MAGE

Release 6.3

Oct 02, 2022

Contents

CHAPTER 1

Introduction

MAGE is an assistant for managing the different environments all big IT projects will eventually end up with (development, user acceptance, integration, preproduction, etc). It will never *do* any action on the environments by itself, such as installing a patch, but will provide many tools to help script these actions (the list of which is specific to each project) and to keep tabs on the environments (this was installed at that time, etc).

Basically, it's a scripting toolbox with some kind of CMDB and DSL designed for out-of-production environments, as well as a publishing portal for all these data.

A partial list of functions:

- **A referential for the environments. What is their content (databases, programs, configurations, ... basically anything is p**
  - **Scripting API**: this enables a huge deal of automation in scripts. For example, a shell script supposed to backup all environments will only have to query MAGE for the elements to backup and will retrieve all necessary data - such as connection information. All it has to know for this is a MAGE account, and submit a query.
  - **Ease of use**: the referential was built with ease of use in mind. For example, every environment can easily be cloned instead of having to define every component. It is even possible to define templates.
  - **Naming conventions**: MAGE can also help enforce naming conventions (which will be automatically applied on an environment duplication for example).
  - **Easy admin**: a fully-featured web site to easily manage your environments, change names, descriptions, every last piece of data stored in the referential. It is based on the Django admin site, for those who know this marvellous little web framework.
  - **Publication**: Web pages with clear graphs and tables are automatically created to provide all the data the environments users will need.
- **A full SCM system**
  - **SCM referential**: stores references to every patchset/installset/binaries/..., the associated versions.
  - **Configuration ordering**: whatever the version naming convention is (A, B, C or 321.1.2, 2.2.3, or really whatever), MAGE will be able to determine a version order thanks to a nifty dependency

system (a delivery is either FULL without dependencies on other components versions OR requires other components to be at a certain version level (exact, or superior to, or inferior to)).

- **Backup tracking**: backups are just another kind of FULL patchset - and so are treated as such. A special API is provided to enable scripts to easily store backup data with simple HTTP GETs.
- **Configuration tracking**: the heart of the SCM module. Each time a configuration is modified through the installation of a patchset (or assimilated), it should be registered (through a simple GET). MAGE will then provide many web pages detailing the history of environments, the current versions of elements, a time machine, etc.: basically, all the SCM views that are usually needed on big IT projects, and then some.
- **Publication**: all the data is made freely accessible to everyone on web pages, limiting the volume of basic and unproductive questions the IT team may receive.

CHAPTER 2

Contents

2.1 Installation with Git

2.1.1 Prerequisites

- OS: every OS with a supported Python 3.10 distribution. (Windows, most Linux distributions, Solaris, . . .)
- The latest Python 3.10.x (not Python 2.x)
- A git client (on Windows, the recommended distribution is GitHub's http://msysgit.github.io/)
- Optionally, a database (Oracle >= 10g, PostgresQL, mysql). Default is sqlite 3 - it is bundled with Python, so nothing special is required. In other databases, you will need an account with the permission to create tables, sequences and indexes (or their equivalent in your database).

2.1.2 MAGE itself

Checkout

Choose a directory in which to install MAGE. This directory will not be accessible to users. It will be referred to as ${MAGE_INSTALL_ROOT} in this document.:

```
git clone https://github.com/marcanpilami/MAGE.git
```

Libraries

These are installed with PIP:

```
## Linux (sh, bash & similar) or Windows (pwsh)
cd $MAGE_INSTALL_ROOT
pip install -r requirements.txt --upgrade
```

Settings

Copy the file ${MAGE_INSTALL_ROOT}/MAGE/local_settings.sample.py to ${MAGE_INSTALL_ROOT}/MAGE/local_settings.py

Edit the file. Every setting is explained in the file. Of particular importance are:

- Database configuration.
- Allowed hosts (unless running in debug mode)
- Static root - this directory will be directly accessible to users

Sync

In the ${MAGE_INSTALL_ROOT} directory, create the database objects by running:

```
python manage.py migrate
python manage.py collectstatic
python manage.py createsuperuser
python manage.py synccheckers
```

You will be asked to create a root account. Accept and do not forget the password you specify.

Test

Run:

```
python manage.py runserver 0.0.0.0:8000
```

This will launch a small web server listening on an address printed on the standard output. With a browser, try this address. You should then access MAGE's homepage.

Initial data

If you just want to play with demo data, run the following commands:

```
python manage.py shell
from scm.demo_items import create_test_is
create_test_is()
exit
```

Otherwise, the database is yours to populate through the GUI and scripts. For writing bootstrap script, inspiration should be taken from the one used above.

