CHAPTER

ONE

DEVELOPERS' GUIDE

1.1 Starting development

This guide will serve as a basic introduction to installing Magik for development purposes. The steps have been tested on Linux and Mac OS X.

Before starting the development process you need to have Python installed on your system. Also Mercurial, a version control system, is required to get the source code.

1.1.1 Setting up Mercurial

To use Mercurial you need to set up your username.

By default Mercurial uses a username of the form `user@localhost` for commits. This is often meaningless. It's best to configure a proper email address in `~/.hgrc` (or on a Windows system in `%USERPROFILE%\Mercurial.ini`) by adding lines such as the following [1]:

```
[ui]
username = John Doe <john@example.com>
```

For more detail refer to the Mercurial wiki, Mercurial: The Definitive Guide or MoinMoin Mercurial guide

The Hg tip has handy tips for Mercurial users.

1.1.2 Getting the source code

To get sources you need to clone the main repository:

```
hg clone https://magik.inf.unibz.it/hg/magik/
```

The command will copy the source code from the server to your local computer to folder *./magik* with the following structure:

```
./magik/
-- bootstrap.py
-- buildout.cfg
-- conftest.py
-- doc
-- magik
```

[1] http://mercurial.selenic.com/wiki/QuickStart#Setting_a_username

```
-- setup.cfg
-- setup.py
```

- `bootstrap.py` and `buildout.py` are files required by Buildout to build the development environment.
- `conftest.py` configures the testing toolkit.
- `doc/` contains package documentation.
- `magik/` is, actually, the source code of the program.
- `setup.cfg` and `setup.py` are files required to cread a Python egg.

1.1.3 Buildout

Buildout Buildout is a Python-based build system for creating, assembling and deploying applications.

It is used for automatic project deployment and dependency installation in an isolated environment.

First of all we need to "cd" to the folder with repository, bootstrap the project, and buildout it:

```
cd magik/
python bootstrap.py
bin/buildout
```

Note: You must have at least Python 2.6 to run Magik.

To specify which Python should be used, run the bootstrap script with it.

```
python2.6 bootstrap.py # Use Python 2.6
/usr/bin/python bootstrap.py # Explicitly define python interpretator.
```

Bootstrapping creates needed directories and generates the `buildout` script, which downloads all the *Magik* dependencies and places command-line scripts to the `bin/` directory.

The most important executables in the `bin/` directory are:

- `bin/buildout`
- `bin/bpython` bpython is a fancy interface to the Python interpreter.
- `bin/py.test` py.test is a command line tool to collect, run and report about automated tests. It is a part of the py library.

You can find more information about buildout at Buildout documentation or A brief introduction to Buildout.

1.2 Developing Magik

1.2.1 Running tests

`py.test` is a testing toolkit that is used in *Magik* to run unit tests. To execute all of them them run:

```
bin/py.test magik
```

In this case the toolkit will run unit tests of the `magik` module and doctests in the `doc/` directory.

You can specify which test to run by providing a directory or a file path which contains test fiels to `py.test`. For example:

```
bin/py.test magik # Run unit tests
bin/py.test doc # Run doctests
bin/py.test magik/statement/_tests/test_select_parser.py # Run tests defined in magik/st
```

Note: Sometimes it is handy to share the output of a test run.

py.test can submit its output to a pastebin, a hosting for text snippets. To send your output run:

```
bin/py.test magik --pastebin=all
```

There is another test runner: `bin/py.test-friends`. It additionally to unit and doc tests performs the **PEP 8** check and static analysis by pyflakes. To execute these additional tests run:

```
bin/py.test-friends
```

It is also possible to check test coverage. Run:

```
bin/py.test --cover=magik --cover-report=html
```

File `coverage/index.html` contains the coverage report in the HTML format.

See to `bin/py.test --help` for more details.

1.2.2 Making changes

Coding standarts

In general, `Magik` is written with **PEP 8** in mind.

- Lines longer then 79 characters are fine.
- The last line of a file is a blank line.
- `some_string.format(...)` is preferred to `some_string % ...`.

Todo

pep8

Actually, there still some work needs to be done to respect pep8 more.

Read MoinMoin coding style to get inspiration.

Todo

codereview

It would be nice to setup codereview at http://magik.inf.unibz.it/rb

Writing tests

Unit tests is an important part of the project, they help to ensure that the program works as was ment by the programmer.

As a rule of thumb, try not to run the code manually to see if the feature you have written works, but write tests that implement such a run. In other words, after running the tests, you should pretty sure that the code works well.

Test are located in the `_tests` directory of the module. For example, tests that test the `magik.statement` module are located in `magik.statement._tests`. Inside the test module (or directory) an `__init__.py` file should be located, and files that contains test classes named `test_*.py`. Test class names should start with `Test` and test methods with `test_`.

Committing changes

Once a feature is implemented, code changes have to be committed.

> **Warning:** Make sure that all test are passed before making a commit.

If you have created new files, they needed to be added to the version control. Use `hg add file1 file1` command to do so.

`hg st` shows the status of the repository: what files were changed, removed, deleted, added and so on.

`hg diff` shows which content of files was removed, added or changed.

Note: To make output of Mercurial more readable, the *color* extension can be enabled.

To enable it ad to the `extensions` sections of your `.hgrc` line `color =`. For example:

```
[extensions]
color =
```

`hg st` and `hg diff` are very handy in finding out what have been changed, what files are missing (were not added to the repository). Try to make commits that refer to one thing: it can be either a bugfix, a formatting fix, a new feature, a new unit test, or anything else, what is important that it is one item, not several.

Having atomic commits is important in teams, where team members need to track how the source code changes.

Once you prepared the code base for a new commit, run `hg ci` to make a commit to the local repository. In general, it better to make small, often commits than rare and big ones.

To share your changes wit the world, you need to push your changes either to the main repository, or to any other repository. `hg push` sends changes sets (commits) to other repository. `hg pull` gets them from other repository.

`hg in` shows commits that you are missing, `hg out` shows commits that other repository is missing.

`hg up` allows to switch to desired commit. The most recent commit has label *tip*. Refer to the Mercurial documentation for more details.

1.3 Writing documentation

Documentation is located in the `doc/` directory. Read Sphinx documentation for more details.