Dingos Documentation

Release 0.2.1

Siemens

September 03, 2015

Contents

Contents:

CHAPTER 1

What DINGOS is all about

Dingos ("Django: INformation in Generic ObjectS") is a Django application that allows you to manage data structured in hierarchies in a generic way. It was written for dealing with cyber-threat information expressed in standards such as CybOX and STIX as part of the MANTIS Cyber Threat Information Management Framework, but may also have other applications.

Consider the following XML-based example:

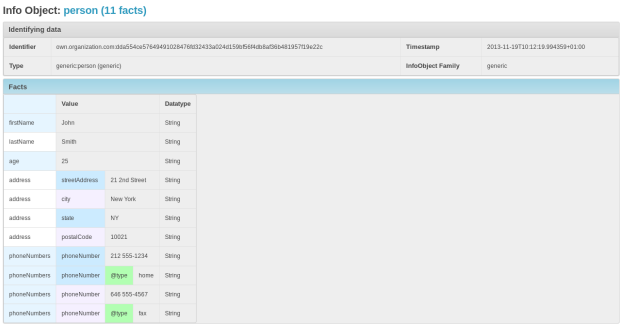
```
<person>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <age>25</age>
    <address>
        <streetAddress>21 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021</postalCode>
    </address>
    <phoneNumbers>
        <phoneNumber type="home">212 555-1234</phoneNumber>
        <phoneNumber type="fax">646 555-4567</phoneNumber>
    </phoneNumbers>
</person>
```

The generic XML import of Dingos stores this information by extracting the following "facts":

Fact Term	Fact Value
person/firstName	John
person/lastName	Smith
person/age	25
person/address/streetAddress	21 2nd Street
person/address/city	New York
person/address/state	NY
person/address/postalCode	10021
person/phoneNumbers/phoneNumber@type	home
person/phoneNumbers/phoneNumber	212 555-1234
person/phoneNumbers/phoneNumber@type	fax
person/phoneNumbers/phoneNumber	646 555-4567

This list of facts is stored as a Dingo "InfoObject". The data model also keeps track of positional information that associates the attribute 'home' with the first and attribute 'fax' with the second telephone number.

Viewing the imported file (without any further customization) currently looks like follows:



Dingos further offers:

- a configurable parser that enables Dingos to deal with rather complicated data structures such as STIX and CybOX that require extraction of embedded structures, derivation of object identifiers, etc.
- a high degree of sharing in the data model: if a piece of data occurs several times (e.g., in several imports), then it is only stored once and referenced at all occurrences.

CHAPTER 2

What DINGOS is still missing

At the current revision: **LOT'S**. Currently, Dingos support import and viewing/search of the imported data, but of course it also requires facilities for editing, authorization management etc., etc., etc.

If there are features you particularly miss and feel like contributing, please have a look at Contributing and Developers' Guide to DINGOS.

CHAPTER 3

Installation

1. Make sure that you have the required dependencies on OS level for building the XML-related packages. For example, on an Ubuntu system, execute the following commands:

```
apt-get install libxml2 libxml2-dev
apt-get install python-dev libxslt1-dev
```

2. Find out the current version of `libxml2-python` by browsing to https://pypi.python.org/pypi/libxml2-python and noting done the version number (at time of writing, this was `2.6.21`).

3. Install `django-dingos` using `pip`:

```
$ pip install ftp://xmlsoft.org/libxml2/python/libxml2-python-<libxml2-python-version-nr>.tar.gz
$ pip install django-dingos
```

4. Add `dingos` and `grappelli` to your `INSTALLED_APPS` list in your settings.

5. To get started, add the `dingos` urls to your `url.py` like so:

```
urlpatterns = patterns('',
                       ...

                       url(r'^dingos/', include('dingos.urls')),

                       ...)
```

6. Dingos uses the `grappelli` application (see django-grappelli). This requires you to run the collect static command once after installing `grappelli`:

```
python manage.py collectstatic
```

7. If you are using south (and you *should* be using *south*), carry out the schemamigration for dingos:

```
python manage.py migrate dingos
```

   Otherwise (this is not recommended, because migrating to future releases of DINGOS will be a pain), run:

```
python manage.py syncdb
```

CHAPTER 4

Usage

To use Dingos, include `grappelli` and `dingos` in Django's `INSTALLED_APPS`. Make sure that you carry out the `collect static` command required for `grappelli`!

CHAPTER 5

Contributing

Contents

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways.

