Developers Guide

Reference for developers and community members

MantisBT Development Team <mantisbt-dev@lists.sourceforge.net>

Developers Guide: Reference for developers and community members

by MantisBT Development Team

Abstract

This book is targeted at MantisBT developers, contributors and plugin authors. It documents the development process and provides reference information regarding the MantisBT core, including the database schema as well as the plugin system including an events reference.

Table of Contents

List of Figures

Chapter 1. Contributing to MantisBT

MantisBT source code is managed with Git [http://git-scm.com/]. If you are new to this version control system, you can find some good resources for learning and installing it in the section called “Git References”.

Initial Setup

There are a few steps the MantisBT team requires of contributors and developers when accepting code submissions. The user needs to configure Git to know their full name (not a screen name) and an email address they can be contacted at (not a throwaway address).

To set up your name and email address with Git, run the following commands, substituting your own real name and email address:

```
git config --global user.name "John Smith"
git config --global user.email "jsmith@mantisbt.org"
```

Optionally, you may want to configure Git to use terminal colors when displaying file diffs and other information, and also alias certain Git actions to shorter phrases to reduce typing:

```
git config --global color.diff "auto"
git config --global color.status "auto"
git config --global color.branch "auto"

git config --global alias.st "status"
git config --global alias.di "diff"
git config --global alias.co "checkout"
git config --global alias.ci "commit"
```

Cloning the Repository

The official MantisBT source code repository is hosted at GitHub [https://github.com/mantisbt/mantisbt]. This document assumes that you have already signed up for and setup a GitHub account.

Determining the Clone URL

Which URL you will use to clone the repository before you start developing depends on your situation.

MantisBT Core Team Developers

MantisBT developers have *push* access to the official repository.

Benefitting from this access requires a special URL that uses your SSH key to handle access permissions: git@github.com:mantisbt/mantisbt.git. Alternatively, an HTTPS link can be used as well, in which case you will have to provide your GitHub User ID and password when Git requests it: https://github.com/mantisbt/mantisbt.git.

Note

Pushes *will fail* if you do not have access or your public SSH key is not set up correctly in your GitHub profile.

Contributors

For other people, the MantisBT repository and the related clone URLs git://github.com/mantisbt/mantisbt.git (SSH) or https://github.com/mantisbt/mantisbt.git (HTTPS) will always be read-only.

It is therefore strongly advised to *create your own fork [https://github.com/mantisbt/mantisbt/fork]* of MantisBT where you will be able to push your changes, and then use the fork's URL instead to clone, which will look like this: git@github.com:MyGithubId/mantisbt.git or https://github.com/MyGithubId/mantisbt.git

Initializing the Clone

To clone the repository, execute the following command from your target workspace:

```
git clone YourCloneURL
```

After performing the cloning operation, you should end up with a new directory in your workspace, `mantisbt/`, containing the MantisBT repository with a `remote` named *origin* pointing to your Clone URL.

MantisBT uses Composer [https://getcomposer.org] to pull libraries and components from Packagist [https://packagist.org] and Github [https://github.com]. Install Composer [https://getcomposer.org/download/] and run the following command:

```
composer install
```

Warning

Failure to execute the submodule initialization commands will result in critical components being missing from `/vendor` folder, which will then cause errors when running MantisBT.

Adding remotes

If you are planning to use your own fork to push and maintain your changes, then we recommend setting up an *upstream* `remote` for MantisBT's official repository, which will make it easier to keep your repository up-to-date.

```
git remote add --tags upstream git://github.com/mantisbt/mantisbt.git
```

Checking out branches

By default, the new clone will only track code from the primary remote branch, `master`, which is the latest development version of MantisBT. If you are planning to work with stable release or other development branches, you will need to set up local tracking branches in your repository.

The following command will set up a tracking branch for the current stable branch, `master-1.3.x`.

```
git checkout -b master-1.3.x origin/master-1.3.x
```

Note

With the introduction of submodules for some of the third-party libraries, you may encounter issues when switching to an older branch which still has code from those libraries in a subdirectory of `/library` rather than a submodule:

```
$ git checkout old_branch
error: The following untracked working tree files would be overwritten by check
 (list of files)
Aborting
```

To resolve this, you first have to get rid of the submodules directories before you can checkout the branch. The command below will move all submodules to `/tmp`:

```
sed -rn "s/^.*path\s*=\s*(.*)$/\1/p" .gitmodules |xargs -I{} mv -v {} /tmp
git checkout old_branch
```

Alernatively, if you don't care about keeping the changes in the submodules directories, you can simply execute

```
git checkout -f old_branch
git clean -df
```

When switching back from the older branch, the submodules directories will be empty. At that point you can either

- Update the submodules to reclone them

  ```
  git submodule update
  ```

- Restore the directories previously moved to `/tmp` back into the empty directories, e.g.

  ```
  sed -rn "s/^.*path\s*=\s*(.*)$/\1/p" .gitmodules |xargs -n 1 basename |xargs
  ```

For further reference: Pro Git book [http://git-scm.com/book/en/Git-Tools-Submodules#Issues-with-Submodules]

Maintaining Tracking Branches

In order to keep your local repository up-to-date with the official one, there are a few simple commands needed for any tracking branches that you may have, including `master` and `master-1.3.x`.

First, you'll need to get the latest information from the remote repository:

```
git fetch origin
```

Note

If you cloned from your personal GitHub fork instead of the official MantisBT repository as explained in the section called “Adding remotes”, then you should instead execute:

```
git fetch upstream
```

Then for each tracking branch you have, enter the following commands:

```
git checkout BranchName
git rebase
```

Alternatively, you may combine the fetch and rebase operations described above into a single pull command (for each remote tracking branch):

```
git checkout master
git pull --rebase
```

Preparing Feature Branches

For each local or shared feature branch that you are working on, you will need to keep it up to date with the appropriate master branch. There are multiple methods for doing this, each better suited to a different type of feature branch. *Both methods assume that you have already performed the previous step, to update your local tracking branches (see the section called “Maintaining Tracking Branches”).*

Private Branches

If the topic branch in question is a local, private branch, that you are not sharing with other developers, the simplest and easiest method to stay up to date with `master` is to use the **rebase** command. This will append all of your feature branch commits into a linear history after the last commit on the `master` branch.

```
git rebase master feature
```

Note

Rebasing changes the ID for each commit in your feature branch, which will cause trouble for anyone sharing and/or following your branch.

The resulting conflict can be fixed by rebasing their copy of your branch onto your branch:

```
git checkout feature
git fetch remote/feature
git rebase remote/feature
```

Public Branches

For any publicly-shared branches, where other users may be watching your feature branches, or cloning them locally for development work, you'll need to take a different approach to keeping it up to date with `master`.

To bring public branch up to date, you'll need to **merge** the current `master` branch, which will create a special "merge commit" in the branch history, causing a logical "split" in commit history where your branch started and joining at the merge. These merge commits are generally disliked, because they can crowd commit history, and because the history is no longer linear. They will be dealt with during the submission process (see the section called “Running PHPUnit tests”).

```
git checkout feature
git merge master
```

At this point, you can push the branch to your public repository, and anyone following the branch can then pull the changes directly into their local branch, either with another merge, or with a rebase, as necessitated by the public or private status of their own changes.

Running PHPUnit tests

MantisBT has a suite of PHPUnit tests found in the `tests` directory. You are encouraged to add your own tests for the patches you are submitting, but please remember that your changes must not break existing tests.

In order to run the tests, you will need to have the PHP Soap extension, PHPUnit 3.4 or newer [http://www.phpunit.de] and Phing 2.4 or newer [http://phing.info] installed. The tests are configured using a `bootstrap.php` file. The `boostrap.php.sample` file contains the settings you will need to adjust to run all the tests.

Running the unit tests is done from root directory using the following command:

```
phing test
```

Running the SOAP tests

MantisBT ships with a suite of SOAP tests which require an initial set up to be executed. The required steps are:

- Install MantisBT locally and configure a project and a category.
- Adjust the `bootstrap.php` file to point to your local installation.
- Customize the `config_inc.php` to enable all the features tested using the SOAP tests. The simplest way to do that is to run all the tests once and adjust it based on the skipped tests.

Submitting Changes

This section describes what you should do to submit a set of changes to MantisBT, allowing the project developers to review and test, your code, and ultimately commit it to the MantisBT repository.

