JXSE v2.7
The JXTA Java™ Standard Edition Implementation

Programmer's Guide

by Jérôme Verstrynge

March, 2011

Table of Contents

Introduction

This document is a follow-up on the JXSE 2.5 and 2.6 Programmers Guide. Its purpose is to describe the new features which have been made available as part of the 2.7 release of the JXTA implementation in the Java Programming language (JXSE). This release is also known as JXSE 2.7.

The basics of JXTA can be found in the JXSE 2.5 and 2.6 Programmer's Guide.

Acknowledgments

I would like to thanks all those who contributed code or other software engineering skills and knowledge to the 2.7 release (including in writing this document):

- Adrian Ivan
- Ariel Ro
- Cees Pieters
- Iain McGinniss
- John Boyle
- Mike (Bondolo) Duigou
- Nick (a.k.a. Buzz Lightyear)
- Simon Temple
- William (Bill) Middleton

Highlights

- *Continuous Integration* – A continuous integration system for JXSE has been implemented via http://mikeci.com/.
- *Hint & Route Suggestions* – Hints objects have been removed from the code. A replacement method has been made available via the EndpointService.
- *Http2* – A new http transport has been implemented next to the current http transport. Amongst other, it solves peer shutdown delays encountered with the current http transport.
- *JUnit Collocated Tests* – A new set of automated JUnit tests has been implemented to test connectivity between peers and most of their basic functionalities. This helps dodging regression issues and significantly improves quality insurance.
- *JXTA Security Enhancement* – A long awaited implementation of secured peergroups and communication is now available.
- *Multiple Peers in Same JVM* – It is now possible to run multiple peers in the same JVM. This feature is used by the collocated tests.
- *Release Process Documentation* – Now that JXSE is available from Maven, via a repository offered by http://www.sonatype.com/, a complete release process documentation is available in the Appendix.

What Is New? What Goes?

This section introduces the new features implemented in the 2.7 release, and those which were removed, in alphabetic order.

Continuous Integration

A continuous integration system has been implemented with the help of http://www.mikeci.com. For more information, send an email on the dev list.

Hint & Route Suggestion

Until this release, the code was cluttered with route hint objects. Most of these were passed from methods to methods and most often not used in corresponding code. As a consequence, hint code has been completely removed from JXSE.

If a software engineer still wants to suggest a route to a peer to the system, he/she can do so by calling:

```
...
MyPeerGroup.getEndpointService().getEndpointRouter().suggestRoute(myRoute);
...
```

The system will check for the quality of the route and keep it locally in its cache for later use, if valid and useful. Else, it will discard it.

REM: Software engineers do not to feed the system with routes to make sure connectivity happens. This should be achieved with proper RDV and Relay seeds. Peers automatically exchange/fetch routes as necessary.

Http2

A new implementation of the HTTP transport is now available in 2.7. It is not compatible with the current Http transport. Endpoint adresses types are different. In theory, both Http and Http2 could run on the same peer together, by connecting to different ports.

Solving Shutdown Delays

Http2 is built upon the Netty library, while current Http transport is built on top of Jetty. The community knows, since release 2.5, that there is delay caused by the Jetty library when shutting down a peer running http. Http2 solves this issue. Therefore, is it recommended to use Http2 rather than Http in JXSE 2.7 applications.

Configuration

The `NetworkConfigurator` has been enhanced to enable Http2 configuration. The new methods are identical to existing methods for current http transport. A `JxseHttp2TransportConfiguration` class has been implemented too and integrated in the `JxsePeerConfiguration` class[1].

Junit Collocated Tests

The set of Junit tests has been cleaned-up. Some tests have been disabled, because they significantly slow down the compilation cycle. In the future, a mechanism will be implemented to enable or disable those tests at will.

This 2.7 release is a turning point regarding automated tests for main functionalities and peer connection modes. Since multiple peers can now run in the same JVM, it makes automated tests a breeze. A set of collocated test has been implemented to make sure that new patches don't break existing functionalities.

Overall, Junit tests currently do not directly cover for all the code, but indirectly, most of the code base is covered.

Memory Leak

1 See the net.jxse.configuration package.

A memory leak has been detected in the in-memory implementation of the SRDI in release 2.6 (c.f. `TernarySearchTreeImpl` class). This has been fixed in release 2.7.

Jxta Security Enhancements

JXTA security has been improved by allowing advertisements and messages to be signed when stored or communicated between peers. This new feature solves one of the weak points of JXTA: cache pollution and (some) man-in-the-middle attacks.

This comes on top of existing secured communication discussed further in this document. Those not familiar with secured communication in JXTA/JXSE should read the 'Under The Hood' section further in this document, before reading the following pages.

Alice & Bob Example

The following simple example explains how to use secured communication between two peers after proper configuration has been set-up. It is extracted from test code:

```
public class DefaultSecureTCPMessageTest extends DefaultTCPMessageTest {

    public static void main(String[] args) {
        try {
            DefaultSecureTCPMessageTest t = new DefaultSecureTCPMessageTest();
            t.init();
            t.run();
            t.end();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    @Override
    protected void run() {
        try {

            PeerID alicePeerID = aliceManager.getNetPeerGroup().getPeerID();
            PeerID bobPeerID = bobManager.getNetPeerGroup().getPeerID();

            PSEMembershipService alicePSEMembershipService =
 (PSEMembershipService)aliceManager.getNetPeerGroup().getMembershipService();
            PSEMembershipService bobPSEMembershipService =
 (PSEMembershipService)bobManager.getNetPeerGroup().getMembershipService();

            PSEConfig alicePSEConfig =
                alicePSEMembershipService.getPSEConfig();
            PSEConfig bobPSEConfig = bobPSEMembershipService.getPSEConfig();

            X509Certificate aliceCertificate =
                alicePSEConfig.getTrustedCertificate(alicePeerID);
            X509Certificate bobCertificate =
                bobPSEConfig.getTrustedCertificate(bobPeerID);

            alicePSEConfig.setTrustedCertificate(bobPeerID, bobCertificate);
            bobPSEConfig.setTrustedCertificate(alicePeerID, aliceCertificate);
            testColocatedPeerBidiPipeComms(true);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Two peers Alice and Bob are started. They exchange their public certificates by extracting them from their respective membership service's PSE configuration, using their peer IDs. Typically, these certificates would be exchanged over the network or provided/fetched from an online key storage in a real application.

Next, they perform the typical bidirectional pipe communication test.

Signing & Verifying Advertisements

A set of new methods has been implemented for advertisements:

```
...
public boolean sign(PSECredential pseCredential, boolean includePublicKey,
    boolean includePeerID);

public boolean verify(PSECredential pseCredential,
    boolean verifyKeyWithKeystore)

public boolean isAuthenticated()

public boolean isCorrectMembershipKey()

public boolean isMember()

public XMLSignature getSignature();
...
```

- *Sign* – Signs the corresponding advertisement using the provided peergroup membership credentials. If the public key is included then the signature can be verified. If the peer id is present, then the certificate can be looked up in the PSE Membership keystore corresponding to the credentials. The method returns true if the signature was successful.
- *Verify* – Indicates whether the advertisement has been tampered with or not. The `verifyKeyWithKeystorePSE` parameter is not used.
- *IsAuthenticated* – Indicates whether the advertisement has been successfully signed or verified.
- *IsMember* – Reveals whether the signing peer is member of the PSE membership keystore.
- *GetSignature* – Provides the valid signature of this advertisement if available.

REM: The method used to sign advertisements is “SHA1”. This is, unfortunately hard-coded. Ideally, this will be made configurable in the future.

PSE Membership & Keystore Configuration

The configuration of the PSE membership service must happen before peer services instances are created (that is, before the JXTA network is started).

Four factory setter/getter pairs of static methods are now available to configure the PSE Membership service. The provided objects are registered in a static variables (if they were not initialized before). It is a one shot operation with no safety net and no second chance.

```
...
public static void setPSEAuthenticatorEngineFactory(...)  { ... }

// ...and correponding factory interface
public interface PSEAuthenticatorEngineFactory {
    PSEAuthenticatorEngine getInstance(PSEMembershipService service,
        PSEConfigAdv config) throws PeerGroupException;
    }

...
public static void setPSEKeyStoreManagerFactory(...) { ... }

// ...and correponding factory interface
public interface PSEKeyStoreManagerFactory {
    KeyStoreManager getInstance(PSEMembershipService service,
        PSEConfigAdv config) throws PeerGroupException;

...
public static void setPSEPeerValidationEngineFactory(...)  { ... }

// ...and correponding factory interface
public interface PSEPeerValidationEngineFactory {
    PSEPeerValidationEngine getInstance(PSEMembershipService service,
        PSEConfigAdv config) throws PeerGroupException;

...
public static void setPSESecurityEngineFactory(...)  { ... }

// ...and correponding factory interface
public interface PSESecurityEngineFactory {
    PSEPeerSecurityEngine getInstance(PSEMembershipService service,
        PSEConfigAdv config) throws PeerGroupException;
...
```

If a getter method is called before its corresponding setter, the corresponding static variable will be initialized with a default instance of the corresponding object.

Each `getInstance(...)` method takes in two parameters: a PSE membership service instance and a PSE configuration advertisement. This enables control on the object instance returned by the factory according to the provided membership service and PSE configuration[2].

In addition to these methods, an extra pair of static factory setter/getter has been implemented within the `StdPeerGroup`[3] object:

```
...
```

2 From the PSE configuration advertisement Javadoc: *"The configuration advertisement can include an optional seed certificate chain and encrypted private key. If this seed information is present the PSE Membership Service will require an initial authentication to unlock the encrypted private key before creating the PSE keystore. The newly created PSE keystore will be "seeded" with the certificate chain and the private key."*

3 As a reminder, this is the actual/standard object implementing the PeerGroup API interface for PeerGroup object instances.

