# Spring Security

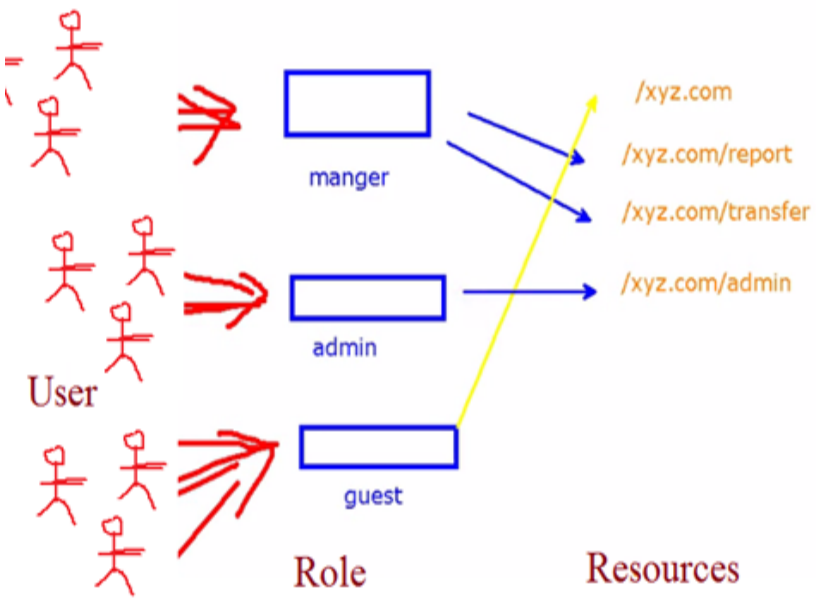
Rajeev Gupta MTech CS
Trainer & Consultant

# Agenda

- Understanding Role access control
- What is Spring Security
- History
- Servlet filter review
- Setting up Spring Security
- How the basic Spring Security flow works
- Simple customizations of Spring Security
- Log In Page
- Custom Log In Controller

# Role based Access Control RAC

- Role based Access Control (RAC)
  - As it is difficult to manage permission for each user, each user is assigned to a role and permission is set for the role
  - Authentication using Spring
    - Http Basic Authentication (uses in XML- pop up form)
    - Http form based Authentication( uses in XML- custom form)
    - Http form based Authentication( uses in DB)



# Spring security

- Spring Security is a security framework that provides declarative security for your Spring-based applications.

- Spring Security handles authentication and authorization at both the web request level and at the method invocation level. Spring Security takes full advantage of DI and AOP

- Security Classification :
  - Wire level security
    - By encryption of data bw client and server
    - By enabling the Https port and security certificates
  - Application level security
    - Authorization and Authentication –Can use Spring API

# History of Spring Security

- Acegi security framework is simplified by Spring framework, hundreds line of xml configuration reduced to few line, thanks to spring new namespace for security, along with annotations and reasonable defaults

Spring Security got its start as Acegi Security. Acegi was a powerful security framework, but it had one big turn-off: it required a *lot* of XML configuration. I'll spare you the intricate details of what such a configuration may have looked like. Suffice it to say that it was common for a typical Acegi configuration to grow to several hundred lines of XML.

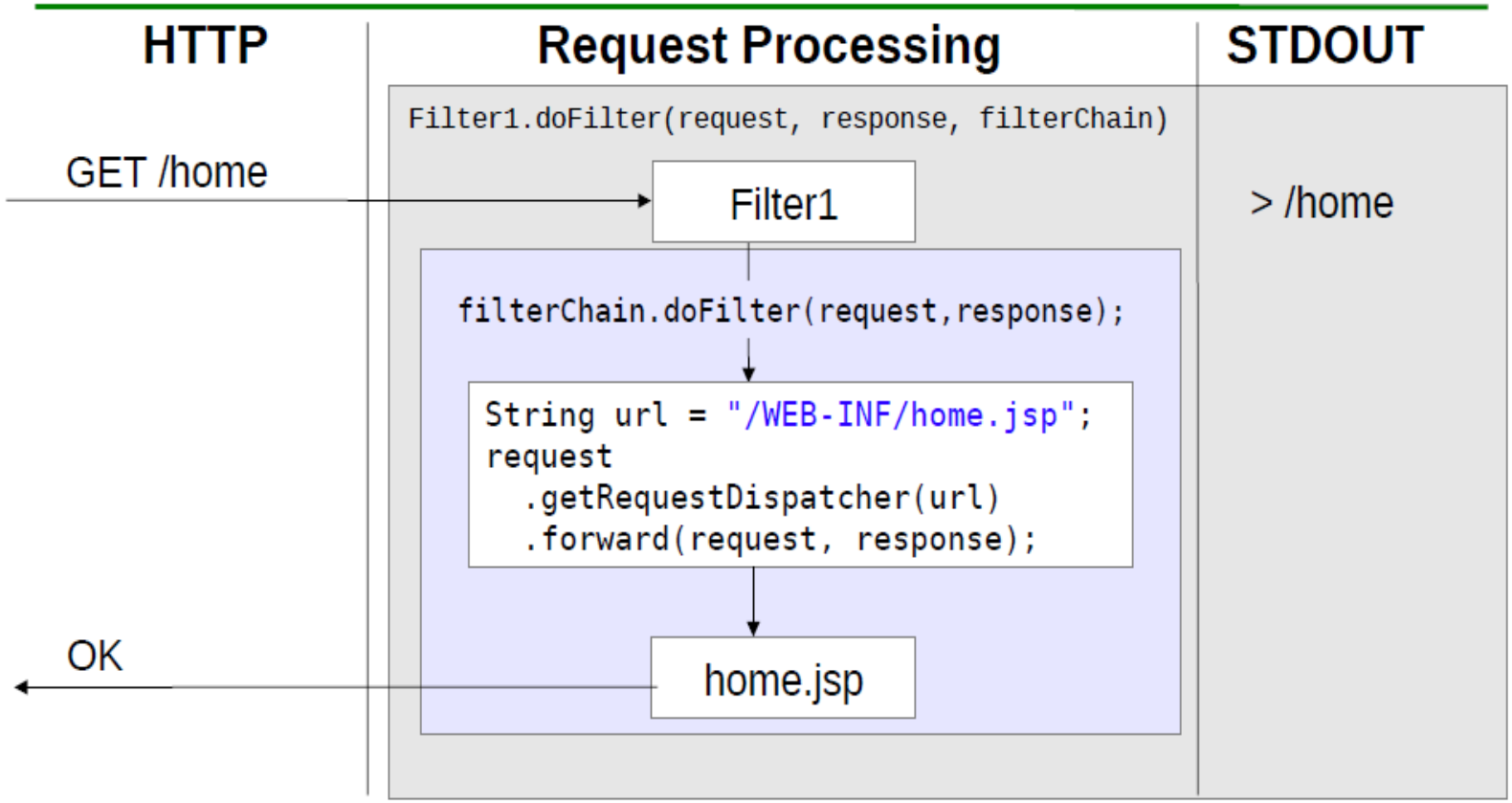
Spring Security tackles security from two angles. To secure web requests and restrict access at the URL level, Spring Security uses servlet filters. Spring Security can also secure method invocations using Spring AOP—proxying objects and applying advice that ensures that the user has proper authority to invoke secured methods.

# Servlet filter review

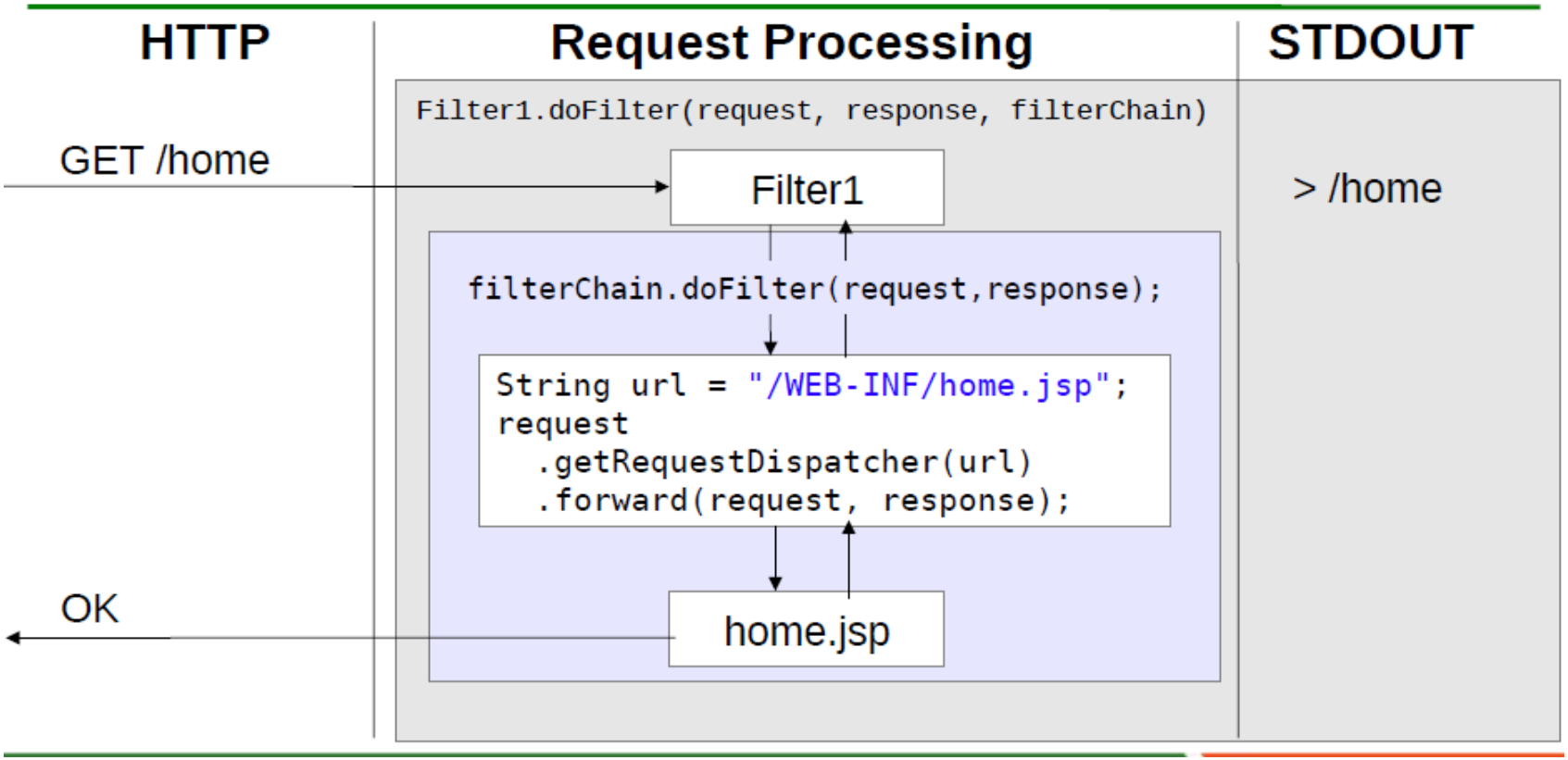
```
<filter>
  <filter-name>filter1</filter-name>
  <filter-class>Filter1</filter-class>
</filter>
<filter-mapping>
  <filter-name>filter1</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain filterChain) … {
  …
  // do something before
  System.out.println("> " + requestUrl);
  // run rest of application
  filterChain.doFilter(request, response);
  // cleanup
  System.out.println("< " + requestUrl);
}
```

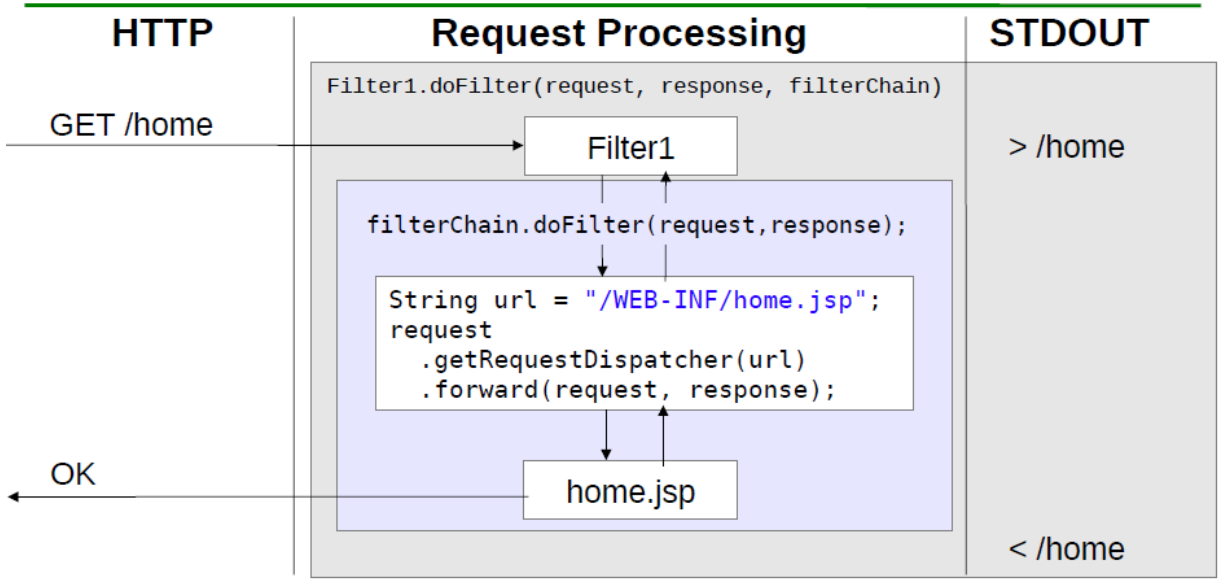
# Servlet filter review-Dispacher



# Servlet filter review-Dispacher



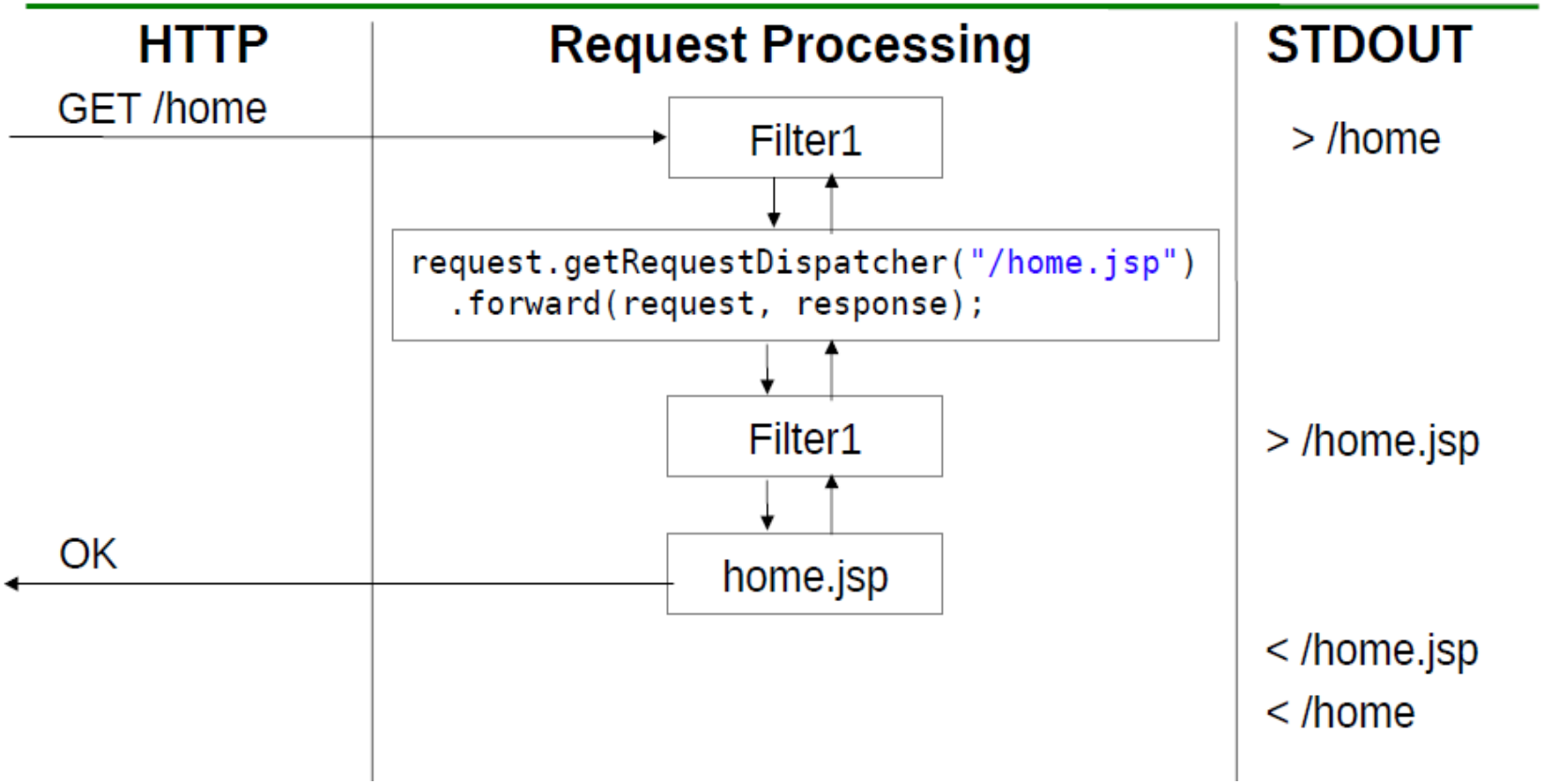
# Servlet filter review-Dispacher




What if we want to log every request and forward.

```
<filter>
  <filter-name>filter1</filter-name>
  <filter-class>Filter1</filter-class>
</filter>
<filter-mapping>
  <filter-name>filter1</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

# Servlet filter review-Dispacher



# Servlet filter review-Dispacher

- Spring Security is based on Servlet Filters
- Rare to process other dispatcher types, but important to be aware of them
- Ensure to include the necessary dispatcher elements
- Other possible dispatcher values include
  - REQUEST (default)
  - INCLUDE
  - FORWARD
  - ERROR

# Servlet filter review-FilterChain

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain filterChain) … {
  try {
    // run rest of application
    filterChain.doFilter(request, response);
    securityFilter.doFilter(request, response, filterChain);
    servlet.service(request, response);
  }
  catch (SecurityException e) {
    // handle error by sending to login page
  }
  finally {
    // cleanup
  }
}
```

# Basic Spring Security Setup

- Steps:
  1. Add Spring Security Dependencies
  2. Update web.xml
  3. Create Spring Security Configuration

# Add Spring Security Dependencies

```
<dependencies>
  ...
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>3.1.3.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.1.3.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.1.3.RELEASE</version>
  </dependency>
</dependencies>
```

# Update web.xml - ContextLoaderListener

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
     /WEB-INF/spring/*.xml
  </param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

## What is the ContextLoaderListener

- Not Specific to Spring Security
- Creates a Spring ApplicationContext using the Spring Configurations (i.e. the value of contextConfigLocation)
- Can be used to lookup objects in ApplicationContext
- Rare to interact with ApplicationContext directly

# ContextLoaderListener pseudocode

```
// init ApplicationContext
XmlWebApplicationContext applicationContext =
    new XmlWebApplicationContext();
applicationContext.setConfigLocation("/WEB-INF/spring/*.xml");
applicationContext.refresh();

// Use ApplicationContext
Filter filter =
    applicationContext.getBean("springSecurityFilterChain",
                               Filter.class);
```

```
// init ApplicationContext
XmlWebApplicationContext applicationContext =
    new XmlWebApplicationContext();
applicationContext.setConfigLocation
applicationContext.refresh();

// Use Applica
Filter filter
    applicatio
```

"/WEB-INF/spring/*.xml"

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
     /WEB-INF/spring/*.xml
  </param-value>
</context-param>
```

# Update web.xml - springSecurityFilterChain

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

# DelegatingFilterProxy pseudocode

```
public class DelegatingFilterProxy implements Filter {
  public void init(FilterConfig config) throws ServletException {
    // applicationContext is obtained from ContextLoaderListener
    this.delegate =
      applicationContext.getBean("springSecurityFilterChain",
                                 Filter.class);
  }
  public void doFilter(...) throws ... {
    this.delegate.doFilter(request, response, filterChain);
  }
  public void destroy() {
    // this may not be invoked depending on the settings
    this.delegate.destroy();
  }
  private Filter delegate;
}
```

# DelegatingFilterProxy pseudocode

```
public class DelegatingFilterProxy implements Filter {
  public void init(FilterConfig config) throws ServletException {
    // applicationContext is obtained from ContextLoaderListener
    this.delegate =
      applicationContext.getBean( springSecurityFilterChain ,
                                  Filter.class);
  }
  <filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
    this.delegate.destroy();
  }
  private Filter delegate;
}
```

# DelegatingFilterProxy pseudocode

```
public class DelegatingFilterProxy implements Filter {
  public void init(FilterConfig config) throws ServletException {
    // applicationContext is obtained from ContextLoaderListener
    this.delegate =
      applicationContext.getBean("springSecurityFilterChain",
                                 Filter.class);
  }
  public void doFilter(...) throws ... {
    this.delegate.doFilter(request, response, filterChain);
  }
  public void destroy() {
    // this may not be invoked depending on the settings
    this.delegate.destroy();
  }
  private Filter delegate;
}
```

# Create security.xml

- The file location should be src/main/webapp/WEB-INF/spring/security.xml to match the contextConfigLocation
- Need to ensure to get the xml namespace declaration correct

```
<security:http use-expressions="true">
  <security:intercept-url pattern="/**"
        access="hasRole('ROLE_USER')"/>
  <security:form-login />
</security:http>
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="user"
          password="password"
          authorities="ROLE_USER"/>
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

