HookMan Documentation

ESSS

Feb 10, 2023

CONTENTS

This documentation covers HookMan usage & API.

For information about HookMan, read the section above. For public changelog and how the project is maintained, please check the GitHub page

CHAPTER

ONE

WHAT IS HOOKMAN?

HookMan is a python package that provides a plugin management system to applications, specially those who are written (in totally or partially) in C++.

It enables external contributors to implement plugins which act as extensions written in C/C++ that interact with the application through well-defined *hooks*.

This system was largely inspired by pluggy, the plugin system which powers pytest, tox, and devpi, but with the intent to be called from a C++ application rather than from Python.

It was conceived to facilitate the application development, allowing hooks to be exposed in a clear way and allowing plugins to be developed without access to classes or data from the application.

With HookMan your application can have access to the hooks implemented on plugins as simple as the example below.

```
# Initializing a class
hm = HookMan(specs=acme_specs, plugin_dirs=['path1','path2'])

hook_caller = hm.get_hook_caller()

# Getting access to the hook implementation
friction_factor = hook_caller.friction_factor()
env_temperature = hook_caller.env_temperature()

# Checking if the hook was implemented
assert friction_factor is not None
assert env_temperature is None

# Executing the hook, wherever it is implemented either in plugin A or B.
ff_result = friction_factor(1, 2.5)
env_tmp_result = env_temperature(35.5, 45.5)
```

1.1 How does it work?

In order to use HookMan in your application, it is necessary to inform which Hooks are available to be implemented through a configuration object.

With this configuration defined, users can create plugins that implement available Hooks extending the behavior of your application.

All plugins informed to your application will be validated by HookMan (to check which hooks are implemented), and an object holding a reference to the Hooks will be passed to the application.

The `HookMan` project uses the library pybind11 to interact between Python and C/C++, allowing a straightforward usage for who is calling the function (either in Python or in C++).

Defining some terminologies:

- `Application` The program that offers the extensions.
- `Hook` An extension of the Application.
- `Plugin` The program that implements the `Hooks`.
- `User` The person who installed the application.

Read the docs to learn more!

- Documentation: https://hookman.readthedocs.io.
- Free software: MIT license

1.2 Credits

Thanks for pluggy, which is a similar project (plugin system) and source for many ideas.

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

WHAT'S NEXT?

To get quick and running with HookMan you can read the *Quick Start* section.

2.1 Quick Start

As previously mentioned, HookMan is a python application that uses plugins written in C/C++.

In order to integrate this project in your application, it's necessary to create a configuration object named *HookSpecs*. This object provides pieces of information related to which hooks are available and which arguments are expected to be sent or received.

The block code below exemplifies a valid *HookSpecs* configuration:

```
from hookman.hooks import HookSpecs

def env_temperature(arg1: 'double', arg2: 'double') -> 'double':
    """
    Docs for Environment Temperature
    """

def friction_factor(arg1: 'int', arg2: 'double') -> 'double':
    """
    Docs for Friction Factor
    """

specs = HookSpecs(
    project_name='Acme',
    version='1',
    pyd_name='_alfasim_hooks',
    hooks=[
        env_temperature,
        friction_factor,
    ]
)
```

With the *HookSpecs* defined, it's possible to generate the necessary files to interact between the application and the plugins implementation.

```
$ python -m hookman generate-project-files hook_specs.py --dst-path <DEST_DIR>
```

The output from the command above will be the following files:

- HookCaller.hpp
- HookCallerPython.cpp
- CMakeLists.txt

These files contain all code necessary to make the project `pybind11_` integrates with your application, and the CMakeLists file contains a boilerplate to compile and generate the binary extensions (`.pyd` file)

Important: Noticed that the macro `PYBIND11_MODULE` (on `HookCallerPython.cpp`) defines the module name that should be used to import these bindings, and this name is used on the *HookSpecs* object with the field "pyd_name".

With the files generated, and compiled., it's possible now to get an instance of the `HookCaller` object that holds all information related with the hooks implementation.

```
from acme_project import specs

# Initializing a class
hook_manager = HookMan(specs=specs, plugin_dirs=['path1', 'path2'])

hook_caller = hook_manager.get_hook_caller()

# Getting access to the hook implementation
friction_factor = hook_caller.friction_factor()
env_temperature = hook_caller.env_temperature()

# Checking if the hook was implemented
assert friction_factor is not None
assert env_temperature is None
```

The object `hook_caller` contains all references for the functions implemented in the plugins, you can access these methods directly or pass this reference to another module or a C++ function.

2.1.1 Executing in python

The example below shows how to execute the method in a python module.

```
from acme_project import specs

# Initializing a class
hook_manager = HookMan(specs=specs, plugin_dirs=['path1', 'path2'])
hook_caller = hook_manager.get_hook_caller()

# Getting access to the hook implementation
friction_factor_function = hook_caller.friction_factor()

#Executing the method implemented in one of the plugins.
ff_result = friction_factor_function(1, 2.5).

print(f"Result from friction_factor hook: {ff_result}")
```

2.1.2 Executing in C++

As mentioned on the pybind11 functional documentation, the C++11 standard brought the generic polymorphic function wrapper `std::function<>` , which enable powerful new ways of working with functions.

Listing 1: Example of a C++ function that takes an arbitrary function and execute it.

```
int friction_factor(const std::function<double(int, double)> &f) {
    return f(10, 2.5);
}
```

With the binding code for this function in place, it's possible to pass a function implemented on one of the plugins directly to C++.

