



Release Management

AID 107

ADITO Academy

Version 1.3 | 21.03.2022



Index

1. Purpose of this document	3
2. Terminology	4
3. Overview	5
4. Branching and associated proceedings	6
4.1. [MAINTAINER] Deriving the "dev" branch	7
4.2. [DEVELOPER] Deriving Working branches and Hotfix branches	8
4.3. [DEVELOPER] Daily Merge und creating Merge Requests	8
4.3.1. Minimize conflicts when merging	12
4.4. [MAINTAINER] Handling Merge Requests	13
4.5. [MAINTAINER] Weekly Merge and Release	15
5. Additional tools	16
5.1. Naming convention	16
5.2. Expandability of the branching model	16
5.3. Dailies	16
5.4. Housekeeping in the ADITO Designer	16
5.5. Customer's Developments	16
5.6. Tagging & Deployment	17
6. Role QA in release management	17



This document is subject to copyright protection. Therefore all contents may only be used, saved or duplicated for designated purposes such as for ADITO workshops or ADITO projects. It is mandatory to consult ADITO first before changing, publishing or passing on contents to a third party, as well as any other possible purposes.

Version	Changes
1.3	Renormalize Newlines
1.2	Typo fix
1.1	QA Additions
1.0	Release version



1. Purpose of this document

This document describes ADITO's release management process. In addition to the strictly defined branching model, this process also includes the execution of code reviews based on merge requests.

The goal of the release management process is to increase code quality.

This document explains the tasks that the roles "Maintainer" and "Developer" are assigned to during the process. Further information on the Git guidelines and the use of Git in the ADITO Designer can be found in document AID 089. To understand this document fully, good knowledge about AID 089 is required.



2. Terminology

- **Developer:**
edits tickets on specifically created branches.
- **Maintainer:**
is responsible for maintenance and management of the Git repositories. He provides particular branches for the corresponding ADITO systems.
- **Branch:**
is a fork of a project for independent development.
- **Source branch:**
is in subject to a merge request always the branch, **from** which the code is being transferred.
- **Target branch:**
is in subject to a merge request always the branch, **in** which the code is being transferred.
- **Child branch:**
is a branch, which is derived from another branch.
- **Parent branch:**
is a branch, from which a child branch was derived.
- **Master branch:**
is the main branch of a Git repository - it includes the code, which is provided on the productive system.
- **Hotfix branch:**
is being derived from the master branch in case of an error in order to fix it.
- **Protected Branch:**
is a branch, which code stand can only be altered with a merge request. Classic push commands are not allowed.
- **Daily Merge:**
includes the guide to daily merge the parent branch into the child branch

3. Overview

The following figure gives you an overview of the release management process.