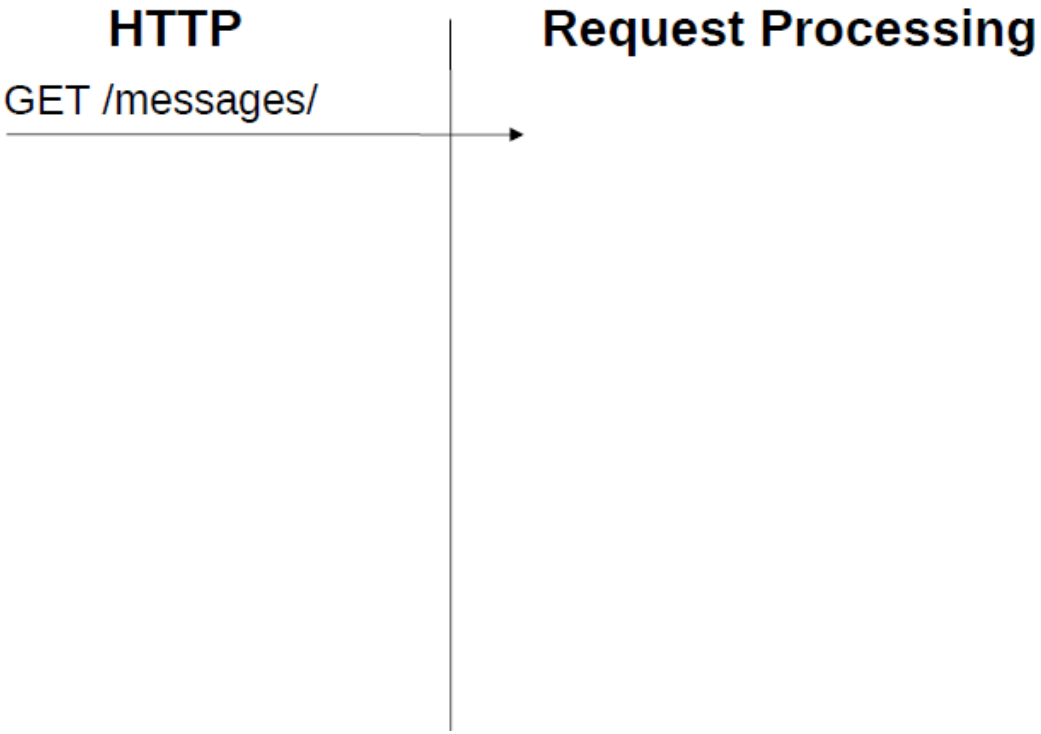
src/main/webapp/WEB-INF/security.xml

# FilterChainProxy (springSecurityFilterChain) Pseudocode

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain filterChain) … {

  Filter[] delegates = lookupDelegates(request);
  for(Filter delegate : delegates) {
    delegate.doFilter(request, response, chain);

