Spring Security

Securing Web Applications

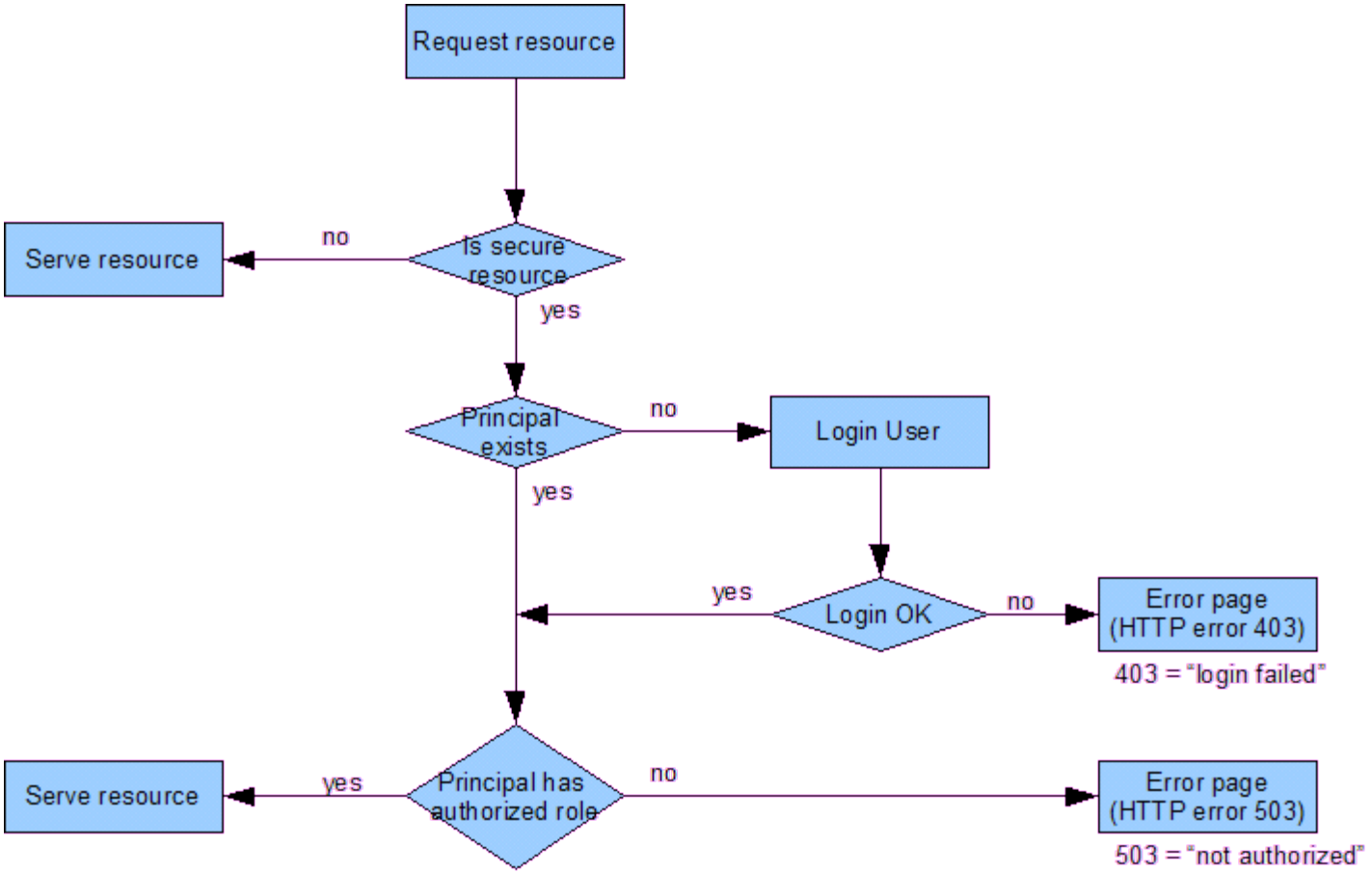
Objectives

Security Concepts

- **Authentication**
  - Authentication pertains to the question “Who are you?”.
  - Usually a user authenticates himself by successfully associating his “principal” (often a username) with his “credentials” (often a password).
- **Authorization**
  - Authorization pertains to the question “What may you do?”.
  - In J2EE applications, this is achieved by making secured resources accessible (“requestable” in web applications) to particular “roles”.
  - Principals (i.e. users) who are associated with one or more of these roles will have access to those resources.

