Wikiprint Book

Title: Contributing for users

Subject: TracMeteo - DRM4G/Development

Version: 34

Date: 05/17/2022 12:17:23 AM

Table of Contents

Contributing for users

To make it easier and more accessible for anyone to play around with the DRM4G, we have decided to host our source code on **GitHub**.

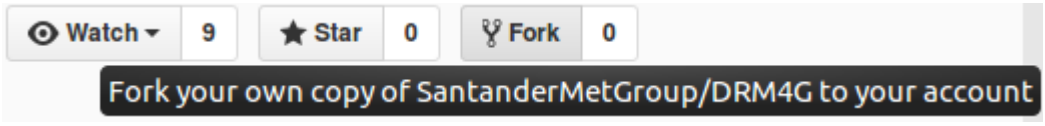
If you want to contribute to this repository please be aware that this project follows a certain gitflow.

So please fork this repository and create a local branch in which to do your work, and then create a pull requests to the main DRM4G repository.

Necessary Steps

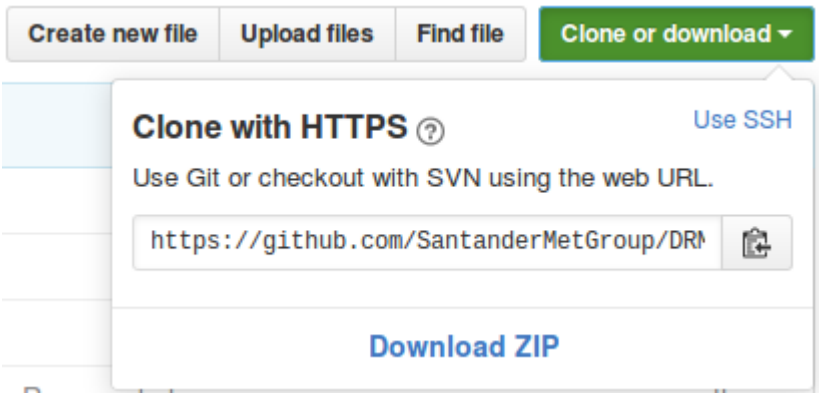
For those of you who wish to help but don't know how, the first thing you need is a ?GitHub account.

In our project's page hit the "***Fork***" button at the top right corner:



This will create a copy of our repository in your account where you'll develop your own feature or implement a bugfix that you may believe is necessary. You'll be submitting changes to this one until you are certain everything works properly, at which point you can request to have your changes integrated into the DRM4G's repository.

To continue, you'll need to setup a local repository where you'll be changing the code and doing your testing. To do this you'll need your repository URL, that can be obtained adding **.git** to your project page or by clicking on the "***Clone or download***" button:



In Linux operating systems:

- Open a terminal on the folder in which you wish your local copy of the repository to be stored (it's recommended to use an empty folder) and run the following commands:

```
git init  #to initialize an empty Git repository
git remote add origin <your_repository_url> #to make your local repository point to your remote repository in GitHub
git fetch
git checkout develop #to create a local copy of the develop branch
```

To be consistent with our gitflow, all you'll be able to do is create **feature or bugfix branches**, and you'll have to follow our naming conventions to do so.

  Our GitHub's default branch is "**develop**", and this is done intentionally, since the "**master**" will only be updated when publishing a new release
  - This means that all pull requests to the "**master**" branch will be rejected
- The naming convention will just be to create branches in lower case letters separated by underscores ("_") that describe what you're trying to accomplish with the branch.

With this, you will now have DRM4G's source code at your disposal.

From here you could create branches for every new feature you'd like to include to the DRM4G, for a more in depth tutorial on how to do that, click ?here.

Testing the DRM4G

Once you've made the changes you wanted to, you'll want to install your version to be sure that your new feature is working properly.

Just in case you would like to try out different versions, we recommend you use a virtual environment to test it.

- ?Here you can find a tutorial on how to install a virtual environment, or you can look for one on your own.

Before you can install and try out your own version, you'll have to build your own package:

- Open a terminal in the folder where your repository is located.
- Run the command `python setup.py sdist`

This will create a a distribution package under a folder called **dist**.

Installing your version in a virtual environment

Go to wherever you have your virtual environment, open a terminal and execute the following commands:

```
source bin/activate
export DRM4G_DIR = $PWD/conf
pip install path/to/drm4g/package
```

DRM4G_DIR is where the configuration files will be installed. More information here.

And that's it. Now you can use and test your own version of DRM4G.

For other ways to install the DRM4G, you can check here.

How to debug code

You have at your disposal a number of ways to check what could have gone wrong when something breaks.

Via the DRM4G CLI

- All of the *drm4g commands* can be executed in debug mode by adding the option "**--dbg**".

