# How to contribute to eZ Publish using GIT

By **André Rømcke**
Collaboration with **Nicolas Pastorino, Robin Muilwijk, Ole Marius Smestad**
Reviews by **Gunnstein Lye, Damien Pobel, Paul Borgermans, Ricardo Goulart & João Pingo**

## Index

## 1 Goal description

At the end of this tutorial, you should be able to contribute to eZ Publish Community Project, eZ Publish's kernel developed by both eZ and the eZ Community. You will learn how to use git and github in the scope of this participation, and be given the bunch of best-practices to follow for smooth collaboration.

## 2 Introduction

With the change to GIT/ Github earlier this year, the 4.4 release and the future change to how we manage our eZ Publish Community Project (CP), now is a good time to learn how to get eZ Publish from GIT and contribute on solving issues and innovating.

Collaborating on eZ Publish development is not really hard or too time-consuming if we take advantage of the tools that we have at hand, and GIT + Github really shines here. First of all it's a way for you to make sure the bug is fixed more quickly while at the same time making sure you are properly attributed for your work.

The Community Project symbolizes eZ's core value of openness in making contribution to eZ Publish kernel happen. Time to roll your sleeves up and make eZ Publish a bit of your own.

## 3 Pre-requisites and target population

This post is aimed at developers and technical users who want to be able to make changes to eZ Publish and share them with others, and in the end the change is bound to be integrated into the product.

If you're just here to find out how to clone / "checkout" eZ Publish from GIT, skip to "Cloning read only", and optionally "Installing GIT and setting up Github".

<u>Note</u>: *This document should be considered as a living document, parts of it can be moved to GitHub wiki at any point for easier maintenance and for updates as the community go forward. This is especially true for coding standards and commit conventions.*

## 4 The different ways of contributing

While waiting for the upcoming changes to how development is done (soon decided upon by the Community Project Governance ), here are your current options when it comes to contributing to eZ Publish Community Project :

- Reporter - Issue reporter, creates issues in issue tracker (http://issues.ez.no/ezpublish)
- Tester - Provide reduction test cases, reproduce and confirm issues
- Collaborator - Contribute code with unit tests and api doc on issues created by yourself or others. A patch could also just fix api doc, improve unit tests or other non functional stuff, in that case there doesn't need to be an issue for it.

As you can see, you don't necessarily need to be a developer to contribute to an open source project, and same goes for eZ Publish Community Project. The whole *"Reporting issues > Reproduce and confirm issue > Adding some more info on steps to reproduce > Finding some more information on what triggers the issue"* process, is of tremendous help and makes sure that issues are looked at more quickly. But if you are a developer, it will be looked at even more quickly if there is already a fix for it…read on !

## 5 How to collaborate

From a bird-eye view, there are two workflows for contributing code. The first one is similar to how it is done with SVN:

*1. Do changes > 2. Create patch > 3. Reset your checkout > 4. Upload somewhere*

And then iterate the whole process starting by applying the latest patch if there are more changes to be made.

But that is not really taking advantage of GIT and Github, so to do that we'll promote the Github workflow in this article:

*1. Fork > 2. Create topic branch > 3. Do changes + Commit > 4. Push*

| **Step** | **How many times do I do this ?** | **What is it exactly ?** |
|---|---|---|
| **1.** Fork | Once | Creating a copy of the main eZ Publish repository, logically linked with the original one. The GIT way. All it takes is one click. |
| **2.** Create topic branch | For every topic you address (bug-fix, feature) | Keeping your development efforts in a clean space in your GIT version of eZ Publish. Lets you work on other parts of eZ Publish if you want to, with no conflict. Branching is seamless with GIT. |
| **3.** Do changes + Commit | At least once per topic | Make changes, commit, and repeat if you want to do several independent changes. |
| **4.** Push | At least once per topic | Synchronizing your local GIT clone with your account on github. After you have pushed you can ask people for feedback on your commits and then re-iterate 2 and 3 until you are happy with the state of your branch, at that point you can create a "Pull Request" (see further below). |
| **5.** Emit a pull request | At least once per topic | Once satisfied with your bug-fixing or feature development job, sending your changes back to the main eZ Publish GIT repository, so that others can see them or elaborate on them. This happens on Github, and is equivalent to asking the 'ezsystems' user to merge in your changes. |

Locally you can iterate #3 several times before you push, keeping in-dependant changes separate from each other. After you have pushed you can ask people for feedback on your commits and then re iterate 2 and 3 until you are happy about the state of your branch, at that point you can create a “Pull Request” on Github asking the ‘ezsystems’ user to merge in your changes.

