## ABOUT RUDIGER HERRMANN

# JGit Authentication Explained

Posted by: Rudiger Herrmann in Enterprise Java December 14th, 2014 0 304 Views

Authentication in JGit is mostly on par with native Git. Commonly used protocols like SSH and HTTP(S) and their authentication methods are supported. This article summarizes how to use the JGit authentication API to securely access remote Git repositories.

Though the examples in this article use the CloneCommand, the described techniques can be applied to all classes that connect to remote repositories like FetchCommand, PushCommand, LsRemoteCommand, etc. All of these commands have a common base class – TransportCommand – which offers the methods discussed here.

## HTTP(S) – https://example.com/repo.git

Authenticating via HTTP and HTTPS is straightforward. An implementation of CredentialsProvider is used to return the authentication credentials when the command requests them. The CredentialsProvider to be used for a certain command can be specified through setCredentialsProvider().

For example, the code below clones a repository over HTTPS and authenticates with username and password.

```
1 CloneCommand cloneCommand = Git.cloneRepository();
2 cloneCommand.setURI( "https://example.com/repo.git" );
3 cloneCommand.setCredentialsProvider( new UsernamePasswordCredentialsProvider( "user", "password" ) );
```

The UsernamePasswordCredentialsProvider is an implementation of CredentialsProvider that comes with JGit and uses the given username and password to authenticate.

Alternatively, JGit (version 3.5 and later) can also read credentials from the user's .netrc file. The NetRCCredentialsProvider uses the first machine entry from the file for authentication.

Though it is not recommendable to send credentials over unsecured connections, the described approach also works for plain HTTP like http://example.com/repo.git.

## SSH with Public Key – ssh://user@example.com/repo.git

JGit delegates creating and destroying SSH connections to the abstract SshSessionFactory. To use public key authentication for an SSH connection, such a session factory has to be specified for the executed command.

With setTransportConfigCallback(), a TransportConfigCallback interface can be specified to intercept the connection process. Its sole method – named configure() – is called just before a connection is established. It is passed a parameter of type Transport which will be used to copy objects between the local and the remote repository. For each protocol there is a distinct subclass of Transport that handles the respective details of that protocol.

Like shown below the callback can be used to configure the Transport instance right before it is put into use:

```
01 SshSessionFactory sshSessionFactory = new JschConfigSessionFactory() {
02   @Override
03   protected void configure( Host host, Session session ) {
04     // do nothing
05   }
06 };
```

```
07  CloneCommand cloneCommand = Git.cloneRepository();
08  cloneCommand.setURI( "ssh://user@example.com/repo.git" );
09  cloneCommand.setTransportConfigCallback( new TransportConfigCallback() {
10    @Override
11    public void configure( Transport transport ) {
12      SshTransport sshTransport = ( SshTransport )transport;
13      sshTransport.setSshSessionFactory( sshSessionFactory );
14    }
15  } );
```

JGit provides an abstract JSchConfigSessionFactory that uses JSch to establish SSH connections and requires its configure() to be overridden. Because in the simplest case there isn't anything to be configured, the example above just overrides the method to let the code compile.

JSchConfigSessionFactory is mostly compatible with OpenSSH, the SSH implementation used by native Git. It loads the known hosts and private keys from their default locations (identity, id_rsa and id_dsa) in the user's .ssh directory.

If your private key file is named differently or located elsewhere, I recommend to override createDefaultJSch(). After calling the base method, custom private keys can be added like so:

```
1  @Override
2  protected JSch createDefaultJSch( FS fs ) throws JSchException {
3    JSch defaultJSch = super.createDefaultJSch( fs );
4    defaultJSch.addIdentity( "/path/to/private_key" )
5    return defaultJSch;
6  }
```

In this example a private key from a custom file location is added. If you look into the JSch JavaDoc you will find further overloaded addIdentity() methods.

For the sake of completeness I should mention that there is also a global session factory. It can be obtained and changed through SshSessionFactory.get/setInstance() and is used as a default if no specific shSessionFactory was configured for a command. However, I recommend to refrain from using it. Apart from making it harder to write isolated tests, there might be code outside of your control that changes the global session factory.

