Maymyo Documentation

Release 0.3.2-beta

ZimSpring Research (SA0225224-K)

September 17, 2012

CONTENTS

Contents:

CHAPTER

ONE

WHAT IS MAYMYO

Maymyo, named after a Myanmarese hill station, is a Django application providing basic infrastructural services for enterprise applications. You can build your own Django application that focuses on your particular industry vertical quicker by using Maymyo to deliver a complete enterprise application.

These infrastructural services are generic to most enterprise applications :-

- *Supporting enterprises with a multi-Business Unit hierachical organisation*
- *Application Security and Access Control*
- *Calendar and business day computations*
- *Auditing of changes to Models*
- *Automatic Filtering of accessible rows by Users*
- *Messaging and Alerts* with optional email or sms delivery
- *Generate Reminders with optional escalation (to superiors)* based on Model state over time
- *General purpose dynamic formulas, conditions, cursors, filters and choices*
- *Background Task Queue for tasks and reports*
- *Job definition and scheduling with business day semantics*
- *Automatic Output spooling to printers and archiving of spool files*
- *Automatic periodic archiving and purging of your Models*
- *Designer defined Application Events and its delivery*
- Designer defined Transaction Selection with Business Rule authorisation
- Workflow engine for your Document approvals
- Reporting Engines to execute Text or JasperReport reports
- Option to use JasperServer as a standalone reporting server
- Charge Computation Engine
- Billing Module

1.1 Who it is for

Maymyo will be useful to experienced Django developers building applications for enterprises. These are internal applications not meant for the public. So if you are looking for tools to build the next billion dollar social application,

keep looking. We are targeting tens to hundreds on users, not millions. However, you can still allow your customers access to self-service applications provided by your organisation.

We would like to encourage developers working within organisations to use Maymyo for their internal projects. We believe that the features of Maymyo would both help you complete your project faster and with less risk.

If you are new to Django, do not despair. You will be proficient in no time by going through Django's tutorial in their excellent documentation.

1.2 Licence

Similar to Django, Maymyo is licensed under Berkeley Software Distribution (BSD). What this means is you can develop your application using Maymyo and distribute your product under closed or open source licence.

1.3 Software redistributed by Maymyo

We would like to acknowledge the contributions of the authors of the following software for making Maymyo possible :-

- Dojo, version 1.6.1, BSD Licence
- jQuery, MIT Licence
- Dojango, version 0.5.2, BSD Licence
- Django Piston, version 0.2.2.1, BSD Licence
- Oxygen icons, for their well polished icons, Creative Common Licence
- **django, dojo or python snippets "borrowed" from the following :-**
  - `YC Loh`, who shared code from his Django project with us
  - Erdem Agaoglu, web services client to JasperServer, Apache Licence
  - Daniel Roseman, Multi-Select, Django snippet
  - Michael Halle, build query from a nested list, Django snippet
  - Louis Riviere, Windows Services helper, an ActiveState Recipe
  - Dan Fairs, for trick to get parent object for inlines
  - Paul Rogalinski, unclosable dijit Dialog
  - Chris Hardin, Dojo idle timeout
  - Apologies to anyone we have inadvertently omitted. Please let us know.

1.4 Steps to take

Briefy, these are the steps to take to quickly start using Maymyo :-

- ***Install Maymyo*** **(and its pre-requisites)**
  - install python, django plus a few required packages
  - install a database and its required python driver.

  - install Maymyo
  - install a Java Runtime Environment (to run JasperReport reports) and jaspercommand (jar files to run JasperReport on the command line). You can defer this until you want to run JasperReport reports
  - auto-start Maymyo daemons in the Operating System You can defer this, just start the daemons manually

- **Create your own application using Maymyo**
  - design your application's models
  - design and develop your programs and reports
  - your models and programs should make use of the services provided by Maymyo.
  - add your programs and reports to Maymyo
- **Add your application's menu(s) under Maymyo**
  - menu items are your menus, programs and reports
- Add your own user groups and users to access your application
- **Create your own Commands for application's house-keeping tasks**
  - create a Job (with Steps and Tasks) that run your Commands and Reports
  - automate the Job execution using Maymyo's Scheduler

The above will allow your application to be accessed by your users through Maymyo's dashboard and automate its housekeeping Jobs. You can *get started* right away, after you have *installed Maymyo*.

There are, of course, more things you can do with Maymyo. Please have a look at *explore and understand Maymyo*, for more in depth ways of using it.

CHAPTER

TWO

HOW TO INSTALL MAYMYO

If you are installing Maymyo to evaluate, then you need to :-

- *Install software packages required to run Maymyo*
- *Download and Install Maymyo*
- *Configure your database and environment*
- *Create and Load Maymyo Database objects*
- *Run Maymyo*

If you want to run our bundled JasperReport reports, then continue with :-

- *Download and Install Java Runtime Environment*
- *Download and Install jaspercommand*

If you want to develop your own JasperReport reports, then :-

- *Download and Install iReport plus Apache Ant*

If you are setting up Maymyo for production, these are the additional actions :-

- *Configure and auto-start Maymyo Daemons*
- *Configure Maymyo for Apache mod_wsgi*

2.1 Install software packages required to run Maymyo

Maymyo is a Django web application that uses a relational database to store its data. Django is a python application and Maymyo uses a few third party python packages.

We will not attempt to repeat the installation instructions of these software packages but we will give you helpful hints where applicable. You should consider yourself ready to install Maymyo only after you have managed to install and run these pre-requisite software packages :-

- **python, version >= 2.5 (2.7 preferred) but not 3 (yet) and** install the following packages:-

> **Hint:** We prefer to download the binary installer for our specific Operating System and Architecture, eg on Windows 64 bit, we would download the Windows Python 2.7 x86-64 Installer. For Debian-based Linux, we prefer to use `apt-get`, eg `sudo apt-get install python2.7` on Ubuntu.

  - setuptools, version >= 0.6. Download its zip or tgz (tar gzip) file, extract to a temporary directory, open a Command Prompt and change to that temporary directory and enter "python setup.py install". On Ubuntu or other Debian-based systems, we would use `sudo apt-get install python-setuptools`.

Hint: If you are installing on Windows 32 bit, just download the installer for your python version and run it.

Hint: If you are installing for the Windows 64 bit, then follow the instructions here. Right-click on the ez_setup.py link and save the link in a working directory. Open a Command Prompt in that working directory and run "python ez_setup.py". This will perform the necessary installation for you. You have to do this because the downloads provide only the 32 bit installer.

Hint: Use easy_install to install python modules *after* installing setuptools.

You can easily install other python modules by entering:

```
> easy_install module_name
```

on the command line. On Windows, ensure that your PATH environment variable has python's Scripts directory. (eg on Windows, `PATH=C:\\Python27;C:\\Python27\\Scripts;....;`) This is where easy_install.exe lives. On Debian-based Linux, you need to use:

```
$ sudo easy_install module_name
```

instead. You will be prompted for the your password.

  - pytz, for timezone handling. Use `easy_install pytz`
  - pyDes, encryption module. Use `easy_install pydes`

- Django, version >= 1.4, because we are using its backward incompatible features. We can use `easy_install django` to install it. Earlier versions will not work.

Note: If you have an earlier version of django installed, you can remove it by deleting the django directory from your Python's *site-packages*. Alternatively, you may choose to use *virtualenv*.

- Databases, you may choose one of the following:-
  - postgresql, version 9 onwards with psycopg2, its python database interface (DBI) driver.
  - mysql, version 5.0.3 onwards with mysql python, version 1.2.3 onwards

    ---

    Hint: On Debian-based Linux, Install postgresql and mysql above by:

    ```
    sudo apt-get install postgresql
    sudo apt-get install python-psycopg2
    sudo apt-get install mysql-server
    sudo apt-get install python-mysqldb
    ```

  - sqlite3 - as at django version 1.4, we have a "Too many SQL variables" error during syncdb. Furthermore, our JasperReport runtime cannot connect to it too. We are not supporting sqlite3 at the moment until we sort these out.

- Please refer to Django's documentation for other supported databases.

2.2 Download and Install Maymyo

You may download the latest Maymyo release file at Maymyo's project home and extract it to a directory that you plan to run in. The release file will have its version number as the root directory, eg 0.3.0. You may rename it.

Note: If you do not want to pollute your environment with our required environment variables, then create a new user and extract the release file in its home directory.

---

on Windows, you may extract to `C:\work`. Then rename the version number to *maymyo*. In this case, your application home directory will be `C:\work\maymyo`. All further steps thereafter will be relative to this Application Home directory. We will refer to this as *maymyo_home*.

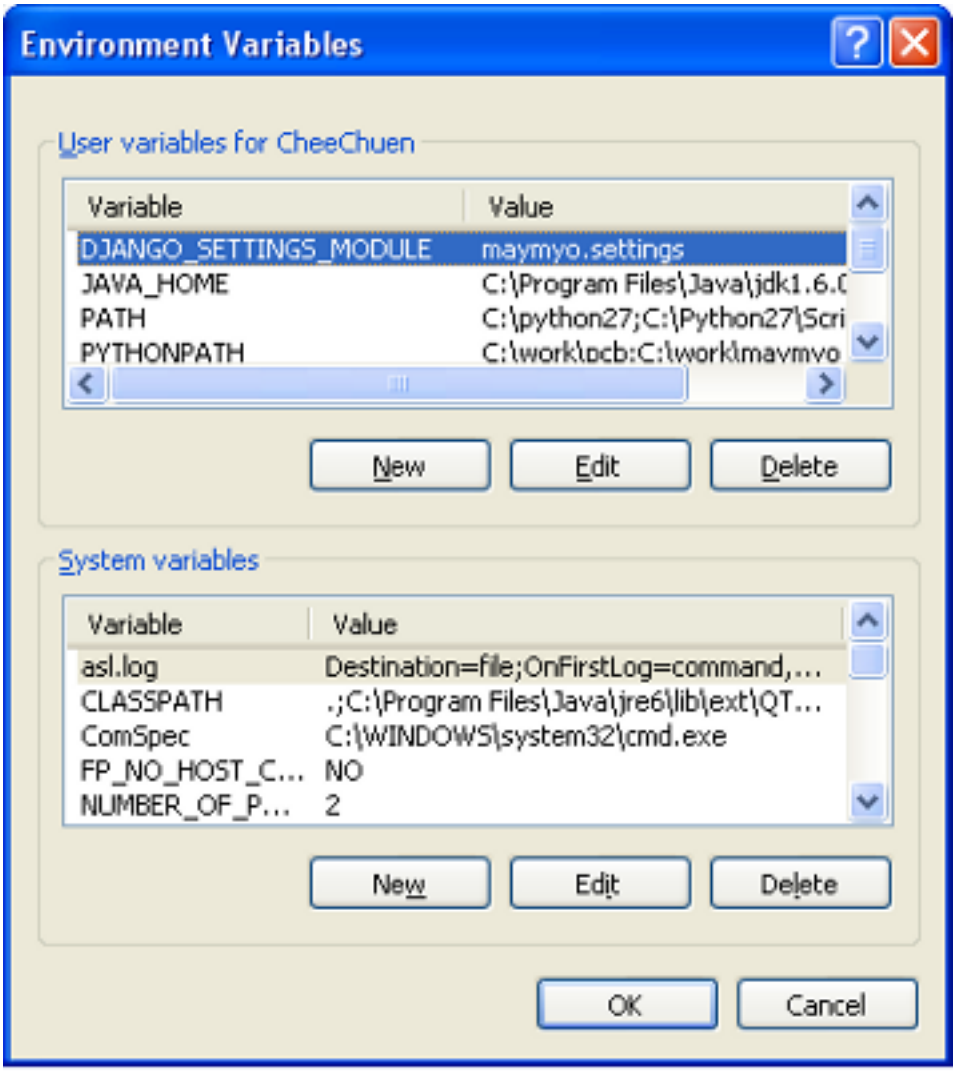
If you plan to add your own application, it will be a sibling directory to Maymyo's applications, eg `C:\work\maymyo\myapp`.

2.3 Configure your database and environment

2.3.1 The first thing to do is to prepare your database.

- We assume you know how to administer a database, either using their command line tool (the hard way) or their GUI tool.
- for postgresql, create a user, database and schema, eg all named *maymyo*. Make the user *maymyo* own both the database and schema. Use pgAdmin III GUI tool for this.
- for mysql, create a database and user, eg all named *maymyo*. Grant all permissions on the database to this user. Use mysql's Administrator GUI tool for this.
- please refer to your database manuals on how to perform the above.
- Django will later create the required database tables automatically. But you need to provide it with a user to connect to the database (in its *settings.py* file).

2.3.2 Then edit your environment variables.



Changing environment variables on Windows. To get to the screen above, right-click on "My Computer" and select "Properties". Then choose the "Advanced" tab and click on "Environment Variables" button somewhere near the bottom.

Edit or Add the following User environment variables:-

- DJANGO_SETTINGS_MODULE = maymyo.settings
- PYTHONPATH = *maymyo_home* eg on Windows, set `PYTHONPATH=C:\work\maymyo`
- add to PATH your Python installation directories and *maymyo_home*/bin eg on Windows, add `C:\Python27;C:\Python27\Scripts;C:\work\maymyo\bin;` to your PATH.

Note: On Ubuntu, you should create a user *maymyo* whose home directory is where Maymyo will reside. Edit the file ".profile" in this home directory. For other unixes, this may be ".bash_profile", depending on the user's shell. A user's default PATH would already include `/usr/bin:/usr/local/bin` where python, easy_install and django-admin.py is installed. So you would just need to add your *maymyo_home*/bin to PATH. Edit the .profile file in Maymyo's home directory. It would also be a good idea to add "." (current directory) to PATH. Saves you the hassle of typing "./". Your changes should look

like this:

```
export DJANGO_SETTINGS_MODULE=maymyo.settings
export PYTHONPATH=maymyo_home
export PATH=.:maymyo_home/bin:$PATH
```

You should logout and login again for the new environment variables to take effect. After re-login, verify that the new environment variables are in effect by, on Windows:

```
> echo %PYTHONPATH%
```

or on Unix platforms:

```
$ echo $PYTHONPATH
```

The output should display your *maymyo_home*. If it is empty or wrong, please repeat the above.

2.3.3 Edit your Maymyo config files to reflect your installation specific values.

1. Edit DATABASES in *settings.py* file in *maymyo_home*/maymyo for your database choice:

```
DATABASES = {
    'default': {
        'NAME': 'maymyo',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        # 'ENGINE': 'django.db.backends.mysql',
        'USER': 'maymyo',
        'PASSWORD': 'maymyo',
        'HOST': '',
        'PORT': '',
    }
}
```

2. Note that NAME above refers to the database name you have created.
3. Uncomment the ENGINE you are using and comment out the one you are not using. ie in the above example, we are using postgresql and not mysql.
4. Set PASSWORD to whatever you used when preparing your database.
5. Leave HOST and PORT blank if you are using *localhost* and the database's default port respectively.
6. Edit RUN_HOME in *settings.py*. This is where Maymyo will send outputs to during runtime. You should choose a separate directory from *maymyo_home*:

   ```
   RUN_HOME = os.path.join(os.path.dirname(APP_HOME), 'mayrun')
   ```

   The above will make the run home directory *mayrun* a sibling of your *maymyo_home*. You can also specify an absolute directory path but ensure you have double back-slashes for Windows.

2.3.4 Prepare your Application Directories and Files

We need to ensure that directory permissions and executable modes are okay.

- goto the *maymyo_home* /bin directory and:

  ```
  $ python prepare_app.py
  ```

- the above will collect all required static files, create the run home directory you specified earler and change the executable modes of our scripts in the *bin* directory.

2.4 Create and Load Maymyo Database objects

1. You should perform a check on your database connection. Open a new Terminal (or Command Prompt on Windows) window:

   ```
   > django-admin.py validate
   ```

   Note: We prefer to use django-admin.py instead of manage.py because it is in the PATH. You can still use manage.py as long as your current directory "." is in the PATH. In this case, use "manage.py validate" instead while you are in *maymyo_home*.

   ---

2. Create the database objects for Django and Maymyo:

   ```
   > django-admin.py syncdb
   ```

3. You will be prompted to create the django superuser and set its password. You must use "admin" as the user name because our fixtures data use this. You will need this to login to Maymyo later.

4. Change directory to your *maymyo_home*. Load Maymyo's standard fixtures data for its applications:

   ```
   > load_fixtures.py
   ```

   You will see a lot of output messages telling you what has been created. *fixtures* is a Django term describing the initial setup data to be loaded into database tables.

   Note: Do not worry if you see "Language ms is not supported". This is because we have not found the the people to volunteer to translate Malay (national language of the authors) for Django. Besides, we have not translated Maymyo to Chinese, so when you choose it on log in, you will only see the Django translations and Maymyo will still be in English.

   ---

5. If you want to load our test data. Skip this if you are setting up for production.:

   ```
   > load_fixtures.py testdata
   ```

2.5 Run Maymyo

1. We need to change the environment for the Task Queue, Output Queue and Reporting engines if you are running on unix-based platforms (because the fixtures defaults to Windows), open a Terminal and do this:

   ```
   > prepare_env.py
   ```

2. You are now ready to run Maymyo and add your own application:

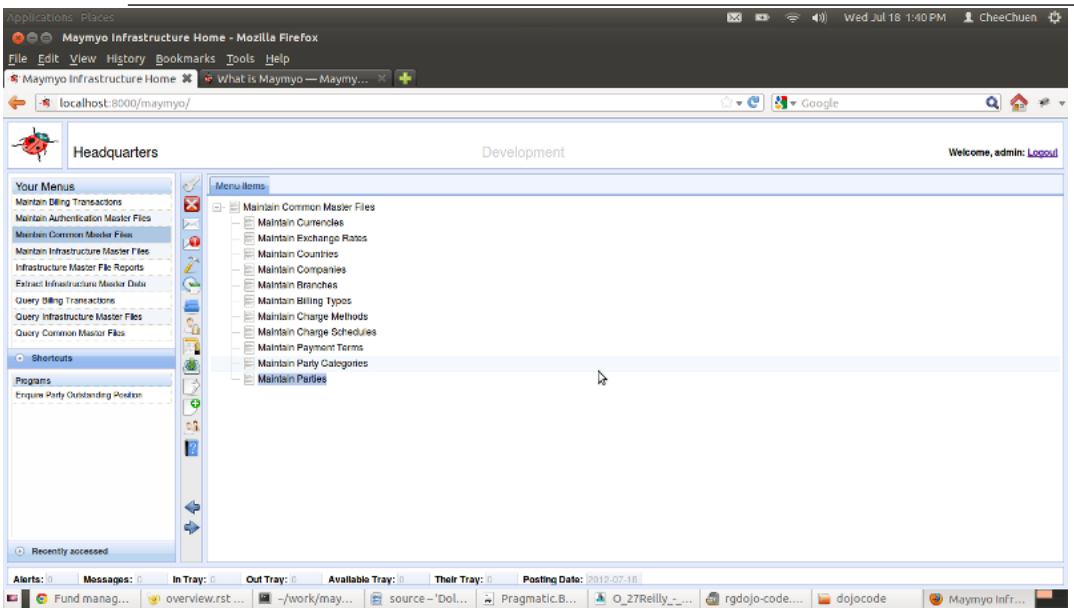
   ```
   > django-admin.py runserver
   ```

3. Open your browser and connect to Maymyo with:

   ```
   http://localhost:8000/maymyo/
   ```

Login with the superuser (and password) you created earlier during syncdb, ie "admin". You should now be connected to Maymyo's Dashboard as shown below.

Note: The above URL assumes that you are logging in from the same computer as the one you installed Maymyo in. If not, please change the host name (ie localhost). If your firewall is on, remember to open up port 8000 for remote connections. You can run on another port by adding the port number after "runserver" above.



Your Dashboard on FireFox.

4. Change the Application Parameter Set used to define your installation's Environment Variables. Click on the Menu "Maintain Infrastructure Master Files" in the "Your Menus" list on the left panel of the Dashboard. Choose the Menu Item "Maintain Application Parameter Sets" and double-click. Find the Parameter Set "WINDOWS-ENV" (or "UNIX-ENV" for posix platforms) to change. This Parameter Set is used dynamically for Task Queues, Output Queues and Reporting Engines. You should change the *PATH*, *PYTHONPATH* and *JAVA_HOME* parameters to your installation's specific directories, similar to the above *environment variables*.

5. If you are just evaluating or setting up a development environment, you can start Maymyo daemons manually by opening a Command Prompt:

   ```
   > start_all_daemons.py
   ```

6. To stop the daemons.py:

   ```
   > stop_all_daemons.py
   ```

7. You can skip the above if you want to start and stop the daemons automatically with the Operating System. This is discussed next.

8. You can stop here if you want to evaluate Maymyo only, enjoy!

2.6 Installing Java Runtime Environment

If you want to run JasperReport reports, then you will need the Java Runtime and *jaspercommand*.

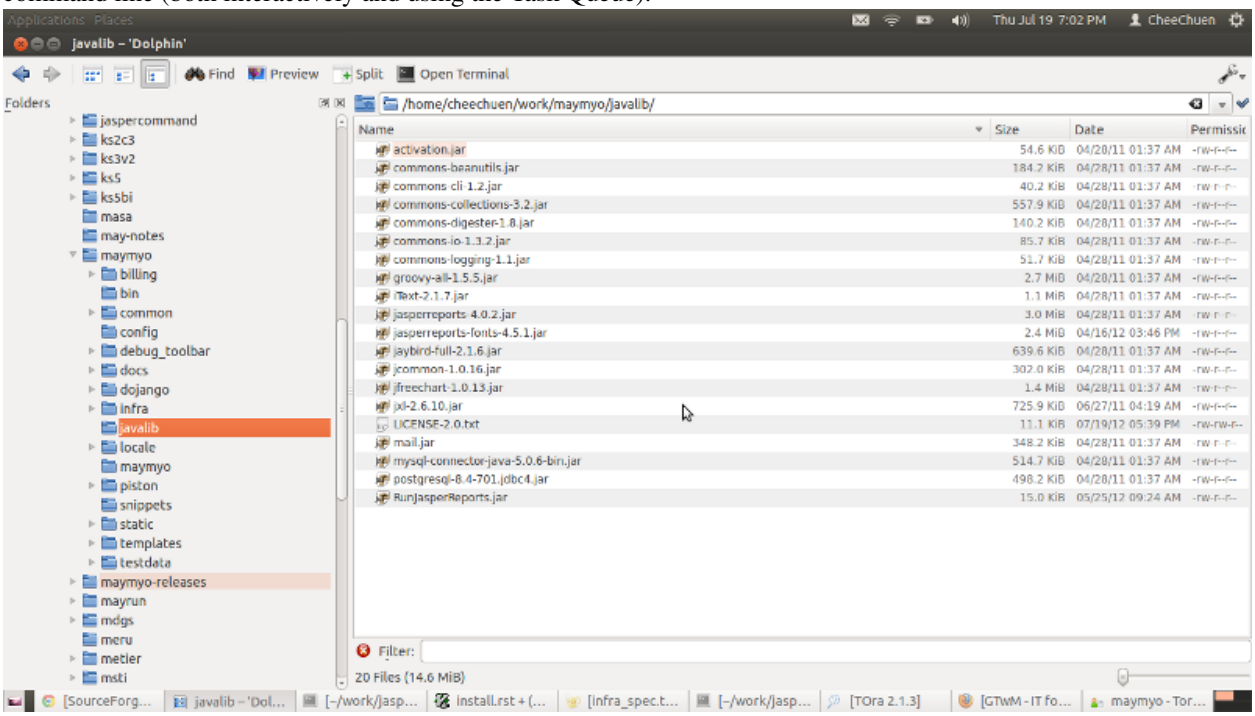
- Download and install Java Runtime Environment, version 1.5 onwards, for running JasperReport reports.

- On Ubuntu, you can use its *Ubuntu Software Center*, search for *java* and install the OpenJDK Java 7 Runtime.

2.7 Download and Install jaspercommand

You need this package to be able to run JasperReport reports on the command line. This package is a fork of GT webMarque Ltd's RunJasperReports. Download either the latest zip or tar compressed file (tgz) from jaspercommand home.

Extract the zip or tgz file to a working directory and copy the entire contents of its *javalib* directory to *maymyo_home*. That's all. Your new directory should look like below. It contains all the jar files needed to run JasperReport on the command line (both interactively and using the Task Queue).



jaspercommand's *javalib* destination

2.8 For developing JasperReport reports

iReport and apache ant are required for developing reports. If you prefer to use other reporting tools, then this is optional. You will however, need to write your own ReportingEngine to allow Maymyo to run your reports. Please refer to our *infra/custom/jasper_command.py* to see how we did it for JasperReport.

Java Runtime Environment and jaspercommand is a pre-requisite.

- iReport, when you want to create or modify JasperReport reports

  ---

  Note: Add Maymyo home directory to iReport's classpath.

  You should specify your reports' external resources (ie images, external styles, subreports) relative to Maymyo home.

- Apache ant, to compile your JasperReport design files in bulk.

Note: Ensure that the ant bin directory is in the PATH environment variable.

Not required if you prefer to use iReport to compile your reports.

2.9 Configure and auto-start Maymyo Daemons for Production

Maymyo Daemons are a set of programs that will run in the background to provide services like Task Queueing, Scheduler, Output Spooling and delivering external messages for your applications. This section shows you how to set them up for Production use.

Note: Skip this step if you are only evaluating or setting up a development environment.

You should edit the *maymyo_home*/snippets/envvars.ini file. Whether on Windows or Unix, the daemons need a user whose initial environment variables PATH and PYTHONPATH having *maymyo_home*/bin and *maymyo_home* respectively to bootstrap the program responsible for starting and stopping individual daemons. Then for the daemons themselves, the envvars.ini is used as their environment. This is where we place additional environment variables required by the daemons. This should be a superset of variables needed by the daemons.:

```
[windows]
# -------------------------------------------------------
# Windows, cannot use %VAR% because not expanded in Services
# Path first, so that we run our scripts.
PATH=C:\work\maymyo\bin;C:\Python27;C:\Python27\Scripts;C:\WINDOWS\system32;
C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\MySQL\MySQL Server 5.1\bin;
C:\Program Files\PostgreSQL\9.1\bin;C:\work\ant\bin;
# then Python path to import our python modules
PYTHONPATH=C:\work\maymyo;
# Let django know where is the settings file name
DJANGO_SETTINGS_MODULE=maymyo.settings
# To find Java
JAVA_HOME=C:\Program Files\Java\jre1.6.0_03;
# Options to pass to Java
JAVA_OPTS=-Xmx256m
# Windows only
SystemDrive=C:
# Where Windows were installed
SystemRoot=C:\Windows
```

The above shows an example on how we would configure the environment for Windows. We need the database bin directory in the PATH when connecting because we need to load their DLLs when running SQL scripts. Also note that you cannot use DOS variables (eg %PATH%) because they cannot be expanded when used in Windows Services.

Then optionally install the following :-

- pywin32, if your are on the Windows platform. Use `easy_install pywin32`
- skype4py, version 1.0.32 onwards, only when you want to send sms from Maymyo. One of the daemons called messenger.py does this.

2.9.1 On Windows, perform the following:-

1. Install the Maymyo Daemons as a Windows Service. Open a new Command Prompt:

```
> winservice.py install
```

2. Then run Services, on Windows XP this would be in *Control Panel->Administrative Tools>Services* Scroll down to find the "Maymyo *Development* Daemons..." Double-click this, then goto to the "Log On" tab and change to "Log On as" to your username. This is because the daemons need your environment variables to be started. You should change the "Startup type" too to "Automatic" so that the daemons will start up at boot time.

Note: *Development* above is the Instance name of Maymyo. You may have several instances of Maymyo on the same computer and we use this to tell them apart. Each instance has its own database (or schema). You can change the instance name by using "Maintain Application Registry". Look for the Registry Key "INSTANCE-NAME". Do this before you install the Windows Service above.

---

3. You may want to tell Windows that the Maymyo Daemons depends on your database service being started first. Run regedit and add a "Multi String Value" entry named "DependOnService" for `HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\Maymyo_Development_Daemons` The value to add is your exact Database Service name, eg "MySQL".

2.9.2 On Red Hat based Linux, perform the following:-

1. log in as root
2. `ln -s *maymyo_home*/bin/auto_daemons.sh /etc/init.d/maymyo`
3. make it auto-start by `chkconfig --level 345 maymyo on`
4. Maymyo daemons will auto-start on next reboot
5. You can start or stop manually by `service maymyo start` or `service maymyo stop`.

2.9.3 On Debian based Linux :-

1. log in with your usual administrator user
2. "sudo su -" to change to root user
3. `ln -s *maymyo_home*/bin/auto_daemons.sh /etc/init.d/maymyo`
4. make it auto-start by `update-rc.d maymyo defaults`
5. Maymyo daemons will auto-start on next reboot
6. You can start or stop manually by `sudo service maymyo start` or `sudo service maymyo stop`.

Note: Replace `*maymyo_home*` with your installation home directory.

For other posix platforms, the steps are similar but the directory and commands may not be exactly the same. At worse, you may run autostart.sh in /etc/rc.local.

There is a config file that controls the daemons. It is in *maymyo_home*/snippets/daemon.tab. The comments in this file explains what the fields are for. By default we run the daemons from 6am to 1am the next morning. This means that the daemons are sleeping from 1:01am to 5:59am. You should perform warm database backup tasks in this time window. If cold database backups are required, then you should use the Operating System's scheduler (eg cron) to stop and restart the daemons or incorporate these actions in your backup script.

2.10 Deploying Maymyo for Production on Apache+mod_wsgi

You should read Django's documentation on deployment on Apache plus mod_wsgi in "The Development Process->Deployment->WSGI Servers" section. Consult Django resources on production deployments on other web servers. What will work for Django will work for us. However read below for the extra static files you need to serve.

Note: Skip this step if you are only evaluating or setting up a development environment.

