**The OpenHFT Handbook**

Welcome to OpenHFT! This Handbook will be your essential guide to the following:

1. **OpenHFT Tutorial** - how to *quickly* get started building and running your very first “Hello OpenHFT World!” application.
2. **OpenHFT Contributors’ Guide** – For those developers that not only want to use OpenHFT to build their applications, but actually want to contribute code, tests and documentation that helps, for both themselves and the global community, to build a better OpenHFT.
3. **OpenHFT Users’ Guide –** For those developers seeking a thorough, unabridged deep-dive guide into each/every and every of the specific OpenHFT capability modules.
4. **OpenHFT Operators’ Guide –** For those developers that need to be able to package a sound/complete/minimal OpenHFT deployment into non-development operational domains. Empowers operations staff to fully manage OpenHFT deployments w/o any developer tool dependencies (no JDK, no IntelliJ, no mvn, no git – just /bin/ksh and /bin/java).

*OpenHFT Tutorial*

[TO BE PROVIDED]

*OpenHFT Contributors’ Guide*

Feel like contributing to OpenHFT? This guide will help you set up your environment, walk you through best practices, and help you test, debug, improve and add features to OpenHFT.

1. The Basics

In this chapter we quickly walk through the basics on contributing; future chapters go into more depth.

Prequisites

|  |  |
| --- | --- |
| Java 6 or 7 | OpenHFT is baselined on Java 6, and is built and tested using  OpenJDK 6/7 and IBM’s JDK. |
| Maven 3 | The OpenHFT build uses Maven, and we recommend using Maven 3. |
| Git | The OpenHFT source code is stored in Git. |

Issue Management

OpenHFT uses Git issue tracker and googlegroups community forums. JIRA? @Ben: provide URLs for all these.

Version control

OpenHFT uses [git](http://git-scm.com/), hosted on [GitHub](http://github.com/), for version control. You can find the upstream git repository at https://github.com/OpenHFT. To clone the repository:

$ git clone git@github.com:OpenHFT/OpenHFT.git

or to clone your fork:

$ git clone git@github.com:YOUR\_GITHB\_USERNAME/OpenHFT.git

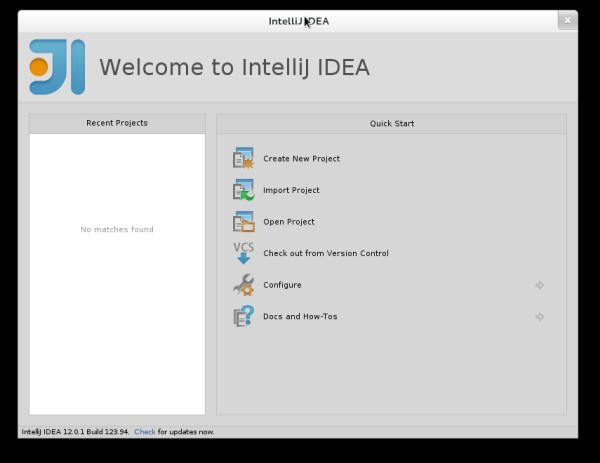
Setting up your IDE

Maven supports generating IDE configuration files for easy setup of a project. This is tested on Eclipse, IntelliJ IDEA and Netbeans.

IntelliJ IDEA

Importing

When you start [IntelliJ IDEA](http://www.jetbrains.com/idea/), you will be greeted by a screen as shown below:



If you have already obtained a copy of the OpenHFT sources via Github (see *'Source Control'*), then follow: *Import Project → /directory/to/downloaded/sources* . IntelliJ will automatically make use of Maven to import the project since it will detect a pom.xml file in the base directory.

If you have not obtained the sources already, you can use the Git integration in IntelliJ IDEA 12. Click on *Check out from Version Control → Github*. After entering your Github credentials, you will then be prompted to enter the git repository URL along with the location that you want to check out the source code to.

Code Style

**(@Ben: all of JPM likes the RedHat IntelliJ code-style plugin, mention it here?)**

Download the code style JAR file from [here](https://github.com/infinispan/infinispan/blob/master/ide-settings/intellij/IntelliJ_IDEA_Code_Style.jar?raw=true) and import this into IntelliJ IDEA.

Builds

OpenHFT uses [Maven](http://maven.apache.org/) for builds. Make sure you have Maven 3 installed, and properly configured.

Continuous Integration

OpenHFT uses [TeamCity](http://www.jetbrains.com/teamcity) for continuous integration. TeamCity polls GitHub for updates and runs whenever updates are available. You can check the status of the latest builds here **(@PETER/ROB: provide?)** .

