



# The Open Student Information System



## **Development and Operations**

Hildeberto Mendonça Filho

# **OSIS - Development and Operations**

Hildeberto Mendonça

© 2016 Hildeberto Mendonça

# Contents

<b>Development</b>	<b>1</b>
<b>The Development Environment</b>	<b>2</b>
Installing and configuring Git	2
Configuring Git to Simplify GitHub Authentication	3
Installing and Configuring PostgreSQL	5
Creating the Database	5
Installing and Configuring Django	6
Installing the Application	6
<b>Development</b>	<b>9</b>
The Layers	9
Internationalization	10
Programming Good Practices	11
Security	11
<b>Development Workflow</b>	<b>12</b>
GitFlow Workflow	12
Following GitFlow	14
<b>Operations</b>	<b>23</b>
<b>Maintenance</b>	<b>24</b>

# Development

# The Development Environment

The development environment is certainly the first thing to do to start contributing to OSIS. In this section, we explain in details how to do that in a Debian-based Linux distribution (Debian Linux, Ubuntu, Mint, etc.). This is the recommended operating system for development and deployment.

If your operating system is not based on Debian Linux then you can rely on a virtual machine as a development environment. For the virtual machine, we suggest [VirtualBox](https://www.virtualbox.org)<sup>1</sup> as virtual runtime and [Ubuntu Mate](https://ubuntu-mate.org)<sup>2</sup> as operating system. Follow the installation instructions on their respective websites according to your own platform. When the Ubuntu virtual machine is created and running, execute the following command to make the operating system aware of the virtual runtime:

```
$ sudo apt-get install virtualbox-guest-dkms virtualbox-guest-x11
```

## Installing and configuring Git

Git is a distributed version control system used to manage the source code of OSIS. The source code is necessary to develop and deploy OSIS and we start the instructions by explaining how to install and configure it. We can use `apt-get` to install Git:

```
$ sudo apt-get update
$ sudo apt-get install git
```

The `update` command downloads package lists from remote repositories to get information about the newest versions of packages and their dependencies. This way, we make sure we are getting the last version of Git and all other dependencies. Then `install git` performs the installation. Next, we add some personal information in the local Git installation to make sure you are well identified in all commits:

---

<sup>1</sup><https://www.virtualbox.org>

<sup>2</sup><https://ubuntu-mate.org>

```
$ git config --global user.name "[Firstname] [Lastname]"
$ git config --global user.email "[firstname.lastname@domain.com]"
```

Since version 2.0, Git has adopted a new behavior to pull and push commits while in a branch. When you execute `git push` or `git pull` Git will consider pushing or pulling just for the current branch. Before, these commands would push and pull all branches. But the change to this new behavior is voluntary, not automatically imposed. So, we have to explicitly say we have to move from the old behavior to the new one. To do that, execute the following command:

```
$ git config --global push.default simple
```

## Configuring Git to Simplify GitHub Authentication

For the moment, every time we push code to GitHub the prompt asks for a username and password. We can bypass this step by registering a SSH key. To do that, we first check whether there is already an existing SSH key we can reuse:

```
$ ls -al ~/.ssh
```

If files with the extension `.pub` are listed then one of them can be reused to authenticate to GitHub. If not, then we can create one:

```
$ ssh-keygen -t rsa -b 4096 -C "[firstname.lastname@domain.com]"
Enter file in which to save the key (/Users/[user]/.ssh/id_rsa): [Press enter]
Enter passphrase (empty for no passphrase): [Type a passphrase]
Enter same passphrase again: [Type passphrase again]
```

The next step is to add the new key - or an existing one - to the `ssh-agent`. This program runs the duration of a local login session, stores unencrypted keys in memory, and communicates with SSH clients using a Unix domain socket. Everyone who is able to connect to this socket also has access to the `ssh-agent`. First, we have to enable the `ssh-agent`:

```
$ eval "$(ssh-agent -s)"
```

And add key to it:

```
$ ssh-add ~/.ssh/id_rsa
```

The next step is to make GitHub aware of the key. For that, we have to copy the exact content of the file `id_rsa.pub` and paste into GitHub. To make no mistake about the copy, install a program called `xclip`:

```
$ sudo apt-get install xclip
```

