Spring Security 6

MISS XING

What is Spring Security?

Spring Security is a framework that focuses on providing both **authentication** and **authorization** (or “access-control”) to Java web application and SOAP/RESTful web services.

At its core, Spring Security is really just a bunch of **servlet filters** that help you add authentication and authorization to your web application.

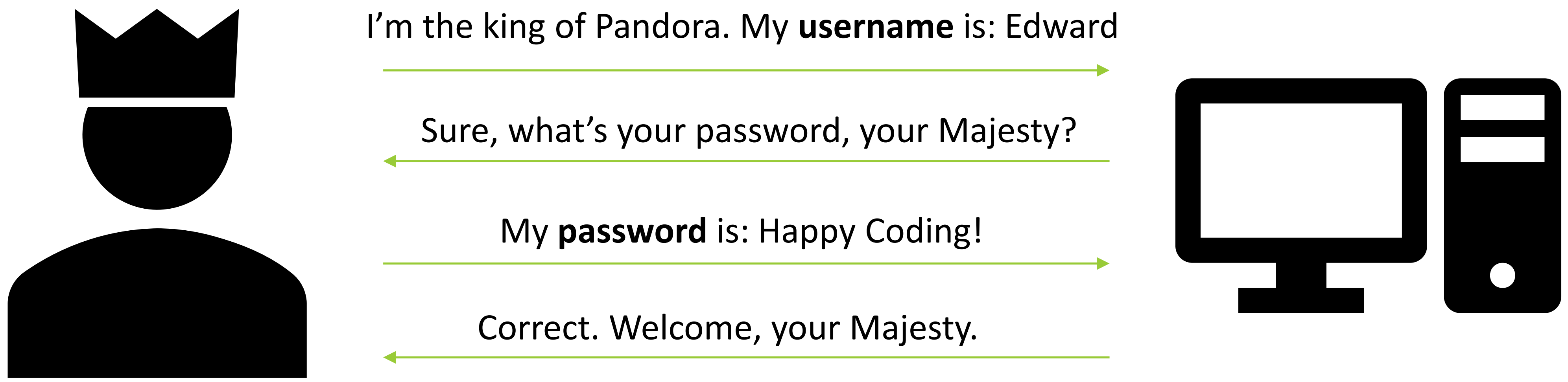
Spring Security currently supports integration with all of the following technologies:

- HTTP basic access authentication
- Form Login
- Outh2
- …..
- Your own authentication systems

It is built on top of Spring Framework

Authentication

Authentication is the process of verifying the identity of an individual, system, or entity to ensure that they are who they claim to be.



I’m the king of Pandora. My **username** is: Edward

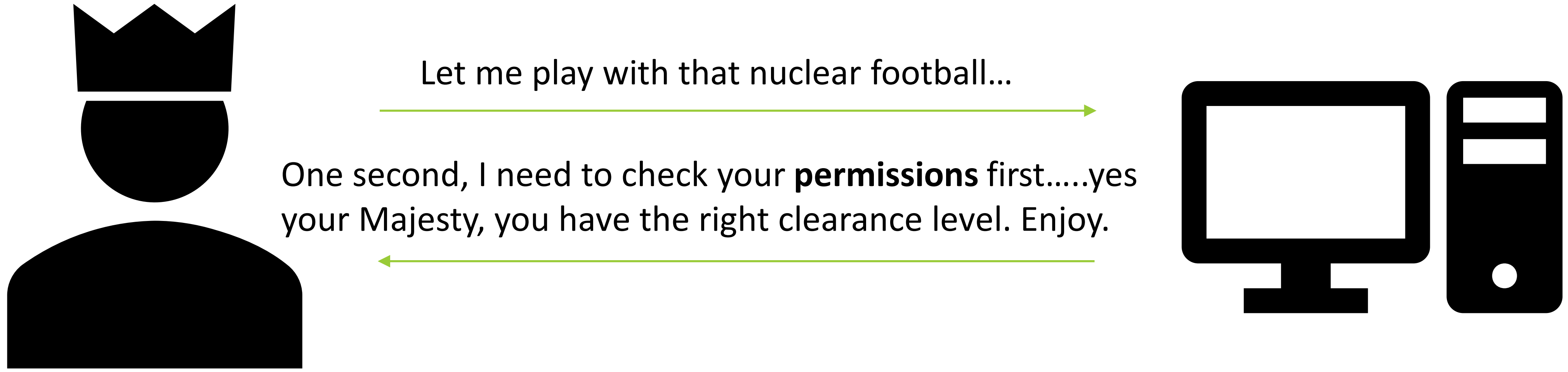
Sure, what’s your password, your Majesty?

My **password** is: Happy Coding!

Correct. Welcome, your Majesty.

Authorization

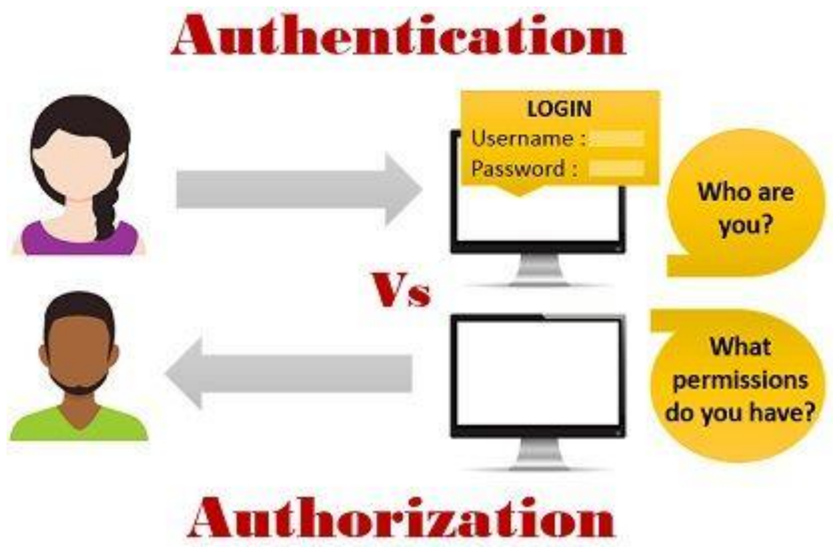
Authorization is the process of determining whether a user, system, or entity has the appropriate permissions and privileges to access specific resources, perform certain actions, or execute particular operations within a computer system, application, or network.



Let me play with that nuclear football…

One second, I need to check your **permissions** first…..yes your Majesty, you have the right clearance level. Enjoy.

Authentication vs Authorization



BASIS FOR COMPARISON	AUTHENTICATION	AUTHORIZATION
Basic	Checks the person's identity to grant access to the system.	Checks the person's privileges or permissions to access the resources.
Includes process of	Verifying user credentials.	Validating the user permissions.
Order of the process	Authentication is performed at the very first step.	Authorization is usually performed after authentication.
Examples	In the online banking applications, the identity of the person is first determined with the help of the user ID and password.	In a multi-user system, the administrator decides what privileges or access rights do each user have.

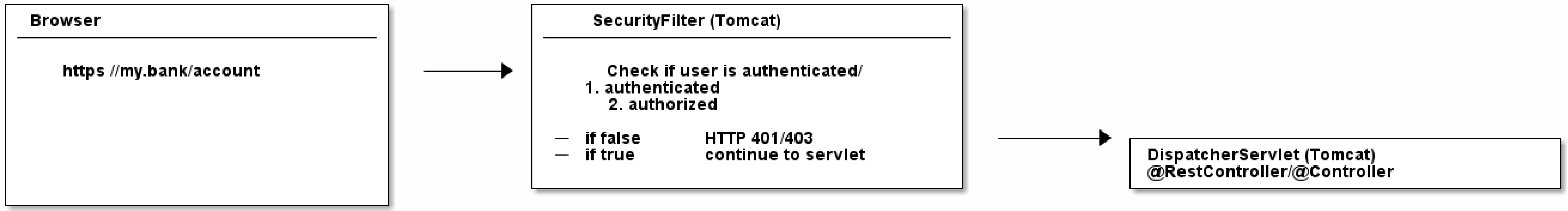
Spring Security Architecture - Filters

MISS XING

Why use Servlet Filters?

