Table of Contents

1. Introduction
2. Legal Notice
3. Working with the Code
4. IDE Integration
5. Building
6. Tests
7. Code Formatting
8. Validating releases
9. Notes for Maintainers

Apache ActiveMQ Artemis Hacking Guide

This hacking guide outlines how developers can get involved in contributing to the Apache ActiveMQ Artemis project.

Legal Notice

Working with the Code

While the canonical Apache ActiveMQ Artemis git repository is hosted on Apache hardware at https://git-wip-us.apache.org/repos/asf?p=activemq-artemis.git contributors are encouraged (but not required) to use a mirror on GitHub for collaboration and pull-request review functionality. Follow the steps below to get set up with GitHub, etc.

If you do not wish to use GitHub for whatever reason you can follow the overall process outlined in the "Typical development cycle" section below but instead attach a patch file to the related JIRA or an email to the dev list.

Initial Steps

1. Create a GitHub account if you don't have one already

   http://github.com

2. Fork the apache-artemis repository into your account

   https://github.com/apache/activemq-artemis

3. Clone your newly forked copy onto your local workspace:

```
$ git clone git@github.com:<your-user-name>/activemq-artemis.git
Cloning into 'activemq-artemis'...
remote: Counting objects: 63800, done.
remote: Compressing objects: 100% (722/722), done.
remote: Total 63800 (delta 149), reused 0 (delta 0), pack-reused 62748
Receiving objects: 100% (63800/63800), 18.28 MiB | 3.16 MiB/s, done.
Resolving deltas: 100% (28800/28800), done.
Checking connectivity... done.

$ cd activemq-artemis
```

4. Add a remote reference to `upstream` for pulling future updates

```
$ git remote add upstream https://github.com/apache/activemq-artemis
```

5. Build with Maven

   Typically developers will want to build using the `dev` profile which disables license and code style checks. For example:

```
$ mvn -Pdev install
...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] ActiveMQ Artemis Parent ........................... SUCCESS [2.298s]
[INFO] ActiveMQ Artemis Commons .......................... SUCCESS [1.821s]
[INFO] ActiveMQ Artemis Selector Implementation .......... SUCCESS [0.767s]
[INFO] ActiveMQ Artemis Native POM ....................... SUCCESS [0.189s]
[INFO] ActiveMQ Artemis Journal .......................... SUCCESS [0.646s]
[INFO] ActiveMQ Artemis Core Client ...................... SUCCESS [5.969s]
[INFO] ActiveMQ Artemis JMS Client ....................... SUCCESS [2.110s]
[INFO] ActiveMQ Artemis Server ........................... SUCCESS [11.540s]
...
[INFO] ActiveMQ Artemis stress Tests ..................... SUCCESS [0.332s]
```

```
[INFO] ActiveMQ Artemis performance Tests ..... SUCCESS [0.174s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Typical development cycle

1. Identify a task (e.g. a bug to fix or feature to implement)

   https://issues.apache.org/jira/browse/ARTEMIS

2. Create a topic branch in your local git repo to do your work

   ```
   $ git checkout -b my_cool_feature
   ```

3. Make the changes and commit one or more times

   ```
   $ git commit
   ```

   When you commit your changes you will need to supply a commit message. We follow the 50/72 git commit message format. An ActiveMQ Artemis commit message should be formatted in the following manner:

   i. Add the ARTEMIS JIRA (if one exists) followed by a brief description of the change in the first line. This line should be limited to 50 characters.
   ii. Insert a single blank line after the first line.
   iii. Provide a detailed description of the change in the following lines, breaking paragraphs where needed. These lines should be wrapped at 72 characters.

   An example correctly formatted commit message:

   ```
   ARTEMIS-123 Add new commit msg format to README

