Mantis Bug Tracker Developers Guide

Mantis Bug Tracker Developers Guide

A reference guide and documentation of the Mantis Bug Tracker for developers and community members.

Build Date: 13 January 2009

Table of Contents

Chapter 1. Contributing to MantisBT

MantisBT uses the source control tool Git[1] for tracking development of the project. If you are new to Git, you can find some good resources for learning and installing Git in the Appendix.

Initial Setup

There are a few steps the MantisBT team requires of contributers 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 also configure Git to use terminal colors when displaying file diffs and other information, and you may want to alias certain Git actions to shorter phrases for less 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 primary repository for MantisBT is hosted and available in multiple methods depending on user status and intentions. For most contributers, the public clone URL is git://mantisbt.org/mantisbt.git. To clone the repository, perform the following from your target workspace:

```
$ git clone git://mantisbt.org/mantisbt.git
```

If you are a member of the MantisBT development team with write access to the repository, there is a special clone URL that uses your SSH key to handle access permissions and allow you read and write access. Note: This action *will fail* if you do not have developer access or do not have your public SSH key set up correctly.

```
$ git clone git@mantisbt.org:mantisbt.git
```

After performing the cloning operation, you should end up with a new directory in your workspace, `mantisbt/`. By default, it will only track code from the primary remote branch, `master`, which is the latest development version of MantisBT. For contributers planning to work with stable release branches, or other development branches, you will need to set up local tracking branches in your repository. The following commands will set up a tracking branch for the current stable branch, `master-1.1.x`.

```
$ git checkout -b master-1.1.x origin/master-1.1.x
```

Maintaining Tracking Branches

In order to keep your local repository up to date with the official, there are a few simple commands needed for any tracking branches that you may have, including `master` and `master-1.1.x`.

First, you'll need to got the latest information from the remote repo:

```
$ git fetch origin
```

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

```
$ git checkout master
$ git rebase origin/master
```

Alternatively, you may combine the above steps into a single 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.*

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 checkout feature
$ git rebase master
```

Do note that this changes the commit ID for each commit in your feature branch, which will cause trouble for anyone sharing and/or following your branch. In this case, if you have rebased a branch that other users are watching or working on, they can fix the resulting conflict 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.

```
$ 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.

Submitting Changes

When you have a set of changes to MantisBT that you would like to contribute to the project, there are two preferred methods of making those changes available for project developers to find, retrieve, test, and commit. The simplest method uses Git to generate a specially-formatted patch, and the other uses a public repository to host changes that developers can pull from.

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. This formatted patch allows anyone to import the enclosed changesets directly into Git, where all of the commit information is preserved.

Using a public repository to host your changes is marginally more complicated than submitting a formatted patch, but is more versatile. It allows you to keep your changesets up to date with the offiicial development repository, and it lets anyone stay up to date with your repository, without needing to constantly upload and download new formatted patches whenever you change anything. There is no need for a special server, as free hosting for public repositories can be found on many sites, such as MantisForge.org[3], GitHub[4], or Gitorious[5].

Via Formatted Patches

Assuming that you have an existing local branch that you've kept up to date with `master` as described in Preparing Feature Branches, 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
```

Via Public Repository

We'll assume that you've already set up a public repository, either on a free repository hosting site, or using **git-daemon** on your own machine, and that you know both the public clone URL and the private push URL for your public repository.

For the purpose of this demonstration, we'll use a public clone URL of `git://mantisbt.org/contrib.git`, a private push URL of `git@mantisbt.org:contrib.git`, and a hypothetical topic branch named `feature`.

You'll need to start by registering your public repository as a 'remote' for your working repository, and then push your topic branch to the public repository. We'll call the remote `public` for this; remember to replace the URL's and branch name as appropriate:

```
$ git remote add public git@mantisbt.org:contrib.git
$ git push public feature
```

Next, you'll need to generate a 'pull request', which will list information about your changes and how to access them. This process will attempt to verify that you've pushed the correct data to the public repository, and will generate a summary of changes that you should paste into a bug report or into an email to the developer mailing list:

```
$ git request-pull origin/master git://mantisbt.org/contrib.git feature
```

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://mantisbt.org/contrib.git
$ git checkout -b feature feature/feature
```

If a remote branch is approved for entry into `master`, then it should first be rebased onto the latest commits, so that Git can remove any unnecessary merge commits, and create a single linear history for the branch. Once that's completed, the branch can be fast-forwarded onto `master`:

```
$ git checkout feature
$ git rebase master
$ git checkout master
$ git merge --ff feature
```

If a feature branch contains commits by non-developers, the branch should be signed off by the developer handling the merge, as a replacement for the above process:

```
$ git checkout feature
$ git rebase master
$ git format-patch --binary --stdout master..HEAD > feature_branch.patch
$ git am --signoff feature_branch.patch
```

Notes

1. http://git.or.cz
2. git://mantisbt.org/mantisbt.git
3. http://git.mantisforge.org
4. http://github.com
5. http://gitorious.com

Chapter 2. Database Schema Management

Schema Definition

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

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_param, ... ) ] ] );
```

Each type of event (and individual events themselves) will use different combinations of parameters and return values, so use of the Event Reference is reccommended 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 use of the Event Reference is reccommended 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 immediatly sent to the output buffer via 'echo'. Example usage:

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

EVENT_TYPE_CHAIN

This event type is designed to allow plugins to succesively 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 ofthe new 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 walkthrough 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 walkthrough will be working towards building a single end result: the "Example" plugin as listed in the next section. 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.

The Basics

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.

Plugin Structure

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:

```
Example/
 Example.php
```

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.

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' 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, array of basename => version pai
            'MantisCore' => '1.2',  #   Should always depend on an appropriate version of M
            );

        $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 section.

Using Events and Pages

Configuration and Languages

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

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. Event 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 type 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

Plugin System

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.

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.

Logging

The following event is used to submit a logging message to the plugins. The event gets the logging string as a parameter. Logging plugins can capture extra context information like timestamp, current logged in user, etc.

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. expected.

Output Modifiers

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_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_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

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_MAIN (Default)

This event gives plugins the opportunity to add new links to the main menu at the top (or bottom) of every page in MantisBT. New links will be added after the 'Docs' link, and before the 'Manage' link. Hooked events may return multiple links, in which case each link in the array will be automatically separated with the '|' symbol as usual.

Return Value

- Array: List of HTML links for the main menu.

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.

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_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.

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 similary 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 immediatly 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 immediatly 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 immediatly 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.

Bug / Bugnote Actions

Chapter 6. Appendix

Git References

- Git Official Documentation[1]
- Git Tutorial[2]
- Everyday Git With 20 Commands[3]
- Git Crash Course for SVN Users[4]
- Git From the Bottom Up[5]
- GitCasts[6]

Notes

1. http://www.kernel.org/pub/software/scm/git/docs/
2. http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html
3. http://www.kernel.org/pub/software/scm/git/docs/everyday.html
4. http://git.or.cz/course/svn.html
5. http://www.newartisans.com/blog_files/git.from.bottom.up.php
6. http://www.gitcasts.com/