The motions of a secured web application

- The diagram below shows the typical rundown of accessing a web resource with security enabled



The Four Checks

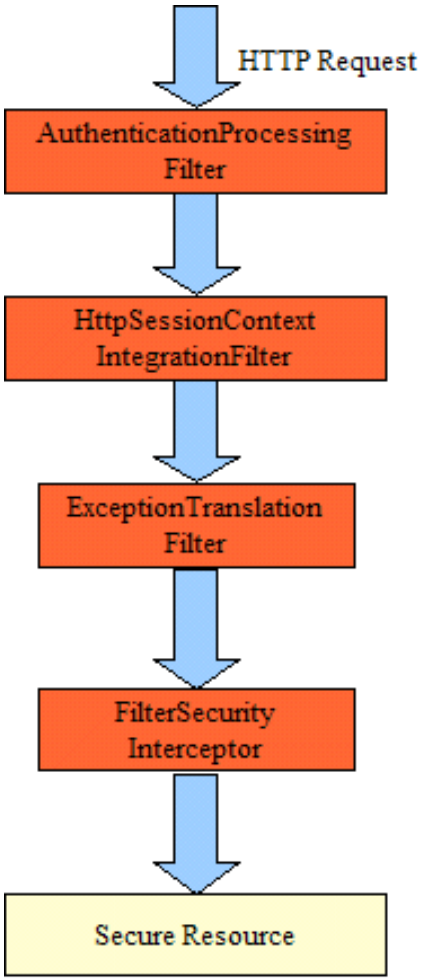
- To make a long story short, security is implemented by these Four Checks:
  - the **Restricted Access** Check
    - is the resource secured?
  - the **Existing Authentication** Check
    - has the user been authenticated?
  - if there is no valid login for the user: the **Authentication Request** Check
    - are the correct username and password provided?
  - the **Authorization** Check
    - does the user have the required roles?

Spring Web Security

The Authentication object

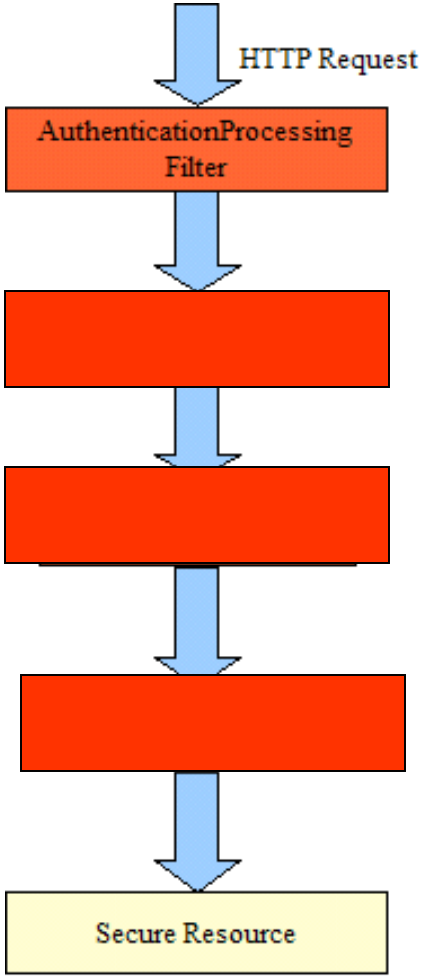
- The Authentication object is pivotal to the Spring Security framework.
- Since “security” basically means “restricted access for specific roles” the framework has to be able to determine, at any time, the roles given to the authenticated user.
- The framework stores this info in the “Authentication” object, which contains the username, password and the roles granted to the user.
- The Authentication object is created and validated by the by the AuthenticationManager.
- Access to resources is controlled by the AccessDecisionManager

Filters



- Spring Security uses a chain of (at least) three filters to enable web application security