And then copy the content of the file `id_rsa.pub` in the clipboard:

```
$ xclip -sel clip < ~/.ssh/id_rsa.pub
```

The command above is the equivalent of opening the file `~/.ssh/id_rsa.pub`, selecting the whole content and pressing `Ctrl+C`. This way, you can paste the content on GitHub when required in the next steps. On the GitHub side:

- Login at <https://github.com>
- In the top right corner of the page, click on the profile photo and select **Settings**
- In the user settings sidebar, click **SSH keys**
- Then click **Add SSH key**
- In the form, define a friendly title for the new key and paste the key in the **Key** field
- Click **Add Key** to finish with GitHub

To make sure everything is working, let's test the connection:

```
$ ssh -T git@github.com
The authenticity of host 'github.com (207.97.227.239)' can't be established.
RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
Are you sure you want to continue connecting (yes/no)? yes
_
Hi [username]! You've successfully authenticated, but GitHub does not
provide shell access.
```

This configuration works only when we use a ssh connection to GitHub. To verify that, go to one of your local GitHub projects and check the url pointing to the server:

```
$ cd ~/python/projects/osis/osis
$ git remote -v
```

If the url starts with `https://` then you are using `https` instead of `ssh`. In this case, you should change the url to the `ssh` one:

```
$ git remote set-url origin git@github.com:uclouvain/osis.git
```

The automatic authentication should work after that.

## Installing and Configuring PostgreSQL

PostgreSQL is the only database supported by OSIS. In theory, the Django ORM would make the application database-independent, but we do not test OSIS with other databases, thus we cannot guarantee that it works on other databases such as MySQL or Oracle. Fortunately, PostgreSQL has a very good reputation, a large community and a generous documentation.

TIP: If you really need a database different from PostgreSQL then you could contribute to the project by testing OSIS on your favorite database.

Execute the following commands to install PostgreSQL:

```
$ sudo apt-get install postgresql
$ sudo su - postgres -c "createuser -s $USER"
$ sudo apt-get install libpq-dev
```

The first command installs PostgreSQL and creates a user named after the current logged OS user. The library `libpq-dev` is also installed for development purposes.

## Creating the Database

Before moving forward, make sure you installed PostgreSQL, as explained in the section <<installing-postgresql>>. Then, follow the steps below to create the backend database:



```
$ createdb osis_backend_dev
$ createuser osis_usr -P      // Inform the password 'osis' when asked for.
$ psql -d osis_backend_dev
    =# grant connect on database osis_backend_dev to osis_usr;
    =# revoke connect on database osis_backend_dev from public;
    =# \q
```

Now, install the necessary dependencies to allow a Python application to connect to PostgreSQL:

```
$ sudo apt-get install python3-dev libpq-dev
```

## Installing and Configuring Django

The Django installation and all other dependencies depends on where we place the OSIS repository in our machine. So, we start by creating the directory where the repository will be placed.

```
$ mkdir -p ~/python/projects/osis
$ cd ~/python/projects/osis
```

The repository `osis` is a full Django project with several modules. To clone the repository locally we executed the following commands:

```
$ cd ~/python/projects/osis
$ git clone git@github.com:uclouvain/osis.git
```

Install the Python virtual environment and other system dependencies:

```
$ sudo apt-get install build-essential python-virtualenv libjpeg-dev libpng-dev
```

In the new repository, create a virtual environment to isolate all dependencies of the project:

```
$ cd osis
$ virtualenv --python=python3.5 venv
```

## Installing the Application

Start the virtual environment and install the dependencies:

```
$ source venv/bin/activate
(venv)$ pip install -r requirements.txt
```

Create the data structure in the database:

```
(venv)$ python manage.py migrate
```

At this point we have two options:

1. we create a super user and go on with an empty database or
2. we load the demonstration data that already contains a superuser

To create the super user and continue with an empty database:

```
(venv)$ python manage.py createsuperuser
Username (leave blank to use '[linux-user]'):
Email address: your@emailaddress.com
Password:
Password (again):
Superuser created successfully.
```

You will need this user to login on OSIS for the first time and be able to create other users.

To load the demonstration data that already contains a superuser:

```
(venv)$ python manage.py loaddata demo_data.json
```

The demonstration data create a super user with the following credentials:

```
Username: osis
Password: osisosis
```

The demonstration data also create several other users. The password for each user is the username typed twice (e.g. user: antonin password: antoninantonin).