In any Spring Web application, there's essentially one main servlet: the `DispatcherServlet`, which directs incoming HTTP requests to the appropriate `@Controllers` or `@RestControllers`. However, it's important to note that the `DispatcherServlet` does not include any built-in security features, and handling raw HTTP Basic Auth headers directly in your `@Controllers` is not ideal. Ideally, authentication and authorization should occur before requests reach your `@Controllers`.

In the Java web environment, this can be achieved by implementing filters in front of your servlets. By creating a `SecurityFilter` and configuring it in your Tomcat (servlet container/application server), you can intercept every incoming HTTP request before it reaches the servlet.



Browser
https //my.bank/account
SecurityFilter (Tomcat)
Check if user is authenticated/
1. authenticated
2. authorized
— if false HTTP 401/403
— if true continue to servlet
DispatcherServlet (Tomcat)
@RestController/@Controller

A naive SecurityFilter

```java
public class SecurityServletFilter extends HttpFilter {

    @Override
    protected void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws IOException, ServletException {

        UsernamePasswordToken token = extractUsernameAndPasswordFrom(request);  // (1)

        if (notAuthenticated(token)) {  // (2)
            // either no or wrong username/password
            // unfortunately the HTTP status code is called "unauthorized", instead of "unauthenticated"
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // HTTP 401.
            return;
        }

        if (notAuthorized(token, request)) { // (3)
            // you are logged in, but don't have the proper rights
            response.setStatus(HttpServletResponse.SC_FORBIDDEN); // HTTP 403
            return;
        }

        // allow the HttpRequest to go to Spring's DispatcherServlet
        // and @RestControllers/@Controllers.
        chain.doFilter(request, response); // (4)
    }

    private UsernamePasswordToken extractUsernameAndPasswordFrom(HttpServletRequest request) {
        // Either try and read in a Basic Auth HTTP Header, which comes in the form of user:password
        // Or try and find form login request parameters or POST bodies, i.e. "username=me" & "password="myPass"
        return checkVariousLoginOptions(request);
    }


    private boolean notAuthenticated(UsernamePasswordToken token) {
        // compare the token with what you have in your database...or in-memory...or in LDAP...
        return false;
    }

    private boolean notAuthorized(UsernamePasswordToken token, HttpServletRequest request) {
        // check if currently authenticated user has the permission/role to access this request's /URI
        // e.g. /admin needs a ROLE_ADMIN , /callcenter needs ROLE_CALLCENTER, etc.
        return false;
    }
}
```

FilterChains Intro

- What’s the problem of the previous slides’ code?
  - A giant monster…

- How can we solve the problem?
  1. First, go through a LoginMethodFilter…
  2. Then, go through an AuthenticationFilter…
  3. Then, go through an AuthorizationFilter…
  4. Finally, hit your servlet.
- The concept is called *FilterChain*.

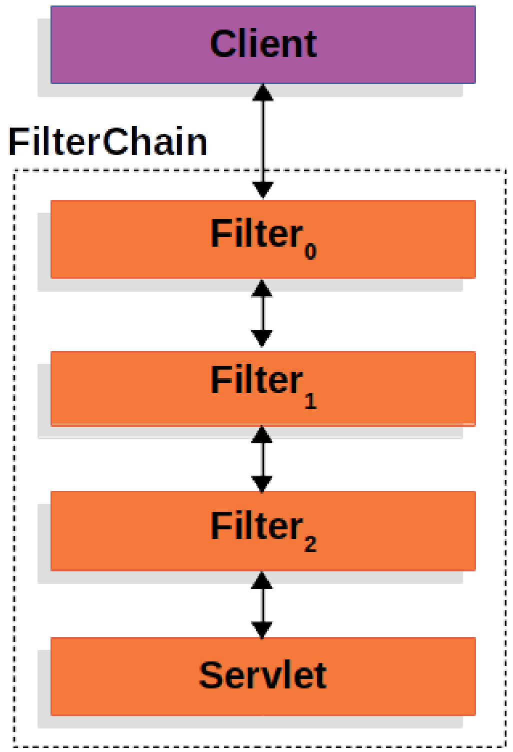
Spring Filters

The right image shows the typical layering of the handlers for a single HTTP request.

The client sends a request to the application, and the container creates a FilterChain, which contains the Filter instances and Servlet that should process the HttpServletRequest, based on the path of the request URI. In a Spring MVC application, the Servlet is an instance of `DispatcherServlet`.

The power of the Filter comes from the FilterChain that is passed into it.

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
   // do something before the rest of the application
   chain.doFilter(request, response); // invoke the rest of the application
   // do something after the rest of the application
}
```



Observe: Spring Security DefaultSecurityFilterChain – Setup Project

1. Create a Spring Boot Project with ProductRestController

2. Add Spring Security Configuration

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

3. Logging configuration

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

```
@EnableWebSecurity(debug = true)
public class SecurityConfigInMemory {

}
```

Observe: Spring Security DefaultSecurityFilterChain – Setup Project

- Log Message when starts the application:
  - Spring Security does not just install *one* filter, instead it installs a whole filter chain consisting of 15 (!) different filters
  - when an HTTPRequest comes in, it will go through *all* these 15 filters, before your request finally hits your @RestControllers or @Controller
  - The order is important, too, starting at the top of that list and going down to the bottom.