WSGI/OSGI/FastCGI/SCGI/AJP integration

For deploying MAGE inside a full-fledged web server, please follow the instructions at https://docs.djangoproject.com/en/4.1/howto/deployment/wsgi/.

Please note that all new deployments should use WSGI and NOT FastCGI which is deprecated in the Apache world.

2.2 Installation with Docker

2.2.1 Prerequisites

- OS: a recent Linux
- Docker daemon or equivalent
- Optionally, a database (Oracle >= 10g, PostgresQL, mysql). Default is sqlite 3 - it is bundled with Python, so nothing special is required. In other databases, you will need an account with the permission to create tables, sequences and indexes (or their equivalent in your database).
  - Obviously the database can also run inside Docker, but this is your choice.

2.2.2 MAGE itself

Just run:

```
docker run -it -e "DJANGO_ROOT_INITIAL_PASSWORD=something" -e "MAGE_CREATE_DEMO_
↪DATA=True" -e "MAGE_ALLOW_MIGRATIONS=True" -p 8000:8000 enioka/mage:nightly
```

Other available image tags are *latest* or specific version tags. Full list available on Docker Hub.

Configuration can be done through environment variables:

- database properties
  - DATABASE_ENGINE: either nothing (sqlite3) or a specific driver (*django.db.backends.postgresql_psycopg2* for postresql, *django.db.backends.mysql* for MySQL).
  - DATABASE_NAME
  - DATABASE_HOST
  - DATABASE_PORT
  - DATABASE_USER
  - DATABASE_PASSWORD
- security properties:
  - DJANGO_ALLOWED_HOSTS: a comma-separated list of HTTP hostnames allowed to query MAGE. Very useful behind a reverse proxy. Wildcard is also possible.
  - DJANGO_SECRET_KEY: set it to a unique random string. Used for some signature purposes.
  - DJANGO_ROOT_INITIAL_PASSWORD: if set, a user named *root* is created with this password if it does not exist yet.

This image contains drivers for sqlite, postgresql and sqlite3. Using this image as a base image, it is possible to create MAGE images with other drivers in custom images.

If you are using sqlite for the database, note the directory */code/deployment/db* is a volume and contains the database file.

The image listens on port 8000.

The image runs database migrations on startup if needed if *MAGE_ALLOW_MIGRATIONS* is set to *True*.

2.3 Preparing MAGE for first use

MAGE is rather generic, allowing to specify a wide variety of environment types in various contexts. Therefore, before starting to use MAGE, there are a few notions that must be defined in the tool in the order given here.

It is recommended to read the page fully before actually setting parameters. Afterwards, connect to the administration interface (default is site_root/admin, there is an 'A' link in the top right corner of the home page), login (with the super user created during install) and for each paragraph create the desired objects.

Warning: the idea is to follow the order given in the page 'new reference items', available from the home page.

2.3.1 Setting the technical context

The first thing to do is to decide how to model the different environments. The main ideas here are:

- every 'item' that can be packaged separately and has to be tracked (as in version tracking) must have its own component description.
- every data needed for administration (DNS, login, ports...) must be available in the model. Links between component descriptions must make this information available simply in the context of a modification of configuration of the tracked elements. (if a Java war package is installed, it must be easy to go from the package to the admin URL for example).
- the simpler, the better!

For each component that was identified, a **component description** must be created. It is simply a list of fields and relationships.

Note: the standard way of creating these is through the administration web UI (follow the links inside the 'new reference items' page). Scripting is also possible.

Basics attributes

- Name - this is a code that will be used when a short and stable in time name is preferable
- Description - the verbose 'public' name.
- Tag - a simple unbound classifier. Please use a tag, it makes many page more readable.
- Self description pattern: a *component expression* which resolution gives the 'name' of instances.

Simple fields. A simple field is just what its name entails - a key/value pair. Values are always stored as strings, even if a data type must be provided. The data type is actually used for widget selection and controls.

- short name: a valid Python identifier. (no spaces, letters and digits and underscores, cannot begin with an underscore or with mage_)
- label: verbose name used in forms
- compulsory: for validation
- sensitive: if True, the values will be hidden to anyone but admins and scripts (NOT published to everyone)
- widget row: if None, won't be published inside the environment description page. Otherwise, this gives the order of fields in that page.

Relationship fields

Computed fields

It is often interesting to make deductions instead of forcing administrators to input everything with many repetitions. Computed fields exist for that. They allow to retrieve data from linked instances, to make basic mathematics operations, to compose strings, to fallback on another field if it is null, etc. Please read the *component expression* reference and the samples for this.

