Nliteme Installation Guide

Contents

1. Overview

An Nliteme is a web based test management application optimized for collecting and presenting test execution results from automated regression test systems. It provide an easy, cUrl based API for uploading the data, allowing an integration “as is” with an existing test systems and feeding data in almost no time.

The web based GUI allows the user to quickly browse, search, compare the test results from different SW releases and provides efficient filtering option to quickly spot the problems in the tested SW and/or HW.

Its configurable dashboard provides a quick overview of the latest testing activities whereas the High Level Report view provides a detailed summary of the test results from different test lines and SW release, which can be useful for project management and planning.

Nliteme is a platform independent application and can be deployed on Unix, Linux, Mac Os X and Windows systems.

This documents describes all steps needed to successfully install Nliteme, configure it and populate it with data.

2. Prerequisites

Nliteme application is platform independent and can be deployed on Unix, Linux, Mac Os X and Windows platforms.

It requires an installation of a

- Web Server e.g. Apache 2.4.25
- MariaDB 10.1 or higher
- PHP 7.2

All the above mentioned SW can be found in an easy to install XAMPP distribution or can be installed as part of LAMP server. However, it shall be noted the XAMPP is not meant for production use, but only for development environments (XAMPP is configured to be open as possible, in a production environment, it could be fatal). Therefore, it is recommended to use the proper SW installation for production mode.

2.1 Mysql/MariaDB initial configuration

2.1.1 Configure InnoDB as a default engine

The default engine can be checked by the following command:

```
# mysql -uroot
MariaDB [(none)]> show engines;
```

To change the setting edit a my.cnf file:

```
# vi my.cnf
default-storage-engine = INNODB
```

2.1.2 Optimize buffer pool size for a better performance

The optimal settings depends on system architecture (32-bit vs. 64-bit system), and available RAM. The bigger the buffer pool size is, the better application performance can be expected (for a big database). For the restriction on setting the size please refer to my.cnf description.

In case of Xampp installation the user may choose already predefined my.ini for big database (my-innodb-heavy-4G.ini).

```
# vi my.cnf
innodb_buffer_pool_size = 2G
```

For further data base optimization options please refer to mysql or mariaDB manuals.

Note. Performance optimization parameters set used in NR PRG project:

```
innodb_buffer_pool_size         = 16G   # Go up to 80% of your available RAM
```

```
innodb_buffer_pool_instances   = 8     # Bigger if huge InnoDB Buffer Pool or high
concurrency

innodb_file_per_table          = 1    # Is the recommended way nowadays
innodb_write_io_threads        = 8    # If you have a strong I/O system or SSD
innodb_read_io_threads         = 8    # If you have a strong I/O system or SSD
innodb_io_capacity             = 1000 # If you have a strong I/O system or SSD

innodb_flush_log_at_trx_commit = 2    # 1 for durability, 0 or 2 for performance
innodb_log_buffer_size         = 8M   # Bigger if innodb_flush_log_at_trx_commit = 0
innodb_log_file_size           = 256M # Bigger means more write throughput but
longer recovery time
```

2.1.3 Create a database, a user and grant privileges

- create user

```
# mysql –uroot
 MariaDB [(none)]> CREATE USER 'nliteme'@'localhost' IDENTIFIED BY
 'nlitemepass';
```

- create database

```
# mysql –uroot
 MariaDB [(none)]> CREATE DATABASE nlitemedb;
```

- allow the user to connect to the server from localhost using the password:

```
# mysql –uroot
MariaDB [(none)]> grant usage on *.* to nliteme@localhost identified by
'nlitemepass';
```

- grant all privileges on the nlitemedb database to this user:

```
# mysql –uroot
MariaDB [(none)]> grant all privileges on nlitemedb.* to nliteme@localhost;
```

3. Installation of Nliteme application

3.1 Download from git

Clone or download nliteme from Git. Copy nliteme subfolder to the web server document root folder (or any other location visible from the web server). This folder will be further referred as *$webSiteRoot*

3.2 Configure database name, user and password

Edit *$webSiteRoot/nliteme/libs/ dbconnect.php* and enter the name of the database, username and password set in sec. 1.1.3

```
# cd $webSiteRoot/nliteme/libs
# vi dbconnect.php
```

3.3 Create a database schema, initialize a preferences table

The following steps creates the nliteme tables and load an initial, default configuration

```
# cd $webSiteRoot/nliteme/struct
# mysql -unliteme -pnlitemepass nlitemedb < nliteme_schema.sql
# mysql -unliteme -pnlitemepass nlitemedb < nliteme_preferences_init.sql
```

3.4 Modify table columns preferences

This step allows the user to customize the handling of certain table columns e.g. enable/disable columns, make them showable (i.e. visible on the web page), make them searchable (i.e. visible and selectable in the search forms).