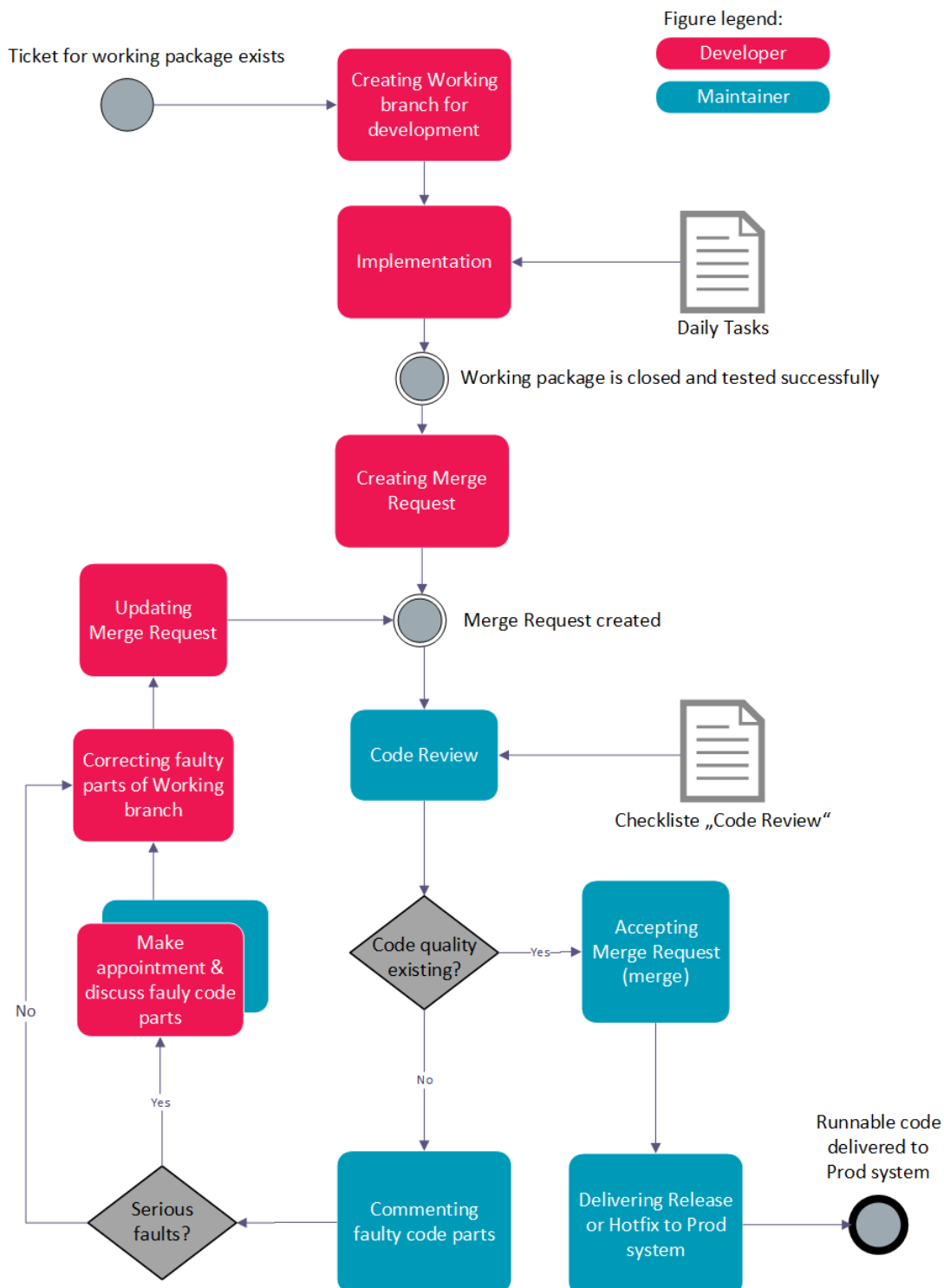


Figure 1. The release management process

4. Branching and associated proceedings

In the following, the branches and procedures shown in Figure 2, which are necessary for successful quality improvement through a structured release management process, are explained.