5.1 The issue tracker for Django DINGOS

Although django-dingos is generic, in the near future its further development will occur mostly within the further development of the Django Mantis Cyber-Threat Intelligence Management Framework. So, for the time being, please use https://github.com/siemens/django-mantis/issues as issue tracker for bugs, feature requests and other feedback regarding django-dingos.

5.2 Types of Contributions

5.2.1 Report Bugs

Report bugs at https://github.com/siemens/django-mantis/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.

- Detailed steps to reproduce the bug.

5.2.2 Fix Bugs

Look through the GitHub issues (https://github.com/siemens/django-mantis/issues) for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

5.2.3 Implement Features

Look through the GitHub issues (https://github.com/siemens/django-mantis/issues) for features. Anything tagged with "feature" is open to whoever wants to implement it.

5.2.4 Write Documentation

Djangos could always use more documentation, whether as part of the official Djangos docs, in docstrings, or even on the web in blog posts, articles, and such.

5.2.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/siemens/django-mantis/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.3 Get Started!

In your contribution, you may want to either modify/add to existing code or create a new Django application that interacts with the existing applications that are part of the Mantis framework.

DINGOS profitted a lot from the advice provided in Two Scoops of Django. Unless you are an absolute Django expert (and maybe even then), please read Daniel Greenfield's and Audrey Roy's excellent Two Scoops of Django. Even though it provides best practices for Django 1.5, most of its advice is also valid for Django 1.6, and likely to be very relevant for quite a few minor revisions to come.

5.3.1 Modifying/adding to existing code

Here's how to set up a repository for local development.

1. Fork the relevant repository repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/<repository>.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv <your_mantis_environment>
$ cd <repository_folder>
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.3.2 Writing your own Django application

Do yourself a favor and set up the directory structure of your Django application in the right way from the very start. The easiest way to do so is to use Daniel Greenfield's cookiecutter-djangopackage template (which uses Audrey Roy's excellent Cookiecutter for creating the directories): this layout has a very sensible directory structure with out-of-the-box configuration of `setup.py` for easy build, submission to PyPi, etc., as well as the start of a Sphinx documentation tree. Once you have the directory structure created, initialize a fresh git repository with it and get to work...

5.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7.

CHAPTER 6

Credits

6.1 Development Lead

- Siemens <mantis.cert@siemens.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.2.1 (2014-03-06)

- Bugfixes
  - *CRITICAL* Remediation of painfully slow import for systems with lot's of imported data

    An illformed query led to extremely slow import of new data in systems that already have lot's of data inside. This bug has been fixed.
  - Problem in link to InfoObjects in which a certain fact can be found on Unique Search Page fixed

    The link was faulty in that it carried a '&page=...' parameter that needed to be removed.
  - Long repetition of '_' in a string lead to HTML display spilling over, because '_' was not regarded as place to insert a possible line break. This has been changed.
- New/Modified views
  - View for listing *all* InfoObjects, also those used internally by DINGOS for bookkeeping (e.g., user preferences). The view is restricted to Django-superusers.
- New/Modified command-line commands
  - In 'dingos_manage_user_settings', added the ability to overwrite settings for 'ALL' users.

7.2 0.2.0 (2014-02-24)

- New base functionality
  - Added framework for managing user-specific data (user configurations, saved searches, etc.) and querying user-specific data in templates and views.
  - Added tracking of namespace information per component of a fact term
- New/Modified views
  - Modifications to all views
    - Added possibility to switch between horizontal and vertical layout ... or have automatic adjustment of the layout depending on screen width.
  - Modifications to filter views
    - Modified date-picker in filters to enable addition of timespans without changing saved searches or messing up order of timespans

        * Added several further filter criteria in InfoObject filter
    - Added view with basic and still rather restricted editing capabilities for InfoObjects – currently only used for editing user preferences or edits by the superuser
    - Added view to edit user configuration
    - Added view to edit saved searches
    - Added per-column ordering to list views
    - Added new filter/search that shows unique Facts rather than all InfoObjects containing a certain fact.
- New/added capabilities for writing views
    - Added framework for ordering list views
    - Added per-user configuration for:
        * layout (horizontal vs. vertical)
        * number of rows to show in list views
        * number of rows to show in widget displaying objects in which a displayed object is embedded
- Bug fixes / Improvements
    - Generation of filter views became unbearably slow when many (> 40,000) InfoObjects are in the system. This was, because of a badly built query within the dynamically built filter form. This has been fixed.
    - Further development of JSON export (still needs work to make the to_dict function of InfoObjects generic and configurable such as the from_dict function)
    - Fixed bug in generation of InfoObjects: when a placeholder for a given ID already existed, it was not reliably found.
- New/Modified command-line commands
    - Import command now fails gracefully if import of a file throws an exception: it continues with import of the next file.
    - Added command line arguments to basic import command:
        * ability to add IDs of marking objects to be added to imported objects
        * ability to automatically move imported XML files to other folder after import
    - Added command to reset user-settings and saved searches for a given user.
    - Added command to re-calculate object names.

