RFP-164 Authentication

Draft

12 Pages

Abstract

This RFP requests a service that can be used to authenticate a user (human or machine) by providing a trusted user id. The service must support the popular authentication schemes like user id/password, OAuth, Kerberos, LDAP, JEE LoginModule, etc. A primary aspect of this authentication service is that it must allows the server code of an application to authenticate a user without becoming entwined in the actual scheme used.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	*24-11-14*	*Initial* *Peter.Kriens@aQute.biz*

1 Introduction

This RFP originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

2 Application Domain

Today virtually all *applications* face an Internet directly or indirectly and therefore must restrict access to its *actions* depending on the *user* that controls the application, where the user can be a human being or another application. To control access, the application must be able to establish a *trusted identity* for this user that is unique within the *realm* that the application lives in. This trusted identity can then be used to associate information about this user. For example, the application can then consult an *authority* to find out if a given action is allowed for this identity.

In general, authentication takes place during a *request*. A *request processor* then executes this request in a context that provides access to the trusted user identity. Since the cost of an authentication can be high, authentication is only performed the first time a user uses the system or after the session expires. An examples of a request processors is a servlet. In the servlet model, authentication is often done with a Servlet Filter.

The process of obtaining the trusted identity is called *authenticating* the user. Users must demonstrate to the authenticator that they are entitled to their trusted identity. The simplest identity is a user name. A password is then a shared secret between the user and the authenticator. If a user can demonstrate knowledge of the secret password, then the authenticator can vouch for this user id (or provide another trusted id).

An *authentication scheme* defines the *roles* and *credentials* used in establishing the trusted identity. In OAuth 2.0, a very popular authentication standard, there is the `Resource Owner`, the `Client Application`, a `Resource Server`, and the `Authorization Server` roles. Credentials are the tokens that are exchanged between the user and the authenticators to demonstrate their identity. Authentication schemes are often coupled to additional information about the user. Authenticators can also provides the authority to execute certain actions and/or user profile information.

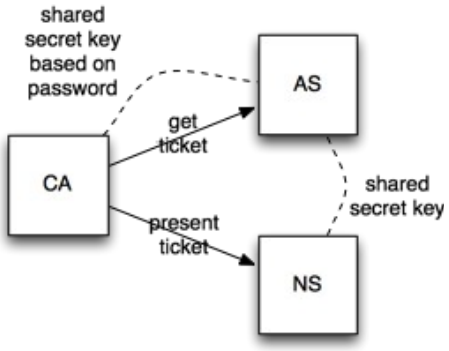
Authentication Schemes come in many forms and are in principal not limited in what they can demand the user to do. The following sections define common authentication schemes.

2.1.1 Basic Authentication

Basic authentication is the scheme built in to every web browser. If the browser receives a security exception from the server it will pop up a dialog and requests the user for a user id and password. These credentials are then base-64 encoded and placed in the WWW-Authorization header. The server decodes this header and use the uses the decoded credentials to establish a trusted identity for this request. This scheme is not secure unless the connection is encrypted.

2.1.2 Kerberos

Kerberos[3] is a popular protocol to authenticate (and authorize) users to execute actions on networked services (NS). It is based on passwords or public keys. An Authentication Server (AS) stores the user credentials. The user logs into a client application that then obtains a ticket for a given network service from the AS. This ticket contains the user id and any authorizations for the (user x NS). This ticket can then be presented to the NS who can then cryptographically verify that the given user id and authorizations can be trusted.

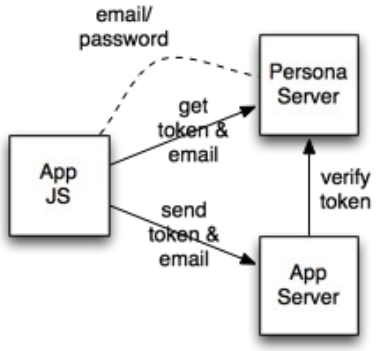


2.1.3 Mozilla Persona

Mozilla Persona (formerly Browser ID) is a network service from the Mozilla foundation geared towards authenticating users in a browser. The purpose is to allow a user to provide a verified email address to an application without sharing a secret with that application. The raison d'être is to minimize the number of secrets a user has to maintain.

