Git Workflow

&

Release Process

Contents

Intro

This document describes the git workflow and release strategy that were successful in multiple real-world projects.

Required git configuration settings

To minimize the potential for problems, please run the following git commands. They will configure GIT to to use safe settings globally (for all repositories):

```
git config --global push.default simple
git config --global rerere.enabled true
```

The option "`push.default`" insures that git will not push multiple branches when executing "`git push`", but only the current branch. Without this configuration option, if used with "`--force`", git will push all branches, and because of the "`--force`" parameter it will override remote history with potentially older history, losing some commits. The option is good to have in any case (and is the default for git 2.0), but is especially critical for the workflow described here, because we need to use forced pushing.

The option "`rerere.enabled`" insures that we don't have to solve the same conflicts multiple times during a rebase. The option is needed because the workflow described here uses rebase. This enables the rerere git feature [https://git-scm.com/blog/2010/03/08/rerere.html], and since it's a configuration setting, you only need to do it once. Of course, if you re-install your OS, you need to do it again.

Development

Branches

- "**master**"
  - This workflow does not use the master workflow. This branch is usually used to always represent the code that is currently deployed to production, but in real-life we didn't find it useful, since every release is tagged, so the production code can always be reproduced by creating a branch from the tag. Moreover, it's very easy to check what version we currently have deployed to production, because each deployable includes such information (e.g. a webapp usually have a "version.html" page or equivalent).
- "**develop**"
  - This is the "integration" branch, into which we merge feature branches.
  - The latest code is here.
- **feature branches**
  - One branch per feature, that follows the pull request / code review process before merging.
  - After merging, these branches are deleted.
- "**release-branches/<major>.<minor>**"
  - This is the branch used to release the version "`<major>.<minor>`" from the `develop` branch.
  - We will re-use this branch for all hotfixes for a release.
  - We will merge this branch into "`develop`" repeatedly - after each normal or hotfix release. If changes from this branch are needed, it can be merged into "`develop`" at any time.
  - After releasing a new "`<major>.<minor>`" version, this branch will be deleted. In other words, we have only one such branch at a time: the one where we stabilize the next release.

Develop and create pull requests

This section gives an example of how development of a new feature is done.

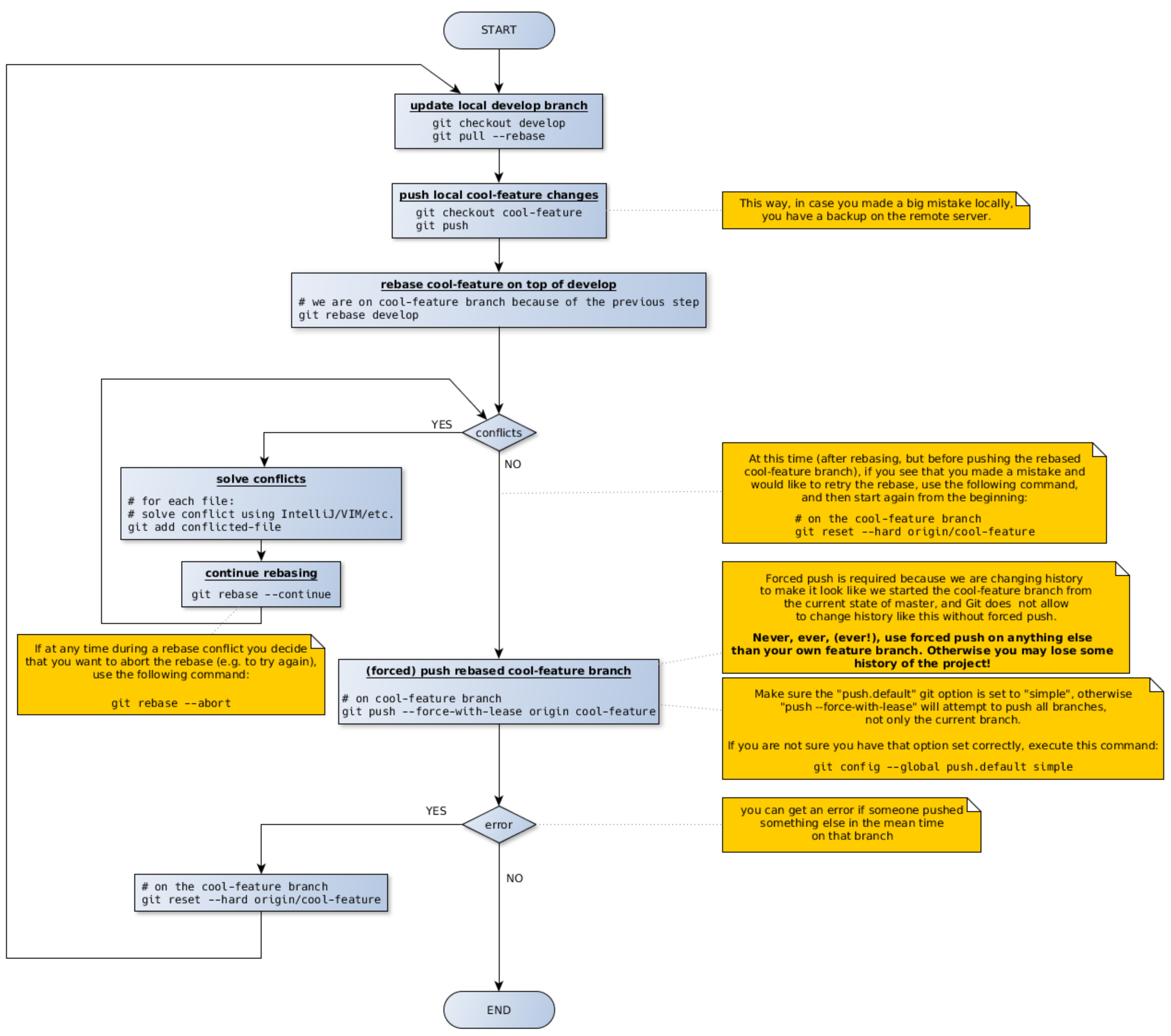
When you start development of a new feature, bug fix, or other piece of work, you **create a new branch from develop**, first making sure it's updated from the remote:

```
git checkout develop
git pull —rebase
git checkout -b cool-feature
git push -u origin cool-feature
```