As you could notice we start using the GIT/Github jargon here. If you are not super acquainted with it, we recommend to read the appendix section.

## *CLA / coding standards*

Make sure you have signed the CLA (Contributor Licensing Agreement), we cannot accept code contributions without it:
http://share.ez.no/community-project/contributor-licensing-agreement-cla
We are considering easier ways for you to sign it, but for now, a scan of the signed, printed-out version, sent through email to community@ez.no is good.

Also make sure you follow our coding standard to save time on several unnecessary review rounds. For instance, always use 4 spaces instead of tabs for indenting. Here are details:

## *PHP*

For PHP you can follow the eZ Components / Apache Zeta Components coding standard :
http://incubator.apache.org/zetacomponents/community/implementation.html
With some notable differences: class name prefix is ‘ezp’ for new classes, use ‘eZDebug’ for error, warnings, notices, strict errors and debug statements instead of ‘trigger_error’ and ‘Component configuration’ / ‘Directory structure’ does not really apply.
Also for exceptions, use with caution, a document / wiki is in the works to define exception hierarchy and it’s going to inherit from PHP/SPL exceptions. Make sure your code works on PHP 5.2 and up, and does not use deprecated functionality.

## *Templates*

Make sure your html validates as html5 and xhtml traditional (in that order when they contradict each other), in the future we will most likely aim for xhtml5 (html5 using XML rules for simplified parsing). The template code itself should not use deprecated functionality and be properly indented.

## *CSS*

Follow CSS 2.1 spec and only use CSS 3.0 for visual enhancements so there are no loss of functionality or major display regressions on older clients, including IE6.

## JavaScript

Follow the official EcmaScript (ES) standard, do feature detection and not browser detection to fallback when browsers do not support [ES] features, but preferably use available functionality in jQuery or YUI which is already doing these things for you. Be unobtrusive (separate html and js code and enhance the html using general CSS selectors progressively meaning basic functionality is still maintained if it fails), and for jQuery / YUI3, use ezjscore.

## Browser testing

We more or less follow Yahoos A-Grade browser list, with one exception, we don't spend time on getting out interfaces to look the same in older browsers, it needs to be functional, but not pixel perfect.
As a simple rule test in latest stable version of FireFox, Chrome/Safari and IE in addition to IE 6.0. When IE 9.0 is stable start testing in that as well, but as it behaves closer to how upcoming FireFox / Web-kit versions do, it's not as important to test as 6.0 and 8.0 (for now).
As for IE 6.0, attention to who your customers are should be taken, but in general we plan to stop testing in IE 6.0 for future products when it dips bellow 10% browser share, and for current products when it's well bellow 5% if current browser share trends continue at the same pace. Considerations for testing on mobile browsers will be defined later, as the mobile browser market is much more fragmented, it will require a much larger discussion.

# 6 Installing GIT and setting-up Github

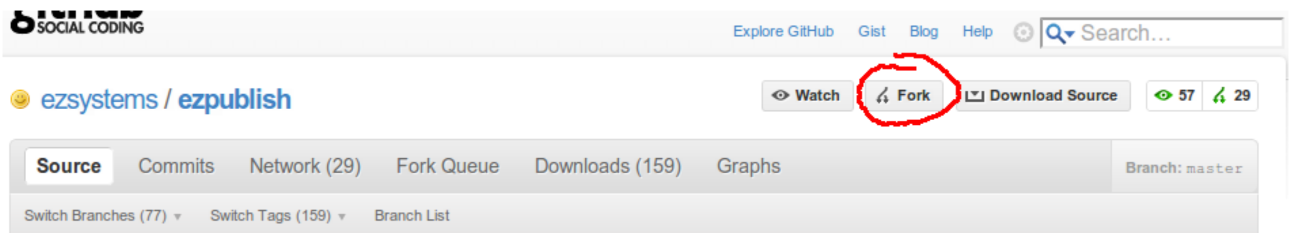
This post will demonstrate GIT using shell (command line). This is mainly because most of the visual GUI's are not good enough yet at the time of writing (end of 2010, beginning of 2011), and they act fairly differently.