```
2023-10-03T13:56:52.791-05:00  INFO 19436 --- [  restartedMain] o.s.s.web.DefaultSecurityFilterChain     : Will secure
any request with [org.springframework.security.web.session.DisableEncodeUrlFilter@6c41fa9,
org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@18f9a02,
org.springframework.security.web.context.SecurityContextHolderFilter@67b67c06,
org.springframework.security.web.header.HeaderWriterFilter@70eb871d,
org.springframework.security.web.csrf.CsrfFilter@7a3ca16c,
org.springframework.security.web.authentication.logout.LogoutFilter@5d99d455,
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@43ae4b4f,
org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@3b034285,
org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter@10483b2e,
org.springframework.security.web.authentication.www.BasicAuthenticationFilter@2f85092a,
org.springframework.security.web.savedrequest.RequestCacheAwareFilter@11d1244a,
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@606a7c9d,
org.springframework.security.web.authentication.AnonymousAuthenticationFilter@566aa182,
org.springframework.security.web.access.ExceptionTranslationFilter@59061fd0,
org.springframework.security.web.access.intercept.AuthorizationFilter@49ffbdf1]
```

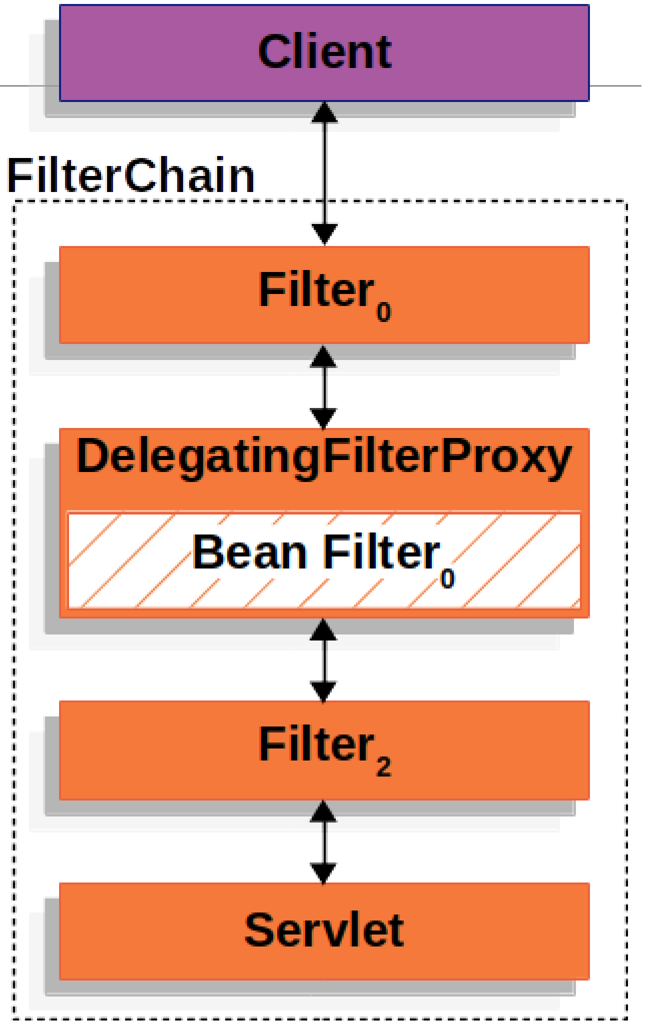
DelegatingFilterProxy

Spring provides a `Filter` implementation named `DelegatingFilterProxy` that allows bridging between the Servlet container's lifecycle and Spring's `ApplicationContext`.

The Servlet container allows registering `Filter` instances by using its own standards, but it is not aware of Spring-defined Beans. You can register `DelegatingFilterProxy` through the standard Servlet container mechanisms but delegate all the work to a Spring Bean that implements Filter.

`DelegatingFilterProxy` looks up Bean `Filter0` from the `ApplicationContext` and then invokes Bean `Filter0`. The following listing shows pseudo code of `DelegatingFilterProxy`:

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

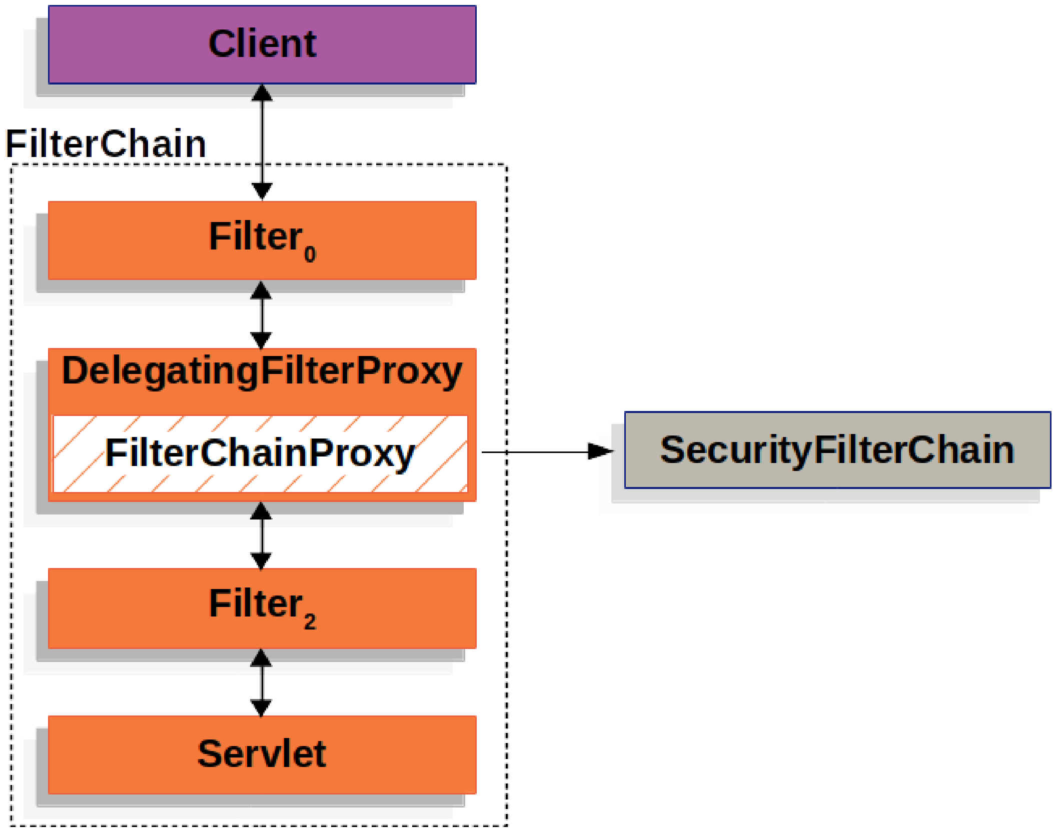


FilterChainProxy

Spring Security's Servlet support is contained within `FilterChainProxy`.

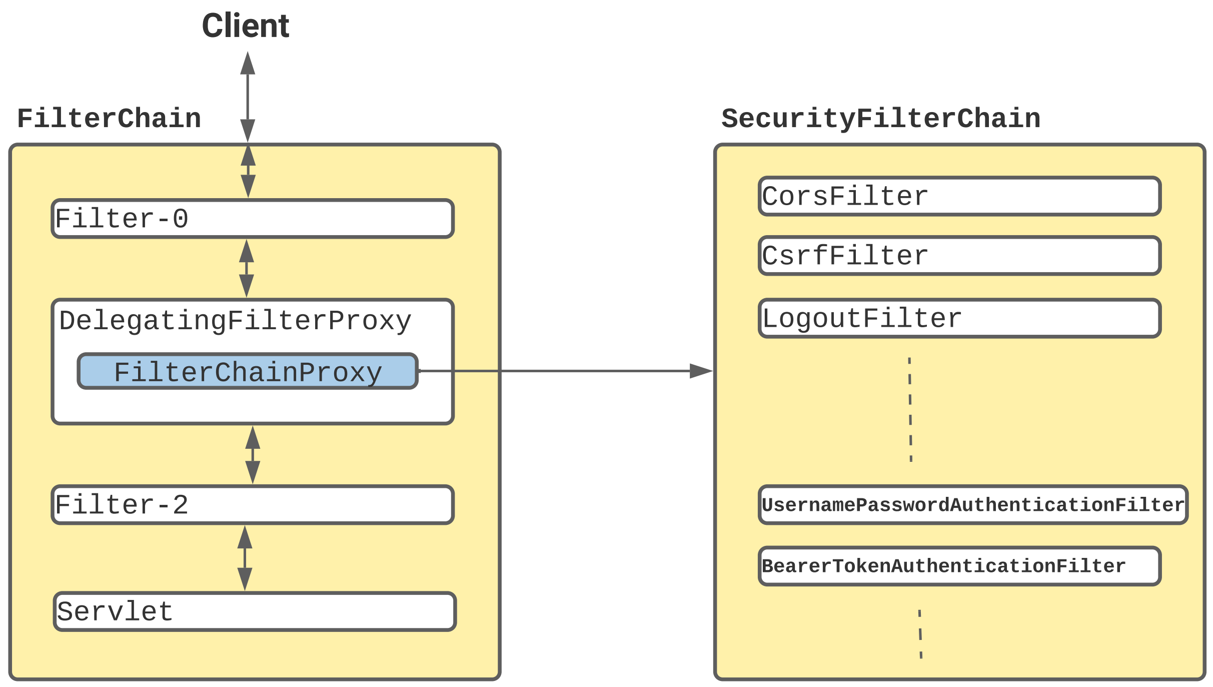
`FilterChainProxy` is a special `Filter` provided by Spring Security that allows delegating to many Filter instances through `SecurityFilterChain`.

Since `FilterChainProxy` is a Bean, it is typically wrapped in a `DelegatingFilterProxy`.



SecurityFilterChain

`SecurityFilterChain` is used by `FilterChainProxy` to determine which Spring Security `Filter` instances should be invoked for the current request.

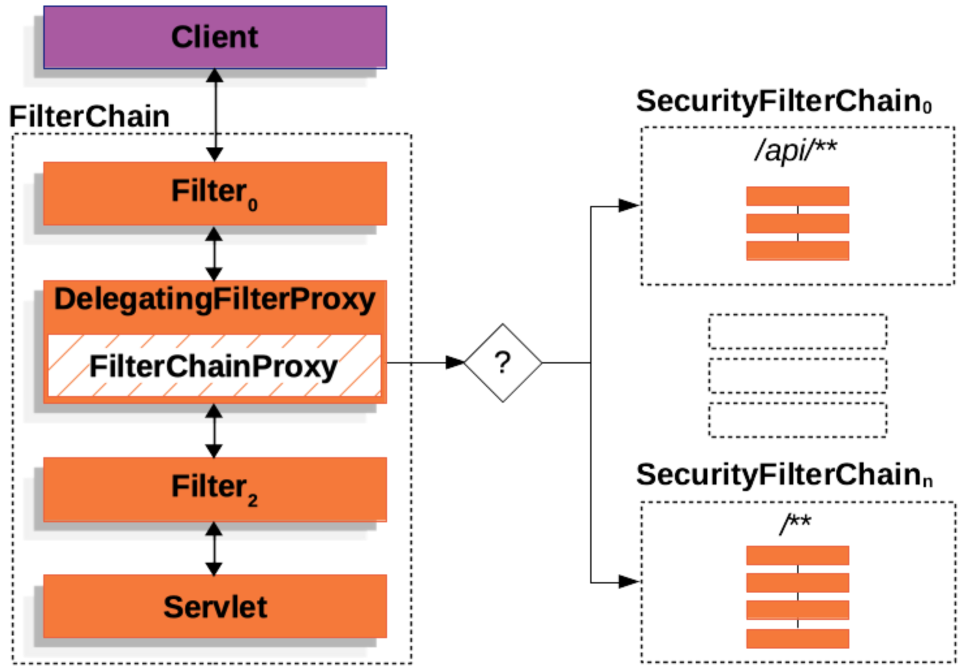


Multiple SecurityFilterChain instances

In the Multiple SecurityFilterChain figure, FilterChainProxy decides which SecurityFilterChain should be used. Only the first SecurityFilterChain that matches is invoked.

If a URL of /api/messages/ is requested, it first matches on the SecurityFilterChain0 pattern of /api/**, so only SecurityFilterChain0 is invoked, even though it also matches on SecurityFilterChainn.

If a URL of /messages/ is requested, it does not match on the SecurityFilterChain0 pattern of /api/**, so FilterChainProxy continues trying each SecurityFilterChain. Assuming that no other SecurityFilterChain instances match, SecurityFilterChainn is invoked.



Security Filters

The Security Filters are inserted into the `FilterChainProxy` with the `SecurityFilterChain` API.

- Those filters can be used for a number of different purposes, like authentication, authorization, exploit protection, and more.
- The filters are executed in a specific order to guarantee that they are invoked at the right time.
  - For example, the `Filter` that performs authentication should be invoked before the `Filter` that performs authorization.

