## What is Spring Security?

spring Security provides comprehensive security services for J2EE-based enterprise software applications. There is a particular emphasis on supporting projects built using The Spring Framework, which is the leading J2EE solution for enterprise software development. If you're not using Spring for developing enterprise applications, we warmly encourage you to take a closer look at it. Some familiarity with Spring - and in particular dependency injection principles - will help you get up to speed with Spring Security more easily.

At an authentication level, Spring Security supports a wide range of authentication models. Most of these authentication models are either provided by third parties, or are developed by relevant standards bodies such as the Internet Engineering Task Force. In addition, Spring Security provides its own set of authentication features. Specifically, Spring Security currently supports authentication integration with all of these technologies:

* HTTP BASIC authentication headers (an IEFT RFC-based standard)
* HTTP Digest authentication headers (an IEFT RFC-based standard)
* HTTP X.509 client certificate exchange (an IEFT RFC-based standard)
* LDAP (a very common approach to cross-platform authentication needs, especially in large environments)
* Form-based authentication (for simple user interface needs)
* OpenID authentication
* Authentication based on pre-established request headers (such as Computer Associates Siteminder)
* JA-SIG Central Authentication Service (otherwise known as CAS, which is a popular open source single sign on system)
* Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (a Spring remoting protocol)
* Automatic "remember-me" authentication (so you can tick a box to avoid re-authentication for a predetermined period of time)
* Anonymous authentication (allowing every call to automatically assume a particular security identity)
* Run-as authentication (which is useful if one call should proceed with a different security identity)
* Java Authentication and Authorization Service (JAAS)
* JEE container autentication (so you can still use Container Managed Authentication if desired)
* Kerberos
* Java Open Source Single Sign On (JOSSO) \*
* OpenNMS Network Management Platform \*
* AppFuse \*
* AndroMDA \*
* Mule ESB \*
* Direct Web Request (DWR) \*
* Grails \*
* Tapestry \*
* JTrac \*
* Jasypt \*
* Roller \*
* Elastic Path \*
* Atlassian Crowd \*
* Your own authentication systems (see below)

(\* Denotes provided by a third party; check our [integration page](http://acegisecurity.org/powering.html) for links to the latest details)

#### Core - spring-security-core.jar

Contains core authentication and access-contol classes and interfaces, remoting support and basic provisioning APIs. Required by any application which uses Spring Security. Supports standalone applications, remote clients, method (service layer) security and JDBC user provisioning. Contains the top-level packages:

* org.springframework.security.core
* org.springframework.security.access
* org.springframework.security.authentication
* org.springframework.security.provisioning

#### Web - spring-security-web.jar

Contains filters and related web-security infrastructure code. Anything with a servlet API dependency. You’ll need it if you require Spring Security web authentication services and URL-based access-control. The main package is org.springframework.security.web.

#### Config - spring-security-config.jar

Contains the security namespace parsing code. You need it if you are using the Spring Security XML namespace for configuration. The main package is org.springframework.security.config. None of the classes are intended for direct use in an application.

#### LDAP - spring-security-ldap.jar

LDAP authentication and provisioning code. Required if you need to use LDAP authentication or manage LDAP user entries. The top-level package is org.springframework.security.ldap.

**Design of the Namespace**

The namespace is designed to capture the most common uses of the framework and provide a simplified and concise syntax for enabling them within an application. The design is based around the large-scale dependencies within the framework, and can be divided up into the following areas:

* *Web/HTTP Security* - the most complex part. Sets up the filters and related service beans used to apply the framework authentication mechanisms, to secure URLs, render login and error pages and much more.
* *Business Object (Method) Security* - options for securing the service layer.
* *AuthenticationManager* - handles authentication requests from other parts of the framework.
* *AccessDecisionManager* - provides access decisions for web and method security. A default one will be registered, but you can also choose to use a custom one, declared using normal Spring bean syntax.
* *AuthenticationProvider*s - mechanisms against which the authentication manager authenticates users. The namespace provides supports for several standard options and also a means of adding custom beans declared using a traditional syntax.
* *UserDetailsService* - closely related to authentication providers, but often also required by other beans.

### web.xml Configuration

The first thing you need to do is add the following filter declaration to your web.xml file:

<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>

This provides a hook into the Spring Security web infrastructure. DelegatingFilterProxy is a Spring Framework class which delegates to a filter implementation which is defined as a Spring bean in your application context. In this case, the bean is named "springSecurityFilterChain", which is an internal infrastructure bean created by the namespace to handle web security. Note that you should not use this bean name yourself. Once you’ve added this to your web.xml, you’re ready to start editing your application context file. Web security services are configured using the <http> element.

The **DelegatingFilterProxy** is a Filter as it was explained above, whose goal is "**delegating to a Spring-managed bean that implements the Filter interface**", that is, it finds a bean ("target bean" or "delegate") in your Spring application context and invokes it. How is it possible? Because this bean implements javax.servlet.Filter, its doFilter method is called.

Which bean is called? the DelegatingFilterProxy "Supports a "targetBeanName" [...], specifying the name of the target bean in the Spring application context."

As you saw in your web.xml that **the bean's name is "springSecurityFilterChain"**.

So, in the context of a web application, a Filter instantiates a bean called "springSecurityFilterChain" in your application context and then delegate to it via the doFilter() method.

Remember, your application context is defined with ALL THE APPLICATION-CONTEXT (XML) files. For instance: applicationContext.xml AND applicationContext-security.xml.

So try to **find a bean called "springSecurityFilterChain"** in the latter...

...and **probably you can't** (for instance if you followed a tutorial or if you configured the security using Roo)

Here is the magic: **there's a new element for configuring the security**, something like

<http auto-config="true" use-expressions="true">

as it is allowed by <http://www.springframework.org/schema/security/spring-security-3.0.xsd>, will do the trick.

When Spring loads the application context using XML files, if it finds a element, it will try to set up the HTTP security, that is, a filter stack and protected URLs and to register the FilterChainProxy named "springSecurityFilterChain".

