Contributing to {brandname}

The {brandname} community

Table of Contents

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

Chapter 1. The Basics

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

1.1. Pre-requisites

Java 11	{brandname} is baselined on Java 8 but needs to be built with Java 11 (we build with OpenJDK)
Maven 3.5	The {brandname} build uses Maven, and you should be using at least Maven 3.5
Git	The {brandname} source code is stored in Git.

1.2. Java compatibility

{brandname} can run with Java 8 or greater. However it **must** be compiled at least with Java 11.

1.3. Issue Management

{brandname} uses JIRA for issue management, hosted on issues.jboss.org. You can log in using your jboss.org username and password.

1.3.1. Reporting Issues

If you need to create a new issue then follow these steps.

1. Choose between
    - *Feature Request* if you want to request an enhancement or new feature for {brandname}
    - *Bug* if you have discovered an issue
    - *Task* if you wish to request a documentation, sample or process (e.g. build system) enhancement or issue
2. Then enter a *Summary* , describing briefly the problem - please try to be descriptive!
3. You should **not** set *Priority*.
4. Now, enter the version you are reporting an issue against in the *Affects Version* field, and leave the *Fix Version* field blank.
5. In the *Environment* text area, provide as much detail as possible about your environment (e.g. Java runtime and version, operating system, any network topology which is relevant).
6. In the *Description* field enter a detailed description of your problem or request.
7. If the issue has been discussed on the forums or the mailing list, enter a reference in the *Forum Reference* field
8. Finally, hit *Create*

1.3.2. Versioning Guidelines

Only {brandname} contributors should set the _Fix Version_ field.

When setting the *Fix Version* field for bugs and issues in JIRA, the following guidelines apply: Version numbers are defined as `${major}.${minor}.${micro}.${modifier}`. For example, `4.1.0.Beta1` would be:

major	4
minor	1
micro	0
modifier	Beta1

If the issue relates to a *Task* or *Feature Request*, please ensure that the `.Final` version is included in the *Fixed In* field. For example, a new feature should contain `4.1.0.Beta1`, `4.1.0.Final` if it is new for 4.1.0 and was first made public in beta1. For example, see ISPN-299 .

If the issue relates to a bug which affected a previous Final version, then the *Fixed In* field should also contain the Final version which contains the fix, in addition to any Alpha, Beta or CR release. For example, see ISPN-546. If the issue pertains to a bug in the current release, then the Final version should not be in the *Fixed In* field. For example, a bug found in 4.1.0.Alpha2 (but not in 4.1.0.Alpha1) should be marked as fixed in 4.1.0.Alpha3, but not in 4.1.0.Final. For example, see ISPN-416.

1.4. Version control

{brandname} uses git, hosted on GitHub, for version control. You can find the upstream git repository at https://github.com/infinispan. To clone the repository:

```
$ git clone git@github.com:infinispan/infinispan.git
```

or to clone your fork:

```
$ git clone git@github.com:YOUR_GITHB_USERNAME/infinispan.git
```

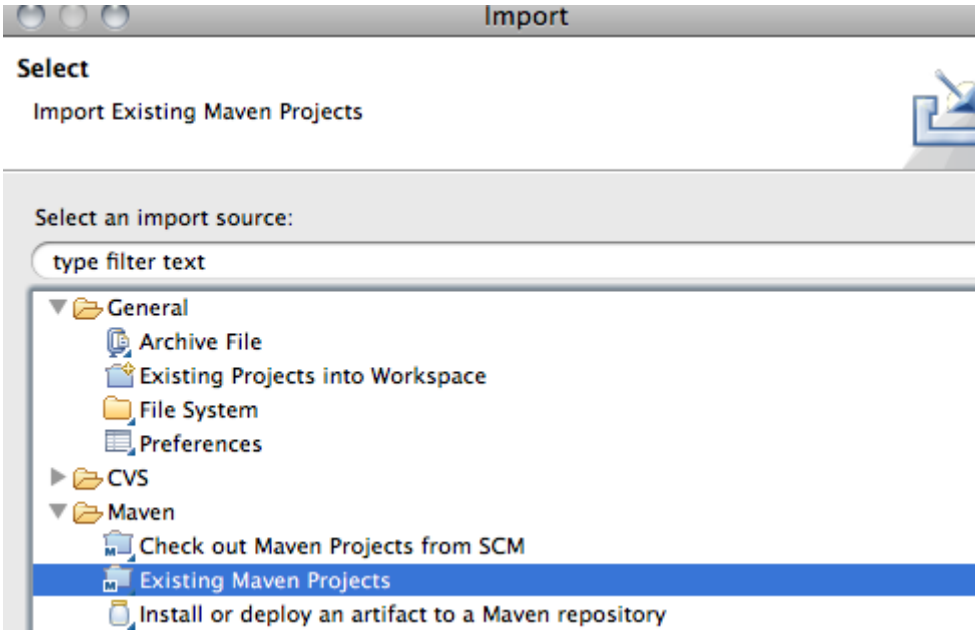
1.4.1. Setting up your IDE

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

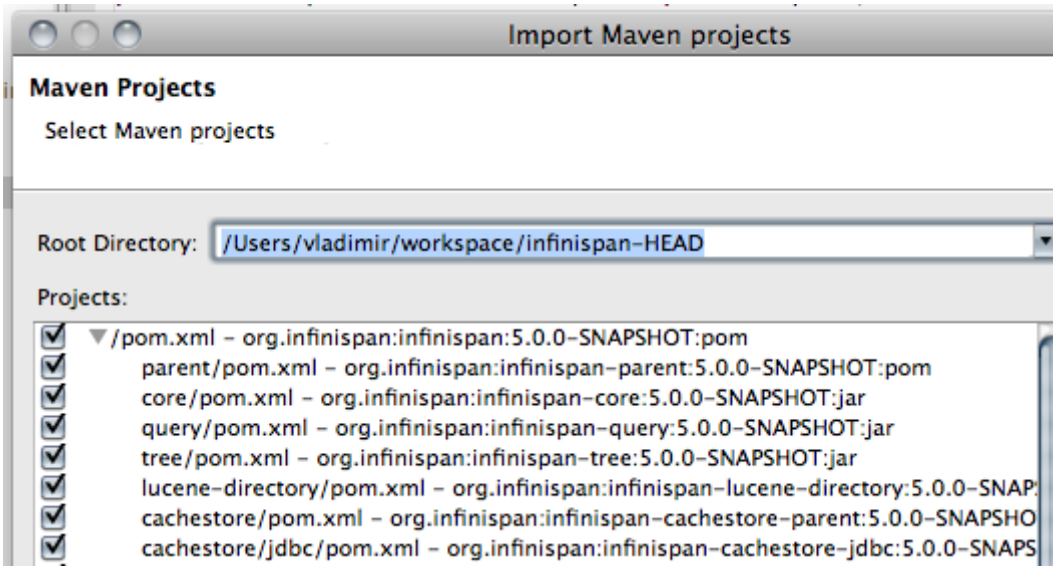
Eclipse

Before we import the project, we need to clone the project as described above.

