

Spring Security Architecture

In terms of Authentication, Authorization, Web Security, Customizations,
Method Security and Threads

17 June 2024

V1.0

Berk Ekim Paşmakoğlu

Authentication and Authorization

Authentication

- who are you?

Authorization

- what are you allowed to do?

Authentication Outline

- AuthenticationManager
- ProviderManager – I
- ProviderManager – II
- AuthenticationManagerBuilder
- Application that configures the **global** (parent) AuthenticationManager
- Application that configures the **local** AuthenticationManager

AuthenticationManager

<<interface>>

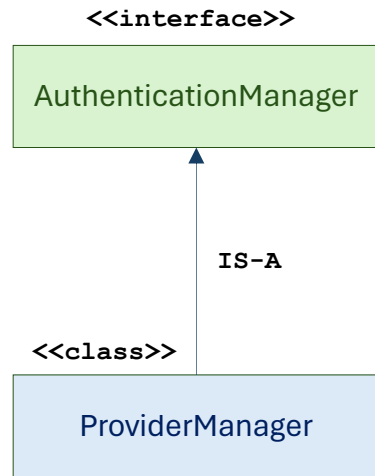
AuthenticationManager

```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication)  
    throws AuthenticationException;  
}
```

- Spring Boot provides a **default global** [AuthenticationManager](#) (with only one user) unless you pre-empt it by providing your own bean of type AuthenticationManager.

- Returns an [Authentication](#)
 - (normally with [authenticated=true](#)) if it can verify that the input represents a valid principal.
 - Throw an [AuthenticationException](#) if it believes that the input represents an invalid principal.
 - Return [null](#) if it cannot decide.
-
- If you do any configuration that builds an [AuthenticationManager](#), you can often ***do it locally to the resources that you are protecting*** and not worry about the **global default**.

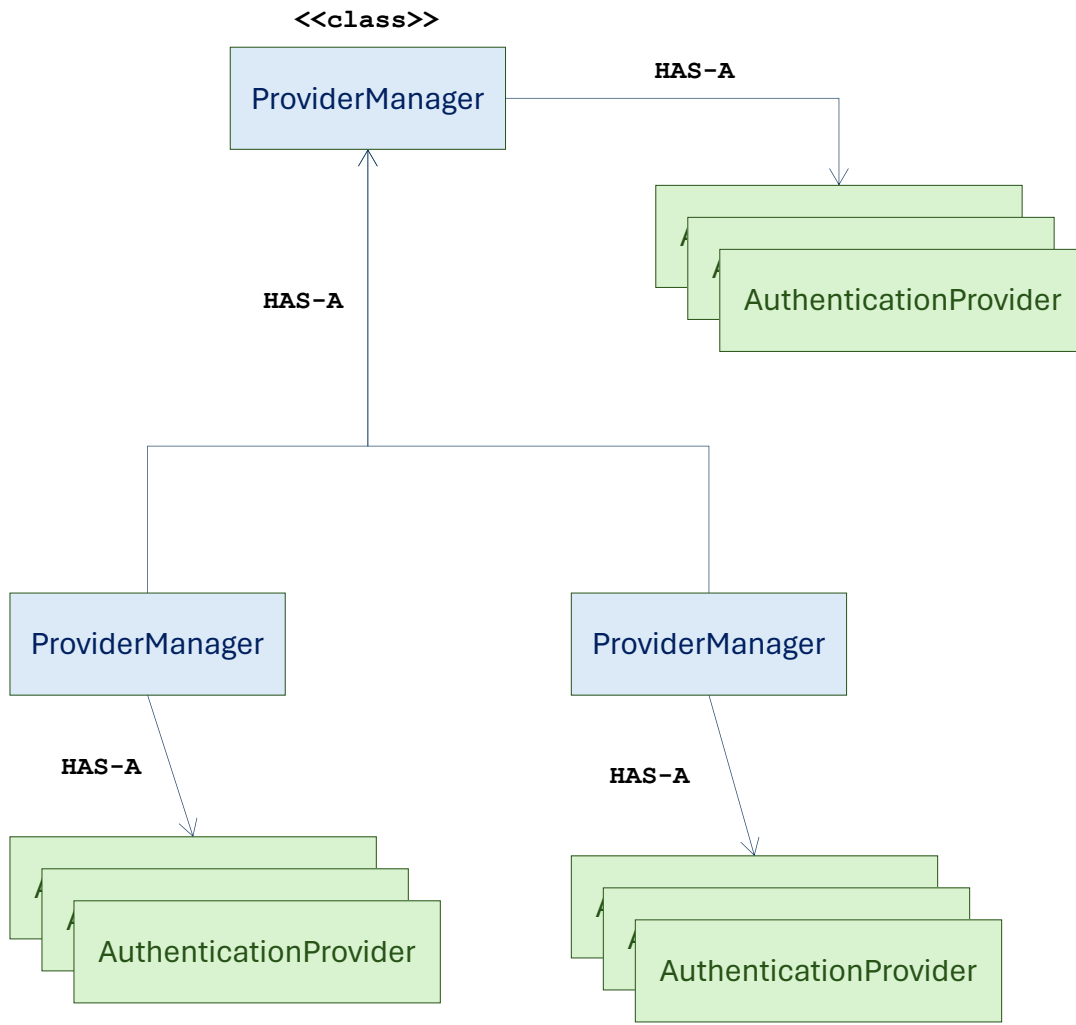
ProviderManager - I



- delegates to a chain of `AuthenticationProvider` instances.
 - can support multiple different authentication mechanisms in the same application by delegating.
 - If it does not recognize a particular `Authentication` instance type, it is skipped.
- has an extra method to allow the caller to query whether it supports a given `Authentication` type.
- The `Class<?>` argument in the `supports()` method is really `Class<? extends Authentication>`.

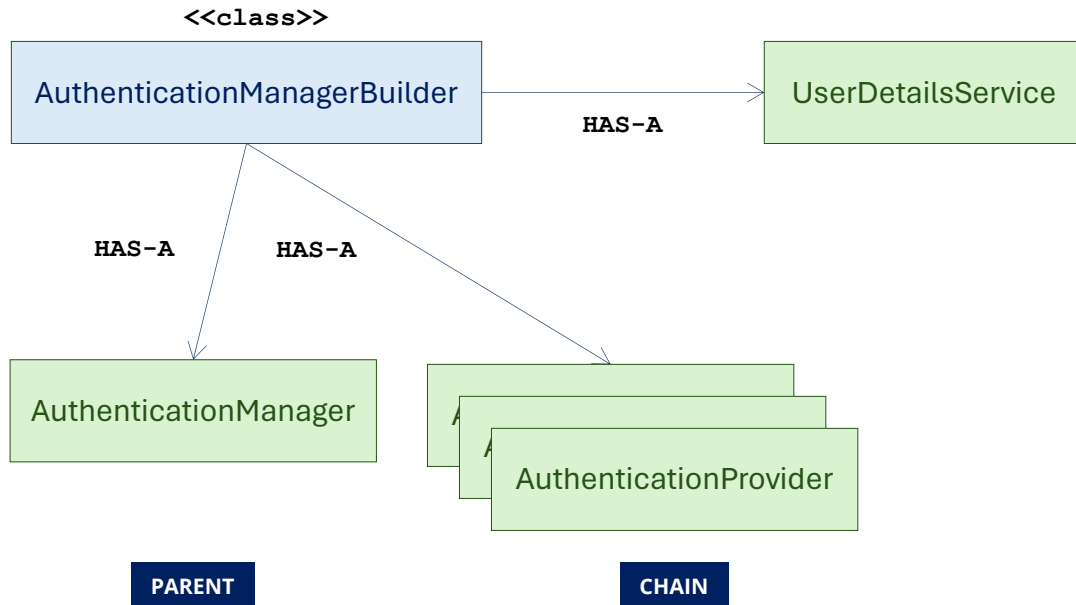
```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication) throws AuthenticationException;
    boolean supports(Class<?> authentication);
}
```

ProviderManager – II



- delegates to a chain of [AuthenticationProvider](#) instances.
 - can support multiple different authentication mechanisms in the same application by delegating.
 - If it does not recognize a particular [Authentication](#) instance type, it is skipped.
- has an optional parent, which it can consult if all providers return [null](#).
 - If the parent is not available, a [null Authentication](#) results in an [AuthenticationException](#).
- Sometimes, an application has logical groups of protected resources
 - i.e., all web resources that match a path pattern, such as [/api/**](#).
- Each group can have its own dedicated [AuthenticationManager](#).
- Often, each of those is a [ProviderManager](#), and they share a parent.
- The parent is then a kind of **global resource** acting as a fallback for all providers.