      This is useful to run right after an import, recalculating the names of 'Observable' InfoObjects created in the past few minutes. Thus, the problem that those Observables that are to be named after the (single) object they contain do not carry a proper name (because at creation time of the Observable, the Object usually does not exist, yet) can be fixed.

7.3 0.1.0 (2013-12-19)

- Bugfixes; added documentation

7.4 0.0.9 (2013-12-16)

- First release on PyPI.

CHAPTER 8

Developers' Guide to DINGOS

Contents:

8.1 Before starting to develop

8.1.1 Read up on techniques and styles used in DINGOS

Django DINGOS profitted a lot from the advice provided in Two Scoops of Django.

Unless you are an absolute Django expert (and maybe even then), please read Daniel Greenfield's and Audrey Roy's excellent Two Scoops of Django. Even though it provides best practices for Django 1.5, most of its advice is also valid for Django 1.6, and likely to be very relevant for quite a few minor revisions to come.

8.1.2 Make sure that DINGOS is the right place to modify / add to

Although DINGOS is likely to be used mainly in the context of the Django MANTIS Cyber Threat Intelligence Management application, DINGOS shold stay a /generic/ application for managing structured information. So whenever you find yourself adding/modifying stuff in DINGOS that is specific to cyber threat intelligence management, the STIX, CybOX standards, etc., **DINGOS is the wrong place to modify/add to**. The same goes for customizations that are particular to your instance of running MANTIS.

Please consider the following places for development instead:

- If you want to add Python code that is particular to cyber threat management, consider adding this in django-mantis-core
- If you want to add Python code that is particular to a certain standard, consider adding it to the respective importer module, e.g., django-mantis-stix-importer or similar
- If you want to make modifications to a DINGOS template that is required for your local instance of MANTIS (or whatever framework is using DINGOS), the right way is probably to override one of the DINGOS base templates. Have a look at how django-mantis overrides the `templates/dingos/grappelli/base.html` template; see also the Django documentation on overriding templates.
- If you want to change the url paths of DINGOS views, do this in the `url.py` of your instance rather than `dingos/url.py`.

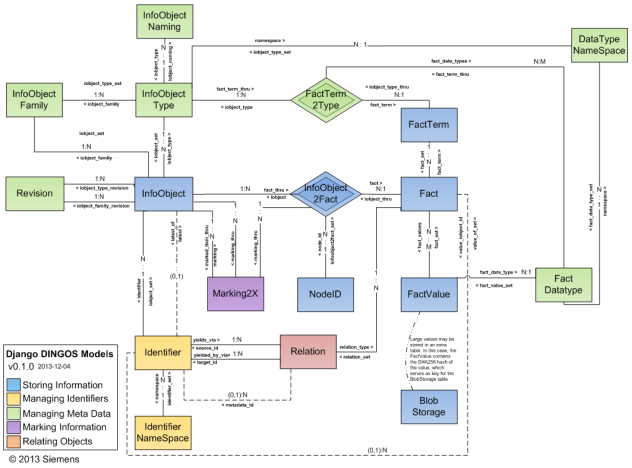


Fig. 8.1: DINGOS Model Overview
Overview of DINGOS models and their relationships.

8.2 DINGOS Model Overview

Please refer to the `DINGOS Developers' Overview` of the DINGOS models. The source code file models.py is extensively documented. Very readable and browsable documentation is generated by Django under the admin url `/admin/docs/models` – be sure to include `django.contrib.admindocs` in your list of installed applications and the following in your `url.py` file:

```
urlpatterns = patterns('',
                       ...
                       # Admin documentation:
                       url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
                       ...
                      )
```