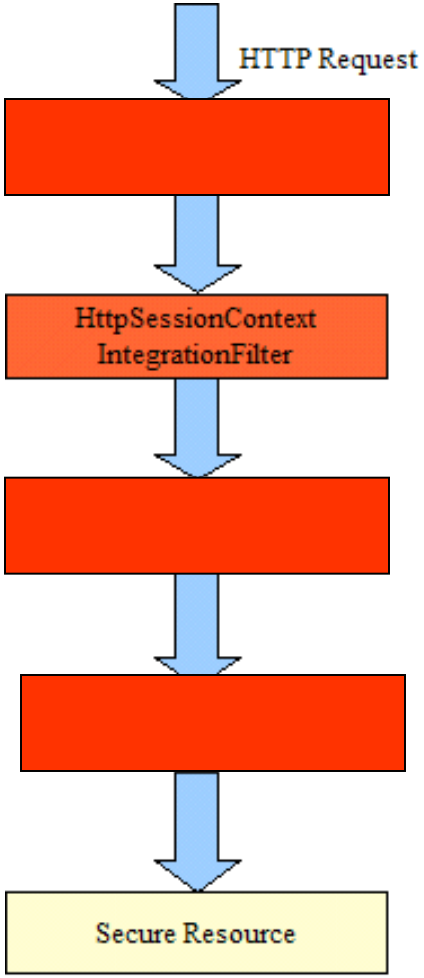
Filters



AuthenticationProcessingFilter

- Handles the Authentication Request Check ("logging into the application").
- It uses the AuthenticationManager to do its work

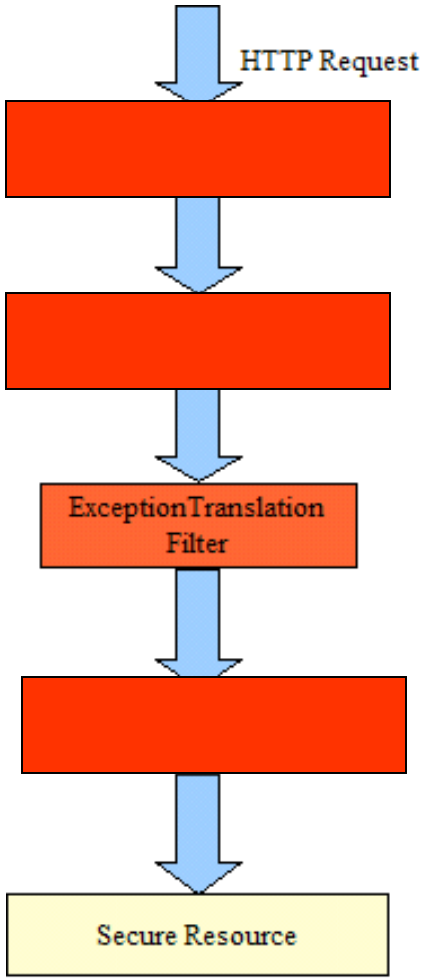
Filters



HttpSessionContextIntegrationFilter

- The HttpSessionContextIntegrationFilter maintains the Authentication object between various requests and passes it around to the AuthenticationManager and the AccessDecisionManager when needed

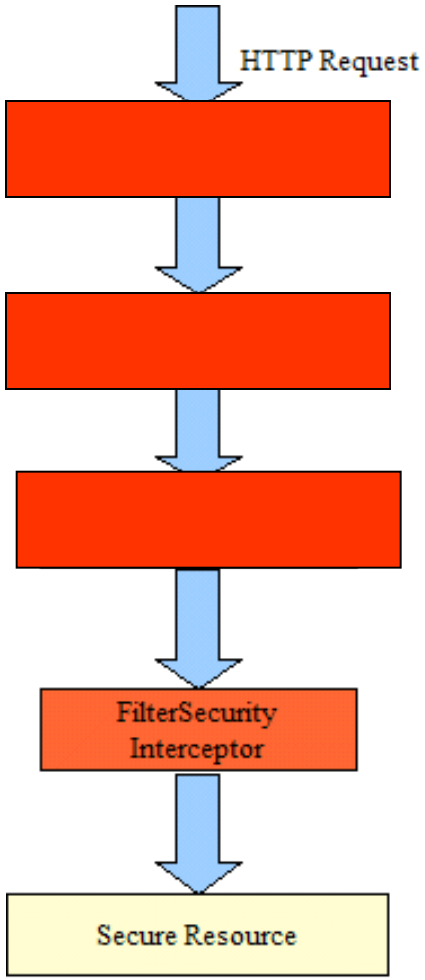
Filters



ExceptonTranslationFilter

- The ExceptonTranslationFilter performs the Existing Authentication Check, handles security exceptions and takes the appropriate action.
- This action can be either spawning the authentication dialog (a.k.a. the login form) or returning the appropriate HTTP security error code.
- ExceptonTranslationFilter depends on the next filter, FilterSecurityInterceptor, to do its work