AuthenticationManagerBuilder



- Spring Security provides some [configuration helpers](#) to quickly get common [authentication manager features](#) set up in your application.
- [AuthenticationManagerBuilder](#)
 - most used helper
 - great for setting up [in-memory](#), [JDBC](#), or [LDAP user details](#) or for adding a custom [UserDetailsService](#).

Application that configures the **global** (parent) AuthenticationManager

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {
    ... // web stuff here

    @Autowired
    public void initialize(AuthenticationManagerBuilder builder, DataSource dataSource) {
        builder.jdbcAuthentication()
            .dataSource(dataSource)
            .withUser("dave")
            .password("secret")
            .roles("USER");
    }
}
```

- `AuthenticationManagerBuilder` is injected (or auto-wired) into a method in a `@Bean`.
- This method body builds the **global** (parent) `AuthenticationManager`.

Application that configures the **local** AuthenticationManager

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {
    @Autowired
    DataSource dataSource;

    ... // web stuff here

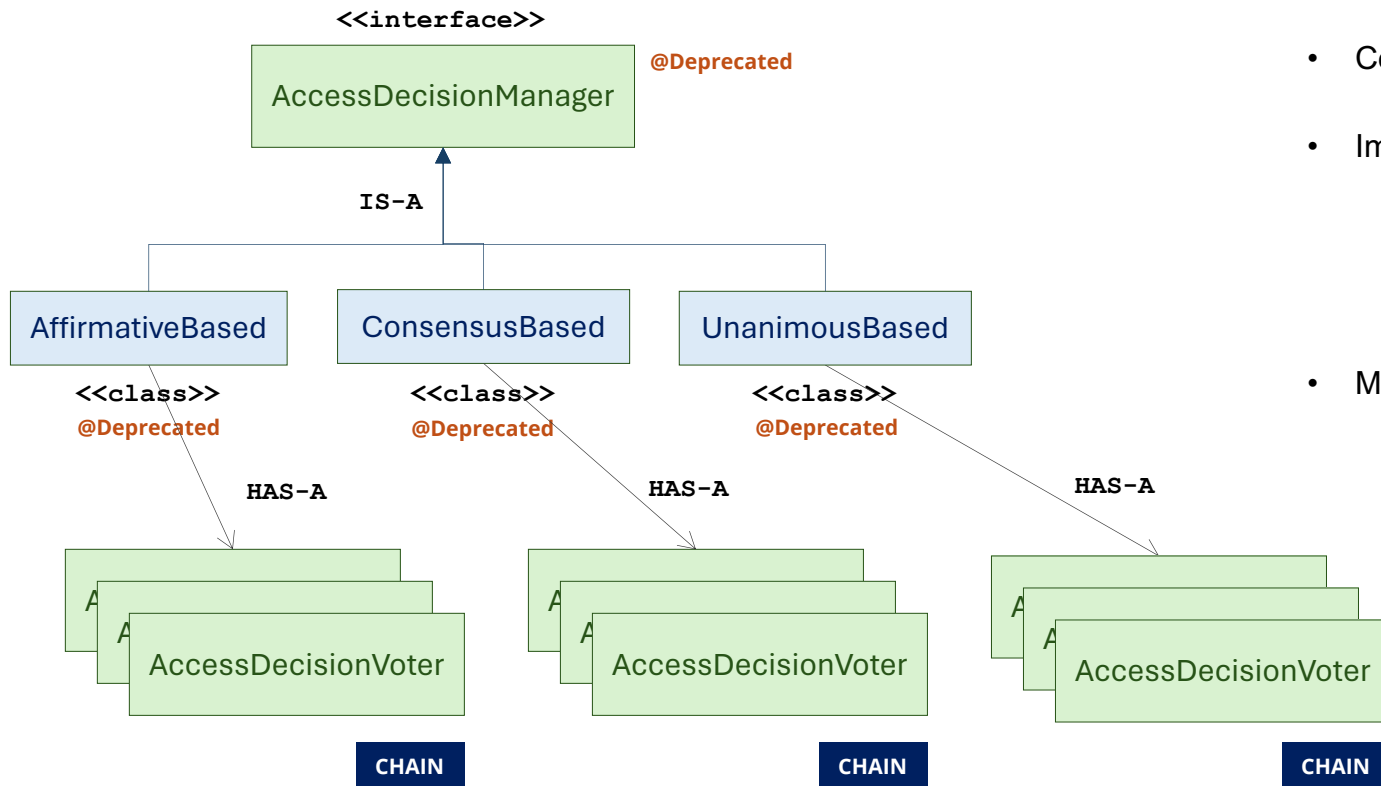
    @Override
    public void configure(AuthenticationManagerBuilder builder) {
        builder.jdbcAuthentication()
            .dataSource(dataSource)
            .withUser("dave")
            .password("secret")
            .roles("USER");
    }
}
```

- We overrode the `configure()` method in the configurer.
- `AuthenticationManagerBuilder` builds a **local** `AuthenticationManager` which would be a **child** of the **global** one.

Authorization Outline

- `AccessDecisionManager`
- `AccessDecisionVoter`
- `ConfigAttribute`

AccessDecisionManager



- Core strategy
- Implementations
 - `AffirmativeBased`, `ConsensusBased` and `UnanimousBased`.
 - All three delegate to a chain of `AccessDecisionVoter` instances
- Most people use `AffirmativeBased` as **default** `AccessDecisionManager`
 - access is granted if any voters return affirmatively

```
void decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes)
    throws AccessDeniedException, InsufficientAuthenticationException;

boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);
```

AccessDecisionVoter

<<interface>>

AccessDecisionVoter

@Deprecated

```
int vote(Authentication authentication, S object, Collection<ConfigAttribute> attributes);

boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);
```

- considers an [Authentication](#) (representing a principal) and a **secure** [Object](#), which has been decorated with [ConfigAttributes](#).

- [Object](#) is completely generic in the signatures of [AccessDecisionManager](#) and [AccessDecisionVoter](#). It represents anything that a user might want to access:
 - i.e., a [web resource](#) or a [method in a Java class](#).
- [ConfigAttributes](#) represent a decoration of the **secure** [Object](#) with some metadata that determines the *level of permission required to access it*.