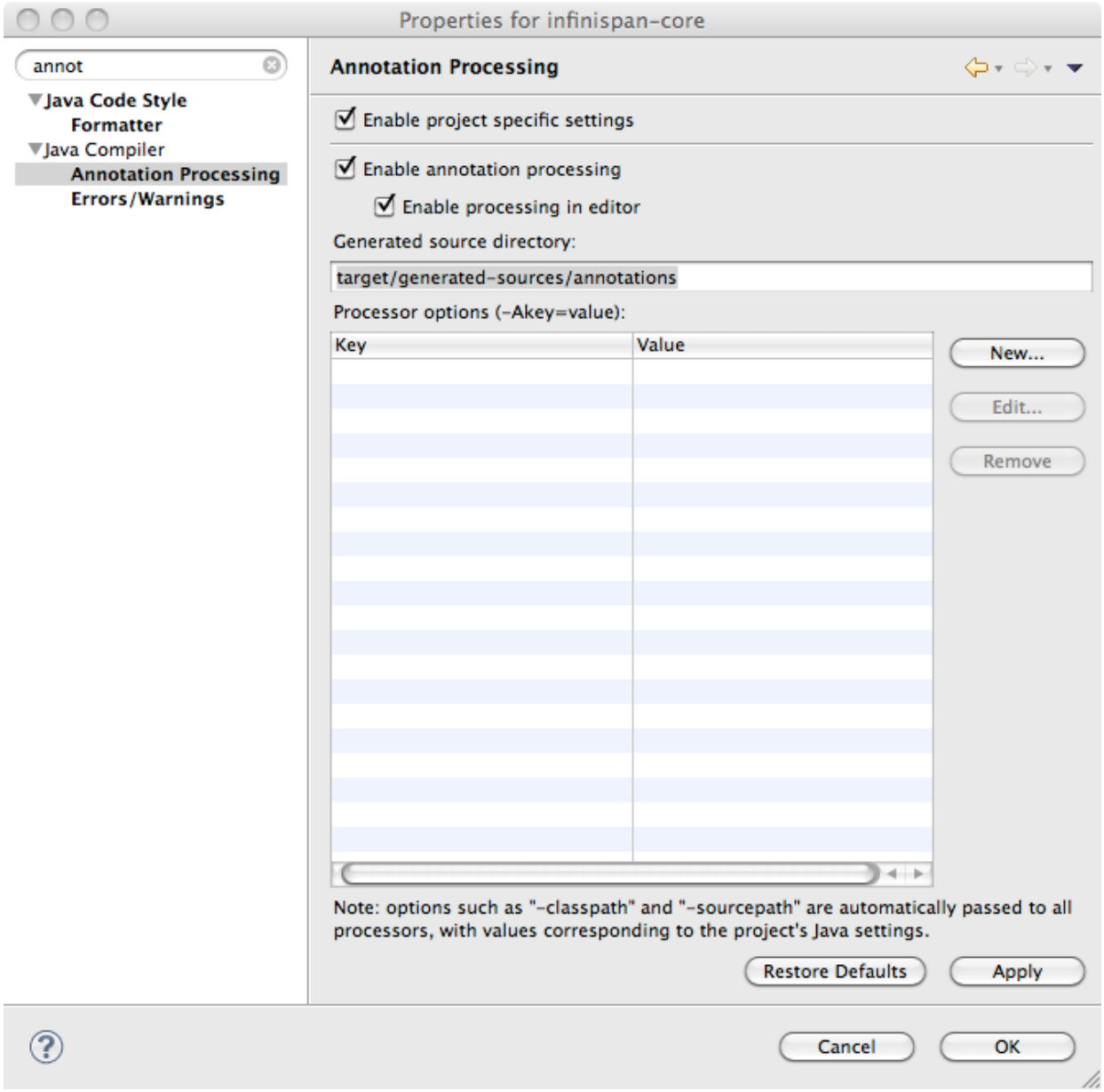
1. Install the m2eclipse plugin if you have not already installed it. This is bundled with Eclipse from version "Indigo" 3.7. For older versions follow instructions on http://eclipse.org/m2e/.
2. Import the {brandname} maven project. Select File → Import in your eclipse workbench. Select the Existing Maven Project importer.



3. Select the root directory of your {brandname} checkout.



4. Select {brandname} modules that you want to import.
5. Finally, from {brandname} 5.0 onwards, annotation processing is used to allow log messages to be internationalized. This processing can be done directly from Eclipse as part of compilation but it requires some set up:
    a. Open the properties for infinispan-core and locate Annotation Processing
    b. Tick Enable project specific settings
    c. Enter `target/generated-sources/annotations` as the `Generated source` directory



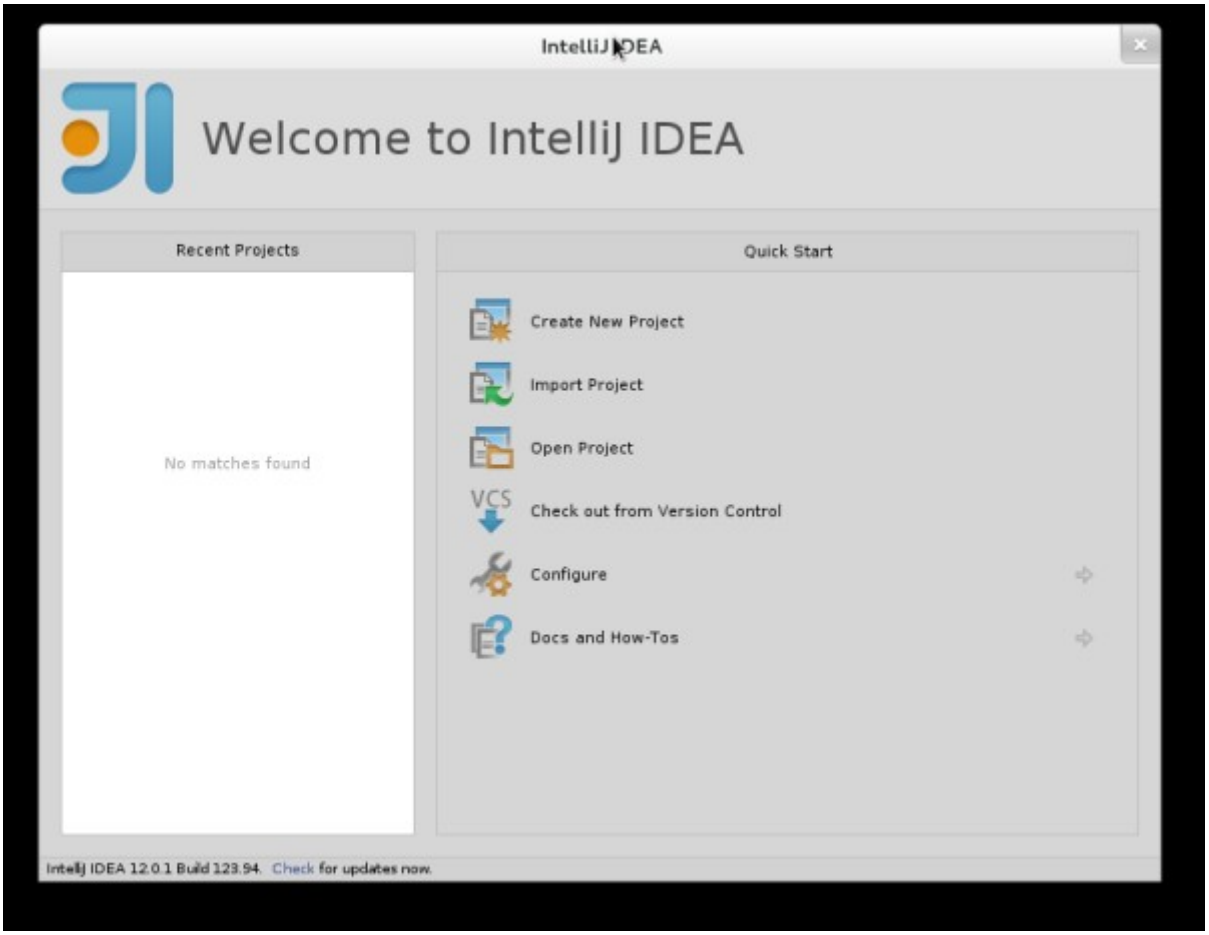
6. Code Formatting. From the menu Window → Preferences → Java → Code Style → Formatter. Import formatter.xml
7. Code templates. From the menu Window → Preferences → Java → Code Style → Code Templates. Import codetemplates.xml

Some modules use Scala, if you plan contributing to one of these modules it's worth installing the Scala IDE. After installing it you need to add "Scala Nature" to a few project of the projects (from the project context menu Configuration → Add Scala Nature), at the moment these projects are: . infinispan-server-core . infinispan-server-hotrod . infinispan-server-memcached . infinispan-server-rest