    if(delegate does not invoke filterChain.doFilter)
      return;
  }

  filterChain.doFilter(request, response);
}
```

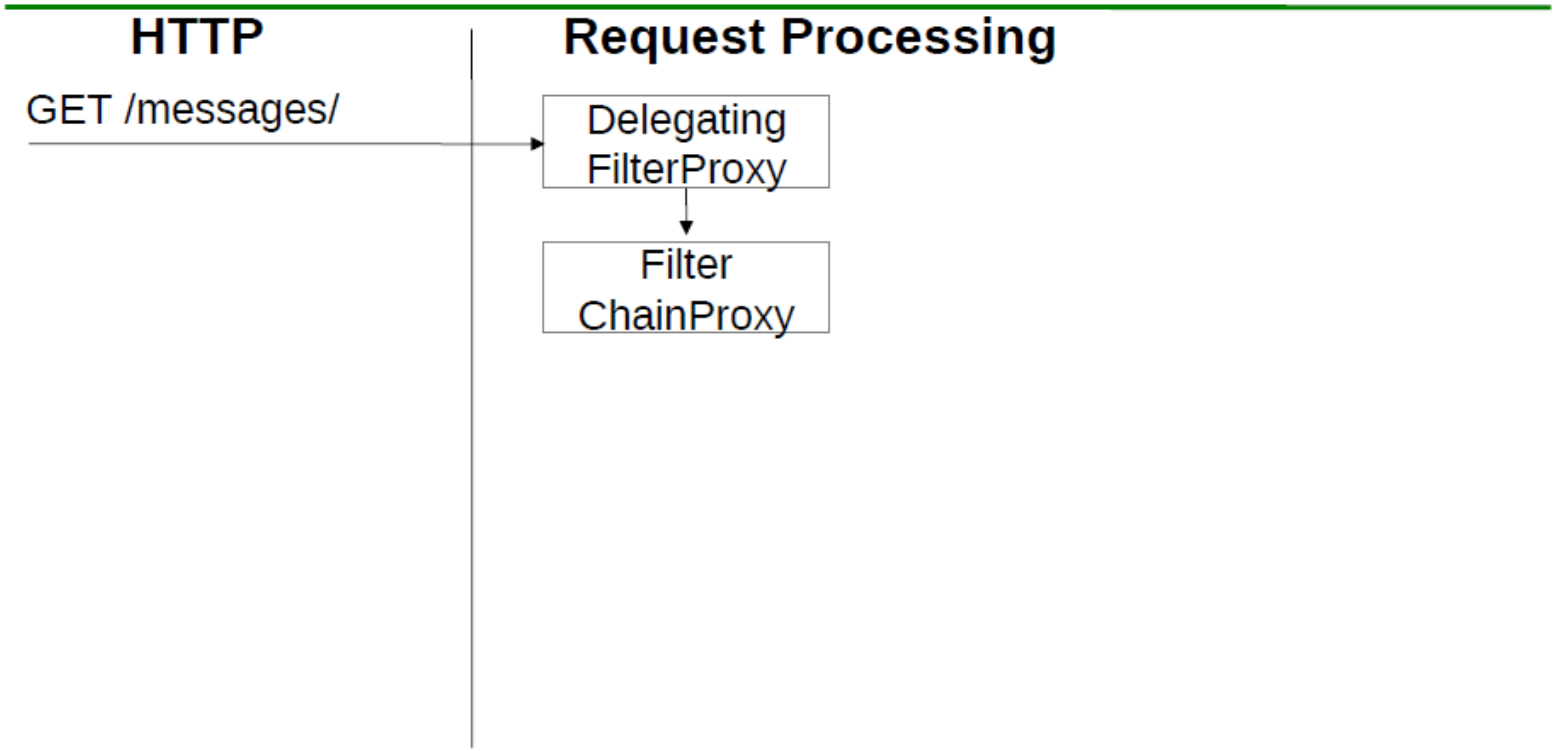
# Understanding How spring security works?

- User look for resource, say /messages



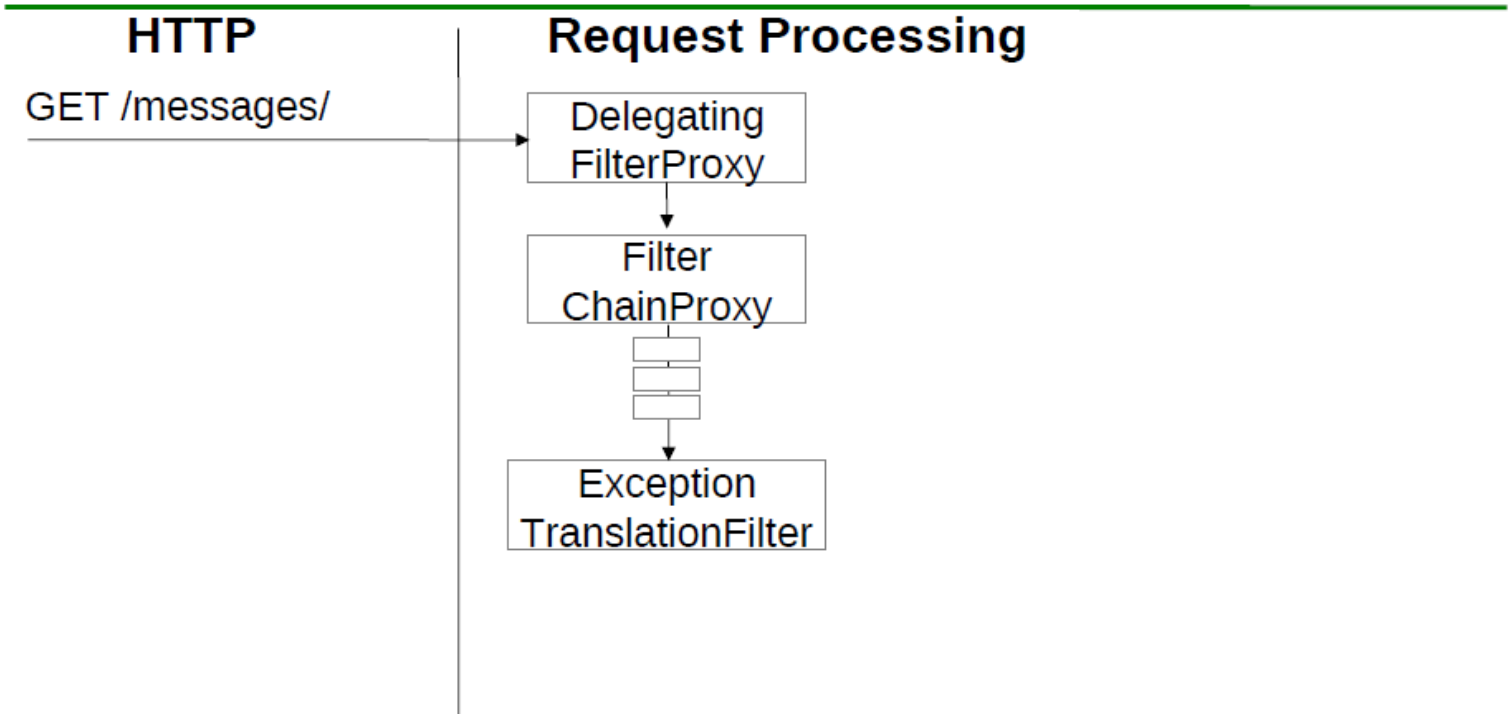

# Understanding How spring security works?

- Request is handled by delegating filter proxy filter that we have configure earlier, it

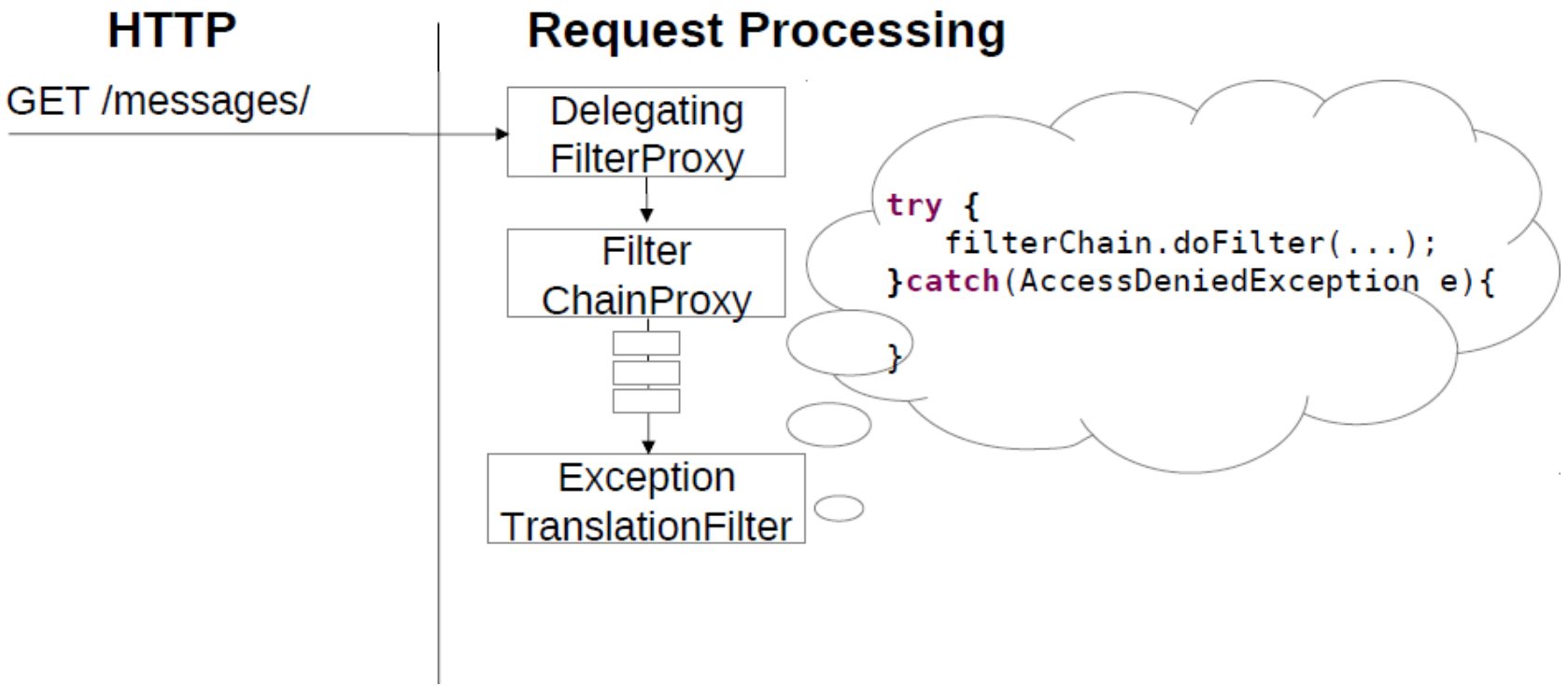



# Understanding How spring security works?

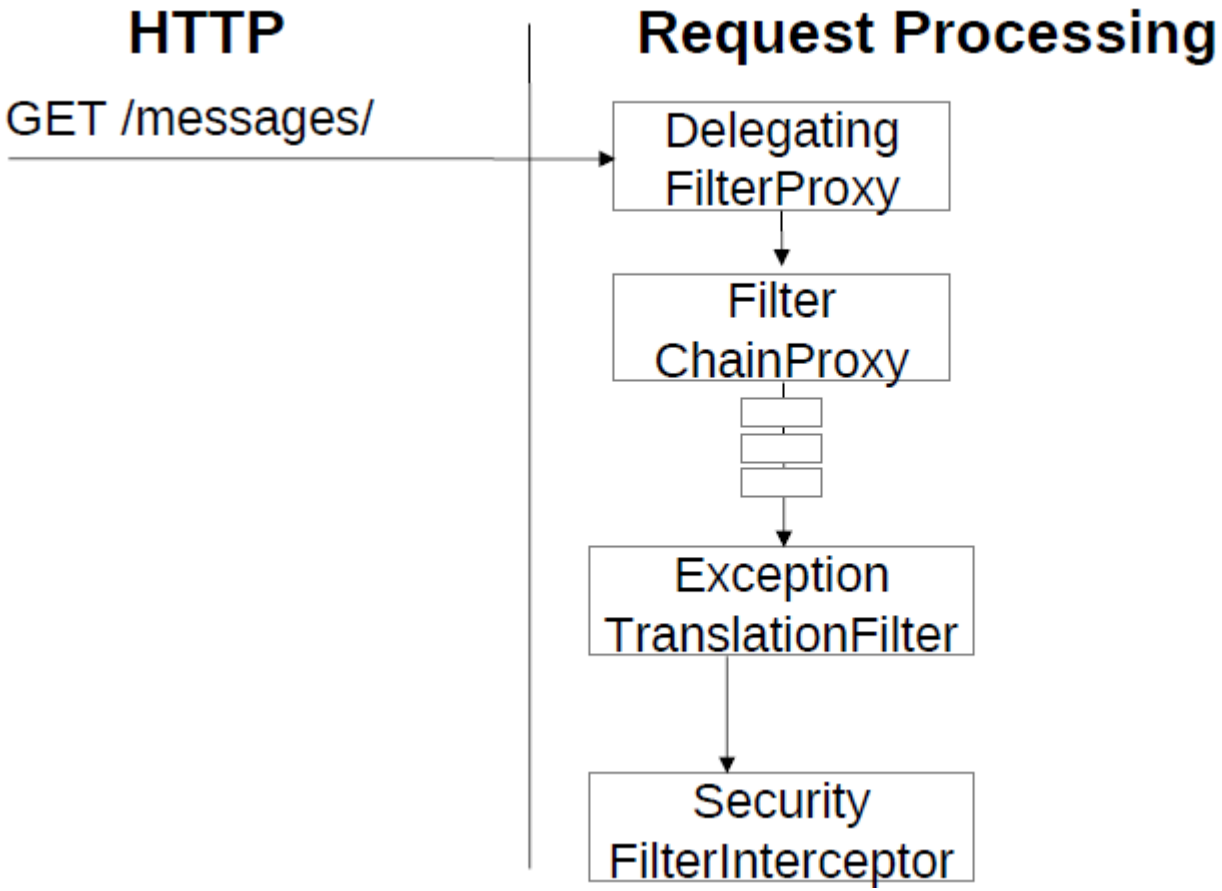
- Then request pass through series of filter, then request meet exception translationfilter



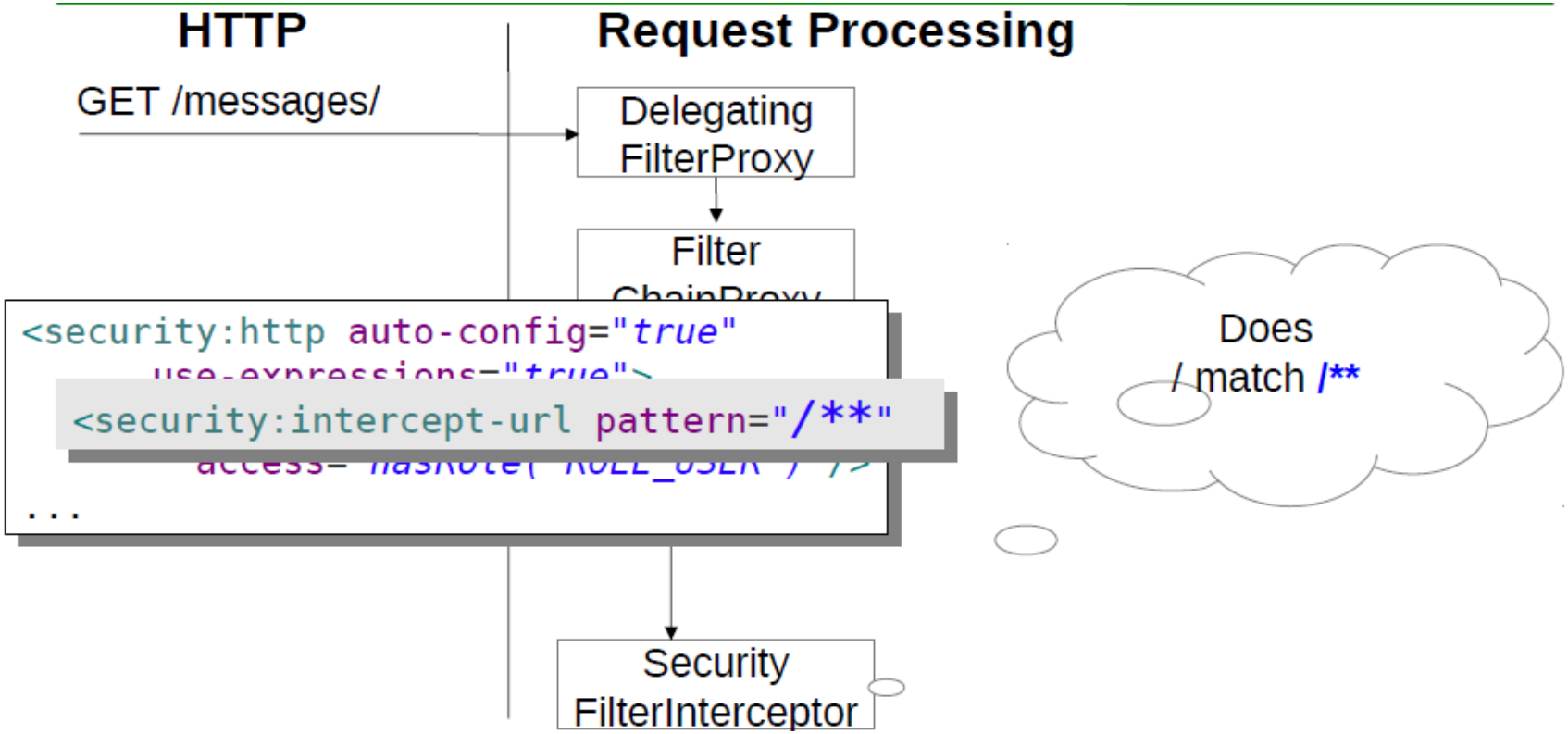

# Understanding How spring security works?



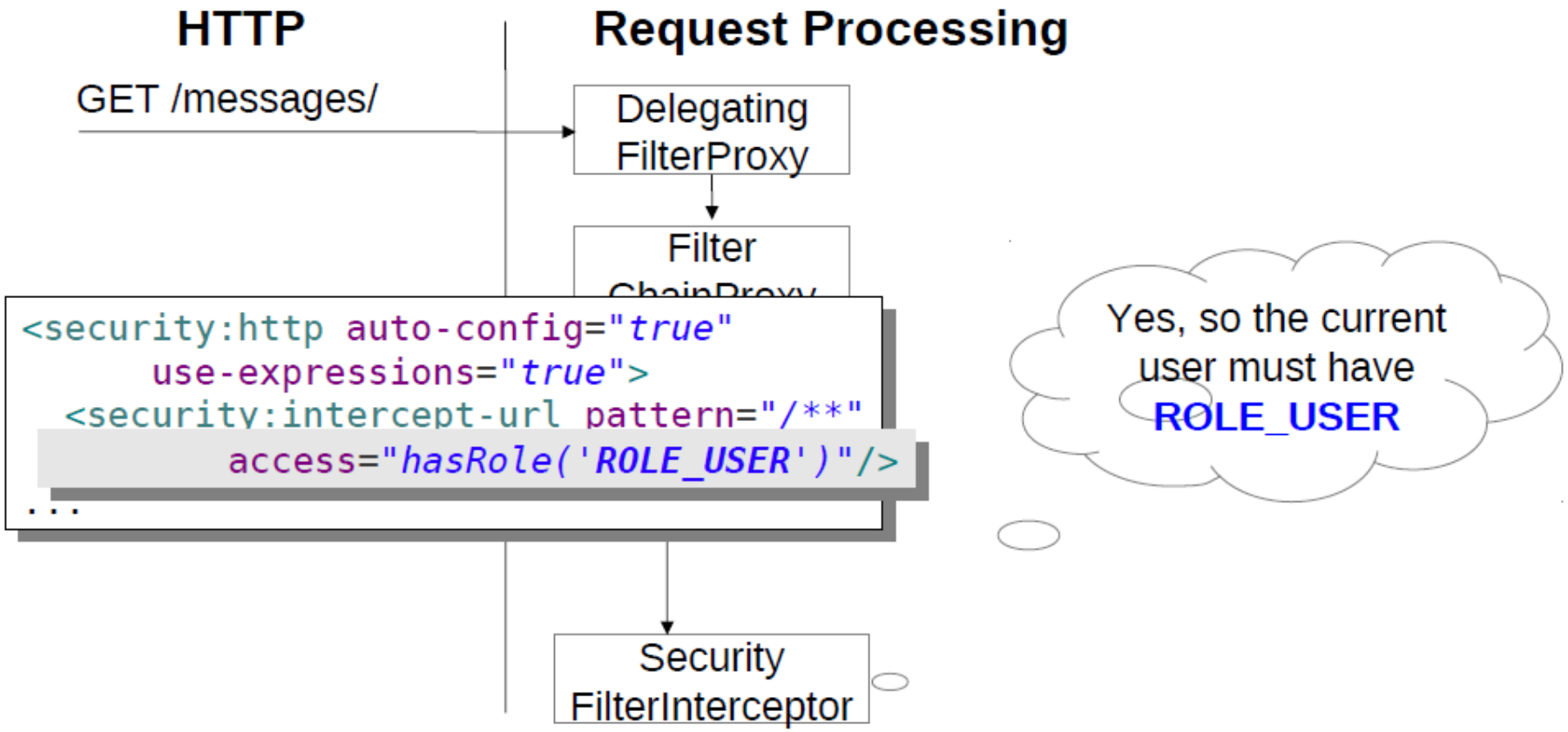
# Understanding How spring security works?



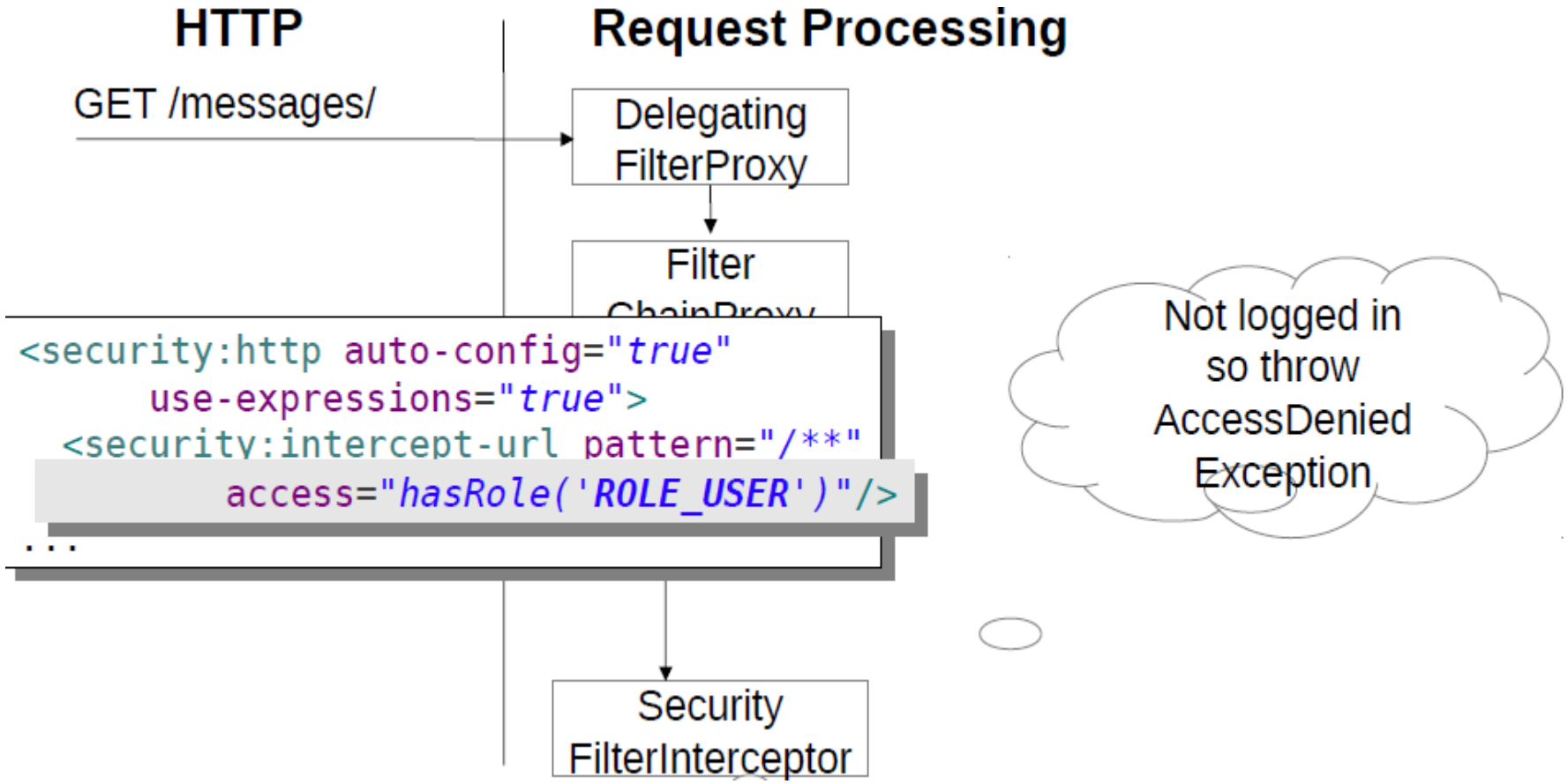
# Understanding How spring security works?



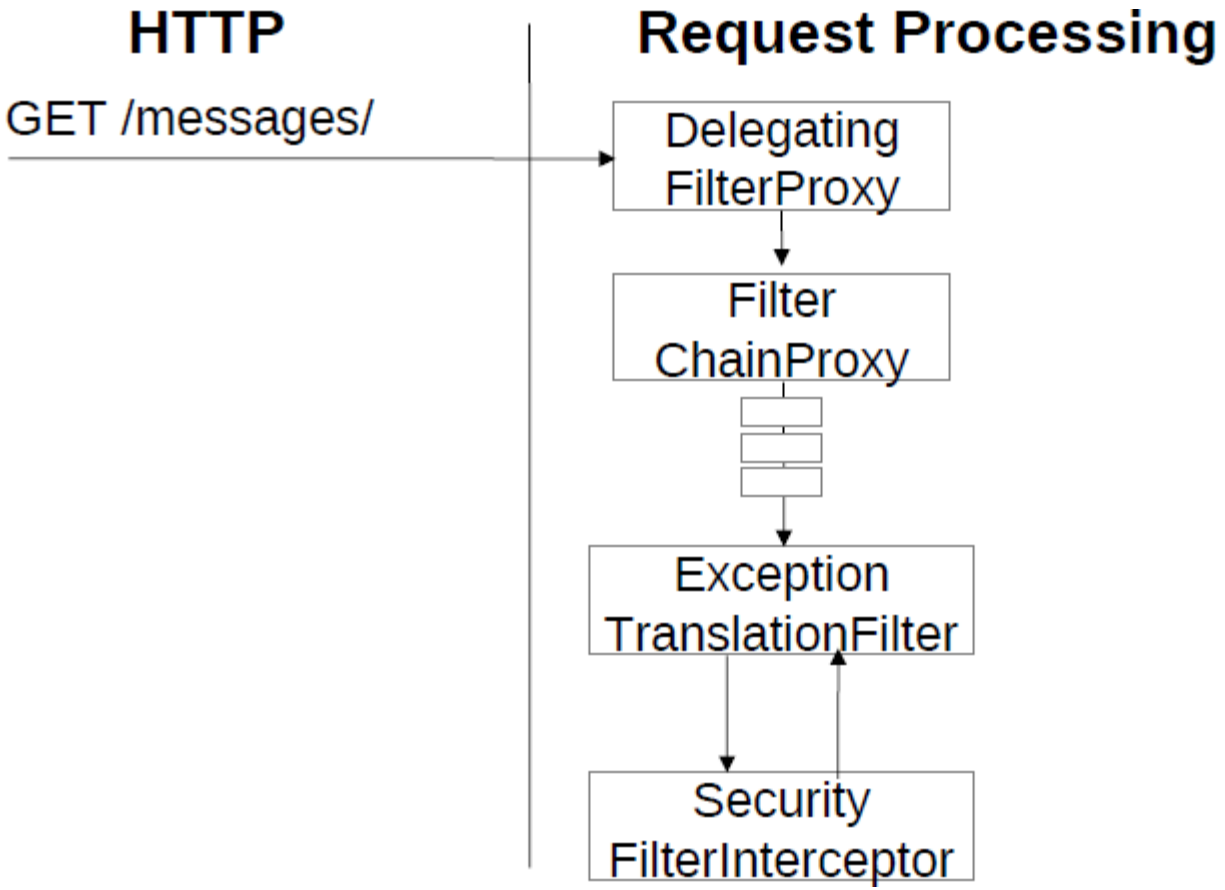
# Understanding How spring security works?



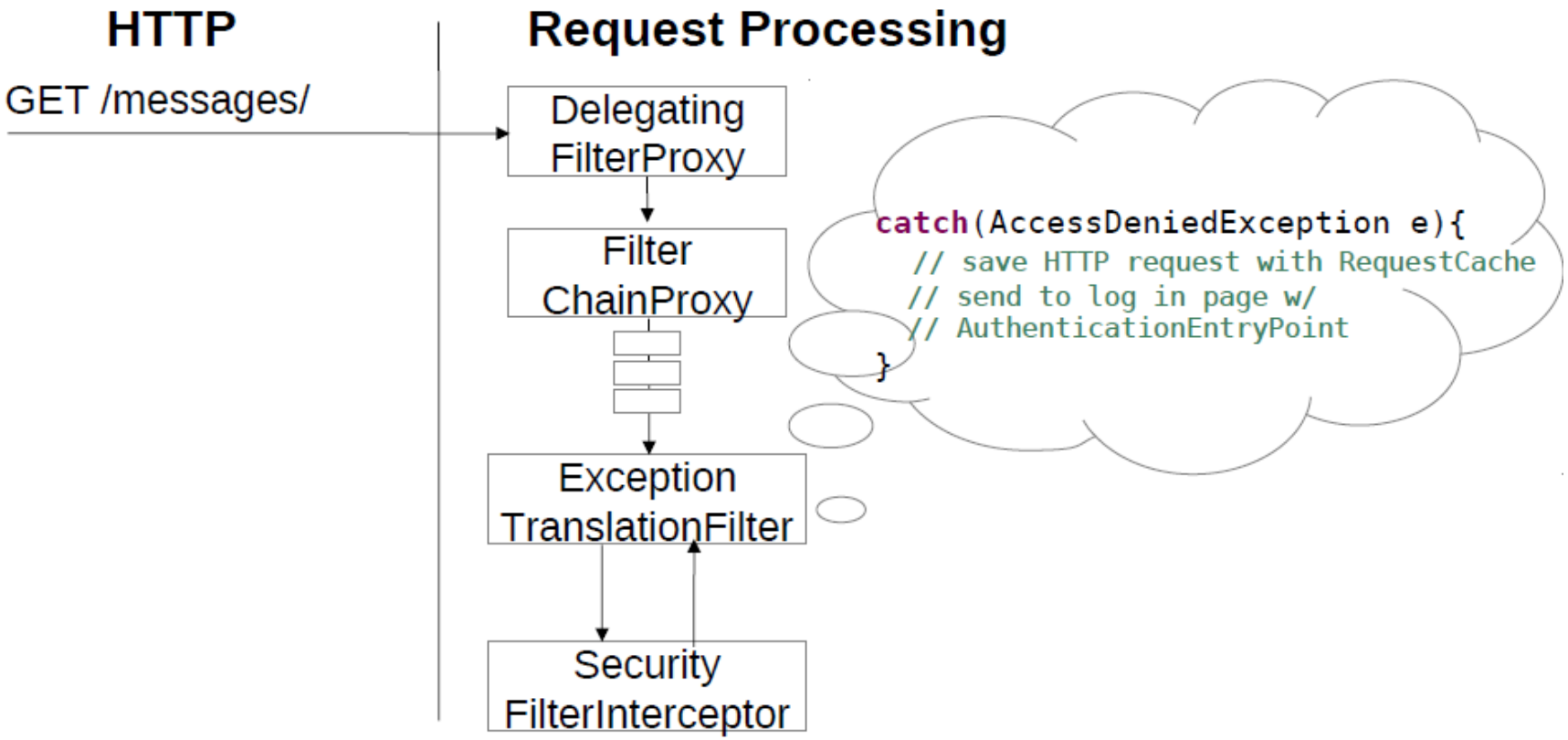
# Understanding How spring security works?



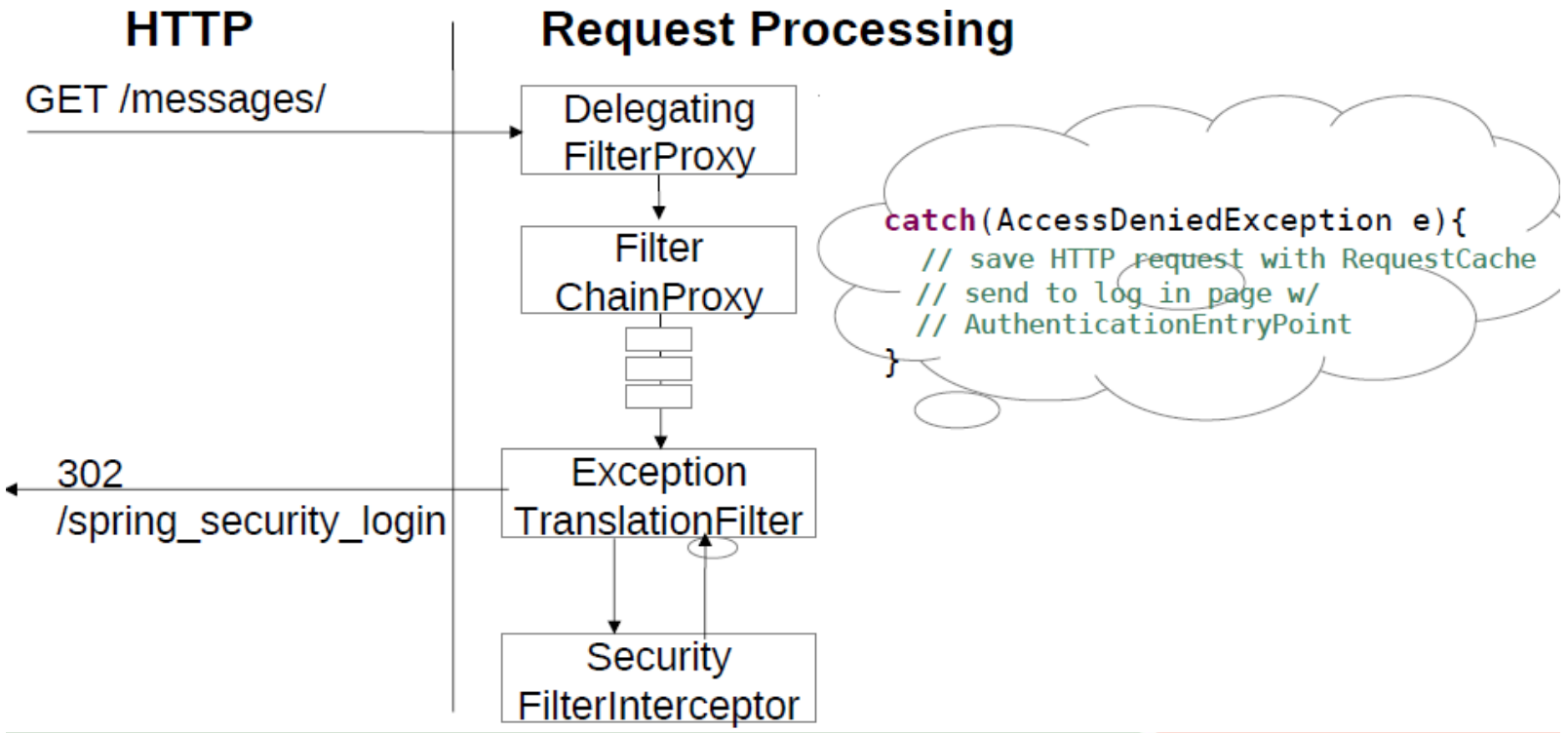
# Understanding How spring security works?



# Understanding How spring security works?



# Understanding How spring security works?



# Ant patterns

- Spring Security uses an AntPathRequestMatcher to determine if a URL
- matches the current URL. The following rules are used when r
  - Query parameters are not included in the match
  - The context path is not included in the match
  - ? matches one character
  - * matches zero or more characters (not a directory delimiter i.e. /)
  - ** matches zero or more 'directories' in a path

# Ant pattern examples

Ant Pattern examples that assume a context path of /messages

| Pattern | Description | Full Path | Path to Match |
|---|---|---|---|
| /** | Matches any URL | | |
| /* | Matches anything in root folder | /messages/1 | /1 |
| | | /messages/2?a=b | /2 |
| | | /messages/1/ | /1/ |