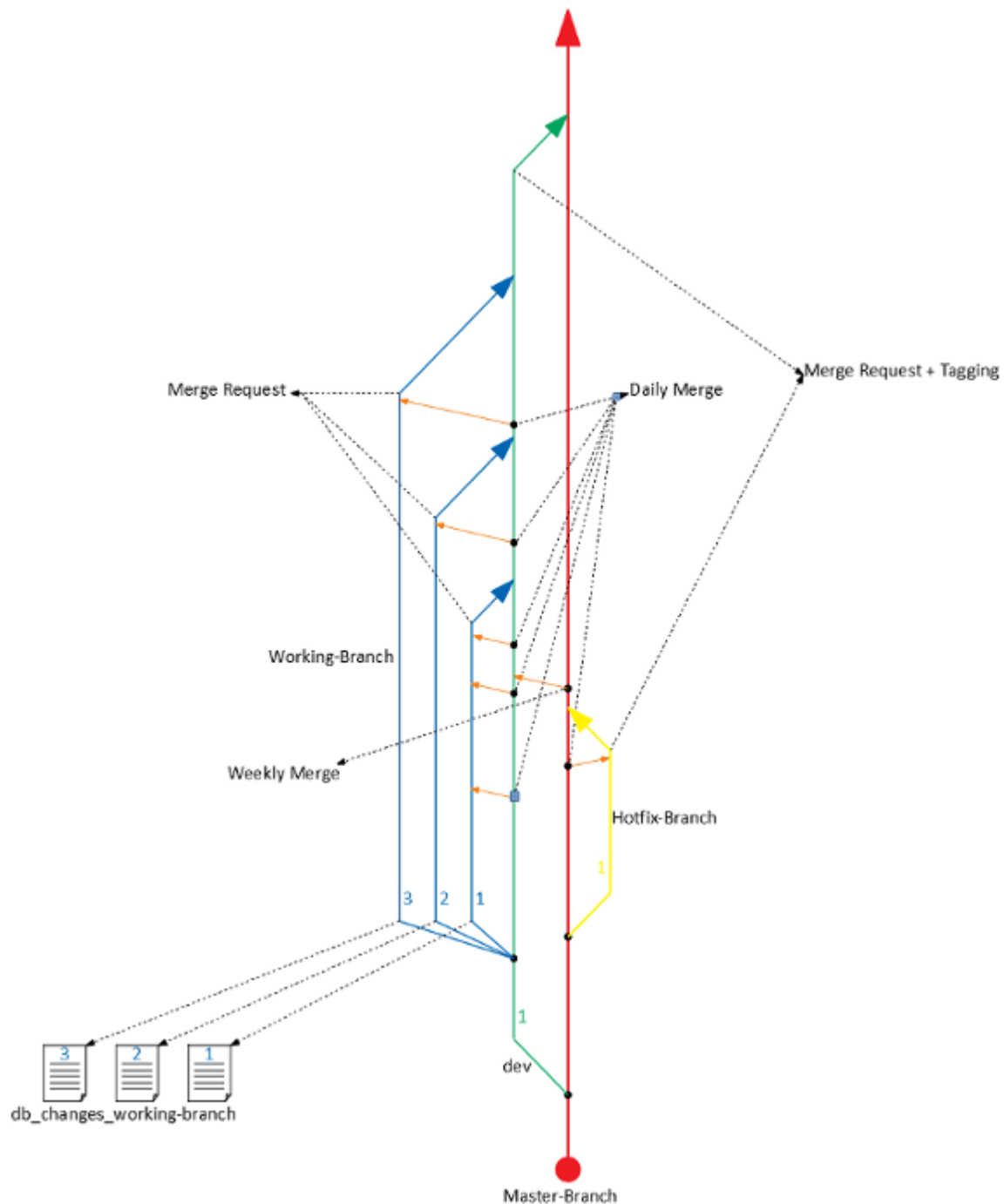


Figure 2. Branches of the Git repository

4.1. [MAINTAINER] Deriving the "dev" branch

After the Git repository was created, the Maintainer is responsible for making sure a development branch is being created. For that he derived the "dev" branch from the master branch. Both branches are protected.



Figure 3. Creation of the dev branch

To the productive ADITO system the master branch is being deployed, to the test system the dev branch is being deployed.

Practical example:

After the dev branch was created, it must be marked as protected in the Git repository of the respective project via Settings > Repository > Protected Branches (see Figure 4). Those settings must be checked or set for the master branch as well.

Protect a branch

Branch:

dev

Wildcards such as `*-stable` or `production/*` are supported

Allowed to merge:

Maintainers

Allowed to push:

Maintainers

Protect

Figure 4. "Protected" branch

4.2. [DEVELOPER] Deriving Working branches and Hotfix branches

The Developer creates a new Working branch for each work package (User-Story, Epic, ect.) within reason.

In those cases the parent branch of every Working branch is the dev branch.

The Developer furthermore creates a DB-changes file for each Working branch in a folder (e.g., Sprint 1 or Release 2.0), in which the database changes of every Working branch are documented.

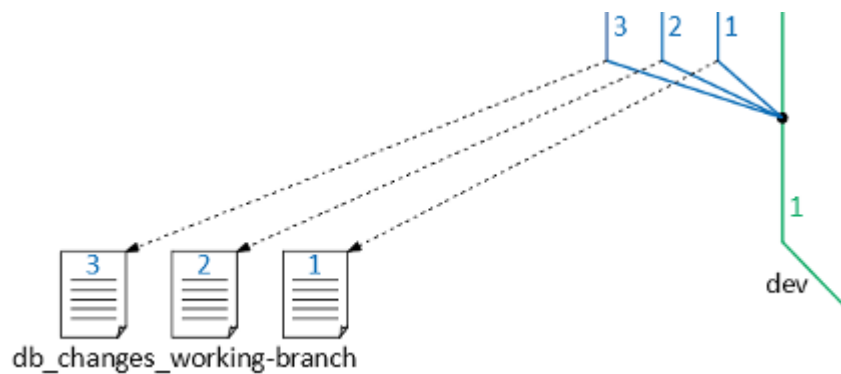


Figure 5. Creation of the Working branches incl. DB-changes files

For the handling of errors in the productive ADITO system, the Developer creates a new branch as well: a so-called Hotfix branch. They are based on the master branch.

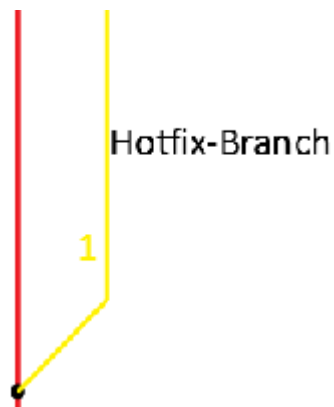


Figure 6. Hotfix branch

4.3. [DEVELOPER] Daily Merge und creating Merge Requests

After the developer has created his working or hotfix branch, he is required to merge the corresponding parent branch into his branch daily to keep it up to date.

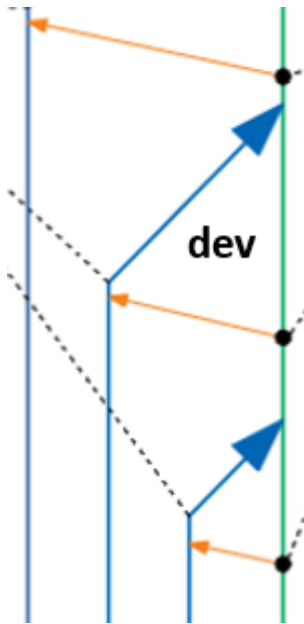


Figure 7. Daily Merge & Merge Requests

Now, when the developer has completed a user story, an epic, or a hotfix and has reached the end of that work package, he creates a merge request to insert the by him completed and successfully tested code into the parent branch.

