

TestLink Developer's Guide

Martin Havlat, Andreas Morsing, Francisco Mancardi

Copyright © 2005 TestLink Development Team

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. The license is available in ["GNU Free Documentation License" homepage](#).

1 General

1.1 Purpose

This document defines standards, practices and procedures for development, testing and publishing.

Refer to *TestLink Architecture Guide* about: architecture, interfaces, relations and general requirements.

2 Coding conventions

2.1 Introduction

This chapter describes the coding conventions which should be confirmed by each developer of TestLink.

2.2 Coding rules

There are some rules which all developers should follow

- Write your **code for humans** and not for computers!
- Don't re-invent the wheel! Take a look at already **existing code first!**
- Initialize each variable, before using them!
- If you are not sure about the existence of indices within associative array, use `isset` to test.
- Avoid scripts with huge size, **use functions** instead (and put them into function only files)
- Avoid functions of huge size, use small to medium sized functions only
- Avoid magic numbers
- Avoid unnecessary comments!
- Don't use fixed paths for accessing files
- Limit your access to `$_GET`, `$_POST`, `$_FILES` and similiar things, to one place in your script (preferable at the beginning)!

- Don't access `$_GET`, `$_POST`, `$_FILES` directly, use wrappers instead!
- Document code changes! That means, at least **document your code change** in the change log. You may also drop a short mail to other developers (maybe the initial author), so the documentation can be also be updated (if needed)
- Use TAB with a width of 4 spaces!
- Set `error_reporting` to **E_ALL** and set `TL_LOG_LEVEL_DEFAULT` to **EXTENDED** to get the maximum error messages while developing!
- Use the shortest possible meaningful names for variables and functions!
- Don't bloat the code with debug output, unnecessary comments, unnecessary blank lines and so on
- Don't ignore bad code! If it's safe to refactor it, just refactor!
- Don't duplicate code!

2.2.1 Code formatting

Always use braces and indent your code in the right way.

Choose one of these styles and be consistent:

```
if ($showFeature == 'editTc') {
    // show edit test cases
    $treeframePath = 'lib/testcases/listTestCases.php?feature=tcEdit';
    $workframePath = 'gui/instructions/tcEdit.html';
}

OR

if ($showFeature == 'editTc')
{
    // show edit test cases
    $treeframePath = 'lib/testcases/listTestCases.php?feature=tcEdit';
    $workframePath = 'gui/instructions/tcEdit.html';
}
```

2.2.2 Functions

for the following rules we use the example below.

```
/**
 * Function verifies if login exists
 * @param string login
 * @return integer number of founded records; i.e. 1 if account exists
 */
function existLogin($login)
{
    $sql = "SELECT * FROM user, rights WHERE user.rightsid = rights.id AND login='" .
        mysql_escape_string($login) . "'";

    $result = mysql_query($sql);
    $userExists = 0;
    if ($result)
    {
        if ($row = mysql_fetch_row($result))
            $userExists = $row;
    }
}
```

```
}  
  
return $userExists;  
}
```

2.2.2.1 Use unambiguous return type

In every situation a function must return the same variable type. As seen in the example the function returns a number (0) or an array, which are two different types, so return null instead of 0

2.2.2.2 Function header contents must be right

The information contained in the header must be coherent with the code. As you can see if the account exists the returned value is not 1, instead an array with the account data is return.

This rule applies also when you change an existing function. So don't forget to update also the function header (if present and needed).

2.2.3 Database access

for the following rules we use the example below.

2.2.3.1 SQL keyword have to be uppercase!

To ease to reading of SQL-Statements write all SQL keyword in uppercase letters

2.2.3.2 Use mysql_fetch_array whenever it's possible

To avoid index errors and simplify introducing new columns use mysql_fetch_array whenever possible.

2.3 File naming conventions

This section describes the file naming conventions used in TestLink.

- Use *only alphanumeric lowercase characters* for any filenames!
- The name of a Smarty Template called/launched from a PHP page called xyz.php must be xyz.tpl, i.e. the filenames without considering the extension must be *identical*.
- Php include files must be named xxx.inc.php
- Smarty templates must be named inc_XXX.tpl

2.4 Comments

To allow automatic creating of code documentation the phpDocumentor syntax will be applied to all code comments

Only BlockComment

```
/* ... */
```

and Line Comments

```
//
```

are allowed (# is not allowed) All comments have to be in plain english.

