salomeTools Documentation

Release 5.4.0

CEA DEN/DANS/DM2S/STMF/LGLS

July 09, 2019

CONTENTS

The **Sa**lome**T**ools (sat) is a suite of commands that can be used to perform operations on SALOME[1].

For example, sat allows you to compile SALOME's codes (prerequisites, products) create application, run tests, create package, etc.

This utility code is a set of Python[2] scripts files.

Find a pdf version of this documentation

[1] http://www.salome-platform.org
[2] https://docs.python.org/2.7

CHAPTER

ONE

QUICK START

1.1 Installation

sat is provided either embedded into a salome package, or as a standalone package.

1.1.1 Embedded sat version

sat is provided in salome packages with sources, in order to be able to recompile the sources (**sat** is not provided in salome packages with only binaries).

Embedded **sat** is always associated to an embedded **sat** project, which contains all the products and application configuration necessary to the package.

```
tar -xf SALOME-9.3.0-CO7-SRC.tgz
cd SALOME-9.3.0-CO7-SRC
ls PROJECT/   # list the embedded sat project
salomeTools/sat config SALOME-9.3.0 -e   # edit the SALOME-9.3.0 configuration pyconf file
```

The user has usually two main use cases with an embedded sat, which are explained in the README file of the archive:

1. recompile the complete application

```
./sat prepare SALOME-9.3.0
./sat compile SALOME-9.3.0
./sat launcher SALOME-9.3.0
```

Please note that the sources are installed in *SOURCES* directory, and the compilation is installed in *INSTALL* directory (therefore they do not overwrite the initial binaries, which are stored in *BINARIES-XXX* directory). The launcher *salome* is overwritten (it will use the new compiled binaries) but the old binaries can still be used in connection with *binsalome* launcher).

2. recompile only a part of the application

It is possible to recompile only a part of the products (those we need to modify et recompile). To enter this (partial recompilation mode), one has initialy to copy the binaries from *BINARIES-XXX* to *INSTALL*, and do the path substitutions by using the **install_bin.sh** script:

```
./install_bin.sh  # pre-installation of all binaries in INSTALL dir, with substitutions
./sat prepare SALOME-9.3.0 -p GEOM  # get GEOM sources, modify them
./sat compile SALOME-9.3.0 -p GEOM --clean_all  # only recompile GEOM
```

1.1.2 Standalone sat packages

sat is also delivered as a standalone package, usually associated to a sat project. The following example is an archive containing salomeTools 5.3.0 and the salome sat project. It can be used to build from scratch any salome application.

```
tar xf salomeTools_5.3.0_satproject_salome.tgz # untar a standalone sat package, with a salome pr
cd salomeTools_5.3.0_satproject_salome
ls projects  # list embedded sat projects
> salome
./sat config -l  # list all salome application available for build
```

Finally, the project also provides bash scripts that get a tagged version of sat from the git repository, and a tagged version of salome projects. This mode is dedicated to the developpersi, and requires an access to the Tuleap git repositories.

1.2 Main usage of SAlomeTools

1.2.1 Purpose, Command Line Interface

sat is a Command Line Interface (CLI[1]) based on python langage. Its purpose is to cover the maintenance and the production of the salome platform and its applications.

Notably: * the definition of the applications content (the products, the resources, the options, the environment, the launcher, etc.) * the description of the products (the environment to set, how to get the sources; how to compilation, the dependencies, etc). * the complete preparation and build * the management of unit or integration tests * the production of binary or source packages

It can be used from interactively from a terminal, or in batch mode.

```
sat [generic_options] [command] [application] [command_options]
```

1.2.2 Getting help

Help option -h

To get help in terminal mode as simple text, use **–help** or **-h** option:

```
sat -h          # or --help : get the list of existing commands and main options
sat compile -h  # get the help on the specific command 'compile'
```

completion mode

When getting started with sat, the use of the competion mode is usefull. This mode will display by type twice on the **tab key** the available options, command, applications or product available. The completion mode has to be activated by sourcing the file **complete_sat.sh** contained in salomeTool directory:

```
source complete_sat.sh      # activate the completion mode

./sat conpile  <TAB> <TAB>  # liste all application available for compilation
> SALOME-7.8.2   SALOME-8.5.0   SALOME-9.3.0   SALOME-master

./sat conpile SALOME-9.3.0 <TAB> <TAB>  # list all available options
> --check              --clean_build_after  --install_flags      --properties
> --stop_first_fail    --with_fathers       --clean_all          --clean_make
> --products           --show               --with_children
```

1.2.3 Verbose and Debug mode

Verbosity

sat has several levels of verbosity. The default value is **3** and correspond to the impression of the main information on what has been done. A verbosity of **0** correspond to no impression at all, while on the opposite a verbosity of **6** prints a lot of information.

Change verbosity level (default is 3).

```
sat -v0 prepare SALOME-9.3.0 -p GEOM  # prepare GEOM product in silent mode
sat -v6 compile SALOME-9.3.0 -p GEOM  # compile GEOM with full verbosity
```

[1] https://en.wikipedia.org/wiki/Command-line_interface

Debug mode -g

This mode is used by developers to see more traces and *stack* if an exception is raised.

1.2.4 Building a SALOME application

Get the list of available applications

To get the list of the current available applications in your context:

```
sat config -l
```

The result depends upon the projects that have been loaded in sat.

