

Securing Java applications with Acegi, Part 5: Protecting JavaBeans in JSF applications

Configurable security for beans used in JSF applications

Skill Level: Intermediate

[Bilal Siddiqui](#)

Freelance consultant

WaxSys

01 Apr 2008

Bilal Siddiqui concludes his [series](#) by demonstrating how to use Acegi to secure access to JavaBeans in Java™ Server Faces (JSF) applications. You can configure secure beans in a variety of ways, including using Acegi-secured inversion-of-control (IOC) beans directly in your JSF tags.

This five-part [series](#) of articles introduces Acegi Security System and demonstrates how to use Acegi to secure enterprise Java applications. This final article of the series continues a discussion of using Acegi to secure JSF applications. In [Part 4](#) I showed how you can secure a JavaServer Faces (JSF) page using Acegi without writing any Java code. I also provided an in-depth explanation of events that happen when you deploy your JSF-Acegi application and when a user accesses it. This time I focus on techniques for securing JavaBeans in your JSF applications.

I'll start the discussion by showing that the bean security concepts I demonstrated in [Part 3](#) of the series can apply to JSF applications too, but are not ideal for them. I'll then demonstrate a couple of new techniques that are especially suitable for securing JavaBeans used in JSF applications. I'll wrap up the discussion by presenting a four-point strategy that lets you use Acegi to secure beans in your JSF applications without writing any Java security code.

A simple technique

The simplest way to use secure beans in a JSF application is to follow the same approach used in the five steps in Listing 4 of [Part 3](#). There I fetch the Spring framework's Web application context object from the servlet context. The Web application context can be used later to access beans securely. [Listing 1](#), below, shows the use of Web application context in a JSF page:

Listing 1. Fetching Web application context from the servlet context and using it in a JSF page

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@page import="sample.CatalogBean"%>
<%@page import="org.springframework.web.context.support.WebApplicationContextUtils" %>
<%@page import="org.springframework.web.context.WebApplicationContext" %>

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
<title>Acegi simple method security application: TEST PAGE</title>
</head>
<body>

    <f:view>
        <h2>
            <h:outputText value="Protected Resource 1:"/>
        </h2>
        <%
            try {
                WebApplicationContext webApplicationContext =
                    WebApplicationContextUtils.getWebApplicationContext(
                        this.getServletConfig().getServletContext());
                CatalogBean privateCatalog = (CatalogBean)
                    webApplicationContext.getBean("privateCatalog");
                String privateData = privateCatalog.getData();
                request.setAttribute("privateData", privateData);
            }
            catch (Exception e) { }
        %>

        <h3>
            <h:outputText value="#{privateData}" />
        </h3>

    </f:view>
</body>
</html>
```

You can see that [Listing 1](#) uses a class named `WebApplicationContextUtils` to fetch an instance of the Web application context.

`WebApplicationContextUtils` is a utility class that Spring provides.

After obtaining the Web application context, you can call its `getBean()` method to obtain a reference to any bean configured in Acegi's configuration file. Then you can call the bean's getter methods and store the data returned by the getter methods as a parameter in the servlet request object. These steps allow the `<outputText>` tag in [Listing 1](#) to serve the data to the user.

Not the best way

Managing bean data directly as shown in [Listing 1](#) is simple but inadvisable. The approach violates JSF's Model-View-Controller (MVC) architecture (see [Resources](#)), which requires the use of *model beans* to hold application data. It's better not to use this strategy in JSF applications, except perhaps in very simple cases.

Bean dependency

Spring's IOC framework provides a useful way of managing model beans by expressing beans' dependencies on one another. I explain the concept of bean dependency in IOC in the "Architecture and components" section of [Part 1](#).

JSF provides comprehensive functionality to manage an application's model beans. Such beans — called *managed beans* — are used in most JSF applications, so most real-world JSF applications need to secure managed beans.

The rest of this article discusses two strategies for using secure beans in JSF applications:

- Using Acegi to secure JSF managed beans
- Using [inversion-of-control \(IOC\) beans](#) that are secured by Acegi directly in JSF tags

Securing JSF managed beans

Look at the JSF page shown in [Listing 2](#), which uses a JSF managed bean named `catalog`:

Listing 2. Simple JSF page using a managed bean

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
<title>JSF Acegi simple method security application: TEST PAGE</title>
</head>

<body>
  <f:view>
    <h2>
      <h:outputText value="Protected Resource 1:" />
    </h2>
    </br>
    <h3>
      <h:outputText value="#{catalog.publicData}" />
    </br>
    <h:outputText value="#{catalog.privateData}" />
  </f:view>
</body>
</html>
```

```
        </h3>
    </f:view>
</body>
</html>
```