ConfigAttribute

<<interface>>

ConfigAttribute

```
String getAttribute();
```

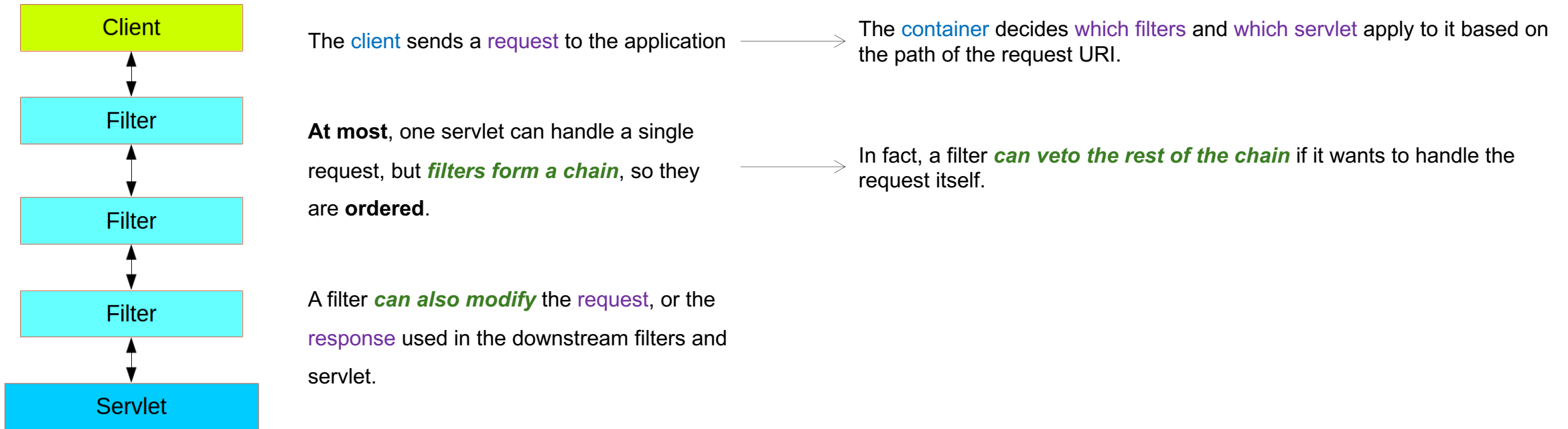
- Has only one API, `getAttribute()` which returns `String`.
- These strings encode the **intention of the owner of the resource**, expressing rules about who is allowed to access it:
 - ❖ A typical example is the name of a user role
 - i.e., `ROLE_ADMIN` or `ROLE_AUDIT`
 - ❖ represent **expressions** that need to be evaluated
 - `Spring Expression Language` (SpEL) expressions
 - `isFullyAuthenticated() && hasRole('user')`

Servlet Filters and Chains Outline

- Servlet Filters
- Order of the Filter Chain - I
- Order of the Filter Chain – II
- FilterChainProxy – I
- FilterChainProxy – II
- FilterChainProxy – III
- DelegatingFilterProxy
- SecurityFilterChain

Servlet Filters

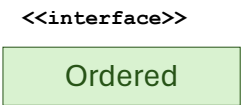
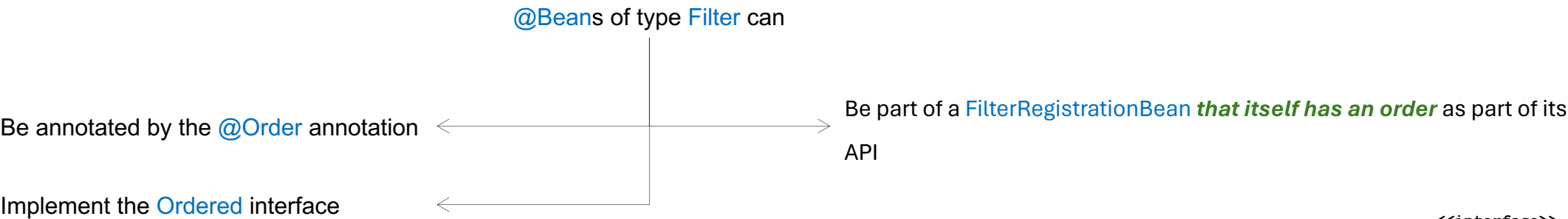
Spring Security in the web tier is based on Servlet **Filters**.



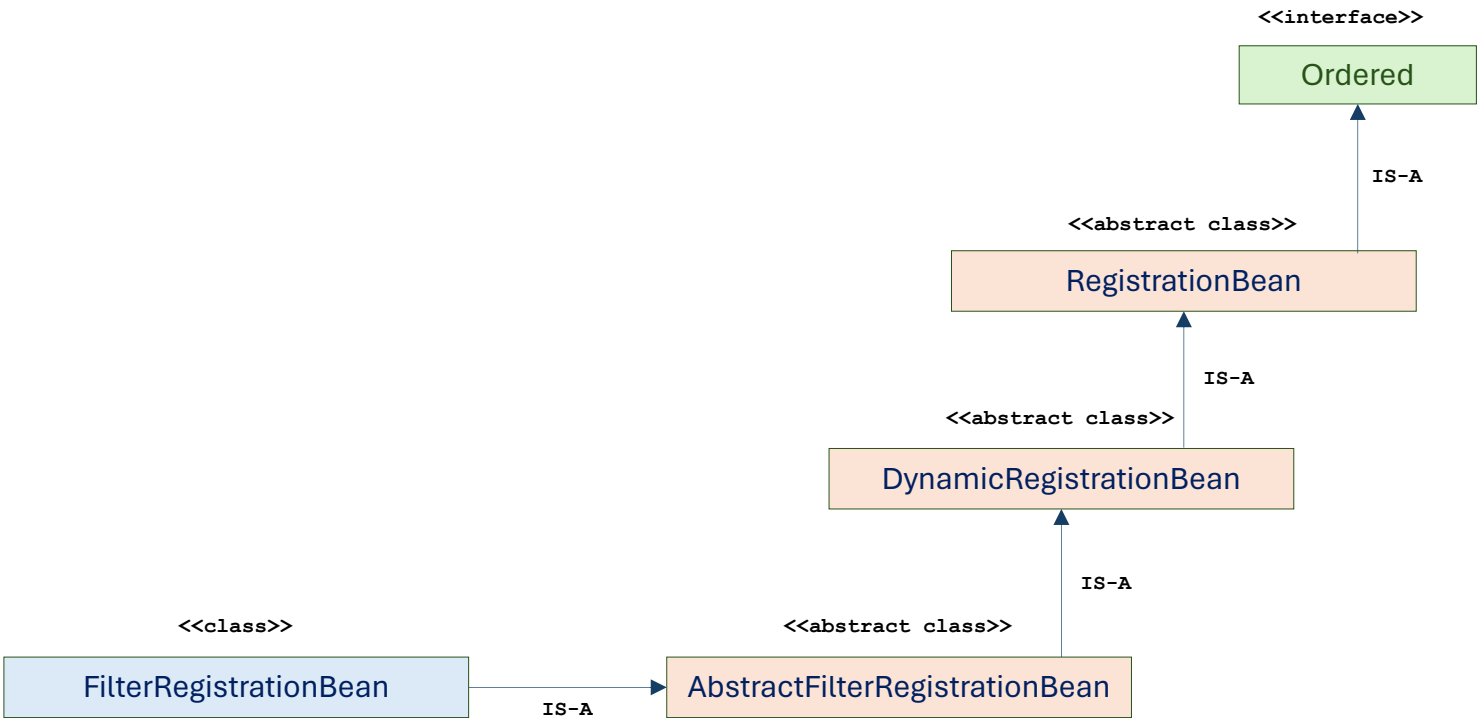
Typical layering of the handlers
for a **single HTTP request**.

Order of the Filter Chain - I

- The **order** of the **filter chain** is **very important**, and Spring Boot manages it through two mechanisms:



```
public interface Ordered {  
    int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;  
    int LOWEST_PRECEDENCE = Integer.MAX_VALUE;  
  
    int getOrder();  
}
```



Order of the Filter Chain - II

- Some **off-the-shelf** filters define their own constants to help signal what order they like to be in relative to each other:
 - `SessionRepositoryFilter` has a `DEFAULT_ORDER` of `Integer.MIN_VALUE + 50`,
 - which tells us it likes to be early in the chain, but it does not rule out other filters coming before it.