The actual submission can be done using several methods, described later in this section:

- *Recommended:* Github Pull Requests (see the section called “Submission Via Github Pull Requests”)
- Other public Git repository Pull Requests (see the section called “Submission Via Public Repository”)
- Git Formatted patches (see the section called “Submission Via Formatted Patches”)

Before you submit

Before submitting your contribution, you should make sure that

1. Your code follows the MantisBT coding guidelines [http://www.mantisbt.org/wiki/doku.php/mantisbt:coding_guidelines]

2. You have tested your changes locally (see the section called “Running PHPUnit tests”)

3. Your local branch has been rebased on top of the current Master branch, as described in the section called “Private Branches”.

Submission Via Github Pull Requests

Since the official MantisBT repository [https://github.com/mantisbt/mantisbt] is hosted there, using GitHub [http://github.com] is the recommended (and easiest) way to submit your contributions.

With this method, you can keep your changesets up-to-date with the official development repository, and likewise let anyone stay up to date with your repository, without needing to constantly upload and download new formatted patches whenever you change anything.

The process below describes a simple workflow that can help you make your submission if you are not familiar with Git; note that it is by no means the only way to do this.

Note

We'll assume that you have already forked MantisBT [https://github.com/mantisbt/mantisbt/fork], cloned it locally as described in the section called “Cloning the Repository” (remote *upstream* being the official MantisBT repository and *origin* your personal fork), and created a new feature branch (see the section called “Preparing Feature Branches”) for your contribution, which we'll call *MyBranch*.

1. Make sure that the *MyBranch* feature branch is up-to-date with the master branch by rebasing it, resolving any conflicts if necessary.

```
git fetch upstream
git rebase upstream/master MyBranch
```

2. Push the branch to your Github fork

```
git push origin MyBranch
```

3. Go to your Fork on Github (https://github.com/MyGithubId/mantisbt)

4. Initiate a Pull Request [https://github.com/MyGithubId/mantisbt/compare/MyBranch] from your feature branch, following the guidelines provided in Github Help [https://help.github.com/articles/using-pull-requests].

   Please make sure you provide a detailed description of the changes you are submitting, including the reason for it and if possible a reference (link) to an existing issue on our bugtracker [http://mantibt.org/bugs/]. The team will usually review your changes and provide feedback within 7 days (but your mileage may vary).

Submission Via Formatted Patches

Formatted patches are very similar to file diffs generated by other tools or source control systems, but contain far more information, including your name and email address, and for every commit in the set, the commit's timestamp, message, author, and more. They allow anyone to import the enclosed changesets directly into Git, where all of the commit information is preserved.

Assuming that you have an existing local that you've kept up to date with `master` as described in the section called “Preparing Feature Branches” currently checked out, generating a formatted patch set should be relatively straightforward, using an appropriate filename as the target of the patch set:

```
git format-patch --binary --stdout origin/master..HEAD > feature_branch.patch
```

Once you've generated the formatted patch file, you can easily attach it to a bug report, or even use the patch file as an email to send to the developer mailing list. Developers, or other users, can then import this patch set into their local repositories using the following command, again substituting the appropriate filename:

```
git am --signoff feature_branch.patch
```

Submission Via Public Repository

If you are not able or not willing to make use of a fork of the official GitHub [http://github.com] repository but have another publicly available one to host your changes, for example on a free hosting for public repository such as

- Bitbucket [https://bitbucket.org]
- Gitorious [http://gitorious.com]

you can still use it to submit a patch in a similar fashion to the Github method described above, although the process is slightly more complicated.

We'll assume you've already set up a publicly accessible repository at URL `git@githosting.com:contrib.git`, kept it up-to-date with MantisBT's official repository, and that you have pushed your feature branch `MyBranch` to it.

1. Generate the Pull Request

   This will list information about your changes and how to access them. The process will attempt to verify that you've pushed the correct data to the public repository, and will generate a summary of changes.

   ```
   git request-pull origin/master git@githosting.com:contrib.git MyBranch
   ```

2. Paste the output of the above command into a bug report or an email to the developer mailing list [mailto:mantisbt-dev@lists.sourceforge.net]

Once your pull request has been posted, developers and other users can add your public repository as a remote, and track your feature branch in their own working repository using the following commands, replacing the remote name and local branch name as appropriate:

```
git remote add feature  git@githosting.com:contrib.git
```

```
git checkout -b MyBranch feature/MyBranch
```

If the feature is approved for entry into MantisBT core, then the branch should first be rebased onto the latest HEAD so that Git can remove any unnecessary merge commits, and create a linear history. Once that's completed, the feature branch can be merged into `master`:

```
git rebase master feature
git checkout master
git merge --no-ff feature
```

Chapter 2. Database Schema Management

The MantisBT schema

The MantisBT database schema (excluding plugins) is described in the Entity-Relationship diagram (ERD) below. There is also a PDF version available for download [http://mantisbt.org/docs/erd/].

Figure 2.1. MantisBT Entity-Relationship Diagram

Schema Definition

TODO: Discuss the ADODB datadict formats and the format MantisBT expects for schema definitions.

Installation / Upgrade Process

TODO: Discuss how MantisBT handles a database installation / upgrade, including the use of the config system and schema definitions.

Chapter 3. Event System

General Concepts

The event system in MantisBT uses the concept of signals and hooked events to drive dynamic actions. Functions, or plugin methods, can be hooked during runtime to various defined events, which can be signalled at any point to initiate execution of hooked functions.

Events are defined at runtime by name and event type (covered in the next section). Depending on the event type, signal parameters and return values from hooked functions will be handled in different ways to make certain types of common communication simplified.

API Usage

This is a general overview of the event API. For more detailed analysis, you may reference the file `core/event_api.php` in the codebase.

Declaring Events

When declaring events, the only information needed is the event name and event type. Events can be declared alone using the form:

```
event_declare( $name, $type=EVENT_TYPE_DEFAULT );
```

or they can be declared in groups using key/value pairs of name => type relations, stored in a single array, such as:

```
$events = array(
 $name_1 => $type_1,
 $name_2 => $type_2,
 ...
 );

event_declare_many( $events );
```

Hooking Events

Hooking events requires knowing the name of an already-declared event, and the name of the callback function (and possibly associated plugin) that will be hooked to the event. If hooking only a function, it must be declared in the global namespace.

```
event_hook( $event_name, $callback, [$plugin] );
```

In order to hook many functions at once, using key/value pairs of name => callback relations, in a single array:

```
$events = array(
 $event_1 => $callback_1,
 $event_2 => $callback_2,
 ...
```

```
 );

event_hook( $events, [$plugin] );
```

Signalling Events

When signalling events, the event type of the target event must be kept in mind when handling event parameters and return values. The general format for signalling an event uses the following structure:

```
$value = event_signal( $event_name, [ array( $param, ... ), [ array( $static
```

Each type of event (and individual events themselves) will use different combinations of parameters and return values, so perusing Chapter 5, *Events Reference* is recommended for determining the unique needs of each event when signalling and hooking them.

Event Types

There are five standard event types currently defined in MantisBT. Each type is a generalization of a certain "class" of solution to the problems that the event system is designed to solve. Each type allows for simplifying a different set of communication needs between event signals and hooked callback functions.

Each type of event (and individual events themselves) will use different combinations of parameters and return values, so perusing Chapter 5, *Events Reference* is recommended for determining the unique needs of each event when signalling and hooking them.

EVENT_TYPE_EXECUTE

This is the simplest event type, meant for initiating basic hook execution without needing to communicate more than a set of immutable parameters to the event, and expecting no return of data.

These events only use the first parameter array, and return values from hooked functions are ignored. Example usage:

```
event_signal( $event_name, [ array( $param, ... ) ] );
```

EVENT_TYPE_OUTPUT

This event type allows for simple output and execution from hooked events. A single set of immutable parameters are sent to each callback, and the return value is inlined as output. This event is generally used for an event with a specific purpose of adding content or markup to the page.

These events only use the first parameter array, and return values from hooked functions are immediately sent to the output buffer via 'echo'. Another parameter `$format` can be used to model how the results are printed. This parameter can be either:

- null, or omitted: The returned values are printed without further processing
- <String>: A string to be used as separator for printed values
- <Array>: An array of (prefix, separator, postfix) to be used for the printed values

Example usage:

```
event_signal( $event_name, [ array( $param, ... ) ], [ $format ] );
```

EVENT_TYPE_CHAIN

This event type is designed to allow plugins to successively alter the parameters given to them, such that the end result returned to the caller is a mutated version of the original parameters. This is very useful for such things as output markup parsers.

The first set of parameters to the event are sent to the first hooked callback, which is then expected to alter the parameters and return the new values, which are then sent to the next callback to modify, and this continues for all callbacks. The return value from the last callback is then returned to the event signaller.