[Listing 2](#) uses two JSF `<outputText>` tags. The first `<outputText>` tag has a `value` attribute of `#{catalog.publicData}`, and the second has a `value` attribute of `#{catalog.privateData}`. These two tags use the `catalog` bean's `publicData` and `privateData` properties, which serve public and private catalog data, respectively.

Recall from the "Accessing proxied Java objects" section of [Part 3](#) that I configured two Acegi beans named `publicCatalog` and `privateCatalog`. Now I am going to map Part 3's `publicCatalog` bean (an unsecured bean meant for public access) to the `catalog` bean's `publicData` property. Similarly, I'll map Part 3's `privateCatalog` (a secured proxied bean configured in [Listing 3 of Part 3](#)) to the `privateData` property of the `catalog` managed bean of [Listing 2](#) above. Once properly mapped, the `catalog` bean simply acts as a wrapper for the JSF catalog application's public and private data.

Defining a managed bean

[Listing 3](#) shows how to define the `catalog` bean so that its `publicData` and `privateData` properties are mapped to Acegi's `publicCatalog` and `privateCatalog` beans, respectively:

Listing 3. Mapping catalog properties to Acegi's beans

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

    <managed-bean>
        <managed-bean-name>catalog</managed-bean-name>
        <managed-bean-class>sample.Catalog</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>

        <managed-property>
            <property-name>publicData</property-name>
            <value>#{publicCatalog.data}</value>
        </managed-property>

        <managed-property>
            <property-name>privateData</property-name>
            <value>#{privateCatalog.data}</value>
        </managed-property>
    </managed-bean>

    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
```

```
</application>
</faces-config>
```

[Listing 3](#) actually shows a configuration file for JSF. Its root tag is `<faces-config>`, which is familiar to most JSF programmers. The root `<faces-config>` tag contains two child tags named `<managed-bean>` and `<application>`. Now I'll explain these two tags in detail.

Declaring bean properties in a faces configuration file

The `<managed-bean>` tag in [Listing 3](#) defines the `catalog` bean and its properties. The `<managed-bean>` tag has three child tags — `<managed-bean-name>`, `<managed-bean-class>`, and `<managed-bean-scope>` — and a couple of `<managed-property>` tags. The first two child tags define the bean's name (`catalog`) and class (`sample.Catalog`), respectively.

Each `<managed-property>` tag in [Listing 3](#) defines a property of the `catalog` bean. Each `<managed-property>` tag has two children — `<property-name>` and `<value>` — that define the property's name and value, respectively. You can see from [Listing 3](#) that the first property's name is `publicData`, and that its value is `#{publicCatalog.data}`. Similarly, the name of the second property is `privateData`, and its value is `#{privateCatalog.data}`.

The two values are actually expressions that resolve to properties of other managed beans. The first expression (`#{publicCatalog.data}`) resolves the `publicCatalog` bean's data property. Similarly, the second expression (`#{privateCatalog.data}`) resolves to the `privateCatalog` bean's data property.

JSF provides a mechanism that resolves expressions such `#{publicData.data}` to instances of actual managed beans. I'll discuss JSF's expression-resolving mechanism of JSF momentarily (in "[Defining an expression resolver](#)").

However, there is a problem here. The JSF configuration file in [Listing 3](#) does not contain the managed beans named `publicCatalog` and `privateCatalog`. Recall that I configured `publicCatalog` and `privateCatalog` IOC beans (rather than a JSF managed bean) in the "Accessing proxied Java objects" section of [Part 3](#). Therefore, JSF's expression resolving-mechanism must be able to resolve Acegi's IOC beans.

Defining an expression resolver

JSF's `javax.faces.el.VariableResolver` class is a default expression resolver capable of resolving expressions to JSF's managed beans. However, the `VariableResolver` can't resolve IOC beans.

JSF provides an extensibility mechanism that allows application developers to write their own expression resolvers. Spring provides an expression resolver for JSF in a class named

`org.springframework.web.jsf.DelegatingVariableResolver`. The `DelegatingVariableResolver` class can resolve expressions to IOC beans. `DelegatingVariableResolver` also uses the default `VariableResolver` to resolve expressions to JSF managed beans.

To use the Spring's `DelegatingVariableResolver`, you must configure it in the JSF configuration file. This is the purpose of including the `<application>` tag in [Listing 3](#) (reproduced in [Listing 4](#) for quick reference):

Listing 4. The `<application>` tag