Now, we can run the application:

```
(venv)$ python manage.py runserver
```

You can leave the server running while you are developing. It will take into account all changes in your code, except the changes in the model. In this case, we have to stop the server to execute the commands `makemigrations` and `migrate` as shown above. When we have finished your daily work, we can deactivate the virtual environment:

```
(venv)$ deactivate
```

# Development

## The Layers

### Model

The model is composed of entities entitled to represent the business' properties and behaviors as well as preserve their state over time with the help of a database. In the Django Framework, entities are represented by classes that inherit from `django.db.models.Model`. Each class of the model represents a table in the database. Instances of the class are records in that table. Attributes of the class are equivalent to columns of the table. Every change in this abstraction is reflected in the database through a mechanism known as database migration.

Django effectively makes things simple, but when we change the model of an application that is deployed multiple times in multiple places, then the database migration is no longer trivial.

1. Modify the entity classes as needed.

### User Interface

#### Testing The User Interface Using Selenium

Selenium relies on HTML elements' ids to identify and interact with user interface elements. In other words, if an element doesn't have an id then Selenium cannot reference it in its test scripts. Therefore, it is important that every element the user interacts directly - such as text fields, links or buttons - must have an id.

The amount of ids to be defined in a single page is not negligible. Since ids must not repeat in the same page, at some point we will lack creativity to think about more unique values to identify the elements. To help with this, we have defined a list of prefixes, one for each type of element, listed in the table below:

Prefix	Element Type	Prefix	Element Type	Prefix	Element Type
bt_	Button	pnl_	Div	tab_	Tab
chb_	Checkbox	rdb_	Radio button	txa_	Text area
fil_	File field	lnk_	Link	txt_	Text field
form_-	Form	num_-	Numeric field		
hdn_	Hidden field	slt_	Combobox		

Some examples of use:

```
<!-- Text field -->
<input type="text" id="txt_start_date">

<!-- Hidden field -->
<input type="hidden" id="hdn_academic_calendar">

<!-- Combobox field -->
<select id="slt_academic_year">
```

## Internationalization

OSIS has adopted a different approach when it comes to internationalization (I18N). Text messages within Python code and template files are keywords instead of text in plain English. Hardcoding keywords forces us to create a translation file for each supported language, instead of relying on hardcoded messages for the default language. For example, instead of:

```
<p>{% trans 'Hello World' %}</p>
```

we use:

```
<p>{% trans 'hello_world' %}</p>
```

which is translated into English and French respectively:

```
locale/en/LC_MESSAGES/django.po
msgid "hello_world"
msgstr "Hello World!"
```

```
locale/fr_BE/LC_MESSAGES/django.po
msgid "hello_world"
msgstr "Salut monde!"
```

It is important to add that we add translations manually in the translation files instead of using `./manage.py makemessages`. We have noticed some strange behaviors, such as confusing “all” with “ill”, and the translation files are changed much more than necessary, causing lots of conflicts when multiple developers contribute to the translations.

Once the translations are done, we finally compile the messages into `.mo` files:

```
$ ./manage.py compilemessages
```

Only `.po` files are committed to the repository. `.mo` files are ignored in the `.gitignore` file.

## Programming Good Practices

Every function returns one and only one value.

The name and the documentation of the function is consistent with what the function really does.

## Security

An important part of the security is the protection of the server where the application runs. However, it is equally important to take some precautions while developing the application to avoid adding vulnerabilities unintentionally. Full attention should be given to the following points:

1. avoid passing user related identifiers (person, tutor, program manager, etc.) in URLs (`?tutor_id=34` or `/score_encoding/print/34/`) because an attacker can try several numbers in sequence to access information he has no privilege to.
2. while importing a file, verify whether the user has the right to add/modify in the target tables.

# Development Workflow

The development workflow defines the sequence of tasks and events that takes place during the development of OSIS, with the goal of producing stable releases. The workflow is controlled by Git, because of its flexible branching features, and complemented by tools directly integrated to it, such as GitHub and GitKraken.