Alternatively, you can define the bean in the classic way, that is:

<beans:bean id="springSecurityFilterChain" class="org.springframework.security.web.FilterChainProxy">

But it's less recommended, since you need to do a lot of configuration (all the filters that you're going to use. And there are more than a dozen of them)

### 4.2.2 A Minimal <http> Configuration

All you need to enable web security to begin with is

<http>

<intercept-url pattern="/\*\*" access="hasRole('USER')" />

<form-login />

<logout />

</http>

Which says that we want all URLs within our application to be secured, requiring the role ROLE\_USER to access them, we want to log in to the application using a form with username and password, and that we want a logout URL registered which will allow us to log out of the application. <http> element is the parent for all web-related namespace functionality. The <intercept-url> element defines a pattern which is matched against the URLs of incoming requests using an ant path style syntax [[2]](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/ns-config.html" \l "ftn.d5e824). You can also use regular-expression matching as an alternative (see the namespace appendix for more details). The access attribute defines the access requirements for requests matching the given pattern. With the default configuration, this is typically a comma-separated list of roles, one of which a user must have to be allowed to make the request. The prefix"ROLE\_" is a marker which indicates that a simple comparison with the user’s authorities should be made. In other words, a normal role-based check should be used. Access-control in Spring Security is not limited to the use of simple roles (hence the use of the prefix to differentiate between different types of security attributes). We’ll see later how the interpretation can vary footnote:[The interpretation of the comma-separated values in the access attribute depends on the implementation of the –1— which is used. In Spring Security 3.0, the attribute can also be populated with an –2—.

|  |
| --- |
| [Note] |
| You can use multiple <intercept-url> elements to define different access requirements for different sets of URLs, but they will be evaluated in the order listed and the first match will be used. So you must put the most specific matches at the top. You can also add a method attribute to limit the match to a particular HTTP method (GET, POST, PUT etc.). |

To add some users, you can define a set of test data directly in the namespace:

<authentication-manager>

<authentication-provider>

<user-service>

<user name="jimi" password="jimispassword" authorities="ROLE\_USER, ROLE\_ADMIN" />

<user name="bob" password="bobspassword" authorities="ROLE\_USER" />

</user-service>

</authentication-provider>

</authentication-manager>

If you are familiar with pre-namespace versions of the framework, you can probably already guess roughly what’s going on here. The <http> element is responsible for creating a FilterChainProxy and the filter beans which it uses. Common problems like incorrect filter ordering are no longer an issue as the filter positions are predefined.

The <authentication-provider> element creates a DaoAuthenticationProvider bean and the <user-service> element creates an InMemoryDaoImpl. All authentication-provider elements must be children of the <authentication-manager> element, which creates a ProviderManager and registers the authentication providers with it. You can find more detailed information on the beans that are created in the [namespace appendix](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/appendix-namespace.html). It’s worth cross-checking this if you want to start understanding what the important classes in the framework are and how they are used, particularly if you want to customise things later.

The configuration above defines two users, their passwords and their roles within the application (which will be used for access control). It is also possible to load user information from a standard properties file using the properties attribute on user-service. See the section on [in-memory authentication](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/core-services.html#core-services-in-memory-service) for more details on the file format. Using the <authentication-provider> element means that the user information will be used by the authentication manager to process authentication requests. You can have multiple <authentication-provider> elements to define different authentication sources and each will be consulted in turn.

At this point you should be able to start up your application and you will be required to log in to proceed. Try it out, or try experimenting with the"tutorial" sample application that comes with the project.

### 4.2.3 Form and Basic Login Options

You might be wondering where the login form came from when you were prompted to log in, since we made no mention of any HTML files or JSPs. In fact, since we didn’t explicitly set a URL for the login page, Spring Security generates one automatically, based on the features that are enabled and using standard values for the URL which processes the submitted login, the default target URL the user will be sent to after logging in and so on. However, the namespace offers plenty of support to allow you to customize these options. For example, if you want to supply your own login page, you could use:

<http>

<intercept-url pattern="/login.jsp\*" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

<intercept-url pattern="/\*\*" access="ROLE\_USER" />

<form-login login-page='/login.jsp'/>

</http>

Also note that we’ve added an extra intercept-url element to say that any requests for the login page should be available to anonymous users [[3]](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/ns-config.html" \l "ftn.d5e864) and also the [AuthenticatedVoter](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/authz-arch.html#authz-authenticated-voter) class for more details on how the value IS\_AUTHENTICATED\_ANONYMOUSLY is processed.]. Otherwise the request would be matched by the pattern /\*\* and it wouldn’t be possible to access the login page itself! This is a common configuration error and will result in an infinite loop in the application. Spring Security will emit a warning in the log if your login page appears to be secured. It is also possible to have all requests matching a particular pattern bypass the security filter chain completely, by defining a separate http element for the pattern like this:

<http pattern="/css/\*\*" security="none"/>

<http pattern="/login.jsp\*" security="none"/>

<http use-expressions="false">

<intercept-url pattern="/\*\*" access="ROLE\_USER" />

<form-login login-page='/login.jsp'/>

</http>

From Spring Security 3.1 it is now possible to use multiple http elements to define separate security filter chain configurations for different request patterns. If the pattern attribute is omitted from an http element, it matches all requests. Creating an unsecured pattern is a simple example of this syntax, where the pattern is mapped to an empty filter chain [[4]](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/ns-config.html" \l "ftn.d5e875). We’ll look at this new syntax in more detail in the chapter on the [Security Filter Chain](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/security-filter-chain.html#filter-chains-with-ns).

It’s important to realise that these unsecured requests will be completely oblivious to any Spring Security web-related configuration or additional attributes such as requires-channel, so you will not be able to access information on the current user or call secured methods during the request. Use access='IS\_AUTHENTICATED\_ANONYMOUSLY' as an alternative if you still want the security filter chain to be applied.

