10

Acegi Security System for Spring

Every enterprise application requires security in one form or another. The problem domain of the application, its intended user base, and importance to the core business are all major influences on the type of security required. For example, there is a major difference in the security requirements of an Internet banking application from those of an internal telephone directory application. This wide range of security requirements between applications has proven a major challenge in the development of generic security frameworks. In this chapter we will explore the major options available for securing Spring-based applications and examine the recommended security framework, *Acegi Security System for Spring*.

Enterprise Application Security Choices

Developers have a wide range of choices in how to implement security within enterprise applications. Let's start by reviewing the typical requirements of such applications and three major security platforms commonly used with Spring.

Typical Requirements

Although Spring provides a highly flexible framework, most of its users leverage its capabilities to deliver robust, multi-tier enterprise applications. Such applications are usually data-centric, have multiple concurrent users, and are of significant importance to the core business. Hence most enterprise applications must provide a robust approach to security.

Let's briefly recap some of the key terms used when discussing computer security. The term *principal* refers to a user, service, or agent who can perform an operation. A principal presents *credentials* such as a password in order to allow *authentication*, which is the process of establishing the identity of a caller. *Authorization* refers to the process of determining whether a principal (authenticated or non-authenticated) is permitted to perform a given operation.

Four main security concerns must be addressed in typical enterprise applications:

- **Authentication:** Enterprise applications typically need to access a variety of *authentication repositories*. Often these repositories will be in the form of LDAP directories, CRM systems, enterprise single-sign-on solutions, and autonomous, application-specific databases. Depending on the repository, the server may never expose the credentials (in which case authentication is performed only by binding to the repository with the correct credentials) or the credentials may be in a hashed format. Each authentication repository also must track the authorities granted to a principal. Security frameworks must integrate with these types of repositories, or new ones, sometimes simultaneously. There may also be a range of client types for an enterprise application, including web applications, Swing clients, and web services. The security framework needs to deal consistently with authentication requests from any such client type.
- **Web request security:** Many enterprise applications are web-based, often using an MVC framework and possibly publishing web services. Security is often required to protect URI patterns. In addition, web views often require integration with a security framework so content can be generated based on the authorities held by the principal.
- **Service layer security:** Service (business) layers should be secured in all but the simplest of applications. Security is usually best modeled as an *aspect.* Using an AOP-based solution allows service layer implementations to be largely or completely unaware of security. It also eliminates the error-prone and tedious approach of enforcing security in user interface tiers via techniques such as URI pattern filtering. Acegi Security can secure an AOP Alliance `MethodInvocation`, using Spring AOP. It can also secure an AspectJ `JoinPoint`.
- **Domain object instance security:** Java applications generally use domain objects to model the problem domain. Different instances of these domain objects may require different security. A principal may have delete permission to one domain object instance but only read permission to a different domain object instance. Permissions are assigned to a *recipient*, which refers to a principal or a role. The list of permissions assigned to different recipients for a given domain object instance is known as an *access control list*, or ACL.

The following common computer security concern is *not* typical of enterprise applications:

- **Limited privilege execution:** Some Java applications need to execute with limited privileges, as the user is concerned about the trustability of the application. The most common examples are applets, which run within a web browser, or Java Web Start applications, which can automatically download from the Internet. Enterprise applications are typically trusted by the computer executing them, such as a corporate server. As such, it is of unnecessary complexity for most enterprise applications to execute with limited privileges.

Given these requirements, we recommend the *Acegi Security System for Spring* (`http://acegisecurity.sourceforge.net`; Apache 2 license) to handle the security requirements of Spring-managed applications. We will now briefly explore Acegi Security with reference to the other major security platforms that target enterprise application development.

Acegi Security in a Nutshell

Acegi Security is widely used within the Spring community for providing comprehensive security services to Spring-powered applications. It comprises a set of interfaces and classes that are configured through a Spring IoC container. In the spirit of Spring, the design of Acegi Security allows many applications to implement the four common enterprise application security requirements listed previously solely via

declarative configuration settings in the IoC container. Acegi Security is heavily interface-driven, providing significant room for customization and extension.

Acegi Security, like Spring, emphasizes *pluggability*. It is based on the assumption that there is no one-size-fits-all approach for security, instead providing a consistent programming model from a security perspective, while allowing a choice between different strategies to implement key security responsibilities.

We will now briefly examine the four common enterprise application security requirements (authentication, web request security, service layer security, and domain object instance security), and how Acegi Security addresses each. Later in this chapter we will revisit Acegi Security architecture and configuration in more detail, so don't be concerned if you don't follow everything mentioned in the following quick overview.

Authentication

Before the other three security requirements can be addressed, the principal must be authenticated. Acegi Security performs authentication using a pluggable `AuthenticationManager`, which can be defined in a Spring application context as follows:

```
<bean id="authenticationManager"
class="net.sf.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
    </list>
  </property>
</bean>

<bean id="daoAuthenticationProvider"
    class="net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="authenticationDao"><ref local="jdbcDaoImpl"/></property>
</bean>

<bean id="jdbcDaoImpl" class="net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl">
  <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
```

The `authenticationManager` bean will delegate through the list of authentication providers (the preceding code sample shows only one). Authentication providers are capable of authenticating from a given authentication repository. In this case, the `daoAuthenticationProvider` bean will be used for authentication, which uses a data access object to retrieve the user information.

Web Request Security

Acegi Security provides web request security via `FilterSecurityInterceptor` and `SecurityEnforcementFilter`. The latter class is implemented as a `web.xml` filter, with the filter itself defined in the application context:

```
<bean id="securityEnforcementFilter"
    class="net.sf.acegisecurity.intercept.web.SecurityEnforcementFilter">
  <property name="filterSecurityInterceptor">
    <ref local="filterInvocationInterceptor"/>
```

```
      </property>
      <property name="authenticationEntryPoint">
        <ref local="authenticationProcessingFilterEntryPoint"/>
      </property>
    </bean>

    <bean id="authenticationProcessingFilterEntryPoint"

    class="net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
      <property name="loginFormUrl"><value>/acegilogin.jsp</value></property>
    </bean>

    <bean id="filterInvocationInterceptor"
        class="net.sf.sf.intercept.web.FilterSecurityInterceptor">
      <property name="authenticationManager">
        <ref bean="authenticationManager"/>
      </property>
      <property name="accessDecisionManager">
        <ref local="httpRequestAccessDecisionManager"/>
      </property>
      <property name="objectDefinitionSource">
        <value>
          CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
          PATTERN_TYPE_APACHE_ANT
          /secure/super/**=ROLE_WE_DONT_HAVE
          /secure/**=ROLE_SUPERVISOR,ROLE_USER
        </value>
      </property>
    </bean>
```

Most of the preceding is boilerplate code, with the actual URI patterns to protect being defined in the `filter InvocationInterceptor` bean `objectDefinitionSource` property. As you can see, the `filter InvocationInterceptor` bean uses Apache Ant–style paths to express the URI patterns. In addition to Apache Ant paths, regular expressions are supported. The `CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON` directs the filter to match URI patterns in a case-insensitive manner. These declarations would prevent a principal from accessing the `/secure` URI unless they held one of the two indicated roles, and would prevent `/secure/super` being accessed unless the principal held the `ROLE_WE_DONT_HAVE` role.

Service Layer Security

Comprehensive services layer security is provided by Acegi Security, again using the application context. Here we define an AOP Alliance method interceptor, which is autowired via Spring's `DefaultAdvisor AutoProxyCreator`:

```
    <bean id="autoproxy"
        class="org.sf.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

    <bean id="methodSecurityAdvisor"

    class="net.sf.acegisecurityf.intercept.method.aopalliance.MethodDefinitionSourceAdv
    isor"
        autowire="constructor"/>

    <bean id="bankManagerSecurity"
```

```
class="net.sf.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor"
>
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
  </property>
  <property name="accessDecisionManager">
    <ref local="businessAccessDecisionManager"/>
  </property>
  <property name="objectDefinitionSource">
    <value>
      com.acegitech.BankManager.createAccount=ROLE_TELLER
      com.acegitech.BankManager.deposit=ROLE_TELLER
      com.acegitech.BankManager.approveLoan=ROLE_MORTGAGE_CENTRE
      com.acegitech.BankManager.viewAuditTrail=ROLE_INVESTIGATIONS
    </value>
  </property>
</bean>
```

In this example, the `MethodSecurityInterceptor` will intercept any `BankManager` implementation defined in the application context. The syntax of `MethodSecurityInterceptor` is very similar to Spring's transaction support, allowing both fully qualified method names as well as wildcards (for example, `create*=ROLE_TELLER`) to be used. If the indicated roles are not held, `MethodSecurityInterceptor` will throw an `AccessDeniedException`. If the principal is not logged in, an `AuthenticationException` will be thrown.