When creating a merge request the developer has to ensure the following:

- The work package is completed 100%
- All tests were successful
- Source and target branch are selected correctly
- Mark for deleting the source branch after accepted merge request is set

Practical example:

The developer performs the daily merge by pulling the parent branch (e.g. dev) into the corresponding child branch (e.g. dev_implementation_bs).

After the developer has committed and pushed the last changes in his branch, he creates a Merge Request on GitLab (see Figure 8 and following).

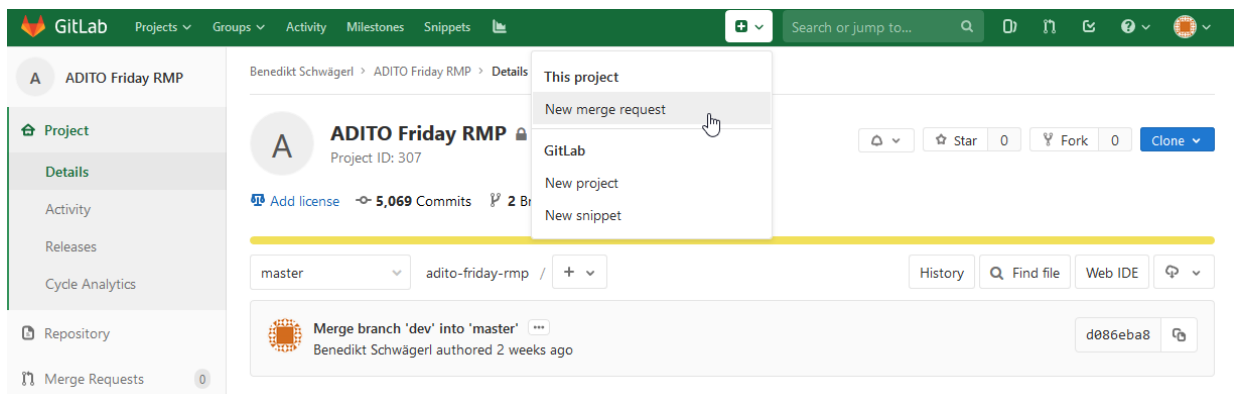


Figure 8. Create Merge Request (1)

At first the respective source and target branch need to be selected.

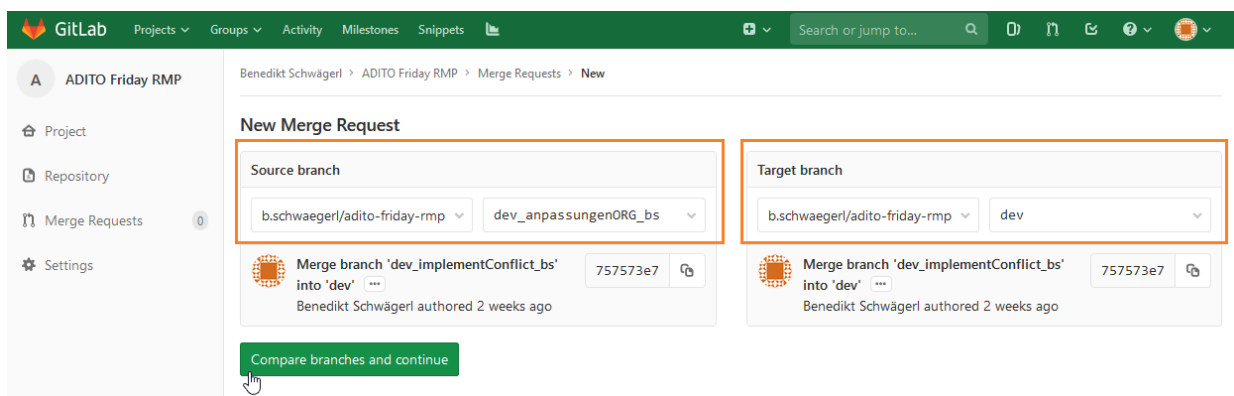


Figure 9. Create Merge Request (2)

As an alternative Gitlab gives the Developer the choice, to create a merge request for his last push.