```
<faces-config>
  .....
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
</faces-config>
```

The `<application>` tag in [Listing 4](#) contains just one child tag, named `<variable-resolver>`, which is meant to configure external resolvers for your JSF application. The `<variable-resolver>` tag wraps name of Spring's resolver class (`org.springframework.web.jsf.DelegatingVariableResolver`), which handles resolving of expressions to IOC beans.

Implementing JSF and IOC beans

You have seen how to configure your JSF application to use Acegi's IOC beans. Now you can have a look at the three beans you have just configured.

[Listing 5](#) shows the implementation of the `Catalog` class, whose instance — named `catalog` — you configured in JSF as a managed bean:

Listing 5. The `Catalog` class

```
package sample;

public class Catalog
{
    private String publicData = null;
    private String privateData = null;

    public Catalog () {
    }
}
```

```
public void setPublicData(String publicData) {
    this.publicData = publicData;
}

public void setPrivateData(String privateData) {
    this.privateData = privateData;
}

public String getPublicData() {
    return publicData;
}

public String getPrivateData() {
    return privateData;
}
} //Catalog
```

You can see from [Listing 5](#) that the `Catalog` class simply contains getter and setter methods from its `publicData` and `privateData` properties. The JSF framework will call the getter and setter methods, as I'll explain in the next section.

Now look at the implementations of the two IOC beans (`publicCatalog` and `privateCatalog`) in [Listing 6](#):

Listing 6. The `publicCatalog` and `privateCatalog` IOC beans

```
//PublicCatalog
package sample;

public class PublicCatalog implements CatalogBean {

    public PublicCatalog () { }

    public String getData() {
        return "This is public catalog data";
    }
}

//PrivateCatalog
package sample;