Testing

OpenHFT uses jUNIT for unit and functional tests, and all OpenHFT tests are run in parallel. For more information see the chapter on the test suite; this chapter gives advice on writing tests which can safely execute in parallel.

Communicating with other OpenHFT contributors

OpenHFT contributors use a mix of technologies to communicate. Google groups. Git issue tracker. **(@BEN: provide links)**

Style Requirements

OpenHFT uses the [K&R code style](http://en.wikipedia.org/wiki/Indent_style#K.26R_style) for all Java source files, with two exceptions:

1. Use 4 space width instead of a tab (8 space width) character for indentations.
2. Curly braces start on the same line for class, interface and method declarations as well as code blocks.

.

Spelling

Ensure correct spelling in code, comments, Javadocs, etc. (use (*American? UK?) English* spelling). It is recommended that you use a spellchecker plugin for your IDE.

Logging

OpenHFT logging standards provided here.

1. Source Control

In this chapter we discuss how to interact with OpenHFT’s source control repository.

Prerequisites

This document assumes some working knowledge of git. We recommend Scott Chacon’s excellent [Pro Git](http://progit.org/) as a valuable piece of background reading. The book is released under the Creative Commons license and can be downloaded in electronic form for free. At very least, we recommend that you read [Chapter 2](http://progit.org/book/ch2-0.html), [Chapter 3](http://progit.org/book/ch3-0.html) and [Chapter 5](http://progit.org/book/ch5-0.html) of *Pro Git* before proceeding.

Repositories

OpenHFT uses http://github.com/OpenHFT/OpenHFT as its canonical repository.

Roles

The project may define *roles* an individual may assume when interacting with the OpenHFT codebase.

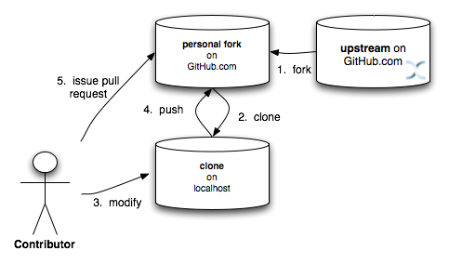
-Contributor

A contributor will only ever submit patches via GitHub’s *pull request* mechanism.

Contributors should *always* fork the upstream project on GitHub and work off a clone of this fork.

All OpenHFT core developers are considered contributors and work off personal forks of the upstream repository. This allows for complex features to be developed in parallel without tripping up over one another. This process is certainly not restricted to just OpenHFT core developers; any contributor should also participate in this manner.

Creating a pull request on GitHub



In this workflow, the contributor forks the OpenHFT upstream repository on GitHub, clones their fork, and makes changes to this private fork. When changes have been tested and are ready to be contributed back to the project, a *pull request* is issued via GitHub so that one of the OpenHFT Admins (Peter’s team) can pull in the change.

**Topic Branches**

It is desirable to work off a *topic branch*, even when using your own, forked repository. A topic branch is created for every feature or bug fix you do. Typically you would create one topic branch per issue, but if several patches are related it’s acceptable to have several commits in the same branch; however different changes should always be identified by different commits.

Before you push your work onto your fork of the repository on GitHub (your *origin*), it is often a good idea to review your commits. Consolidating them (squashing) or breaking them up as necessary and cleaning up commit messages should all be done while still working off your local clone. Also, prior to issuing a pull request, you should make sure you rebase your branch against the upstream branch you expect it to be merged into.  Also, only submit pull requests for your topic branch - not for your master!

A worked example

1. Make sure your master is synced up with upstream.
2. Create new branch for your topic and switch to it. For the example issue, OHFT-1234:

git checkout -b t\_OHFT-12345 master

1. Do your work. Test. Repeat
2. Commit your work on your topic branch
3. Push your topic branch to GitHub. For example:

git push origin t\_OHFT-12345

1. Issue a pull request using the [GitHub pull request system](http://help.github.com/send-pull-requests/)
2. Once your pull request has been applied upstream, delete the topic branch both locally and on your fork. For example:

git branch -d t\_OHFT-12345 && git push origin :t\_OHFT-12345

1. Sync with upstream again so that your changes now appear in your master branch

If your topic branch has been open for a while and you are afraid changes upstream may clash with your changes, it makes sense to rebase your topic branch before you issue a pull request. To do this:

1. Sync your master branch with upstream
2. git checkout master

git pull upstream master

1. Switch to your topic branch. For example:

git checkout t\_OHFT-12345

1. Rebase your topic branch against master:

git rebase master

1. During the rebase process you might need to fix conflicts
2. When you’re done test your code again.
3. Push your rebased topic branch to your repo on GitHub (you will likely need to force this with the -f option).

git push -f origin OHFT-12345

1. Continue your work on your topic branch.

If you are sharing your forked OpenHFT repo with others, then do not rebase! Use a merge instead.

|  |  |
| --- | --- |
|  | . |

Multi-step coordination between developers using forked repositories

Sometimes a feature/task is rather complex to implement and requires competence from multiple areas of the projects. In such occasions it is not uncommon for developers to coordinate feature implementation using personal forks of OpenHFT prior to finally issuing request to integrate into OpenHFT main repository on GitHub.

