Contribution Process

Contribution Processes

Special thanks to Apache Kafka from which most of this documented process was blatantly copied.

Code Change Governing Policy

Apache Knox follows the Commit-Then-Review policy as defined in: http://www.apache.org/foundation/glossary.html

Commit-Then-Review

(Often abbreviated 'CTR' or 'C-T-R'.) A policy governing code changes which permits developers to make changes at will, with the possibility of being retroactively vetoed. C-T-R is an application of decision making through lazy consensus. The C-T-R model is useful in rapid-prototyping environments, but because of the lack of mandatory review it may permit more bugs through in daily practice than the R-T-C alternative. Compare R-T-C , and see the description of the voting process.

Acknowledging the potential negative aspects of not mandating a review for every commit, we have a short set of guidelines and a notification mechanism for ensuring the rapid development that CTR affords the community while providing communication and attention to review comments which are all important to the Knox community.

The following guidelines are prescribed for committers on the Apache Knox project and should be considered for each change commit:

Guidelines

1. For any patches that make fundamental, architectural or security related changes - committers should solicit feedback on the design and document it clearly in an document attached to JIRA or wiki linked to from the JIRA
2. For any patches that are extensions to existing patterns for features - such as adding new service API support, the committer may commit freely - given sufficient tests and documentation (to the committer's discretion)
3. For any patches that are simple bug fixes, the committer may commit freely

Notification

In order to draw attention to review topics the use of the [REVIEW] tag in an email subject will be used for:

1. Reviews from community members post commit - we will use an email with a [REVIEW] tag to indicate that comments are being provided for a merged change and that it needs attention. Committers must review such comments and assess whether they should manifest in:
   a. New JIRAs to address the review comments
   b. A veto discussion and revert upon justification - see the veto process definition
   c. Further documentation is needed
   d. Any combination of above
2. The [REVIEW] tag may also be used by committers or any contributor in order to solicit feedback on design or technical details from the community at large - see guideline #1 above.

Casual Browsing

```
https://gitbox.apache.org/repos/asf?p=knox.git
```

Contributor Workflow

This is the simple workflow and will work well for small features development for people who don't have direct access to check in to the Apache repository. Let's assume you are working on a feature or bug called, KNOX-nnn:

1. Checkout a new repository:

```
git clone https://gitbox.apache.org/repos/asf/knox.git
```

2. Create and checkout a branch to work in:

```
git checkout -b KNOX-nnn remote/origin/master
```

3. Do some work on this branch and periodically checkin locally:

```
git commit -a
```

4. When done (or periodically) rebase your branch to take any changes from trunk:

```
git pull --rebase origin master
```

5. Make a patch containing your work and upload it to JIRA:

```
git format-patch master --stdout > KNOX-nnn.patch
```

6. You may need to iterate/rebase your patch a few times as people comment on the code until a commit checks it in to the main repository.

You will also want to ensure you have your username and email setup correctly so that we correctly record the source of the contribution:

```
git config --global user.name "Joe Coder"
git config --global user.email "jcoder@fake.org"
```

Reviewer Workflow

This assumes you already have a copy of the repository (https://git-wip-us.apache.org/repos/asf/knox.git).

1. Make sure your code is up-to-date:

```
git fetch
```

2. Checkout the destination branch:

```
git checkout master
```

3. See what the patch will do:

```
git apply --stat KNOX-nnn.patch
```

4. See that the patch will apply cleanly (otherwise prod the contributor to rebase):

```
git apply --check KNOX-nnn.patch
```

6. Apply the patch to trunk

```
git am --signoff < KNOX-nnn.patch
```

If you get an error that says "Patch does not have a valid e-mail address." then the patch might have been created by doing git diff in which case you can apply the patch using.

```
patch -p1 < KNOX-nnn.patch
```

If the am operation failed you will also need to remove the .git/rebase-apply/ that gets created

7. If things go wrong (tests fail, you find some problem, etc), you can back out:

```
git reset --hard HEAD
git clean -f
```

8. If after review and running the test you want to push the change into the Apache repo:

```
git push origin master
```

Committer Workflow

If you have commit access on the Apache repository (https://git-wip-us.apache.org/repos/asf/knox.git) then you will not be applying patches in the manner described in the reviewer workflow.
Instead, once your patch is ready, you will check it in yourself as follows:

1. Create a branch to work on:

```
git fetch
git checkout -b KNOX-nnn remotes/origin/master
```

2. Implement the feature. Commit as desired to phase the work.

```
commit -am "<short message about progress.>"
```

3. Rebase as required to track the master branch.

```
git stash
git rebase remotes/origin/master
git stash pop
```

4. Run the tests. Should always rebase before testing prior to submitting a patch.

```
ant verify
```

5. Create a patch, if desired. **Note that commit messages should include the issue identifier and the issue title (e.g., KNOX-{NNNN} - {JIRA TITLE}).**

```
git reset --soft origin/master
git commit -am "KNOX-nnn: <short, meaningful message>"
git format-patch origin/master --stdout > KNOX-nnn.patch
```

6. Post the change patch file to JIRA and optionally get it reviewed.

7. Push the change back to Apache. Pick one of the following:

```
git push
```

Committer Workflow using Git Branches

If you are working on a sizable set of code, for instance, implementing a significant feature, then it is recommended to use a Git remote branch. This will not only help backup your code but also ensure the master branch is never in an unstable state. Also, using a branch will make for easy code reviews and collaboration.

The workflow for creating a remote branch and using it is the following (presuming that you have done a git clone and are in the master branch) :

1. Create a local branch from master:

```
git checkout -b KNOX-nnn
```

2. Push the local branch remote and setup remote tracking

```
git push -u origin KNOX-nnn
```

3. Implement the feature and commit and push as desired. **Note that commit messages should include the issue identifier and the issue title (e.g., KNOX-{NNNN} - {JIRA TITLE}).**

```
git commit -am "<short message about commit>"
git push
```

4. To update the branch from master at any time

```
git pull --rebase origin master
```

5. To merge the branch back into master, first switch to the master branch and then merge.

```
git checkout master
git merge KNOX-nnn
git push
```

6. After all code has been merged and the branch is no longer needed, it can be deleted remotely with:

```
git push origin :KNOX-nnn
```

To delete the branch locally

```
git branch -d KNOX-nnn
```

Github Workflow

Recently the Knox project converted our repository to Gitbox. This allows us to merge pull requests directly using the Github interface. In this chapter you can read about the recommended Github workflow on how to use this approach as a Knox contributor/committer instead of the old style JIRA patches.

1. Checkout source code

Source code checkout in clean environments (i.e. this is the first time you checkout Knox's source)

1. Fork the project on your Github account at https://github.com/apache/knox if you haven't already
2. Clone this fork:

```
# Replace [forked-repository-url] with your git clone url. This sets up the remote alias "origin"
automatically which refers back to your forked repo.
$ git clone [forked-repository-url] knox
```

3. Set upstream remote:

```
$ cd knox
$ git remote add upstream https://github.com/apache/knox.git
```

4. Make sure that your master branches remote to the newly created upstream. Therefore your local `master` branch will be tracking `upstream/master` so when you do a `git pull` by default it will look at `upstream`:

```
$ git checkout master
$ git branch --set-upstream-to upstream/master
```

Source code settings in case you already have Knox's source checked out locally

1. If you are already a Knox contributor and you followed the old contribution flow using JIRA patches you are very likely to have https://github.com/apache/knox checked out locally as follows:

```
$ cd [your knox project path]

$ git checkout master

$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean

$ git remote -v
origin https://github.com/apache/knox.git (fetch)
origin https://github.com/apache/knox.git (push)
```

2. You also need to fork https://github.com/apache/knox.git on your Github account the same way as described above
3. To support consistency with the approach we described above the following changes are recommended in your local repository:

```
# renaming the existing origin to upstream
$ git remote rename origin upstream

# adding your forked repository as a new remote with name origin
$ git remote add origin [forked-repository-url]

$ git remote -v
origin [forked-repository-url] (fetch)
origin [forked-repository-url] (push)
upstream https://github.com/apache/knox.git (fetch)
upstream https://github.com/apache/knox.git (push)

$ git checkout master

$ git branch --set-upstream-to upstream/master
```

2. Keep your fork up to date

```
# Checkout the branch that needs to sync (i.e. master)
git checkout master

# Pulling latest changes from upstream remote (assuming your branch does not have any local changes so that
remote updates can be fast-forwarded)
git pull

# Optionally, push missing changes to your forked repo
git push origin master
```

3. Recommended workflow

3.1. Commit and Push changes

1. Create a branch named KNOX-#### in your local repository and check it out

   ```
   # make sure to track local master branch
   git checkout master

   # creating and checking out the new branch locally
   git checkout -b KNOX-####
   ```

2. Mark the status of the related JIRA as "In Progress" to let others know that you have started working on the JIRA.
3. Run all the tests that are applicable and make sure that all unit tests pass
4. Make changes to the code and commit them to the newly created branch.