public class PrivateCatalog implements CatalogBean {

    public PrivateCatalog () { }

    public String getData() {
        return "This is private catalog data";
    }
}
```

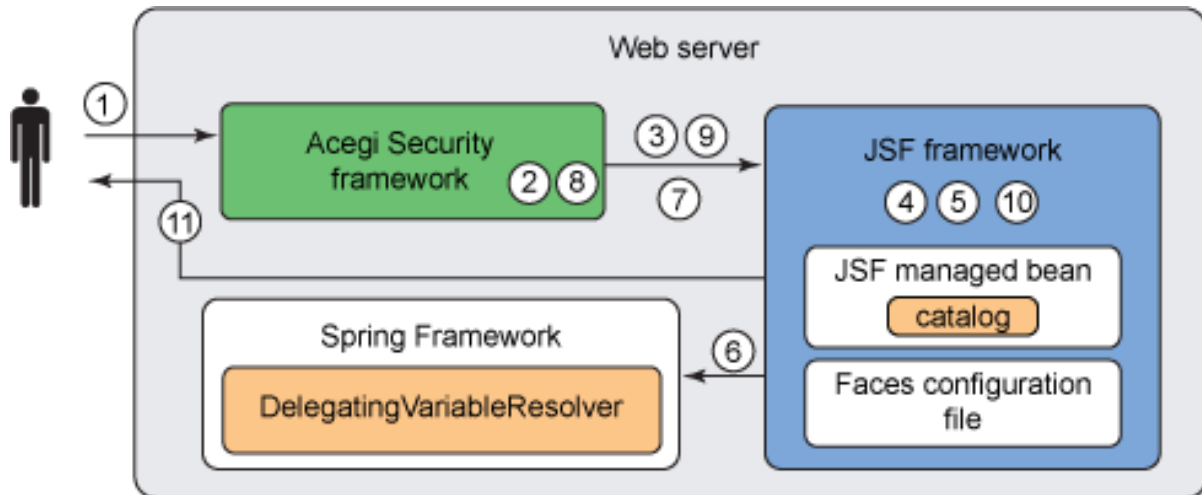
You can see in [Listing 6](#) that I have hardcoded the actual public and private data in the two IOC beans. In a real application, these beans would read the data from a database.

You have seen all the components and configurations that you need to secure the data wrapped inside your JSF managed beans. Now it's time to see how JSF and Acegi work together to use the components and configurations.

JSF and Acegi working together to secure managed beans

The sequence of events shown in [Figure 1](#) takes place when a user tries to access the JSF page in [Listing 2](#). I have included all the events that enable Acegi's URL security as well as bean security in JSF applications.

Figure 1. JSF and Acegi components working together



The events shown in [Figure 1](#) follow this sequence:

1. The user accesses the JSF page.
2. Acegi checks whether the user is authorized to access the JSF page. (See the "Processing a request for an Acegi protected JSF page" section of [Part 4](#).)
3. If the authorization process succeeds, control is transferred to the faces servlet, which prepares to serve the JSF page.
4. While preparing, JSF finds the `catalog` bean in the JSF page shown in [Listing 2](#).
5. JSF checks the configuration file shown in [Listing 3](#) to find definition of the `catalog` bean and instantiates it. JSF also checks properties of the `catalog` bean in the configuration file. It finds that the `catalog` bean's `publicData` and `privateData` properties are mapped to `publicCatalog` and `privateCatalog` beans, which are not configured as JSF managed beans in [Listing 3](#).
6. JSF uses Spring's `DelegatesVariableResolver` variable resolver (configured in [Listing 4](#)) to resolve the `publicCatalog` and

`privateCatalog` beans.

7. JSF uses Acegi to invoke getter methods of the `publicCatalog` and `privateCatalog` beans to fetch public and private data.
8. Acegi again executes its authorization process to access the beans. (See [Part 3](#) for a detailed discussion of this authorization process for Java objects.)