If you want to use basic authentication instead of form login, then change the configuration to

<http use-expressions="false">

<intercept-url pattern="/\*\*" access="ROLE\_USER" />

<http-basic />

</http>

Basic authentication will then take precedence and will be used to prompt for a login when a user attempts to access a protected resource. Form login is still available in this configuration if you wish to use it, for example through a login form embedded in another web page.

#### Setting a Default Post-Login Destination

If a form login isn’t prompted by an attempt to access a protected resource, the default-target-url option comes into play. This is the URL the user will be taken to after successfully logging in, and defaults to "/". You can also configure things so that the user always ends up at this page (regardless of whether the login was "on-demand" or they explicitly chose to log in) by setting the always-use-default-target attribute to "true". This is useful if your application always requires that the user starts at a "home" page, for example:

<http pattern="/login.htm\*" security="none"/>

<http use-expressions="false">

<intercept-url pattern='/\*\*' access='ROLE\_USER' />

<form-login login-page='/login.htm' default-target-url='/home.htm'

always-use-default-target='true' />

</http>

For even more control over the destination, you can use the authentication-success-handler-ref attribute as an alternative to default-target-url. The referenced bean should be an instance of AuthenticationSuccessHandler. You’ll find more on this in the [Core Filters](http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/core-web-filters.html#form-login-flow-handling) chapter and also in the namespace appendix, as well as information on how to customize the flow when authentication fails.

