Securing Java applications with Acegi, Part 1: Architectural overview and security filters

URL-based security with Acegi Security System

Skill Level: Intermediate

Bilal Siddiqui
Freelance consultant
WaxSys

27 Mar 2007

This series introduces Acegi Security System, a formidable open source security framework for Java™ enterprise applications. In this first article, consultant Bilal Siddiqui introduces you to the architecture and components of Acegi and shows you how to use it to secure a simple Java enterprise application.

Acegi Security System is a formidable, easy-to-use alternative to writing endless security code for your Java enterprise applications. While intended especially for applications written using the Spring framework, there is no reason why Acegi cannot be used for any type of Java application. This series introduces you to Acegi from the ground up and shows you how to use it to secure both simple enterprise applications and ones that are more complex.

This series starts with an introduction to the common security concerns of enterprise applications and explains how Acegi resolves them. You will see Acegi's architectural model and its security filters, which embody most of the functionality you'll use to secure your applications. You will learn how filters work individually, how they can be combined, and how a filter chain functions from start to finish in an enterprise security implementation. This article concludes with a sample application that demonstrates Acegi's implementation of a URL-based security system. The following articles in the series will explore some of the more advanced uses of Acegi, including how to design and host access control policies and then configure Acegi to use them.

You must download Acegi to compile the example code and run the sample applications for this series. You will also need to have a Tomcat server running as part of your workstation.

Enterprise application security

Because enterprise content management (ECM) applications manage the authoring and processing of enterprise content stored in different types of data sources (such as file systems, relational databases, and directory services), ECM security requires controlling access to those data sources. For example, an ECM application might control who is authorized to read, edit, or delete data related to the design, marketing, production, and quality control of a manufacturing enterprise.

In an ECM security scenario, it is common to implement access control by applying security on the locators (or network addresses) of enterprise resources. This simple security model is called *Universal Resource Locator*, or URL security. As I demonstrate later in this article (and in the series), Acegi offers comprehensive features for implementing URL security.

In many enterprise scenarios, however, URL security is not enough. For example, consider a PDF document containing the data about a particular product manufactured by the manufacturing company. Part of the document contains design data meant to be edited and updated by the company's design department. Another part of the document contains production data, which production managers will use. For a scenario like this one, you would need to implement more fine-grained security applying different access rights to different portions of the document.

This article introduces you to Acegi's facilities for implementing URL security. The next article in the series will demonstrate the framework's method-based security, which gives you more fine-grained control over enterprise data access.

Acegi Security System

Acegi Security System uses security filters to provide authentication and authorization services to enterprise applications. The framework offers several types of filters that you can configure according to your application requirements. You will learn about the various types of security filters later in the article; for the moment, just note that you can configure Acegi's security filters for the following tasks:

1. Prompt the user for login before accessing a secure resource.
2. Authenticate the user by checking a security token such as a password.

3. Check whether an authenticated user has the privilege to access a secure resource.
4. Redirect a successfully authenticated and authorized user to the secure resource requested.
5. Display an Access Denied page to a user who does not have the privilege to access a secure resource.
6. Remember a successfully authenticated user on the server and set a secure cookie on the user's client. The next authentication can then be performed using the cookie and without asking the user to log in.
7. Store authentication information in server-side session objects to securely serve subsequent requests for resources.
8. Build and maintain a cache of security information in server-side objects to optimize performance.
9. When the user signs out, destroy server-side objects maintained for the user's secure session.
10. Communicate with a variety of back-end data storage services (like a directory service or a relational database) that are used to store users' security information and the ECM's access-control policies.

As this list suggests, Acegi's security filters allow you to do almost anything you might require to secure your enterprise applications.

Architecture and components

The more you understand Acegi, the easier it will be for you to work with it. This section introduces Acegi's components; next you will learn how the framework uses inversion of control (IOC) and XML configuration files to combine components and express their dependencies.

The big four

Acegi Security System consists of four main types of component: filters, managers, providers, and handlers.

Filters
These most high-level components provide common security services like authentication processing, session handling, and logout. I discuss filters in depth later in the article.

Managers

Filters are only a high-level abstraction of security-related functionality: managers and providers are used to actually implement authentication processing and logout services. Managers manage lower level security services offered by different providers.

Providers

A variety of providers are available to communicate with different types of data storage services, such as directory services, relational databases, or simple in-memory objects. This means you can store your user base and access control policies in any of these data storage services and Acegi's managers will select appropriate providers at run time.