| Pattern | Description | Full Path | Path to Match |
|---|---|---|---|
| /1/** | Matches anything that starts with /1 | /messages/1 | /1 |
| | | /messages/1?a=b | /1 |
| | | /messages/1/ | /1/ |
| | | /messages/1/view | /1/view |
| | | /messages/other/ | /other/ |
| | | /messages/2/view | /2/view |

# Ant pattern examples

Be careful when using pattern matching

| Pattern | Description | Full Path | Path to Match |
|---|---|---|---|
| /**/*.css | Matches anything that ends with .css | /messages/styles/main.css | **/styles/main.css** |
| | | /messages/1 | **/1** |
| | | /messages/1.css | **/1.css** |

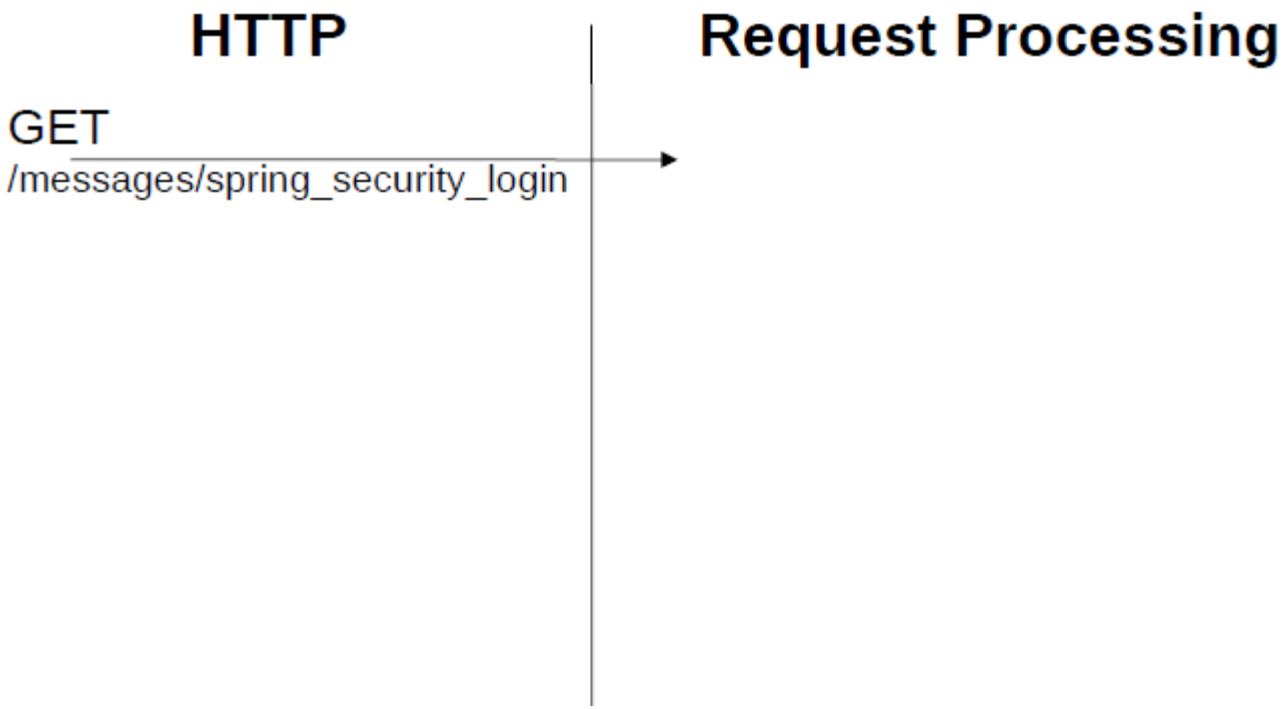
The less restrictive the mapping the easier it is for a malicious user to bypass

Spring MVC will treat /1.css the same as /1, so a malicious user can use this to bypass security constraints

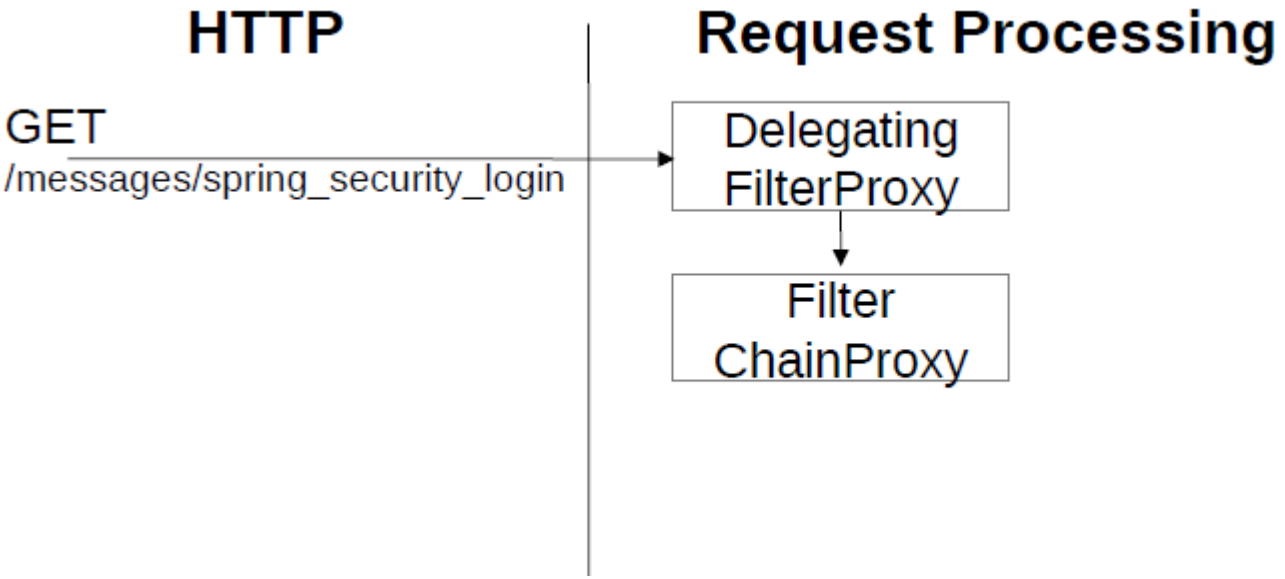
Other ways to bypass URL based security (i.e. path variables, non-normalized URLs, etc). Spring Security does have things in place to help protect you (i.e. HttpFirewall)

Best to combine URL Security with Method Security to provide defense in depth

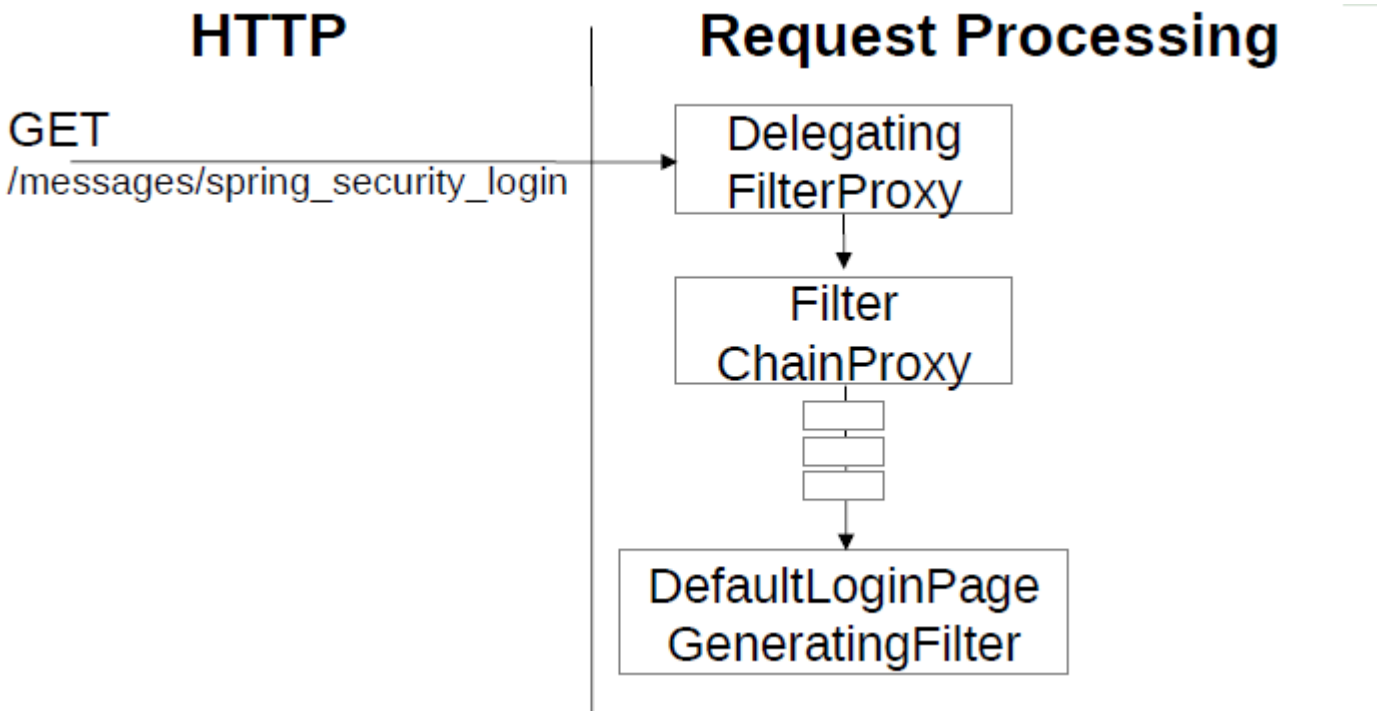
# Requesting log in page



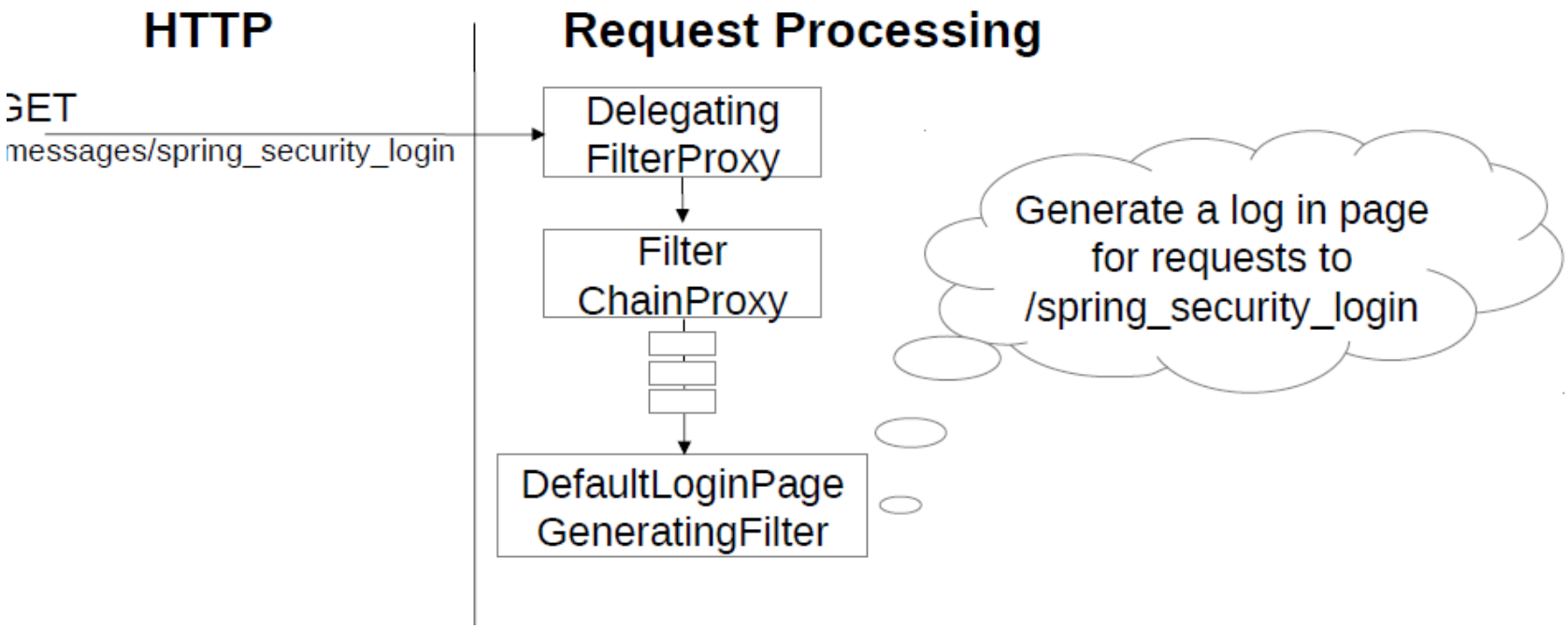

# Requesting log in page



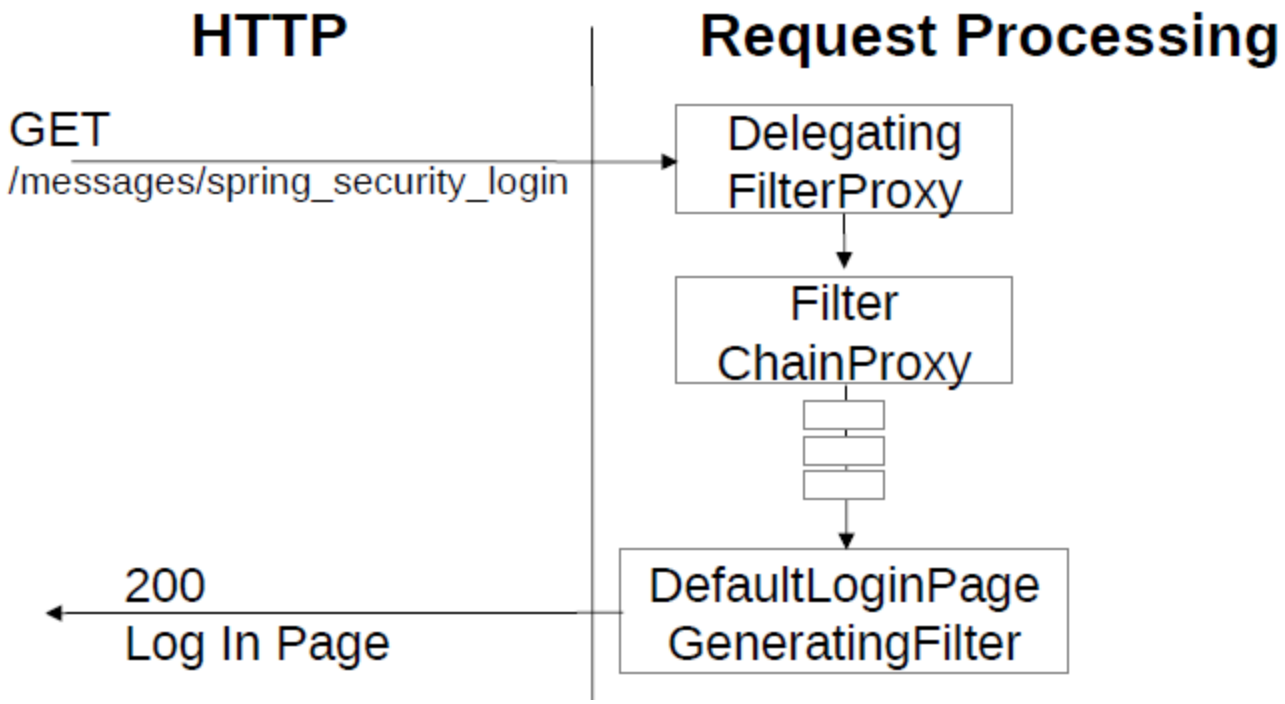

# Requesting log in page



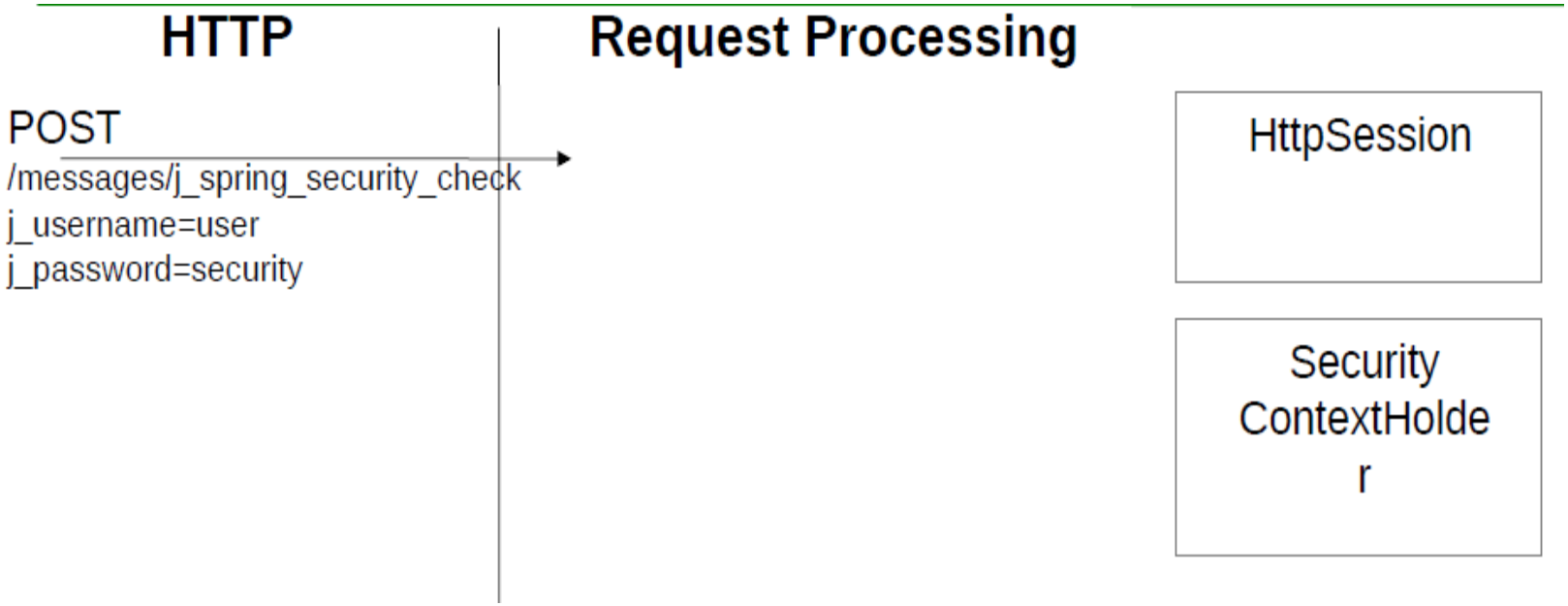

# Requesting log in page



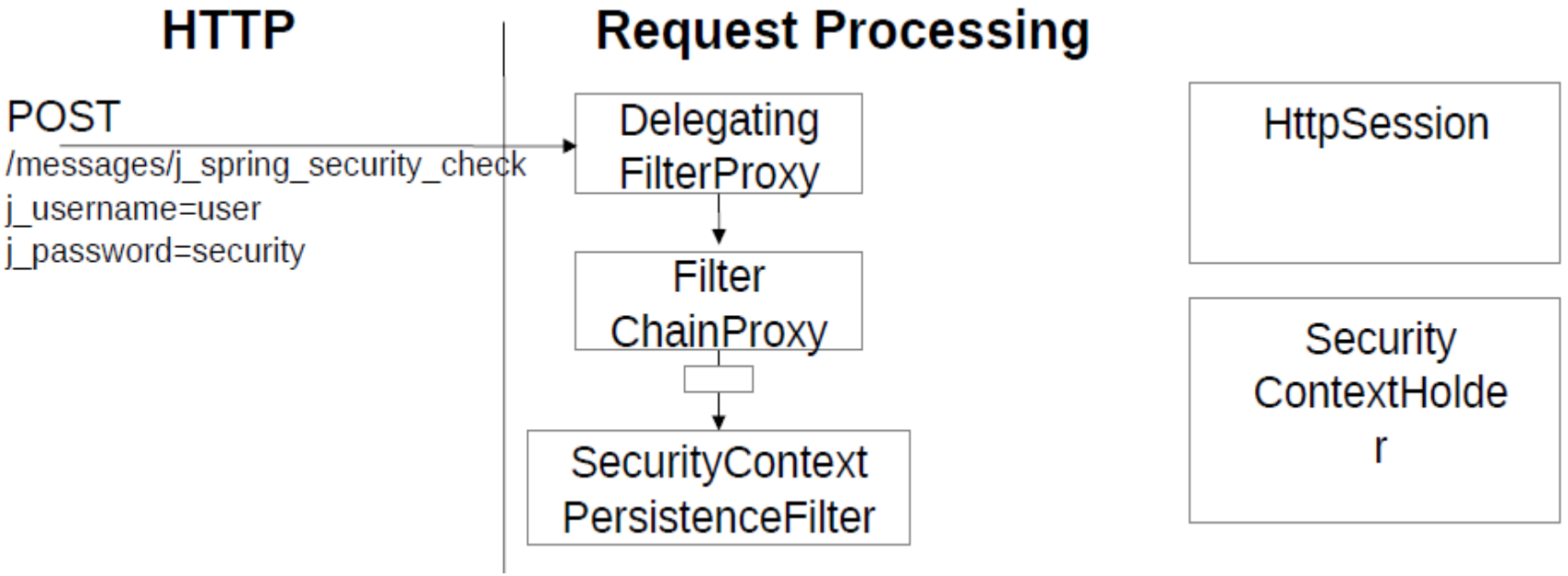
# Requesting log in page



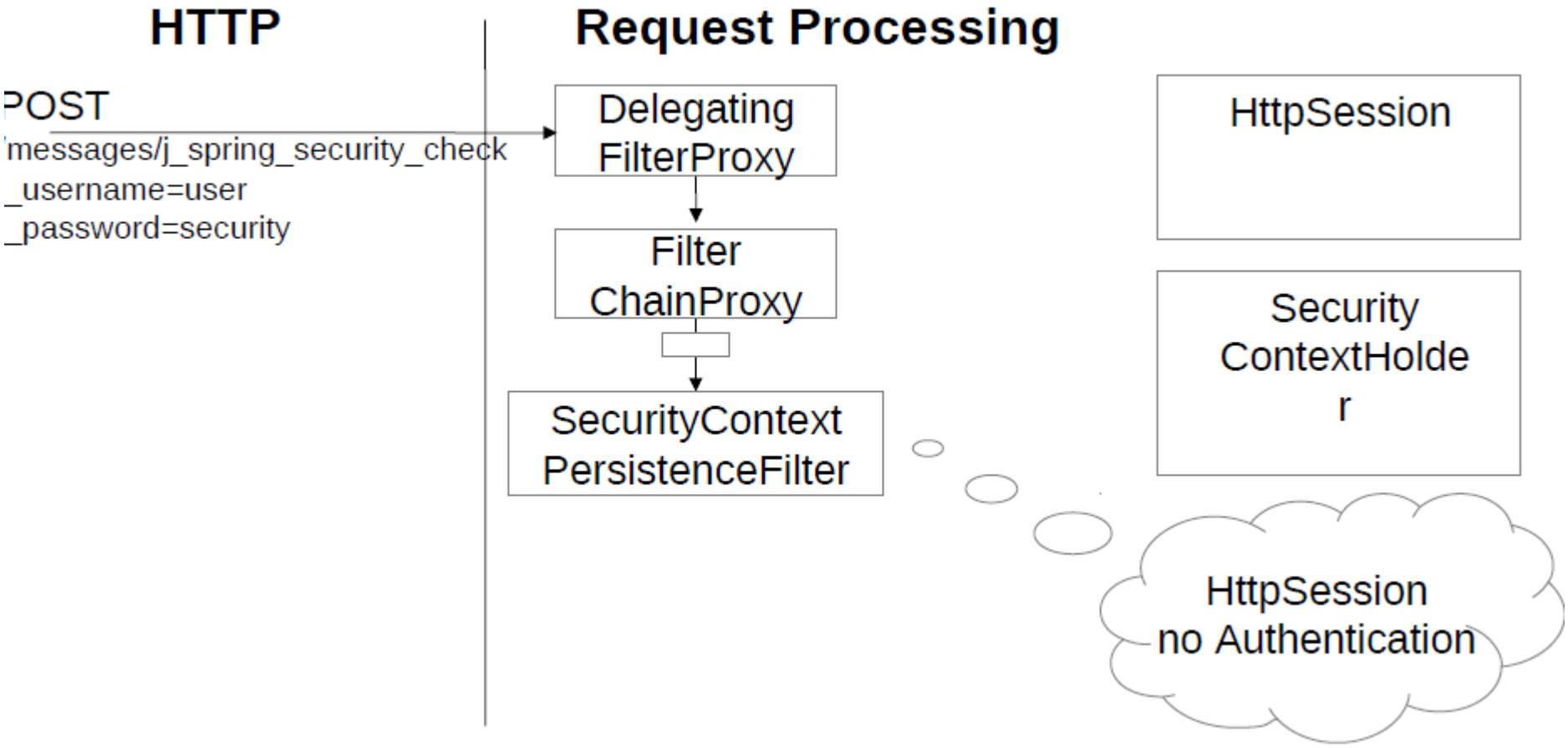
# Authentication via username and password



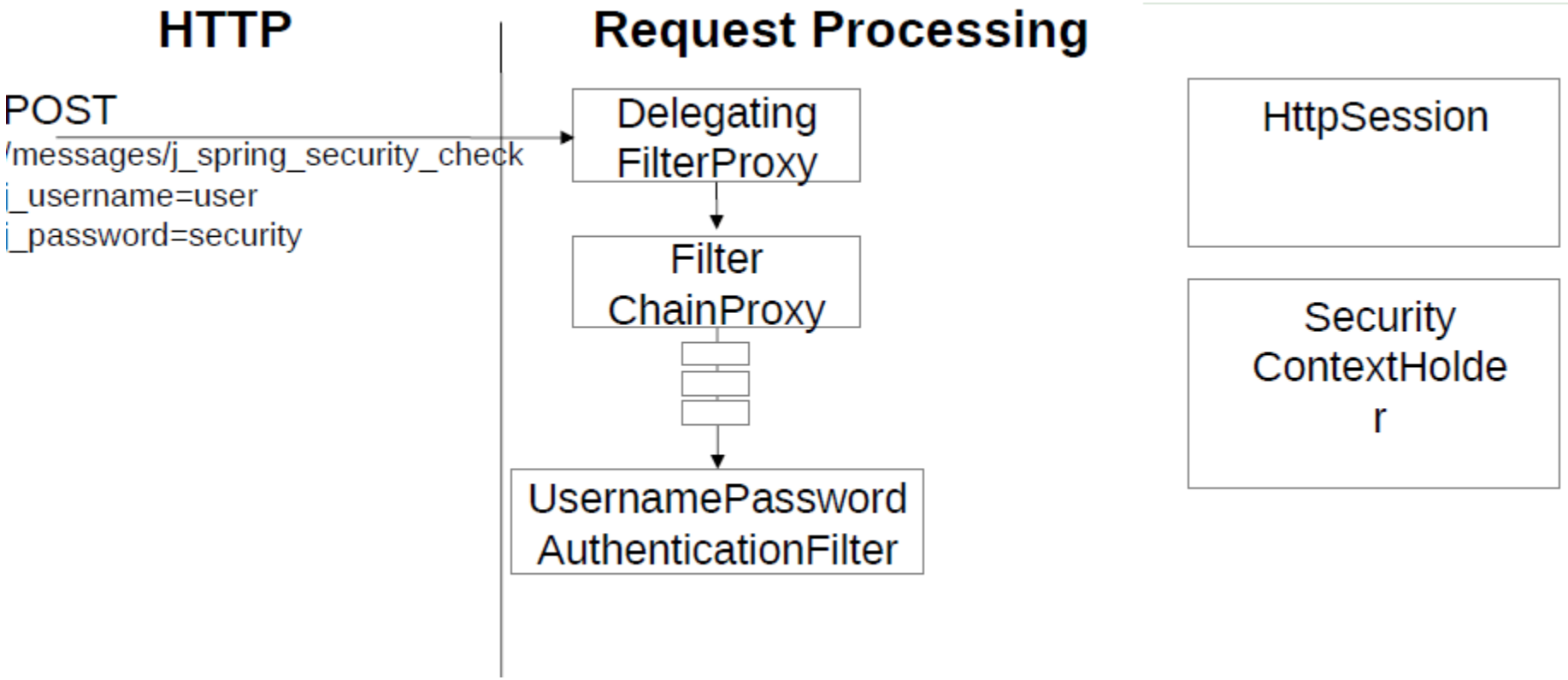
# Requesting log in page



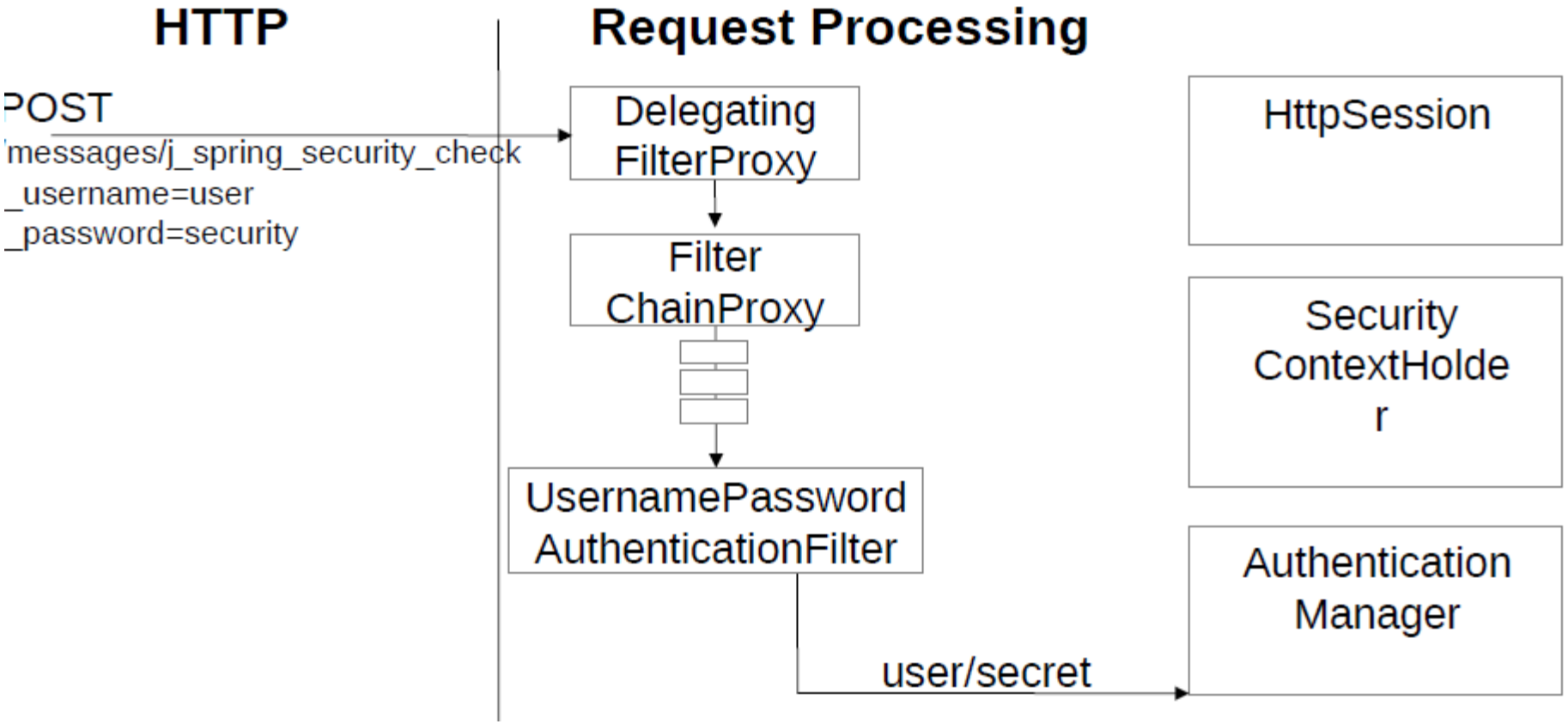
# Requesting log in page



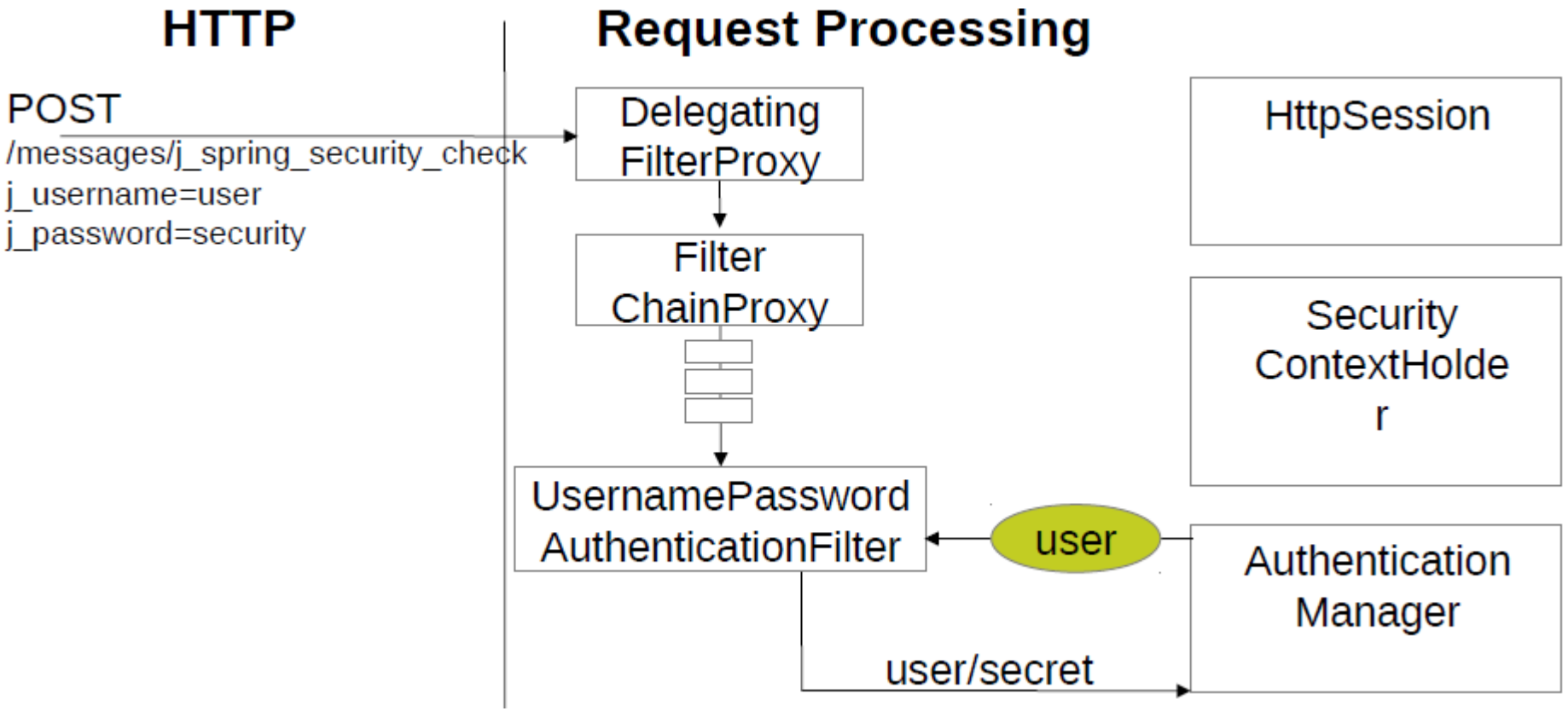
# Requesting log in page

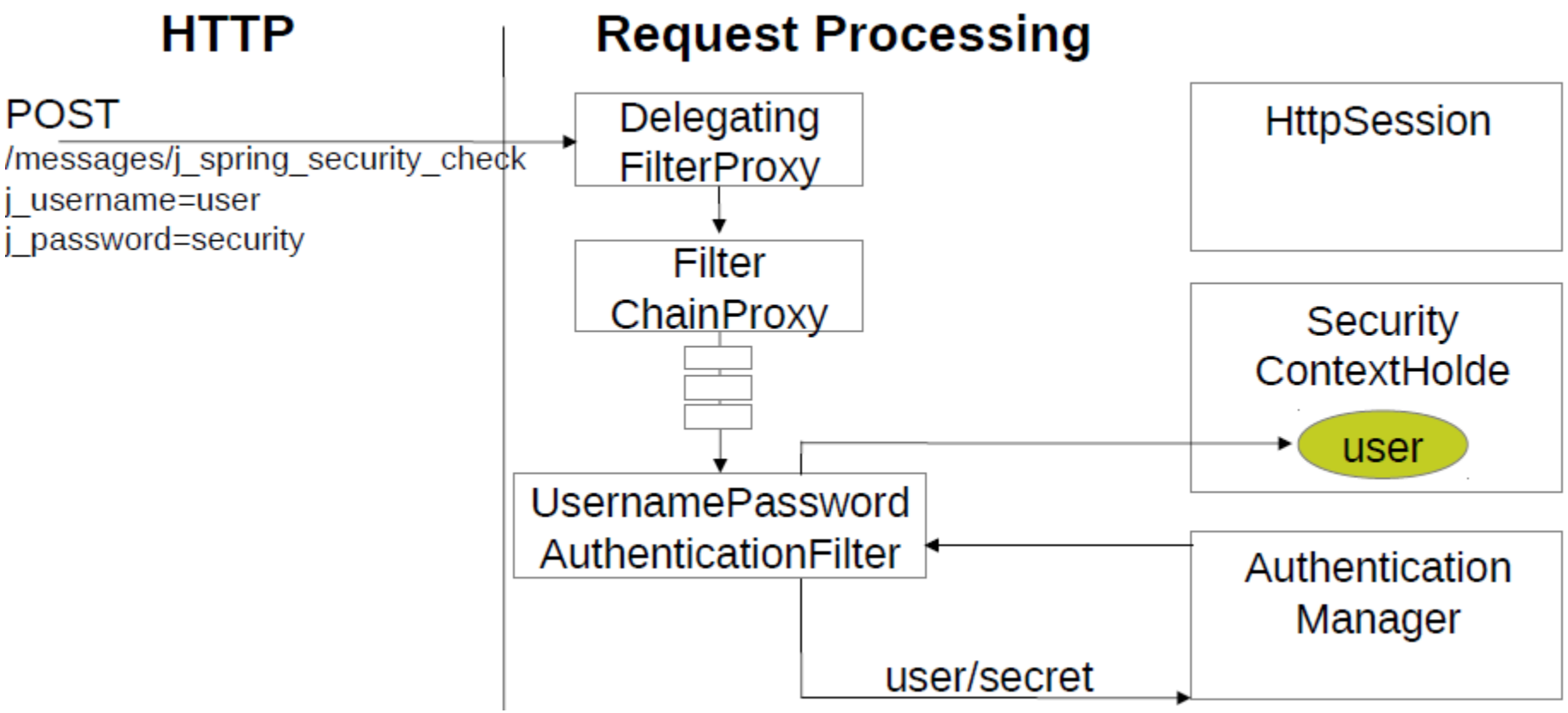


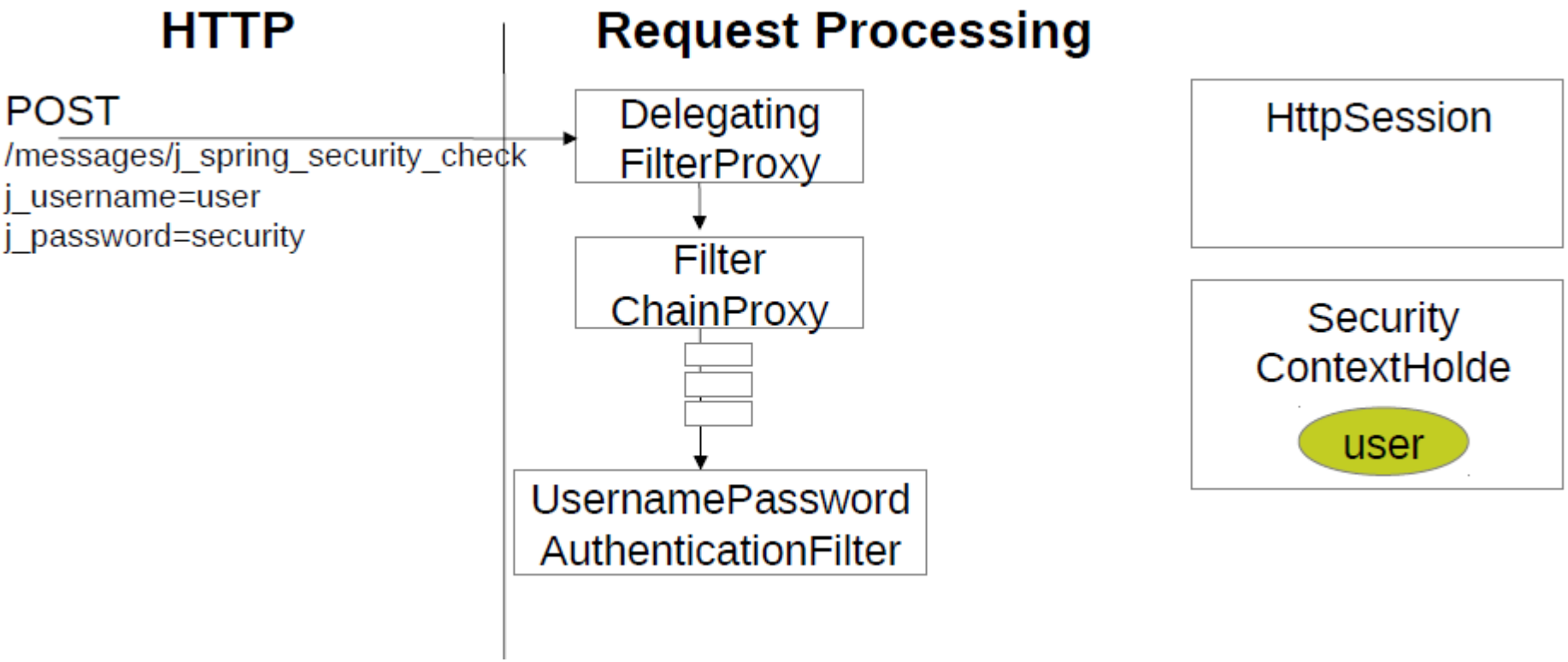
# Understanding How spring security works?

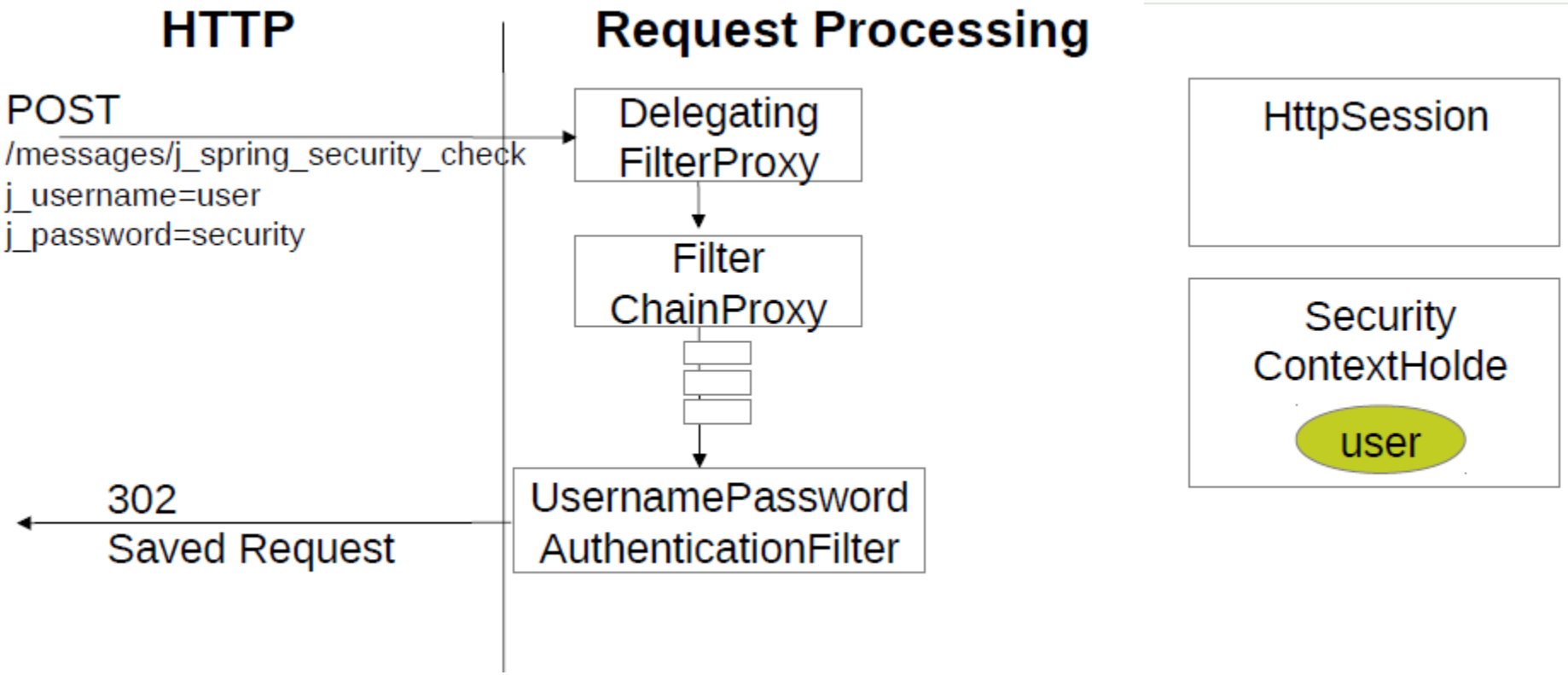


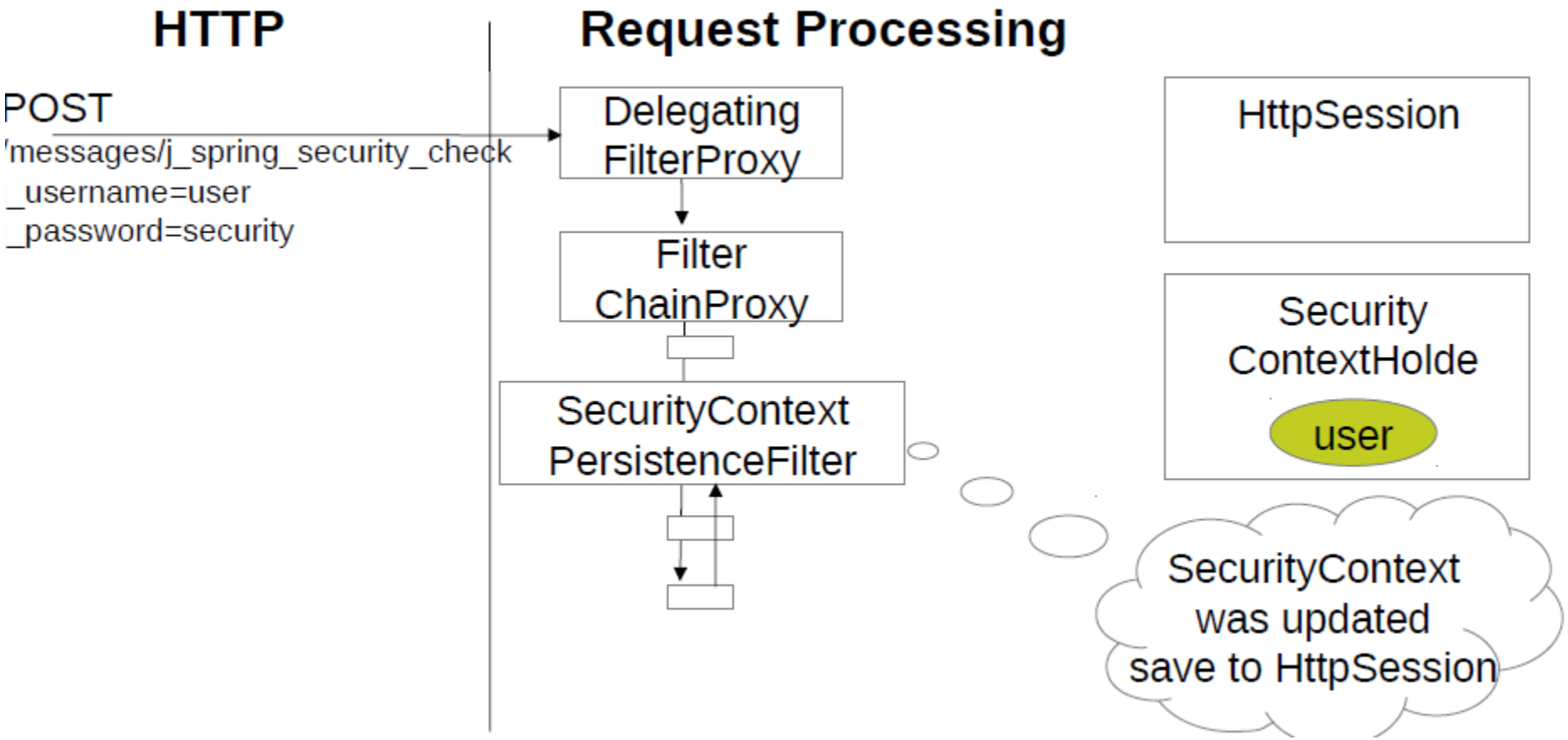
