Table of Contents

Development

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.

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 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.

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>`

Our Git workflow

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

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 musn'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 hyphens ("**-**"), but always adding the prefix "**drm4g-**".

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

```
git checkout develop
git checkout -b drm4g-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 drm4g-new-feature
git branch -d drm4g-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>
```

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
git push origin --tags
git push github master
git push github --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 use 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 tag X.X.X
git push origin develop
git push origin --tags
git push github develop
git push github --tags
```

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

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

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.