Handlers

Tasks are sometimes broken up into multiple steps, with each step performed by a specific handler. For example, Acegi's *logout filter* uses two handlers to sign out an HTTP client. One handler invalidates a user's HTTP session and another handler destroys the user's cookie. Having multiple handlers provides flexibility while configuring Acegi to work according to your application requirements. You can select the handlers of your choice to execute the steps required to secure your application.

Inversion of control

Acegi's components are dependent on each other to secure your enterprise applications. For example, an authentication processing filter requires an authentication manager to select an appropriate authentication provider. This means you must be able to express and manage the dependency of Acegi's components.

An IOC implementation is commonly used to manage the dependencies of Java components. IOC offers two important features:

1. It provides a syntax to express what components are required in an application and how they depend on each other.

2. It ensures that required components are available at run time.

The XML configuration file

Acegi uses the popular open source IOC implementation that comes with the Spring framework (see Resources) to manage its components. Spring uses an XML configuration file to express the dependency of components, as shown in Listing 1:

Listing 1. Structure of a Spring configuration file

```
<beans>
    <bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
        <property name="filterInvocationDefinitionSource">
            <value> value here </value>
        </property>
    </bean>

    <bean id="authenticationProcessingFilter"
        class="org.acegisecurity.ui.webapp.AuthenticationProcessingFitler">
        <property name="authenticationManager" ref="authManager"/>
        <!-- Other properties -->
    </bean>

    <bean id="authManager"
        class="org.acegisecurity.providers.ProviderManager">
        <property name="providers">
            <!--  List of providers here -->
        </property>
    </bean>
    <!-- Other bean tags -->
</beans>
```

As you can see, the Spring XML configuration file used by Acegi contains a single `<beans>` tag that wraps a number of other `<bean>` tags. All Acegi components (that is, filters, managers, providers, etc.) are actually JavaBeans. Each `<bean>` tag in the XML configuration file represents an Acegi component.

Further notes about the XML configuration file

The first thing you'll note is that each `<bean>` tag has a *class* attribute, which identifies the class that the component uses. The `<bean>` tag also has an *id* attribute that identifies the instance (Java object) acting as an Acegi component.

For instance, the first `<bean>` tag of Listing 1 identifies a component instance named `filterChainProxy`, which is an instance of a class named `org.acegisecurity.util.FilterChainProxy`.

The dependency of a bean is expressed using child tags of the `<bean>` tag. For example, notice the `<property>` child tag of the first `<bean>` tag. The `<property>` child tag defines values or other beans on which the `<bean>` tag depends.

So in Listing 1, the `<property>` child tag of the first `<bean>` tag has a name attribute and a `<value>` child tag, which respectively define the name and value of the property on which the bean depends.

Likewise, the second and third `<bean>` tags of Listing 1 define that a filter bean depends on a manager bean. The second `<bean>` tag represents the filter bean and the third `<bean>` tag represents the manager bean.

The `<bean>` tag for the filter contains a `<property>` child tag with two attributes, `name` and `ref`. The `name` attribute defines a property of the filter bean and the `ref` attribute refers to the instance (name) of the manager bean.

The next section shows you how to configure Acegi filters in an XML configuration file. Later in the article, you will use the filters in a sample Acegi application.

Security filters

As I previously mentioned, Acegi uses security filters to provide authentication and authorization services to enterprise applications. You can use and configure various types of filters according to your application requirements. This section introduces the five most important Acegi security filters.

Session Integration Filter

Acegi's *Session Integration Filter* (SIF) is normally the first filter you will configure. SIF creates a *security context object*, which is a placeholder for security-related information. Other Acegi filters save security information in the security context and also use the information available in the security context.

SIF creates the security context and calls other filters in the filter chain. Other filters then retrieve the security context and make changes to it. For example, the Authentication Processing Filter (which I discuss next) stores user information such as username, password, and e-mail address in the security context.

When all the filters have finished processing, SIF checks the security context for updates. If any filter has made changes to the security context, SIF saves the changes into a server-side session object. If no changes are found in the security context, SIF discards it.

SIF is configured in the XML configuration file as shown in Listing 2:

Listing 2. Configuring SIF

```
<bean id="httpSessionContextIntegrationFilter"
     class="org.acegisecurity.context.HttpSessionContextIntegrationFilter"/>
```

Authentication Processing Filter

Acegi uses the *Authentication Processing Filter* (APF) for authentication. APF uses an authentication (or login) form, in which a user enters a username and password and triggers authentication.

APF performs all back-end authentication processing tasks such as extracting the username and password from the client request, reading the user's parameters from the back-end user base, and using the information to authenticate the user.