# Understanding How spring security works?

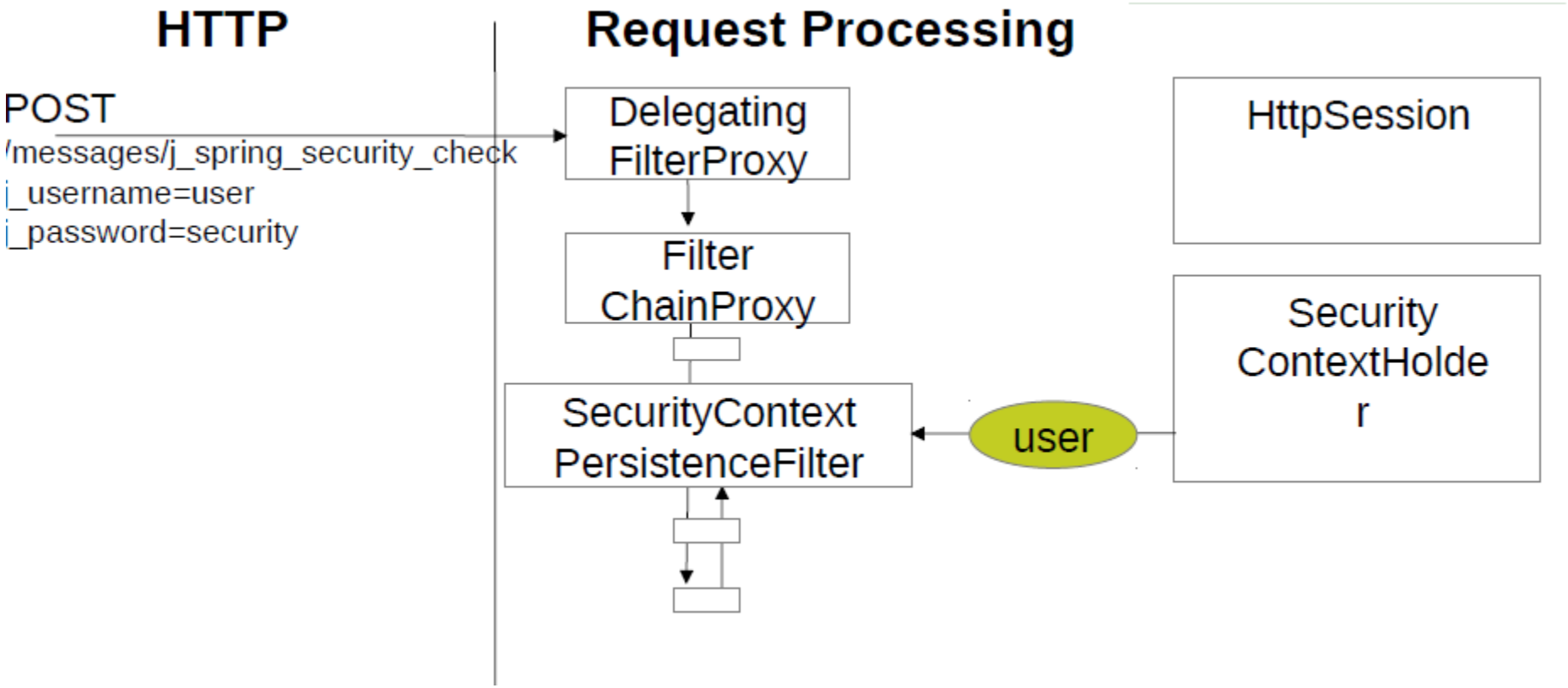


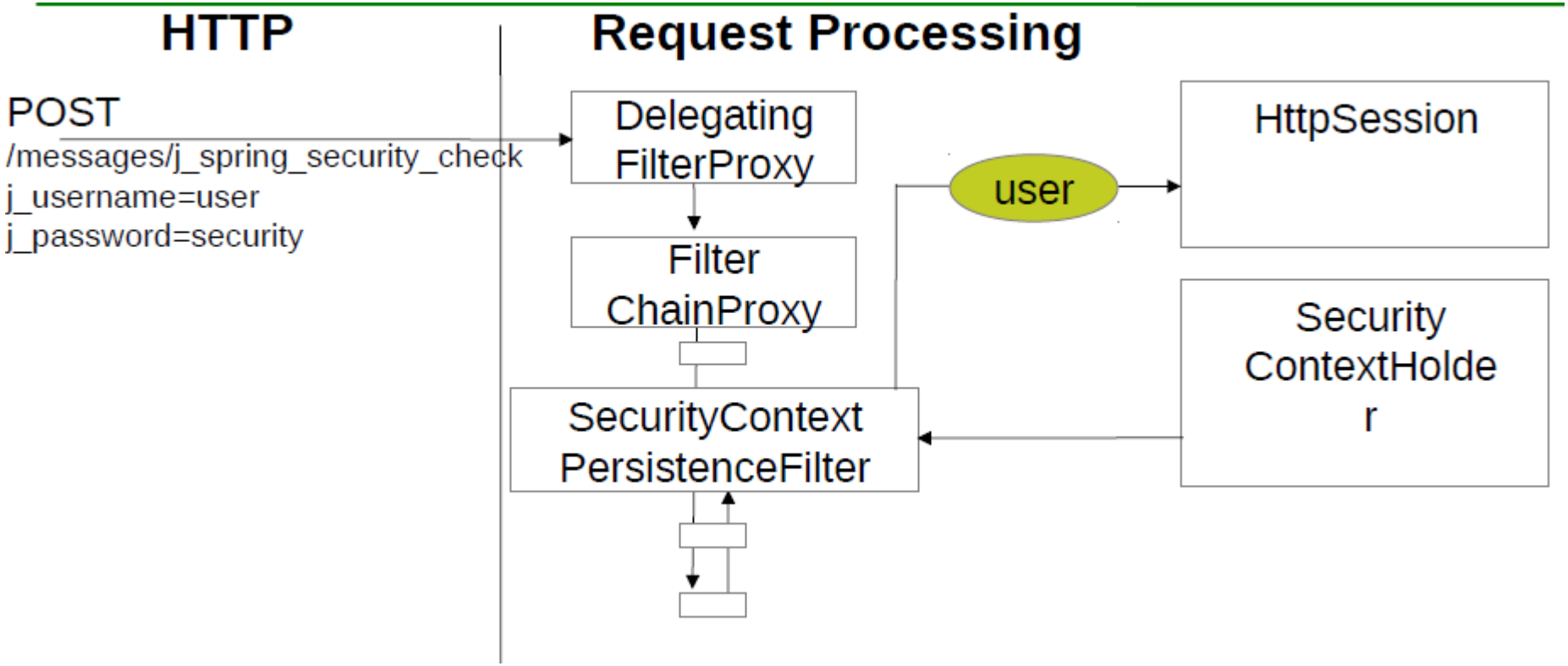

HTTP
Request Processing
POST
/messages/j_spring_security_check
j_username=user
j_password=security
Delegating
FilterProxy
Filter
ChainProxy
UsernamePassword
AuthenticationFilter
HttpSession
Security
ContextHolde
user
Authentication
Manager
user/secret

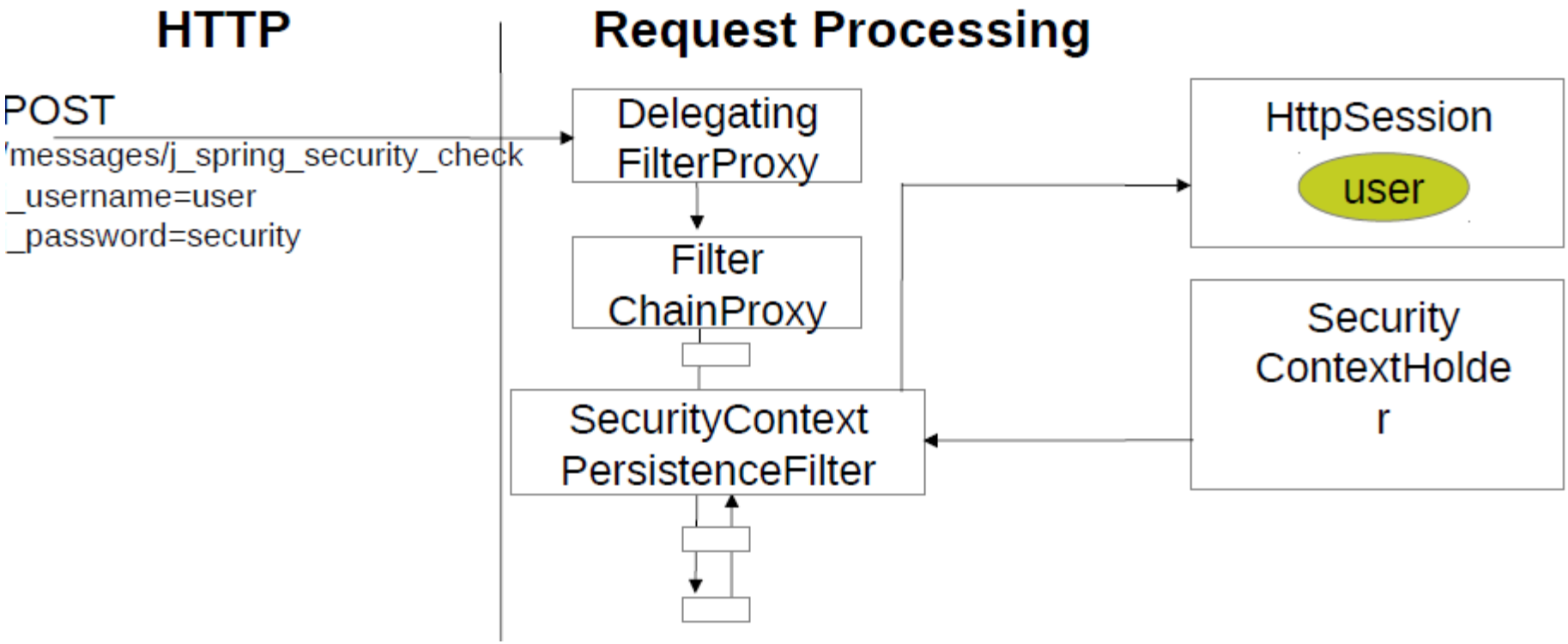

HTTP
Request Processing
POST
/messages/j_spring_security_check
j_username=user
j_password=security
Delegating
FilterProxy
Filter
ChainProxy
UsernamePassword
AuthenticationFilter
HttpSession
Security
ContextHolde
user

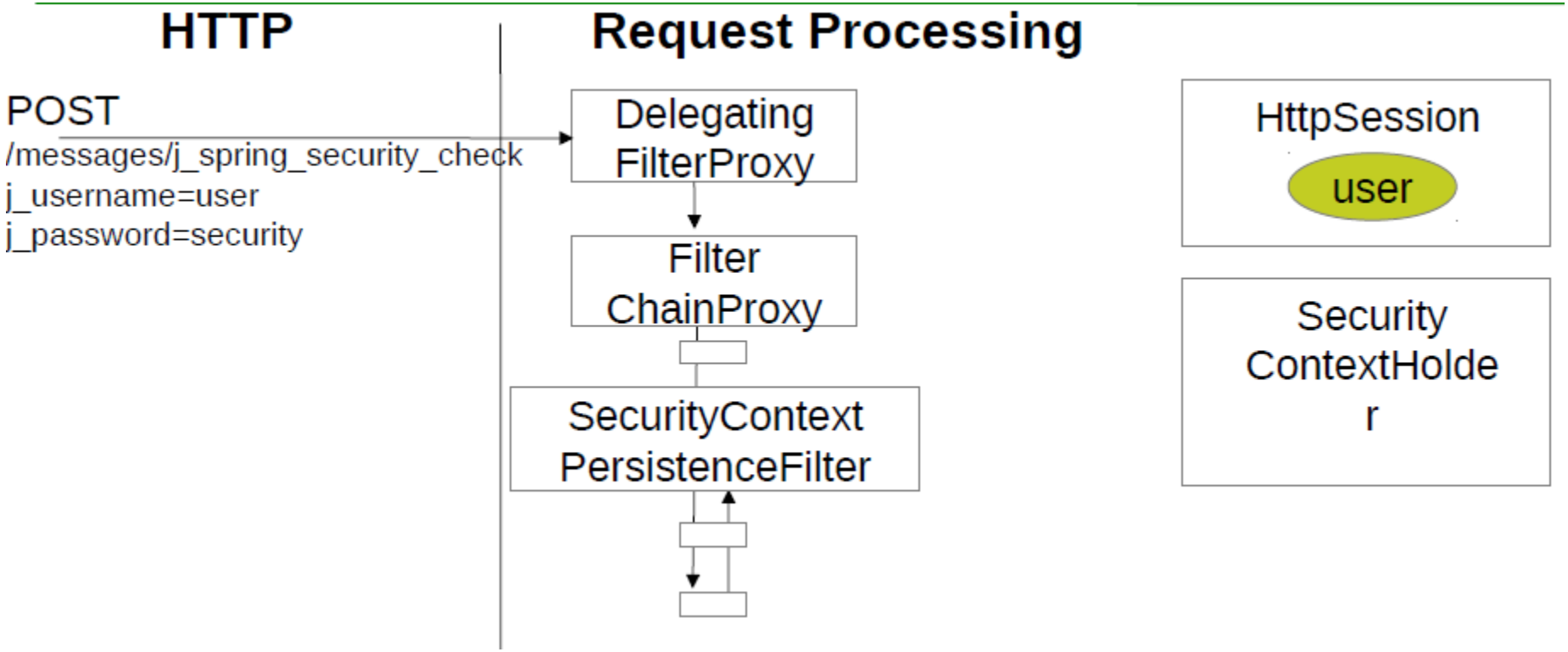

HTTP
Request Processing
POST
/messages/j_spring_security_check
j_username=user
j_password=security
Delegating
FilterProxy
Filter
ChainProxy
UsernamePassword
AuthenticationFilter
302
Saved Request
HttpSession
Security
ContextHolde
user

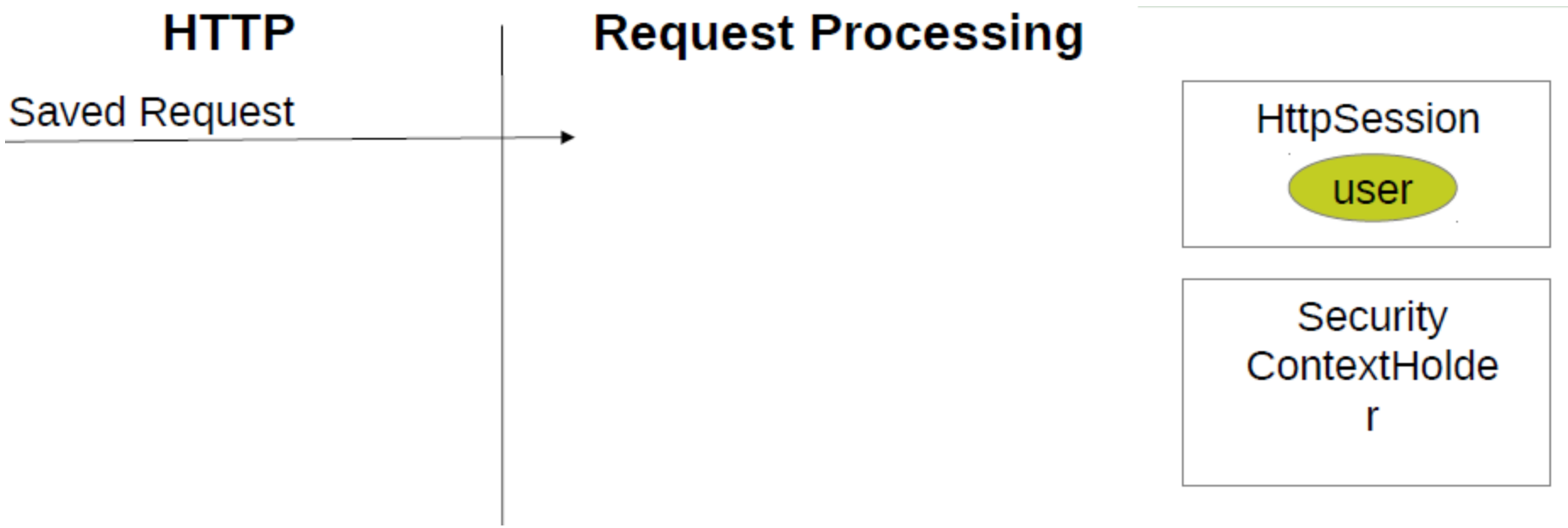

HTTP
Request Processing
POST
'messages/j_spring_security_check
_username=user
_password=security
Delegating
FilterProxy
Filter
ChainProxy
SecurityContext
PersistenceFilter
HttpSession
Security
ContextHolde
user
SecurityContext
was updated
save to HttpSession

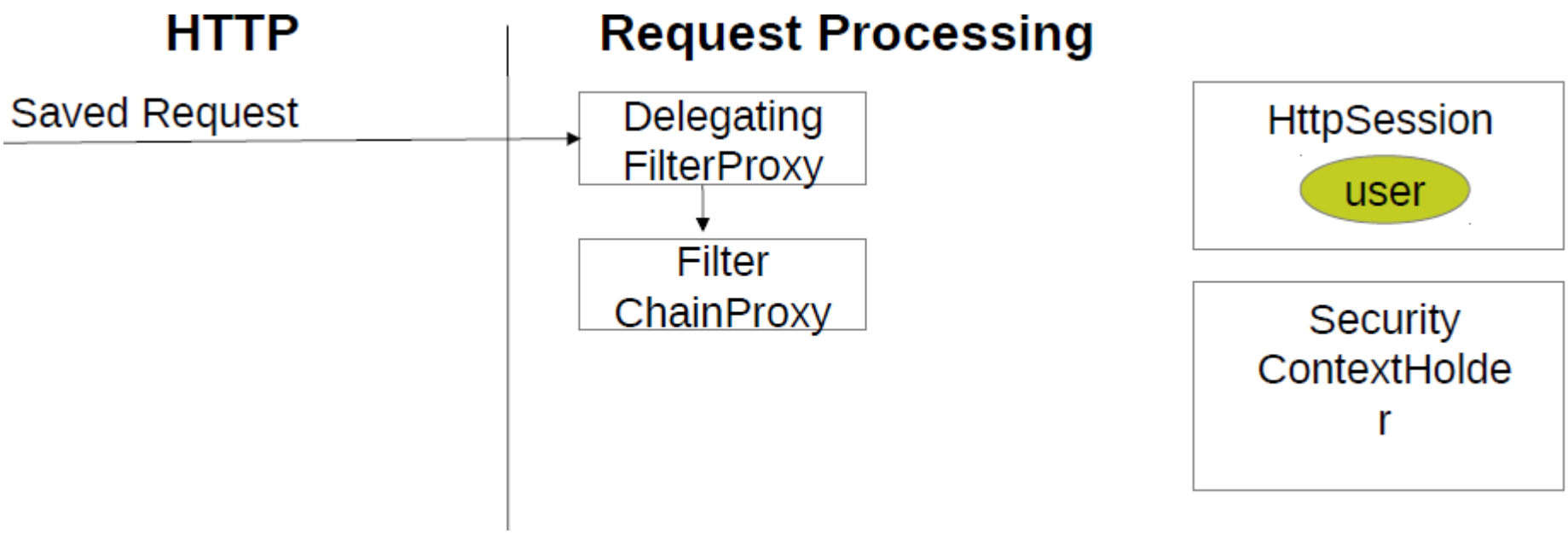

HTTP
Request Processing
POST
/messages/j_spring_security_check
j_username=user
j_password=security
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
user
HttpSession
Security ContextHolde r

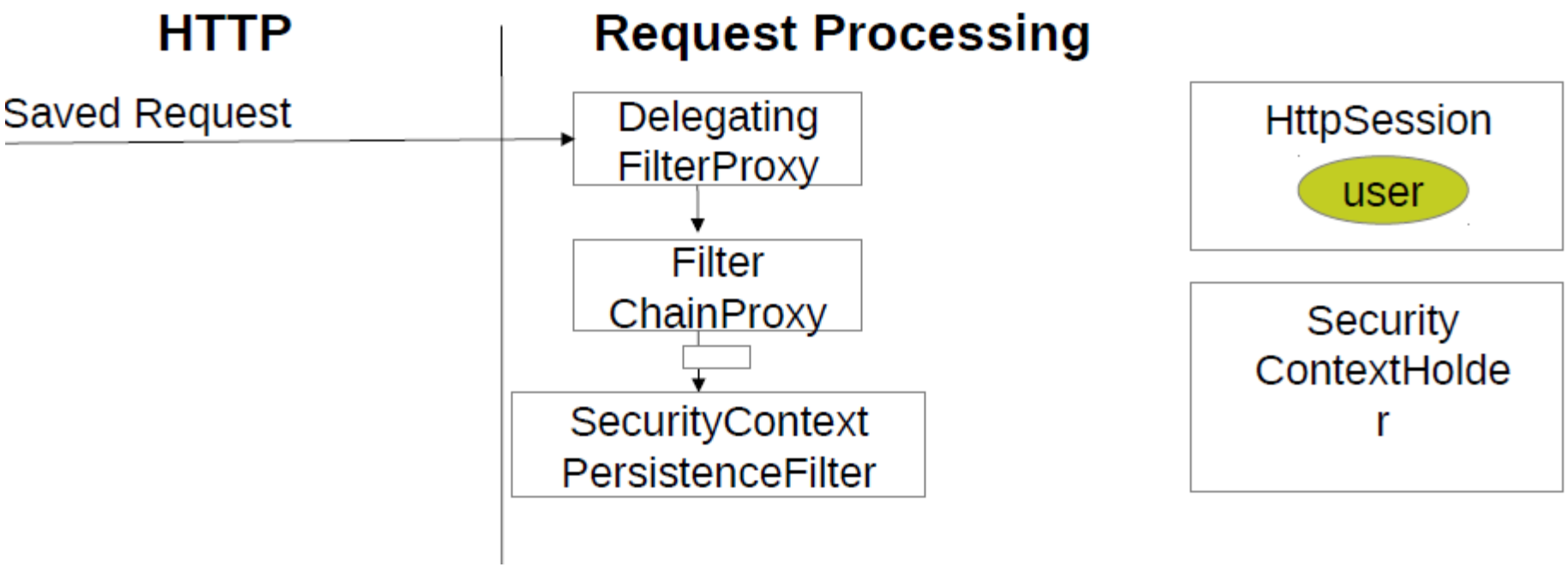

HTTP
Request Processing
POST
/messages/j_spring_security_check
j_username=user
j_password=security
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
user
HttpSession
Security ContextHolde r

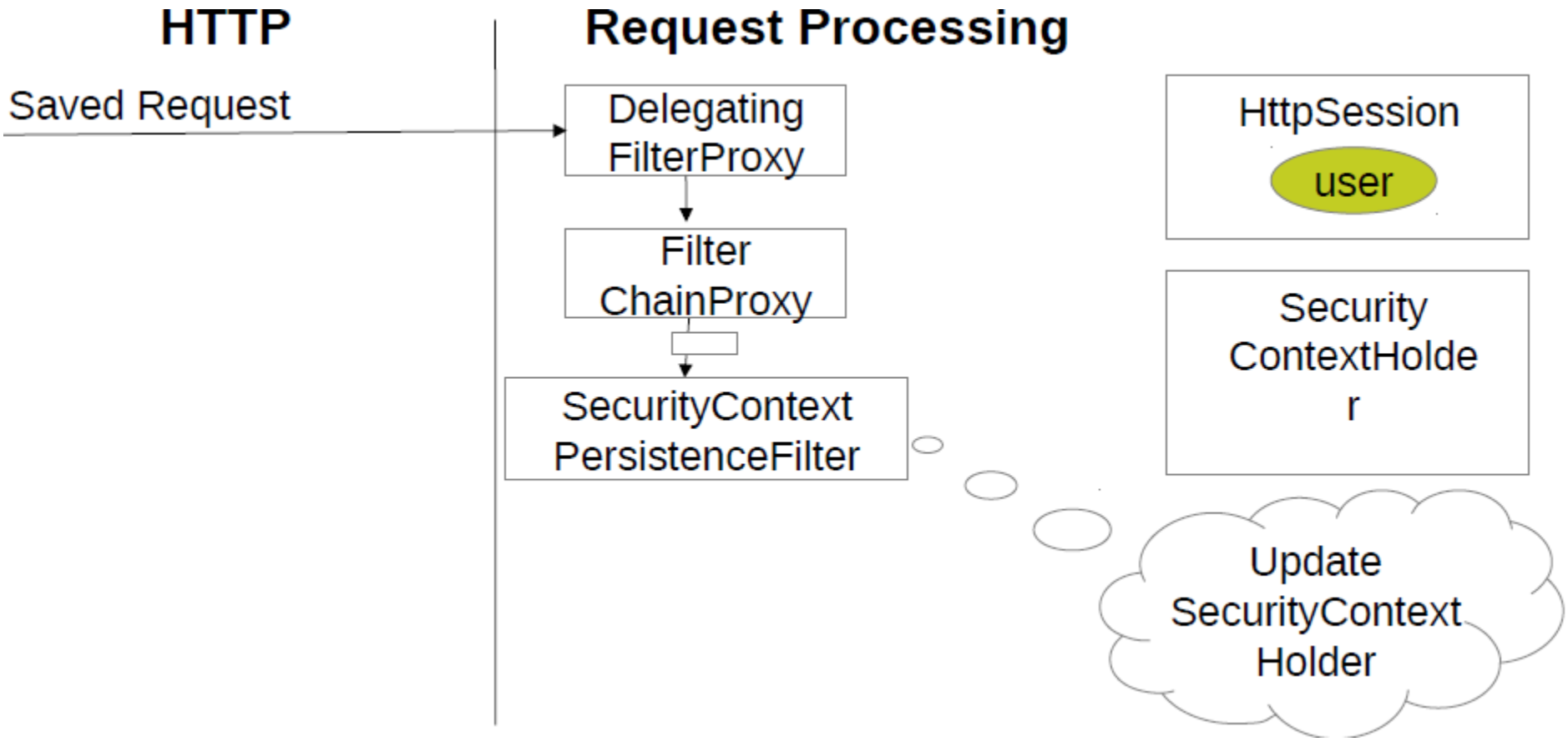

HTTP
Request Processing
POST
'messages/j_spring_security_check
_username=user
_password=security
Delegating
FilterProxy
Filter
ChainProxy
SecurityContext
PersistenceFilter
HttpSession
user
Security
ContextHolde
r