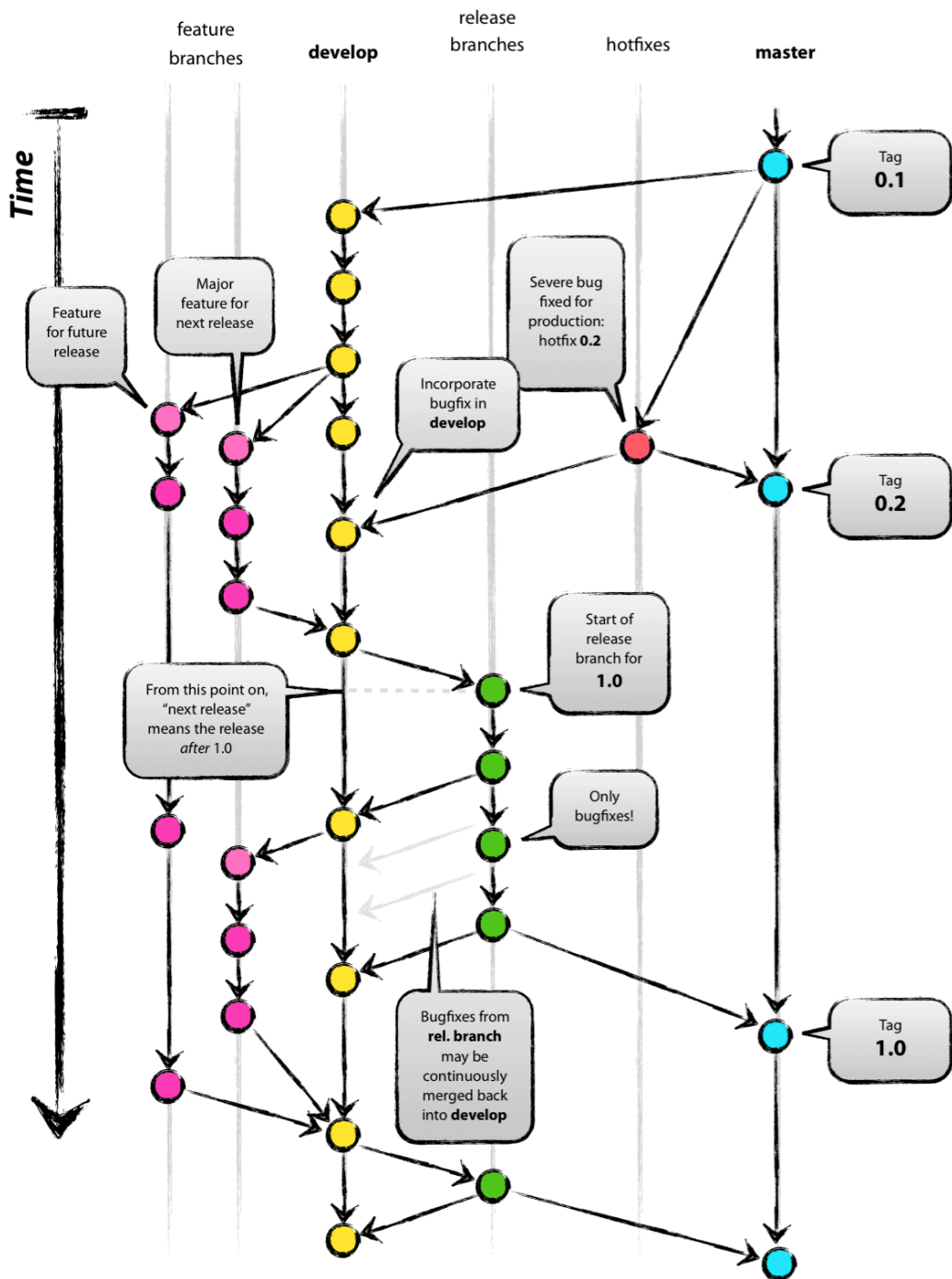
Git is flexible enough to support any workflow we have in mind. But, instead of reinventing the wheel, we started from a widely used workflow called GitFlow then we built and adapted it for our special needs.

## GitFlow Workflow

[GitFlow](#)<sup>3</sup> was created by Vincent Driessen back in 2010 and it has been recognized as the de facto workflow for Git. The following figure illustrates how it works.