When you configure APF, you must provide the following parameters:

- **Authentication manager** specifies the authentication manager to be used to manage authentication providers.
- **Filter processes URL** specifies the URL to be accessed when the client presses the **Sign In** button on the login form. Upon receiving a request for this URL, Acegi invokes APF.
- **Default target URL** specifies the page to be presented to the user if authentication and authorization is successful.
- **Authentication failure URL** specifies the page the user sees if authentication fails.

APF fetches the username, password, and other information from the user's request object. It passes this information to the authentication manager. The authentication manager uses an appropriate provider to read detailed user information (such as username, password, e-mail address, and the user's access rights or privileges) from the back-end user base, authenticates the user, and stores the information in an `Authentication` object.

Finally, APF saves the `Authentication` object in the security context created earlier by SIF. The `Authentication` object stored in the security context will be used later to make authorization decisions.

Configure APF as shown in Listing 3:

Listing 3. Configuring APF

```
<bean id="authenticationProcessingFilter"
    class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
    <property name="authenticationManager"
        ref="authenticationManager" />
    <property name="filterProcessesUrl"
        value="/j_acegi_security_check" />
    <property name="defaultTargetUrl"
        value="/protected/protected1.jsp" />
    <property name="authenticationFailureUrl"
        value="/login.jsp?login_error=1" />
</bean>
```

You can see from this code that APF depends on the four parameters discussed above. Each parameter is configured as a `<property>` tag in Listing 3.

Logout Processing Filter

Acegi uses a *Logout Processing Filer* (LPF) to manage logout processing. LPF operates when a logout request comes from a client. It identifies the logout request from the URL invoked by the client.

LPF is configured as shown in Listing 4:

Listing 4. Configuring LPF

```
<bean id="logoutFilter" class="org.acegisecurity.ui.logout.LogoutFilter">
    <constructor-arg value="/logoutSuccess.jsp"/>
    <constructor-arg>
        <list>
            <bean class="org.acegisecurity.ui.logout.SecurityContextLogoutHandler"/>
        </list>
    </constructor-arg>
</bean>
```

You can see that LPF takes two parameters in its constructor: the logout success URL (`/logoutSuccess.jsp`) and a list of handlers. The logout success URL is used to redirect the client after the logout process is complete. Handlers perform the actual logout process; I have configured only one handler because it is enough to invalidate the HTTP session. I will discuss more handlers in the second article of this series.

Exception Translation Filter

The *Exception Translation Filter* (ETF) handles exceptional cases in the authentication and authorization procedure, such as when authorization fails. In these exceptional cases, ETF decides what to do.

For example, if a non-authenticated user attempts to access a protected resource, ETF serves the login page inviting the user to authenticate. Similarly, in case of authorization failure, you can configure ETF to serve an Access Denied page.

ETF is configured as shown in Listing 5:

Listing 5. Configuring ETF

```
<bean id="exceptionTranslationFilter"
    class="org.acegisecurity.ui.ExceptionTranslationFilter">
    <property name="authenticationEntryPoint">
        <bean
            class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
            <property name="loginFormUrl" value="/login.jsp" />
        </bean>
    </property>
    <property name="accessDeniedHandler">
        <bean class="org.acegisecurity.ui.AccessDeniedHandlerImpl">
            <property name="errorPage" value="/accessDenied.jsp" />
        </bean>
    </property>
</bean>
```

As you can see from Listing 5, ETF takes two parameters named `authenticationEntryPoint` and `accessDeniedHandler`. The `authenticationEntryPoint` property specifies the login page and the `accessDeniedHandler` specifies the Access Denied page.

Interceptor filters

Acegi's *interceptor filters* are used to make authorization decisions. You need to configure interceptor filters to act after APF has performed a successful authentication. Interceptors use your application's access control policy to make authorization decisions.

The next article in this series shows you how to design access control policies, how to host them on a directory service, and how to configure Acegi to read your access control policy. For the moment, however, I'll stick to showing you how to configure a simple access control policy using Acegi. Later in the article, you'll see the simple access control policy used in building a sample application.

You can divide configuring a simple access control policy into two steps:

1. Writing the access control policy.
2. Configuring Acegi's interceptor filter according to the policy.

Step 1. Writing a simple access control policy
Start by looking at Listing 6, which shows how to define a user and the user's role:

Listing 6. Defining simple access control policy for a user

```
    alice=123,ROLE_HEAD_OF_ENGINEERING
```

The access control policy shown in Listing 6 defines a user named `alice`, whose password is `123` and whose role is `ROLE_HEAD_OF_ENGINEERING`. (The next section explains how you can define any number of users and their roles in a file and then configure an Acegi Interceptor filter to use the file.)