For example, **developer A** using his personal OpenHFT fork creates a topic branch T and completes as much work as he/she can before requesting for assistance from **developer B**. **Developer A** pushes topic T to his personal OpenHFT fork where **developer B** picks it up and brings it down to his local repo. **Developer B** then in turn completes necessary work, commits his/her changes on branch T, and finally pushes back T to his own personal fork. After issuing request for pull to developer A, **developer B** waits for notification that **developer A** integrated his changes. This exchange can be repeated as much as it is necessary and can involve multiple developers.

A worked example

This example assumes that **developer A** and B have added each others OpenHFT forked repositories with the git add remote command. For example, **developer B** would add developer A’s personal OpenHFT fork repository with the command

git remote add devA https://github.com/developerA/OpenHFT.git

1. **Developer A** starts implementing feature OHFT-244 and works on a local topic branch t\_OHFT244. Developer A pushes t\_OHFT244 to personal OpenHFT fork. For example:

git push origin t\_OHFT244

1. **Developer B** fetches branch t\_OHFT244 to local repository. For example:

git fetch devA t\_OHFT244:my\_t\_ohft244

1. **Developer B** works on local branch my\_t\_ohft244
2. **Developer B** commits changes, pushes my\_t\_ohft244 to own fork.

git push origin my\_t\_ohft244

1. Developer B sends pull request to developer A to integrate changes from my\_t\_ohft244 to t\_ohft244

Project Admin

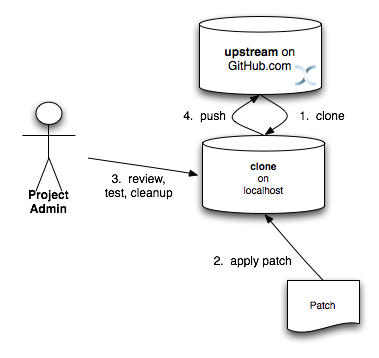
OpenHFT Project Admins have a very limited role. Only Project Admins are allowed to push to upstream, and Project Admins *never* write any code directly on the upstream repository. All Project Admins do is pull in and merge changes from contributors (even if the "contributor" happens to be themselves) into upstream, perform code reviews and either commit or reject such changes.

All Contributors who are also Project Admins are encouraged to not merge their own changes, to ensure that all changes are reviewed by someone else.

|  |  |
| --- | --- |
|  |  |

This approach ensures OpenHFT maintains quality on the main code source tree, and allows for important code reviews to take place again ensuring quality. Further, it ensures clean and easily traceable code history and makes sure that more than one person knows about the changes being performed.

Handling pull requests



Project Admins are also responsible for responding to pull requests. When pulling in changes from a forked repository, more than a single commit may be pulled in. Again, this should be done on a newly created working branch, code reviewed, tested and cleaned up as necessary.

If commits need to be altered - e.g., rebasing to squash or split commits, or to alter commit messages - it is often better to contact the Contributor and ask the Contributor to do so and re-issue the pull request, since doing so on the upstream repo could cause update issues for other contributors later on. If commits were altered or three-way merge was performed during a merge instead of fast-forward, it’s also a good idea to check the log to make sure that the resulting repository history looks OK:

$ git log --pretty=oneline --graph --abbrev-commit  # History messed up due to a bad merge (@QUANT-RISK: what happened?)

\*   3005020 Merge branch 'OHFT-786' of git://github.com/Cotton-Ben/OpenHFT

|\

| \* e757265 OHFT-786 Make dependency to XYZ optional  <-- Same with cb4e5d6 - unnecessary

\* | cb4e5d6 OHF-786 Make dependency to XYZ optional  <-- Ben must have Cherry-picked commit by other admin

|/

\* ...