Please note the modifications done here without understanding may cause a wrong behavior of the application.

The user may also execute the scripts without modifying them to ensure the correct settings are applied.

Note. This step shall not be done on working system, as it may delete some existing data e.g. predefined values for table columns marked as predefined. If the user want to modify column's preferences on the running system, he needs to prepare a php code similar to $webSiteRoot/nliteme/scripts/modifyPrefs.php (see section Modifying column preferences for more information)

To modify the test results table columns preferences edit the following file:

```
# cd $webSiteRoot/nliteme/struct
# vi genColumnConfigTestresults.php
# php genColumnConfigTestresults.php
```

To modify all other tables columns preferences edit the following file:

```
# cd $webSiteRoot/nliteme/struct
# vi genColumnConfigRest.php
# php genColumnConfigRest.php
```

3.5 Set build compare preferences

This step is optional and is needed only if some custom preferences shall be applied for a build compare feature

```
# cd $webSiteRoot/nliteme/struct
# vi initBCPrefs.php
# php initBCPrefs.php
```

4. Naming conventions

The data stored in the Nliteme database can be divided in the following types:

- Build - a software build to be tested
- Build Increment – a software branch, iteration or stream which groups builds using the same interface release and/or sharing the same feature set etc.
- Feature - a feature to be tested by the test cases.
- Test Case – a test scenario testing a certain feature of the software
- Test Suite – a collection of test cases with common properties e.g. using the same API
- Test line – an entity consisting of HW and SW being tested, control PC running an automation SW and additional devices (e.g. Power supply, Ues etc.)
- Test Result – a result of a single execution of a test case on a particular test line using a particular SW build. It provides a test verdict, a link to traces, an indication of the used test line, build as well as a description of the test execution sequence

From Nliteme point of view there are no restrictions on the names of the above types. The only restriction is the names must be unique i.e. two different build cannot have the same name (the same applies to the other data type). The test results are distinguished by the trace archive file path and it also has to be unique.

5. Uploading data to the database

The Nliteme offers an API for uploading a data. This is a preferred way of feeding the database with a data. The input data shall be a UTF-8 encoded json string.

5.1 Input data format

The input data shall be provided in the form of a UTF-8 encoded json string containing key, value pairs representing data to be put in to certain columns in the database tables. The column specific key value pair need to have the format as follows:

```
"<column_name>":{"realname":"<column_name>","value":"<column_value>"}
```

For example a "build" column information would be structured as follows:

```
"build":{"realname":"build","value":"v_1.1"}
```

The list of columns specific key, value pairs constitute a record information. Please note it does not mean a single record in a specific single table in the database. It is rather a single information record which is used to update several database tables, depending on it's content and an upload type. The upload type specifies what kind of data are represented in the json string i.e. if it is a build, test case, test result, test line etc. specific information.

The record information can be collected in a json encoded associative list with keys being consecutive indices or stored in a file, where each line contains a single record information. The 1st approach is used to upload a single or several record information, whereas the 2nd for a mass upload i.e. for uploading several hundreds or thousands record information at once.

Below is an example of an associative list containing a single record information for uploading "*build*" information (uploadtype = "builds"):

```
{"0":{
"build":{"realname":"build","value":" v_1.1"}
,"increment":{"realname":"increment","value":" v_1"}
,"description":{"realname":"description","value":"Detailed info"}
}}
```

Below is an example content of a file used for mass upload of "build" information (uploadtype = "builds"):

```
{"build":{"realname":"build","value":"v_1.1"},"increment":
{"realname":"increment","value":"v_1"},"description":
{"realname":"description","value":"Detailed info"}}
{"build":{"realname":"build","value":"v_1.2"},"increment":
{"realname":"increment","value":"v_1"},"description":
{"realname":"description","value":"Detailed info"}}
{"build":{"realname":"build","value":"v_2.1"},"increment":
{"realname":"increment","value":"v_2"},"description":
{"realname":"description","value":"Detailed info"}}
```

The type of the record information is passed to the application via HTTP POST argument 'uploadtype'. The following upload types are currently supported:

- 'testresults' – for test result's specific record information
- 'builds' – for build's specific record information
- 'buildincrements' - for build increment's specific record information
- 'testcases' – for test case's specific record information
- 'testsuites' – for test suite's specific record information
- 'testlines' – for test line's specific record information

The following subsections describe structures of the record information for all those types.

