concept video - What is JWT? JSON Web Tokens Explained (Java Brains) - YouTube
- [ ] JWT Structure video - What is the structure of a JWT - Java Brains - YouTube
- [ ] Spring Security Authorization Server talk - Deep diving in Spring Security with the latest trends and additions By Laur Spilca - YouTube
- [ ] OWASP Top 10 talk - Philippe de Ryck - SEVEN things about API security
- [ ] Cryptography talk - Cryptography 101 for Java developers by Michel Schudel - YouTube