$ git reset cb4e5d6  # revert the bad merge

It is therefore *strongly recommended* that you use the [handle\_pull\_request](https://github.com/maniksurtani/githelpers/blob/master/project_admins/handle_pull_request) script that ensures a clean merge. If you *still* wish to do this manually, please consider reading through the script first to get an idea of what needs to happen.

|  |
| --- |
|  |

Possible trouble handling pull requests

1. If you have warnings about "Merge made by recursive" you have to fix it rebasing.
2. If you have warnings about "non-fast-forward" you have to rebase.
3. If you see "non-fast-forward updates were rejected" you **must never** use --force on upstream! It means that another patch was merged before you and you have to update your master again, and rebase again.
4. --force is allowed only in special maintenance circumstances. If you find you’re needing it to handle a pull request, then you’re doing it wrong, and the mistake might be a dangerous one! It’s like the good rule of never commit when you’re drunk (drunk coding, however, is allowed).
5. **Never use**--force**on**git push

Using --force while pushing on a shared repository such as *upstream* you could effectively erase other committed patches. No one should ever use this option unless unanimously approved on the public mailing list: the most dangerous aspect of it is that nobody gets any notification if this happens, and we might think issues are solved but you silently removed the fix and it’s history from the repository.

|  |
| --- |
|  |

Cutting releases

Releases can only be cut by OpenHFT Project Admins, and must be done off a recently updated (git fetch and git pull origin) clone of the upstream repo.

Release branches

OpenHFT has several main release branches. These are master (ongoing work on the current unstable release), and maintenance branches for previous minor releases Work should never be committed directly to any of these release branches directly; topic branches should always be used for work, and these topic branches should be merged in using the process outlined above.

Topic branches

Some of the biggest features of git are speed and efficiency of branching, and accuracy of merging. As a result, best practices involve making frequent use of branches. Creating several topic branches a day, even, should not be considered excessive, and working on several topic branches simultaneously again should be commonplace.

[Chapter 3, Section 4](http://progit.org/book/ch3-4.html) of Pro Git has a detailed discussion of topic branches. For OpenHFT, it makes sense to create a topic branch and name it after the JIRA/GitIssue it corresponds to. (if it doesn’t correspond to a JIRA, a simple but descriptive name should be used).

Topic Branches Affecting More Than One Release Branch

Most topic branches will only affect a single release branch, e.g. features targeted at the current unstable release will only affect the master release branch. So a topic branch should be created based off master. Create topic branch off OpenHFT 2.x. For example:

git checkout -b <topic>\_2.x 2.x

1. Create topic branch off master. For example:

git checkout -b <topic>\_master master

1. Do your work on <topic>\_master, test and commit your fixes
2. Switch to <topic>\_2.x. For example:

git checkout <topic>\_2.x

1. Cherry-pick your commit from <topic>\_master onto <topic>\_2.x. For example:

git cherry-pick <commit\_id>

1. Test <topic>\_2.x for correctness, modify as necessary
2. Issue two separate pull requests for both branches

2.3.5. Comments

It is *extremely important* that comments for each commit are clear and follow certain conventions. This allows for proper parsing of logs by git tools. Read [this article](http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html) on how to format comments for git and adhere to them. Further to the recommendations in the article, the short summary of the commit message should be in the following format:

OHFT-XXX Subject line of the JIRA/GitIssue in question

This can optionally be followed by a detailed explanation of the commit. Why it was done, how much of it was completed, etc. You may wish to express this as a list, for example:

* Add a unit test
* Add more unit tests
* Fix regressions
* Solve major NP-Complete problems

Make sure however to split separate concerns - especially if they are unrelated - in separate commits.

Commits

Sometimes work on your topic branch may include several commits. For example, committing a test. Then committing another test. Then perhaps committing a fix. And perhaps fixing your own fix in the next commit… Before issuing a pull request for this topic branch, consider cleaning up these commits. Interactive rebasing helps you squash several commits into a single commit, which is often more coherent to deal with for others merging in your work. [Chapter 6, Section 4](http://progit.org/book/ch6-4.html) of Pro Git has details on how to squash commits and generally, clean up a series of commits before sharing this work with others. Note that you can also easily reorder them, just change the order of lines during the interactive rebase process.

Also, it is important to make sure you don’t accidentally commit files for which no real changes have happened, but rather, whitespace has been modified. This often happens with some IDEs. git diff --check should be run before you issue such a pull request, which will check for such "noise" commits and warn you accordingly. Such files should be reverted and not be committed to the branch.

Adhering to OpenHFT’s code style guidelines will help minimise "noise" commits. Project Admins are going to ask contributors to reformat their code if necessary.