There are extra static files to serve :-

- collect all the static files needed into the settings.STATIC_ROOT directory. Note that this step has been done during the prepare Application Directories above. You will need to do this again if you have added your own static files or you have changed the static directory STATIC_ROOT location in settings.py, run the following:

```
$ django-admin.py collectstatic
```

- You will also need to serve our dojo files, which includes a few of our own Javascript files. These are in `dojango/dojo-media/release/1.6.1-dojango-optimized-with-dojo`. Below is our own changes to the Apache httpd.conf file to serve Maymyo:

```
# Django wsgi setup -- start
# Using / overrides the above doc root but we can't use
# any prefix as we will then need to change our urls.py
WSGIScriptAlias / "/home/maymyo/maymyo024/maymyo/wsgi.py"
WSGIPythonPath "/home/maymyo/maymyo024"

<Directory "/home/maymyo/maymyo024/maymyo">
<Files wsgi.py>
Order deny,allow
Allow from all
</Files>
</Directory>

# Django wsgi setup -- end

# Django static files setup -- start
# You need to run "django-admin collectstatic" to copy all apps' static files
# to STATIC_ROOT directory.
# For dojango, we need to allow only the release directories to be accessed.
# Alias /robots.txt "/home/maymyo/mayrun024/static/robots.txt"
Alias /favicon.ico "/home/maymyo/mayrun024/static/images/custlogo.png"

Alias /uploads/ "/home/maymyo/mayrun024/uploads/"
Alias /static/ "/home/maymyo/mayrun024/static/"
Alias /dojango/dojo-media/release/1.6.1-dojango-optimized-with-dojo/ "/home/maymyo/maymyo024/doj

<Directory "/home/maymyo/mayrun024/static">
Order deny,allow
Allow from all
</Directory>

<Directory "/home/maymyo/mayrun024/uploads">
Order deny,allow
Allow from all
</Directory>

<Directory "/home/maymyo/maymyo024/dojango/dojo-media/release/1.6.1-dojango-optimized-with-dojo"
```

```
Order deny,allow
Allow from all
</Directory>

# Django static files setup -- end
```

- the example above assumes that *maymyo_home* is in "/home/maymyo/maymyo024".

CHAPTER

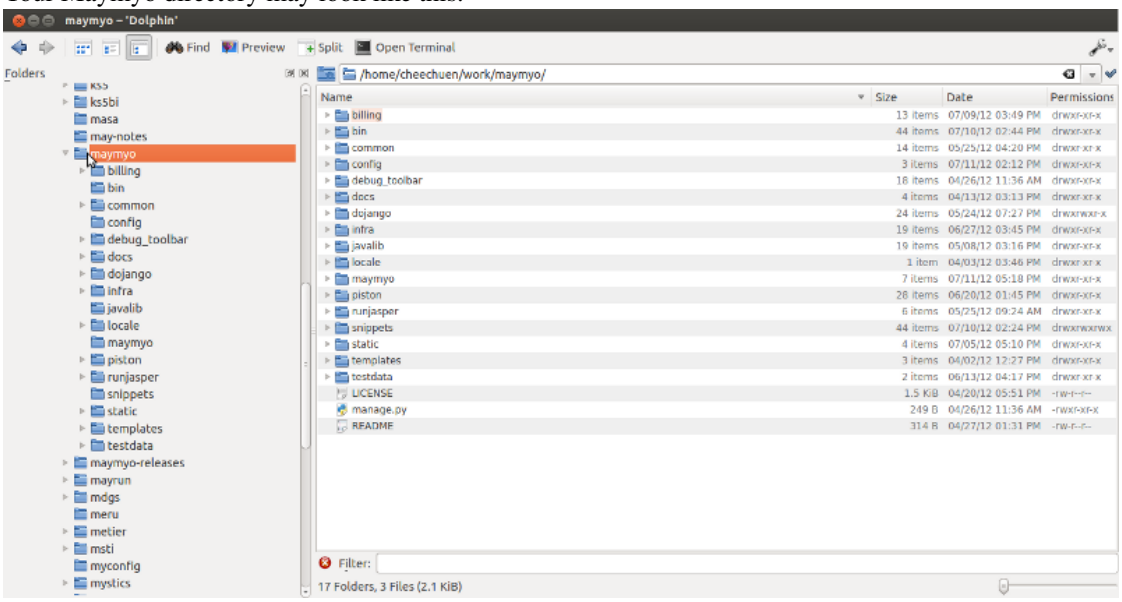
THREE

GETTING STARTED

After you have successfully *installed* Maymyo, you start by adding your own application under Maymyo. Your application will be a sibling of Maymyo's applications, ie infra, common or billing. You should use the services provided by Maymyo in your own application, eg use our Calendar when you need business day computations.

Please go through our *Explore and understand Maymyo* to learn how you may use these services.

Your Maymyo directory may look like this.



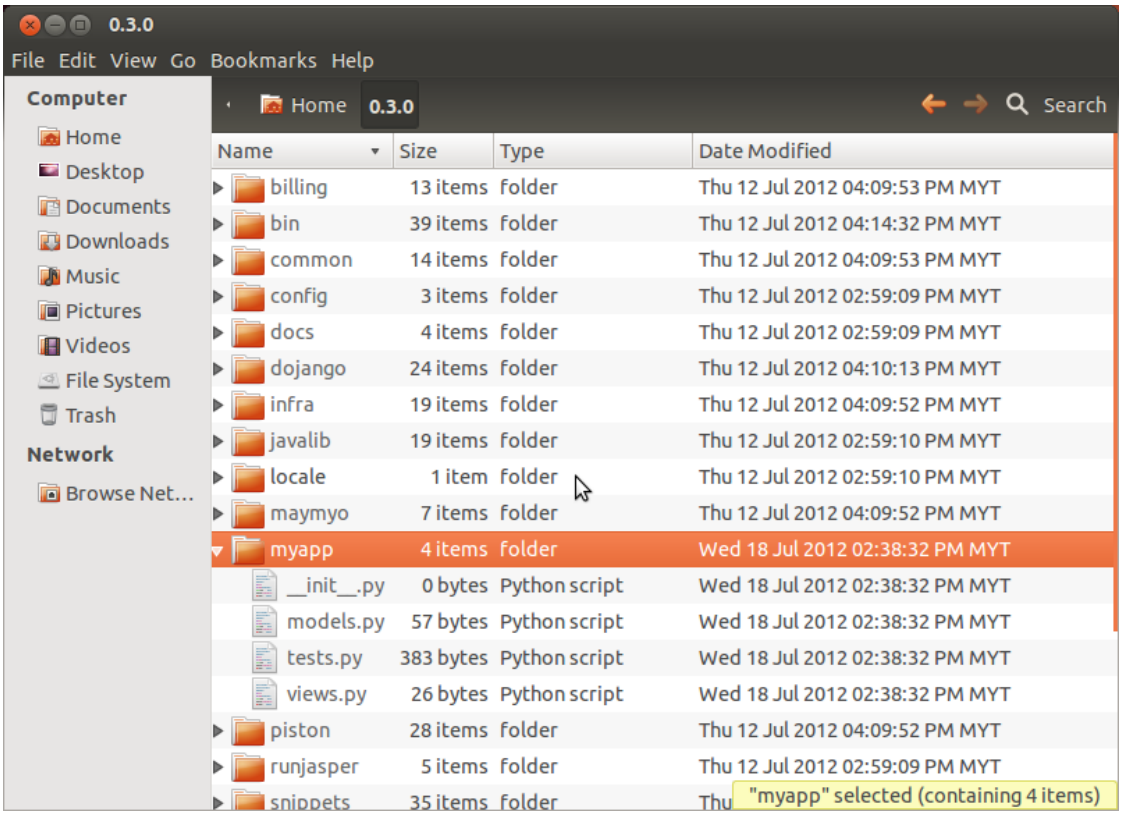
Directory layout of Maymyo

Open a terminal (or command prompt on Windows) and change to Maymyo home and type on the command line:

```
$ manage.py startapp myapp
```

Note: If you have problems running manage.py, make sure you are in Maymyo home and also ".", ie the current directory is in your PATH. Otherwise use "./manage.py ...".

This will create your application named "myapp" under the Maymyo home directory.



Directory layout of your new application

Then add your application to the *maymyo/settings.py* INSTALLED_APPS setting, right below our applications. Use your favorite text editor to edit the above settings.py. Find INSTALL_APPS and add the line 'myapp' like below:

```
INSTALLED_APPS = (
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.sites',
'django.contrib.messages',
'django.contrib.admin',
'django.contrib.staticfiles',
'django.contrib.humanize',
'dojango',                      # Dojango
'infra',                        # Our infrastructure app
'common',                       # Common Business Objects
'billing',                      # Billing Module
'myapp',                        # My new app!
)
```

If your application is small, ie just a few models, then you can leave the directory structure and files as is. Otherwise, we suggest that you create module directories for

- models
- forms
- admin
- views

under your myapp directory and create one file per model (and its children). Remove the original models.py and

views.py. *forms* and *admin* directories are our preferred way of organising our model forms and admin programs. This is better when you are using a source control application with a team of developers who will then not have conflicts accessing the same source file. Module directories means that they must have a special file named *__init__.py* to tell python that the directory is a module.

Start by creating your models (either in the models directory or models.py). You should inherit our RecordOwner model, which automatically adds some fields to keep track of who created and last updated a row (in database-speak or instance, in django-speak). The best way to do this is to open an existing Maymyo model (like infra.ValueSet) and use "Save As" to your own application's models directory.

If you want your model *MyModel* to be audited (you should, it comes free), ie all adds, updates and deletes are tracked and is available as History, then import the following at the top of your *my_model.py* file:

```
from django.db.models.signals import pre_save, post_save, pre_delete
from infra.custom.audit_handlers import audit_update_handler, audit_add_handler, audit_delete_handler
```

and right after your model's class definition, add:

```
# Register the audit update handler
pre_save.connect(audit_update_handler, sender=MyModel)
# Register the audit add handler
post_save.connect(audit_add_handler, sender=MyModel)
# Register the audit delete handler
pre_delete.connect(audit_delete_handler, sender=MyModel)
```

The above will register your model with the audit handlers to record all changes to your model done through Maymyo. If you write your own scripts to change your model, we will still audit the changes but since we have no knowledge of who did it, *nobody* will be assumed to be the user. However, if your scripts are run by our Task Queue, we will know who the user is.

Now you can go ahead to program your Django application using your Model (remember, you are an experienced Django developer). As a matter of convention, we prefer to name our forms and admin files using our model's name. Your program can be a Django admin program, a transactional program using views, forms and templates or a report using the TextEngine or JasperEngine reporting engines. You can also build rest servers for your models using piston, please see our *infra/api* directory for some examples.

To develop new admin programs for your models, please copy our existing admin programs. You need to create a form for your model under your *forms* directory. Please copy our *infra/forms/value_set.py* and use "Save As" to your own *forms* directory. Then copy our *infra/admin/value_set.py* and use "Save As" to your own *admin* directory.

As you would have noted, you register your model and admin class with our infra_admin site. This makes your admin program available under the relative url "infra_admin/myapp/yourmodel/", where *yourmodel* is the name of your model.

You need to edit the *maymyo/urls.py* file. This should be in *maymyo_home/maymyo*. Add the line "from myapp.admin import *" at the top of this file, at about line 17, right after our own bundled applications (ie infra, common, billing...):

```
# Our own admin site
from infra.custom.infra_admin_site import infra_admin_site
# Import for all apps, its admin programs so that they can register themselves with our site.
# Please add your own apps under ours.
from infra.admin import *
from common.admin import *
from billing.admin import *
from myapp.admin import *
```

If you have problems importing *myapp.admin*, it may be because you did not create *__init__.py* file in the admin directory.

3.1 Setting up your application's URLs for your views

You should be familiar with editing urls.py to map urls to your views. Add a new urls.py to your *myapp* directory. Copy our *infra/urls.py* and remove (or comment out) lines as necessary. Then include your *myapp*'s urls in *maymyo/urls.py* by adding:

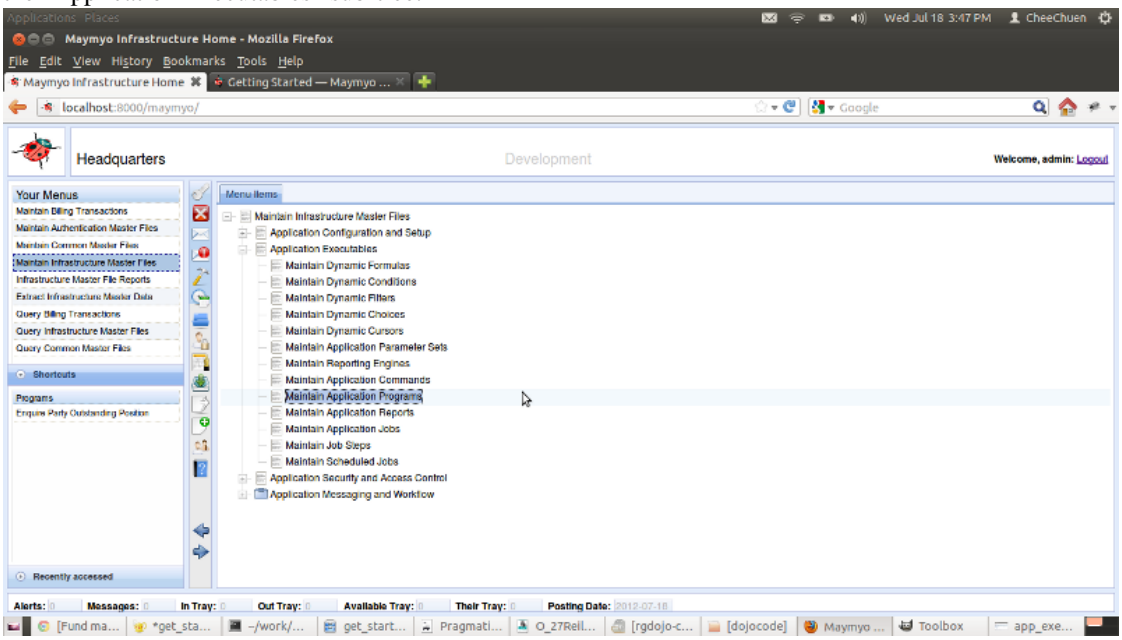
```
# Please include your own app's urls below this line....
(r'^myapp/', include('myapp.urls')),
```

This should be added to the *urlpatterns* tuple.

3.2 Accessing your new program from the Dashboard

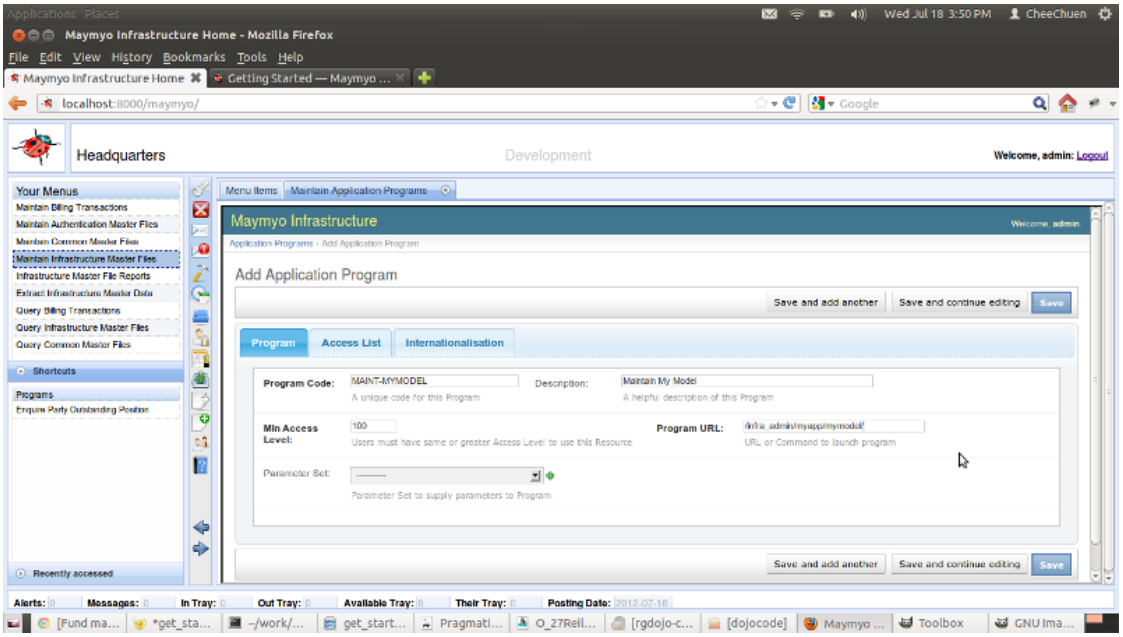
Once you have completed a program or report and are ready to publish it to your users, you need to add an AppProgram (or AppReport) instance to Maymyo. You can do this either by using our "Maintain Infrastructure Master Files" menu's "Maintain Application Programs" or permanently by adding it to your fixtures directory. The latter is more complicated, for now, we point you to our own *common/fixtures/menu_item.py* and *common/fixtures/app_program.py* to copy and modify.

Assuming you have a new admin program for your *MyModel*. Its relative url should be "/infra_admin/myapp/mymodel/". Login as "admin" and click on "Maintain Infrastructure Master Files" and expand the "Application Executables" sub-tree.



Application Executables Menu Items

Select "Maintain Application Programs" and click on the "Add Application Program" button on the top right. Add your new program as below.



Adding your new program

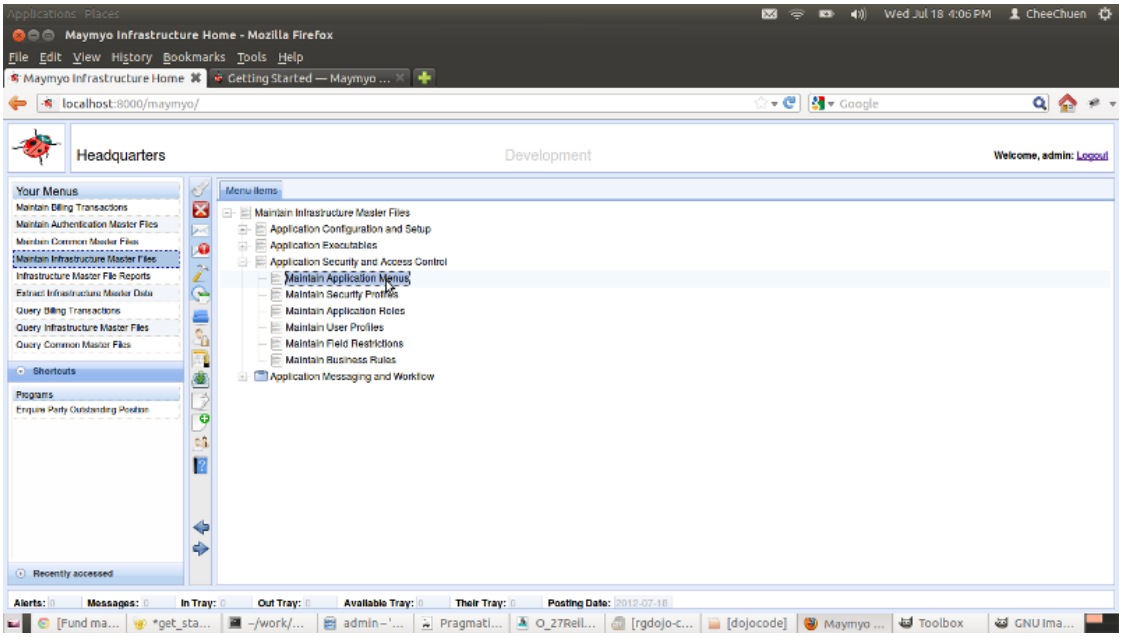
- each program must have a unique Program Code. By convention, we prepend admin programs with "MAINT-".
- leave the Minimum Access Level at 100. This controls which users can access this program. See *Application Security and Access Control* for more information.
- The program URL is relative to the host portion of your URL.
- Click "Save" to add your new program.

3.3 Creating a Menu for your programs

The next thing you need to do is to create a new Menu (or add your program to an existing Menu). Your application should have its own module menu under our "Application Main Menu". Under your module menu, you should typically have menus for :-

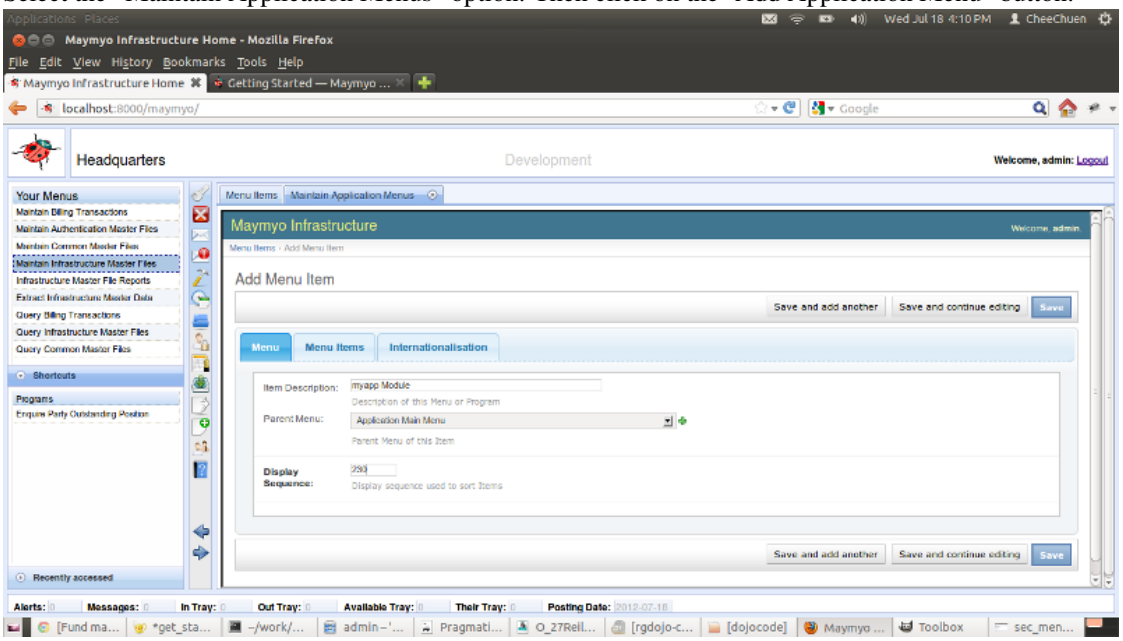
- Maintain *myapp* Master Files - for your admin programs
- Query *myapp* Master Files - your admin programs in query mode. If you inherit from our VersionModelAdmin, you can create a new query program by appending "query/" to any existing admin program url.
- *myapp* Transactions
- *myapp* Enquiries
- *myapp* Reports

In "Maintain Infrastructure Master Files", expand on the "Application Security and Access Control" sub-tree.



Application Security and Access Control menu

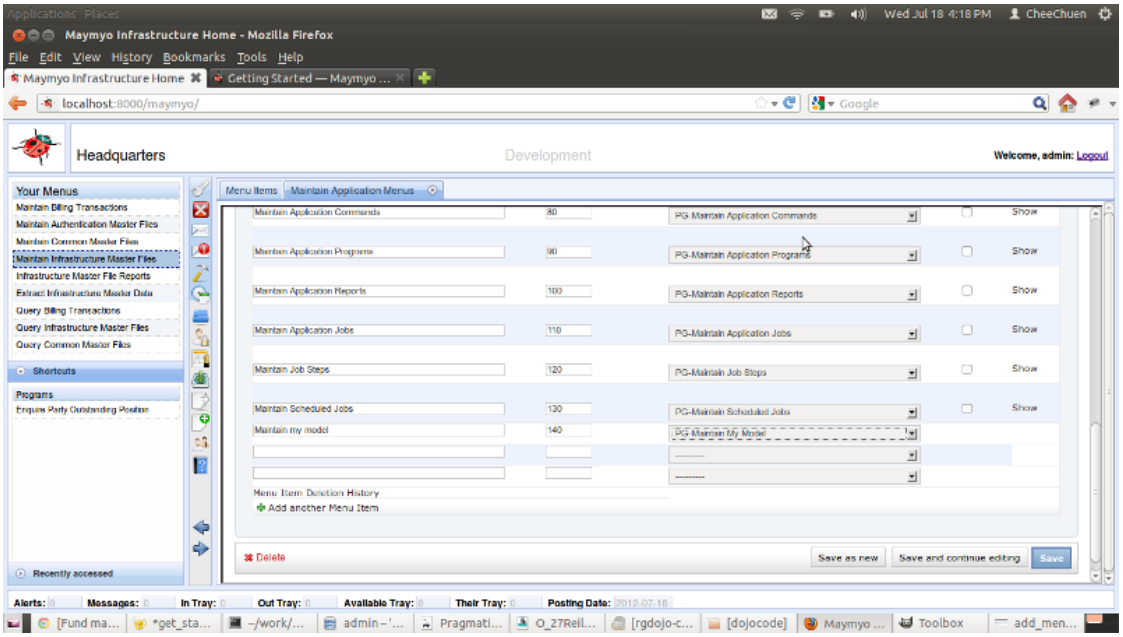
Select the "Maintain Application Menus" option. Then click on the "Add Application Menu" button.



Adding a new menu

Your module menu should use "Application Main Menu" as its parent. The "Display Sequence" field is for its ordering under the parent menu. To find out the next sequence, display the "Menu Items" tab for the parent menu. Simply add 10 to the last sequence you see.

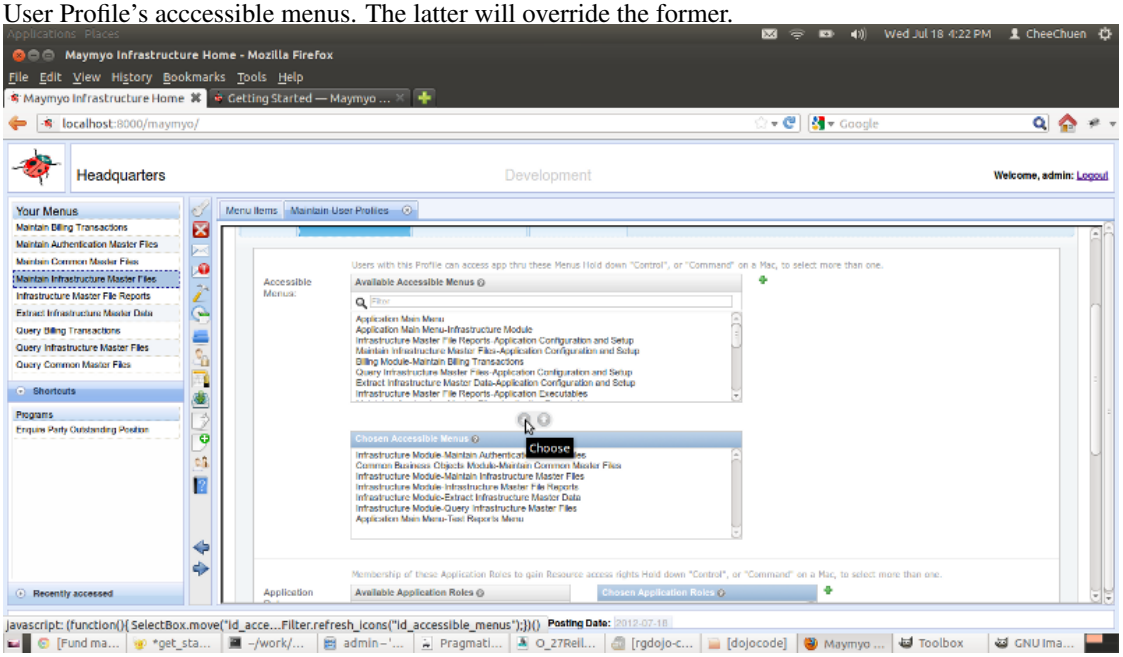
Then add the list of menus as suggested above under your new Module menu. Your admin programs should be added under the "Menu Items" tab of your "Maintain *myapp* Master File" using the "Resource" drop-down list.



Adding a new menu item

3.4 Allowing access to your menu to a User

Finally, you should allow selected users access to your menus. You may either add your menu to a Security Profile or User Profile's acccessible menus. The latter will override the former.



Adding a menu to a Security or User Profile

Click a menu in the "Available Accessible Menus" combo-box and click the down-arrow icon to allow the menu to be accessed. You should see your menu added to the "Chosen Accessible Menus" combo-box.

You can try adding your menu to a specific user and then login as that user. Your menu should appear in her Dashboard.

CHAPTER

FOUR

EXPLORE AND UNDERSTAND MAYMYO

Features:

4.1 Multi-Business Unit organisation

4.1.1 Business Unit Hierarchy

Maymyo is designed to support an Enterprise that operates multiple Business Units. A Business Unit may be a actual (ie a Branch) or virtual location. A location is a place where the enterprise conducts business. In Maymyo, this actual location is called a Branch.

The Business Units can be organised in a hierarchical manner, eg we can have a virtual HeadQuarters as the root. HeadQuarters may be parent to several Regional HeadQuarters under which are the State Offices. All these area virtual. Branches will be housed under the State Offices.

If you load the test data, you will see our Business Unit Hierarchy that resides in Malaysia, home of the authors.

These Business Units may be within the same country (using the same Functional Currency) or operate in different countries. We support this by making Branches operate under a Company's Profit Centre. Each Company may use a different Functional Currency. A Company owns an accounting Set of Books.

4.1.2 Users belong to a Business Unit

Each User of Maymyo must belong to a Business Unit. By default, she will only be allowed to access the data of this Business Unit. You can specify other Accessible Business Units in order to allow her to access data of other Business Units. Maintain User Profile allows you to do this.

If the Business Unit is virtual, then the User can access all data of its children Business Units. Note that transactions can only happen at actual locations, ie Branches, so each User will in effect have a list of accessible Branches whose data she can access. So if a User belongs to a Regional HeadQuarters then she can access data of all Branches in the States under this Region.

We implement this row-level access using *Dynamic Filtering Manager* for the django models that we want to control access to.

4.2 Application Security and Access Control

4.2.1 Access to your application

Every enterprise application needs to control who can be given access. It is usually not open to public. The people who can access it are usually its employees, customers, auditors or consultants.

For every person who can access the application, you must also decide which programs this person can run. For a non-trivial application, the number of programs can be in the hundreds. You will usually organise your programs into a set of menus. The items in a menu can also be other menus, so you can define a hierarchy of menus that a user can access.

4.2.2 Security Profile of your users

Besides controlling *what* your users can access, you may also want to define the *how* of their access. This includes attributes like :-

- *when* they can access, eg Monday to Friday, starting from 8am till 6pm. No access allowed on Rest Days and Holidays.
- their Password attributes like its quality (minimum length, number of digits), aging (eg must change after 30 days), recycling of old passwords and maximum login attempts.
- within how many days a new User must log into the application, otherwise the User account will be disabled.
- their resource usage, ie whether allowed to schedule Jobs, maximum active Sessions, maximum open Programs per Session, maximum Jobs and maximum Tasks per Job.
- how long they can leave their Session idle and how often to check for messages and alerts of the user.
- the list of Menus they can access. These will be displayed in the User's Dashboard.