9. If Acegi finds that the user is authorized to access the beans, it invokes the getter methods, fetches public and private data, and provides the data to JSF.
10. JSF calls the `catalog` bean's setter methods to set public and private data in the `catalog` bean.
11. JSF executes its life cycle and serves the JSF page.

A sample JSF-Acegi application with secure managed beans

A sample application named `JSFAcegiSampleWithSecureManagedBeans` accompanies this article (see [Download](#)). It uses the technique you've learned in the previous couple of sections to secure access to data wrapped inside JSF managed beans.

To deploy the sample application, follow the two steps in the "Deploy and run the application" section of [Part 1](#). You also need to download and unzip `jsf-1_1_01.zip` from Sun's JSF site (see [Resources](#)). Copy all the files you find in `jsf-1.1.X.zip` into the `WEB-INF/lib` folder of your `JSFAcegiSampleWithSecureManagedBeans` application. You also need to download the `cglib-full-2.0.2.jar` file (which you used in [Part 3](#) of this series) and copy it in `WEB-INF/lib` folder of the `JSFAcegiSampleWithSecureManagedBeans` application. You can invoke the sample application by accessing `http://localhost:8080/JSFAcegiSampleWithSecureManagedBeans` from your browser.

Using Acegi's IOC beans directly in a JSF application

You have learned how to map properties of JSF managed beans to Acegi's IOC beans and how to configure `DelegatingVariableResolver` to resolve expressions to IOC beans. You have also seen how JSF and Acegi work together to secure access to bean data.

Alternatively, you can use IOC beans directly in your JSF pages, as shown in the JSF page in [Listing 7](#):

Listing 7. Using IOC beans directly in a JSF page

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
<title>JSF Acegi simple method security application: TEST PAGE</title>
</head>

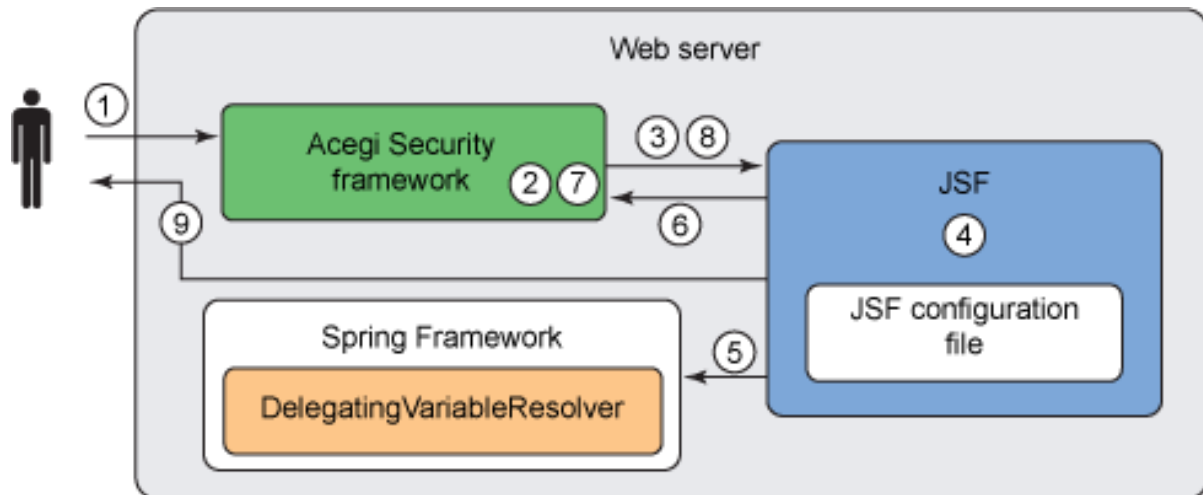
<body>
  <f:view>
    <h2>
      <h:outputText value="Protected Resource 1:" />
    </h2>
    <br>
    <h3>
      <h:outputText value="#{publicCatalog.data}" />
    </h3>
    <br>
    <h3>
      <h:outputText value="#{privateCatalog.data}" />
    </h3>
  </f:view>
</body>
</html>
```