IntelliJ IDEA

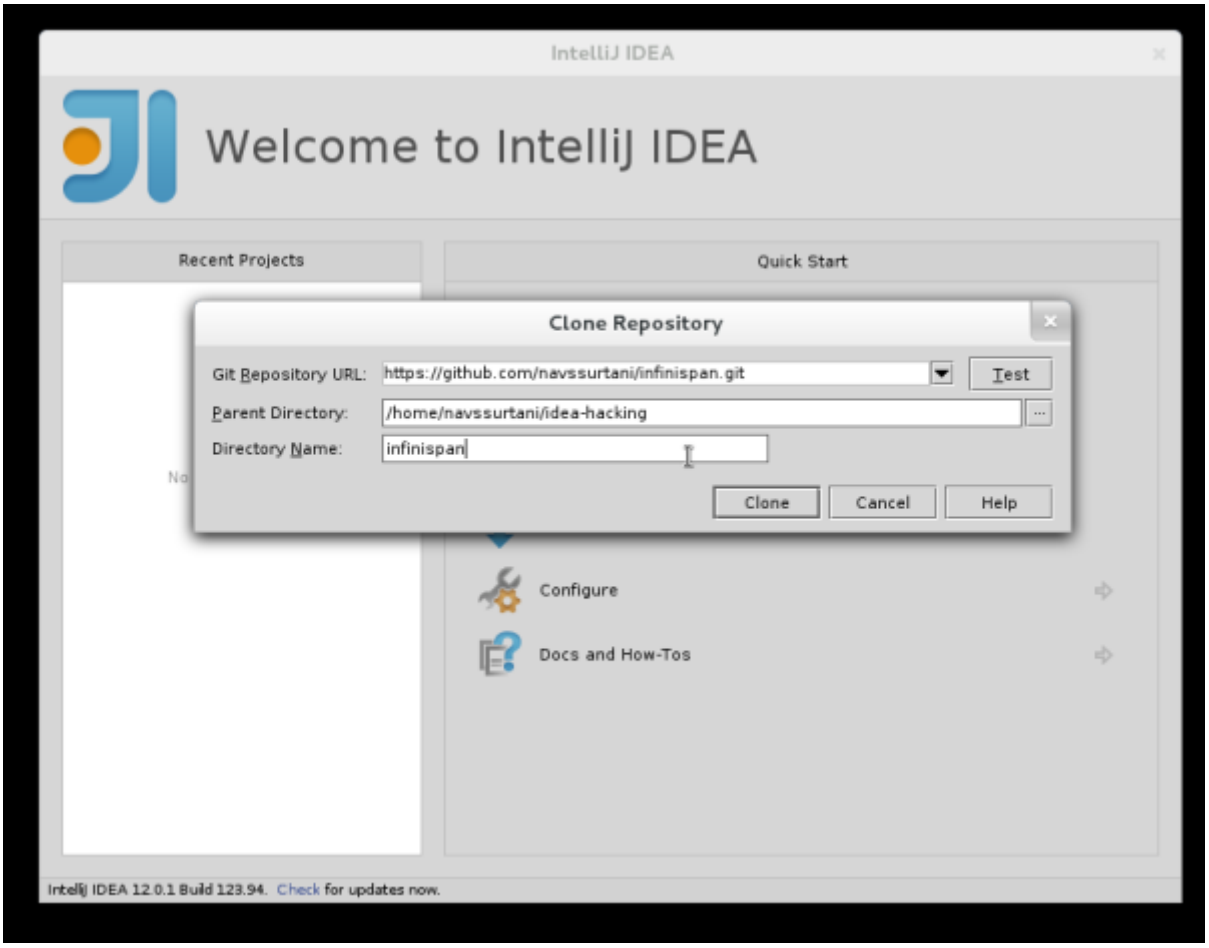
Importing

When you start IntelliJ IDEA, you will be greeted by a screen as shown below:



If you have already obtained a copy of the {brandname} 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.

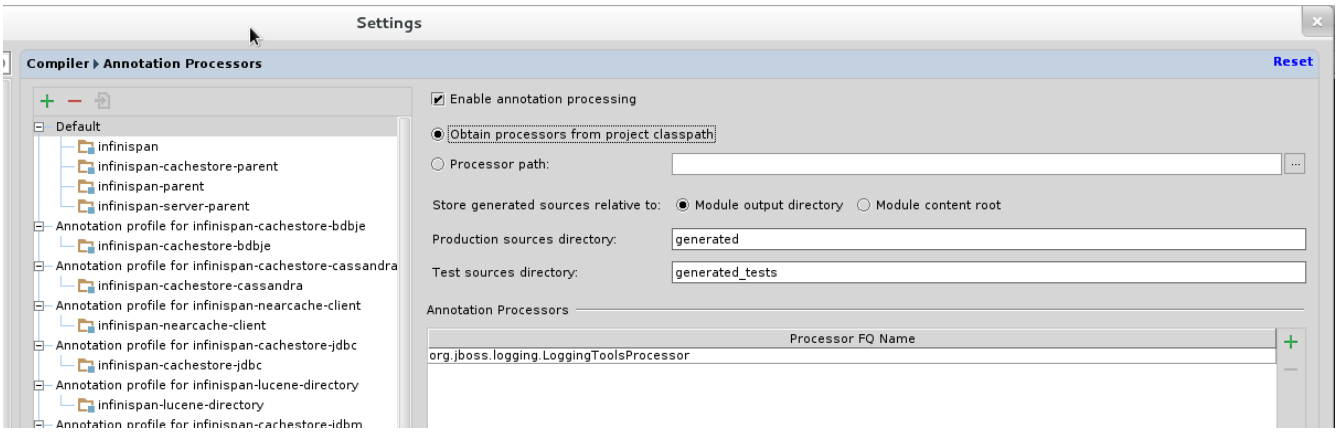
If you have not obtained the sources already, you can use the Git integration in IntelliJ IDEA 12. Click on *Check out from Version Control → Github*. After entering your Github credentials, you will then be prompted to enter the git repository URL along with the location that you want to check out the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



Compiler settings

From {brandname} 5.0 onwards, annotation processing is used to allow log messages to be internationalized. This processing can be done directly from IntelliJ as part of compilation but it requires some set up:

1. Go to Preferences → Compiler → Annotation Processor" and click on *Enable annotation processing*
2. Add an annotation processor with "Processor FQN Name" as `org.jboss.logging.LoggingToolsProcessor`
3. In "Processed Modules", add all modules except the root and the parent modules.

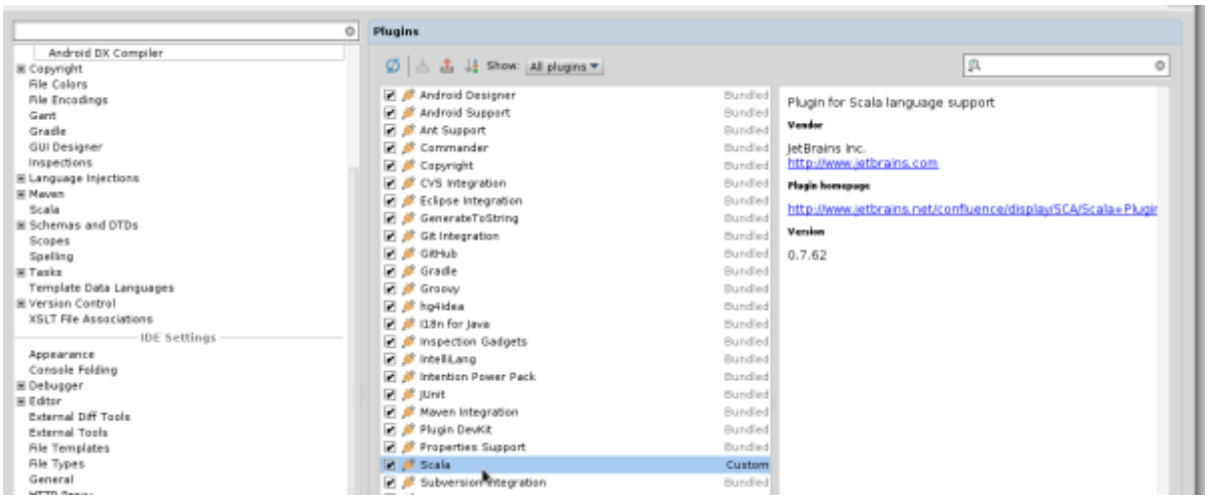


There can sometimes be issues with the generated logging classes on rebuild (particularly when you switch Git branches). If these issues do crop up then simply run `mvn clean install -DskipTests` on the command line to clear them out.