This type allows events to optionally make use of the second parameter set, which are sent to every callback in the series, but should not be returned by each callback. This allows the signalling function to send extra, immutable information to every callback in the chain. Example usage:

```
$value = event_signal( $event_name, $param, [ array( $static_param, ... ) ]
```

EVENT_TYPE_FIRST

The design of this event type allows for multiple hooked callbacks to 'compete' for the event signal, based on priority and execution order. The first callback that can satisfy the needs of the signal is the last callback executed for the event, and its return value is the only one sent to the event caller. This is very useful for topics like user authentication.

These events only use the first parameter array, and the first non-null return value from a hook function is returned to the caller. Subsequent callbacks are never executed. Example usage:

```
$value = event_signal( $event_name, [ array( $param, ... ) ] );
```

EVENT_TYPE_DEFAULT

This is the fallback event type, in which the return values from all hooked callbacks are stored in a special array structure. This allows the event caller to gather data separately from all events.

These events only use the first parameter array, and return values from hooked functions are returned in a multi-dimensional array keyed by plugin name and hooked function name. Example usage:

```
$values = event_signal( $event_name, [ array( $param, ... ) ] );
```

Chapter 4. Plugin System

General Concepts

The plugin system for MantisBT is designed as a lightweight extension to the standard MantisBT API, allowing for simple and flexible addition of new features and customization of core operations. It takes advantage of the new Event System (see Chapter 3, *Event System*) to offer developers rapid creation and testing of extensions, without the need to modify core files.

Plugins are defined as implementations, or subclasses, of the `MantisPlugin` class as defined in `core/classes/MantisPlugin.php`. Each plugin may define information about itself, as well as a list of conflicts and dependencies upon other plugins. There are many methods defined in the `MantisPlugin` class that may be used as convenient places to define extra behaviors, such as configuration options, event declarations, event hooks, errors, and database schemas. Outside a plugin's core class, there is a standard method of handling language strings, content pages, and files.

At page load, the core MantisBT API will find and process any conforming plugins. Plugins will be checked for minimal information, such as its name, version, and dependencies. Plugins that meet requirements will then be initialized. At this point, MantisBT will interact with the plugins when appropriate.

The plugin system includes a special set of API functions that provide convenience wrappers around the more useful MantisBT API calls, including configuration, language strings, and link generation. This API allows plugins to use core API's in "sandboxed" fashions to aid interoperation with other plugins, and simplification of common functionality.

Building a Plugin

This section will act as a walk through of how to build a plugin, from the bare basics all the way up to advanced topics. A general understanding of the concepts covered in the last section is assumed, as well as knowledge of how the event system works. Later topics in this section will require knowledge of database schemas and how they are used with MantisBT.

This walk through will be working towards building a single end result: the "Example" plugin as listed in the section called “Example Plugin Source Listing”. You may refer to the final source code along the way, although every part of it will be built up in steps throughout this section.

Plugin Structure

This section will introduce the general concepts of plugin structure, and how to get a barebones plugin working with MantisBT. Not much will be mentioned yet on the topic of adding functionality to plugins, just how to get the development process rolling.

The backbone of every plugin is what MantisBT calls the *basename*, a succinct, and most importantly, unique name that identifies the plugin. It may not contain any spacing or special characters beyond the ASCII upper- and lowercase alphabet, numerals, and underscore. This is used to identify the plugin everywhere except for what the end-user sees. For our "Example" plugin, the basename we will use should be obvious enough: `Example`.

Every plugin must be contained in a single directory, named to match the plugin's basename, as well as contain at least a single PHP file, also named to match the basename, as such:

Note that for plugins that require a database schema to operate, the basename is also used to build the table names, using the MantisBT table prefixes and suffix (please refer to the Admin Guide's *Configura-*

tion section for further information). If our Example plugin were to create a table named 'foo', assuming default values for prefixes and suffix in MantisBT configuration, the physical table name would be `mantis_plugin_Example_foo_table`.

```
Example/
 Example.php
```

Warning

Depending on case sensitivity of the underlying file system, these names must *exactly match* the plugin's base name, i.e. `example` will not work.

This top-level PHP file must then contain a concrete class deriving from the `MantisPlugin` class, which must be named in the form of `%Basename%Plugin`, which for our purpose becomes `ExamplePlugin`.

Because of how `MantisPlugin` declares the `register()` method as `abstract`, our plugin must implement that method before PHP will find it semantically valid. This method is meant for one simple purpose, and should never be used for any other task: setting the plugin's information properties including the plugin's name, description, version, and more. Please refer to the section called “Properties” below for details about available properties.

Once your plugin defines its class, implements the `register()` method, and sets at least the name and version properties, it is then considered a "complete" plugin, and can be loaded and installed within MantisBT's plugin manager. At this stage, our Example plugin, with all the possible plugin properties set at registration, looks like this:

Example/Example.php

```
<?php
class ExamplePlugin extends MantisPlugin {
    function register() {
        $this->name = 'Example';    # Proper name of plugin
        $this->description = '';    # Short description of the plugin
        $this->page = '';           # Default plugin page

        $this->version = '1.0';     # Plugin version string
        $this->requires = array(    # Plugin dependencies
            'MantisCore' => '2.0',  # Should always depend on an appropriate
                                    # version of MantisBT
        );

        $this->author = '';         # Author/team name
        $this->contact = '';        # Author/team e-mail address
        $this->url = '';            # Support webpage
    }
}
```

This alone will allow the Example plugin to be installed with MantisBT, and is the foundation of any plugin. More of the plugin development process will be continued in the next sections.

Properties

This section describes the properties that can be defined when registering the plugin.

name — Your plugin's full name. *Required value.*

description — A full description of your plugin.

page — The name of a plugin page for further information and administration of the plugin. This is used to create a link to the specified page on Mantis' manage plugin page.

version — Your plugin's version string. *Required value.* We recommend following the Semantic Versioning [http://semver.org/] specification, but you are free to use any versioning scheme that can be handled by PHP's version_compare() [http://php.net/manual/en/function.version-compare.php] function.

requires — An array of key/value pairs of basename/version plugin dependencies.

Note

The special, reserved basename `MantisCore` can be used to specify the minimum requirement for MantisBT core.

The version string can be defined as:

- *Minimum requirement*: the plugin specified by the given basename must be installed, and its version must be equal or higher than the indicated one.
- *Maximum requirement*: prefixing a version number with '<' will allow the plugin to specify the highest version (non-inclusive) up to which the required dependency is supported.

Note

If the plugin's minimum dependency for MantisCore is unspecified or lower than the current release (i.e. it does not specifically list the current core version as supported) and the plugin does not define a maximum dependency, a default one will be set to the next major release of MantisBT. (i.e. for 2.x.y we would add `'<2'`).

This effectively disables plugins which have not been specifically designed for a new major Mantis release, thus forcing authors to review their code, adapt it if necessary, and release a new version of the plugin with updated dependencies.

- *Both minimum and maximum*: the two version numbers must be separated by a comma.

Here are a few examples to illustrate the above explanations, assuming that the current Mantis release (*MantisCore* version) is 2.1:

- Old release without a maximum version specified

```
$this->requires = array( 'MantisCore' => '1.3.1' );
```

The plugin is compatible with MantisBT >= 1.3.1 and < 2.0.0 - note that the maximum version (`<2`) was added by the system.

- Current release without a maximum version specified

```
$this->requires = array( 'MantisCore' => '2.0' );
```

The plugin is compatible with MantisBT >= 2.0 and < 3.0 (the latter is implicit); code supporting older releases (e.g. 1.3) must be maintained separately (i.e. in a different branch).

- Only specify a maximum version

```
$this->requires = array( 'MantisCore' => '< 3.1' );
```

The plugin is compatible up to MantisBT 3.1 (not inclusive).

- Old release with a maximum version

```
$this->requires = array( 'MantisCore' => '1.3, < 4.0' );
```

The plugin is compatible with MantisBT >= 1.3 and < 4.0.

uses	An array of key/value pairs of basename/version optional (soft) plugin dependencies. See `requires` above for details on how to specify versions.
author	Your name, or an array of names.
contact	An email address where you can be contacted.
url	A web address for your plugin.

Pages and Files

The plugin API provides a standard hierarchy and process for adding new pages and files to your plugin. For strict definitions, pages are PHP files that will be executed within the MantisBT core system, while files are defined as a separate set of raw data that will be passed to the client's browser exactly as it appears in the filesystem.