[Listing 7](#) is similar to the JSF page in [Listing 2](#). The only difference between the two is the value attributes of the `<outputText>` tags. In [Listing 2](#), the value attributes refer to `catalog`, which is a JSF managed bean. In [Listing 7](#), the value attribute refers directly to IOC beans (that is, `publicCatalog` and `privateCatalog`). This means that when a user accesses the JSF page in [Listing 7](#), JSF directly resolves Acegi IOC using Spring's `DelegatingVariableResolver`.

Note that in order to resolve IOC beans used in a JSF page, `DelegatingVariableResolver` works in the same manner that I explained while discussing [Figure 1](#).

[Figure 2](#) shows sequence of events that happens when user accesses the JSF page of [Listing 7](#).

Figure 2. JSF and Acegi components working to serve a JSF page with secure IOC beans



The sequence of events shown in [Figure 2](#) has only minor differences from the sequence in [Figure 1](#):

1. The user accesses the JSF page.
2. Acegi checks whether the user is authorized to access the JSF page.
3. If the authorization process succeeds, control is transferred to JSF, which prepares to serve the JSF page.
4. While preparing, JSF finds the `publicCatalog` and `privateCatalog` beans in the JSF page in [Listing 7](#).
5. JSF checks the configuration file of [Listing 3](#) and finds that `publicCatalog` and `privateCatalog` beans are not configured as JSF managed beans in the configuration file. JSF uses Spring's `DelegatingVariableResolver` to resolve the `publicCatalog` and `privateCatalog` beans.
6. JSF uses Acegi to invoke getter methods of the `publicCatalog` and `privateCatalog` beans to fetch public and private data.
7. Acegi executes its authorization process to access the beans.
8. If Acegi finds that the user is authorized to access the beans, it invokes the getter methods, fetches public and private data, and provides the data to JSF.
9. JSF executes its life cycle and serves the JSF page.

You can see that the JSF page of [Listing 7](#) does not use any managed beans, so [Figure 2](#) does not include events related to JSF managed beans.

A second sample application, named `JSFAcegiSampleWithIOCBears`, is in this article's source code (see [Download](#)). `JSFAcegiSampleWithIOCBears` uses the JSF page in [Listing 7](#) to demonstrate the use of IOC beans in JSF pages.

Using Acegi to secure existing JSF applications

The preceding section demonstrated that you can use IOC beans directly in your JSF applications. If you have an existing JSF application and want to use Acegi to secure it, you just need to perform four configuration steps:

1. Write Acegi's configuration file as described in first three articles in this series.
2. Write a `web.xml` file, as described in [Part 4](#).
3. Use Spring's `DelegatingVariableResolver` in JSF's configuration file as in this article's "[Defining an expression resolver](#)" section.
4. Reconfigure the JSF managed beans that you wish to secure as IOC beans by declaring your beans in Acegi's configuration file instead of JSF's configuration file.

This technique lets you can develop your JSF applications without considering security issues. After developing the application, you can follow the four configuration steps to deploy Acegi without writing any Java security code.

Conclusion

In this series of articles, you have learned how to use Acegi to secure your Java applications. You now know how to enforce both URL-based security and method-based security. You've learned how to design access-control policies and host them in a directory server, how to configure Acegi to communicate with a directory service, and how to make authentication and authorization decisions based on the access-control policies hosted there. In this article and the preceding one, you focused on JSF applications and learned how you can secure your JSF applications without writing any Java security code.

Downloads

Description	Name	Size	Download method
Sample code for this article	j-acegi5-source.zip	42KB	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 1: Architectural overview and security filters"](#) (Bilal Siddiqui, developerWorks, March 2007): This article introduces you to the architecture and components of Acegi Security System.
 - ["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 shows how to protect JSF applications that run in a servlet container.
- [Acegi Security System](#): The Acegi Web site is your first stop for reference documentation.
- ["JSF for nonbelievers: Clearing the FUD about JSF"](#) (Richard Hightower, developerWorks, February 2005): This article discusses and demonstrates JSF's MVC framework fundamentals.
- ["Getting started with JavaServer Faces 1.2"](#) (Richard Hightower, developerWorks, December 2007 and January 2008): Get up to speed quickly with the latest JSF version in this two-part tutorial.
- ["Acegi Security System for Spring Framework, Part 1"](#) and ["Acegi Security System for Spring Framework, Part 2"](#): Check out this presentation on Acegi security.
- ["Using JSF technology for XForms applications"](#) (Faheem Khan, developerWorks, February 2005): This tutorial explains the JSF life cycle in detail.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download [cglib-full-2.0.2.jar](#), which Acegi uses to create proxies.
- [JavaServer Faces technology](#): Download the JSF implementation from Sun's Web site.
- [Acegi Security System](#): Download Acegi.

Discuss

- Check out [developerWorks blogs](#) and 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.