8.3 DINGOS Application Layout

Contents

8.3.1 Overview of the directory layout

The layout of the DINGOS Django application is as follows:

```
.
-- dingos
|   -- core
|   |   -- datastructures.py
|   |   -- ...
|   -- management
|   |   -- commands
|   |       -- dingos_generic_xml_import.py
|   |
|   -- migrations
|   |   -- 0001_initial.py
|   |   -- ...
|   -- templates
|   |   -- dingos
|   |       -- grappelli
|   |           -- base.html
|   |           -- details
|   |           |   -- ...
|   |           -- includes
|   |           |   -- ...
|   |           -- lists
|   |           |   -- ...
|   |           -- searches
|   |               -- ...
|   -- templatetags
|   |   -- dingos_tags.py
|   -- admin.py
|   -- filter.py
|   -- import_handling.py
|   -- importer.py
|   -- models.py
|   -- __init__.py
|   -- read_settings.py
```

```
|   -- urls.py
|   -- view_classes.py
|   -- views.py
```

8.3.2 `core`: internal DINGOS libraries

Internal libraries with helper functions are placed in the `core` folder. The most important library probably is `core/datastructures.py`, which contains `DingosObjDict`, the dictionary structure into which imported data is written. `DingosObjDict` preserves the order in which keys have been added and knows how to /flatten/ itself into a list of facts.

8.3.3 `management/commands`

This folder contains code for the command-line scripts that can be executed via Django's `django-admin` or `manage.py` interface. Refer to Django documentation on custom django-admin commands for a description of how commands can be added.

8.3.4 `templates\dingos\grappelli`

DINGOS uses Django templates (see Django documentation on the template language) for rendering HTML pages. These are located in the `template\dingos\grappelli` folder. The reason for this nesting is the following:

- by having `dingos` in the file path, also other apps are able to refer to templates defined in DINGOS
- by having `grappelli` in the file path, we are open to supporting different CSS frameworks at a later point of time: for supporting, e.g., `bootstrap`, a folder `templates\bootstrap` would have to be added and would then contain the bootstrap-based templates.

In order to learn how to use the Django Grappelli CSS, make sure to include `(r'^grappelli/', include('grappelli.urls'))` in your url patterns in `url.py`. You can then view the Grappelli CSS documentation under `<your Django server url>/grappelli/grp-doc/`.

8.3.5 `templatetags\dingos_tags.py`

When you are viewing a template and find something like `{% show_InfoObjectIDData object %}` that seems to do something magical (in this case, rendering a box containg identifier data of an object), then you are looking at a Django /template tag/. Those are defined in `templatetags\dingos_tags.py`; the template snippets used by the tags are defined in ‘‘templatesdingosgrappelliincludes‘.

8.3.6 `admin.py`

Configuration for the Django admin interface: via the admin interface, you can access the DINGOS models. That is useful for viewing certain data (e.g., which namespaces do I have in my system?) and configuring data (e.g., managing naming schemas via the `InfoObjectType` objects). Refer to the Django documentation on the admin site for details about the contents of `admin.py` – you may also want to have a look at the documentation of Django Grappelli, since `admin.py` uses some extensions provided by Grappelli.

8.3.7 filter.py

DINGOS uses the django-filter app to generate filters for list views. The configuration for the filters is located in `filter.py`: for background on how to configure filters, please refer to the django-filter documentation.

8.3.8 import_handling.py

Next to `models.py` (see below), this is the heart of DINGOS: it defines the class `DingoImportHandling` that contains the `xml_import` function, a highly configurable function for turning XML into DINGOS dictionary objects, and `create_iobject`, the function used to write a DINGOS dictionary object to a `InfoObject` in the database.

8.3.9 importer.py

The most important content of this file is the generic class `DingoImportCommand` which provides the basis for easy implementation of import scripts to be carried out via the command-line (see above under `management/commands` and Django documentation on custom django-admin commands.

This file also contains a very simple generic XML importer, which is mostly for demonstration purposes.

8.3.10 models.py

The heart of DINGOS. The code is extensively documented; please refer to the `DINGOS Developers' Overview` of the DINGOS models for an overview.

8.3.11 __init__.py

DINGOS uses the `__init__.py` file to define a number of defaults used within the DINGOS code.

8.3.12 read_settings.py

Code for reading DINGOS-specific settings configured in the Django settings file(s). Some of the defaults defined in `__init__.py` can be overwritten here.

8.3.13 urls.py

The Django URL configuration. See the Django documentation on the URL dispatcher.

8.3.14 view_classes.py

DINGOS makes extensive use of Django's class-based views (see the Django documentation on class-based views). In `view_classes.py`, we define mixins (see also the Django documentation on using mixins in class-based views) and base classes that are used for defining views in DINGOS.

