Acegi Security

Reference Documentation

1.0.7

Ben Alex

Table of Contents

Preface

Acegi Security provides a comprehensive security solution for J2EE-based enterprise software applications. As you will discover as you venture through this reference guide, we have tried to provide you a useful and highly configurable security system.

Security is an ever-moving target, and it's important to pursue a comprehensive, system-wide approach. In security circles we encourage you to adopt "layers of security", so that each layer tries to be as secure as possible in its own right, with successive layers providing additional security. The "tighter" the security of each layer, the more robust and safe your application will be. At the bottom level you'll need to deal with issues such as transport security and system identification, in order to mitigate man-in-the-middle attacks. Next you'll generally utilise firewalls, perhaps with VPNs or IP security to ensure only authorised systems can attempt to connect. In corporate environments you may deploy a DMZ to separate public-facing servers from backend database and application servers. Your operating system will also play a critical part, addressing issues such as running processes as non-privileged users and maximising file system security. An operating system will usually also be configured with its own firewall. Hopefully somewhere along the way you'll be trying to prevent denial of service and brute force attacks against the system. An intrusion detection system will also be especially useful for monitoring and responding to attacks, with such systems able to take protective action such as blocking offending TCP/IP addresses in real-time. Moving to the higher layers, your Java Virtual Machine will hopefully be configured to minimize the permissions granted to different Java types, and then your application will add its own problem domain-specific security configuration. Acegi Security makes this latter area - application security - much easier.

Of course, you will need to properly address all security layers mentioned above, together with managerial factors that encompass every layer. A non-exhaustive list of such managerial factors would include security bulletin monitoring, patching, personnel vetting, audits, change control, engineering management systems, data backup, disaster recovery, performance benchmarking, load monitoring, centralised logging, incident response procedures etc.

With Acegi Security being focused on helping you with the enterprise application security layer, you will find that there are as many different requirements as there are business problem domains. A banking application has different needs from an ecommerce application. An ecommerce application has different needs from a corporate sales force automation tool. These custom requirements make application security interesting, challenging and rewarding.

This reference guide has been largely restructured for the 1.0.0 release of Acegi Security. Please read Part I, Overall Architecture, in its entirety. The remaining parts of the reference guide are structured in a more traditional reference style, designed to be read on an as-required basis.

We hope that you find this reference guide useful, and we welcome your feedback and suggestions.

Finally, welcome to the Acegi Security community.

Part I. Overall Architecture

Like most software, Acegi Security has certain central interfaces, classes and conceptual abstractions that are commonly used throughout the framework. In this part of the reference guide we will introduce Acegi Security, before examining these central elements that are necessary to successfully planning and executing an Acegi Security integration.

Chapter 1. Introduction

1.1. What is Acegi Security?

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

People use Acegi Security for many reasons, but most are drawn to the project after finding the security features of J2EE's Servlet Specification or EJB Specification lack the depth required for typical enterprise application scenarios. Whilst mentioning these standards, it's important to recognise that they are not portable at a WAR or EAR level. Therefore, if you switch server environments, it is typically a lot of work to reconfigure your application's security in the new target environment. Using Acegi Security overcomes these problems, and also brings you dozens of other useful, entirely customisable security features.

As you probably know, security comprises two major operations. The first is known as "authentication", which is the process of establishing a principal is who they claim to be. A "principal" generally means a user, device or some other system which can perform an action in your application. "Authorization" refers to the process of deciding whether a principal is allowed to perform an action in your application. To arrive at the point where an authorization decision is needed, the identity of the principal has already been established by the authentication process. These concepts are common, and not at all specific to Acegi Security.

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

- HTTP BASIC authentication headers (an IEFT RFC-based standard)
- HTTP Digest authentication headers (an IEFT RFC-based standard)
- HTTP X.509 client certificate exchange (an IEFT RFC-based standard)
- LDAP (a very common approach to cross-platform authentication needs, especially in large environments)
- Form-based authentication (for simple user interface needs)
- Computer Associates Siteminder
- JA-SIG Central Authentication Service (otherwise known as CAS, which is a popular open source single sign on system)
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (a Spring remoting protocol)
- Automatic "remember-me" authentication (so you can tick a box to avoid re-authentication for a predetermined period of time)
- Anonymous authentication (allowing every call to automatically assume a particular security identity)
- Run-as authentication (which is useful if one call should proceed with a different security identity)
- Java Authentication and Authorization Service (JAAS)

- Container integration with JBoss, Jetty, Resin and Tomcat (so you can still use Container Manager Authentication if desired)
- Your own authentication systems (see below)

Many independent software vendors (ISVs) adopt Acegi Security because of this rich choice of authentication models. Doing so allows them to quickly integrate their solutions with whatever their end clients need, without undertaking a lot of engineering or requiring the client to change their environment. If none of the above authentication mechanisms suit your needs, Acegi Security is an open platform and it is quite simple to write your own authentication mechanism. Many corporate users of Acegi Security need to integrate with "legacy" systems that don't follow any particular security standards, and Acegi Security is happy to "play nicely" with such systems.

Sometimes the mere process of authentication isn't enough. Sometimes you need to also differentiate security based on the way a principal is interacting with your application. For example, you might want to ensure requests only arrive over HTTPS, in order to protect passwords from eavesdropping or end users from man-in-the-middle attacks. Or, you might want to ensure that an actual human being is making the requests and not some robot or other automated process. This is especially helpful to protect password recovery processes from brute force attacks, or simply to make it harder for people to duplicate your application's key content. To help you achieve these goals, Acegi Security fully supports automatic "channel security", together with JCaptcha integration for human user detection.

Irrespective of how authentication was undertaken, Acegi Security provides a deep set of authorization capabilities. There are three main areas of interest in respect of authorization, these being authorizing web requests, authorizing methods can be invoked, and authorizing access to individual domain object instances. To help you understand the differences, consider the authorization capabilities found in the Servlet Specification web pattern security, EJB Container Managed Security and file system security respectively. Acegi Security provides deep capabilities in all of these important areas, which we'll explore later in this reference guide.

1.2. History

Acegi Security began in late 2003, when a question was posed on the Spring Developers' mailing list asking whether there had been any consideration given to a Spring-based security implementation. At the time the Spring community was relatively small (especially by today's size!), and indeed Spring itself had only existed as a SourceForge project from early 2003. The response to the question was that it was a worthwhile area, although a lack of time currently prevented its exploration.

With that in mind, a simple security implementation was built and not released. A few weeks later another member of the Spring community inquired about security, and at the time this code was offered to them. Several other requests followed, and by January 2004 around twenty people were using the code. These pioneering users were joined by others who suggested a SourceForge project was in order, which was duly established in March 2004.

In those early days, the project didn't have any of its own authentication modules. Container Managed Security was relied upon for the authentication process, with Acegi Security instead focusing on authorization. This was suitable at first, but as more and more users requested additional container support, the fundamental limitation of container-specific authentication realm interfaces was experienced. There was also a related issue of adding new JARs to the container's classpath, which was a common source of end user confusion and misconfiguration.

Acegi Security-specific authentication services were subsequently introduced. Around a year later, the

Acegi Security became an official Spring Framework subproject. The 1.0.0 final release was published in May 2006 - after more than two and a half years of active use in numerous production software projects and many hundreds of improvements and community contributions.

Today Acegi Security enjoys a strong and active open source community. There are thousands of messages about Acegi Security on the support forums. Fourteen developers work on the code itself, with an active community who also regularly share patches and support their peers.

1.3. Release Numbering

It is useful to understand how Acegi Security release numbers work, as it will help you identify the effort (or lack thereof) involved in migrating to future releases of the project. Officially, we use the Apache Portable Runtime Project versioning guidelines, which can be viewed at `http://apr.apache.org/versioning.html`. We quote the introduction contained on that page for your convenience:

“Versions are denoted using a standard triplet of integers: MAJOR.MINOR.PATCH. The basic intent is that MAJOR versions are incompatible, large-scale upgrades of the API. MINOR versions retain source and binary compatibility with older minor versions, and changes in the PATCH level are perfectly compatible, forwards and backwards.”

Chapter 2. Technical Overview

2.1. Runtime Environment

Acegi Security is written to execute within a standard Java 1.3 Runtime Environment. It also supports Java 5.0, although the Java types which are specific to this release are packaged in a separate package with the suffix "tiger" in their JAR filename. As Acegi Security aims to operate in a self-contained manner, there is no need to place any special configuration files into your Java Runtime Environment. In particular, there is no need to configure a special Java Authentication and Authorization Service (JAAS) policy file or place Acegi Security into common classpath locations.

Similarly, if you are using an EJB Container or Servlet Container there is no need to put any special configuration files anywhere, nor include Acegi Security in a server classloader.

This above design offers maximum deployment time flexibility, as you can simply copy your target artifact (be it a JAR, WAR or EAR) from one system to another and it will immediately work.

2.2. Shared Components

Let's explore some of the most important shared components in Acegi Security. Components are considered "shared" if they are central to the framework and the framework cannot operate without them. These Java types represent the building blocks of the remaining system, so it's important to understand that they're there, even if you don't need to directly interact with them.

The most fundamental object is `SecurityContextHolder`. This is where we store details of the present security context of the application, which includes details of the principal currently using the application. By default the `SecurityContextHolder` uses a `ThreadLocal` to store these details, which means that the security context is always available to methods in the same thread of execution, even if the security context is not explicitly passed around as an argument to those methods. Using a `ThreadLocal` in this way is quite safe if care is taken to clear the thread after the present principal's request is processed. Of course, Acegi Security takes care of this for you automatically so there is no need to worry about it.

Some applications aren't entirely suitable for using a `ThreadLocal`, because of the specific way they work with threads. For example, a Swing client might want all threads in a Java Virtual Machine to use the same security context. For this situation you would use the `SecurityContextHolder.MODE_GLOBAL`. Other applications might want to have threads spawned by the secure thread also assume the same security identity. This is achieved by using `SecurityContextHolder.MODE_INHERITABLETHREADLOCAL`. You can change the mode from the default `SecurityContextHolder.MODE_THREADLOCAL` in two ways. The first is to set a system property. Alternatively, call a static method on `SecurityContextHolder`. Most applications won't need to change from the default, but if you do, take a look at the JavaDocs for `SecurityContextHolder` to learn more.

Inside the `SecurityContextHolder` we store details of the principal currently interacting with the application. Acegi Security uses an `Authentication` object to represent this information. Whilst you won't normally need to create an `Authentication` object yourself, it is fairly common for users to query the `Authentication` object. You can use the following code block - from anywhere in your application - to do this:

```
Object obj = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

if (obj instanceof UserDetails) {
  String username = ((UserDetails)obj).getUsername();
} else {
  String username = obj.toString();
}
```

The above code introduces a number of interesting relationships and key objects. First, you will notice that there is an intermediate object between `SecurityContextHolder` and `Authentication`. The `SecurityContextHolder.getContext()` method is actually returning a `SecurityContext`. Acegi Security uses a few different `SecurityContext` implementations, such as if we need to store special information related to a request that is not principal-specific. A good example of this is our JCaptcha integration, which needs to know whether the current request came from a human user or not. Because such a decision has nothing at all to do with the principal the request may or may not be authenticated as, we store it in the `SecurityContext`.

Another item to note from the above code fragment is that you can obtain a principal from the `Authentication` object. The principal is just an `Object`. Most of the time this can be cast into a `UserDetails` object. `UserDetails` is a central interface in Acegi Security. It represents a principal, but in an extensible and application-specific way. Think of `UserDetails` as the adapter between your own user database and what Acegi Security needs inside the `SecurityContextHolder`. Being a representation of something from your own user database, quite often you will cast the `UserDetails` to the original object that your application provided, so you can call business-specific methods (like `getEmail()`, `getEmployeeNumber()` and so on).

By now you're probably wondering, so when do I provide a `UserDetails` object? How do I do that? I thought you said this thing was declarative and I didn't need to write any Java code - what gives? The short answer is that there is a special interface called `UserDetailsService`. The only method on this interface accepts a `String`-based username argument and returns a `UserDetails`. Most authentication providers that ship with Acegi Security delegate to a `UserDetailsService` as part of the authentication process. The `UserDetailsService` is used to build the `Authentication` object that is stored in the `SecurityContextHolder`. The good news is that we provide a number of `UserDetailsService` implementations, including one that uses an in-memory map and another that uses JDBC. Most users tend to write their own, though, with such implementations often simply sitting on top of an existing Data Access Object (DAO) that represents their employees, customers, or other users of the enterprise application. Remember the advantage that whatever your UserDetailsService returns can always be obtained from the `SecurityContextHolder`, as per the above code fragment.

Besides the principal, another important method provided by `Authentication` is `getAuthorities()`. This method provides an array of `GrantedAuthority` objects. A `GrantedAuthority` is, not surprisingly, an authority that is granted to the principal. Such authorities are usually "roles", such as `ROLE_ADMINISTRATOR` or `ROLE_HR_SUPERVISOR`. These roles are later on configured for web authorization, method authorization and domain object authorization. Other parts of Acegi Security are capable of interpreting these authorities, and expect them to be present. `GrantedAuthority` objects are usually loaded by the `UserDetailsService`.

Usually the `GrantedAuthority` objects are application-wide permissions. They are not specific to a given domain object. Thus, you wouldn't likely have a `GrantedAuthority` to represent a permission to `Employee` object number 54, because if there are thousands of such authorities you would quickly run out of memory (or, at the very least, cause the application to take a long time to authenticate a user). Of course, Acegi Security is expressly designed to handle this common requirement, but you'd instead use the project's domain object security capabilities for this purpose.

Last but not least, sometimes you will need to store the `SecurityContext` between HTTP requests. Other times the principal will re-authenticate on every request, although most of the time it will be stored. The `HttpSessionContextIntegrationFilter` is responsible for storing a `SecurityContext` between HTTP requests. As suggested by the name of the class, the `HttpSession` is used to store this information. You should never interact directly with the `HttpSession` for security purposes. There is simply no justification for doing so - always use the `SecurityContextHolder` instead.

Just to recap, the major building blocks of Acegi Security are:

- `SecurityContextHolder`, to provide any type access to the `SecurityContext`.
- `SecurityContext`, to hold the `Authentication` and possibly request-specific security information.
- `HttpSessionContextIntegrationFilter`, to store the `SecurityContext` in the `HttpSession` between web requests.
- `Authentication`, to represent the principal in an Acegi Security-specific manner.
- `GrantedAuthority`, to reflect the application-wide permissions granted to a principal.
- `UserDetails`, to provide the necessary information to build an Authentication object from your application's DAOs.
- `UserDetailsService`, to create a `UserDetails` when passed in a `String`-based username (or certificate ID or alike).

Now that you've gained an understanding of these repeatedly-used components, let's take a closer look at the process of authentication.

2.3. Authentication

As mentioned in the beginning of this reference guide, Acegi Security can participate in many different authentication environments. Whilst we recommend people use Acegi Security for authentication and not integrate with existing Container Managed Authentication, it is nevertheless supported - as is integrating with your own proprietary authentication system. Let's first explore authentication from the perspective of Acegi Security managing web security entirely on its own, which is illustrative of the most complex and most common situation.

Consider a typical web application's authentication process:

1. You visit the home page, and click on a link.

2. A request goes to the server, and the server decides that you've asked for a protected resource.

3. As you're not presently authenticated, the server sends back a response indicating that you must authenticate. The response will either be an HTTP response code, or a redirect to a particular web page.

4. Depending on the authentication mechanism, your browser will either redirect to the specific web page so that you can fill out the form, or the browser will somehow retrieve your identity (eg a BASIC authentication dialogue box, a cookie, a X509 certificate etc).

5. The browser will send back a response to the server. This will either be an HTTP POST containing the contents of the form that you filled out, or an HTTP header containing your authentication details.

6. Next the server will decide whether or not the presented credentials are valid. If they're valid, the next step will happen. If they're invalid, usually your browser will be asked to try again (so

you return to step two above).

7. The original request that you made to cause the authentication process will be retried. Hopefully you've authenticated with sufficient granted authorities to access the protected resource. If you have sufficient access, the request will be successful. Otherwise, you'll receive back an HTTP error code 403, which means "forbidden".

Acegi Security has distinct classes responsible for most of the steps described above. The main participants (in the order that they are used) are the `ExceptionTranslationFilter`, an `AuthenticationEntryPoint`, an authentication mechanism, and an `AuthenticationProvider`.

`ExceptionTranslationFilter` is an Acegi Security filter that has responsibility for detecting any Acegi Security exceptions that are thrown. Such exceptions will generally be thrown by an `AbstractSecurityInterceptor`, which is the main provider of authorization services. We will discuss `AbstractSecurityInterceptor` in the next section, but for now we just need to know that it produces Java exceptions and knows nothing about HTTP or how to go about authenticating a principal. Instead the `ExceptionTranslationFilter` offers this service, with specific responsibility for either returning error code 403 (if the principal has been authenticated and therefore simply lacks sufficient access - as per step seven above), or launching an `AuthenticationEntryPoint` (if the principal has not been authenticated and therefore we need to go commence step three).

The `AuthenticationEntryPoint` is responsible for step three in the above list. As you can imagine, each web application will have a default authentication strategy (well, this can be configured like nearly everything else in Acegi Security, but let's keep it simple for now). Each major authentication system will have its own `AuthenticationEntryPoint` implementation, which takes actions such as described in step three.

After your browser decides to submit your authentication credentials (either as an HTTP form post or HTTP header) there needs to be something on the server that "collects" these authentication details. By now we're at step six in the above list. In Acegi Security we have a special name for the function of collecting authentication details from a user agent (usually a web browser), and that name is "authentication mechanism". After the authentication details are collected from the user agent, an "`Authentication` request" object is built and then presented to an `AuthenticationProvider`.

The last played in the Acegi Security authentication process is an `AuthenticationProvider`. Quite simply, it is responsible for taking an `Authentication` request object and deciding whether or not it is valid. The provider will either throw an exception or return a fully populated `Authentication` object. Remember our good friends, `UserDetails` and `UserDetailsService`? If not, head back to the previous section and refresh your memory. Most `AuthenticationProvider`s will ask a `UserDetailsService` to provide a `UserDetails` object. As mentioned earlier, most application will provide their own `UserDetailsService`, although some will be able to use the JDBC or in-memory implementation that ships with Acegi Security. The resultant `UserDetails` object - and particularly the `GrantedAuthority[]`s contained within the `UserDetails` object - will be used when building the fully populated `Authentication` object.

After the authentication mechanism receives back the fully-populated `Authentication` object, it will deem the request valid, put the `Authentication` into the `SecurityContextHolder`, and cause the original request to be retried (step seven above). If, on the other hand, the `AuthenticationProvider` rejected the request, the authentication mechanism will ask the user agent to retry (step two above).

Whilst this describes the typical authentication workflow, the good news is that Acegi Security doesn't mind how you put an `Authentication` inside the `SecurityContextHolder`. The only critical requirement is

that the `SecurityContextHolder` contains an `Authentication` that represents a principal before the `AbstractSecurityInterceptor` needs to authorize a request.

You can (and many users do) write their own filters or MVC controllers to provide interoperability with authentication systems that are not based on Acegi Security. For example, you might be using Container Managed Authentication which makes the current user available from a ThreadLocal or JNDI location. Or you might work for a company that has a legacy proprietary authentication system, which is a corporate "standard" over which you have little control. In such situations it's quite easy to get Acegi Security to work, and still provide authorization capabilities. All you need to do is write a filter (or equivalent) that reads the third-party user information from a location, build an Acegi Security-specific Authentication object, and put it onto the SecurityContextHolder. It's quite easy to do this, and it is a fully-supported integration approach.

2.4. Secure Objects

If you're familiar with AOP, you'd be aware there are different types of advice available: before, after, throws and around. An around advice is very useful, because an advisor can elect whether or not to proceed with a method invocation, whether or not to modify the response, and whether or not to throw an exception. Acegi Security provides an around advice for method invocations as well as web requests. We achieve an around advice for method invocations using AOP Alliance, and we achieve an around advice for web requests using a standard Filter.

For those not familiar with AOP, the key point to understand is that Acegi Security can help you protect method invocations as well as web requests. Most people are interested in securing method invocations on their services layer. This is because the services layer is where most business logic resides in current-generation J2EE applications (for clarification, the author disapproves of this design and instead advocates properly encapsulated domain objects together with the DTO, assembly, facade and transparent persistence patterns, but as anemic domain objects is the present mainstream approach, we'll talk about it here). If you just need to secure method invocations to the services layer, using the Spring's standard AOP platform (otherwise known as AOP Alliance) will be adequate. If you need to secure domain objects directly, you will likely find that AspectJ is worth considering.

You can elect to perform method authorization using AspectJ or AOP Alliance, or you can elect to perform web request authorization using filters. You can use zero, one, two or three of these approaches together. The mainstream usage is to perform some web request authorization, coupled with some AOP Alliance method invocation authorization on the services layer.

Acegi Security uses the term "secure object" to refer to any object that can have security applied to it. Each secure object supported by Acegi Security has its own class, which is a subclass of `AbstractSecurityInterceptor`. Importantly, by the time the `AbstractSecurityInterceptor` is run, the `SecurityContextHolder` will contain a valid `Authentication` if the principal has been authenticated.

The `AbstractSecurityInterceptor` provides a consistent workflow for handling secure object requests. This workflow includes looking up the "configuration attributes" associated with the present request. A "configuration attribute" can be thought of as a String that has special meaning to the classes used by `AbstractSecurityInterceptor`. They're normally configured against your `AbstractSecurityInterceptor` using XML. Anyway, the `AbstractSecurityInterceptor` will ask an `AccessDecisionManager` "here's the configuration attributes, here's the current `Authentication` object, and here's details of the current request - is this particular principal allowed to perform this particular operation?".

Assuming `AccessDecisionManager` decides to allow the request, the `AbstractSecurityInterceptor` will

normally just proceed with the request. Having said that, on rare occasions users may want to replace the `Authentication` inside the `SecurityContext` with a different `Authentication`, which is handled by the `AccessDecisionManager` calling a `RunAsManager`. This might be useful in reasonably unusual situations, such as if a services layer method needs to call a remote system and present a different identity. Because Acegi Security automatically propagates security identity from one server to another (assuming you're using a properly-configured RMI or HttpInvoker remoting protocol client), this may be useful.

Following the secure object proceeding and then returning - which may mean a method invocation completing or a filter chain proceeding - the `AbstractSecurityInterceptor` gets one final chance to handle the invocation. At this stage the `AbstractSecurityInterceptor` is interested in possibly modifying the return object. We might want this to happen because an authorization decision couldn't be made "on the way in" to a secure object invocation. Being highly pluggable, `AbstractSecurityInterceptor` will pass control to an `AfterInvocationManager` to actually modify the object if needed. This class even can entirely replace the object, or throw an exception, or not change it in any way.

Because `AbstractSecurityInterceptor` is the central template class, it seems fitting that the first figure should be devoted to it.

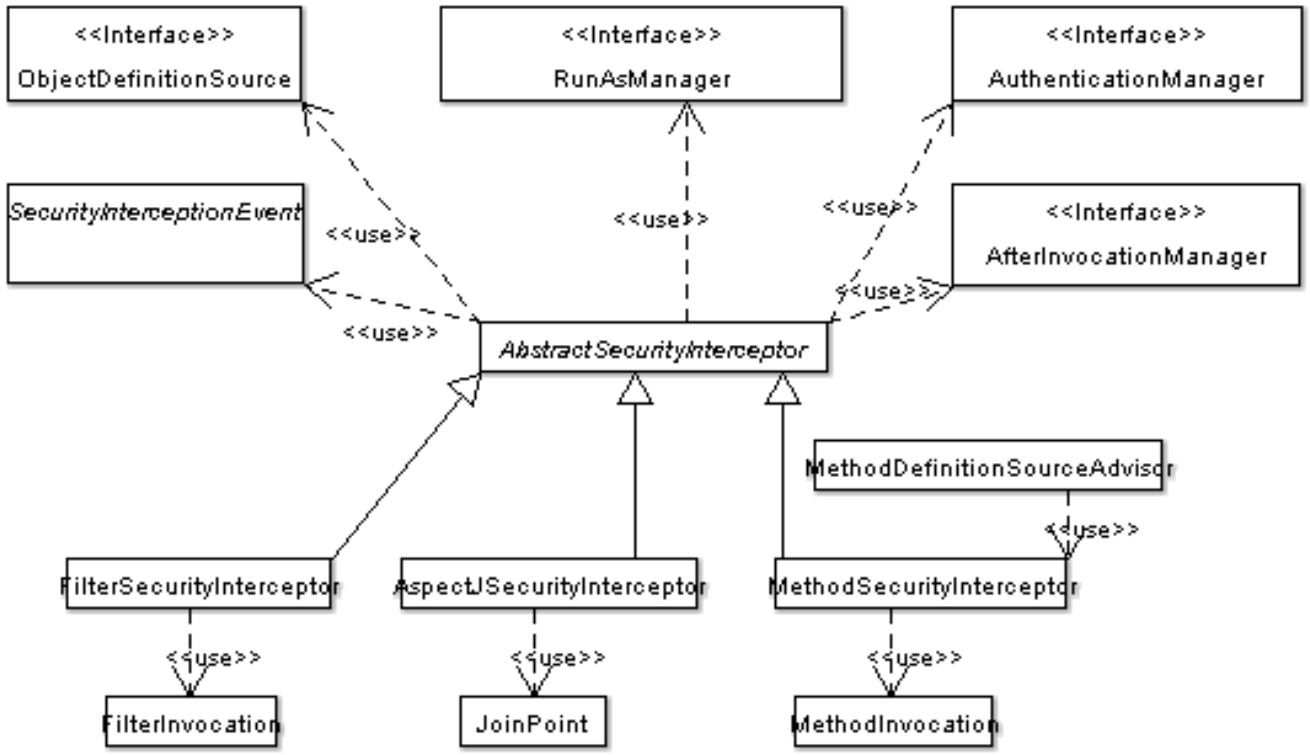

Figure 1: The key "secure object" model

Only developers contemplating an entirely new way of intercepting and authorizing requests would need to use secure objects directly. For example, it would be possible to build a new secure object to secure calls to a messaging system. Anything that requires security and also provides a way of intercepting a call (like the AOP around advice semantics) is capable of being made into a secure object. Having said that, most Spring applications will simply use the three currently supported secure object types (AOP Alliance `MethodInvocation`, AspectJ `JoinPoint` and web request `FilterInterceptor`) with complete transparency.