You have now created the new branch from `develop`, and pushed it to Github. You can delay pushing until after you have done some work, but remember to **push often**, to avoid losing work in case your computer crashes.

Let's say the next day you require in your feature branch some other changes which were merged into `master` in the mean time. In this case, you should **rebase your feature branch on top of `develop`**:

Rebase a feature branch on top of develop



update local develop branch

```
git checkout develop
git pull --rebase
```

push local cool-feature changes

```
git checkout cool-feature
git push
```

This way, in case you made a big mistake locally, you have a backup on the remote server.

rebase cool-feature on top of develop

```
# we are on cool-feature branch because of the previous step
git rebase develop
```

solve conflicts

```
# for each file:
# solve conflict using IntelliJ/VIM/etc.
git add conflicted-file
```

continue rebasing

```
git rebase --continue
```

If at any time during a rebase conflict you decide that you want to abort the rebase (e.g. to try again), use the following command:

```
git rebase --abort
```

At this time (after rebasing, but before pushing the rebased cool-feature branch), if you see that you made a mistake and would like to retry the rebase, use the following command, and then start again from the beginning:

```
# on the cool-feature branch
git reset --hard origin/cool-feature
```

Forced push is required because we are changing history to make it look like we started the cool-feature branch from the current state of master, and Git does not allow to change history like this without forced push.

Never, ever, (ever!), use forced push on anything else than your own feature branch. Otherwise you may lose some history of the project!

(forced) push rebased cool-feature branch

```
# on cool-feature branch
git push --force-with-lease origin cool-feature
```

Make sure the "push.default" git option is set to "simple", otherwise "push --force-with-lease" will attempt to push all branches, not only the current branch.

If you are not sure you have that option set correctly, execute this command:

```
git config --global push.default simple
```

you can get an error if someone pushed something else in the mean time on that branch

```
# on the cool-feature branch
git reset --hard origin/cool-feature
```

Only rebase after pushing all your local changes to GitHub. This way, you make sure you don't lose your code if you make a mistake locally during rebase.

Only use "`--force-with-lease`" to push your feature branch. Don't use "`--force`" because it's dangerous: it will override the remote branch with your code even if someone else pushed there in the mean time, dropping their code.