   Adds a description of the new commit message format as well as examples
   of well formatted commit messages to the README.md.  This is required
   to enable developers to quickly identify what the commit is intended to
   do and why the commit was added.
   ```

4. Occasionally you'll want to push your commit(s) to GitHub for safe-keeping and/or sharing with others.

   ```
   git push origin my_cool_feature
   ```

   Note that git push references the branch you are pushing and defaults to `master`, not your working branch.

5. Discuss your planned changes (if you want feedback)

   On mailing list - http://activemq.apache.org/mailing-lists.html On IRC - irc://irc.freenode.org/apache-activemq or https://webchat.freenode.net/?channels=apache-activemq

6. Once you're finished coding your feature/fix then rebase your branch against the latest master (applies your patches on top of master)

   ```
   git fetch upstream
   ```

```
git rebase -i upstream/master
# if you have conflicts fix them and rerun rebase
# The -f, forces the push, alters history, see note below
git push -f origin my_cool_feature
```

The `rebase -i` triggers an interactive update which also allows you to combine commits, alter commit messages etc. It's a good idea to make the commit log very nice for external consumption (e.g. by squashing all related commits into a single commit. Note that rebasing and/or using `push -f` can alter history. While this is great for making a clean patch, it is unfriendly to anyone who has forked your branch. Therefore you'll want to make sure that you either work in a branch that you don't share, or if you do share it, tell them you are about to revise the branch history (and thus, they will then need to rebase on top of your branch once you push it out).

7. Get your changes merged into upstream

    i. Send a GitHub pull request, by clicking the pull request link while in your repo's fork.
    ii. An email will automatically be sent to the ActiveMQ developer list.
    iii. As part of the review you may see an automated test run comment on your request.
    iv. After review a maintainer will merge your PR into the canonical git repository at which point those changes will be synced with the GitHub mirror repository (i.e. your `master`) and your PR will be closed by the `asfgit` bot.

Other common tasks

1. Pulling updates from upstream

```
$ git pull --rebase upstream master
```

(`--rebase` will automatically move your local commits, if any, on top of the latest branch you pull from; you can leave it off if you do not have any local commits).

One last option, which some prefer, is to avoid using pull altogether, and just use fetch + rebase (this is of course more typing). For example:

```
$ git fetch upstream
$ git pull
```

2. Pushing pulled updates (or local commits if you aren't using topic branches) to your private GitHub repo (origin)

```
$ git push
Counting objects: 192, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (44/44), done.
Writing objects: 100% (100/100), 10.67 KiB, done.
Total 100 (delta 47), reused 100 (delta 47)
To git@github.com:<your-user-name>/apache-artemis.git
   3382570..1fa25df  master -> master
```

You might need to say -f to force the changes.

Adding New Dependencies

Due to incompatibilities between some open source licenses and the Apache v2.0 license (that this project is licensed under) care must be taken when adding new dependencies to the project. The Apache Software Foundation 3rd party

licensing policy has more information here: http://www.apache.org/legal/3party.html

To keep track of all licenses in ActiveMQ Artemis, new dependencies must be added in either the top level pom.xml or in test/pom.xml (depending on whether this is a test only dependency or if it is used in the main code base). The dependency should be added under the dependency management section with version and labelled with a comment highlighting the license for the dependency version. See existing dependencies in the main pom.xml for examples. The dependency can then be added to individual ActiveMQ Artemis modules *without* the version specified (the version is implied from the dependency management section of the top level pom). This allows ActiveMQ Artemis developers to keep track of all dependencies and licenses.

IDE Integration

IntelliJ IDEA

Importing the Project

The following steps show how to import ActiveMQ Artemis source into IntelliJ IDEA and setup the correct maven profile to allow running of JUnit tests from within the IDE. (Steps are based on version: 13.1.4)

- File --> Import Project --> Select the root directory of the ActiveMQ Artemis source folder. --> Click OK

This should open the import project wizard. From here:

- Select "Import from existing model" toggle box, then select Maven from the list box below. Click Next.
- Leave the defaults set on this page and click next.
- On the "Select profiles page", select the checkbox next to "dev" and click next.
- From here the default settings should suffice. Continue through the wizard, clicking next until the wizard is complete.

Once the project has been imported and IDEA has caught up importing all the relevant dependencies, you should be able to run JUnit tests from with the IDE. Select any test class in the tests -> integration tests folder. Right click on the class in the project tab and click "Run ". If the "Run " option is present then you're all set to go.

Note about IBM JDK on Idea

If you are running IBM JDK it may be a little tricky to get it working.

After you add the JDK to the IDE, add also the vm.jar specific to your platform under that jdk.

