1 TextGridLab Release HOWTO

We maintain a set of maintenance releases for TextGridLab 2.0

WORK IN PROGRESS

2 Bug Fixing / Development

1. Everything that needs to be fixed for a 2.x release needs a proper focused issue report in ChiliProject.
2. Bug fixes should be applied to a topic branch (feature/issue) and be merged into the `develop` branch first, unless we are in a release phase (see below). Small issues can also be implemented first and
3. Each commit should be focused, it MUST mention a short summary about the commit, and it SHOULD mention a the relevant issue number (#12345) in its commit message:
    a. use `fixes #12345` or `resolves #12345` to automatically resolve the issue
    b. use `refs #12345` or `references #12345` to add a link between issue and commit in the issue tracker.
4. You should increase the relevant artifact versions accordingly
5. If your bug should be included in yet unreleased TextGridLab 2.n, include the relevant release in the bug reports *fix for* field

We use the *git flow* model (and the corresponding tools), or an approximation thereof: New development generally happens in the `develop` branch, when we create a new release (e.g., 2.1), we fork a branch `release/2.1` off the `develop` branch.

3 Preparing the Release Candidate

In preparation of the release candidate, it is best to work on the aggregate `textgrid-laboratory` project to deal with all components (= submodules) in parallel.

```
git clone --recursive -b develop ssh://git@projects.gwdg.de/dariah-de/tg/textgrid-laboratory.git lab-3.3
cd lab-3.3
git checkout develop; git submodule foreach "git checkout develop"
git flow init -d
git flow release start 3.3
git submodule foreach "git flow init -d; git flow release start 3.3; :"
$EDITOR .gitmodules # replace all branch=develop entries with branch=release/3.3
# e.g.: sed -i 's/branch = develop/branch = release\/3.3/g' .gitmodules
git commit -am "Started release branch for 3.3"
```

1. Clone the repository, and all its submodules, to a new folder called `lab-3.3`
2. change into that directory
3. checkout the `develop` branch everywhere
4. initialize the git-flow tooling for the top-level project
5. start the release branch, this effectively creates a new branch called `release/3.3` and checks it out
6. do the same steps for all submodules
7. register the branch names; this allows you to do `git submodule update --remote` later to get them to the tip of their branch
8. commit the changes

Now we should adjust the version numbers at the various places in the source code. Additionally, we should set the application name to "TextGridLab RC" to indicate it's a release candidate. There is a shell script in the textgridlab-modules project for that:

```
./set-version.sh 3.3.0 RC
cd base
git diff
# possibly some &amp;apos;&amp;apos; are replaced with '', also there may be other unnecessary xml reformating
# check the diff, undo unwanted things and commit
git commit -am "Updated version information to 3.3.0 RC"
```

Publish the release branch:

```
git flow release publish 3.3
git submodule foreach "git flow release publish 3.3
```

Now Jenkins should start building RC on https://ci.de.dariah.eu/jenkins/view/TextGridLab/ and and a lab-3.3rc dir (updatesite) should show up in https://ci.de.dariah.eu/p2/ . It is possible that the order of the builds is not fine on first run, and triggers are failing. Try to get this fixed by: re-running the job 'lab-dependencies' in Jenkins. Trigger the "Scan Multibranch Pipeline Now" button in Jenkins for builds without release-job. Fix the jenkins release build in future!

Once Jenkins is done the rc build pipeline should be in place.

4 Testing the Release Candidate

After the build with all included releases has finished, the RC should be tested

1. Manual test against the features
2. Functional test
3. Test updates from previous lab version by
    a. downloading a clean copy of the previous version of the lab
    b. adding the update site
    c. *Check for updates* or restart the lab
4. Call for user tests

5 Release

When everything is tested we need to finish stuff in git, prepare the update site, upload the product zips and update the download website.

5.1 Sourcecode preparation

1. in the `base` module, update the update site enabledness state in `info.textgrid.lab.feature.base/p2.inf` to the following settings:
    - stable: enabled
    - beta: enabled
    - nightly: disabled
    - prerelease: disabled

   Remember that each site has *two* entries.
2. Run the version script again: `./set-version.sh 3.3.0` to remove the 'Special Version' and generate the welcome screen look for the released version.
3. commit the changes, update the submodule in the `textgrid-laboratory` project.

5.2 Release in Git

The *git flow* workflow's *release finish* feature performs the following steps:

1. the release branch is merged into master
2. this is tagged
3. the master branch is merged into develop
4. the release branch is removed.

These steps have to be performed in every submodule, *then* the `textgrid-laboratory` module needs to be updated, *then* this needs to be released. If we're using an older working tree, we should make sure all branches are up-to-date:

```
git checkout develop; git pull
git checkout master; git pull
git checkout release/3.3
git submodule foreach "
  git checkout develop; git pull
  git checkout master; git pull
  git flow release finish -m 'TextGridLab 3.3' 3.3
  :
"
```

Now, we need to **make sure the nightly branch *does* enable the nightly update site**. To do so,

1. `cd` /base
2. `git checkout develop`
3. edit `info.textgrid.lab.feature.base/p2.inf` to enable the Nightly update site (two lines)
4. cd ..
5. ./set-version.sh 3.4.0 Nightly
6. cd base
7. check the changes, remove unwanted modifications e.g. 'meld .' or 'git diff'
8. git commit -am "Started 3.4.0 development cycle"
9. `git checkout master`

If everything above went cleanly, now push the submodules

```
git submodule foreach "git push origin master develop --tags; :"
```

Now commit the submodule updates in the `textgrid-laboratory` module, still in the `release/3.3` branch, perform the release there, and push the master branch:

```
git submodule foreach "git checkout master; :"
sed -i 's/branch = release\/3.3/branch = master/g' .gitmodules
git commit -am "released submodules"
git flow release finish -m 'TextGridLab 3.3' 3.3
git push origin master:master
```

The prerelease jenkins job should now build the actual release version off the master branches. Look into the dashboard, all release/3.3 branch builds should be gone, trigger a scan if not.