The above attributes are defined in a Security Profile, which is then shared among many Users. You can only override a User's accessible menu list. All other Security Profile attributes will be enforced upon the User.

You should create your own Security Profiles that conform to your (or your Customer's) policies.

4.2.3 Maymyo Users

Maymyo extends the authentication mechanism of Django by defining a model `infra.UserProfile` to hold additional information of a User.

You create a user by :-

- adding the User using Django. At minimum you must enter :-
  - a unique user name
  - any password but this will be changed later to the user's Security Profile's initial password.
- adding a User Profile in Maymyo. At minimum, enter :-
  - which Django User this profile is linked to. This is a 1 to 1 relationship. One User cannot have more than 1 User Profiles.
  - which Business Unit and Department she belongs to.
  - which Security Profile to use
  - when she can start using the Application

Or you can take our tutorial to learn how to *create a user*.

4.2.4 Access Control to Application Resources

Access Control means who (ie the User) can access what. The what in this case is an Application Resource, which includes many objects besides Programs and Reports. So what we discuss here will apply equally to other types of Resources, which we will discover later.

Programs created using Django's admin

For Programs which are created using Django's admin (a contrib module used to quickly create programs that can perform CRUD on our Models), we inherit Django's Permission access control.

Note: CRUD means Create, Read (usually with Search ability), Update and Delete, aka known as Master File Maintenance in Maymyo.

Note: Django's Permission access control

Every Model that you create will automatically have 3 permissions, "Add, Change and Delete". By default, any superuser will have these permissions to all Models. All programs created using Django's admin will inherit these permission checking, ie if you did not explicitly grant permissions to your Model to a non-superuser User then this User will not be able to access the admin program that maintains the Model. This Permissions does not apply to non-admin Programs (as we do not use it).

Django allows you to either grant these Permissions to the User directly or indirectly via User Groups. The latter will save you a lot of headache if you have many Users. You will first grant the Add, Change and Delete permissions to the Group and then specify the Group(s) that your User belongs to.

As part of installation, we have auto-created Groups for each of our applications. They are named *app_label-maint* (eg infra-main) and you can use them for your own Users whom you allow to maintain our models.

Maymyo's Access Control

The above Django Permissions applies only to admin Programs. Maymyo has its own access control against its own Application Resources, which may be a :-

- **Application Program** - any web page called using an URL
- **Application Report** - either a Text or JasperReport report, as these are the only 2 Reporting Engines supported (for now).
- **Application Command** - anything that can be executed on the Command Line (in the Application Server where Maymyo is installed).
- **Application Job** - comprises a series of Steps that execute many Tasks using the Scheduler. Each Task may either be a Application Report or Command.
- **Application Event** - designer defined event that may be raised when a transaction has above ordinary values, eg a Payment Transaction involving more than 1 million (this is attached to a Business Rule, so it is triggered by the rule)

- **Business Rule** - designer defined Business Rule attached to a Transaction. Similar to Event but has Authorisation mechanism. This is a simple 2 step approval process where a User without permission for a Business Rule makes an authorisation request from her superior (who has permission).

Maymyo access control works in either of 2 methods :-

- a simple Access Level clearance check. Every Application Resource has a Minimum Access Level which is compared with a User's assigned Access Level. This Access Level is a positive Integer. For example, a Program "X" has a Minimum Access Level of 200. This means that only Users with Access Level 200 and above may access it. So even though a User may see this program on her menu, she will be stopped from running it.
- an Access List may be defined for an Application Resource. This will **override** the Access Level checking. The designer will define for a Resource a field which is the Key to the Access List and up to 5 other fields to be used as Attributes.
  - This Key allows you to define many Access List for one Resource, one each per Key Value. You may also use the ALL-ROWS wildcard to define a catch-all Access List. This is used when a particular Key Value has no Access List defined.
  - The Key field is used only for Business Rules and Application Events. For all other Resources you should use the ALL-ROWS wildcard for its value.
  - You can choose not to use the Key field by defining only one Access List using the wildcard ALL-ROWS.
  - the 5 Attributes fields are optional. They are useful for Events and Business Rules only.
  - the designer supplies these Key and Attribute values in a python dictionary (ie a list of Key:Value pairs) during runtime. So these values may not necessary come from the fields of the Resource. It is entirely up to the designer.

How does an Access List work?

- an Access List for a Resource is a list of Roles or Users who are allowed to access it.
- a Role is similar to a User Group. You define which Resources a Role can access and then share it with many Users. This will save you a lot of work defining Access List by Users.
- A User may have 0, 1 or many Roles. She will inherit all the access permissions of these Roles, unless if it is overridden for specific Resources by her own User Access List.
- so an Access List, whether by Role or User behaves the same way. You define an Access List row for a Resource, Role (or User) with a Key Value and up to 5 Attribute values as defined for that Resource.
- what Attribute values to use for a Resource is decided by the designer. But this applies only to Business Rules and Application Events. For the other Resources, we should not be using the Key or Attribute fields.
- For the Attributes, you define in the Access List the Access Values. You may use real values or our wildcards, ALL-ROWS or NO-ROW. For the latter, no access is allowed no matter what.
- When a User wants to access a Resource (which has its Key and Attribute values), Maymyo will look at whether *any* Access List is defined for that Resource. If none, then it will use the Access Level mechanism described earlier.
- If there is at least 1 Access List defined for that Resource, then we will look for the User's Access List for that Resource. If none, then we will look for Access Lists for all the Roles that this User have.
- So if neither is found for the User, then she will be denied access.
- But if an Access List is defined for that User, it will be used, otherwise we will check the Access List of all her Roles until the first one allows her access.
- A User Access List may have start and end datetimes. You can use this to give temporary access to a User, eg for someone who is acting as replacement for the usual User who is on leave.

- How we check the Key and Attribute values
  - with the Key Value of the Resource, we will look for the Access List with this exact Key Value. If none is found, then we will fallback on the ALL-ROWS Key Value. If both not found, then the User (or Role) has no access.
  - When an Access List is found, we will check the Resource's Attribute values against the Access List's access values, one by one. All Attribute values checked must return True, ie they are logically ANDed together. If any one returns False, then access is denied.
    - When the access value is ALL-ROWS, then we return True.
    - When the access value is NO-ROW, then we return False.
    - When either the access value or Resource Attribute value is null (or blank), then we return False.
    - When the access value is numeric, then we check that the access value is greater or equal the Resource attribute value. (ie access_value >= actual_value)
    - When the access value is a numeric or string range, "20-30" or "AAA-CCC", then we will check that the Resource attribute value is between these 2 values, eg 20 <= access_value <= 30.
    - When the access value is a single string, eg "ABC" then we will check that for equality of the Resource attribute value against the access value, eg access_value = 'ABC'
    - When the access value is a comma delimited list, eg "A,B,C", we will check that the Resource Attribute value is a member of this list, eg access_value in ('A', 'B', 'C')
  - You (as the Administrator) may decide to toggle the results of the Attribute checks. What this means is that the result of the check above is negated (except for the first 3 items, ALL-ROWS, NO-ROW or when either is null), ie True becomes False and vice-versa.

Example usage of Access List

- you have a special Report that only the CEO can access. Define a single Access List for this Report with the CEO's user. Key Value should be ALL-ROWS.
- We (as the designer) have a Business Rule called "RECEIVE-PAYMENT". This Business Rule is checked during the Selection for the "Receive Payment from Customer" transaction. We decide to use the Product Code (of the Customer) as the Key field and Receipt Amount as the Attribute value. Your business may have Products called "GOLD" and "SILVER". "GOLD" customers are high net-worth, so we will not be surprised if they routinely pay us more than 100,000 but for "SILVER" customers this would be extra-ordinary. Assume we have 2 Roles, "CLERK" and "MANAGER". So we should define the Access List for this Business Rule as follows :-

  1. Role "MANAGER", with Key Value "ALL-ROWS" and Attribute 1 "ALL-ROWS"
  2. Role "CLERK", with Key Value "GOLD" and Attribute 1 "1000000"
  3. Role "CLERK", with Key Value "ALL-ROWS" and Attribute 1 "100000"

  We will be toggling the check for Attribute 1. This is because we want to use the "Less Than" operator instead of the normal "Greater or Equal".

  The effects of these will be :-

  1. Users who have the Role "MANAGER" has permission to perform Receipt of any Amount.
  2. Users who have the Role "CLERK" can perform Receipt for "GOLD" customers for Amounts up to 999,999.99 and all other Customers, up to 99,999.99. If the amount is Equal or more than this threshold, she will have to request for an Authorisation from her Manager to allow the transaction selection to go through.
  3. You should not have users having both Roles. In this event, the more powerful Role will dominate.

What the above means is that "GOLD" customers will surprise us only when they pay us more than 1 million while for other customers we set this threshold at 100,000. Notice that the wildcard ALL-ROWS will apply to Products which are not "GOLD". This is a good practice to avoid disallowing access when new Products are added.

We would usually link the above Business Rule with an Event to allow other Users to register the attribute values of this transaction, above which they want to be informed (via our Messaging service).

4.3 Calendar and business day computations

In an enterprise application, many business transactions may require business day computations, eg an Invoice due date may be 14 business days from billing date.

A Calendar in Maymyo will specify 2 things, its rest days in a week and its Holidays. When we compute business days, we will omit counting these.

4.3.1 Methods available using Calendars

We provide the following for each calendar. Some are class instance methods and others are normal functions where the calendar is passed in as a parameter.

Class instance methods defined in *infra/models/calendar.py*

- Is holiday - class instance method, returns True or False for an input date
- Is rest day - class instance method, returns True or False for an input date
- Is business day - class instance method, returns True or False for an input date, ie is either a rest day or holiday.

Normal functions, in *infra/objects/app_calendar.py*

- Last Day in the month for an input date
- First Day in the month for an input date
- Add Months for an input date. A second parameter is an integer for the number of months. When negative then the result will be the number of months in the past.
- Return the Application default calendar. This is maintained in the Application Registry with the key DEFAULT-CALENDAR. All methods that compute business days will use this when its optional Calendar is not passed in.
- Compute Business Days between 2 dates. An optional Calendar can be passed in, will use Application default when not passed in.
- Get next Business Date for an input date and number of days. The number of days when negative will compute an earlier date. This is the most often used to compute due dates.
- Get the Month's first business day for an input date and optional Calendar.
- Get the Month's last business day for an input date and optional Calendar.
- Get the Year's first business day for an input date and optional Calendar.
- Get the Year's last business day for an input date and optional Calendar.
- Get a Quarter's canonical number for an input date, ie 1 to 4.
- Get a Quarter start or end date for an input date.

4.3.2 How we use calendars

First of all, when computing business days or deciding if a date is a business day or not, we need a Calendar. Calendars are attached to the following :-

- Business Unit, eg all Users must belong to one, so to decide if a user is logging in on a rest day or not, her Business Unit's calendar is used. Also all Branches are also Business Units, so any business transactions arising from it will use its Calendar.
- Scheduled Job - we have a scheduler daemon that can automatically start Application Jobs. If a Job is to be run on the last business day of the month, then we will use the Scheduled Job's Calendar, which when not defined, we will use its Job Owner's Business Unit's Calendar. When all else fails, then the Application default. If a Scheduled Job does not use any business day computation, then the Calendar is ignored.
- Workflow Type - when using Duration Type of BUSINESS-DAY, then we need a calendar. We will use its Calendar, failing which the Application Default.

You may also use Calendar in your models. An example of how we use it is in our Billing module, which uses the Party model from the Common module. Each Party (ie a person or company who is a Customer or Vendor) must have a home Branch and Payment Term for each Billing Type (for Invoices or Debit Notes). If a Payment Term uses Business Day computation, then we will use the Party's home Branch's Calendar.

4.4 Auditing of changes to Models

There are 2 services provided by Maymyo for auditing changes to model instances.

- recording who and when an instance was created and last updated, plus incrementing a Last Version number which is used for concurrency control. We call this the RecordOwner pattern.
- recording the actual changes to each field in the model instance since creation.

If your models uses these services, then their admin (ie maintenance programs) will automatically have links that show the history of changes to particular instances.

4.4.1 How to use these services in your models

We shall use an existing model in Maymyo to illustrate:

```
# From Django
from django.utils.translation import ugettext_lazy as _ # To mark strings for translations
from django.db import models
from django.db.models.signals import pre_save, post_save, pre_delete

# Our modules
from infra.models.record_owner import RecordOwner
from infra.custom.fields import CodeField, CharNullField, DescriptionField
# Too many circular import errors when we use Filter Manager with Value Set
from infra.custom.filter_manager import FilterManager
from infra.custom.audit_handlers import audit_update_handler, audit_add_handler, audit_delete_handler
from bin.constants import RANGE_SEPARATOR

class ValueSet(RecordOwner):
    """
    Application specific List of Values.

    A value set is a List of Values used in this application.
    Instead of hardcoding these lists, we create a set of values
```

```
    here. Each ValueSet has as children one or more ValueSetMembers
    whose Value Code and Description are used to
    build the drop-down list (for the HTML Select Input).
    Use the class method get_choices() for this.

    Whenever you want to create a model that has just a Code and
    a Value, avoid doing so and just create a new ValueSet. Use
    your model name (in uppercase) as the Value Set Code. In
    addition, we allow up to 5 attributes per Value, which your
    own program can use in anyway it likes.
    """
    # Each Value Set has a Unique Code
    value_set_code = CodeField(verbose_name=_("Value Set Code"), unique=True)
    # and a helpful description
    value_set_description = DescriptionField(verbose_name=_("Description"),
        help_text=_("A helpful description of this Value Set"))
    # Optionally restrict Value Set Members' Value Code to this size (1-200)
    maximum_length = models.PositiveSmallIntegerField(verbose_name=_("Maximum Value Code Length"),
        blank=True, null=True,
        help_text=_("Optionally constrain its Value Code length (must be 1 to 200)"))
    # Is this an Application Constant, when True, only Administrator can change it
    is_app_constant = models.BooleanField(verbose_name=_("Is this an Application Constant?"), default
        help_text=_("Application Constants can only be changed by Administrators"))

    # Override Manager to our own FilterManager
    objects = FilterManager()
    # Need another Manager without filtering for use in basic system functions
    all_objects = models.Manager()

    class Meta:
        ordering = ['value_set_code']
        verbose_name = _("Value Set")
        app_label = 'infra'
        db_table = 'if_value_set'

    def __unicode__(self):
        # we prefer to describe ourself using code then description
        return self.value_set_code

    def get_choices(self):
        choices = []
        if self.id:
            # Must prepend an empty selection for Fields that allows blank=True
            choices = [(u'', _("No Selection"))] + [(row.value_code, row.value_description)
                for row in self.valuesetmember_set.all().order_by('value_code')]
        # Return a List of 2 value Tuples as choices
        return choices

# Register the audit update handler
pre_save.connect(audit_update_handler, sender=ValueSet)
# Register the audit add handler
post_save.connect(audit_add_handler, sender=ValueSet)
# Register the audit delete handler
pre_delete.connect(audit_delete_handler, sender=ValueSet)

class ValueSetMember(RecordOwner):
    """
    Children of a Value Set.
```

```
    """
    # Parent Value Set
    value_set = models.ForeignKey(ValueSet, on_delete=models.PROTECT, verbose_name=_("Value Set"))
    # A unique Value Code
    value_code = models.CharField(verbose_name=_("Value Code"), max_length=200,
        help_text=_("A Unique Value Code within thapp_labelis Value Set"))
    # with a helpful description
    value_description = DescriptionField(verbose_name=_("Description"),
        help_text=_("A helpful description of this Value Code"))
    # up to 5 optional Attributes, with dynamic Choices if Value Set defined a choice
    attribute_1 = CharNullField(verbose_name=_("Attribute 1"), max_length=200,
        help_text=_("Optional Attribute 1 value"))
    attribute_2 = CharNullField(verbose_name=_("Attribute 2"), max_length=200,
        help_text=_("Optional Attribute 2 value"))
    attribute_3 = CharNullField(verbose_name=_("Attribute 3"), max_length=200,
        help_text=_("Optional Attribute 3 value"))
    attribute_4 = CharNullField(verbose_name=_("Attribute 4"), max_length=200,
        help_text=_("Optional Attribute 4 value"))
    attribute_5 = CharNullField(verbose_name=_("Attribute 5"), max_length=200,
        help_text=_("Optional Attribute 5 value"))

    class Meta:
        unique_together = ('value_set', 'value_code')
        ordering = ['value_set', 'value_code']
        verbose_name = _("Value Set Member")
        app_label = 'infra'
        db_table = 'if_value_set_member'

    def __unicode__(self):
        return self.value_description

# Register the audit update handler
pre_save.connect(audit_update_handler, sender=ValueSetMember)
# Register the audit add handler
post_save.connect(audit_add_handler, sender=ValueSetMember)
# Register the audit delete handler
pre_delete.connect(audit_delete_handler, sender=ValueSetMember)
```

The above defines the models ValueSet and its child model ValueSetMember. It can be found in *infra/models/value_set.py*. Whenever you find yourself needing to create a new model with just a code and description, resist it and just create a new ValueSet entry. The *value_set_code* in ValueSet becomes the name of your model while its code-description pairs will reside in ValueSetMember's *value_code* and *value_description*. As an added bonus, each ValueSetMember allows you to define up to 5 attributes, which you can use as you please. A common use of ValueSets is for choices of drop-down lists.

Now if you look at the definition above, you will see the necessary imports to use auditing:

```
from django.db.models.signals import pre_save, post_save, pre_delete
...
from infra.models.record_owner import RecordOwner
...
from infra.custom.audit_handlers import audit_update_handler, audit_add_handler, audit_delete_handler
```

These 3 imports will allow you to use the 2 services mentioned above. The 1st and 3rd imports are for auditing while the 2nd is for RecordOwner.

RecordOwner

To use the RecordOwner service, just inherit your model from RecordOwner. Then every time your model instance is saved, who and when it was created and last updated is recorded automatically. We do not normally show these fields in our admin programs. You will have to use a Query Browser tool against your database to view them. You can include these fields in your reports.

Note: How do we know who is the user updating the instance? We cheat by using a MiddleWare class called ThreadLocals. We know that you can get this from *request.user* but we want to centralise the code with the model so that batch processing programs that execute in the Task Queue will also work. Please have a look at *infra/custom/threadlocals.py*

Concurrency Control

Besides this we will also increment the Last Version number on updates. What do we use this for? Well, if your model admin form and program inherits from our VersionModelForm and VersionModelAdmin (and VersionTabularInline for inline models) respectively, we will ensure that no 2 users can concurrently update the same model instance.

How can 2 (or more) users update the same instance at the same time? When a model is queried (ie read) and displayed for users to select for update, no database locks is placed on the row (in the table that stores your model instances). So 2 users can view the same set of instances. If they were to select the same instance to update, let's say the same field to different values, then both of them will be successful (if our concurrency control is not used). Because django admin uses auto-commit, a database lock will be placed on the row for a very short time. The second user need not hit the Save button at the same time as the first user to be able to save her changes, thereby over-writing the updates of the first user.

When you use our VersionModelForm, we will automatically add a validation method that will compare the Last Version number (as queried earlier) with its current value (we perform a quick select against the database). If there are different, we will raise a validation error. When the first user successfully updates the same instance, she would have incremented the Last Version. So when the second user hit Save, the validation would have read the latest version number, which will be different.

You will have noticed that there is a small performance hit taken to requery the model instance on every form validation. From a business point of view, for example, preventing a Bank Account balance from over withdrawn, means that this is a price worth paying.

This works also for inline models (ie child models in a parent-child admin program) if it inherits from VersionTabularInline or VersionStackedInline.

Auditing

Besides the 1st and 3rd imports above, you will need to place the block of code below:

```
# Register the audit update handler
pre_save.connect(audit_update_handler, sender=ValueSet)
# Register the audit add handler
post_save.connect(audit_add_handler, sender=ValueSet)
# Register the audit delete handler
pre_delete.connect(audit_delete_handler, sender=ValueSet)
```

right after your model's definition. You must change the sender parameter to the name of your model class.

It must be mentioned that auditing works independently from RecordOwner, ie your model need not inherit from RecordOwner to use auditing.

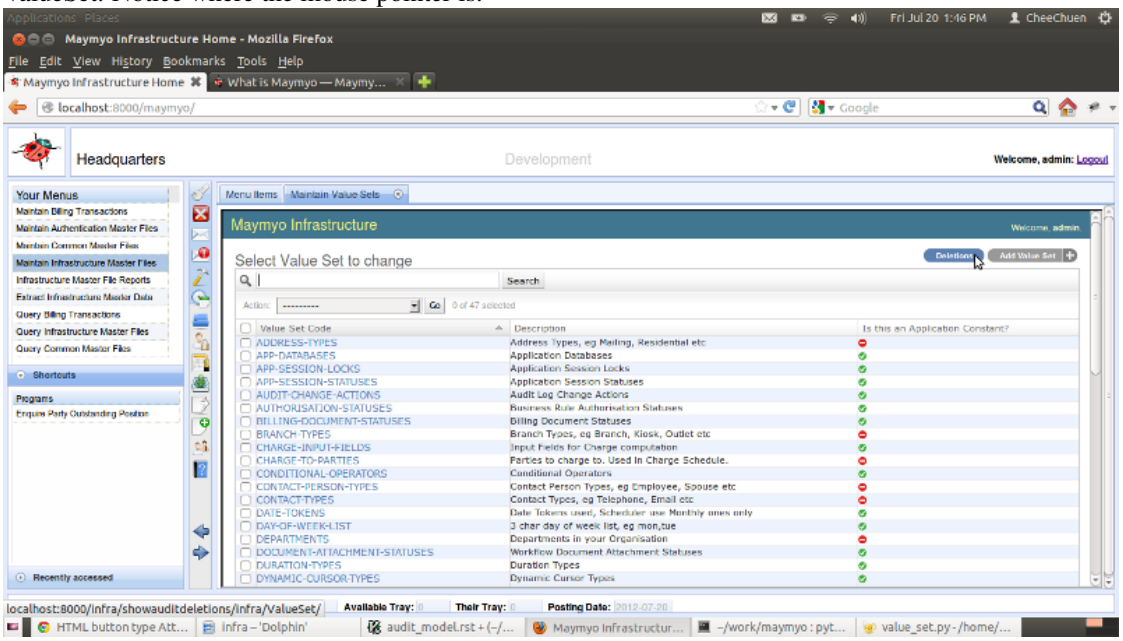
We use django's pre and post save signals to implement our auditing. What the above 3 lines of code does is to tell django to call our handler functions before saving (pre_save), after saving (post_save) and before deleting (pre_delete) a model instance. This works at the model level so it does not matter if the updates happens in an admin program, your own views or batch processing task.

We use 2 generic models to save your audit data. They are *infra.AuditHeader* and *infra.AuditLog*. AuditHeader will save the instance primary key and who and when information, while AuditLog will store its changed field values.

When adding a new model instance, we will save only non-null field values in AuditLog. When updating a model instance, we will save only the newly changed field values. When deleting a model instance, we will save only the AuditHeader information.

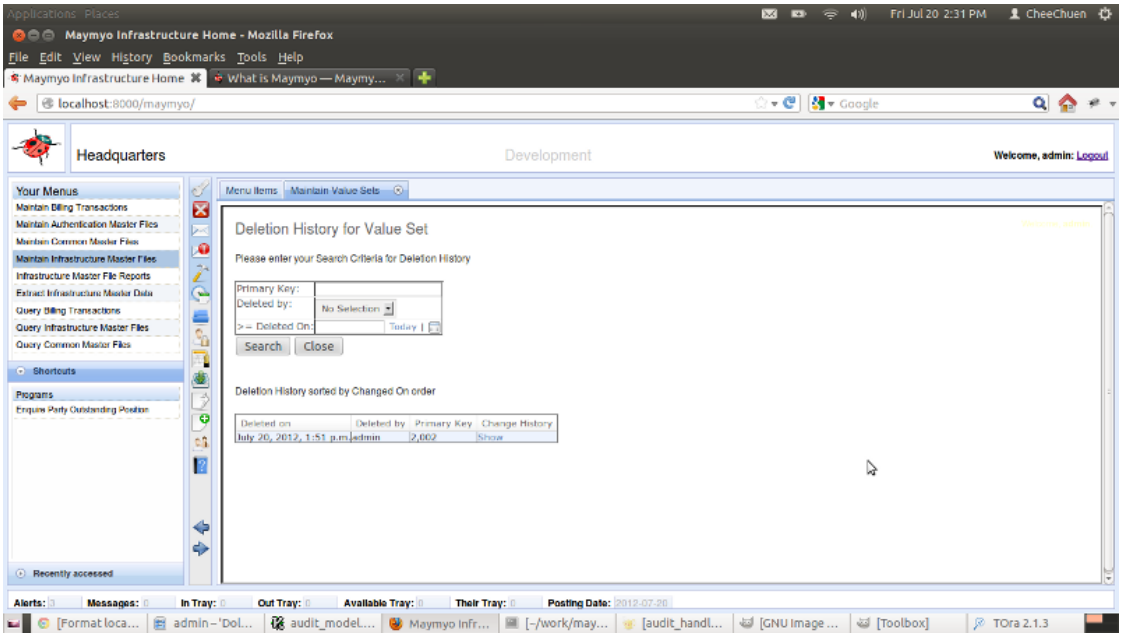
4.4.2 Viewing Audit History

If you were to run any of our admin programs, you will notice the *Deletions* button in the *Change List*. *Change List* is the page where you see a list of model instances that you can select to change or delete. This is shown below for ValueSet. Notice where the mouse pointer is.



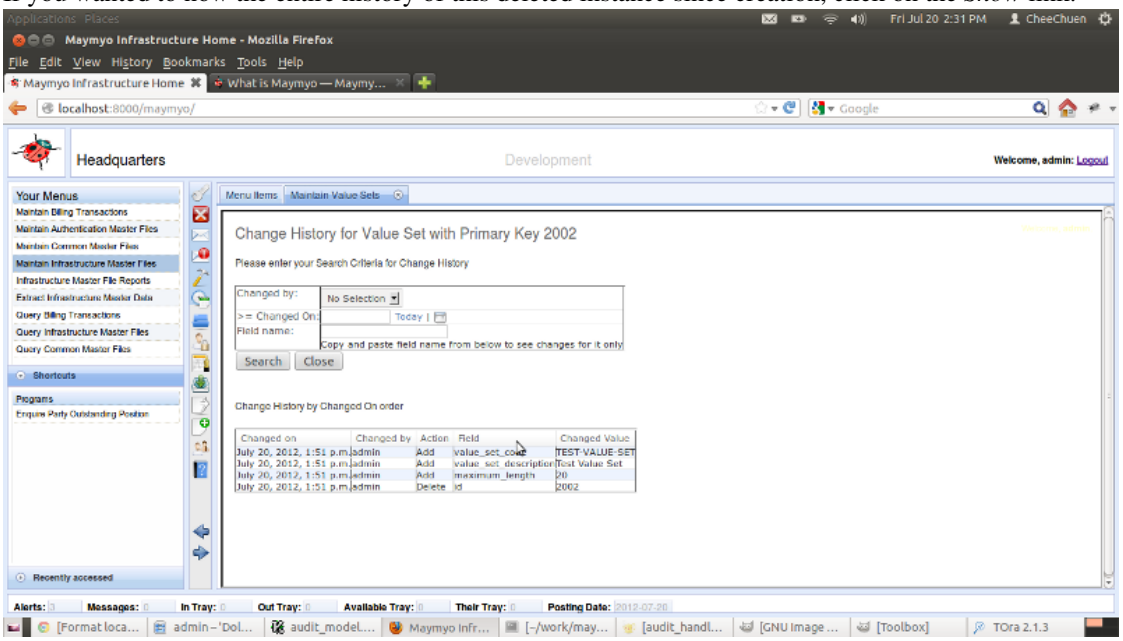
ValueSet Change List page

When you click on *Deletions*, you should be able to see all the model instances that has been deleted.



ValueSet deletions history

If you wanted to now the entire history of this deleted instance since creation, click on the *Show* link.



ValueSet instance change history

The same audit history is available for all Maymyo's models except transactional or log models, which by nature is already an audit model.

4.5 Automatic Filtering of accessible rows by Users

In database-speak, this is known as row-based security or horizontal filtering. Oracle calls it Virtual Private Database. This feature will filter the instances of a model that can be accessed by a particular user, behind the scenes, ie she will

not be shown the instances at all and will not be aware of their existence.

This should not be confused with column-based security (vertical filtering) which is implemented using database views. In Maymyo, you can implement this by creating specific views or admin programs that shows certain fields only and allowing access to these programs to certain users using our menus or access list (ie programs may be shown in a menu but access is controlled using either Access Level or Access List).

4.5.1 What does it mean?

To understand this, let's look at an organisation using Maymyo which has Branches (which are also Business Units) located in several cities in a large country. Every User who can access Maymyo must belong to a Business Unit.

You want to control access to your transactional data by allowing Users to access data that belong to her Business Unit only and not other Business Units. eg a User in the Chicago office should not see any data of any other cities except her own.

Maymyo will implement this by defining a Field Restriction for the Business Unit field. This is a Foreign Key field that refers to our BusinessUnit model. Then for all transactional models that has this BusinessUnit field, we will define a Model Restriction with a Default Restriction Value of MY-BUSINESS-UNITS. Then any super-users who can see all data can have an entry in User Restrictions with the Restriction Value of ALL-ROWS. The Default Restriction Value will be used on all Users who are not defined in the User Restrictions.

In our FilterManager class, MY-BUSINESS-UNITS will return a list of accessible Business Units of a User. Remember that *Business Units* can be a hierarchy? If a User belongs to a parent Business Unit, then she can access all its children Business Units. This is what we mean by accessible Business Units.

4.5.2 How to use it for your own models

If you look at our *ValueSet model definition* in the *Auditing topic*, you will notice the following lines:

```
# Override Manager to our own FilterManager
objects = FilterManager()
# Need another Manager without filtering for use in basic system functions
all_objects = models.Manager()
```

This is right after all the field definitions of ValueSet and just before the inner *Meta* class. What this does is to override the normal model manager to our custom FilterManager. You will also need to create a reference to the normal manager using *all_objects*. There may be certain situations when we need to allow a user to access all instances, eg when displaying a list of choices for a ValueSet. But this is within the control of the programmer (ie you).