5.1.1 'testresults' record information structure

A test results specific record has a following structure. The "*filepath*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

For the new record also the "*build*", "*increment*", "*tlname*", "*tsname*", "*tcname*", "*tcverdict*", "*duration*" fields shall be provided.

Optionally, the "*fname*", "*coverage*" can be provided to define the feature associated with the testcase together with the percentage of the feature covered by that test case. Note. It is recommended to create feature and test case records separately (see sec. 5.1.4 and sec. 5.1.5)

The "*extracolumn_2*" is reserved for the defect number/id. The defect number can be associated with the test result either when creating the test result record or later for already existing test result record (the details on how to update the defect number can be found in sample upload script (described in sec. 5.2.). Note that it is assumed the

details of the defects associated to the failing test results are hosted externally (using dedicated defect tracking tools)

```
{"filepath": {"realname": "filepath", "value": "<must be unique for every test
result record>"},
"createdate": {"realname": "createdate", "value": "YYYY-MM-DD hh:mm:ss"},
"build": {"realname": "build", "value": "<build_name>"},
"increment": {"realname": "increment", "value": "<increment_name>"},
"tlname": {"realname": "tlname", "value": "<test line name>"},
"tsname": {"realname": "tsname", "value": "<test suite name>"},
"tcname": {"realname": "tcname", "value": "<test case name>"},
"fname": {"realname": "fname", "value": "<feature name>"},
"coverage": {"realname": "coverage", "value": "<feature coverage>"},
"tcverdict": {"realname": "tcverdict", "value": "<set "ok" if passed, anything for
failed>"},
"duration": {"realname": "duration", "value": <int number of seconds it took to
execute>},
"extracolumn_0": {"realname": "extracolumn_0", "value": "<value depends how/if a
columns is configured>"},
"extracolumn_1": {"realname": "extracolumn_1", "value": "<value depends how/if a
columns is configured>"}
"extracolumn_2": {"realname": "extracolumn_2", "value": "<value depends how/if a
columns is configured>"},
"extracolumn_3": {"realname": "extracolumn_3", "value": "<value depends how/if a
columns is configured>"},
"description": {"realname": "description", "value": "<put a description here in UTF-
8 encoded format>"},
}
```

5.1.2 'builds' record information structure

A build specific record has a following structure. The "*build*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

```
{"build":{"realname":"build","value":"<build name, must be unique for different
builds>"}
,"increment":{"realname":"increment","value":"<increment name>"}
,"createdate": {"realname": "createdate", "value": "YYYY-MM-DD hh:mm:ss"},
,"description":{"realname":"description","value":"<put a description here in UTF-8
encoded format>"}}
```

5.1.3 'buildincrements' record information structure

A build increment specific record has a following structure. The "*increment*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

```
{"increment":{"realname":"increment","value":"<increment name, must be unique for
different increments>"}
,"description":{"realname":"description","value":"<put a description here in UTF-8
encoded format>"}}
```

5.1.4 'features' record information structure

A feature specific record has a following structure. The "*fname*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

```
{"fname":{"realname":"fname","value":"<test case name, must be unique for different
test cases>"}
```

```
,"hlink": {"realname": "hlink", "value": "<optional hyperlink to external server
hosting features definitions"},
,"createdate": {"realname": "createdate", "value": "YYYY-MM-DD hh:mm:ss"},
,"description":{"realname":"description","value":"<put a description here in UTF-8
encoded format>"}}
```

5.1.5 'testcases' record information structure

A test case specific record has a following structure. The "*tcname*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

```
{"tcname":{"realname":"tcname","value":"<test case name, must be unique for
different test cases>"}
,"fname": {"realname": "fname", "value": "<feature name>"},
,"coverage": {"realname": "coverage", "value": "<percentage feature coverage>"},
,"createdate": {"realname": "createdate", "value": "YYYY-MM-DD hh:mm:ss"},
,"description":{"realname":"description","value":"<put a description here in UTF-8
encoded format>"}}
```