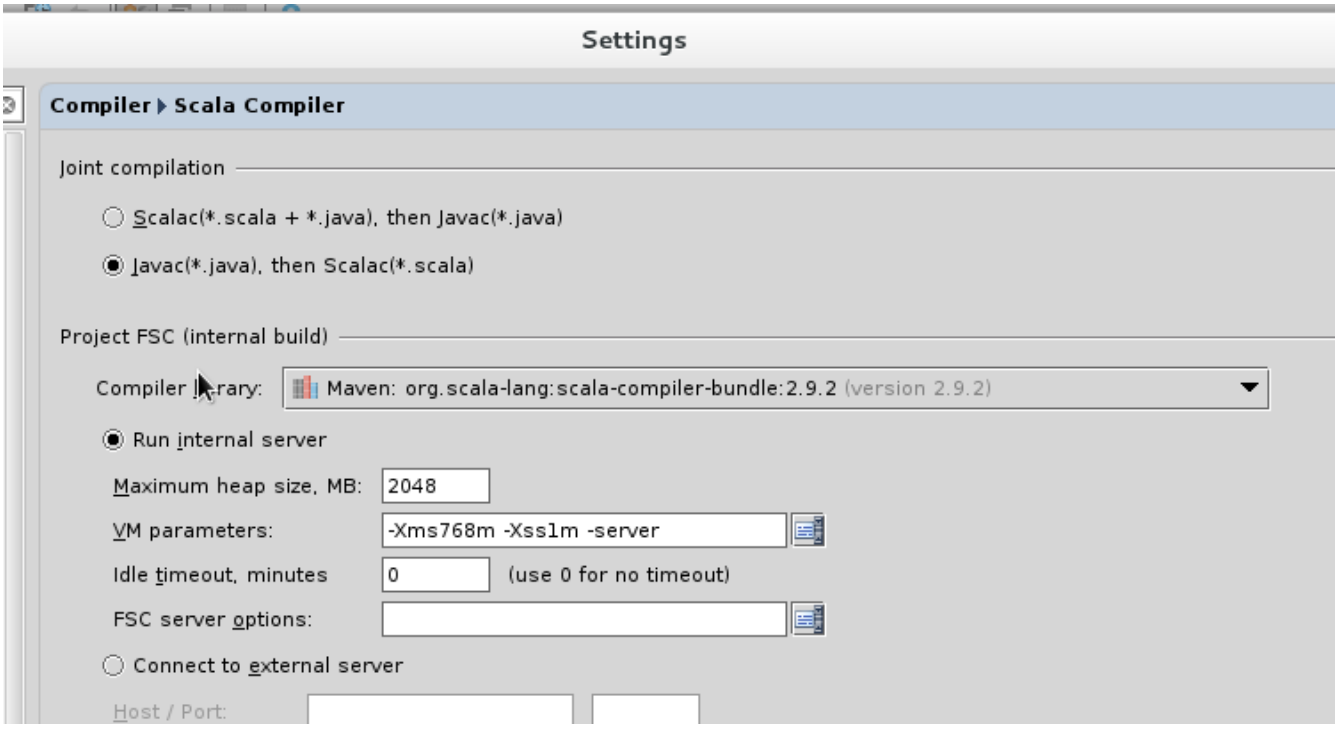
If you are running a multi-core environment (e.g. quad-core or above) then you can follow the instructions on making use of parallelized compilation in IntelliJ 12. Information on how to do this can be found here .

Scala Plugin

You will need to download the Scala plugin for IntelliJ as well. This can be done in *Project Settings → Plugins → Browse Repositories*. Then run a search for 'Scala'. JetBrains themselves are the vendor for this plugin and more information on it can be found here .



You will then have to configure the Scala plugin to use the Scala compiler for Scala files and the Java compiler for Java files. You can do this by going into *Settings → Compiler → Scala Compiler* . Be sure to add the Scala compiler bundle as shown in the screenshot below.



Code Style

Download the code style JAR file from here and import this into IntelliJ IDEA.

1.5. Builds

{brandname} uses Maven for builds. Make sure you have Maven 3 installed, and properly configured. For more information, read the Maven chapter.

1.5.1. Continuous Integration

{brandname} uses 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 .

1.6. Testing

{brandname} uses TestNG for unit and functional tests, and all {brandname} 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.

1.7. Communicating with other {brandname} contributors

{brandname} contributors use a mix of technologies to communicate. Visit this page to learn more.

1.8. Style Requirements

{brandname} uses the K&R code style for all Java source files, with two exceptions:

1. use 3 spaces instead of a tab character for indentations.
2. braces start on the same line for class, interface and method declarations as well as code blocks.

In addition, sure all new line characters used must be LF (UNIX style line feeds). Most good IDEs allow you to set this, regardless of operating system used.

All patches or code committed must adhere to this style. Code style settings which can be imported into IntelliJ IDEA and Eclipse are committed in the project sources, in ide-settings .

1.8.1. Spelling

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

1.8.2. Check-in comments

Please ensure any commit comments use this format if related to a task or issue in JIRA. This helps JIRA pick out these checkins and display them on the issue, making it very useful for back/forward

porting fixes. If your comment does not follow this format, your commit may not be merged into upstream.

1.9. Logging

{brandname} follows the JBoss logging standards, which can be found here .

From {brandname} 5.0 onwards, {brandname} uses JBoss Logging to abstract over the logging backend. {brandname} supports localization of log message for categories of INFO or above as explained in the JBoss Logging guidelines . As a developer, this means that for each INFO, WARN, ERROR and FATAL message your code emits, you need to modify the Log class in your module and add an explicit method for it with the right annotations.

For example:

```
@LogMessage(level = INFO)
@Message(value = "An informative message: %s - %s", id = 600)
void fiveTransactionsHaveCompleted(String param1, String param2);
```

And then, instead of calling `log.info(…)`, you call the method, for example `log.fiveTransactionsHaveCompleted(param1, param2)`. If what you're trying to log is an error or similar message and you want an exception to be logged as cause, simply use `@Cause` annotation:

```
@LogMessage(level = ERROR)
@Message(value = "An error message: %s", id = 600)
void anErrorMessage(String param1, @Cause IllegalStateException e);
```

The last thing to figure out is which id to give to the message. Each module that logs something in production code that could be internationalized has been given an id range, and so the messages should use an available id in the range for the module where the log call resides. Here are the id range assignments per module:

Module name	Id range
core	1 - 1000
tree	2001 - 3000
bdbje cache store	2001 - 3000
cassandra cache store	3001 - 4000
hotrod client	4001 - 5000
server core	5001 - 6000
server hotrod	6001 - 7000
cloud cache store	7001 - 8000
jdbc cache store	8001 - 9000
jdbm cache store	9001 - 10000

Module name	Id range
remote cache store	10001 - 11000
server memcached	11001 - 12000
server rest	12001 - 13000
server websocket	13001 - 14000
query	14001 - 14800
query-dsl	14801 - 15000
lucene directory	15001 - 16000
<no longer used>	16001 - 17000
cdi integration	17001 - 18000
hbase cache store	18001 - 19000
cli interpreter	19001 - 20000
cli client	20001 - 21000
mongodb cache store	21001 - 22000
rest cache store	22001 - 23000
leveldb cache store	23001 - 24000
couchbase cache store	24001 - 25000
redis cache store	25001 - 26000
extended statistics	25001 - 26000
infinispan directory provider	26001 - 27000
tasks	27001 - 27500
scripting	27501 - 28000
remote query server	28001 - 28500
object filter	28501 - 29000
soft-index file store	29001 - 29500
clustered counter	29501 - 30000

When editing the above table, remember to update the README-i18n.txt file in the project sources!

You will need to enable annotation processing in order to be able to compile {brandname} and have the logger implementation generated.

Chapter 2. Source Control

In this chapter we discuss how to interact with {brandname}'s source control repository.

As a convention, *upstream* is used as the name of the http://github.com/infinispan/infinispan repository. This repository is the canonical repository for {brandname}. We usually name *origin* the fork on GitHub of each contributor. So the exact meaning of *origin* is relative to the developer: you could think of *origin* as your own fork.

2.1. Prerequisites

This document assumes some working knowledge of git. We recommend Scott Chacon's excellent Pro Git 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, Chapter 3 and Chapter 5 of *Pro Git* before proceeding.

2.2. Repositories

{brandname} uses http://github.com/infinispan/infinispan as its canonical repository, and this repository contains the stable code on master as well as the maintenance branches for previous minor versions (e.g., 5.0.x, 4.2.x, 4.1.x, etc)

Typically, only *Project Admins* would be able to push to this repository while all else may clone or fork the repository.

2.3. Roles

The project assumes one of 2 *roles* an individual may assume when interacting with the {brandname} codebase. The 2 roles here are:

Roles

- *Contributor*
- *Project Admin* (typically, no more than a small handful of people)

None of the roles assume that you are a Red Hat employee. All it assumes is how much responsibility over the project has been granted to you. Typically, someone may be in more than one role at any given time, and puts on a different "hats" based on the task at hand.

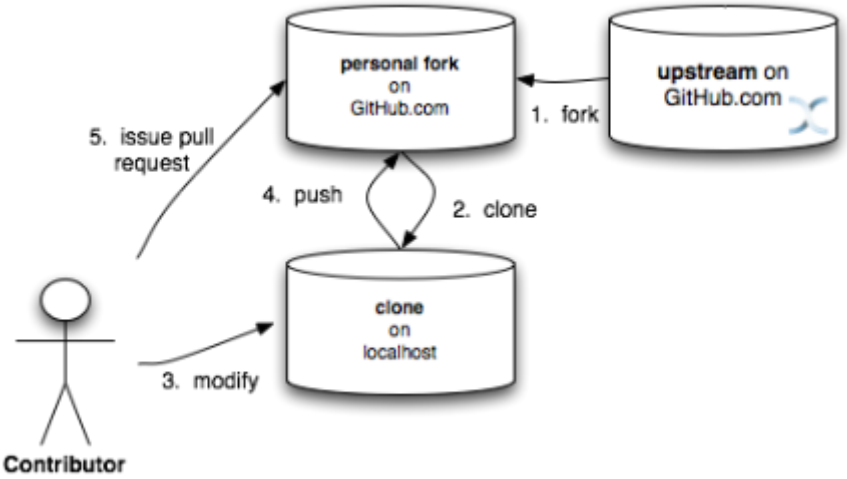
2.3.1. Contributor

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 {brandname} 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 {brandname} core developers; any contributor should also participate in this manner.