Prepare sources of an application

To prepare (get) *all* the sources of an application (*SALOME_xx* for example):

```
# get all sources
sat prepare SALOME_xx

# get (git) sources of SALOME modules
sat prepare SALOME_xx  --properties is_SALOME_module:yes
```

The sources are usually copied in directory *$APPLICATION.workdir + $VARS.sep + 'SOURCES'*

Compile an application

To compile an application

```
# compile all prerequisites/products
sat compile SALOME_xx

# compile only three products (KERNEL, GUI and SHAPER), if not done yet
sat compile SALOME_xx -p KERNEL,GUI,SHAPER

# compile only two products, unconditionaly
sat compile SALOME_xx -p KERNEL,GUI --clean_all

# (re)compile only salome modules
sat compile SALOME_xx --properties is_SALOME_module:yes  --clean_all
```

The products are usually build in the directory
$APPLICATION.workdir + $VARS.sep + 'BUILD'

The products are usually installed in the directory
$APPLICATION.workdir + $VARS.sep + 'INSTALL'

1.3 Configuration

salomeTools uses files with **.pyconf** extension to store its configuration parameters. These pyconf configuration files are provided by the salomeTool projects that are set by sat init command.

When executing a command, sat will load several configuration files in a specific order. When all the files are loaded a *config* object is created. Then, this object is passed to all command scripts.

1.3.1 Syntax

The configuration files use a python-like structure format (see config module[2] for a complete description).

- **{}** define a dictionary,
- **[]** define a list,
- **@** can be used to include a file,
- **$prefix** reference to another parameter (ex: `$PRODUCT.name`),
- **#** comments.

Note: in this documentation a reference to a configuration parameter will be noted `XXX.YYY`.

1.3.2 Description

VARS section

This section is dynamically created by salomeTools at run time.
It contains information about the environment: date, time, OS, architecture etc.

```
# to get the current setting
sat config --value VARS
```

APPLICATION section

This section is defined in the application pyconf file.
It contains instructions on how to build a version of SALOME (list of products and versions, compilation options, etc.)

```
# to get the current setting
sat config SALOME-xx --value APPLICATION
```

PRODUCTS section

This section contains all the information required to build the products contained in the application.
It is build from the products configuration files.

```
# to get the current setting
sat config SALOME-xx --value PRODUCT
```

[2] http://www.red-dove.com/config-doc/

USER section

This section is defined by the user configuration file, `~/.salomeTools/SAT.pyconf`.

The `USER` section defines some parameters (not exhaustive):

- **pdf_viewer** : the pdf viewer used to read pdf documentation
- **browser** : The web browser to use (*firefox*).
- **editor** : The editor to use (*vi, pluma*).
- and other user preferences.

```
# to get the current setting
sat config SALOME-xx --value USER

# to edit your personal configuration file
sat config -e
```

Other sections

- **PROJECTs** : This section contains the configuration of the projects loaded in salomeTool by sat init –add_project command.
- **PATHS** : This section contains paths used by saloeTools.
- **LOCAL** : contains information relative to the local installation of salomeTool.
- **INTERNAL** : contains internal salomeTool information

All these sections can be printed with sat config command:

```
# It is possible to use sat completion mode to print available sections.
sat config SALOME-xx --value <TAB> <TAB>
> APPLICATION.       INTERNAL.          LOCAL.             PATHS.
> PRODUCTS.          PROJECTS.          USER.              VARS.

# get paths used by sat
sat config SALOME-xx --value PATHS
```

It is possible to use sat completion mode to print available sections.

CHAPTER

TWO

LIST OF COMMANDS

2.1 Command doc

2.1.1 Description

The **doc** command displays sat documentation.

2.1.2 Usage

- Show (in a web browser) the sat documentation in format xml/html:

  ```
  sat doc --xml
  ```

- Show (in evince, for example) the (same) sat documentation in format pdf:

  ```
  sat doc --pdf
  ```

- Edit and modify in your preference user editor the sources files (rst) of sat documentation:

  ```
  sat doc --edit
  ```

- get information how to compile locally sat documentation (from the sources files):

  ```
  sat doc --compile
  ```

2.1.3 Some useful configuration pathes

- **USER**
  - **browser** : The browser used to show the html files (*firefox* for example).
  - **pdf_viewer** : The viewer used to show the pdf files (*evince* for example).
  - **editor** : The editor used to edit ascii text files (*pluma or gedit* for example).

2.2 Command config

2.2.1 Description

The **config** command manages sat configuration. It allows display, manipulation and operation on configuration files

2.2.2 Usage

- Edit the user personal configuration file `$HOME/.salomeTools/SAT.pyconf`. It is used to store the user personal choices, like the favorite editor, browser, pdf viewer:

  ```
  sat config --edit
  ```

- List the available applications (they come from the sat projects defined in `data/local.pyconf`:

  ```
  sat config --list
  ```

- Edit the configuration of an application:

  ```
  sat config <application> --edit
  ```

- Copy an application configuration file into the user personal directory:

  ```
  sat config <application> --copy [new_name]
  ```

- Print the value of a configuration parameter.
  Use the automatic completion to get recursively the parameter names.
  Use *–no_label* option to get *only* the value, *without* label (useful in automatic scripts).
  Examples (with *SALOME-xx* as *SALOME-8.4.0*):

  ```
  # sat config --value <parameter_path>
  sat config --value .          # all the configuration
  sat config --value LOCAL
  sat config --value LOCAL.workdir

  # sat config <application> --value <parameter_path>
  sat config SALOME-xx --value APPLICATION.workdir
  sat config SALOME-xx --no_label --value APPLICATION.workdir
  ```

- Print in one-line-by-value mode the value of a configuration parameter,
  with its source *expression*, if any.
  This is a debug mode, useful for developers.
  Prints the parameter path, the source expression if any, and the final value:

  ```
  sat config SALOME-xx -g USER
  ```

  Note: And so, *not only for fun*, to get **all expressions** of configuration

  ```
  sat config SALOME-xx -g . | grep -e "-->"
  ```

- Print the patches that are applied:

  ```
  sat config SALOME-xx --show_patchs
  ```

- Get information on a product configuration:

```
# sat config <application> --info <product>
sat config SALOME-xx --info KERNEL
sat config SALOME-xx --info qt
```

2.2.3 Some useful configuration pathes

Exploring a current configuration.