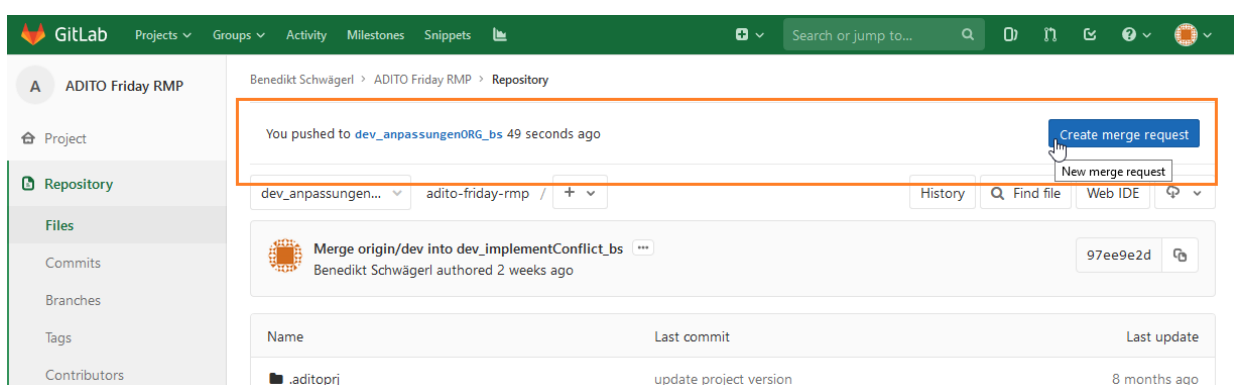


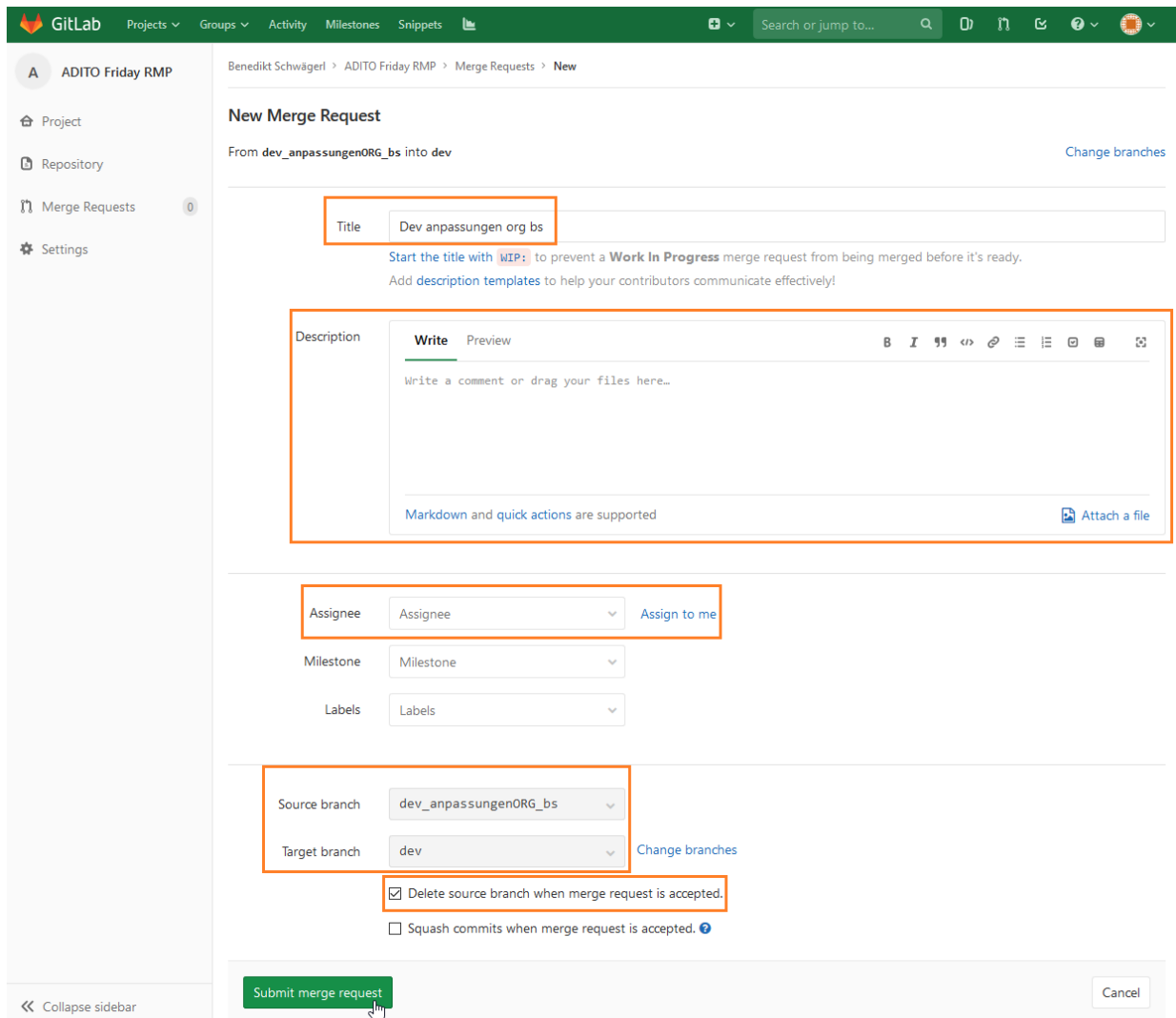
Figure 10. Create Merge Request (3)

Both the button "Compare branches and continue" from Figure 9 and the button "Create merge request" from Figure 10, take the creator to the overview page where he can see a summary of the merge request (see Figure 11).