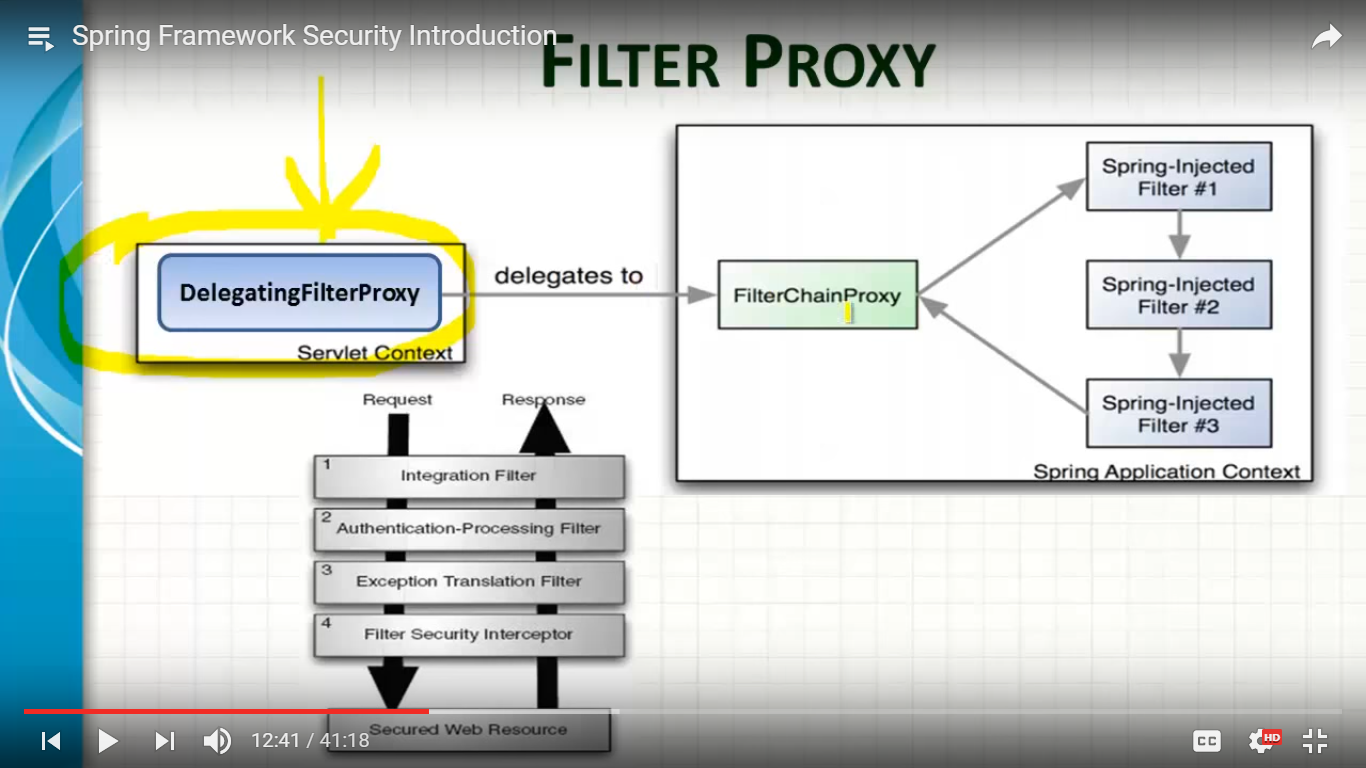
### 4.2.4 Logout Handling

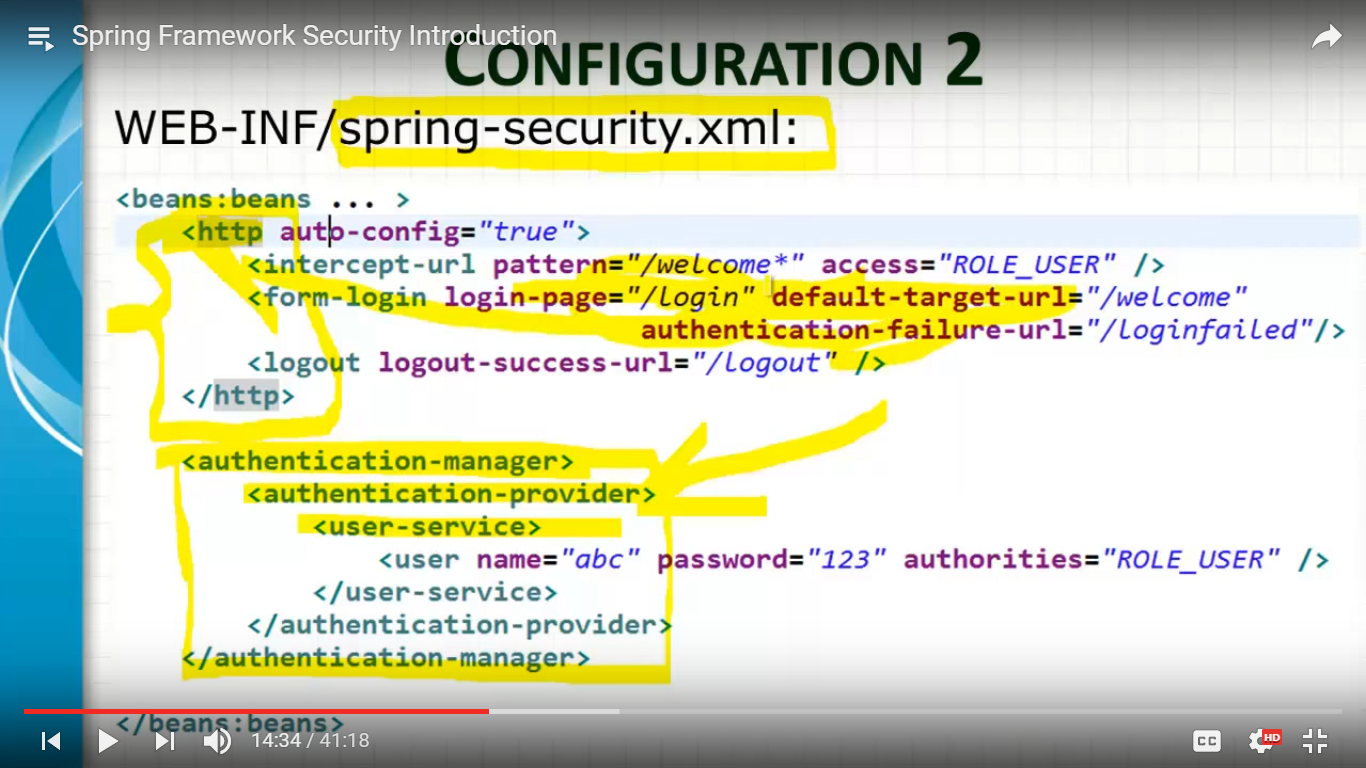
The logout element adds support for logging out by navigating to a particular URL. The default logout URL is /logout, but you can set it to something else using the logout-url attribute. More information on other available attributes may be found in the namespace appendix.

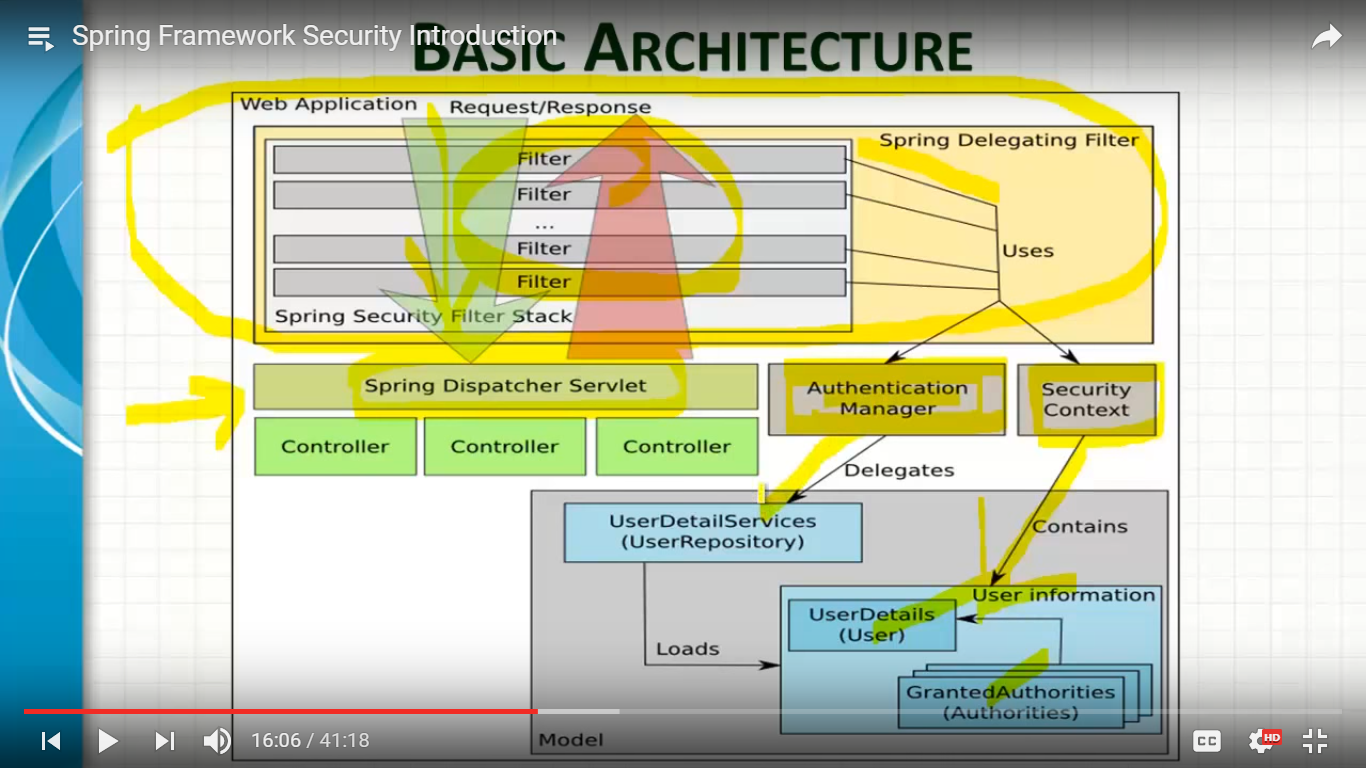
Spring security :

