Contributing

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Definitions 1

2 Discussions 1

3 Support 1

4 Bug reports 1

5 Security reports 1

6 Feature requests 2

7 Documentation submissions 2

8 Examples submissions 2

9 Code submissions 2

9.1 Cloning . . . 2

9.2 Branching . . . 3

9.3 Source editing . . . 3

9.4 Committing . . . 3

9.5 Cleaning the commit history . . . 4

9.6 Submitting the pull request . . . 4

9.7 Updating the pull request . . . 5

9.8 Merging . . . 5

This document is a guide on how to best contribute to this project.

1 Definitions

SHOULD describes optional steps. **MUST** describes mandatory steps.

SHOULD NOT and **MUST NOT** describes pitfalls to avoid.

Your local copy refers to the copy of the repository that you have on your computer. *origin* refers to your fork of the project. *upstream* refers to the official repository for this project.

2 Discussions

For general discussion about this project, please open a ticket. Feedback is always welcome and may transform in tasks to improve the project, so having the discussion start there is a plus.

Alternatively you may try the Discord server or, if you need the discussion to stay private, you can send an email at contact@ninenines.eu.

3 Support

Free support is generally not available. The rule is that free support is only given if doing so benefits most users. In practice this means that free support will only be given if the issues are due to a fault in the project itself or its documentation.

Paid support is available for all price ranges. Please send an email to contact@ninenines.eu for more information.

4 Bug reports

You **SHOULD** open a ticket for every bug you encounter, regardless of the version you use. A ticket not only helps the project ensure that bugs are squashed, it also helps other users who later run into this issue. You **SHOULD** give as much information as possible including what commit/branch, what OS/version and so on.

You **SHOULD NOT** open a ticket if another already exists for the same issue. You **SHOULD** instead either add more information by commenting on it, or simply comment to inform the maintainer that you are also affected. The maintainer **SHOULD** reply to every new ticket when they are opened. If the maintainer didn't say anything after a few days, you **SHOULD** write a new comment asking for more information.

You **SHOULD** provide a reproducible test case, either in the ticket or by sending a pull request and updating the test suite.

When you have a fix ready, you **SHOULD** open a pull request, even if the code does not fit the requirements discussed below. Providing a fix, even a dirty one, can help other users and/or at least get the maintainer on the right tracks.

You **SHOULD** try to relax and be patient. Some tickets are merged or fixed quickly, others aren't. There's no real rules around that. You can become a paying customer if you need something fast.

5 Security reports

You **SHOULD** open a ticket when you identify a DoS vulnerability in this project. You **SHOULD** include the resources needed to DoS the project; every project can be brought down if you have the necessary resources.

You **SHOULD** send an email to contact@ninenines.eu when you identify a security vulnerability. If the vulnerability originates from code inside Erlang/OTP itself, you **SHOULD** also consult with OTP Team directly to get the problem fixed upstream.

6 Feature requests

Feature requests are always welcome. To be accepted, however, they must be well defined, make sense in the context of the project and benefit most users.

Feature requests not benefiting most users may only be accepted when accompanied with a proper pull request.

You **MUST** open a ticket to explain what the new feature is, even if you are going to submit a pull request for it.

All these conditions are meant to ensure that the project stays lightweight and maintainable.

7 Documentation submissions

You **SHOULD** follow the code submission guidelines to submit documentation.

The documentation is available in the *doc/src/* directory. There are three kinds of documentation: manual, guide and tutorials. The format for the documentation is Asciidoc.

You **SHOULD** follow the same style as the surrounding documentation when editing existing files.

You **MUST** include the source when providing media.

8 Examples submissions

You **SHOULD** follow the code submission guidelines to submit examples.

The examples are available in the *examples/* directory.

You **SHOULD** focus on exactly one thing per example.

9 Code submissions

You **SHOULD** open a pull request to submit code.

You **SHOULD** open a ticket to discuss backward incompatible changes before you submit code. This step ensures that you do not work on a large change that will then be rejected.

You **SHOULD** send your code submission using a pull request on GitHub. If you can't, please send an email to contact@ninenines.eu with your patch.

The following sections explain the normal GitHub workflow.

9.1 Cloning

You **MUST** fork the project's repository on GitHub by clicking on the *Fork* button.

On the right page of your fork's page is a field named *SSH clone URL*. Its contents will be identified as `$ORIGIN_URL` in the following snippet.

On the right side of the project's repository page is a similar field. Its contents will be identified as `$UPSTREAM_URL`.

Finally, `$PROJECT` is the name of this project.