8.3.15 views.py

The DINGOS views. Refert to the Django documentation on class-based views.

When writing and testing views, do not even start without the excellent Django Debug Toolbar: it shows you, for example, how many which SQL queries were executed (which will help you to find the right configuration for the prefetch_related and select_related

8.4 Writing views and templates for Dingos

Contents

8.4.1 Relevant folders and files

Templates

DINGOS uses Django templates (see Django documentation on the template language) for rendering HTML pages. These are located in the `template\dingos\grappelli` folder. The reason for this nesting is the following:

- by having `dingos` in the file path, also other apps are able to refer to templates defined in DINGOS
- by having `grappelli` in the file path, we are open to supporting different CSS frameworks at a later point of time: for supporting, e.g., `bootstrap`, a folder `templates\bootstrap` would have to be added and would then contain the bootstrap-based templates.

In order to learn how to use the Django Grappelli CSS, make sure to include `(r'^grappelli/', include('grappelli.urls'))` in your url patterns in `url.py`. You can then view the Grappelli CSS documentation under `<your Django server url>/grappelli/grp-doc/`.

Views

DINGOS makes extensive use of Django's class-based views. In `view_classes.py`, we define mixins and base classes that are used for defining views in DINGOS; the views themselves are defined in `views.py`.

8.4.2 Supporting Documentation

- For basics on templates, refer to the Django documentation on the template language.
- In order to learn how to use the Django Grappelli CSS, make sure to include `(r'^grappelli/', include('grappelli.urls'))` in your url patterns in `url.py`. You can then view the Grappelli CSS documentation under `<your Django server url>/grappelli/grp-doc/`.

- For information on class-based views see: - Django documentation on class-based views - Django documentation on using mixins in class-based views

8.4.3 Dingos-specific features

User configurations

Since Dingos 0.2.0, Dingos offers resources for structured management of user-specific data such as user-configurations. Please refer to Dingos User Configuration Facilities for more information.

8.4.4 Tips and tricks

When writing and testing views, do not even start without the excellent Django Debug Toolbar: it shows you, for example, how many which SQL queries were executed (which will help you to find the right configuration for the prefetch_related and select_related

8.5 Dingos User Configuration Facilities

8.5.1 Defining user configurations

The default user configuration is defined in the constant `DINGOS_DEFAULT_USER_PREFS`. `dingos/__init__.py` but can be overwritten in the settings, e.g. as follows:

```
DINGOS = {
          (...)
          'DINGOS_DEFAULT_USER_PREFS' : {
              'dingos' : { 'widgets' :
                                  {'embedded_in_objects' :
                                       {'lines' : {'@description': """Max. number of objects displayed i
                                                                   widget listing the objects in which the
                                                                   current object is embedded.""",
                                                   '_value' : '5'}
                                       } ,
                                   },
                              'view' :
                                  {'pagination':
                                       {'lines' : {'@description': """Max. number of lines displayed in
                                                                paginated views.""",
                                                   '_value' : '20'},
                                        },
                                    'orientation' : {'@description': """Layout orientation. Possible valu
                                                     '_value' : 'horizontal'}
                                  }

              }
          }

 (...)
}
```

When a user logs in for the first time, the standard user configuration is copied over to his personal user configuration. The user configuration can be viewed with the view named `url.dingos.admin.view.userprefs` – the standard URL for this view is `../Admin/ViewUserPrefs`.

A logged in user can edit the settings under the `ViewUserPrefs`. Alternatively, for testing purposes, you can change the preferences via the command line interface:

```
python manage.py dingos_manage_user_settings --reset preferences <user_name1> <user_name2> --settings
```

After doing this, go to the above-mentioned view of the user preferences for a user to also refresh the user data that has been cached in the session.

8.5.2 Accessing user configurations in templates

In templates, user configurations are accessed as follows:

```
customization.<default_value>.<path>.<to>.<value>.<in>.<config>.<dictionary>
```

For example to access the orientation of the display defined as `horizontal` above, you would write:

```
customization.horizontal.dingos.view.orientation
```

Or, to access the number of lines to be shown on a list display (with a default value of *15*), you would write:

```
customization.15.dingos.view.pagination.lines
```

8.5.3 Accessing user configuration in views

The Dingos standard views all include the mixin `ViewMethodsMixin`, which defines the function `lookup_customization`. In order to look up the number of lines to be shown on a list display (with a default value of *15*), you would write:

```
self.lookup_customization('dingos','view','pagination','lines',default=15)
```