- **PATHS**: To get list of directories where to find files.
- **USER**: To get user preferences (editor, pdf viewer, web browser, default working dir).

sat commands:

```
sat config SALOME-xx -v PATHS
sat config SALOME-xx -v USERS
```

2.3 Command prepare

2.3.1 Description

The **prepare** command brings the sources of an application in the *sources application directory*, in order to compile them with the compile command.

The sources can be prepared from VCS software (*cvs, svn, git*), an archive or a directory.

Warning: When sat prepares a product, it first removes the existing directory, except if the development mode is activated. When you are working on a product, you need to declare in the application configuration this product in **dev** mode.

2.3.2 Remarks

VCS bases (git, svn, cvs)

The *prepare* command does not manage authentication on the cvs server. For example, to prepare modules from a cvs server, you first need to login once.

To avoid typing a password for each product, you may use a ssh key with passphrase, or store your password (in .cvspass or .gitconfig files). If you have security concerns, it is also possible to use a bash agent and type your password only once.

Dev mode

By default *prepare* uses *export* mode: it creates an image of the sources, corresponding to the tag or branch specified, without any link to the VCS base. To perform a *checkout* (svn, cvs) or a *git clone* (git), you need to declare the product in dev mode in your application configuration: edit the application configuration file (pyconf) and modify the product declaration:

```
sat config <application> -e
# and edit the product section:
#   <product> : {tag : "my_tag", dev : "yes", debug : "yes"}
```

The first time you will execute the *sat prepare* command, your module will be downloaded in *checkout* mode (inside the SOURCES directory of the application. Then, you can develop in this repository, and finally push them in the base when they are ready. If you type during the development process by mistake a *sat prepare* command, the sources in dev mode will not be altered/removed (Unless you use -f option)

2.3.3 Usage

- Prepare the sources of a complete application in SOURCES directory (all products):

  ```
  sat prepare <application>
  ```

- Prepare only some modules:

  ```
  sat prepare <application>  --products <product1>,<product2> ...
  ```

- Use –force to force to prepare the products in development mode (this will remove the sources and do a new clone/checkout):

  ```
  sat prepare <application> --force
  ```

- Use –force_patch to force to apply patch to the products in development mode (otherwise they are not applied):

```
sat prepare <application> --force_patch
```

- Prepare only products that are not present in SOURCES. This completion mode is used to complete the preparation when it was interrupted, or when the product list was increased:

```
sat prepare <application> --complete
```

2.3.4 Some useful configuration pathes

Command *sat prepare* uses the *pyconf file configuration* of each product to know how to get the sources.

Note: to verify configuration of a product, and get name of this *pyconf files configuration*

```
sat config <application> --info <product>
```

- **get_method**: the method to use to prepare the module, possible values are cvs, git, archive, dir.
- **git_info** : (used if get_method = git) information to prepare sources from git.
- **svn_info** : (used if get_method = svn) information to prepare sources from cvs.
- **cvs_info** : (used if get_method = cvs) information to prepare sources from cvs.
- **archive_info** : (used if get_method = archive) the path to the archive.
- **dir_info** : (used if get_method = dir) the directory with the sources.

2.4 Command compile

2.4.1 Description

The **compile** command allows compiling the products of a SALOME[1] application.

2.4.2 Usage

- Compile a complete application:

```
sat compile <application>
```

- Compile only some products:

```
sat compile <application> --products <product1>,<product2> ...
```

- Use *sat -t* to duplicate the logs in the terminal (by default the log are stored and displayed with *sat log* command):

```
sat -t compile <application> --products <product1>
```

- Compile a module and its dependencies:

```
sat compile <application> --products med --with_fathers
```

- Compile a module and the modules depending on it (for example plugins):

```
sat compile <application> --products med --with_children
```

- Clean the build and install directories before starting compilation:

```
sat compile <application> --products GEOM  --clean_all
```

Note:

a warning will be shown if option *–products* is missing
(as it will clean everything)

- Clean only the install directories before starting compilation:

```
sat compile <application> --clean_install
```

- Add options for make:

```
sat compile <application> --products <product> --make_flags <flags>
```

- Use the *–check* option to execute the unit tests after compilation:

```
sat compile <application> --check
```

- Remove the build directory after successful compilation (some build directory like qt are big):

```
sat compile <application> --products qt --clean_build_after
```

- Stop the compilation as soon as the compilation of a module fails:

```
sat compile <product> --stop_first_fail
```

- Do not compile, just show if products are installed or not, and where is the installation:

[1] http://www.salome-platform.org

```
sat compile <application> --show
```

2.4.3 Some useful configuration pathes

The way to compile a product is defined in the *pyconf file configuration*. The main options are:

- **build_source** : the method used to build the product (cmake/autotools/script)
- **compil_script** : the compilation script if build_source is equal to "script"
- **cmake_options** : additional options for cmake.
- **nb_proc** : number of jobs to use with make for this product.
- **check_install** : allow to specify a list of path (relative to install directory), that sat will check after installation. This flag allow to check an installation is complete.