Filters

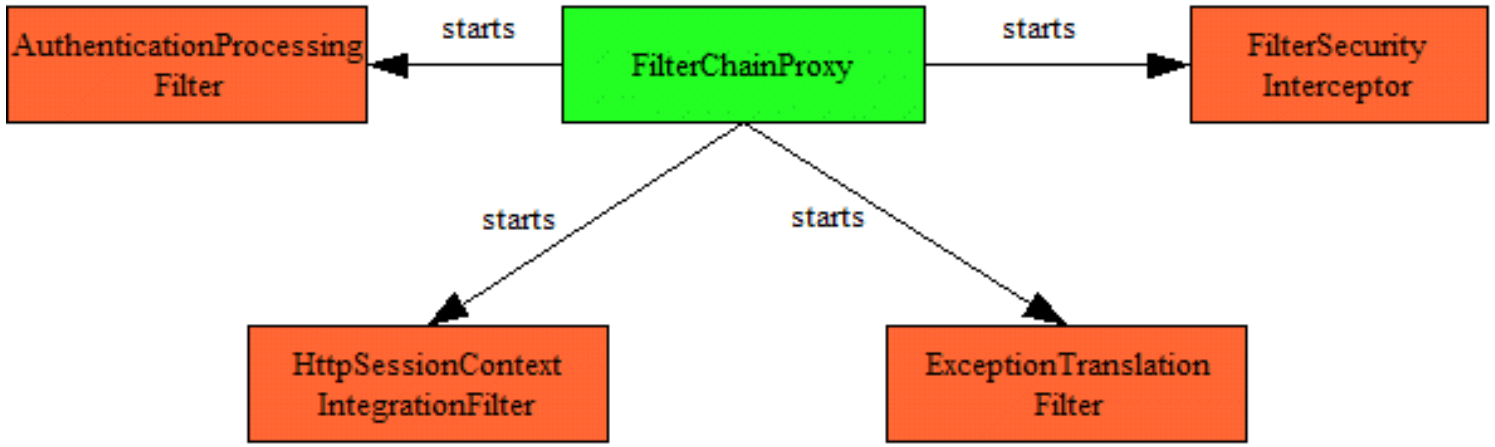


FilterSecurityInterceptor

- FilterSecurityInterceptor manages the Restricted Acces Check,and the Authorisation check.
- It knows which resources are secure and which roles have access to them. FilterSecurityInterceptor uses the AuthenticationManager and AccessDecisionManager to do its work

Filter Chain

- The Previous three filters form a chain through which every HTTP request passes.
- Together, the three filters perform the task of securing the application.
- The three filters are “chained” together by an object called the “filterChainProxy”, which in turn creates and starts the three filters



XML Configuration for FilterChainProxy

```
<bean id="filterChainProxy"
   class="org.springframework.security.util.FilterChainProxy">
<property name="filterInvocationDefinitionSource">
<value>
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
PATTERN_TYPE_APACHE_ANT
/
   **=httpSessionContextIntegrationFilter,formAuthenticationProcessingFilter,
exceptionTranslationFilter,filterSecurityInterceptor
</value>
</property>
</bean>
```

The above configuration states the filter beans which will be started by the proxy

The AuthenticationProcessingFilter

- The first filter through which the HTTP request will pass is the formAuthenticationProcessingFilter bean.
- This filter specializes in handling authentication request, i.e. The validation of username/password combinations. Let's take a look at the configuration XML