New pages for your plugin should be placed in your plugin's `pages/` directory, and should be named using only letters and numbers, and must have a ".php" file extension. To generate a URI to the new page in MantisBT, the API function `plugin_page()` should be used. Our Example plugin will create a page named `foo.php`, which can then be accessed via `plugin_page.php?page=Example/foo`, the same URI that `plugin_page()` would have generated:

`Example/pages/foo.php`

```
<?php
echo '<p>Here is a link to <a href="', plugin_page( 'foo' ), '">page foo</a>.</p>'
```

Adding non-PHP files, such as images or CSS stylesheets, follows a very similar pattern as pages. Files should be placed in the plugin's `files/` directory, and can only contain a single period in the name. The file's URI is generated with the `plugin_file()` function. For our Example plugin, we'll create a basic CSS stylesheet, and modify the previously shown page to include the stylesheet:

`Example/files/foo.css`

```
p.foo {
    color: red;
}
```

Example/pages/foo.php

```
<?php
echo '<p>Here is a link to <a href="', plugin_page( 'foo' ), '">page foo</a>.</p>'
echo '<link rel="stylesheet" type="text/css" href="', plugin_file( 'foo.css' ), '"
     '<p class="foo">This is red text.</p>';
```

Note that while `plugin_page()` expects only the page's name without the extension, `plugin_file()` requires the entire filename so that it can distinguish between `foo.css` and a potential file `foo.png`.

The plugin's filesystem structure at this point looks like this:

```
Example/
 Example.php
 pages/
  foo.php
 files/
  foo.css
```

Events

Plugins have an integrated method for both declaring and hooking events, without needing to directly call the event API functions. These take the form of class methods on your plugin.

To declare a new event, or a set of events, that your plugin will trigger, override the `events()` method of your plugin class, and return an associative array with event names as the key, and the event type as the value. Let's add an event "foo" to our Example plugin that does not expect a return value (an "execute" event type), and another event 'bar' that expects a single value that gets modified by each hooked function (a "chain" event type):

Example/Example.php

```
<?php
class ExamplePlugin extends MantisPlugin {
    ...

    function events() {
        return array(
            'EVENT_EXAMPLE_FOO' => EVENT_TYPE_EXECUTE,
            'EVENT_EXAMPLE_BAR' => EVENT_TYPE_CHAIN,
        );
    }
}
```

When the Example plugin is loaded, the event system in MantisBT will add these two events to its list of events, and will then allow other plugins or functions to hook them. Naming the events "EVENT_PLUGINNAME_EVENTNAME" is not necessary, but is considered best practice to avoid conflicts between plugins.

Hooking other events (or events from your own plugin) is almost identical to declaring them. Instead of passing an event type as the value, your plugin must pass the name of a class method on your plugin that will be called when the event is triggered. For our Example plugin, we'll create a `foo()` and `bar()` method on our plugin class, and hook them to the events we declared earlier.

Example/Example.php

```
<?php
class ExamplePlugin extends MantisPlugin {
    ...

    function hooks() {
        return array(
            'EVENT_EXAMPLE_FOO' => 'foo',
            'EVENT_EXAMPLE_BAR' => 'bar',
        );
    }

    function foo( $p_event ) {
        ...
    }

    function bar( $p_event, $p_chained_param ) {
        ...
        return $p_chained_param;
    }
}
```

Note that both hooked methods need to accept the *`$p_event`* parameter, as that contains the event name triggering the method (for cases where you may want a method hooked to multiple events). The `bar()` method also accepts and returns the chained parameter in order to match the expectations of the "bar" event.

Now that we have our plugin's events declared and hooked, let's modify our earlier page so that triggers the events, and add some real processing to the hooked methods:

Example/Example.php

```
<?php
class ExamplePlugin extends MantisPlugin {
    ...

    function foo( $p_event ) {
        echo 'In method foo(). ';
    }

    function bar( $p_event, $p_chained_param ) {
        return str_replace( 'foo', 'bar', $p_chained_param );
    }
}
```

Example/pages/foo.php

```
<?php
echo '<p>Here is a link to <a href="', plugin_page( 'foo' ), '">page foo</a>.</p>'
     '<link rel="stylesheet" type="text/css" href="', plugin_file( 'foo.css' ), '"
```

```
     '<p class="foo">';

event_signal( 'EVENT_EXAMPLE_FOO' );

$t_string = 'A sentence with the word "foo" in it.';
$t_new_string = event_signal( 'EVENT_EXAMPLE_BAR', array( $t_string ) );

echo $t_new_string, '</p>';
```

When the first event "foo" is signaled, the Example plugin's `foo()` method will execute and echo a string. After that, the second event "bar" is signaled, and the page passes a string parameter; the plugin's `bar()` gets the string and replaces any instance of "foo" with "bar", and returns the resulting string. If any other plugin had hooked the event, that plugin could have further modified the new string from the Example plugin, or vice versa, depending on the loading order of plugins. The page then echos the modified string that was returned from the event.

Configuration

Similar to events, plugins have a simplified method for declaring configuration options, as well as API functions for retrieving or setting those values at runtime.

Declaring a new configuration option is achieved just like declaring events. By overriding the `config()` method on your plugin class, your plugin can return an associative array of configuration options, with the option name as the key, and the default option as the array value. Our Example plugin will declare an option "foo_or_bar", with a default value of "foo":

Example/Example.php

```
<?php
class ExamplePlugin extends MantisPlugin {
    ...

    function config() {
        return array(
            'foo_or_bar' => 'foo',
        );
    }
}
```

Retrieving the current value of a plugin's configuration option is achieved by using the plugin API's `plugin_config_get()` function, and can be set to a modified value in the database using `plugin_config_set()`. With these functions, the config option is prefixed with the plugin's name, in attempt to automatically avoid conflicts in naming. Our Example plugin will demonstrate this by adding a secure form to the "config_page", and handling the form on a separate page "config_update" that will modify the value in the database, and redirect back to page "config_page", just like any other form and action page in MantisBT:

Example/pages/config_page.php

```
<form action="<?php echo plugin_page( 'config_update' ) ?>" method="post">
<?php echo form_security_field( 'plugin_Example_config_update' ) ?>

<label>Foo or Bar?<br/><input name="foo_or_bar" value="<?php echo string_attribute
<br/>
```

```
<label><input type="checkbox" name="reset"/> Reset</label>
<br/>
<input type="submit"/>

</form>
```

Example/pages/config_update.php

```
<?php
form_security_validate( 'plugin_Example_config_update' );

$f_foo_or_bar = gpc_get_string( 'foo_or_bar' );
$f_reset = gpc_get_bool( 'reset', false );

if( $f_reset ) {
    plugin_config_delete( 'foo_or_bar' );
} else {
    if( $f_foo_or_bar == 'foo' || $f_foo_or_bar == 'bar' ) {
        plugin_config_set( 'foo_or_bar', $f_foo_or_bar );
    }
}

form_security_purge( 'plugin_Example_config_update' );
print_successful_redirect( plugin_page( 'foo', true ) );
```

Note that the `form_security_*()` functions are part of the form API, and prevent CSRF attacks against forms that make changes to the system.

Language and Localization

MantisBT has a very advanced set of localization tools, which allow all parts of of the application to be localized to the user's preferred language. This feature has been extended for use by plugins as well, so that a plugin can be localized in much the same method as used for the core system. Localizing a plugin involves creating a language file for each localization available, and using a special API call to retrieve the appropriate string for the user's language.

All language files for plugins follow the same format used in the core of MantisBT, should be placed in the plugin's `lang/` directory, and named the same as the core language files. Strings specific to the plugin should be "namespaced" in a way that will minimize any risk of collision. Translating the plugin to other languages already supported by MantisBT is then as simple as creating a new strings file with the localized content; the MantisBT core will find and use the new language strings automatically.

We'll use the "configuration" pages from the previous examples, and dress them up with localized language strings, and add a few more flourishes to make the page act like a standard MantisBT page. First we need to create a language file for English, the default language of MantisBT and the default fallback language in the case that some strings have not yet been localized to the user's language:

Example/lang/strings_english.txt

```
<?php

$s_plugin_Example_configuration = "Configuration";
$s_plugin_Example_foo_or_bar = "Foo or Bar?";
$s_plugin_Example_reset = "Reset Value";
```

```
Example/pages/config_page.php
<?php

layout_page_header( plugin_lang_get( 'configuration' ) );
layout_page_begin();
$t_foo_or_bar = plugin_config_get( 'foo_or_bar' );

?>

<br/>

<form action="<?php echo plugin_page( 'config_update' ) ?>" method="post">
<?php echo form_security_field( 'plugin_Example_config_update' ) ?>
<table class="width60">

<tr>
    <td class="form-title" rowspan="2"><?php echo plugin_lang_get( 'configuration'
</tr>

<tr <?php echo helper_alternate_class() ?>>
    <td class="category"><php echo plugin_lang_get( 'foo_or_bar' ) ?></td>
    <td><input name="foo_or_bar" value="<?php echo string_attribute( $t_foo_or_bar
</tr>

<tr <?php echo helper_alternate_class() ?>>
    <td class="category"><php echo plugin_lang_get( 'reset' ) ?></td>
    <td><input type="checkbox" name="reset"/></td>
</tr>

<tr>
    <td class="center" rowspan="2"><input type="submit"/></td>
</tr>

</table>
</form>

<?php

layout_page_end();
```

The two calls to `layout_page_being()` and `layout_page_end()` trigger the standard MantisBT header and footer portions, respectively, which also displays things such as the menus and triggers other layout-related events. `layout_page_header()` pulls in the CSS classes for alternating row colors in the table. The rest of the HTML and CSS follows the "standard" MantisBT markup styles for content and layout.