2.5 Command launcher

2.5.1 Description

The **launcher** command creates a SALOME launcher, a python script file to start SALOME[2].

2.5.2 Usage

- Create a launcher:

  ```
  sat launcher <application>
  ```

  Generate a launcher in the application directory, i.e `$APPLICATION.workdir`.

- Create a launcher with a given name (default name is `APPLICATION.profile.launcher_name`)

  ```
  sat launcher <application> --name ZeLauncher
  ```

  The launcher will be called *ZeLauncher*.

- Set a specific resources catalog:

  ```
  sat launcher <application>  --catalog  <path of a salome resources catalog>
  ```

  Note that the catalog specified will be copied to the profile directory.

- Generate the catalog for a list of machines:

  ```
  sat launcher <application> --gencat <list of machines>
  ```

  This will create a catalog by querying each machine (memory, number of processor) with ssh.

- Generate a mesa launcher (if mesa and llvm are parts of the application). Use this option only if you have to use salome through ssh and have problems with ssh X forwarding of OpengGL modules (like Paravis):

  ```
  sat launcher <application> --use_mesa
  ```

2.5.3 Configuration

Some useful configuration pathes:

- **APPLICATION.profile**
  - **product** : the name of the profile product (the product in charge of holding the application stuff, like logos, splashscreen)
  - **launcher_name** : the name of the launcher.

[2] http://www.salome-platform.org

2.6 Command application

2.6.1 Description

The **application** command creates a virtual SALOME[3] application. Virtual SALOME applications are used to start SALOME when distribution is needed.

2.6.2 Usage

- Create an application:

  ```
  sat application <application>
  ```

  Create the virtual application directory in the salomeTool application directory `$APPLICATION.workdir`.

- Give a name to the application:

  ```
  sat application <application> --name <my_application_name>
  ```

  Remark: this option overrides the name given in the virtual_app section of the configuration file `$APPLICATION.virtual_app.name`.

- Change the directory where the application is created:

  ```
  sat application <application> --target <my_application_directory>
  ```

- Set a specific SALOME[4] resources catalog (it will be used for the distribution of components on distant machines):

  ```
  sat application <application> --catalog <path_to_catalog>
  ```

  Note that the catalog specified will be copied to the application directory.

- Generate the catalog for a list of machines:

  ```
  sat application <application> --gencat machine1,machine2,machine3
  ```

  This will create a catalog by querying each machine through ssh protocol (memory, number of processor) with ssh.

- Generate a mesa application (if mesa and llvm are parts of the application). Use this option only if you have to use salome through ssh and have problems with ssh X forwarding of OpengGL modules (like Paravis):

  ```
  sat launcher <application> --use_mesa
  ```

2.6.3 Some useful configuration pathes

The virtual application can be configured with the virtual_app section of the configutation file.

- **APPLICATION.virtual_app**
  - **name** : name of the launcher (to replace the default runAppli).
  - **application_name** : (optional) the name of the virtual application directory, if missing the default value is `$name + _appli`.

[3] http://www.salome-platform.org
[4] http://www.salome-platform.org

2.7 Command log

2.7.1 Description

The **log** command displays sat log in a web browser or in a terminal.

2.7.2 Usage

- Show (in a web browser) the log of the commands corresponding to an application:

  ```
  sat log <application>
  ```

- Show the log for commands that do not use any application:

  ```
  sat log
  ```

- The –terminal (or -t) display the log directly in the terminal, through a CLI[5] interactive menu:

  ```
  sat log <application> --terminal
  ```

- The –last option displays only the last command:

  ```
  sat log <application> --last
  ```

- To access the last compilation log in terminal mode, use –last_compile option:

  ```
  sat log <application> --last_compile
  ```

- The –clean (int) option erases the n older log files and print the number of remaining log files:

  ```
  sat log <application> --clean 50
  ```

2.7.3 Some useful configuration pathes

- **USER**
  - **browser** : The browser used to show the log (by default *firefox*).
  - **log_dir** : The directory used to store the log files.

[5] https://en.wikipedia.org/wiki/Command-line_interface

2.8 Command environ

2.8.1 Description

The **environ** command generates the environment files used to run and compile your application (as SALOME[6] is an example).

Note: these files are **not** required, salomeTool set the environment himself, when compiling. And so does the salome launcher.

These files are useful when someone wants to check the environment. They could be used in debug mode to set the environment for *gdb*.

The configuration part at the end of this page explains how to specify the environment used by sat (at build or run time), and saved in some files by *sat environ* command.

2.8.2 Usage

- Create the shell environment files of the application:

```
sat environ <application>
```

- Create the environment files of the application for a given shell. Options are bash, bat (for windows) and cfg (the configuration format used by SALOME[7]):

```
sat environ <application> --shell [bash|cfg|all]
```

- Use a different prefix for the files (default is 'env'):

```
# This will create file <prefix>_launch.sh, <prefix>_build.sh
sat environ <application> --prefix <prefix>
```

- Use a different target directory for the files:

```
# This will create file env_launch.sh, env_build.sh
# in the directory corresponding to <path>
sat environ <application> --target <path>
```

- Generate the environment files only with the given products:

```
# This will create the environment files only for the given products
# and their prerequisites.
# It is useful when you want to visualise which environment uses
# sat to compile a given product.
sat environ <application> --product <product1>,<product2>, ...
```

2.8.3 Configuration

The specification of the environment can be done through several mechanisms.

1. For salome products (the products with the property `is_SALOME_module` as `yes`) the environment is set automatically by sat, in respect with SALOME[8] requirements.
2. For other products, the environment is set with the use of the environ section within the pyconf file of the product. The user has two possibilities, either set directly the environment within the section, or specify a python script which wil be used to set the environment programmatically.