Creating a pull request on GitHub



In this workflow, the contributor forks the {brandname} 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 *Project Administrators* can pull in the change.

Topic Branches

NOTE: 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!

The section on *Public Small Project* in Chapter 5, Section 2 of Pro Git has more information on this style of workflow.

A worked example

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

```
git checkout -b t_ISPN-12345 master
```

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

```
git push origin t_ISPN-12345
```

6. Issue a pull request using the GitHub pull request system
7. Once your pull request has been applied upstream, delete the topic branch both locally and on your fork. For example:

```
git branch -d t_ISPN-12345 && git push origin :t_ISPN-12345
```

8. 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

```
git checkout master
git pull upstream master
```

2. Switch to your topic branch. For example:

```
git checkout t_ISPN-12345
```

3. Rebase your topic branch against master:

```
git rebase master
```

4. During the rebase process you might need to fix conflicts
5. When you're done test your code again.
6. 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 ISPN-12345
```

7. Continue your work on your topic branch.

If you are sharing your forked {brandname} 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 {brandname} prior to finally issuing request to integrate into {brandname} main repository on GitHub.

For example, developer A using his personal {brandname} 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 {brandname} 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 {brandname} forked repositories with the `git add remote` command. For example, developer B would add developer A's personal {brandname} fork repository with the command

```
git remote add devA https://github.com/developerA/infinispan.git
```

1. Developer A starts implementing feature ISPN-244 and works on a local topic branch `t_ISPN244`. Developer A pushes `t_ISPN244` to personal {brandname} fork. For example:

   ```
   git push origin t_ISPN244
   ```

2. Developer B fetches branch `t_ISPN244` to local repository. For example:

   ```
   git fetch devA t_ispn244:my_t_ispn244
   ```

3. Developer B works on local branch `my_t_ispn244`
4. Developer B commits changes, pushes `my_t_ispn244` to own fork.

   ```
   git push origin my_t_ispn244
   ```

5. Developer B sends pull request to developer A to integrate changes from `my_t_ispn244` to `t_ispn244`

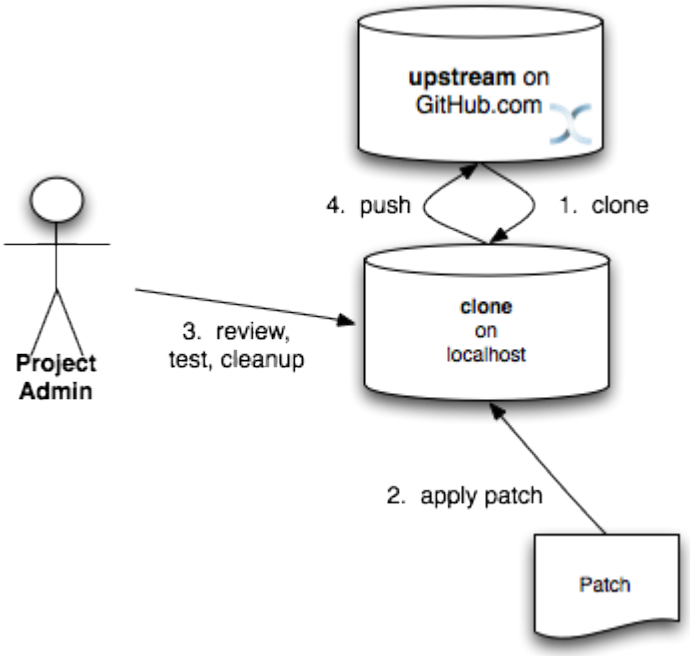
2.3.2. Project Admin

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 {brandname} 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
*   3005020 Merge branch 'ISPN-786' of git://github.com/Sanne/infinispan
|\
| * e757265 ISPN-786 Make dependency to log4j optional  <-- Same with cb4e5d6 -
unnecessary
* | cb4e5d6 ISPN-786 Make dependency to log4j optional  <-- 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` 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.

More information on pulling changes from remote, forked repos can be found in Chapter 5, Section 3 of Pro Git, under *Checking Out Remote Branches* .

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).

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 Project Admins, and must be done off a recently updated (`git fetch` and `git pull origin`) clone of the upstream repo. {brandname}'s `bin/release.py` script takes care of the rest.

2.3.3. Release branches

{brandname} has several main release branches. These are master (ongoing work on the current unstable release), and maintenance branches for previous minor releases (e.g., `5.0.x`, `4.2.x`, `4.1.x`). 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.

2.3.4. 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 of Pro Git has a detailed discussion of topic branches. For {brandname}, it makes sense to create a topic branch and name it after the JIRA 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. However, occasionally, fixes may apply to both release branches 4.2.x as well as master. In this case, the following workflow should apply:

1. Create topic branch off 4.2.x. For example:

```
git checkout -b <topic>_4.2.x 4.2.x
```

2. Create topic branch off master. For example:

```
git checkout -b <topic>_master master
```

3. Do your work on `<topic>_master`, test and commit your fixes
4. Switch to `<topic>_4.2.x`. For example:

```
git checkout <topic>_4.2.x
```

5. Cherry-pick your commit from `<topic>_master` onto `<topic>_4.2.x`. For example:

```
git cherry-pick <commit_id>
```

6. Test `<topic>_4.2.x` for correctness, modify as necessary
7. 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 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:

```
ISPN-XXX Subject line of the JIRA 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.

2.3.6. 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 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 {brandname}'s code style guidelines will help minimise "noise" commits. Project Admins are going to ask contributors to reformat their code if necessary.

2.4. Keeping your repo in sync with upstream

2.4.1. 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 4.2.x
$ git pull origin 4.2.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_4.2.x
$ git rebase 4.2.x
```

2.4.2. 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/infinispan/infinispan.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 4.2.x
$ git pull upstream 4.2.x
$ git push origin 4.2.x
```

A script can do this for you - have a look at `sync_with_upstream` .

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_4.2.x
$ git rebase 4.2.x
```

The `sync_with_upstream` script can do this for you if your topic branch naming conventions match the script.

2.5. Tips on enhancing git

2.5.1. Completions

Save this script 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 even `git checkout my_br` followed by `TAB` will complete to `git checkout my_branch`!

You get git autocompletion for free if you use zsh instead of bash.

2.5.2. 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
```

2.5.3. 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
```

2.5.4. Visual History

Git ships with gitk, a GUI that visually represents a log. If you use Mac OS X, GitX 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.

2.5.5. Visual diff and merge tools

There are several options available, including KDiff3, meld and Perforce's P4Merge which are all

either open source or available for free. See this link on setting these up (section under *External Merge and Diff Tools*)

2.5.6. Choosing an Editor

You can customise the editor used by git editing `~/.gitconfig`. The following fires up MacVIM instead of the default vi editor:

~/.gitconfig

```
[core]
     editor = mvim -f
```

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

2.5.7. 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:

```
trustin@matrix:infinispan-4.2][4.2.x]$
```

If you're a zsh user, you can get even more interesting branch information thanks to this blog post , 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_ispn775_master' and that it is behind remote:

```
[~/Go/code/infinispan.git]% (t_ispn775_master ↓)
```

Chapter 3. Building {brandname}

{brandname} uses Maven as a build system.

3.1. Requirements

- Java 11 or above
- Maven 3.5 or above

Make sure you follow the steps outlined in Maven Getting Started - Users to set up your JBoss repository correctly. This step is *crucial* to ensure your Maven setup can locate JBoss artifacts! If you also want to test the EAP integration modules you should also add the appropriate Enterprise Red Hat Maven Repository.

3.2. Maven

The following is an example `settings.xml` to get you started:

settings.xml

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd" >
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <proxies/>
  <profiles>
    <profile>
      <id>jboss-public-repository</id>
      <repositories>
        <repository>
          <id>jboss-public-repository-group</id>
          <name>JBoss Public Maven Repository Group</name>
          <url> https://repository.jboss.org/nexus/content/groups/public-jboss/ </url>
          <layout>default</layout>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </snapshots>
        </repository>
      </repositories>
```

```
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-public-repository-group</id>
          <name>JBoss Public Maven Repository Group</name>
          <url> https://repository.jboss.org/nexus/content/groups/public-jboss/ </url>
          <layout>default</layout>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>

    <!-- Include early access of application server and other products -->
    <profile>
      <id>redhat-earlyaccess-repository</id>
        <repositories>
          <repository>
            <id>redhat-earlyaccess-repository-group</id>
            <name>Red Hat early access repository</name>
            <url> http://maven.repository.redhat.com/earlyaccess/all/ </url>
            <layout>default</layout>
            <releases>
              <enabled>true</enabled>
              <updatePolicy>never</updatePolicy>
            </releases>
            <snapshots>
              <enabled>true</enabled>
              <updatePolicy>never</updatePolicy>
            </snapshots>
          </repository>
        </repositories>
     </profile>
  </profiles>
  <activeProfiles>
    <activeProfile>jboss-public-repository</activeProfile>
    <activeProfile>redhat-earlyaccess-repository</activeProfile>
  </activeProfiles>
</settings>
```

3.2.1. 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 all modules,scripts...etc but no javadoc or source code. This 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 JBoss snapshots repository.