```
@Configuration
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
                .formLogin(Customizer.withDefaults())
                .authorizeHttpRequests(authorize -> authorize.requestMatchers("/products/*").authenticated());

        return http.build();
    }

}
```

Adding a Custom Filter to the Filter Chain

Most of the time, the default security filters are enough to provide security to your application. However, there might be times that you want to add a custom Filter to the security filter chain.

For example, let’s say that you want to add a Filter that gets a tenant id header and check if the current user has access to that tenant. The previous description already gives us a clue on where to add the filter, since we need to know the current user, we need to add it after the authorization filters.

```
public class TenantFilter extends OncePerRequestFilter {

    @Override
    public void doFilterInternal(HttpServletRequest servletRequest, HttpServletResponse servletResponse, FilterChain filterChain)
                    throws ServletException, IOException {

        String tenantId = request.getHeader("X-Tenant-Id");
        boolean hasAccess = isUserAllowed(tenantId);
        if (hasAccess) {
            filterChain.doFilter(request, response);
            return;
        }
        throw new AccessDeniedException("Access denied");
    }

    private boolean isUserAllowed(String tenantId){
        return tenantId.contains(“admin");
    }
}
```

Adding a Custom Filter to the Filter Chain

```
@Configuration
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
                .formLogin(Customizer.withDefaults())
                .httpBasic(Customizer.withDefaults())
                .addFilterAfter(new TenantFilter(), AuthorizationFilter.class)
                .authorizeHttpRequests(authorize -> authorize.requestMatchers("/products/*").authenticated());

        return http.build();
    }

}
```

```
GET http://localhost:8080/products/123
Authorization: Basic user f9f4896b-4729-44c4-bfbb-155e40393333
X-Tenant-Id: MIU123456
```

rest.http

Spring Security Common Filters

- BasicAuthenticationFilter
  - When using HTTP Basic Authentication, this filter extracts and verifies the credentials sent in the '**`Authorization`**' header of HTTP requests.
- UsernamePasswordAuthenticationFilter
  - This filter handles form-based authentication by intercepting requests to the login endpoint (typically '**`/login`**') and processing submitted username and password credentials to authenticate users.
- DefaultLoginPageGeneratingFilter
  - Is responsible for generating a default login page in web applications that use form-based authentication. This filter is automatically added to the Spring Security filter chain when you configure form-based authentication with Spring Security.
- DefaultLogoutPageGeneratingFilter
  - It's responsible for generating a default logout page for web applications that use Spring Security for authentication and session management. This filter is automatically added to the Spring Security filter chain when you configure logout functionality.
- FilterSecurityInterceptor
  - This is the heart of Spring Security's authorization mechanism. It performs access control checks (authorization) based on the configured security rules, typically using expressions like '`hasRole`' and '`hasPermission`'.

Authentication Architecture

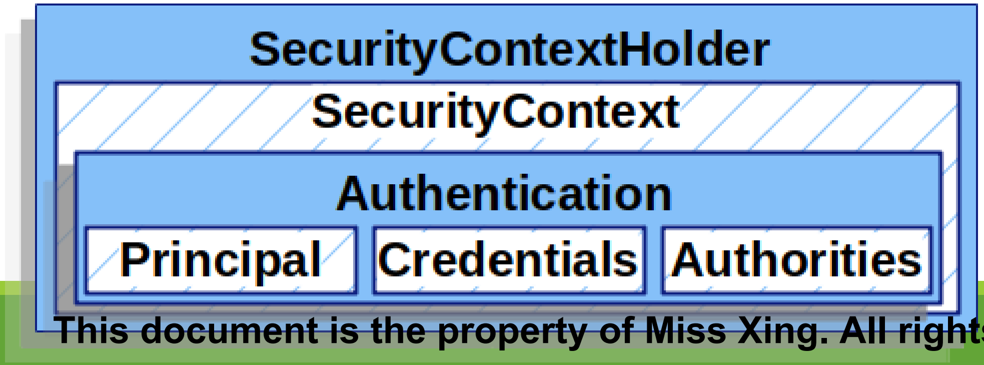
MISS XING

Servlet Authentication Architecture

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



Authentication

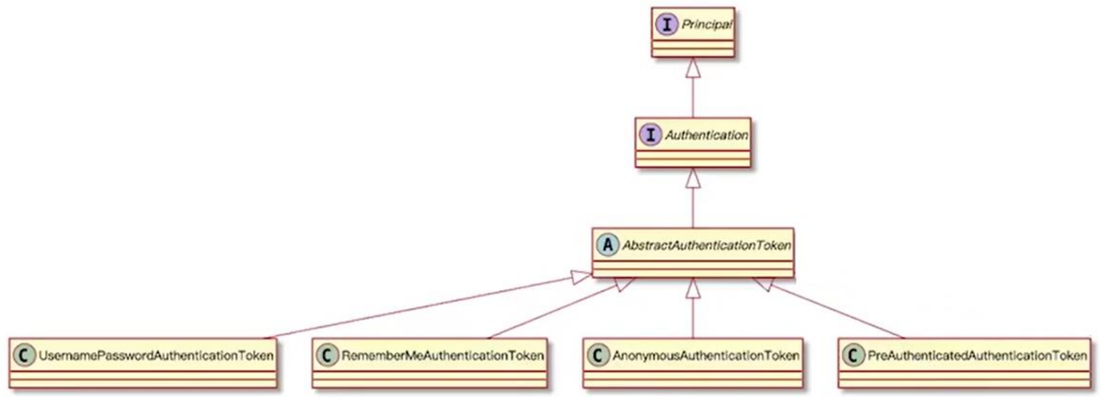
The `Authentication` interface serves two main purposes within Spring Security:

- 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.

Authentication


Principal
Authentication
AbstractAuthenticationToken
UsernamePasswordAuthenticationToken
RememberMeAuthenticationToken
AnonymousAuthenticationToken
PreAuthenticatedAuthenticationToken