Keeping your repo in sync with upstream

If you have cloned upstream

If you have a clone of the upstream, you may want to update it from time to time. Running:

$ git fetch origin

$ git fetch origin --tags

will often do the trick. You could then pull the specific branches you would need to update:

$ git checkout master

$ git pull origin master

$ git checkout 2.x

$ git pull origin 2.x

Updating topic branches

You should rebase your topic branches at this point so that they are up-to-date and when pulled by upstream, upstream can fast-forward the release branches:

$ git checkout <topic>\_master

$ git rebase master

and/or

$ git checkout topic 2.x

$ git rebase 2.x

If you have forked upstream

If you have a fork of upstream, you should probably define upstream as one of your remotes:

$ git remote add upstream git://github.com/OpenHFT/OpenHFT.git

You should now be able to fetch and pull changes from upstream into your local repository, though you should make sure you have no uncommitted changes. (You *do* use topic branches, right?)

$ git fetch upstream

$ git fetch upstream --tags

$ git checkout master

$ git pull upstream master

$ git push origin master

$ git checkout 2.x

$ git pull upstream 2.x

$ git push origin 2.x

Updating topic branches

Again, you should rebase your topic branches at this point so that they are up-to-date and when pulled by upstream, upstream can fast-forward the release branches:

$ git checkout topic\_master

$ git rebase master

and/or

$ git checkout topic\_2.x

$ git rebase 2.x

.

Tips on enhancing git

Completions

Save [this script](http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;h=168669bbf79cb33c527a688fb906e276beadaf79;hb=HEAD) as ~/.git-completion.bash and in ~/.bash\_profile, add the following on one line:

source ~/.git-completion.bash

After logging out and back in again, typing git followed by TAB will give you a list of git commands, as would git c followed by TAB, etc. This even works for options, e.g. git commit -- followed by TAB. The completions are even aware of your refs, so evengit checkout my\_br followed by TAB will complete to git checkout my\_branch!

|  |  |
| --- | --- |
|  | You get git autocompletion for free if you use [zsh](http://zsh.sourceforge.net/) instead of bash. |

Terminal colors

Add the following to your ~/.gitconfig

**~/.gitconfig**

[color]

ui = yes

[color "branch"]

current = yellow reverse

local = yellow

remote = green

[color "diff"]

meta = yellow bold

frag = magenta bold

old = red bold

new = green bold

[color "status"]

added = yellow

changed = green

untracked = cyan

Aliases

Some git commands are pretty long to type, especially with various switches. Aliases help you to map shortcuts to more complex commands. Again, For example, add the following to ~/.gitconfig:

**~/.gitconfig**

[alias]

     co = checkout

     undo = reset --hard

     cb = checkout -b

     br = branch

     cp = cherry-pick

     st = status

     l = log --pretty=oneline --decorate --abbrev-commit

     lg = log --decorate --abbrev-commit

     last = log --decorate -1 -p --abbrev-commit

     ci = commit -a

     pom = push origin master

     graph = log --pretty=oneline --graph --abbrev-commit

     dt = difftool

Visual History

Git ships with gitk, a GUI that visually represents a log. If you use Mac OS X, [GitX](http://gitx.frim.nl/) is a good alternative. Try typing gitk or gitx in a git project directory. For Linux users, there are lots of alternatives: *gitk* , *gitg* , *giggle*, … up to *egit* for Eclipse.

Visual diff and merge tools

There are several options available, including [KDiff3](http://kdiff3.sourceforge.net/), [meld](http://meld.sourceforge.net/) and Perforce’s [P4Merge](http://www.perforce.com/perforce/products/merge.html) which are all either open source or available for free. See [this link](http://progit.org/book/ch7-1.html) on setting these up (section under *External Merge and Diff Tools*)

Choosing an Editor

You can customise the editor used by git editing ~/.gitconfig. The following fires up [MacVIM](http://code.google.com/p/macvim/) instead of the default vi editor:

**~/.gitconfig**

[core]

     editor = mvim -f

Alternatively, you could fire up TextMate or another editors of your choice.

Shell prompt

You can change your bash shell prompt to print the current repository’s branch name. Add the following to your ~/.bashrc

**~/.bashrc**

function git\_current\_branch {

  git branch --no-color 2> /dev/null | sed -e '/^[^\*]/d' -e 's/\* \(.\*\)/[\1]/'

}

if [ "$PS1" ]; then

  PS1='[\u@\h:\W]$(git\_current\_branch)\$ '

fi

The resulting shell prompt will look like:

bcotton@quant.risk.ny.jpm.com>OpenHFT-2][2.x]$

If you’re a zsh user, you can get even more interesting branch information thanks to [this blog post](http://sebastiancelis.com/2009/nov/16/zsh-prompt-git-users/) , such as:

* whether your branch is dirty (*X*)
* whether it’s ahead of the remote(↑)
* whether it diverges with the remote (↕)
* whether it’s behind (↓)

For example, the following prompt indicates that the current branch is *t\_ohft775\_master* and that it is behind remote:

[~/Go/code/OpenHFT.git]% (t\_OHFT775\_master ↓)

3. Building OpenHFT

OpenHFT uses [Maven](http://maven.apache.org/) as a build system.