Example Plugin Source Listing

The code in this section, for the Example plugin, is available for use, modification, and redistribution without any restrictions and without any warranty or implied warranties. You may use this code however you want.

```
Example/
```

```
Example.php
files/
    foo.css
lang/
    strings_english.txt
pages/
    config_page.php
    config_update.php
    foo.php
```

Example/Example.php

```
Example/Example.php
<?php
class ExamplePlugin extends MantisPlugin {
    function register() {
        $this->name = 'Example';    # Proper name of plugin
        $this->description = '';    # Short description of the plugin
        $this->page = '';           # Default plugin page

        $this->version = '1.0';     # Plugin version string
        $this->requires = array(    # Plugin dependencies
            'MantisCore' => '2.0',  # Should always depend on an appropriate
                                    # version of MantisBT
        );

        $this->author = '';         # Author/team name
        $this->contact = '';        # Author/team e-mail address
        $this->url = '';            # Support webpage
    }

    function events() {
        return array(
            'EVENT_EXAMPLE_FOO' => EVENT_TYPE_EXECUTE,
            'EVENT_EXAMPLE_BAR' => EVENT_TYPE_CHAIN,
        );
    }

    function hooks() {
        return array(
            'EVENT_EXAMPLE_FOO' => 'foo',
            'EVENT_EXAMPLE_BAR' => 'bar',
        );
    }

    function config() {
        return array(
            'foo_or_bar' => 'foo',
        );
    }

    function foo( $p_event ) {
        echo 'In method foo(). ';
```

```
    }

    function bar( $p_event, $p_chained_param ) {
        return str_replace( 'foo', 'bar', $p_chained_param );
    }

}
```

Example/files/foo.css

```
Example/files/foo.css
p.foo {
    color: red;
}
```

Example/lang/strings_english.txt

```
Example/lang/strings_english.txt
<?php

$s_plugin_Example_configuration = "Configuration";
$s_plugin_Example_foo_or_bar = "Foo or Bar?";
$s_plugin_Example_reset = "Reset Value";
```

Example/page/config_page.php

```
Example/pages/config_page.php
<?php

layout_page_header( plugin_lang_get( 'configuration' ) );
layout_page_begin();
$t_foo_or_bar = plugin_config_get( 'foo_or_bar' );

?>

<br/>

<form action="<?php echo plugin_page( 'config_update' ) ?>" method="post">
<?php echo form_security_field( 'plugin_Example_config_update' ) ?>
<table class="width60">

<tr>
    <td class="form-title" rowspan="2"><?php echo plugin_lang_get( 'configuration'
</tr>

<tr <?php echo helper_alternate_class() ?>>
    <td class="category"><php echo plugin_lang_get( 'foo_or_bar' ) ?></td>
    <td><input name="foo_or_bar" value="<?php echo string_attribute( $t_foo_or_bar
</tr>

<tr <?php echo helper_alternate_class() ?>>
    <td class="category"><php echo plugin_lang_get( 'reset' ) ?></td>
    <td><input type="checkbox" name="reset"/></td>
```

```
</tr>

<tr>
    <td class="center" rowspan="2"><input type="submit"/></td>
</tr>

</table>
</form>

<?php

layout_page_end();
```

Example/pages/config_update.php

```
Example/pages/config_update.php
<?php
form_security_validate( 'plugin_Example_config_update' );

$f_foo_or_bar = gpc_get_string( 'foo_or_bar' );
$f_reset = gpc_get_bool( 'reset', false );

if( $f_reset ) {
    plugin_config_delete( 'foo_or_bar' );
} else {
    if( $f_foo_or_bar == 'foo' || $f_foo_or_bar == 'bar' ) {
        plugin_config_set( 'foo_or_bar', $f_foo_or_bar );
    }
}

form_security_purge( 'plugin_Example_config_update' );
print_successful_redirect( plugin_page( 'foo', true ) );
```

Example/page/foo.php

```
Example/pages/foo.php
<?php
echo '<p>Here is a link to <a href="', plugin_page( 'foo' ), '">page foo</a>.</p>'
     '<link rel="stylesheet" type="text/css" href="', plugin_file( 'foo.css' ), '"
     '<p class="foo">';

event_signal( 'EVENT_EXAMPLE_FOO' );

$t_string = 'A sentence with the word "foo" in it.';
$t_new_string = event_signal( 'EVENT_EXAMPLE_BAR', array( $t_string ) );

echo $t_new_string, '</p>';
```

API Usage

This is a general overview of the plugin API. For more detailed analysis, you may reference the file `core/plugin_api.php` in the codebase.

Chapter 5. Events Reference

Introduction

In this chapter, an attempt will be made to list all events used (or planned for later use) in the MantisBT event system. Each listed event will include details for the event type, when the event is called, and the expected parameters and return values for event callbacks.

Here we show an example event definition. For each event, the event identifier will be listed along with the event types (see the section called “Event Types”) in parentheses. Below that should be a concise but thorough description of how the event is called and how to use it. Following that should be a list of event parameters (if any), as well as the expected return value (if any).

EVENT_EXAMPLE (Default)

This is an example event description.

Parameters

- <Type>: Description of parameter one
- <Type>: Description of parameter two

Return Value

- <Type>: Description of return value

System Events

These events are initiated by the plugin system itself to allow certain functionality to simplify plugin development.

EVENT_PLUGIN_INIT (Execute)

This event is triggered by the MantisBT plugin system after all registered and enabled plugins have been initialized (their `init()` functions have been called). This event should *always* be the first event triggered for any page load. No parameters are passed to hooked functions, and no return values are expected.

This event is the first point in page execution where all registered plugins are guaranteed to be enabled (assuming dependencies and such are met). At any point before this event, any or all plugins may not yet be loaded. Note that the core system has not yet completed the bootstrap process when this event is signalled.

Suggested uses for the event include:

- Checking for plugins that aren't require for normal usage.
- Interacting with other plugins outside the context of pages or events.

EVENT_CORE_HEADERS (Execute)

This event is triggered by the MantisBT bootstrap process just before emitting the headers. This enables plugins to emit their own headers or use API that enables tweaking values of headers emitted by core. An example, of headers that can be tweaked is Content-Security-Policy header which can be tweaked using http_csp_*() APIs.

EVENT_CORE_READY (Execute)

This event is triggered by the MantisBT bootstrap process after all core APIs have been initialized, including the plugin system, but before control is relinquished from the bootstrap process back to the originating page. No parameters are passed to hooked functions, and no return values are expected.

This event is the first point in page execution where the entire system is considered loaded and ready.

EVENT_REST_API_ROUTES (Execute)

This event is triggered by MantisBT to enable plugins to register their own routes to be accessible via the REST API. All APIs belonging to a plugin named 'Example', MUST live under 'https://.../api/rest/plugins/Example/'. The route registration is done using the Slim Framework app instance that is passed as a parameter. A route group should be used to include all routes for the plugin. The name of the route group should be retrieved via calling plugin_route_group(). See MantisGraph core plugin for an example and Slim Framework router documentation [https://www.slimframework.com/docs/objects/router.html].

Before calling into the plugin routes, the user would be already authenticated and authorized for API access in general. However, it is the responsibility of the plugin to do its own plugin specific authorization.

EVENT_LOG (Execute)

This event is triggered by MantisBT to log a message. The contents of the message should be hyper linked based on the following rules: #123 means issue 123, ~123 means issue note 123, @P123 means project 123, @U123 means user 123. Logging plugins can capture extra context information like timestamp, current logged in user, etc. This event receives the logging string as a parameter.