<<class>>

SessionRepositoryFilter

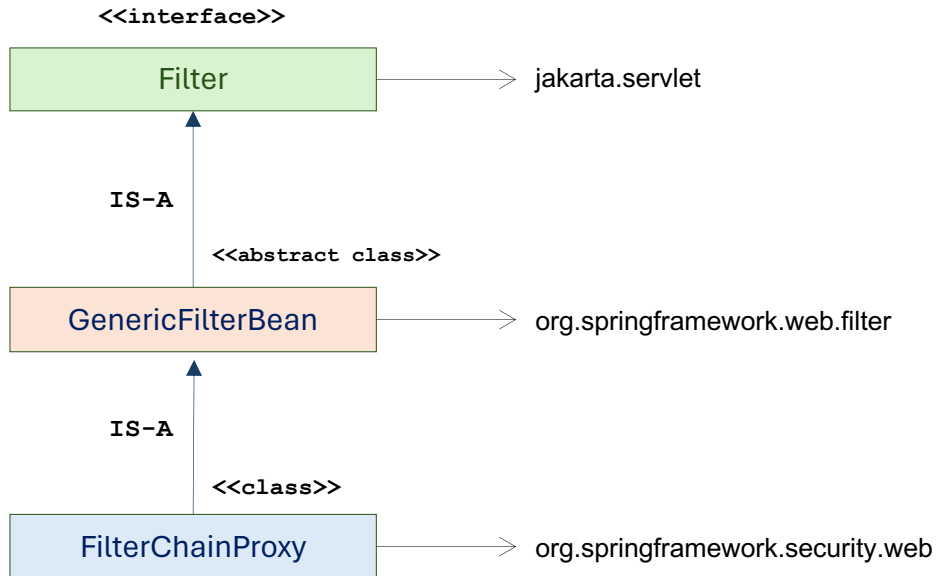
```
@Order(SessionRepositoryFilter.DEFAULT_ORDER)
public class SessionRepositoryFilter<S extends Session> extends OncePerRequestFilter {
    . . . // intentionally skipped

    public static final int DEFAULT_ORDER = Integer.MIN_VALUE + 50;

    . . . // intentionally skipped
}
```

FilterChainProxy - I

- Spring Security is installed as a single [Filter](#) in the chain, and its concrete type is [FilterChainProxy](#).



```
public interface Filter {
    default void init(FilterConfig filterConfig) throws ServletException {
    }

    void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException;

    default void destroy() {
    }
}
```

```
package jakarta.servlet;

import java.io.IOException;

public interface FilterChain {
    void doFilter(ServletRequest request, ServletResponse response)
    throws IOException, ServletException;
}
```

FilterChainProxy - II

- In a Spring Boot application, the security filter is a `@Bean` in the `ApplicationContext`, and it is *installed by default* so that it is applied to every request.
 - It is **installed at a position** defined by `SecurityProperties.DEFAULT_FILTER_ORDER` which in turn is anchored by `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER`.
 - `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER` is the maximum order that a Spring Boot application expects filters to have if they wrap the request, modifying its behavior.

```
package org.springframework.boot.web.servlet.filter;

public interface OrderedFilter extends Filter, Ordered {
    int REQUEST_WRAPPER_FILTER_MAX_ORDER = 0;
}
```

```
package org.springframework.boot.autoconfigure.security;

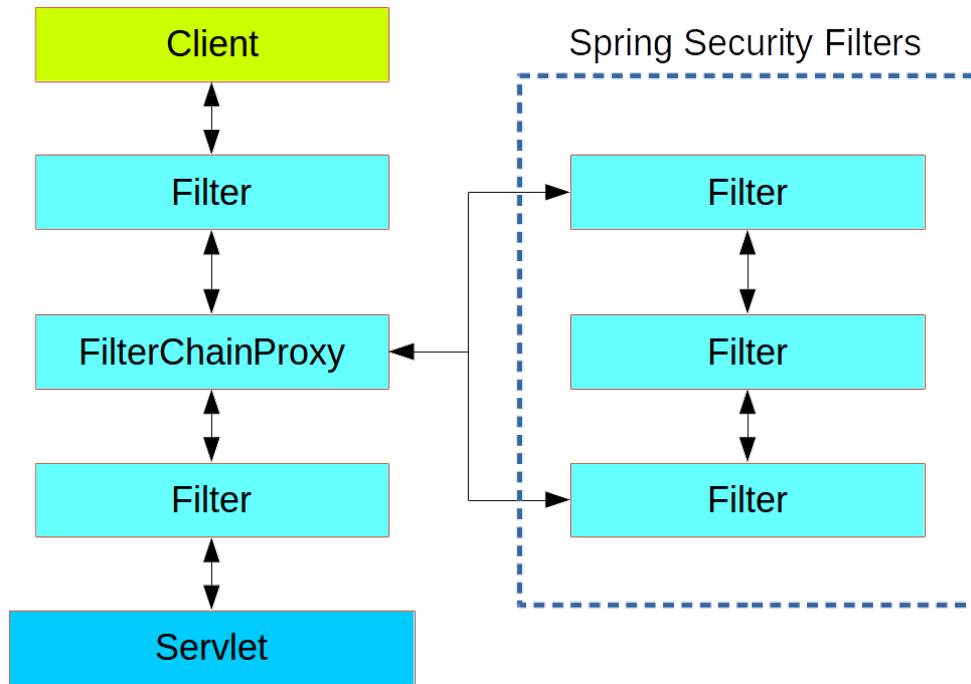
public class SecurityProperties {
    . . . // intentionally skipped

    public static final int DEFAULT_FILTER_ORDER = OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER -100;

    . . . // intentionally skipped
}
```

FilterChainProxy - III

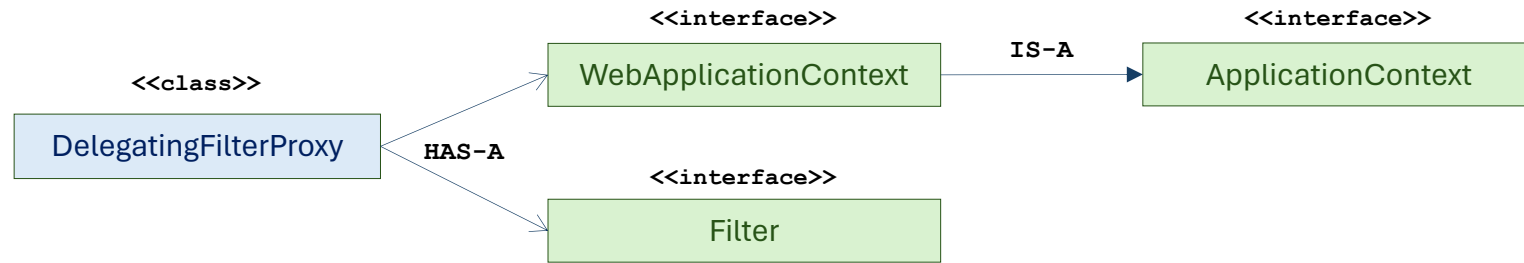
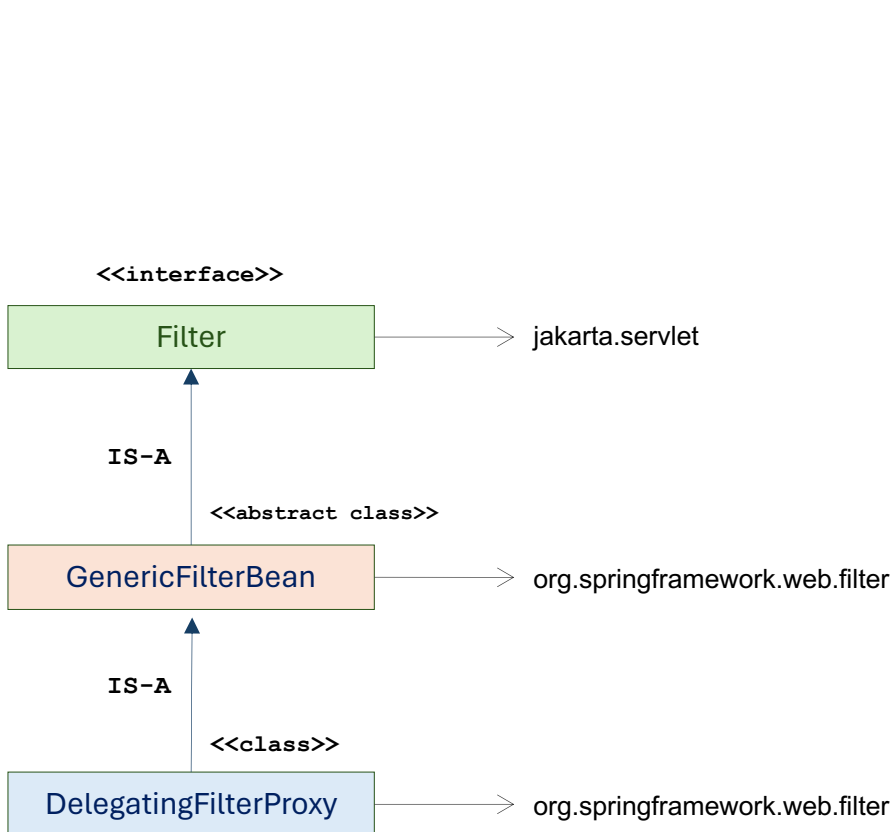
- From the point of view of the container, **Spring Security** is a single physical **Filter**, but, **inside of it**, there are additional filters, each playing a special role.
 - delegates **processing** to a **chain of internal filters**.