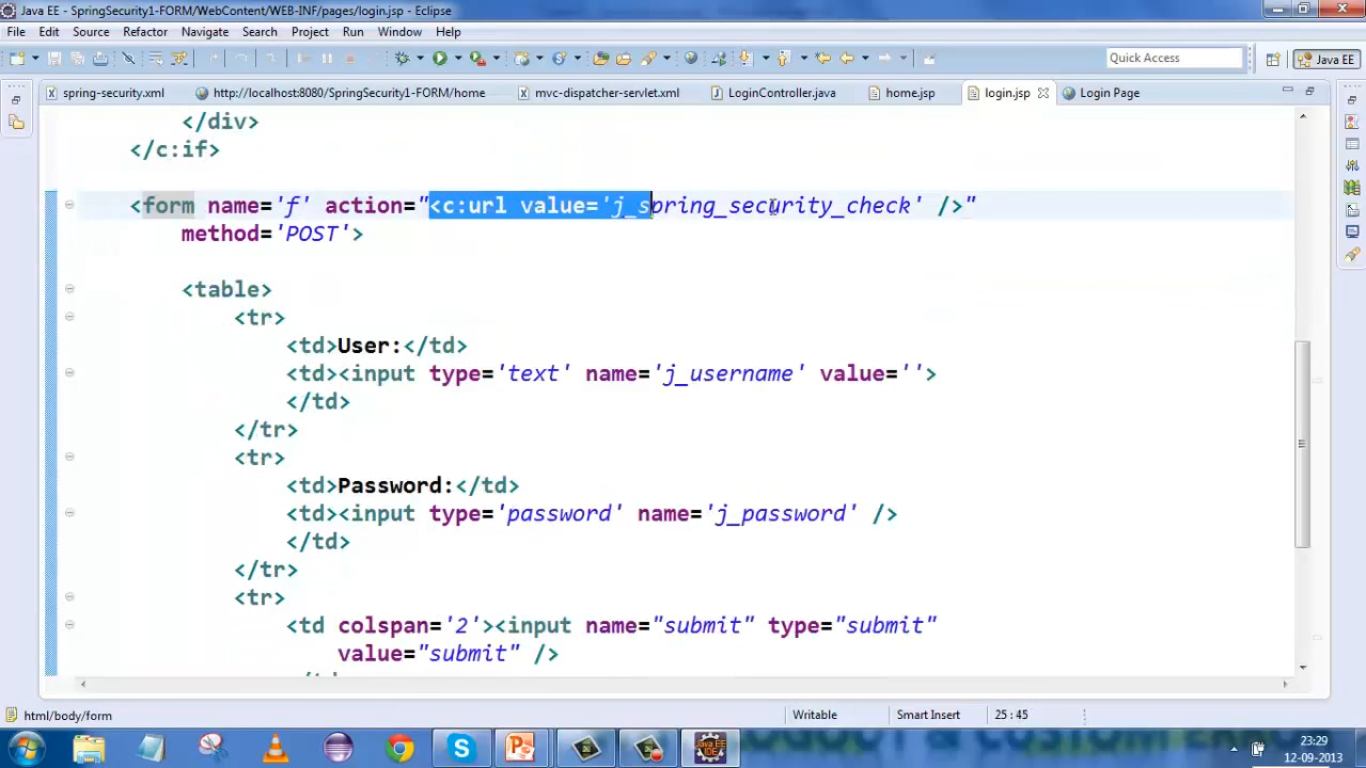
DeligatingFilterProxy

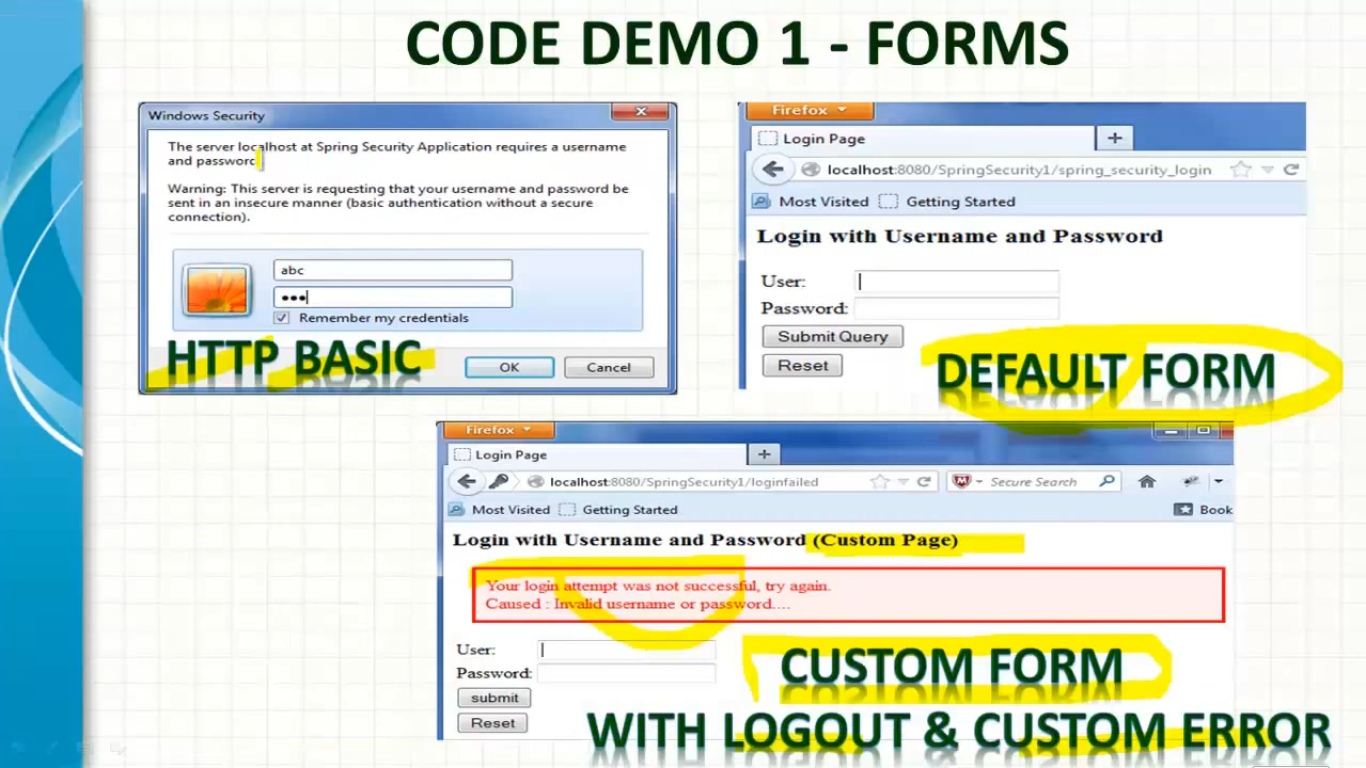
SpringsecurityFilterChain as url pattern