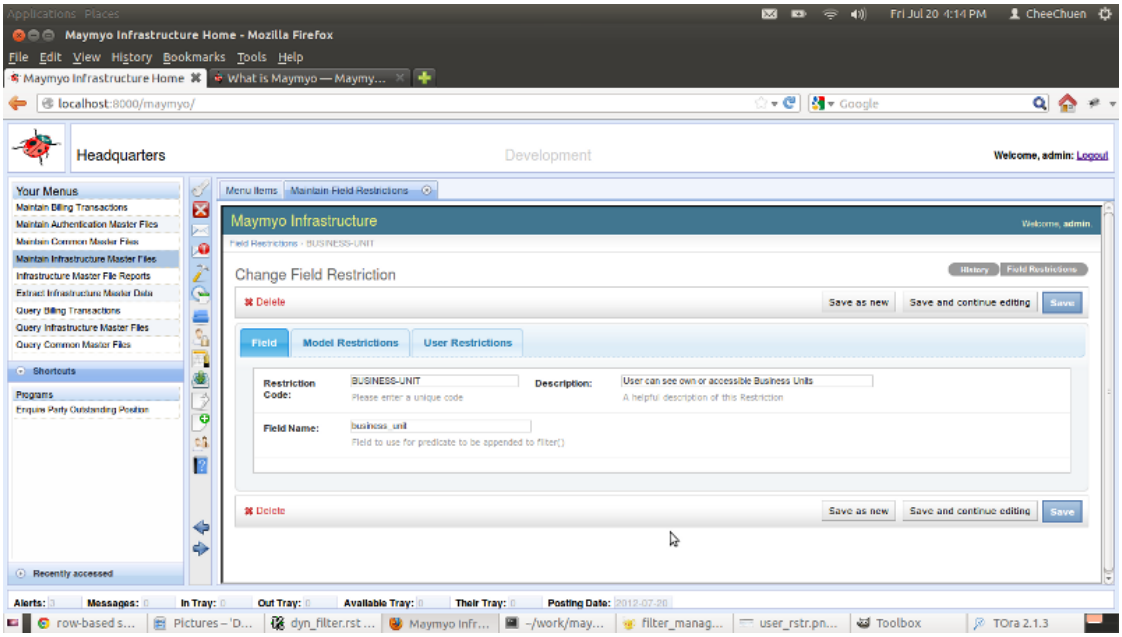
You will also need to import the FilterManager first, before you can use it.:

```
from infra.custom.filter_manager import FilterManager
```

This is the first step. Next you will need to define the filtering criteria. These are the following maintenance. You will find this in the "Maintain Infrastructure Master Files" menu under the "Application Security and Access Control" sub-tree's "Maintain Field Restrictions".

1. Maintain a Field Restriction for the field
2. Maintain a Model Restriction for models which has this field
3. Maintain User Restrictions (one per user) for those allowed access to instances.

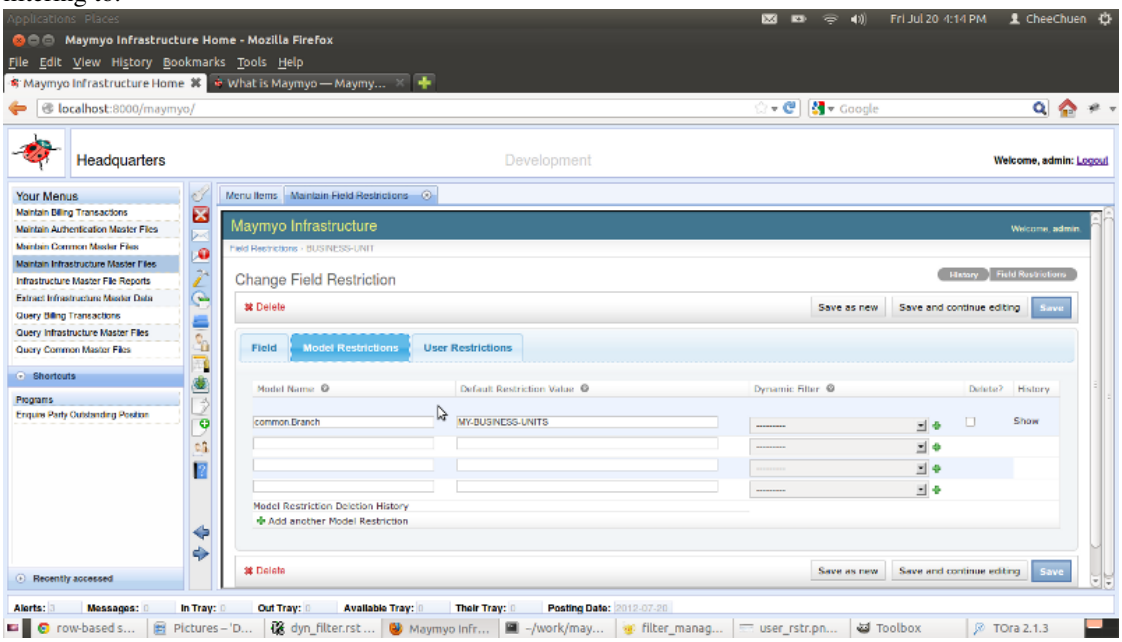
You must have a field in your model whose value is used to decide access to certain users. In our Business Unit example above, we have a field called *business_unit* in our *common.Branch* model.



Dynamic Filtering of accessible rows of Users Maintain Field Restrictions

We define a unique "Restriction Code" called BUSINESS-UNIT and the "Field Name" is *business_unit*.

"Restriction Code" must be a unique field code in upper case and "Field Name" must be the same as your model's field name. In the "Model Restrictions" tab, you will define which model (which has this field) we will apply automatic filtering to.



Maintain Model Restrictions

You will notice that you can define many Models per Field Restriction. This is because the same Field can exist in many Models, eg you may have a Branch field that exist in several transactional Models. You need only define a list of Users allowed access to specific field values and it shall be applied to all Models defined in Model Restrictions.

For each Model, you can define the "Default Restriction Value" for Users who are NOT in the "User Restrictions"

(next tab). Besides using the actual value, there are certain special tokens you can use as Restriction Value.:-

- ALL-ROWS - all instances of a Model can be accessed
- NO-ROW - no instance can be accessed
- MY-SUBORD - my (ie the user's) subordinates only, ie used when the Field is a User Foreign Key field.
- MY-SUBORD-BY-NAME - same as My Subordinates but when Field uses the username field.
- MY-PEERS - my peers, ie same position in the Hierarchy.
- MY-PEERS-BY-NAME - same as My Peers but using the username
- MY-BUSINESS-UNITS - a User's accessible Business Unit(s). A User must belong to a Business Unit and can have a list (comma delimited) of Business Units (by its code) that she can access. All these are her Accessible Business Units. If a Business Unit is a parent, then all its children (and grand-children) will be added to the list.

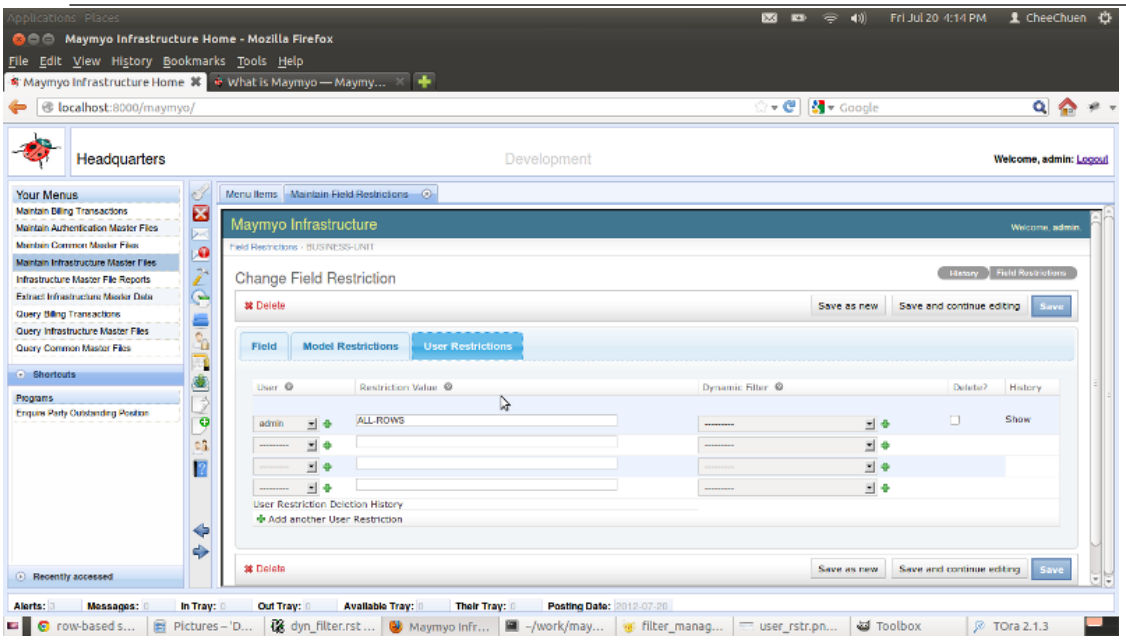
In addition, instead of using "Default Restriction Value", you can use a Dynamic Filter, which returns a dictionary of filter predicate(s) which will be used against the Model. If you are familiar with Django's ORM (Object Relational Mapper), you would have used its QuerySet filter() method which access predicates like "field__operator=value". This is the predicate which is returned by a Dynamic Filter. You must create the Dynamic Filter yourself in "Maintain Dynamic Filters" in the "Application Executables" sub-tree in the same menu. The predicate returned must operate in your model's field(s).

With Dynamic Filters, you can do virtually anything. We will always pass in the current User instance into your Dynamic Filter. It is up to you to define how to work out which rows the user can access by using the full power of QuerySet's filter.

Note: Application Hierarchies

The tokens above that applies to subordinates and peers make use of Maymyo's Application Hierarchies. You can define a Hierarchy of Users in Maymyo. A Hierarchy is a pyramid of Job Positions, defining who reports to whom. For each Position, you may have Users who are holders of that position, starting from an effective date (until superseded by holding another position in the same Hierarchy).

A User may hold positions in 0, 1 or more Hierarchies. Hierarchies are used in Workflow document approvals and Reminder escalations.



Maintain User Restrictions

You should use the "Default Restriction Value" to define access to your Models for all Users and use the "User Restrictions" to define the exceptions. It is useful to allow super-users like "admin" to have access to all rows by using the ALL-ROWS token.

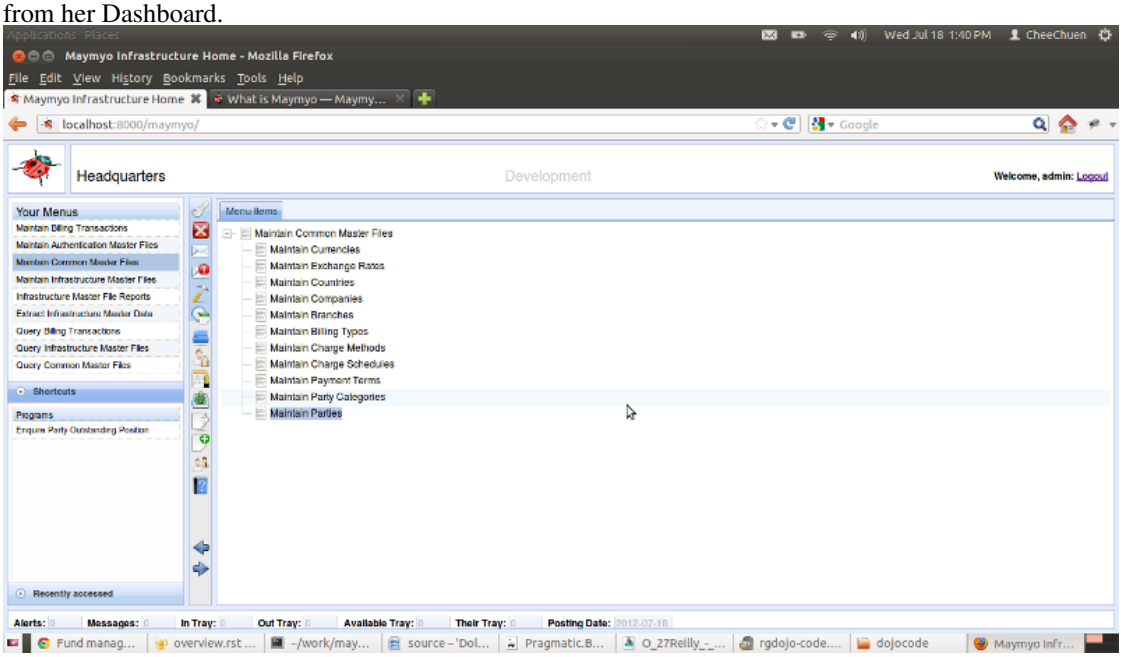
4.6 Messaging and Alerts

Messaging is intended for developer use. Messaging is asynchronous in nature, ie the message needs to be delivered to the recipient out of band. Synchronous messaging is very familiar to the user. We do it all the time, eg every time we raise a validation error when an invalid value is entered in an input field of a form.

An Alert is the same as a Message, just that its Message Type has a Delivery Method of 'A' and its delivery is segregated from normal messages.

4.6.1 Message and Alert delivery

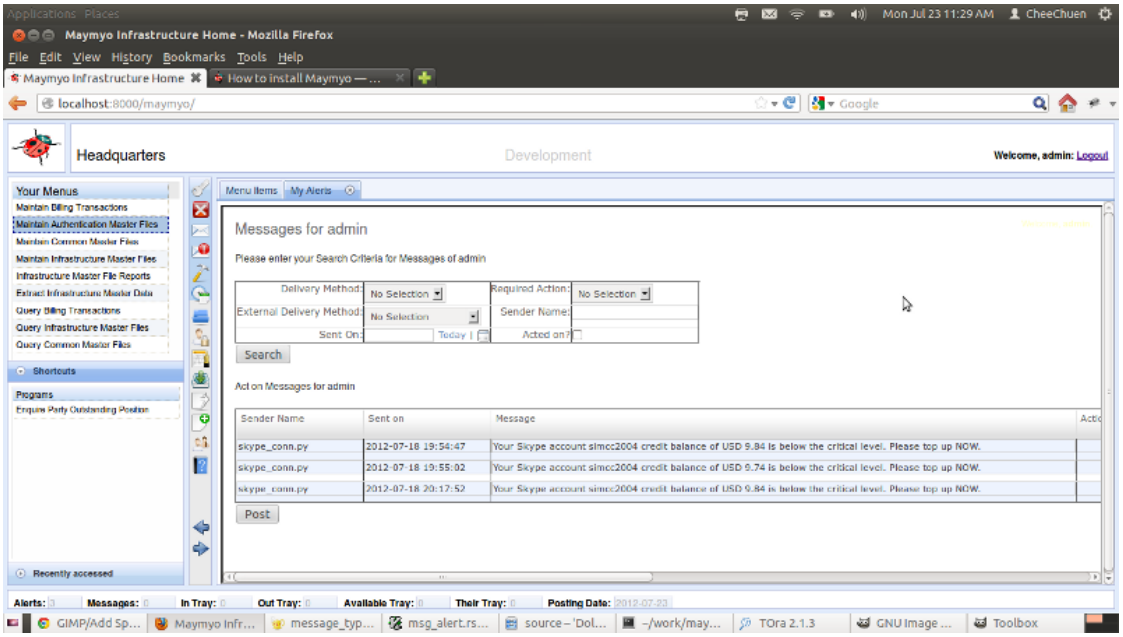
Messages to the users usually comes from the Task Queue, Output Queue, Workflow engine, Application Events or Reminder Events. When a user submits a task, eg a report, to the Task Queue she will immediately regain control of her session and can continue with another transaction. So there is no way we can send an in band message to her. Later when the Task Queue has completed her report, it will send a message to her Message Queue, which can be retrieved from her Dashboard.



Messaging in the Dashboard

In the Dashboard above, a user will know whether she has Alerts or Messages pending her attention by looking at the counts at the bottom left of the page. If the count is above 0, then she can double-click the count field and an Alert (or Message) tab will be launched. The Dashboard will refresh the counts at regular intervals. Double-clicking when the count is 0 has no effect.

Alternatively, she can also double-click on the Message or Alert icons in the vertical Toolbar next to "Your Menus". She can allow the mouse to hover on each icon to display the tooltip on what each is for. The messaging icon uses the envelope symbol.



Check my Messages or Alerts

The top part of this page allows the user to search for specific messages or alerts. This is useful when there are a lot of messages accumulated for her attention.

Dismissing Messages and Alerts

Messages and Alerts will always be pending action from the recipient and will only stop displaying after one of the actions below has been taken. There are 3 types of actions :-

1. View only - user should click on the "Act" checkbox(es) and the click the "Post" button. However, you can specify a "Expiry Days" for its Message Type to automatically purge view messages.
2. Reply - these needs a reply value to be entered before clicking the checkbox as above. Sometimes, the reply value is a Drop-down list to be selected, usually a Yes/No choice.
3. Action - some messages or alerts requires the user to click on the "Act" button which will launch another tab for further action by the user. An example is when the message is an Authorisation request for a Business Rule.

When a new Alert arrives (or whenever the User logs in and there are pending alerts (which are awaiting action)), then a message will be displayed at the bottom right of the Dashboard. We have decided not to deliver alerts via pop-up windows because from past experience, we find that they annoy the users.

External Delivery

The same Message or Alert can also be delivered to the recipient via sms and/or email. This is controlled by its Message Type. This is in addition to delivery via the Dashboard.

Before you can send email, you must define your SMTP Server and user account details in the Application Registry. Please update the following Registry Keys :-

1. SMTP-SERVER : hostname of your Simple Mail Transport Protocol server, eg smtp.mymail.org
2. SMTP-PORT : Your SMTP host listens for connections using this TCP Port, usually 25.
3. SMTP-USERNAME : Your user account name with your SMTP host, usually your email address.

4. SMTP-PASSWORD : Your DES encrypted password to log in to your SMTP host. You need to use our *bin/des_engine.py* to encrypt your password.

Encrypting your SMTP account password for updating to the Application Registry. From the shell or Command Prompt:

```
$ python
>>> from bin.des_engine import des_engine
>>> des = des_engine()
>>> des.encrypt('MYPASSWORD')
>>> 'c6d37916e59c1eb35f54d234317349cc'
```

Copy and paste the last line (without the quotes) and update to the Application Registry. Test your email connection by running "email_conn.py" on the command line.

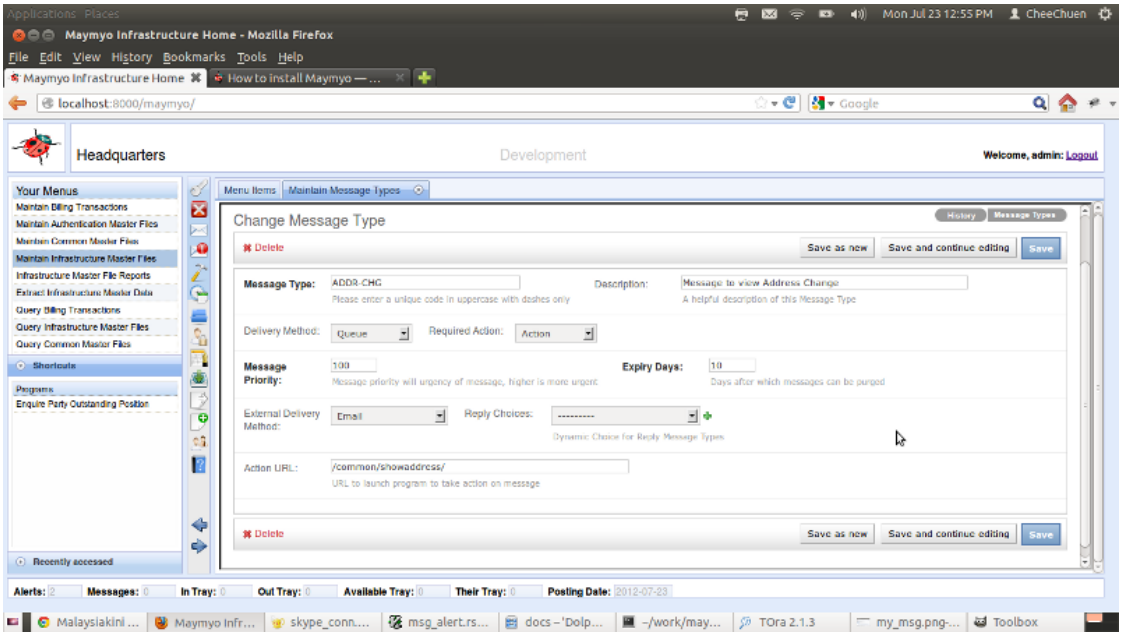
As for sending sms, we use Skype only. We do not forsee a high volume of sms. However if your application needs to send a higher volume of sms than us, then you can write your own using the API provided your sms service provider, best if it is a python API. Please look at our bin/skype_conn.py as a guide. You need to install Skype on the same server where Maymyo Daemons are running, sign up for an account and buy some credit. You must login to Skype before you start the daemons.

When Maymyo is installed, sending of sms is turned off by default. You can turn it back on by editing *snippets/daemon.tab* and changing the last parameter to "N":

```
# Deliver Messages in MessageQueue (whose MessageType has external
# delivery method specified) by sms and/or email.
# skype (for sms) and smtp server must be connected to App Server and
# its parameters defined in AppRegistry.
# Parameters: P1 = Y/N to disable sending sms
msgr|Y|messenger.py|Y|06:00:00|01:00:00|1|Y
```

4.6.2 Using Messaging in your programs

Maintain Message Type



Maintain Message Types

When you want to use the Messaging mechanism in your programs, the first thing is to add a new Message Type. Give your Message Type a unique code. Then in the Delivery Method you have to choose how the message is to be delivered to the recipient, either "Queue" or "Alert". The former is a normal message. Then the "Required Action" specifies how the recipient acts on the message. "Expiry Days" applies to how many days View messages are kept before they are purged from the recipients' queue. So View Only messages should not be important messages as they can be purged without the recipient ever seeing them. Reply and Action messages are never purged until the recipient has acted on them.

"External Delivery Method" tells us whether you want the message to be sent via email and/ or sms to the recipient.

"Reply Choices" applies only to Reply messages, ie when the recipient must enter a Reply Value against the message. For example, selecting the YESNO-CHOICES will force the recipient to confirm the contents of the message. You can create your own Choices. Please refer to TODO.

Finally, "Action URL" refers to Action messages, when you want the message to directly launch a program (with parameters values you saved earlier with the message) so that the recipient can take action immediately without going through the Menu. A GET request will be formed using this URL plus its parameter values.

Using our standard Message Types

If your messages are not Action or need special Reply values, then you can choose to use our standard Message Types :-

- VIEW-NORMAL : view only message that is purged after 10 days
- REPLY-YESNO : reply Yes/No message
- ALERT-VIEW : view only alerts that is purged after 10 days
- ALERT-YESNO : reply Yes/No alert
- ALERT-SMS : alert that will also be sent as sms, purged after 10 days
- ALERT-EMAIL : alert that will also be sent as email, purged after 10 days
- ALERT-BOTH : alert that will also be sent as sms and email, purged after 10 days

Maintain Message Templates

If your program is sending email messages, then you may want to format a longer message that can use field values from your model instance (only one allowed per template, but we allow you to use its related fields, eg "fk.fk_field"). Which model instance to use is completely up to you.

Using our Reminder Events as an example, we generate reminders against a model whose status has not changed after a time interval (from its initial date and time). We expect the recipient to be a user in the same model instance. In this case we can construct a message using the fields of this model instance. An example would be to generate reminders from the Message Queue model whose Action messages has not been acted on after 14 days.

You can use these tokens in your message template :-

1. Model instance field values using *#field1#* or *#fk.fk_field#*, where *fk* is the Foreign Key field. The enclosing # is required.
2. Function calls using the fields as parameters, *@package.module.function(#field1#,#field2#,'literal1')@*. Your function must start from your package and module that can be found in PYTHONPATH. The enclosing @ is required.

We will replace the tokens with the actual field values using *infra.objects.app_tokens.replace_token_values*.

Sending Messages in your programs

You will need to import our send_message function:

```
from infra.objects.message_type import send_message
```

If you are using a Message Template, then you need to prepare the message:

```
from infra.objects.app_token import replace_token_values
....
message_text = replace_token_values(your_model_instance, message_template.message_text)
external_message = replace_token_values(your_model_instance, message_template.external_message)
```

Then send the message to the recipient by:

```
# Send message to recipient
msg_que = send_message(recipient=recipient, message_text=message_text,
    sender=your_model_name, message_type=your_message_type,
    sender_instance=str(your_model_instance.id), external_message=external_message)
```

If you are not using Message Template, then omit the previous block and also the *external_message* parameter for *send_message*. The *sender* and *sender_instance* is for the recipient to figure out who sent the message. You can replace *sender* with a user instance and leave the *sender_instance* blank if using a user as the sender.

send_message will return back the Message Queue instance created (to send the message to the recipient). *recipient* must be a valid User instance.

Sending Action Messages

If your Message Type is an Action, then you may need to pass in a python list of parameter values using the *action_parameters* parameter to *send_message*. We allow only scalar types like int, float or str. If you pass in dates or datetimes, stringify them first to ISO format and let your action program convert them back to dates or datetimes.

The "Act" button in "Check your Messages" will append these as GET parameters to the action url.

Sending Messages to external recipients

You can send sms or email to external recipients, ie people who are not users of Maymyo. After getting a Message Queue instance (in *msg_que* variable in the above example), call its *add_recipient* method:

```
msg_que.add_recipient(contact_reference='mobile_or_email', email_mode='Cc')
```

The *contact_reference* should be a mobile phone number if sending by sms or email address if otherwise. You cannot send both sms and email (actually if you want to do that, then call add_recipient twice).

The *email_mode* applies only if sending email. If can be *To* (default if you do not pass in this parameter), *Cc* or *Bcc*.

It is up to you how to prepare a list of external recipients. It may be a list of Contact Persons for a Customer. Have a look at our Reminder Events for an example on how to get external recipients dynamically from related model instances.

4.7 Generate Reminders with optional escalation (to superiors)

Reminder Events uses the *Messaging services* to deliver its reminders. Your models can trigger reminders if it fulfills the following criteria.

1. a Date or Datetime field, whose value is compared with the current time.
2. a User field (to send reminders to). This can be left blank if you use Reminder Recipients to select at least one User instance. But this is required if you want to escalate to her superior.
3. a status field which tells us that nothing has been done by User above after a specific period of days.

The Reminder is triggered based on a source Model instance which is still waiting for action by a User (specified in a field with a User foreign key). Waiting for action means a certain status value has not changed after a period of days from initiation. This can be a status code or balance unpaid amount.

You can optionally send reminders to the user's superior by attaching an Application Hierarchy to the Reminder Event definition. The hierarchy will tell us who is the superior of the user in question.

Reminder Recipients are other users or external people you want to send the same reminder to. External people are not users of Maymyo and we can only send sms or email to them. You will need to use a Message Type that sends sms or email for this. External people could be found in another model that can be joined to your source model. For example, if your source model is the Billing Ledger (ie outstanding invoices) which has a Party field (ie the customer), then you can join to the Contact Persons of that Party to select one or more people to send sms or email to.

The sort of Models within Maymyo that can be used to generate reminders are like Workflow Document in-trays, Messages and Alerts, Business Rules awaiting authorisation, anything that involves a user and an action by that user.

When it does not involve users, then it may be an Invoice that has not been paid a number of days after (or before) its due date. We may want to send an email reminder to the Customer's contact person. (However, our messaging service requires at least 1 user before it can send any external messages, so this person could be the Salesperson).

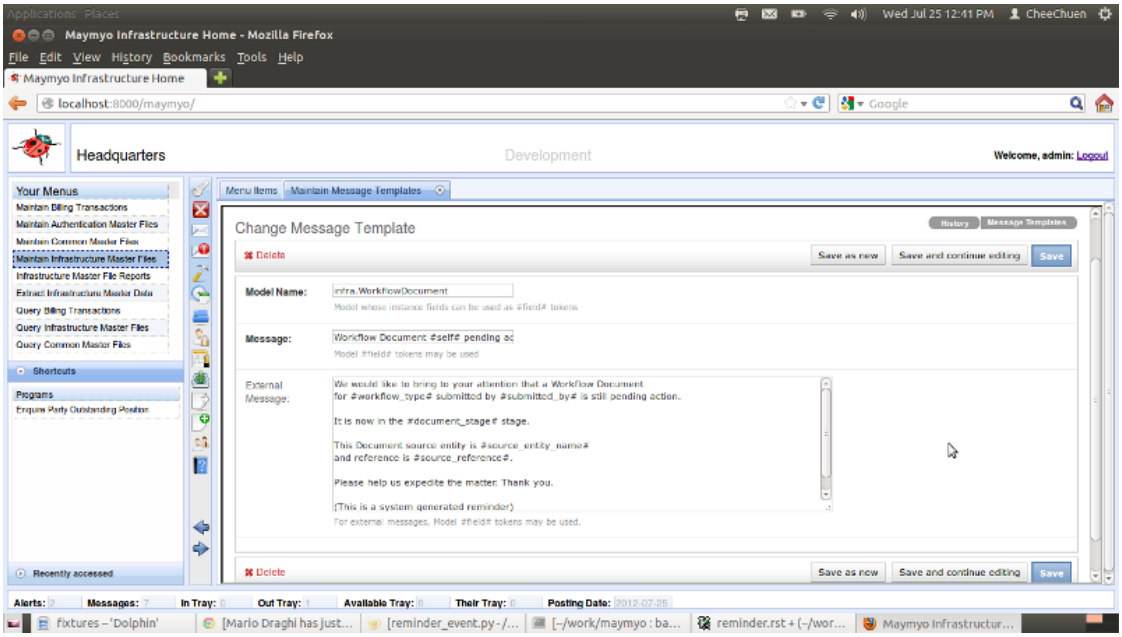
As part of the fixtures data, we have set up a daily scheduled job that generate reminders at 8pm every business day. You can either use this or disable it and add a generate reminders task to your own daily job.