Code should not be clustered with comments, but instead the code should be written in a readable way, so its meaning is clear and therefore it doesn't need to be commented. Only comment things where it is absolutely needed and avoid useless comments like those bad examples:

```
//Initializing variables  
$pass = 0;
```

or

```
// execute SQL request  
$sqlTC = "select id,title,summary,steps,exresult,version,keywords,"
```

or

```
$result = mysql_query($sql); //Execute query
```

or

```
//I had to write this code so that the loop before would  
work.. I'm sure there is a better way to do it but hell if I know how to  
figure it out..  
if(count($testCaseArray) > 0){
```

These comments are quite unnecessary and useless (as they give no additional clues for the reader) and should be removed, before they bloat the code.

Example for a good comment:

```
//Chop the trailing comma off of the end of the keywords field  
$myrowTC[6] = substr($myrowTC[6], 0, -1);
```

Here it isn't clear in the first view that there is always a trailing comma at the end of the keyword list, so it's helpful to comment this intrinsic fact.

2.4.1 Standard file Header

Each file must contain the standard file header as seen in the example below below:

```
/**  
 * TestLink Open Source Project - http://testlink.sourceforge.net/  
 * This script is distributed under the GNU General Public License 2 or later.  
 *  
 * Filename $RCSfile: codingstandard.xml,v $  
 *  
 * @version $Revision: 1.4 $  
 * @modified $Date: 2005/11/15 19:28:24 $ by $Author: schlundus $  
 *  
 * @author <LastName, FirstName>  
 *  
 *  
 **/
```

2.4.2 Standard function header

Each function should be prefixed with a standard function header as seen in the example below

```
/**
 * generates data for tree menu of Test Case Suite (in Test Plan)
 *
 * @author <LastName, FirstName>
 *
 * @param string $linkto path for generated URL
 * @param integer $hidetc [0: show TCs, 1: disable TCs ]
 * @param string $getArguments additional $_GET arguments
 *
 * @return string input string for layersmenu
 *
 * @todo
 */
```

This header consists of:

- a comment: What does this function
- @author tags describing the author of the function
- @param tags for each parameter the function takes
- @return tag describing the return value [if any]
- @todo tag [if there are todos} describing the todo for this function

2.4.3 Code attributes

Any Code (this includes HTML, JS and PHP) may be attributed with

```
/// <tagname>
```

-tags Line Comments with

```
///
```

have a special meaning, they can be used to tag the code and ease the automatic generating of automatic reports like changelog, todo list.... At the moment there are currently four possible tags:

```
<change>
```

marking a change in the code

```
<fix>
```

a change in the code which is a bugfix

```
<enhancement>
```

a change (or an extension) of the code which is an improvement or an addition feature

```
<todo>
```

a reminder for something which has to be done with the code

Each tag can have up to four attributes (if applicable) namely:

```
version
```

this describes the version to which the action was applied

```
id
```

this refers to a bug id on sourceforge

```
author
```

the initials of the author

```
date
```

date of the action (should be given in YYYY-mm-dd)

Example:

```
/// <fix version="1.5.1" date="2005-02-15" author="scs" id="5678543">corrected the  
uninitialized use of the variable $key</fix>
```

Any code change has to be at least commented by one code-tag, if the change applies to one line only the code-tag has to be applied to the line before the change. If the change modifies many different lines of a function body, the code tag has to be included in the function comment. If the change modifies many different lines within the whole file, the code tag has to be included in the file header.

2.5 CVS

This section presents some short rules regarding to CVS handling. A good rule is to have two TestLink-source directories while developing. One which holds the current CVS contents and the other one in your web servers document root for developing. So its easy to revert any developing mistakes. Applying your changes to CVS is also easy, as the only thing you have to do is diffing your developing directory with the CVS directory and merge your changes and commit into CVS.

As you are not the only developing (-:, some rules should be applied while plaing with CVS.

- Update your CVS directory before applying your changes to it! As other developers may also change things on parts you are working on, updating avoids conflicts.
- Don't commit anything you are working on, which isn't complete! As CVS is accessible by all most people aren't happy if they checked out something, which isn't working at all. So commit only when your changes (e.G. new feature) are working and complete. This doesn't meant that we expect only bugfree commits (-:. So don't commit your nearly 50%-ready changes, go in a two-weeks holiday and after that commit the rest.
- If you intend to work on a bigger feature it makes sense to communicate this to the other developers, so the amount of interferences are minimized.
- If you intend to work on a new feature which wasn't assigned to you, please contact the other developers, nothing is more frustrating and resource wasting when two different people are implementing same thing in parallel.