```json
{
  "authorities": [],
  "details": {
    "remoteAddress": "0:0:0:0:0:0:0:1",
    "sessionId": "A7387F558714B2FC56D3DD17B18C627D"
  },
  "authenticated": true,
  "principal": {
    "password": null,
    "username": "user",
    "authorities": [],
    "accountNonExpired": true,
    "accountNonLocked": true,
    "credentialsNonExpired": true,
    "enabled": true
  },
  "credentials": null,
  "name": "user"
}
```

GrantedAuthority

The `GrantedAuthority` objects are inserted into the Authentication object by the `AuthenticationManager` and are later read by `AccessDecisionManager` instances when making authorization decisions.

- An authority (in its simplest form) is just a string, it can be anything like: *user*, *ADMIN*, *ROLE_ADMIN* or *53cr37_r0l3*.
- A role is an authority with a *ROLE_* prefix, so a role called *ADMIN* is the same as an authority called *ROLE_ADMIN*.

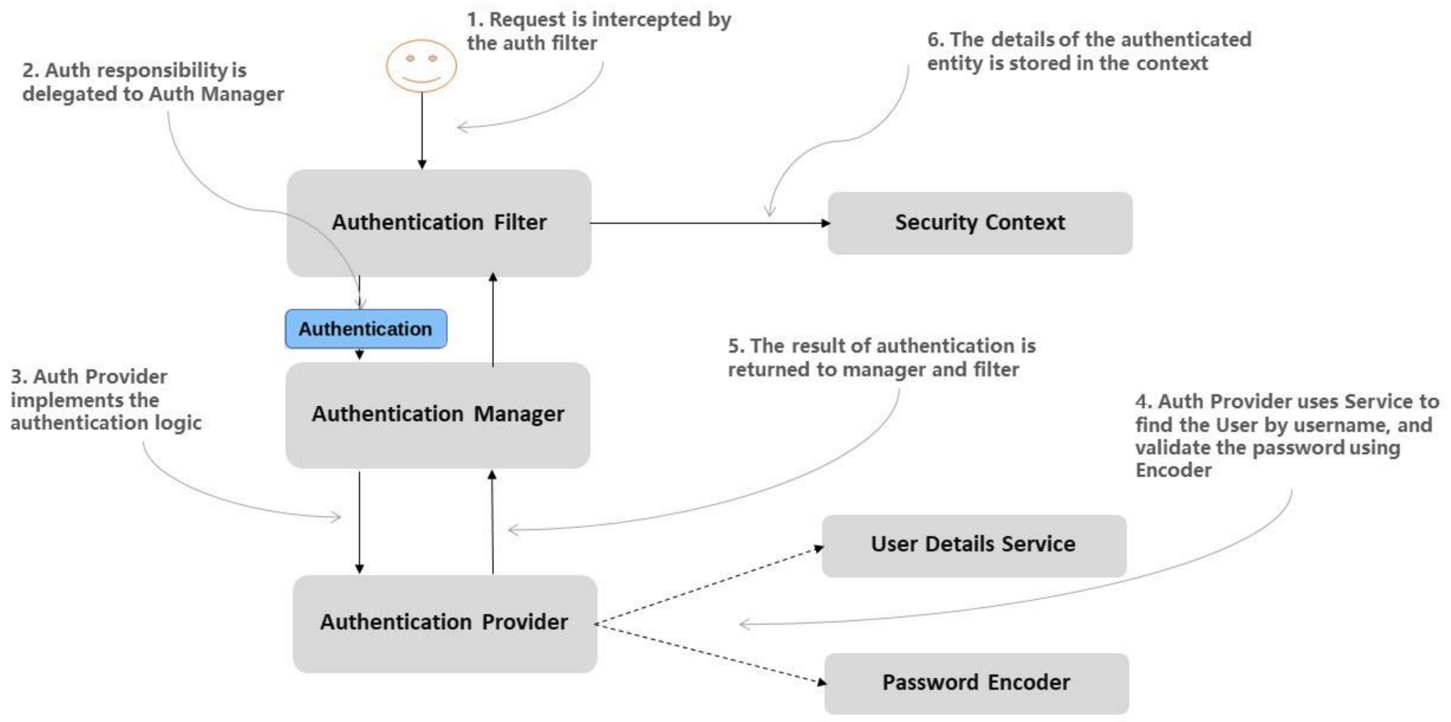
application.properties

```
spring.security.user.name=user
spring.security.user.password=12345678
spring.security.user.roles=ADMIN,USER
```

```
{
    "authorities": [
        {
            "authority": "ROLE_ADMIN"
        },
        {
            "authority": "ROLE_USER"
        }
    ],
    "details": {
        "remoteAddress": "0:0:0:0:0:0:0:1",
        "sessionId": "2008653CAE25911B71D493B3AA0B5877"
    },
    "authenticated": true,
    "principal": {
        "password": null,
        "username": "user",
        "authorities": [
            {
                "authority": "ROLE_ADMIN"
            },
            {
                "authority": "ROLE_USER"
            }
        ],
        "accountNonExpired": true,
        "accountNonLocked": true,
        "credentialsNonExpired": true,
        "enabled": true
    },
    "credentials": null,
    "name": "user"
}
```

Authentication Flow

Spring Security provides comprehensive support for different kinds of authentication mechanisms such as Username/password, OAuth2, SAML 2, JAAS, SiteMinder etc.


1. Request is intercepted by the auth filter
2. Auth responsibility is delegated to Auth Manager
6. The details of the authenticated entity is stored in the context
Authentication Filter
Security Context
Authentication
Authentication Manager
3. Auth Provider implements the authentication logic
5. The result of authentication is returned to manager and filter
4. Auth Provider uses Service to find the User by username, and validate the password using Encoder
User Details Service
Authentication Provider
Password Encoder

Authentication

`AuthenticationManager`: `AuthenticationManager` is the API that defines how Spring Security's Filters perform authentication.

`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.

`AuthenticationProvider`: You can inject multiple `AuthenticationProviders` 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.

UserDetailsService

An interface in Spring Security that defines a method for retrieving user details based on a username.

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

The default implementation of `UserDetailsService` only registers the default credentials in the memory of the application. These default credentials are "user" with a default password that's a randomly generated universally unique identifier (UUID) written to the application console when the spring context loads.

Note that `UserDetailsService` is always associated with a `PasswordEncoder` that encodes a supplied password and verifies if the password matches an existing encoding. When we replace the default implementation of the `UserDetailsService`, we must also specify a `PasswordEncoder`.

```
Using generated security password: c281a0ae-dfe4-4353-ae8c-301a9fa2fe64
```

PasswordEncoder

- The primary purpose of `PasswordEncoder` is to match the user-supplied password with the password stored in `UserDetails` object stored in the `SecurityContext`.
- Perform only a one-way transformation of a password to allow it to be stored securely.
- Cannot decode an encoded password using the PasswordEncoder interface.

```
public interface PasswordEncoder {
    String encode(CharSequence rawPassword);
    boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

Built-in PasswordEncoder PasswordEncoderFactories

- **noop** – NoOpPasswordEncoder : doesn't encode the password but keeps it in cleartext. It can be used for unit testing only.
- **sha256** – StandardPasswordEncoder uses SHA-256 to hash the password that is not strong enough anymore. Its deprecated now.
- **scrypt** – SCryptPasswordEncoder : uses a scrypt hashing function to encode the password.
- **bcrypt** – BCryptPasswordEncoder : uses a bcrypt hashing function to encode the password.
- **ldap** – LdapShaPasswordEncoder : legacy purposes only and is not considered secure. It supports LDAP SHA and SSHA (salted-SHA) encodings.
- **pbkdf2** – Pbkdf2PasswordEncoder : uses PBKDF2 invoked on the concatenated bytes of the salt, secret and password. The default is based upon aiming for .5 seconds to validate the password when this class was added. Users should tune password verification to their own systems.
- **argon2** – Argon2PasswordEncoder : uses the Argon2 hashing function. It can accept the length of the salt, length of generated hash, a CPU cost parameter, a memory cost parameter and a parallelization parameter.
- **MD4** – Md4PasswordEncoder : is deprecated for not being secured.
- **MD5**, **SHA-1**, **SHA-256** – MessageDigestPasswordEncoder : is deprecated because digest-based password encoding is not considered secure.