# SSH with Password – ssh://user@example.com/repo.git

As with using SSH with public keys, an SshSessionFactory must be specified to use password-secured SSH connections. But this time, the session factory's configure() method has a purpose.

```
01  SshSessionFactory sshSessionFactory = new JschConfigSessionFactory() {
02    @Override
03    protected void configure( Host host, Session session ) {
04      session.setPassword( "password" );
05    }
06  } );
07
08  CloneCommand cloneCommand = Git.cloneRepository();
09  cloneCommand.setURI( "ssh://user@example.com/repo.git" );
10  cloneCommand.setTransportConfigCallback( new TransportConfigCallback() {
11    @Override
12    public void configure( Transport transport ) {
13      SshTransport sshTransport = ( SshTransport )transport;
14      sshTransport.setSshSessionFactory( sshSessionFactory );
15    }
16  } );
```

A JSch session represents a connection to an SSH server and in line 4, the password for the current session is set. The rest of the code is the same that was used to connect via SSH with public key authentication.

# Which Authentication Method to Use?

Some authentication methods discussed here can also be combined. For example, setting a credentials provider while attempting to connect to a remote repository via SSH with public-key won't harm. However, you usually want to know what Transport will be used for a given repository-URL beforehand.

To determine that, the TransportProtocol's canHandle() method can be used. It returns true if the protocol can handle the given URL and false otherwise. A list of all registered TransportProtocols can be obtained from Transport.getTransportProtocols(). And once the protocol is known, the appropriate authentication method can be chosen.

# Authentication @ GitHub

GitHub supports a variety of protocols and authentications methods, but certainly not all possible combinations. A common mistake, for example, is to try to use SSH with password authentication. But this combination is not supported – only SSH with public keys is.

This comparison of protocols offered by GitHub lists what is supported and what not. Summarized, there is:

- Plain Git (e.g. git://github.com/user/repo.git): The transfer is unencrypted and the server is not verified.
- HTTPS (e.g. https://github.com/user/repo.git): Works practically everywhere. Uses password authentication for pushing but allows anonymous fetch and clone.
- SSH (e.g. ssh://git@github.com:user/repo.git): Uses public key authentication, also for fetch and clone.

# Concluding JGit Authentication

While I find the authentication facilities are a bit widely scattered over the JGit API, they get the task done. The recipes given here hopefully provide you with the necessary basics to authenticate connections in JGit and hiding the complexities of the API could be seen as an exercise to practice clean code !

If you have difficulties or questions, feel free to leave a comment or ask the friendly and helpful JGit community for assistance.

Reference: JGit Authentication Explained from our JCG partner Rudiger Herrmann at the Code Affine blog.

Tagged with: GIT JGIT

(*0 rating, 0 votes*)

*You need to be a registered member to rate this.* Start the discussion 304 Views Tweet it!



## LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS



**Window closing event handling**

**Tomcat server.xml Configuration Example**

**Java Servlet Exception Handling Example**

**Java Servlet Tutorials**

**Microservices for Java Developers: Distributed Tracing**

**JAVA Swing Application Example**

**Spring Security Concurrent Session Control Example Tutorial – How to limit number of User...**

**5 Quest Asking Microse (Questi**

## Leave a Reply

Start the discussion...

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Subscribe

## KNOWLEDGE BASE

- Courses
- Examples
- Minibooks
- Resources
- Tutorials

## PARTNERS

- Mkyong

## THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- System Code Geeks
- Web Code Geeks

## HALL OF FAME

- "Android Full Application Tutorial" series
- 11 Online Learning websites that you should check out
- Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
- Android Google Maps Tutorial
- Android JSON Parsing with Gson Tutorial
- Android Location Based Services Application – GPS location
- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet

## ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community f o ultimate Java to Java developers resource center; targeted at t technical team lead (senior developer), project manager and j JCGs serve the Java, SOA, Agile and Telecom communities with daily n domain experts, articles, tutorials, reviews, announcements, code snipp source projects.

## DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Geel property of their respective owners. Java is a trademark or registered t Oracle Corporation in the United States and other countries. Examples is not connected to Oracle Corporation and is not sponsored by Oracle