Domain Object Instance Security

Finally, domain object instance security in Acegi Security leverages the services layer security interceptors. Acegi Security also provides a repository, known as an `AclManager`, that can determine the access control list (ACL) that applies for any domain object instance it is passed:

```
<bean id="aclFolderWriteVoter"
class="net.sf.acegisecurity.vote.BasicAclEntryVoter">
  <property name="processConfigAttribute">
    <value>ACL_FOLDER_WRITE</value>
  </property>
  <property name="processDomainObjectClass">
    <value>com.acegitech.Folder</value>
  </property>
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.WRITE"/>
    </list>
  </property>
</bean>

<bean id="businessAccessDecisionManager"
class="net.sf.acegisecurity.vote.AffirmativeBased">
  <property name="allowIfAllAbstainDecisions"><value>false</value></property>
  <property name="decisionVoters">
    <list>
      <ref local="roleVoter"/>
      <ref local="aclFolderWriteVoter"/>
    </list>
```

```
    </property>
  </bean>

  <bean id="folderManagerSecurity"

  class="net.sf.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor"
  >
    <property name="authenticationManager">
      <ref bean="authenticationManager"/>
    </property>
    <property name="accessDecisionManager">
      <ref local="businessAccessDecisionManager"/>
    </property>
    <property name="objectDefinitionSource">
      <value>
        com.acegitech.FolderManager.createFolderWithParent=ACL_FOLDER_WRITE
        com.acegitech.FolderManager.writeFolder=ACL_FOLDER_WRITE
        com.acegitech.FolderManager.readFolder=ROLE_USER
      </value>
    </property>
  </bean>
```

The preceding settings would cause Acegi Security to intercept every method invocation for the listed methods of `FolderManager`. For the two methods marked `ACL_FOLDER_WRITE`, the `aclFolderWriteVoter` would be asked to vote on access. The `aclFolderWriterVoter` would then query the `AclManager` (which is not shown in the preceding sample) for the ACL entries applying to the current principal. If that returned list contains a "read" or "administration" permission, the method invocation would be allowed to proceed.

Acegi Security also publishes `AuthenticationEvent` and `SecurityInterceptionEvent` objects for authentication and authorization operations respectively to the Spring `ApplicationContext`. This provides a convenient hook for loosely coupled auditing, logging, and monitoring systems.

Let's now consider the alternative options available for securing Spring-based enterprise applications.

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) was introduced as an optional package to the Java 2 SDK version 1.3, being integrated into the SDK from version 1.4. JAAS provides pluggable authentication and authorization services for the Java platform, integrating into the platform layer security via system security policies. Many web and EJB containers provide some form of JAAS support, although, as of early 2005, there are significant compatibility issues between the various containers.

JAAS uses a `Subject` to represent an actual person or other entity. When authenticated, a range of identities is added to the `Subject` in the form of `Principal` objects. A JAAS `Principal` is used to represent an identity of the Subject, such as its name. `Subjects` can also contain public and private credentials, such as X.509 certificates.

Authentication is handled by JAAS through a `LoginModule`, a list of which is typically contained in an external policy file. A `LoginModule` is responsible for authenticating the principal and populating the `Subject`. A `LoginModule` requires a `CallbackHandler` to collect credentials from the principal.

Acegi Security provides an authentication provider that can delegate to JAAS, allowing integration between the two systems for authentication.

Once authenticated, JAAS's integration with the platform security, via a `SecurityManager`, allows authorization to take place. Authorization is typically defined in an external policy file. The operation of JAAS authorization is involved and beyond the scope of this book, although interested readers can refer to the Java 2 SDK documentation for a detailed background.

The following table briefly discusses some of the reasons commonly provided for using JAAS.

JAAS Benefit	Why It Matters	Mitigating Factors and Alternatives
Range of `LoginModules`	Saves you from having to write a login module for a given authentication repository.	— Most authentication repositories are relatively easy to integrate with in the first place, which means that it is easy to write a new authentication provider for a framework such as Acegi Security. — Acegi Security can use JAAS for authentication, effectively making all its `LoginModules` available to Acegi Security–managed applications.
JVM-level authorization	In a limited privilege execution environment, the JVM can ensure the code itself as well as that the principal is not requesting an unauthorized operation.	— Acegi Security, on the other hand, requires each secure object to be intercepted, typically via a Filter or an AOP "around" advice. — As Acegi Security provides AspectJ support, AOP "around" advices can even be applied to POJOs without any special step such as creating a proxy through an application context. — Rarely is the code of enterprise applications required to execute in a limited privilege environment. — Significant configuration is required for JAAS authorization, often involving JAR deployments in the SDK lib directory and editing of external policy files. — Acegi Security's use of the application context means Spring users find it a natural and convenient approach to configuration, particularly given its similarity (both in terms of security configuration and AOP interception) with Spring transaction management.

Table continued on following page

JAAS Benefit	Why It Matters	Mitigating Factors and Alternatives
		— Many web and application containers are incompatible with JVM-level authorization and cannot, for example, support two web applications using different security policies. — Using JVM-level authorization will limit web and application server portability, whereas using Acegi Security in its recommended configuration delivers complete compatibility and portability. — Even if JVM-level authorization was necessary for an unusual enterprise application, it would only be as secure as the security policy, and as that security policy will likely be provided by the same people who wrote the enterprise application itself, it becomes debatable just how much more security is realistically being derived from JVM-level authorization. Consider using AspectJ with Acegi Security instead, to gain portability and configuration benefits.
Automatic propagation of security identity from a web container to an EJB container, or from one EJB container to a remote EJB container	If you're using EJBs, this means less work for developers.	— Most new Spring applications don't use EJBs. — As most EJB developers are acutely aware, the EJB specification is limited in terms of authorization expressiveness and this typically requires an application to perform authorization itself anyway. — Through container adapters, Acegi Security can provide authentication for EJB containers, and in doing so provide a `Principal` simultaneously acceptable to both the EJB security subsystem as well as the Acegi Security framework. — Acegi Security performs transparent remote propagation of security identity when using Spring's RMI or `HttpInvoker`, effectively delivering an equivalent security propagation benefit as JAAS for remote POJOs.

JAAS Benefit	Why It Matters	Mitigating Factors and Alternatives
JAAS is an official standard	In some cases this is important for funding or buy-in.	— Acegi Security is the de facto and official standard for securing Spring-based applications, and as such has a large and active community with peer design review, implementation patterns, comprehensive documentation, extensions, and support readily available. — JAAS is only occasionally used in enterprise applications for authentication (usually the web or EJB container is relied upon for authentication) and it is rarely used for authorization because of the compatibility constraints. — Acegi Security can use JAAS for authentication if desired, effectively equalling the level of JAAS support available in most practical cases to enterprise application developers who consider the standard issue an important one.

Certainly JAAS is an important standard when developing applications intended for a limited privilege execution environment, such as an applet. However, unless web and EJB container support for JAAS greatly improves, it is unlikely to prove practical for efficiently developing portable, secure enterprise applications.

Servlet Specification

The Servlet Specification provides a simple way of authenticating principals and securing web URIs.

Each web container is responsible for performing authentication. Typically each web container will provide its own interface that authentication providers must implement to perform authentication. After configuring the web container to use a particular implementation of that interface, `web.xml` is used to finalize the authentication configuration:

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Servlet Specification Secured Realm</realm-name>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/login.jsp?login_error=1</form-error-page>
  </form-login-config>
</login-config>
```

Because each web container has its own special authentication provider interface, this introduces web container portability constraints. In addition, the web container will often require those authentication providers to be placed in a common library directory (along with support JARs such as JDBC drivers) and this further complicates configuration and classloader operation.

As far as authorization goes, the Servlet Specification provides just one of the three types of authorization areas typically required by enterprise applications: web request security. Again, `web.xml` is used to specify the roles that are required for given URI patterns:

```
<security-constraint>
  <display-name>Secured Area Security Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Secured Area</web-resource-name>
    <url-pattern>/secure/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_USER</role-name>
    <role-name>ROLE_SUPERVISOR</role-name>
  </auth-constraint>
</security-constraint>
```

The major limitation with this approach to web request authorization is that authorization decisions cannot take into account anything but roles. For example, they cannot consider in the authorization logic if a request has a particular query parameter. In addition, the specification provides only one way to express the URI patterns to be secured. (Acegi Security interprets two standard formats, with additional formats easily added.)

As mentioned already, the absence of support for services layer security or domain object instance security translates into serious limitations for multi-tiered applications. Typically developers find themselves either ignoring these authorization requirements, or implementing the security logic within their MVC controller code (or even worse, inside their views). There are disadvantages with this approach:

- **Separation of concerns:** Authorization is a crosscutting concern and should be implemented as such. MVC controllers or views implementing authorization code make it more difficult to test both the controller and authorization logic, more difficult to debug, and will often lead to code duplication.
- **Support for rich clients and web services:** If an additional client type must ultimately be supported, any authorization code embedded within the web layer is non-reusable. It should be considered that Spring remoting exporters export only service layer beans (not MVC controllers). As such, authorization logic needs to be located in the services layer to support a multitude of client types.
- **Layering issues:** An MVC controller or view is simply the incorrect architectural layer to implement authorization decisions concerning services layer methods or domain object instances. Though the `Principal` may be passed to the services layer to enable it to make the authorization decision, doing so would introduce an additional argument on every services layer method. A more elegant approach is to use a `ThreadLocal` to hold the `Principal`, although this would likely increase development time to a point where it would become more economical (on a cost-benefit basis) to simply use a dedicated security framework.

- **Authorization code quality:** It is often said of web frameworks that they "make it easier to do the right things, and harder to do the wrong things." Security frameworks are the same, because they are designed in an abstract manner for a wide range of purposes. Writing your own authorization code from scratch does not provide the "design check" a framework would offer, and in-house authorization code will typically lack the improvements that emerge from widespread deployment, peer review, and new versions.

For simple applications, Servlet Specification security may be sufficient. Although when considered within the context of web container portability, configuration requirements, limited web request security flexibility, and non-existent services layer and domain object instance security, it becomes clear why developers often look to alternative solutions.

Acegi Security Fundamentals

Let's now get to grips with the Acegi Security solution.

Authentication

The major interfaces and implementations of Acegi Security's authentication services are provided in Figure 10-1. As shown, the central interface is `Authentication`, which contains the identity of the principal, its credentials, and the `GrantedAuthority`s the principal has obtained. `GrantedAuthority` is an interface and implementations can have any meaning appropriate to the application. The `GrantedAuthorityImpl` is typically used, as it stores a `String` representation of an authority the principal has been granted.

As shown in the figure, the `AuthenticationManager` is the central interface responsible for processing an `Authentication` "request" object and determining if the presented principal and credentials are valid. If they are valid, the `AuthenticationManager` populates the `Authentication` with the `GrantedAuthority`s that apply. The main implementation of `AuthenticationManager` is known as `ProviderManager`, which is responsible for polling a list of `AuthenticationProvider`s.

There are many `AuthenticationProvider`s, each devoted to processing a particular `Authentication` concrete implementation. The implementation supported by different providers is indicated in Figure 10-1. The `ProviderManager` uses the first `AuthenticationProvider` capable of processing a given `Authentication` request object. It should be noted that because of space constraints, not every `AuthenticationProvider` included with Acegi Security is shown in the figure.

Most of the time the `DaoAuthenticationProvider` is used to process an authentication request. Figure 10-2 provides a class diagram for this important provider. As shown, `DaoAuthenticationProvider` will retrieve a `UserDetails` from an `AuthenticationDao`. Most developers write their own `AuthenticationDao` (to use Hibernate or their preferred persistence strategy), although Acegi Security also ships with a production-quality `JdbcDaoImpl`. `DaoAuthenticationProvider` also provides other useful features such as caching layer integration and decoding of encoded passwords (for example, SHA or MD5 with salts).

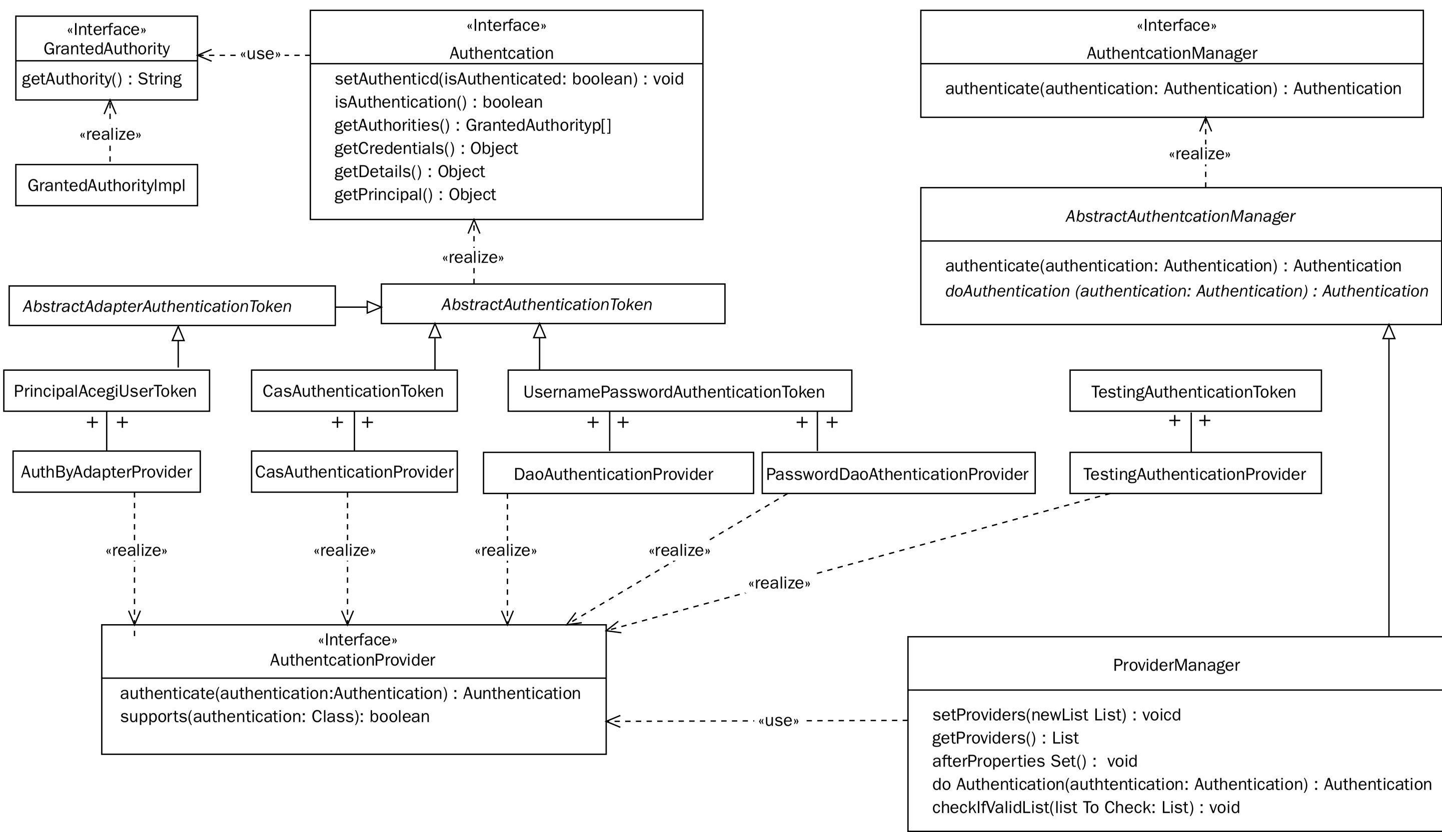

Figure 10-1

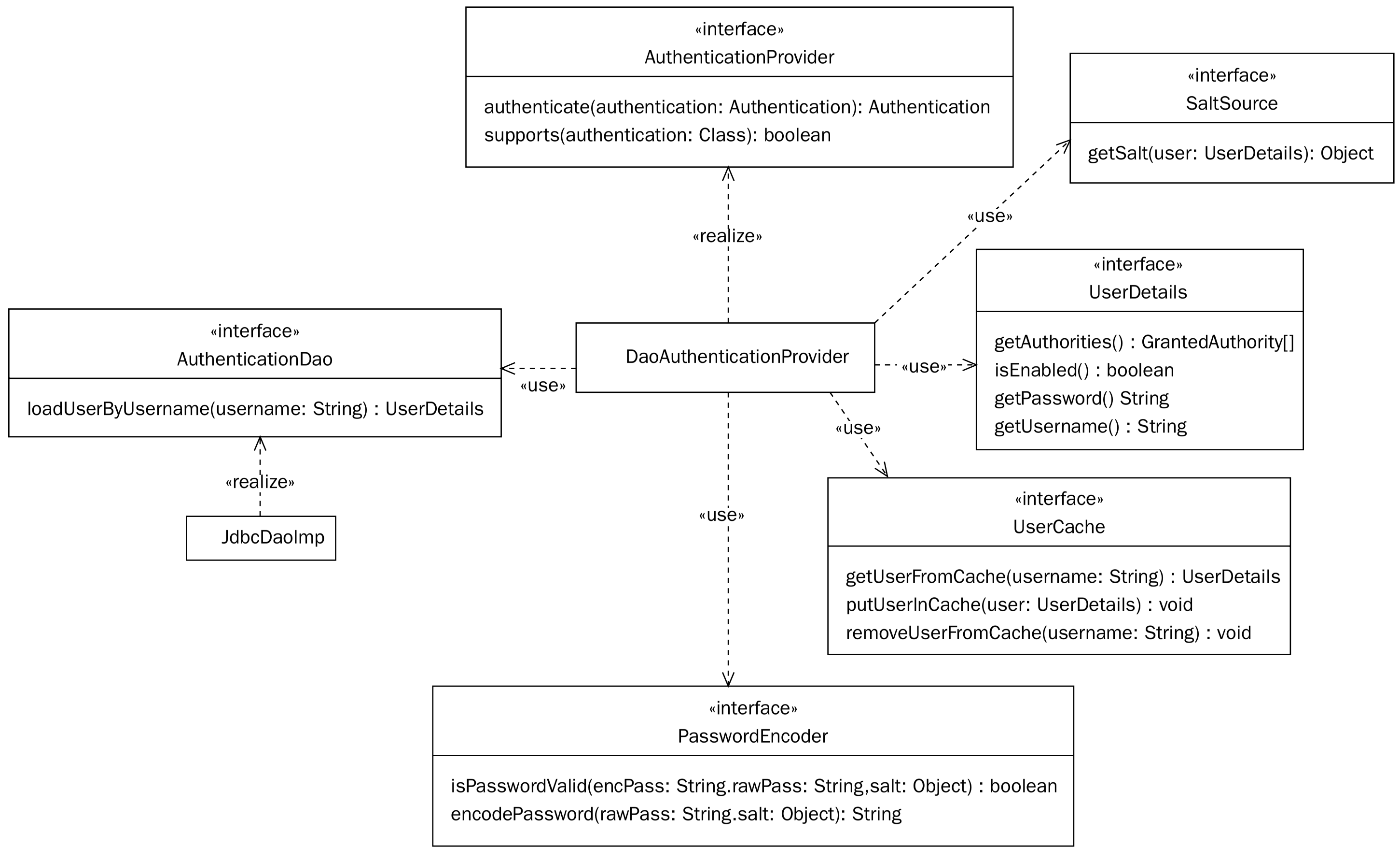

Figure 10-2

Although the preceding text explains how an authentication request is processed into a populated Authentication object, we have not covered how the user interacts with Acegi Security to actually request authentication. This is the role of various *authentication mechanisms*, as shown in the following table.

Module	What It Does	Why and When to Use It
`BasicProcessingFilter`	Processes BASIC authentication requests in accordance with RFC 1945.	— To publish secured web services — If a simple single sign on approach is desired — Should a simple authentication dialog be desired in the web browser rather than a login form
`CasProcessingFilter`	Processes Yale University's Central Authentication Service (CAS) tickets	— If your organization already has a CAS server — If you are looking for a comprehensive open source single sign-on solution
`AuthenticationProcessingFilter`	Processes an HTTP FORM POST similar to the Servlet Specification's `j_security_check`	— When single sign-on is not a requirement — Should your web application prefer a form-based login

Each authentication mechanism has two classes. The `[Strategy]ProcessingFilter` is used to process an authentication request as described in the previous table. The `[Strategy]ProcessingFilterEntryPoint` implements `AuthenticationEntryPoint` and is used to cause a web browser to commence the specified authentication mechanism, generally by sending an HTTP redirect message or some browser status code. Each web application also defines a `SecurityEnforcementFilter`, which is used to catch various Acegi Security exceptions and delegate to the `AuthenticationEntryPoint` or return a 403 (access denied) as appropriate.