2.3.2 Setting the applicative context

Project

A project is nothing more than a classifier. It has no other interest than to regroup Environments (an Environment belongs to zero or one Project).

It's main use is in *Conventions patterns*, which can make use of its name and alternative names.

Application

This is a second level of classification: a project may have zero to many Applications. It can also be used in Conventions. The different Logical Components (see below) all belong to one (and only one) Application, so this is a very important classifier.

Note: 'Project' and 'Application' are just names. They can be considered as "Big project" and "Sub project", or "Program" and "Project", etc.

Logical Component (LC)

It represents the "essence" of an item of the project. It can be an application, a configuration, a program... whatever. Choosing the right granularity for LC is crucial - they are the foundation of everything else. As a rule of thumb, a LC corresponds to an element you want to track in configuration/version on its own. The more there are, the more complicated it will get but the more precise the collected data will be.

Now, choosing the configuration tracking granularity is up to the user, as no tool will ever automate this - there are many trade-offs and therefore many different solutions.

class ref.models.**LogicalComponent**

name
The name of the logical component

application
The application the component belongs to (compulsory)

description
A (very) short text describing the use of the LC

scm_trackable
Default is True. If False, this LC will never be used in any Configuration Management operation (backup, update, ...)

Implementation Offer (CIC)

Note: internally, MAGE refers to this as a Component Implementation Class (CIC)

This is a technical way of actually implementing a logical component.

For example, if the LC is "Application B data storage", there may be many CICs :

- an Oracle database schema
- a PostgreSQL database
- ... whatever RDBMS

In a single project, all these possibilities may be used. To build on the previous example, Oracle will be used in production but as Oracle is expensive, developers will use PostgreSQL. This is why the distinction (an abstraction level, actually) between the CIC and the LC is very important.

Note: obviously, in in simple project, nothing prevents you from having only one CIC for a LC.

class ref.models.**ComponentImplementationClass**

name

description
The description object that will be used to actually instantiate the CIC. See above.

implements
The *LogicalComponent* implemented

sla
An optional SLA object

2.3.3 Environments

Environment Type

Each environment has - optionally - one associated Type. It provides common values for :

- an optional SLA
- a typology (production, conformity, ...)
- backup related parameters
- types of component that are allowed for these environments.

At the beginning of the project, the first few types should be referenced. The list can be completed later - but never purged, as it would allow to re-write history.

Environment

A potentially partial implementation of the functional and logical architecture of an IT perimeter, aiming at fulfilling the needs of a certain population at a given period of the life-cycle of a system.

Basically: a bunch of items that may belong to other environments too. 'Item' will be defined later.

Most of the time, they are built (in MAGE, but also in reality) by copying another one. Save, obviously, for the first one.

class ref.models.**Environment**

name

buildDate
Default is at the time the Python object is created.

destructionDate
Planned destruction. Nullable.

description
Not nullable. Displayed pretty much everywhere.

manager
Name of the person in charge of using the environment. (often: team leader). Nullable.

project
Nullable.

typology
An environment type. Not nullable.

template_only
Default False. All environments can be copied and serve as templates for creating others. If this is ticked, the environment will only be used for templating (there should be no actual implementation of the template)

At the beginning of a project, a first representative environment should be created though the admin (complete with component instance, described below) for every different environment "template" you'll have. This template will then be copied each time a new environment is created. During copy, the following elements are preserved or remapped:

- members of the environment are all copied (the list can be filtered as a parameter - so the source template can be "too complete")
- relationships between members of the source environment become relationships between the copied members
- relationships between members of the source environment and other items not member of the environment are preserved as-is in the copy, unless explicitly remapped (parameter). For example, an application server belonging to the source environment runs on a Windows server that does not belong to the environment. The copy of the environment will have a new application server running on the same server.
- some naming conventions will be applied to the copy (for example, to change the component instance names)

2.3.4 Environment content

After the context is fully described, it is time to fill in the environments with data that will be useful for scripting, configuration tracking, ...