```
public static void setPSEMembershipServiceKeystoreInfoFactory(...) {...}

// Correponding factory interface
public interface PSEMembershipServiceKeystoreInfoFactory {
    PSEMembershipServiceKeystoreInfo getInstance(PeerGroup peerGroup)
        throws PeerGroupException;
}
...
```

PSE Membership Service Keystore Info

The objects returned by the PSE membership service keystore info factory implement the following interface:

```
...
public interface PSEMembershipServiceKeystoreInfo {
    PeerGroup getPeerGroup();
    String getAuthenticationType();
    String getPassword();
}
...
```

Since keystores are often locked with a password, the factory can return objects providing information to unlock these. By default, the system sets up the membership authentication type to “StringAuthentication” and the password to “the!one!password” across all peergroups.

These settings can be changed via:

```
...
System.setProperty("impl.membership.pse.authentication.type",
    "StringAuthentication");
System.setProperty("impl.membership.pse.authentication.password",
    "the!one!password");
...
```

REM: if a password is set in `NetworkConfigurator` then the System properties are ignored !!!

The corresponding factory enables the automatic joining of `PSEMembershipService` each time a new PeerGroup is created.

Authenticator Engine

Back to the PSE membership static factories, an authenticator engine has the following API:

```
public interface PSEAuthenticatorEngine {

    public PublicKey getPublicKey() throws SecurityException;

    public byte[] sign(byte[] data)
        throws InvalidKeyException, SignatureException, IOException;
    public String getSignatureAlgorithm();

    public boolean isEnginePresent() throws SecurityException;

    public X509Certificate getX509Certificate() throws SecurityException;

    public char[] getKeyPass(PeerGroup peerGroup) throws SecurityException;

    public char[] getStorePass(PeerGroup peerGroup) throws SecurityException;

}
```

These object are used by `EngineAuthenticator`, that is, a type of authenticator delivered by JXSE.

Keystore Manager

A keystore manager has the following API:

```
public interface KeyStoreManager {

    /**
     * Returns true if the KeyStore has been initialized (created).
     * Since this method does not provide a passphrase it is really only
     * useful for determining if a new KeyStore needs to be created.
     */
    boolean isInitialized() throws KeyStoreException;

    /**
     * Returns true if the Keystore has been initialized (created).
     * This method also ensures that the provided passphrase is valid for the
     * keystore.
     */
    boolean isInitialized(char[] password) throws KeyStoreException;

    /**
     * Create the KeyStore using the specified KeyStore passphrase.
     */
    void createKeyStore(char[] password)
        throws IOException, KeyStoreException;

    /**
     * Load the KeyStore.
     */
    KeyStore loadKeyStore(char[] password)
        throws IOException, KeyStoreException;

    /**
     * Save the provided KeyStore using the specified KeyStore passphrase.
     */
    void saveKeyStore(KeyStore store, char[] password)
```

```
        throws IOException, KeyStoreException;

    /**
     * Erase the KeyStore. Some KeyStore implementations may not allow the
     * KeyStore container itself to be erased and in some cases specific
     * certificates and keys may be unerasable. All implementations should
     * erase all user provided certificates and keys.
     */
    void eraseKeyStore() throws IOException;

}
```

PSE Peer Validation Engine

This new feature allows the membership service to bind to a Certificate Authority. It is an opportunity for engineers to have more control on security.

```
/**
 * PSEPeerValidationEngine validates the certificate chain of a peer
 * in the pse keystore against a Certificate Authority
 */
public interface PSEPeerValidationEngine {
    void validatePeer(PeerID peerID, X509Certificate[] certList)
        throws CertPathValidatorException;
}
```

REM: The default `PSEPeerValidationEngine` does nothing – no CertPathValidatorException is thrown.

PSE Peer Security Engine

These objects assist with the signing and verifying of streams of bytes. This new feature allows the use of devices with an inaccessible private key (for example SmartCards).

```
public interface PSEPeerSecurityEngine {

    /**
     *  Cryptographically sign an input stream using the specified credential
     *  with the specified algorithm.
     **/
    public byte[] sign(String algorithm, PSECredential credential,
        InputStream bis)
        throws InvalidKeyException, SignatureException, IOException;

    /**
     * Cryptographically verify a signature against an input stream using the
     * specified credential with the specified algorithm.
     **/
    public boolean verify(String algorithm, PSECredential credential,
        byte[] signature, InputStream bis)
        throws InvalidKeyException, SignatureException, IOException;
    /**
     * Generate a new service certificate.
     **/
    public IssuerInfo generateCertificate(PSECredential credential)
        throws SecurityException;

    /**
     * Returns the default signature algorithm for this security engine.
     **/
    public String getSignatureAlgorithm();

}
```

PSE Membership Enhancement

Two methods have been added to the PSE Membership:

```
...
public void validateOffererCredential(
    PSECredential accessServiceOffererCredential)
    throws CertPathValidatorException {...}

public void validateOffererCredential(
    PSECredential accessServiceOffererCredential, String[] aliases)
    throws CertPathValidatorException {...}
...
```

The first method is a convenient one to perform a PKIX certpath validation on the certificate chain (if available in the corresponding keystore). The second method takes the credential and performs a certpath validation against all of the alias certificates found in the keystore.

How Does It All Work?

Once the factories have been set-up (i.e., before starting the JXTA network[4]), all messages sent via the peergroup are signed before sending. On receiving, messages are converted to empty messages if the signature is not valid. The overhead per message is ~800 bytes (encoded certificate, peerid and signature).

Outgoing messages are passed though the `toWireExternal`[5] object. They are appended with the encoded peer certificate and peerid and signed. On destination, `fromWireExternal`[6] verifies the message. `CbjxFilter`[7] checks that the message source endpoint address is valid and not a spoof.

Of course, if nothing is configured, the messages are not signed or verified by default. They bear no overhead. This can be verified by running the code examples provided in the `ServiceClient`[8] and `ServiceServer` classes.

Advertisements can be signed before publishing and verified after discovery.

Encryption Over JXTA Sockets

4 Some engineers will complain that they have to reference implementation code, rather than API only, and rightfully so. There are other parts of the API code explicitely referencing implementation code too. This is a know issue and an area for improvement in JXSE. Guice (see http://code.google.com/p/google-guice/) would most probably help.

5 See `net.jxta.impl.endpoint.WireFormatMessageBinary.toWireExternal`

6 See `net.jxta.impl.endpoint.WireFormatMessageBinary.fromWireExternal`

7 See `net.jxta.impl.endpoint.cbjx.CbjxFilter`

8 See `net.jxta.document.ServiceClient and net.jxta.document.ServiceServer`

So far, we have only mentioned one aspect of cryptography: authentication. The new features delivered as part of JXTA also cover for encryption, on top of existing TLS communication.

Secured communication over JXTA sockets is now available. This approach is still experimental. The idea is to use a cipher to perform encryption on messages sent via Jxta socket.

New constructors have been added to `JXTASocket` and `JXTAServerSocket` to enabled encryption. The `PSEUtils` object has been enhanced with some static methods to facilitate encryption and decryption of messages too. The cipher is hard-coded to “DESede” in `PSEUtil`[9].

To enable encryption, sofware engineers need to set the encryption boolean when creating the `JxtaSocketServer` and `JxtaSocket` objects. This, of course, should follow proper membership service configuration.

Software engineers can check the code examples in the `\contrib\net\jxta\socket\examples` when checking out trunk.

Jxse Configuration Tool (for Legacy)

A new tool class to convert legacy configuration files, called `JxseConfigurationTool`, is now available. It offers two main public static methods:

```
...
public static NetworkManager getConfiguredNetworkManager(
    JxsePeerConfiguration inConfig) throws JxtaConfigurationException,
    IOException {…}

public static JxsePeerConfiguration
    getJxsePeerConfigurationFromNetworkManager(NetworkManager inNM)
    throws Exception {…}
...
```

The first method provides a configured `NetworkManager` object instance from a provided JXSE peer configuration. This instance can be used to start the JXTA Network.

The second method provides a JXSE peer configuration object instance from a `NetworkManager` object. The method extracts the configuration from the corresponding `NetworkConfigurator`. It can be used to generate a JXSE peer configuration object by first creating a `NetworkManager` object based on an old configuration file.

Maven Sonatype

Sonatype offers an automated process to release a version's artifacts in our Sonatype Maven repositories. However, this process is prone to failures. There is also a manual procedure, which we have used to release 2.7, to post artifacts. This procedure has been documented in the apendix.

It is now easier to release versions of JXTA/JXSE and we see more frequent releases (2.6.1, 2.7.1 or 2.7.2 for example).

Multiple Peers in Same JVM

At last, a feature long awaited by the JXTA community is now available: running multiple peers in the same Java Virtual Machine. Yes, you can now call `startNetwork()` on multiple instances of the `NetworkManager()` within the same application.

9 Hopefully, this will be configurable in a future version.

Of course, make sure each instance is configured to listen to different ports. Some engineers have experienced issues when running multicasting, but others don't experience those issues at all. Make sure firewalls on local PC allow multicasting[10].

This feature proves very useful for testing purposes.

OSGi

The level-1 implementation of JXTA released as part of 2.6 is not a true success[11]. We are short of designing a proper API interface offering access to the JXTA functionalities. Some members of the community have started to work on this.

Hopefully, this will be available in a future release.

PeerGroup Publication

The methods offered by the PeerGroup API interface did not give any control on the publication or not of the peergroup advertisement. A new `newGroup(...)` method including a publish boolean has been made available to solve this is issue:

```
...
public PeerGroup newGroup(PeerGroupID gid, Advertisement impl,
 String name, String description, boolean publish) throws PeerGroupException;
...
```

Reference JXTA Loader

The reference JXTA loader is the class responsible for loading JXTA modules in JXSE. It extends the `JxtaLoader` abstract class which extends `URLClassLoader`. Hence, there is no alternative to class loader for implementation of a JXTA loader. Ideally, an API should not be that restrictive.

Time To Toss It?

Typically, the reference JXTA loader attempts to organize a secured way to load JXTA modules (or JXTA applications) on peers. Unfortunately, this approach is ill-borned, since it does not handle the loading of different code versions of a service implementation, amongst others (c.f. previous version of this guide).

OSGi has already solved such issues. It may be time to remove this feature and stop any further efforts aiming at re-inventing an existing wheel. Of course, an alternative to enable the chose amongst different implementation of peer services would be welcome: Guice[12] would fit this requirement.

So, is it time to toss the reference JXTA loader? It probably is. The corresponding code is complex and offer no added operation value. To some extent, it also prevents the proper implementation of an OSGi API interface. Remove the reference JXTA loader would also eliminate the need of multiple module advertisements, which have never proved useful so far.

Removed & Deprecated Code

Config Dialog

The infamous ConfigDialog box, that would pop-up from time-to-time is the application (when it missed some configuration), has now been definitely removed.

Default Community Seeds

10 Microsoft VISTA is known to silently disable multicasting, which is problematic when running multiple peers instances on the same PC.
11 The functionalities where not exported properly in the manifest file. This has been solved in release 2.7.
12 See http://code.google.com/p/google-guice/.

The default community seeds have not been maintained by Sun Microsystems (now Oracle) for quite some time. The code still contained explicit references to URLs. These have been removed. The default seed methods in the `NetworkManager` and the `NetworkConfigurator` have been removed too.

PROXY Module

All code corresponding to the proxy module has been removed or disactivated from the core JXSE project. This code was not maintained, tested and used by the community in recent releases.

This feature was originally implemented to let devices with small capacity access the JXTA network via another more powerfull device capable of running the JXTA stack. This is not to be confused with an EDGE → SUPER peer connection (for example).

The consequences are:

- The tiniest devices will have to run the JXTA stack themselves if they want to participate to the JXTA network. Remember, that a device does not necessarily have to implement/operate the full JXTA stack to participate to the JXTA network.
- Some configuration files generated with earlier version of JXSE may have to be regenerate by software engineers, since they won't be loaded successfully with 2.7.

Service Interfaces

The service interfaces have now been completely been removed to the code, making it much simpler to read and to understand. These interfaces were originally implemented to make sure end users would not call `stop()` on services unecessarily (i.e. leaving the application is an illegal/unsafe state). Unfortunately, these interfaces were also used to implement code which should have remained in the service object itself.

Ultimately, these state issues should not be solved with interfaces, but with a proper API not exposing 'dangerous' methods. Moreover, those methods are typically trying to solve some of the issues any service loader like OSGi would have to solve, but incompletely and unsafely.

Shell Code

The Shell project has not been maintained by the community since somewhere between release 2.4.1 and 2.5. This project was known to be buggy and no one volunteered to take care of it. The JXSE code base still contained some code specifically implemented to accommodate the Shell project.

The application concept in JXTA as opposed to the service concept has already been hard to understand and to justify. The Shell was the only known implementation of the application concept. Ideally, it should have been implemented as a service.

Although JXTA initially aimed at offering functionalities similar to what became OSGi, it never managed to deliver a safe a complete implementation of these. The Shell code implementation was also a clear breach of general recommendations regarding security.

Corresponding code has been definitely removed.

Technical Debt

The 2.7 release has been another opportunity to clean-up the code and work on the technical debt. Many patches have been applied to simplify the code where possible and make it more readable.

FindBugs & PMD

The FindBugs and PMD tools were used to scan core code for potential issues. About 85% of all reported issues were corrected with patches. Some of the remaining issues need significant rework or re-implementation of the code in order to be solved. These do not represent a significant threat to the application.

PeerGroup Resource Sharing

In theory, channel messengers are redundant, since canonical messengers could do the job as well. They could (in theory) be removed from the code, but an attempt revealed that there are cross references between messengers over peergroup instances (REM: peergroups are organized in a parent-child relationship in JXSE).

The corresponding code is very complex. Solving this issue would require rewriting the whole of it in order to prevent the sharing of resources across peergroups. This is also a cause for concern regarding security.

The parent-child relationship between peergroups is questionable implementation[13]. The mathematical group theory does not necessarily imply a parent-child relationship between groups. Some have intersections, some don't, etc...

If one needs to control access to a secure peergroup, via a chain of certificate for example, it does necessarily mean that one should implement a peergroup per level of certificate, under the hood. Messenger resources should not have to be attached to peergroup instances. Ideally, they should be centralized in the application.

13 Some may disagree with this opinion of course.

Hunder The Hood

This chapter provides additional information about what is happening under the hood. It is a follow-up of the corresponding chapter in release 2.6 of the Programmer's Guide.

Membership Service & Access Service

Conceptually, the membership service and the access service in JXTA provide means to implement access control to peer groups. Each peergroup can implement its own membership service along the parent-child relationship between peergroups.

Implementations & Selection

Remember that implementations for the membership service and the access service are defined in the `META-INF.service/net.jxta.platform.Module`:

```
...

#urn:jxta:uuid-deadbeefdeafbabafeedbabe000000050106
net.jxta.impl.membership.none.NoneMembershipService None Membership Service

#urn:jxta:uuid-deadbeefdeafbabafeedbabe000000050206
net.jxta.impl.membership.passwd.PasswdMembershipService   Password   Membership
Service.

urn:jxta:uuid-deadbeefdeafbabafeedbabe000000050306
net.jxta.impl.membership.pse.PSEMembershipService PSE Membership Service

urn:jxta:uuid-deadbeefdeafbabafeedbabe0000000D0106
net.jxta.impl.endpoint.tls.TlsTransport  Reference  Implementation  of  the  TLS
Message Transport

urn:jxta:uuid-deadbeefdeafbabafeedbabe000000100106
net.jxta.impl.access.always.AlwaysAccessService Always Access Service

urn:jxta:uuid-deadbeefdeafbabafeedbabe000000100206
net.jxta.impl.access.simpleACL.SimpleACLAccessService   Simple   ACL   Access
Service

urn:jxta:uuid-deadbeefdeafbabafeedbabe000000100306
net.jxta.impl.access.pse.PSEAccessService PSE Access Service

...
```

The definition of default implementations is defined in the `PeerGroup` interface:

```
...

/* Well known access specification identifier: standard access service */

public final static ModuleSpecID refAccessSpecID =

    ModuleSpecID.create(URI.create(WK_ID_PREFIX + "000000100106"));

/* Well known service specification identifier: the standard membership */

public final static ModuleSpecID refMembershipSpecID =

    ModuleSpecID.create(URI.create(WK_ID_PREFIX + "000000050106"));

...
```

WARNING: However, this does not mean that these defaults will be used when creating peergroup objet instances. The key item defining selected modules for any peergroup instance is its corresponding module implementation advertisement.

Generic Peergroup

The default module implementation advertisement for standard peergroups is returned by the `getDefaultModuleImplAdvertisement()` method of the `StdPeerGroup` class. In this method, the defined services are :

```
...
  paramAdv.addService(PeerGroup.membershipClassID,
      PSEMembershipService.pseMembershipSpecID);


  paramAdv.addService(PeerGroup.accessClassID,
      PSEAccessService.PSE_ACCESS_SPEC_ID);
...
```

That is, the `net.jxta.impl.membership.pse.PSEMembershipService` and the `net.jxta.impl.access.pse.PSEAccessService`.

NetPeerGroup

The default module implementation advertisement for standard peergroups is returned by the `getDefaultModuleImplAdvertisement()` method of the `ShadowPeerGroup` class. In this method, the defined services are :