---

<sup>3</sup><http://nvie.com/posts/a-successful-git-branching-model/>





## Following GitFlow

The code repository is organized in three fixed branches:

- *dev*: aggregates developers' contributions that are intended to be in production, but they still need to be validated.
- *qa*: at the end of the sprint, when all features are frozen, the branch *dev* is merged into *qa* to allow testers to validate the release before it gets into production.
- *master*: once the version in *qa* is fully validated, it is merged into the branch *master*, which is the one to be deployed in production.

Developers should not commit directly to any of these branches. By convention, these branches can only be changed if there is an issue in the <https://github.com/uclouvain/osis/issues> [issue tracking tool] that justifies the creation of an exclusive branch for that issue. For instance: if the issue's number is *#234* then its respective branch is named *issue#234*, created to isolate the changes described in the issue. To create a new branch for the issue, perform the following commands:

```
$ git checkout dev
$ git pull origin dev
$ git checkout -b issue#234
```

The first command enters in the branch *dev*, if the developer is not already in there. Within the branch *dev*, the latest commits in the remote branch *origin/dev* are downloaded and merged within the local branch *dev*. Then, the new branch *issue#234* is created from the local branch *dev*.

The developer in charge of the issue *#234* changes the code within the branch *issue#234*. Two commands are very useful to keep track of what has been done:

```
$ git status
$ git diff models.py
```

The first command shows all created, modified, removed and untracked files that are candidates to be committed. The second shows the changes in one of the modified files. When we are ready to commit, we should decide whether all changed files will be included in the commit or just a subset of them. To include all files:

```
$ git commit -a -m "New entities added."
```

To include a subset of files, we have to add each file individually:

```
$ git add base/models/academic_year.py
$ git add base/models/academic_calendar.py
$ git add base/models/__init__.py
$ git commit -m "New entities added."
```

Committing often is encouraged. All commits are done locally, thus there is no risk of conflicts until all commits are sent to the server. The push option sends all commits in a local branch to the server, identified by `origin`.

```
$ git push origin issue#234
```

## Branching to Fix a Bug

### Fixing Mistakes

Version control doesn't always happens smoothly. We will certainly face some problems and fortunately Git is very gentle on which concerns recovering from mistakes. These are some common situations we may face during development.

## Moving to another branch before finishing the work in the current branch

Sometimes we are working in a branch and a more urgent problem arrives, requiring us to move to or create another branch. In this case, we have to commit all changes in the current branch before moving to another one, otherwise we risk to have our changes to the current branch commi## Following the Git Workflow

The code repository is organized in three fixed branches:

- *dev*: agregates developers' contributions that are intended to be in production, but they still need to be validated.
- *qa*: at the end of the sprint, when all features are frozen, the branch *dev* is merged into *qa* to allow testers to validate the release before it gets into production.

- *master*: once the version in *qa* is fully validated, it is merged into the branch *master*, which is the one to be deployed in production.

Developers should not commit directly to any of these branches. By convention, these branches can only be changed if there is an issue in the <https://github.com/uclouvain/osis/issues> [issue tracking tool] that justifies the creation of an exclusive branch for that issue. For instance: if the issue's number is *#234* then its respective branch is named *issue#234*, created to isolate the changes described in the issue. To create a new branch for the issue, perform the following commands:

```
$ git checkout dev
$ git pull origin dev
$ git checkout -b issue#234
```

The first command enters in the branch *dev*, if the developer is not already in there. Within the branch *dev*, the latest commits in the remote branch *origin/dev* are downloaded and merged within the local branch *dev*. Then, the new branch *issue#234* is created from the local branch *dev*.

The developer in charge of the issue *#234* changes the code within the branch *issue#234*. Two commands are very useful to keep track of what has been done:

```
$ git status
$ git diff models.py
```

The first command shows all created, modified, removed and untracked files that are candidates to be committed. The second shows the changes in one of the modified files. When we are ready to commit, we should decide whether all changed files will be included in the commit or just a subset of them. To include all files:

```
$ git commit -a -m "New entities added."
```

To include a subset of files, we have to add each file individually:

```
$ git add base/models/academic_year.py
$ git add base/models/academic_calendar.py
$ git add base/models/__init__.py
$ git commit -m "New entities added."
```

Committing often is encouraged. All commits are done locally, thus there is no risk of conflicts until all commits are sent to the server. The push option sends all commits in a local branch to the server, identified by `origin`.

```
$ git push origin issue#234
```

## Branching to Fix a Bug

### Fixing Mistakes

Version control doesn't always happens smoothly. We will certainly face some problems and fortunately Git is very gentle on which concerns recovering from mistakes. These are some common situations we may face during development.