4.7.1 How to configure a Reminder Event

It is best to illustrate this by using an example from Maymyo. We have a Workflow engine that it used for approval of Documents. A Document is mapped to a source model and it will track the approval stages that the document must go through. This Workflow Document will flow from the in-tray of each user it is assigned to throughout the workflow stages. If it sits for too long in somebody's in-tray, then we want to generate a reminder to the assignee (and also her superior).

The first step is to decide on a Message Type to use for the reminder message. This will determine whether you send an alert or a normal message and in addition to that, send sms and/or email. You can reuse our *standard Message Types*.

The next step is to create a Message Template, which can use field values from your source document. The following shows our Message Template for Workflow Documents that is awaiting action by an assignee.



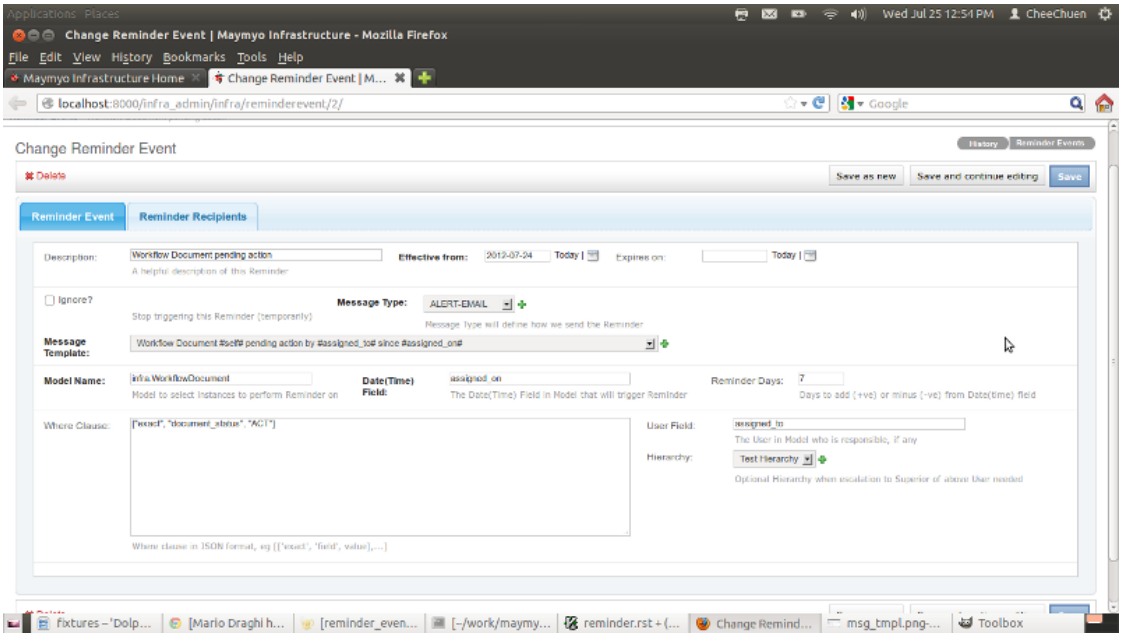
Maintain Message Templates

The Model Name should be your source model. In our case it is the Workflow Document model. The Message Text will be used for sending internal messages and sms. For emails, the Message Text will be used as the Subject while the External Message will be used for the email body.

You will notice the *#self#* and similar tokens. These will be replaced with actual values from the Workflow Document instance. *#self#* is a special field for the source model instance itself. It will be replaced with the unicode value as returned by the *__unicode__* method of the model's class. Other fields are like *#workflow_type#* or *#submitted_by#*. When a field is a foreign key field, then its unicode value is used. You can also use related fields, ie fields of the foreign key by using *#fk.fk_field#*, eg *#workflow_type.workflow_description#*.

In addition, you can call functions that returns a unicode value by using the syntax *@pkg.mod.func(#field1#,#field2#,'literal1')@*. Just make sure that this function is importable from your PYTHONPATH. An example usage would be to format a date or numeric value.

Then create the Reminder Event.



Maintain Reminder Events

You must select a previously created Message Type and Template. Each Reminder must have an effective date from when to generate reminders (until its expiry date, which when blank means never expires). You can also temporarily suspend a Reminder by checking the Ignore Flag. Just remember to uncheck it later.

The Model Name must be the same as used in the Message Template. The Date or Datetime field is *assigned_on* which is the date and time when this document was assigned. If this is a Date field, then the time portion will be 12am midnight. The Reminder Days is the number of days since assignment (ie the value of *assigned_on*). This can be negative, in which case we will send a reminder before the assignment. But this will only make sense for forward dated fields, eg an Invoice's Payment Due Date.

The User field tells us who to send the reminder to, in this case the Assignee. If you want to escalate the (same) reminder to her superior, then select an Application Hierarchy to use.

The Where Clause is a bit more complicated to define. We use the Where Clause to select instances whose status value has not changed. We use a JSON list (enclosed in []), which can be nested (ie list of lists). Each list must have 3 elements, the operator (as used by django's QuerySet filter method), the field and the value operand. In our example, we have:

```
["exact", "document_status", "ACT"]
```

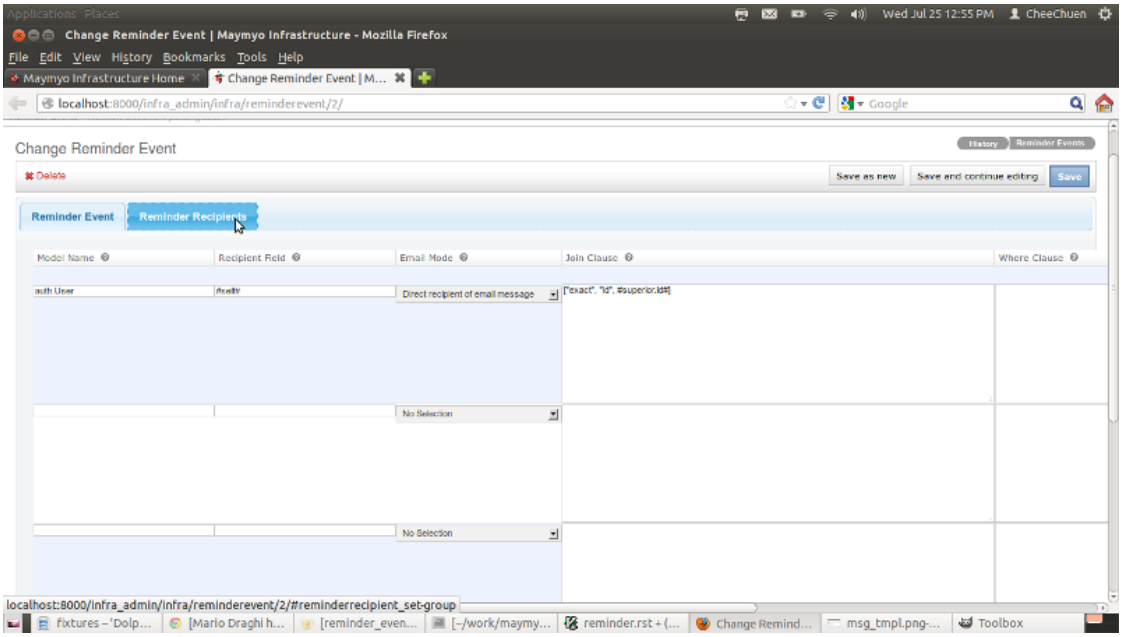
which will be translated to:

```
source_model.objects.filter(document_status__exact, 'ACT')
```

during execution. If you have nested lists, ie "[[...], [...]]", then they will be ANDed together. Please look at django's QuerySet documentation for the list of operators you can use. The value operand can support literals only, ie strings and numbers, so date(time) values are not supported. When your operator is *in*, then the value operand should be a list, eg ["A", "B", "C"].

JSON uses double quotes only (but single quotes will be converted automatically to double quotes).

You may also want to send the same reminder to other recipients.



Maintain Reminder Recipients

Each line in this maintenance allows the selection of one or more recipients. The recipients can be Maymyo Users (if the Recipient Field yields an auth.User instance) or external people, ie the Recipient Field yields either a mobile number or email address.

The Recipient(s) instance will come from another Model, which can be joined with the source Model (in this example, the Workflow Document) using the Join Clause. You can omit the Join Clause and use the Where Clause if these models are not related, eg you want to send reminders to certain Users in the IT Department.

The Email Mode is either *To*, ie the direct recipient, *Cc* or *Bcc*, ie the carbon-copied or blind carbon-copied recipient. This applies only when the Message Type sends emails.

The Join Clause is similar to the Where Clause described earlier with one exception. You can use field or function tokens from the Reminder Model for the operand value, ie fields from the Workflow Document. In the above example, we are joining to the *auth.User* model using the *#superior_user.id#* field. The second element of a list in the Join Clause must be a field of the Recipient's Model (ie *auth.User*).

The Recipient Field is either a field or function token against the Recipient Model (ie *auth.User*). It must return a User instance, mobile number (for sending sms) or email address. If not, then no reminders can be sent.

What our example above does is to select the User instance of the Workflow Document's *superior_user*. (Actually, this is redundant as we can escalate the same reminder to this superior by using the same Application Hierarchy as the Workflow Type. We setup this just to illustrate how to use Reminder Recipients).

The final step is to create a daily Scheduled Job to generate reminders. However, our fixtures data has already set this up to run every 8pm on business days. You can change the timing using "Maintain Scheduled Jobs" in the "Maintain Infrastructure Master Files" menu or "View your Scheduled Jobs" button in your Dashboard's Toolbar.

4.8 General purpose dynamic formulas, conditions, cursors, filters and choices

When you want to design complex products for a specific industry, you may inevitably start to write more and more complex pieces of code to support the myriad features required by these products. What if these features changes in mid-course after going into production? Or some customers requires unique combinations of these product attributes?

You would normally modify your code and provide a new version for your customers to upgrade to. Or worse, you may have different versions for each customer.

By using our Dynamic Formula, you can send them an SQL script instead. This script will modify the dynamic formulas used by the products in the database instead of your codebase. (It may be wiser to add a new formula and replace it in the product instead so that you can revert if anything goes wrong).

This will also allow you to provide different custom features to different customers using the same codebase. Just write variations of similar formulas and allow your customer to pick and choose.

Using our dynamic features will require a major redesign of your application, so it is best to gain a thorough understanding before you embark on your design. You need to figure out the what, when and how of each of the dynamics above.

4.8.1 What is a Dynamic Formula?

A Dynamic Formula is a python function that returns a python datatype, ie scalars (eg int, float, date), list or dict. It can also use the return values of other Dynamic Formulas. We call these Embedded Formulas. You can pass into this function a list of your model instances and named parameters (in a python dict). Which models and parameters to use depends on you, eg if you are designing a Payroll Computation, the model list could include the Employee instance and the parameters may have the Payroll Date.

4.8.2 When to use a Dynamic Formula?

Whenever you have computations that differ in their methodology, eg simple interest vs monthly rest, rather than having a flag as an attribute of your model, use a Dynamic Formula that returns a number instead. Pass in as parameters the required model instances, eg Invoice instance plus a Current Date (to compute interest until) as named parameter. You do this in your code, for example:

```
int_amt = my_model.interest_formula.execute(model_instances={'Invoice': inv_inst},
    params={'current_date': date.today()})
```

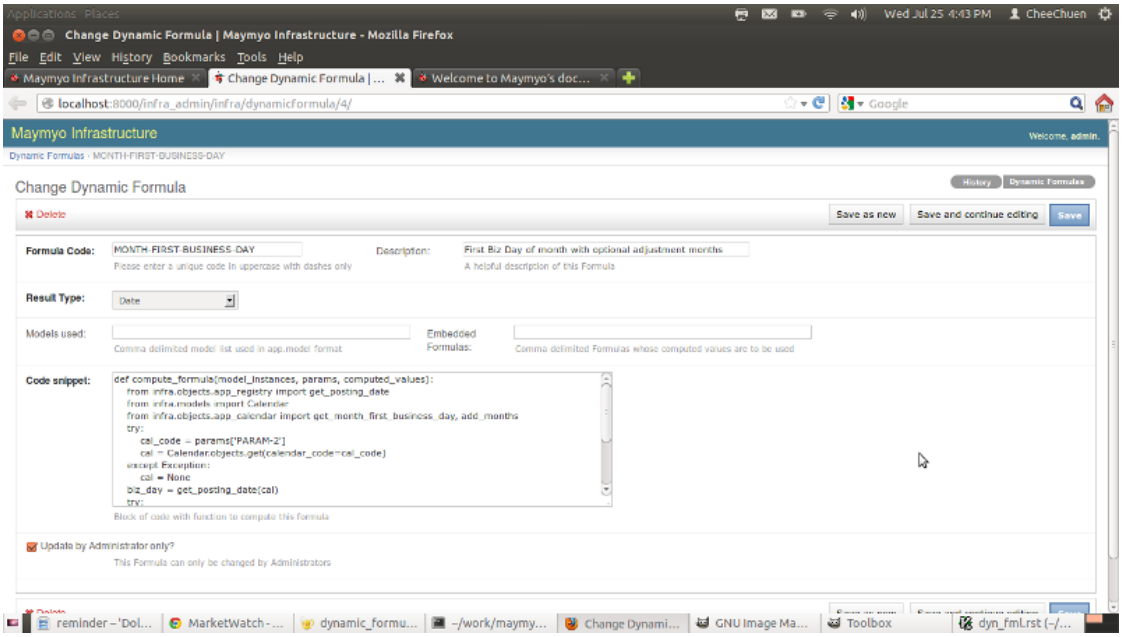
The above shows the code snippet where the *interest_formula* is called to compute the interest for a particular invoice. The model name (ie key in the *model_instance* dict) should not have the *app_label*.

The code snippet for that formula above should be using the fields of the Invoice instance (eg Balance Unpaid Amount, Last Interest Date etc) plus the named parameters to compute the interest. It should also use the Party (ie Customer) field in Invoice to get the interest rate to use.

4.8.3 How to use a Dynamic Formula?

First, you have to design your application, ie its models and transactions. Identify which transactions will benefit from the use of Dynamic Formulas. Then add a dynamic_formula foreign key field to your relevant models and call our dynamic formula execute() method, like above, in your transaction posting code.

Defining a formula is straight-forward.



Maintain Dynamic Formulas

Give your formula a unique code and description. The description should describe the methodology used. The description will be used in drop-down list of your models to select a formula. Then you must select a Result Type, ie for interest computation, this should be a float or Decimal.

Next you can specify the model(s) to be used as input to your formula. You can leave this blank if your formula does not use any. The Model used must be a comma separated list of *app_label.model_name*, eg "myapp.Invoice, myapp.Customer". When you call execute(), you must pass in instances of this model in the *model_instances* dict, using just the *model_name* as key (without the *app_label*).

The *Update by Administrators only* flag is used to protect Maymyo's formulas from being accessed by non-administrators. We use *Automatic Filters* for this to prevent non-administrators from accessing these formulas. Please have a look at the Model Restriction definition for *infra.DynamicFormula*.

Embedded Formulas

When you have many formulas which are long and complex computation, many steps may be similar. In that case, it would be wiser to split them into many formulas and reuse their results as Embedded Formulas in specific formulas which are different for each customer.

Using the results of other formulas requires 3 steps. The first is to define that formula. The second is to update their unique formula codes (as a comma separated list) in the target formula's Embedded Formula field. Then in the Code Snippet, use the results of the embedded formula, let's say 'FORMULA-ONE' as follows:

```
int_amt = computed_values['FORMULA-ONE']['result_1'] * int_rate / ... rest of computation ...
```

When we see an Embedded Formula list, we will execute them first and save their values (your embedded formula's results will be the value in the *computed_values* dict. The key will be the formula code .So in the above example, the embedded formula FORMULA-ONE returns a dict with *{'result_1': 123, 'result_2': 234, ...}*. If your embedded formula returns a scalar (eg float), then you can simply use *computed_values['FORMULA-ONE']*. The Embedded Formula can be recursive, ie calling another embedded formula.

The Code Snippet

This is where you insert your python code that does what your formula is supposed to. It must be a function named *compute_formula* like snippet below:

```
def compute_formula(model_instances, params, computed_values):
    from infra.objects.app_registry import get_posting_date
    from infra.models import Calendar
    from infra.objects.app_calendar import get_month_first_business_day, add_months
    try:
        cal_code = params['PARAM-2']
        cal = Calendar.objects.get(calendar_code=cal_code)
    except Exception:
        cal = None
    biz_day = get_posting_date(cal)
    try:
        adj_months = int(params['PARAM-1'])
        biz_day = add_months(biz_day, adj_months)
    except KeyError:
        pass
    return get_month_first_business_day(biz_day, cal)
```

It must use 3 parameters, named exactly (no differences in parameter names allowed) as above. The *model_instances* is for you to pass in a dict of instances as per your formula's *Models used*. *params* is a dict of the arbitrary values you choose to pass in. The imports must be relative to your PYTHONPATH (of the OS user who runs Maymyo). If your formula is executed in the Task Queue, then it will use the Environment Parameter Set of the Application Command or Task Queue or the user who runs Maymyo daemons, in that order. (If however, the daemons are started up automatically by the Operating System, then we will be using *snippets/envvars.ini*.)

The above code will compute the current posting's date minus an input *adj_months*, if any, first business day of the month. *adj_months* will be a positive or negative integer, eg 1 or -3, meaning next month's and prior 3 month's first business day respectively.

Your code snippet must *return* a value of the same datatype as specified in your Formula's Result Type. If not, an exception (uncaught resulting in an ugly stack trace which will scare your users) will be raised.

You may wonder where did the named params *PARAM-1* and *PARAM-2* came from. It is passed in by our code. This formula is used to compute an automatic parameter value (to be submitted to reports or commands executed in Scheduled Jobs where it is not convenient to for a user to input a value at the scheduled time) for an Application Parameter. When we define that parameter value, we would use "1,CAL-CODE" and specify the formula above in its Dynamic Formula field. This is done in "Maintain Application Parameter Sets". Please have a look at the code in *infra.objects.app_parameter_set*'s *get_formula_params* function. By the way, the second parameter is the Calendar Code to use to compute the first business day (otherwise the Application Default will be used). You can see from the code that both parameters are optional, when *PARAM-1* not passed in we will use the current posting date.

It is better to code your snippet using a Text Editor or your favorite IDE, then copy and paste. Upon saving, we will compile your snippet and raise a validation error if your code fails. However, we cannot protect you from runtime errors.

Be familiar with Maymyo first

Before you start writing any code snippets, it pays to be familiar with Maymyo, ie its models and object methods besides the functions in *bin.helpers*. For example, if you need something to do with dates, there may already be a method available in *infra.objects.app_calendar* or *infra.objects.app_datetimes*.

4.8.4 Dynamic Conditions

Dynamic Conditions are similar to Dynamic Formulas except for 2 things :-

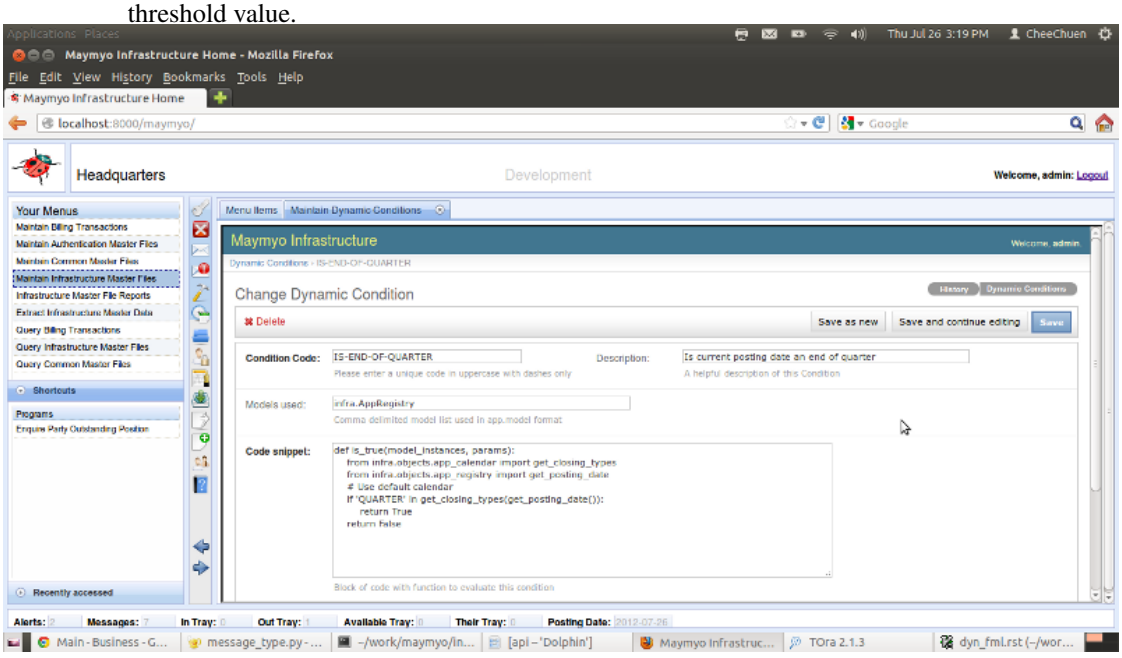
1. It must return a Boolean
2. It cannot used embedded conditions.

Use it like a Dynamic Formula that returns a Boolean, eg when deciding to allow or disallow certain transactions. Usually you will use the named parameters to pass in attributes of your transaction to check against a model instance, eg a User to see if she can be allowed to enter a certain amount.

Where it is used in Maymyo

We use Dynamic Conditions in :-

1. Application Job and its Job Steps, to decide whether to allow execution of the Job or any of its steps. The typical conditions we use will test if a particular closing date is a End of Business Day, Week, Month and the like. We can then setup a big job that combines End of Day, Week, Month, Quarter and Year closings to their own steps and run this job daily. If current day is a end of week, then only the End of Day and Week steps will run, the others will be skipped. every day. It will then only run those tasks when the current day meet its condition.
2. Application Resource access control, to decide if a user can access a resource, ie Programs, Reports, Commands, Business Rules and Events.
3. Workflow document routing, to route a document depending on its attributes, eg when more than 1 Million, than route to Senior Manager, otherwise can be posted immediately.
4. Archiving of Model's data. Decide when to archive and purge its data (in conjunction with a Dynamic Filter to select data to archive or purge).
5. Application Events, to decide which Registrant should be informed of an event based on its attributes. Registrants can choose to be informed only when certain transaction attributes are above a threshold value.



Maintain Dynamic Conditions

The above shows an example of a Dynamic Condition that test where the Current Posting Date is the End of Quarter or not.

4.8.5 Dynamic Cursors

Cursors are code snippets that yield rows of data, which can either be selected using SQL directly from the database (SQL Cursor Type) or using django's ORM (python Cursor Type). The row returned is a dict with the field name (or column name) as key.

We use it to generate text reports against our models. Please have a look at *infra/custom/text_engine.py*. This is a Reporting Engine that produces text files, either CSV or fixed-format. We will explain how we use them.

Let's look at the Code Snippet of 2 existing cursors that we use. Both of them produces exactly the same set of rows against our ValueSet and ValueSetMember models (they are parent and child).

SQL Cursor Type:

```
SELECT vs.value_set_code, vs.value_set_description, vs.maximum_length, vs.is_app_constant,
    vm.value_code, vm.value_description, vm.attribute_1, vm.attribute_2,
    vm.attribute_3, vm.attribute_4, vm.attribute_5
FROM :db_name.if_value_set AS vs
JOIN :db_name.if_value_set_member AS vm
ON (vm.value_set_id = vs.id)
WHERE <CONDITIONS>
ORDER BY vs.value_set_code, vm.value_code
```

python Cursor Type:

```
from infra.models import ValueSet, ValueSetMember
from django.forms.models import model_to_dict
def yield_rows(pos_params, named_params, filter_predicates, limit):
    qs = ValueSet.objects.filter(**filter_predicates)
    for vs in qs:
        vm = vs.valuesetmember_set.all()
        for row in vm.values():
            # Add ValueSet fields
            row.update(model_to_dict(vs))
            yield dict(row)
```

Let's start with the SQL one first. The snippet must be a SELECT statement against the database tables, which needs to be prefixed with the database name (:db_name, a named parameter passed in). The database table name is usually named by django automatically, using the app_label_model_name, eg myapp_mymodel. You should use standard SQL 92 syntax so that it will run against all the databases supported. You must always provide a `WHERE <CONDITIONS>` as a placeholder for Maymyo to replace with Auto-filtering criteria plus user entered parameter values prompted when the report is run from the menu. If your query has a where clause, then you should use `WHERE (myfield = 'field_value' OR ....) AND <CONDITIONS>`. Always enclose your Where Clause with `()`. Each column in the SELECT statement will be returned in a dict as `{'column_name': column_value,,}`.

For the python cursor type, you will notice that we use a generate function which must be named *yield_rows*. This function must accept 4 parameters as shown above. The code you see above is uses django ORM. We import the necessary models and the *model_to_dict* function from *django.forms*. Because we are returning a row that combines a child with its parent, we have 2 `for` loops to combine the child row's *values()* dict with the parent (`row.update(model_to_dict(vs))` above). Then the function must yield a dict for each row.

So you can use either of the above cursor in a `for` loop, eg:

```
dyn_cur = DynamicCursor.objects.get(cursor_code='MY-CURSOR')
for row in dyn_cur.get_cursor(pos_params, named_params, conditions):
    ... do something with row
```

Your code will have to prepare the positional parameters (*pos_params*), named parameters (*named_params*) and *conditions*, which is a dict with the *field_name* as key and a list of [operator, search_value] as value. Maymyo will construct the *conditions* dict from the values prompted from the users when the report is run. See our *infra.objects.app_parameter_set.prepare_parameters* for this.

So take look at our *infra.custom.text_engine*, which is instantiated by an Application Report which uses a TextEngine Reporting Engine. Its *run_report* method is called after a user has entered the parameters (eg to select a subset of rows to print) and selected the output format. The relevant section of the code that uses the dynamic cursor is

```
# Get Cursor returns a row which is a dict of name:value
for row in app_report.dynamic_cursor.get_cursor(pos_params, named_params, conditions):
    if output_format.value_code == 'CSV':
        field_values = [row[field] for field in field_order]
        csv_writer.writerow(map(stringify_value, field_values))
    else:
        # Handle as text
        for field in field_order:
            # Right pad field value
            output_handle.write(stringify_value(row[field]).ljust(field_defs[field][1]))
        # add a newline to end each row
        output_handle.write("\n")
```

The `for` loop will call the dynamic cursor's *get_cursor* method to yield rows. In the above code, we output each row in either CSV or fixed-format text to an output file (using its handle *output_handle*).

So you would have to put on your thinking cap to see how you can use Dynamic Cursors in your application. It will be most useful for a python based reporting tool like *Dabo*, which we have not enough time to explore.

4.8.6 Dynamic Filters

The Code Snippet of a Dynamic Filter must return a dict of filter predicates that matches the parameters you can use with QuerySet's *filter()* method, eg `{'field__operator': search_value,,}`. Each Dynamic Filter must have a single Queried Model. Just like the other dynamics, you can pass in *model_instances* to your snippet. By using Dynamic Filter, you can allow your users (we mean their Administrators) to customise your application, instead of changing your code.

Maymyo uses Dynamic Filters in :-

1. *Auto-Filtering of accessible rows by Users*. A Model's or User's default restriction can be specified using a dynamic filter instead of being a literal value (or list).
2. Archiving and Purging, the rows to be archived and purged for each model uses a dynamic filter, so that the window of available rows can be customised. Some may prefer 2 years of transactions kept online (instead of the standard 2 months).
3. Dynamic Choice, normally its Queried Model will select all rows. We can filter it by applying a Dynamic Filter so that it will return a subset of rows.

Let's look at an example that we use in Auto-filtering:

```
def get_predicates(model_instances, params):
    from django.utils.translation import ugettext as _
    from common.objects.branch import get_user_branches
    from infra.custom import CustomException
    # Get the user that is passed in
    usr = model_instances.get('User')
    if usr is None:
        raise CustomException(_('User must be passed in to get his Branches'))
    # Get user's accessible Branches
```

```
    branch_list = get_user_branches(usr)
    if branch_list:
        if len(branch_list) == 1:
            # Return a = predicate
            return {'branch': branch_list[0]}
        else:
            # Return a proper in predicate
            return {'branch__in': branch_list}
    else:
        return {'branch': -1}
```

The above Dynamic Filter is used to return a filter predicate for a User's accessible Branches. Some transactional models may have a Branch field and we want to allow a User to see only those Branches in her accessible list. The above code snippet will return a filter predicate to use against that transactional model.

As usual, the function name is fixed, same as its parameters. The function must return a dict that looks like `{'field__operator': search_value„„}`. It can also be an empty dict or a dict with multiple elements (as long as the key has a field of the Queried Model). The `field__operator` syntax is exactly what is used by QuerySet's *filter()* method. You can also use related fields, eg `field__related_field__operator`.

Back to our code snippet above. After the necessary imports, we retrieve the User instance passed in by the caller (from the *model_instances* dict). Then we get the user's accesible branches by calling the function *get_user_branches* which returns a list. If the list has 1 element, then we use the *exact* operator, ie `{'branch': branch_list[0]}`. When more than 1 element, then we use the *in* operator and supply the *branch_list* unchanged. Finally, when the user cannot access any branches, then we return a predicate that queries for a branch with id *-1*, which should return 0 rows (as there are no ids which are negative).

If you want to know how *get_user_branches* really works; from the User instance, we know which Business Unit she belongs to (she must belong to one). She may also have a list of accessible Business Units (in addition to her home Business Unit) defined in her User Profile. These becomes a list of Business Units. Now each Business Unit may have children Business Units. The code will traverse down each accessible Business Unit. Finally, we know that a Branch is actually a physical Business Unit, ie we will return all occurences of Business Units which are also a Branch.

4.8.7 Dynamic Choices

Dynamic Choices returns a list of [code, description] pairs to be used in ChoiceFields, which uses the HTML Select form widget. You can use it for your model's field whose choices can only be decided at runtime.

We use it for :-

1. Application Parameter prompted choices rather than entered values
2. Message Type's Reply Value, eg user can only reply Yes or No only rather than entered values.