The application code in the browser calls the browser id Javascript library. This opens up a new window that provides a secure connection to the Persona servers. The user selects an email as user id and authenticates itself to the user with a password. When successful, the Persona server creates a temporary token that is returned, together with the email, to the Javascript application code. This code then forwards the token and the

email to the application server. The server then sends the token to the Persona server over a secure line to verify the validity of the token.

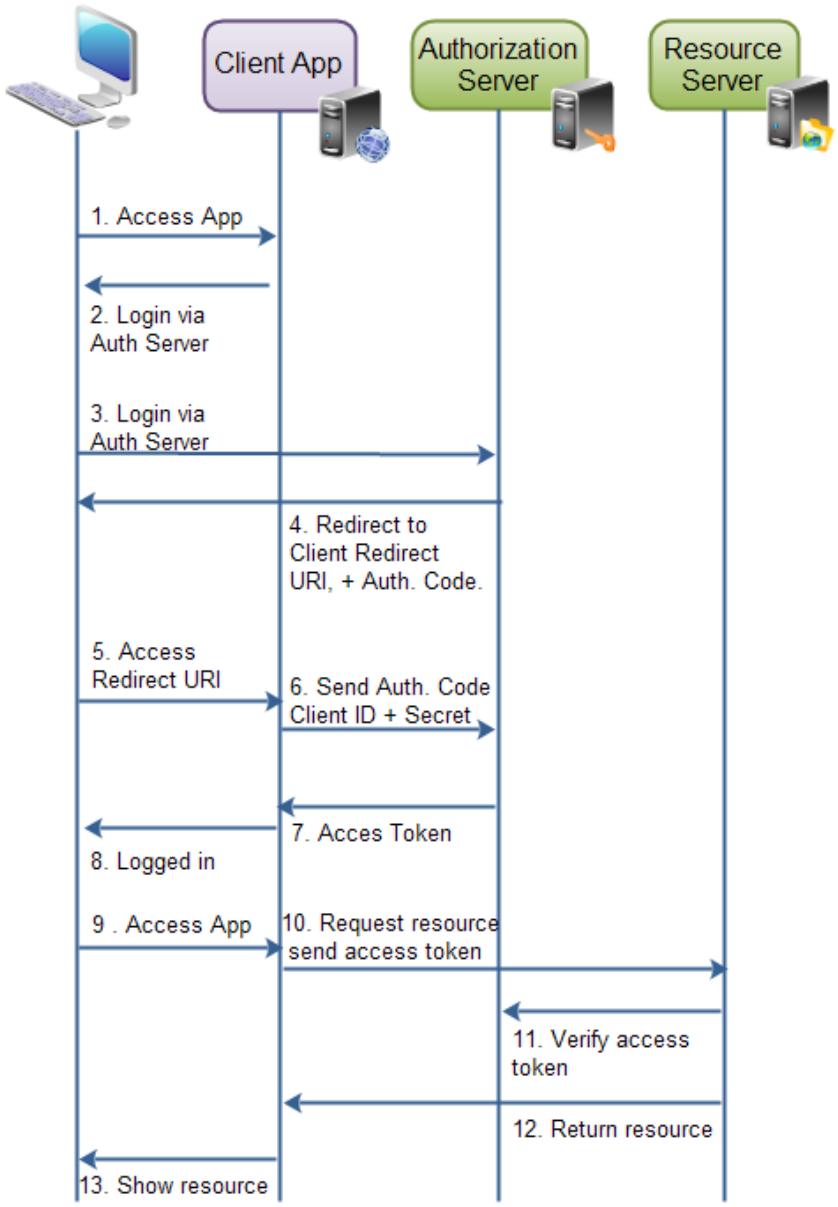


2.1.4 OAuth 2.0

OAuth 2 is an authentication and authorization scheme developed to address the problem of delegated authorization. Users of Facebook or google want to authorize third party applications to act on their behalf. Sharing passwords for this purpose would make it very hard to withdraw an authorization since it would require changing the password on the master side and all third party applications that were authorized. Also, users tend to reuse the same password creating an interesting venue of attack.