Acegi Security supports multiple authentication mechanisms at the same time. For example, it is possible to handle authentication requests received both from a web form (with `AuthenticationProcessingFilter`) and from a Web Services client (with `BasicProcessingFilter`).

Client-server–rich clients are also fully supported. In the client-side application context a `RemoteAuthenticationManager` is configured. This `RemoteAuthenticationManager` receives `Authentication` request objects and passes the contained username and password to a corresponding server-side web service. The server-side web service then builds a new Authentication request object containing the passed username and password, before passing it to a server-side `AuthenticationManager`. If successful, a list of `GrantedAuthority`s is passed back to the `RemoteAuthenticationManager`. This allows the rich client to make authorization decisions such as the visibility of GUI actions. The rich client will also typically set each remoting proxy factory with the validated username and password.

The authors of Acegi Security recognize that many existing web applications would ideally be modified to leverage Acegi Security's authentication, ACL, and services layer authorization services, yet these apps have an existing investment in taglibs and other web-layer code that depends on the Servlet

Specification's `isUserInRole()`, `getUserPrincipal()`, and `getRemoteUser()` methods. Acegi Security provides two solutions:

- `HttpServletRequestWrapper`: Recall that Acegi Security's `ContextHolder` stores an `Authentication`, which is an extension of the `Principal` interface. As such, `ContextHolderAwareRequestWrapper` implements the Servlet Specification methods by delegation to these Acegi Security objects instead of the web server authentication system. The replacement wrapper is enabled simply by adding the `ContextHolderAwareRequestFilter` to `web.xml`.
- **Container adapters:** Although use is discouraged in all but very special circumstances, Acegi Security also provides several implementations of proprietary web or application server authentication interfaces. These allow the server to delegate an authentication decision to an Acegi Security `AuthenticationManager`, which returns an Authentication object that is in turn used by both the server and Acegi Security authorization classes. This may be useful if refactoring a large application that uses EJBs to a pure Spring implementation, as the existing EJBs (secured by the application server) can be used in parallel with new Spring POJOs (secured by Acegi Security).

Although `ContextHolderAwareRequestWrapper` is commonly used, container adapters are of very limited usefulness in web servers given that Acegi Security provides a far more extensive and flexible implementation of security functionality than offered by the Servlet Specification. One container adapter for which use is encouraged is `CasPasswordHandler`, which allows an `AuthenticationManager` to be used by an enterprise's CAS server. At the time of this writing CAS does not provide any usable implementations of its `PasswordHandler` interface, so this container adapter makes deployment of CAS much easier.

> **You do not need to "throw away" your existing investment in code and helper classes to move to Acegi Security. Integration hooks have been carefully considered so that you can migrate over time, leveraging the benefit of Acegi Security's additional features and container portability without major refactoring.**

Storing the Authentication Object

All major Acegi Security classes, and many user-specific classes, need to access the currently logged on user. More specifically, they need access to the populated Authentication object that applies to the current principal. Acegi Security ensures any interested class can access the Authentication via the `ContextHolder`, which is a `ThreadLocal` object usually used to hold a `SecureContext`. The `SecureContext` provides a getter and setter for the `Authentication` object, as shown in Figure 10-3.

A `ContextHolder` must be set with the correct `Authentication` for the duration of a given principal's request. For a unit test, that duration is typically a single test case. For a rich client, that duration is typically an entire application execution period. For a web application or web service, that duration is typically a single HTTP request. The `ContextHolder` needs to be made null at the end of a given principal's request; otherwise other principals might reuse the Thread, which would compromise security.

For JUnit tests, the `setUp()` *and* `tearDown()` *methods are often used to configure the* `ContextHolder` *with an appropriate* `Authentication` *object. This is of course assuming security actually needs to be tested. Normally, the POJOs making up Spring applications should be unit tested in isolation without transaction, security, and other AOP concerns.*

In the case of rich clients, the developer is responsible for setting up the `ContextHolder` for the request duration. With Spring Rich, classes supporting the use of Acegi Security are included in the

`org.springframework.richclient.security` package. This package contains a `LoginCommand` and `LogoutCommand`, which hide `ContextHolder` management from developers.

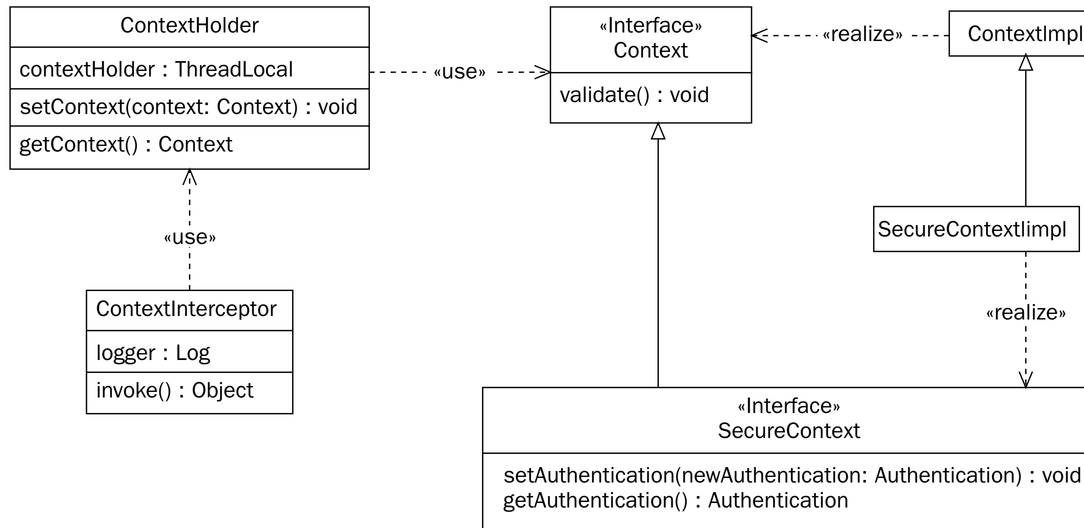

Figure 10-3

For web applications and web services, a Filter is used to address the `ContextHolder` requirement. A series of *authentication integration filters* is provided with Acegi Security, as detailed in the following table.

Module	What It Does	Why and When to Use It
`HttpSessionIntegrationFilter`	Uses the `HttpSession` to store the `Authentication` between requests	— Because container adapters are not recommended, this authentication integration filter is used in almost every situation. — `HttpSession` is used to store the Authentication between requests. — It provides a portable and flexible approach that does not depend on container adapters.
`HttpRequestIntegrationFilter`	Obtains the Authentication from `HttpServletRequest.getUserPrincipal()`, but cannot write the Authentication back to this location at the end of a request	— Mandatory if using most container adapters, as this is the only location the Authentication produced by most container adapters is available from.

		— The container is responsible for internally storing the Authentication between requests.
`JbossIntegrationFilter`	Obtains the Authentication from Jboss's `java:comp/env/security/subject` JNDI location, but cannot write the Authentication back to this location at the end of a request	— Mandatory if using the JBoss container adapter. — The container is responsible for internally storing the Authentication between requests. — Do not use the JBoss container adapter if one of the standard web application authentication mechanisms (see the previous table) would suffice.

Authorization