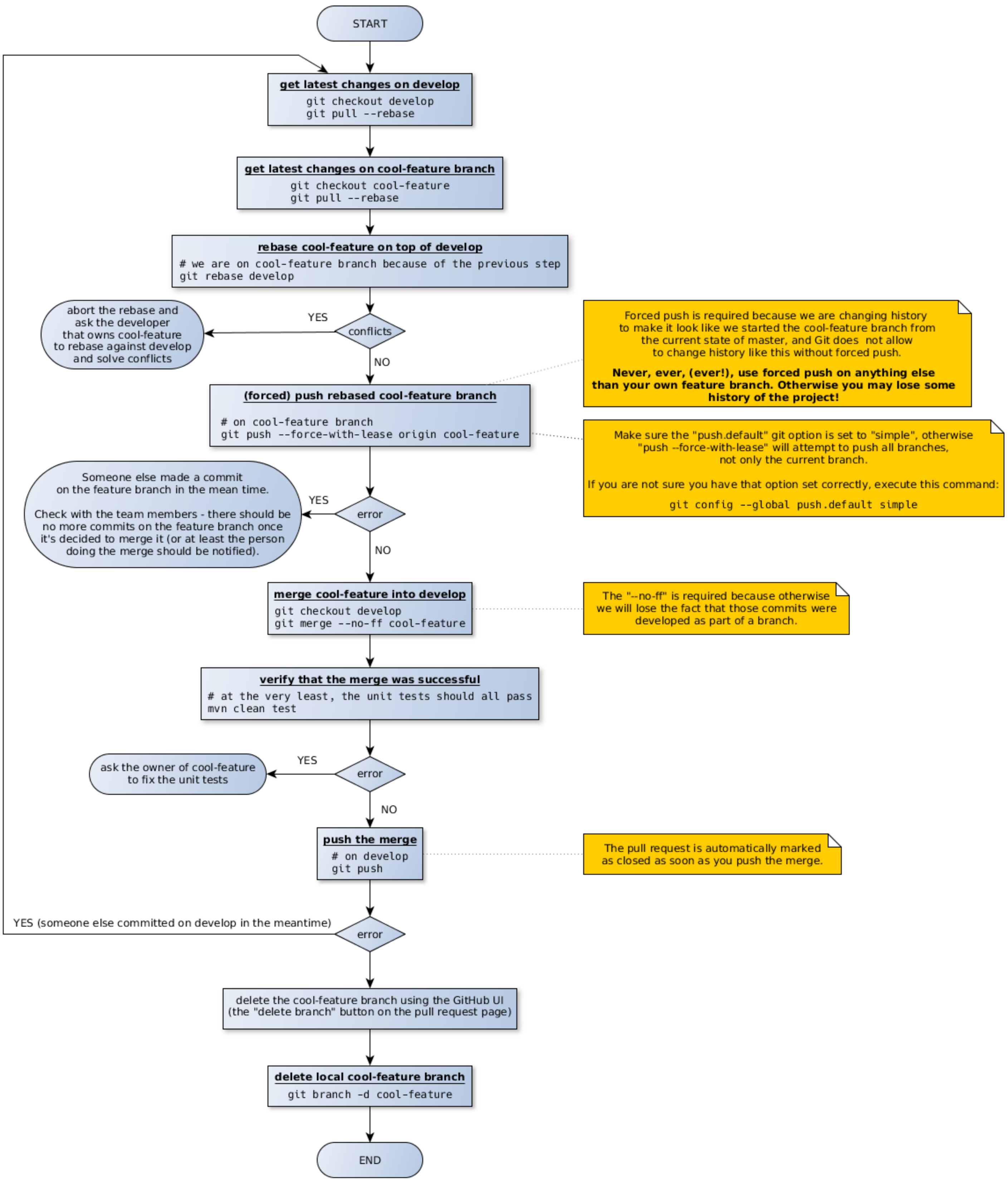
Once you are done with your feature **create a pull request** on Github [https://help.github.com/articles/about-pull-requests/].