OAuth is primary web based where the Application and the Authentication Server use web redirects to create a seamless flow for the user. To authenticate, users are redirected to the authentication provider where they can then login with a password or some other means. If the user is successfully logged in, the browser is redirected back to the application with a URL that contains the user id and authentication token. The Application Server then sends this authentication code to the Authentication Server and receives an access token.

When the application access a resource it hands over the access token. The Resource Server verifies this access token with the Authentication server. It then executes the action if granted permission.



(from http://tutorials.jenkov.com/oauth2/authorization.html)

2.1.5 JAAS & Login Module

Java provides the Login Module interface that is geared to authentication. This security subsystem provides a number of implementations in all JVMs.

JndiLoginModule

The module prompts for a username and password and then verifies the password against the password stored in a directory service configured under JNDI.

- `KeyStoreLoginModule` – Provides a JAAS login module that prompts for a key store alias and populates the subject with the alias's principal and credentials.

- `Krb5LoginModule` – This LoginModule authenticates users using Kerberos protocols.
- `LdapLoginModule` – This LoginModule performs LDAP-based authentication.
- `NTLoginModule` – This LoginModule renders a user's NT security information as some number of Principals and associates them with a Subject.
- `NTSystem` – This class implementation retrieves and makes available NT security information for the current user.
- `SolarisLoginModule` – As of JDK1.4, replaced by `UnixLoginModule`.
- `SolarisSystem` - This class implementation retrieves and makes available Solaris UID/GID/groups information for the current user.
- `UnixLoginModule` – This LoginModule imports a user's Unix Principal information (UnixPrincipal, UnixNumericUserPrincipal, and UnixNumericGroupPrincipal) and associates them with the current Subject.

Login Modules are created by a Login Context. This Login Context is in general configured using System properties and other global singletons. The login context is passed a callback handler that is used when the Login Module requires authentication information from a user, e.g. the user id and a password. The type of the callback (it must implement the empty marker interface) defines what kind of callback information the callback should provide. For example, the standard PasswordCallback should answer the password and the NameCallback the name of the user.

2.1.6 SASL

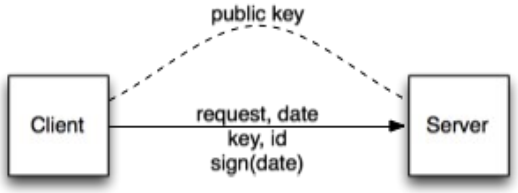
Simple Authentication and Security Layer (SASL) is a framework for authentication and data security in Internet protocols. It decouples authentication mechanisms from application protocols, in theory allowing any authentication mechanism supported by SASL to be used in any application protocol that uses SASL. Java Vms provide a number of SASL server mechanisms.

The SASL server support in Java consists of:

CRAM-MD5
DIGEST-MD5
GSSAPI

2.1.7 Signed Requests

AWS and others use a signed requests model. In this model, an HTTPS request is signed in such a way that the receiver can verify the identity of the sender. In this model, the URL contains the details of the request and the user id. During sending, the Date header is signed with the AWS secret. On the receiving side, the secret is obtained for the user id and then the signature is verified agains the date header. If the date is fresh then the request is assumed to come from the claimed user.



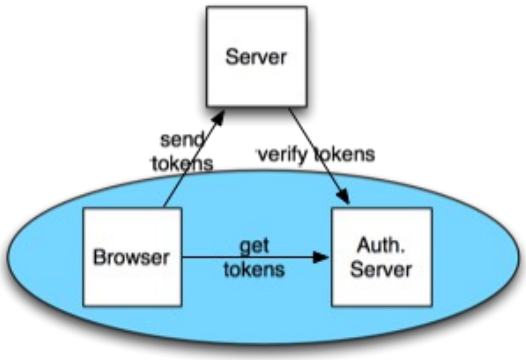
2.2 Terminology + Abbreviations

3 Problem Description

Java has excellent implementation support for popular authentication mechanisms. However, as with many Java APIs, the way these implementations are used is riddled with class loader based factories, singletons, and surprising complexity. Additionally, the Login Modules are hard to use with browser based authentication schemes.

The collaboration scenario for all described authentication schemes is that the browser collects a token that is then verified by the server using an external communication channel or local file, a quite simple scenario.