5.1.6 'testsuites' record information structure

A test suite specific record has a following structure. The "*tsname*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

```
{"tsname":{"realname":"tsname","value":"<test suite name, must be unique for
different test suites>"}
,"createdate": {"realname": "createdate", "value": "YYYY-MM-DD hh:mm:ss"},
,"description":{"realname":"description","value":"<put a description here in UTF-8
encoded format>"}}
```

5.1.7 'testlines' record information structure

A test line specific record has a following structure. The "*tlname*" is a mandatory key. For a new record has to be unique value, otherwise the existing record will be updated.

```
{"tlname":{"realname":"tlname","value":"<test line name, must be unique for
different test lines>"}
,"createdate": {"realname": "createdate", "value": "YYYY-MM-DD hh:mm:ss"},
,"description":{"realname":"description","value":"<put a description here in UTF-8
encoded format>"}}
```

5.2 Sample upload script

Sample python code scripts containing ready to use examples of uploading different type of data into the database can be found in *$webSiteRoot/nliteme/uploadclients/*

Please note that the application is able to create new entries for the build, build increment, test case, test suite, test line, tc verdict etc. based on the information included in the data structure used for the upload test results. This means that it is not necessary to separately create entries in the data base for each type of information. For example uploading the 1st test result of a test case "x" for a build "y" tested on test line "z" will also create entries for the test case, build and test line, if they are not present yet.

In some case it might be beneficial to create some entries separately. For example the user may want to create a build entry as soon as the build process is done and additionally include a detailed description of what changes, features and fixes this build contains. Please refer to the examples in the above mentioned folder.

Here is a short summary of the example scripts:

- *$webSiteRoot\nliteme\uploadclients\uploadTestResultExample\uploadTestResultExample.py*

  shows how to upload a single test result (note. the build, test line, test case etc. entries will also be created, if not present) or update the defect number for the existing test result record

```
# python.exe uploadTestResultExample.py
```

- *$webSiteRoot\nliteme\uploadclients\uploadTestResultExample\generateTrAndUpload.py*
shows how to upload multiple test results at once. The script generates an example test results content, stores it in a file and uploads the file into the nliteme application.

```
# python.exe generateTrAndUpload.py
```

- *$webSiteRoot\nliteme\uploadclients\uploadBuildExample\uploadBuildInfo2nliteme.py*
A PRG LTE proprietary build information upload script, supporting command line options for providing files with description (e.g. list of RTC Changesets, Baselines etc.):

```
# python.exe uploadBuildInfo2nliteme.py -n <buildname> -c <changesetdifffile>
-b <baselinefile> -t <buildtype>

where:
⇨   buildname is a name of the SW build. To make the script work properly the
build name needs to have a certain format e.g. 33.10_2017-03-16-13-16
⇨   changesetdifffile file containing changeset differences to previous build
(optional)
⇨   baselinefile file containing list of RTC baselines (optional)
⇨   buildtype additional, short string to differentiate between base and
integration build (internal feature) (optional)

e.g simply create an entry for a build 33.10_2017-03-16-13-16
# python.exe uploadBuildInfo2nliteme.py -n 33.10_2017-03-16-13-16
```

- sample scripts for creating single entries for builds, build increments, test lines, test cases, test suites, test feature in the data base
$webSiteRoot\nliteme\uploadclients\uploadBuildExample\uploadBuildSimple.py
$webSiteRoot\nliteme\uploadclients\uploadBuildIncrementExample\uploadBuildIncrementSimple.py
$webSiteRoot\nliteme\uploadclients\uploadTestCaseExample\uploadTestCaseSimple.py
$webSiteRoot\nliteme\uploadclients\uploadTestSuiteExample\uploadTestSuiteSimple.py
$webSiteRoot\nliteme\uploadclients\uploadTestSuiteExample\uploadTestLineSimple.py
$webSiteRoot\nliteme\uploadclients\uploadFeatureExample\uploadFeatureSimple.py

6. Configuring dashboard

Currently the dashboard configuration requires several manual steps, due to a missing configuration GUI and afact that the widgets and a dashboard configurations have not been moved to the database yet.