Parameters

- <String>: the logging string

EVENT_AUTH_USER_FLAGS (First)

An event that enables plugins to return a set of flags that control the authentication behaviors for the user who is logging in or logged in. In some cases, the user will be in the system, but there will be cases

where the username provided by the user doesn't exist. In case the user doesn't exist, it is up to the authentication plugin whether to fail the login, validate credentials then fail, or validate credentials then auto-provision the user based on information the plugin is aware of (e.g. IDP or some db of accounts). If no plugin is registered for events, then defaults are used. If plugin sets a subset of the options, then the default will be used for the rest.

Checkout SampleAuth plugin [https://github.com/mantisbt-plugins/SampleAuth] for more details.

Output Modifier Events

String Display

These events make it possible to dynamically modify output strings to interpret or add semantic meaning or markup. Examples include the creation of links to other bugs or bugnotes, as well as handling urls to other sites in general.

EVENT_DISPLAY_BUG_ID (Chained)

This is an event to format bug ID numbers before being displayed, using the `bug_format_id()` API call. The result should be plaintext, as the resulting string is used in various formats and locations.

Parameters

- <String>: bug ID string to be displayed
- <Integer>: bug ID number

Return Value

- <String>: modified bug ID string

EVENT_DISPLAY_EMAIL (Chained)

This is an event to format text before being sent in an email. Callbacks should be used to process text and convert it into a plaintext-readable format so that users with textual email clients can best utilize the information. Hyperlinks and other markup should be removed, leaving the core content by itself.

Parameters

- <String>: input string to be displayed

Return Value

- <String>: modified input string

EVENT_DISPLAY_EMAIL_BUILD_SUBJECT (Chained)

This is an event to format the subject line of an email before it is sent.

Parameters

- <String>: input string for email subject

Return Value

- <String>: modified subject string

EVENT_DISPLAY_FORMATTED (Chained)

This is an event to display generic formatted text. The string to be displayed is passed between hooked callbacks, each taking a turn to modify the output in some specific manner. Text passed to this may be processed for all types of formatting and markup, including clickable links, presentation adjustments, etc.

Parameters

- <String>: input string to be displayed

Return Value

- <String>: modified input string
- <Boolean>: multiline input string

EVENT_DISPLAY_RSS (Chained)

This is an event to format content before being displayed in an RSS feed. Text should be processed to perform any necessary character escaping to preserve hyperlinks and other appropriate markup.

Parameters

- <String>: input string to be displayed
- <Boolean>: multiline input string

Return Value

- <String>: modified input string

EVENT_DISPLAY_TEXT (Chained)

This is an event to display generic unformatted text. The string to be displayed is passed between hooked callbacks, each taking a turn to modify the output in some specific manner. Text passed to this event should only be processed for the most basic formatting, such as preserving line breaks and special characters.

Parameters

- <String>: input string to be displayed

- <Boolean>: multiline input string

Return Value

- <String>: modified input string

Menu Items

These events allow new menu items to be inserted in order for new content to be added, such as new pages or integration with other applications.

EVENT_MENU_ACCOUNT (Default)

This event gives plugins the opportunity to add new links to the user account menu available to users from the 'My Account' link on the main menu.

Return Value

- <Array>: List of HTML links for the user account menu.

EVENT_MENU_DOCS (Default)

This event gives plugins the opportunity to add new links to the documents menu available to users from the 'Docs' link on the main menu.

Return Value

- <Array>: List of HTML links for the documents menu.

EVENT_MENU_FILTER (Default)

This event gives plugins the opportunity to add new links to the issue list menu available to users from the 'View Issues' link on the main menu.

Return Value

- <Array>: List of HTML links for the issue list menu.

EVENT_MENU_ISSUE (Default)

This event gives plugins the opportunity to add new links to the issue menu available to users when viewing issues.

Parameters

- <Integer>: bug ID

Return Value

- <Array>: List of HTML links for the documents menu.

EVENT_MENU_MAIN (Default)

This event gives plugins the opportunity to add new menu options to the main menu. New links will be added AFTER the standard menu options.

Return Value

- <Array>: Hooked events may return an array of menu options. Each array entry will contain an associate array with keys 'title', 'url', 'access_level', and 'icon' (e.g. fa-pencil from Font Awesome [http://fontawesome.io/icons/]).

```
return array(
    array(
  'title' => 'My Link',
  'access_level' => DEVELOPER,
  'url' => 'my_link.php',
  'icon' => 'fa-random'
 ),
 array(
  'title' => 'My Link2',
  'access_level' => DEVELOPER,
  'url' => 'my_link2.php',
  'icon' => 'fa-shield'
 )
);
```

EVENT_MENU_MAIN_FRONT (Default)

This event gives plugins the opportunity to add new menu options to main menu. New links will be added BEFORE the standard menu options.

Return Value

- <Array>: Hooked events may return an array of menu options. Each array entry will contain an associate array with keys 'title', 'url', 'access_level', and 'icon' (e.g. fa-pencil from Font Awesome [http://fontawesome.io/icons/]).

```
return array(
    array(
  'title' => 'My Link',
  'access_level' => DEVELOPER,
  'url' => 'my_link.php',
  'icon' => 'fa-random'
 ),
 array(
  'title' => 'My Link2',
  'access_level' => DEVELOPER,
  'url' => 'my_link2.php',
  'icon' => 'fa-shield'
 )
```

```
);
```

EVENT_MENU_MANAGE (Default)

This event gives plugins the opportunity to add new links to the management menu available to site administrators from the 'Manage' link on the main menu. Plugins should try to minimize use of these links to functions dealing with core MantisBT management.

Return Value

- <Array>: List of HTML links for the management menu.

EVENT_MENU_MANAGE_CONFIG (Default)

This event gives plugins the opportunity to add new links to the configuration management menu available to site administrators from the 'Manage Configuration' link on the standard management menu. Plugins should try to minimize use of these links to functions dealing with core MantisBT configuration.

Return Value

- <Array>: List of HTML links for the manage configuration menu.

EVENT_MENU_SUMMARY (Default)

This event gives plugins the opportunity to add new links to the summary menu available to users from the 'Summary' link on the main menu.

Return Value

- <Array>: List of HTML links for the summary menu.

Page Layout

These events offer the chance to create output at points relevant to the overall page layout of MantisBT. Page headers, footers, stylesheets, and more can be created. Events listed below are in order of runtime execution.

EVENT_LAYOUT_RESOURCES (Output)

This event allows plugins to output HTML code from inside the `<head>` tag, for use with CSS, Javascript, RSS, or any other similar resources. Note that this event is signaled after all other CSS and Javascript resources are linked by MantisBT.

Return Value

- <String>: HTML code to output.

EVENT_LAYOUT_BODY_BEGIN (Output)

This event allows plugins to output HTML code immediately after the `<body>` tag is opened, so that MantisBT may be integrated within another website's template, or other similar use.

Return Value

- <String>: HTML code to output.

EVENT_LAYOUT_PAGE_HEADER (Output)

This event allows plugins to output HTML code immediately after the MantisBT header content, such as the logo image.

Return Value

- <String>: HTML code to output.

EVENT_LAYOUT_CONTENT_BEGIN (Output)

This event allows plugins to output HTML code after the top main menu, but before any page-specific content begins.

Return Value

- <String>: HTML code to output.

EVENT_LAYOUT_CONTENT_END (Output)

This event allows plugins to output HTML code after any page- specific content has completed, but before the bottom menu bar (or footer).

Return Value

- <String>: HTML code to output.

EVENT_LAYOUT_PAGE_FOOTER (Output)

This event allows plugins to output HTML code after the MantisBT version, copyright, and webmaster information, but before the query information.

Return Value

- <String>: HTML code to output.

EVENT_LAYOUT_BODY_END (Output)

This event allows plugins to output HTML code immediately before the `</body>` end tag, to so that MantisBT may be integrated within another website's template, or other similar use.

Return Value

- <String>: HTML code to output.

EVENT_VIEW_BUG_ATTACHMENT (Output)

This event allows plugins to output HTML code immediately after the line of an attachment. Receives the attachment data as a parameter, in the form of an attachment array from within the array returned by the `file_get_visible_attachments()` function.

Parameters

- <Array>: the attachment data as an array (see `core/file_api.php`)

Return Value

- <String>: HTML code to output.

Bug Filter Events

Custom Filters and Columns

EVENT_FILTER_FIELDS (Default)

This event allows a plugin to register custom filter objects (based on the `MantisFilter` class) that will allow the user to search for issues based on custom criteria or datasets. The plugin can return either a class name (which will be instantiated at runtime) or an already instantiated object. The plugin must ensure that the filter class has been defined before returning the class name for this event.