```
(e.g: JAVA_HOME/jre/lib/amd64/default/jclSC180/vm.jar
```

There's a SOF Question about this that could be useful in case you are running into this issue.

Style Templates for Idea

We have shared the style templates that are good for this project. If you want to apply them use these steps:

- File->Import Settings
- Select the file under ./artemis-cloned-folder/etc/IDEA-style.jar
- Select both Code Style Templates and File templates (it's the default option)
- Select OK and restart Idea

Issue: My JUnit tests are not runnable with in the IDE.

If the "Run " or "Run all tests" option is not present. It is likely that the default profile has not been imported properly. To (re)import the "tests" Maven profile in an existing project.

- Open the Maven Projects Tool Window: View -> Tool Windows -> Maven Projects
- Select the "profiles" drop down
- Unselect then reselect the checkbox next to "tests".
- Click on the "Reimport all maven projects" button in the top left hand corner of the window. (It looks like a ciruclar blue

arrow.
- Wait for IDEA to reload and try running a JUnit test again. The option to run should now be present.

Eclipse

We recommend using Eclipse Kepler (4.3), due to the built-in support for Maven and Git. Note that there are still some Maven plugins used by sub-projects (e.g. documentation) which are not supported even in Eclipse Kepler (4.3).

Eclipse m2e is already included in "Eclipse IDE for Java Developers", or it can be installed from Eclipse Kepler release repository.

Git setup

It is strongly recommended to turn off the auto-updating of .gitignore files by the Git Team extension. Otherwise, it generates new .gitignore files in many directories that are not needed due to the top level .gitignore file. To turn it off, go to Preferences->Team->Git->Projects and deselect the "Automatically ignore derived resources" checkbox.

Annotation Pre-Processing

ActiveMQ Artemis uses JBoss Logging and that requires source code generation from Java annotations. In order for it to 'just work' in Eclipse you need to install the *Maven Integration for Eclipse JDT Annotation Processor Toolkit* m2e-apt. See this JBoss blog post for details.

M2E Connector for Javacc-Maven-Plugin

Eclipse Indigo (3.7) has out-of-the-box support for it.

As of this writing, Eclipse Kepler (4.3) still lacks support for Maven's javacc plugin. The available m2e connector for javacc-maven-plugin requires a downgrade of Maven components to be installed. manual installation instructions (as of this writing you need to use the development update site). See this post for how to do this with Eclipse Juno (4.2).

The current recommended solution for Eclipse Kepler is to mark `javacc-maven-plugin` as ignored by Eclipse, run Maven from the command line and then modify the project `activemq-core-client` adding the folder `target/generated-sources/javacc` to its build path.

Use *Project Working Sets*

Importing all ActiveMQ Artemis subprojects will create *too many* projects in Eclipse, cluttering your *Package Explorer* and *Project Explorer* views. One way to address that is to use Eclipse's Working Sets feature. A good introduction to it can be found at a Dzone article on Eclipse Working Sets.

Building

We use Apache Maven to build the code, docs, distribution, etc. and to manage dependencies.

The minimum required Maven version is 3.0.0.

Note that there are some compatibility issues with Maven 3.X still unsolved. This is specially true for the 'site' plugin.

Full Release

The full release uses `gitbook` to build a static website from the documentation, if you don't have `gitbook` installed then you can build the distribution without docs (see below) or install `gitbook` using `npm`:

```
$ npm install -g gitbook gitbook-cli
```

If you don't have `npm` installed then you would need to install it first.

Install npm On Fedora

```
$ yum install npm
```

Install npm On Mac-OS

The easiest way would be through brew brew

You first install brew using the instructions on the brew website.

After you installed brew you can install npm by:

```
brew install npm
```

To build the full release with documentation, Javadocs, and the full web site:

```
$ mvn -Prelease package
```

To install it to your local maven repo:

```
$ mvn -Prelease install
```

Build the distribution without docs

It is possible to build a distribution with out the manuals and Javadocs if you don't have or want `gitbook` installed, simply run

```
$ mvn package
```

Tests

Running Tests

To run the unit tests:

```
$ mvn -Ptests test
```

Generating reports from unit tests:

```
$ mvn install site
```

Running tests individually

```
$ mvn -Ptests -DfailIfNoTests=false -Dtest=<test-name> test
```

where <test-name> is the name of the Test class without its package name

Writing Tests

The broker is comprised of POJOs so it's simple to configure and run a broker instance and test particular functionality. Even complex test-cases involving multiple clustered brokers are relatively easy to write. Almost every test in the test-suite follows this pattern - configure broker, start broker, test functionality, stop broker.

The test-suite uses JUnit to manage test execution and life-cycle. Most tests extend `org.apache.activemq.artemis.tests.util.ActiveMQTestBase` which contains JUnit setup and tear-down methods as well as a wealth of utility functions to configure, start, manage, and stop brokers as well as perform other common tasks.

Check out `org.apache.activemq.artemis.tests.integration.SimpleTest`. It's a very simple test-case that extends `org.apache.activemq.artemis.tests.util.ActiveMQTestBase` and uses its methods to configure a server, run a test, and then `super.tearDown()` cleans it up once the test completes. The test-case includes comments to explain everything. As the name implies, this is a simple test-case that demonstrates the most basic functionality of the test-suite. A simple test like this takes less than a second to run on modern hardware.

Although `org.apache.activemq.artemis.tests.integration.SimpleTest` is simple it could be simpler still by extending `org.apache.activemq.artemis.tests.util.SingleServerTestBase`. This class does all the setup of a simple server automatically and provides the test-case with a `ServerLocator`, `ClientSessionFactory`, and `ClientSession` instance. `org.apache.activemq.artemis.tests.integration.SingleServerSimpleTest` is an example based on `org.apache.activemq.artemis.tests.integration.SimpleTest` but extends `org.apache.activemq.artemis.tests.util.SingleServerTestBase` which eliminates all the setup and class variables which are provided by `SingleServerTestBase` itself.

Keys for writing good tests

Avoid leaks

An important task for any test-case is to clean up all the resources it creates when it runs. This includes the server instance itself and any resources created to connect to it (e.g. instances of `ServerLocator`, `ClientSessionFactory`, `ClientSession`, etc.). This task is typically completed in the test's `tearDown()` method. However, `ActiveMQTestBase` (and other classes which extend it) simplifies this process. As `org.apache.activemq.artemis.tests.integration.SimpleTest` demonstrates, there are several methods you can use when creating your test which will ensure proper clean up *automatically* when the test is torn down. These include:

- All the overloaded `org.apache.activemq.artemis.tests.util.ActiveMQTestBase.createServer(..)` methods. If you choose *not* to use one of these methods to create your `ActiveMQServer` instance then use the `addServer(ActiveMQServer)` method to add the instance to the test-suite's internal resource ledger.
- Methods from `org.apache.activemq.artemis.tests.util.ActiveMQTestBase` to create a `ServerLocator` like `createInVMNonHALocator` and `createNettyNonHALocator`. If you choose *not* to use one of these methods then use `addServerLocator(ServerLocator)` to add the locator to the test-suite's internal resource ledger.
- `org.apache.activemq.artemis.tests.util.ActiveMQTestBase.createSessionFactory(ServerLocator)` for creating your session factory. If you choose *not* to use this method then use `org.apache.activemq.artemis.tests.util.ActiveMQTestBase.addSessionFactory` to add the factory to the test-suite's internal resource ledger.

Create configurations

There are numerous methods in `org.apache.activemq.artemis.tests.util.ActiveMQTestBase` to create a configuration. These methods are named like create*Config(..). Each one creates a slightly different configuration but there is a lot of overlap between them.

In any case, `org.apache.activemq.artemis.core.config.Configuration` is a *fluent* interface so it's easy to customize however you need.

Look at other test-cases

If you need ideas on how to configure something or test something try looking through the test-suite at other test-cases which may be similar. This is one of the best ways to learn how the test-suite works and how you can leverage the testing infrastructure to test your particular case.

Code Formatting

Eclipse code formatting and (basic) project configuration files can be found at the `etc/` folder. You should manually copy them *after importing all your projects*:

```
for settings_dir in `find . -type d -name .settings`; do
   \cp -v etc/org.eclipse.jdt.* $settings_dir
done
```

Do not use the maven-eclipse-plugin to copy the files as it conflicts with m2e.

Validating releases

Setting up the maven repository

When a release is proposed a maven repository is staged.

This information was extracted from Guide to Testing Staged Releases

For examples, the 1.1.0 release had the Maven Repository statged as https://repository.apache.org/content/repositories/orgapacheactivemq-1066.

The first thing you need to do is to be able to use this release. The easiest way we have found is to change your maven settings at `~/.m2/settings.xml`, setting up the staged repo.

file ~/.m2/settings.xml:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<settings>
   <profiles>
      <profile>
         <id>apache-artemis-test</id>
         <repositories>

            <repository>
               <id>artemis-test</id>
               <name>Apache Artemis Test</name>
               <url>https://repository.apache.org/content/repositories/orgapacheactivemq-1066</url>
               <layout>default</layout>
               <releases>
                  <enabled>true</enabled>
               </releases>
               <snapshots>
                  <enabled>true</enabled>
               </snapshots>
            </repository>
         </repositories>

         <pluginRepositories>

            <pluginRepository>
               <id>artemis-test2</id>
               <name>Apache Artemis Test</name>
               <url>https://repository.apache.org/content/repositories/orgapacheactivemq-1066</url>
               <releases>
                  <enabled>true</enabled>
               </releases>
               <snapshots>
                  <enabled>true</enabled>
               </snapshots>
            </pluginRepository>
         </pluginRepositories>
      </profile>
   </profiles>

   <activeProfiles>
      <activeProfile>apache-artemis-test</activeProfile>
   </activeProfiles>
</settings>
```

After you configure this, all the maven objects will be available to your builds.

Using the examples

The Apache ActiveMQ Artemis examples will create servers and use most of the maven components as real application were supposed to do. You can do this by running these examples after the .m2 profile installations for the staged repository.

Of course you can use your own applications after you have staged the maven repository.

Notes for Maintainers

Core ActiveMQ Artemis members have write access to the Apache ActiveMQ Artemis repositories and will be responsible for acknowledging and pushing commits contributed via pull requests on GitHub.

Core ActiveMQ Artemis members are also able to push their own commits directly to the canonical Apache repository. However, the expectation here is that the developer has made a good effort to test their changes and is reasonably confident that the changes that are being committed will not break the build.

What does it mean to be reasonably confident? If the developer has run the same maven commands that the pull-request builds are running they can be reasonably confident. Currently the PR build runs this command:

```
mvn compile test-compile javadoc:javadoc -Pfast-tests -Pextra-tests test
```

However, if the changes are significant, touches a wide area of code, or even if the developer just wants a second opinion they are encouraged to engage other members of the community to obtain an additional review prior to pushing. This can easily be done via a pull request on GitHub, a patch file attached to an email or JIRA, commit to a branch in the Apache git repo, etc. Having additional eyes looking at significant changes prior to committing to the main development branches is definitely encouraged if it helps obtain the "reasonable confidence" that the build is not broken and code quality has not decreased.

If the build does break then developer is expected to make their best effort to get the builds fixed in a reasonable amount of time. If it cannot be fixed in a reasonable amount of time the commit can be reverted and re-reviewed.

Commit Messages

Please ensure the commit messages follow the 50/72 format as described here.