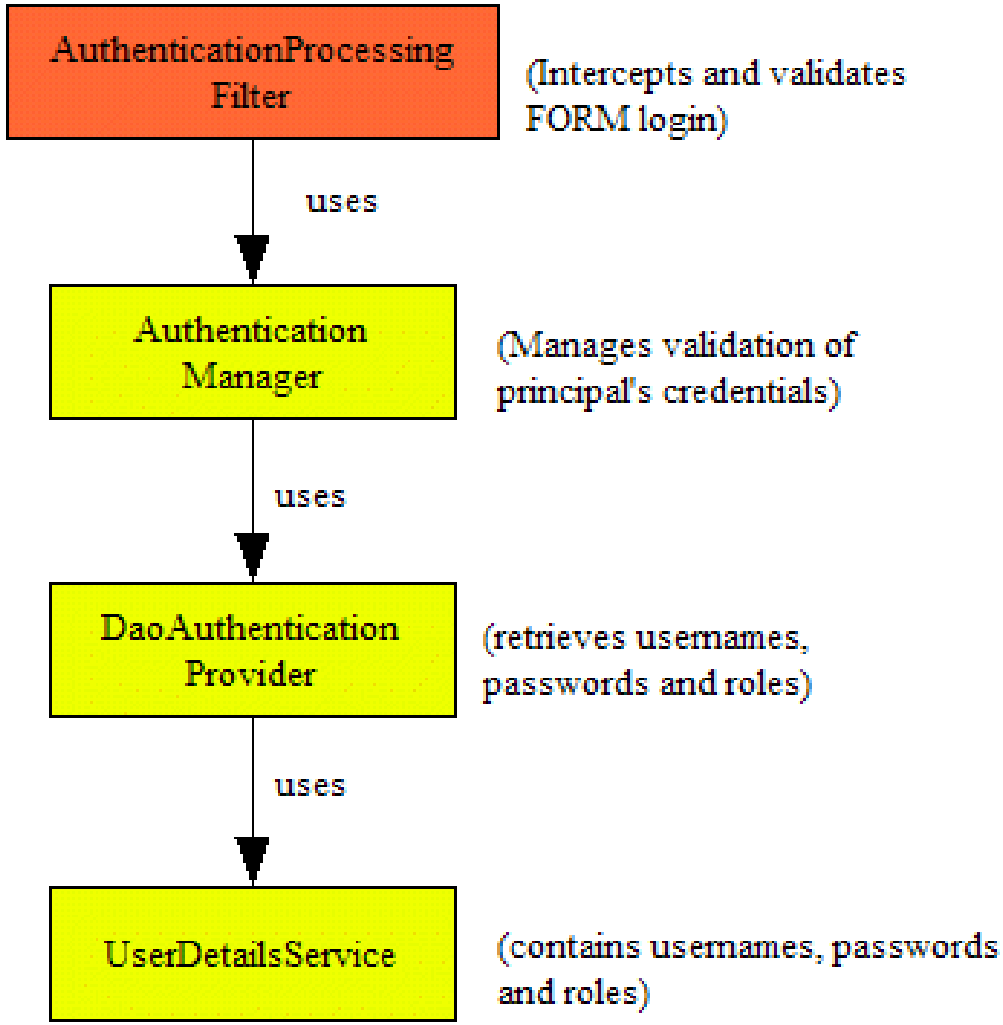
The AuthenticationProcessingFilter

– XML Configuration

```
<bean id="formAuthenticationProcessingFilter"
class="org.springframework.security.ui.webapp.AuthenticationProcess
   ingFilter">
<property name="filterProcessesUrl">
<value>/j_acegi_security_check</value>
</property>
<property name="authenticationFailureUrl">
<value>/loginFailed.html</value>
</property>
<property name="defaultTargetUrl">
<value>/</value>
</property>
<property name="authenticationManager">
<ref bean="authenticationManager" />
</property>
</bean>
```

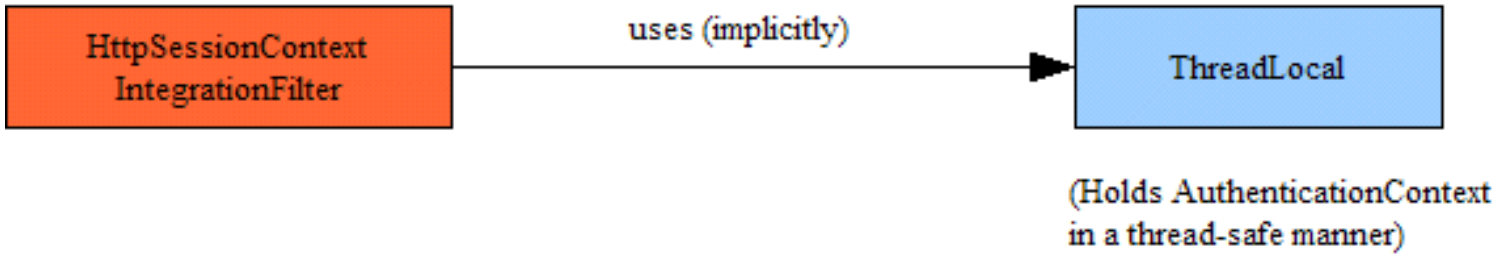
- The filter bean is of type org.springframework.security.ui.webapp.AuthenticationProcessingFilter. This filter class is specifically used for Form logins, which is why the form-submit URL (“filterProcessUrl”), the login-failed page (“authenticationFailureUrl”) are configured with this bean.
- In case you are wondering where the login page itself is configured: with the security realm, which we will get to later on. Remember that the AuthenticationProcessingFilter specialised in *handling* authentication requests. Spawning a login dialog is enables a user to log in, but has nothing to do with actually validating the provided username/password combination and is therefore not configured in this filter.