Component Instance

A component Instance is the representation of an actual "thing" managed on the project. Basically, it is an instance of CIC. To clarify things :

- Logical Component = "Application B data store"
- Component Implementation Class = "Oracle schema for B data store with High Availability"

- Component Instance = “schema my_schema_name (described by the items listed in Component Description “Oracle Schema”)”

The component instance is described by the *ComponentImplementationClass.description* attribute of the CIC. However, all CI have a few common attributes.

class ref.models.**ComponentInstance**

name
The meaning of this attribute depends of the described CIC. However, it should always enable the user to identify an instance.

instantiates
The *ComponentImplementationClass* implemented. It is optional - not all component instances need to be version tracked.

deleted
Instances are never deleted - they are hidden when they do not exist any more in the real world. This enables to having a consistent configuration tracking (for example, backups still exist when an environment is destroyed, and the user may want one day to restore it without loosing all the version data associated to it)

environments
The different environments the instance belongs to. It may belong to multiple environment (may be the case for a shared middleware) or to none (it may make no sense to attribute a shared server to all the environments it supports)

Warning: often, only “component” is used instead of “Component Instance”.

At the beginning of a project, the new environments (created at the beginning of this page) should be filled with component instances. Contrary to all other elements described on this page, there is no “Component Instance” page in the admin site. This page instead sits inside the main MAGE portal.

2.4 Mage Query Language (MQL)

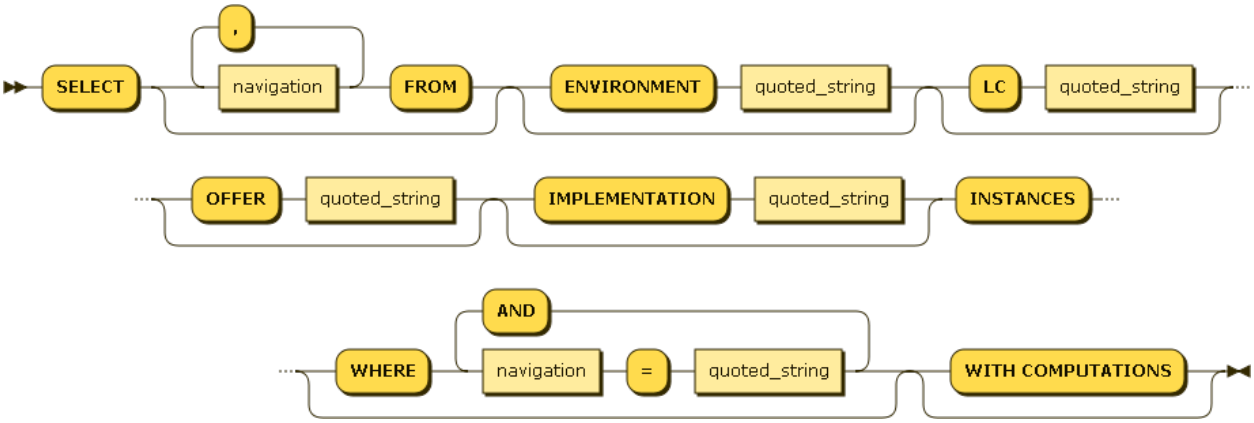
MQL is an SQL-like language allowing to query the component instance referential easily from everywhere, including from shell scripts.

It allows to find component instances based on their attributes and, most important, based on the relationships they has with other components. That way, scripts may easily find all the data they need to run. For example, just knowing the hostname of the server it runs on, a script will be able to query all database accounts running on database instances running on this server, and then backup or export them.

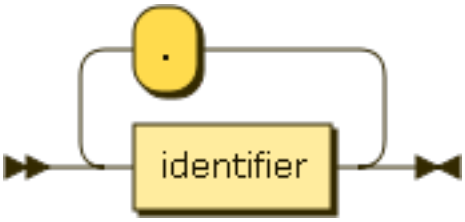
2.4.1 Grammar (EBNF)

```
quoted_string  ::=  := "'" .* "'"
identifier     ::=  := [a-zA-Z] [a-zA-Z0-9]*
navigation     ::=  := identifier ('.' identifier)*
query          ::=  := "SELECT" (navigation (',' navigation)* "FROM")? ("ENVIRONMENT" quote
```

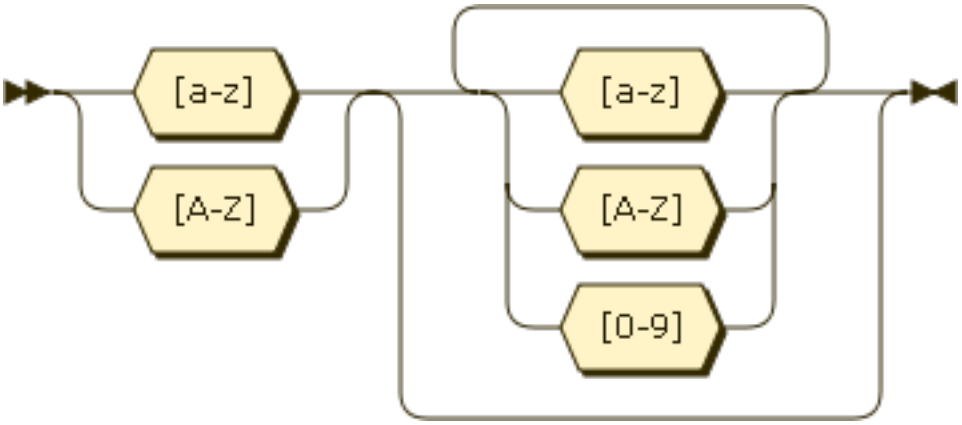
mql_query:



navigation:



identifier:



Diagrams generated with http://bottlecaps.de/rr/ui

Note: keywords such as SELECT are not actually case sensitive.

2.4.2 How to query

MQL is a **filter** language. That means that it works in a negative fashion: you begin with every existing component instances, and the more filters you add, the less instances survive. It also means that everything is optional in MQL, but "SELECT INSTANCES" - in this case, every single last instance is returned.

This section introduces all the different filters, from the simplest to the more complicated.

Query on attributes (simple)

In this case, the user knows a few attributes (usually the name) of the required component instance.:

```
SELECT INSTANCES WHERE port='1789'
```

In this case, 'port' is the name of a field of some (perhaps none!) components. This will retrieve all the component instances that have a port field and which port field is equal to 1789.

Simple inverted commas are compulsory surrounding the value. In case there are some inside the value itself, double the quotes to escape them.

Other examples::

```
SELECT INSTANCES WHERE name='ERP_TEC2_PU6'
SELECT INSTANCES WHERE description='it''s "beautiful" AND name='marsupilami'
```

Query on relations' attributes

This is exactly the same as previously, except we use dotted-path notation to access the fields of the related component instances.

Example:

```
SELECT INSTANCES WHERE server.dns='my.server.dns.org'
```

will select every component instance that has a relationship named 'server' pointing to another component instance which field 'dns' equals 'my.server.dns.org'.

Query on type

There are many component instance types: Oracle instances, web servers, batch programs, etc. To filter on this::

```
SELECT IMPLEMENTATION 'compo type' INSTANCES
```

Example:

```
SELECT IMPLEMENTATION 'jbossas' INSTANCES
```

will return every JBoss application server in the system.

Query on environment

To filter by environment, just use:

SELECT ENVIRONMENT 'envt name' INSTANCES

Example:

```
SELECT ENVIRONMENT 'DEV1' INSTANCES
```

Selecting attributes

Computed fields

To avoid useless computations, computed fields are not present in the query results by default.

To get them, add " WITH COMPUTATIONS" at the end of the query.

```
SELECT INSTANCES WITH COMPUTATIONS
```

Other fields

One can add fields from related instances at the beginning of the query.

```
SELECT name, server.dns FROM INSTANCES WHERE ...
```

This will only give two attributes: the name of the compoennt instance, and the dns of the server the instance runs on.

Warning: this is not efficient. Using computed fields is far better.

Final example

```
SELECT IMPLEMENTATION 'jbossapplication' INSTANCES WHERE datastore.name='prd_int' AND␣
↪group.domain.name='jbossproddomain'
```

will look for applications named integration that:

- are linked to an Oracle Schema named prd_int
- **run on a group (which is not named here)**
  - the group must be inside a domain named jbossproddomain

2.5 Conventions patterns

2.5.1 Introduction

Conventions are a way to rationalize the way components are named and created and speed up their creation. They should reflect both the naming conventions of your project and its environment templates.

A convention can be defined for each standard field inside the component description (it is the 'default' field). They are used:

- when a new component instance is created. All fields that have a default/convention value are set with it.
- when an environment is duplicated, all fields with a default/convention are reset with it.

Please note it is not compulsory to use conventions patterns as a default - a simple value - or nothing - is enough. But it is **strongly recommended** to use them as they allow one click component instance creation, minimizing the risk of error.

2.5.2 Simple naming patterns

Environment

Pattern	Interpretation
%e%	environment name (lower case)
%E%	environment name (upper case)

If the component does not belong to an environment, NOENVIRONMENT will be used.

Application

Pattern	Interpretation
%a%	application name (lower case)
%A%	application name (upper case)
%a1%	application alternate name 1 (lower case)
%A1%	application alternate name 1 (upper case)
%a2%	application alternate name 2 (lower case)
%A2%	application alternate name 2 (upper case)
%a3%	application alternate name 3 (lower case)
%A3%	application alternate name 3 (upper case)

If no application, NOAPPLICATION will be used.