4
After creating the pull request, move the associated JIRA ticket from status "In progress" to "Code review".

Merging the pull request

Now, other team members will **check the pull request**, **provide feedback**, and possibly **point out defects** in the code. **Defects have to be fixed** before the pull request may be merged. Once the pull request is approved, it should be **merged to develop** (note that anyone can do this, not only the owner of the feature branch):

Merge a feature branch into develop


START
get latest changes on develop
get latest changes on cool-feature branch
rebase cool-feature on top of develop
conflicts
YES
NO
abort the rebase and ask the developer that owns cool-feature to rebase against develop and solve conflicts
(forced) push rebased cool-feature branch
error
YES
NO
merge cool-feature into develop
verify that the merge was successful
error
YES
NO
ask the owner of cool-feature to fix the unit tests
push the merge
error
YES (someone else committed on develop in the meantime)
delete the cool-feature branch using the GitHub UI (the "delete branch" button on the pull request page)
delete local cool-feature branch
END

```
git checkout develop
git pull --rebase
```

```
git checkout cool-feature
git pull --rebase
```

```
# we are on cool-feature branch because of the previous step
git rebase develop
```

```
# on cool-feature branch
git push --force-with-lease origin cool-feature
```

Forced push is required because we are changing history to make it look like we started the cool-feature branch from the current state of master, and Git does not allow to change history like this without forced push.

Never, ever, (ever!), use forced push on anything else than your own feature branch. Otherwise you may lose some history of the project!

Make sure the "push.default" git option is set to "simple", otherwise "push --force-with-lease" will attempt to push all branches, not only the current branch.

If you are not sure you have that option set correctly, execute this command:

```
git config --global push.default simple
```

Someone else made a commit on the feature branch in the mean time.

Check with the team members - there should be no more commits on the feature branch once it's decided to merge it (or at least the person doing the merge should be notified).

```
git checkout develop
git merge --no-ff cool-feature
```

The "--no-ff" is required because otherwise we will lose the fact that those commits were developed as part of a branch.

```
# at the very least, the unit tests should all pass
mvn clean test
```

```
# on develop
git push
```

The pull request is automatically marked as closed as soon as you push the merge.

```
git branch -d cool-feature
```

After the pull request is merged and the merge pushed (in other words, after following the process in the above diagram), please move the associated JIRA ticket from "Code review" status to "Ready for deployment".

Release

Release candidates

It's best practice to deploy the same exact binary in ACCEPTANCE and PRODUCTION. For this reason, we will only deploy releases produced by the maven release plugin (no snapshots). We will call these builds "release candidates".

Because there is a chance that a release candidate will be rejected after testing in ACCEPTANCE, we may have multiple "release candidates" for a single release, until one of them is accepted.

The release candidate that passes testing in ACCEPTANCE will be deployed to PRODUCTION.

Versioning scheme

The versioning scheme is "`<major>.<minor>.<hotfix>.<release-candidate>`".

All field are required.

They have the following meaning:

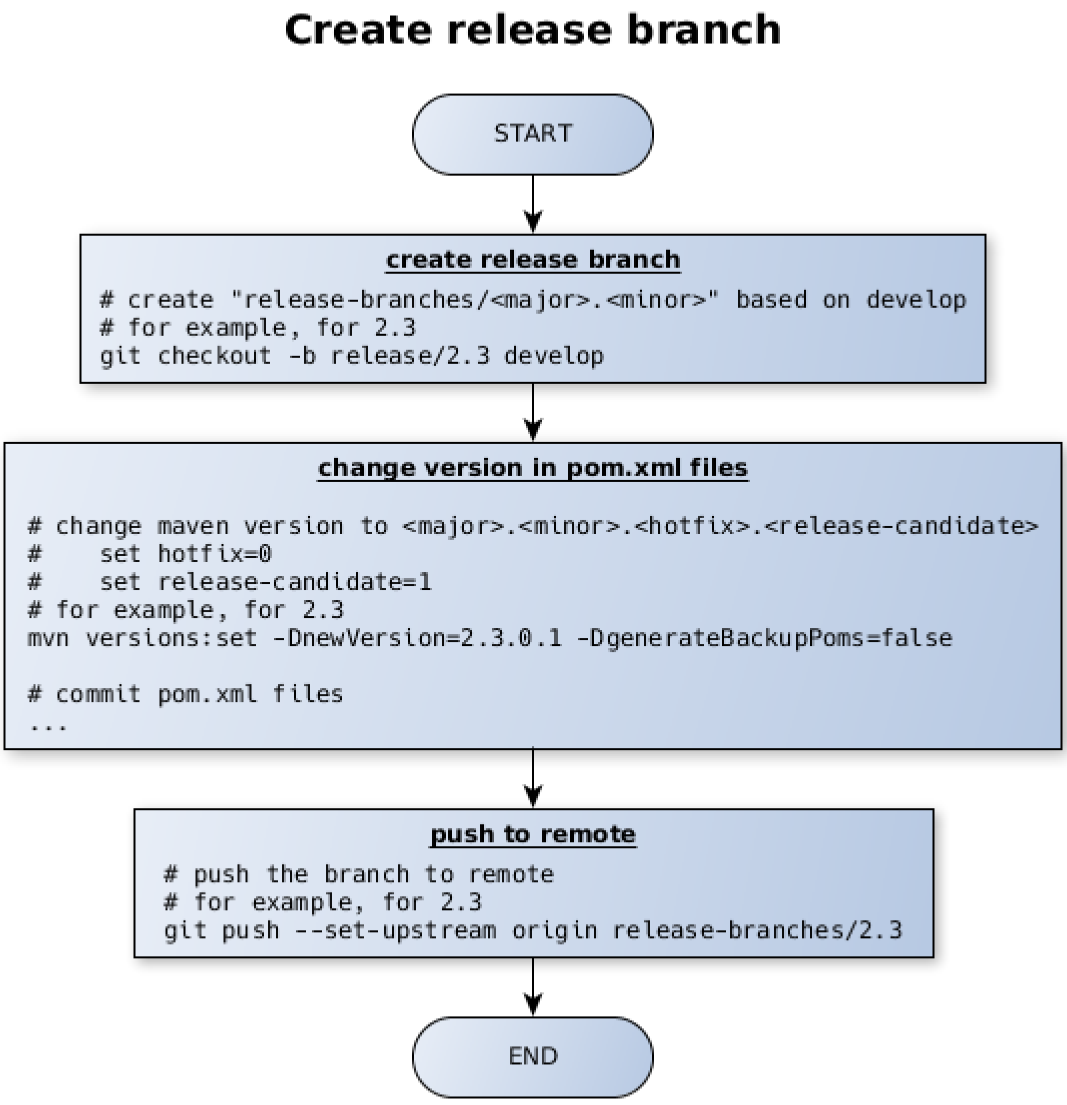
- "`<major>.<minor>`" is the business version
- "`<hotfix>`" is used because we can have multiple releases for the same "`<major>.<minor>`" version.
  - it starts with "0"
  - the first release is "hotfix 0". Hotfix releases after this will have this field incremented, so that the first hotfix will have `hotfix=1`.
- "`<release-candidate>`" is used to track the release candidate number.
  - it starts at "1"
  - ideally, this will always be "1". In real life, a release candidate can be rejected after testing in ACCEPTANCE. If a release candidate is rejected after testing in ACCEPTANCE, we fix the problems, then increment this field, and create a new release candidate to be tested in ACCEPTANCE. We do this until a release candidate is accepted.
- on the "`develop`" branch, the version will be always "`develop-SNAPSHOT`". This makes things easier: we don't need to track the version number on this branch, since we release from a separate branch. We will track the version number on the release branch.

Releasing

Create "<major>.<minor>" release branch

When the "develop" is considered ready to release, we need to create a release branch using the following steps:

- create the release branch and push it to GitHub ("`release-branches/<major>.<minor>`").
- change the version in the `pom.xml` files to "`<major>.<minor>.0.1-SNAPSHOT`" (`hotfix=0`, `release-candidate=1`), commit, and push to GitHub



```
# create "release-branches/<major>.<minor>" based on develop
# for example, for 2.3
git checkout -b release/2.3 develop
```

```
# change maven version to <major>.<minor>.<hotfix>.<release-candidate>
#    set hotfix=0
#    set release-candidate=1
# for example, for 2.3
mvn versions:set -DnewVersion=2.3.0.1 -DgenerateBackupPoms=false

# commit pom.xml files
...
```

```
# push the branch to remote
# for example, for 2.3
git push --set-upstream origin release-branches/2.3
```

This is not needed for hotfix releases, since for hotfixes we will reuse the current release branch.