Here a title can be assigned, a description can be maintained and/or a person responsible for the

merge request can be selected. With click on "Submit merge request" the creator releases the merge request for processing and thus assures the following points:

- The work package is completed 100%
- All tests were successful
- Source and target branch are selected correctly
- Mark for deleting the source branch after accepted merge request is set



GitLab Projects Groups Activity Milestones Snippets

Benedikt Schwägerl > ADITO Friday RMP > Merge Requests > New

New Merge Request

From `dev_anpassungenORG_bs` into `dev` [Change branches](#)

Title `Dev anpassungen org bs`

Start the title with `WIP:` to prevent a **Work In Progress** merge request from being merged before it's ready. Add [description templates](#) to help your contributors communicate effectively!

Description Write Preview

Write a comment or drag your files here...

Markdown and quick actions are supported [Attach a file](#)

Assignee `Assignee` [Assign to me](#)

Milestone `Milestone`

Labels `Labels`

Source branch `dev_anpassungenORG_bs`

Target branch `dev` [Change branches](#)

☒ Delete source branch when merge request is accepted.

☐ Squash commits when merge request is accepted. [?](#)

[Submit merge request](#) [Cancel](#)

Figure 11. Create Merge Request (4)

Within or after creation of a merge request, it's always possible that errors occur and the merge request can therefore no longer be executed by the Maintainer (see Figure 12).

If that's the case, the Developer is requested to merge the target/parent branch into his source branch, to fix the conflicts locally. After that he needs to update the merge request with a commit and push command.