Command	Meaning
`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.py` script.

3.2.2. Publishing releases to Maven

To be able to publish releases to Maven, you need to have the following in your `settings.xml` file:

settings.xml

```
<settings>
  ...
  <servers>
    ...
    <server>
      <id>jboss-snapshots-repository</id>
      <username>your JBoss.org username</username>
      <password>your JBoss.org password</password>

    </server>
    <server>
      <id>jboss-releases-repository</id>
      <username>your JBoss.org username</username>
      <password>your JBoss.org password</password>

    </server>
    ...
  </servers>
  ...
</settings>
```

Publishing snapshots

Simply running

```
$ mvn clean deploy -DskipTests
```

in the {brandname} root directory will deploy a snapshot.

Publishing releases

Use the `bin/release.py` script.

3.2.3. The Maven Archetypes

{brandname} currently has 2 separate Maven archetypes you can use to create a skeleton project and get started using {brandname}. This is an easy way to get started using {brandname} as the archetype generates sample code, a sample Maven pom.xml with necessary dependencies, etc.

You don't need to have any experience with or knowledge of Maven's Archetypes to use this! Just follow the simple steps below.

Starting a new project

Use the `newproject-archetype` project. The simple command below will get you started, and

```
$ mvn archetype:generate \
   -DarchetypeGroupId=org.infinispan.archetypes \
   -DarchetypeArtifactId=newproject-archetype \
   -DarchetypeVersion=1.0.5 \
   -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

You will be prompted for a few things, including the artifactId , groupId and version of your new project. And that's it - you're ready to go!

Exploring your new project

The skeleton project ships with a sample application class for interacting with {brandname}. 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 and this guide. 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 {brandname}

This archetype is useful if you wish to contribute a test to the {brandname} project and helps you get set up to use {brandname}'s testing harness and related tools. Use

```
$ mvn archetype:generate \
   -DarchetypeGroupId=org.infinispan.archetypes \
   -DarchetypeArtifactId=testcase-archetype \
   -DarchetypeVersion=1.0.5 \
   -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

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 {brandname} making use of {brandname}'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 rather than {brandname}'s embedded transaction manager

Contributing tests back to {brandname}

If you have written a functional, unit or stress test for {brandname} and want to contribute this back to {brandname}, your best bet is to fork the {brandname} 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 {brandname}'s test suite. Make your changes, add your test, prove that it fails even on {brandname}'s upstream source tree and issue a pull request .

Checking coding style

If you have written any new code, it is highly recommended to validate formatting before submitting a Pull Request. This might be done by invoking:

```
$ mvn checktyle:check
```

Versions

The archetypes generate poms with dependencies to specific versions of {brandname}. You should edit these generated poms by hand to point to other versions of {brandname} 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!

Chapter 4. API, Commons and Core

In order to provide proper separation between public APIs, common utilities and the actual implementation of {brandname}, these are split into 3 Maven modules: `infinispan-api`, `infinispan-commons` and `infinispan-core`. This separation also makes sure that modules, such as the remote clients, don't have to depend on infinispan-core and its transitive dependencies. The following paragraphs describe the role of each of these modules and give indication as to what goes where.

4.1. API

The `infinispan-api` module should only contain the public interfaces which can be used in any context (local, remote, etc). Any additions and/or modifications to this module *must* be discussed and approved by the core team beforehand. Ideally it should not contain any concrete classes: rare exceptions may be made for small, self-contained classes which need to be referenced from the API interfaces and for which the introduction of an interface would be deemed cumbersome.

4.2. Commons

The `infinispan-commons` module contains utility classes which can be reused across other modules. Classes in infinispan-commons should be self-contained and not pull in any dependencies (apart from the existing `jboss-logging` and `infinispan-api`). They should also make no reference to configuration aspects specific to a particular environment.

4.3. Core

The `infinispan-core` module contains the actual implementation used for local/embedded mode. When adding new functionality to the APIs, it is generally safe to start by putting them in `infinispan-core` and promoting them to `infinispan-api` only when it is deemed mature enough to do so and it makes sense across the various use-cases.

Chapter 5. Running and Writing Tests

Tests are written using the TestNG framework.

5.1. 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"
```

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 {brandname} 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

```
rhusar    soft    nofile    16384
rhusar    hard    nofile    16384
rhusar    soft    nproc     16384
rhusar    hard    nproc     16384
```

5.1.1. 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=org/infinispan/api/MixedModeTest test
```

Patterns are also supported:

```
$ mvn -Dtest=org/infinispan/api/* test
```

Also, you can always pass your own Log4j configuration file via `-Dlog4.configuration` with your own logging settings.

5.1.2. Skipping the test run

It is sometimes desirable to install the {brandname} package in your local repository without performing a full test run. To do this, simply use the `skipTests` 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.

5.1.3. Running tests using `@Parameters`

If you want to execute tests relying on TestNG's `@Parameters` from an IDE (such as Eclipse or IntelliJ IDEA), please read this blog entry .

5.1.4. Enabling TRACE in test logs

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

```
$ mvn test -PtraceTests
```

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

```
$ gunzip -c trace-infinispan.log.gz > trace-infinispan.log
```

5.1.5. Running tests with JDK 8

While building requires at least JDK 11, the testsuite can be run with JDK 8. In order to do so, you should set the `JAVA8_HOME` environment variable to point to your JDK 8 installation, e.g.:

```
$ export JAVA8_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
```

And enable the `java8-test` maven profile:

```
$ mvn -Pjava8-test test
```

5.1.6. Enabling code coverage generation

When you run tests, you can enable code coverage recorder for calculating and analysing the {brandname} code coverage. You can do this using `coverage` and `jacocoReport` profiles. As a code coverage evaluation tool, the JaCoCo is used.

```
$ mvn test -Pcoverage -Dmaven.test.failure.ignore=true
```

Please note, that `-Dmaven.test.failure.ignore=true` is used for generating JaCoCo code coverage report, even if there are test failures.

Executing this will generate `jacoco.exec` file in each module's `target/` directory. These are the JaCoCo execution data files, which contain full data about the specific module's coverage.

As soon as the coverage execution command is finished, you will need to generate the JaCoCo report, which will merge the generated `jacoco.exec` files as well as will create the code coverage report.

For having the report in place, run the following command from your {brandname} home directory:

```
$ mvn install -pl parent -PjacocoReport
```

The `jacoco-html/` directory will be generated in {brandname} Home directory, which will contain the code coverage report.

5.2. 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.

5.2.1. Which group should I use?

The following test groups are used by {brandname}.

If your test does not fit into one of these groups, a new group should be added.

Test Group	Description
unit	Unit tests using stubs to isolate and test each major class in {brandname}. This is the default group run if no test group is specified
functional	Tests which test the general functionality of {brandname}

Test Group	Description
jgroups	Tests which need to send data on a JGroups Channel
transaction	Tests which use a transaction manager
profiling	Tests used for manual profiling, not meant for automated test runs
manual	Other tests that are run manually

Every test (except those not intended to be run by continuous integration) should at least be in the **functional** or **unit** groups, since these are the default test groups executed by Maven, and are run when preparing a release.