6.1 Configure the content of widgets to display.

The widget's definitions are located in *$webSiteRoot\nliteme\libs\controllers\widgetcontroller.php* in a class TempConfig (Temporary used, before widget config is places in the DB).

Here is an example of such a definition:

```
$widgetParams = new WidgetParams(null);
$widgetParams->setParam('numOfDays', 3); # last n days within which to search for the results
$widgetParams->setParam('numOfBuilds', 5); # limit the search to the latest n builds (if no results are found within last n days)
$widgetParams->setParam('groupingColumns',
array('increment','build','tsname','tlname','extracolumn_0'));
# the 'groupingColumns' parameter defines how the results shall be grouped in the summary (in the above example the statistics will be printed grouped by build increment, build, test suite, test line and UE type – by default the extracolumn_0 is used for storing UE types
$widgetParams->setParam('staticConditions',
array('incid'=>array('39','40'),'tsid'=>array('1','4')));
# the 'staticConditions' parameter defines the filtering for search query i.e. which incid (build increment ids) and which tsid (test suites id) shall be used for searching. Note the indexes must be manually read from the DB (as no GUI is available yet for widget configuration), whenever a new widget is supposed to be added or a new value (e.g. new build increment e.g. 33.19) is added to the DB
```

```
$widgetConf = new WidgetConfig(null);
$widgetConf->setName('TestActivities33.1x_TS1'); # name referenced in the dashboard config
$widgetConf->setTitle('MLAPI-RT Latest Activities'); # Title printed in the widget header
$widgetConf->setUrl('action=com.nliteme.TestActivities&widget=TestActivities33.1x_TS1');
$widgetConf->setUseIframe('no');
$widgetConf->setWidgetParams($widgetParams);
self::$widgets->addItem($widgetConf);
```

6.2 Configure the dashboard by adding tabs and filling them with widget references

First the dashboard configuration string has to be generated for that:

- Edit *$webSiteRoot\nliteme\scripts\genDashboardContent.php* file and add the definitions for tab with columns containing widget references e.g. a definition of one tab containing 3 columns, 1st with one, 2nd with 3 and 3rd with 2 widgets could look like that:

```
 $wCL_0 = new WidgetConfigList(array( getWidgetConfigStripped(TempConfig::getInstance()-
>getWidgetConfigList()->getItemByName('TestActivities33.1x'))));
 $wCL_1 = new WidgetConfigList(array( getWidgetConfigStripped(TempConfig::getInstance()-
>getWidgetConfigList()->getItemByName('TestActivities33.1x_TS1'))
                                    ,getWidgetConfigStripped(TempConfig::getInstance()-
>getWidgetConfigList()->getItemByName('TestActivities33.1x_TS4'))
                                    ,getWidgetConfigStripped(TempConfig::getInstance()-
>getWidgetConfigList()->getItemByName('TestActivities33.1x_TS5'))));

 $wCL_2 = new WidgetConfigList(array( getWidgetConfigStripped(TempConfig::getInstance()-
>getWidgetConfigList()->getItemByName('TestActivities33.1x_TS2'))
                                    ,getWidgetConfigStripped(TempConfig::getInstance()-
>getWidgetConfigList()->getItemByName('TestActivities33.1x_TS3'))));
    $dColumn_1 = new DashboardColumn(array($wCL_0));
    $dColumn_2 = new DashboardColumn(array($wCL_1));
    $dColumn_3 = new DashboardColumn(array($wCL_2));
    $dColumnList = new DashboardColumnList(array($dColumn_1));
    $dColumnList->addItem($dColumn_2);
    $dColumnList->addItem($dColumn_3);
    $dTab_1 = new DashboardTab(array($dColumnList));
    $dTab_1->setName('v33.1x');
```

Note that the widget references must point to the existing widget definitions defined as in step 1 (see section 4.1)

- Generate the corresponding json string:

```
# php $webSiteRoot\nliteme\genDashboardContent.php
```

The newly generated json string shall be assigned to the parameter $dashboardConfigString in the *$webSiteRoot\nliteme\libs\models\dashboardmodel.php*

7. Modifying column preferences

When the database has already been initialized and tables contain some data it might be necessary to modify some table columns settings. For example the user may want to enable one of the extracolumns in the testresults table.