The perfect OSGi authentication model is installing a bundle that then provides a way for a request processor to establish the trusted identity for the requestor. The problem in this model is coupling. The different authentication schemes require different information and this is multiplied by the variation in request processors. In an ideal world, the request processor should not have to be aware of what authentication scheme is used. That said, any authentication scheme will require some user interface that is unique to the scheme. Java has tried to abstract this UI aspect in the CallbackHandler but this method is awkward to use and often collides with the model that the GUI runs in a remote web browser.



The key goal of this RFP is to obtain a service model where the server code can remain generic. It is assumed that the server will receive credentials from a login module (likely running in the browser) that is specific for the authentication scheme in use but will be able to interpret those credentials itself. It will then verify these credentials agains a service that must recognize the type of the credentials and then perform the authentication scheme specific verification, returning a user id if it succeeds.

The request processor can then assume that the user id is trusted since the authentication module is accessible to it.

Therefore, the primary goal of this RFP is to find a solution that allows a server to authenticate a user without the server code having to be aware of the actual scheme used. Though the primary request processor is envisioned to be Servlets, the solution must not require servlets to allow other types of request processors.

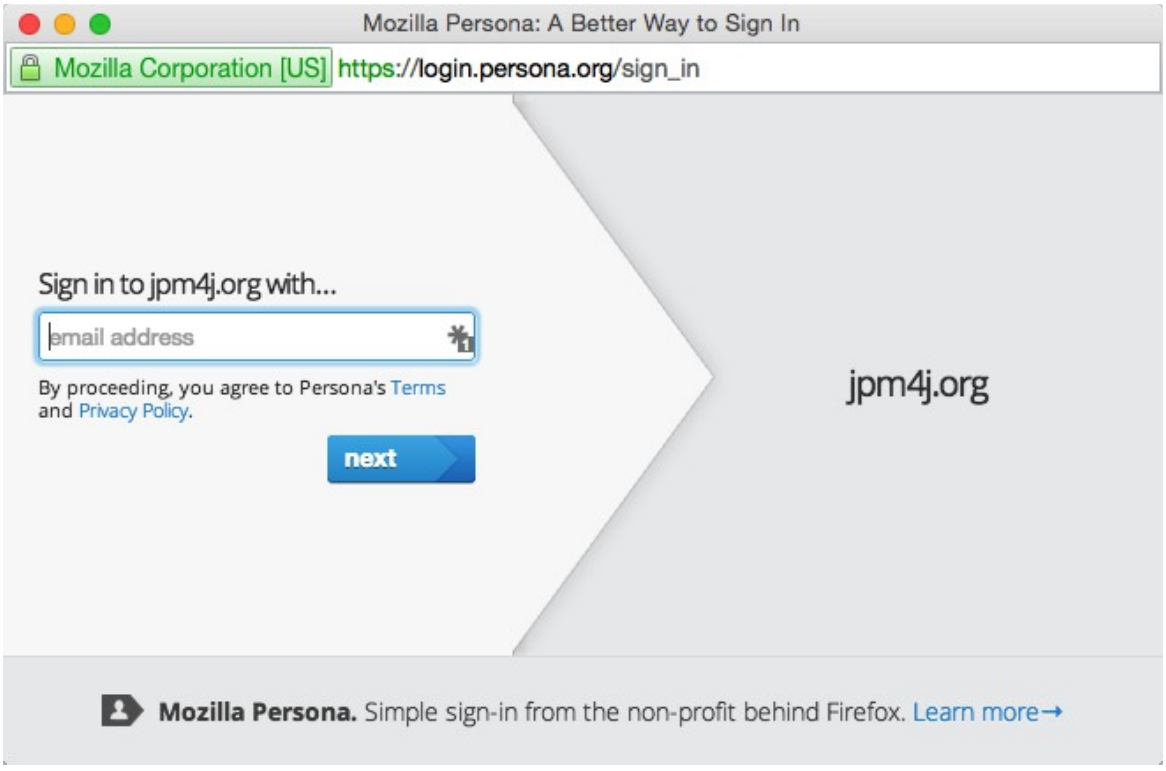
4 Use Cases

4.1.1 Persona

The jpm4j.org web site uses Mozilla Persona to log in. The site uses a single page web app based on angular. The user can login by clicking on the login icon on the left top, this calls the login code.

```
function getAssertion() {
  navigator.id.get(callback, {siteName: "JPM4J"});
}
```

This opens a new window (unconnected to JPM) on the Persona web site.