- It is the **FilterChainProxy** that **contains all the security logic** arranged internally as a **chain** (or chains) **of filters**.
 - All the filters have the **same API** since they all implement the **Filter** interface.
 - They all have the opportunity to **veto** the rest of the chain.

DelegatingFilterProxy

- There is even one more **layer of indirection** in the security filter
 - usually installed in the container as a [DelegatingFilterProxy](#), which **does not have** to be a Spring [@Bean](#).
 - delegates to a [FilterChainProxy](#), which is **always** a [@Bean](#), usually with a fixed name of [springSecurityFilterChain](#).



```
package org.springframework.boot.autoconfigure.security;

public class DelegatingFilterProxy extends GenericFilterBean {
    . . . // intentionally skipped

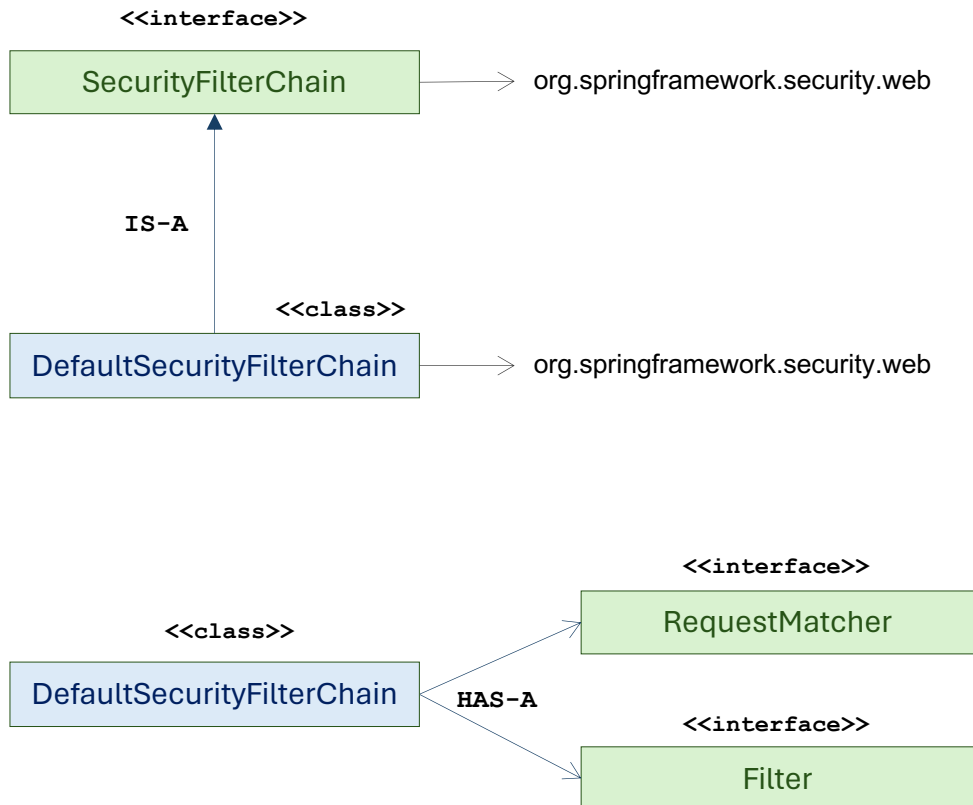
    private WebApplicationContext webApplicationContext;

    private volatile Filter delegate;

    . . . // intentionally skipped
}
```

SecurityFilterChain

- Defines a **filter chain** which is capable of being matched against an [HttpServletRequest](#) to decide whether it applies to that request.
- Used to configure a [FilterChainProxy](#).



```
package org.springframework.security.web;

public interface SecurityFilterChain {
    boolean matches(HttpServletRequest request);

    List<Filter> getFilters();
}
```

```
package org.springframework.security.web;

public interface DefaultSecurityFilterChain implements SecurityFilterChain {
    . . . // intentionally skipped

    private final RequestMatcher requestMatcher;

    private final List<Filter> filters;

    . . . // intentionally skipped
}
```

Web Security Outline

- WebSecurityConfiguration
- WebSecurity – I
- WebSecurity – II
- WebSecurity – III
- WebSecurityConfigurerAdapter
- WebSecurityConfigurer
- SecurityConfigurer

WebSecurityConfiguration

- Uses a [WebSecurity](#) to create the [FilterChainProxy](#) that performs the **web-based** security for Spring Security.
- **Customizations** can be made to WebSecurity
 - by implementing [WebSecurityConfigurer](#) and exposing it as a [Configuration](#) or
 - exposing a [WebSecurityCustomizer](#) bean.
- This configuration is imported when using [EnableWebSecurity](#).

```
Creates the Spring Security Filter Chain
Returns: the Filter that represents the security filter chain
Throws: Exception

@Bean(name = AbstractSecurityWebApplicationInitializer.DEFAULT_FILTER_NAME)
public Filter springSecurityFilterChain() throws Exception {
    boolean hasFilterChain = !this.securityFilterChains.isEmpty();
    if (!hasFilterChain) {
        this.webSecurity.addSecurityFilterChainBuilder(() -> {
            this.httpSecurity.authorizeHttpRequests((authorize) -> authorize.anyRequest().authenticated());
            this.httpSecurity.formLogin(Customizer.withDefaults());
            this.httpSecurity.httpBasic(Customizer.withDefaults());
            return this.httpSecurity.build();
        });
    }
    for (SecurityFilterChain securityFilterChain : this.securityFilterChains) {
        this.webSecurity.addSecurityFilterChainBuilder(() -> securityFilterChain);
    }
    for (WebSecurityCustomizer customizer : this.webSecurityCustomizers) {
        customizer.customize(this.webSecurity);
    }
    return this.webSecurity.build();
}
```

```
package org.springframework.security.config.annotation.web.configuration;

@Configuration(
    proxyBeanMethods = false
)
public interface WebSecurityConfiguration implements ImportAware, BeanClassLoaderAware {
    . . . // intentionally skipped

    private WebSecurity webSecurity;
    private HttpSecurity httpSecurity;

    private List<SecurityFilterChain> securityFilterChains = Collections.emptyList();

    . . . // intentionally skipped
}
```