GUI - Smarty templates

2.6 Introduction

TestLink has independent GUI rendering by Smarty templates. See more [Smarty homepage](#).

All Smarty Templates are stored in the directory <tl_root>/gui/templates/. Also CSS is described in this chapter. CSS 1.0 is preferred standard. Parameters from standard CSS 2.0 and later must be verified in both IE and Firefox before using.

2.7 Conventions

2.7.1 Filename

The file name of template follows the name of a related php script. There are also general pieces of template which are used in more page templates. Such template has **inc_** prefix (e.g. `inc_jsTree.tpl`).

2.7.2 Header

The simple header is used in each template. See the next example. The last line of the example informs about changes. No needs to describe changes in body of template more.

```
{* TestLink Open Source Project - http://testlink.sourceforge.net/ *} {* $Id:
execNavigator.tpl,v 1.3 2005/08/27 20:53:30 schlundus Exp $ *} {* Purpose:
smarty template - show test set tree *} {* 20050828 - scs - added searching
for tcID *}
```

2.8 Common practices

This section describe common using of some parts of templates.

2.8.1 HTML page header

Html header is defined in `inc_header.tpl`. This template is included to each page template immediately after file description. E.g. `{include file="inc_head.tpl" jsTree="yes"}`.

The header template can have the next parameters:

- `$openHead="yes"` - The `head.tpl` includes a closing of html header as default. This parameter cause that header remains open for another definition e.g. javascripts call. `</head>` must be defined in page then.
- `$jsValidate="yes"` - includes `validate.js` javascript to the html header.
- `$jsTree="yes"` - includes `inc_jsTree.tpl` (javascript to the html header).

All files (except generated documents) are expected to fulfill the XHTML 1.0 Transitional standard. A browser is instructed to no cache the pages. Functions in `testlink_library.js` are everytime loaded.

Included modifiable parameters (via `$smarty->assign()`):

- `$pageCharset` - UTF-8 is default;

- \$css - <tl_root>/gui/css/testlink.css is default for pages and tl_doc_basic.css for generated documents;
- \$SP_html_help_file

2.8.2 Tables

There are four basic types of style:

- class="common" - used to view data
- class="simple" - used to forms (input) data
- class="invisible" - used where div/span formatting is not sufficient
- class="smallGrey" - used to define options, filters in left (navigation) pane

2.8.3 Help reference

Use the next example of code to show help icon in title:

```
<h1> 
{lang_get s='your_string_identifier'}: { $arrReq.title|escape} </h1>
```

Use the next example of code to call help by click a string:

```
<span class="help"
onclick="javascript:open_popup('../help/glosary.html#testspec');">{lang_get
s='your_string_identifier'}</span>
```

2.8.4 Buttons

Each button should be included in <div class="groupBtn">. Navigation buttons should be in a top part of a page. Input data buttons should be in a bottom part of a page. Both top and bottom button should be used if page is long (e.g. the test execution page with tens of test cases).

2.8.5 Action results

The template inc_update.tpl is useful generalized template. See for more in.

Styles div.error, div.info should be also used for information about some result.

2.8.6 Combobox Menu

Use smarty component html_options. See more into Smarty documentation. E.g.

```
<select name="optReq"> {html_options options=$option_yes_no
selected=$reqs_default} </select>
```

Debugging

2.9 Logging

See `config.inc.php` to set logging level and target file.

Now any log messages from the levels ERROR or INFO will be recorded. DEBUG messages will be ignored. We can have as many log entries as we like. They take the form: `tLog("testing level ERROR", 'ERROR');` `tLog("testing level INFO", 'INFO');` `tLog("testing level DEBUG");` This will add the following entries to the log: `[05/Jan/27 13:05:56][INFO][guest] - Login ok. (Timing: 0.000763) [05/Jan/27 13:06:03][DEBUG][havlatm] - User id = 10, Rights = admin`

`tLogSqlError($sql)` - function specified for unsuccessful database query.

Timing is available for performance optimization. Use the next functions: `tlTimingStart ($name)`, `tlTimingStop ($name)` and `tlTimingCurrent ($name)`. The last one returns the measured time.