Return Value

- <Array>: Array of class names or objects for custom filters

EVENT_FILTER_COLUMNS (Default)

This event allows a plugin to register custom column objects (based on the `MantisColumn` class) that will allow the user to view data for issues based on custom datasets. The plugin can return either a class name (which will be instantiated at runtime) or an already instantiated object. The plugin must ensure that the column class has been defined before returning the class name for this event.

Return Value

- <Array>: Array of class names or objects for custom columns

Bug and Bugnote Events

Bug View

EVENT_VIEW_BUG_DETAILS (Execute)

This event allows a plugin to either process information or display some data in the bug view page. It is triggered after the row containing the target version and product build fields, and before the bug summary is displayed.

Any output here should be defining appropriate rows and columns for the surrounding

`<table>`

elements.

Parameters

- <Integer>: Bug ID

EVENT_VIEW_BUG_EXTRA (Execute)

This event allows a plugin to either process information or display some data in the bug view page. It is triggered after the bug notes have been displayed, but before the history log is shown.

Any output here should be contained within its own

`<table>`

element.

Parameters

- <Integer>: Bug ID

Bug Actions

EVENT_REPORT_BUG_FORM (Execute)

This event allows plugins to do processing or display form elements on the Report Issue page. It is triggered immediately before the summary text field.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Project ID

EVENT_REPORT_BUG_FORM_TOP (Execute)

This event allows plugins to do processing or display form elements at the top of the Report Issue page. It is triggered before any of the visible form elements have been created.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Project ID

EVENT_REPORT_BUG_DATA (Chain)

This event allows plugins to perform pre-processing of the new bug data structure after being reported from the user, but before the data is saved to the database. At this point, the issue ID is not yet known, as the data has not yet been persisted.

Parameters

- <Complex>: Bug data structure (see `core/bug_api.php`)

Return Value

- <Complex>: Bug data structure

EVENT_REPORT_BUG (Execute)

This event allows plugins to perform post-processing of the bug data structure after being reported from the user and being saved to the database. At this point, the issue ID is actually known, and is passed as a second parameter.

Parameters

- <Complex>: Bug data structure (see `core/bug_api.php`)
- <Integer>: Bug ID

EVENT_UPDATE_BUG_FORM (Execute)

This event allows plugins to do processing or display form elements on the Update Issue page. It is triggered immediately before the summary text field.

Parameters

- <Integer>: Bug ID

EVENT_UPDATE_BUG_FORM_TOP (Execute)

This event allows plugins to do processing or display form elements on the Update Issue page. It is triggered immediately before before any of the visible form elements have been created.

Parameters

- <Integer>: Bug ID

EVENT_UPDATE_BUG_STATUS_FORM (Execute)

This event allows plugins to do processing or display form elements in the bug change status form. It is triggered immediately before the add bugnote fields.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Bug ID
- <Integer>: New Status

EVENT_UPDATE_BUG_DATA (Chain)

This event allows plugins to perform pre-processing of the updated bug data structure after being modified by the user, but before being saved to the database.

Parameters

- <Complex>: Updated bug data structure (see `core/bug_api.php`)
- <Complex>: Original bug data structure (see `core/bug_api.php`)

Return Value

- <Complex>: Updated bug data structure (see `core/bug_api.php`)

EVENT_UPDATE_BUG (Execute)

This event allows plugins to perform post-processing of the bug data structure after being updated.

Parameters

- <Complex>: Original bug data structure (see `core/bug_api.php`)
- <Complex>: Updated bug data structure (see `core/bug_api.php`)

EVENT_BUG_ACTIONGROUP_FORM (Execute)

This event allows plugins to do processing or display form elements in the bug group action page form. It is triggered immediately after the standard fields, and before bugnote fields (if applicable).

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Array>: Array of event data elements

Parameter array contents

The parameter array contains elements for these indexes:

- 'action' => <String>: Action title (see `bug_actiongroup.php`)
- 'bug_ids' => <Array>: Array of selected bug ids
- 'multiple_projects' => <Boolean>: Flag if selected bug ids span multiple projects
- 'has_bugnote' => <Boolean>: Flag if current group action form contains a bugnote input

Depending on the action, any of these indexes may appear:

- 'custom_field_id' => <Integer>: If action is 'CUSTOM', contains the custom field id selected for update

EVENT_BUG_ACTION (Execute)

This event allows plugins to perform post-processing of group actions performed from the View Issues page. The event will get called for each bug ID that was part of the group action event.

Parameters

- <String>: Action title (see `bug_actiongroup.php`)
- <Integer>: Bug ID

EVENT_BUG_DELETED (Execute)

This event allows plugins to perform pre-processing of bug deletion actions. The actual deletion will occur after execution of the event, for compatibility reasons.

Parameters

- <Integer>: Bug ID

Bugnote View

EVENT_VIEW_BUGNOTES_START (Execute)

This event allows a plugin to either process information or display some data in the bug notes section, before any bug notes are displayed. It is triggered after the bug notes section title.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Bug ID
- <Complex>: A list of all bugnotes to be displayed to the user

EVENT_VIEW_BUGNOTE (Execute)

This event allows a plugin to either process information or display some data in the bug notes section, interleaved with the individual bug notes. It gets triggered after every bug note is displayed.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Bug ID
- <Integer>: Bugnote ID
- <Boolean>: Private bugnote (false if public)

EVENT_VIEW_BUGNOTES_END (Execute)

This event allows a plugin to either process information or display some data in the bug notes section, after all bugnotes have been displayed.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Bug ID

Bugnote Actions

EVENT_BUGNOTE_ADD_FORM (Execute)

This event allows plugins to do processing or display form elements in the bugnote adding form. It is triggered immediately after the bugnote text field.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Bug ID

EVENT_BUGNOTE_ADD (Execute)

This event allows plugins to do post-processing of bugnotes added to an issue.

Parameters

- <Integer>: Bug ID
- <Integer>: Bugnote ID

EVENT_BUGNOTE_EDIT_FORM (Execute)

This event allows plugins to do processing or display form elements in the bugnote editing form. It is triggered immediately after the bugnote text field.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Integer>: Bug ID
- <Integer>: Bugnote ID

EVENT_BUGNOTE_EDIT (Execute)

This event allows plugins to do post-processing of bugnote edits.

Parameters

- <Integer>: Bug ID
- <Integer>: Bugnote ID

EVENT_BUGNOTE_DELETED (Execute)

This event allows plugins to do post-processing of bugnote deletions.

Parameters

- <Integer>: Bug ID
- <Integer>: Bugnote ID

EVENT_TAG_ATTACHED (Execute)

This event allows plugins to do post-processing of attached tags.

Parameters

- <Integer>: Bug ID
- <Array of Integers>: Tag IDs

EVENT_TAG_DETACHED (Execute)

This event allows plugins to do post-processing of detached tags.

Parameters

- <Integer>: Bug ID
- <Array of Integers>: Tag IDs

Notification Events

Recipient Selection

EVENT_NOTIFY_USER_INCLUDE (Default)

This event allows a plugin to specify a set of users to be included as recipients for a notification. The set of users returned is added to the list of recipients already generated from the existing notification flags and selection process.

Parameters

- <Integer>: Bug ID
- <String>: Notification type

Return Value

- <Array>: User IDs to include as recipients

EVENT_NOTIFY_USER_EXCLUDE (Default)

This event allows a plugin to selectively exclude individual users from the recipient list for a notification. The event is signalled for every user in the final recipient list, including recipients added by the event NOTIFY_USER_INCLUDE as described above.

Parameters

- <Integer>: Bug ID
- <String>: Notification type
- <Integer>: User ID

Return Value

- <Boolean>: True to exclude the user, false otherwise

User Account Events

Account Preferences

EVENT_ACCOUNT_PREF_UPDATE_FORM (Execute)

This event allows plugins to do processing or display form elements on the Account Preferences page. It is triggered immediately after the last core preference element.

Any output here should follow the format found in account_prefs_inc.php. As of 1.3.x this is no longer table elements.

Parameters

- <Integer>: User ID

EVENT_ACCOUNT_PREF_UPDATE (Execute)

This event allows plugins to do pre-processing of form elements from the Account Preferences page. It is triggered immediately before the user preferences are saved to the database.

Parameters

- <Integer>: User ID

EVENT_USER_AVATAR (First)

This event gets the user's avatar as an instance of the Avatar class. The first plugin to respond with an avatar wins. Hence, in case of multiple avatar plugins, make sure to tweak the priorities. Avatars should return null if they don't have an avatar for the specified user id.

Parameters

- <Avatar>: Avatar instance or null.

Management Events

EVENT_MANAGE_OVERVIEW_INFO (Output)

This event allows plugins to display special information on the Management Overview page.