Create a release candidate

Creating a release candidate should only be done from a release branch ("`release-branches/develop-<major>.<minor>`").

To create a release candidate, just use the Jenkins release job, with the following parameters:

- **release branch**: this is the name of the branch from which to release, for example "`release-branches/2.3`". This needs to be a release branch, not "`develop`"

- **version to release**: "`<major>.<minor>.<hotfix>.<release-candidate>`" (same as the current version in `pom.xml`, without "`-SNAPSHOT`")
- **next development version**: "`<major>.<minor>.<hotfix>.<release-candidate + 1>-SNAPSHOT`" (increment the `<release-candidate>`)

If successful, the release job will upload the artifacts on Nexus.

After the Jenkins build is done, merge the release branch into master, for example:

```
# pom.xml files were changed by Maven in the Jenkins build; get them
git checkout release/2.3
git pull --rebase origin release/2.3

# make sure develop is also up-to-date
git checkout develop
git pull --rebase origin develop

# merge
# the "--no-commit" flag will instruct git not to automatically commit the merge
git merge --no-ff --no-commit release/2.3

# manually revert the version changes in the pom.xml files, to keep the version
at "develop-SNAPSHOT"
...

# commit the merge
git commit

# push new master that contains the merge of the release branch
git push origin develop
```

Once a release candidate is created, it will be deployed to ACCEPTANCE and tested. If it passes testing on ACCEPTANCE, it will become the release. We will take the same maven artifact that we deployed on ACCEPTANCE and we will deploy it to PRODUCTION.

If the release candidate is rejected after testing on ACCEPTANCE, we need to:

- do the necessary code fixes (see the next section)
- create a new release candidate and deploy again on ACCEPTANCE. This one will have an incremented <release-candidate> field in the version.

Work on a release branch

Issues are fixed using pull requests, just as we do on the "develop" branch.

Deploy release candidate to PRODUCTION

After a release candidate has passed testing on ACCEPTANCE, we will deploy it also on PRODUCTION.

After deployment to PRODUCTION, if this is a new "`<major>.<minor>`" version (not a hotfix), we need to delete the release branch associated with the old version. For example, after we deploy, the version 2.3, we need to delete the branch "`release-branches/2.2`". We do this, because we need only one active release branch - the one corresponding to the current PRODUCTION release. We need it in case we need to create hotfixes.

Merge the release branch

After creating a release candidate, the release branch needs to be merged into "`develop`". After merging, please make sure that the maven version in `pom.xml` files still "`develop-SNAPSHOT`".