Since Persona can be used by many different applications it lists 3 email addresses for John Doe, our user. John selects jd@morgue.com and provides his password. The password is correct and the window closes. The browser id calls the callback function with an assertion, which is an opaque token.

The callback function POSTS the opaque token to the server on the /login url with the assertion as a parameter. A Servlet Filter behind this URL then consults all authentication services in the registry. In this case, the browser id implementation recognizes the assertion object and contacts the Persona server over an SSL connection. The Persona server recognizes the assertion and returns the user's email address, the authenticated id. The Servlet Filter then sends the user id in the servlet session so that can become used by the servlets as long as the session does not expire.

The JPM server then uses the email as a trusted id and looks up the profile and authorizations in User Admin.

The browser notices that the login was successful and modifies the user interface accordingly.

John Doe now selects an import action and provides a url to http://example.com/mybundle.jar. The request is send to the server. Another Servlet Filter sees the user id in the session and creates an authority context with the authenticated user and authority information. Since this was a JSON RPC request, the actual action is executed in the JPM service. This service verifies if the user has the required authorization and executes the import.

When John Doe logs out, the login filter removes the trusted user id from the session. From then on, the other servlet filter will no longer see a logged in user and establish an anonymous user context.

4.1.2 Signed Requests

John Doe wants to submit JARs to JPM directly from a continuous integration server. Since John is not present during the build to provide a password he uses bnd to create a private key on the machine. John then forces a release to JPM. Since there is a private key, the HTTP request gets an additional header containing the date-time, the public key, and the associated email address of John Doe. This header is then signed by another header using the private key.

When JPM receives the request, it iterates over the authenticators. The second authenticator recognizes the headers. It verifies the proper signing and then looks up the public key. Since this was never used before, the request is denied. However, the authenticator records the public key and IP address with the John Doe's profile as a potential delegate.

The release fails since the headers could not be authenticated.

John Doe now logs into JPM and goes to its delegates window. There he checks the 'allow' checkbox for the just added public key.

He then releases again. This time the authenticator sees that the public key is allowed for that email address and authenticates the user.

4.1.3 OSGi enRoute Authentication

The OSGi enRoute project has developed an implementation of an Authentication API [7]. that closely follows the model of this RFP, actually, was inspired by it.

5 Requirements

5.1.1 General

- G00010 – The solution must support multiple independent authenticators in an OSGi framework
- G00020 – It must be possible to order authenticators
- G00030 – It must be possible to implement authenticators for OAuth 1, OAuth 2, LDAP, Kerberos, password, Persona, Unix, Windows NT, and signed requests.
- G00040 – The solution must work in a Servlet/Servlet Filter based environment
- G00050 – It must be possible for the request processor to indicate to the request processor that the user id is no longer used in the system, e.g. when the user gets logged out. The rationale is that sometimes authenticators are tightly integrated with the authority module.
- G00060 – It must be possible to leverage the existing Login Module implements both available in the VM and outside with minimal adaptations.
- G00070 – There must be no dependency on the Servlet API, it must be possible to use the solution for applications other than servlets.
- G00080 – Solutions must allow Java security to restrict access to the authenticators
- G00090 – Authenticators must not require special permissions from the callers

5.1.2 Optimizations

- O00010 – The solution should attempt to be compatible with the OSGi enRoute Authentication API
- O00020 – The solution will primarily be used for web based applications. It should therefore optimize for servlets and single page apps.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. Kerberos. http://technet.microsoft.com/en-us/library/cc961976.aspx

[4]. http://en.wikipedia.org/wiki/Simple_Authentication_and_Security_Layer
[5]. http://www.ietf.org/rfc/rfc2853.txt
[6]. https://docs.oracle.com/javase/7/docs/technotes/guides/security/sasl/sasl-refguide.html
[7]. http://enroute.osgi.org/services/osgi.enroute.authentication.api.html

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33 467542167
e-mail	Peter.kriens@aQute.biz

6.3 End of Document