5.3. 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 `-Dinfinispan.test.jgroups.protocol=udp` test suite is executed using UDP config.

```
$ mvn -Dinfinispan.test.jgroups.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.

5.3.1. Running permutations manually or in an IDE

Sometimes you want to run a test using settings other than the defaults (such as UDP for `jgroups` group tests or the EmbeddedTransactionManager for `transaction` group tests). This can be achieved by referring to the Maven POM file to figure out which system properties are passed in to the test when doing something different. For example to run a `jgroups` group test in your IDE using TCP instead of the default UDP, set `-Dinfinispan.test.jgroups.protocol=tcp`. Or, to use JBoss JTA (Arjuna TM) instead of the EmbeddedTransactionManager in a `transaction` group test, set `-Dinfinispan.test.jta.tm=jbosstm`. Please refer to the POM file for more properties and permutations.

5.4. The Parallel Test Suite

{brandname} runs its unit test suite in parallel; {brandname} tests are often IO rather than processor bound, so executing them in parallel offers significant speed up in executing the entire test suite.

5.4.1. Tips for writing and debugging parallel tests

There are a number of constraints and best practices that need to be followed in order to ensure correctness and keep the execution time to a minimum. If you follow these guidelines you will find your tests are more reliable:

- *Each test class is run in a single thread* There is no need to synchronize unit test's fixture, as test methods will be run in sequence. However, multiple test classes are executed in parallel.
- *Each test class must have an unique test name* As a convention, the name of the test should be the fully qualified class name of the test class with the `org.infinispan` prefix removed. For example, given a test class `org.infinispan.mypackage.MyTest` the name of the test should be `mypackage.MyTest`. This convention guarantees a unique name.

MyTest.java

```
package org.infinispan.mypackage;
@Test (testName = "mypackage.MyTest")
public class MyTest { ... }
```

- Use `TestCacheManagerFactory.createXyzCacheManager` and **never** create managers using `new DefaultCacheManager()`. This ensures that there are no conflicts on resources e.g. a cluster created by one test won't interfere with a cluster created by another test.
- Where possible, extend `SingleCacheManagerTestorMultipleCacheManagersTest`. Tests inheriting from these template method classes will only create a cache/cluster once for all the test methods, rather than before each method. This helps keep the execution time down.
- **Never** rely on `Thread.sleep()`. When running in heavily threaded environments this will most often not work. For example, if using ASYNC_REPL , don't use a `sleep(someValue)` and expect the data will be replicated to another cache instance after this delay has elapsed. Instead, use a ReplicationListener (look up javadocs for more information on this class). Generally speaking, if you expect something will happen and you don't have a guarantee when, a good approach is to try that expectation in a loop, several times, with an generous (5-10secs) timeout. For example:

```
while (Systet.currentTimeMillis() - startTime < timeout) {
   if (conditionMeet()) break;
   Thread.sleep(50);
}
```

- `Thread.sleep(10)` may not work in certain OS/JRE combos (e.g. Windows XP/Sun JRE 1.5). Use values grater than 10 for these statements, e.g. 50. Otherwise, a `System.currentTimeMillis()` might return same value when called before and after such a sleep statement.
- For each cache that is create with `TestCacheManagerFactory.createXyzCacheManager()` the test harness will allocate a unique JMX domain name which can be obtained through `CacheManager.getJmxDomain()`. This ensures that no JMX collisions will takes place between any tests executed in parallel. If you want to enforce a JMX domain name, this can be done by using one of the `TestCacheManagerFactory.createCacheManagerEnforceJmxDomain` methods. These methods must be used with care, and you are responsible for ensuring no domain name

collisions happen when the parallel suite is executed.

- Use obscure words. Insert uncommon or obscure words into the cache that has been generated with a random word generator. In a multi-threaded environment like {brandname}'s testsuite, grepping for these words can greatly help the debugging process. You may find this random word generator useful.
- Use the test method name as the key. Grab the test method and use it as part of the cached key. You can dynamically grab the test method using code like this:

```
Thread.currentThread().getStackTrace()(1).getMethodName()
```

Even though we've tried to reduce them to a minimum, intermittent failures might still appear from time to time. If you see such failures *in existing code* please report them in the issue tracker.

Chapter 6. Helping Others Out

{brandname} 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 How to ask a forum question guide.

Forum discussions should be posed as questions (we encourage people to do this). Questions can be marked as answered, indicating to the community that they no longer require answering, allowing easy tracking of open questions. Open questions can be easily viewed using this filter. Community members are encouraged to regularly view the open questions and answer any questions they can.

In order to track what questions are still open, you are encouraged to mark questions as "assumed answered" if you provide information that you think resolves the query and you don’t hear back to the contrary after a week or so.

Approximately every month, a member of the {brandname} team will go through any open questions for the past month and clear up any unanswered questions, either by chasing for an answer from core team, or by checking with the user if they require more info.

Chapter 7. Adding Configuration

We still use the old configuration system internally within {brandname}. This makes things a little complicated. This will be switched out soon! For now, you need to also add your property to the old config system as well as the new.

Note, these guides assume you are adding an element to the cache configuration, but apply equally to the global configuration.

Before you start adding a configuration property, identify whether you want to add a property to an existing configuration group/element, or whether you need to create a child object. We call the configuration group XXX in the steps below.

7.1. Adding a property

1. Add the property to the relevant XXXConfiguration class. Add a private final field, add a parameter to the constructor, and assign the value to the field in the constructor body. Add a accessor for the property. If the property should be mutable at runtime, then add a mutator as well. Most configuration properties are not mutable at runtime - if the configuration is runtime mutable, then {brandname} needs to take notice of this update whilst the cache is running (you can't cache the value of the configuration in your implementation class). Mutators and accessors don't use the classic JavaBean pattern of prepending accessors with "get" and mutators with "set". Instead, the name of the property is used for an accessor. A mutator is an overloaded version of the accessor which takes a parameter, the new value.
2. Add the property to the matching XXXConfigurationBuilder. You'll need to add a mutable field to the class, and initialise it to it's default value in the field declaration. Add a mutator (following the above pattern).
3. The `create()` method is called by the parent object in order to instantiate the XXXConfiguration object from the builder. Therefore, make sure to pass the value of the field in the builder to the XXXConfiguration object's constructor here. Additionally, if you require a complex default (for example, the value of a configuration property is defaulted conditionally based on the value of some other configuration property), then this is the place to do this.
4. The `validate()` method is called by the parent object to validate the values the user has passed in. This method may also be called directly by user code, should they wish to manually validate a configuration object. You should place any validation logic here related to your configuration property. If you need to "cross-validate" properties (validate the value of your property conditionally upon the value of another property), and the other property is on another builder object, increase the visibility of that other property field to "default", and reference it from this builder, by calling the `getBuilder()` method, which will gives you a handle on the root configuration builder.
5. The final step is to add parsing logic to the `Parser` class. First, add the attribute to name to the `Attribute` enum (this class simply provides a mapping between the non-type-safe name of the attribute in XML and a type-safe reference to use in the parser). Locate the relevant `parseXXX()` method on the class, and add a case to the switch statement for the attribute. Call the builder mutator you created above, performing any XML related validation (you are unlikely to need

this), and type conversion (using the static methods on the primitive wrapper classes, String class, or relevant enum class).

7.2. Adding a group

In some situations you may additionally want to add a configuration grouping object, represented in XML as an element. You might want to do this if you are adding a new area of functionality to {brandname}. Identify the location of the new configuration grouping object. It might be added to the root Configuration object, or it might be added to one it's children, children's children. We'll call the parent YYY in the steps below. Create the XXXConfiguration object. Add any properties required following the guide for adding properties. The constructors visibility should be "default".

1. Create the XXXConfigurationBuilder object. It should subclass the relevant configuration child builder – use the YYYConfigurationChildBuilder as the superclass. This will ensure that all builder methods that allow the user to "escape" are provided correctly (i.e provide access to other grouping elements), and also require you to provide a `create()` and `validate()` method. The constructor needs to take the the YYYConfigurationBuilder as an argument, and pass this to the superclass (this simply allows access to the root of the builder tree using the `getBuilder()` method).
2. Follow the property adding guide to add any properties you need to the builder. The `create()` method needs to return a new instance of the XXXConfiguration object. Implement any validation needed in the `validate()` method.
3. In the YYYConfiguration object, add your new configuration class as a private final field, add an accessor, and add initialiser assignment in the constructor

` In the YYYConfigurationBuilder, add your new configuration builder as a private final field, and initialise it in the constructor with a new instance. Finally, add an accessor for it following the standard pattern discussed in the guide.

1. In the YYYConfigurationBuilder ensure that your validate method is called in it's validate method, and that result of the XXXConfiguration instances' create method is passed to the constructor of YYYConfiguration.
2. Finally, add this to the parser. First, add the element to the `Element` class, which provides a type safe representation of the element name in XML. In the `Parser` class, add a new `parseXXX` method, copying one of the others that most matches your requirements (parse methods either parse elements only - look for `ParseUtils.requireNoAttributes()`, attributes only – look for `ParseUtils.requireNoContent()` or a combination of both – look for an iterator over both elements and attributes). Add any attributes as discussed in the adding a property guide. Finally, wire this in by locating the `parseYYY()` method, and adding an element to the switch statement, that calls your new `parseXXX()` method.

7.3. Don't forget to update the XSD and XSD test

1. Add your new elements or attributes to the XSD in `src/main/resources`.
2. Update `src/test/resources/configs/all.xml` to include your new elements or attributes.

7.4. Bridging to the old configuration

Until we entirely swap out the old configuration you will need to add your property to the old configuration (no need to worry about JAXB mappings though!), and then add some code to the `LegacyConfigurationAdaptor` to adapt both ways. It's fairly straightforward, just locate the relevant point in the `adapt()` method (near the configuration group you are using) and map from the legacy configuration to the new configuration, or vs versa. You will need to map both ways, in both adapt methods.

Chapter 8. Writing Documentation and FAQs

{brandname} makes use of AsciiDoc as a lightweight markup language to write all documentation. AsciiDoc is quick and easy to write, easy to learn, and there are a number of ways to render AsciiDoc, including HTML5, PDF and DocBook.

AsciiDoc resources

- AsciiDoc website
- Quick Syntax Reference

8.1. Practicalities

8.1.1. Style guide

AsciiDoctor has some excellent resources on authoring documentation effectively, including:

- Writers' Guide
- Style Guide

8.1.2. Editing

You will want to install the entire AsciiDoctor toolchain on your computer.

- Installing AsciiDoctor
  - On a Mac
- Text editors

In order to see the result of your editing, from the infinispan's root dir (embedded):