WebSecurity - I

- The `WebSecurity` is created by `WebSecurityConfiguration` to create the `FilterChainProxy` known as the **Spring Security Filter Chain** (`springSecurityFilterChain`).
 - The `springSecurityFilterChain` is the `Filter` that the `DelegatingFilterProxy` delegates to.
 - Customizations to the `WebSecurity` can be made by
 - *creating* a `WebSecurityConfigurer` or
 - *exposing* a `WebSecurityCustomizer` bean.

```
package org.springframework.security.config.annotation.web.builders;

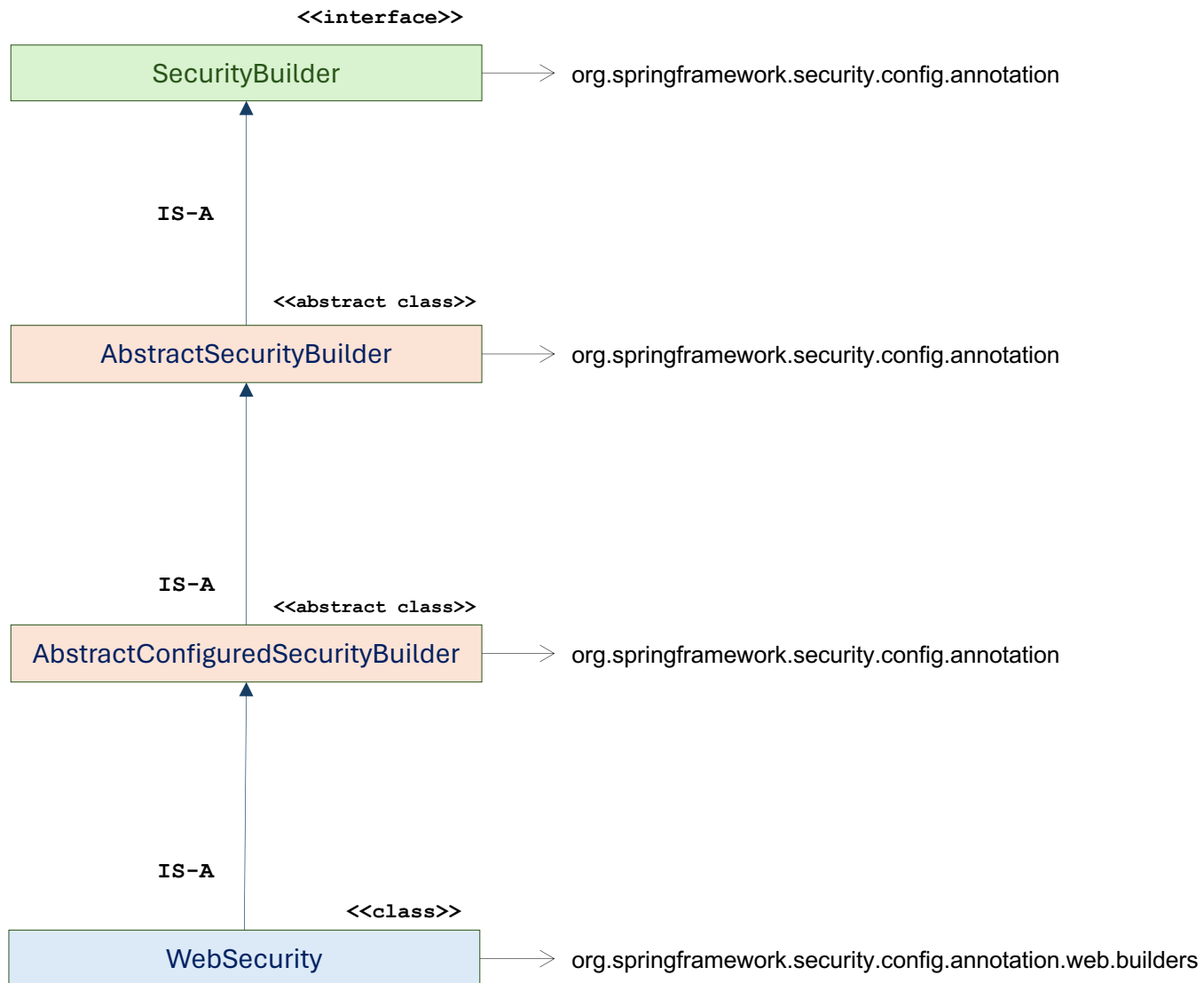
public final class WebSecurity extends AbstractConfiguredSecurityBuilder<Filter, WebSecurity> implements SecurityBuilder<Filter>, ApplicationContextAware, ServletContextAware {

    . . . // intentionally skipped

    private final List<RequestMatcher> ignoredRequests;
    private final List<SecurityBuilder<? extends SecurityFilterChain>> securityFilterChainBuilders;
    private HttpFirewall httpFirewall;
    private ServletContext servletContext;

    . . . // intentionally skipped
}
```

WebSecurity - II



```
package org.springframework.security.config.annotation;

public interface SecurityBuilder<O> {
    O build() throws Exception;
}
```

- Builds an `Object`.

```
package org.springframework.security.config.annotation;

public abstract class AbstractSecurityBuilder<O> implements SecurityBuilder<O> {
    private AtomicBoolean building = new AtomicBoolean();
    private O object;

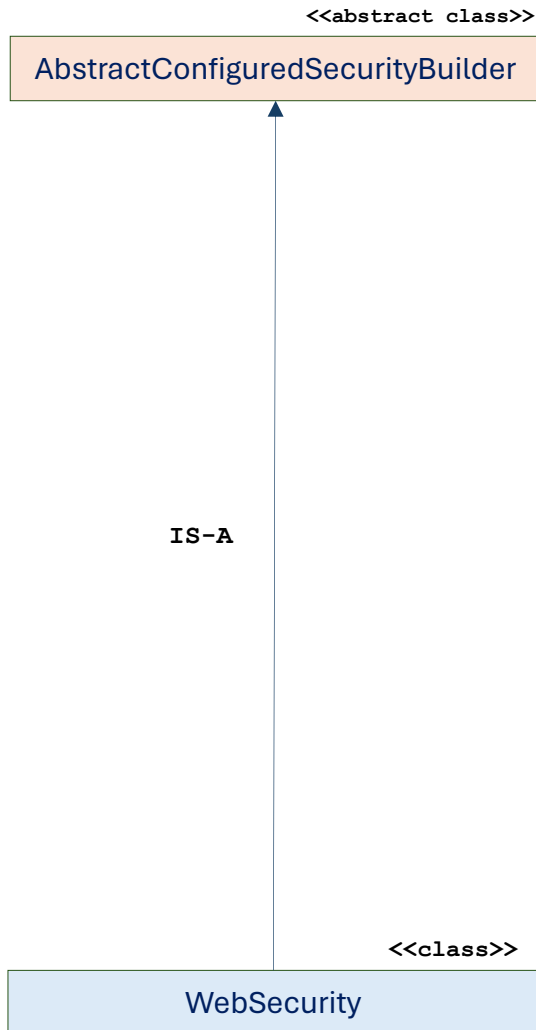
    @Override
    public final O build() throws Exception {
        if (this.building.compareAndSet(false, true)) {
            this.object = doBuild();
            return this.object;
        }
        throw new AlreadyBuiltException("This object has already been built");
    }

    public final O getObject() {
        if (!this.building.get()) {
            throw new IllegalStateException("This object has not been built");
        }
        return this.object;
    }

    protected abstract O doBuild() throws Exception;
}
```

- A base `SecurityBuilder` that ensures the object being built is only built one time.

WebSecurity - III



```
package org.springframework.security.config.annotation;

public abstract class AbstractConfiguredSecurityBuilder<O, B extends SecurityBuilder<O>> extends
AbstractSecurityBuilder<O> {
    private final LinkedHashMap<Class<? extends SecurityConfigurer<O, B>>, List<SecurityConfigurer<O, B>>> configurers;
    private BuildState buildState = BuildState.UNBUILT;

    . . . // intentionally skipped

    @Override
    protected final O doBuild() throws Exception {
        synchronized (this.configurers) {
            this.buildState = BuildState.INITIALIZING;
            beforeInit();
            init();
            this.buildState = BuildState.CONFIGURING;
            beforeConfigure();
            configure();
            this.buildState = BuildState.BUILDING;
            O result = performBuild();
            this.buildState = BuildState.BUILT;
            return result;
        }
    }

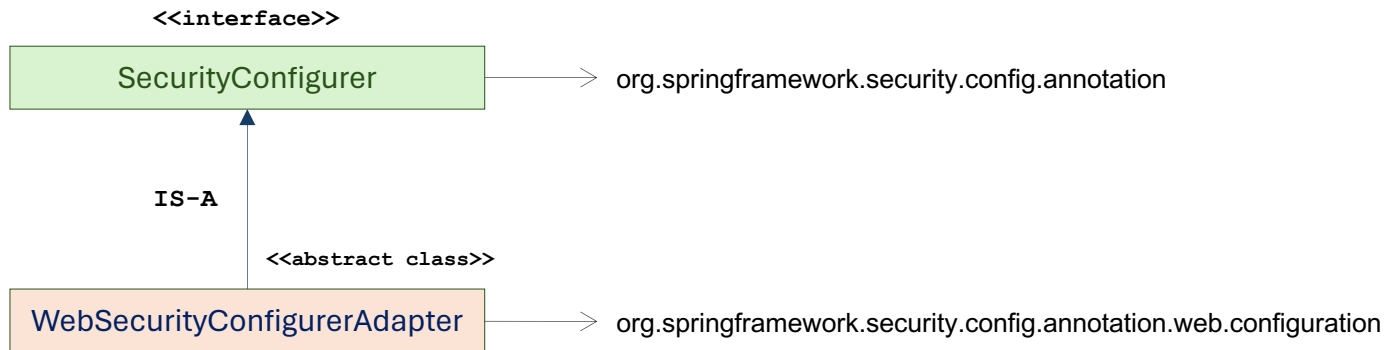
    protected abstract O performBuild() throws Exception;

    . . . // intentionally skipped
}
```

- A base `SecurityBuilder` that allows `SecurityConfigurer` to be applied to it.
- This makes modifying the `SecurityBuilder` a strategy that can be customized and broken up into several `SecurityConfigurer` objects that have more specific goals than that of the `SecurityBuilder`.
 - For example, a `SecurityBuilder` may build an `DelegatingFilterProxy`, but a `SecurityConfigurer` might populate the `SecurityBuilder` with the filters necessary for **session management**, **form-based login**, **authorization**, etc.