Via the logger

If there has been any error, chances are that you can find the cause by looking at DRM4G's log files, found in "$DRM4G_DIR/.drm4g/var"

- The DRM4G is divided into different parts, and each one of them has its own logger.
- By default, the logger level is set to *INFO* for all of them. To see all of the log messages that the DRM4G can record, you'll have to modify the file "**$DRM4G_DIR/.drm4g/etc/logger.conf**" and change the level of the logger to "**DEBUG**" for each part that you wish to check.

Via the job's logs

- In addition, all of the individual jobs submitted have their own log files. They can help you see in which phase did the program stop working.
  They are grouped in folders by their job ID, every hundred jobs. They can be located in "$DRM4G_DIR/.drm4g/var".
  - The folders will look like this: "**000-099**", "**100-199**", ...
- Another way to view them is to use the command `drm4g job log <job_id>`

Commiting changes

After you've tested that everything is in working order, it's time to update your GitHub fork.

```
git add .
git commit -m "Description of the changes you've made"
git push origin develop
```

From here you'll have to create a **Pull request**.

Creating a Pull Request

As time passes, chances are that the main DRM4G repository has had other contributors merge changes into it, so you should first update your local repository and check that there are no conflicts.

```
git checkout develop
git remote add github https://github.com/SantanderMetGroup/DRM4G.git #just do run this command the first time to add the m
git pull github develop
```

If there have been any changes, but there are no conflicts, you can just *push* your changes into your remote repository.

```
git push origin develop
```

In the case that there are conflicts, resolve them and commit the changes, after do the *push* to update your remote repository.

After, you just have to go to your repositories web page and click on the "***New pull request***" button.

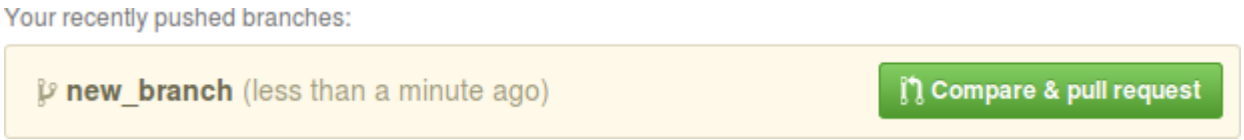
A message will tell you if there are any conflicts that need to be solved or if an automatic merge is possible. Then you just have to click on the "***Create pull request***", give this merge a title (which will serve as the commit message if the pull request is accepted), a comment if you want (can be useful to explain in more detail why this should be integrated into the DRM4G), and click a second time on the "***Create pull request***" button.

Now you'll have to wait and see if the administrator of the project accepts the changes you're proposing.

Another option would be to create a new branch from the "**develop**" branch and then try to make a pull request from that.

```
git checkout develop
git checkout -b new_branch
#make some changes
git add .
git commit -m "Description of the changes you've made"
git push origin new_branch
```

The next time you enter your repository's web page, you'll see a new button:



Click on "***Compare & pull request***" to perform a pull request on to the main repository. The same will happen as when clicking on "New pull request" when on a previously existing branch.

Contributing for developers

The aim of this section will be to explain the process we follow to contribute to the development of the DRM4G.

We use something similar to a ?gitflow workflow. Internally we synchronize our work with **GitBucket**, which is what will be considered as the ***central repository*** of the **Santander Meteorology Group**.

- You can find the project ?here, but to be able to access it you must first get a ?GitBucket account and get access to the private repository.
- Have in mind that we try to adopt the DevOps development methodology.

Necessary steps

There are a lot of tutorials showing how to use git, including our own, but this is more straightforward explanation on how to get started.

In Linux operating systems:

- Open a terminal on the folder in which you wish your local copy of the repository to be stored (it's recommended to use an empty folder) and run the following commands:

```
git init  #to initialize an empty Git repository
git remote add origin https://meteo.unican.es/gitbucket/git/DRM4G/DRM4G.git #to make your local repository point to the re
git fetch
git checkout master #to create a local repository of the master branch
git checkout develop #to create a local repository of the develop branch
```

If you want to see the changes your doing, check the file "**.git/config**" under your repository directory.

With this, you will now have DRM4G's source code at your disposal.

From here you should follow our gitflow and create branches for every new feature you'd like to include to the DRM4G.

- To create a new branch and switch to it you can use the command `git checkout -b <branch_name>`.
- For a more in depth tutorial on how branching works, click ?here.

Our Git workflow

In this section you'll find the guidelines of how we tackle the development process.

- A section has already been written specifying how to test the DRM4G.

As mentioned above, to us, our *central* repository will be a private one hosted by ***GitBucket***, but in addition we have a second public one hosted in ***GitHub*** to make the DRM4G accessible to anyone that might want to contribute to the project.

To add the **GitHub** repository as a new remote:

```
git remote add github https://github.com/SantanderMetGroup/DRM4G.git
```

Both this two repositories have at least two branches, the **master** and the **develop** branches, that must be synchronized at all times.