Step 2. Configuring Acegi's interceptor filter
Interceptor filters use three components to make authorization decisions, which I have configured in Listing 7:

Listing 7. Configuring an interceptor filter

```
<bean id="filterInvocationInterceptor"
    class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
    <property name="authenticationManager" ref="authenticationManager" />
    <property name="accessDecisionManager" ref="accessDecisionManager" />
    <property name="objectDefinitionSource">
        <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /protected/**=ROLE_HEAD_OF_ENGINEERING
            /**=IS_AUTHENTICATED_ANONYMOUSLY
        </value>
    </property>
    <!--  More properties of the interceptor filter -->
</bean>
```

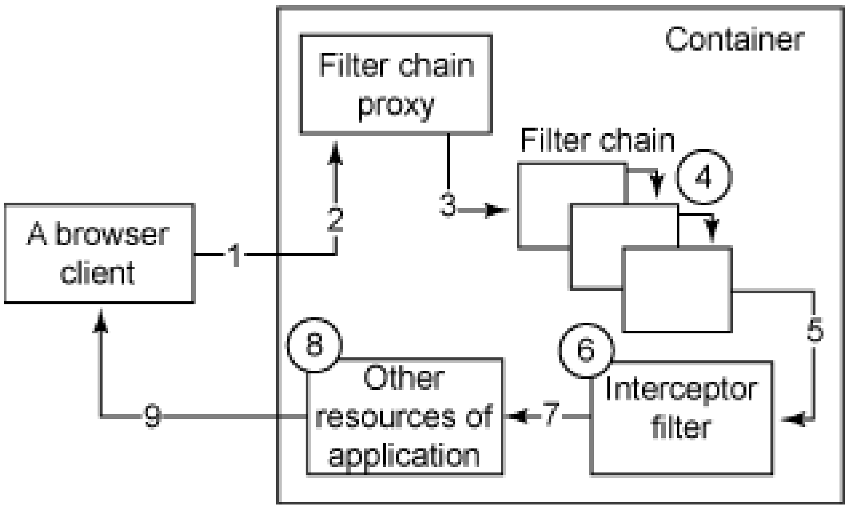
As Listing 7 shows, the three components you need to configure are the `authenticationManager`, `accessDecisionManager`, `objectDefinitionSource`:

- The `authenticationManager` component is the same as the authentication manager discussed when I introduced the Authentication Processing Filter. The interceptor filter can use the `authenticationManager` to re-authenticate a client during the authorization process.
- The `accessDecisionManager` component manages the process of authorization, which the next article of this series will discuss in more detail.
- The `objectDefinitionSource` component contains access control definitions according to which authorization will take place. For example, the `objectDefinitionSource` property in Listing 7 contains two URLs (`/protected/*` and `/*`) in its value. The value defines roles for these URLs. The role for the `/protected/*` URL is `ROLE_HEAD_OF_ENGINEERING`. You can define any roles you like, according to your application requirements.
  Recall from Listing 6 that you defined `ROLE_HEAD_OF_ENGINEERING` for the user named `alice`. This means `alice` will be able to access the `/protected/*` URL.

How filters work

As you have learned already, Acegi's components are dependent on each other to secure your applications. Later in the series, you will see how to configure Acegi to apply security filters in a specific order, thus creating a *chain of filters*. For this purpose, Acegi maintains a filter chain object, which wraps all the filters you have configured to secure your application. Figure 1 shows the life cycle of the Acegi filter chain, which starts when a client sends an HTTP request to your application. (Figure 1 shows a container serving a browser client.)

Figure 1. A container hosting Acegi's filter chain to securely serve a browser client



The following steps describe the life cycle of the filter chain:

1. A browser client sends an HTTP request to your application.
2. The container receives the HTTP request and creates a request object that wraps information contained in the HTTP request. The container also creates a response object, which different filters can process to prepare an HTTP response for the requesting client. The container then invokes Acegi's filter chain proxy, which is a proxy filter. The proxy knows the sequences of actual filters to be applied. When the container invokes the proxy, it passes request, response, and filter chain objects to it.
3. The proxy filter invokes the first filter in the filter chain, passing request, response, and filter chain objects to the filter.
4. Filters in the chain do their processing one by one. A filter can terminate its processing at any time by calling the next filter in the chain. A filter may even choose to not perform any processing at all (for example, APF might terminate its processing upon discovering that an incoming request did not require authentication).
5. When authentication filters have finished processing, they pass request and response objects to the interceptor filter configured in your application.
6. The interceptor decides whether the requesting client is authorized to access the requested resource.
7. The interceptor transfers control to your application (for example, the JSP page requested by the client in case of successful authentication and authorization).