2.5. Conclusion

Congratulations! You have enough of a high-level picture of Acegi Security to embark on your project. We've explored the shared components, how authentication works, and reviewed the common authorization concept of a "secure object". Everything that follows in this reference guide may or may not apply to your particular needs, and can be read in any order.

Chapter 3. Supporting Infrastructure

This chapter introduces some of the supplementary and supporting infrastructure used by Acegi Security. If a capability is not directly related to security, yet included in the Acegi Security project, we will discuss it in this chapter.

3.1. Localization

Acegi Security supports localization of exception messages that end users are likely to see. If your application is designed for English users, you don't need to do anything as by default all Acegi Security messages are in English. If you need to support other locales, everything you need to know is contained in this section.

All exception messages can be localized, including messages related to authentication failures and access being denied (authorization failures). Exceptions and logging that is focused on developers or system deployers (including incorrect attributes, interface contract violations, using incorrect constructors, startup time validation, debug-level logging) etc are not localized and instead are hard-coded in English within Acegi Security's code.

Shipping in the `acegi-security-xx.jar` you will find an `org.acegisecurity` package that in turn contains a `messages.properties` file. This should be referred to by your `ApplicationContext`, as Acegi Security classes implement Spring's `MessageSourceAware` interface and expect the message resolver to be dependency injected at application context startup time. Usually all you need to do is register a bean inside your application context to refer to the messages. An example is shown below:

```
<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename"><value>org/acegisecurity/messages</value></property>
</bean>
```

The `messages.properties` is named in accordance with standard resource bundles and represents the default language supported by Acegi Securtiy messages. This default file is in English. If you do not register a message source, Acegi Security will still work correctly and fallback to hard-coded English versions of the messages.

If you wish to customize the `messages.properties` file, or support other languages, you should copy the file, rename it accordingly, and register it inside the above bean definition. There are not a large number of message keys inside this file, so localization should not be considered a major initiative. If you do perform localization of this file, please consider sharing your work with the community by logging a JIRA task and attaching your appropriately-named localized version of `messages.properties`.

Rounding out the discussion on localization is the Spring `ThreadLocal` known as `org.springframework.context.i18n.LocaleContextHolder`. You should set the `LocaleContextHolder` to represent the preferred `Locale` of each user. Acegi Security will attempt to locate a message from the message source using the `Locale` obtained from this `ThreadLocal`. Please refer to Spring documentation for further details on using `LocaleContextHolder` and the helper classes that can automatically set it for you (eg `AcceptHeaderLocaleResolver`, `CookieLocaleResolver`, `FixedLocaleResolver`, `SessionLocaleResolver` etc)

3.2. Filters

Acegi Security uses many filters, as referred to throughout the remainder of this reference guide. You have a choice in how these filters are added to your web application, in that you can use either `FilterToBeanProxy` or `FilterChainProxy`. We'll look at both below.

Most filters are configured using the `FilterToBeanProxy`. An example configuration from `web.xml` follows:

```
<filter>
  <filter-name>Acegi HTTP Request Security Filter</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.ClassThatImplementsFilter</param-value>
  </init-param>
</filter>
```

Notice that the filter in `web.xml` is actually a `FilterToBeanProxy`, and not the filter that will actually implement the logic of the filter. What `FilterToBeanProxy` does is delegate the `Filter`'s methods through to a bean which is obtained from the Spring application context. This enables the bean to benefit from the Spring application context lifecycle support and configuration flexibility. The bean must implement `javax.servlet.Filter`.

The `FilterToBeanProxy` only requires a single initialization parameter, `targetClass` or `targetBean`. The `targetClass` parameter locates the first object in the application context of the specified class, whilst `targetBean` locates the object by bean name. Like standard Spring web applications, the `FilterToBeanProxy` accesses the application context via `WebApplicationContextUtils.getWebApplicationContext(ServletContext)`, so you should configure a `ContextLoaderListener` in `web.xml`.

There is a lifecycle issue to consider when hosting `Filter`s in an IoC container instead of a servlet container. Specifically, which container should be responsible for calling the `Filter`'s "startup" and "shutdown" methods? It is noted that the order of initialization and destruction of a `Filter` can vary by servlet container, and this can cause problems if one `Filter` depends on configuration settings established by an earlier initialized `Filter`. The Spring IoC container on the other hand has more comprehensive lifecycle/IoC interfaces (such as `InitializingBean`, `DisposableBean`, `BeanNameAware`, `ApplicationContextAware` and many others) as well as a well-understood interface contract, predictable method invocation ordering, autowiring support, and even options to avoid implementing Spring interfaces (eg the `destroy-method` attribute in Spring XML). For this reason we recommend the use of Spring lifecycle services instead of servlet container lifecycle services wherever possible. By default `FilterToBeanProxy` will not delegate `init(FilterConfig)` and `destroy()` methods through to the proxied bean. If you do require such invocations to be delegated, set the `lifecycle` initialization parameter to `servlet-container-managed`.

Rather than using `FilterToBeanProxy`, we strongly recommend to use `FilterChainProxy` instead. Whilst `FilterToBeanProxy` is a very useful class, the problem is that the lines of code required for `<filter>` and `<filter-mapping>` entries in `web.xml` explodes when using more than a few filters. To overcome this issue, Acegi Security provides a `FilterChainProxy` class. It is wired using a `FilterToBeanProxy` (just like in the example above), but the target class is `org.acegisecurity.util.FilterChainProxy`. The filter chain is then declared in the application context, using code such as this:

```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
```

```
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /webServices/**=httpSessionContextIntegrationFilterWithASCFalse,basicProcessingFilter,exceptionTranslation
      /**=httpSessionContextIntegrationFilterWithASCTrue,authenticationProcessingFilter,exceptionTranslationFilt
    </value>
  </property>
</bean>
```

You may notice similarities with the way `FilterSecurityInterceptor` is declared. Both regular expressions and Ant Paths are supported, and the most specific URIs appear first. At runtime the `FilterChainProxy` will locate the first URI pattern that matches the current web request. Each of the corresponding configuration attributes represent the name of a bean defined in the application context. The filters will then be invoked in the order they are specified, with standard `FilterChain` behaviour being respected (a `Filter` can elect not to proceed with the chain if it wishes to end processing).

As you can see, `FilterChainProxy` requires the duplication of filter names for different request patterns (in the above example, `exceptionTranslationFilter` and `filterSecurityInterceptor` are duplicated). This design decision was made to enable `FilterChainProxy` to specify different `Filter` invocation orders for different URI patterns, and also to improve both the expressiveness (in terms of regular expressions, Ant Paths, and any custom `FilterInvocationDefinitionSource` implementations) and clarity of which `Filter`s should be invoked.

You may have noticed we have declared two `HttpSessionContextIntegrationFilter`s in the filter chain (`ASC` is short for `allowSessionCreation`, a property of `HttpSessionContextIntegrationFilter`). As web services will never present a `jsessionid` on future requests, creating `HttpSession`s for such user agents would be wasteful. If you had a high-volume application which required maximum scalability, we recommend you use the approach shown above. For smaller applications, using a single `HttpSessionContextIntegrationFilter` (with its default `allowSessionCreation` as `true`) would likely be sufficient.

In relation to lifecycle issues, the `FilterChainProxy` will always delegate `init(FilterConfig)` and `destroy()` methods through to the underlaying `Filter`s if such methods are called against `FilterChainProxy` itself. In this case, `FilterChainProxy` guarantees to only initialize and destroy each `Filter` once, irrespective of how many times it is declared by the `FilterInvocationDefinitionSource`. You control the overall choice as to whether these methods are called or not via the `lifecycle` initialization parameter of the `FilterToBeanProxy` that proxies `FilterChainProxy`. As discussed above, by default any servlet container lifecycle invocations are not delegated through to `FilterChainProxy`.

You can also omit a URI pattern from the filter chain by using the token `#NONE#` on the right-hand side of the `<URI Pattern> = <Filter Chain>` expression. For example, using the example above, if you wanted to exclude the `/webservices` location completely, you would modify the corresponding line in the bean declaration to be

```
/webServices/**=#NONE#
```

Note that anything matching this path will then have no authentication or authorization services applied and will be freely accessible.

The order that filters are defined in `web.xml` is very important. Irrespective of which filters you are actually using, the order of the `<filter-mapping>`s should be as follows:

1. `ChannelProcessingFilter`, because it might need to redirect to a different protocol

2. `ConcurrentSessionFilter`, because it doesn't use any `SecurityContextHolder` functionality but needs to update the `SessionRegistry` to reflect ongoing requests from the principal

3. `HttpSessionContextIntegrationFilter`, so a `SecurityContext` can be setup in the `SecurityContextHolder` at the beginning of a web request, and any changes to the `SecurityContext` can be copied to the `HttpSession` when the web request ends (ready for use with the next web request)

4. Authentication processing mechanisms - `AuthenticationProcessingFilter`, `CasProcessingFilter`, `BasicProcessingFilter`, `HttpRequestIntegrationFilter`, `JbossIntegrationFilter` etc - so that the `SecurityContextHolder` can be modified to contain a valid `Authentication` request token

5. The `SecurityContextHolderAwareRequestFilter`, if you are using it to install an Acegi Security aware `HttpServletRequestWrapper` into your servlet container

6. `RememberMeProcessingFilter`, so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`, and the request presents a cookie that enables remember-me services to take place, a suitable remembered `Authentication` object will be put there

7. `AnonymousProcessingFilter`, so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`, an anonymous `Authentication` object will be put there

8. `ExceptionTranslationFilter`, to catch any Acegi Security exceptions so that either an HTTP error response can be returned or an appropriate `AuthenticationEntryPoint` can be launched

9. `FilterSecurityInterceptor`, to protect web URIs

All of the above filters use `FilterToBeanProxy` or `FilterChainProxy`. It is recommended that a single `FilterToBeanProxy` proxy through to a single `FilterChainProxy` for each application, with that `FilterChainProxy` defining all of Acegi Security `Filter`s.

If you're using SiteMesh, ensure Acegi Security filters execute before the SiteMesh filters are called. This enables the `SecurityContextHolder` to be populated in time for use by SiteMesh decorators

Chapter 4. Channel Security

4.1. Overview

In addition to coordinating the authentication and authorization requirements of your application, Acegi Security is also able to ensure unauthenticated web requests have certain properties. These properties may include being of a particular transport type, having a particular `HttpSession` attribute set and so on. The most common requirement is for your web requests to be received using a particular transport protocol, such as HTTPS.

An important issue in considering transport security is that of session hijacking. Your web container manages a `HttpSession` by reference to a `jsessionid` that is sent to user agents either via a cookie or URL rewriting. If the `jsessionid` is ever sent over HTTP, there is a possibility that session identifier can be intercepted and used to impersonate the user after they complete the authentication process. This is because most web containers maintain the same session identifier for a given user, even after they switch from HTTP to HTTPS pages.

If session hijacking is considered too significant a risk for your particular application, the only option is to use HTTPS for every request. This means the `jsessionid` is never sent across an insecure channel. You will need to ensure your `web.xml`-defined `<welcome-file>` points to an HTTPS location, and the application never directs the user to an HTTP location. Acegi Security provides a solution to assist with the latter.

4.2. Configuration

To utilise Acegi Security's channel security services, add the following lines to `web.xml`:

```
<filter>
  <filter-name>Acegi Channel Processing Filter</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.securechannel.ChannelProcessingFilter</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Acegi Channel Processing Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

As usual when running `FilterToBeanProxy`, you will also need to configure the filter in your application context:

```
<bean id="channelProcessingFilter" class="org.acegisecurity.securechannel.ChannelProcessingFilter">
  <property name="channelDecisionManager"><ref bean="channelDecisionManager"/></property>
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      \A/secure/.*\Z=REQUIRES_SECURE_CHANNEL
      \A/acegilogin.jsp.*\Z=REQUIRES_SECURE_CHANNEL
      \A/j_acegi_security_check.*\Z=REQUIRES_SECURE_CHANNEL
      \A.*\Z=REQUIRES_INSECURE_CHANNEL
```

```
    </value>
  </property>
</bean>

<bean id="channelDecisionManager" class="org.acegisecurity.securechannel.ChannelDecisionManagerImpl">
  <property name="channelProcessors">
    <list>
      <ref bean="secureChannelProcessor"/>
      <ref bean="insecureChannelProcessor"/>
    </list>
  </property>
</bean>

<bean id="secureChannelProcessor" class="org.acegisecurity.securechannel.SecureChannelProcessor"/>
<bean id="insecureChannelProcessor" class="org.acegisecurity.securechannel.InsecureChannelProcessor"/>
```

Like `FilterSecurityInterceptor`, Apache Ant style paths are also supported by the `ChannelProcessingFilter`.

The `ChannelProcessingFilter` operates by filtering all web requests and determining the configuration attributes that apply. It then delegates to the `ChannelDecisionManager`. The default implementation, `ChannelDecisionManagerImpl`, should suffice in most cases. It simply delegates through the list of configured `ChannelProcessor` instances. A `ChannelProcessor` will review the request, and if it is unhappy with the request (eg it was received across the incorrect transport protocol), it will perform a redirect, throw an exception or take whatever other action is appropriate.

Included with Acegi Security are two concrete `ChannelProcessor` implementations: `SecureChannelProcessor` ensures requests with a configuration attribute of `REQUIRES_SECURE_CHANNEL` are received over HTTPS, whilst `InsecureChannelProcessor` ensures requests with a configuration attribute of `REQUIRES_INSECURE_CHANNEL` are received over HTTP. Both implementations delegate to a `ChannelEntryPoint` if the required transport protocol is not used. The two `ChannelEntryPoint` implementations included with Acegi Security simply redirect the request to HTTP and HTTPS as appropriate. Appropriate defaults are assigned to the `ChannelProcessor` implementations for the configuration attribute keywords they respond to and the `ChannelEntryPoint` they delegate to, although you have the ability to override these using the application context.

Note that the redirections are absolute (eg `http://www.company.com:8080/app/page`), not relative (eg `/app/page`). During testing it was discovered that Internet Explorer 6 Service Pack 1 has a bug whereby it does not respond correctly to a redirection instruction which also changes the port to use. Accordingly, absolute URLs are used in conjunction with bug detection logic in the `PortResolverImpl` that is wired up by default to many Acegi Security beans. Please refer to the JavaDocs for `PortResolverImpl` for further details.

You should note that using a secure channel is recommended if usernames and passwords are to be kept secure during the login process. If you do decide to use `ChannelProcessingFilter` with form-based login, please ensure that your login page is set to `REQUIRES_SECURE_CHANNEL`, and that the `AuthenticationProcessingFilterEntryPoint.forceHttps` property is `true`.

4.3. Conclusion

Once configured, using the channel security filter is very easy. Simply request pages without regard to the protocol (ie HTTP or HTTPS) or port (eg 80, 8080, 443, 8443 etc). Obviously you'll still need a way of making the initial request (probably via the `web.xml` `<welcome-file>` or a well-known home page URL), but once this is done the filter will perform redirects as defined by your application context.

You can also add your own `ChannelProcessor` implementations to the `ChannelDecisionManagerImpl`. For example, you might set a `HttpSession` attribute when a human user is detected via a "enter the contents of this graphic" procedure. Your `ChannelProcessor` would respond to say `REQUIRES_HUMAN_USER` configuration attributes and redirect to an appropriate entry point to start the human user validation process if the `HttpSession` attribute is not currently set.

To decide whether a security check belongs in a `ChannelProcessor` or an `AccessDecisionVoter`, remember that the former is designed to handle unauthenticated requests, whilst the latter is designed to handle authenticated requests. The latter therefore has access to the granted authorities of the authenticated principal. In addition, problems detected by a `ChannelProcessor` will generally cause an HTTP/HTTPS redirection so its requirements can be met, whilst problems detected by an `AccessDecisionVoter` will ultimately result in an `AccessDeniedException` (depending on the governing `AccessDecisionManager`).

Chapter 5. Tag Libraries

5.1. Overview

Acegi Security comes bundled with several JSP tag libraries that eases JSP writing. The tag libraries are known as `authz` and provide a range of different services.

5.2. Configuration

All taglib classes are included in the core `acegi-security-xx.jar` file, with the `authz.tld` located in the JAR's `META-INF` directory. This means for JSP 1.2+ web containers you can simply include the JAR in the WAR's `WEB-INF/lib` directory and it will be available. If you're using a JSP 1.1 container, you'll need to declare the JSP taglib in your `web.xml file`, and include `authz.tld` in the `WEB-INF/lib` directory. The following fragment is added to `web.xml`:

```
<taglib>
  <taglib-uri>http://acegisecurity.org/authz</taglib-uri>
  <taglib-location>/WEB-INF/authz.tld</taglib-location>
</taglib>
```

5.3. Usage

Now that you've configured the tag libraries, refer to the individual reference guide sections for details on how to use them.

Part II. Authentication

In this part of the reference guide we will examine individual authentication mechanisms and their corresponding `AuthenticationProvider`s. We'll also look at how to configure authentication more generally, including if you have several authentication approaches that need to be chained together.

Chapter 6. Common Authentication Services

6.1. Mechanisms, Providers and Entry Points

If you're using Acegi Security-provided authentication approaches, you'll usually need to configure a web filter, together with an `AuthenticationProvider` and `AuthenticationEntryPoint`. In this section we are going to explore an example application that needs to support both form-based authentication (ie so a nice HTML page is presented to a user for them to login) plus BASIC authentication (ie so a web service or similar can access protected resources).

In the web.xml, this application will need a single Acegi Security filter in order to use the FilterChainProxy. Nearly every Acegi Security application will have such an entry, and it looks like this:

```
<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>
```

The above declarations will cause every web request to be passed through to Acegi Security's FilterChainProxy. As explained in the filters section of this reference guide, the `FilterChainProxy` is a generally-useful class that enables web requests to be passed to different filters based on the URL patterns. Those delegated filters are managed inside the application context, so they can benefit from dependency injection. Let's have a look at what the FilterChainProxy bean definition would look like inside your application context:

```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /**=httpSessionContextIntegrationFilter,logoutFilter,authenticationProcessingFilter,basicProcessingFilter,
    </value>
  </property>
</bean>
```

Internally Acegi Security will use a `PropertyEditor` to convert the string presented in the above XML fragment into a `FilterInvocationDefinitionSource` object. What's important to note at this stage is that a series of filters will be run - in the order specified by the declaration - and each of those filters are actually the `<bean id>` of another bean inside the application context. So, in our case some extra beans will also appear in the application context, and they'll be named `httpSessionContextIntegrationFilter`, `logoutFilter` and so on. The order that the filters should appear is discussed in the filters section of the reference guide - although they are correct in the above example.

In our example we have the `AuthenticationProcessingFilter` and `BasicProcessingFilter` being used. These are the "authentication mechanisms" that respond to form-based authentication and BASIC

HTTP header-based authentication respectively (we discussed the role of authentication mechanisms earlier in this reference guide). If you weren't using form or BASIC authentication, neither of these beans would be defined. You'd instead define filters applicable to your desired authentication environment, such as `DigestProcessingFilter` or `CasProcessingFilter`. Refer to the individual chapters of this part of the reference guide to learn how to configure each of these authentication mechanisms.

Recall that `HttpSessionContextIntegrationFilter` keeps the contents of the `SecurityContext` between invocations inside an HTTP session. This means the authentication mechanisms are only used once, being when the principal initially tries to authenticate. The rest of the time the authentication mechanisms sit there and silently pass the request through to the next filter in the chain. That is a practical requirement due to the fact that few authentication approaches present credentials on each and every call (BASIC authentication being a notable exception), but what happens if a principal's account gets cancelled or disabled or otherwise changed (eg an increase or decrease in `GrantedAuthority[]`s) after the initial authentication step? Let's look at how that is handled now.

The major authorization provider for secure objects has previously been introduced as `AbstractSecurityInterceptor`. This class needs to have access to an `AuthenticationManager`. It also has configurable settings to indicate whether an `Authentication` object should be re-authenticated on each secure object invocation. By default it just accepts any `Authentication` inside the `SecurityContextHolder` is authenticated if `Authentication.isAuthenticated()` returns true. This is great for performance, but not ideal if you want to ensure up-to-the-moment authentication validity. For such cases you'll probably want to set the `AbstractSecurityInterceptor.alwaysReauthenticate` property to true.

You might be asking yourself, "what's this `AuthenticationManager`?". We haven't explored it before, but we have discussed the concept of an `AuthenticationProvider`. Quite simply, an `AuthenticationManager` is responsible for passing requests through a chain of AuthenticationProviders. It's a little like the filter chain we discussed earlier, although there are some differences. There is only one `AuthenticationManager` implementation shipped with Acegi Security, so let's look at how it's configured for the example we're using in this chapter:

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

It's probably worth mentioning at this point that your authentication mechanisms (which are usually filters) are also injected with a reference to the `AuthenticationManager`. So both `AbstractSecurityInterceptor` as well as the authentication mechanisms will use the above `ProviderManager` to poll a list of `AuthenticationProvider`s.

In our example we have three providers. They are tried in the order shown (which is implied by the use of a `List` instead of a `Set`), with each provider able to attempt authentication, or skip authentication by simply returning `null`. If all implementations return null, the `ProviderManager` will throw a suitable exception. If you're interested in learning more about chaining providers, please refer to the `ProviderManager` JavaDocs.

The providers to use will sometimes be interchangeable with the authentication mechanisms, whilst at other times they will depend on a specific authentication mechanism. For example, the `DaoAuthenticationProvider` just needs a string-based username and password. Various authentication

mechanisms result in the collection of a string-based username and password, including (but not limited to) BASIC and form authentication. Equally, some authentication mechanisms create an authentication request object which can only be interpreted by a single type of `AuthenticationProvider`. An example of this one-to-one mapping would be JA-SIG CAS, which uses the notion of a service ticket which can therefore only be authenticated by `CasAuthenticationProvider`. A further example of a one-to-one mapping would be the LDAP authentication mechanism, which can only be processed an the `LdapAuthenticationProvider`. The specifics of such relationships are detailed in the JavaDocs for each class, plus the authentication approach-specific chapters of this reference guide. You need not be terribly concerned about this implementation detail, because if you forget to register a suitable provider, you'll simply receive a `ProviderNotFoundException` when an attempt to authenticate is made.

After configuring the correct authentication mechanisms in the `FilterChainProxy`, and ensuring that a corresponding `AuthenticationProvider` is registered in the `ProviderManager`, your last step is to configure an `AuthenticationEntryPoint`. Recall that earlier we discussed the role of `ExceptionTranslationFilter`, which is used when HTTP-based requests should receive back an HTTP header or HTTP redirect in order to start authentication. Continuing on with our earlier example:

```
<bean id="exceptionTranslationFilter" class="org.acegisecurity.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint"><ref local="authenticationProcessingFilterEntryPoint"/></property>
  <property name="accessDeniedHandler">
    <bean class="org.acegisecurity.ui.AccessDeniedHandlerImpl">
      <property name="errorPage" value="/accessDenied.jsp"/>
    </bean>
  </property>
</bean>

<bean id="authenticationProcessingFilterEntryPoint" class="org.acegisecurity.ui.webapp.AuthenticationProcessingF
  <property name="loginFormUrl"><value>/acegilogin.jsp</value></property>
  <property name="forceHttps"><value>false</value></property>
</bean>
```

Notice that the `ExceptionTranslationFilter` requires two collaborators. The first, `AccessDeniedHandlerImpl`, uses a `RequestDispatcher` forward to display the specified access denied error page. We use a forward so that the `SecurityContextHolder` still contains details of the principal, which may be useful for display to the user (in old releases of Acegi Security we relied upon the servlet container to handle a 403 error message, which lacked this useful contextual information). `AccessDeniedHandlerImpl` will also set the HTTP header to 403, which is the official error code to indicate access denied. In the case of the `AuthentionEntryPoint`, here we're setting what action we would like taken when an unauthenticated principal attempts to perform a protected operation. Because in our example we're going to be using form-based authentication, we specify `AuthenticationProcessinFilterEntryPoint` and the URL of the login page. Your application will usually only have one entry point, and most authentication approaches define their own specific `AuthenticationEntryPoint`. Details of which entry point to use for each authentication approach is discussed in the authentication approach-specific chapters of this reference guide.

6.2. UserDetails and Associated Types

As mentioned in the first part of the reference guide, most authentication providers take advantage of the `UserDetails` and `UserDetailsService` interfaces. The contract for this latter interface consists of a single method:

```
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException, DataAccessException;
```

The returned `UserDetails` is an interface that provides getters that guarantee non-null provision of

basic authentication information such as the username, password, granted authorities and whether the user is enabled or disabled. Most authentication providers will use a `UserDetailsService`, even if the username and password are not actually used as part of the authentication decision. Generally such provider will be using the returned `UserDetails` object just for its `GrantedAuthority[]` information, because some other system (like LDAP or X509 or CAS etc) has undertaken the responsibility of actually validating the credentials.

A single concrete implementation of `UserDetails` is provided with Acegi Security, being the `User` class. Acegi Security users will need to decide when writing their `UserDetailsService` what concrete `UserDetails` class to return. In most cases `User` will be used directly or subclassed, although special circumstances (such as object relational mappers) may require users to write their own `UserDetails` implementation from scratch. This is not such an unusual situation, and users should not hesitate to simply return their normal domain object that represents a user of the system. This is especially common given that `UserDetails` is often used to store additional principal-related properties (such as their telephone number and email address), so that they can be easily used by web views.

Given `UserDetailsService` is so simple to implement, it should be easy for users to retrieve authentication information using a persistence strategy of their choice. Having said that, Acegi Security does include a couple of useful base implementations, which we'll look at below.

6.2.1. In-Memory Authentication

Whilst it is easy to use create a custom `UserDetailsService` implementation that extracts information from a persistence engine of choice, many applications do not require such complexity. This is particularly true if you're undertaking a rapid prototype or just starting integrating Acegi Security, when you don't really want to spend time configuring databases or writing `UserDetailsService` implementations. For this sort of situation, a simple option is to configure the `InMemoryDaoImpl` implementation:

```
<bean id="inMemoryDaoImpl" class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      marissa=koala,ROLE_TELLER,ROLE_SUPERVISOR
      dianne=emu,ROLE_TELLER
      scott=wombat,ROLE_TELLER
      peter=opal,disabled,ROLE_TELLER
    </value>
  </property>
</bean>
```

In the above example, the `userMap` property contains each of the usernames, passwords, a list of granted authorities and an optional enabled/disabled keyword. Commas are used to delimit each token. The username must appear to the left of the equals sign, and the password must be the first token to the right of the equals sign. The `enabled` and `disabled` keywords (case insensitive) may appear in the second or any subsequent token. Any remaining tokens are treated as granted authorities, which are created as `GrantedAuthorityImpl` objects (this is just for your reference - most applications don't need custom `GrantedAuthority` implementations, so using the default implementation in this manner is just fine). Note that if a user has no password and/or no granted authorities, the user will not be created in the in-memory authentication repository.

`InMemoryDaoImpl` also offers a `setUserProperties(Properties)` method, which allows you to externalise the `java.util.Properties` in another Spring configured bean or an external properties file. You might like to use Spring's `PropertiesFactoryBean`, which is useful for loading such external properties files. This setter might prove useful for simple applications that have a larger number of users, or

deployment-time configuration changes, but do not wish to use a full database for handling authentication details.

6.2.2. JDBC Authentication

Acegi Security also includes a `UserDetailsService` that can obtain authentication information from a JDBC data source. Internally Spring JDBC is used, so it avoids the complexity of a fully-featured object relational mapper (ORM) just to store user details. If your application does use an ORM tool, you might prefer to write a custom `UserDetailsService` to reuse the mapping files you've probably already created. Returning to `JdbcDaoImpl`, an example configuration is shown below:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"><value>org.hsqldb.jdbcDriver</value></property>
  <property name="url"><value>jdbc:hsqldb:hsql://localhost:9001</value></property>
  <property name="username"><value>sa</value></property>
  <property name="password"><value></value></property>
</bean>

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

You can use different relational database management systems by modifying the `DriverManagerDataSource` shown above. You can also use a global data source obtained from JNDI, as per normal Spring options. Irrespective of the database used and how a `DataSource` is obtained, a standard schema must be used as indicated in `dbinit.txt`. You can download this file from the Acegi Security web site.

If your default schema is unsuitable for your needs, `JdbcDaoImpl` provides two properties that allow customisation of the SQL statements. You may also subclass the `JdbcDaoImpl` if further customisation is necessary. Please refer to the JavaDocs for details, although please note that the class is not intended for complex custom subclasses. If you have complex needs (such as a special schema or would like a certain `UserDetails` implementation returned), you'd be better off writing your own `UserDetailsService`. The base implementation provided with Acegi Security is intended for typical situations, and does not offer infinite configuration flexibility.

6.3. Concurrent Session Handling

Acegi Security is able to prevent a principal from concurrently authenticating to the same application more than a specified number of times. Many ISVs take advantage of this to enforce licensing, whilst network administrators like this feature because it helps prevent people from sharing login names. You can, for example, stop user "Batman" from logging onto the web application from two different sessions.

To use concurrent session support, you'll need to add the following to `web.xml`:

```
<listener>
  <listener-class>org.acegisecurity.ui.session.HttpSessionEventPublisher</listener-class>
</listener>
```

In addition, you will need to add the `org.acegisecurity.concurrent.ConcurrentSessionFilter` to your `FilterChainProxy`. The `ConcurrentSessionFilter` requires two properties, `sessionRegistry`, which generally points to an instance of `SessionRegistryImpl`, and `expiredUrl`, which points to the page to

display when a session has expired.

The `web.xml HttpSessionEventPublisher` causes an `ApplicationEvent` to be published to the Spring `ApplicationContext` every time a `HttpSession` commences or terminates. This is critical, as it allows the `SessionRegistryImpl` to be notified when a session ends.

You will also need to wire up the `ConcurrentSessionControllerImpl` and refer to it from your `ProviderManager` bean:

```
<bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <!-- your providers go here -->
  </property>
  <property name="sessionController"><ref bean="concurrentSessionController"/></property>
</bean>

<bean id="concurrentSessionController" class="org.acegisecurity.concurrent.ConcurrentSessionControllerImpl">
  <property name="maximumSessions"><value>1</value></property>
  <property name="sessionRegistry"><ref local="sessionRegistry"/></property>
</bean>

<bean id="sessionRegistry" class="org.acegisecurity.concurrent.SessionRegistryImpl"/>
```

6.4. Authentication Tag Libraries

`AuthenticationTag` is used to simply output a property of the current principal's `Authentication.getPrincipal()` object to the web page.

The following JSP fragment illustrates how to use the `AuthenticationTag`:

```
<authz:authentication operation="username"/>
```

This tag would cause the principal's name to be output. Here we are assuming the `Authentication.getPrincipal()` is a `UserDetails` object, which is generally the case when using the typical `DaoAuthenticationProvider`.

Chapter 7. DAO Authentication Provider

7.1. Overview

Acegi Security includes a production-quality `AuthenticationProvider` implementation called `DaoAuthenticationProvider`. This authentication provider is compatible with all of the authentication mechanisms that generate a `UsernamePasswordAuthenticationToken`, and is probably the most commonly used provider in the framework. Like most of the other authentication providers, the DaoAuthenticationProvider leverages a UserDetailsService in order to lookup the username, password and GrantedAuthority[]s. Unlike most of the other authentication providers that leverage UserDetailsService, this authentication provider actually requires the password to be presented, and the provider will actually evaluate the validity or otherwise of the password presented in an authentication request object.

7.2. Configuration

Aside from adding DaoAuthenticationProvider to your ProviderManager list (as discussed at the start of this part of the reference guide), and ensuring a suitable authentication mechanism is configured to present a UsernamePasswordAuthenticationToken, the configuration of the provider itself is rather simple:

```
<bean id="daoAuthenticationProvider" class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService"><ref bean="inMemoryDaoImpl"/></property>
  <property name="saltSource"><ref bean="saltSource"/></property>
  <property name="passwordEncoder"><ref bean="passwordEncoder"/></property>
</bean>
```

The `PasswordEncoder` and `SaltSource` are optional. A `PasswordEncoder` provides encoding and decoding of passwords presented in the `UserDetails` object that is returned from the configured `UserDetailsService`. A `SaltSource` enables the passwords to be populated with a "salt", which enhances the security of the passwords in the authentication repository. `PasswordEncoder` implementations are provided with Acegi Security covering MD5, SHA and cleartext encodings. Two `SaltSource` implementations are also provided: `SystemWideSaltSource` which encodes all passwords with the same salt, and `ReflectionSaltSource`, which inspects a given property of the returned `UserDetails` object to obtain the salt. Please refer to the JavaDocs for further details on these optional features.

In addition to the properties above, the `DaoAuthenticationProvider` supports optional caching of `UserDetails` objects. The `UserCache` interface enables the `DaoAuthenticationProvider` to place a `UserDetails` object into the cache, and retrieve it from the cache upon subsequent authentication attempts for the same username. By default the `DaoAuthenticationProvider` uses the `NullUserCache`, which performs no caching. A usable caching implementation is also provided, `EhCacheBasedUserCache`, which is configured as follows:

```
<bean id="daoAuthenticationProvider" class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService"><ref bean="userDetailsService"/></property>
  <property name="userCache"><ref bean="userCache"/></property>
</bean>

<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
  <property name="configLocation">
    <value>classpath:/ehcache-failsafe.xml</value>
  </property>
```

```
</bean>

<bean id="userCacheBackend" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <property name="cacheManager">
    <ref local="cacheManager"/>
  </property>
  <property name="cacheName">
    <value>userCache</value>
  </property>
</bean>

<bean id="userCache" class="org.acegisecurity.providers.dao.cache.EhCacheBasedUserCache">
  <property name="cache"><ref local="userCacheBackend"/></property>
</bean>
```

All Acegi Security EH-CACHE implementations (including `EhCacheBasedUserCache`) require an EH-CACHE `Cache` object. The `Cache` object can be obtained from wherever you like, although we recommend you use Spring's factory classes as shown in the above configuration. If using Spring's factory classes, please refer to the Spring documentation for further details on how to optimise the cache storage location, memory usage, eviction policies, timeouts etc.

A design decision was made not to support account locking in the `DaoAuthenticationProvider`, as doing so would have increased the complexity of the `UserDetailsService` interface. For instance, a method would be required to increase the count of unsuccessful authentication attempts. Such functionality could be easily provided by leveraging the application event publishing features discussed below.

`DaoAuthenticationProvider` returns an `Authentication` object which in turn has its `principal` property set. The principal will be either a `String` (which is essentially the username) or a `UserDetails` object (which was looked up from the `UserDetailsService`). By default the `UserDetails` is returned, as this enables applications to add extra properties potentially of use in applications, such as the user's full name, email address etc. If using container adapters, or if your applications were written to operate with `String`s (as was the case for releases prior to Acegi Security 0.6), you should set the `DaoAuthenticationProvider.forcePrincipalAsString` property to `true` in your application context

Chapter 8. Java Authentication and Authorization Service (JAAS) Provider

8.1. Overview

Acegi Security provides a package able to delegate authentication requests to the Java Authentication and Authorization Service (JAAS). This package is discussed in detail below.

Central to JAAS operation are login configuration files. To learn more about JAAS login configuration files, consult the JAAS reference documentation available from Sun Microsystems. We expect you to have a basic understanding of JAAS and its login configuration file syntax in order to understand this section.

8.2. Configuration

The `JaasAuthenticationProvider` attempts to authenticate a user's principal and credentials through JAAS.

Let's assume we have a JAAS login configuration file, `/WEB-INF/login.conf`, with the following contents:

```
JAASTest {
  sample.SampleLoginModule required;
};
```

Like all Acegi Security beans, the `JaasAuthenticationProvider` is configured via the application context. The following definitions would correspond to the above JAAS login configuration file:

```
<bean id="jaasAuthenticationProvider" class="org.acegisecurity.providers.jaas.JaasAuthenticationProvider">
  <property name="loginConfig">
    <value>/WEB-INF/login.conf</value>
  </property>
  <property name="loginContextName">
    <value>JAASTest</value>
  </property>
  <property name="callbackHandlers">
    <list>
      <bean class="org.acegisecurity.providers.jaas.JaasNameCallbackHandler"/>
      <bean class="org.acegisecurity.providers.jaas.JaasPasswordCallbackHandler"/>
    </list>
  </property>
  <property name="authorityGranters">
    <list>
      <bean class="org.acegisecurity.providers.jaas.TestAuthorityGranter"/>
    </list>
  </property>
</bean>
```

The `CallbackHandler`s and `AuthorityGranter`s are discussed below.

8.2.1. JAAS CallbackHandler

Most JAAS `LoginModule`s require a callback of some sort. These callbacks are usually used to obtain the username and password from the user.

In an Acegi Security deployment, Acegi Security is responsible for this user interaction (via the authentication mechanism). Thus, by the time the authentication request is delegated through to JAAS, Acegi Security's authentication mechanism will already have fully-populated an `Authentication` object containing all the information required by the JAAS `LoginModule`.

Therefore, the JAAS package for Acegi Security provides two default callback handlers, `JaasNameCallbackHandler` and `JaasPasswordCallbackHandler`. Each of these callback handlers implement `JaasAuthenticationCallbackHandler`. In most cases these callback handlers can simply be used without understanding the internal mechanics.

For those needing full control over the callback behavior, internally `JaasAutheticationProvider` wraps these `JaasAuthenticationCallbackHandler`s with an `InternalCallbackHandler`. The `InternalCallbackHandler` is the class that actually implements JAAS' normal `CallbackHandler` interface. Any time that the JAAS `LoginModule` is used, it is passed a list of application context configured `InternalCallbackHandler`s. If the `LoginModule` requests a callback against the `InternalCallbackHandler`s, the callback is in-turn passed to the `JaasAuthenticationCallbackHandler`s being wrapped.

8.2.2. JAAS AuthorityGranter

JAAS works with principals. Even "roles" are represented as principals in JAAS. Acegi Security, on the other hand, works with `Authentication` objects. Each `Authentication` object contains a single principal, and multiple `GrantedAuthority`[]s. To facilitate mapping between these different concepts, Acegi Security's JAAS package includes an `AuthorityGranter` interface.

An `AuthorityGranter` is responsible for inspecting a JAAS principal and returning a `String`. The `JaasAuthenticationProvider` then creates a `JaasGrantedAuthority` (which implements Acegi Security's `GrantedAuthority` interface) containing both the `AuthorityGranter`-returned `String` and the JAAS principal that the `AuthorityGranter` was passed. The `JaasAuthenticationProvider` obtains the JAAS principals by firstly successfully authenticating the user's credentials using the JAAS `LoginModule`, and then accessing the `LoginContext` it returns. A call to `LoginContext.getSubject().getPrincipals()` is made, with each resulting principal passed to each `AuthorityGranter` defined against the `JaasAuthenticationProvider.setAuthorityGranters(List)` property.

Acegi Security does not include any production `AuthorityGranter`s given that every JAAS principal has an implementation-specific meaning. However, there is a `TestAuthorityGranter` in the unit tests that demonstrates a simple `AuthorityGranter` implementation.

Chapter 9. Siteminder Authentication Mechanism

9.1. Overview

Siteminder is a commercial single sign on solution by Computer Associates.

Acegi Security provides a filter, `SiteminderAuthenticationProcessingFilter` and provider, `SiteminderAuthenticationProvider` that can be used to process requests that have been pre-authenticated by Siteminder. This filter assumes that you're using Siteminder for *authentication*, and that you're using Acegi Security for *authorization*. The use of Siteminder for *authorization* is not yet directly supported by Acegi Security.

When using Siteminder, an agent is setup on your web server to intercept a principal's first call to your application. The agent redirects the web request to a single sign-on login page, and once authenticated, your application receives the request. Inside the HTTP request is a header - such as `SM_USER` - which identifies the authenticated principal (please refer to your organization's "single sign-on" group for header details in your particular configuration).

9.2. Configuration

The first step in setting up Acegi Security's Siteminder support is to define the authentication mechanism that will inspect the HTTP header discussed earlier. It will be responsible for generating a `UsernamePasswordAuthenticationToken` that is later sent to the `SiteminderAuthenticationProvider`. Let's look at an example:

```
<bean id="authenticationProcessingFilter" class="org.acegisecurity.ui.webapp.SiteminderAuthenticationProcessingF
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="authenticationFailureUrl"><value>/login.jsp?login_error=1</value></property>
  <property name="defaultTargetUrl"><value>/security.do?method=getMainMenu</value></property>
  <property name="filterProcessesUrl"><value>/j_acegi_security_check</value></property>
  <property name="siteminderUsernameHeaderKey"><value>SM_USER</value></property>
  <property name="formUsernameParameterKey"><value>j_username</value></property>
</bean>
```

In our example above, the bean is being provided an `AuthenticationManager`, as is normally needed by authentication mechanisms. Several URLs are also specified, with the values being self-explanatory. It's important to also specify the HTTP header that Acegi Security should inspect. If you additionally want to support form-based authentication (i.e. in your development environment where Siteminder is not installed), specify the form's username parameter as well - just don't do this in production!

Note that you'll need a `SiteminderAuthenticationProvider` configured against your `ProviderManager` in order to use the Siteminder authentication mechanism. Normally an `AuthenticationProvider` expects the password property to match what it retrieves from the `UserDetailsSource`, but in this case, authentication has already been handled by Siteminder, so password property is not even relevant. This may sound like a security weakness, but remember that users have to authenticate with Siteminder before your application ever receives the requests, so the purpose of your custom `UserDetailsService` should simply be to build the complete `Authentication` object (ie with suitable `GrantedAuthority[]`s).

Advanced tip and word to the wise: If you additionally want to support form-based authentication in your development environment (where Siteminder is typically not installed), specify the form's username parameter as well. Just don't do this in production!

Chapter 10. Run-As Authentication Replacement

10.1. Overview

The `AbstractSecurityInterceptor` is able to temporarily replace the `Authentication` object in the `SecurityContext` and `SecurityContextHolder` during the secure object callback phase. This only occurs if the original `Authentication` object was successfully processed by the `AuthenticationManager` and `AccessDecisionManager`. The `RunAsManager` will indicate the replacement `Authentication` object, if any, that should be used during the `SecurityInterceptorCallback`.

By temporarily replacing the `Authentication` object during the secure object callback phase, the secured invocation will be able to call other objects which require different authentication and authorization credentials. It will also be able to perform any internal security checks for specific `GrantedAuthority` objects. Because Acegi Security provides a number of helper classes that automatically configure remoting protocols based on the contents of the `SecurityContextHolder`, these run-as replacements are particularly useful when calling remote web services

10.2. Configuration

A `RunAsManager` interface is provided by Acegi Security:

```
public Authentication buildRunAs(Authentication authentication, Object object, ConfigAttributeDefinition config)
public boolean supports(ConfigAttribute attribute);
public boolean supports(Class clazz);
```

The first method returns the `Authentication` object that should replace the existing `Authentication` object for the duration of the method invocation. If the method returns `null`, it indicates no replacement should be made. The second method is used by the `AbstractSecurityInterceptor` as part of its startup validation of configuration attributes. The `supports(Class)` method is called by a security interceptor implementation to ensure the configured `RunAsManager` supports the type of secure object that the security interceptor will present.

One concrete implementation of a `RunAsManager` is provided with Acegi Security. The `RunAsManagerImpl` class returns a replacement `RunAsUserToken` if any `ConfigAttribute` starts with `RUN_AS_`. If any such `ConfigAttribute` is found, the replacement `RunAsUserToken` will contain the same principal, credentials and granted authorities as the original `Authentication` object, along with a new `GrantedAuthorityImpl` for each `RUN_AS_ ConfigAttribute`. Each new `GrantedAuthorityImpl` will be prefixed with `ROLE_`, followed by the `RUN_AS ConfigAttribute`. For example, a `RUN_AS_SERVER` will result in the replacement `RunAsUserToken` containing a `ROLE_RUN_AS_SERVER` granted authority.

The replacement `RunAsUserToken` is just like any other `Authentication` object. It needs to be authenticated by the `AuthenticationManager`, probably via delegation to a suitable `AuthenticationProvider`. The `RunAsImplAuthenticationProvider` performs such authentication. It simply accepts as valid any `RunAsUserToken` presented.

To ensure malicious code does not create a `RunAsUserToken` and present it for guaranteed acceptance by

the `RunAsImplAuthenticationProvider`, the hash of a key is stored in all generated tokens. The `RunAsManagerImpl` and `RunAsImplAuthenticationProvider` is created in the bean context with the same key:

```
<bean id="runAsManager" class="org.acegisecurity.runas.RunAsManagerImpl">
  <property name="key"><value>my_run_as_password</value></property>
</bean>

<bean id="runAsAuthenticationProvider" class="org.acegisecurity.runas.RunAsImplAuthenticationProvider">
  <property name="key"><value>my_run_as_password</value></property>
</bean>
```

By using the same key, each `RunAsUserToken` can be validated it was created by an approved `RunAsManagerImpl`. The `RunAsUserToken` is immutable after creation for security reasons

Chapter 11. Form Authentication Mechanism

11.1. Overview

HTTP Form Authentication involves using the `AuthenticationProcessingFilter` to process a login form. This is the most common way that application authenticate end users. Form-based authentication is entirely compatible with the DAO and JAAS authentication providers.

11.2. Configuration

The login form simply contains `j_username` and `j_password` input fields, and posts to a URL that is monitored by the filter (by default `j_acegi_security_check`). The filter is defined in `web.xml` behind a `FilterToBeanProxy` as follows:

```
<filter>
  <filter-name>Acegi Authentication Processing Filter</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.ui.webapp.AuthenticationProcessingFilter</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Acegi Authentication Processing Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

For a discussion of `FilterToBeanProxy`, please refer to the Filters section. The application context will need to define the `AuthenticationProcessingFilter`:

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

The configured `AuthenticationManager` processes each authentication request. If authentication fails, the browser will be redirected to the `authenticationFailureUrl`. The `AuthenticationException` will be placed into the `HttpSession` attribute indicated by `AbstractProcessingFilter.ACEGI_SECURITY_LAST_EXCEPTION_KEY`, enabling a reason to be provided to the user on the error page.

If authentication is successful, the resulting `Authentication` object will be placed into the `SecurityContextHolder`.

Once the `SecurityContextHolder` has been updated, the browser will need to be redirected to the target URL. The target URL is usually indicated by the `HttpSession` attribute specified by `AbstractProcessingFilter.ACEGI_SECURITY_TARGET_URL_KEY`. This attribute is automatically set by the `ExceptionTranslationFilter` when an `AuthenticationException` occurs, so that after login is completed the user can return to what they were trying to access. If for some reason the `HttpSession` does not indicate the target URL, the browser will be redirected to the `defaultTargetUrl` property.

Chapter 12. BASIC Authentication Mechanism

12.1. Overview

Acegi Security provides a `BasicProcessingFilter` which is capable of processing basic authentication credentials presented in HTTP headers. This can be used for authenticating calls made by Spring remoting protocols (such as Hessian and Burlap), as well as normal user agents (such as Internet Explorer and Navigator). The standard governing HTTP Basic Authentication is defined by RFC 1945, Section 11, and the `BasicProcessingFilter` conforms with this RFC. Basic Authentication is an attractive approach to authentication, because it is very widely deployed in user agents and implementation is extremely simple (it's just a Base64 encoding of the username:password, specified in an HTTP header).

12.2. Configuration

To implement HTTP Basic Authentication, it is necessary to define `BasicProcessingFilter` in the filter chain. The application context will need to define the `BasicProcessingFilter` and its required collaborator:

```
<bean id="basicProcessingFilter" class="org.acegisecurity.ui.basicauth.BasicProcessingFilter">
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="authenticationEntryPoint"><ref bean="authenticationEntryPoint"/></property>
</bean>

<bean id="authenticationEntryPoint" class="org.acegisecurity.ui.basicauth.BasicProcessingFilterEntryPoint">
  <property name="realmName"><value>Name Of Your Realm</value></property>
</bean>
```

The configured `AuthenticationManager` processes each authentication request. If authentication fails, the configured `AuthenticationEntryPoint` will be used to retry the authentication process. Usually you will use the `BasicProcessingFilterEntryPoint`, which returns a 401 response with a suitable header to retry HTTP Basic authentication. If authentication is successful, the resulting `Authentication` object will be placed into the `SecurityContextHolder`.

If the authentication event was successful, or authentication was not attempted because the HTTP header did not contain a supported authentication request, the filter chain will continue as normal. The only time the filter chain will be interrupted is if authentication fails and the `AuthenticationEntryPoint` is called, as discussed in the previous paragraph

Chapter 13. Digest Authentication

13.1. Overview

Acegi Security provides a `DigestProcessingFilter` which is capable of processing digest authentication credentials presented in HTTP headers. Digest Authentication attempts to solve many of the weaknesses of Basic authentication, specifically by ensuring credentials are never sent in clear text across the wire. Many user agents support Digest Authentication, including FireFox and Internet Explorer. The standard governing HTTP Digest Authentication is defined by RFC 2617, which updates an earlier version of the Digest Authentication standard prescribed by RFC 2069. Most user agents implement RFC 2617. Acegi Security `DigestProcessingFilter` is compatible with the "`auth`" quality of protection (`qop`) prescribed by RFC 2617, which also provides backward compatibility with RFC 2069. Digest Authentication is a highly attractive option if you need to use unencrypted HTTP (ie no TLS/HTTPS) and wish to maximise security of the authentication process. Indeed Digest Authentication is a mandatory requirement for the WebDAV protocol, as noted by RFC 2518 Section 17.1, so we should expect to see it increasingly deployed and replacing Basic Authentication.

Digest Authentication is definitely the most secure choice between Form Authentication, Basic Authentication and Digest Authentication, although extra security also means more complex user agent implementations. Central to Digest Authentication is a "nonce". This is a value the server generates. Acegi Security's nonce adopts the following format:

```
base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))

expirationTime:   The date and time when the nonce expires, expressed in milliseconds
key:              A private key to prevent modification of the nonce token
```

The `DigestProcessingFilterEntryPoint` has a property specifying the `key` used for generating the nonce tokens, along with a `nonceValiditySeconds` property for determining the expiration time (default 300, which equals five minutes). Whist ever the nonce is valid, the digest is computed by concatenating various strings including the username, password, nonce, URI being requested, a client-generated nonce (merely a random value which the user agent generates each request), the realm name etc, then performing an MD5 hash. Both the server and user agent perform this digest computation, resulting in different hash codes if they disagree on an included value (eg password). In Acegi Security implementation, if the server-generated nonce has merely expired (but the digest was otherwise valid), the `DigestProcessingFilterEntryPoint` will send a `"stale=true"` header. This tells the user agent there is no need to disturb the user (as the password and username etc is correct), but simply to try again using a new nonce.