Built-in In-memory UserDetailsService - InMemoryUserDetailsManager

The first very basic example of overriding the UserDetailsService is InMemoryUserDetailsManager. This class stores credentials in the memory, which can then be used by Spring Security to authenticate an incoming request.

A UserDetailsManager extends the UserDetailsService contract.

```
@Configuration
public class SecurityConfigInMemory {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        …
   }


   @Bean
   public UserDetailsService userDetailsService() {
      UserDetails user1 = User.builder()
            .username("user1")
            .password(passwordEncoder().encode("1234"))
            .roles("USER")
            .build();
      UserDetails user2 = User.builder()
            .username("admin")
            .password(passwordEncoder().encode("1234"))
            .roles("ADMIN")
            .build();
      return new InMemoryUserDetailsManager(user1, user2);
   }

   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }
}
```

Built-in JDBC UserDetailsService - JdbcUserDetailsManager

The default schema is also exposed as a classpath resource named `org/springframework/security/core/userdetails/jdbc/users.ddl`.

When using JdbcUserDetailsManager in Spring Security, it doesn't create the necessary tables automatically. Instead, you need to set up the database schema yourself.

```
create table users(
                    username varchar(50) not null primary key,
                    password varchar(500) not null,
                    enabled boolean not null
);

create table authorities (
                          username varchar(50) not null,
                          authority varchar(50) not null,
                          constraint fk_authorities_users foreign key(username) references users(username)
);
create unique index ix_auth_username on authorities (username,authority);
```

Built-in JDBC UserDetailsService - JdbcUserDetailsManager

Here, we use `JdbcUserDetailsManager` generates two users to save into DB.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.url=jdbc:mysql://localhost:3306/springsecurity
```

By default, Spring Security queries user details, including authorities, with the following SQL statements:

```
SELECT username, password, enabled FROM users
WHERE username = ?

SELECT username, authority FROM authorities WHERE
username = ?
```

```
@Configuration
public class SecurityConfigJDBC {

   @Autowired
   private DataSource dataSource;

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      …
   }

   @Bean
   public UserDetailsService userDetailsService() {
      UserDetails user1 = User.builder()
            .username("user")
            .password(passwordEncoder().encode("1234"))
            .roles("USER")
            .build();
      UserDetails user2 = User.builder()
            .username("admin")
            .password(passwordEncoder().encode("1234"))
            .roles("ADMIN")
            .build();
     var users = new JdbcUserDetailsManager(dataSource);
     users.createUser(user1);
     users.createUser(user2);
     return users;
   }

   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }
}
```

Built-in JDBC UserDetailsService - JdbcUserDetailsManager

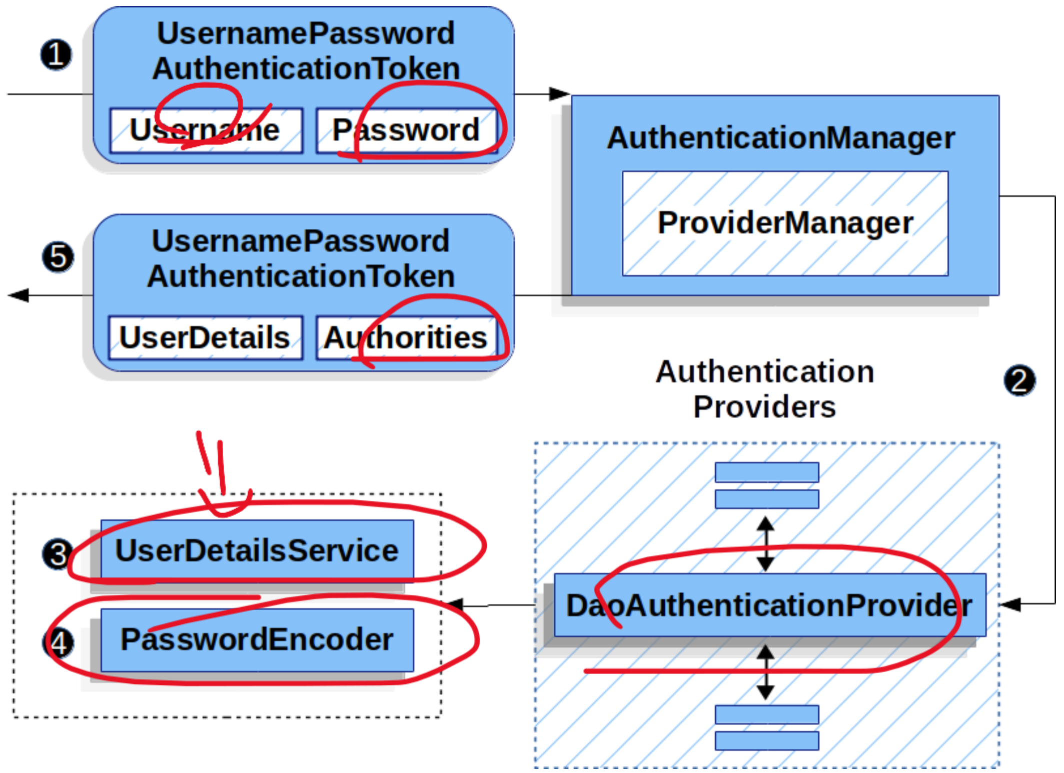
We can also write custom SQL queries to fetch the user and authorities' details if we are using a custom DDL schema that uses different table or column names.

NOTE: didn't list `SecurityFilterChain` and `PasswordEncoder` beans in he code below:

```
@Configuration
public class SecurityConfigJDBCCustomQuery {

      @Bean
   public UserDetailsService jdbcUserDetailsService(DataSource dataSource) {
      String usersByUsernameQuery = "select username, password, enabled from tbl_users where username = ?";
      String authsByUserQuery = "select username, authority from tbl_authorities where username = ?";

      JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
      users.setUsersByUsernameQuery(usersByUsernameQuery);
      users.setAuthoritiesByUsernameQuery(authsByUserQuery);

      return users;
   }

}
```

Customized AuthenticationProvider - UserDetailsService

- DaoAuthenticationProvider is an AuthenticationProvider implementation that uses a UserDetailsService and PasswordEncoder to authenticate a username and password.
- UserDetails is returned by the UserDetailsService. The DaoAuthenticationProvider validates the UserDetails and then returns an Authentication that has a principal that is the UserDetails returned by the configured UserDetailsService.



Customized AuthenticationProvider - UserDetailsService Demo

```java
@Entity
@Data
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "firstname")
    private String firstName;

    @Column(name = "lastname")
    private String lastName;

    private String email;
    private String password;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "roles", joinColumns = @JoinColumn(name = "user_id"))
    @Column(name = "role")
    private List<String> roles;

}
```

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
}
```

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

    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        Optional<User> userOptional = userRepository.findByEmail(email);
        if (userOptional.isPresent()) {
            User user = userOptional.get();
            List<SimpleGrantedAuthority> authorities = user.getRoles().stream()
                    .map(str -> new SimpleGrantedAuthority(str))
                    .collect(Collectors.toList());
            UserDetails userDetails = org.springframework.security.core.userdetails.User.builder()
                    .username(user.getEmail())
                    .password(user.getPassword())
                    .authorities(authorities)
                    .build();

            return userDetails;
        } else {
            throw new UsernameNotFoundException(email);
        }
    }
}
```

Customized AuthenticationProvider - UserDetailsService Demo (Cont.)