Project

Pattern	Interpretation
%p%	project name (lower case)
%P%	project name (upper case)
%p1%	project alternate name 1 (lower case)
%P1%	project alternate name 1 (upper case)
%p2%	project alternate name 2 (lower case)
%P2%	project alternate name 2 (upper case)
%p3%	project alternate name 3 (lower case)
%P3%	project alternate name 3 (upper case)

NOPROJECT if no project.

Current date

Pattern	Interpretation
%d%	current date (japanese format: YYYYMMDD)

Offer

Pattern	Interpretation
%ic1%	CIC field ref1
%ic2%	CIC field ref2
%ic3%	CIC field ref3

Note: a CIC name is supposed to be descriptive and therefore ill-adapted to naming use. This explains its absence from the list above.

Logical component

Pattern	Interpretation
%lc1%	LC field ref1
%lc2%	LC field ref2
%lc3%	LC field ref3

2.5.3 Naming convention counters

Counters are incremented each time they are used. This is why creating a component instance always begins with a specific form, and not inside the admin application - this allow MAGE to always use the counters. However, should a counter become desynchronized, it is possible to set its value through the admin.

They all begin at 1, and have different scopes. The scope is what defines the counter to increment in each case. For example, an environment scope means there is one counter per environment. Two different items inside the same environment using an environment scope will actually use the same counter.

Counters can be formatted by giving a number of figures after the classifier of the counter.

Environment counter

This counter has an environment scope.

%ce%

Project counter

This counter has a project scope (therefore multiple applications)

%cp%

Global counter

This counter has no scope. There is only one such counter.

%cg%

Model + environment counter

This counter has an environment + component description scope. There is one counter for each description in an environment. Therefore, you can track your different databases without incrementing the counter for your application servers.

%cem%

Model counter

This counter has a component description scope. There is one counter for each description.

%cm%

Instance counter

This counter is scoped by another component instance designated by a dotted-path expression ending on an instance ID.

%cidotted.expression.to.id%

Example for the 'port' field of an application server, sitting on a Unix server designated by the 'server' field':

```
1788+%ci2server.mage_id%
```

This will give a port, starting at 1789, which is unique for the Unix server (that means if there are many servers each hosting multiple AS, each server will have ASs running on ports 1789, 1790, ... without duplicates or holes in the sequence.)

Note: MAGE always adds a field named mage_id to every component instance to help using this feature.

Relation counter

This counter has a component description scope. There is one counter for each description.

%n%

2.5.4 Expression patterns

This is a special pattern that can only be applied after all fields and relationships have been set on an instance. It consists in a *navigation expression*, as detailed elsewhere.

For example, if your convention is that the JMX monitoring port is HTTP port + 1000, it can be specified as:

```
%nport%+1000
```

%n means expression pattern (n means navigation).

As this is applied in a second place, it is still possible to have a counter giving the value of port. This will work perfectly:

```
port => 1788+%ci2server.mage_id%
jmx_port => %nport%+1000
```

Note: every result of the application of any type of pattern is eventually valued as an integer - it it doesn't work nothing is done, if it works the result of the valuation becomes the new value for the field.

2.6 Security

2.6.1 Access security

By default MAGE gives everyone (anonymous users) read-only access to non-sensitive data. It means all projects, all environments.

A middleware can be enabled inside MAGE settings (a line to uncomment) in order to prevent anonymous access.

The following permissions are defined at project level and are always applied (even to anonymous users):

- sensitive data (fields marked as such in the compoennt description, usually password-like fields) are only available to accounts with the *allfields_componentinstance* permission.
- uploading a new delivery requires the *modify_delivery* permission
- modifying an environment requires the *modify_project* permission

The following permissions are defined at project level and are only applied if not allowing anonymous access:

- viewing a project (including on the home page) requires the *view_project* permission