An appropriate value for `DigestProcessingFilterEntryPoint`'s `nonceValiditySeconds` parameter will depend on your application. Extremely secure applications should note that an intercepted authentication header can be used to impersonate the principal until the `expirationTime` contained in the nonce is reached. This is the key principle when selecting an appropriate setting, but it would be unusual for immensely secure applications to not be running over TLS/HTTPS in the first instance.

Because of the more complex implementation of Digest Authentication, there are often user agent issues. For example, Internet Explorer fails to present an "`opaque`" token on subsequent requests in the same session. Acegi Security filters therefore encapsulate all state information into the "`nonce`" token instead. In our testing, Acegi Security implementation works reliably with FireFox and Internet Explorer, correctly handling nonce timeouts etc.

13.2. Configuration

Now that we've reviewed the theory, let's see how to use it. To implement HTTP Digest Authentication, it is necessary to define `DigestProcessingFilter` in the fitler chain. The application context will need to define the `DigestProcessingFilter` and its required collaborators:

```
<bean id="digestProcessingFilter" class="org.acegisecurity.ui.digestauth.DigestProcessingFilter">
  <property name="userDetailsService"><ref local="jdbcDaoImpl"/></property>
  <property name="authenticationEntryPoint"><ref local="digestProcessingFilterEntryPoint"/></property>
  <property name="userCache"><ref local="userCache"/></property>
</bean>

<bean id="digestProcessingFilterEntryPoint" class="org.acegisecurity.ui.digestauth.DigestProcessingFilterEntryPo
  <property name="realmName"><value>Contacts Realm via Digest Authentication</value></property>
  <property name="key"><value>acegi</value></property>
  <property name="nonceValiditySeconds"><value>10</value></property>
</bean>
```

The configured `UserDetailsService` is needed because `DigestProcessingFilter` must have direct access to the clear text password of a user. Digest Authentication will NOT work if you are using encoded passwords in your DAO. The DAO collaborator, along with the `UserCache`, are typically shared directly with a `DaoAuthenticationProvider`. The `authenticationEntryPoint` property must be `DigestProcessingFilterEntryPoint`, so that `DigestProcessingFilter` can obtain the correct `realmName` and `key` for digest calculations.

Like `BasicAuthenticationFilter`, if authentication is successful an `Authentication` request token will be placed into the `SecurityContextHolder`. If the authentication event was successful, or authentication was not attempted because the HTTP header did not contain a Digest Authentication request, the filter chain will continue as normal. The only time the filter chain will be interrupted is if authentication fails and the `AuthenticationEntryPoint` is called, as discussed in the previous paragraph.

Digest Authentication's RFC offers a range of additional features to further increase security. For example, the nonce can be changed on every request. Despite this, Acegi Security implementation was designed to minimise the complexity of the implementation (and the doubtless user agent incompatibilities that would emerge), and avoid needing to store server-side state. You are invited to review RFC 2617 if you wish to explore these features in more detail. As far as we are aware, Acegi Security implementation does comply with the minimum standards of this RFC.

Chapter 14. Anonymous Authentication

14.1. Overview

Particularly in the case of web request URI security, sometimes it is more convenient to assign configuration attributes against every possible secure object invocation. Put differently, sometimes it is nice to say `ROLE_SOMETHING` is required by default and only allow certain exceptions to this rule, such as for login, logout and home pages of an application. There are also other situations where anonymous authentication would be desired, such as when an auditing interceptor queries the `SecurityContextHolder` to identify which principal was responsible for a given operation. Such classes can be authored with more robustness if they know the `SecurityContextHolder` always contains an `Authentication` object, and never `null`.

14.2. Configuration

Acegi Security provides three classes that together provide an anonymous authentication feature. `AnonymousAuthenticationToken` is an implementation of `Authentication`, and stores the `GrantedAuthority`[]s which apply to the anonymous principal. There is a corresponding `AnonymousAuthenticationProvider`, which is chained into the `ProviderManager` so that `AnonymousAuthenticationTokens` are accepted. Finally, there is an AnonymousProcessingFilter, which is chained after the normal authentication mechanisms and automatically add an `AnonymousAuthenticationToken` to the `SecurityContextHolder` if there is no existing `Authentication` held there. The definition of the filter and authentication provider appears as follows:

```
<bean id="anonymousProcessingFilter" class="org.acegisecurity.providers.anonymous.AnonymousProcessingFilter">
  <property name="key"><value>foobar</value></property>
  <property name="userAttribute"><value>anonymousUser,ROLE_ANONYMOUS</value></property>
</bean>

<bean id="anonymousAuthenticationProvider" class="org.acegisecurity.providers.anonymous.AnonymousAuthenticationP
  <property name="key"><value>foobar</value></property>
</bean>
```

The `key` is shared between the filter and authentication provider, so that tokens created by the former are accepted by the latter. The `userAttribute` is expressed in the form of `usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]`. This is the same syntax as used after the equals sign for `InMemoryDaoImpl`'s `userMap` property.

As explained earlier, the benefit of anonymous authentication is that all URI patterns can have security applied to them. For example:

```
<bean id="filterInvocationInterceptor" class="org.acegisecurity.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
      /index.jsp=ROLE_ANONYMOUS,ROLE_USER
      /hello.htm=ROLE_ANONYMOUS,ROLE_USER
      /logoff.jsp=ROLE_ANONYMOUS,ROLE_USER
```

```
      /acegilogin.jsp*=ROLE_ANONYMOUS,ROLE_USER
      /**=ROLE_USER
    </value>
  </property>
</bean>
```

Rounding out the anonymous authentication discussion is the `AuthenticationTrustResolver` interface, with its corresponding `AuthenticationTrustResolverImpl` implementation. This interface provides an `isAnonymous(Authentication)` method, which allows interested classes to take into account this special type of authentication status. The `ExceptionTranslationFilter` uses this interface in processing `AccessDeniedException`s. If an `AccessDeniedException` is thrown, and the authentication is of an anonymous type, instead of throwing a 403 (forbidden) response, the filter will instead commence the `AuthenticationEntryPoint` so the principal can authenticate properly. This is a necessary distinction, otherwise principals would always be deemed "authenticated" and never be given an opportunity to login via form, basic, digest or some other normal authentication mechanism

Chapter 15. Remember-Me Authentication

15.1. Overview

Remember-me authentication refers to web sites being able to remember the identity of a principal between sessions. This is typically accomplished by sending a cookie to the browser, with the cookie being detected during future sessions and causing automated login to take place. Acegi Security provides the necessary hooks so that such operations can take place, along with providing a concrete implementation that uses hashing to preserve the security of cookie-based tokens.

15.2. Configuration

Remember-me authentication is not used with basic authentication, given it is often not used with `HttpSession`s. Remember-me is used with `AuthenticationProcessingFilter`, and is implemented via hooks in the `AbstractProcessingFilter` superclass. The hooks will invoke a concrete `RememberMeServices` at the appropriate times. The interface looks like this:

```
public Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);
public void loginFail(HttpServletRequest request, HttpServletResponse response);
public void loginSuccess(HttpServletRequest request, HttpServletResponse response, Authentication successfulAuth
```

Please refer to JavaDocs for a fuller discussion on what the methods do, although note at this stage `AbstractProcessingFilter` only calls the `loginFail()` and `loginSuccess()` methods. The `autoLogin()` method is called by `RememberMeProcessingFilter` whenever the `SecurityContextHolder` does not contain an `Authentication`. This interface therefore provides the underlaying remember-me implementation with sufficient notification of authentication-related events, and delegates to the implementation whenever a candidate web request might contain a cookie and wish to be remembered.

This design allows any number of remember-me implementation strategies. In the interests of simplicity and avoiding the need for DAO implementations that specify write and create methods, Acegi Security's only concrete implementation, `TokenBasedRememberMeServices`, uses hashing to achieve a useful remember-me strategy. In essence a cookie is sent to the browser upon successful interactive authentication, with that cookie being composed as follows:

```
base64(username + ":" + expirationTime + ":" + md5Hex(username + ":" + expirationTime + ":" password + ":" + key

username:         As identifiable to TokenBasedRememberMeServices.getUserDetailsService()
password:         That matches the relevant UserDetails retrieved from TokenBasedRememberMeServices.getUserDetai
expirationTime:   The date and time when the remember-me token expires, expressed in milliseconds
key:              A private key to prevent modification of the remember-me token
```

As such the remember-me token is valid only for the period specified, and provided that the username, password and key does not change. Notably, this has a potential security issue in that a captured remember-me token will be usable from any user agent until such time as the token expires. This is the same issue as with digest authentication. If a principal is aware a token has been captured, they can easily change their password and immediately invalidate all remember-me tokens on issue. However, if more significant security is needed a rolling token approach should be used (this would require a database) or remember-me services should simply not be used.

`TokenBasedRememberMeServices` generates a `RememberMeAuthenticationToken`, which is processed by `RememberMeAuthenticationProvider`. A `key` is shared between this authentication provider and the `TokenBasedRememberMeServices`. In addition, `TokenBasedRememberMeServices` requires A UserDetailsService from which it can retrieve the username and password for signature comparison purposes, and generate the `RememberMeAuthenticationToken` to contain the correct `GrantedAuthority`[]s. Some sort of logout command should be provided by the application (typically via a JSP) that invalidates the cookie upon user request. See the Contacts Sample application's `logout.jsp` for an example.

The beans required in an application context to enable remember-me services are as follows:

```
<bean id="rememberMeProcessingFilter" class="org.acegisecurity.ui.rememberme.RememberMeProcessingFilter">
  <property name="rememberMeServices"><ref local="rememberMeServices"/></property>
</bean>

<bean id="rememberMeServices" class="org.acegisecurity.ui.rememberme.TokenBasedRememberMeServices">
  <property name="userDetailsService"><ref local="jdbcDaoImpl"/></property>
  <property name="key"><value>springRocks</value></property>
</bean>

<bean id="rememberMeAuthenticationProvider" class="org.acegisecurity.providers.rememberme.RememberMeAuthenticati
  <property name="key"><value>springRocks</value></property>
</bean>
```

Don't forget to add your `RememberMeServices` implementation to your `AuthenticationProcessingFilter.setRememberMeServices()` property, include the `RememberMeAuthenticationProvider` in your `AuthenticationManager.setProviders()` list, and add a call to `RememberMeProcessingFilter` into your `FilterChainProxy` (typically immediately after your `AuthenticationProcessingFilter`)

Chapter 16. X509 Authentication

16.1. Overview

The most common use of X509 certificate authentication is in verifying the identity of a server when using SSL, most commonly when using HTTPS from a browser. The browser will automatically check that the certificate presented by a server has been issued (ie digitally signed) by one of a list of trusted certificate authorities which it maintains.

You can also use SSL with “mutual authentication”; the server will then request a valid certificate from the client as part of the SSL handshake. The server will authenticate the client by checking that it's certificate is signed by an acceptable authority. If a valid certificate has been provided, it can be obtained through the servlet API in an application. Acegi Security X509 module extracts the certificate using a filter and passes it to the configured X509 authentication provider to allow any additional application-specific checks to be applied. It also maps the certificate to an application user and loads that user's set of granted authorities for use with the standard Acegi Security infrastructure.

You should be familiar with using certificates and setting up client authentication for your servlet container before attempting to use it with Acegi Security. Most of the work is in creating and installing suitable certificates and keys. For example, if you're using Tomcat then read the instructions here http://jakarta.apache.org/tomcat/tomcat-5.0-doc/ssl-howto.html. It's important that you get this working before trying it out with Acegi Security

16.2. Using X509 with Acegi Security

With X509 authentication, there is no explicit login procedure so the implementation is relatively simple; there is no need to redirect requests in order to interact with the user. As a result, some of the classes behave slightly differently from their equivalents in other packages. For example, the default “entry point” class, which is normally responsible for starting the authentication process, is only invoked if the certificate is rejected and it always returns an error to the user. With a suitable bean configuration, the normal sequence of events is as follows

1. The `X509ProcessingFilter` extracts the certificate from the request and uses it as the credentials for an authentication request. The generated authentication request is an `X509AuthenticationToken`. The request is passed to the authentication manager.

2. The `X509AuthenticationProvider` receives the token. Its main concern is to obtain the user information (in particular the user's granted authorities) that matches the certificate. It delegates this responsibility to an `X509AuthoritiesPopulator`.

3. The populator's single method, `getUserDetails(X509Certificate userCertificate)` is invoked. Implementations should return a `UserDetails` instance containing the array of `GrantedAuthority` objects for the user. This method can also choose to reject the certificate (for example if it doesn't contain a matching user name). In such cases it should throw a `BadCredentialsException`. A DAO-based implementation, `DaoX509AuthoritiesPopulator`, is provided which extracts the user's name from the subject “common name” (CN) in the certificate. It also allows you to set your own regular expression to match a different part of the subject's distinguished name. A UserDetailsService is used to load the user information.

4. If everything has gone smoothly then there should be a valid `Authentication` object in the secure context and the invocation will procede as normal. If no certificate was found, or the certificate was rejected, then the `ExceptionTranslationFilter` will invoke the `X509ProcessingFilterEntryPoint` which returns a 403 error (forbidden) to the user.

16.3. Configuration

There is a version of the Contacts Sample Application which uses X509. Copy the beans and filter setup from this as a starting point for configuring your own application. A set of example certificates is also included which you can use to configure your server. These are

- `marissa.p12`: A PKCS12 format file containing the client key and certificate. These should be installed in your browser. It maps to the user “marissa” in the application.
- `server.p12`: The server certificate and key for HTTPS connections.
- `ca.jks`: A Java keystore containing the certificate for the authority which issued marissa's certificate. This will be used by the container to validate client certificates.

For JBoss 3.2.7 (with Tomcat 5.0), the SSL configuration in the `server.xml` file looks like this

```
<!-- SSL/TLS Connector configuration -->
<Connector port="8443" address="${jboss.bind.address}"
  maxThreads="100" minSpareThreads="5" maxSpareThreads="15"
  scheme="https" secure="true"
  sslProtocol = "TLS"
  clientAuth="true" keystoreFile="${jboss.server.home.dir}/conf/server.p12"
  keystoreType="PKCS12" keystorePass="password"
  truststoreFile="${jboss.server.home.dir}/conf/ca.jks"
  truststoreType="JKS" truststorePass="password"
/>
```

`clientAuth` can also be set to *`want`* if you still want SSL connections to succeed even if the client doesn't provide a certificate. Obviously these clients won't be able to access any objects secured by Acegi Security (unless you use a non-X509 authentication mechanism, such as BASIC authentication, to authenticate the user)

Chapter 17. LDAP Authentication

17.1. Overview

LDAP is often used by organizations as a central repository for user information and as an authentication service. It can also be used to store the role information for application users.

There are many different scenarios for how an LDAP server may be configured so Acegi LDAP provider is fully configurable. It uses separate strategy interfaces for authentication and role retrieval and provides default implementations which can be configured to handle a wide range of situations.

You should be familiar with LDAP before trying to use it with Acegi. The following link provides a good introduction to the concepts involved and a guide to setting up a directory using the free LDAP server OpenLDAP: http://www.zytrax.com/books/ldap/. Some familiarity with the JNDI APIs used to access LDAP from Java may also be useful. We don't use any third-party LDAP libraries (Mozilla/Netscape, JLDAP etc.) in the LDAP provider.

17.2. Using LDAP with Acegi Security

The main LDAP provider class is `org.acegisecurity.providers.ldap.LdapAuthenticationProvider`. This bean doesn't actually do much itself other than implement the `retrieveUser` method required by its base class, `AbstractUserDetailsAuthenticationProvider`. It delegates the work to two other beans, an `LdapAuthenticator` and an `LdapAuthoritiesPopulator` which are responsible for authenticating the user and retrieving the user's set of `GrantedAuthority`s respectively.

17.2.1. LdapAuthenticator Implementations

The authenticator is also responsible for retrieving any required user attributes. This is because the permissions on the attributes may depend on the type of authentication being used. For example, if binding as the user, it may be necessary to read them with the user's own permissions.

There are currently two authentication strategies supplied with Acegi Security:

- Authentication directly to the LDAP server ("bind" authentication).
- Password comparison, where the password supplied by the user is compared with the one stored in the repository. This can either be done by retrieving the value of the password attribute and checking it locally or by performing an LDAP "compare" operation, where the supplied password is passed to the server for comparison and the real password value is never retrieved.

17.2.1.1. Common Functionality

Before it is possible to authenticate a user (by either strategy), the distinguished name (DN) has to be obtained from the login name supplied to the application. This can be done either by simple pattern-matching (by setting the setUserDnPatterns array property) or by setting the userSearch property. For the DN pattern-matching approach, a standard Java pattern format is used, and the login name will be substituted for the parameter `{0}`. The pattern should be relative to the DN that the

configured `InitialDirContextFactory` will bind to (see the section on connecting to the LDAP server for more information on this). For example, if you are using an LDAP server specified by the URL `ldap://monkeymachine.co.uk/dc=acegisecurity,dc=org`, and have a pattern `uid={0},ou=greatapes`, then a login name of "gorilla" will map to a DN `uid=gorilla,ou=greatapes,dc=acegisecurity,dc=org`. Each configured DN pattern will be tried in turn until a match is found. For information on using a search, see the section on search objects below. A combination of the two approaches can also be used - the patterns will be checked first and if no matching DN is found, the search will be used.

17.2.1.2. BindAuthenticator

The class `org.acegisecurity.providers.ldap.authenticator.BindAuthenticator` implements the bind authentication strategy. It simply attempts to bind as the user.

17.2.1.3. PasswordComparisonAuthenticator

The class `org.acegisecurity.providers.ldap.authenticator.PasswordComparisonAuthenticator` implements the password comparison authentication strategy.

17.2.1.4. Active Directory Authentication

In addition to standard LDAP authentication (binding with a DN), Active Directory has its own non-standard syntax for user authentication.

17.2.2. Connecting to the LDAP Server

The beans discussed above have to be able to connect to the server. They both have to be supplied with an `InitialDirContextFactory` instance. Unless you have special requirements, this will usually be a `DefaultInitialDirContextFactory` bean, which can be configured with the URL of your LDAP server and optionally with the username and password of a "manager" user which will be used by default when binding to the server (instead of binding anonymously). It currently supports "simple" LDAP authentication.

`DefaultInitialDirContextFactory` uses Sun's JNDI LDAP implementation by default (the one that comes with the JDK). It also supports the built in connection pooling offered by Sun's provider. Connections which are obtained either anonymously or with the "manager" user's identity will be pooled automatically. Connections obtained with a specific user's identity will not be pooled. Connection pooling can be disabled completely by setting the useConnectionPool property to false.

See the class Javadoc and source for more information on this bean and its properties.

17.2.3. LDAP Search Objects

Often more a more complicated strategy than simple DN-matching is required to locate a user entry in the directory. This can be encapsulated in an `LdapUserSearch` instance which can be supplied to the authenticator implementations, for example, to allow them to locate a user. The supplied implementation is `FilterBasedLdapUserSearch`.

17.2.3.1. `FilterBasedLdapUserSearch`

This bean uses an LDAP filter to match the user object in the directory. The process is explained in the Javadoc for the corresponding search method on the JDK DirContext class. As explained there, the

search filter can be supplied with parameters. For this class, the only valid parameter is `{0}` which will be replaced with the user's login name.

17.3. Configuration

There is a version of the Contacts Sample Application which uses LDAP. You can copy the beans and filter setup from this as a starting point for configuring your own application.

A typical configuration, using some of the beans we've discussed above, might look like this:

```
<bean id="initialDirContextFactory"
        class="org.acegisecurity.ldap.DefaultInitialDirContextFactory">
  <constructor-arg value="ldap://monkeymachine:389/dc=acegisecurity,dc=org"/>
  <property name="managerDn"><value>cn=manager,dc=acegisecurity,dc=org</value></property>
  <property name="managerPassword"><value>password</value></property>
</bean>

<bean id="userSearch"
        class="org.acegisecurity.ldap.search.FilterBasedLdapUserSearch">
  <constructor-arg index="0">
    <value></value>
  </constructor-arg>
  <constructor-arg index="1">
    <value>(uid={0})</value>
  </constructor-arg>
  <constructor-arg index="2">
    <ref local="initialDirContextFactory" />
  </constructor-arg>
  <property name="searchSubtree">
    <value>true</value>
  </property>
</bean>

<bean id="ldapAuthProvider"
        class="org.acegisecurity.providers.ldap.LdapAuthenticationProvider">
  <constructor-arg>
    <bean class="org.acegisecurity.providers.ldap.authenticator.BindAuthenticator">
       <constructor-arg><ref local="initialDirContextFactory"/></constructor-arg>
       <property name="userDnPatterns"><list><value>uid={0},ou=people</value></list></property>
    </bean>
  </constructor-arg>
  <constructor-arg>
    <bean class="org.acegisecurity.providers.ldap.populator.DefaultLdapAuthoritiesPopulator">
       <constructor-arg><ref local="initialDirContextFactory"/></constructor-arg>
       <constructor-arg><value>ou=groups</value></constructor-arg>
       <property name="groupRoleAttribute"><value>ou</value></property>
    </bean>
  </constructor-arg>
</bean>
```

This would set up the provider to access an LDAP server with URL `ldap://monkeymachine:389/dc=acegisecurity,dc=org`. Authentication will be performed by attempting to bind with the DN `uid=<user-login-name>,ou=people,dc=acegisecurity,dc=org`. After successful authentication, roles will be assigned to the user by searching under the DN `ou=groups,dc=acegisecurity,dc=org` with the default filter `(member=<user's-DN>)`. The role name will be taken from the “ou” attribute of each match.

We've also included the configuration for a user search object, which uses the filter `(uid=<user-login-name>)`. This could be used instead of the DN-pattern (or in addition to it), by setting the authenticator's userSearch property. The authenticator would then call the search object to obtain the correct user's DN before attempting to bind as this user.

Chapter 18. CAS Authentication

18.1. Overview

JA-SIG produces an enterprise-wide single sign on system known as CAS. Unlike other initiatives, JA-SIG's Central Authentication Service is open source, widely used, simple to understand, platform independent, and supports proxy capabilities. Acegi Security fully supports CAS, and provides an easy migration path from single-application deployments of Acegi Security through to multiple-application deployments secured by an enterprise-wide CAS server.

You can learn more about CAS at `http://www.ja-sig.org/products/cas/`. You will need to visit this URL to download the CAS Server files. Whilst Acegi Security includes two CAS libraries in the "-with-dependencies" ZIP file, you will still need the CAS Java Server Pages and `web.xml` to customise and deploy your CAS server.

18.2. How CAS Works

Whilst the CAS web site above contains two documents that detail the architecture of CAS, we present the general overview again here within the context of Acegi Security. The following refers to both CAS 2.0 (produced by Yale) and CAS 3.0 (produced by JA-SIG), being the versions of CAS that Acegi Security supports.

Somewhere in your enterprise you will need to setup a CAS server. The CAS server is simply a standard WAR file, so there isn't anything difficult about setting up your server. Inside the WAR file you will customise the login and other single sign on pages displayed to users.

If you are deploying CAS 2.0, you will also need to specify in the web.xml a `PasswordHandler`. The `PasswordHandler` has a simple method that returns a boolean as to whether a given username and password is valid. Your `PasswordHandler` implementation will need to link into some type of backend authentication repository, such as an LDAP server or database.

If you are already running an existing CAS 2.0 server instance, you will have already established a `PasswordHandler`. If you do not already have a `PasswordHandler`, you might prefer to use Acegi Security `CasPasswordHandler` class. This class delegates through to the standard Acegi Security `AuthenticationManager`, enabling you to use a security configuration you might already have in place. You do not need to use the `CasPasswordHandler` class on your CAS server if you do not wish. Acegi Security will function as a CAS client successfully irrespective of the `PasswordHandler` you've chosen for your CAS server.

If you are deploying CAS 3.0, you will also need to specify an `AuthenticationHandler` in the deployerConfigContext.xml included with CAS. The `AuthenticationHandler` has a simple method that returns a boolean as to whether a given set of Credentials is valid. Your `AuthenticationHandler` implementation will need to link into some type of backend authentication repository, such as an LDAP server or database. CAS itself includes numerous `AuthenticationHandler`s out of the box to assist with this.

If you are already running an existing CAS 3.0 server instance, you will have already established an `AuthenticationHandler`. If you do not already have an `AuthenticationHandler`, you might prefer to use Acegi Security `CasAuthenticationHandler` class. This class delegates through to the standard Acegi

Security `AuthenticationManager`, enabling you to use a security configuration you might already have in place. You do not need to use the `CasAuthenticationHandler` class on your CAS server if you do not wish. Acegi Security will function as a CAS client successfully irrespective of the `AuthenticationHandler` you've chosen for your CAS server.

Apart from the CAS server itself, the other key player is of course the secure web applications deployed throughout your enterprise. These web applications are known as "services". There are two types of services: standard services and proxy services. A proxy service is able to request resources from other services on behalf of the user. This will be explained more fully later.

Services can be developed in a large variety of languages, due to CAS 2.0's very light XML-based protocol. The JA-SIG CAS home page contains a clients archive which demonstrates CAS clients in Java, Active Server Pages, Perl, Python and others. Naturally, Java support is very strong given the CAS server is written in Java. You do not need to use any of CAS' client classes in applications secured by Acegi Security. This is handled transparently for you.

The basic interaction between a web browser, CAS server and an Acegi Security for System Spring secured service is as follows:

1. The web user is browsing the service's public pages. CAS or Acegi Security is not involved.

2. The user eventually requests a page that is either secure or one of the beans it uses is secure. Acegi Security's `ExceptionTranslationFilter` will detect the `AuthenticationException`.

3. Because the user's `Authentication` object (or lack thereof) caused an `AuthenticationException`, the `ExceptionTranslationFilter` will call the configured `AuthenticationEntryPoint`. If using CAS, this will be the `CasProcessingFilterEntryPoint` class.