```
@Configuration
@RequiredArgsConstructor
public class SecurityConfigCustomUserDetailService {

    private final UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService);
        authenticationProvider.setPasswordEncoder(passwordEncoder());
        return authenticationProvider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

}
```

Spring Security 6 Authorization

MISS XING

Authorities

Authentication discusses how all Authentication implementations store a list of `GrantedAuthority` objects.

These represent the authorities that have been granted to the principal. The `GrantedAuthority` objects are inserted into the `Authentication` object by the `AuthenticationManager` and are later read by `AccessDecisionManager` instances when making authorization decisions.

Authorities - GrantedAuthority

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`.

Spring Security includes one concrete `GrantedAuthority` implementation: `SimpleGrantedAuthority`. This implementation lets any user-specified `String` be converted into a `GrantedAuthority`.

AuthorizationManager

`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:

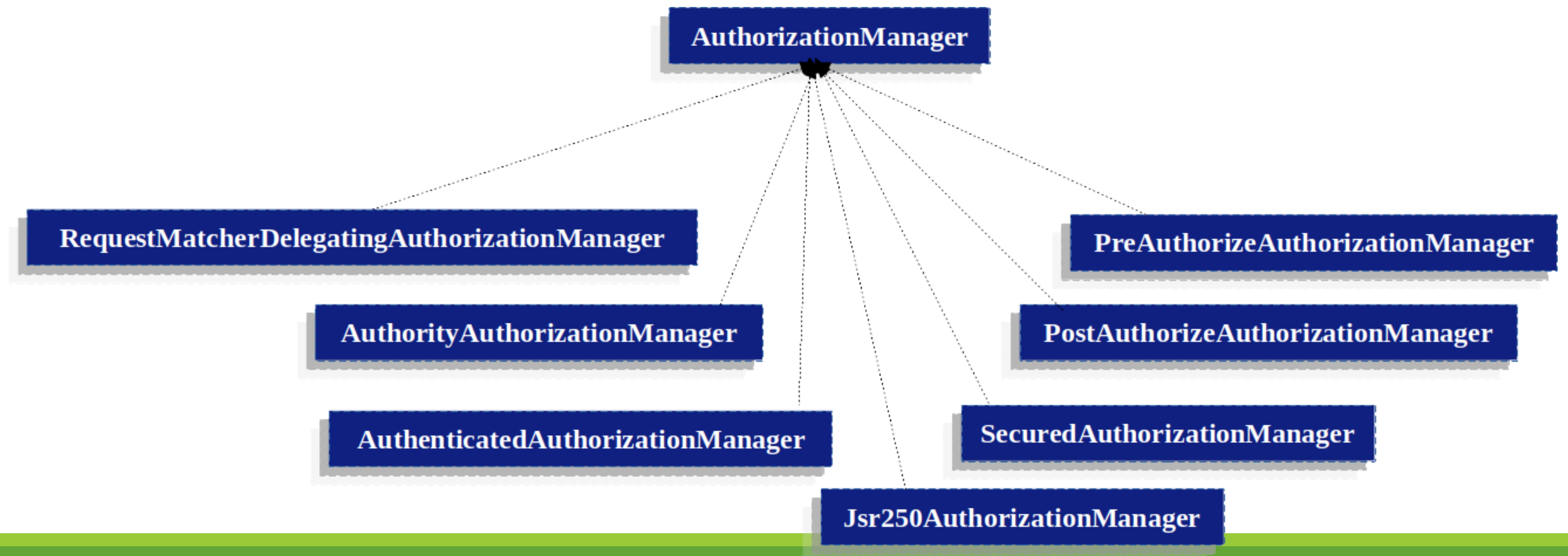
```
@FunctionalInterface
public interface AuthorizationManager<T> {
    default void verify(Supplier<Authentication> authentication, T object) {
        AuthorizationDecision decision = this.check(authentication, object);
        if (decision != null && !decision.isGranted()) {
            throw new AccessDeniedException("Access Denied");
        }
    }

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

Delegate-based AuthorizationManager Implementations

Whilst users can implement their own `AuthorizationManager` to control all aspects of authorization, Spring Security ships with a delegating `AuthorizationManager` that can collaborate with individual `AuthorizationManagers`.

`RequestMatcherDelegatingAuthorizationManager` will match the request with the most appropriate delegate `AuthorizationManager`. For method security, you can use `AuthorizationManagerBeforeMethodInterceptor` and `AuthorizationManagerAfterMethodInterceptor`.

Authorization Manager Implementations illustrates the relevant classes.


AuthorizationManager
RequestMatcherDelegatingAuthorizationManager
PreAuthorizeAuthorizationManager
AuthorityAuthorizationManager
PostAuthorizeAuthorizationManager
AuthenticatedAuthorizationManager
SecuredAuthorizationManager
Jsr250AuthorizationManager

Authorize HttpServletRequests

Spring Security allows you to model your authorization at the request level. For example, with Spring Security you can say that all pages under `/admin` require one authority while all other pages simply require authentication.

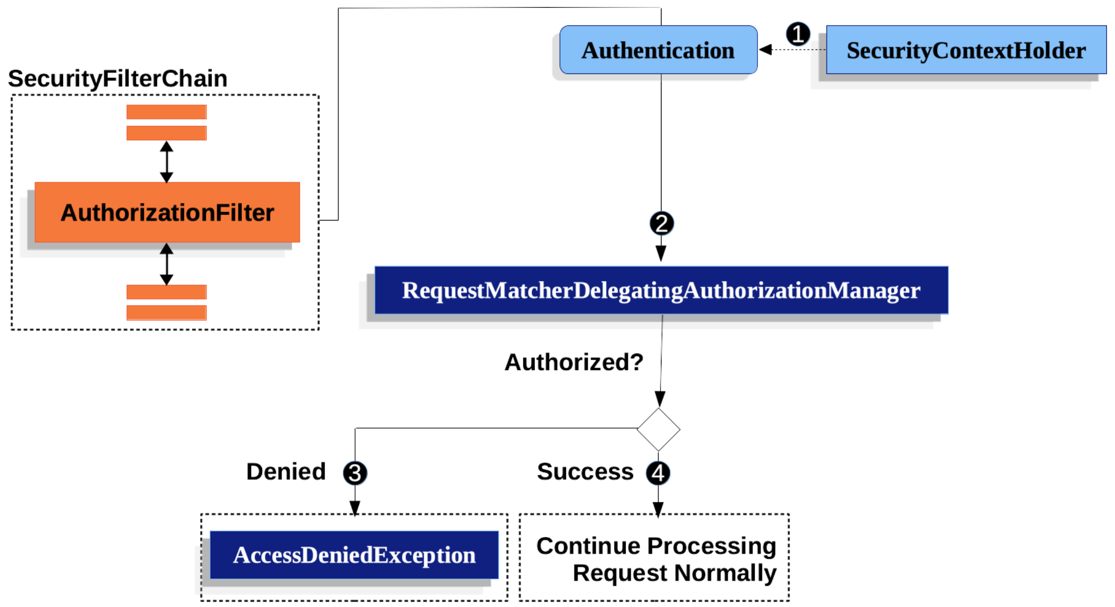
By default, Spring Security requires that every request be authenticated. That said, any time you use an `HttpSecurity` instance, it's necessary to declare your authorization rules.

Whenever you have an `HttpSecurity` instance, you can do:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
            .formLogin(Customizer.withDefaults()) //UsernamePasswordAuthenticationFilter
            .httpBasic(Customizer.withDefaults()) //BasicAuthenticationFilter
            .addFilterBefore(new TenantFilter(), AuthorizationFilter.class)
            .authorizeHttpRequests(authorize -> authorize.anyRequest().authenticated())//AuthorizationFilter
            .build();
}
```

How Request Authorization Components Work

1. First, the `AuthorizationFilter` constructs a `Supplier` that retrieves an Authentication from the `SecurityContextHolder`.
2. Second, it passes the `Supplier<Authentication>` and the `HttpServletRequest` to the `AuthorizationManager`. The `AuthorizationManager` matches the request to the patterns in `authorizeHttpRequests`, and runs the corresponding rule.
   1. If authorization is denied, an `AuthorizationDeniedEvent` is published, and an `AccessDeniedException` is thrown. In this case the `ExceptionTranslationFilter` handles the `AccessDeniedException`.
   2. If access is granted, an `AuthorizationGrantedEvent` is published and `AuthorizationFilter` continues with the `FilterChain` which allows the application to process normally.



Authorizing an Endpoint

You can configure Spring Security to have different rules by adding more rules in order of precedence.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
    http.formLogin(Customizer.withDefaults())
            .authorizeHttpRequests(req -> req.requestMatchers("/users").hasAuthority(“admin")
                    .anyRequest().authenticated());
    return http.build();
}
```

`AuthorizationFilter` processes these pairs in the order listed, applying only the first match to the request. This means that even though **`/**`** would also match for **`/users`** the above rules are not a problem. The way to read the above rules is "if the request is **`/users`**, then require the **`ADMIN`** authority; else, only require authentication".