8. Your application writes contents over the response object.

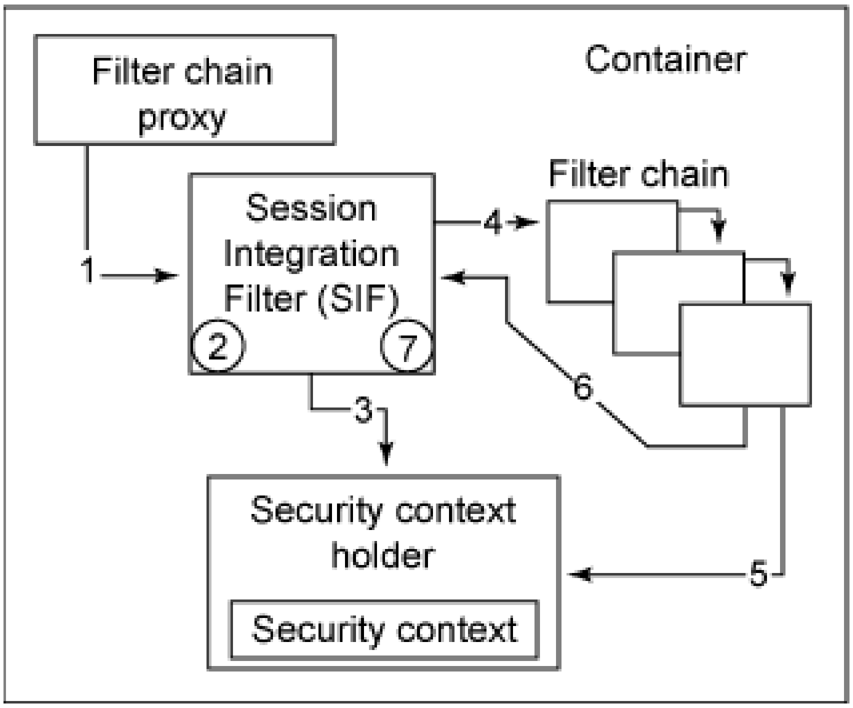
9. The response object is now ready. The container translates the response object into an HTTP response and sends the response to the requesting client.

To help you further understand Acegi filters, I'll give you a closer look at the operation of two of them: the Session Integration Filter and the Authentication Processing Filter.

How SIF creates a security context

Figure 2 shows the steps involved in SIF's creation of a security context:

Figure 2. SIF creates a security context



Now consider these steps in detail:

1. Acegi's filter chain proxy invokes SIF and passes request, response, and filter chain objects to it. Note that normally you will configure SIF as the first filter in the filter chain.

2. SIF checks whether it has already processed this Web request or not. If it finds that it has, it does no further processing and transfers control to the next filter in the filter chain (see Step 4 below). If SIF finds that this is the first time it has been called during the given Web request, it sets a flag,