2.10 Smarty variables parsing

Smarty allows to view all assigned variable via special window.

Set line 101: `var $debugging = true; in`
`<tl_root>/third_party/smarty/Smarty.class.php.`

How-to

2.11 How to write interface for Bug Tracking systems

This section shortly describes how to write a new interface for bug tracking systems. A bugtracking interface mainly consists of two files: First a config file located in the `cfg`-directory and a interface definition file located in the `lib/bugtracking`-folder.

2.11.1 Name the interface

First choose a name for your interface and make this name available in `config.inc.php` (search for `TL_INTERFACE_BUGS`). Just use some simple word like 'BUGZILLA' or 'MANTIS'.

2.11.2 Create the config file

As already mentioned the config file is located in the `cfg`-folder. The file should contain all the configuration options for the interface like `dbhost` and such stuff. For the already used options just look at the `bugzilla.cfg.php` file.

For simplicity just name the config file similar to the name you've chosen in 1.1 like `bugzilla.cfg.php` for 'BUGZILLA'

2.11.3 Create the interface file

The interface file is located in lib/bugtracking and should be named similar to the name chosen in 1.1 like int_bugzilla.php for 'BUGZILLA'.

Create a new interface which extends the base interface in the file int_bugtracking.php. It is important to make the interface name available via the define "BUG_INTERFACE_CLASSNAME". For an example of a bugtracking interface look at int_bugzilla.php file.

2.11.3.1 Description of the base class

The base class already contains code for some basic stuff like connecting. So mostly there is no need to overload these functions.

The only function which is directly called by TestLink is the buildViewBugLink-Function which returns a link to the bugtracking system. The base already has a default implementation of it, so maybe you don't have to change it. The default implementation calls to helpers name buildViewBugURL and getBUGStatusString. The first one builds the URL for accessing the bugtracking system and the second one returns a user friendly description of the bug like BugID and such stuff. So these two ones are the first to change. Another helper which may come in handy is getBugStatus which should be overloaded to get the bug status from the bugtracking-db. This status can be used in the two other helpers.

For an example how to implement these functions just look at the mantis and bugzilla interfaces. For comments and the purposes of the different base interface functions just look at the int_bugtracking.php file. The code is well documented and straight forward.

2.11.4 Setup your bugtracking interface

After you named the interface and created the two files it is necessary to add these filenames to the int_bugtracking.php file. This can be done by adding the interface to the two associative arrays \$configFiles and \$interfaceFiles and the beginning of the file.

2.11.5 Testing the interface

If the connection to the bugtracking system is successfully done. You should test the interface. So enter a dummy bug in your bugtracking system and execute a testcase from a dummy-testplan within TestLink. You should now see an input field for entering bugs. So mark the testcase as failed and enter the id of the created dummy bug. Now switch to the reports and execute the report of failed testcases. The report should not show a link to the bug. Clicking the link should open a new window which shows the bug in your bugtracking.

2.11.6 Some words

As the bugtracking base interface is quite new, it may be that it will not always be sufficient. So feel free to contact us. I would also be very appreciated if you send your created and working interface(s) to us, so we can integrate them to our TestLink distribution.

How to write documentation

Docbook is chosen type of documentation for testLink. All documentation sources are available in CVS in module *documents* (no branch). Exported html is stored in module *docs* (together with homepage).

All documentation is stored as OpenOffice.org 2.0 format. The documents are exported for releases to html and pdf format.

Old way (valid for TL 1.5):

First you need a **XML Parser** (Transformer). E.g. *Saxon* (free open source), it can be called via command line, so you can automate the process via batch file or maybe a console php script (which adds more power).

Second you need the docbook toolchain (can be easily downloaded in the internet at <http://docbook.sourceforge.net/>). A good editor for docbook related stuff is **xmimind** they offer a free standard edition for download which is capable of editing xml files in a tree or WYSIWYG and can also directly render to HTML which is good for testing the files. But sometimes it is faster to write directly in XML so a simple XML Editor (or notepad) may be useful.

Appendixes

Revision history:

#	Description	Date	Authors
0.1 - 0.5	Coding standard create	2005/3/13	SCS, FM, MHT
0.6	Initial compilation from the all related documentation	2005/07/15	MHT
0.7	Conversion to OO2, layout correction, doc section updated	2005/12/11	MHT