Configuring git repositories

Aside from the traditional `origin` and `upstream` repositories committers will need an additional reference for the canonical Apache git repository where they will be merging and pushing pull-requests. For the purposes of this document, let's assume these ref/repo associations already exist as described in the Working with the Code section:

- `origin` : https://github.com/(your-user-name)/activemq-artemis.git
- `upstream` : https://github.com/apache/activemq-artemis

- Add the canonical Apache repository as a remote. Here we call it `apache`.

  ```
  $ git remote add apache https://git-wip-us.apache.org/repos/asf/activemq-artemis.git
  ```

- Add the following section to your /.git/config statement to fetch all pull requests sent to the GitHub mirror. We are using `upstream` as the remote repo name (as noted above), but the remote repo name may be different if you choose. Just be sure to edit all references to the remote repo name so it's consistent.

  ```
  [remote "upstream"]
      url = git@github.com:apache/activemq-artemis.git
      fetch = +refs/heads/*:refs/remotes/upstream/*
      fetch = +refs/pull/*/head:refs/remotes/upstream/pr/*
  ```

Merging and pushing pull requests

Here are the basic commands to retrieve pull requests, merge, and push them to the canonical Apache repository:

1. Download all the remote branches etc... including all the pull requests.

```
$ git fetch --all
Fetching origin
Fetching upstream
remote: Counting objects: 566, done.
remote: Compressing objects: 100% (188/188), done.
remote: Total 566 (delta 64), reused 17 (delta 17), pack-reused 351
Receiving objects: 100% (566/566), 300.67 KiB | 0 bytes/s, done.
Resolving deltas: 100% (78/78), done.
From github.com:apache/activemq-artemis
 * [new ref]         refs/pull/105/head -> upstream/pr/105
```

2. Checkout the pull request you wish to review

```
$ git checkout pr/105
```

3. Once you've reviewed the change and are ready to merge checkout `master`.

```
$ git checkout master
```

4. Ensure you are up to date

```
$ git pull
```

5. Create a new merge commit from the pull-request. IMPORTANT: The commit message here should be something like: "This closes #105" where "105" is the pull request ID. The "#105" shows up as a link in the GitHub UI for navigating to the PR from the commit message.

```
$ git merge --no-ff pr/105
```

6. Push to the canonical Apache repo.

```
$ git push apache master
```

Rebasing before you merge

If the pull request gets too behind master, it would be better to rebase the branch before merging it. You can do that by either asking the author of the pull request to do such rebase (what is not always possible) or you could do it yourself during merging.

In case you rebase the pull request it is mandatory that write "This closes #105" Where 105 is the pull request ID.

There is a script that helps doing such thing at /scripts/merge-PR.sh.

The script is assuming you have defined remotes named at upstream, apache and origin like specified previously here.

to execute such script simply use:

```
$ <checkout-directory>/scripts/merge-pr.sh <PR number> Message on the PR
```

Example:

```
$  pwd
/checkouts/apache-activemq-artemis

$  ./scripts/merge-pr.sh 175 ARTEMIS-229 address on Security Interface
```

The previous example was taken from a real case that generated this merge commit on #175.

- After this you can push to the canonical Apache repo.

  ```
  $ git push apache master
  ```

Notes:

The GitHub mirror repository (i.e. `upstream`) is cloning the canonical Apache repository. Because of this there may be a slight delay between when a commit is pushed to the Apache repo and when that commit is reflected in the GitHub mirror. This may cause some difficulty when trying to push a PR to `apache` that has been merged on the out-of-date GitHub mirror. You can wait for the mirror to update before performing the steps above or you can change your local master branch to track the master branch on the canonical Apache repository rather than the master branch on the GitHub mirror:

```
$ git branch master -u apache/master
```

Where `apache` points to the canonical Apache repository.

If you'd like your local master branch to always track `upstream/master` (i.e. the GitHub mirror) then another way to achieve this is to add another branch that tracks `apache/master` and push from that branch e.g.

```
$ git checkout master
$ git branch apache_master --track apache/master
$ git pull
$ git merge --no-ff pr/105
$ git push
```