Armed with a solid understanding of how a principal is authenticated and the resulting Authentication is stored between requests and made available through the `ContextHolder`, we can now examine the fundamental objective of security: authorization.

Figure 10-4 provides an overview of the key authorization interfaces and classes. An important interface is `ConfigAttribute`, which represents a configuration setting that applies to a secure object invocation (recall a secure object invocation is any type of object that can have its invocation intercepted and security logic applied). Configuration attributes are similar to Spring's transaction attributes, such as `PROPAGATION_REQUIRED`.

The `AccessDecisionManager` is responsible for making an authorization decision. It is passed the secure object, in case it needs access to properties of the secure object invocation (such as arguments in the case of a `MethodInvocation` or `HttpServletRequest` properties in the case of a `FilterInvocation`). The `AccessDecisionManager` is also passed the configuration attributes that apply to the secure object invocation, along with the validated Authentication object. This information is sufficient to make an authorization decision by returning void or throwing an `AccessDeniedException`.

Similar to the authentication-related `ProviderManager`, the `AbstractAccessDecisionManager` adopts a provider-based approach whereby it will poll a series of `AccessDecisionVoter`s. Each voter returns a veto, grant, or deny vote. The final decision based on these votes is left to the specific `AbstractAccessDecisionManager` implementation.

The `RoleVoter` is the most commonly used `AccessDecisionVoter` with Acegi Security. It iterates through each configuration attribute, voting to grant access if one matches a `GrantedAuthority` held by the principal. As such it's a simple way of implementing role-based access control. `BasicAclVoter` is a more complex voter and is discussed in the following "Domain Object Instance Security" section.

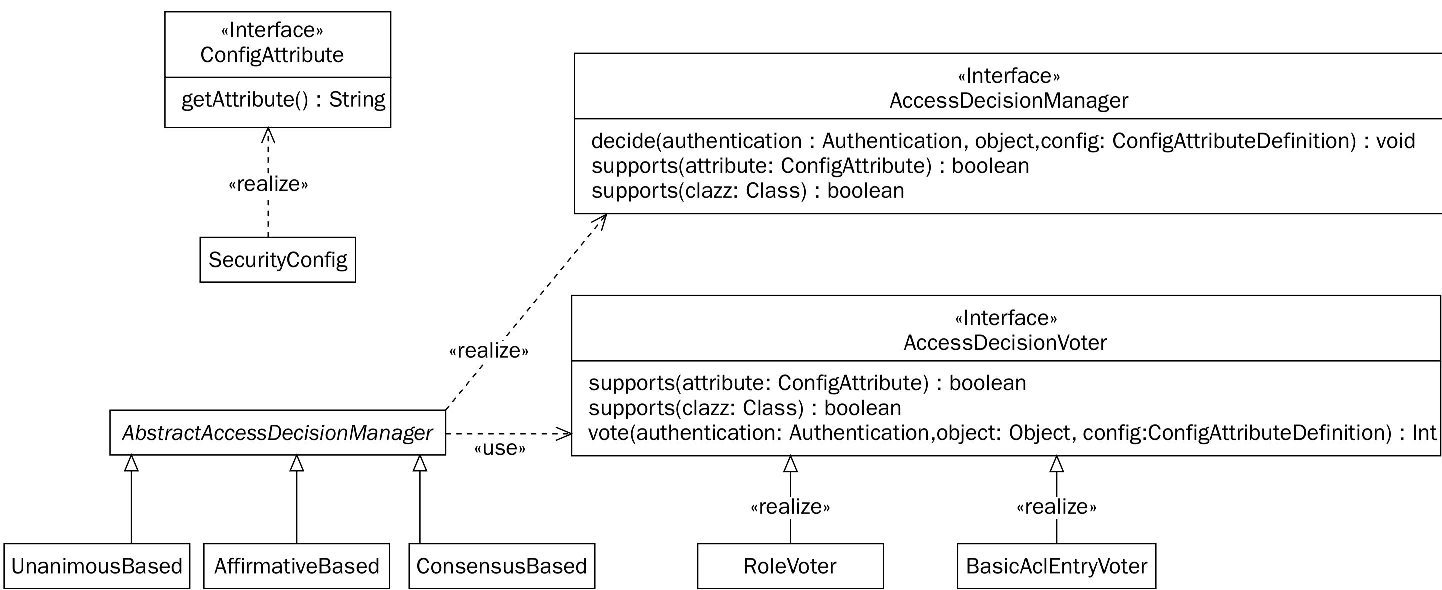

Figure 10-4

In order to actually authorize a secure object invocation, Acegi Security needs to intercept the invocation to begin with. Figure 10-5 shows the key interfaces and classes involved in security interception. As shown, the `AbstractSecurityInterceptor` has a series of subclasses that each handle a specific type of secure object.

The `FilterSecurityInterceptor` is implemented by way of a `<filter>` declaration in `web.xml`. Acegi Security provides a useful `FilterToBeanProxy` class, which allows a `web.xml`-defined Filter to be set up in the Spring bean container. This allows collaborating beans to be easily injected. `MethodSecurityInterceptor` is used to protect beans defined in the bean container, requiring the use of Spring's `ProxyFactoryBean` or `DefaultAdvisorAutoProxyCreator` along with the `MethodDefinitionSourceAdvisor`. The `AspectJSecurityInterceptor` is woven in using AspectJ's compiler.

Upon detection of a secure object invocation, an `AbstractSecurityInterceptor` will look up the configuration attributes that apply to that invocation. If there aren't any configuration attributes, the invocation is deemed to be public and the invocation proceeds. If configuration attributes are found, the `Authentication` contained in the `ContextHolder` is authenticated via the `AuthenticationManager`, and the `AccessDecisionManager` is asked to authorize the request. If successful, the `RunAsManager` may replace the identity of the `Authentication` and then the invocation will proceed. Upon completing the invocation, the `AbstractSecurityInterceptor` will clean up by updating the `ContextHolder` to contain the actual `Authentication` object, and if one is defined, calling the `AfterInvocationManager`.

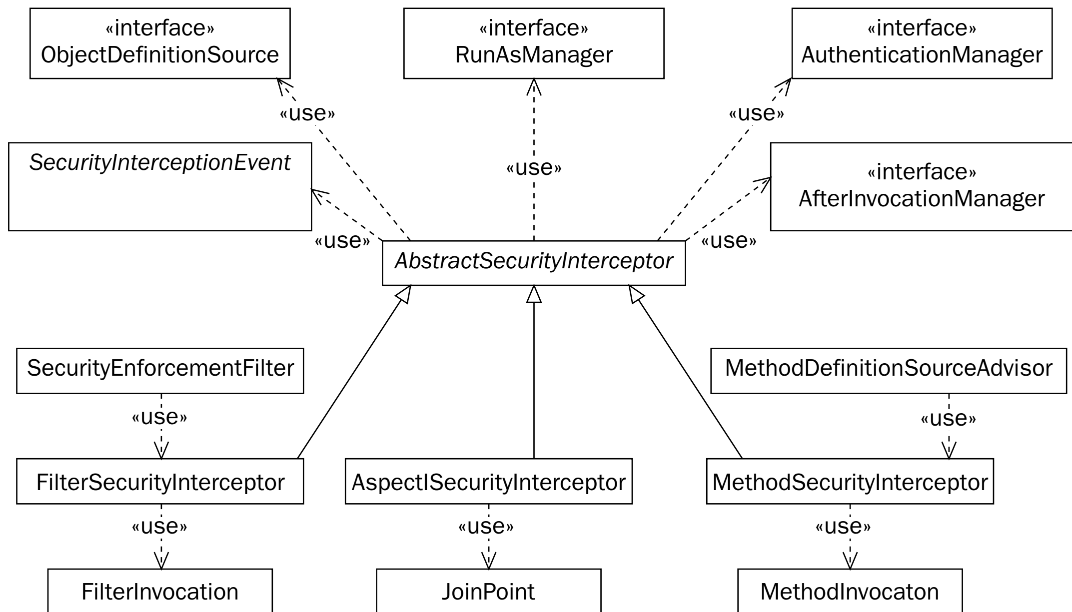

Figure 10-5

The `AfterInvocationManager` is able to modify the object to be returned from the secure object invocation. This is typically used to filter `Collections` only to contained, authorized elements, or mutate instance variables if the information is protected. It can also throw an `AccessDeniedException` if, for example, the principal does not have permission to the object about to be returned.

`AfterInvocationManager` is similar to `AccessDecisionManager`. It is passed the `Authentication`, secure object invocation, applicable configuration attributes, and of course the object to be returned. It is required to return an object, which will then become the result of the secure object invocation. A concrete implementation named `AfterInvocationProviderManager` is used to poll `AfterInvocationProvider` instances. The first instance that indicates it can process the after invocation request will be used.

Domain Object Instance Security

Protecting individual domain object instances is a common requirement of large enterprise applications. Often it is necessary to prevent unauthorized principals from accessing certain domain object instances, or preventing services layer methods from being invoked when principals are not authorized for a particular domain object instance. Other times it is desirable to mutate sensitive properties of a domain object instance, depending on the principal retrieving the instance. Acegi Security uses the secure object model discussed previously to enforce this form of ACL security within your applications.

Figure 10-6 illustrates Acegi Security's ACL services. As shown, the `AclManager` interface allows any `Object` to be presented and its ACLs obtained. An additional method is provided that allows those ACLs to be filtered to contain only ACLs that apply to the presented Authentication. The `AclProviderManager` polls `AclProviders` until one indicates it is capable of retrieving the `AclEntrys` applying to a given `Object`.

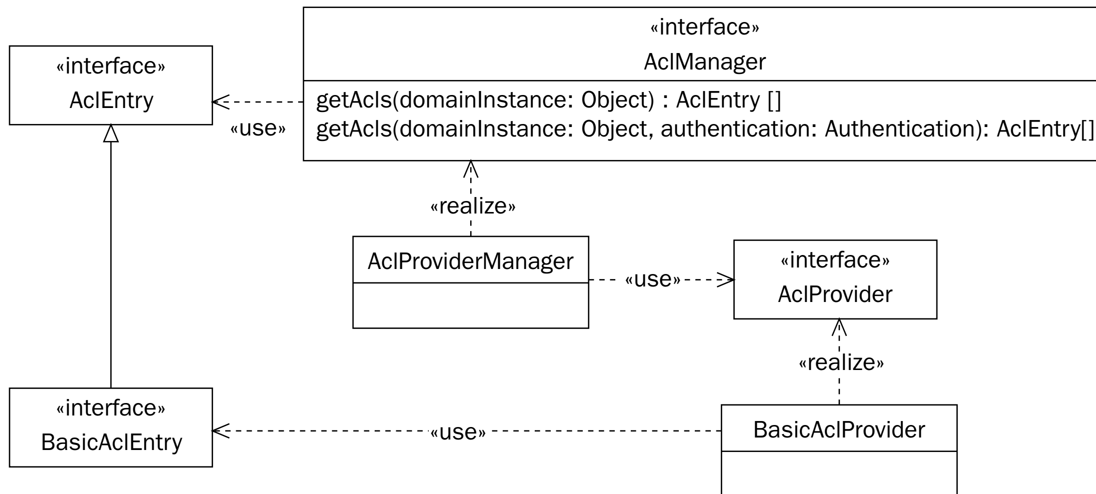

Figure 10-6

Acegi Security provides a `BasicAclProvider`, which uses integer bit masking and a DAO-based approach to retrieving ACL information. The implementation basically converts the domain `Object` into an `AclObjectIdentity`, which is able to uniquely identify the `Object`. This is then used as a key against a `BasicAclDao` that can retrieve the `BasicAclEntrys` for the `Object`. `BasicAclProvider` (shown in Figure 10-7) uses a pluggable caching layer to avoid expensive ACL retrievals. Other interfaces are also used by the `BasicAclProvider` to implement the remainder of the `AclProvider` contract.

Working in concert with the `AclManager` are several key classes. These provide the linkage between the `AbstractSecurityInterceptor` and actual enforcement of ACL security. These are listed in the following table.

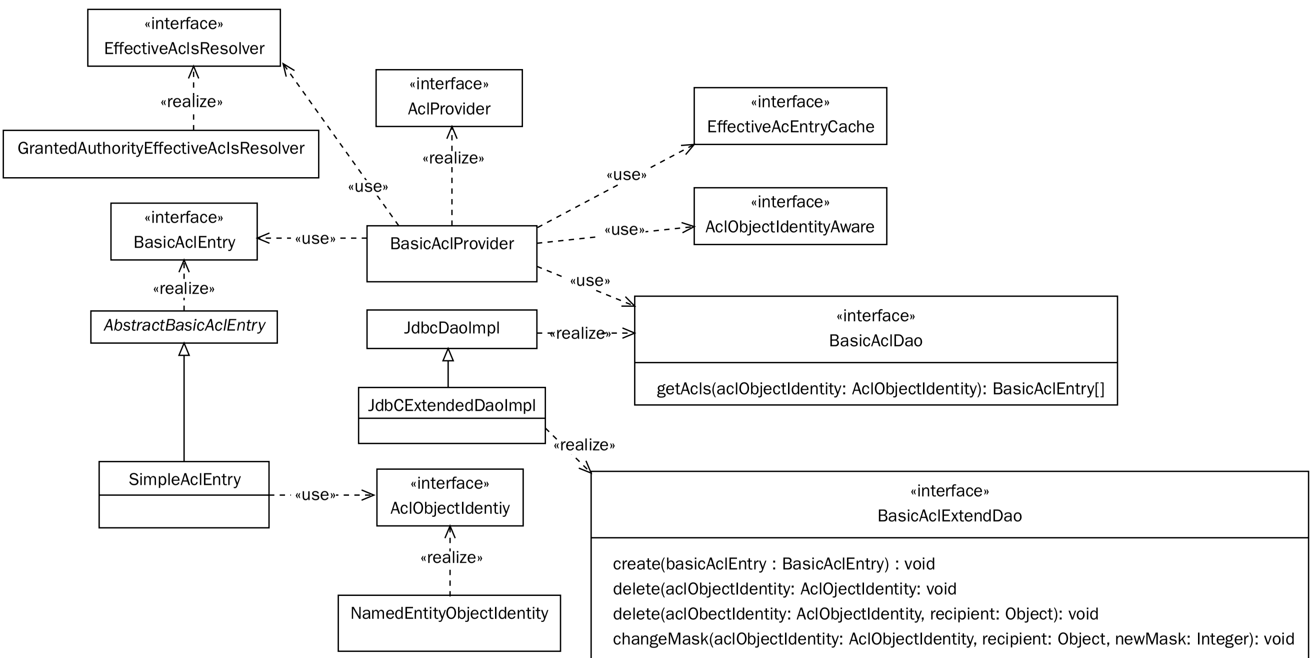

Figure 10-7

Module	What It Does	Why and When to Use It
`BasicAclEntryVoter`	Called by an `AbstractAccessDecisionManager`, looks at method arguments of the secure object invocation and locates one that matches, and then retrieves the ACL details for that `Object` and principal, and votes to grant or deny access based on a list of allowed permissions defined against the `BasicAclEntryVoter`.	— Typically several `BasicAclEntryVoter`s are used together. — Different instances respond to different configuration attributes, with each instance typically targeting a different domain object class and/or list of allowed permissions. — Used when you need to deny access *before* a secure object invocation is allowed to proceed.