Any output here should be defining appropriate rows and columns for the surrounding <table> elements.

Parameters

- <Boolean>: whether user is administrator

Projects and Versions

EVENT_MANAGE_PROJECT_PAGE (Execute)

This event allows plugins to do processing or display information on the View Project page. It is triggered immediately before the project access blocks.

Any output here should be contained within its own <table> element.

Parameters

- <Integer>: Project ID

EVENT_MANAGE_PROJECT_CREATE_FORM (Execute)

This event allows plugins to do processing or display form elements on the Create Project page. It is triggered immediately before the submit button.

Any output here should follow the format found in manage_proj_create_page.php. As of 1.3.x this is no longer table elements.

EVENT_MANAGE_PROJECT_CREATE (Execute)

This event allows plugins to do post-processing of newly-created projects and form elements from the Create Project page.

Parameters

- <Integer>: Project ID

EVENT_MANAGE_PROJECT_UPDATE_FORM (Execute)

This event allows plugins to do processing or display form elements in the Edit Project form on the View Project page. It is triggered immediately before the submit button.

Any output here should follow the format found in manage_proj_edit_page.php. As of 1.3.x this is no longer table elements.

Parameters

- <Integer>: Project ID

EVENT_MANAGE_PROJECT_UPDATE (Execute)

This event allows plugins to do post-processing of modified projects and form elements from the Edit Project form.

Parameters

- <Integer>: Project ID

EVENT_MANAGE_PROJECT_DELETE (Execute)

This event allows plugins to do pre-processing of project deletion. This event is triggered prior to the project removal from the database.

Parameters

- <Integer>: Project ID

EVENT_MANAGE_VERSION_CREATE (Execute)

This event allows plugins to do post-processing of newly-created project versions from the View Project page, or versions copied from other projects. This event is triggered for each version created.

Parameters

- <Integer>: Version ID

EVENT_MANAGE_VERSION_UPDATE_FORM (Execute)

This event allows plugins to do processing or display form elements on the Update Version page. It is triggered immediately before the submit button.

Any output here should follow the format found in manage_proj_ver_edit_page.php. As of 1.3.x this is no longer table elements.

Parameters

- <Integer>: Version ID

EVENT_MANAGE_VERSION_UPDATE (Execute)

This event allows plugins to do post-processing of modified versions and form elements from the Edit Version page.

Parameters

- <Integer>: Version ID

EVENT_MANAGE_VERSION_DELETE (Execute)

This event allows plugins to do pre-processing of version deletion. This event is triggered prior to the version removal from the database.

Parameters

- <Integer>: Version ID
- <String>: Replacement version to set on issues that are currently using the version that is about to be deleted.

EVENT_MANAGE_USER_CREATE_FORM (Execute)

This event allows plugins to do processing or display form elements on the Create User page. It is triggered immediately before the submit button.

Any output here should follow the format found in manage_user_create_page.php.

EVENT_MANAGE_USER_CREATE (Execute)

This event allows plugins to do post-processing of newly-created users. This event is triggered for each user created. The Manage Users create form is one possible case for triggering such events, but there can be other ways users can be created.

Parameters

- <Integer>: User ID

EVENT_MANAGE_USER_UPDATE_FORM (Execute)

This event allows plugins to do processing or display form elements in the Manage User page. It is triggered immediately before the submit button.

Any output here should follow the format found in manage_user_edit_page.php.

Parameters

- <Integer>: User ID

EVENT_MANAGE_USER_UPDATE (Execute)

This event allows plugins to do post-processing of modified users. This may be triggered by the Manage User page or some other path.

Parameters

- <Integer>: User ID

EVENT_MANAGE_USER_DELETE (Execute)

This event allows plugins to do pre-processing of user deletion.

Parameters

- <Integer>: User ID

EVENT_MANAGE_USER_PAGE (Execute)

This event allows plugins to do processing or display information on the View User page. It is triggered immediately after the reset password segment.

Any output here should be contained within its own container.

Parameters

- <Integer>: User ID

Chapter 6. Integrating with MantisBT

The primary means of integrating with MantisBT with web services is with the bundled SOAP API, which is accessible at `http://server.com/mantis/api/soap/mantisconnect.php`.

Java integration

Prebuilt SOAP stubs using Axis

For ease of integration of the Java clients, SOAP stubs are maintained and deployed in the Maven central repository [http://maven.org/]. For example:

```
<dependency>
    <groupId>biz.futureware.mantis</groupId>
    <artifactId>mantis-axis-soap-client</artifactId>
    <version>1.2.15</version>
</dependency>
```

To include them in your project, download the latest available version [http://search.maven.org/#search|ga|1|g%3A%22biz.futureware.mantis%22].

Usage in OSGi environments

If you would like to use Axis in an OSGi environment, it is recommended that you use a ready-made bundle, such as the Axis bundle available from Eclipse Orbit [http://download.eclipse.org/tools/orbit/downloads/]

Compatibility between releases

The SOAP API signature will change between minor releases, typically to add new functionality or to extend existing features.

Some of these changes might require a refresh of the client libraries generated, for instance Apache Axis 1 SOAP stubs must be regenerated if a complex type receives a new property. Such changes will be announced before the release of the new MantisBT version on the mantisbt-soap-dev mailing list [http://lists.sourceforge.net/mailman/listinfo/mantisbt-soap-dev]. Typically there will be two weeks time to integrate the new SOAP stubs.

Support

The primary means of obtaining support for Web Services and the SOAP API is through the mantisbt-soap-dev mailing list [http://lists.sourceforge.net/mailman/listinfo/mantisbt-soap-dev].

Chapter 7. Appendix

Git References

- The Git SCM web site [http://git-scm.com/documentation/] offers a full reference of Git commands, as well Scott Chacon's excellent *Pro Git* book.
- Github's Git Reference [http://gitref.org/]
- Official documentation (from kernel.org)
  - Manual Page [http://www.kernel.org/pub/software/scm/git/docs/]
  - Tutorial [http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html]
  - Everyday Git With 20 Commands [http://www.kernel.org/pub/software/scm/git/docs/everyday.html]
- Git Crash Course for SVN Users [https://git.wiki.kernel.org/index.php/GitSvnCrashCourse]
- Git From the Bottom Up [http://ftp.newartisans.com/pub/git.from.bottom.up.pdf] (PDF)

Appendix A. Revision History

Revision History		
Revision 2.15-0	Tue Jun 5 2018	VictorBoctor<vboctor@mantisbt.org>
Release 2.15.0		
Revision 2.14-0	Sun Apr 29 2018	VictorBoctor<vboctor@mantisbt.org>
Release 2.14.0		
Revision 2.13-1	Wed Apr 4 2018	VictorBoctor<vboctor@mantisbt.org>
Release 2.13.1		
Revision 2.13-0	Sun Apr 1 2018	VictorBoctor<vboctor@mantisbt.org>
Release 2.13.0		
Revision 2.12-0	Sat Mar 3 2018	VictorBoctor<vboctor@mantisbt.org>
Release 2.12.0		
Revision 2.11-0	Tue Feb 6 2018	VictorBoctor<vboctor@mantisbt.org>
Release 2.11.0		
Revision 2.10-0	Sat Dec 30 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.10.0		
Revision 2.9-0	Sun Dec 3 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.9.0		
Revision 2.8-0	Sat Oct 28 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.8.0		
Revision 2.7-0	Sun Oct 8 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.7.0		
Revision 2.6-0	Sun Sep 3 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.6.0		
Revision 2.5-1	Sat Jun 17 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.5.1		
Revision 2.5-0	Sun Jun 4 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.5.0		
Revision 2.4-1	Sat May 20 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.4.1		
Revision 2.4-0	Sun Apr 30 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.4.0		
Revision 2.3-3	Sun Apr 30 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.3.2		

Revision 2.3-2	Sun Apr 17 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.3.1		
Revision 2.3-1	Fri Mar 31 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.3.0		
Revision 2.2-3	Wed Mar 22 2017	DamienRegad<dregad@mantisbt.org>
Release 2.2.2		
Revision 2.2-2	Sun Mar 12 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.2.1		
Revision 2.2-1	Sun Feb 26 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.2.0		
Revision 2.1-2	Sun Feb 26 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.1.1		
Revision 2.1-1	Tue Jan 31 2017	VictorBoctor<vboctor@mantisbt.org>
Release 2.1.0		
Revision 2.0-2	Fri Dec 30 2016	VictorBoctor<vboctor@mantisbt.org>
Release 2.0.0		
Revision 2.0-1	Sat Nov 26 2016	DamienRegad<dregad@mantisbt.org>
Release 2.0.0-rc.2		