Requirements

* Java 6.0 or above
* Maven 3 or above

|  |
| --- |
|  |

Maven

The following is an example settings.xml to get you started. [Nilesh provide?]

mvn Quick command reference

Maven places it’s output in target/

|  |  |
| --- | --- |
|  |  |

| **Command** | **Meaning** |
| --- | --- |
| mvn clean | Cleans out any old builds and binaries |
| mvn compile | Compiles java source code |
| mvn test | Runs the TestNG unit test suite on the compiled code. Will also compile the tests.  See the testing section below for more information on running different test groups.  The default test group run is the "unit" group. |
| mvn package | Packages the module as a JAR file, the resulting JAR file will be in target/ |
| mvn package -DskipTests | Creates a JAR file without running tests |
| mvn package -DskipTests -P minimal-distribution | Creates a reduced version of the distribution with al  l modules,scripts…etc but no javadoc or source code. T  his is very handy to quickly build the distribution in order to run some tests. |
| mvn install -DskipTests | Installs the artifacts in your local repo for use  by other projects/modules, including inter-module  dependencies within the same project. |
| mvn install -P distribution | In addition to install, will also use  Maven’s assembly plugin to build ZIP files  for distribution (in target/distribution ).  Contents of various distribution are controlled by the files in src/main/resources/assemblies . |
| mvn deploy | Builds and deploy the project to the OpenHFT snapshots repository. |
| mvn -Pgenerate-schema-doc install -DskipTests -pl core | Builds the configuration reference HTML file,  often followed up withfirefox core/target/xsd\_doc |
| mvn install -P-extras | Avoids the extras profile disables  the enforce plugin, generation of  source jars and OSGI bundleconstruction,  hence making builds run faster.  Clearly, this option should not be used  when making a release or publishing a snapshot. |

|  |  |
| --- | --- |
|  | For non-snapshot releases (e.g., alphas, betas, release candidates and final releases) you should use the bin/release.pyscript. |

Publishing releases to Maven

To be able to publish releases to Maven, you need to have the following in settings.xml [provide].

Publishing snapshots

Simply running

$ mvn clean deploy -DskipTests

in the OpenHFT root directory will deploy a snapshot.

Publishing releases

Use the bin/release.py script.

The Maven Archetypes

With OpenHFT [archetypes](http://maven.apache.org/guides/introduction/introduction-to-archetypes.html) you can use to create a skeleton project and get started using OpenHFT.

Exploring your new project

The skeleton project ships with a sample application class for interacting with OpenHFT. You can open this new project in your IDE - most good IDEs such as IntelliJ and Eclipse allow you to import Maven projects, see [this guide](http://www.jetbrains.com/idea/webhelp/importing-maven-project.html) and [this guide](http://m2eclipse.sonatype.org/). Once you open your project in your IDE, you should examine the generated classes and read through the comments.

On the command line…

Try running

$ mvn install -Prun

in your newly generated project. This runs the main() method in the generated application class.

Writing a test case for OpenHFT

This archetype is useful if you wish to contribute a test to the OpenHFT project and helps you get set up to use OpenHFT’s testing harness and related tools

As above, this will prompt you for project details and again as above, you should open this project in your IDE. Once you have done so, you will see some sample tests written for OpenHFT making use of OpenHFT’s test harness and testing tools along with extensive comments and links for further reading.

On the command line…

Try running

$ mvn test

in your newly generated project to run your tests. The generated project has a few different profiles you can use as well, using Maven’s -P flag. For example:

$ mvn test -Pudp

Available profiles

The profiles available in the generated sample project are:

* udp: use UDP for network communications rather than TCP
* tcp: use TCP for network communications rather than UDP
* jbosstm: Use the embedded [JBoss Transaction Manager](http://www.jboss.org/jbosstm) rather than OpenHFT’s dummy test transaction manager

Contributing tests back to OpenHFT

If you have written a functional, unit or stress test for OpenHFT and want to contribute this back to OpenHFT, your best bet is to fork the OpenHFT sources on GitHub. The test you would have prototyped and tested in an isolated project created using this archetype can be simply dropped in to OpenHFT’s test suite. Make your changes, add your test, prove that it fails even on OpenHFT’s upstream source tree and issue a [pull request](http://help.github.com/pull-requests/) .