```
...
  paramAdv.addService(PeerGroup.membershipClassID,
    PSEMembershipService.pseMembershipSpecID);
  paramAdv.addService(PeerGroup.accessClassID,
    PeerGroup.refAccessSpecID);
...
```

That is, the `net.jxta.impl.membership.pse.PSEMembershipService` and the `net.jxta.impl.access.always.AlwaysAccessService`. In other words, if a peer tries to become a member of the `NetPeerGroup`, it will always be accepted, which is a requirement of the JXTA protocol specifications.

General PeerGroup 'Login' Scheme

The general procedure to 'log' into a peergroup is the following:

1. Retrieve the membership service.
2. Create some credentials (using `AuthenticationCredential`).

   2.1. One has to provide a method of authentication. This is specific to the corresponding membership service. For the PSEMembershipService, several values are accepted:

   a) *String Authentication* – A typical password method.

   b) *Engine Authentication* – An engine operating automatically (i.e., without requiring user interaction).
   c) *Dialog Authentication* – A method implying user interaction.

3. Call the `apply(...)` method on the membership service providing the credentials as a parameter. This method returns an `Authenticator` (or a sub-class implementation of it).
4. If necessary, call methods on the authenticator object (for example, to set a password). See corresponding Javadoc for more information.
5. Authenticators implement a method called `isReadyToJoin()`. Its purpose is to allow asynchronous authentication implementations. Typically, one should wait until this method returns `true` before proceeding.
6. Call the `join(...)` method on the membership service by providing the authenticator as a parameter. This method returns a `Credential` object corresponding to the login approval, when successful.

TLS Transport

The TLS (i.e. secure transport) is a transport method not discussed in the previous version of the guide. By default, it is enabled in the `NetPeerGroup` (c.f.. its module implementation advertisement).

Configuration

The general configuration procedure is the following, assuming that a keystore containing proper X509 certificates and a private key for *this* peer is available:

1. Create an instance of the `NetworkManager`.
2. Retrieve the `NetworkConfigurator` from the `NetworkManager` and set the keystore location URI with the `setKeystoreLocation(...)` method.
3. If you want to check the content of the keystore, you can invoke the `getPSEConfig()` method on the peergroup's membership service. This object provides many methods to interact with the keystore.

That's it! Don't forget to read the earlier section on Jxta Security Improvements regarding PSE Membership configuration.

Implementation

Just like the other types of transports, TLS transportation has its own messengers attached to a TLS transport object. Contrary to other transports, there is an extra `TlsManager` class managing TLS connections to remote peers. The TLS endpoint addresses bear a “jxtatls” prefix.

The JXTA TLS transport is end-to-end. The middle peers, like relays and rdvs, do not have access to the content of exchanged messages between Alice and Bob. This is how it works:

- Alice starts by creating a `JTLSInput/OutputStream` pair into a `TLSSocket` object instance[14].
- This operation is performed by `TlsConn` objects, which are created by the `TlsManager` object, which is attached to the `TlsTransport` object. `TlsMessengers` are attached to `TlsConn` objects and operate like other messenger over the TLS plumbing.
- Next, a real TLS connection is created on top of the `TLSSocket` object using the `createSocket(...)` method of a/the `SSLSocketFactory` object.
- Whatever is poured in the TLS connection is transformed by the `JTLSOutputStream` into encrypted JXTA messages, which are sent over the JXTA network.
- These are read by Bob's `JTLSInputStream` and feed into Bob's side of the TLS banana connection.
- Bob can send messages to Alice by feeding its `JTLSOutputStream` too. Alice will read Bob's love letters from her side of the TLS banana connection.
- Of course, the beginning of the communication is only about letting both side of the TLS banana set-up the secure channel. Love letters are only sent if successful.

Secured pipe are implemented over `SecureInputPipeImpl` and `SecureOutputPipe` objects.

14 This is performed by the core code automatically. Engineers using JXTA/JXSE do not have to deal with this.

Caveats

- TLS will verify the validity of provided certificates when establishing a TLS connection to a remote peer. However, the JXSE implementation does not offer a certificate authority service, or means to retrieve certificates from a public key repository for example.
- The TLS connection predates the introduction of the SSLEngine in Java 5. The code could be improved with this new feature.
- JXSE uses TLS 1.0. Some people may not be comfortable with this, since one can't prevent the selection of weak ciphers by TLS (or benefit from the improvements or earlier versions). TLS 1.0 is unfortunately hard-coded in the code base.
- A possible solution to this issue is to organize Diffe-Hellman on a JXTA socket manually in order to create a secret key which can be used with a cipher. The message content would be encrypted by Alice, transported over multiple peers if necessary (with no access to content), and decrypted by Bob only.
- Advertisement publication and fetching does not happen over secure TLS communication in JXTA. Hence, content is not encrypted.

Remote Password Login

Those willing to implement remote login to their application over JXTA can investigate Standford's Secure Remote Password protocol available at: http://srp.stanford.edu/. This could be implemented on JXTA sockets.

Conclusion

Release 2.7 is finalizing the cycle of efforts initiated after release 2.5 to put JXTA/JXSE back on tracks regarding technical debt and testing. There are still areas for improvement, for sure, but overall, the code has proven to be a stable TURN-like P2P framework. We have now a stable code base to move forward with improving the API, OSGi integration and NAT traversal (amongst others).

Examples

Unfortunately, no one volunteered to update the JXTA examples for release 2.7. Engineers can still refer to collocated Junit test code for peer configuration and running. The examples available at www.practicaljxta.com can be used too.

The 2.7 release does not modify existing functionalities, meaning that code written under 2.6 should run under 2.7 without hassle. A couple deprecated public API methods have been removed in 2.7. If necessary, check 2.6 Javadoc for more information on conversion.

Book

The 'Practical JXTA II' book is now available from Scribd for online reading.

Appendix A: Sonatype Maven Release Process

This appendix contains a copy of the document describing the generation of artifacts and release in the Sonatype Maven repository. This document was created when version 2.7 beta was released.

Introduction

Following the Mavenization of the JXTA/JXSE project in release 2.6, the community has decided to select Sonatype's to host its artifacts.

Sonatype[15] offers free repository service for open source projects. It is one of the few approved forges[16] for synchronization with the central repository.

READ ME

Sonatype offers an automated release process to automatically create a tag from trunk, sign and upload artifacts in the JXTA/JXSE's community OSS repository. Unfortunately, this process is pretty fragile, constraining and leaves quite a mess when it fails. Unfortunately, the poor service offered by Java.net tends to break that process even more often.

A clean manual procedure to generate, sign and upload artifacts from trunk (or branches or elsewhere) is also included in this document. It can be used to create maintenance releases too.

The JXTA/JXSE community is strongly advised to use the manual procedure described in this document (and not Sonatype's process) as it has proven to be easier to use and less prone to errors.

Sonatype Account

We have create a 'jxse' account at Sonatype. The corresponding password has been communicated to leaders in the community.

It is accessible from:

https://oss.sonatype.org/index.html#welcome

For general questions, register to the user mailing:

ossrh-users@sonatype.org

To report issues, use the following URL:

https://issues.sonatype.org/secure/Dashboard.jspa

15 See http://nexus.sonatype.org/oss-repository-hosting.html
16 See http://maven.apache.org/guides/mini/guide-central-repository-upload.html

Release With Sonatype's Process

This chapter describe the procedure to perform a release of JXTA/JXSE. It is based on the Sonatype OSS documentation available at:

https://docs.sonatype.org/display/Repository/Sonatype+OSS+Maven+Repository+Usage+Guide

Prerequisites

1. If you have not read the READ ME section earlier in the document, please do so before proceding.
2. Make sure you have a commit access to our Subversion repository[17].
3. Make sure you have access to a PC where:
    - JDK 1.5 or higher is installed
    - Subversion[18] 1.5 or higher installed
    - Maven 2.2.1 or higher

      REM: Maven 2.1.0 and 2.2.0 produce incorrect GPG signatures and checksums.
    - Ant 1.7.1 or higher is installed

Maven Settings

4. Make sure you have configured your Maven settings.xml as decribed in section 7.1.a of the Sonatype documentation.

   You can use the JXSE Sonatype account or your own account if you have one at Sonatype OSS[19].

PGP Keys

5. In order to sign the artifacts, PGP keys must be available[20] on your PC.

   REM: Write down the pass-phrase you used to generate the keys.

Preparation

REM: Sonatype only allows releases from trunk (not branches or tags).

1. Check-out the trunk content from our repository.
2. Open pom.xml in your favorite editor.
3. Make sure the version ends with '-SNAPSHOT', for example:

   <version>2.7-Beta-SNAPSHOT</version>

   REM: This is a Sonatype process requirement to perform a release.
4. Make sure you check-in any version update.

REM: The Sonatype process to perform a release assumes/requires that all Junit tests (c.f. Surefire) pass successfully.

5. Perform '*mvn clean install*' to make sure the complete compilation, testing and install process performs successfully.

   This can be performed from your favorite IDE or from a command line session from within the repository where trunk was checked out.
6. Fix any issues encountered in the above maven process.

17 Currently → scm:svn:https://jxta-jxse.dev.java.net/svn/jxta-jxse/
18 See http://subversion.apache.org/
19 To create an account, see point 2 (Sign up) of the Sonatype OSS documentation.
20 See http://www.sonatype.com/people/2010/01/how-to-generate-pgp-signatures-with-maven/ to generate such keys.

Dealing With Failing Tests

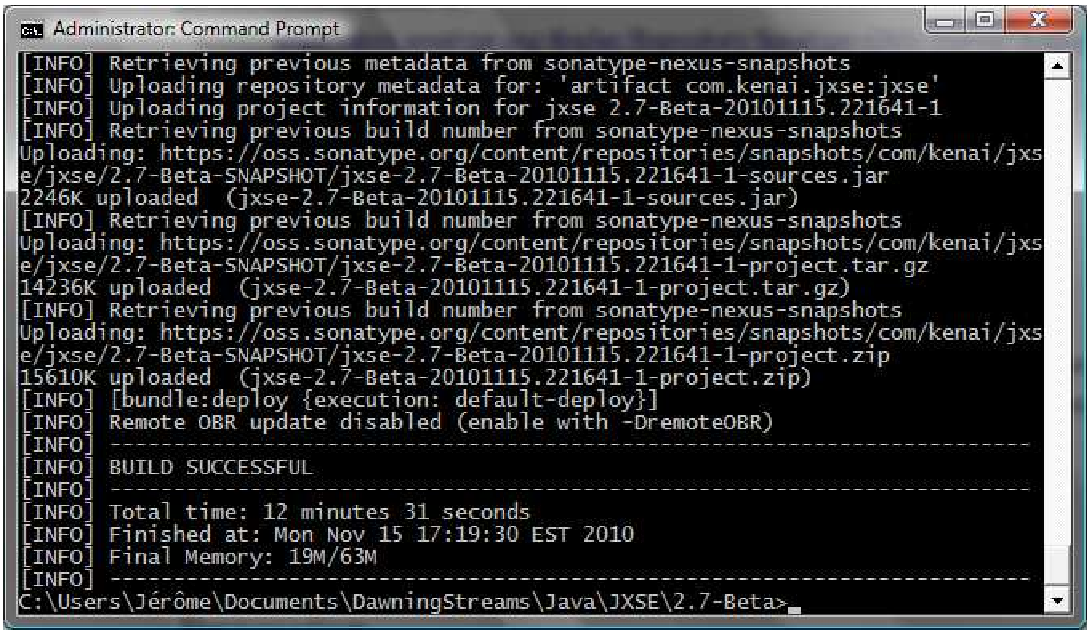
- If some tests systematiaclly fail during the above process and you are nevetheless OK to proceed with a release, annotate theses tests with:

  @Ignore("<any comment>")

- Sometimes, tests fail because of a time-out, but not always. To avoid such issues, use '`-DskipTests=true`' on the command line.

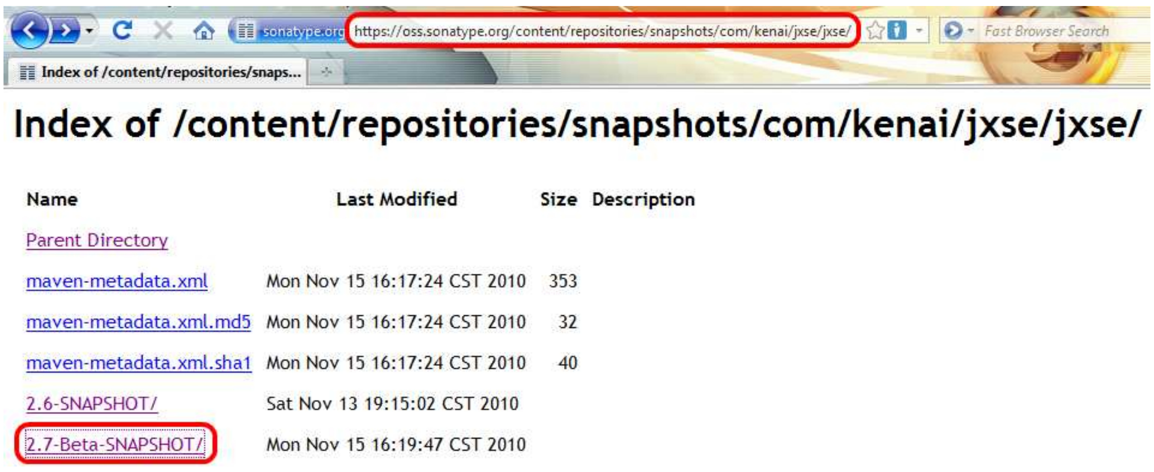
Preliminary Check With A Snapshot

In order to test a Sonatype configuration, one can use the snapshot release with the '*mvn clean deploy*' command. If successful, one should obtain something like this:



```
[INFO] Retrieving previous metadata from sonatype-nexus-snapshots
[INFO] Uploading repository metadata for: 'artifact com.kenai.jxse:jxse'
[INFO] Uploading project information for jxse 2.7-Beta-20101115.221641-1
[INFO] Retrieving previous build number from sonatype-nexus-snapshots
Uploading: https://oss.sonatype.org/content/repositories/snapshots/com/kenai/jxs
e/jxse/2.7-Beta-SNAPSHOT/jxse-2.7-Beta-20101115.221641-1-sources.jar
2246K uploaded  (jxse-2.7-Beta-20101115.221641-1-sources.jar)
[INFO] Retrieving previous build number from sonatype-nexus-snapshots
Uploading: https://oss.sonatype.org/content/repositories/snapshots/com/kenai/jxs
e/jxse/2.7-Beta-SNAPSHOT/jxse-2.7-Beta-20101115.221641-1-project.tar.gz
14236K uploaded  (jxse-2.7-Beta-20101115.221641-1-project.tar.gz)
[INFO] Retrieving previous build number from sonatype-nexus-snapshots
Uploading: https://oss.sonatype.org/content/repositories/snapshots/com/kenai/jxs
e/jxse/2.7-Beta-SNAPSHOT/jxse-2.7-Beta-20101115.221641-1-project.zip
15610K uploaded  (jxse-2.7-Beta-20101115.221641-1-project.zip)
[INFO] [bundle:deploy {execution: default-deploy}]
[INFO] Remote OBR update disabled (enable with -DremoteOBR)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12 minutes 31 seconds
[INFO] Finished at: Mon Nov 15 17:19:30 EST 2010
[INFO] Final Memory: 19M/63M
[INFO] ------------------------------------------------------------------------
C:\Users\Jérôme\Documents\DawningStreams\Java\JXSE\2.7-Beta>_
```

This can be double-checked from Sonatype's snapshot repository[21]:

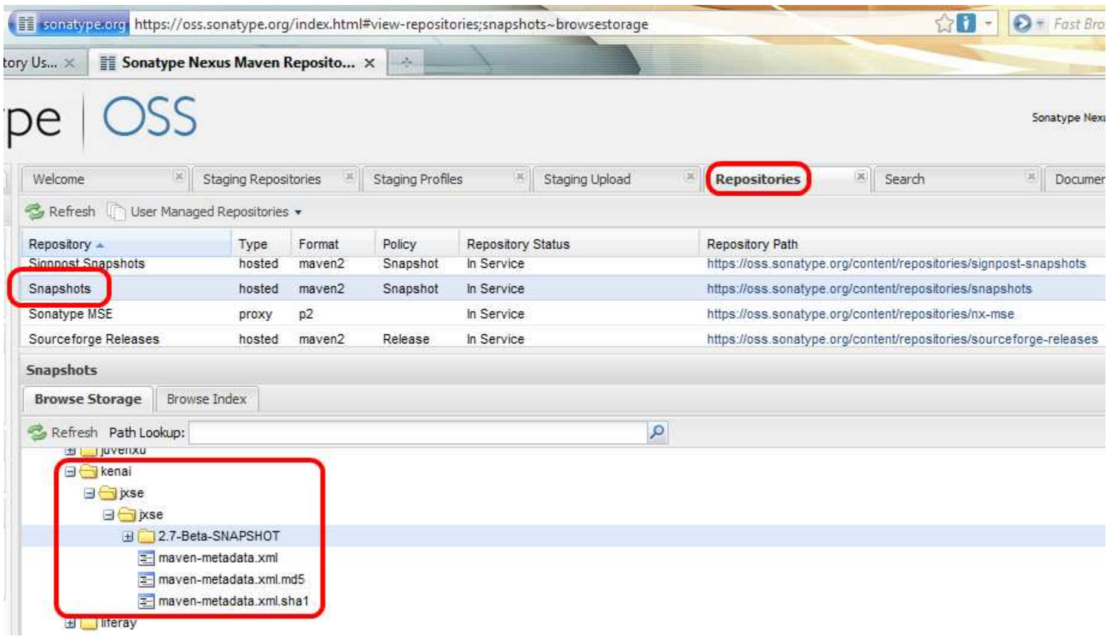

sonatype.org
https://oss.sonatype.org/content/repositories/snapshots/com/kenai/jxse/jxse/
Fast Browser Search
Index of /content/repositories/snaps...

Index of /content/repositories/snapshots/com/kenai/jxse/jxse/

Name	Last Modified	Size	Description
Parent Directory			
maven-metadata.xml	Mon Nov 15 16:17:24 CST 2010	353	
maven-metadata.xml.md5	Mon Nov 15 16:17:24 CST 2010	32	
maven-metadata.xml.sha1	Mon Nov 15 16:17:24 CST 2010	40	
2.6-SNAPSHOT/	Sat Nov 13 19:15:02 CST 2010		
2.7-Beta-SNAPSHOT/	Mon Nov 15 16:19:47 CST 2010		

Or from Sonatype's interface[22]:

21 See point 4 of OSS Sonatype documentation for the list of OSS Sonatype repositories.
22 Login from: https://oss.sonatype.org/index.html#welcome



Repository	Type	Format	Policy	Repository Status	Repository Path
Signpost Snapshots	hosted	maven2	Snapshot	In Service	https://oss.sonatype.org/content/repositories/signpost-snapshots
Snapshots	hosted	maven2	Snapshot	In Service	https://oss.sonatype.org/content/repositories/snapshots
Sonatype MSE	proxy	p2		In Service	https://oss.sonatype.org/content/repositories/nx-mse
Sourceforge Releases	hosted	maven2	Release	In Service	https://oss.sonatype.org/content/repositories/sourceforge-releases