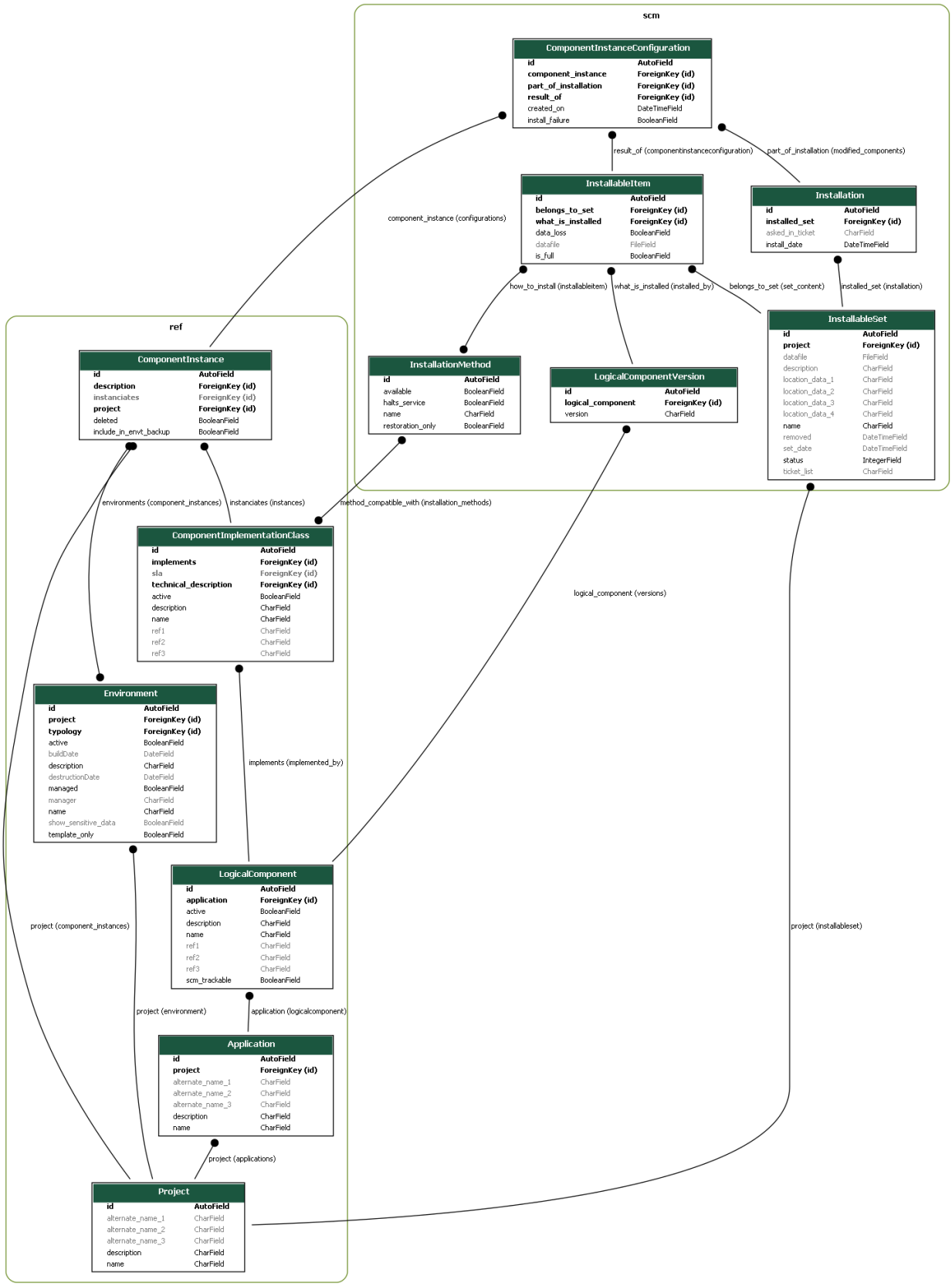
Super administrator accounts have access to everything without limitation.