4. The `CasProcessingFilterEntry` point will redirect the user's browser to the CAS server. It will also indicate a `service` parameter, which is the callback URL for Acegi Security service. For example, the URL to which the browser is redirected might be `https://my.company.com/cas/login?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Fj_acegi_cas_secu`

5. After the user's browser redirects to CAS, they will be prompted for their username and password. If the user presents a session cookie which indicates they've previously logged on, they will not be prompted to login again (there is an exception to this procedure, which we'll cover later). CAS will use the `PasswordHandler` (or `AuthenticationHandler` if using CAS 3.0) discussed above to decide whether the username and password is valid.

6. Upon successful login, CAS will redirect the user's browser back to the original service. It will also include a `ticket` parameter, which is an opaque string representing the "service ticket". Continuing our earlier example, the URL the browser is redirected to might be `https://server3.company.com/webapp/j_acegi_cas_security_check?ticket=ST-0-ER94xMJmn6pha35CQRoZ`.

7. Back in the service web application, the `CasProcessingFilter` is always listening for requests to `/j_acegi_cas_security_check` (this is configurable, but we'll use the defaults in this introduction). The processing filter will construct a `UsernamePasswordAuthenticationToken` representing the service ticket. The principal will be equal to `CasProcessingFilter.CAS_STATEFUL_IDENTIFIER`, whilst the credentials will be the service ticket opaque value. This authentication request will then be handed to the configured `AuthenticationManager`.

8. The `AuthenticationManager` implementation will be the `ProviderManager`, which is in turn configured with the `CasAuthenticationProvider`. The `CasAuthenticationProvider` only responds to

`UsernamePasswordAuthenticationToken`s containing the CAS-specific principal (such as `CasProcessingFilter.CAS_STATEFUL_IDENTIFIER`) and `CasAuthenticationToken`s (discussed later).

9. `CasAuthenticationProvider` will validate the service ticket using a `TicketValidator` implementation. Acegi Security includes one implementation, the `CasProxyTicketValidator`. This implementation a ticket validation class included in the CAS client library. The `CasProxyTicketValidator` makes an HTTPS request to the CAS server in order to validate the service ticket. The `CasProxyTicketValidator` may also include a proxy callback URL, which is included in this example: `https://my.company.com/cas/proxyValidate?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Fj_acegi`

10. Back on the CAS server, the proxy validation request will be received. If the presented service ticket matches the service URL the ticket was issued to, CAS will provide an affirmative response in XML indicating the username. If any proxy was involved in the authentication (discussed below), the list of proxies is also included in the XML response.

11. [OPTIONAL] If the request to the CAS validation service included the proxy callback URL (in the `pgtUrl` parameter), CAS will include a `pgtIou` string in the XML response. This `pgtIou` represents a proxy-granting ticket IOU. The CAS server will then create its own HTTPS connection back to the `pgtUrl`. This is to mutually authenticate the CAS server and the claimed service URL. The HTTPS connection will be used to send a proxy granting ticket to the original web application. For example, `https://server3.company.com/webapp/casProxy/receptor?pgtIou=PGTIOU-0-R0zlgrl4pdAQwBvJWO3vnNpevwqStbSG`
We suggest you use CAS' `ProxyTicketReceptor` servlet to receive these proxy-granting tickets, if they are required.

12. The `CasProxyTicketValidator` will parse the XML received from the CAS server. It will return to the `CasAuthenticationProvider` a `TicketResponse`, which includes the username (mandatory), proxy list (if any were involved), and proxy-granting ticket IOU (if the proxy callback was requested).

13. Next `CasAuthenticationProvider` will call a configured `CasProxyDecider`. The `CasProxyDecider` indicates whether the proxy list in the `TicketResponse` is acceptable to the service. Several implementations are provided with Acegi Security System: `RejectProxyTickets`, `AcceptAnyCasProxy` and `NamedCasProxyDecider`. These names are largely self-explanatory, except `NamedCasProxyDecider` which allows a `List` of trusted proxies to be provided.

14. `CasAuthenticationProvider` will next request a `CasAuthoritiesPopulator` to advise the `GrantedAuthority` objects that apply to the user contained in the `TicketResponse`. Acegi Security includes a `DaoCasAuthoritiesPopulator` which simply uses the `UserDetailsService` infrastructure to find the `UserDetails` and their associated `GrantedAuthority`s. Note that the password and enabled/disabled status of `UserDetails` returned by the `UserDetailsService` are ignored, as the CAS server is responsible for authentication decisions. `DaoCasAuthoritiesPopulator` is only concerned with retrieving the `GrantedAuthority`s.

15. If there were no problems, `CasAuthenticationProvider` constructs a `CasAuthenticationToken` including the details contained in the `TicketResponse` and the `GrantedAuthority`s. The `CasAuthenticationToken` contains the hash of a key, so that the `CasAuthenticationProvider` knows it created it.

16. Control then returns to `CasProcessingFilter`, which places the created `CasAuthenticationToken` into the `HttpSession` attribute named

`HttpSessionIntegrationFilter.ACEGI_SECURITY_AUTHENTICATION_KEY`.

17. The user's browser is redirected to the original page that caused the `AuthenticationException`.

18. As the `Authentication` object is now in the well-known location, it is handled like any other authentication approach. Usually the `HttpSessionIntegrationFilter` will be used to associate the `Authentication` object with the `SecurityContextHolder` for the duration of each request.

It's good that you're still here! It might sound involved, but you can relax as Acegi Security classes hide much of the complexity. Let's now look at how this is configured

18.3. Optional CAS Server Setup

Acegi Security can even act as the backend which a CAS version 2.0 or 3.0 server utilises. The configuration approach is described below. Of course, if you have an existing CAS environment you might just like to use it instead.

18.3.1. CAS Version 2.0

As mentioned above, Acegi Security includes a `PasswordHandler` that bridges your existing `AuthenticationManager` into CAS 2.0. You do not need to use this `PasswordHandler` to use Acegi Security on the client side (any CAS `PasswordHandler` will do).

To install, you will need to download and extract the CAS server archive. We used version 2.0.12. There will be a `/web` directory in the root of the deployment. Copy an `applicationContext.xml` containing your `AuthenticationManager` as well as the `CasPasswordHandler` into the `/web/WEB-INF` directory. A sample `applicationContext.xml` is included below:

```
<bean id="inMemoryDaoImpl" class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      marissa=koala,ROLES_IGNORED_BY_CAS
      dianne=emu,ROLES_IGNORED_BY_CAS
      scott=wombat,ROLES_IGNORED_BY_CAS
      peter=opal,disabled,ROLES_IGNORED_BY_CAS
    </value>
  </property>
</bean>

<bean id="daoAuthenticationProvider" class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService"><ref bean="inMemoryDaoImpl"/></property>
</bean>

<bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref bean="daoAuthenticationProvider"/>
    </list>
  </property>
</bean>

<bean id="casPasswordHandler" class="org.acegisecurity.adapters.cas.CasPasswordHandler">
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
</bean>
```

Note the granted authorities are ignored by CAS because it has no way of communicating the granted authorities to calling applications. CAS is only concerned with username and passwords (and the

enabled/disabled status).

Next you will need to edit the existing `/web/WEB-INF/web.xml` file. Add (or edit in the case of the `authHandler` property) the following lines:

```
<context-param>
  <param-name>edu.yale.its.tp.cas.authHandler</param-name>
  <param-value>org.acegisecurity.adapters.cas.CasPasswordHandlerProxy</param-value>
</context-param>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Copy the `spring.jar` and `acegi-security.jar` files into `/web/WEB-INF/lib`. Now use the `ant dist` task in the `build.xml` in the root of the directory structure. This will create `/lib/cas.war`, which is ready for deployment to your servlet container.

Note CAS heavily relies on HTTPS. You can't even test the system without an HTTPS certificate. Whilst you should refer to your web container's documentation on setting up HTTPS, if you need some additional help or a test certificate you might like to check the `samples/contacts/etc/ssl` directory

18.3.2. CAS Version 3.0

As mentioned above, Acegi Security includes an `AuthenticationHandler` that bridges your existing `AuthenticationManager` into CAS 3.0. You do not need to use this `AuthenticationHandler` to use Acegi Security on the client side (any CAS `AuthenticationHandler` will do).

To install, you will need to download and extract the CAS server archive. We used version 3.0.4. There will be a `/webapp` directory in the root of the deployment. Edit the an `deployerConfigContext.xml` so that it contains your `AuthenticationManager` as well as the `CasAuthenticationHandler`. A sample `applicationContext.xml` is included below:

```
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
	<beans>
		<bean
			id="authenticationManager"
			class="org.jasig.cas.authentication.AuthenticationManagerImpl">
			<property name="credentialsToPrincipalResolvers">
				<list>
					<bean class="org.jasig.cas.authentication.principal.UsernamePasswordCred
					<bean class="org.jasig.cas.authentication.principal.HttpBasedServiceCred
				</list>
			</property>

			<property name="authenticationHandlers">
				<list>
					<bean class="org.jasig.cas.authentication.handler.support.HttpBasedServi
					<bean class="org.acegisecurity.adapters.cas3.CasAuthenticationHandler">
						<property name="authenticationManager" ref="acegiAuthenticationM
					</bean>
				</list>
			</property>
		</bean>
```

```
        <bean id="inMemoryDaoImpl" class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
                  <property name="userMap">
                        <value>
                                marissa=koala,ROLES_IGNORED_BY_CAS
                                dianne=emu,ROLES_IGNORED_BY_CAS
                                scott=wombat,ROLES_IGNORED_BY_CAS
                                peter=opal,disabled,ROLES_IGNORED_BY_CAS
                        </value>
                </property>
        </bean>

        <bean id="daoAuthenticationProvider" class="org.acegisecurity.providers.dao.DaoAuthenticationPro
             <property name="userDetailsService"><ref bean="inMemoryDaoImpl"/></property>
        </bean>

        <bean id="acegiAuthenticationManager" class="org.acegisecurity.providers.ProviderManager">
                <property name="providers">
                  <list>
                    <ref bean="daoAuthenticationProvider"/>
                  </list>
                </property>
        </bean>
</beans>
```

Note the granted authorities are ignored by CAS because it has no way of communicating the granted authorities to calling applications. CAS is only concerned with username and passwords (and the enabled/disabled status).

Copy `acegi-security.jar` and `acegi-security-cas.jar` files into `/localPlugins/lib`. Now use the `ant war` task in the `build.xml` in the /localPlugins directory. This will create `/localPlugins/target/cas.war`, which is ready for deployment to your servlet container.

Note CAS heavily relies on HTTPS. You can't even test the system without an HTTPS certificate. Whilst you should refer to your web container's documentation on setting up HTTPS, if you need some additional help or a test certificate you might like to check the CAS documentation on setting up SSL: `http://www.ja-sig.org/products/cas/server/ssl/index.html`

18.4. Configuration of CAS Client

The web application side of CAS is made easy due to Acegi Security. It is assumed you already know the basics of using Acegi Security, so these are not covered again below. Only the CAS-specific beans are mentioned.

You will need to add a `ServiceProperties` bean to your application context. This represents your service:

```
<bean id="serviceProperties" class="org.acegisecurity.ui.cas.ServiceProperties">
  <property name="service"><value>https://localhost:8443/contacts-cas/j_acegi_cas_security_check</value></proper
  <property name="sendRenew"><value>false</value></property>
</bean>
```

The `service` must equal a URL that will be monitored by the `CasProcessingFilter`. The `sendRenew` defaults to false, but should be set to true if your application is particularly sensitive. What this parameter does is tell the CAS login service that a single sign on login is unacceptable. Instead, the

user will need to re-enter their username and password in order to gain access to the service.

The following beans should be configured to commence the CAS authentication process:

```
<bean id="casProcessingFilter" class="org.acegisecurity.ui.cas.CasProcessingFilter">
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="authenticationFailureUrl"><value>/casfailed.jsp</value></property>
  <property name="defaultTargetUrl"><value>/</value></property>
  <property name="filterProcessesUrl"><value>/j_acegi_cas_security_check</value></property>
</bean>

<bean id="exceptionTranslationFilter" class="org.acegisecurity.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint"><ref local="casProcessingFilterEntryPoint"/></property>
</bean>

<bean id="casProcessingFilterEntryPoint" class="org.acegisecurity.ui.cas.CasProcessingFilterEntryPoint">
  <property name="loginUrl"><value>https://localhost:8443/cas/login</value></property>
  <property name="serviceProperties"><ref bean="serviceProperties"/></property>
</bean>
```

You will also need to add the `CasProcessingFilter` to web.xml:

```
<filter>
  <filter-name>Acegi CAS Processing Filter</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.ui.cas.CasProcessingFilter</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Acegi CAS Processing Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The `CasProcessingFilter` has very similar properties to the `AuthenticationProcessingFilter` (used for form-based logins). Each property is self-explanatory.

For CAS to operate, the `ExceptionTranslationFilter` must have its `authenticationEntryPoint` property set to the `CasProcessingFilterEntryPoint` bean.

The `CasProcessingFilterEntryPoint` must refer to the `ServiceProperties` bean (discussed above), which provides the URL to the enterprise's CAS login server. This is where the user's browser will be redirected.

Next you need to add an `AuthenticationManager` that uses `CasAuthenticationProvider` and its collaborators:

```
<bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref bean="casAuthenticationProvider"/>
    </list>
  </property>
</bean>

<bean id="casAuthenticationProvider" class="org.acegisecurity.providers.cas.CasAuthenticationProvider">
  <property name="casAuthoritiesPopulator"><ref bean="casAuthoritiesPopulator"/></property>
  <property name="casProxyDecider"><ref bean="casProxyDecider"/></property>
  <property name="ticketValidator"><ref bean="casProxyTicketValidator"/></property>
  <property name="statelessTicketCache"><ref bean="statelessTicketCache"/></property>
```

```
  <property name="key"><value>my_password_for_this_auth_provider_only</value></property>
</bean>

<bean id="casProxyTicketValidator" class="org.acegisecurity.providers.cas.ticketvalidator.CasProxyTicketValidato
  <property name="casValidate"><value>https://localhost:8443/cas/proxyValidate</value></property>
  <property name="proxyCallbackUrl"><value>https://localhost:8443/contacts-cas/casProxy/receptor</value></proper
  <property name="serviceProperties"><ref bean="serviceProperties"/></property>
  <!-- <property name="trustStore"><value>/some/path/to/your/lib/security/cacerts</value></property> -->
</bean>

<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
  <property name="configLocation">
    <value>classpath:/ehcache-failsafe.xml</value>
  </property>
</bean>

<bean id="ticketCacheBackend" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <property name="cacheManager">
    <ref local="cacheManager"/>
  </property>
  <property name="cacheName">
    <value>ticketCache</value>
  </property>
</bean>

<bean id="statelessTicketCache" class="org.acegisecurity.providers.cas.cache.EhCacheBasedTicketCache">
  <property name="cache"><ref local="ticketCacheBackend"/></property>
</bean>

<bean id="casAuthoritiesPopulator" class="org.acegisecurity.providers.cas.populator.DaoCasAuthoritiesPopulator">
  <property name="userDetailsService"><ref bean="inMemoryDaoImpl"/></property>
</bean>

<bean id="casProxyDecider" class="org.acegisecurity.providers.cas.proxy.RejectProxyTickets"/>
```

The beans are all reasonable self-explanatory if you refer back to the "How CAS Works" section. Careful readers might notice one surprise: the `statelessTicketCache` property of the `CasAuthenticationProvider`. This is discussed in detail in the "Advanced CAS Usage" section.

Note the `CasProxyTicketValidator` has a remarked out `trustStore` property. This property might be helpful if you experience HTTPS certificate issues. Also note the `proxyCallbackUrl` is set so the service can receive a proxy-granting ticket. As mentioned above, this is optional and unnecessary if you do not require proxy-granting tickets. If you do use this feature, you will need to configure a suitable servlet to receive the proxy-granting tickets. We suggest you use CAS' `ProxyTicketReceptor` by adding the following to your web application's `web.xml`:

```
<servlet>
  <servlet-name>casproxy</servlet-name>
  <servlet-class>edu.yale.its.tp.cas.proxy.ProxyTicketReceptor</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>casproxy</servlet-name>
  <url-pattern>/casProxy/*</url-pattern>
</servlet-mapping>
```

This completes the configuration of CAS. If you haven't made any mistakes, your web application should happily work within the framework of CAS single sign on. No other parts of Acegi Security need to be concerned about the fact CAS handled authentication.

There is also a `contacts-cas.war` file in the sample applications directory. This sample application uses the above settings and can be deployed to see CAS in operation

18.5. Advanced Issues

The `CasAuthenticationProvider` distinguishes between stateful and stateless clients. A stateful client is considered any that originates via the `CasProcessingFilter`. A stateless client is any that presents an authentication request via the `UsernamePasswordAuthenticationToken` with a principal equal to `CasProcessingFilter.CAS_STATELESS_IDENTIFIER`.

Stateless clients are likely to be via remoting protocols such as Hessian and Burlap. The `BasicProcessingFilter` is still used in this case, but the remoting protocol client is expected to present a username equal to the static string above, and a password equal to a CAS service ticket. Clients should acquire a CAS service ticket directly from the CAS server.

Because remoting protocols have no way of presenting themselves within the context of a `HttpSession`, it isn't possible to rely on the `HttpSession`'s `HttpSessionIntegrationFilter.ACEGI_SECURITY_AUTHENTICATION_KEY` attribute to locate the `CasAuthenticationToken`. Furthermore, because the CAS server invalidates a service ticket after it has been validated by the `TicketValidator`, presenting the same service ticket on subsequent requests will not work. It is similarly very difficult to obtain a proxy-granting ticket for a remoting protocol client, as they are often deployed on client machines which rarely have HTTPS URLs that would be accessible to the CAS server.

One obvious option is to not use CAS at all for remoting protocol clients. However, this would eliminate many of the desirable features of CAS.

As a middle-ground, the `CasAuthenticationProvider` uses a `StatelessTicketCache`. This is used solely for requests with a principal equal to `CasProcessingFilter.CAS_STATELESS_IDENTIFIER`. What happens is the `CasAuthenticationProvider` will store the resulting `CasAuthenticationToken` in the `StatelessTicketCache`, keyed on the service ticket. Accordingly, remoting protocol clients can present the same service ticket and the `CasAuthenticationProvider` will not need to contact the CAS server for validation (aside from the first request).

The other aspect of advanced CAS usage involves creating proxy tickets from the proxy-granting ticket. As indicated above, we recommend you use CAS' `ProxyTicketReceptor` to receive these tickets. The `ProxyTicketReceptor` provides a static method that enables you to obtain a proxy ticket by presenting the proxy-granting IOU ticket. You can obtain the proxy-granting IOU ticket by calling `CasAuthenticationToken.getProxyGrantingTicketIou()`.

It is hoped you find CAS integration easy and useful with Acegi Security classes. Welcome to enterprise-wide single sign on!

Chapter 19. Container Adapter Authentication

19.1. Overview

Very early versions of Acegi Security exclusively used Container Adapters for interfacing authentication with end users. Whilst this worked well, it required considerable time to support multiple container versions and the configuration itself was relatively time-consuming for developers. For this reason the HTTP Form Authentication and HTTP Basic Authentication approaches were developed, and are today recommended for almost all applications.

Container Adapters enable Acegi Security to integrate directly with the containers used to host end user applications. This integration means that applications can continue to leverage the authentication and authorization capabilities built into containers (such as `isUserInRole()` and form-based or basic authentication), whilst benefiting from the enhanced security interception capabilities provided by Acegi Security (it should be noted that Acegi Security also offers `ContextHolderAwareRequestWrapper` to deliver `isUserInRole()` and similar Servlet Specification compatibility methods).

The integration between a container and Acegi Security is achieved through an adapter. The adapter provides a container-compatible user authentication provider, and needs to return a container-compatible user object.

The adapter is instantiated by the container and is defined in a container-specific configuration file. The adapter then loads a Spring application context which defines the normal authentication manager settings, such as the authentication providers that can be used to authenticate the request. The application context is usually named `acegisecurity.xml` and is placed in a container-specific location.

Acegi Security currently supports Jetty, Catalina (Tomcat), JBoss and Resin. Additional container adapters can easily be written

19.2. Adapter Authentication Provider

As is always the case, the container adapter generated `Authentication` object still needs to be authenticated by an `AuthenticationManager` when requested to do so by the `AbstractSecurityInterceptor`. The `AuthenticationManager` needs to be certain the adapter-provided `Authentication` object is valid and was actually authenticated by a trusted adapter.

Adapters create `Authentication` objects which are immutable and implement the `AuthByAdapter` interface. These objects store the hash of a key that is defined by the adapter. This allows the `Authentication` object to be validated by the `AuthByAdapterProvider`. This authentication provider is defined as follows:

```
<bean id="authByAdapterProvider" class="org.acegisecurity.adapters.AuthByAdapterProvider">
  <property name="key"><value>my_password</value></property>
</bean>
```

The key must match the key that is defined in the container-specific configuration file that starts the adapter. The `AuthByAdapterProvider` automatically accepts as valid any `AuthByAdapter` implementation that returns the expected hash of the key.

To reiterate, this means the adapter will perform the initial authentication using providers such as `DaoAuthenticationProvider`, returning an `AuthByAdapter` instance that contains a hash code of the key. Later, when an application calls a security interceptor managed resource, the `AuthByAdapter` instance in the `SecurityContext` in the `SecurityContextHolder` will be tested by the application's `AuthByAdapterProvider`. There is no requirement for additional authentication providers such as `DaoAuthenticationProvider` within the application-specific application context, as the only type of `Authentication` instance that will be presented by the application is from the container adapter.

Classloader issues are frequent with containers and the use of container adapters illustrates this further. Each container requires a very specific configuration. The installation instructions are provided below. Once installed, please take the time to try the sample application to ensure your container adapter is properly configured.

When using container adapters with the `DaoAuthenticationProvider`, ensure you set its `forcePrincipalAsString` property to `true`.

19.3. Jetty

The following was tested with Jetty 4.2.18.

`$JETTY_HOME` refers to the root of your Jetty installation.

Edit your `$JETTY_HOME/etc/jetty.xml` file so the `<Configure class>` section has a new `addRealm` call:

```
<Call name="addRealm">
  <Arg>
    <New class="org.acegisecurity.adapters.jetty.JettyAcegiUserRealm">
      <Arg>Spring Powered Realm</Arg>
      <Arg>my_password</Arg>
      <Arg>etc/acegisecurity.xml</Arg>
    </New>
  </Arg>
</Call>
```

Copy `acegisecurity.xml` into `$JETTY_HOME/etc`.

Copy the following files into `$JETTY_HOME/ext`:

- `aopalliance.jar`
- `commons-logging.jar`
- `spring.jar`
- `acegi-security-jetty-XX.jar`
- `commons-codec.jar`
- `burlap.jar`
- `hessian.jar`

None of the above JAR files (or `acegi-security-XX.jar`) should be in your application's `WEB-INF/lib`. The realm name indicated in your `web.xml` does matter with Jetty. The `web.xml` must express the same `<realm-name>` as your `jetty.xml` (in the example above, "Spring Powered Realm").

19.4. JBoss

The following was tested with JBoss 3.2.6.

`$JBOSS_HOME` refers to the root of your JBoss installation.

There are two different ways of making spring context available to the Jboss integration classes.

The first approach is by editing your `$JBOSS_HOME/server/your_config/conf/login-config.xml` file so that it contains a new entry under the `<Policy>` section:

```
<application-policy name = "SpringPoweredRealm">
   <authentication>
      <login-module code = "org.acegisecurity.adapters.jboss.JbossAcegiLoginModule"
        flag = "required">
        <module-option name = "appContextLocation">acegisecurity.xml</module-option>
        <module-option name = "key">my_password</module-option>
     </login-module>
   </authentication>
</application-policy>
```

Copy `acegisecurity.xml` into `$JBOSS_HOME/server/your_config/conf`.

In this configuration `acegisecurity.xml` contains the spring context definition including all the authentication manager beans. You have to bear in mind though, that `SecurityContext` is created and destroyed on each login request, so the login operation might become costly. Alternatively, the second approach is to use Spring singleton capabilities through `org.springframework.beans.factory.access.SingletonBeanFactoryLocator`. The required configuration for this approach is:

```
<application-policy name = "SpringPoweredRealm">
   <authentication>
      <login-module code = "org.acegisecurity.adapters.jboss.JbossAcegiLoginModule"
        flag = "required">
        <module-option name = "singletonId">springRealm</module-option>
        <module-option name = "key">my_password</module-option>
        <module-option name = "authenticationManager">authenticationManager</module-option>
     </login-module>
   </authentication>
</application-policy>
```

In the above code fragment, `authenticationManager` is a helper property that defines the expected name of the `AuthenticationManager` in case you have several defined in the IoC container. The `singletonId` property references a bean defined in a `beanRefFactory.xml` file. This file needs to be available from anywhere on the JBoss classpath, including `$JBOSS_HOME/server/your_config/conf`. The `beanRefFactory.xml` contains the following declaration:

```
```

```
<beans>
  <bean id="springRealm" singleton="true" lazy-init="true" class="org.springframework.context.support.ClassPathX
    <constructor-arg>
      <list>
        <value>acegisecurity.xml</value>
      </list>
    </constructor-arg>
  </bean>
</beans>
```

Finally, irrespective of the configuration approach you need to copy the following files into `$JBOSS_HOME/server/your_config/lib`:

- `aopalliance.jar`
- `spring.jar`
- `acegi-security-jboss-XX.jar`
- `commons-codec.jar`
- `burlap.jar`
- `hessian.jar`

None of the above JAR files (or `acegi-security-XX.jar`) should be in your application's `WEB-INF/lib`. The realm name indicated in your `web.xml` does not matter with JBoss. However, your web application's `WEB-INF/jboss-web.xml` must express the same `<security-domain>` as your `login-config.xml`. For example, to match the above example, your `jboss-web.xml` would look like this:

```
<jboss-web>
  <security-domain>java:/jaas/SpringPoweredRealm</security-domain>
</jboss-web>
```

JBoss is a widely-used container adapter (mostly due to the need to support legacy EJBs), so please let us know if you have any difficulties.