### Moving to another branch before finishing the work in the current branch

Sometimes we are working in a branch and a more urgent problem arrives, requiring us to move to or create another branch. In this case, we have to commit all changes in the current branch before moving to another one, otherwise we risk to have our changes to the current branch committed in another branch. So, first add your changes and commit:

```
$ git commit -a -m "New entities added but still incomplete."
```

and then move to another branch:

```
$ git checkout issue#261
```

or create another branch from dev:

```
$ git checkout dev
$ git pull origin dev
$ git checkout -b issue#262
```

It also happens that we start fixing an issue (#262) but we forget to checkout its respective branch (issue#262). In this case, we have to commit only the files related to the current branch and leave in the workspace the changes related to another branch:

```
$ git add quick_sort.py
$ git commit -m "Sort algorithm started."
$ git checkout issue#260
```

The files that were not committed in the previous branch will be available for commit in the branch issue#260.

This practical approach of moving from a branch to another while leaving some files in the workspace may not work if at least one of the files we have left in the workspace was also changed in the branch issue#260. We may see a message like this:

```
From https://github.com/uclouvain/osis
* branch      dev      -> FETCH_HEAD
Updating 57c4a6d..9839a25
error: Your local changes to the following files would be overwritten
      by merge:
      __openerp__.py
Please, commit your changes or stash them before you can merge.
Aborting
```

In this case, we have to commit local changes before moving to another branch, as we first explained. However, things can get worse when the current branch is related to a closed issue, thus committing to it doesn't make sense anymore. In this case, we can use `git stash`. It moves all changes in the current workspace to a transit area that can be recovered later on. To move all changes to the stash area, simply type:

```
$ git stash
```

Now, if we type `git status` we find the working directory clean, which means we can move to another branch. To see the stashes we have stored we can use:

```
$ git stash list
```

After moving to another branch, we can recover the changes from the stash using:

```
$ git stash apply
```

but if there is more than one stash in the list we can apply a specific one by referencing its identifier:

```
$ git stash apply stash@{2}
```

## Fixing the latest commit message

```
$ git commit --amend -m "message"
```

When we work with branches it's very common to fool around with commits. There are many branches locally and sometimes we forget to switch to the proper branch and we end up committing on the wrong branch. When it happens before pushing the commits to the server, we can undo the last commit done with the command:

```
$ git reset --soft HEAD~1
```

But if the commit was already pushed to the server, it is still possible to undo the push as long as other people have not pushed to the same branch after the wrong push. So, after resetting local branches to the desired state, we can force a push to overwrite the remote branch:

```
$ git push origin dev -f
```

It is also possible undo remote modifications by reverting commits. Git figures out how to undo the changes introduced by the commit and appends a new commit with the resulting content. We first update our local branch, perform the revert and push back to the origin:

```
$ git checkout dev
$ git pull origin dev
$ git revert 7b4ff2cc82db88d1f5778babdcefae08266b37d0
$ git push origin dev
```

Stop tracking a file without deleting it locally:

```
$ git rm --cached [file]
```

Deleting remote branches:

```
$ git push origin --delete issue#530
```

tted in another branch. So, first add your changes and commit:

```
$ git commit -a -m "New entities added but still incomplete."
```

and then move to another branch:

```
$ git checkout issue#261
```

or create another branch from dev:

```
$ git checkout dev
$ git pull origin dev
$ git checkout -b issue#262
```

It also happens that we start fixing an issue (#262) but we forget to checkout its respective branch (issue#262). In this case, we have to commit only the files related to the current branch and leave in the workspace the changes related to another branch:

```
$ git add quick_sort.py
$ git commit -m "Sort algorithm started."
$ git checkout issue#260
```

The files that were not committed in the previous branch will be available for commit in the branch issue#260.