Matching Requests

Spring Security supports two languages for URI pattern-matching: `Ant` (as seen above) and `Regular Expressions`.

- Matching Using Ant
  - Ant is the default language that Spring Security uses to match requests.

```
http
   .authorizeHttpRequests((authorize) -> authorize
      .requestMatchers("/resource/**").hasRole("USER")
      .anyRequest().authenticated()
   )
```

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

Matching Using Regular Expressions

Spring Security supports matching requests against a regular expression. This can come in handy if you want to apply more strict matching criteria than ****** on a subdirectory.

For example, consider a path that contains the username and the rule that all usernames must be alphanumeric. You can use RegexRequestMatcher to respect this rule, like so:

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

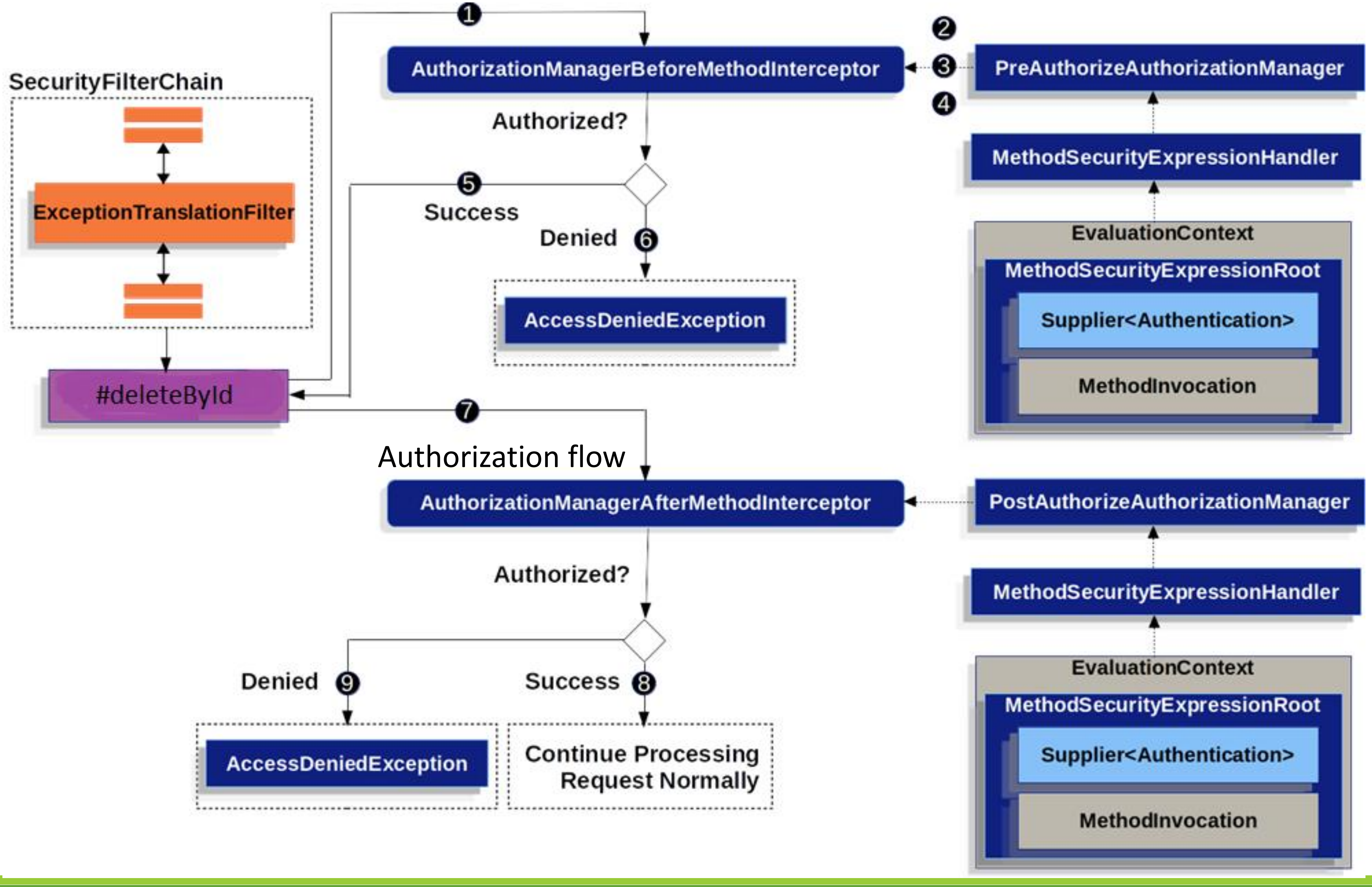
Method Security

In addition to modeling authorization at the request level, Spring Security also supports modeling at the method level.

You can activate it in your application by annotating any `@Configuration` class with `@EnableMethodSecurity`.

Then, you are immediately able to annotate any Spring-managed class or method with `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, and `@PostFilter` to authorize method invocations, including the input parameters and return values.

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService{

    private final UserRepository userRepository;

    @PreAuthorize("hasAuthority('ADMIN')")
    @Override
    public void deleteById(Long id) {
        userRepository.deleteById(id);
    }
}
```


SecurityFilterChain
ExceptionTranslationFilter
#deleteById
1
AuthorizationManagerBeforeMethodInterceptor
2
3
4
PreAuthorizeAuthorizationManager
MethodSecurityExpressionHandler
EvaluationContext
MethodSecurityExpressionRoot
Supplier<Authentication>
MethodInvocation
Authorized?
5
Success
Denied
6
AccessDeniedException
7
Authorization flow
AuthorizationManagerAfterMethodInterceptor
PostAuthorizeAuthorizationManager
MethodSecurityExpressionHandler
EvaluationContext
MethodSecurityExpressionRoot
Supplier<Authentication>
MethodInvocation
Authorized?
Denied
9
Success
8
AccessDeniedException
Continue Processing Request Normally

Authorization flow

Authorization flow

1. Spring AOP invokes its proxy method for deleteById. Among the proxy's other advisors, it invokes an AuthorizationManagerBeforeMethodInterceptor that matches the @PreAuthorize pointcut

2. The interceptor invokes PreAuthorizeAuthorizationManager#check

3. The authorization manager uses a MethodSecurityExpressionHandler to parse the annotation's SpEL expression and constructs a corresponding EvaluationContext from a MethodSecurityExpressionRoot containing a Supplier<Authentication> and MethodInvocation.

4. The interceptor uses this context to evaluate the expression; specifically, it reads the Authentication from the Supplier and checks whether it has permission:read in its collection of authorities

5. If the evaluation passes, then Spring AOP proceeds to invoke the method.

6. If not, the interceptor publishes an AuthorizationDeniedEvent and throws an AccessDeniedException which the ExceptionTranslationFilter catches and returns a 403 status code to the response

7. After the method returns, Spring AOP invokes an AuthorizationManagerAfterMethodInterceptor that matches the @PostAuthorize pointcut, operating the same as above, but with PostAuthorizeAuthorizationManager

8. If the evaluation passes (in this case, the return value belongs to the logged-in user), processing continues normally

9. If not, the interceptor publishes an AuthorizationDeniedEvent and throws an AccessDeniedException, which the ExceptionTranslationFilter catches and returns a 403 status code to the response

Reference

https://docs.spring.io/spring-security/reference/servlet/architecture.html

https://www.marcobehler.com/guides/spring-security

https://howtodoinjava.com/spring-security/password-encoders/

https://howtodoinjava.com/spring-security/inmemory-jdbc-userdetails-service/

https://howtodoinjava.com/spring-security/spring-security-tutorial/