`BasicAclAfterInvocationProvider`	Called by the `AfterInvocationProviderManager`, looks at the returned `Object`, and then retrieves the ACL details for that `Object` and principal, throwing an `AccessDeniedException` if the principal does not have access according to the list of allowed permissions defined against the `BasicAclAfterInvocationProvider`.	— If you cannot determine *before* a method is invoked whether the principal should be allowed access, such as `getById(int)` methods. — Whenever you do not mind an `AccessDeniedException` being thrown *after* a secure object invocation is allowed to proceed.
`BasicAclEntryAfterInvocation-CollectionFilteringProvider`	Similar to `BasicAclAfterInvocationProvider`, but expects the returned `Object` to be a `Collection` and iterates that `Collection`, locating and removing unauthorized elements (rather than throwing `AccessDeniedException`) based on the list of allowed permissions defined against the `BasicAclEntryAfterInvocation-CollectionFilteringProvider`.	— If you need to filter a `Collection` to remove unauthorized elements. — When the method returns a `Collection` (no other return object types are allowed).

Example Code

We've provided a broad overview of Acegi Security and some of its key capabilities. Let's now look at a sample application that demonstrates the ACL services in detail.

Introducing the Sample

For this sample we are going to assume the role of a developer delivering a domain name system (DNS) administration tool for a major Internet hosting provider. The central function of the new application is to provide the hosting provider's customers with a self-service facility to manage the domain names they have registered. The main customer base comprises Internet entrepreneurs who own multiple domain names, so it is important the new DNS administration tool be able to accommodate each customer having many domain names. It is also a requirement that customers be able to delegate DNS administration responsibility to other people, such as their web designers or branch office IT managers.

It is assumed you have a basic understanding of DNS, in that its main function is to resolve hostnames such as `www.springhost.zz` into TCP/IP addresses. Each domain name is hierarchical, in that each subdomain falls under a parent domain until eventually reaching a top-level domain (such as an .org or .com or .au). For each domain name, an authoritative DNS server is specified. The role of the authoritative DNS server is to respond to requests for records within the domain.

Returning to our sample, there are a variety of DNS server implementations that could be used to implement the DNS administration tool. In order to promote maximum flexibility and reduce development time, your team has decided to store the DNS data in an RDBMS. With a little searching you discover an open source project called myDns (`mydns.bboy.net`), which interfaces with Postgres and mySql.

Security Unaware Implementation

After reviewing the myDns schema, for this book we developed a simple core for the new DNS administration tool. The three domain objects and service layer are shown in Figure 10-8. The `StartOfAuthority` and `ResourceRecord` objects are both modeled after myDns's `soa` and `rr` tables respectively. The `Domain` object is an addition to allow the hierarchy of the domain to be reflected, as the myDns tables, being focused on a DNS server's requirements, do not provide this support. There is a one-to-one relationship between a given `Domain` and a `StartOfAuthority` instance. There is a many-to-one relationship between `ResourceRecord` and `StartOfAuthority`. `DomainDao` provides CRUD operations for the `Domain` (including its contained `StartOfAuthority`) and `ResourceRecord` objects. `BasicAclExtendedDao` is an Acegi Security class that provides a simple interface to perform CRUD operations for ACL entries.

Also included in the Java package is a `DataSourcePopulator` class that configures the schema and inserts sample data. The `applicationContext-business.xml` included with the sample configures the persistence layer and services layer, and adds transaction support.

An in-memory Hypersonic SQL database is also configured in the application context, as it facilitates rapid prototyping and testing. Of course, a production implementation would need to use a database supported by myDns.

The classes, interfaces, and application context just described could be used to provide a basic DNS administration tool in conjunction with myDns.

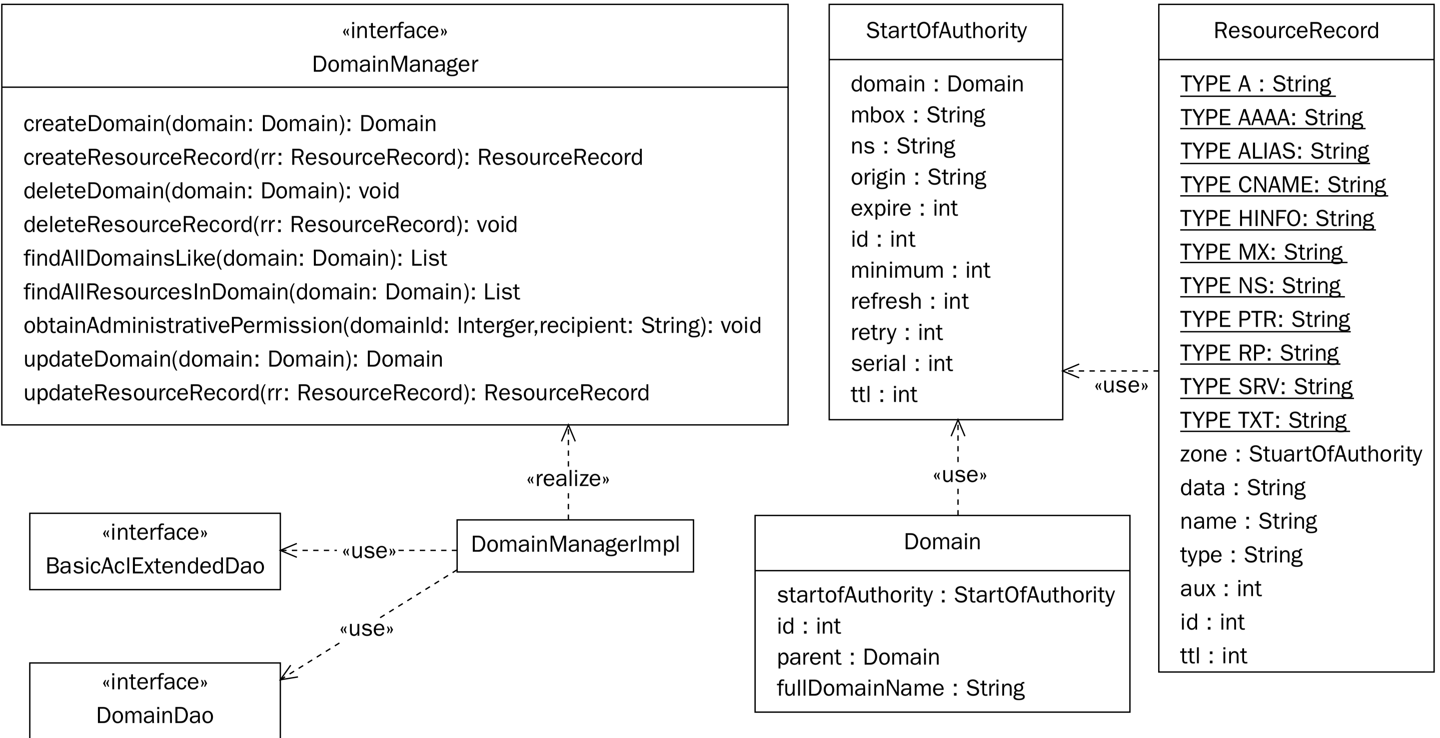

Figure 10-8

Security Approach

The interfaces and classes in Figure 10-8 are unaware of security, except as follows:

- When creating `Domains` and `ResourceRecord`s, the `DomainManager` implementation needs to create an ACL entry via the `BasicAclExtendedDao`.
- When deleting `Domains` and `ResourceRecord`s, the `DomainManager` implementation needs to delete all ACL entries via the `BasicAclExtendedDao`.
- To allow an administrator to gain full access to a domain (perhaps the owner of that domain revoked the administrator's existing access), `DomainManager` provides an `obtainAdministrativePermission` method that operates via the `BasicAclExtendedDao`.

Though the application does require a small amount of security awareness, this is only to link between the problem domain itself and the Acegi Security ACL services. The remainder of the security requirements of the application are declaratively configured in the Spring bean container. Configuration attributes will be used to tag how we require `ContactManager` methods to be secured. At an implementation level, we'll use `MethodSecurityInterceptor` as the application does not use AspectJ or provide web access that requires URI filtering.

Most of the application will be secured using ACL security. The only method on the `DomainManager` interface that is not secured via ACL security is the `obtainAdministrativePermission` method. For this special method we'll use role-based security, meaning the principal will need to hold a particular `GrantedAuthority`.

Over the following pages we'll be discussing the configuration of a security environment appropriate for integration testing using JUnit. The test cases have been developed to test and demonstrate security integration is functioning correctly for the sample application. Each of the unit tests extends from `AbstractDomainManagerTest`, which extends `AbstractTransactionSpringContextTests` because we leverage the latter's automatic database rollback capability.

Authentication

`AbstractDomainManagerTest` is responsible for providing several methods to assist in handling the `ContextHolder` during our JUnit tests. They allow our tests to simply call `makeActiveUser(String username)` for the `ContextHolder` to be configured with an appropriate `Authentication` request object.