- Commits in the **master** branch represent all the stable versions of the DRM4G. This means that merges to this branch are only done when ready to publish a release.
- The **develop** branch reflects the whole evolution of the project. It shows all of the new features and bug fixes that have been included. This is the main branch where all of the work will be done.

If during development, you find yourself working in a team and wish to share your code, push your work to our *central* repository in *GitBucket*, not *GitHub*.

```
git push -u origin <branch_name>
```

Something to remember then, is that when you've finished your work on the branch, you mustn't forget to eliminate it from the remote repository.

```
git branch -d <branch_name> #deletes the branch from your local repository
git push origin :<branch_name> #deletes the same branch from the remote repository
```

Another reminder, is that before *pushing* your work onto the remote repositories, you should always check that no changes have been made in them.

```
git remote -v update #updates your remote refs and shows the state of the branches
git status -uno #informs you whether your current branch is ahead, behind or has diverged from the remote
```

In the case that changes exist in the remote repository, you should perform a *pull*, resolve any possible conflicts, and then make the *push*.

```
git pull origin <branch_name>
#do necessary operations
git push origin <branch_name>
```

Adding new features

Each time some big change or new feature is going to be implemented, a new **feature branch** has to be created from the **develop** branch.

The **feature branches** are how new functionalities are introduced and tested. Normally the finalization of one of them will also define when a new release will be published, but that's not always the case since a new release may include more than one new feature.

As for the naming convention for these branches, they can be anything separated by underscores ("_").

- However the name should try to be describe what is being implemented with it.

```
git checkout develop
git checkout -b new_feature #it will switch you to the new branch
```

When finished, the branch has to be merged back into the **develop** branch. After it can be deleted.

- To keep the log history of the branches created, the merges to the **develop** branch, will always be done with the no fast-forward option "**--no-ff**".

```
git checkout develop
git merge --no-ff new_feature
git branch -d new_feature
git push origin develop
git push github develop
```

To make the log more readable, in the cases in which a lot of small commits were made in the **feature branch**, it may be advisable to *rebase* to merge some commits before doing the merge into **develop**:

```
git rebase -i <hash_code>
```

Updating a branch

Sometimes you'll want to incorporate the changes or improvements made in one branch into another one. To avoid making the log history too complicated when merging the **feature branch** back into the **develop** branch, the merges will be fast forwarded.

```
git checkout branch_a
git merge branch_b
```

Getting ready for a release

Once all the new changes for the next release have been added to the **develop** branch and it is at a stable point, it's time to publish a new release.

From this point on, all the following changes done on the **develop** branch will be for the following release. So, to avoid halting the development of the project a new **release branch** is created indicating the new release number.

The nomenclature for all **release branches** will be a string followed by the new version in numbers in the form of **drm4g-MAJOR.MINOR.PATCH** and they will have their numbers increase following these guidelines:

- **MAJOR** version when you add some new functionality or you make incompatible API changes
- **MINOR** version when you improve some part of the DRM4G's functionality in a backwards-compatible manner
- **PATCH** version when you make backwards-compatible bug fixes.

- For creating **tags**, the same naming convention will be followed, albeit just the digits will be used.

It is in this branch where the version number of all the files will be modified, and where the code will be brought to a release ready state. That means that it is where minor bug fixes will be made, where comments and unnecessary code snippets will be removed and other maintenance tasks will be carried out.

- During this process you might want to merge the new version number or some of the bug fixes on to the **develop** branch so that future features may incorporate them.

When ready, all that needs to be done is merge it into the **master**, create a tag, update the remote repositories and publish the new release.

```
git checkout master
git merge --squash drm4g-X.X.X
git commit -v
git tag X.X.X
git push origin master --tags
git push github master --tags
```

- When performing the commit, the commit message will include all the previous commits made, but at the beginning a detailed message describing what is being added should be included.
- Squashing all the commits when doing the merge, will ensure that every commit in the master is a stable version and will make the log history cleaner, allowing us to easily see the changes made between commits.

Updating the **develop** will be a bit different. If it is ever needed to rollback to a previous version, its commit has to be accessible. So merges to the **develop** branch *won't be squashed*.

```
git checkout develop
git merge --no-ff drm4g-X.X.X
git push origin develop
git push github develop
```

Aferwards you can just delete the **release branch**.

```
git branch -d drm4g-X.X.X
```

Problems merging branches

When performing a merge, there may be some complications.