[6] http://www.salome-platform.org
[7] http://www.salome-platform.org
[8] http://www.salome-platform.org

Within the section, the user can define environment variables. He can also modify PATH variables, by appending or prepending directories. In the following example, we prepend *<install_dir>/lib* to `LD_LIBRARY_PATH` (note the *left first* underscore), append *<install_dir>/lib* to `PYTHONPATH` (note the *right last* underscore), and set `LAPACK_ROOT_DIR` to *<install_dir>*:

```
environ :
{
  _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
  PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  LAPACK_ROOT_DIR : $install_dir
}
```

It is possible to distinguish the build environment from the launch environment: use a subsection called *build* or *launch*. In the example below, `LD_LIBRARY_PATH` and `PYTHONPATH` are only modified at run time, not at compile time:

```
environ :
{
  build :
  {
    LAPACK_ROOT_DIR : $install_dir
  }
  launch :
  {
    LAPACK_ROOT_DIR : $install_dir
    _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
    PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  }
}
```

3. The last possibility is to set the environment with a python script. The script should be provided in the *products/env_scripts* directory of the sat project, and its name is specified in the environment section with the key `environ.env_script`:

```
environ :
{
  env_script : 'lapack.py'
}
```

Please note that the two modes are complementary and are both taken into account. Most of the time, the first mode is sufficient.

The second mode can be used when the environment has to be set programmatically. The developer implements a handle (as a python method) which is called by sat to set the environment. Here is an example:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import os.path
import platform

def set_env(env, prereq_dir, version):
    env.set("TRUST_ROOT_DIR",prereq_dir)
    env.prepend('PATH', os.path.join(prereq_dir, 'bin'))
    env.prepend('PATH', os.path.join(prereq_dir, 'include'))
    env.prepend('LD_LIBRARY_PATH', os.path.join(prereq_dir, 'lib'))
    return
```

SalomeTools defines four handles:

- **set_env(env, prereq_dir, version)** : used at build and run time.
- **set_env_launch(env, prereq_dir, version)** : used only at run time (if defined!)
- **set_env_build(env, prereq_dir, version)** : used only at build time (if defined!)

- **set_native_env(env)** : used only for native products, at build and run time.

2.9 Command clean

2.9.1 Description

The **clean** command removes products in the *source, build, or install* directories of an application. Theses directories are usually named `SOURCES, BUILD, INSTALL`.

Use the options to define what directories you want to suppress and to set the list of products

2.9.2 Usage

- Clean all previously created *build* and *install* directories (example application as *SALOME_xx*):

```
# take care, is long time to restore, sometimes
sat clean SALOME-xx --build --install
```

- Clean previously created *build* and *install* directories, only for products with property *is_salome_module*:

```
sat clean SALOME-xxx --build --install \
                     --properties is_salome_module:yes
```

2.9.3 Availables options

- **–products** : Products to clean.
- **–properties** :

  Filter the products by their properties.
  Syntax: *–properties <property>:<value>*

- **–sources** : Clean the product source directories.
- **–build** : Clean the product build directories.
- **–install** : Clean the product install directories.
- **–generated** : Clean source, build and install directories for generated products.
- **–package** : Clean the application package directory.
- **–all** : Clean the product source, build and install directories.
- **–sources_without_dev** :

  Do not clean the products in development mode,
  (they could have VCS[9] commits pending).

2.9.4 Some useful configuration pathes

No specific configuration.

[9] https://en.wikipedia.org/wiki/Version_control

2.10 Command package

2.10.1 Description

The **package** command creates a SALOME[10] archive (usually a compressed Tar[11] file .tgz). This tar file is used later to intall SALOME on other remote computer.

Depending on the selected options, the archive includes sources and binaries of SALOME products and prerequisites.

Usually utility *salomeTools* is included in the archive.

Note: By default the package includes the sources of prerequisites and products. To select a subset use the *–without_property* or *–with_vcs* options.

2.10.2 Usage

- Create a package for a product (example as *SALOME_xx*):

  ```
  sat package SALOME_xx
  ```

  This command will create an archive named `SALOME_xx.tgz` in the working directory (`USER.workDir`). If the archive already exists, do nothing.

- Create a package with a specific name:

  ```
  sat package SALOME_xx --name YourSpecificName
  ```

Note: By default, the archive is created in the working directory of the user (`USER.workDir`).

If the option *–name* is used with a path (relative or absolute) it will be used.

If the option *–name* is not used and binaries (prerequisites and products) are included in the package, the OS[12] architecture will be appended to the name (example: `SALOME_xx-CO7.tgz`).

Examples:

```
# Creates SALOME_xx.tgz in $USER.workDir
sat package SALOME_xx

# Creates SALOME_xx_<arch>.tgz in $USER.workDir
sat package SALOME_xx --binaries

# Creates MySpecificName.tgz in $USER.workDir
sat package SALOME_xx --name MySpecificName
```

- Force the creation of the archive (if it already exists):

  ```
  sat package SALOME_xx --force
  ```

- Include the binaries in the archive (products and prerequisites):

  ```
  sat package SALOME_xx --binaries
  ```

  This command will create an archive named `SALOME_xx _<arch>.tgz` where <arch> is the OS[13] architecture of the machine.

[10] http://www.salome-platform.org
[11] https://en.wikipedia.org/wiki/Tar_(computing)
[12] https://en.wikipedia.org/wiki/Operating_system
[13] https://en.wikipedia.org/wiki/Operating_system

- Do not delete Version Control System (VCS[14]) informations from the configurations files of the embedded salomeTools:

  ```
  sat package SALOME_xx --with_vcs
  ```

  The version control systems known by this option are CVS[15], SVN[16] and Git[17].

2.10.3 Some useful configuration pathes

No specific configuration.

[14] https://en.wikipedia.org/wiki/Version_control
[15] https://fr.wikipedia.org/wiki/Concurrent_versions_system
[16] https://en.wikipedia.org/wiki/Apache_Subversion
[17] https://git-scm.com

2.11 Command generate

2.11.1 Description

The **generate** command generates and compile SALOME modules from cpp modules using YACSGEN.

Note: This command uses YACSGEN to generate the module. It needs to be specified with *–yacsgen* option, or defined in the product or by the environment variable `$YACSGEN_ROOT_DIR`.

2.11.2 Remarks

- This command will only apply on the CPP modules of the application, those who have both properties:

  ```
  cpp : "yes"
  generate : "yes"
  ```

- The cpp module are usually computational components, and the generated module brings the CORBA layer which allows distributing the compononent on remore machines. cpp modules should conform to YACSGEN/hxx2salome requirements (please refer to YACSGEN documentation)

2.11.3 Usage

- Generate all the modules of a product:

  ```
  sat generate <application>
  ```

- Generate only specific modules:

  ```
  sat generate <application> --products <list_of_products>
  ```

  Remark: modules which don't have the *generate* property are ignored.

- Use a specific version of YACSGEN:

  ```
  sat generate <application> --yacsgen <path_to_yacsgen>
  ```

2.12 Command config

2.12.1 Description

The **init** command manages the sat local configuration (which is stored in the data/local.pyconf file). It allows to initialise the content of this file.

2.12.2 Usage

- A sat project provides all the pyconf files relatives to a project (salome for example). Use the *–add_project* command to add a sat project locally, in data/local.pyconf (by default sat comes without any project). It is possible to add as many projects as required.

```
sat init --add_project <path/to/a/sat/project/project.pyconf>
```

- If you need to remove a sat project from the local configuration, use the *–reset_projects* command to remove all projects and then add the nex ones with *–add_project*:

```
sat init --reset_projects
sat init --add_project <path/to/a/new/sat/project/project.pyconf>
```

- By default the product archives are stored locally within the directory containing salomeTool, in a subdirectory called ARCHIVES. If you want to change the default, use the *–archive_dir* option:

```
sat init --archive_dir  <local/path/where/to/store/product/archives>
```

- sat enable a **base** mode, which allow to mutualize product builds between several applications. By defaut, the mutualised builds are stored locally within the directory containing salomeTool, in a subdirectory called BASE. To change the default, use the *–base* option:

```
sat init --base <local/path/where/to/store/product/mutualised/product/builds>
```

- In the same way, you can use the *–workdir* and *–log_dir* commands to change the default directories used to store the application builds, and sat logs:

```
sat init --workdir <local/path/where/to/store/applications>
sat init --log_dir <local/path/where/to/store/sat/logs>
```

2.12.3 Some useful configuration pathes

All the sat init commands update the local pyconf salomeTool file data/local.pyconf. The same result can be achieved by editing the file directly. The content of data/local.pyconf is dumped into two sat configuration variables:

- **LOCAL**: Contains notably all the default paths in the fields archive_dir, base, log_dir and workdir.
- **PROJECTS**: The field project_file_paths contains all the project files that have been included with –add_project option.

sat commands:

```
sat config -v LOCAL
sat config -v PROJECTS
```

2.13 Command environ

2.13.1 Description

The **environ** command generates the environment files used to run and compile your application (as SALOME[18] is an example).

Note: these files are **not** required, salomeTool set the environment himself, when compiling. And so does the salome launcher.

These files are useful when someone wants to check the environment. They could be used in debug mode to set the environment for *gdb*.

The configuration part at the end of this page explains how to specify the environment used by sat (at build or run time), and saved in some files by *sat environ* command.

2.13.2 Usage

- Create the shell environment files of the application:

```
sat environ <application>
```

- Create the environment files of the application for a given shell. Options are bash, bat (for windows) and cfg (the configuration format used by SALOME[19]):

```
sat environ <application> --shell [bash|cfg|all]
```

- Use a different prefix for the files (default is 'env'):

```
# This will create file <prefix>_launch.sh, <prefix>_build.sh
sat environ <application> --prefix <prefix>
```

- Use a different target directory for the files:

```
# This will create file env_launch.sh, env_build.sh
# in the directory corresponding to <path>
sat environ <application> --target <path>
```

- Generate the environment files only with the given products:

```
# This will create the environment files only for the given products
# and their prerequisites.
# It is useful when you want to visualise which environment uses
# sat to compile a given product.
sat environ <application> --product <product1>,<product2>, ...
```

2.13.3 Configuration

The specification of the environment can be done through several mechanisms.

1. For salome products (the products with the property `is_SALOME_module` as `yes`) the environment is set automatically by sat, in respect with SALOME[20] requirements.
2. For other products, the environment is set with the use of the environ section within the pyconf file of the product. The user has two possibilities, either set directly the environment within the section, or specify a python script which wil be used to set the environment programmatically.

[18] http://www.salome-platform.org
[19] http://www.salome-platform.org
[20] http://www.salome-platform.org

Within the section, the user can define environment variables. He can also modify PATH variables, by appending or prepending directories. In the following example, we prepend *<install_dir>/lib* to `LD_LIBRARY_PATH` (note the *left first* underscore), append *<install_dir>/lib* to `PYTHONPATH` (note the *right last* underscore), and set `LAPACK_ROOT_DIR` to *<install_dir>*:

```
environ :
{
  _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
  PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  LAPACK_ROOT_DIR : $install_dir
}
```

It is possible to distinguish the build environment from the launch environment: use a subsection called *build* or *launch*. In the example below, `LD_LIBRARY_PATH` and `PYTHONPATH` are only modified at run time, not at compile time:

```
environ :
{
  build :
  {
    LAPACK_ROOT_DIR : $install_dir
  }
  launch :
  {
    LAPACK_ROOT_DIR : $install_dir
    _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
    PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  }
}
```

3. The last possibility is to set the environment with a python script. The script should be provided in the *products/env_scripts* directory of the sat project, and its name is specified in the environment section with the key `environ.env_script`:

```
environ :
{
  env_script : 'lapack.py'
}
```

Please note that the two modes are complementary and are both taken into account. Most of the time, the first mode is sufficient.

The second mode can be used when the environment has to be set programmatically. The developer implements a handle (as a python method) which is called by sat to set the environment. Here is an example:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import os.path
import platform

def set_env(env, prereq_dir, version):
    env.set("TRUST_ROOT_DIR",prereq_dir)
    env.prepend('PATH', os.path.join(prereq_dir, 'bin'))
    env.prepend('PATH', os.path.join(prereq_dir, 'include'))
    env.prepend('LD_LIBRARY_PATH', os.path.join(prereq_dir, 'lib'))
    return
```

SalomeTools defines four handles:

- **set_env(env, prereq_dir, version)** : used at build and run time.
- **set_env_launch(env, prereq_dir, version)** : used only at run time (if defined!)
- **set_env_build(env, prereq_dir, version)** : used only at build time (if defined!)

- **set_native_env(env)** : used only for native products, at build and run time.

CHAPTER

THREE

DEVELOPER DOCUMENTATION

3.1 Add a user custom command

3.1.1 Introduction

Note: This documentation is for Python[1] developers.

The salomeTools product provides a simple way to develop commands. The first thing to do is to add a file with *.py* extension in the `commands` directory of salomeTools.

Here are the basic requirements that must be followed in this file in order to add a command.

3.1.2 Basic requirements

By adding a file *mycommand.py* in the `commands` directory, salomeTools will define a new command named `mycommand`.

In *mycommand.py*, there must be the following method:

```
def run(args, runner, logger):
    # your algorithm ...
    pass
```

In fact, at this point, the command will already be functional. But there are some useful services provided by salomeTools :

- You can give some options to your command:

```
import src

# Define all possible option for mycommand command :  'sat mycommand <options>'
parser = src.options.Options()
parser.add_option('m', 'myoption', \
                  'boolean', 'myoption', \
                  'My option changes the behavior of my command.')

def run(args, runner, logger):
    # Parse the options
    (options, args) = parser.parse_args(args)
    # algorithm
```

- You can add a *description* method that will display a message when the user will call the help:

```
import src

# Define all possible option for mycommand command : 'sat mycommand <options>'
parser = src.options.Options()
parser.add_option('m', 'myoption', \
                  'boolean', 'myoption', \
                  'My option changes the behavior of my command.')

def description():
    return _("The help of mycommand.")

def run(args, runner, logger):
    # Parse the options
    (options, args) = parser.parse_args(args)
    # algorithm
```

[1] https://docs.python.org/2.7

3.1.3 HowTo access salomeTools config and other commands

The *runner* variable is an python instance of *Sat* class. It gives access to *runner.cfg* which is the data model defined from all *configuration pyconf files* of salomeTools For example, *runner.cfg.APPLICATION.workdir* contains the root directory of the current application.

The *runner* variable gives also access to other commands of salomeTools:

```
# as CLI_ 'sat prepare ...'
runner.prepare(runner.cfg.VARS.application)
```

3.1.4 HowTo logger

The logger variable is an instance of the Logger class. It gives access to the write method.

When this method is called, the message passed as parameter will be displayed in the terminal and written in an xml log file.

```
logger.write("My message", 3) # 3 as default
```

The second argument defines the level of verbosity that is wanted for this message. It has to be between 1 and 5 (the most verbose level).

3.1.5 HELLO example

Here is a *hello* command, file *commands/hello.py*:

```
import src

"""
hello.py
Define all possible options for hello command:
sat hello <options>
"""

parser = src.options.Options()
parser.add_option('f', 'french', 'boolean', 'french', "french set hello message in french.")

def description():
    return _("The help of hello.")

def run(args, runner, logger):
    # Parse the options
    (options, args) = parser.parse_args(args)
    # algorithm
    if not options.french:
        logger.write('HELLO! WORLD!\n')
    else:
        logger.write('Bonjour tout le monde!\n')
```

A first call of hello:

```
# Get the help of hello:
./sat --help hello

# To get bonjour
./sat hello --french
Bonjour tout le monde!

# To get hello
./sat hello
```

```
HELLO! WORLD!

# To get the log
./sat log
```

CHAPTER

FOUR

RELEASE NOTES

4.1 SAT version 5.5.0

4.1.1 Release Notes, November 2019

Warning: This documentation is under construction!