To setup your clone and be able to rebase when requested, run the following commands:

```
$ git clone $ORIGIN_URL
$ cd $PROJECT
$ git remote add upstream $UPSTREAM_URL
```

9.2 Branching

You **SHOULD** base your branch on *master*, unless your patch applies to a stable release, in which case you need to base your branch on the stable branch, for example *1.0.x*.

The first step is therefore to checkout the branch in question:

```
$ git checkout 1.0.x
```

The next step is to update the branch to the current version from *upstream*. In the following snippet, replace *1.0.x* by *master* if you are patching *master*.

```
$ git fetch upstream
$ git rebase upstream/1.0.x
```

This last command may fail and ask you to stash your changes. When that happens, run the following sequence of commands:

```
$ git stash
$ git rebase upstream/1.0.x
$ git stash pop
```

The final step is to create a new branch you can work in. The name of the new branch is up to you, there is no particular requirement. Replace `$BRANCH` with the branch name you came up with:

```
$ git checkout -b $BRANCH
```

Your local copy is now ready.

9.3 Source editing

There are very few rules with regard to source code editing.

You **MUST** use horizontal tabs for indentation. Use one tab per indentation level.

You **MUST NOT** align code. You can only add or remove one indentation level compared to the previous line.

You **SHOULD NOT** write lines more than about a hundred characters. There is no hard limit, just try to keep it as readable as possible.

You **SHOULD** write small functions when possible.

You **SHOULD** avoid a too big hierarchy of case clauses inside a single function.

You **SHOULD** add tests to make sure your code works.

9.4 Committing

You **SHOULD** run Dialyzer and the test suite while working on your patch, and you **SHOULD** ensure that no additional tests fail when you finish.

You can use the following command to run Dialyzer:

```
$ make dialyze
```

You have two options to run tests. You can either run tests across all supported Erlang versions, or just on the version you are currently using.

To test across all supported Erlang versions:

```
$ make -k ci
```

To test using the current version:

```
$ make tests
```

You can then open Common Test logs in *logs/all_runs.html*.

By default Cowboy excludes a few test suites that take too long to complete. For example all the examples are built and tested, and one Websocket test suite is very extensive. In order to run everything, do:

```
$ make tests FULL=1
```

Once all tests pass (or at least, no new tests are failing), you can commit your changes.

First you need to add your changes:

```
$ git add src/file_you_edited.erl
```

If you want an interactive session, allowing you to filter out changes that have nothing to do with this commit:

```
$ git add -p
```

You **MUST** put all related changes inside a single commit. The general rule is that all commits must pass tests. Fix one bug per commit. Add one feature per commit. Separate features in multiple commits only if smaller parts of the feature make sense on their own.

Finally once all changes are added you can commit. This command will open the editor of your choice where you can put a proper commit title and message.

```
$ git commit
```

Do not use the `-m` option as it makes it easy to break the following rules:

You **MUST** write a proper commit title and message. The commit title is the first line and **MUST** be at most 72 characters. The second line **MUST** be left blank. Everything after that is the commit message. You **SHOULD** write a detailed commit message. The lines of the message **MUST** be at most 80 characters. You **SHOULD** explain what the commit does, what references you used and any other information that helps understanding why this commit exists. You **MUST NOT** include commands to close GitHub tickets automatically.

9.5 Cleaning the commit history

If you create a new commit every time you make a change, however insignificant, you **MUST** consolidate those commits before sending the pull request.

This is done through *rebasing*. The easiest way to do so is to use interactive rebasing, which allows you to choose which commits to keep, squash, edit and so on. To rebase, you need to give the original commit before you made your changes. If you only did two changes, you can use the shortcut form `HEAD^^`:

```
$ git rebase -i HEAD^^
```

9.6 Submitting the pull request

You **MUST** push your branch to your fork on GitHub. Replace `$BRANCH` with your branch name:

```
$ git push origin $BRANCH
```

You can then submit the pull request using the GitHub interface. You **SHOULD** provide an explanatory message and refer to any previous ticket related to this patch. You **MUST NOT** include commands to close other tickets automatically.

9.7 Updating the pull request

Sometimes the maintainer will ask you to change a few things. Other times you will notice problems with your submission and want to fix them on your own.

In either case you do not need to close the pull request. You can just push your changes again and, if needed, force them. This will update the pull request automatically.

```
$ git push -f origin $BRANCH
```

9.8 Merging

This is an open source project maintained by independent developers. Please be patient when your changes aren't merged immediately.

All pull requests run through a Continuous Integration service to ensure nothing gets broken by the changes submitted.

Bug fixes will be merged immediately when all tests pass. The maintainer may do style changes in the merge commit if the submitter is not available. The maintainer **MUST** open a new ticket if the solution could still be improved.

New features and backward incompatible changes will be merged when all tests pass and all other requirements are fulfilled.