Wait for the final base build to finish and test it, and then go on to publish the released version.

5.3 Prepare the Update Site

- The update sites are maintained on textgridlab.org in `/var/www/nginx-root/textgridlab.org/updates/stable`. We have the following directory structure:
  - `stable/composite/2.0.5` etc. each contain an update site for the respective version *only*.
  - `stable/composite` is an composite update site of *all* these version-specific update sites.
  - `stable` is a (non-composite) mirror of `stable/composite`.

  To prepare this, there are a bunch of scripts linked to /usr/local/bin. Do the following, in this order, replacing 3.3.0 with the appropriate version:

  1. in `stable/composite`, run `generate-aggregate-repo https://ci.de.dariah.eu/p2/prerelease/base 3.3.0` – this will mirror the repository from the integration build to the directory `3.3.0`.
  2. in `stable/composite`, run `generate-composite-p2-repo "TextGridLab Stable" [23].*` – this will create the composite repository metadata for the 2.* and 3.* subdirectories in the current (composite) directory.
  3. in `stable`, run `generate-aggregate-repo $PWD/composite candidate` – this will create an aggregate (= faster) mirror of the composite stuff in the `candidate` directory.
  4. For each directory or file in `candidate`, remove their respective counterpart from `stable` and move the version from `candidate` in place.
  5. remove the (now empty) `candidate` subdirectory.
- Details for managing the directory structure and the `composite*.xml` files are at Update Site Management.

5.4 Prepare the Downloads

- The download site is on textgridlab.org at `/var/www/download`. Files are in subdirectories with the version number
- It is probably best to just mirror the original downloads from the automatic build

```
cd /var/www/nginx-root/textgridlab.org/download/
mkdir 3.3.0 && cd 3.3.0
wget https://ci.de.dariah.eu/jenkins/job/lab-base/job/master/lastSuccessfulBuild/artifact/*zip*/archive.zip
unzip -j archive.zip
rm archive.zip
generate-lab-index.py --full > index.html
generate-lab-index.py > fragment.html
```

5.5 Download Website

1. on textgridlab.org, cd into `/var/www/download/2.0.2` and run the generate-lab-index.py script
2. open the download page in the typo3 backend and
    a. paste the generate-lab-index.py output into the download table field in the hero block, replacing the existing content
    b. write a short release note in German and English into the corresponding field
    c. generate the changelog ~~in Jira:~~
        i. ~~go to the TextGrid project's Versions list~~
        ii. ~~click the version to release~~
        iii. ~~click the *Release Notes* link in the upper right~~
    d. paste the changelog into the corresponding field in typo3
    e. add last version to the list of archived versions

5.6 ChiliProject

6 Hotfix Versions

According to the git-flow model, we have regular development in `develop` and in `feature/xxx` branches which get merged into `develop`, release preparation branches called `release/<Version>`, and a stable `master` branch that features just the releases. Regular releases follow the following workflow:

> Develop in `develop` branch `release/2.1` off `develop` test & last bugfixes release: (merge `release/2.1` into `master` tag master as `2.1` merge master (or rather the tag) back into `develop` (for each module, and for the umbrella project `textgrid-laboratory`)

However, it might be neccessary to apply just one or two bug fixes to the current release version and to publish the fixed version rather fast. To do so, git-flow supports the notion of hotfixes. Hotfixes branch off the latest released version (= master branch), perform the fixes, and then get merged into master as well as develop. Here it is explained for a case with a slight modification in the `base` module (I forgot to disable the nightly update site for the 2.1 release):

```
git hotfix start 2.1.1
( cd base; git hotfix start 2.1.1 )
./set-version.sh 2.1.1      # updates the base module
cd base
git commit -a -m "Bumped version number to 2.1.1"
# fix the bug
git commit -a               # useful bugfix message
```

Now you shoud cd up to the umbrella project, run `./build-locally` to compile the fixed version, and test (the products can be found in subdirectories of `base/base-repository/target/products` after the build). When everything is fine, you can release the hotfix, first in the submodule, then (commit the modified submodule first!) in the umbrella project.

```
cd base
git flow hotfix finish 2.1.1
cd ..
git commit -a          # commit the updated submodule
git flow hotfix finish 2.1.1
```

Now you need to push the changes:

```
git submodule foreach "git push --all; git push --tags"
git push --all
git push --tags
```

The prerelease build will build your hotfix. Afterwards, upload the hotfix release and update the update sites as described above.

7 TODOs

- [ ] What to do with the parent pom in the different steps of the release / development cycle? it contains the repo locations, and should possibly also have mathching releases and snapshots.