If you use it, you should dynamically change your form field's widget, like what we did for a MessageQueue's *reply_value* field in its *__init__* method, as below:

```
# Reply Value to be turned into choice field?
if instance.message_type.dynamic_choice:
    self.fields['reply_value'].widget = forms.Select()
    self.fields['reply_value'].widget.choices = instance.message_type.dynamic_choice.get_choices()
```

The block of code is from *infra.forms.message_type*, at about line 176, in the *ActionMessageQueueForm* class. When it detects the message's Message Type uses a Dynamic Choice, then it will change its form widget to use the Select widget and change its choices to whatever is returned by the Dynamic Choice. When the user selects a choice, the code will be saved in the Select input field.

You choose to sort by its Description (default is by code, which the user cannot see) and change the sort order to Descending. There is no Code Snippet for Dynamic Choice. You will need to specify which fields of the Queried Model will yield the code and description and we will construct the code automatically.

4.9 Background Task Queue for tasks and reports

The Task Queue will execute tasks submitted by users in the background. These tasks include Application Commands (anything that can be run on the command line) and Reports. After completing each task, it will send a message to the user. All output generated by the task is saved and can be downloaded or viewed by users from their Dashboard's "Check your Messages" button (on the Toolbar).

The Task Queue is a daemon that can be started automatically during Operating System boot-up. Each Maymyo's installation will by default start up 2 Task Queue daemons, for Immediate and Queued tasks. The former will execute each task submitted immediately without any queueing while the latter queues task (based on priority and FIFO basis for those with same priority) subject to a maximum number of active tasks.

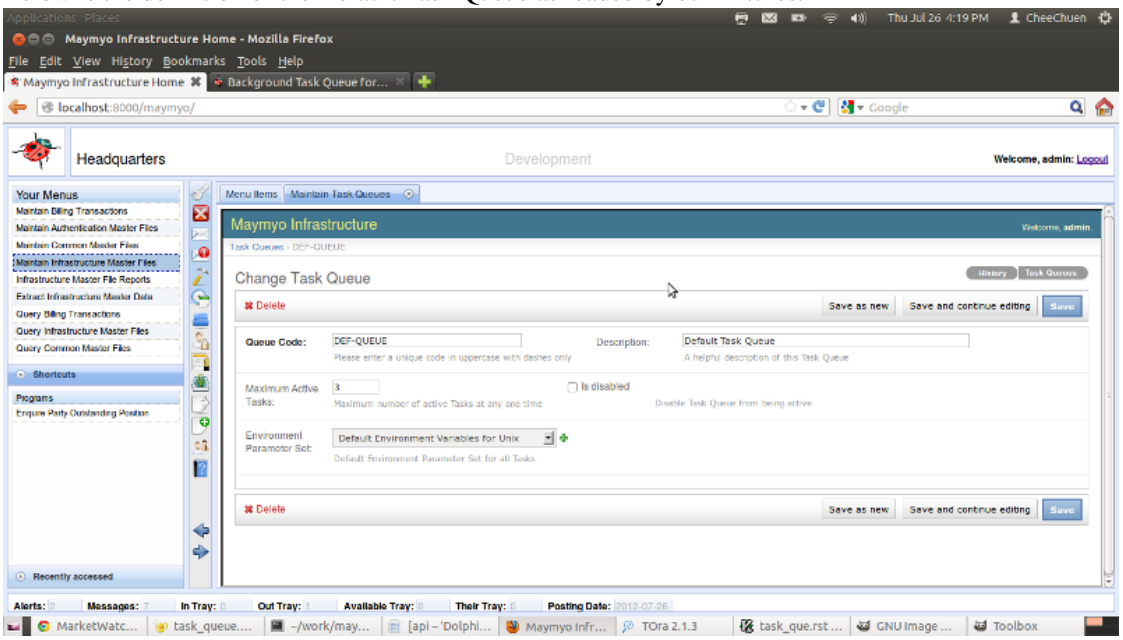
How many Task Queue daemons to startup is controlled by the configuration file *snippets/daemons.tab*. You would not normally need to add any more daemons. Adding more daemons on the same server will not improve performance or throughput.

We allow users to submit reports to the Task Queue. Whenever a report is run, we will prompt for the report parameters and allow the user to click on the *Run* or *Queue* buttons. The former will run the report interactively (a HTTP Response will be returned to the browser) while the latter will submit the report to a Task Queue and return control back to the user immediately so that she can continue with other transactions. A message will be returned to inform the user which task id is allocated to her report task. Later, when the report is completed, the Task Queue will send her a Message which can be read using the "Check your Messages" button in the Dashboard's Toolbar. She can also monitor the completion of the report task using the "View your Tasks" button on the same Toolbar.

The Task Queue is also used by the Scheduler daemon for automatically scheduled jobs. Each task of that job will be submitted for execution to the Task Queue(s).

4.9.1 Defining a Task Queue

Below is the definition of the Default Task Queue as loaded by our fixtures.



Maintain Task Queues

The fields of interest are :-

1. Maximum Active Tasks which controls how many task can be active at any one time. Other tasks will queue until an active task completes.
2. Is Disable allows you to temporarily disable the Task Queue from starting.
3. Environment Parameter Set defines which Parameter Set to use for the Task Queue daemon. This is where you would set the PATH, PYTHONPATH and DJANGO_SETTINGS_MODULE variables. Normally during installation, we will change this according to your operating system (ie either Windows or Posix-based).

If you change any of the above fields, it will only take effect the next time the Task Queue daemons are restarted.

4.9.2 Configuring the daemons table

The configuration file that controls which daemons to start up (and shut down) is in *snippets/daemons.tab*. We show below the section to control the Task Queue:

```
# Task Queues. Parameters: P1 = Queue Code in TaskQueue, P2 = Execute Mode (Queued or Immediate)
# Should be run in tandem with Scheduler, ie if scheduler runs 7 days a week, Task Queue
# should also run 7 days a week.
tkqq|Y|task_queue.py|N|06:00:00|01:00:00|1|DEF-QUEUE|Q
tkqi|Y|task_queue.py|N|06:00:00|01:00:00|1|DEF-QUEUE|I
```

Those lines that starts with the **#** are comments. The *tkqq* line has **|** character as separators between fields. Each line will start one daemon instance. There are 9 fields per line :-

1. Unique entry name field, ie *tkqq*. This entry name is used to name the log and pid files saved in the *runtime* directory.
2. Start Flag, Y/N to start this daemon.
3. python program to run for this daemon. For Task Queue, this must be as above (*task_queue.py*). This program can be found in the *bin* directory.
4. Business Day flag, Y/N to run this daemon only on business days (according to the Application default Calendar). When Y, it will not run on rest days or holidays. (Actually it will run but will be sleeping).
5. Start Time, when this daemon will start working. Above shows that we want the Task Queue to start working at 6am in the morning. This is in 24 hour format.
6. End Time, when this daemon stops working, ie until 1am. So our daemon works from 6am to 1am the next morning, leaving a window of 5 hours (from 1am - 5:59am) which should be used for database backups.
7. Debug mode, 1 (on) or 0 (off). When 1, the log messages will be more verbose. Even security sensitive information will be logged. You may want to turn this off in Production.
8. Task Queue code, the unique code to use, as defined earlier. We use this to get the maximum task and Environment Parameter Set.
9. Queued or Immediate mode for this Task Queue.

The first 7 fields are the same for the other types of daemons. The 8th field onwards are daemon specific. The above shows 2 Task Queue daemons are started, in Queue and Immediate modes respectively.

In the *Installation guide*, we have already shown you how to start up and shut down the daemons *manually* and *automatically*.

4.9.3 Using the Task Queue in your code

You can submit Application Commands or Reports (we call it a Resource) to a Task Queue. Which Task Queue to use depends on the Resource. Each Command may define its own default Task Queue. A Report will get its default Task Queue from its Reporting Engine. When this default is not defined, then you should use the Application default. Your code would probably look like this for a Report:

```
from infra.objects.task_queue import get_default_task_queue, submit_task
....

# Get Task Queue to submit into
task_queue = app_rep.reporting_engine.default_task_queue or get_default_task_queue()
```

app_rep is the Application Report instance. You should have prepared the positional and named parameters, plus the *conditions* dict using *infra.objects.app_parameter_set.prepare_parameters* after prompting the users to select her report criteria. See *infra.views.run_report* for an example.

Then submit the report to the Task Queue by:

```
queued_task = submit_task(task_queue=task_queue, app_resource=app_rep, app_session=app_session,
    pos_params=pos_params, named_params=named_params, conditions=conditions,
    output_format_code=output_format.value_code)
```

The full parameter list is:

```
submit_task(task_queue, app_resource, app_session, pos_params=None, named_params=None,
    conditions=None, output_format_code=None, submitter_model=None, instance_id=None)
```

where *task_queue*, *app_resource* and *app_session* is mandatory. The *app_resource* should either be an Application Command or Report instance. The *app_session* is the submitting user's current Application Session instance. You can always get it by

```
from infra.custom.threadlocals import get_app_session
....
app_ssn = get_app_session()
```

Of course, this only works in your views (which can only be run when logged into Maymyo) or for tasks executed using our Task Queue (where we use the *app_session* of *submit_task* to simulate a login).

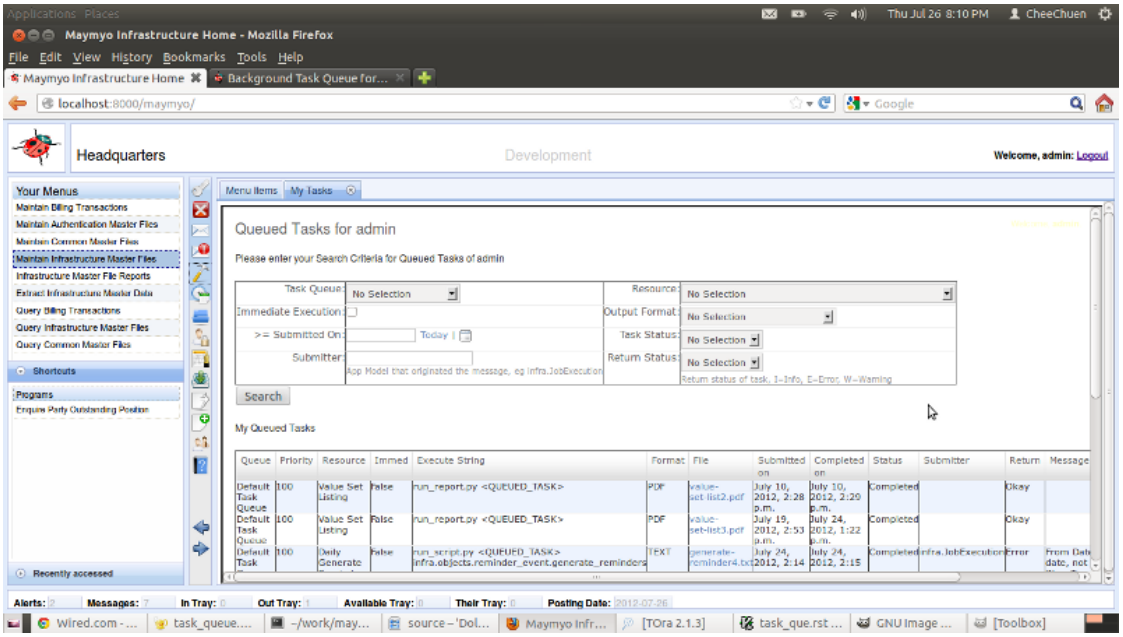
The *output_format_code* should be passed in for Reports. It should be a Value Code from the Value Set *REPORT-OUTPUT-FORMATS*.

The *submitter_model* and *instance_id* are used only for Scheduled Jobs, ignore them.

4.9.4 Viewing your tasks

If you are a user who has submitted a task, naturally you would want to know where to view the output of the task.

When the task has completed, the Task Queue will send you a completion message. You can view the message using the "Check your Messages" button on your Dashboard's Toolbar. You can then use the "View your Tasks" button to view its output.



View your Tasks

Just click on the file name link (*value-set-list2.pdf* in above screen for the 1st task) and then the browser will prompt you on what to do with the download. You can either open or save it. If you open it with the appropriate program, you can send it to your own printer or save it to your own disk. The output file is saved as a Spool Item on the Maymyo server until it is purged (which depends on the configuration in Archiving and Purging). You can download it multiple times (until it is purged).

If the link is not there, it may mean that the task is not completed yet or there has been an error. Scroll further to the right to view the status and error message.

4.10 Job definition and scheduling with business day semantics

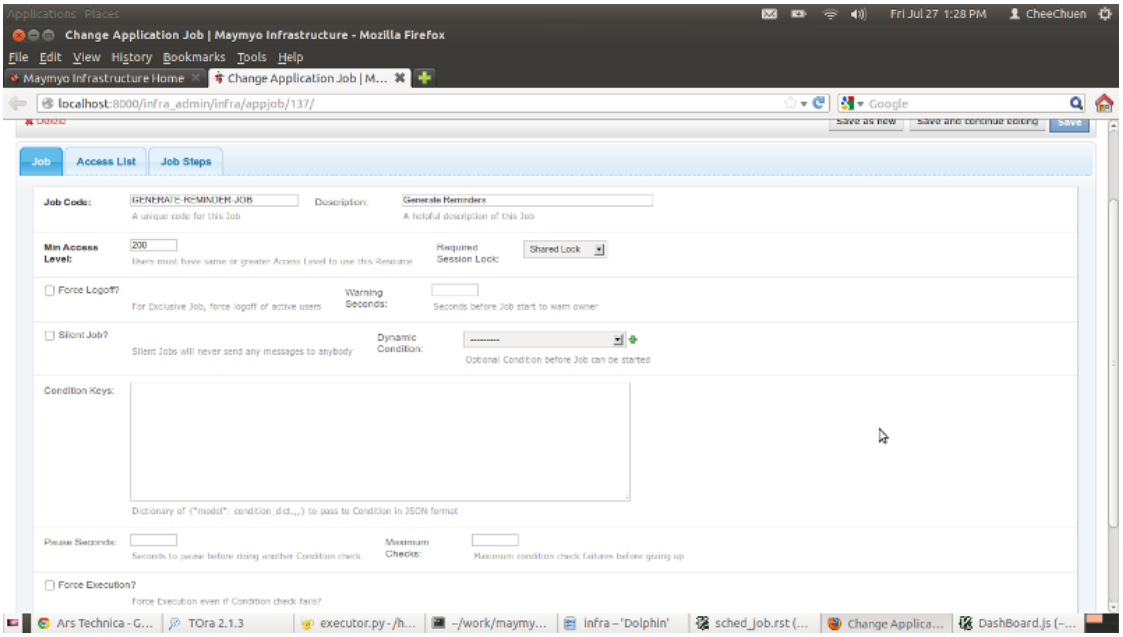
Maymyo has a Scheduler that works similarly to *cron* or Windows Scheduler. The difference is our Scheduler will only run our Application Jobs, which has Steps that can be run conditionally. Each Step will execute 1 or more Tasks simultaneously. A later Step cannot start until all the Tasks of an earlier Step has completed. A Task can be either an Application Command or Report which will be submitted to the *Task Queue for execution*.

To use the Scheduler, you must first define an Application Job. Then create a Scheduled Job that will execute this Job at a specific time, either once or repeatedly. The output of your Job's tasks will be available using the "View your Tasks" button in your Dashboard. A single Job can have many schedules.

4.10.1 Defining an Application Job

An Application Job can be very simple, comprising 1 Step and 1 Task or very complicated, comprising multiple conditional Steps, each with tens of Tasks. A simple Job can be one where a User wants to run a few reports at 7am every Monday. A complex Job usually performs a lot of batch processing and reports for your application, eg End of Day closings.

Before you create a new Job, you must be clear on what tasks you want to run. You also have to figure out if your tasks has dependencies on each other, ie a certain task can only be run after the completion of another task. You have to split these tasks into different Steps. For example, reports may depend on transactions performed by a batch processing task, so it must be in a later Step.



Maintain Application Jobs

An Application Job is a subtype of an Application Resource, so it can use the same Access List control as other Resources.

You must assign a unique code to your Job and give it a helpful description, as it will be used in drop-down lists. The *Minimum Access Level* and *Access List* tab controls the access to this Job. Leave the *Access List* tab alone because we do not use the Key or any of the Job's Attributes to control access.

If you want to disallow any other Users from scheduling your Job, then goto "Maintain User Profiles" and add this Job to your Access List with *Access Key Value* of ALL-ROWS. Alternately, you can also allow only your Role to access this Job. In this case, use "Maintain Application Roles" and add this Job to your Role's Access List.

Exclusive Lock

For a simple Job, you can leave all the other fields alone. Otherwise, a Job with a *Required Session Lock* of *Exclusive Lock* can only be performed when no Users are logged in to Maymyo. We call this an Exclusive Job and is used for system closing Jobs when we want exclusive access to the application to perform batch processing or housekeeping tasks.

When an Exclusive Job (that requires an Exclusive Lock when run) is about to start, it will send Alerts to all logged in Users *Warning Seconds* before its expected Start Time.

You can also *Force Logoff* for all active Users when your Exclusive Job is going to start. For unattended system closing Jobs, this would be a preferred option because you do not have Operators around to call the Users to logoff.

If your Job does something trivial very frequently, like every 5 minutes, then it will generate a lot of Messages. This can be annoying. You can turn off all messaging from a Job by checking the *Silent Job?*.

Dynamic Condition checking before Job starts

Sometimes, your Job can only start after somebody has uploaded a certain file that contains market data (that is needed for valuations). In this case, you can use a *Dynamic Condition* that tests whether the market data for the closing date has been uploaded or not. The *Dynamic Condition* can be tested multiple times by using the *Pause Seconds* and *Maximum Checks*. You can set *Maximum Checks* to 10 times with a *Pause Seconds* of 60 seconds between each

Check. This gives you a grace period of 10 minutes for the Condition to become True. Checking multiple times can only work in conjunction with the Job Owner. When a Condition test fails, a Message will be sent to the Job Owner (or *Rostered User*, which will be explained later) who is expected to do something, eg get other Users to log off or perform uploading of a file. Your standard operating procedure may require that this file be uploaded 30 minutes before the Job's expected Start Time but sometimes this may be overlooked. So using repeated Condition checking allows you to rectify the situation and prevent your Job from being aborted.

If there is no resolution and the Condition is still False after *Maximum Checks*, then you can force the Job to start by checking the *Force Execution* flag. You should do this when the omission (that cause the Condition to fail) is fixable after the fact. If not, consider using sms to send the Alerts to the Job Owner or Rostered User.

Passing Model Instances to Dynamic Condition

Sometimes you need to pass in some Model Instances to your *Dynamic Condition*. Since you cannot program this (the most you can do is pass in the known model instances when executing the Job, eg the Application Job or Job Owner instances), you need to let the Job Owner have a way to pass other Model Instances to the Dynamic Condition. This is where the *Condition Keys* come in. This is a dict where the key is the *app_label.model_name* and the value is a nested dict of filter predicates that yields exactly one instance, exactly what QuerySet get() method accepts, eg *{'infra.ValueSet': {value_set_code: 'ADDRESS-TYPES'}}*. This will pass in a single ValueSet instance in the *model_instance* parameter to the Dynamic Condition. Of course, you can pass in more model instances by adding other *key:value* pairs to this dict.

4.10.2 Defining Job Steps

Each Job may have one or more Steps. Each Step in turn can have one or more Tasks, to be executed simultaneously. You will need more than one Step when some Tasks has dependencies on earlier Tasks, for example reporting tasks must wait for batch processing tasks to complete first.

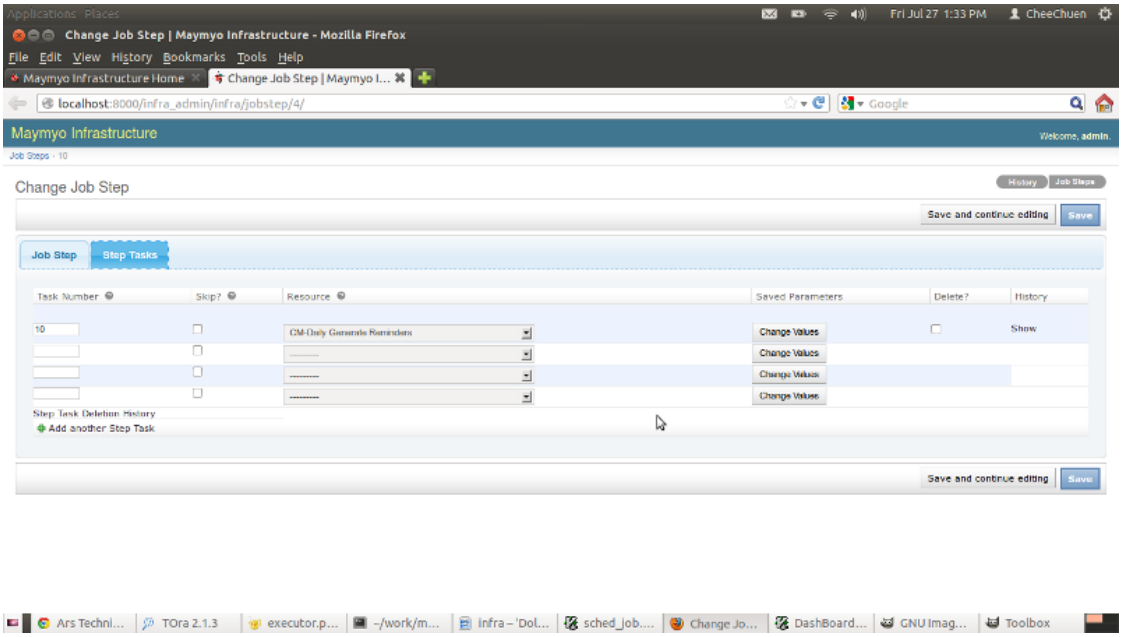
Each Step is identified by its unique (within the Job) Step Number, preferably in increments of 10 so that you can squeeze in a step that you overlooked later. You may also check the *Skip?* checkbox to temporarily prevent a Step from executing. The order of running of the Steps are by its Step Number, lowest will be run first.

Each Step may have a *Dynamic Condition* attached to control its execution. An example on how this would be useful is when you have a combined Job that is run daily but has separate Steps that run depending on the type of day it is, eg End of Week, Month or Quarter. You can then maintain one Job rather than separate End of Day, Week, Month and Quarter Jobs (and repeat with different combinations because some days may be a end of Day and Week, while others may be end of Day, Week and Month).

You can use the *Condition Keys* the same way as described for Application Job. Finally, the *Abort Job* flag can be used to stop execution of the whole Job when the Condition fails for a particular Step. All subsequent Steps will be ignored.

Adding Tasks to a Step

For each Job Step, you should add at least one Task. Each Task may be an Application Command or Report.

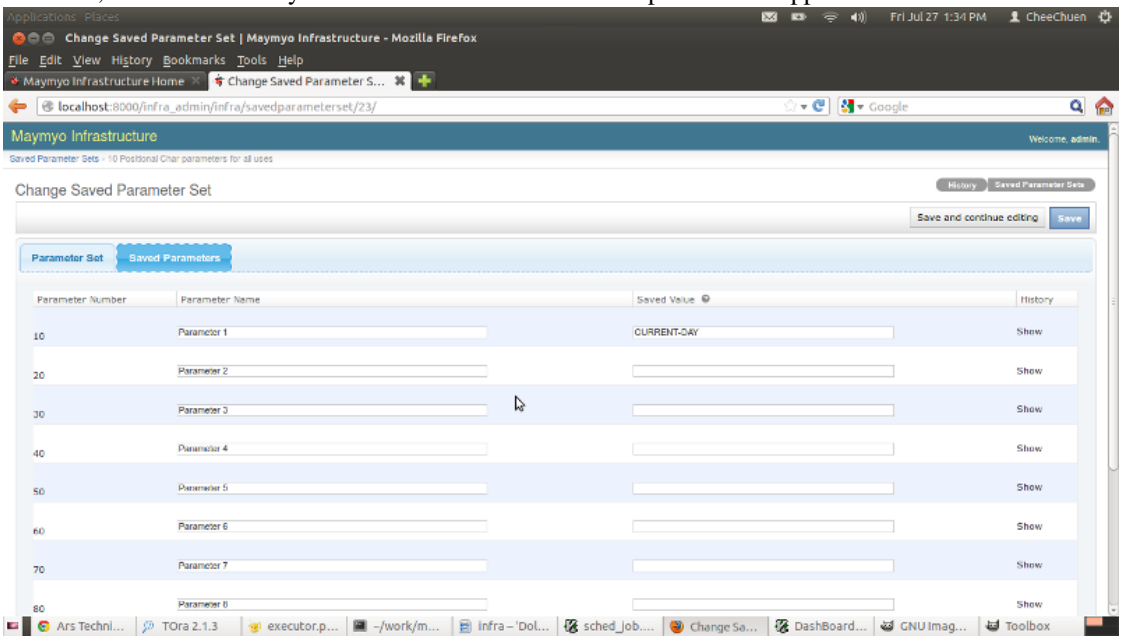


Maintain Step Tasks

Give your task a unique Task Number. This field determines the sequence in submitting the Tasks to the Task Queue. The *Skip?* checkbox allows you to temporarily omit a Task from being executed. The *Resource* field is where you select which Command or Report to run. This is mandatory.

Attaching Saved Parameters to a Task

Since a Task in a Scheduled Job will run without human intervention, we need a way to input parameter values (especially date values) to the Task. The *Saved Parameters* button (that says *Change Values*) allows you to do this. However, this button is only clickable if the Command or Report has an Application Parameter Set defined.



Maintain Saved Parameters

Maymyo will auto-copy all Parameters (of the Parameter Set of the Command or Report) into a set of Saved Parameters. These will be displayed for you to update when you click the *Change Values* button. The above shows the *Generate Daily Reminders* Task which can accept up to 10 positional parameters. The first parameter we pass in is a special token *CURRENT-DAY*. This is a Maymyo Date token that will be replaced with the current date in ISO format, ie YYYY-MM-DD, when the Task is run.

Besides Date Tokens, you can use String or Numeric literals as Saved Parameters. This must match the parameter expected by the Command or Report.

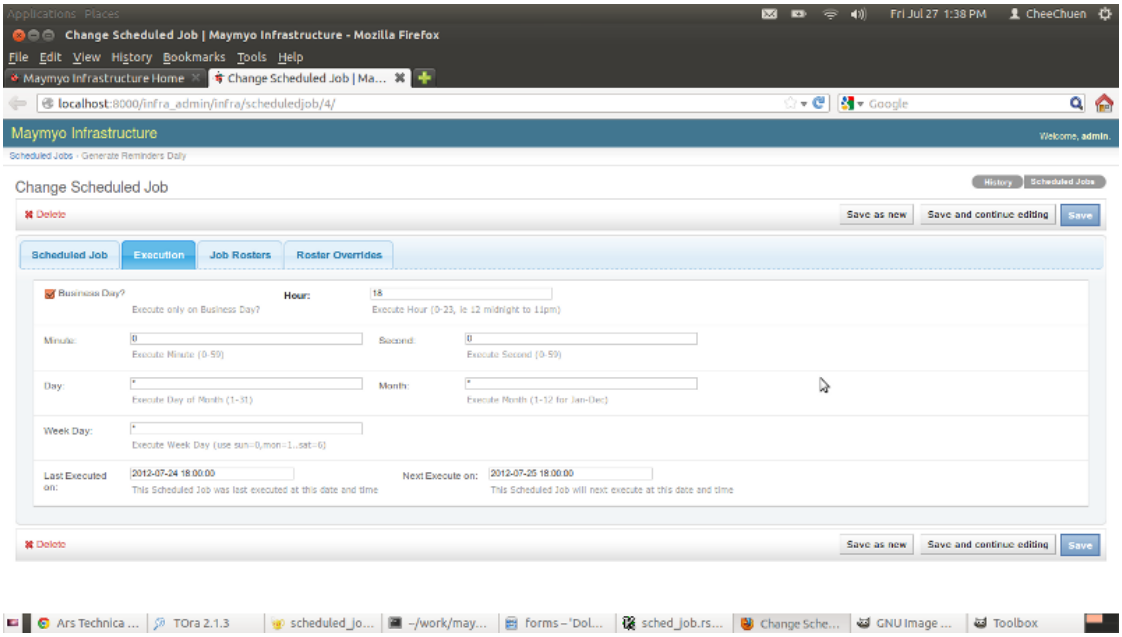
List of Date Tokens

This is the list of Date tokens that you can use for Saved Parameters. They are members of the *DATE-TOKENS* Value Set. You can add your own Date Tokens to this Value Set.

1. *BUSINESS-DAY* : Return Today if is a Business Day, else previous Business Day. eg if today is Saturday (ie a Rest Day), then will return Friday's Date. If current time is before the normal Start Business Time (AppRegistry's *START-BUSINESS-TIME* defaults to 6am), then revert to previous Business Day too.
2. *CURRENT-DAY* : Return Today regardless of Rest Day, Holiday or Time of day..
3. *NEXT-BUSINESS-DAY*, *PREVIOUS-BUSINESS-DAY*, *NEXT-DAY*, *PREVIOUS-DAY* are variations of the above 2 tokens. You can add more tokens for different plus (NEXT) or minus (PREVIOUS) number of days from current day.
4. *MONTH-FIRST-BUSINESS-DAY* : Return the current month's first business day.
5. *MONTH-LAST-BUSINESS-DAY* : Return the last business day of current month.
6. *MONTH-FIRST-DAY* : Return first calendar day of the month, ie the 1st.
7. *MONTH-LAST-DAY* : Return last calendar day of the month, ie the 30th, 31st or 28th (or 29th) for February.
8. *YEAR-XXXXXX-DAY* : Similar to *MONTH-XXXXXX-DAY*.
9. *QUARTER-FIRST-DAY*: Return the first calendar day of current quarter, eg 1st April.
10. *QUARTER-LAST-DAY*: Return the last calendar day of current quarter, eg 30th June.

4.10.3 Scheduling your Application Job

After you have defined the Tasks for your Job, you can schedule it for execution.



Maintain Scheduled Jobs

Give your Scheduled Job a helpful description and select the Application Job you have defined earlier. The *Job Owner* will default to you, ie whoever added this Scheduled Job.

If your Job executes only on a business day, then you may want to choose a *Calendar* to use. Otherwise it will use the Application default. Clicking *Skip?* will allow you to temporarily skip the execution of this Scheduled Job.

The *Effective From* date controls when the Scheduler will start considering this Scheduled Job for execution. If left blank, then it will never be executed. If you enter an earlier date for a non-repeated job, then it will also never be executed. This is because the Scheduler works in the current day and time, no back-dating.