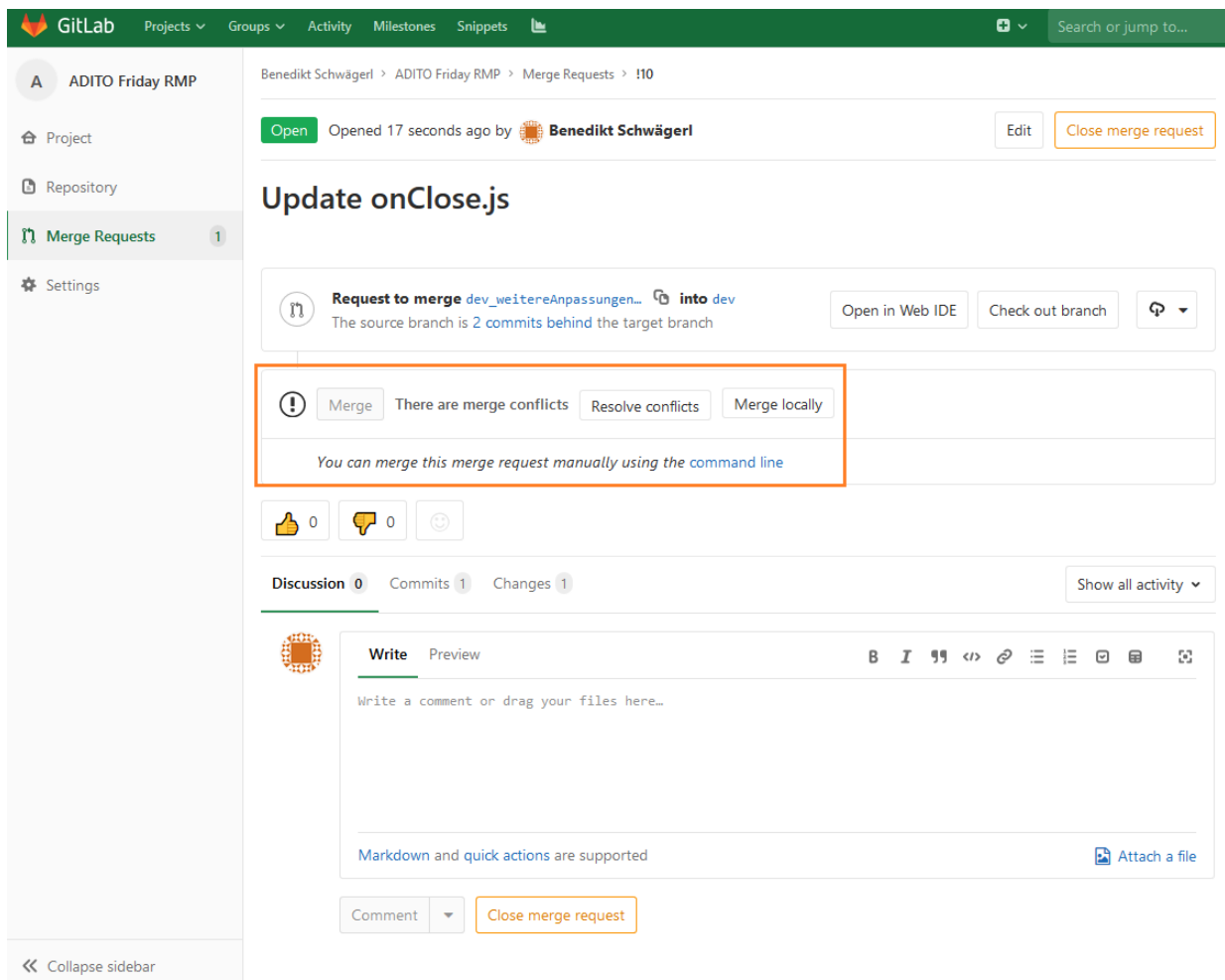


Figure 12. Conflicts in a Merge Request

4.3.1. Minimize conflicts when merging

It is recommended to first raise the two branches to the same version before initiating the merge. This results in you not having to worry about the version number changes of the entities/contexts/views etc.

You should also use the feature "Renormalize Newlines" on both the Master Branch and Working Branch. If anything did change with the execution of this feature you have to make a commit again.

4.4. [MAINTAINER] Handling Merge Requests

If a developer's merge request is free of conflicts, the Maintainer can process it. The Maintainer must observe the code review rules (see practical example). If a code passes the review, it is merged into the specified target branch.

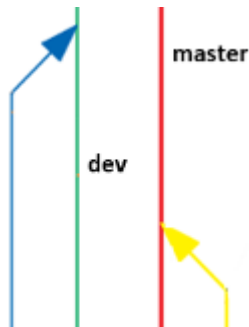


Figure 13. Handling Merge Requests

The Maintainer needs to make sure that the source branch is being deleted after a successful merge. Deleting working and hotfix branches in a Git repository is essential to maintain clarity.