The AuthenticationProcessingFilter



The HttpSessionContextIntegrationFilter

- The work of the HttpSessionContextIntegrationFilter is very specialized and therefor very easy to configure.
- The only thing this filter does, is propagating the established authentication object through all requests.
- The filter wraps the authentication object a ThreadLocal and hands that wrapper over to the other filters in the chain.



The HttpSessionContextIntegrationFilter

- The XML Configuration

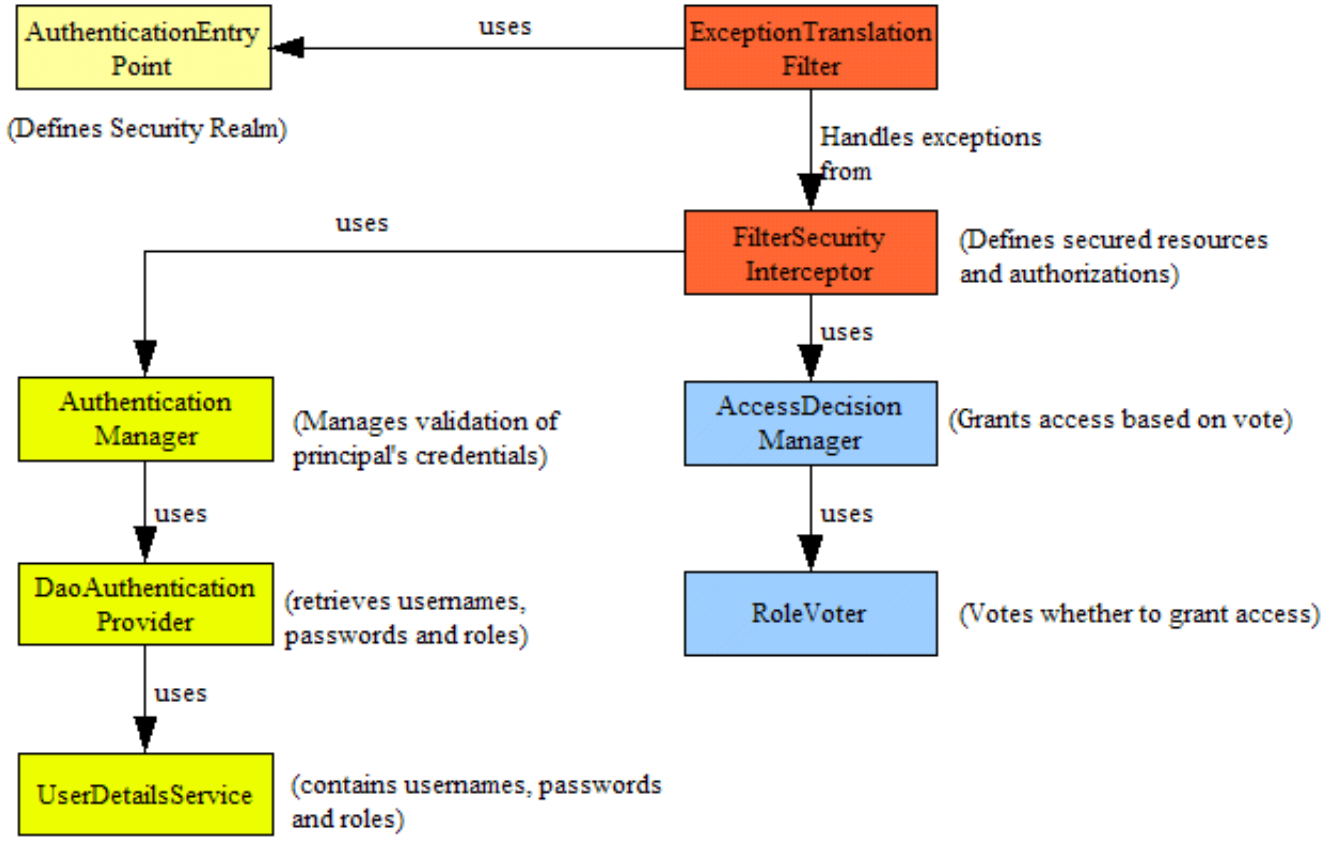
```
<bean id="httpSessionContextIntegrationFilter"
class="org.springframework.security.context.HttpSessionCon
   textIntegrationFilter">
</bean>
```