The *Expire On* date tells the Scheduler when to stop executing this Job. Use this only for *Repeat?* jobs. When you check the *Repeat?* checkbox, then the Scheduler will consider the Job for execution from Effective to Expiry dates. When *Expire On* is left blank, then it will execute forever.

Defining the Execution Time of your Job

This is in the *Execution* tab as shown above. At the very minimum, you must specify the *Hour* when your job is to be executed. For non-repeated jobs, you should enter a time in the future, otherwise your job will never be executed. The following are the time fields that you can enter :-

1. *Business Day?* : When checked, the job will only be executed during business days, using the Calendar of the Scheduled Job or Application default.
2. *Hour* : Mandatory field, you must define at the very least which hour you want to run your job. This field uses the 24 hour format, ie 0 to 23 where 0 means 12 midnight.
3. *Minute* : 0 to 59, defaults to 0 when blank.
4. *Second* : 0 to 59, defaults to 0 when blank.
5. *Day* : 1 to 31, ie the day of the month. You can also use Date Tokens, eg *CURRENT-DAY*. When token used, then the *Month* and *Week Day* fields are ignored. Defaults to the * wildcard, ie every day of the month.
6. *Month* : 1 to 12 for Jan to Dec. Defaults to the * wildcard, ie every month of the year.

7. *Week Day* : Day of Week, Sunday starts as 0, Monday is 1 and Saturday is 6. Defaults to the * wildcard, ie every day of the week.

List, Range and Skip values

For all the time fields above, you can also use List and Range (with optional Skip values).

- List : a comma separated list of values, eg if you want to execute a job at 6am, 8am and 1pm, then enter "6,8,13" in the *Hour* field.
- Range : a range of 2 numbers (with a - in the middle), eg "6-12" for the *Day* field means you want to execute on the 6th, 7th, 8th, 9th, 10th, 11th and 12th of the Month.
- Range with Skip: a range of 2 numbers using the Skip value to increment, eg "6-12/2" in the *Day* field means 6th, 8th, 10th and 12th of the Month.

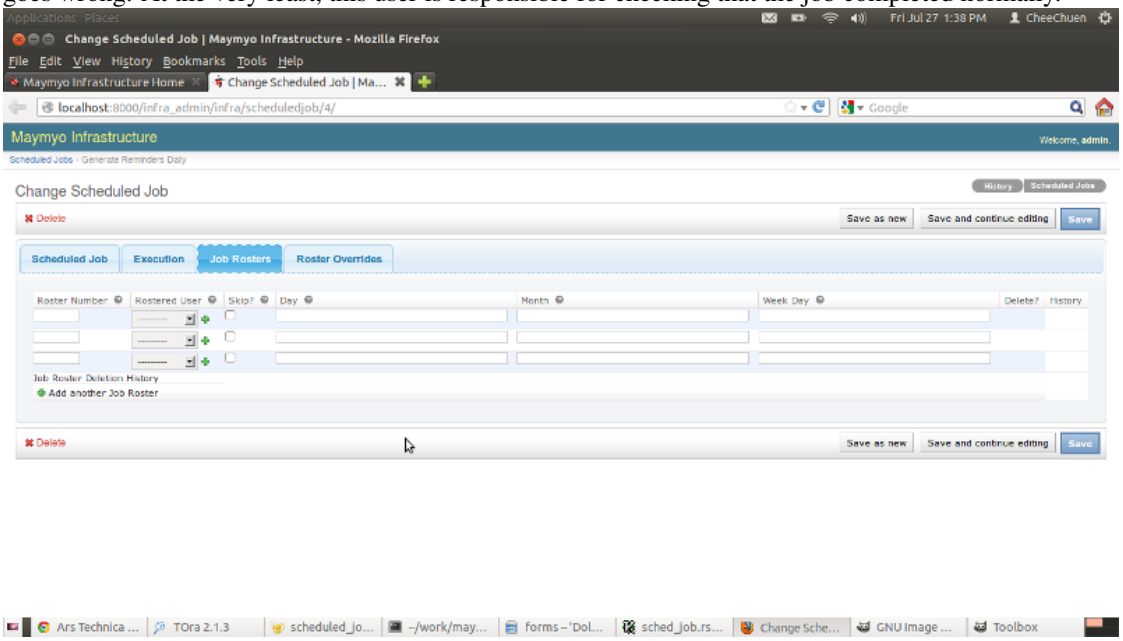
Day of Month semantics

The *Day*, *Week Day* and *Business Day* fields are considered together. If any one does not agree with the other(s), then the Scheduler will move on to the next computed day.

For example, *Day* may be "1-7" and *Week Day* is "1" means execute only on the Monday that falls between the 1st and 7th of the Month, ie the first Monday. And if *Business Day?* is also checked, then it will execute only when the first Monday is a business day. If you want the First Business Day of the month, you can use the Date Token *MONTH-FIRST-BUSINESS-DAY*.

Job Rosters

If your Job is an unattended closing that performs mission-critical batch processing and reporting, then you will want to create a Roster of users who will standby (with mobile phones or email client) to receive Alerts when something goes wrong. At the very least, this user is responsible for checking that the job completed normally.



Maintain Job Rosters

The Roster works on a daily, monthly and day of week combinations. You specify the *Roster Number* sequence of a *Rostered User* which will be used as the order of evaluation (in case many users are rostered on the same day, then the first one found is returned).

You enter the *Day*, *Month* and *Week Day* the same way are *above*.

If you want to have a weekly roster, where *UserA* is rostered on Monday, Wednesday and Friday and *UserB* on Tuesday and Thursdays (the job runs on business days only), then use :-

User	Day	Month	Week Day
UserA	blank	blank	1,3,5
UserB	blank	blank	2,4

If you want to make it fairer, then swap them forthnightly :-

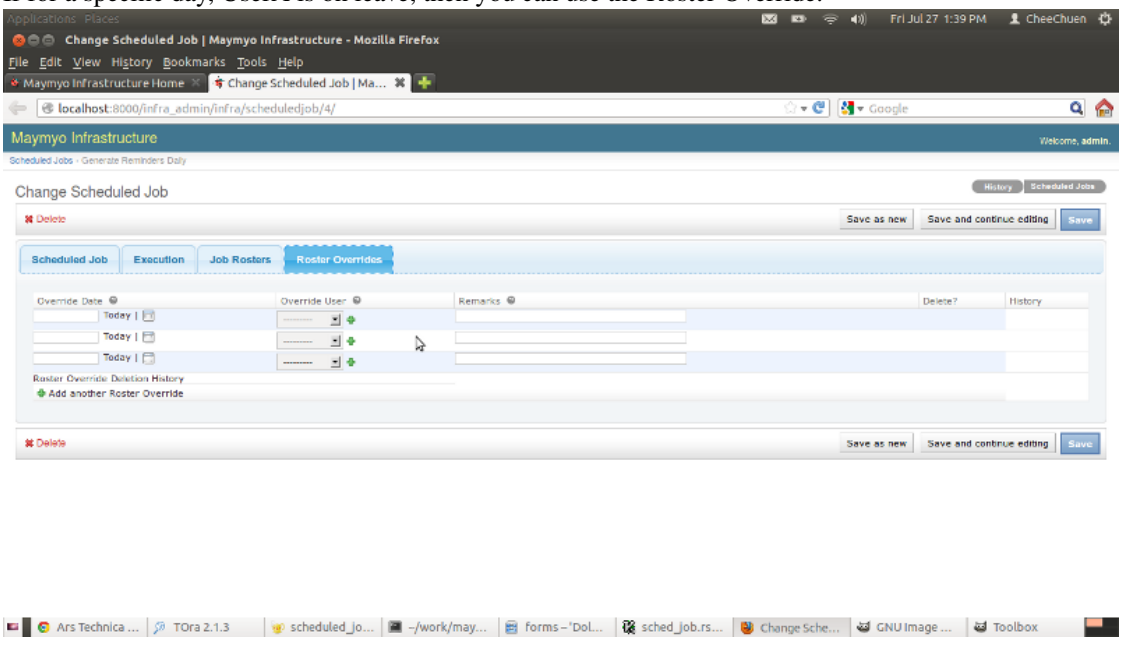
User	Day	Month	Week Day
UserA	1-14	blank	1,3,5
UserA	15-31	blank	2,4
UserB	1-14	blank	2,4
UserB	15-31	blank	1,3,5

blank means leave the field blank.

If there are *holes* in your roster, ie no rostered user found for a specific day, then we will use the *Job Owner* (of the Scheduled Job, not the Application Job) as the rostered user.

Roster Overrides

If for a specific day, UserA is on leave, then you can use the Roster Override.



Maintain Roster Overrides

Just enter the *Override Date* and select the *Override User*. The *Remarks* can be used to record the reason for the override.

4.11 Automatic Output spooling to printers and archiving of spool files

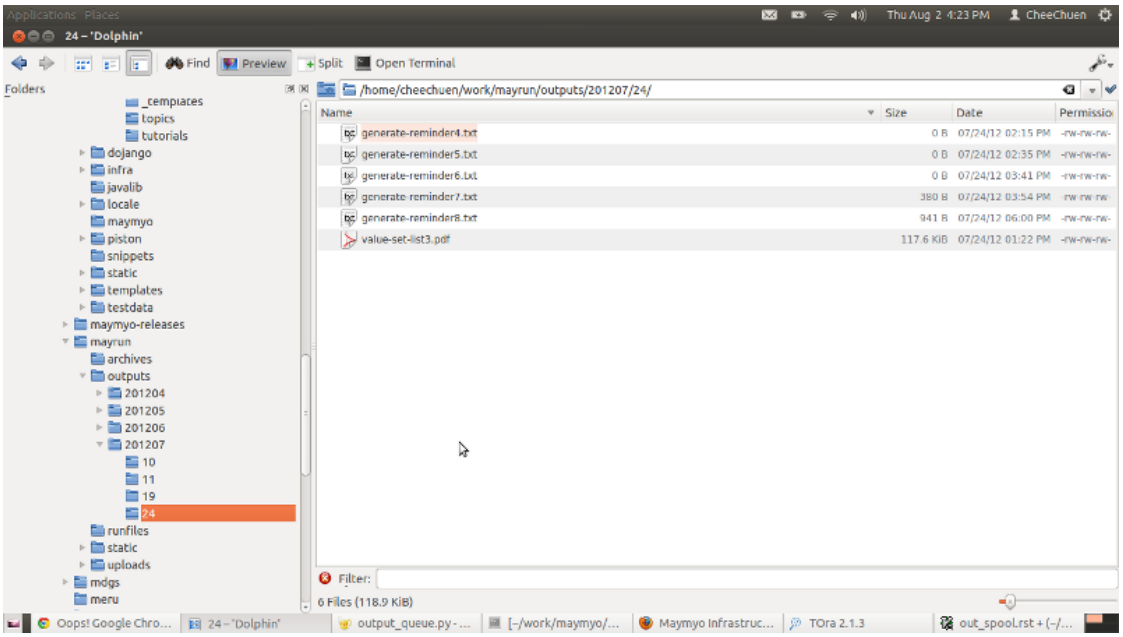
Application Reports that run on the Task Queue will generate output which are saved (in the run directory on the server where Maymyo Daemons are running) and may need to be printed (by our automatic spooling daemon).

Reports can be sent to the Task Queue by the user by pressing the *Queue* button when running a report from the Dashboard. It can also be sent to the Task Queue by a *Scheduled Job*.

Application Commands can generate outputs that are saved in the same run directory too. Anything that is printed to the standard output are saved in an output file. Unlike Report outputs, these output files may only be viewed or saved by the browser using the Dashboard's "View your Tasks" button. They will not be automatically printed. Your Commands can log debug information to be viewed later.

When an Application Report is run interactively, then its output is sent (as an HttpResponse) back to the browser, which will ask to open or save (on the user's computer) the output. The user can then view or print them using printers connected to their own computer. So spooling does not involve these outputs as Maymyo does not save interactive Report's output.

4.11.1 Layout of the output directory



Layout of output directory

The above shows the directories containing the outputs of Reports and Commands. All Queued Task will be assigned a handle to an output file in this directory. A Command need not be aware of this handle as the Task Queue will redirect all its standard output to this file (its errors will go to the Task Queue's log). A Report's Reporting Engine will be passed this handle to save its output.

The directory *mayrun* is a sibling of *maymyo_home* and is specified in the *settings.py*. Under it will be the *outputs* directory where output of all queued Reports are saved. There are 2 levels of sub-directories. The first directory is named using the month the Report was run, in *YYYYMM* format. The second directory is named using the day of month the Report was run, ie 01 to 31.

The output files are named using the Application Report's Resource Code (in lower-case) appended with the Queued Task id. Its suffix follows its Output Format type, eg for TEXT is .txt. (See the *REPORT-OUTPUT-FORMATS* Value

Set's Attribute 1).

4.11.2 Output Queues and automatic spooling

The automatic spooling daemon will read all queued Report output as soon as they are generated and send them to their designated printer (as long as that printer is Online and have the correct Paper mounted). If the designated printer is not ready, then the output will be held until somebody change the Output Queue's Online status and Paper mounted.

Each Report can have an Output Queue defined, overriding the default of its Reporting Engine. When both Output Queues are not defined (ie at Report and Engine level), then the Application Default will be used.

Printers are pointed to by Output Queues. Each Output Queue will point to an Operating System printer device (on the server where the daemon is running). The Output Queue will have a Paper (ie type of paper or form) currently mounted and an Online status. A Report must define what sort of Paper it needs to use when being printed (using its Output Format attribute 4).

You can define a null printer using the OS null device (/dev/null or NUL on Windows). All Report output sent to this printer will not be printed. This is useful if you want your output to be archived for viewing only. In fact, the *NULL-PRINT* Output Queue is already provided by Maymyo.

4.11.3 Standard Output Queues installed as fixtures

These are the standard Output Queues provided by Maymyo :-

1. ***USER-QUEUE* - Output Queue assigned to the logged in User,** as maintained in her User Profile. Some Reports should be printed to the printer of the User who generated them.

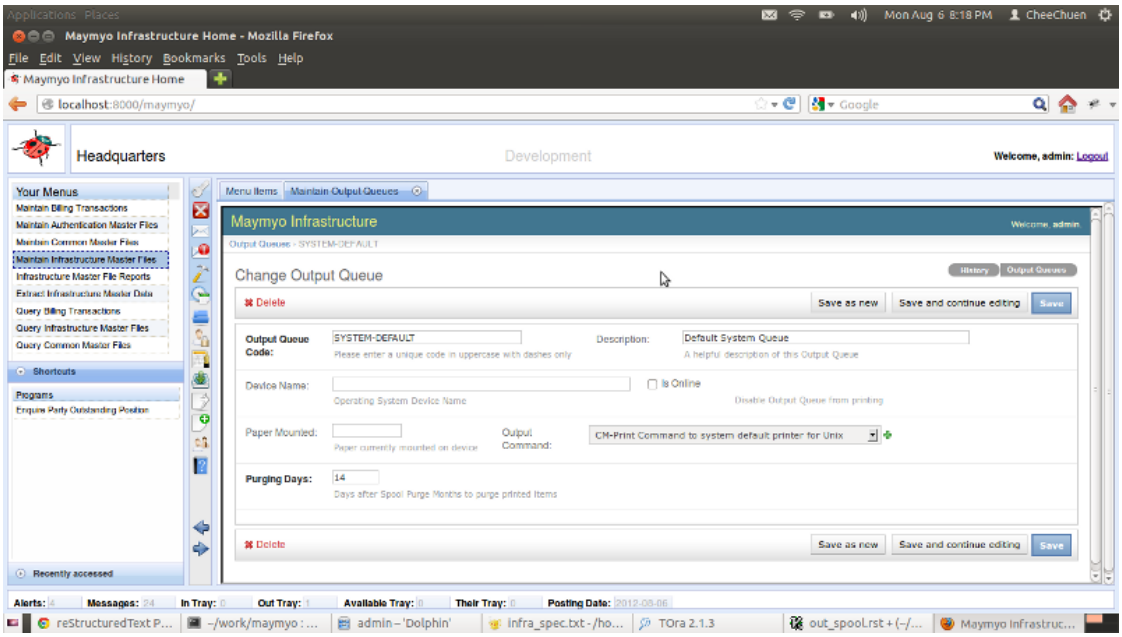
   Note: the printer device name must be defined to the server where our Spooling Daemon runs. If it is a printer connected to the User's computer, then you need to use *samba* (or Windows Network) to connect it to the server.

   ---

2. ***WORKSTATION-QUEUE* - similar to the User Queue but this will** use the printer attached to the workstation the user is logged into. The Value Set *WORKSTATION-MAPPINGS* defines the Output Queue. Attribute 2 is the Output Queue's code. This printer device must be visible to the server where the Spooling Daemon runs.
3. ***SYSTEM-DEFAULT* - Default printer on the server where the spooler** daemon runs. Any Report which do not specify any Output Queue (at Report or Engine level) will use this printer.
4. ***NULL-PRINT* - This is the null device printer which will never** print anything.

You would use *USER-QUEUE* or *WORKSTATION-QUEUE* to automatically print reports as they are completed without any human intervention. The printer to use will depend on the User or the Workstation she is logged in to, For example, a Cashier can print a Receipt automatically to the printer attached to her computer when she posts her transaction.

4.11.4 Defining Output Queues



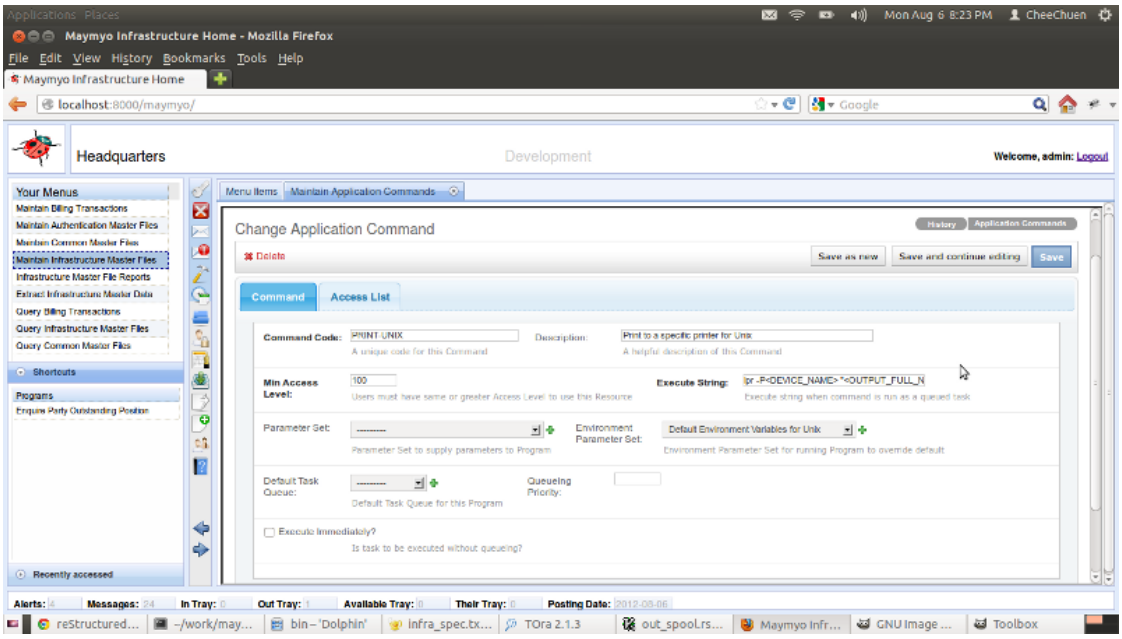
Maintain Output Queues

An Output Queue is always linked to a printer device on the server where the Spooler Daemon is running. This is the *Device Name* field. This printer device must be spelled exactly as defined to the Operating System. You should link printers to Output Queues on a one to one basis.

Is Online and *Paper Mounted* should be used by the operator of the printer, to inform the Spooler that it is ready to print spool items.

The *Purging Days* will be added to the default Spool Purge Months (as maintained in the Application Registry) before printed spool items are purged.

Defining Output Commands



Maintain Output Commands

We will use the *Output Command* to create the actual command to send the output file to the printer device. The default command provided by Maymyo should be adequate. If you have unique commands, you can always add your own Commands. Your *Execute String* can include the following tokens, which will be replaced with the actual Queued Task or Spool Item values just before running.

1. ***<INSTANCE_NAME>* - Maymyo Instance name, eg Development, Production.** as maintained in your AppRegistry *APP-INSTANCE-NAME*. Maymyo's reports will accept this as a parameter to print in the background. This will allow users to differentiate reports printed from their Production vs Test instances. However, you do not use this token in the Output Command (because it is too late, the report output has been generated. The right way is to define this as a parameter to the report).
2. ***<QUEUED_TASK>* - the Queued Task id, use this if your command needs to** read or write the Queued Task instance.
3. <*APP_SESSION*> - the connected session's id
4. <*TASK_OWNER*> - the Task Owner's username
5. <*APP_RESOURCE*> - the AppResource id of the Report
6. ***<OUTPUT_DIRECTORY>* - the absolute directory where the output file is,** normally will be *may_run*/outputs/YYYY/MM*.
7. <*OUTPUT_FILE*> - the output file name, usually Report code + Queued Task id + Output Format suffix (eg .pdf)
8. ***<OUTPUT_FULL_NAME>* - the output directory plus file name, this is the token** that you need to use to send the file to your printer device.
9. <*SPOOL_ITEM*> - the Spool Item id
10. ***<ARCHIVE_DIRECTORY>* - the absolute archive directory (will be blank for non-** archived Reports) where archive file is. The YYYY/MM is based on the archival date, currently the first Business Day of the month.

11. *<ARCHIVE_FILE>* - the archive file name. This is the Output File name plus the compression tool's suffix, currently .zip.
12. *<ARCHIVE_FULL_NAME>* - the archive directory plus file name.
13. ***<DEVICE_NAME>* - the printer device name, as read from the Output Queue's** device name. You will need this in your command.
14. *<PAPER>* - the Paper required by the Report. You may need this in your command.

An example of how we define the *Execute String* is the Command *UNIX-PRINT*:

```
lpr -P<DEVICE_NAME> "<OUTPUT_FULL_NAME>"
```

The above will use the *lpr* command to send the full output file name (absolute path name) to the printer device name. You can do more for your printer device that accepts other command options, eg line printers that can accept page length or character per inch options.

The other field that you should use is the *Environment Parameter Set* to export before running the *Execute String* above. If you leave this empty, it will use the Application default, as used by the Spooler daemon.

4.11.5 Archiving of output files

If your Report is important enough to be saved permanently, then check the *To Archive* checkbox in "Maintain Application Reports".

The archival of output files will run once a month and save compressed output files to the *archives* directory. This is a sibling of the *outputs* directory.

Output files are purged in 3 months (default setting which can be changed) and their corresponding Spool Item instances are purged in 6 months, so you can reprint archive spool items up to 6 months. After this, you will have to go to the *archives* directory directly to retrieve the archive files.

4.11.6 Purging of output files

Report output files will be purged after 3 months. This is the default setting in the Application Registry. This task is run in the *PURGE-INFRA-MODELS* job (together with purging of Maymyo *infra* model instances) that is run on the last business day of every month. This is the default as loaded by our fixtures and can be changed.

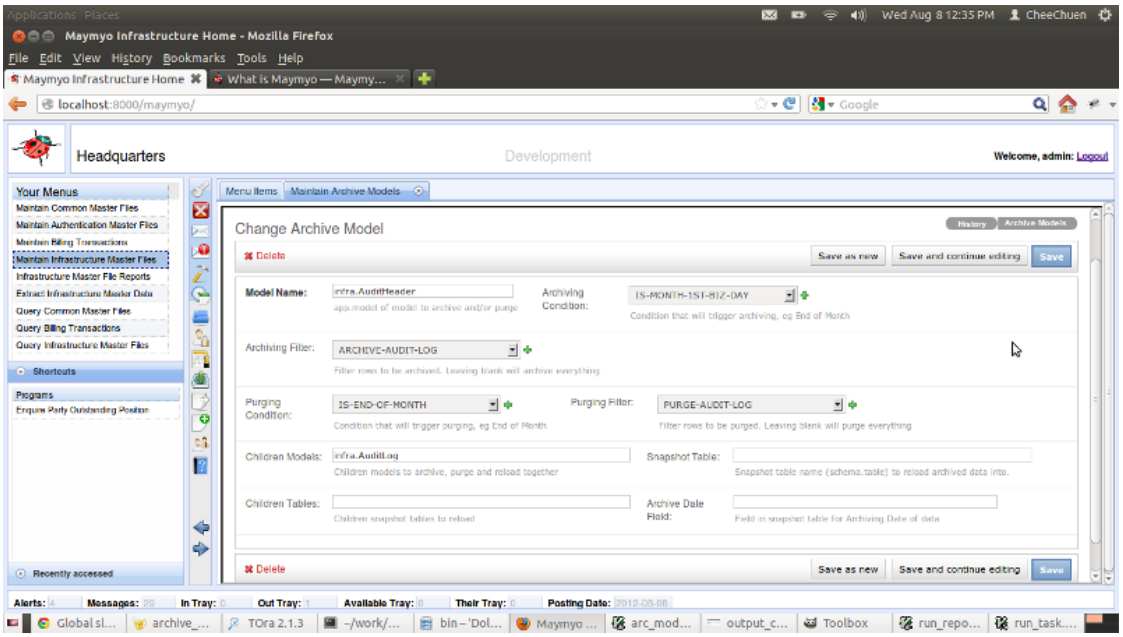
You may disable this Scheduled Job and add its Task to your own closing Job.

4.12 Automatic periodic archiving and purging of your Models

Business applications that generates transactional data need to perform periodic housekeeping to optimise between database space usage and performance. Maymyo allows you to archive, purge and reload (to snapshot tables) your transactional data in your models automatically (using *Scheduled Jobs*).

However, database tables that go through frequent inserts and deletes will become fragmented over time. You should also perform periodic database housekeeping in conjunction with our archiving and purging.

4.12.1 Defining your model's archiving and purging criteria



Maintain Archive Models

You can archive and purge your models together with its children models (joined by a Foreign Key field of your model). Each Archive Model is identified uniquely by its *Model Name*, ie its *app_label.model_name*.

First you have to decide when and which rows to archive or purge by using Dynamic Conditions and Filters. The *Archiving Condition* and *Archiving Filter* will decide when and which rows to archive. In the example above, the *infra.AuditHeader* and its child *infra.AuditLog* are archived on the First Business Day of the Month. The rows to be archived are found in the predicate returned by the *ARCHIVE-AUDIT-LOG* Dynamic Filter, which we show below:

```
def get_predicates(model_instances, params):
    from infra.objects.app_registry import get_posting_date
    from infra.objects.app_calendar import add_months
    # Archive previous month's data (not this month because data is still being updated)
    archive_date = add_months(get_posting_date(), -1)
    return {'changed_on__year': archive_date.year, 'changed_on__month': archive_date.month}
```

The predicate returned by the Filter above tells us that the *AuditHeader* rows to be archived are those whose *changed_on* date falls in the previous month (using the current posting date). This predicate is applied only on the model *infra.AuditHeader*. All its children *AuditLog* rows will also be archived and output into the same archive file.

The purging of *AuditHeader* happens at the End of the Month, ie the last business day of the month. The rows to be purged are *changed_on* more than 12 months ago (from current posting date):

```
def get_predicates(model_instances, params):
    from infra.objects.app_registry import get_posting_date
    from infra.objects.app_calendar import add_months, first_day
    from infra.objects.app_datetimes import to_aware_date
    # 12 month window, ie purge anything older
    purge_date = first_day(add_months(get_posting_date(), -12))
    purge_on = to_aware_date(purge_date)
    return {'changed_on__lt': purge_on}
```

If you have more than one *Children Models*, then use comma to separate them.

Note: You can choose to purge without archiving or vice-versa. You should ensure that the rows you purge has been archived previously, ie the purging window should be longer.

The *Snapshot Table* field specifies the database table, using *schema.table_name* format, to reload previously archived rows to. Archived children data are reloaded to *Children Tables*. The snapshot tables are clones of the archived models (parent and its children), except that they must have an additional field for the *Archive Date field* (ie the archiving date, which is the First Business Day of the Month). If you were to write reports against these snapshot tables, make sure your where clause has the archive date.

If you archive a parent and 2 children models together, then you must provide 3 snapshot tables to reload the archive data to, otherwise the reloading will fail.

The django view to reload archive data to snapshot tables are still in our TODO list. However the function to do so is in *infra.objects.archive_model.perform_reloading* and you may use it in your own Application Commands.

Where are archive files kept?

Archive files are kept in the same directory as *Archived Spool files*. The date of the archive sub-directories follows the archival date, which is the First Business Day of the Month.

Archiving, Purging and Reloading History

Every time an archiving and purging is performed, an entry is made in the Archiving History (*infra.ArchivingHistory*) and Purging History (*infra.PurgingHistory*) respectively.

The former is also used to select archived data to reload to the snapshot tables. A successful reloading will also create a Reloading History (*infra.ReloadingHistory*) instance.

Currently, these are log tables that are accessible only directly from the database. Creating views for them are in our TODO list.

4.12.2 The Archiving and Purging of Maymyo models

These are Application Jobs provided by our fixtures data. They are *ARCHIVE-INFRA-MODELS* and *PURGE-INFRA-MODELS*. We currently schedule the former on the First Business Day and the latter on the Last Business Day of the month respectively.

You can append your models' archiving and purging to our Jobs or create your own.

4.13 Designer defined Application Events and its delivery

Application Events are triggered when something happens in Maymyo. This something is designed by you. It should be a business event that is of importance to your users, for example whenever a payment more than X amount is made.

An Event will have an Access Key and up to 5 Attributes values, defined by the Designer. The Key is mandatory but you can always use the ALL-ROWS wildcard to make it redundant. The Access Key allows your Registrants to register with different Attribute Values, eg if your Access Key is by Business Unit, then a Registrant can register for an Event with different Attribute values for each Business Unit.

The end result of an Event is a Message sent to interested Registrants. This is what is meant by Delivery. It works on a Publisher-Subscriber basis, where the Publisher is Maymyo (through the Event that the Designer raised) and the Subscribers are Registrants who register their interest in an Event for a set of Key and Attribute values.

The Event Access Key and Attribute values (ie the actual values associated with an actual event incident) is compared with the Registrants' Access Key and Attribute values. If they do not match, then no Message will be sent to the Registrant. This allows Registrants to be informed only when certain Attribute are above a specific value, usually monetary in nature.