Documentation on installing GIT for any platform (even Windows!) and setting up Github is best explained on Githubs excellent help pages: http://help.github.com/
There are also some useful links and comments on the earlier share.ez.no blog post: "eZ running on GIT"

# 7 "Forking" eZ Publish

If you would like to make some changes to eZ Publish like proposing a fix for a issue directly on Github, click on the fork button. This will create a fork of eZ Publish on your own account page (yes, this implies you created your github account beforehand. This is free).

*<u>Note</u>: the fork will not update itself with changes done in the original repository, this is done by pulling in changes and pushing it to your fork ( how to do this is explained a bit below).*



# 8 Cloning eZ Publish

Using command line, go to the place you want to checkout eZ Publish in, for instance the localhost / www folder that your web server points to, to be able to execute eZ Publish directly for testing.
At this point there are two options:

## *Cloning read-only*

Only do a clone of eZ Publish to track development, get source, potentially create patches and test (optionally specifying target <folder_name>):

```
$ git clone git://github.com/ezsystems/ezpublish.git [<folder_name>]
$ cd [ezpublish|<folder_name>]
```

After that keeping your checkout up-to-date is a matter of:

```
$ git pull
```

If you want to checkout other branches than master, try to use the same name as remote branch to avoid issues when you later try to push changes somewhere.
eg. when you start following a branch (track):

```
$ git checkout -b stable-4.2 origin/stable-4.2
```

eg. When later checking out a local branch:

```
$ git checkout stable-4.2
```

## *Cloning with write access on Fork*

If you are an external contributor that would like to make changes to eZ Publish and share it on github using the fork you created in "Forking eZ Publish" (optionally specifying target <folder_name>, and <user> as your github account alias):

```
$ git clone git@github.com:<user>/ezpublish.git [<folder_name>]
$ cd [ezpublish|<folder_name>]
```

Now we need to add ezsystems as an additional read only remote location, in the following code examples we call it 'upstream':

```
$ git remote add upstream git://github.com/ezsystems/ezpublish.git
$ git fetch upstream
```

Optionally we make master branch track upstream, making it easier to pull in changes:

```
$ git config branch.master.remote upstream
```

After that keeping your checkout up to date is a matter of:

```
$ git pull
```

If you want to checkout other branches than master, try to use the same name as remote branch to avoid issues when you later try to push changes.
eg. when you start following a branch (track):

```
$ git checkout -b stable-4.2 upstream/stable-4.2
```

eg. When later checking out a local branch:

```
$ git checkout stable-4.2
```

If you don't want to make any changes to the code, you're done! Just point your browser to http://localhost/[ezpublish|<folder_name>]/ and you should get the setup wizard straight away given that you cloned eZ Publish lies into your www dir. No worries though, git folders are self contained so you can move it around as you wish (the whole cloned folder, not it's sub-folders).

# 9 Working with topic branches

A topic branch is a branch specifically created for an issue or a collection of related issues, or even a new feature. From a SVN background this might sound crazy. But with GIT branching is cheap and makes the whole process simpler, retains history and allows online reviews on github. Here is how to create the topic-branch:

## *Checkout*

1. If you haven't created topic branch yet:
   `$ git checkout -b <topic-branch> upstream/master`

2. If you already have the branch, but only remotely in your fork:
   `$ git checkout -b <topic-branch> origin/<topic-branch>`

   And specify that it should track upstream/master:
   `$ git config branch.<topic-branch>.remote upstream/master`

3. If you have the branch locally already (check with "`$ git branch`"):
   `$ git checkout <topic-branch>`

Topic branch name should be something that identifies issue, so one or two keywords and issue number separated by hyphen, eg: *rss-nbsp-016953 for "#016953: &nbsp; breaks RSS/ATOM feeds"* issue, or if your going to address a collection of related issues something like eg: *rss-improvments*. If you are starting a feature, be explanatory in the name too, yet don't write your life in it.