```
juvenxu
kenai
  jxse
    jxse
      2.7-Beta-SNAPSHOT
      maven-metadata.xml
      maven-metadata.xml.md5
      maven-metadata.xml.sha1
liferay
```

REM: This is also where snapshots can be deleted too.

Releasing From Trunk

WARNING: Staging a release automatically creates a tag from trunk. If the process fails – for example, because of an unstable Junit test –, the Sonatype process does not clean the mess. One can end up with a modified pom.xml (amongst others).

Hence, it is highly recommended to run the following processes using the '`-DskipTests=true`' option.

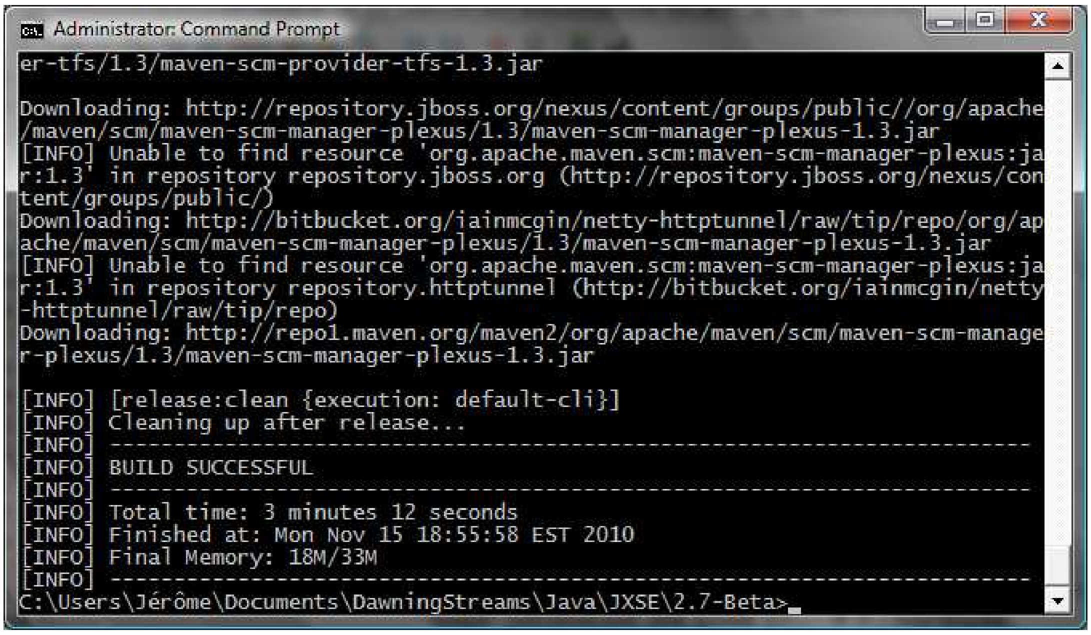
REM: If the process should fail, (locally) revert all subversion modifications made by the Sonatype process and start with a clean run first.

Clean

The first step is to perform a clean-up. Run the following command:

```
mvn release:clean
```

If successful, one should obtain something like the following:



```
er-tfs/1.3/maven-scm-provider-tfs-1.3.jar

Downloading: http://repository.jboss.org/nexus/content/groups/public//org/apache
/maven/scm/maven-scm-manager-plexus/1.3/maven-scm-manager-plexus-1.3.jar
[INFO] Unable to find resource 'org.apache.maven.scm:maven-scm-manager-plexus:ja
r:1.3' in repository repository.jboss.org (http://repository.jboss.org/nexus/con
tent/groups/public/)
Downloading: http://bitbucket.org/iainmcgin/netty-httptunnel/raw/tip/repo/org/ap
ache/maven/scm/maven-scm-manager-plexus/1.3/maven-scm-manager-plexus-1.3.jar
[INFO] Unable to find resource 'org.apache.maven.scm:maven-scm-manager-plexus:ja
r:1.3' in repository repository.httptunnel (http://bitbucket.org/iainmcgin/netty
-httptunnel/raw/tip/repo)
Downloading: http://repo1.maven.org/maven2/org/apache/maven/scm/maven-scm-manage
r-plexus/1.3/maven-scm-manager-plexus-1.3.jar

[INFO] [release:clean {execution: default-cli}]
[INFO] Cleaning up after release...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 minutes 12 seconds
[INFO] Finished at: Mon Nov 15 18:55:58 EST 2010
[INFO] Final Memory: 18M/33M
[INFO] ------------------------------------------------------------------------
C:\Users\Jérôme\Documents\DawningStreams\Java\JXSE\2.7-Beta>_
```

Prepare

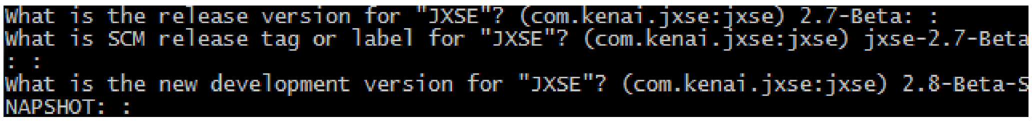
The second step is to perform a prepare. Run the following command where the username is the one defined for svn access:

```
mvn -Dusername=adamman71 -Dpassword=<yourpwd> -DskipTests=true
    release:prepare
```

Eventually, a dry run can be performed with the '`-DdryRun=true`' option.

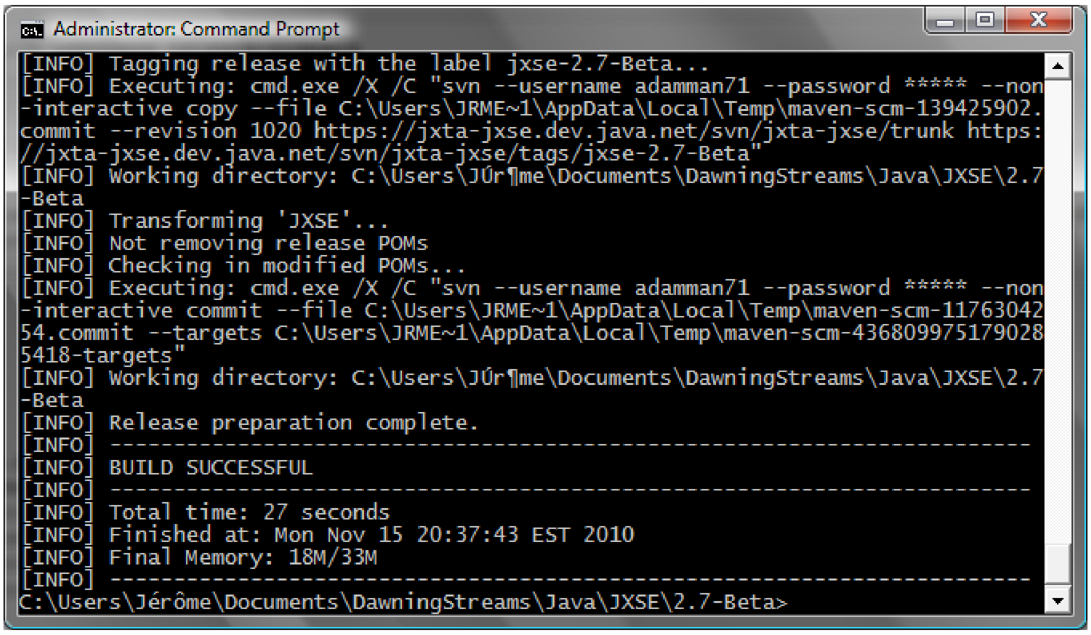
REM: To perform this stage, all local modifications in the check-out must have been deleted (for example, Junit test results in files) or previously checked-in.

Sonatype's process will ask a couple of questions:



```
What is the release version for "JXSE"? (com.kenai.jxse:jxse) 2.7-Beta: :
What is SCM release tag or label for "JXSE"? (com.kenai.jxse:jxse) jxse-2.7-Beta
: :
What is the new development version for "JXSE"? (com.kenai.jxse:jxse) 2.8-Beta-S
NAPSHOT: :
```

If the process if successful, one should get something like this:



```
[INFO] Tagging release with the label jxse-2.7-Beta...
[INFO] Executing: cmd.exe /X /C "svn --username adamman71 --password ***** --non
-interactive copy --file C:\Users\JRME~1\AppData\Local\Temp\maven-scm-139425902.
commit --revision 1020 https://jxta-jxse.dev.java.net/svn/jxta-jxse/trunk https:
//jxta-jxse.dev.java.net/svn/jxta-jxse/tags/jxse-2.7-Beta"
[INFO] Working directory: C:\Users\JÚr¶me\Documents\DawningStreams\Java\JXSE\2.7
-Beta
[INFO] Transforming 'JXSE'...
[INFO] Not removing release POMs
[INFO] Checking in modified POMs...
[INFO] Executing: cmd.exe /X /C "svn --username adamman71 --password ***** --non
-interactive commit --file C:\Users\JRME~1\AppData\Local\Temp\maven-scm-11763042
54.commit --targets C:\Users\JRME~1\AppData\Local\Temp\maven-scm-436809975179028
5418-targets"
[INFO] Working directory: C:\Users\JÚr¶me\Documents\DawningStreams\Java\JXSE\2.7
-Beta
[INFO] Release preparation complete.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 27 seconds
[INFO] Finished at: Mon Nov 15 20:37:43 EST 2010
[INFO] Final Memory: 18M/33M
[INFO] ------------------------------------------------------------------------
C:\Users\Jérôme\Documents\DawningStreams\Java\JXSE\2.7-Beta>
```

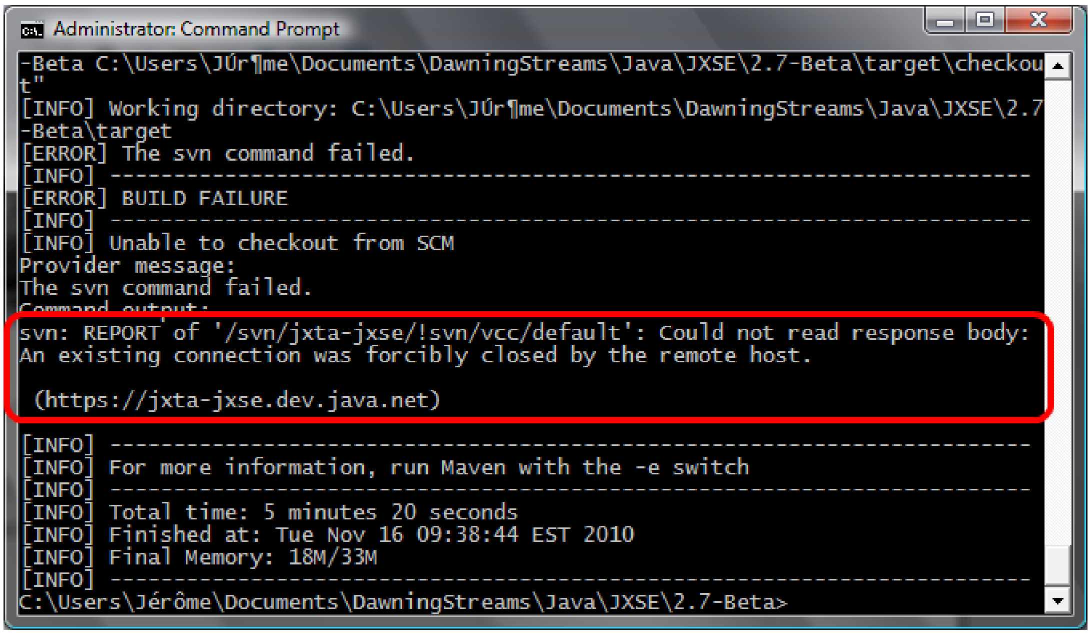
Perform

The third step to perform the release. Run the following command:

```
mvn -Dusername=adamman71 -Dpassword=<yourpwd> -DskipTests=true
    -Darguments=-Dgpg.passphrase=<passphrase> release:perform
```

Dealing With Java.net Timeouts

Often, Java.net will time-out the connection:



```
-Beta C:\Users\JÚr¶me\Documents\DawningStreams\Java\JXSE\2.7-Beta\target\checkou
t"
[INFO] Working directory: C:\Users\JÚr¶me\Documents\DawningStreams\Java\JXSE\2.7
-Beta\target
[ERROR] The svn command failed.
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Unable to checkout from SCM
Provider message:
The svn command failed.
Command output:
svn: REPORT of '/svn/jxta-jxse/!svn/vcc/default': Could not read response body:
An existing connection was forcibly closed by the remote host.

 (https://jxta-jxse.dev.java.net)

[INFO] ------------------------------------------------------------------------
[INFO] For more information, run Maven with the -e switch
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5 minutes 20 seconds
[INFO] Finished at: Tue Nov 16 09:38:44 EST 2010
[INFO] Final Memory: 18M/33M
[INFO] ------------------------------------------------------------------------
C:\Users\Jérôme\Documents\DawningStreams\Java\JXSE\2.7-Beta>
```

One can retry the operation to solve this issue, but it does not always work. If unsuccessful, consider posting artifacts with the manual procedure.

Staging Artifacts Manually

This is the preferred method to post releases on our Sonatype repositories.

Prerequisites

Prerequisites are identical to those for a release with Sonatype's process (describe earlier in the document). Make sure you meet these requirements before proceeding.

Preparation

1. If you want to create a maintenance release, consider creating a branch with the corresponding code and apply maintenance patches.
2. Make sure the proper version has been set in the pom.xml. It should not contain -SNAPSHOT:

   <version>2.7-Beta</version>
3. Generate a tag (preferably).
4. Eventually, export (not check-out) the directory content in a separate directory locally (use the tag content if one has been generated).

   REM: In our example, we will take the 2.7-beta tag.
5. Open a command session (DOS, shell...) and go to the directory containing the export.

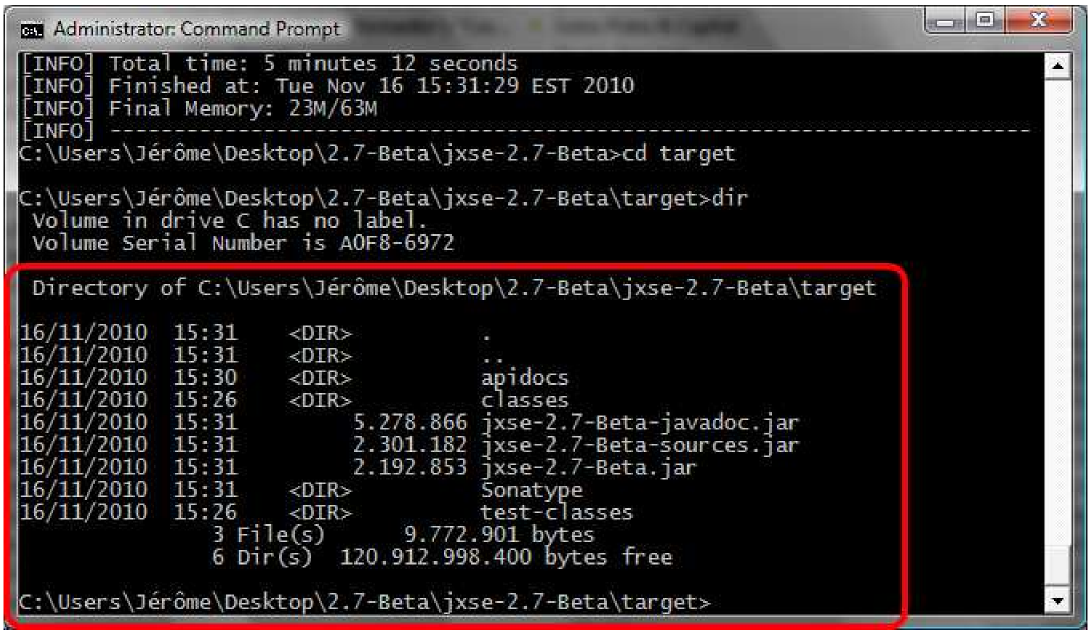
Building Process

6. Make sure you have read the section about dealing with failing tests in the previous chapter.
7. If necessary, temporarily disactivate tests in the pom.xml (c.f. maven surefire plugin configuration):

   <skip>true</skip>
8. Perform '*mvn clean install -P sonatype-packaging*'

   REM: 'sonatype-packaging' is a new maven profile which adds the generation of Javadoc to the compilation process.

   If successful, the content of the target directory should look like this:



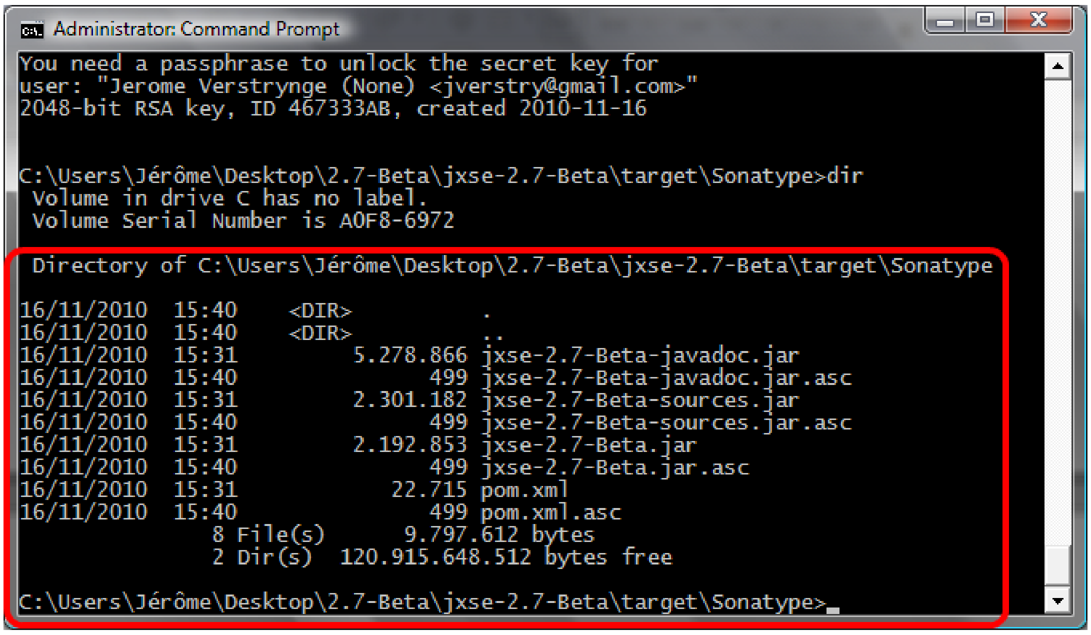
```
[INFO] Total time: 5 minutes 12 seconds
[INFO] Finished at: Tue Nov 16 15:31:29 EST 2010
[INFO] Final Memory: 23M/63M
[INFO] ------------------------------------------------------------------------
C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta>cd target

C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta\target>dir
 Volume in drive C has no label.
 Volume Serial Number is A0F8-6972

 Directory of C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta\target

16/11/2010  15:31    <DIR>          .
16/11/2010  15:31    <DIR>          ..
16/11/2010  15:30    <DIR>          apidocs
16/11/2010  15:26    <DIR>          classes
16/11/2010  15:31         5.278.866 jxse-2.7-Beta-javadoc.jar
16/11/2010  15:31         2.301.182 jxse-2.7-Beta-sources.jar
16/11/2010  15:31         2.192.853 jxse-2.7-Beta.jar
16/11/2010  15:31    <DIR>          Sonatype
16/11/2010  15:26    <DIR>          test-classes
               3 File(s)      9.772.901 bytes
               6 Dir(s)  120.912.998.400 bytes free