The ExceptionTranslationFilter

- The ExceptionTranslationFilter is the one of the two“pivotal” filters in the security system (the other being FilterSecurityInterceptor).
- ExceptionTranslationFilter catches any authentication or authorization error (in the form of an springframework.securityException) and may do one of the following two things.
  - If the exception was caused by the absence of an Authentication object (i.e. the user has not logged in yet), it spawns the configured AuthenticationEntryPoint to prompt the user for login (more on AuthenticationEntryPoint later).
  - If the exception was caused by an authorization exception thrown by FilterSecurityInterceptor (i.e. the user is logged in but is not authorized for the resource requested), ExceptionTranslationFilter will send an SC_FORBIDDEN (HTTP 403) error to the browser, which will display it’s built-in version of an ‘unauthorized access’ page

The ExceptionTranslationFilter

- The filter leaves all the hard work to it's collaborators: FilterSecurityInterceptor (to which it is linked through the filter chain) and authenticationEntryPoint.
- Before we examine those two in more detail, it will definitely be useful to take a look at the following diagram, which shows the two filters and their dependencies



XML configuration for ExceptionTranslationFilter

```
<bean id="exceptionTranslationFilter"
class="org.springframework.security.ui.ExceptionTranslationFil
   ter">
<property name="authenticationEntryPoint">
<ref bean="formLoginAuthenticationEntryPoint" />
</property>
</bean>
```

FilterSecurityInterceptor

- FilterSecurityInterceptor contains the definitions of the secured resources

```
<bean id="filterSecurityInterceptor"
class="org.springframework.security.intercept.web. FilterSecurityInterceptor">
<property name="authenticationManager">
<ref bean="authenticationManager" />
</property>
<property name="accessDecisionManager">
<ref bean="accessDecisionManager" />
</property>
```

FilterSecurityInterceptor

```
<property name="objectDefinitionSource">
<value>
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
PATTERN_TYPE_APACHE_ANT
/secure/admin/*=ROLE_ADMIN
/secure/app/*=ROLE_USER
</value>
</property>
</bean>
```

AuthenticationManager (1/3)

```
<bean id="authenticationManager"
   class="org.acegisecurity.providers.ProviderManager">
<property name="providers">
<list>
<ref bean="daoAuthenticationProvider" />
</list>
</property>
</bean>
<bean id="daoAuthenticationProvider"
class="org.acegisecurity.providers.dao.DaoAuthenticationPr
   ovider">
<property name="authenticationDao">
<ref bean="authenticationDao" />
</property>
</bean>
```

AuthenticationManager (2/3)

```
<bean id="authenticationDao"
   class="org.acegisecurity.userdetails.memory.InMemo
   ryDaoImpl">

<property name="userMap">

<value>

jklaassen=4moreyears,ROLE_ADMIN

bouerj=ineedsleep,ROLE_USER

</value>

</property>

</bean>
```

AuthenticationManager (3/3)

- The AuthenticationManager is of type ProviderManager, which means that it forms a proxy to the AuthenticationProvider.
- In Acegi, an AuthenticationProvider validates the inputted username/password combination and extracts the role(s) appointed to that user.
- AuthenticationProvider is itself a proxy to an AuthenticationDao, which is basically an registry containing usernames, passwords and roles.
- There a several types of AuthenticationDao (in-memory, database via JDBC or even LDAP), but for simplicity the standard InMemoryDaoImpl type is used.
- In the Dao, two users have been defined (jklaassen and bouerj), each with a different role.

AccessDecisionManager

- Validating the correctness of the username/password combination and the retrieval of associated roles is one thing, deciding whether to grant access is another. In other words: once a user has been *authenticated*, he must also be *authorized*. This decision is the responsibilty of the AccessDecisionManager. The AccessDecisionManager takes the available user information and decides to grant access (or not, of course). The AccessDecisionManager uses a Voter to determine if the user will be authorized.