Since so far, there is no configuration GUI, this action requires writing a small piece of code. Here is an example on how to enable an extracolumn_3, make it visible in the web page and searchable i.e. enable it in the filtering options in the search form of the test results:

```
$columnConfig = Config::getInstance()->getColumnConfig('testresults');

$columnConfig->getDbColumnByRealName('extracolumn_3')->setParam('showable','yes');
$columnConfig->getDbColumnByRealName('extracolumn_3')->setParam('iteratable','yes');
$columnConfig->getDbColumnByRealName('extracolumn_3')->setParam('enabled','yes');

Config::getInstance()->updateColumnConfig($columnConfig, 'testresults');
```

Note. The example can be found here: *$webSiteRoot\nliteme\scripts\modifyPrefs.php*

8. Use of extracolumns

The testresults table contains additional columns which may be used for project specific extension without a need to modify the existing Nliteme code. These columns have predefined names and data types which are:

extracolumn_0	smallint	custom use, small integer value
extracolumn_1	smallint	custom use, small integer value
extracolumn_2	mediumint	custom use, medium integer value
extracolumn_3	decimal	custom use, decimal value

Note. The extracolumn_2 has been reserved for mapped defect number and shall not be used for other purposes

To enable and configure the extracolumns the column settings have to be modified in the testresults table preferences. Please refer to section Modifying column preferences for a description on how to do that.

The columns can be used to store values “as is” or just an index to a predefined values, which will be stored in column preferences. If the column shall use predefined values the parameter 'predefined' has to be set to ‘yes’ in column settings.

Below is a code snippet on how to enable extracolumn_0, make it visible in search forms and details view and configure it to use predefined values:

```
$columnConfig = Config::getInstance()->getColumnConfig('testresults');

$columnConfig->getDbColumnByRealName('extracolumn_3')->setParam('showable','yes');
$columnConfig->getDbColumnByRealName('extracolumn_3')->setParam('iteratable','yes');
$columnConfig->getDbColumnByRealName('extracolumn_3')->setParam('enabled','yes');
$columnConfig->getDbColumnByRealName('extracolumn_0')->setParam('predefined','yes');

Config::getInstance()->updateColumnConfig($columnConfig, 'testresults');
```

Please note that the extracolumn_0 is by default configured with the above settings as it is used by PRG LTE project to store the UE types information.

9. Changing text printed on the web page

The communications messages, texts, table descriptions etc. are located in *$webSiteRoot \nliteme\etc\text.ini*. The file has a standard php ini format (i.e. left value = right value). The left value is referenced in the templates and shall not be modified. The user may change the messages by editing the right argument.

For example to change the name of the extracolumn_3 to “My Column 3” on the web page the following entry has to be modified:

```
EXTRACOLUMN_3 = 'My Column 3'
```

10. Other settings

10.1 Enable automatic hyperlink prefix to external server hosting Feature definitions

In case the Feature name (`fname` parameter - see sec. 5.1.4) is an index to the feature definition on the separate/external server, it is possible to predefine the hyperlink prefix which would be added automatically to hyperlink (that redirects to feature definition outside of nliteme domain) in case the user does not provide it explicitly (via `hlink` parameter - see sec. 5.1.4). The setting is stored under *'feature_otherserver_hlink'* key in the preferences table (by default disabled) and can be modified by executing the following code (example available also in `$webSiteRoot/nliteme/scripts/initFeatureExtServerPrefs.php`):

```
Config::getInstance()->setPreference('feature_otherserver_hlink',
                                     'https://<external_feature_server_url_prefix_up_to_id=>');
```

10.2 Enable automatic hyperlink prefix to external server hosting defects

It is assumed the details of the defects associated to the failing test results are hosted externally (using dedicated defect tracking tools) To enable linking between the defect number and the corresponding defect definition in the defect tracking tool the user can set the hyperlink prefix that would prepend the defect number in the generated hyperlink (whenever the user hovers on the defect number in the web views). The setting is stored under '*defect_otherserver_hlink'* key in the preferences table (by default disabled) and can be modified by executing the following code (example available also in `$webSiteRoot/nliteme/scripts/initDefectExtServerPrefs.php`):

```
Config::getInstance()->setPreference('defect_otherserver_hlink',

'https://<external_defect_tracking_server_url_prefix_up_to_id=>');
```