   ```
   git add [your-changes]
   git commit -m "KNOX-#### - Your meaningful commit message"
   ```

5. Push your changes. Provide your Github user id and personal access token when asked for user name and password

   ```
   git push origin KNOX-####
   ```

3.2. Creating a Pull Request

1. Navigate to your fork in Github and create a pull request. The pull request needs to be opened against the branch you want the patch to land.
2. The pull request title should be of the form KNOX-#### Title, where KNOX-#### is the relevant JIRA number
3. If the pull request is still a work in progress, and so is not ready to be merged, but needs to be pushed to Github to facilitate review, then add [WIP] before the KNOX-####
4. Consider identifying committers or other contributors who have worked on the code being changed. Find the file(s) in Github and click “Blame” to see a line-by-line annotation of who changed the code last. You can add @username in the PR description or as a comment to request review from a developer.
   Note: Contributors do not have access to edit or add reviewers in the "Reviewers" widget. Contributors can only @mention to get the attention of committers.
5. The related JIRA will automatically have a link to the PR as shown below. Mark the status of JIRA as "Patch Available" manually.

3.3. Travis CI

A Travis CI job is configured to be triggered every time

- a new pull request is created, or
- a new commit is pushed to an already created PR

The job is configured to perform the following tasks:

- Validate the merge
- Build Knox (using JDK-8 and JDK-11) on AMD64 and ARM64 CPU architectures
- Run unit/integration tests

It is the responsibility of the contributor of the pull request to make sure that the build passes. Pull requests should not be merged if the TravisCI job fails to validate the merge.

3.4. GitHub Actions

GitHub Actions is also enabled for the project. It is used as an alternative to Travis CI and might become the main CI in the future.

Pros: It is a bit faster than Travis CI

Cons: it doesn't have non-x86_64 nodes for testing on alternative CPU architectures

4. Review Process

All committers who are invited for review are required to follow the instructions in this page and link their Github accounts with Gitbox to gain Merge access to `apache/knox` in Github.

To try out the changes locally, you can checkout the pull request locally by following the instructions in this guide.

Rules:

- Other reviewers, including committers can try out the changes locally and either approve or give their comments as suggestions on the pull request by submitting a review on the pull request. More help can be found here.
- If more changes are required, reviewers are encouraged to leave their comments on the lines of code that require changes. The author of the pull request can then update the code and push another commit to the same branch to update the pull request and notify the committers.
- The pull request can be merged if at least one committer has approved it or commented "LGTM" which means "Looks Good To Me" and the Travis CI job validated the merge successfully. If you comment LGTM you will be expected to help with bugs or follow-up issues on the patch. (Remember committers cannot review their own patch. If a committer opens a PR, they should make sure that another committer reviews it.
- Sometimes, other changes might be merged which conflict with the pull request's changes. The PR can't be merged until the conflict is resolved. This can be resolved by running `git fetch upstream` followed by `git rebase upstream/[KNOX-####]` and resolving the conflicts by hand, then pushing the result to your branch.
- If a PR is merged, promptly close the PR and resolve the JIRA as "Fixed".

Documentation Contributor Workflow

All of the documentation is maintained in a separate SVN repository to facilitate independent updates.

1. Checkout the SVN repository.

```
svn checkout https://svn.apache.org/repos/asf/knox knox-site
cd knox-site
```

2. Edit site or documentation files. Note that there are no version branches in the SVN repo. Each version of the guides are maintained in separate directories identified by {version} in the example command blow. In addition, most of the guides are in Markdown format and are broken into multiple sections as indicated by {section}.md in the command below. These fils can be edited with the tool of your choice vi is only used as an example.

```
vi trunk/books/{version}/{section}.md
```

3. Generate the new site and review. The command below should be run from the root of the product. The 'review' target causes the local site to be launched in a browser so that you can review the generation result.

```
ant generate review
```

4. If you're only creating a patch, then remove any changes to the generated site. Those changes will be regenerated by the committer as part of committing the patch. If you don't do this it makes it patch review process very difficult. We recommend naming the patch using the JIRA ID. Upload the patch to the JIRA and request review from a committer.

```
svn revert --recursive site
svn diff > KNOX-nnn.patch
```

5. If you're committing changes, then DO NOT revert the generated site content. Commit all the changes.

```
svn ci <modified_files> -m "Change comment"
```

Trademarks

License

Apache Knox uses the standard Apache license.

Privacy Policy

Apache Knox uses the standard Apache privacy policy.