AccessDecisionManager

```
<bean id="accessDecisionManager"
   class="org.acegisecurity.vote.UnanimousBased">
<property name="decisionVoters">
<list>
<ref bean="roleVoter" />
</list>
</property>
</bean>
<bean id="roleVoter"
   class="org.acegisecurity.vote.RoleVoter">
<property name="rolePrefix">
<value>ROLE_</value>
</property>
</bean>
```

AccessDecisionManager

- The above configuration is usually sufficient for webapplications, so we will leave it at that. Note though, that you will have to specify which rolenames should be handled by a specific voter by specifying the role prefix.
- Note also that it is possible to wire multiple voters into one AccessDecisionManager and that multiple ProviderManagers can be wired to an AuthenticationManager. So it is possible to let Acegi consult several different username/password registries available (say a mixture of LDAP, Database and NT Domain registries), with many different rolenames configured and voted on by several Voters. I admit that elaborate scenario to be a bit far fetched and certainly far too complex for one webapplication, but in huge enterprise systems such (legacy) complextity may very well exist.

AuthenticationEntryPoint

- Now only one configuration item needs to be specified: the AuthenticationEntryPoint, which is the starting point of the authentication dialog.
- If the FilterSecurityInterceptor determines that there is no available authentication object present, the SecurityEnforcementFilter will pass control to the AuthenticationEntryPoint

AuthenticationEntryPoint

```
<bean id="formLoginAuthenticationEntryPoint"
class="org.acegisecurity.ui.webapp.AuthenticationPro
   cessingFilterEntryPoint">
<property name="loginFormUrl">
<value>/login.jsp</value>
</property>
<property name="forceHttps">
<value>false</value>
</property>
</bean>
```

- The AuthenticationEntryPoint in this example is of type AuthenticationProcessingFilterEntryPoint, which is specifically suitable for FORM login dialogs. Configuration is fairly straightforward, only the URL of the login form to spawn needs to be specified. An optional parameter “forceHttps”, may be set to “true” if you would like the username/password data to be sent as encrypted data.

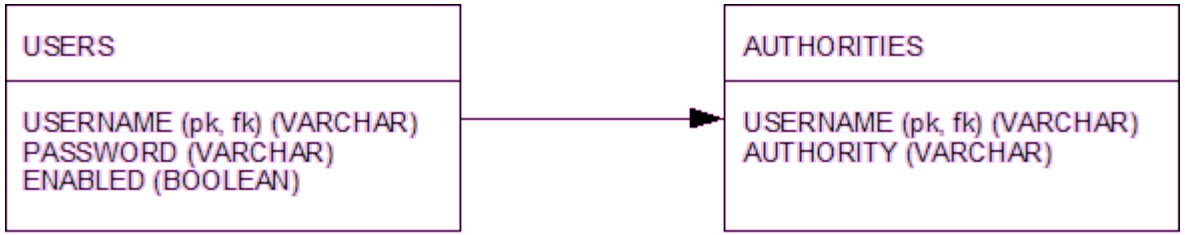
Using an authentication database through JDBC

- The above example uses the InMemoryDaoImpl as the AuthenticationDAO, to store usernames, passwords and roles. That's fine for simple testing purposes, but in the real world a user registry is usually a database. So here's some configuration for using the JdbcDaoImpl class of Acegi

```
<bean id="userDetailsService"
   class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
<property name="dataSource">
<ref bean="dataSource" />
</property>
</bean>
```

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName">
<value>com.mysql.jdbc.Driver</value>
</property>
<property name="url">
<value>jdbc:mysql://localhost:3306/spring
</value>
</property>
<property name="username">
<value>j2ee</value>
</property>
<property name="password">
<value>password</value>
</property>
</bean>
```

- Now all you need is the database specification: the tables. There are two ways to go here. The hard way involves using custom tables and thus specifying custom queries and rowmappers for the AuthenticationDao. This is beyond the scope of this tutorial. The documentation included in the full Acegi framework download contains more information on using a user database.
- The easy way is to create a database tailored to Acegi's wishes, which would mean the following schema:



- The “ENABLED” column could also be a CHAR or VARCHAR holding the values “true” and “false”. Obviously, the AUTHORITY column would hold the role names. Populate the database and you are all set!