

Authentication and Authorization

Jive SBS has multiple facilities to handle the three primary facets of network application security. This topic will discuss each and highlight APIs commonly of interest to developers customizing installations.

This topic describes the framework. For an example of how to add customizations based on these technologies, see [Example: Authentication and Authorization](#).

Fundamental Terminology

The following terms are used commonly in the remainder of this section and are outlined here for clarification:

- **User (or Principal)** — Representation of an authenticated party in the application system. Each HTTP request processed by the system is performed as a User. In the case of guest or anonymous access, Jive SBS uses the `AnonymousUser` class to represent the security context of the current request.
- **Federated User** — A user belonging to a system of record external to the installation. For example, if configured for LDAP authentication, user representations are federated.
- **External User** — An external user is a federated user who has no formal account in the system with which to authenticate.
- **Authentication** — The act of identifying a user given a set of credentials.
- **Credentials** — Data used by an untrusted user to prove their identity to the system. Examples of credentials include passwords, X509 certificates and SAMLRequests.
- **Application Security Layer** — A series of J2EE Servlet Filters which examine every inbound request and enforce security constraints.
- **Application Layer** — Application-level code downstream of the Application Security Layer. Each HTTP request must pass through the Application Security Layer before entering the Application Layer.

Supporting Libraries

Jive SBS relies on common security patterns established in the Spring Security (formerly Acegi Security) library. By leveraging Spring Security, the application uses terminology familiar to Spring users in an effort to standardize integration and leverage existing Spring libraries and idioms.

Authentication

Fundamentally, authentication is performed by a series of Spring Security filter (implementations of J2EE Servlet Filters) chains, linked together. Each element in a given chain has a dedicated responsibility, while each chain is responsible for accomplishing high-level goals towards the handling of a request. Ultimately, these chains must prepare a request to fulfill a single contract enforced by the last link in the primary security filter chain.

Security Context

Each thread of execution, including background jobs and asynchronous tasks, is associated with a Spring Security `SecurityContext` instance. The `SecurityContext` holds information about the Authentication associated with the request.

Internally within application code, Jive extends the Spring Security `Authentication` interface with the `JiveAuthentication` class. This class serves a number of purposes, including directly exposing a User implementation representing the current user through a strongly-typed contract as well as exposing meta data about the user such as whether or not the user is anonymous.

URI Mappings

Each URI handled by the system passes through a series of J2EE Servlet Filters at the Application Security Layer before entering the Application Layer. The following URI contexts are defined in a standard installation:

- `/upgrade/**` - All upgrade requests are handled by the given sequence of filters
- `/post-upgrade/**` - All requests after an upgrade is complete
- `/admin/**` - All requests for the Admin Console
- `/rpc/xmlrpc` - All requests for XML-RPC web services
- `/rpc/rest` - All requests for RESTful HTTP services
- `/rpc/soap` - All requests for SOAP HTTP services
- `/**` - All requests which do not fit one of the above patterns - most requests to an instance are handled by this chain of Servlet Filters

The series of filters handling each request can be altered through the Plugin system when customization of authentication behavior is needed (see below).

Security Filter Chains

Jive SBS defines several Security Filter Chains, each mapped to a specific URL pattern described above. The default filter chain is defined in `spring-securityContext.xml` as the following set of filters:

Place in Chain	Filter Used	Description	As Defined in <code>spring.xml</code>
1	Session Integration filter	Associates HTTP requests with a security context when a user has previously authenticated or entered the system as a guest.	<code>httpSessionContextIntegrationFilter</code>
2	Authentication filters	The default authentication filter is an implementation of Spring Security's <code>FormAuthenticationProcessingFilter</code> which delegates to an internal set of <code>AuthenticationProvider</code> implementations.	<code>formAuthenticationFilter</code>
3	Cookie Authentication filter	Processes "RememberMe" cookies, long-lived HTTP cookies used to authenticate a given user beyond any given session.	<code>rememberMeProcessingFilter</code>
4	Feed Basic Authentication filter	Performs HTTP Basic Authentication of requests for RSS/Atom feeds. It is generally intended to authenticate standalone feed readers and not browser-based requests.	<code>feedBasicAuthenticationFilter</code>
5	Exception Translation filter	Routes redirects of various security-related exceptions to URLs within the application. Security-related exceptions from application-level code are caught and processed by this filter and interceptors in the Struts 2 layer depending on the exception.	<code>exceptionTranslationFilter</code>
6	Authentication Translation filter	Enforces the authentication contract between the Application Security Layer and Application Code.	<code>jiveAuthenticationTranslationFilter</code>

Authentication Contract

The *authentication contract* is a fundamental set of assumptions made by application-level code about the security context of any given request. In a standalone configuration (one in which the application is the system of record for user information), the authentication contract is met by out of the box application functionality. Likewise, for LDAP-based authentication the application fulfills the contract. In the case of custom authentication, third-party code must meet the terms of the contract in order to perform a successful authentication.

The authentication contract is enforced by the last filter in the Security Filter Chain, the `JiveAuthenticationTranslationFilter`. This ensures that the authentication associated with the `SecurityContext` is a valid `JiveAuthentication` before transferring control of the request handling to the application layer downstream.

The contract between the application security layer and the application layer requires that one of the following is true before control is passed from the security layer to the application layer:

- The `SecurityContext` of the request contains an instance of the `JiveAuthentication` interface (established through the `SecurityContext.setAuthentication` method).
- The `Authentication` associated with the `SecurityContext` returns `true` for `isAuthenticated()` and an implementation of Jive's `User` interface is present in either the `getPrincipal()` method or in the `getDetails()` method.

As part of the authentication contract, if no authentication is present when the `JiveAuthenticationTranslationFilter` is invoked, the `AnonymousAuthentication` will be set to the `SecurityContext` prior to transferring control to the application layer. As a result, application-level code needn't check to see if user references obtained from the `SecurityContext` are null.

Jive SBS includes several implementations of the `JiveAuthentication` interface, a subclass of Spring Security's `Authentication` interface. Most commonly used is `JiveUserAuthentication` which requires an implementation of the Jive `User` interface as its sole constructor argument.

As an example, once a handle to a `User` implementation has been obtained (directly created or through the `UserManager` API), that implementation instance can fulfill the authentication

contract by creating an instance of `JiveUserAuthentication` and setting that instance to the `SecurityContext`.

```
UserTemplate ut = new UserTemplate();  
SecurityContextHolder.getContext().setAuthentication(new JiveUserAuthentication(ut));
```

Authorization

Authorization in Jive SBS is addressed via three constructs in the Application Layer.

1. Permissions - The admin console provides a user interface with which to grant a series permissions to users or groups. Permissions are granted on containers within the application. Containers include communities, blogs and social groups.
2. Groups - Groups act as a union of permissions and users within the system. Permissions may be assigned to a group, and all users belonging to that group will be entitled to the group permissions unless overridden by more-specific user permissions.
3. Proxies - Proxies secure application layer objects by restricting access to operations on those objects to users with the appropriate permissions. The proxying of application layer objects is generally transparent and does not impact security layer code.

Permissions behavior is governed by the `PermissionsManager` API, group membership by the `GroupManager` API. Proxies are used to secure access to application API methods and domain objects as they move through the system. Proxies enforce security based on the Acegi `SecurityContext` associated with a request. Jive SBS associates instances of an Acegi subclass — `JiveAuthentication` — with each request by the time the servlet stack leaves the filter chain. That `JiveAuthentication` contains the effective user for the current call stack, which is in turn used to drive proxy authorization checks.