Versions

The archetypes generate poms with dependencies to specific versions of OpenHFT. You should edit these generated poms by hand to point to other versions of OpenHFT that you are interested in.

Source Code

The source code used to generate these archetypes are on GitHub. If you wish to enhance and contribute back to the project, fork away!

4. OpenHFT Modules

In order to provide proper separation between public APIs, common utilities and the actual implementation of OpenHFT is structured as a custodian Parent Module and functional Child Sub-Modules.

* 1. OpenHFT
  2. Java-Lang
  3. Java-Chronicle
  4. Java-Direct
  5. Java-Thread-Affinity
  6. Java-Runtime-Compiler
  7. HugeCollections
  8. TransFIX
  9. SAXophone

.

5. Running and Writing Tests

Tests are written using the jUnit framework.

Running the tests

Before running the actual tests it is highly recommended to configure adjust suite’s memory setting by updating the MAVEN\_OPTS variables. E.g.

$ export MAVEN\_OPTS="-Xms512m -Xmx2048m -XX:MaxPermSize=384m"

The default run executes all tests in the functional and unit groups. To just run the tests with txt and xml output the command is:

$ mvn test

Alternatively, you can execute the tests *and* generate a report with:

$ mvn surefire-report:report

If you are running the tests on a Unix-like operating system, the default limits per user are typically low. The OpenHFT test suite creates a lot of processes/threads, thus you will have to increase your user’s limits and reboot the system to pick up the new values. Open up/etc/security/limits.conf and add the following lines replacing the user name with your username.

**/etc/security/limits.conf**

bigben    soft    nofile    16384

bigben    hard    nofile    16384

bigben    soft    nproc    16384

bigben    hard    nproc    16384

Specifying which tests to run

A single test can be executed using the test property. The value is the short name (not the fully qualified package name) of the test. For example:

$ mvn -Dtest=FqnTest test

Alternatively, if there is more than one test with a given classname in your test suite, you could provide the path to the test.

$ mvn -Dtest=net/OpenHFT/collections/SharedHashMapTest test

Patterns are also supported:

$ mvn -Dtest=net/OpenHFT/collections/\* test

.

Skipping the test run

It is sometimes desirable to install the OpenHFT package in your local repository without performing a full test run. To do this, simply use theskipTests property:

$ mvn -DskipTests install

Note that you should *never* use -Dmaven.test.skip=true since modules' test classes depend on other module test classes, and this will cause compilation errors.

.

Enabling TRACE in test logs

When you run tests, you can get TRACE logging via using the traceTests profile

$ mvn test -PtraceTests

executing this will generate a GZIP file called trace-OpenHFT.log.gz. This file is not fully closed, so to extract the log file, execute:

$ gunzip -c trace-OpenHFT.log.gz > trace-ohft.log

Test groups

Each test should belong to one or more group. The group acts as a filter, and is used to select which tests are ran as part of the maven test lifecycle.

Which group should I use?

The following test groups are used by OpenHFT.

Test permutations

We use the term permutation to describe a test suite execution against a particular configuration. This allows us to test a variety of environments and configurations without rewriting the same basic test over and over again. For example, if we pass JVM parameter-DOpenHFT.test.jpm.risk.protocol=udp test suite is executed using UDP config.

$ mvn -DOpenHFT.test.jpm.risk.protocol=udp test

Each permutation uses its own report directory, and its own html output file name. This allows you to execute multiple permutations without wiping the results from the previous run. Note that due to the way Maven operates, only one permutation can be executed per mvn invocation. So automating multiple runs requires shell scripting, or some other execution framework to make multiple calls to Maven.

6. Helping Others Out

OpenHFT is reliant on the whole community helping each other out. Less experienced contributors are often able to help out answering the "newbie" questions, leaving more experienced contributors to handle the more complex questions.

Users are encouraged to follow the “what topic is hot?” by subscribing to both the Git Issue tracker and google groups community forums.

.

7. Writing Documentation and FAQs

OpenHFT makes use of GoogleDocs as a lightweight collaborative concurrent markup language to write all documentation.

Style guide

By using a consistent voice throughout the documentation, the OpenHFT documentation appears more professional The aim is to make it feel to the user like the documentation was written by a single person. This can only be completely achieved by regular editing, however in order to make the workload of the editor lighter, following these rules will produce a pretty consistent voice.

* Never use abbreviations. On the other hand, contractions are fine.
* Always use the project name "OpenHFT". Never abbreviate it, for example, to "OHFT"
* Always write in the second or third person, never the first (plural or singular forms). Use the second person to emphasize you are giving instructions to the user.

|  |
| --- |
|  |

Writing entirely in the third person can produce quite "dry" text, so it is recommended that you use the second person when you are giving instructions to the user. This could be when you are walking through a sequence of steps they should perform, or could be when you are stating that they *must* do something in order for them to succeed.