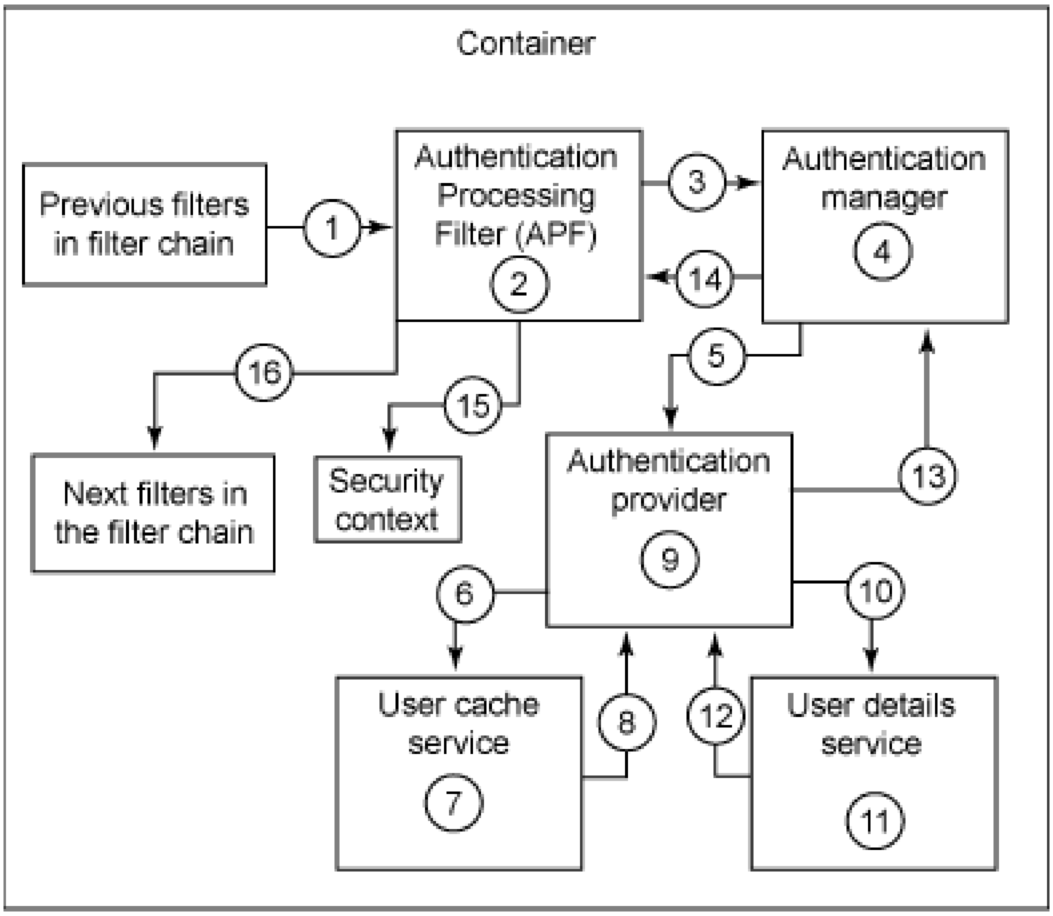
which will be used next time to indicate that SIF has been called.

3. SIF checks whether a session object exists and contains a security context. It retrieves the security context from the session object and places it in a temporary placeholder called *security context holder*. If the session object does not exist, SIF creates a new security context and puts it in the security context holder. Note that the security context holder exists in the application scope so that it is accessible to other security filters.
4. SIF calls the next filter in the filter chain.
5. Other filters may edit the security context.
6. SIF receives control after the filter chain processing completes.
7. SIF checks whether any other filter changed the security context during its processing (for example, APF may have stored user details in the security context). If so, it updates the security context in the session object. This means that any changes made to the security context during filter chain processing now reside in the session object.

How APF authenticates a user

Figure 3 shows the steps involved in APF's authentication of a user:

Figure 3. APF authenticates a user



Now consider these steps in detail:

1. The previous filter in the filter chain passes request, response, and filter chain objects to APF.
2. APF creates an authentication token with the username, password, and other information fetched from the request object.
3. APF passes the authentication token to the authentication manager.
4. The authentication manager may contain one or more authentication providers. Each provider supports exactly one type of authentication. The manager checks which of its providers support the authentication token it received from APF.
5. The authentication manager passes the authentication token to the provider suitable for authentication.
6. The authentication provider extracts the username from the authentication token and passes it to a service called *user cache service*. Acegi

maintains a cache of users who have been authenticated. The next time the user signs in, Acegi can load his or her details (such as username, password, and privileges) from the cache instead of reading from back-end data storage. This improves performance.

7. The user cache service checks whether details of the user exist in the cache.

8. The user cache service returns the details of the user to the authentication provider. If the cache does not contain user details, it returns null.

9. The authentication provider checks whether the cache service returned details of the user or null.

10. If the cache returned null, the authentication provider passes the username (extracted in Step 6) to another service called *user details service*.

11. The user details service communicates with the back-end data storage (such as a directory service) that contains details of the user.

12. The user details service returns details of the user or throws an authentication exception if it cannot find details of the user.

13. If either the user cache service or the user details service returns valid user details, the authentication provider matches the security token (such as a password) supplied by the user with the password returned by the cache or user details service. If a match is found, the authentication provider returns the details of the user to the authentication manager. Otherwise, it throws an authentication exception.