```
## compile the user guide and the index
$ asciidoctor ./documentation/src/main/asciidoc/user_guide/user_guide.adoc
$ asciidoctor ./documentation/src/main/asciidoc/index.adoc

## open the documentation and make sure it renders correctly (this is OS-X specific)
$ open /Applications/Google\ Chrome.app documentation/src/main/asciidoc/index.html
```

8.1.3. Linefeed

A soft limit of 80 characters is recommended. At the end of each sentence, go to the next line. Consider going to the next line for each new clause, in particular if the sentence would go beyond 80 characters. But do not obsess: if a multi-clause sentence is below 80 characters, don't split it to limit the *verticality* of the document. For long links, tend to go to the next line.

The 80 characters limit is used because GitHub diffs are around 90 chars long.

For more information, read this blog post

8.1.4. End of file

If you intend your file to be included in another file (aggregated), such as the technique used in the User Guide, then you **must** end the file with two blank lines.

Failure to do so *will* mess up formatting and layout.

8.1.5. Diagrams

As far as possible, use OmniGraffle for diagrams. Please store diagram sources if possible as well, XML files in `src/main/omnigraffle`. Export the omnigraffle files as `png` with a dot per inch of 72. This will create a file of the right size for the web.

Binary images should be stored under `documentation/src/main/asciidoc/${your_document}/images`

8.1.6. Live editing

Naturally, while editing the docs, you don't want to have to build the entire docs to see your changes. A good way to do this is to set up *live previews* as described here.

8.2. Who can contribute documentation?

Anyone. Just fork {brandname} on GitHub, and edit away in the `documents` directory. Then submit a *pull request*, just as you would do with any code contributions. See the Source Control chapter for more information on this process.

8.3. Layout

8.3.1. What goes where?

Official documentation - a User Guide, Getting Started Guide, FAQs, guides to contributing, extending and upgrading {brandname} - live in {brandname}'s source code repository on GitHub.

Additional documentation - such as for cache stores, Hot Rod clients and modules that live outside of {brandname}'s core repository - live alongside the code in their respective repositories following a similar structure.

In addition, {brandname} maintains a wiki for design documents and the like, on this site.

{brandname} no longer uses Confluence for online documentation.

8.3.2. Headers, Page Structure and the Table of Contents

Each *book* has its own structure. Longer books such as the User Guide and the Getting Started Guide may have a table of contents, while shorter books like the guide to extending or upgrading {brandname} may skip this.

Table of Contents

Tables of contents should use the `:toc2:` AsciiDoctor directive at the start of the page.

Each book may choose how many levels deep the TOC should go to, by using the `:toclevels: N` directive, where `1 ⇐ N ⇐ 5`.

Headers

Each book should have a title, along with a section for authors, TOC information, etc. For example, copied from the User Guide:

```
= {brandname} User Guide
Manik Surtani, Mircea Markus, Galder Zamarreño, Pete Muir, and others from the
{brandname} community
:toc2:
:icons: font
:toclevels: 3
:numbered:
```

Chapters

Each chapter thereafter - whether written directly into the book or included via the `include::` directive - should then begin with `==`. For example, from this very document:

```
== Writing Documentation and FAQs
{brandname} makes use of link:http://www.methods.co.nz/asciidoc/[AsciiDoc] as a
lightweight markup language to write all documentation.
```

8.3.3. Images and other media

If you are describing the use of a GUI, or showing results of some operation, images embedded in the page can bring the documentation to life for the reader. Images can be included via the `image::[]` directive.

See this section for details on where to store your images.

8.3.4. Code samples

CodeRay is used for image highlighting. Visit the CodeRay site for a list of supported languages. Highlighting code is as simple as:

```
[source,java]
.MyClass.java
----
// some Java code
----
```

Only include snippets you want to use to demonstrate an idea. If you want to share a reusable block of code or a configuration file, consider storing it in GitHub as a gist and linking to it.

8.3.5. Versioning

If you are writing about a feature that has existed from {brandname} 5.0 onwards, there is no need to specify a version that the feature existed from. However, if you are writing about a new feature, use a `TIP` callout to specify the version it applies to. Also, only specify a MINOR version rather than a detailed version.

Example 1. A bad version statement

This section talks about a new API in {brandname}, called a WidgetMeister. The WidgetMeister has the power to rule all widgets in your cluster, and is included in {brandname} from version 6.2.3.Beta2 onwards.

Why is this bad? A number of reasons.

- It mixes feature detail (what the WidgetMeister does) with versioning (when it was released)
- It points to a beta version!!

How *should* this be written?

Example 2. A good version statement

This section talks about a new API in {brandname}, called a WidgetMeister. The WidgetMeister has the power to rule all widgets in your cluster.

The WidgetMeister API is new in {brandname} 6.2.x.

8.4. Voice and grammar guide

By using a consistent voice throughout the documentation, the {brandname} 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 "{brandname}". Never abbreviate it, for example, to "ISPN"
- 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.

Naturally, most people write in the first person, and, typically find it the easiest form to write, however without a lot of care it can produce the most "unprofessional" text. Conversely, writing in the third person is trickier, but will produce text that feels well written almost without fail. The first person can be used for emphasis but in general it is recommended to avoid it unless you feel confident!

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 "{brandname}" is capitalized (it's a project/product name)
- You should always use American spelling. **Enable a spell checker!**
- 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.

Example 3. Articles used correctly

> *{brandname} uses **a** logging framework to communicate messages to the user, **the** logging framework used by {brandname} is JBoss Logging.*

Let's dig into this. . The sentence states that "{brandname} uses logging", and the indefinite article is used - we are not stating which of many possibilities is used. . The sentence goes on to discuss the logging framework {brandname} uses, and here the definite article is used, as the specific framework in use is discussed. . The sentence is concluded by stating that the logging framework used is called "JBoss Logging", and as this is a product name, no article is used.

This is not a formal or complete description, but is a good rule of thumb.

- Keep the tense the same. It's very easy to slip between the present, past and future tenses, but this produces text that is feels "unnatural" to the reader.

Example 4. Bad tenses

> Data is collected from {brandname} every hour. Upon analysis the data showed that {brandname} is 2 million times faster than it's nearest competitor.

You may not have noticed, but the phrase starts using the present tense (*is*) and slips into the past tense (*showed*). This is clearly not actually the order in which the events happened!

Of course, if you are actually describing the progression of time, then changing tenses is fine.

Example 5. Tenses used correctly

> In the last section you *were* shown how to configure {brandname} using XML, and in the next section you *will be* shown how to configure {brandname} programmatically.

- If you are telling the user about a procedure they can follow, do be explicit about this, and enumerate the steps clearly

8.4.1. Colloquialisms

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

Example 6. Bad colloquialisms

> *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.*

8.5. Glossary and FAQs

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.