So, are there any tricks to reformulate a sentence so the first person is not used?

* Use the passive voice. "I recommend" can become "It is recommended". However, extensive use of the can produce boring, dry and indefinite text, so don’t do this too much!
* Change the subject. For example you can change "Here we discuss" to "This section discusses"
* Use a "chatty" style. Although the use of the first person is avoided, the documentation shouldn’t be too dry. Use the second person as needed. Short sentences and good use of punctuation help too!
* If you define a list, keep the ordering of the list the same whenever you express the list. For example, if you say "In this section you will learn about interceptors, commands and factories" do not go on to say "First, let’s discuss factories". This will subconsciously confuse the user
* You should only capitalize proper nouns only. For example "data grid" is lower case (it’s a concept), whilst "OpenHFT" is capitalized (it’s a project/product name)
* Use the definite article when discussing a specific instance or the indefinite article when describing a generalization of something; generally you omit the article when using a name for a project or product.

Please stay away from colloquialisms at all cost. This impacts the professionalism and readability of the documentation. The examples below probably need no explanation.

*You should use the WidgetMeister API for this sort of problem, coz it’s the fastest way and its pretty cool.*

*You’ve then gotta install the downloaded archive.*

*If the dload fails, contact the SA who runs the svr.*

When writing a glossary or FAQ entry, you should follow the existing entries as a template.

* If the entry is commonly referred to using an acronym, then the title should consistent of the fully expanded name, with the acronym in brackets. You can then use the acronym always within the main text body.
* If you want to refer to other glossary articles using links in the text body, then just link them with no alternative text
* If you want to make external links (e.g. wikipedia, user guide), then add a bulleted list with title "More resources", and list them there. This clearly indicates to users when they are moving outside of our definitions

*The OpenHFT Users’ Guide*

[to be provided]

*The OpenHFT Operators’ Guide*

[Ben’s blunt musings for now– lots of work to be done here – do not underestimate the importance of this content being sound/complete. OpenHFT is a marvelous developer joy, but it has to be operational from the non-developer view. It has to be a non-developer joy too (no exaggeration!)]

Liquidity Risk team:

Progress has been made wrt to our only having demo’d OpenHFT from IntelliJ on a fully loaded developer domain or desktop.

We now have the ability to compellingly demonstrate how a developer can package a suite of OpenHFT applications for deployment to a bare bones non-dev operational domain (on which  only /bin/java and KornShell are available).

Here are the steps the developer should take  to affect a no-Git, no-mvn, no-IntelliJ operational demo of a deployed and operationally executable suite of OpenHFT apps:

      # from developer machine

1.       **cd   /apps/OpenHFT/**                                     #developer’s Fork/Clone OpenHFT root

2.      **mvn dependency:copy-dependencies**         #render sound/complete/minimal .jar hierarchy

3.      # zip up and scp /apps/OpenHFT/ to operational machine

# from operator machine (only /bin/ksh and /bin/java )

1.      Unzip into /apps/OpenHFT

2.      Render sound/complete/minimal CLASSPATH from the .jar hierarchy

**unset CLASSPATH**

**for i in `find /apps/OpenHFT -name \*jar -print `**

**do**

**export CLASSPATH=$i:$CLASSPATH**

**done**

**for i in `find /apps/OpenHFT -name \*classes -print `**

**do**

**export CLASSPATH=$i:$CLASSPATH**

**done**

3.      #all set .. so now demo to them a /bin/ksh script that executes all of OpenHFT Test suits – with only /bin/java

**tput clear**

**cd /apps/OpenHFT  
for i in `find . -name \*.class -print | grep test-classes | grep -v "\\$[0-9a-Z]" `  
do  
export TEST=`echo $i | sed -e 's/\/net/\^net/g'  | cut -f2 -d'^' | sed -e 's/\//\./g' | sed -e 's/.class//g'`  
echo "===================== Running: $TEST ==================================="  
/bin/java org.junit.runner.JUnitCore  $TEST 2>/dev/null > /tmp/testResult  
if [[ `cat /tmp/testResult | grep -i "no runnable methods" | wc -l`  -eq 0 ]]; then  
        cat /tmp/testResult  
else  
        echo "No runnable @Test methods.  Skipping"  
fi  
echo  
echo "====================== FINISHED: $TEST ==================================="  
echo  
#echo "sleeping 5 seconds prior to running next Test"  
sleep 1  
echo  
echo  
done**