19.5. Resin

The following was tested with Resin 3.0.6.

`$RESIN_HOME` refers to the root of your Resin installation.

Resin provides several ways to support the container adapter. In the instructions below we have elected to maximise consistency with other container adapter configurations. This will allow Resin users to simply deploy the sample application and confirm correct configuration. Developers comfortable with Resin are naturally able to use its capabilities to package the JARs with the web application itself, and/or support single sign-on.

Copy the following files into `$RESIN_HOME/lib`:

- `aopalliance.jar`

- commons-logging.jar
- spring.jar
- acegi-security-resin-XX.jar
- commons-codec.jar
- burlap.jar
- hessian.jar

Unlike the container-wide acegisecurity.xml files used by other container adapters, each Resin web application will contain its own WEB-INF/resin-acegisecurity.xml file. Each web application will also contain a resin-web.xml file which Resin uses to start the container adapter:

```
<web-app>
  <authenticator>
    <type>org.acegisecurity.adapters.resin.ResinAcegiAuthenticator</type>
    <init>
      <app-context-location>WEB-INF/resin-acegisecurity.xml</app-context-location>
      <key>my_password</key>
    </init>
  </authenticator>
</web-app>
```

With the basic configuration provided above, none of the JAR files listed (or acegi-security-XX.jar) should be in your application's WEB-INF/lib. The realm name indicated in your web.xml does not matter with Resin, as the relevant authentication class is indicated by the <authenticator> setting

19.6. Tomcat

The following was tested with Jakarta Tomcat 4.1.30 and 5.0.19.

$CATALINA_HOME refers to the root of your Catalina (Tomcat) installation.

Edit your $CATALINA_HOME/conf/server.xml file so the <Engine> section contains only one active <Realm> entry. An example realm entry:

```
      <Realm className="org.acegisecurity.adapters.catalina.CatalinaAcegiUserRealm"
             appContextLocation="conf/acegisecurity.xml"
             key="my_password" />
```

Be sure to remove any other <Realm> entry from your <Engine> section.

Copy acegisecurity.xml into $CATALINA_HOME/conf.

Copy acegi-security-catalina-XX.jar into $CATALINA_HOME/server/lib.

Copy the following files into $CATALINA_HOME/common/lib:

- aopalliance.jar

- `spring.jar`
- `commons-codec.jar`
- `burlap.jar`
- `hessian.jar`

None of the above JAR files (or `acegi-security-XX.jar`) should be in your application's `WEB-INF/lib`. The realm name indicated in your `web.xml` does not matter with Catalina.

We have received reports of problems using this Container Adapter with Mac OS X. A work-around is to use a script such as follows:

```
#!/bin/sh
export CATALINA_HOME="/Library/Tomcat"
export JAVA_HOME="/Library/Java/Home"
cd /
$CATALINA_HOME/bin/startup.sh
```

Finally, restart Tomcat.

Part III. Authorization

The advanced authorization capabilities within Acegi Security represent one of the most compelling reasons for its popularity. Irrespective of how you choose to authenticate - whether using an Acegi Security-provided mechanism and provider, or integrating with a container or other non-Acegi Security authentication authority - you will find the authorization services can be used within your application in a consistent and simple way.

In this part we'll explore the different `AbstractSecurityInterceptor` implementations, which were introduced in Part I. We then move on to explore how to fine-tune authorization through use of domain access control lists.

Chapter 20. Common Authorization Concepts

20.1. Authorities

As briefly mentioned in the Authentication section, all `Authentication` implementations are required to store an array of `GrantedAuthority` objects. These represent the authorities that have been granted to the principal. The `GrantedAuthority` objects are inserted into the `Authentication` object by the `AuthenticationManager` and are later read by `AccessDecisionManagers` when making authorization decisions.

`GrantedAuthority` is an interface with only one method:

```
public String getAuthority();
```

This method allows `AccessDecisionManagers` to obtain a precise `String` representation of the `GrantedAuthority`. By returning a representation as a `String`, a `GrantedAuthority` can be easily "read" by most `AccessDecisionManagers`. If a `GrantedAuthority` cannot be precisely represented as a `String`, the `GrantedAuthority` is considered "complex" and `getAuthority()` must return `null`.

An example of a "complex" `GrantedAuthority` would be an implementation that stores a list of operations and authority thresholds that apply to different customer account numbers. Representing this complex `GrantedAuthority` as a `String` would be quite complex, and as a result the `getAuthority()` method should return `null`. This will indicate to any `AccessDecisionManager` that it will need to specifically support the `GrantedAuthority` implementation in order to understand its contents.

Acegi Security includes one concrete `GrantedAuthority` implementation, `GrantedAuthorityImpl`. This allows any user-specified `String` to be converted into a `GrantedAuthority`. All `AuthenticationProviders` included with the security architecture use `GrantedAuthorityImpl` to populate the `Authentication` object.

20.2. Pre-Invocation Handling

The `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` and is responsible for making final access control decisions. The `AccessDecisionManager` interface contains three methods:

```
public void decide(Authentication authentication, Object object, ConfigAttributeDefinition config) throws Access
public boolean supports(ConfigAttribute attribute);
public boolean supports(Class clazz);
```

As can be seen from the first method, the `AccessDecisionManager` is passed via method parameters all information that is likely to be of value in assessing an authorization decision. In particular, passing the secure `Object` enables those arguments contained in the actual secure object invocation to be inspected. For example, let's assume the secure object was a `MethodInvocation`. It would be easy to query the `MethodInvocation` for any `Customer` argument, and then implement some sort of security logic in the `AccessDecisionManager` to ensure the principal is permitted to operate on that customer. Implementations are expected to throw an `AccessDeniedException` if access is denied.

The `supports(ConfigAttribute)` method is called by the `AbstractSecurityInterceptor` at startup time to

determine if the `AccessDecisionManager` can process the passed `ConfigAttribute`. The `supports(Class)` method is called by a security interceptor implementation to ensure the configured `AccessDecisionManager` supports the type of secure object that the security interceptor will present.

Whilst users can implement their own `AccessDecisionManager` to control all aspects of authorization, Acegi Security includes several `AccessDecisionManager` implementations that are based on voting. Figure 4 illustrates the relevant classes.

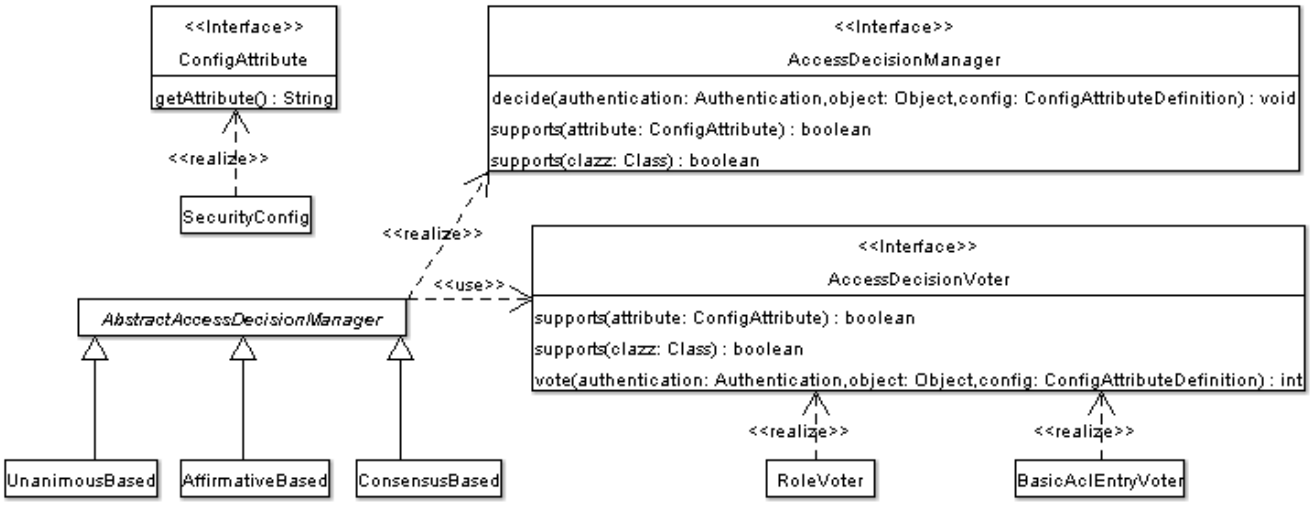

Figure 4: Voting Decision Manager

Using this approach, a series of `AccessDecisionVoter` implementations are polled on an authorization decision. The `AccessDecisionManager` then decides whether or not to throw an `AccessDeniedException` based on its assessment of the votes.

The `AccessDecisionVoter` interface has three methods:

```
public int vote(Authentication authentication, Object object, ConfigAttributeDefinition config);
public boolean supports(ConfigAttribute attribute);
public boolean supports(Class clazz);
```

Concrete implementations return an `int`, with possible values being reflected in the `AccessDecisionVoter` static fields `ACCESS_ABSTAIN`, `ACCESS_DENIED` and `ACCESS_GRANTED`. A voting implementation will return `ACCESS_ABSTAIN` if it has no opinion on an authorization decision. If it does have an opinion, it must return either `ACCESS_DENIED` or `ACCESS_GRANTED`.

There are three concrete `AccessDecisionManager`s provided with Acegi Security that tally the votes. The `ConsensusBased` implementation will grant or deny access based on the consensus of non-abstain votes. Properties are provided to control behavior in the event of an equality of votes or if all votes are abstain. The `AffirmativeBased` implementation will grant access if one or more `ACCESS_GRANTED` votes were received (ie a deny vote will be ignored, provided there was at least one grant vote). Like the `ConsensusBased` implementation, there is a parameter that controls the behavior if all voters abstain. The `UnanimousBased` provider expects unanimous `ACCESS_GRANTED` votes in order to grant access, ignoring abstains. It will deny access if there is any `ACCESS_DENIED` vote. Like the other implementations, there is a parameter that controls the behaviour if all voters abstain.

It is possible to implement a custom `AccessDecisionManager` that tallies votes differently. For example, votes from a particular `AccessDecisionVoter` might receive additional weighting, whilst a deny vote from a particular voter may have a veto effect.

There are two concrete `AccessDecisionVoter` implementations provided with Acegi Security. The `RoleVoter` class will vote if any ConfigAttribute begins with `ROLE_`. It will vote to grant access if there is a `GrantedAuthority` which returns a `String` representation (via the `getAuthority()` method) exactly equal to one or more `ConfigAttributes` starting with `ROLE_`. If there is no exact match of any `ConfigAttribute` starting with `ROLE_`, the `RoleVoter` will vote to deny access. If no `ConfigAttribute` begins with `ROLE_`, the voter will abstain. `RoleVoter` is case sensitive on comparisons as well as the `ROLE_` prefix.

`BasicAclEntryVoter` is the other concrete voter included with Acegi Security. It integrates with Acegi Security's `AclManager` (discussed later). This voter is designed to have multiple instances in the same application context, such as:

```
<bean id="aclContactReadVoter" class="org.acegisecurity.vote.BasicAclEntryVoter">
  <property name="processConfigAttribute"><value>ACL_CONTACT_READ</value></property>
  <property name="processDomainObjectClass"><value>sample.contact.Contact</value></property>
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="org.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="org.acegisecurity.acl.basic.SimpleAclEntry.READ"/>
    </list>
  </property>
</bean>

<bean id="aclContactDeleteVoter" class="org.acegisecurity.vote.BasicAclEntryVoter">
  <property name="processConfigAttribute"><value>ACL_CONTACT_DELETE</value></property>
  <property name="processDomainObjectClass"><value>sample.contact.Contact</value></property>
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="org.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="org.acegisecurity.acl.basic.SimpleAclEntry.DELETE"/>
    </list>
  </property>
</bean>
```

In the above example, you'd define `ACL_CONTACT_READ` or `ACL_CONTACT_DELETE` against some methods on a `MethodSecurityInterceptor` or `AspectJSecurityInterceptor`. When those methods are invoked, the above applicable voter defined above would vote to grant or deny access. The voter would look at the method invocation to locate the first argument of type `sample.contact.Contact`, and then pass that `Contact` to the `AclManager`. The `AclManager` will then return an access control list (ACL) that applies to the current `Authentication`. Assuming that ACL contains one of the listed `requirePermissions`, the voter will vote to grant access. If the ACL does not contain one of the permissions defined against the voter, the voter will vote to deny access. `BasicAclEntryVoter` is an important class as it allows you to build truly complex applications with domain object security entirely defined in the application context. If you're interested in learning more about Acegi Security's ACL capabilities and how best to apply them, please see the ACL and "After Invocation" sections of this reference guide, and the Contacts sample application.

It is also possible to implement a custom `AccessDecisionVoter`. Several examples are provided in Acegi Security unit tests, including `ContactSecurityVoter` and `DenyVoter`. The `ContactSecurityVoter` abstains from voting decisions where a `CONTACT_OWNED_BY_CURRENT_USER ConfigAttribute` is not found. If voting, it queries the `MethodInvocation` to extract the owner of the `Contact` object that is subject of the method call. It votes to grant access if the `Contact` owner matches the principal presented in the `Authentication` object. It could have just as easily compared the `Contact` owner with some `GrantedAuthority` the `Authentication` object presented. All of this is achieved with relatively few lines of code and demonstrates the flexibility of the authorization model.

TODO: Remove references to the old ACL package when it's deprecated, and have all references to

the replacement package limited to the chapter describing the new ACL implementation.

20.3. After Invocation Handling

Whilst the `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` before proceeding with the secure object invocation, some applications need a way of modifying the object actually returned by the secure object invocation. Whilst you could easily implement your own AOP concern to achieve this, Acegi Security provides a convenient hook that has several concrete implementations that integrate with its ACL capabilities.

Figure 5 illustrates Acegi Security's `AfterInvocationManager` and its concrete implementations.

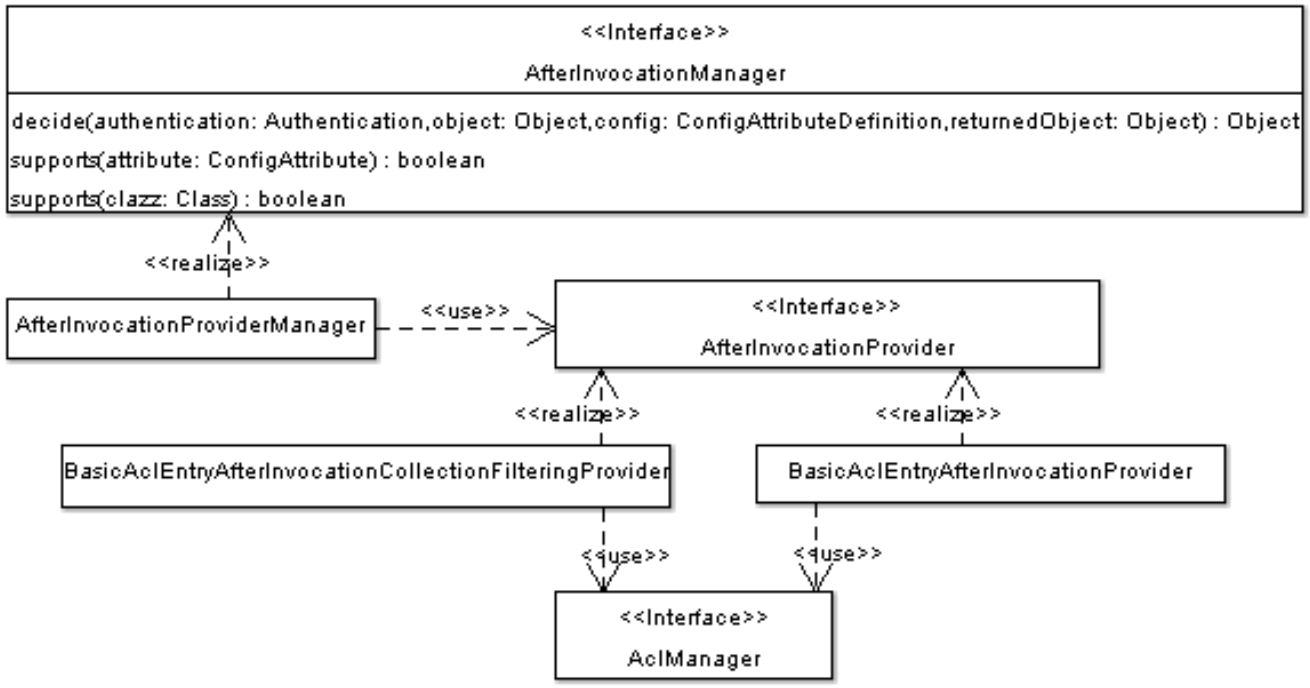

Figure 5: After Invocation Implementation

Like many other parts of Acegi Security, `AfterInvocationManager` has a single concrete implementation, `AfterInvocationProviderManager`, which polls a list of `AfterInvocationProvider`s. Each `AfterInvocationProvider` is allowed to modify the return object or throw an `AccessDeniedException`. Indeed multiple providers can modify the object, as the result of the previous provider is passed to the next in the list. Let's now consider our ACL-aware implementations of `AfterInvocationProvider`.

Please be aware that if you're using `AfterInvocationManager`, you will still need configuration attributes that allow the `MethodSecurityInterceptor`'s `AccessDecisionManager` to allow an operation. If you're using the typical Acegi Security included `AccessDecisionManager` implementations, having no configuration attributes defined for a particular secure method invocation will cause each `AccessDecisionVoter` to abstain from voting. In turn, if the `AccessDecisionManager` property "`allowIfAllAbstainDecisions`" is `false`, an `AccessDeniedException` will be thrown. You may avoid this potential issue by either (i) setting "`allowIfAllAbstainDecisions`" to `true` (although this is generally not recommended) or (ii) simply ensure that there is at least one configuration attribute that an `AccessDecisionVoter` will vote to grant access for. This latter (recommended) approach is usually achieved through a `ROLE_USER` or `ROLE_AUTHENTICATED` configuration attribute

20.3.1. ACL-Aware AfterInvocationProviders

PLEASE NOTE: Acegi Security 1.0.3 contains a preview of a new ACL module. The new ACL module is a significant rewrite of the existing ACL module. The new module can be found under the `org.acegisecurity.acls` package, with the old ACL module under `org.acegisecurity.acl`. We encourage users to consider testing with the new ACL module and build applications with it. The old ACL module should be considered deprecated and may be removed from a future release. The following information relates to the new ACL package, and is thus recommended.

A common services layer method we've all written at one stage or another looks like this:

```
public Contact getById(Integer id);
```

Quite often, only principals with permission to read the `Contact` should be allowed to obtain it. In this situation the `AccessDecisionManager` approach provided by the `AbstractSecurityInterceptor` will not suffice. This is because the identity of the `Contact` is all that is available before the secure object is invoked. The `AclAfterInvocationProvider` delivers a solution, and is configured as follows:

```
<bean id="afterAclRead" class="org.acegisecurity.afterinvocation.AclEntryAfterInvocationProvider">
  <constructor-arg>
    <ref bean="aclService"/>
  </constructor-arg>
  <constructor-arg>
    <list>
      <ref local="org.acegisecurity.acls.domain.BasePermission.ADMINISTRATION"/>
      <ref local="org.acegisecurity.acls.domain.BasePermission.READ"/>
    </list>
  </constructor-arg>
</bean>
```

In the above example, the `Contact` will be retrieved and passed to the `AclEntryAfterInvocationProvider`. The provider will thrown an `AccessDeniedException` if one of the listed `requirePermission`s is not held by the `Authentication`. The `AclEntryAfterInvocationProvider` queries the `Acl`Service to determine the ACL that applies for this domain object to this `Authentication`.

Similar to the `AclEntryAfterInvocationProvider` is `AclEntryAfterInvocationCollectionFilteringProvider`. It is designed to remove `Collection` or array elements for which a principal does not have access. It never thrown an `AccessDeniedException` - simply silently removes the offending elements. The provider is configured as follows:

```
<bean id="afterAclCollectionRead" class="org.acegisecurity.afterinvocation.AclEntryAfterInvocationCollectionFilt
  <constructor-arg>
    <ref bean="aclService"/>
  </constructor-arg>
  <constructor-arg>
    <list>
      <ref local="org.acegisecurity.acls.domain.BasePermission.ADMINISTRATION"/>
      <ref local="org.acegisecurity.acls.domain.BasePermission.READ"/>
    </list>
  </constructor-arg>
</bean>
```

As you can imagine, the returned `Object` must be a `Collection` or array for this provider to operate. It will remove any element if the `AclManager` indicates the `Authentication` does not hold one of the listed `requirePermissionS`.

The Contacts sample application demonstrates these two `AfterInvocationProviderS`.

20.3.2. ACL-Aware AfterInvocationProviders (old ACL module)

PLEASE NOTE: Acegi Security 1.0.3 contains a preview of a new ACL module. The new ACL module is a significant rewrite of the existing ACL module. The new module can be found under the `org.acegisecurity.acls` package, with the old ACL module under `org.acegisecurity.acl`. We encourage users to consider testing with the new ACL module and build applications with it. The old ACL module should be considered deprecated and may be removed from a future release.

A common services layer method we've all written at one stage or another looks like this:

```
public Contact getById(Integer id);
```

Quite often, only principals with permission to read the `Contact` should be allowed to obtain it. In this situation the `AccessDecisionManager` approach provided by the `AbstractSecurityInterceptor` will not suffice. This is because the identity of the `Contact` is all that is available before the secure object is invoked. The `BasicAclAfterInvocationProvider` delivers a solution, and is configured as follows:

```
<bean id="afterAclRead" class="org.acegisecurity.afterinvocation.BasicAclEntryAfterInvocationProvider">
  <property name="aclManager"><ref local="aclManager"/></property>
  <property name="requirePermission">
    <list>
      <ref local="org.acegisecurity.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="org.acegisecurity.acl.basic.SimpleAclEntry.READ"/>
    </list>
  </property>
</bean>
```

In the above example, the `Contact` will be retrieved and passed to the `BasicAclEntryAfterInvocationProvider`. The provider will thrown an `AccessDeniedException` if one of the listed `requirePermissionS` is not held by the `Authentication`. The `BasicAclEntryAfterInvocationProvider` queries the `AclManager` to determine the ACL that applies for this domain object to this `Authentication`.

Similar to the `BasicAclEntryAfterInvocationProvider` is `BasicAclEntryAfterInvocationCollectionFilteringProvider`. It is designed to remove `Collection` or array elements for which a principal does not have access. It never thrown an `AccessDeniedException` - simply silently removes the offending elements. The provider is configured as follows:

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

As you can imagine, the returned `Object` must be a `Collection` or array for this provider to operate. It will remove any element if the `AclManager` indicates the `Authentication` does not hold one of the listed `requirePermissionS`.

The Contacts sample application demonstrates these two `AfterInvocationProviderS`.

20.4. Authorization Tag Libraries

`AuthorizeTag` is used to include content if the current principal holds certain `GrantedAuthority`s.

The following JSP fragment illustrates how to use the `AuthorizeTag`:

```
<authz:authorize ifAllGranted="ROLE_SUPERVISOR">
  <td>
    <A HREF="del.htm?id=<c:out value="${contact.id}"/>">Del</A>
  </td>
</authz:authorize>
```

This tag would cause the tag's body to be output if the principal has been granted ROLE_SUPERVISOR.

The `authz:authorize` tag declares the following attributes:

- `ifAllGranted`: All the listed roles must be granted for the tag to output its body.
- `ifAnyGranted`: Any of the listed roles must be granted for the tag to output its body.
- `ifNotGranted`: None of the listed roles must be granted for the tag to output its body.

You'll note that in each attribute you can list multiple roles. Simply separate the roles using a comma. The `authorize` tag ignores whitespace in attributes.

The tag library logically ANDs all of it's parameters together. This means that if you combine two or more attributes, all attributes must be true for the tag to output it's body. Don't add an `ifAllGranted="ROLE_SUPERVISOR"`, followed by an `ifNotGranted="ROLE_SUPERVISOR"`, or you'll be surprised to never see the tag's body.

By requiring all attributes to return true, the authorize tag allows you to create more complex authorization scenarios. For example, you could declare an `ifAllGranted="ROLE_SUPERVISOR"` and an `ifNotGranted="ROLE_NEWBIE_SUPERVISOR"` in the same tag, in order to prevent new supervisors from seeing the tag body. However it would no doubt be simpler to use `ifAllGranted="ROLE_EXPERIENCED_SUPERVISOR"` rather than inserting NOT conditions into your design.

One last item: the tag verifies the authorizations in a specific order: first `ifNotGranted`, then `ifAllGranted`, and finally, `if AnyGranted`.

`AccessControlListTag` is used to include content if the current principal has an ACL to the indicated domain object.

The following JSP fragment illustrates how to use the `AccessControlListTag`:

```
<authz:accesscontrollist domainObject="${contact}" hasPermission="8,16">
  <td><A HREF="<c:url value="del.htm"><c:param name="contactId" value="${contact.id}"/></c:url>">Del</A></td>
</authz:accesscontrollist>
```

This tag would cause the tag's body to be output if the principal holds either permission 16 or permission 1 for the "contact" domain object. The numbers are actually integers that are used with `BasePermission` bit masking. Please refer to the ACL section of this reference guide to understand more about the ACL capabilities of Acegi Security.

`AclTag` is part of the old ACL module and should be considered deprecated. For the sake of historical reference, works exactly the samae as `AccessControlListTag`.