- If you get a merge conflict when for example merging a release branch (**drm4g-X.X.X**) into **master**, which would make you lose the whole automatically generated squash commit message since you'd have to resolve conflicts and commit on your own, do this:

```
git checkout master
git merge --squash drm4g-X.X.X
#conflict occurred
git merge --abort #or in case you commited without realizing it "git reset --hard HEAD~1" to go back to the previous commi
#"git reset --hard HEAD" if the "--abort" commands gives you this message "fatal: There is no merge to abort (MERGE_HEAD m
git merge --squash -Xtheirs drm4g-X.X.X
git commit -v
```

- This will choose all of the changes made in the branch over the ones in the **master** branch, so only use this if you're completely sure of what you're doing.

Hotfixes

As an exception to the normal flow of the development, in the cases when an important error is discovered only after the release has been published, a **hotfix branch** can be created from the **master** to fix it.

- This is done like this as to not disrupt the teams working on the **develop** branch.
  The nomenclature followed would be **hotfix-MAJOR.MINOR.PATCH**
  - The version number would only have to bump up by one the **PATCH** version.
  - All of the files would have to have their version numbers modified.

After the problem was solved, you'd have to follow the instructions to prepare for a new release, except considering the **hotfix branch** to be the **release branch**.

- That means that a new release branch shouldn't be created. But this branch would have to be merged back into the **master** and the **develop** branch following the instructions set in the previous section.

If it turns out to be a critical error and not something fixable in a few hours, the release would have to be retracted until the issue got resolved.

Publishing a release

After having updated the **master**, it's time to publish the release.

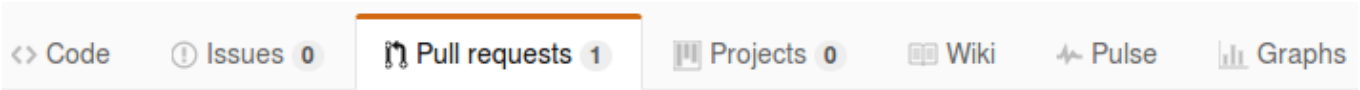
1. First, the package has to be uploaded to PyPI, there's a few ways to do this, but you'll need to have an account with access to the groups ?PyPI page.
   Run the command `python setup.py register` or modify your ?$HOME/.pypirc file and add your credentials to be able to submit package data to PyPI.
   - This step is only needed the first time.
3. Finally submit the distribution files.
   - Run the command `python setup.py sdist upload`
     or
   - Run the command `python setup.py sdist` and then upload the package though PyPI's user interface.

At the end, the last step is to update the realese wiki page and the references in the section "New releases" of the DRM4G's main wiki page.

Managing the GitHub account

For the administrator of the DRM4G's GitHub repository:

When you see that there's a new pull request, either because you saw it on the projects page or because you received a message in your mail informing you about it, you can click on the "***Pull requests***" button.



If you agree with the changes made, and there are no conflicts, you can just click on the "***Merge pull request***" button.

- You'll automatically see a commit message that will look something like this: "Merge pull request #X from <contributor>/<branch_name>"
- Bellow that, you can write an "optional extended description".
  - When running `git log` from the command line, you'll just see both lines shown as the commit message.
- When done just click on the "***Confirm merge***" button to finish.

If don't agree or there are conflicts, you can click right next to the "Merge pull request" button, where it says "***command line instructions***" to perform a manual merge.

- Just follow the instructions shown there.
- Optionally, you could also try to solve the conflicts wiht the web editor, but this isn't recommended since you can't test what you're changing, unless the conflict is in a comment or a text message, not in the code.

SVN repository migration to Git

This section exists in case there's a need to convert any other project from ***Subversion*** to ***Git***, including *tags*, the same way that has been done for the *DRM4G* and *WRF4G*. There may be better ways of doing it, but this is the only one that we've tested.

The program used is called ?svn2git. Follow the link to see how to install it.

Once installed, you can try to use the options they have listed in ?usage.

- First create a new folder where you want to have your local repository and change into it

```
mkdir <local_repository_folder>
cd <local_repository_folder>
```

This next command will then **convert the whole remote svn repository** into a local git repository.

- You'll need the `--username` option since our repository is protected by password.
- This process will take a long time.

```
svn2git https://meteo.unican.es/svn/repos/WRF4G/ --username <user_name>
```

When it's done, get into the branch you want to start a project from and rename it as the *master*

- You might have to clean up a bit the root folder, since there might be some folders that shouldn't be there. But they should be empty and should be fine to just delete them.

```
git checkout <project_branch>
rm -rf WRF* #if you need to clean up
#if they were empty you won't need to make a commit
```

```
#rename the project branch
git branch -m master old-master
git branch -m <project_branch> master
```

- Then push it to our remote repository in ***GitHub***

```
git remote add origin https://github.com/SantanderMetGroup/<project_name>.git
git push -u origin master
```

- After delete the tags that don't have anything to do with the project and push them onto the remote repository

```
git tag
git tag -d <tag-names>
```

```
git push origin --tags
```

- Finally, depending on whether you'll need the rest of them or not, delete the remaining branches.

```
git branch
git branch -D <every_branch>
```