## Make changes

Make all independent changes one at a time, code should be workable, clean and testable before you go ahead and push. Meaning no debug code or var_dump calls.
Bug fixing example, full work-flow :

1. Pull in changes using "`$ git pull`"
2. First add a unit test that triggers the issue and fails
    1. commit (see below)
3. Fix the issue
    1. commit (see below)
4. Switch from doxygen to phpdoc on the php functions you touch
    1. commit (see below)
5. Push (see below)

## Commit

If you have added new files, use "`$ git add <file>`" to stage them for commit.
Then the most common ways to do commits (#B is preferred):

A) Only summary:
```
$ git commit -am "Fixed #<issue_id>: <issue_title>"
```

B) With a message:
Useful when you do several commits to same issue / topic and want to add a note on the changes you do in-between. Also, and most importantly, write why you do the change, especially if there is no issue that explains the why! Start with the summary as above and add an empty line before the body of the commit message.
*Remember that lines starting with # are treated as comments and will not be part of the commit message, and please always use utf-8. For more info, see the git user guide*
*For reviewer / issue-creator / patch-provider credit and attribution follow the svn convention*
```
$ git commit -a
```

Git will open your default editor to edit commit message, change it to something like this and save + close (example text):

```
Fixed #<issue_id>: <issue_title>

Extended signature of function X to allow caller to specify the user
instance instead of only using current logged in user.

Review by: Eagle Eyes
          ar
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
(...)
```

*<u>Note</u>: "Fixed" is used if issue is a bug, if it is an enhancement, use "Implemented"!*

If you are not confident in your fix then it is better to create a patch and upload it to the issue tracker under the related issue (cf "Working with GIT patches" a bit below). When uploading your patch, feel free to engage a conversation around it (adding a comment to your patch upload, simply), explaining why you are not sure about it, what you would like to clarify, etc.

## Push

When you feel confident that you want to share your work with the world, you'll want to push. But since our topic branch tracks another branch (probably upstream/master) we should always specify target using:

```
$ git push origin <topic-branch>
```

When this is done, git will respond with something like "Your branch is ahead of 'upstream/master' by 1 commit." basically giving you an idea on how many differences there are between your topic branch and the branch you track (master). This is also shown when you use "`$ git status`".
You may also see statuses like "Your branch .. is ahead .. with 1 commits and behind by 50 commits" meaning you have 1 commit in your topic branch and 50 new commits in master since you last pulled in changes.

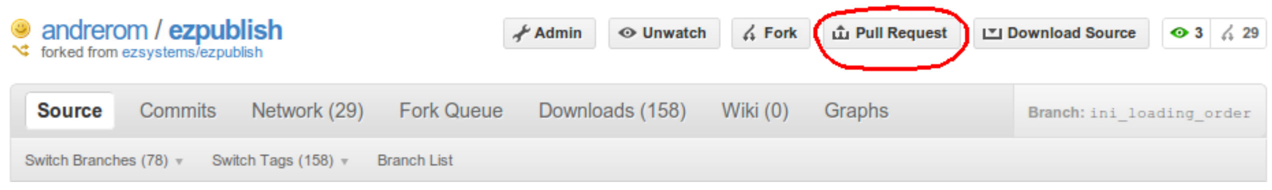
## Add comment on issue

Now to make anyone following the issue aware of the fix, add a comment like this on issue in issue tracker with correct link to the commit :

```
Fixed in <user> fork:
master(4.5.0alpha3) http://github.com/<user>/ezpublish/commit/<commit_sha>
```

## Pull Request

When you are confident that your contribution is ready for inclusion in eZ Publish, then all you need to do is click on "Pull request" in the Github GUI **on the specific topic branch** as selected using the "Switch Branches" drop-down.



### *Tagging a "finished" branch (optional)*

When work on a branch has concluded, it is a good idea to convert it into a tag, to avoid cluttering up your branch list with inactive branches, but to still keep track of the changes and commits inside that branch. This can be done by the following command, when being inside the working directory of your checked out branch:

```
$ git tag -am "Message describing the tag" <tagname>
```

Leaving out the last parameter will tell git to point the tag to the latest commit on that branch, which is what you want in this case. Converting branches to tags, is particularly useful when you are dealing with a form of old, inactive release branches, where you do want to keep it, but are no longer working actively on it.
If a branch has been merged back to its upstream source, been tagged or both it can be deleted. You accomplish this by:

```
$ git -d <branch name>
```

Note the lower case 'd', this is a safety precaution, git will warn you and stop, if it detects that you are trying to delete a branch which has not yet been merged, or has another ref tracking it (such as a tag). This is for your benefit, and to make sure that you don't lose any data.
Following these suggestions you should be able to keep mind and repository clean, but yet keep all history of your work as you please.