14. The authentication manager returns details of the user back to APF. The user is now successfully authenticated.

15. APF saves the user details in the security context created in Step 3 of Figure 2.

16. APF transfers control to the next filter in the filter chain.

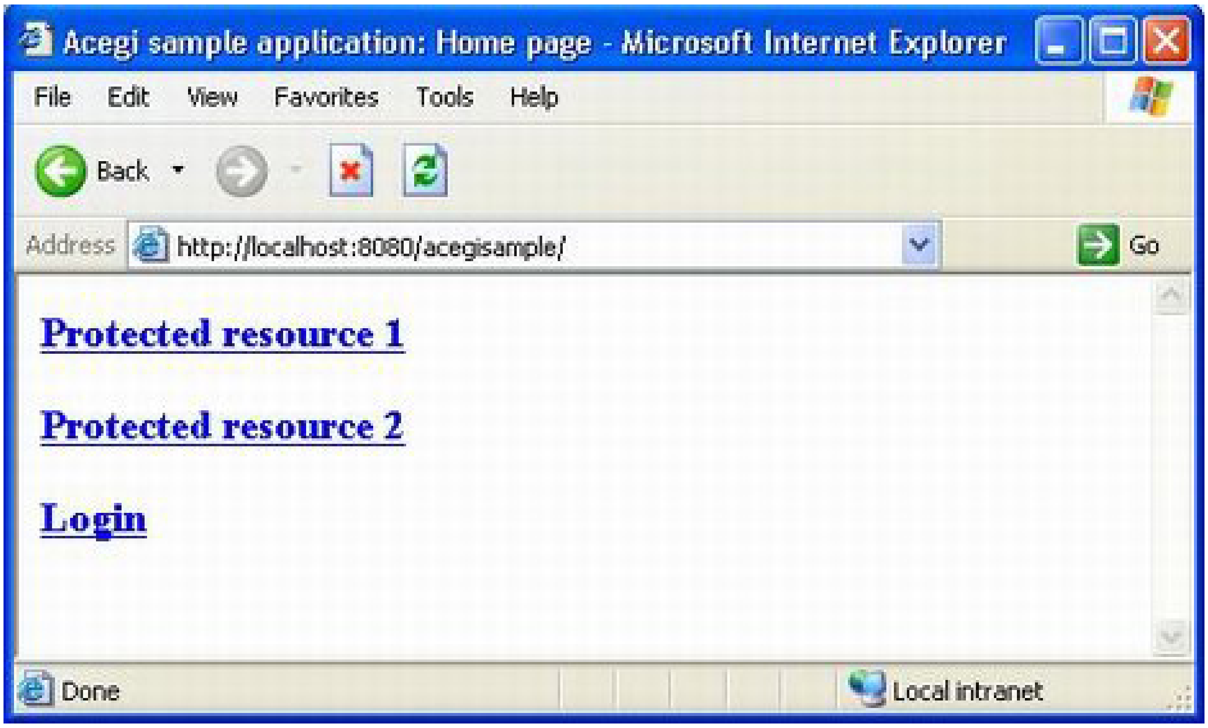
A simple Acegi application

You have learned quite a bit about Acegi in this article, so I'll conclude with a look at what you can do with what you've learned so far. For this simple demonstration, I designed a sample application (see Download) and configured Acegi to secure some of its resources.

The sample application contains five JSP pages: index.jsp, protected1.jsp, protected2.jsp, login.jsp, and accessDenied.jsp.

index.jsp is the welcome page of the application. It presents three hyperlinks to the user, as shown in Figure 4:

Figure 4. Welcome page of the sample application


Acegi sample application: Home page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost:8080/acegisample/
Go
Protected resource 1
Protected resource 2
Login
Done
Local intranet

Two of the links shown in Figure 4 point to protected resources (protected1.jsp and protected2.jsp) and the third link points to a login page (login.jsp). The accessDenied.jsp page is presented only if Acegi finds that the user is not authorized to access a protected resource.

If the user attempts to access any of the protected resources, the sample application presents the login page. When the user signs in using the login page, the application automatically redirects to the requested protected resource.

The user can directly request the login page by clicking a third link (Login) on the welcome page. In this case, the application presents the login page where the user can sign in. After signing in, the application redirects the user to protected1.jsp, which is the default resource presented whenever a user signs in without requesting a particular protected resource.

Configuring the sample application

The source code download for this article contains an XML configuration file named acegi-config.xml, which contains the configuration of Acegi's filters. The configuration should be familiar to you based on the examples in the discussion of security filters.

I have also written a `web.xml` file for the sample application, as shown in Listing 8:

Listing 8. The web.xml file for the sample application

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/acegi-config.xml</param-value>
    </context-param>
    <filter>
        <filter-name>Acegi Filter Chain Proxy</filter-name>
        <filter-class>
            org.acegisecurity.util.FilterToBeanProxy
        </filter-class>
        <init-param>
            <param-name>targetClass</param-name>
            <param-value>
                org.acegisecurity.util.FilterChainProxy
            </param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>Acegi Filter Chain Proxy</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

The web.xml file configures the following:

- The URL of the `acegi-config.xml` file in a `<context-param>` tag.
- The name of Acegi's filter chain proxy class in a `<filter>` tag.
- The mapping of URLs to Acegi's filter chain proxy in a `<filter-mapping>` tag. Note that you can simply map all URLs of your application (`/*`) to Acegi's filter chain proxy. Acegi applies security to all URLs you map to Acegi's filter chain proxy.
- An application context loader in a `<listener>` tag, which loads Spring's IOC framework

Deploy and run the application

Deploying and running the sample application is very simple. You just need to do two things:

1. Copy the acegisample.war file from the source code download of this tutorial into the `webapps` directory of your Tomcat installation.