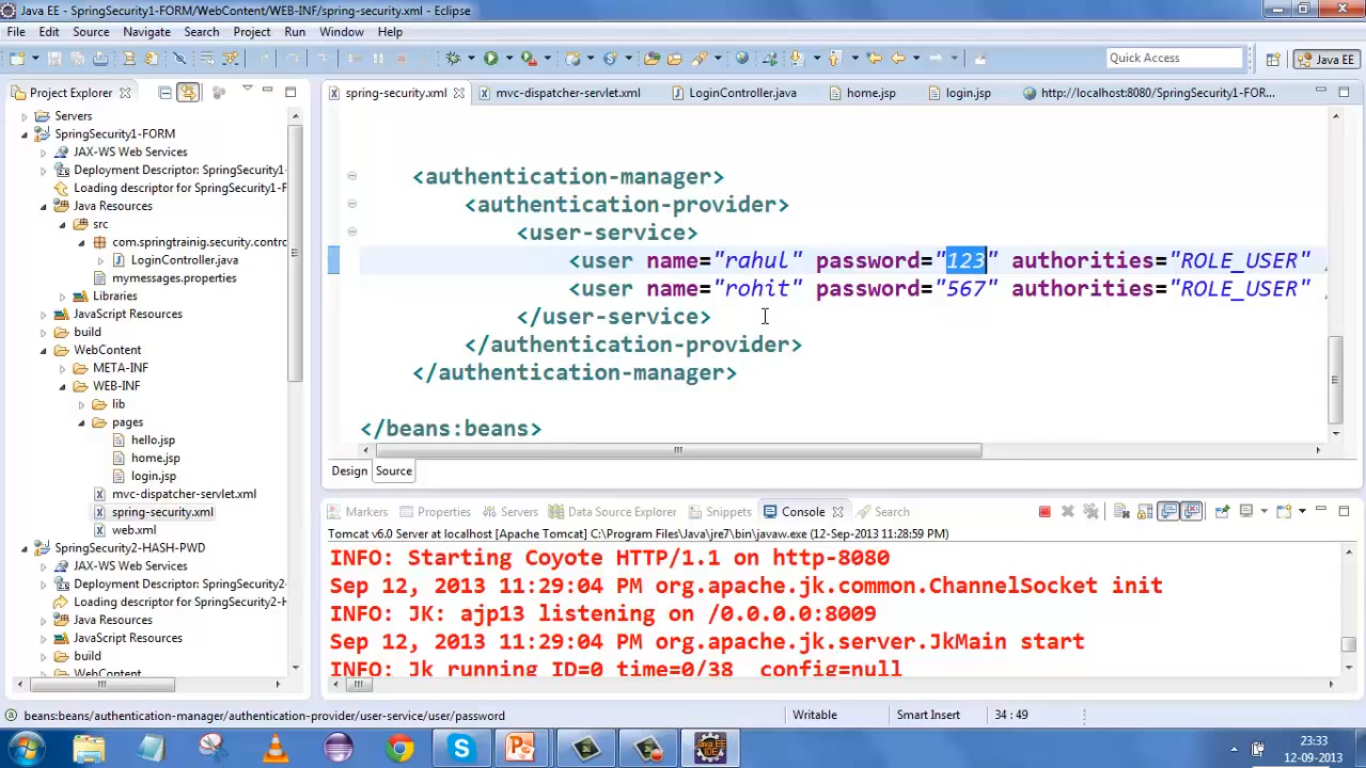


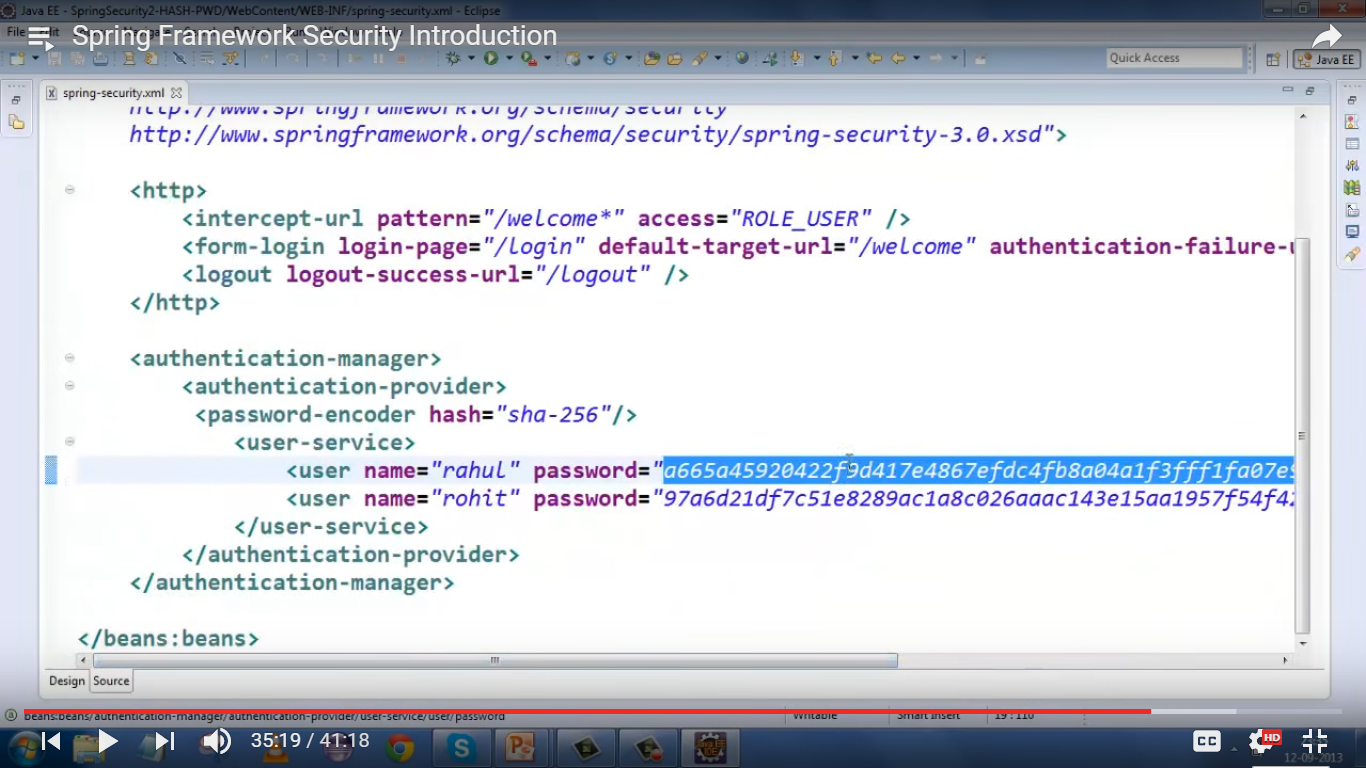


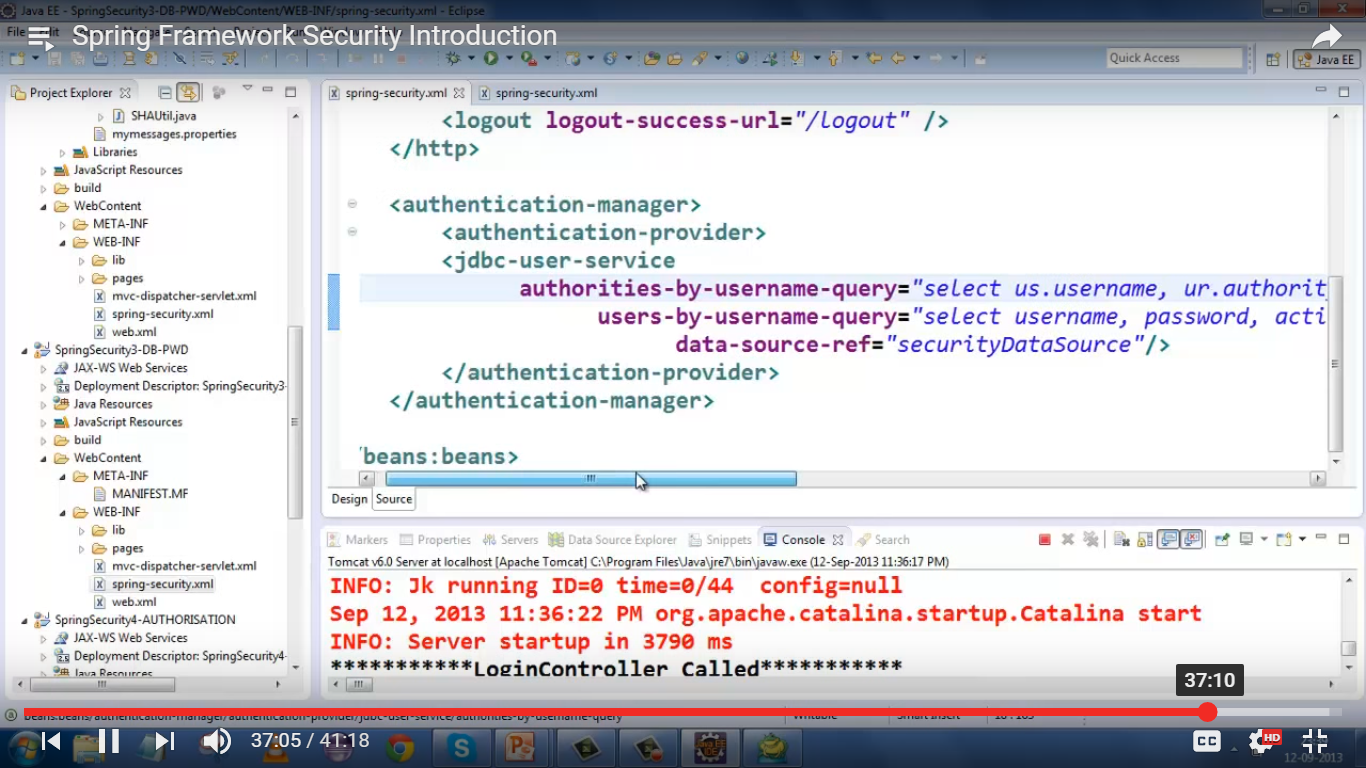


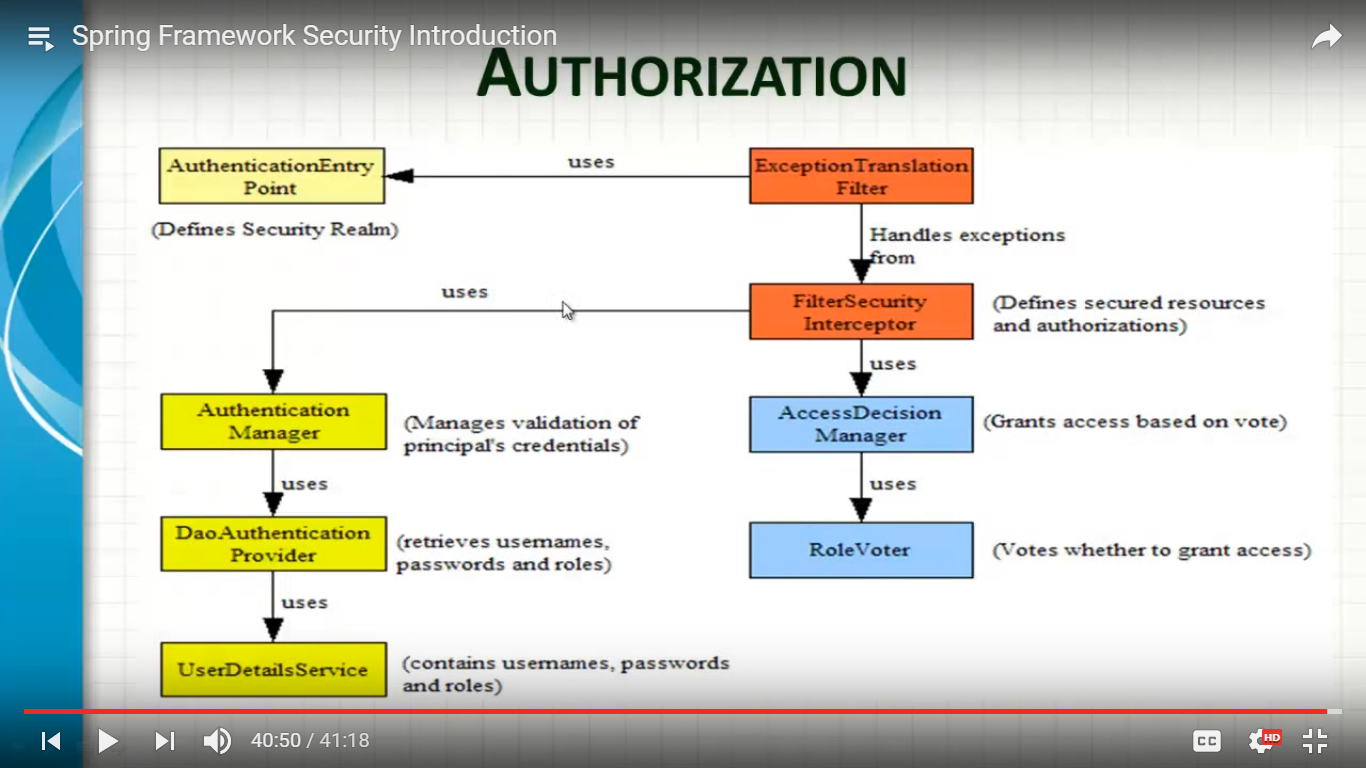












**Authorizing Web URLs**

Activate expression-based web URL authorization as follows:

<http auto-config="true" use-expressions="true">

Then implement access rules for URLs by adding <intercept-url> children directly under the <http> element. The <intercept-url> attributes are as follows:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| pattern | URL pattern to match. Uses Ant syntax by default (e.g. \* and \*\* wildcards) but regex is supported as well. | Yes |
| method | Optional HTTP method to narrow the match | No |
| access | When expressions are activated, this contains the security expression to apply to the URL and (if applicable) HTTP method. Legacy behavior is to store a comma-delimited list of user roles. | No |
| filters | Only possible value is “none”, which indicates that the request is to bypass the Spring Security filter chain. The request will have no SecurityContext. This is mostly for static resources like images, JavaScript, CSS and so forth. | No |
| requires-channel | Can be either “http” or “https”. | No |