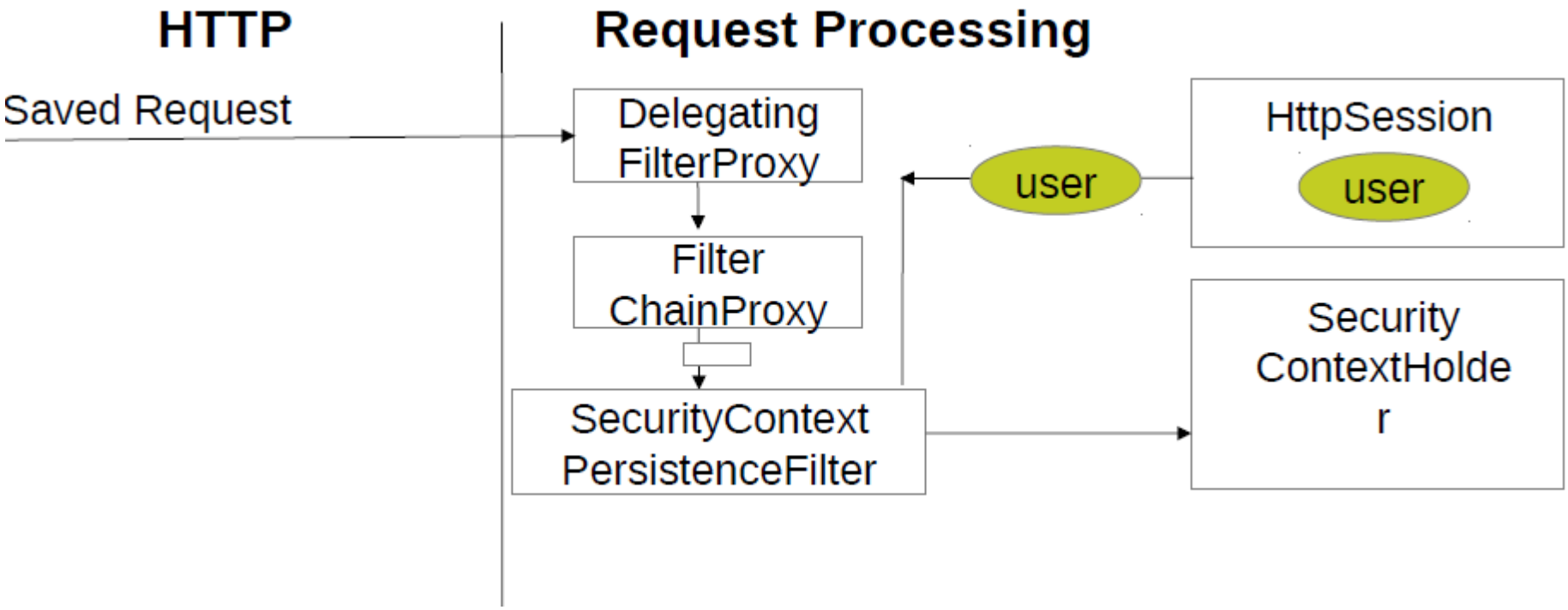

HTTP
Request Processing
POST
/messages/j_spring_security_check
j_username=user
j_password=security
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde r

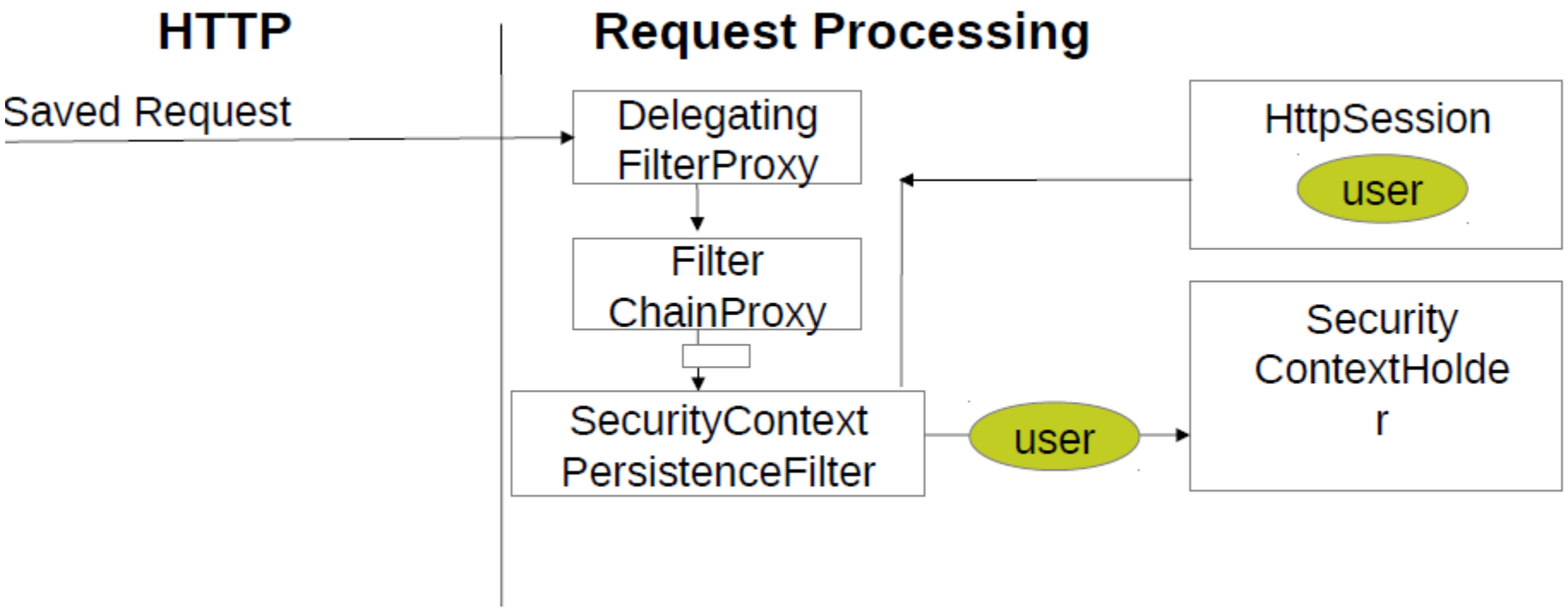

HTTP
Request Processing
Saved Request
HttpSession
user
Security
ContextHolde
r

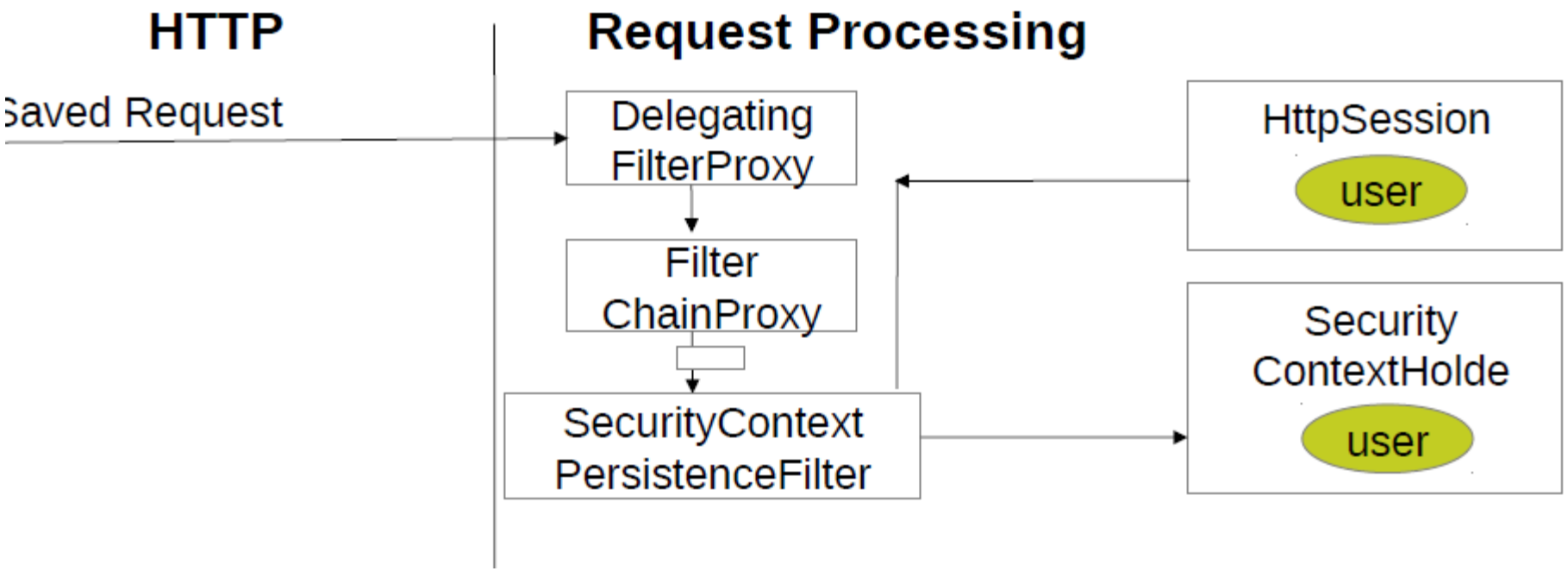

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
HttpSession
user
Security ContextHolde r

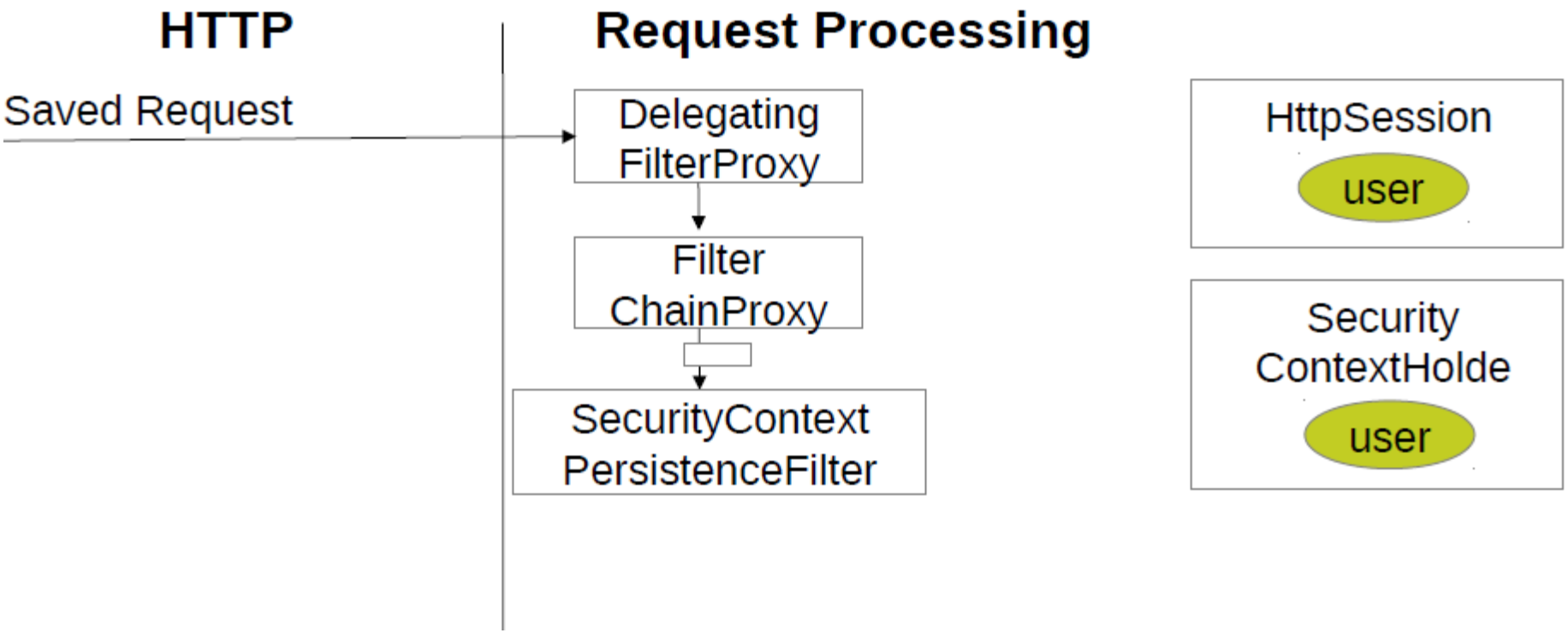

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde r

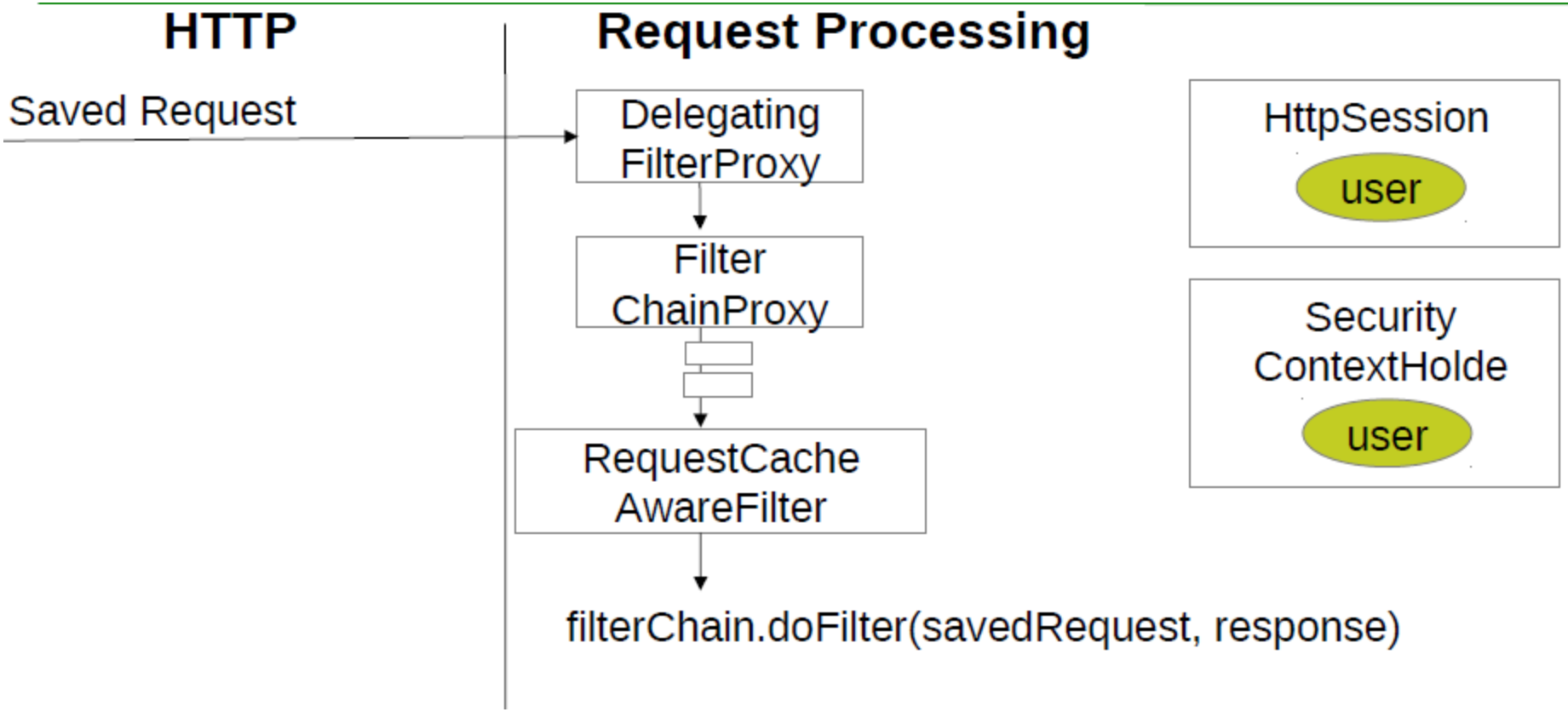

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde r
Update SecurityContext Holder

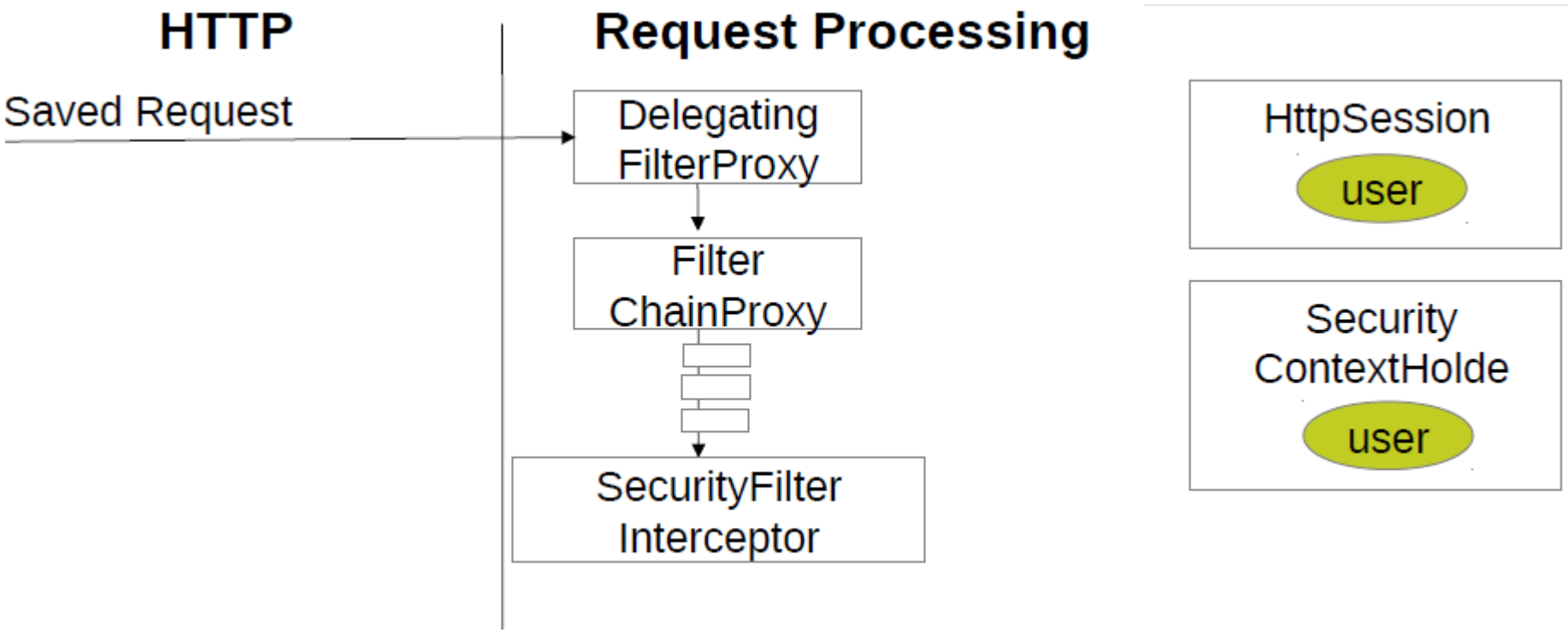

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
user
HttpSession
user
Security ContextHolde r

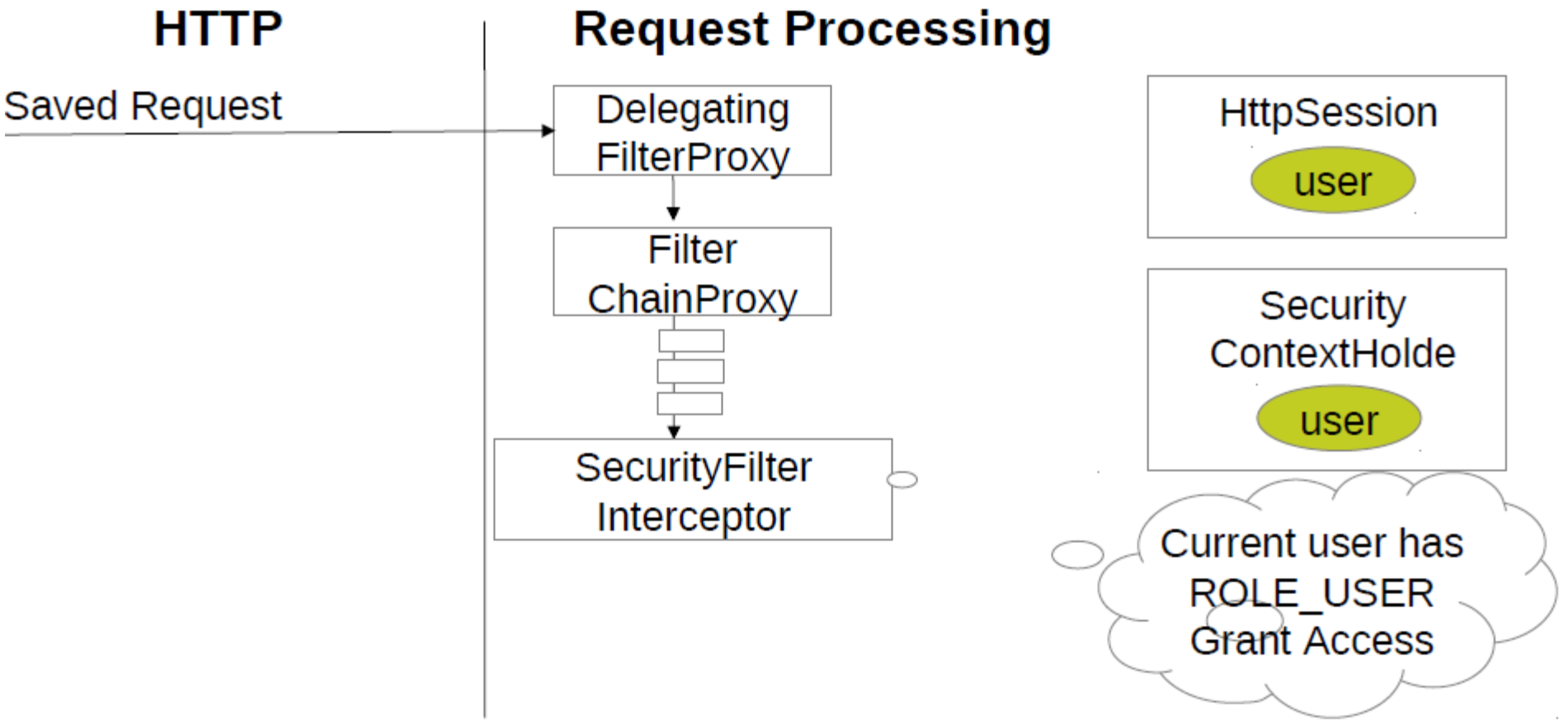

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde r
user

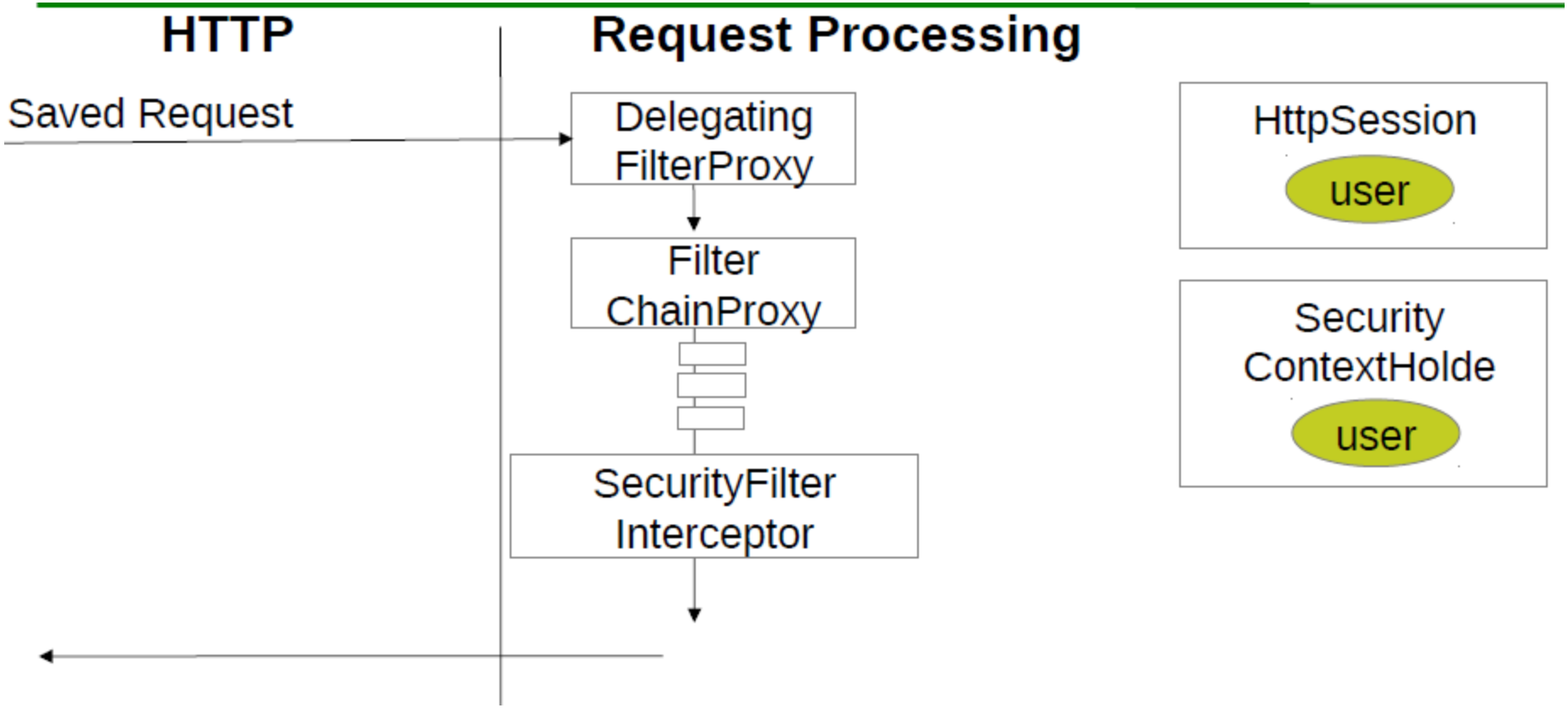

HTTP
Request Processing
Saved Request
Delegating
FilterProxy
Filter
ChainProxy
SecurityContext
PersistenceFilter
HttpSession
user
Security
ContextHolde
user

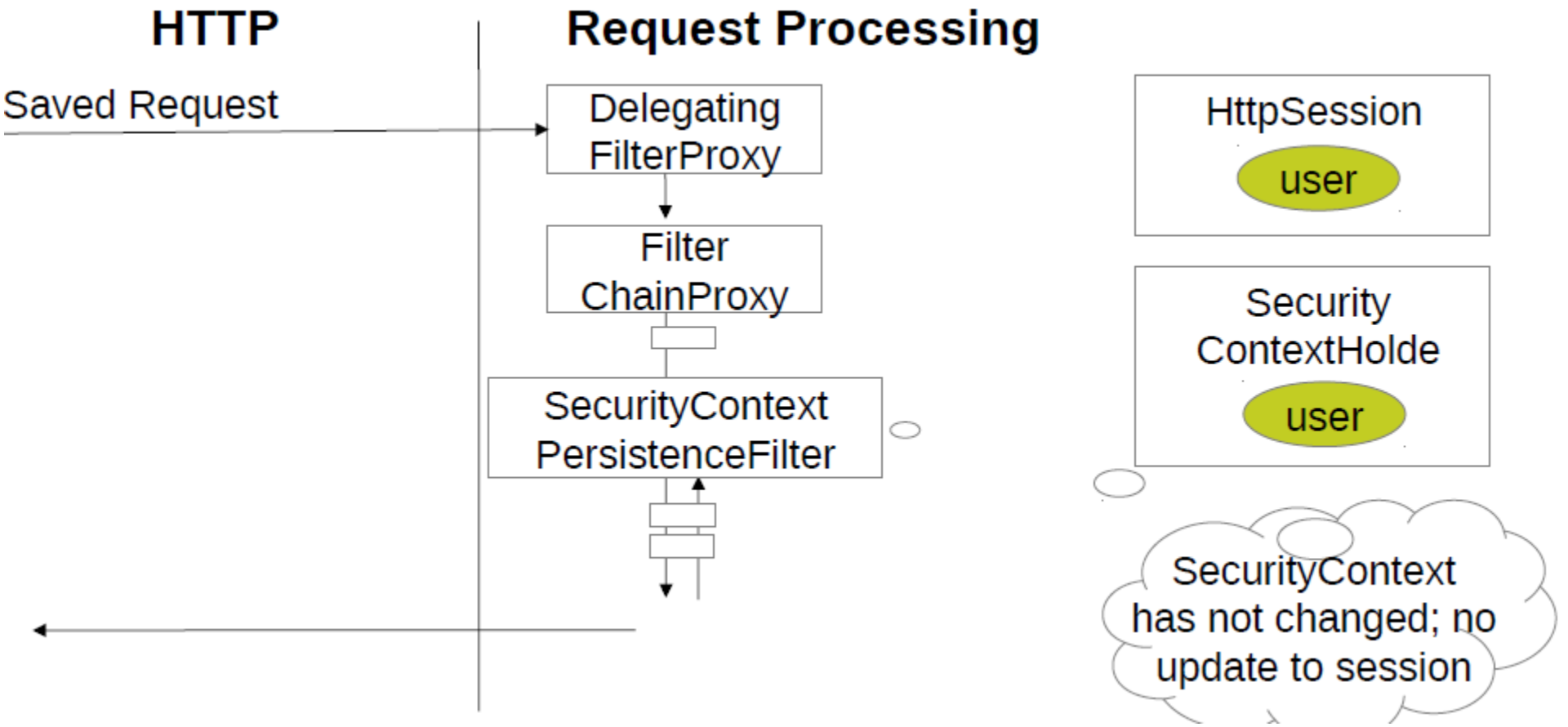

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde
user

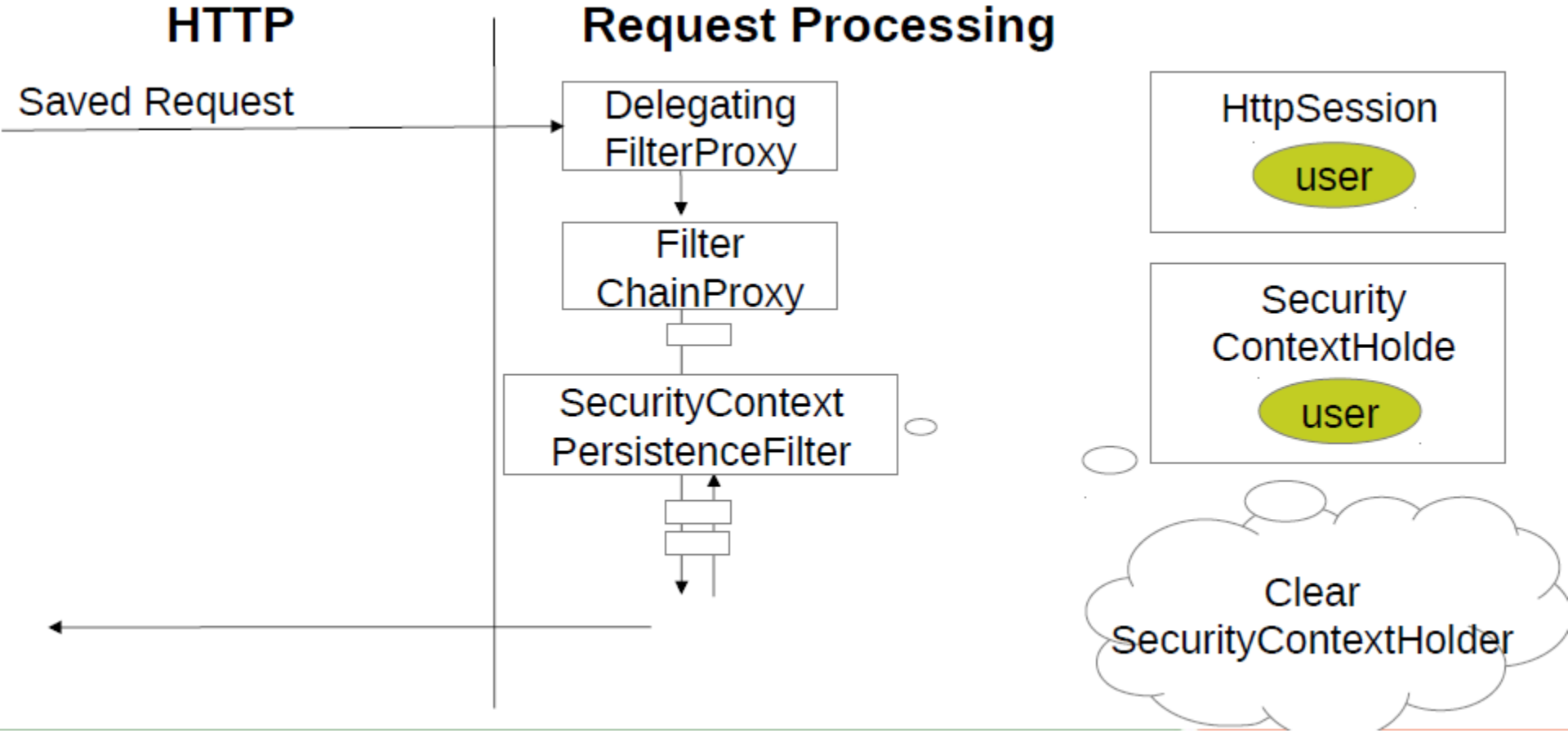

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
RequestCache AwareFilter
filterChain.doFilter(savedRequest, response)
HttpSession
user
Security ContextHolde
user

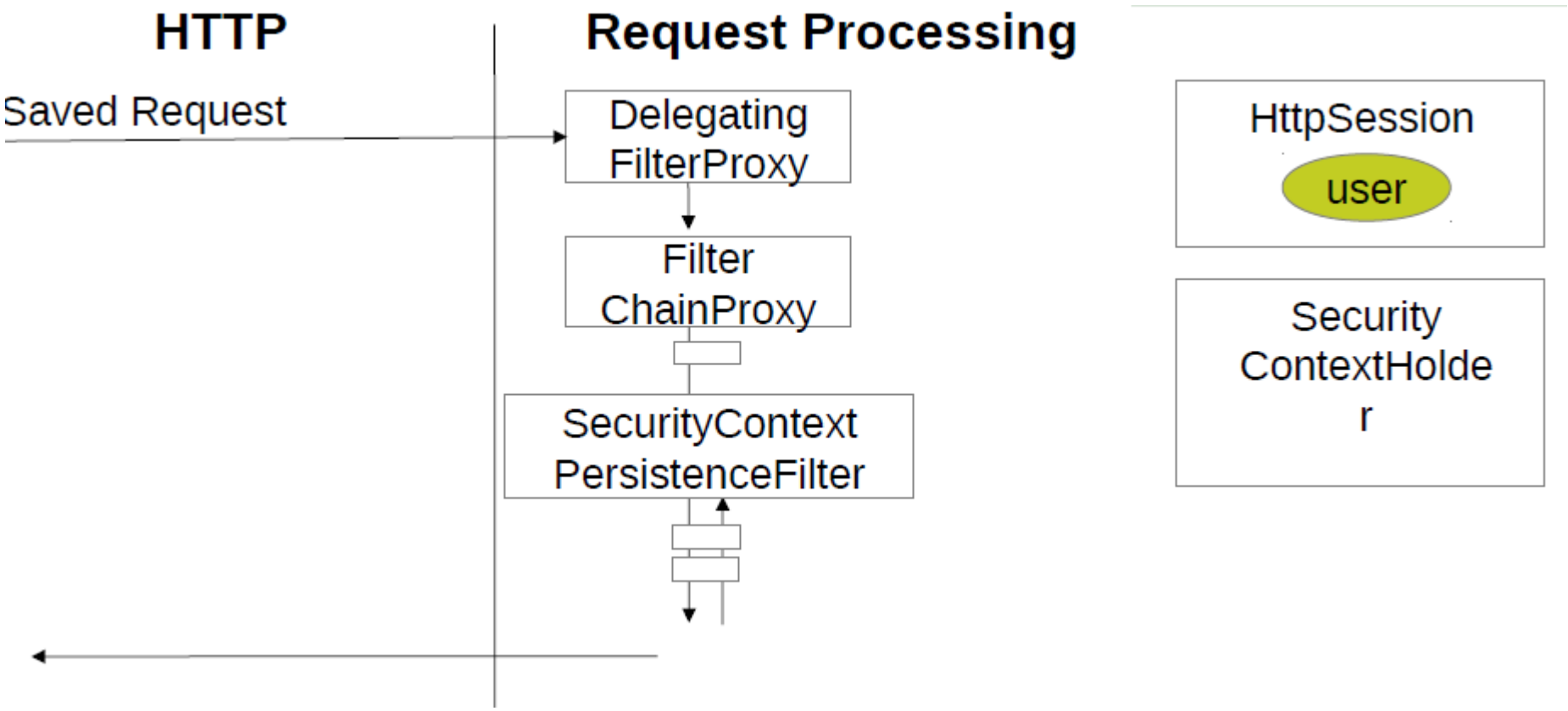

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityFilter Interceptor
HttpSession
user
Security ContextHolde
user

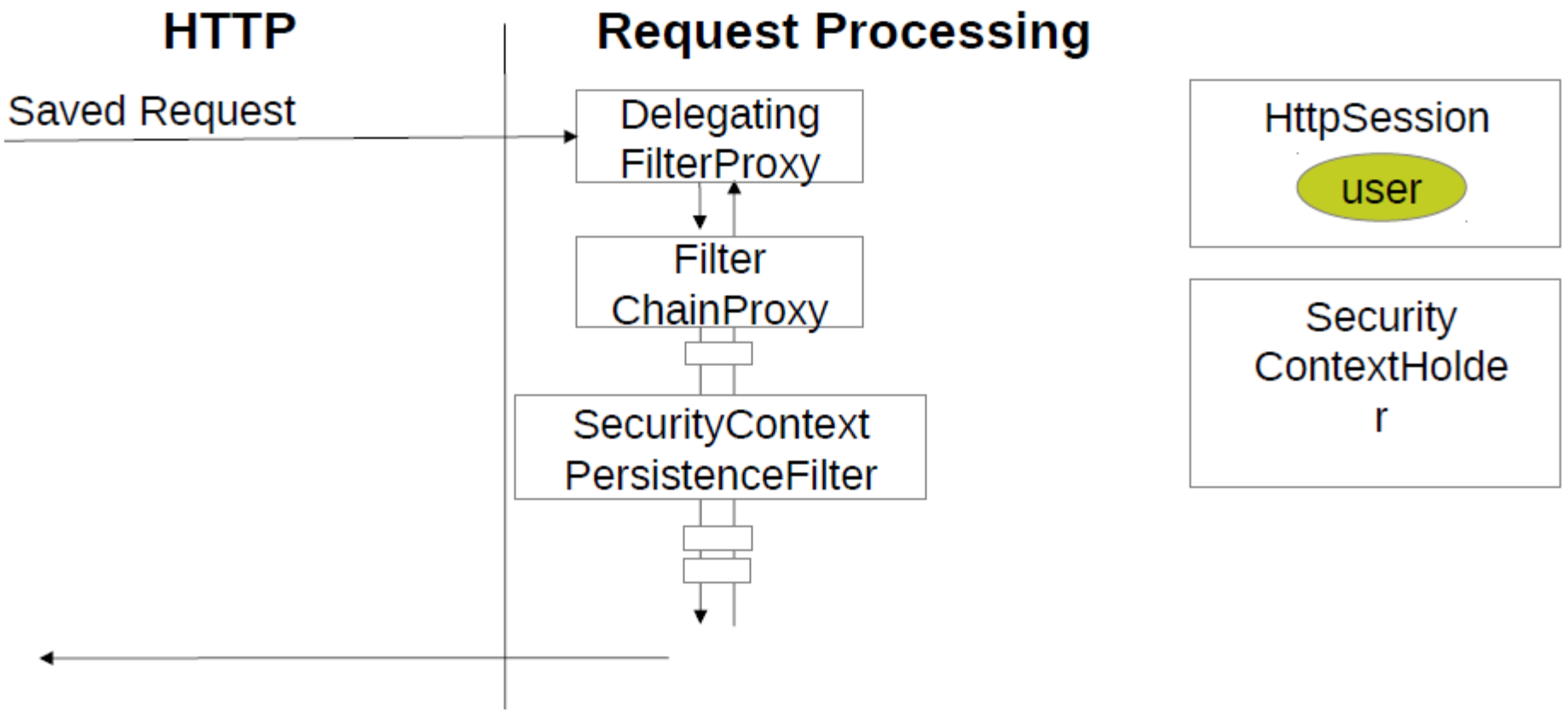

HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityFilter Interceptor
HttpSession
user
Security ContextHolde
user
Current user has ROLE_USER Grant Access


HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityFilter Interceptor
HttpSession
user
Security ContextHolde
user


HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde
user
SecurityContext has not changed; no update to session


HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde
user
Clear SecurityContextHolder


HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde r


HTTP
Request Processing
Saved Request
Delegating FilterProxy
Filter ChainProxy
SecurityContext PersistenceFilter
HttpSession
user
Security ContextHolde r

# Getting started with Spring Security

- No matter what kind of application you want to secure using Spring Security, the first thing to do is to add the Spring Security modules to the application's classpath. Spring Security 3.0 is divided into eight modules, as listed
- At the least, you'll want to include the Core and Configuration modules in your application's classpath. Spring Security is often used to secure web applications.

**Table 9.1 Spring Security is partitioned into eight modules.**

| Module | Description |
|---|---|
| ACL | Provides support for domain object security through access control lists (ACLs) |
| CAS Client | Provides integration with JA-SIG's Central Authentication Service (CAS) |
| Configuration | Contains support for Spring Security's XML namespace |
| Core | Provides the essential Spring Security library |
| LDAP | Provides support for authentication using the Lightweight Directory Access Protocol (LDAP) |
| OpenID | Provides integration with the decentralized OpenID standard |
| Tag Library | Includes a set of JSP tags for view-level security |
| Web | Provides Spring Security's filter-based web security support |

# Configuration

- For an Simple web application, we've separated all of the security-specific configuration into a separate Spring configuration file called xxx-security.xml. Since all of the configuration in this file will be from the security namespace, we've changed the security namespace to be the primary namespace for that file

**Listing 9.2 Using the security namespace as the default namespace**