Create hotfixes

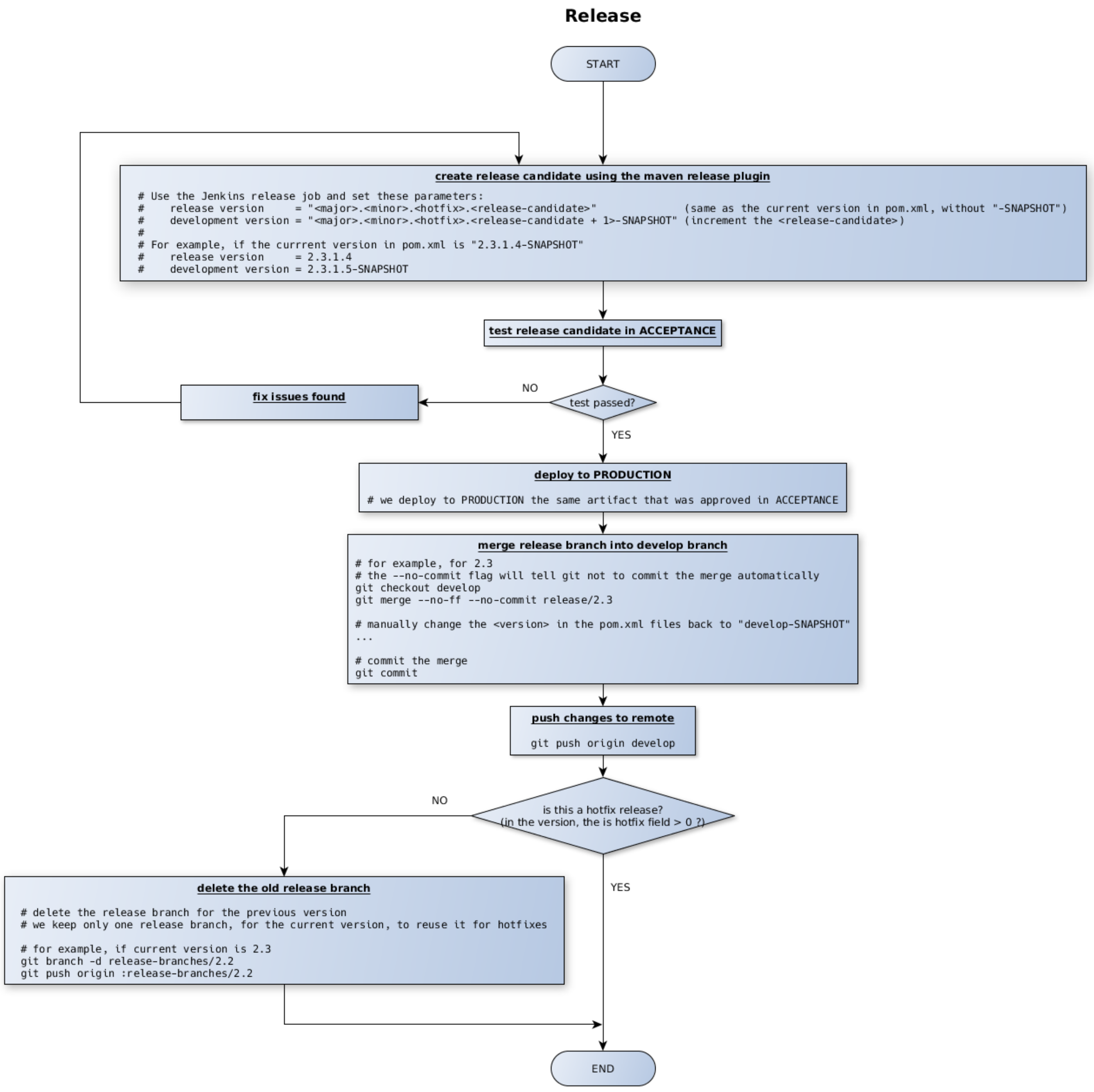
Hotfixes are created on the release branch corresponding with the current release. No new branch is needed.

To create a hotfix:

- change version in `pom.xml` files
  - increment `<hotfix>`
  - set `<release-candidate>` to `1`
  - example: `2.7.2.3` will become `2.7.3.1`
- fix the issue in a feature branch
- create a pull request
- merge the pull request into the release branch

Release diagram

The following diagram summarizes the release process:


Release
START
create release candidate using the maven release plugin
test release candidate in ACCEPTANCE
test passed?
NO
fix issues found
YES
deploy to PRODUCTION
merge release branch into develop branch
push changes to remote
is this a hotfix release?
(in the version, the is hotfix field > 0 ?)
NO
delete the old release branch
YES
END

```
# Use the Jenkins release job and set these parameters:
#    release version     = "<major>.<minor>.<hotfix>.<release-candidate>"              (same as the current version in pom.xml, without "-SNAPSHOT")
#    development version = "<major>.<minor>.<hotfix>.<release-candidate + 1>-SNAPSHOT" (increment the <release-candidate>)
#
# For example, if the currrent version in pom.xml is "2.3.1.4-SNAPSHOT"
#    release version     = 2.3.1.4
#    development version = 2.3.1.5-SNAPSHOT
```

```
# we deploy to PRODUCTION the same artifact that was approved in ACCEPTANCE
```

```
# for example, for 2.3
# the --no-commit flag will tell git not to commit the merge automatically
git checkout develop
git merge --no-ff --no-commit release/2.3

# manually change the <version> in the pom.xml files back to "develop-SNAPSHOT"
...

# commit the merge
git commit
```

```
git push origin develop
```

```
# delete the release branch for the previous version
# we keep only one release branch, for the current version, to reuse it for hotfixes

# for example, if current version is 2.3
git branch -d release-branches/2.2
git push origin :release-branches/2.2
```

Tips

Many of the procedures above requires to change the version in the pom.xml files. To do it safely and easily, you can use the "versions" maven plugin like this:

```
mvn versions:set -DnewVersion=<the-version-we-need-to-set> -DgenerateBackupPoms=false
```