Rules are processed in order, so the first pattern/method match determines which security expression will be used to make the access decision. Therefore, place more specific patterns before more general patterns.

Implement a whitelist by placing a <intercept-url pattern="/\*\*" access="denyAll" /> at the end of the list of rules.

---===--000—

### In-Memory Authentication

Is easy to use create a custom UserDetailsService implementation that extracts information from a persistence engine of choice, but many applications do not require such complexity. This is particularly true if you're building a prototype application or just starting integrating Spring Security, when you don't really want to spend time configuring databases or writing UserDetailsService implementations. For this sort of situation, a simple option is to use the user-service element from the security [namespace](http://docs.spring.io/spring-security/site/docs/3.0.x/reference/ns-config.html#ns-minimal):

<user-service id="userDetailsService">

<user name="jimi" password="jimispassword" authorities="ROLE\_USER, ROLE\_ADMIN" />

<user name="bob" password="bobspassword" authorities="ROLE\_USER" />

</user-service>

This also supports the use of an external properties file:

<user-service id="userDetailsService" properties="users.properties"/>

The properties file should contain entries in the form

username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

For example

jimi=jimispassword,ROLE\_USER,ROLE\_ADMIN,enabled

bob=bobspassword,ROLE\_USER,enabled

**Spring Security:**

[Spring Security Custom Authentication - AuthenticationProvider vs UserDetailsService](http://stackoverflow.com/questions/31630818/spring-security-custom-authentication-authenticationprovider-vs-userdetailsser)

when you are using a different authentication system, and the password is not provided in your own database/data model, you have to use the AuthenticationProvider. for example, I've worked in a project that the customer had a centralized authentication system (CAS), so my system had no idea about the password, I had to implement the AuthenticationProvider and send the given password to the CAS, and act according to its answer.

But in another system, I was storing the password in my database, so all I had to do was implementing the UserDetailsService and check if the user exists in my database or not, spring-security had to do the rest.