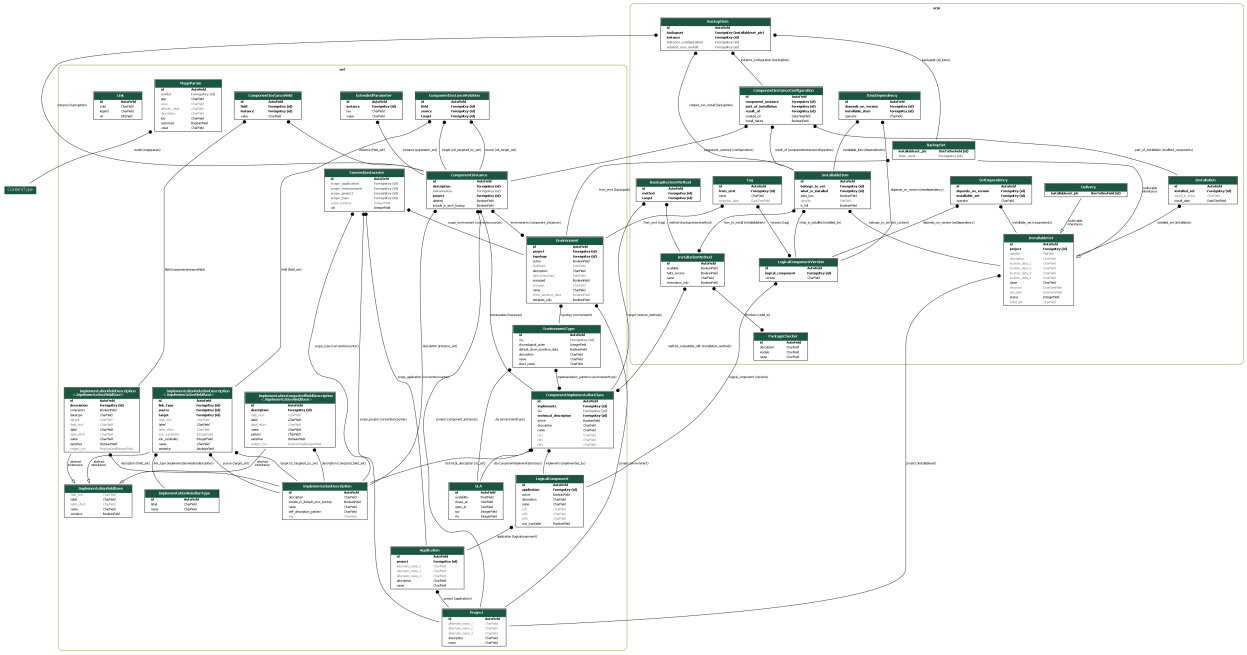
The permissions above are specialized per project: one account can be allowed to see sensitive fields on one project and not on another. There is also a set of permissions that are always applied to all the projects for which the user has the *modify_project* permission:

- *modify_delivery* allows to create and modify deliveries
- *install_installableset* allows to reference a new installation of a given package
- *del_backupset* allows to archive, unarchive a backup
- *validate_installableset* allows to validate a package or backupset
- *add_tag* allows to create , modify or remove a tag.

2.7 Development help: model diagrams

These two diagrams are developer helpers.


scm
ComponentInstanceConfiguration
id AutoField
component_instance ForeignKey (id)
part_of_installation ForeignKey (id)
result_of ForeignKey (id)
created_on DateTimeField
install_failure BooleanField
result_of (componentinstanceconfiguration)
part_of_installation (modified_components)
component_instance (configurations)
InstallableItem
id AutoField
belongs_to_set ForeignKey (id)
what_is_installed ForeignKey (id)
data_loss BooleanField
datafile FileField
is_full BooleanField
Installation
id AutoField
installed_set ForeignKey (id)
asked_in_ticket CharField
install_date DateTimeField
how_to_install (installableitem)
what_is_installed (installed_by)
belongs_to_set (set_content)
installed_set (installation)
InstallableSet
id AutoField
project ForeignKey (id)
datafile FileField
description CharField
location_data_1 CharField
location_data_2 CharField
location_data_3 CharField
location_data_4 CharField
name CharField
removed DateTimeField
set_date DateTimeField
status IntegerField
ticket_list CharField
ref
ComponentInstance
id AutoField
description ForeignKey (id)
instanciates ForeignKey (id)
project ForeignKey (id)
deleted BooleanField
include_in_envt_backup BooleanField
InstallationMethod
id AutoField
available BooleanField
halts_service BooleanField
name CharField
restoration_only BooleanField
LogicalComponentVersion
id AutoField
logical_component ForeignKey (id)
version CharField
environments (component_instances)
instanciates (instances)
method_compatible_with (installation_methods)
ComponentImplementationClass
id AutoField
implements ForeignKey (id)
sla ForeignKey (id)
technical_description ForeignKey (id)
active BooleanField
description CharField
name CharField
ref1 CharField
ref2 CharField
ref3 CharField
logical_component (versions)
Environment
id AutoField
project ForeignKey (id)
typology ForeignKey (id)
active BooleanField
buildDate DateField
description CharField
destructionDate DateField
managed BooleanField
manager CharField
name CharField
show_sensitive_data BooleanField
template_only BooleanField
implements (implemented_by)
LogicalComponent
id AutoField
application ForeignKey (id)
active BooleanField
description CharField
name CharField
ref1 CharField
ref2 CharField
ref3 CharField
scm_trackable BooleanField
project (component_instances)
project (installableset)
project (environment)
application (logicalcomponent)
Application
id AutoField
project ForeignKey (id)
alternate_name_1 CharField
alternate_name_2 CharField
alternate_name_3 CharField
description CharField
name CharField
project (applications)
Project
id AutoField
alternate_name_1 CharField
alternate_name_2 CharField
alternate_name_3 CharField
description CharField
name CharField



Python Module Index

r

Index