WebSecurityConfigurerAdapter

- Provides a **convenient base class** for creating a [WebSecurityConfigurer](#) instance.
 - The implementation allows **customization** by overriding methods.



```
package org.springframework.security.config.annotation.web.configuration;

public abstract class WebSecurityConfigurerAdapter extends SecurityConfigurer<Filter, WebSecurity> {
    private final AuthenticationManagerBuilder authenticationBuilder;
    private final AuthenticationManagerBuilder parentAuthenticationBuilder;

    . . . // intentionally skipped

    private AuthenticationManager authenticationManager;
    private HttpSecurity http;

    . . . // intentionally skipped
}
```

- Configures the [SecurityBuilder](#) instance, *i.e.*, [WebSecurity](#).

WebSecurityConfigurer

- Allows customization to the [WebSecurity](#).
- In most instances users will use [EnableWebSecurity](#) and create a [Configuration](#) that exposes a [SecurityFilterChain](#) bean.
 - This will automatically be applied to the [WebSecurity](#) by the [EnableWebSecurity](#) annotation.

```
package org.springframework.security.config.annotation.web;  
  
public interface WebSecurityConfigurer<T extends SecurityBuilder<Filter>> extends SecurityConfigurer<Filter, T> {  
}
```

SecurityConfigurer

- Allows for configuring a [SecurityBuilder](#).
 - All [SecurityConfigurer](#) first have their **init**([SecurityBuilder](#)) method invoked.
 - After all **init**([SecurityBuilder](#)) methods have been invoked, each **configure**([SecurityBuilder](#)) method is invoked.

```
package org.springframework.security.config.annotation.web.configuration;

// Type parameters:
//   <O> - The object being built by the SecurityBuilder B
//   <B> - The SecurityBuilder that builds objects of type O.
//   This is also the SecurityBuilder that is being configured.
public interface SecurityConfigurer<O, B extends SecurityBuilder<O>> {

    void init(B builder) throws Exception;
    void configure(B builder) throws Exception;

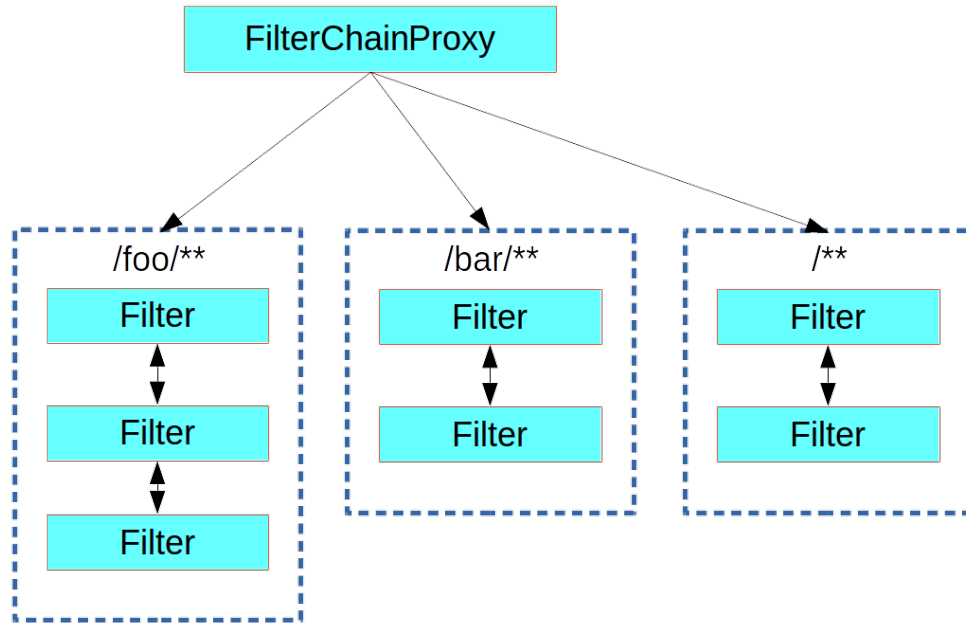
}
```

Customizations Outline

- Dispatching Requests to the First Chain That Matches
- Configuration of Filter Chains
- Creating and Customizing Filter Chains - I
- Creating and Customizing Filter Chains – II
- Request Matching for Dispatch and Authorization
- Combining Application Security Rules with Actuator Rules

Dispatching Requests to the First Chain That Matches

- There can be **multiple filter chains** all managed by Spring Security in the same top level `FilterChainProxy` and all are *unknown to the container*.
- The Spring Security filter contains a **list of filter chains** and *dispatches a request to the first chain that matches it*.
- The **most important feature** of this dispatch process is that *only one chain ever handles a request*.



- The **dispatch** happening based on matching the request path
 - `/foo/**` matches before `/**`.

Configuration of Filter Chains

- A **vanilla Spring Boot application** with no custom security configuration has a several (call it n) **filter chains**, where usually **n=6**.
 - The first (n-1) chains are there just to **ignore static resource patterns**, like **/css/**** and **/images/****, and the error view: **/error**.
 - The last chain matches the **catch-all** path (**/****) and is more active, containing logic for **authentication**, **authorization**, **exception handling**, **session handling**, **header writing**, and so on.
 - There are **a total of 11 filters** in this chain by default.
 - Users **don't have to** concern themselves with which filters are used and when.
-

- All filters internal to Spring Security are **unknown to the container** is important, especially in a Spring Boot application, where, by default, all **@Beans** of type **Filter** are **registered automatically with the container**.
- If you want to add a **custom filter** to the security chain, you need to either
 - **not make** it be a **@Bean** or
 - wrap it in a **FilterRegistrationBean** that explicitly disables the **container registration**.

Creating and Customizing Filter Chains - I

- The **default fallback filter** chain in a Spring Boot application (the one with the `/**` request matcher) has a **predefined order** of `SecurityProperties.BASIC_AUTH_ORDER`.
 - Order applied to the `SecurityFilterChain` that is used to configure **basic authentication** for application endpoints.
- The **actual order** can be interpreted as **prioritization**, with the first object (with the **lowest order value**) having the **highest priority**.

```
package org.springframework.boot.autoconfigure.security;

public class SecurityProperties {
    . . . // intentionally skipped

    public static final int BASIC_AUTH_ORDER = Ordered.LOWEST_PRECEDENCE - 5;

    public static final int IGNORED_ORDER = Ordered.HIGHEST_PRECEDENCE;

    . . . // intentionally skipped
}
```

- `SecurityProperties.IGNORED_ORDER` is applied to the `WebSecurityCustomizer` that ignores standard **static resource paths**.