This practical approach of moving from a branch to another while leaving some files in the workspace may not work if at least one of the files we have left in the workspace was also changed in the branch issue#260. We may see a message like this:

```
From https://github.com/uclouvain/osis
* branch          dev          -> FETCH_HEAD
Updating 57c4a6d..9839a25
error: Your local changes to the following files would be overwritten
      by merge:
      __openerp__.py
Please, commit your changes or stash them before you can merge.
Aborting
```

In this case, we have to commit local changes before moving to another branch, as we first explained. However, things can get worse when the current branch is related to a closed issue, thus committing to it doesn't make sense anymore. In this case, we can use `git stash`. It moves all changes in the current workspace to a transit area that can be recovered later on. To move all changes to the stash area, simply type:

```
$ git stash
```

Now, if we type `git status` we find the working directory clean, which means we can move to another branch. To see the stashes we have stored we can use:

```
$ git stash list
```

After moving to another branch, we can recover the changes from the stash using:

```
$ git stash apply
```

but if there is more than one stash in the list we can apply a specific one by referencing its identifier:

```
$ git stash apply stash@{2}
```

## Fixing the latest commit message

```
$ git commit --amend -m "message"
```

When we work with branches it's very common to fool around with commits. There are many branches locally and sometimes we forget to switch to the proper branch and we end up committing on the wrong branch. When it happens before pushing the commits to the server, we can undo the last commit done with the command:



```
$ git reset --soft HEAD~1
```

But if the commit was already pushed to the server, it is still possible to undo the push as long as other people have not pushed to the same branch after the wrong push. So, after resetting local branches to the desired state, we can force a push to overwrite the remote branch:

```
$ git push origin dev -f
```

It is also possible undo remote modifications by reverting commits. Git figures out how to undo the changes introduced by the commit and appends a new commit with the resulting content. We first update our local branch, perform the revert and push back to the origin:

```
$ git checkout dev  
$ git pull origin dev  
$ git revert 7b4ff2cc82db88d1f5778babdcefae08266b37d0  
$ git push origin dev
```

Stop tracking a file without deleting it locally:

```
$ git rm --cached [file]
```

Deleting remote branches:

```
$ git push origin --delete issue#530
```

# Operations

# Maintenance

When a maintenance window is planned for OSIS, it is possible to prevent users about this event directly on the user interface, so they can plan their usage according to the application's availability.

The **Application Notice** is available in the administration and allows the administrator to create temporary notices to be shown in all screens of OSIS. This way, all active users are notified during the period the notice is set to be shown. To create or maintain notices, go to the *Administration* > section *Base* > subsection *Application Notices*.

OSIS

BIENVENUE, HILDEBERTO. [VOIR LE SITE](#) / [DÉCONNEXION](#)

Accueil > Base > Application notices > Ajouter application notice

Ajout application notice

Subject :

Mise à jour de sécurité


Notice :

Une mise à jour de sécurité est nécessaire sur tous les serveurs OSIS, il risque donc d'y avoir une courte coupure si vous êtes connectés.

Start publish :


Date :

04/05/2016

Aujourd'hui 

Heure :


08:11:52

Maintenant 

Stop publish :


Date :

06/05/2016

Aujourd'hui 

Heure :

08:11:56

Maintenant 

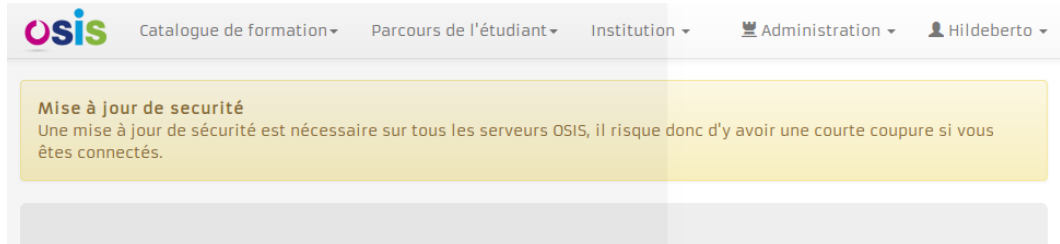
Enregistrer et ajouter un nouveau

Enregistrer et continuer les modifications

ENREGISTRER

Application Notice form

In the form, define a subject, a short notice, and the period in which the notice should become visible. The result is shown in the following image.



Application Notice shown in the user interface

Within the publishing period, the notice appears right below the main menu and on top of the main content.

It is recommended to create a new notice for every event, instead of changing existing ones. This way, we preserve the history, which may help to make decisions in the future. But nothing blocks you to use the same notice over and over again.