The Message can also be linked to an Action URL that the Registrant can click to see details of the event (in her Dashboard). Our Business Rule events have Action URLs that allows the Recipient to see the details of the transaction Selection.

How the Message is delivered to the Registrant depends on the Message Type defined for the Application Event. It can be queued to the Dashboard, sent as an Alert and emailed or sms'ed to mobile phones. See *Messaging and Alerts*.

4.13.1 A Maymyo Event example

In Maymyo, all Business Rules will trigger an Application Event whenever an Authorisation is requested (by a User who does not have sufficient permission).

In our Billing module, we have the Billing Entry transaction. A User may enter Debit or Credit Bills against a Party. For Credit Bills, we have a Business Rule CREDIT-BILL-RULE which controls who can enter Credit Bills and up to what amount. A Business Rule can trigger an Application Event using the same Access Key and Attribute values whenever an Authorisation request is raised. (This means that no event is triggered if a user has sufficient permission to perform the transaction).

The Access Key for this Business Rule is the Business Unit Code of the Party being billed. The 1st Attribute is the Billing Type Code used (eg Trade Discount, Cash Discount etc) and the 2nd Attribute is the Billing Amount (absolute amount in Functional Currency). The 1st Attribute uses the IN operator (set membership) while the 2nd uses "<", ie less than.

In tandem with this, we also have a special User called Trivial User. Whatever Access and Attribute values granted to this special User is also granted to everybody. This means that Maymyo should allow everyone to enter Credit Bills that the Trivial User can enter and no Application Events will be triggered. This allows you to define a lower threshold to Business Rules and Events below which every one is given permission automatically.

The Access Key allows you to control different Attribute values for different Business Units. For example, for Head-Quarters, its Users can enter Credit Bill Amounts up to 10k without restriction (by using the Trivial User, see below). Only Finance Managers can enter up to 100k. Then in the other Branches, you want to set these amounts to 5k and 50k respectively. You should define the following Access List :-

1. Define Access List against the Trivial User in "Maintain User Profiles" as follows :-
    - Access Key HQ with Attribute 1 ALL-ROWS and Attribute 2 10k
    - Access Key ALL-ROWS with Attribute 1 ALL-ROWS and Attribute 2 5k This applies to all other Branches except HQ.
2. Define Access List for Application Role "Finance Manager" in "Maintain Application Roles" as follows :-
    - Access Key HQ with Attribute 1 ALL-ROWS and Attribute 2 100k
    - Access Key ALL-ROWS with Attribute 1 ALL-ROWS and Attribute 2 50k This applies to all other Branches except HQ.
3. Define Access list for General Manager, Finance user in "Maintain User Profiles" :-
    - Access Key ALL-ROWS with Attribute 1 ALL-ROWS and Attribute 2 ALL-ROWS

No 1 above will apply to all Users except those with the "Finance Manager" Role while No 2 will apply to the latter only. You should always define a super-user (ie General Manager) for every Business Rule otherwise transactions above 50k (or 100k for HQ) cannot be performed at all.

If you are the Chief Operating Officer, you may want to know if there are any Credit Bills above 100k being approved. You can register your interest in the CREDIT-BILL-EVENT with the following :-

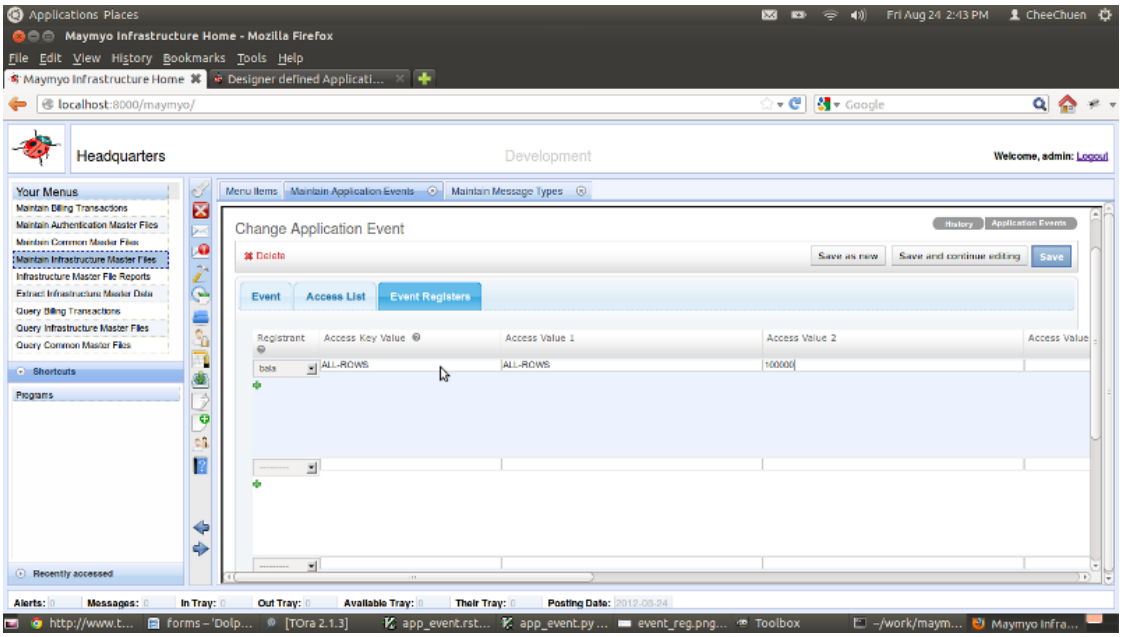
- Access Key ALL-ROWS with Attribute 1 ALL-ROWS and Attribute 2 of 100k

Whenever a CREDIT-BILL-EVENT is triggered, it will go through all the interested Registrants and send a Message to those whose Access and Attribute conditions matches. There could be other Registrants with different Attribute values who may or may not be sent Messages.

Note that when registering for an Event, if an Attribute is a numeric value, we will always toggle the operator used for the underlying Business Rule, ie when checking permission to the Business Rule we will use the "<" operator but when registering interest to an Event we will toggle it to ">=". This is controlled by the "Toggle Event" flags in "Maintain Application Events".

The mode of delivery of the Messages depends on the Event's Message Type. When not defined, we will fall back on the Application default, which is to send a queued message to the Dashboard.

4.13.2 How to register your interest in an Event



Maintain Application Events

You must first have access to the "Maintain Application Events" program in the "Maintain Infrastructure Master Files" menu. The Registrant must also have *access* to the specific Application Event, either directly through her User Profile or indirectly through her Application Role(s). You can check which fields are the Access and Attribute values in the "Maintain Application Events" Access List tab.

The "Event Registers" tab allows you to add users who are interested to be informed of an Event. Select a user in the "Registrant" field, then enter the Access Key and the Attribute values and scroll to the right-most part of the page to enter the "Starting On" date and time when the registration takes effect. The "Ending On" date and time allows registrations for a short period of time, after which the Registrant will no longer be informed.

As mentioned earlier, the operator for numeric Attributes are toggled for registrations. If you were to click on the "Event" tab, you will see that we have checked the "Toggle Event 2" checkbox. So even though the "2nd Value Name" in the "Access List" tab says "< Billing Amount", the "<" operator is used only for checking access to the Application Event. When the Event is triggered, we will decide to send a Message to the Registrant using the "Toggle Event X" checkboxes.

4.13.3 How to program a new Event

To create a new Application Event, you must first decide how it should be triggered and which fields to use as Access Key and Attribute values. Your Event will usually be associated with a Business Transaction, an Application wide event (eg an End of Day task that looks for customers whose overdue amounts has exceeded X days) or possibly an application error event, eg when an automatically uploaded file cannot be found in its usual directory by a certain time. You can see that it can be anything at all.

So, first step is to create a new Application Event in "Maintain Application Events". You should also maintain the *Access List* to your Application Event. This will control who can register for your Event.

Next, decide on how the Message will be delivered to Registrants. You can use our default Message Types or if your Message has an "Action URL" to allow the Registrant to view details of the event, then create a new Message Type. What parameters to be passed to the Action URL is decided by you, when you call our "trigger_event" function. The parameters will be passed as GET parameters (ie appended to the Action URL). The Action URL is usually relative to where your serving Maymyo, ie no need the hostname and port part. See our *BILL_ENTRY* Message Type.

Note: Action URL should be a *view* program coded by you. Please read django's documentation on views and URLs to learn how to create new *view* programs.

Finally, in your code, call our *trigger_event* function:

```
from infra.objects.app_event import trigger_event
from infra.objects.app_resource import prepare_access_params
....
your program code....
....
params = prepare_access_params([access_key_value, attribute_1_value, attribute_2_value,...])
action_params = [your_event_id]
rsr = trigger_event(app_event, user, event_message, params, action_params)
if rsr.return_status == 'E':
    # raise your own error
....
rest of your program
....
```

The parameters that you pass in to *trigger_event* are:-

1. *app_event* - ApplicationEvent instance that you created. Usually you would perform a *get* using its unique *resource_code*.
2. *user* - the User instance who triggered the event, eg if a business transaction, then the user who started it. If Application wide event in a Scheduled Job, then the Job Owner or Rostered User.
3. *event_message* - a text message describing the event, eg _("Customer %(customer_name)s has overdue Bills of %(overdue_amount)d more than %(overdue_days)d days"). Notice the "_" which is our alias for *ugettext* to support internationalisation. You should do the same if your application is internationalised.
4. *params* - this is a dictionary of the Access Key and Attribute actual values (ie actual field values of the event). You call *prepare_access_params* with a list of the actual field values to prepare the dict. The Access Key and Attributes must match what you defined for your Application Event.
5. *action_params* - if your Event's Message Type uses an "Action URL" to allow the Registrant to view details of the Event, then you should supply the parameter values as a list. This parameter is optional.

rsr is the ReturnStatusRecord returned by *trigger_event*. It is a class with *return_status*, *return_messages* list and *return_values* list. It is our standard way of returning from a non-trivial function call.

The triggering of an event will immediately broadcast the event to all Registrants. If the values of the Access Key and Attributes matches for a Registrant then she will be sent a Message.

Do not forget to add Registrants to your Event. See *previous section*.

CHAPTER

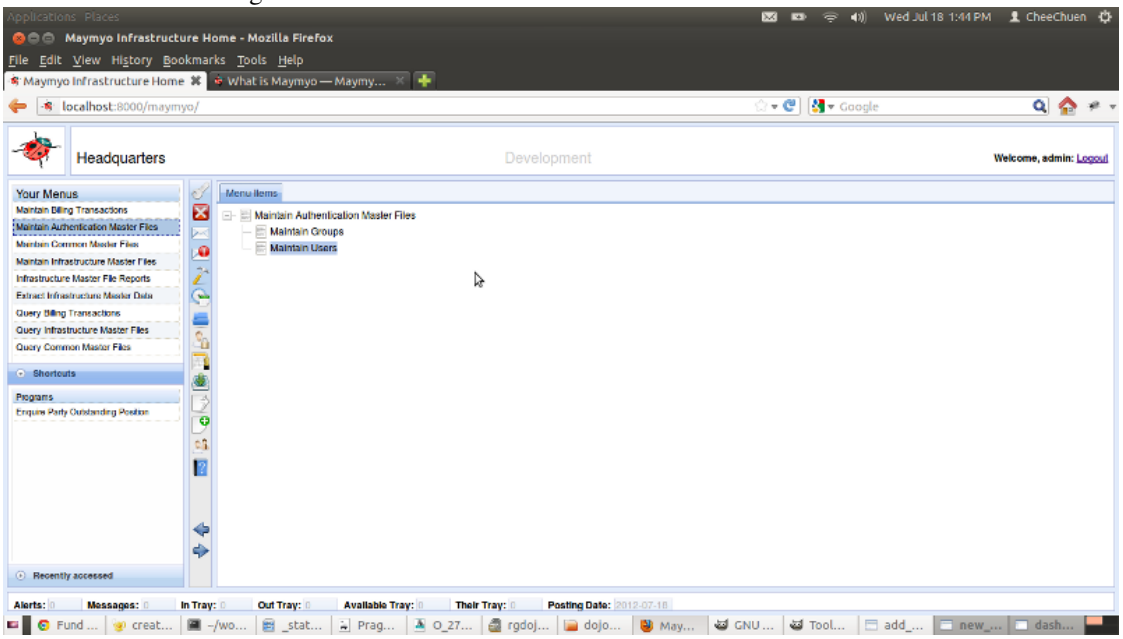
FIVE

MAYMYO TUTORIALS

Tutorials:

5.1 Let's try creating a User

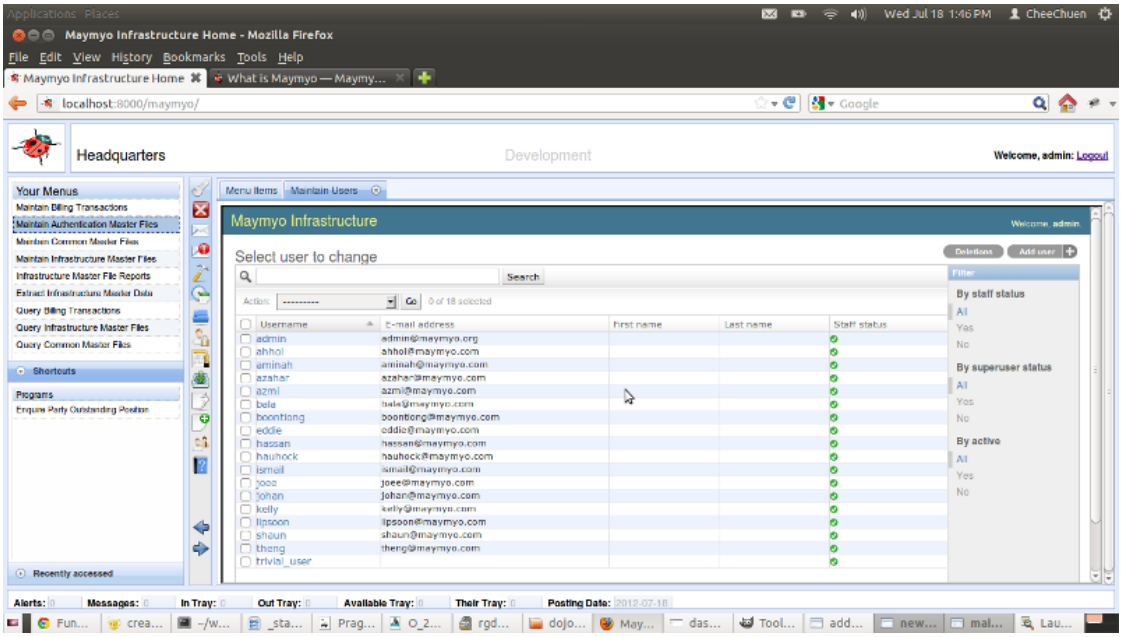
Log in as "admin" to Maymyo. If you are lost, check out *running Maymyo* in the Installation guide. After you log in, click on the "Maintain Authentication Master Files" in "Your Menus" (its in the top left of your Dashboard). You should see the following :-



Maintain Authentication Master Files menu items

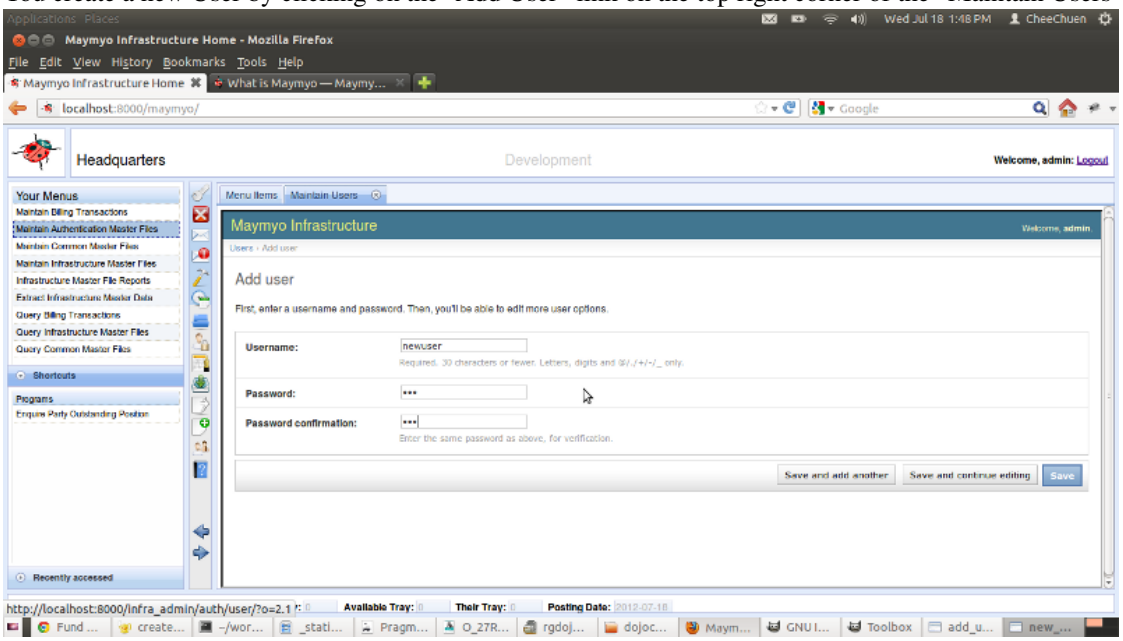
The Menu Items are displayed as a Tree. Similar to other Tree widgets that you may have used, you expand the items in a sub-tree by clicking on the "+" icon. After expanding, this icon becomes a "-" icon. Clicking "-" will collapse the sub-tree.

Double-click on the "Maintain Users" menu item.



Maintain Users - list of users

You create a new User by clicking on the "Add User" link on the top right corner of the "Maintain Users" tab.

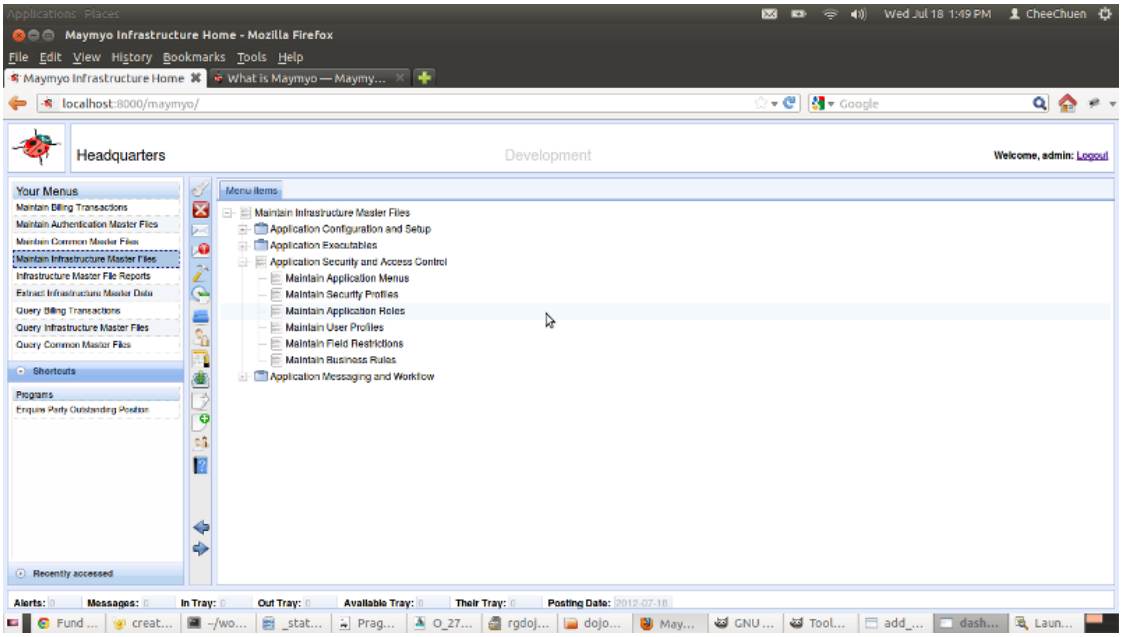


Adding a new User

You must enter :-

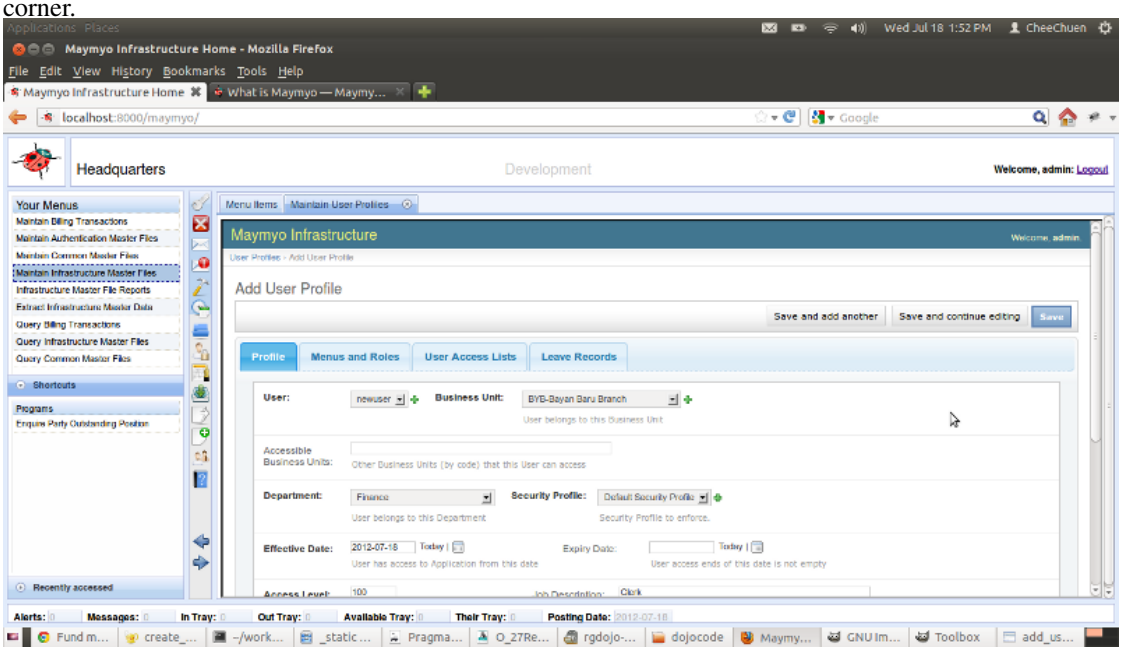
- a unique user name. In above, this is 'newuser'
- you can enter anything in the password fields. It won't be used as it will be changed later automatically when you Maintain User Profile.
- Click "Save" to create this user. This is the button on the bottom right.

Then close this tab by clicking the circled "x" (this is on the tab itself, right next to "Maintain Users") and click on the "Maintain Infrastructure Master Files" in "Your Menus". Then click on "Application Security and Access Control".



Application Security and Access Control menu items

Double-click on the "Maintain User Profiles" menu item and click on the "Add User Profile" link on the top right corner.



Adding a new User Profile

You must enter :-

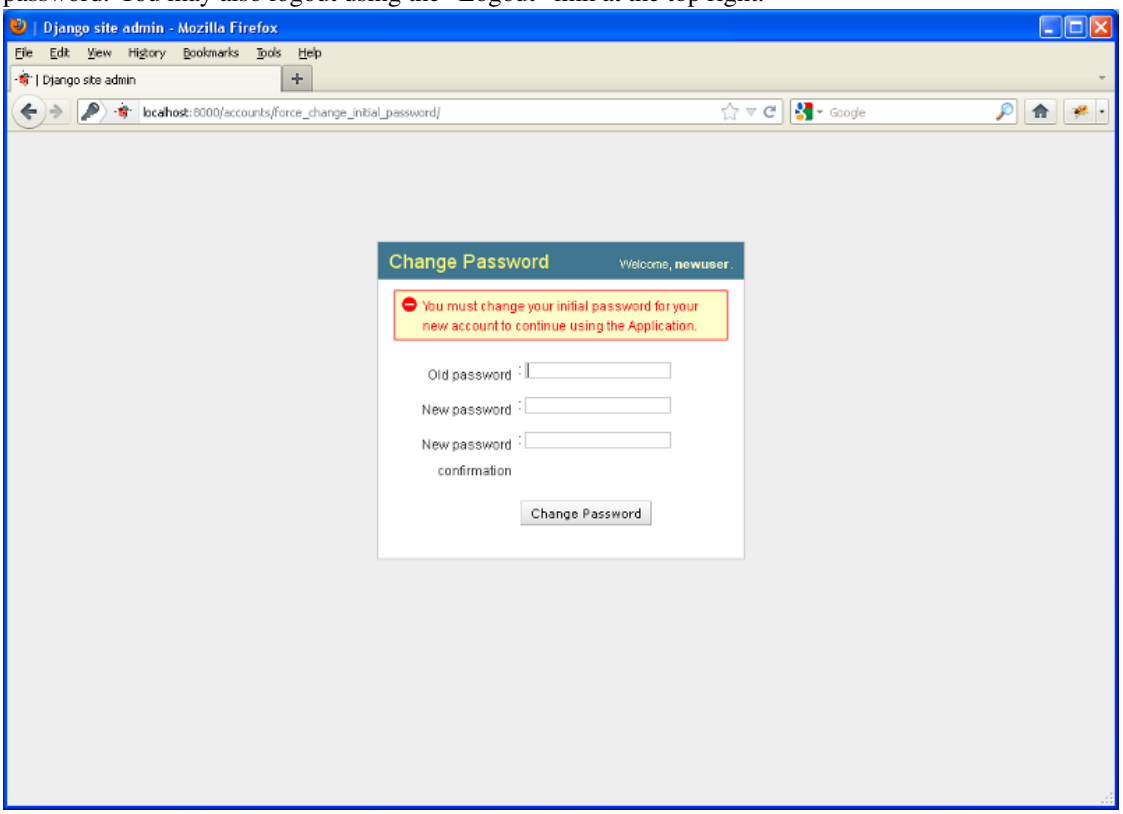
- select a new user (you have just created). The drop-down list will show only new users that do not have a User Profile.
- the Business Unit the user belongs to. Some models which has a Business Unit field can auto-filter the rows accessible for this User by the Business Unit she belongs to.
- the Department the user belongs to.

- the Effective Date when this user can start using the Application. Click on 'today' so that you can test logging in later.
- Click on "Save" to create this User Profile.

Now try logging in with this User. Note that the User's password has been changed to the Security Profile's Initial Password upon saving.

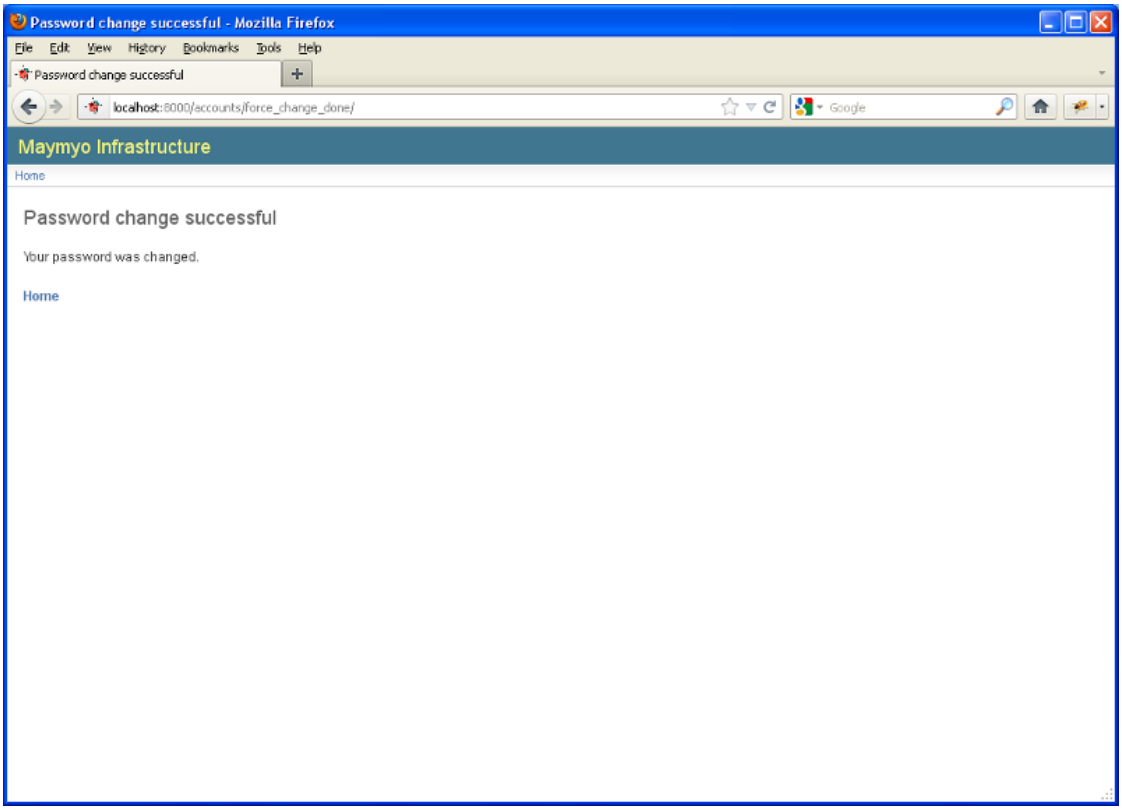
Note: The reason we do this is that the Administrator should **not** know the User's password. We do this by forcing the new User to change password when we detect that the Initial Password is used. On Installation, this Initial Password is "abc123" for the default Security Profile. You should change it in Production.

Logout as "admin" and click on the "Logout" icon (which is a red cross on the vertical toobar in the left of Menu Item tab, just hover your mouse on each icon to discover its use) and then click on the "Login again" link. You should see Maymyo's login prompt. Login with Username "newuser" and password "abc123". You will be forced to change the password. You may also logout using the "Logout" link at the top right.



You are forced to change password on first log in

Enter the old password "abc123" and then a new and different password twice. This password must be at least 6 characters and have at least 2 digits (our installation default).



Your password has been changed

Click on the "Home" link to go to the Dashboard. You will notice that the new user inherits the "Your Menus" from her Security Profile. For Production, this is not a good idea. You should create your own Security Profiles which allows access to your own application programs and leave the default Security Profile for "admin" use only. You may also assign menus directly to a user in Maintain User Profile. This will override the menus of her Security Profile.

It is now time to talk about *Access Control*.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*