Creating and Customizing Filter Chains - II

- You can **switch it off completely** by setting `security.basic.enabled = false`, or
- You can use it as a fallback and define other rules **with a lower order**.
 - Add a `@Bean` of type `WebSecurityConfigurer` and
 - Decorate the class with `@Order`:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/match1/**")
        ...;
    }
}
```

- This bean causes **Spring Security** to add a new filter chain and order it before the **fallback**.

Request Matching for Dispatch and Authorization

- A **security filter chain** (or, equivalently, a [WebSecurityConfigurerAdapter](#)) has a **request matcher** that is used to decide whether to apply it to an **HTTP request**.
- Once the decision is made to apply a particular filter chain, no others are applied.
- However, within a filter chain, you can have more **fine-grained control of authorization** by setting additional matchers in the [HttpSecurity](#) configurer, as follows:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/match1/**").authorizeRequests() // this one is a request matcher for the whole filter chain
            .antMatchers("/match1/user").hasRole("USER") // this one is only to choose the access rule to apply
            .antMatchers("/match1/spam").hasRole("SPAM")
            .anyRequest().isAuthenticated();
    }
}
```

Combining Application Security Rules with Actuator Rules

- If you use the **Spring Boot Actuator** for [management endpoints](#), you probably want them to be **secure**, and, by default, **they are**.
- In fact, as soon as you add the **Actuator** to a secure application, **you get an additional filter chain** that applies only to the actuator endpoints.
 - It is defined with a **request matcher** that matches only actuator endpoints and
 - it has an order of `ManagementServerProperties.BASIC_AUTH_ORDER`, which is 5 fewer than the default `SecurityProperties` fallback filter, so it is consulted before the fallback.

```
@Configuration
@Order(ManagementServerProperties.BASIC_AUTH_ORDER + 1)
public class ApplicationConfigurerAdapter extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/foo/**")
        . . .;
    }
}
```

Applying Custom Application Security Rules to the Actuator Endpoints:

- Add a [filter chain](#) that
 - is ordered earlier than the actuator one and
 - has a [request matcher](#) that includes all actuator endpoints.

Default Security Settings for the Actuator Endpoints

- Add your own filter
 - **later than the actuator one**,
 - but **earlier than the fallback**,
i.e., `ManagementServerProperties.BASIC_AUTH_ORDER + 1`

Method Security Outline

- Method Security

Method Security

- **Spring Security** offers support for [applying access rules](#) to Java [method executions](#).
- [Access rules](#) are declared using the same format of [ConfigAttribute](#) strings.

```
@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SampleSecureApplication {
}
```

```
@Service
public class MyService {

    @Secured("ROLE_USER")
    public String secure() {
        return "Hello Security";
    }
}
```

- Spring creates a [@Bean](#) of this type, it is proxied and callers must go through a **security interceptor** before the method is executed.
- If access is denied, the caller gets an [AccessDeniedException](#) instead of the actual method result.

Working with Threads Outline

- SecurityContext – I
- SecurityContext – II
- AuthenticationPrincipal
- Authentication
- Principal
- Processing Secure Methods Asynchronously
- AsyncConfigurerSupport
- AsyncConfigurer

SecurityContext - I

- Spring Security is fundamentally **thread-bound** because it needs to make the current **authenticated principal** available to a wide variety of downstream consumers.
- `SecurityContext` contains an `Authentication`.

- You can access and manipulate the `SecurityContext` through static convenience methods in `SecurityContextHolder`, which, in turn, manipulate a `ThreadLocal`.

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
assert(authentication.isAuthenticated());
```

- If you need access to the **currently authenticated user** in a web endpoint, you can use a method parameter in a `@RequestMapping`, annotated by `@AuthenticationPrincipal`.
 - pulls the current `Authentication` out of the `SecurityContext` and calls the `getPrincipal()` method on it to yield the method parameter.

```
@RequestMapping("/foo")
public String foo(@AuthenticationPrincipal User user) {
    // do stuff with user
}
```

SecurityContext - II

- The type of the [Principal](#) in an [Authentication](#) is dependent on the [AuthenticationManager](#) used to validate the authentication, so this can be a useful little trick to get a **type-safe** reference to user data.
- If Spring Security is in use, the [Principal](#) from the [HttpServletRequest](#) is of type [Authentication](#), so you can also use that directly.

```
@RequestMapping("/foo")
public String foo(Principal principal) {
    Authentication authentication = (Authentication) principal;
    User = (User) authentication.getPrincipal();
    // do stuff with user
}
```

AuthenticationPrincipal

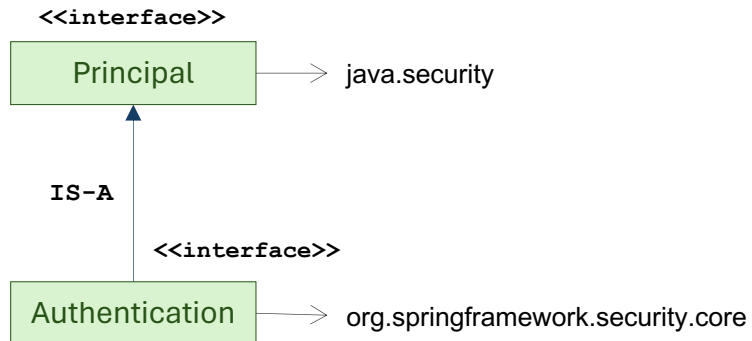
- Annotation that is used to resolve `Authentication.getPrincipal()` to a method argument.

```
package org.springframework.security.core.annotation;

@Target({ElementType.PARAMETER, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface AuthenticationPrincipal {
    // True if a ClassCastException should be thrown when
    // the current Authentication.getPrincipal() is the incorrect type.
    boolean errorOnInvalidType() default false;
    String expression() default "";
}
```

Authentication

- Represents the **token** for an **authentication request** or for an **authenticated principal** once the request has been processed by the `AuthenticationManager.authenticate(Authentication)` method.
- Once the request has been authenticated, the `Authentication` will usually be stored in a **thread-local** `SecurityContext` managed by the `SecurityContextHolder` by the `authentication mechanism` which is being used.



```
package org.springframework.security.core;

public interface Authentication extends Principal, Serializable {
    // returns the Principal being authenticated or the authenticated principal after authentication.
    //
    // In the case of an authentication request with username and password, this would be the username.
    //
    // The AuthenticationManager implementation will often return an Authentication containing richer
    // information as the principal for use by the application.
    //
    // Many of the authentication providers will create a UserDetails object as the principal.
    Object getPrincipal();

    // other APIs were intentionally skipped

    Collection<? extends GrantedAuthority> getAuthorities();
    boolean isAuthenticated();
}
```

Principal

- Interface represents the abstract notion of a [Principal](#), which can be used to represent any entity, such as an [individual](#), a [corporation](#), and a [login id](#).

```
package java.security;

public interface Principal {
    boolean equals(Object another);
    String toString();
    int hashCode();
    String getName();
}
```

Processing Secure Methods Asynchronously

- Since the `SecurityContext` is **thread-bound**, if you want to do any background processing that calls secure methods, i.e., with `@Async`, you need to ensure that the context is propagated.
 - This boils down to wrapping the `SecurityContext` with the task (Runnable, Callable, and so on) that is executed in the background.
- To propagate the `SecurityContext` to `@Async` methods, you need to supply an `AsyncConfigurer` and ensure the `Executor` is of the correct type

```
@Configuration
public class ApplicationConfiguration extends AsyncConfigurerSupport {
    @Override
    public Executor getAsyncExecutor() {
        return new DelegatingSecurityContextExecutorService(Executors.newFixedThreadPool(5));
    }
}
```

AsyncConfigurerSupport

- A convenience [AsyncConfigurer](#) that implements all methods so that the defaults are used.
 - Provides a **backward compatible alternative** of implementing [AsyncConfigurer](#) directly.
- **Deprecated**
 - as of 6.0 in favor of implementing [AsyncConfigurer](#) directly.

```
package org.springframework.scheduling.annotation;

@Deprecated(since = "6.0")
public class AsyncConfigurerSupport implements AsyncConfigurer {
    @Override
    public Executor getAsyncExecutor() {
        return null;
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return null;
    }
}
```

AyncConfigurer

- Interface to be implemented by classes annotated with `@EnableAsync` and `@Configuration` that wish to customize
 - the `Executor` instance used when **processing async method invocations** or
 - the `AsyncUncaughtExceptionHandler` instance used to process **exceptions thrown from async method** with `void` return type.

```
package org.springframework.scheduling.annotation;

public interface AsyncConfigurer {

    default Executor getAsyncExecutor() {
        return null;
    }

    default AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return null;
    }
}
```