Practical example

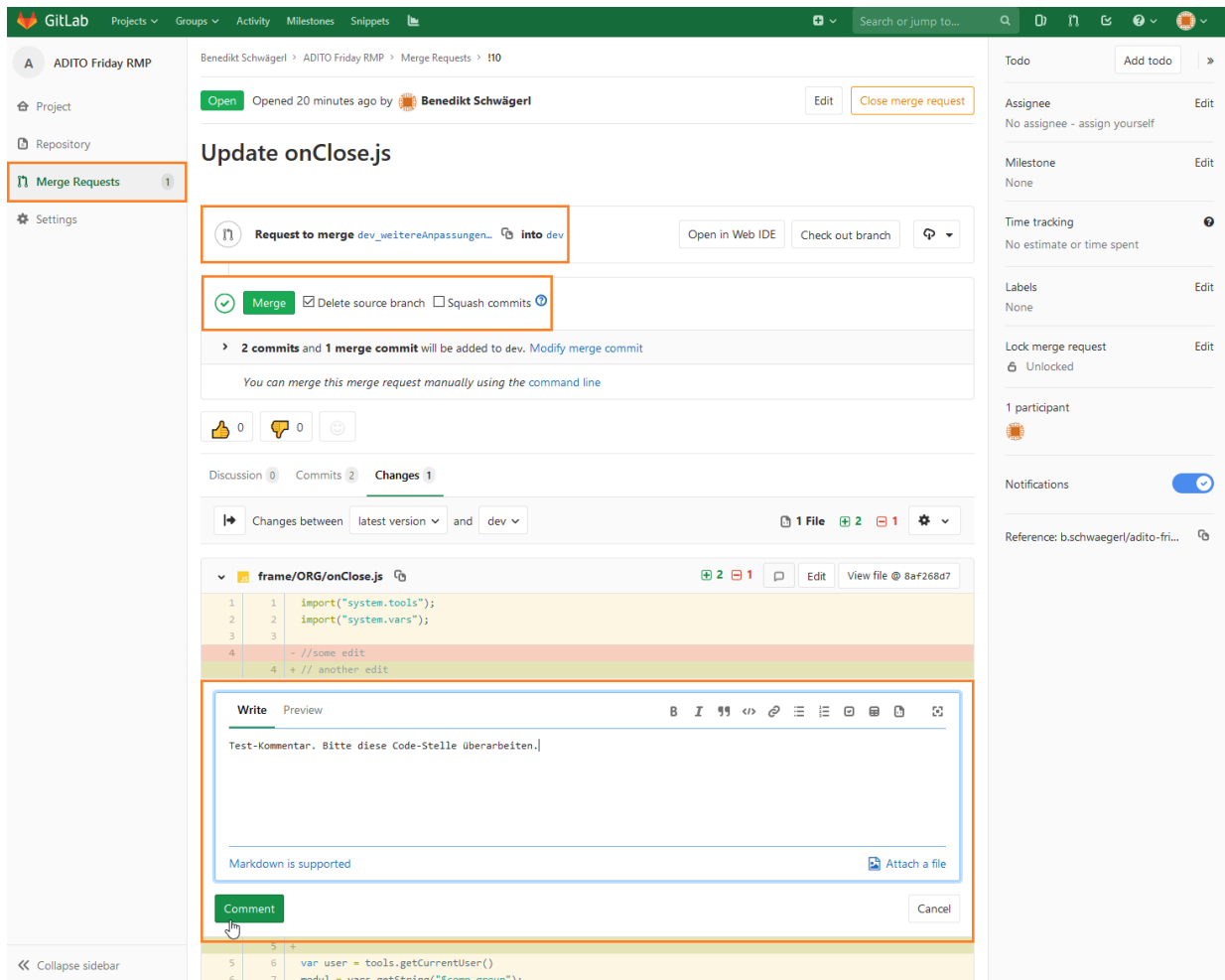


Figure 14. Detail page of a merge requests

The tab Merge Requests contains all merge requests of a project. After selecting a merge request, the detailed view appears (see Figure 14). The Maintainer must now check the following points

1. Has the **correct target branch** been selected? (that's only possible with correct usage of the naming conventions of section 5.1.)
2. Has the **tick for deleting** the source branch been selected?
3. Complies the code with the requested **quality**?
 - Does it meet the *Coding-Guidelines*?
 - Can the *checklist "Code-Review"* in ADITO internally be processed without gaps?



With a merge request, only the entire request can be accepted or rejected. A partial acceptance is not possible.

[*ACCEPTED*] If all requirements have been met, the merge request can be accepted.

-> By clicking on the "Merge" button, the new code is now available on the target branch and the source branch is deleted from the repository.

[*REJECTED*] If the Maintainer finds quality issues in the code, he uses the comment function to inform the Developer about the found issues. The Developer then improves the requested parts of the code. After that he commits and pushes the changes to the source branch of the merge request, which automatically updates the merge request.

4.5. [MAINTAINER] Weekly Merge and Release

In order to ensure that changes on the parent branches are also applied to the child branches, the Maintainer is required to merge the parent branch into the child branch at least once a week when changes are made. In the opposite direction, the Maintainer is responsible for transferring the code from the child to the parent branch in the process of a release.

If the master branch is updated in the latter case, a new tag must be specified (see chapter [Tagging & Deployment](#)).



Figure 15. Weekly Merge & Release

Practical example:

Since the dev branch is protected, the Maintainer executes the weekly merge analog to section 4.3. He submits the merge request (source branch: master; target branch: dev) which he then also merges by himself. The other way round it's the same: He submits a merge request (source branch: dev; target branch: master) and merges by himself.

5. Additional tools

In addition to the branching model the release management process includes other important tools, which contribute to quality improvement.

5.1. Naming convention

The naming convention for Working and Hotfix branches needs to have the following structure:

„Parent branch“ _ „theme“ _ „Employee Initials“

Example for Hotfix branch: master_hotfix_performanceOrg_bs

Example for Working branch: dev_sapImport_bs

5.2. Expandability of the branching model

In some cases it can be necessary to expand the branching model. Here's an example for general clarification:

In a project a partial project was commissioned, which had to go live before the official Go-Live date. In that case the branching model had to be modified: A second development branch was needed. Now a separate development state was ensured, that could go live on a different date.

5.3. Dailies

A daily is a 15 minutes long, daily meeting in which Developers discuss their daily tasks in the release management process, among other things.

For agile projects, where dailies take place anyway, an extra daily for the points concerning the release management process is not needed. They can be included in the "main" daily.

5.4. Housekeeping in the ADITO Designer

By deleting the branches after an accepted Merge Request, the clarity in the "origin" repository is guaranteed.

To prevent the Git repository of the respective Developer from ending up in chaos, the Developer can use a Fetch command to remove "remote" branches that no longer exist. Branches of the Developer's "local" repository are removed via RMB → "delete Branch" (see AID 089 section 3.5).

5.5. Customer's Developments

In case a customer's Developer develops code in a project within the implementation phase, their code needs to be subjected to the release management process as well.

The customer must pay for the resulting expenses. Such a service can be stated as quality assurance.

After the implementation phase, responsibility for the code's quality is transferred to the customer's developer team.

5.6. Tagging & Deployment

A tag must be specified for every accepted merge request that changes the code state of the master branch. The following tagging model is **for guidance only**.



If a development branch was merged in the master, the first two digits of the tag would be changed. If a hotfix branch was merged successfully, the last digit would be raised.

In addition to tagging, the code states of course also need to be **deployed**. The release management process intends the deployment as follows:

- Master branch exclusively on Prod system
- Dev branch on systems like "Test" or "Training"

6. Role QA in release management

A QA-Engineer (Quality Assurance Engineer) has to view merge requests and see if code reviews are executed in ordinary method. The code review is a part of the complete QA-Process and has to be viewed and secured by the QA-Engineer. Normally code reviews will be held by the teams developers.

If the maintainer finds weak spots due to insufficient quality of the code, the QA-Engineer can, depending on the topic, call a meeting to clarify the issues.