```
protected void makeActiveUser(String username) {
String password = "";
if ("marissa".equals(username)) {
    password = "koala";
  } else if ("dianne".equals(username)) {
    password = "emu";
  } else if ("scott".equals(username)) {
    password = "wombat";
  } else if ("peter".equals(username)) {
    password = "opal";
  }
```

```
Authentication authRequest = new UsernamePasswordAuthenticationToken(
                                  username, password);
  SecureContextImpl secureContext = new SecureContextImpl();
  secureContext.setAuthentication(authRequest);
  ContextHolder.setContext(secureContext);
}

protected void onTearDownInTransaction() {
  ContextHolder.setContext(null);
}
```

Though Acegi Security provides a `TestingAuthenticationToken` and `TestingAuthenticationProvider` that allow you to specify any `GrantedAuthority`s, principal, and credentials you wish (and they will be automatically accepted), in the DNS administration tool tests we've decided to use the popular `DaoAuthenticationProvider`, which accesses user information using `JdbcDaoImpl`. Key authentication-related XML is listed here:

```
<bean id="authenticationManager"
  class="net.sf.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
    </list>
  </property>
</bean>
<bean id="jdbcDaoImpl" class="net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl">
  <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
<bean id="daoAuthenticationProvider"
  class="net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="authenticationDao"><ref local="jdbcDaoImpl"/></property>
</bean>
```

Authorization

None of the `DomainManager` methods allow anonymous access, meaning every one of them needs to have one or more configuration attributes defined against the `MethodSecurityInterceptor`. The configuration attributes we've used for the DNS administration tool are listed here:

```
DomainManager.createDomain=ACL_DOMAIN_CREATE
DomainManager.createResourceRecord=ACL_RESOURCE_RECORD_CREATE
DomainManager.deleteDomain=ACL_DOMAIN_DELETE
DomainManager.deleteResourceRecord=ACL_RESOURCE_RECORD_DELETE
DomainManager.findAllDomainsLike=ROLE_USER,AFTER_ACL_COLLECTION_READ
DomainManager.findAllResourceRecordsInDomain=ROLE_USER,AFTER_ACL_COLLECTION_READ
DomainManager.obtainAdministrativePermission=ROLE_SUPERVISOR
DomainManager.updateDomain=ACL_DOMAIN_WRITE
DomainManager.updateResourceRecord=ACL_RESOURCE_RECORD_WRITE
```

We suggest a convention when naming configuration attributes to help you understand when they will be used. All configuration attributes that will be processed by an `AfterInvocationManager` should have a name starting with `AFTER_`. No configuration attributes that will be processed by an `AccessDecisionManager` should start with `AFTER_`.

Based on this convention, every method has a configuration attribute indicating an AccessDecisionManager will process it. This is because we want to ensure every principal either holds the ROLE_USER GrantedAuthority or has a specific ACL for the domain object passed as a method argument. Let's consider several examples.

The deleteDomain method uses the ACL_DOMAIN_DELETE configuration attribute. The Spring bean container contains the following declaration:

```
<bean id="aclDomainDeleteVoter"
class="net.sf.acegisecurity.vote.BasicAclEntryVoter">
  <property
name="processConfigAttribute"><value>ACL_DOMAIN_DELETE</value></property>
  <property
name="processDomainObjectClass"><value>com.acegitech.dns.domain.Domain</value></pro
perty>
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.DELETE"/>
    </list>
  </property>
</bean>
```

The SimpleAclEntry.ADMINISTRATION and DELETE references actually relate to a FieldRetrieveingFactoryBean that enables the integer values of these static variables to be obtained from the specified class, rather than using integers directly inside the XML.

The aclDomainDeleteVoter will vote only when a secure object invocation presents the ACL_DOMAIN_DELETE attribute. Based on our earlier list configuration attributes, only DomainManager .deleteDomain(Domain) uses this configuration attribute. If voting, the voter will look at the method arguments for the first instance of the processDomainObjectClass, which in this case is the Domain object. The Domain will then be presented to the AclManager defined by the property of the same name, and the BasicAclEntrys that apply to the current principal will be obtained. Finally, aclDomainDeleteVoter will determine if the principal has any of the ACL permissions listed in the requirePermission property. If the principal does not, it will vote to deny access, or vote to grant access otherwise.

An important feature of the BasicAclProvider is its support for hierarchical domain object instances. Consider the following unit test:

```
public void testDeleteAsPeterWhenPermittedViaDeepInheritance() {
  makeActiveUser("peter");
  Domain domain = getDomain("compsci.science.zoo-uni.edu.zz.");
  domainManager.deleteDomain(domain);

  // Ensure it was deleted (transaction will rollback post-test)
  assertNull(getDomain("compsci.science.zoo-uni.edu.zz."));
}
```

In the preceding example, user peter has been granted delete permission for domain science.zoo-uni.edu.zz. The fact peter is authorized to delete this subdomain is only because BasicAclProvider delivered the parent object's ACLs.

ACL_DOMAIN_CREATE is an interesting use of the BasicAclEntryVoter class because it demonstrates an additional feature. In this case we do not want to vote on the Domain presented in the method argument, but on whether the principal has create permission for that Domain's parent Domain. To accommodate this we use the internalMethod property, as shown here:

```
<bean id="aclDomainCreateVoter"
class="net.sf.acegisecurity.vote.BasicAclEntryVoter">
  <property
name="processConfigAttribute"><value>ACL_DOMAIN_CREATE</value></property>
  <property
name="processDomainObjectClass"><value>com.acegitech.dns.domain.Domain</value></pro
perty>
  <property name="internalMethod"><value>getParent</value></property>
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.CREATE"/>
    </list>
  </property>
</bean>
```

Let's consider role-based access control, as used by the DomainManager.obtainAdministrativePermission method. RoleVoter will process the relevant configuration attribute, ROLE_SUPERVISOR. It will simply compare every GrantedAuthority assigned to the principal with the ROLE_SUPERVISOR String. A single RoleVoter is used for all role voting, irrespective of the attribute name (for example, ROLE_USER, ROLE_FOO):

```
<bean id="roleVoter" class="net.sf.acegisecurity.vote.RoleVoter"/>
```

The preceding techniques are used to secure almost every method of DomainManager. The only exceptions are the two find* methods, which use the configuration attribute AFTER_ACL_COLLECTION_READ. The corresponding declaration for this AfterInvocationProvider is:

```
<bean id="afterAclCollectionRead"
class="net.sf.acegisecurity.afterinvocation.BasicAclEntryAfterInvocationCollectionF
ilteringProvider">
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="net.sf.acegisecurity.acl.basic.SimpleAclEntry.READ"/>
    </list>
  </property>
</bean>
```

The afterAclCollectionRead bean will present every element in the Collection to the AclManager, removing that element if the current principal does not hold one of the listed permissions (administration or read). It is used in this case to remove Domains and ResourceRecords the principal is not permitted to access. This is demonstrated by the following JUnit test code:

```
public void testFindAllDomainsLikeAsDianne() {
  makeActiveUser("dianne"); // has ROLE_STAFF
  List domains = domainManager.findAllDomainsLike("");
```

```
    assertNotContainsDomain("scotty.com.zz.", domains);
    assertEquals(8, domains.size()); // all domains except scotty.com.zz.
  }
```

As indicated by the comments, user dianne has the ROLE_STAFF role and is thus granted administrative access to all domains except scotty.com.zz. The test demonstrates that scotty.com.zz was removed from the Collection.

Each DomainManager method has a comprehensive suite of JUnit that tests operation using different domain object instances and principals. We recommend you have a look at those tests to gain a fuller understanding of how Acegi Security is being used in this sample application. Acegi Security also provides a sample application that can be used to query the domains. The sample application is a client-server tool that demonstrates the ease at which Acegi Security supports web services clients. A simple command that displays all resource records in the DNS system is shown here:

```
maven -Dusername=marissa -Dpassword=koala
      -Dcommand=findRecordsByDomainName -Ddomain=. run
run:
    [java] Acegi DNS Console Query Tool
    [java] Domain: 10   arts.zoo-uni.edu.zz.
    [java]   10 www     10.1.43.65
    [java] Domain: 11   compsci.science.zoo-uni.edu.zz.
    [java]   12 ssl     10.1.12.88
    [java]   11 www     10.1.12.87
    [java] Domain: 4    jackpot.com.zz.
    [java]   4  www     192.168.0.3
    [java] Domain: 2    petes.com.zz.
    [java]   2  www     192.168.0.1
    [java] Domain: 9    science.zoo-uni.edu.zz.
    [java]   9  www     10.1.12.4
    [java] Domain: 7    tafe.edu.zz.
    [java]   7  www     192.168.0.5
    [java] Domain: 6    zoo-uni.edu.zz.
    [java]   6  www     10.1.0.1
    [java] Domain: 8    zoohigh.edu.zz.
    [java]   8  www     192.168.0.6

maven -o -Dusername=scott -Dpassword=wombat -Dcommand=findAll run
run:
    [java] Acegi DNS Console Query Tool
    [java] Domain: 3    scotty.com.zz.
    [java] Domain: 7    tafe.edu.zz.
```

Finally, Acegi Security itself includes a Contacts sample application that uses domain object instance security and provides a full web layer and web services. This sample also demonstrates form-based, CAS, and container adapter–managed authentication.

Summary

In this chapter we've looked at security within enterprise applications, the most common type of application built with the Spring Framework.

We've seen that enterprise applications typically have four security-related requirements:

- Authentication
- Web request security
- Service layer security
- Domain object instance security

We briefly considered three common approaches to implementing these requirements:

- Acegi Security System for Spring
- JAAS
- Servlet Specification security

We recommended that Acegi Security be used in all but the simplest of cases, with a simple case being one that does not need any form of services layer or domain object instance authorization, or web container portability.

We examined Acegi Security's architecture, with an emphasis on authentication, authorization, and domain object instance security. We saw how Acegi Security is configured exclusively through the Spring application context, and is similar in many ways to Spring's own transaction services (discussed in Chapter 6). We discussed many of the most commonly used Acegi Security classes in detail.

We saw that Acegi Security supports a wide range of authentication providers, and can not only authorize a web or method invocation before it happens, but also throw an `AccessDeniedException` or modify an `Object` returned from a method invocation. We saw the interface-driven nature of Acegi Security, and we looked at how role-based and access control list (ACL) authorization could be achieved solely using extensible, declarative means.

Following a review of the architecture, we reviewed a DNS application that could be used in a multi-customer Internet hosting provider. The application used Acegi Security — in particular its ACL capabilities — to restrict different customers to their own domains only. The application also leveraged DNS's inherent hierarchical nature so that ACL permissions granted to higher level domains trickled down to subdomains and their contained resource records.

As we have shown in this chapter, with proper design and an effective security framework, enterprise applications rarely require their own custom security code. Using a high-quality, generic security framework such as Acegi Security can help to free developers to focus on truly domain-specific concerns. The value proposition is just as compelling as with other generic functionality addressed by Spring and other leading frameworks.

As with transactions, security demonstrates the true potential of AOP in delivering transparent crosscutting concerns, and the real-world productivity and quality benefits this delivers.