New features and improvments

sat package

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description

4.2 SAT version 9.4.0

4.2.1 Release Notes, April, 2019

New features and improvments

sat package

The sat package command has been completed and finalised, in order to manage standalone packages of sat, with or without an embedded project. Options **–ftp** and **–with_vcs** have been added, in order to reduce the size of salome project packages (without these options, the archive of the sat salome project is huge, as it includes all the prerequisites archives. The **–ftp** option allows pointing directly to salome ftp site, which provides the prerequisites archives. These are therefore not included. With the same approach, **–with_vcs** option specify an archive that points directly to the git bases of SALOME. Sources of SALOME modules are therefore not embedded in the archive, reducing the size.

```
# produce a standalone archive of salomeTool
sat package -t

# produce a HUGE standalone archive of salomeTool with the salome project embedded.
sat package -t -p salome

# produce a small archive with salomeTool and embedded salome project,
# with direct links to ftp server and git repos
sat package -t -p salome --ftp --with_vcs
```

repo_dev property

This new application property **repo_dev** was introduced to trigger the use of the development git repositories for all the git bases of an application. Before, the only way to use the development git repositories was to declare all products in dev mode. This was problematic, for example one had to use *–force_patch* option to apply patches, or to use *–force* option to reinstall sources.

The use of the development git repository is now triggered by declaring this new **repo_dev** property in the application. And products are declared in dev mode only if we develop them.

```
# add this section in an application to force the use of the development git bases
# (from Tuleap)
properties :
{
    mesa_launcher_in_package : "yes"
    repo_dev : "yes"
}
```

windows compatibility

The compatibility to windows platform has been improved. The calls to lsb_release linux command have been replaced by the use of python platform module. Also the module med has been renamed medfile, and module Homard has been renamed homard_bin, in order to avoid lower/upper case conflicts.

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #12099	Add a new field called check_install to verify the correct installation
sat #8607	Suppression of sat profile command, replaced by sat template command (AppModule)
69d6a69f43	Introduction of a new property called repo_dev to trigger the use of the dev git repository.
scs #13187	Update of PythonComponent template
sat #16728	Replace call to lsb_release by platform module
sat #13318 sat #16713	command sat package -t -p salome –ftp –with_vcs debug of sat packages containing sat and embedded projects
sat #16787	Rename product med by medfile and Homard by homard_bin

4.3 SAT version 5.3.0

4.3.1 Release Notes, February, 2019

New features and improvments

sat init

The command *sat init* has been finalized, with the addition of options **–add_project** and **–reset_projects**. It is now able to manage projects after an intiale git clone of salomeTool. The capacity is used by users installing salomeTool from the git repositories:

```
# get sources of salomeTool
git clone https://codev-tuleap.cea.fr/plugins/git/spns/SAT.git salomeTool

# get SAT_SALOME project (the sat project that contains the configuration of SALOME)
git clone https://codev-tuleap.cea.fr/plugins/git/spns/SAT_SALOME.git

# initialise sat with this project
salomeTool/sat init --add_project $(pwd)/SAT_SALOME/salome.pyconf
```

It is possible to initialise sat with several projects by calling several times *sat init –add_project*

sat prepare : git retry functionnality

With large git repositories (>1GB) *git clone* command may fail. To decrease the risk, sat prepare will now retry three times the *git clone* function in case of failure.

Reset of LD_LIBRARY_PATH and PYTHONPATH before setting the environment

Every year, a lot of problems occur, due to users (bad) environment. This is most of the time caused by the presence (out-of-date) .bashrc files. To prevent these (time-consuming) problems, sat now reset LD_LIBRARY_PATH and PYTHONPATH variables before setting the environment thus avoiding side effects. Users who wish anayway to start SALOME with a non empty LD_LIBRARY_PATH or PYTHONPATH may comment the reset in salome launcher or in env_launch.sh file.

New option –complete for sat prepare

This option is used when an installation is interrupted or incomplete. It allows downloading only the sources of missing products

```
# only get sources of missing products (i.e products not present in INSTALL dir)
git prepare SALOME-master -c
```

** New option –packages for sat clean**

SALOME packages are big... It is usefull to be able to clean them whith this new option.

```
# remove packages present in PACKAGES directory of SALOME-master
git clean SALOME-master --packages
```

Global configuration keys “debug”, “verbose” and “dev” in applications

These new keys can be defined in applications in order to triger the debug, verbose and dev mode for all products. In the following example, the SALOME-master application will be compiled in debug mode (use of **-g** flag), but with no verbosity. Its products are not in development mode.

```
APPLICATION :
{
    name : 'SALOME-master'
    workdir : $LOCAL.workdir + $VARS.sep + $APPLICATION.name + '-' + $VARS.dist
    tag : 'master'
    dev : 'no'
    verbose :'no'
    debug : 'yes'
    ...
}
```

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #16548 sat #8566	Finalisation of sat init command (options -add_project and –reset_projects)
sat #12994	new git retry functionnality for sat prepare : give three trials in case of failures
sat #8581	traceability : save tag of salomeTool and its projects
sat #8588	reset LD_LIBRARY_PATH and PYTHONPATH before launching SALOME
sat #9575	Improvment of the DISTENE licences management (notably for packages)
sat #8597	Implementation of option sat prepare -c (–complete) for preparing only the sources that are not yet installed
sat #8655	implementation of option sat clean –packages
sat #8532 sat #8594	sat log : remane option –last_terminal in –last_compile Extension of sat log –last_compile to the logs of make check
sat #13271	hpc mode trigered by product “hpc” key in state of MPI_ROOT variable
sat #8606	sat generate clean old directories before a new generation
sat #12952	Add global keys “debug”, “verbose” and “dev” to manage globally these modes for all the products of an application
sat #8523	protection of call to ssh on windows platform

4.4 SAT version 5.5.0

4.4.1 Release Notes, December, 2018

Warning: This documentation is under construction!

New features and improvments

sat package

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description