C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta\target>
```

Upload

9. Go into the Sonatype directory.
10. Sign all file with the following command (if using DOS):

```
FOR %1 IN (*) DO gpg -ab --passphrase <yourpassphrase> %1
```

If all goes fine, the content of the directory should look like this ('.asc' files are created):

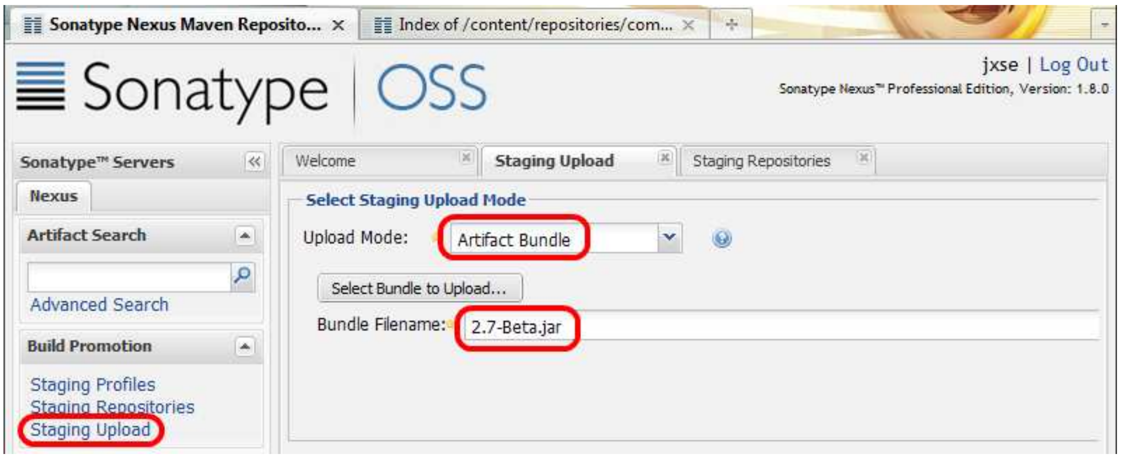


```
You need a passphrase to unlock the secret key for
user: "Jerome Verstrynge (None) <jverstry@gmail.com>"
2048-bit RSA key, ID 467333AB, created 2010-11-16

C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta\target\Sonatype>dir
 Volume in drive C has no label.
 Volume Serial Number is A0F8-6972

 Directory of C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta\target\Sonatype

16/11/2010  15:40    <DIR>          .
16/11/2010  15:40    <DIR>          ..
16/11/2010  15:31         5.278.866 jxse-2.7-Beta-javadoc.jar
16/11/2010  15:40               499 jxse-2.7-Beta-javadoc.jar.asc
16/11/2010  15:31         2.301.182 jxse-2.7-Beta-sources.jar
16/11/2010  15:40               499 jxse-2.7-Beta-sources.jar.asc
16/11/2010  15:31         2.192.853 jxse-2.7-Beta.jar
16/11/2010  15:40               499 jxse-2.7-Beta.jar.asc
16/11/2010  15:31            22.715 pom.xml
16/11/2010  15:40               499 pom.xml.asc
               8 File(s)      9.797.612 bytes
               2 Dir(s)  120.915.648.512 bytes free

C:\Users\Jérôme\Desktop\2.7-Beta\jxse-2.7-Beta\target\Sonatype>
```

11. Next, create a package for Sonatype using the following command (you can use a different name than 2.7-Beta.jar, it does not matter):

```
jar -cvf 2.7-Beta.jar *
```

12. Next, log into Sonatype[23] and load the .jar created above:

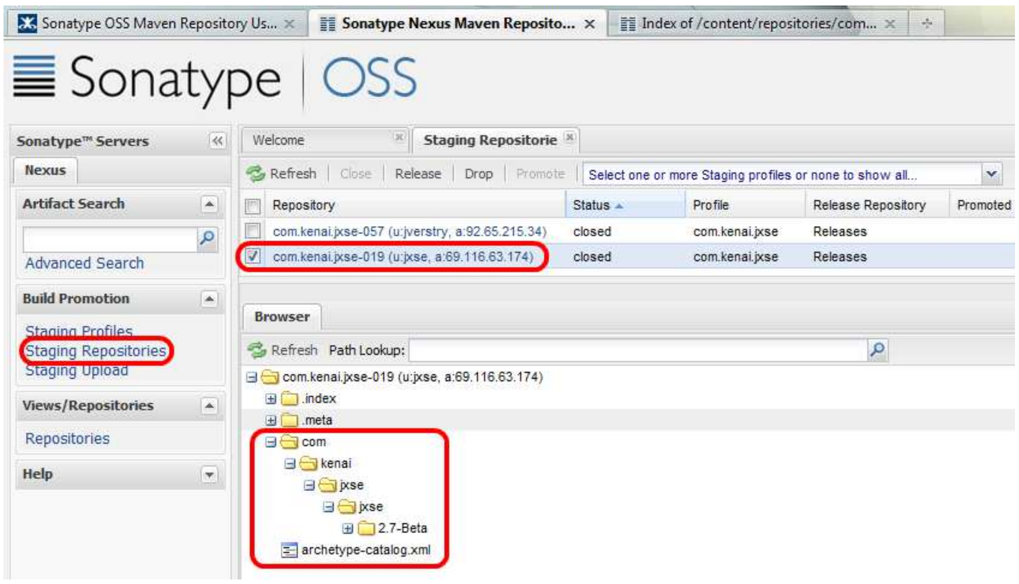


Upload Mode: Artifact Bundle

Bundle Filename: 2.7-Beta.jar

13. If all goes fine, it should be visible from here:

23 See https://oss.sonatype.org/index.html#welcome



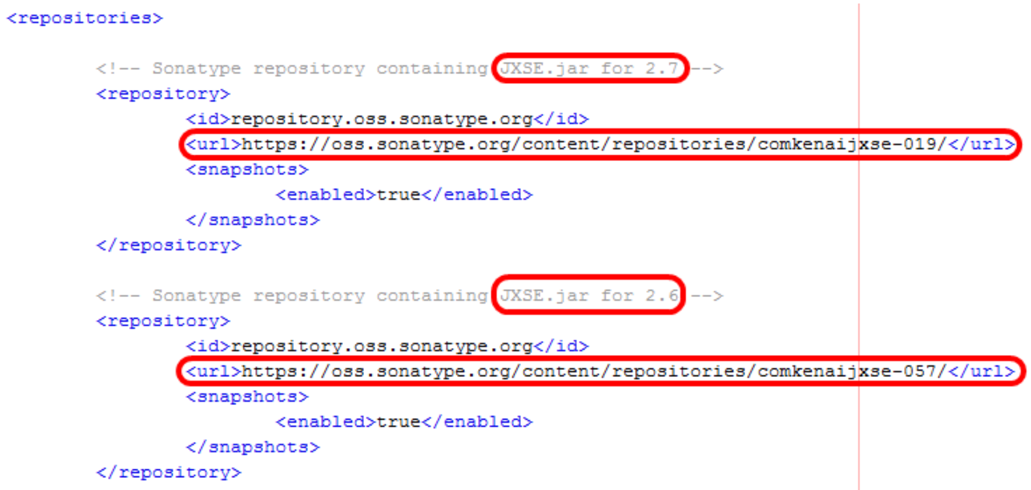
Repository	Status	Profile	Release Repository	Promoted
com.kenai.jxse-057 (u:jverstry, a:92.65.215.34)	closed	com.kenai.jxse	Releases	
✓ com.kenai.jxse-019 (u:jxse, a:69.116.63.174)	closed	com.kenai.jxse	Releases	

```
com.kenai.jxse-019 (u:jxse, a:69.116.63.174)
  .index
  .meta
  com
    kenai
      jxse
        jxse
          2.7-Beta
    archetype-catalog.xml
```

14. That's it, the local export of the project can be deleted if it was created.
15. The branch can be closed without synchronizing modifications (merging) with trunk.

User Project Pom.xml

The user's project must contain a repository entry such as:



```
<repositories>

        <!-- Sonatype repository containing JXSE.jar for 2.7-->
        <repository>
                <id>repository.oss.sonatype.org</id>
                <url>https://oss.sonatype.org/content/repositories/comkenaijxse-019/</url>
                <snapshots>
                        <enabled>true</enabled>
                </snapshots>
        </repository>

        <!-- Sonatype repository containing JXSE.jar for 2.6 -->
        <repository>
                <id>repository.oss.sonatype.org</id>
                <url>https://oss.sonatype.org/content/repositories/comkenaijxse-057/</url>
                <snapshots>
                        <enabled>true</enabled>
                </snapshots>
        </repository>
```

REM: It is unclear why Sonatype has created two repositories, instead of keeping one. May be it is because I have used my own profile for 2.6 and the JXSE account for 2.7.

References

- Sonatype OSS Maven Repository Usage Guide: https://docs.sonatype.org/display/Repository/Sonatype+OSS+Maven+Repository+Usage+Guide
- How to generate PGP signatures with Maven: http://www.sonatype.com/people/2010/01/how-to-generate-pgp-signatures-with-maven/
- Sonatype OSS Repository Hosting: http://nexus.sonatype.org/oss-repository-hosting.html
- Sonatype OSS Repository login: https://oss.sonatype.org/index.html#welcome
- Sonatype Dashboard (for issues): https://issues.sonatype.org/secure/Dashboard.jspa
- Guide to central repository upload: http://maven.apache.org/guides/mini/guide-central-repository-upload.html