Chapter 21. Secure Object Implementations

21.1. AOP Alliance (MethodInvocation) Security Interceptor

To secure MethodInvocations, developers simply add a properly configured MethodSecurityInterceptor into the application context. Next the beans requiring security are chained into the interceptor. This chaining is accomplished using Spring's ProxyFactoryBean or BeanNameAutoProxyCreator, as commonly used by many other parts of Spring (refer to the sample application for examples). Alternatively, Acegi Security provides a MethodDefinitionSourceAdvisor which may be used with Spring's DefaultAdvisorAutoProxyCreator to automatically chain the security interceptor in front of any beans defined against the MethodSecurityInterceptor. The MethodSecurityInterceptor itself is configured as follows:

```
<bean id="bankManagerSecurity" class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
  <property name="validateConfigAttributes"><value>true</value></property>
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="accessDecisionManager"><ref bean="accessDecisionManager"/></property>
  <property name="runAsManager"><ref bean="runAsManager"/></property>
  <property name="afterInvocationManager"><ref bean="afterInvocationManager"/></property>
  <property name="objectDefinitionSource">
    <value>
      org.acegisecurity.context.BankManager.delete*=ROLE_SUPERVISOR,RUN_AS_SERVER
      org.acegisecurity.context.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISOR,BANKSECURITY_CUSTOMER,RUN_AS_
    </value>
  </property>
</bean>
```

As shown above, the MethodSecurityInterceptor is configured with a reference to an AuthenticationManager, AccessDecisionManager and RunAsManager, which are each discussed in separate sections below. In this case we've also defined an AfterInvocationManager, although this is entirely optional. The MethodSecurityInterceptor is also configured with configuration attributes that apply to different method signatures. A full discussion of configuration attributes is provided in the High Level Design section of this document.

The MethodSecurityInterceptor can be configured with configuration attributes in three ways. The first is via a property editor and the application context, which is shown above. The second is via defining the configuration attributes in your source code using Jakarta Commons Attributes or Java 5 Annotations. The third is via writing your own ObjectDefinitionSource, although this is beyond the scope of this document. Irrespective of the approach used, the ObjectDefinitionSource is responsible for returning a ConfigAttributeDefinition object that contains all of the configuration attributes associated with a single secure method.

It should be noted that the MethodSecurityInterceptor.setObjectDefinitionSource() method actually expects an instance of MethodDefinitionSource. This is a marker interface which subclasses ObjectDefinitionSource. It simply denotes the ObjectDefinitionSource understands MethodInvocations. In the interests of simplicity we'll continue to refer to the MethodDefinitionSource as an ObjectDefinitionSource, as the distinction is of little relevance to most users of the MethodSecurityInterceptor.

If using the application context property editor approach (as shown above), commas are used to delimit the different configuration attributes that apply to a given method pattern. Each configuration attribute is assigned into its own SecurityConfig object. The SecurityConfig object is discussed in the High Level Design section.

If you are using the Jakarta Commons Attributes approach, your bean context will be configured differently:

```
<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
<bean id="objectDefinitionSource" class="org.acegisecurity.intercept.method.MethodDefinitionAttributes">
  <property name="attributes"><ref local="attributes"/></property>
</bean>

<bean id="bankManagerSecurity" class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
  <property name="validateConfigAttributes"><value>false</value></property>
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="accessDecisionManager"><ref bean="accessDecisionManager"/></property>
  <property name="runAsManager"><ref bean="runAsManager"/></property>
  <property name="objectDefinitionSource"><ref bean="objectDefinitionSource"/></property>
</bean>
```

In addition, your source code will contain Jakarta Commons Attributes tags that refer to a concrete implementation of `ConfigAttribute`. The following example uses the `SecurityConfig` implementation to represent the configuration attributes, and results in the same security configuration as provided by the property editor approach above:

```
public interface BankManager {

    /**
     * @@SecurityConfig("ROLE_SUPERVISOR")
     * @@SecurityConfig("RUN_AS_SERVER")
     */
    public void deleteSomething(int id);

    /**
     * @@SecurityConfig("ROLE_SUPERVISOR")
     * @@SecurityConfig("RUN_AS_SERVER")
     */
    public void deleteAnother(int id);

    /**
     * @@SecurityConfig("ROLE_TELLER")
     * @@SecurityConfig("ROLE_SUPERVISOR")
     * @@SecurityConfig("BANKSECURITY_CUSTOMER")
     * @@SecurityConfig("RUN_AS_SERVER")
     */
    public float getBalance(int id);
}
```

If you are using the Acegi Security Java 5 Annotations approach, your bean context will be configured as follows:

```
<bean id="attributes" class="org.acegisecurity.annotation.SecurityAnnotationAttributes"/>
<bean id="objectDefinitionSource" class="org.acegisecurity.intercept.method.MethodDefinitionAttributes">
  <property name="attributes"><ref local="attributes"/></property>
</bean>

<bean id="bankManagerSecurity" class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
  <property name="validateConfigAttributes"><value>false</value></property>
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="accessDecisionManager"><ref bean="accessDecisionManager"/></property>
  <property name="runAsManager"><ref bean="runAsManager"/></property>
  <property name="objectDefinitionSource"><ref bean="objectDefinitionSource"/></property>
</bean>
```

In addition, your source code will contain Acegi Java 5 Security Annotations that represent the `ConfigAttribute`. The following example uses the `@Secured` annotations to represent the configuration attributes, and results in the same security configuration as provided by the property editor approach:

```
import org.acegisecurity.annotation.Secured;

public interface BankManager {

    /**
     * Delete something
     */
```

```
    @Secured({"ROLE_SUPERVISOR","RUN_AS_SERVER" })
    public void deleteSomething(int id);

    /**
     * Delete another
     */
    @Secured({"ROLE_SUPERVISOR","RUN_AS_SERVER" })
    public void deleteAnother(int id);

    /**
     * Get balance
     */
    @Secured({"ROLE_TELLER","ROLE_SUPERVISOR","BANKSECURITY_CUSTOMER","RUN_AS_SERVER" })
    public float getBalance(int id);
}
```

You might have noticed the `validateConfigAttributes` property in the above `MethodSecurityInterceptor` examples. When set to `true` (the default), at startup time the `MethodSecurityInterceptor` will evaluate if the provided configuration attributes are valid. It does this by checking each configuration attribute can be processed by either the `AccessDecisionManager` or the `RunAsManager`. If neither of these can process a given configuration attribute, an exception is thrown. If using the Jakarta Commons Attributes method of configuration, you should set `validateConfigAttributes` to `false`.

Please note that when using `BeanNameAutoProxyCreator` to create the required proxy for security, the configuration must contain the property `proxyTargetClass` set to `true`. Otherwise, the method passed to `MethodSecurityInterceptor.invoke` is the proxy's caller, not the proxy's target. Note that this introduces a requirement on CGLIB. See an example of using `BeanNameAutoProxyCreator` below:

```
<bean id="autoProxyCreator" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="interceptorNames">
    <list><value>methodSecurityInterceptor</value></list>
  </property>
  <property name="beanNames">
    <list><value>targetObjectName</value></list>
  </property>
  <property name="proxyTargetClass" value="true"/>
</bean>
```

21.2. AspectJ (JoinPoint) Security Interceptor

The AspectJ security interceptor is very similar to the AOP Alliance security interceptor discussed in the previous section. Indeed we will only discuss the differences in this section.

The AspectJ interceptor is named `AspectJSecurityInterceptor`. Unlike the AOP Alliance security interceptor, which relies on the Spring application context to weave in the security interceptor via proxying, the `AspectJSecurityInterceptor` is weaved in via the AspectJ compiler. It would not be uncommon to use both types of security interceptors in the same application, with `AspectJSecurityInterceptor` being used for domain object instance security and the AOP Alliance `MethodSecurityInterceptor` being used for services layer security.

Let's first consider how the `AspectJSecurityInterceptor` is configured in the Spring application context:

```
<bean id="bankManagerSecurity" class="org.acegisecurity.intercept.method.aspectj.AspectJSecurityInterceptor">
  <property name="validateConfigAttributes"><value>true</value></property>
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="accessDecisionManager"><ref bean="accessDecisionManager"/></property>
  <property name="runAsManager"><ref bean="runAsManager"/></property>
  <property name="afterInvocationManager"><ref bean="afterInvocationManager"/></property>
  <property name="objectDefinitionSource">
    <value>
      org.acegisecurity.context.BankManager.delete*=ROLE_SUPERVISOR,RUN_AS_SERVER
      org.acegisecurity.context.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISOR,BANKSECURITY_CUSTOMER,RUN_AS_
    </value>
  </property>
```

```
</bean>
```

As you can see, aside from the class name, the `AspectJSecurityInterceptor` is exactly the same as the AOP Alliance security interceptor. Indeed the two interceptors can share the same `objectDefinitionSource`, as the `ObjectDefinitionSource` works with `java.lang.reflect.Method`s rather than an AOP library-specific class. Of course, your access decisions have access to the relevant AOP library-specific invocation (ie `MethodInvocation` or `JoinPoint`) and as such can consider a range of addition criteria when making access decisions (such as method arguments).

Next you'll need to define an AspectJ `aspect`. For example:

```
package org.acegisecurity.samples.aspectj;

import org.acegisecurity.intercept.method.aspectj.AspectJSecurityInterceptor;
import org.acegisecurity.intercept.method.aspectj.AspectJCallback;
import org.springframework.beans.factory.InitializingBean;

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {

  private AspectJSecurityInterceptor securityInterceptor;

  pointcut domainObjectInstanceExecution(): target(PersistableEntity)
            && execution(public * *(..)) && !within(DomainObjectInstanceSecurityAspect);

  Object around(): domainObjectInstanceExecution() {
    if (this.securityInterceptor != null) {
      AspectJCallback callback = new AspectJCallback() {
        public Object proceedWithObject() {
        return proceed();
      }
    };
    return this.securityInterceptor.invoke(thisJoinPoint, callback);
    } else {
      return proceed();
    }
  }

  public AspectJSecurityInterceptor getSecurityInterceptor() {
    return securityInterceptor;
  }

  public void setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor) {
    this.securityInterceptor = securityInterceptor;
  }

  public void afterPropertiesSet() throws Exception {
    if (this.securityInterceptor == null)
      throw new IllegalArgumentException("securityInterceptor required");
  }
}
```

In the above example, the security interceptor will be applied to every instance of `PersistableEntity`, which is an abstract class not shown (you can use any other class or `pointcut` expression you like). For those curious, `AspectJCallback` is needed because the `proceed();` statement has special meaning only within an `around()` body. The `AspectJSecurityInterceptor` calls this anonymous `AspectJCallback` class when it wants the target object to continue.

You will need to configure Spring to load the aspect and wire it with the `AspectJSecurityInterceptor`. A bean declaration which achieves this is shown below:

```
<bean id="domainObjectInstanceSecurityAspect"
    class="org.acegisecurity.samples.aspectj.DomainObjectInstanceSecurityAspect"
    factory-method="aspectOf">
  <property name="securityInterceptor"><ref bean="aspectJSecurityInterceptor"/></property>
</bean>
```

That's it! Now you can create your beans from anywhere within your application, using whatever means you think fit (eg `new Person();`) and they will have the security interceptor applied.

21.3. FilterInvocation Security Interceptor

To secure `FilterInvocation`s, developers need to add a filter to their `web.xml` that delegates to the `FilterSecurityInterceptor`. A typical configuration example is provided below:

```
<filter>
  <filter-name>Acegi HTTP Request Security Filter</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.intercept.web.FilterSecurityInterceptor</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Acegi HTTP Request Security Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Notice that the filter is actually a `FilterToBeanProxy`. Most of the filters used by Acegi Security use this class. Refer to the Filters section to learn more about this bean.

In the application context you will need to configure three beans:

```
<bean id="exceptionTranslationFilter" class="org.acegisecurity.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint"><ref local="authenticationEntryPoint"/></property>
</bean>

<bean id="authenticationEntryPoint" class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint"
  <property name="loginFormUrl"><value>/acegilogin.jsp</value></property>
  <property name="forceHttps"><value>false</value></property>
</bean>

<bean id="filterSecurityInterceptor" class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="accessDecisionManager"><ref bean="accessDecisionManager"/></property>
  <property name="objectDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      \A/secure/super/.*\Z=ROLE_WE_DONT_HAVE
      \A/secure/.*\Z=ROLE_SUPERVISOR,ROLE_TELLER
    </value>
  </property>
</bean>
```

The `ExceptionTranslationFilter` provides the bridge between Java exceptions and HTTP responses. It is solely concerned with maintaining the user interface. This filter does not do any actual security enforcement. If an `AuthenticationException` is detected, the filter will call the AuthenticationEntryPoint to commence the authentication process (e.g. a user login).

The `AuthenticationEntryPoint` will be called if the user requests a secure HTTP resource but they are not authenticated. The class handles presenting the appropriate response to the user so that authentication can begin. Three concrete implementations are provided with Acegi Security: `AuthenticationProcessingFilterEntryPoint` for commencing a form-based authentication, `BasicProcessingFilterEntryPoint` for commencing a HTTP Basic authentication process, and `CasProcessingFilterEntryPoint` for commencing a JA-SIG Central Authentication Service (CAS) login. The `AuthenticationProcessingFilterEntryPoint` and `CasProcessingFilterEntryPoint` have optional properties related to forcing the use of HTTPS, so please refer to the JavaDocs if you require this.

`FilterSecurityInterceptor` is responsible for handling the security of HTTP resources. Like any other security interceptor, it requires a reference to an `AuthenticationManager` and an `AccessDecisionManager`, which are both discussed in separate sections below. The `FilterSecurityInterceptor` is also configured with configuration attributes that apply to different HTTP URL requests. A full discussion of configuration attributes is provided in the High Level Design section of this document.

The `FilterSecurityInterceptor` can be configured with configuration attributes in two ways. The first is via a property editor and the application context, which is shown above. The second is via writing your own `ObjectDefinitionSource`, although this is beyond the scope of this document. Irrespective of the approach used, the `ObjectDefinitionSource` is responsible for returning a `ConfigAttributeDefinition` object that contains all of the configuration attributes associated with a single secure HTTP URL.

It should be noted that the `FilterSecurityInterceptor.setObjectDefinitionSource()` method actually expects an instance of `FilterInvocationDefinitionSource`. This is a marker interface which subclasses `ObjectDefinitionSource`. It simply denotes the `ObjectDefinitionSource` understands `FilterInvocation`s. In the interests of simplicity we'll continue to refer to the `FilterInvocationDefinitionSource` as an `ObjectDefinitionSource`, as the distinction is of little relevance to most users of the `FilterSecurityInterceptor`.

If using the application context property editor approach (as shown above), commas are used to delimit the different configuration attributes that apply to each HTTP URL. Each configuration attribute is assigned into its own `SecurityConfig` object. The `SecurityConfig` object is discussed in the High Level Design section. The `ObjectDefinitionSource` created by the property editor, `FilterInvocationDefinitionSource`, matches configuration attributes against `FilterInvocations` based on expression evaluation of the request URL. Two standard expression syntaxes are supported. The default is to treat all expressions as regular expressions. Alternatively, the presence of a `PATTERN_TYPE_APACHE_ANT` directive will cause all expressions to be treated as Apache Ant paths. It is not possible to mix expression syntaxes within the same definition. For example, the earlier configuration could be generated using Apache Ant paths as follows:

```
<bean id="filterInvocationInterceptor" class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager"><ref bean="authenticationManager"/></property>
  <property name="accessDecisionManager"><ref bean="accessDecisionManager"/></property>
  <property name="runAsManager"><ref bean="runAsManager"/></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_TELLER
    </value>
  </property>
</bean>
```

Irrespective of the type of expression syntax used, expressions are always evaluated in the order they are defined. Thus it is important that more specific expressions are defined higher in the list than less specific expressions. This is reflected in our example above, where the more specific `/secure/super/` pattern appears higher than the less specific `/secure/` pattern. If they were reversed, the `/secure/` pattern would always match and the `/secure/super/` pattern would never be evaluated.

The special keyword `CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON` causes the `FilterInvocationDefinitionSource` to automatically convert a request URL to lowercase before comparison against the expressions. Whilst by default the case of the request URL is not converted, it is generally recommended to use `CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON` and write each expression assuming lowercase.

As with other security interceptors, the `validateConfigAttributes` property is observed. When set to

`true` (the default), at startup time the `FilterSecurityInterceptor` will evaluate if the provided configuration attributes are valid. It does this by checking each configuration attribute can be processed by either the `AccessDecisionManager` or the `RunAsManager`. If neither of these can process a given configuration attribute, an exception is thrown.

Chapter 22. Domain Object Security

22.1. Overview

PLEASE NOTE: Acegi Security 1.0.3 contains a preview of a new ACL module. The new ACL module is a significant rewrite of the existing ACL module. The new module can be found under the `org.acegisecurity.acls` package, with the old ACL module under `org.acegisecurity.acl`. We encourage users to consider testing with the new ACL module and build applications with it. The old ACL module should be considered deprecated and may be removed from a future release.

Complex applications often will find the need to define access permissions not simply at a web request or method invocation level. Instead, security decisions need to comprise both who (`Authentication`), where (`MethodInvocation`) and what (`SomeDomainObject`). In other words, authorization decisions also need to consider the actual domain object instance subject of a method invocation.

Imagine you're designing an application for a pet clinic. There will be two main groups of users of your Spring-based application: staff of the pet clinic, as well as the pet clinic's customers. The staff will have access to all of the data, whilst your customers will only be able to see their own customer records. To make it a little more interesting, your customers can allow other users to see their customer records, such as their "puppy preschool "mentor or president of their local "Pony Club". Using Acegi Security as the foundation, you have several approaches that can be used:

1. Write your business methods to enforce the security. You could consult a collection within the `Customer` domain object instance to determine which users have access. By using the `SecurityContextHolder.getContext().getAuthentication()`, you'll be able to access the `Authentication` object.

2. Write an `AccessDecisionVoter` to enforce the security from the `GrantedAuthority[]`s stored in the `Authentication` object. This would mean your `AuthenticationManager` would need to populate the `Authentication` with custom `GrantedAuthority`[]s representing each of the `Customer` domain object instances the principal has access to.

3. Write an `AccessDecisionVoter` to enforce the security and open the target `Customer` domain object directly. This would mean your voter needs access to a DAO that allows it to retrieve the `Customer` object. It would then access the `Customer` object's collection of approved users and make the appropriate decision.

Each one of these approaches is perfectly legitimate. However, the first couples your authorization checking to your business code. The main problems with this include the enhanced difficulty of unit testing and the fact it would be more difficult to reuse the `Customer` authorization logic elsewhere. Obtaining the `GrantedAuthority[]`s from the `Authentication` object is also fine, but will not scale to large numbers of `Customer`s. If a user might be able to access 5,000 `Customer`s (unlikely in this case, but imagine if it were a popular vet for a large Pony Club!) the amount of memory consumed and time required to construct the `Authentication` object would be undesirable. The final method, opening the `Customer` directly from external code, is probably the best of the three. It achieves separation of concerns, and doesn't misuse memory or CPU cycles, but it is still inefficient in that both the `AccessDecisionVoter` and the eventual business method itself will perform a call to the DAO responsible for retrieving the `Customer` object. Two accesses per method invocation is clearly undesirable. In addition, with every approach listed you'll need to write your own access control list (ACL) persistence and business logic from scratch.

Fortunately, there is another alternative, which we'll talk about below.

22.2. Key Concepts

The org.acegisecurity.acls package should be consulted for its major interfaces. The key interfaces are:

- `Acl`: Every domain object has one and only one `Acl` object, which internally holds the `AccessControlEntry`s as well as knows the owner of the `Acl`. An Acl does not refer directly to the domain object, but instead to an `ObjectIdentity`.
- `AccessControlEntry`: An Acl holds multiple `AccessControlEntry`s, which are often abbreviated as ACEs in the framework. Each ACE refers to a specific tuple of `Permission`, `Sid` and `Acl`. An ACE can also be granting or non-granting and contain audit settings.
- `Permission`: A permission represents an immutable particular bit mask, and offers convenience functions for bit masking and outputting information.
- `Sid`: The ACL module needs to refer to principals and `GrantedAuthority[]`s. A level of indirection is provided by the `Sid` interface. Common classes include `PrincipalSid` (to represent the principal inside an `Authentication` object) and `GrantedAuthoritySid`.
- `ObjectIdentity`: Each domain object is represented internally within the ACL module by an `ObjectIdentity`.
- `AclService`: Retrieves the `Acl` applicable for a given `ObjectIdentity`.
- `MutableAclService`: Allows a modified `Acl` to be presented for persistence. It is not essential to use this interface if you do not wish.

The ACL module was based on extensive feedback from the user community following real-world use of the original ACL module. This feedback resulted in a rearchitecture of the ACL module to offer significantly enhanced performance (particularly in the area of database retrieval), significantly better encapsulation, higher cohesion, and enhanced customisation points.

The Contacts Sample that ships with Acegi Security 1.0.3 offers a demonstration of the new ACL module. Converting Contacts from using the old module to the new module was relatively simple, and users of the old ACL module will likely find their applications can be modified with relatively little work.

We will document the new ACL module more fully with a subsequent release. Please note that the new ACL module should be considered a preview only (ie do not use in production without proper prior testing), and there is a small chance there may be changes between 1.0.3 and 1.1.0 when it will become final. Nevertheless, compatibility-affecting changes are considered quite unlikely, especially given the module is already based on several years of feedback from users of the original ACL module.

Chapter 23. Domain Object Security (old ACL module)

23.1. Overview

PLEASE NOTE: Acegi Security 1.0.3 contains a preview of a new ACL module. The new ACL module is a significant rewrite of the existing ACL module. The new module can be found under the `org.acegisecurity.acls` package, with the old ACL module under `org.acegisecurity.acl`. We encourage users to consider testing with the new ACL module and build applications with it. The old ACL module should be considered deprecated and may be removed from a future release.

Complex applications often will find the need to define access permissions not simply at a web request or method invocation level. Instead, security decisions need to comprise both who (`Authentication`), where (`MethodInvocation`) and what (`SomeDomainObject`). In other words, authorization decisions also need to consider the actual domain object instance subject of a method invocation.

Imagine you're designing an application for a pet clinic. There will be two main groups of users of your Spring-based application: staff of the pet clinic, as well as the pet clinic's customers. The staff will have access to all of the data, whilst your customers will only be able to see their own customer records. To make it a little more interesting, your customers can allow other users to see their customer records, such as their "puppy preschool "mentor or president of their local "Pony Club". Using Acegi Security as the foundation, you have several approaches that can be used:

1. Write your business methods to enforce the security. You could consult a collection within the `Customer` domain object instance to determine which users have access. By using the `SecurityContextHolder.getContext().getAuthentication()`, you'll be able to access the `Authentication` object.

2. Write an `AccessDecisionVoter` to enforce the security from the `GrantedAuthority[]`s stored in the `Authentication` object. This would mean your `AuthenticationManager` would need to populate the `Authentication` with custom `GrantedAuthority`[]s representing each of the `Customer` domain object instances the principal has access to.

3. Write an `AccessDecisionVoter` to enforce the security and open the target `Customer` domain object directly. This would mean your voter needs access to a DAO that allows it to retrieve the `Customer` object. It would then access the `Customer` object's collection of approved users and make the appropriate decision.

Each one of these approaches is perfectly legitimate. However, the first couples your authorization checking to your business code. The main problems with this include the enhanced difficulty of unit testing and the fact it would be more difficult to reuse the `Customer` authorization logic elsewhere. Obtaining the `GrantedAuthority[]`s from the `Authentication` object is also fine, but will not scale to large numbers of `Customer`s. If a user might be able to access 5,000 `Customer`s (unlikely in this case, but imagine if it were a popular vet for a large Pony Club!) the amount of memory consumed and time required to construct the `Authentication` object would be undesirable. The final method, opening the `Customer` directly from external code, is probably the best of the three. It achieves separation of concerns, and doesn't misuse memory or CPU cycles, but it is still inefficient in that both the `AccessDecisionVoter` and the eventual business method itself will perform a call to the DAO responsible for retrieving the `Customer` object. Two accesses per method invocation is clearly

undesirable. In addition, with every approach listed you'll need to write your own access control list (ACL) persistence and business logic from scratch.

Fortunately, there is another alternative, which we'll talk about below.

23.2. Basic ACL Package

Please note that our Basic ACL services are currently being refactored. We expect release 1.1.0 will contain this new code. Planned code is already in the Acegi Security Subversion sandbox, so please check there if you have a new application requiring ACLs or are in the planning stages. The Basic ACL services will be deprecated from release 1.1.0.

The `org.acegisecurity.acl` package is very simple, comprising only a handful of interfaces and a single class, as shown in Figure 6. It provides the basic foundation for access control list (ACL) lookups.

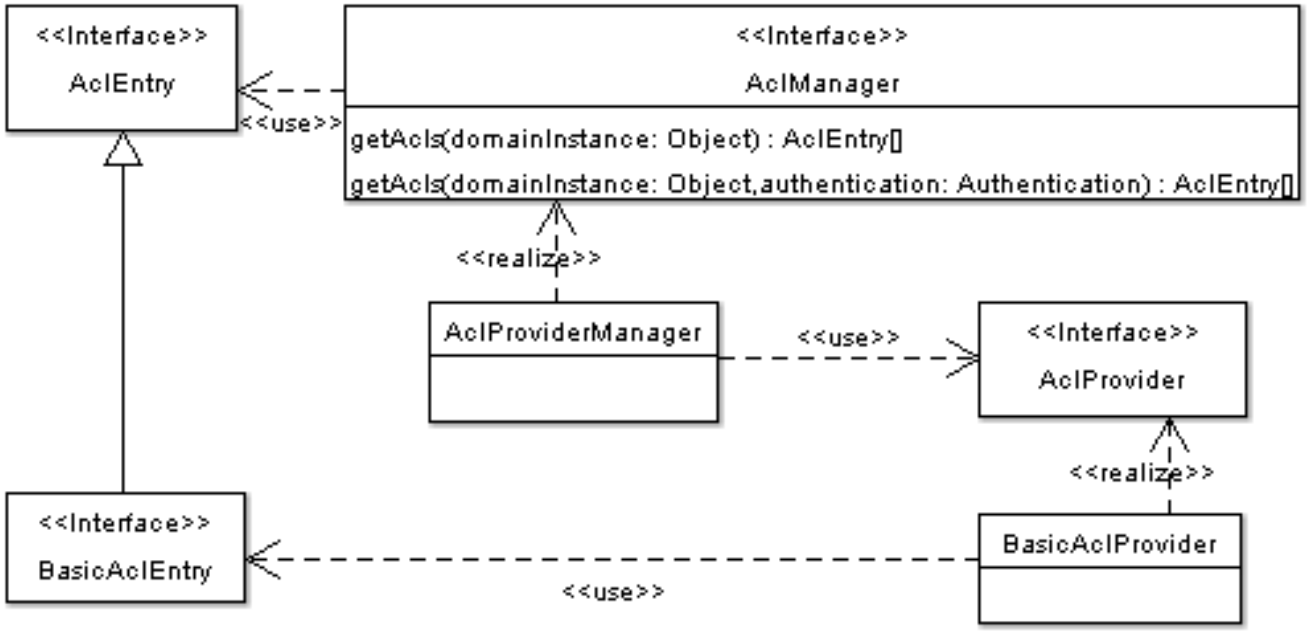

Figure 6: Access Control List Manager

The central interface is `AclManager`, which is defined by two methods:

```
public AclEntry[] getAcls(java.lang.Object domainInstance);
public AclEntry[] getAcls(java.lang.Object domainInstance, Authentication authentication);
```

`AclManager` is intended to be used as a collaborator against your business objects, or, more desirably, `AccessDecisionVoter`s. This means you use Spring's normal `ApplicationContext` features to wire up your `AccessDecisionVoter` (or business method) with an `AclManager`. Consideration was given to placing the ACL information in the `ContextHolder`, but it was felt this would be inefficient both in terms of memory usage as well as the time spent loading potentially unused ACL information. The trade-off of needing to wire up a collaborator for those objects requiring ACL information is rather minor, particularly in a Spring-managed application.

The first method of the `AclManager` will return all ACLs applying to the domain object instance passed to it. The second method does the same, but only returns those ACLs which apply to the passed

`Authentication` object.

The `AclEntry` interface returned by `AclManager` is merely a marker interface. You will need to provide an implementation that reflects that ACL permissions for your application.

Rounding out the `org.acegisecurity.acl` package is an `AclProviderManager` class, with a corresponding `AclProvider` interface. `AclProviderManager` is a concrete implementation of `AclManager`, which iterates through registered `AclProvider`s. The first `AclProvider` that indicates it can authoritatively provide ACL information for the presented domain object instance will be used. This is very similar to the `AuthenticationProvider` interface used for authentication.

With this background, let's now look at a usable ACL implementation.

Acegi Security includes a production-quality ACL provider implementation, which is shown in Figure 7.

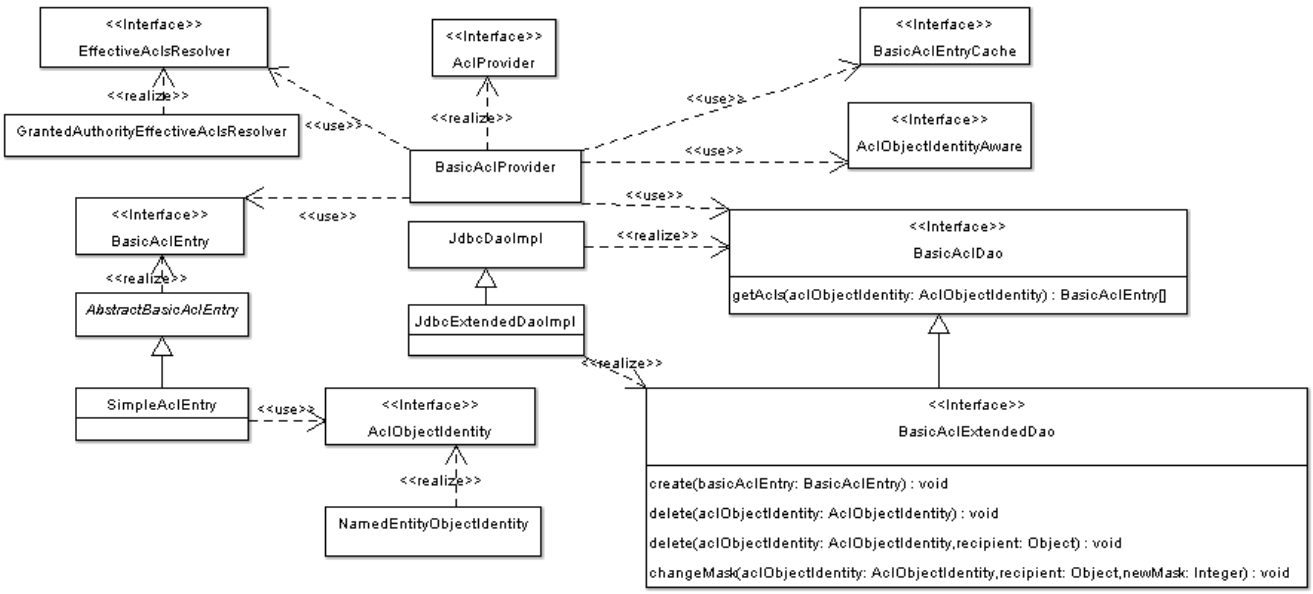

Figure 7: Basic ACL Manager

The implementation is based on integer masking, which is commonly used for ACL permissions given its flexibility and speed. Anyone who has used Unix's `chmod` command will know all about this type of permission masking (eg `chmod 777`). You'll find the classes and interfaces for the integer masking ACL package under `org.acegisecurity.acl.basic`.

Extending the `AclEntry` interface is a `BasicAclEntry` interface, with the main methods shown below:

```
public AclObjectIdentity getAclObjectIdentity();
public AclObjectIdentity getAclObjectParentIdentity();
public int getMask();
public java.lang.Object getRecipient();
```

As shown, each `BasicAclEntry` has four main properties. The `mask` is the integer that represents the permissions granted to the `recipient`. The `aclObjectIdentity` is able to identify the domain object instance for which the ACL applies, and the `aclObjectParentIdentity` optionally specifies the parent of the domain object instance. Multiple `BasicAclEntry`s usually exist against a single domain object instance, and as suggested by the parent identity property, permissions granted higher in the object hierarchy will trickle down and be inherited (unless blocked by integer zero).

`BasicAclEntry` implementations typically provide convenience methods, such as `isReadAllowed()`, to avoid application classes needing to perform bit masking themselves. The `SimpleAclEntry` and `AbstractBasicAclEntry` demonstrate and provide much of this bit masking logic.

The `AclObjectIdentity` itself is merely a marker interface, so you need to provide implementations for your domain objects. However, the package does include a `NamedEntityObjectIdentity` implementation which will suit many needs. The `NamedEntityObjectIdentity` identifies a given domain object instance by the classname of the instance and the identity of the instance. A `NamedEntityObjectIdentity` can be constructed manually (by calling the constructor and providing the classname and identity `String`s), or by passing in any domain object that contains a `getId()` method.

The actual `AclProvider` implementation is named `BasicAclProvider`. It has adopted a similar design to that used by the authentication-related `DaoAuthenticationProvder`. Specifically, you define a `BasicAclDao` against the provider, so different ACL repository types can be accessed in a pluggable manner. The `BasicAclProvider` also supports pluggable cache providers (with Acegi Security including an implementation that fronts EH-CACHE).

The `BasicAclDao` interface is very simple to implement:

```
public BasicAclEntry[] getAcls(AclObjectIdentity aclObjectIdentity);
```

A `BasicAclDao` implementation needs to understand the presented `AclObjectIdentity` and how it maps to a storage repository, find the relevant records, and create appropriate `BasicAclEntry` objects and return them.

Acegi Security includes a single `BasicAclDao` implementation called `JdbcDaoImpl`. As implied by the name, `JdbcDaoImpl` accesses ACL information from a JDBC database. There is also an extended version of this DAO, `JdbcExtendedDaoImpl`, which provides CRUD operations on the JDBC database, although we won't discuss these features here. The default database schema and some sample data will aid in understanding its function:

```
CREATE TABLE acl_object_identity (
     id IDENTITY NOT NULL,
     object_identity VARCHAR_IGNORECASE(250) NOT NULL,
     parent_object INTEGER,
     acl_class VARCHAR_IGNORECASE(250) NOT NULL,
     CONSTRAINT unique_object_identity UNIQUE(object_identity),
     FOREIGN KEY (parent_object) REFERENCES acl_object_identity(id)
);

CREATE TABLE acl_permission (
     id IDENTITY NOT NULL,
     acl_object_identity INTEGER NOT NULL,
     recipient VARCHAR_IGNORECASE(100) NOT NULL,
     mask INTEGER NOT NULL,
     CONSTRAINT unique_recipient UNIQUE(acl_object_identity, recipient),
     FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity(id)
);

INSERT INTO acl_object_identity VALUES (1, 'corp.DomainObject:1', null, 'org.acegisecurity.acl.basic.SimpleAclEr
INSERT INTO acl_object_identity VALUES (2, 'corp.DomainObject:2', 1, 'org.acegisecurity.acl.basic.SimpleAclEntry
INSERT INTO acl_object_identity VALUES (3, 'corp.DomainObject:3', 1, 'org.acegisecurity.acl.basic.SimpleAclEntry
INSERT INTO acl_object_identity VALUES (4, 'corp.DomainObject:4', 1, 'org.acegisecurity.acl.basic.SimpleAclEntry
INSERT INTO acl_object_identity VALUES (5, 'corp.DomainObject:5', 3, 'org.acegisecurity.acl.basic.SimpleAclEntry
INSERT INTO acl_object_identity VALUES (6, 'corp.DomainObject:6', 3, 'org.acegisecurity.acl.basic.SimpleAclEntry

INSERT INTO acl_permission VALUES (null, 1, 'ROLE_SUPERVISOR', 1);
INSERT INTO acl_permission VALUES (null, 2, 'ROLE_SUPERVISOR', 0);
INSERT INTO acl_permission VALUES (null, 2, 'marissa', 2);
INSERT INTO acl_permission VALUES (null, 3, 'scott', 14);
INSERT INTO acl_permission VALUES (null, 6, 'scott', 1);
```

As can be seen, database-specific constraints are used extensively to ensure the integrity of the ACL information. If you need to use a different database (Hypersonic SQL statements are shown above), you should try to implement equivalent constraints. The equivalent Oracle configuration is:

```
CREATE TABLE ACL_OBJECT_IDENTITY (
     ID number(19,0) not null,
     OBJECT_IDENTITY varchar2(255) NOT NULL,
     PARENT_OBJECT number(19,0),
     ACL_CLASS varchar2(255) NOT NULL,
     primary key (ID)
);
ALTER TABLE ACL_OBJECT_IDENTITY ADD CONTRAINT FK_PARENT_OBJECT foreign key (ID) references ACL_OBJECT_IDENTITY

CREATE SEQUENCE ACL_OBJECT_IDENTITY_SEQ;

CREATE OR REPLACE TRIGGER ACL_OBJECT_IDENTITY_ID
BEFORE INSERT ON ACL_OBJECT_IDENTITY
FOR EACH ROW
BEGIN
  SELECT ACL_OBJECT_IDENTITY_SEQ.NEXTVAL INTO :new.id FROM dual;
END;

CREATE TABLE ACL_PERMISSION (
     ID number(19,0) not null,
     ACL_OBJECT_IDENTITY number(19,0) NOT NULL,
     RECIPIENT varchar2(255) NOT NULL,
     MASK number(19,0) NOT NULL,
     primary key (ID)
);

ALTER TABLE ACL_PERMISSION ADD CONTRAINT UNIQUE_ID_RECIPIENT unique (acl_object_identity, recipient);

CREATE SEQUENCE ACL_PERMISSION_SEQ;

CREATE OR REPLACE TRIGGER ACL_PERMISSION_ID
BEFORE INSERT ON ACL_PERMISSION
FOR EACH ROW
BEGIN
  SELECT ACL_PERMISSION_SEQ.NEXTVAL INTO :new.id FROM dual;
END;

<bean id="basicAclExtendedDao" class="org.acegisecurity.acl.basic.jdbc.JdbcExtendedDaoImpl">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    <property name="objectPropertiesQuery" value="${acegi.objectPropertiesQuery}"/>
</bean>

<prop key="acegi.objectPropertiesQuery">SELECT CHILD.ID, CHILD.OBJECT_IDENTITY, CHILD.ACL_CLASS, PARENT.OBJECT_I
```

The `JdbcDaoImpl` will only respond to requests for `NamedEntityObjectIdentity`s. It converts such identities into a single `String`, comprising the `NamedEntityObjectIdentity.getClassname() + ":" + NamedEntityObjectIdentity.getId()`. This yields the type of `object_identity` values shown above. As indicated by the sample data, each database row corresponds to a single `BasicAclEntry`. As stated earlier and demonstrated by `corp.DomainObject:2` in the above sample data, each domain object instance will often have multiple `BasicAclEntry`[]s.

As `JdbcDaoImpl` is required to return concrete `BasicAclEntry` classes, it needs to know which `BasicAclEntry` implementation it is to create and populate. This is the role of the `acl_class` column. `JdbcDaoImpl` will create the indicated class and set its `mask`, `recipient`, `aclObjectIdentity` and `aclObjectParentIdentity` properties.

As you can probably tell from the sample data, the `parent_object_identity` value can either be null or in the same format as the `object_identity`. If non-null, `JdbcDaoImpl` will create a `NamedEntityObjectIdentity` to place inside the returned `BasicAclEntry` class.

Returning to the `BasicAclProvider`, before it can poll the `BasicAclDao` implementation it needs to convert the domain object instance it was passed into an `AclObjectIdentity`. `BasicAclProvider` has a

`protected AclObjectIdentity obtainIdentity(Object domainInstance)` method that is responsible for this. As a protected method, it enables subclasses to easily override. The normal implementation checks whether the passed domain object instance implements the `AclObjectIdentityAware` interface, which is merely a getter for an `AclObjectIdentity`. If the domain object does implement this interface, that is the identity returned. If the domain object does not implement this interface, the method will attempt to create an `AclObjectIdentity` by passing the domain object instance to the constructor of a class defined by the `BasicAclProvider.getDefaultAclObjectIdentity()` method. By default the defined class is `NamedEntityObjectIdentity`, which was described in more detail above. Therefore, you will need to either (i) provide a `getId()` method on your domain objects, (ii) implement `AclObjectIdentityAware` on your domain objects, (iii) provide an alternative `AclObjectIdentity` implementation that will accept your domain object in its constructor, or (iv) override the `obtainIdentity(Object)` method.

Once the `AclObjectIdentity` of the domain object instance is determined, the `BasicAclProvider` will poll the DAO to obtain its `BasicAclEntry`[]s. If any of the entries returned by the DAO indicate there is a parent, that parent will be polled, and the process will repeat until there is no further parent. The permissions assigned to a `recipient` closest to the domain object instance will always take priority and override any inherited permissions. From the sample data above, the following inherited permissions would apply:

```
--- Mask integer 0  = no permissions
--- Mask integer 1  = administer
--- Mask integer 2  = read
--- Mask integer 6  = read and write permissions
--- Mask integer 14 = read and write and create permissions

---------------------------------------------------------------------
--- *** INHERITED RIGHTS FOR DIFFERENT INSTANCES AND RECIPIENTS ***
--- INSTANCE  RECIPIENT         PERMISSION(S) (COMMENT #INSTANCE)
---------------------------------------------------------------------
---    1      ROLE_SUPERVISOR   Administer
---    2      ROLE_SUPERVISOR   None (overrides parent #1)
---           marissa           Read
---    3      ROLE_SUPERVISOR   Administer (from parent #1)
---           scott             Read, Write, Create
---    4      ROLE_SUPERVISOR   Administer (from parent #1)
---    5      ROLE_SUPERVISOR   Administer (from parent #3)
---           scott             Read, Write, Create (from parent #3)
---    6      ROLE_SUPERVISOR   Administer (from parent #3)
---           scott             Administer (overrides parent #3)
```

So the above explains how a domain object instance has its `AclObjectIdentity` discovered, and the `BasicAclDao` will be polled successively until an array of inherited permissions is constructed for the domain object instance. The final step is to determine the `BasicAclEntry`[]s that are actually applicable to a given `Authentication` object.

As you would recall, the `AclManager` (and all delegates, up to and including `BasicAclProvider`) provides a method which returns only those `BasicAclEntry`[]s applying to a passed `Authentication` object. `BasicAclProvider` delivers this functionality by delegating the filtering operation to an `EffectiveAclsResolver` implementation. The default implementation, `GrantedAuthorityEffectiveAclsResolver`, will iterate through the `BasicAclEntry`[]s and include only those where the `recipient` is equal to either the `Authentication`'s `principal` or any of the `Authentication`'s `GrantedAuthority`[]s. Please refer to the JavaDocs for more information.

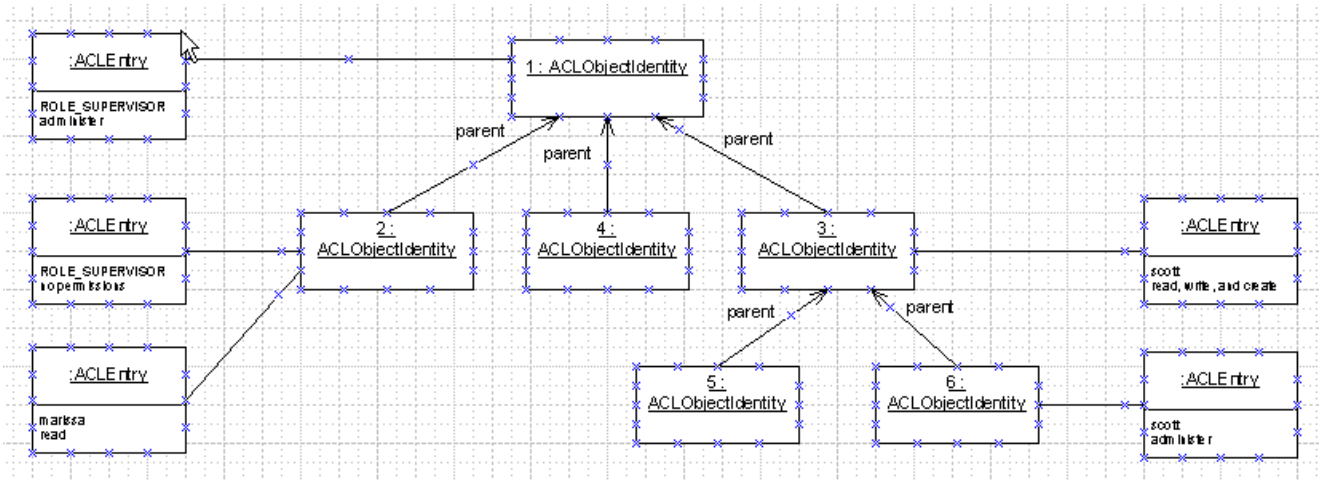

Figure 8: ACL Instantiation Approach

The above figure explains the key relationships between objects in the Basic ACL package.

Part IV. Other Resources

In addition to this reference guide, a number of other resources exist to help you learn how to use Acegi Security. These resources are discussed in this section.

Chapter 24. Sample Applications

24.1. Contacts

Included with Acegi Security is a very simple application that can demonstrate the basic security facilities provided by the system (and confirm your Container Adapter is properly configured if you're using one).

If you build from Subversion, the Contacts sample application includes three deployable versions: `acegi-security-sample-contacts-filter.war` is configured with the HTTP Session Authentication approach. Acegi`-security-sample-contacts-ca.war` is configured to use a Container Adapter. Finally, `acegi-security-sample-contacts-cas.war` is designed to work with a JA-SIG CAS server. If you're just wanting to see how the sample application works, please use `acegi-security-sample-contacts-filter.war` as it does not require special configuration of your container. This is also the artifact included in official release ZIPs.

To deploy, simply copy the relevant WAR file from Acegi Security distribution into your container's `webapps` directory.

After starting your container, check the application can load. Visit `http://localhost:8080/acegi-security-sample-contacts-filter` (or whichever URL is appropriate for your web container and the WAR you deployed). A random contact should be displayed. Click "Refresh" several times and you will see different contacts. The business method that provides this random contact is not secured.

Next, click "Debug". You will be prompted to authenticate, and a series of usernames and passwords are suggested on that page. Simply authenticate with any of these and view the resulting page. It should contain a success message similar to the following:

> Context on SecurityContextHolder is of type: org.acegisecurity.context.SecurityContextImpl
>
> The Context implements SecurityContext.
>
> Authentication object is of type: org.acegisecurity.adapters.PrincipalAcegiUserToken
>
> Authentication object as a String: org.acegisecurity.adapters.PrincipalAcegiUserToken@e9a7c2: Username: marissa; Password: [PROTECTED]; Authenticated: true; Granted Authorities: ROLE_TELLER, ROLE_SUPERVISOR
>
> Authentication object holds the following granted authorities:
>
> ROLE_TELLER (getAuthority(): ROLE_TELLER)
>
> ROLE_SUPERVISOR (getAuthority(): ROLE_SUPERVISOR)
>
> SUCCESS! Your [container adapter|web filter] appears to be properly configured!

If you receive a different message, and deployed `acegi-security-sample-contacts-ca.war`, check you

have properly configured your Container Adapter as described elsewhere in this reference guide.

Once you successfully receive the above message, return to the sample application's home page and click "Manage". You can then try out the application. Notice that only the contacts available to the currently logged on user are displayed, and only users with `ROLE_SUPERVISOR` are granted access to delete their contacts. Behind the scenes, the `MethodSecurityInterceptor` is securing the business objects. If you're using `acegi-security-sample-contacts-filter.war` or `acegi-security-sample-contacts-cas.war`, the `FilterSecurityInterceptor` is also securing the HTTP requests. If using either of these WARs, be sure to try visiting `http://localhost:8080/contacts/secure/super`, which will demonstrate access being denied by the `FilterSecurityInterceptor`. Note the sample application enables you to modify the access control lists associated with different contacts. Be sure to give this a try and understand how it works by reviewing the sample application's application context XML files.

The Contacts sample application also include a `client` directory. Inside you will find a small application that queries the backend business objects using several web services protocols. This demonstrates how to use Acegi Security for authentication with Spring remoting protocols. To try this client, ensure your servlet container is still running the Contacts sample application, and then execute `client marissa koala`. The command-line parameters respectively represent the username to use, and the password to use. Note that you may need to edit `client.properties` to use a different target URL.

Please note the sample application's `client` does not currently support CAS. You can still give it a try, though, if you're ambitious: try `client _cas_stateless_ YOUR-SERVICE-TICKET-ID`.

24.2. Tutorial Sample

Whilst the Contacts Sample is quite advanced in that it illustrates the more powerful features of domain object access control lists and so on, sometimes you just want to start with a nice basic example. The tutorial sample is intended to provide this for you.

The compiled tutorial is included in the distribution ZIP file, ready to be deployed into your web container. Authentication is handled by the DaoAuthenticationProvider, using the in-memory `UserDetailsService` that sources information from the `users.properties` file located in the WAR's `/WEB-INF` directory. The form-based authentication mechanism is used, with the commonly-used remember-me authentication provider used to automatically remember the login using cookies.

In terms of authorization, to keep things simple we've configured the tutorial to only perform some basic web filter authorization. We've wired two common pre-invocation access decision voters, being the `RoleVoter` and `AuthenticatedVoter`, such that `ROLE_*` configuration attributes and `IS_AUTHENTICATED_*` configuration attributes may be used. Of course, it's extremely easy to add in other providers, with most users probably starting with some services-layer security using MethodSecurityInterceptor.

We recommend you start with the tutorial sample, as the XML is minimal and easy to follow. All of the needed filters are configured properly, and using best practise. Most importantly, you can easily this one XML file (and its corresponding `web.xml` entries) to your existing application. Only when this basic integration is achieved do we suggest you attempt adding in method authorization or domain object security.

Chapter 25. Community Support

25.1. Use JIRA for Issue Tracking

Acegi Security uses JIRA to manage bug reports and enhancement requests. If you find a bug, please log a report using JIRA. Do not log it on the support forum, mailing list or by emailing the project's developers. Such approaches are ad-hoc and we prefer to manage bugs using a more formal process.

If possible, in your JIRA report please provide a JUnit test that demonstrates any incorrect behaviour. Or, better yet, provide a patch that corrects the issue. Similarly, enhancements are welcome to be logged in JIRA, although we only accept commit enhancement requests if you include corresponding unit tests. This is necessary to ensure project test coverage is adequately maintained.

You can access JIRA at http://opensource.atlassian.com/projects/spring/secure/BrowseProject.jspa?id=10040.

25.2. Becoming Involved

We welcome you to become involved in Acegi Security project. There are many ways of contributing, including reading the mailing list and responding to questions from other people, writing new code, improving existing code, assisting with documentation, developing samples or tutorials, or simply making suggestions.

Please read our project policies web page that is available on Acegi Security home page. This explains the path to become a committer, and the administration approaches we use within the project.

25.3. Further Information

Questions and comments on Acegi Security are welcome. Please use the Spring Community Forum web site at http://forum.springframework.org for all support issues. Remember to use JIRA for bug reports, as explained above. Everyone is also welcome to join the Acegisecurity-developer mailing list and participate in design discussions. It's also a good way of finding out what's happening with regard to release timing, and the traffic volume is quite light. Finally, our project home page (where you can obtain the latest release of the project and convenient links to Subversion, JIRA, mailing lists, forums etc) is at http://acegisecurity.org.