```
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security-3.0.xsd">

</beans:beans>
```

Non-prefixed security elements go here

# *Securing web requests*

- Security in an typical web application start with an HttpServletRequest
- The most basic form of request-level security involves declaring one or more URL patterns as requiring some level of granted authority and preventing users without that authority from accessing the content behind those URLs. We may require that certain URLs can only be accessed over HTTPS
- Before you can restrict access to users with certain privileges, there must be a way to know who’s using the application. Therefore, the application will need to authenticate the user, prompting them to log in and identify themselves.

# *Proxying servlet filters*

- Spring Security employs several servlet filters to provide various aspects of security , but we do not need to configure all ( Relax!)
- We only need to configure one filter in the application's web.xml file (Spring magic). Specifically, we'll need to add the following <filter>

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

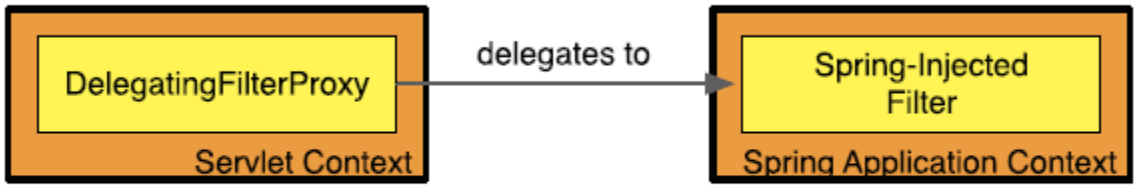



**Figure 9.1 `DelegatingFilterProxy` proxies filter handling to a delegate filter bean in the Spring application context.**

# How it works?

- DelegatingFilterProxy is a special servlet filter that, by itself, doesn’t do much. Instead, it delegates to an implementation of javax.servlet.
- Filter that’s registered as a <bean> in the Spring application context
- The value given as DelegatingFilterProxy’s <filter-name> is significant. This is the name used to look up the filter bean from the Spring application context.
- Spring Security will automatically create a filter bean whose ID is springSecurityFilter- Chain, so that’s the name we’ve given to DelegatingFilterProxy in web.xml

# Configuring minimal web security

- We need to use bellow snippet of XML for configuration of Spr

```
<http auto-config="true">
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

- These humble three lines of XML configure Spring security to intercept requests for all URLs (as specified by the Ant-style path in the pattern attribute of <intercept- url>) and restrict access to only authenticated users who have the ROLE_SPITTER role.
- The <http> element automatically sets up a FilterChainProxy (which is delegated to by the DelegatingFilterProxy we configured in web.xml) and all of the filter beans in the chain.
- In addition to those filter beans, we also get a few more freebies by setting the auto-config attribute to true. Autoconfiguration gives our application a free
- logg
- aski

```
<http>
    <form-login />
    <http-basic />
    <logout />
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

ort for explicitly