## 10 Some notes on *rebase*

When you work towards one branch it is perfectly fine to use "`$ git pull --rebase`" before you *push* to avoid unneeded merges. But not when you work towards a public topic branch that tracks another branch (master in the cases described above), as it will rewrite history of your topic branch causing issues for everyone else who uses it as well as losing relevant history when the topic branch is merged back to master.
So only use it optionally on topic branches before you push the first time, otherwise it'll do more damage than good.

## 11 Merging an 'alien' topic branch

In the following example we merge code from one eZ Publish fork, to our own.
The same approach is used when eZ Engineers merge an external topic branch into eZ Publish, ( if Github's GUI functionality of directly accepting a pull request is not used ).
If not already done, we need to add remote to the fork we want to take a look at.

```
$ git remote add <user> git://github.com/<user>/ezpublish.git
$ git fetch <user>
```

Assuming you have already checked out the local branch you would like to merge the changes into using "`$ git checkout <branch>`" we can now merge the topic branch using:

```
$ git merge --no-ff <user>/<topic-branch>
```

## 12 Working with GIT patches

There are predominantly two main paths for creating patches in our workflow, not including regular diffs. The first way, is to use git's *format-patch* command. This command will take the commits you specify, and save them out as mbox compatible files. These files can then be passed on to another developer, or attached the issue tracker. The good thing about a git patch is that it retains author information, date and commit message from the original contributor, allowing for a correct attribution of work (git can distinguish between the author of a patch and its committer).  This command has a lot of options, please see the git help, for more details here.

```
$ git format-patch <sha1(commit-hash)>
```

So if you have received a patch, reviewed it, and would like to add it the repository, you can use the *am* command. It takes a patch produced with f*ormat-patch* and commits it to the repository, with all the information intact, provided the patch still cleanly applies.

```
$ git am <patch file> [<patch file>] [<patch file>]
```

## 13 Conclusion

You have hopefully now a much better overview of how you can take advantage of git and github, both for eZ Publish contributions as well as for your own open source code development.
It should once again be noted that this is considered a living document, so things like code standards and commit conventions might move out to a wiki pages on GitHub later where it can more easily be maintained and adjusted as we move on.

## 14 Resources

If you have questions when it comes to git, first place to look should probably be Github's FAQ, as it explains things straight forward and also covers Github-related questions. But for a GIT specific article, be sure to check out one from A list Apart on the subject which also mentions the most important additional GIT resources like Git ready, Pro Git and the official documentation. If you are a cheat-sheet addict, check this out : http://help.github.com/git-cheat-sheets.

If there are any eZ Publish + GIT specific questions, feel free to post below!

## 15 About the authors

Author

- Andre Rømcke - http://share.ez.no/community/profile/9757

Collaborators :

- Ole Marius Smestad - http://share.ez.no/community/profile/9710
- Nicolas Pastorino - http://share.ez.no/community/profile/9804
- Robin Muilwijk - http://share.ez.no/community/profile/10838

## 16 License choice

## 17 Appendix : GIT/Github jargon

### *Git terms:*

- Clone: Clones a whole repository, giving you access to all it's branches and tags locally in one folder (but you'll need to use checkout to select one at a time).
- Commit: Like in svn, but only done locally. Allows you to commit offline and do several small commits to simplify reviews
- Push: Push your local commits to a remote repository
- Pull: Like svn up, pulls in changes from a remote repository
- origin: The origin remote server, this refers to the origin remote GIT server you cloned a repository from, and has nothing to do with forks.
- upstream: And open source term for a third party project your code relies on, in GIT often used for the original project, the one you forked in case of Github. aka ezystems/ezpublish when dealing with eZ Publish.

### *Github terms:*

- Fork: Like a copy of a repository with some knowledge of original repository and compare / status features. To be able to easily share changes you do, as opposed to keeping your changes locally on custom branches or setting up your own GIT server that contains your changes.
- Pull request: A Github feature, makes it possible to notify the original repository committers about your branch and ask him to integrate  your work in (make sure you comply with CLA / coding standards before you do)