2. Download and unzip acegi-security-1.0.3.zip from the Acegi Security System home page. You will find a sample application named acegi-security-sample-tutorial.war. Unzip the war file to extract all the jars you find in its WEB-INF/lib folder. Copy all the JARs from the WEB-INF/lib folder into the WEB-INF/lib folder of theacegisample.war application.

Now you are ready to try the sample application. Start Tomcat and point your browser to `http://localhost:8080/acegisample/`.

You will see the welcome page shown in Figure 4, but this time it will be live. Go ahead and see what happens when you try to access different links presented on the welcome page.

In conclusion

In this first article in the *Securing Java applications with Acegi* series, you have learned about the features, architecture, and components of Acegi Security System. You've learned quite a bit about Acegi's security filters, which are integral to its security framework. You've also learned how to configure component dependencies using an XML configuration file, and seen Acegi's security filters at work in a sample application that implements URL-based security.

The security technique described in this article is fairly simple, and so are the Acegi facilities used to implement it. The next article in this series will get you started with some more advanced uses of Acegi, beginning with writing an access control policy and storing it in a directory service. You will also learn how to configure Acegi to interact with the directory service to implement your access control policy.

Downloads

Description	Name	Size	Download method
Source code for this article	j-acegi1.zip	10KB	HTTP

Information about download methods

Resources

Learn

- *Securing Java applications with Acegi* series: Get an introduction to using Acegi Security System to secure Java enterprise applications.
    - "Securing Java applications with Acegi, Part 2: Working with an LDAP directory server" (Bilal Siddiqui, developerWorks, May 2007): This article demonstrates the working of directory servers with Acegi.
    - "Securing Java applications with Acegi, Part 3: Access control for Java objects" (Bilal Siddiqui, developerWorks, September 2007): This article demonstrates access control for Java objects.
    - "Securing Java applications with Acegi, Part 4: Protecting JSF applications" (Bilal Siddiqui, developerWorks, February 2008): This article helps you configure JSF and Acegi to work together in a servlet container and explores how JSF and Acegi components cooperate with one another.
- The Acegi Security System home page: Your first stop for reference documentation and to download Acegi.
- "Java security evolution and concepts, Part 1: Security nuts and bolts" (Raghavan Srinivas, developerWorks, May 2000): An overview of Java security concepts and terminology.
- "Java security, Part 1: Crypto basics" (Brad Rubin, developerWorks, July 2002): Launches a comprehensive tutorial introduction to application security on the Java platform. See also "Java security, Part 2: Authentication and authorization."
- "Web app security using Struts, servlet filters, and custom taglibs" (Swaminathan Radhakrishnan, developerWorks, September 2004): Showcases the ingenuity that often goes into developing Java security solutions.
- "Introduction to Spring 2 and JPA" (Sing Li, developerWorks, August 2006): This tutorial gets you started with the second-generation Spring framework.
- The Spring reference documentation: Spring documentation from the source.
- "The Essentials of Filters" (Sun Developer Network): Learn more about filters and what you can do with them.
- Java SE Security: A repository of Java security information.
- IBM security providers (Yanni Zhang, Audrey Timkovich, and John Peck; developerWorks, October 2004): Learn about the enhanced security features of the IBM developer kit for the Java platform (1.4.2 Java SDK) and how it differs from the Sun Microsystems release.

- developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download Acegi Security System: Secure your Java enterprise applications.

Discuss

- developerWorks blogs: Get involved in the developerWorks community.

About the author

Bilal Siddiqui
Bilal Siddiqui is an electronics engineer, an XML consultant, and the co-founder of WaxSys, a company focused on simplifying e-business. After graduating in 1995 with a degree in electronics engineering from the University of Engineering and Technology, Lahore, he began designing software solutions for industrial control systems. Later, he turned to XML and used his experience programming in C++ to build Web- and Wap-based XML processing tools, server-side parsing solutions, and service applications. Bilal is a technology evangelist and a frequently-published technical author.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.