Listing 2: binding_code.cpp

```
#include <pybind11/functional.h>

PYBIND11_MODULE(my_cpp_binding_module, m) {
    m.def("func_friction_factor", &friction_factor);

}
```

The example below shows how to create an object `hook_caller`, and pass a function implemented on one of the plugins directly to C++ a function.

```
from acme_project import specs

# Initializing a class
hook_manager = HookMan(specs=specs, plugin_dirs=['path1', 'path2'])
hook_caller = hook_manager.get_hook_caller()

# Getting access to the hook implementation
friction_factor_function = hook_caller.friction_factor()

# Importing the binding with the cpp code
import my_cpp_binding_module

# Passing the Friction Factor function to C++
my_cpp_binding_module.func_friction_factor(friction_factor_function)
```

After reading the quick start section, check out these additional resources to help better understand the project flow:

2.2 Creating HookSpecs

In order to inform to `HookMan` which Hooks are available in your application, it's necessary to create a *HookSpecs* object.

```
from hookman.hooks import HookSpecs

def env_temperature(arg1: 'float', arg2: 'float') -> 'float':
    """
```

(continues on next page)

(continued from previous page)

```
    Docs for Environment Temperature
    """

specs = HookSpecs(
    project_name='Acme',
    version='1',
    pyd_name='_alfasim_hooks',
    hooks=[
        env_temperature,
    ]
)
```

This object has the following fields:

- **Project Name:**
  An display name to be used to identify the project and to name the hook functions.
- **Version:**
  Current version of the spec, when a new hook is created or modified this version should be incremented
- **pyd_name:**
  Name of the module exported by `PyBind11` on `HookCallerPython.cpp` file.
- **Hooks:**
  A list with the hooks available for the project, each hook is represented by a python function.

The field hooks should be a list of Python functions, with the following fields filled:

- **Function Name**: The name of hook.
- **Arguments**: The arguments that the Hooks will receive.
- **Type Hints**: The type of argument type.
- **Doc String**: The documentation of the Hook.

Noticed that all the fields are necessary in order to create the *HookSpecs* object, if any of the fiels are not correctly informed a `TypeError` exception will be raised

2.3 Creating plugins

A plugin consists of a `ZipFile` with the extension `.hmplugin` that has the following folder hierarchy.

```
\---<plugin_id>
    |
    +---assets
    |       plugin.yaml
    |       README.md
    |
    \---artifacts
            Linux:    lib<plugin_id>.so
            Windows:  <plugin_id>.dll
```

The `HookMan` project has some utilities to help with the task to generate plugins for the user.

To generate the initial boilerplate for a plugin, execute:

```
$ python -m hookman generate-plugin-template <specs-path> <plugin-name> <shared-lib-name>
↪ <author-name> <author-email> [dst-path]
```

The arguments are:

- **specs-path:**
  Path to the `hook_specs.py` file of the application
- **plugin-name:**
  Name of the plugin to be displayed
- **shared-lib-name:**
  The filename of the compiled plugin
- **author-name:**
  Name of the plugin author to be displayed
- **author-email:**
  Email of the plugin author to be displayed
- **dst-path:**
  A path to where the template generated should be placed, if not given the current directory will be used

The generated template has the following structure:

```
\---<plugin_id>
    |   compile.py
    |   CMakeLists.txt
    |
    +---assets
    |       plugin.yaml
    |       README.md
    |
    \---src
            CMakeLists.txt
            hook_specs.h
            plugin.c
```

Where:

- **plugin.yaml:**
  File with necessary information about the plugin to the application using this plugin
- **plugin.c**
  The source file of the plugin
- **hook_specs.h**
  The header file with all the information necessary to create a plugin for the given application
- **CMakeLists**
  CMake file with the minimum configuration necessary to build a shared library across different platforms
- **README**
  Readme file with the description of the Plugin, to be used by the application.
- **compile.py**
  Script file to generate the shared library, this command will create a folder name artifacts.

2.3.1 Distributing

Plugins should be packaged for distribution and installation in the target software. HookMan plugins are deployed with the `.hmplugin` extension, which is a zip file with the binaries and assets necessary for execution.

To create a `.hmplugin` extension, use this command:

```
$ python -m hookman package-plugin <specs-path> <package-name> <plugin-dir> [dst-path]
```

Where:

- **specs-path:**
  Path to the `hook_specs.py` file of the application
- **package-name:**
  Output name of the package file, without extension. For example: `myplugin-1.0`
- **plugin-dir:**
  Directory where the plugin is located
- **plugin-dir:**
  Directory where the plugin is located
- **dst-path:**
  A path to where put the generated package file, if not given the package will be generated in the same directory as `plugin-dir`.

In order to integrate the `HookMan` project in your application, by listing available plugins and checking conflicts read the section:

2.4 Application utilities

`HookMan` offers a few utilities to help with the task to manage and handle plugins used by the application.

With `HookMan` you can:

- **Install Plugins:**
  Install a plugun by informing the path to where the `.hmplugin` is located.
- **Remove a Plugin:**
  Remove the plugin by informing the name of the plugin.
- **List all plugins availables:**
  Get a list of plugins with all the plugin details.
- **Get the status of all plugins:**
  Check if there are conflicts between them.

The method that list all plugins available return a list of `PluginInfo`, for more details read the *PluginInfo* API section.

Dig deeper into specific topics:

2.5 Creating an SDK

In progress

For more details look the Alfasim SDK project that uses the `HookMan`.

2.6 API

This page contains the full reference to `HookMan` API.

2.6.1 HookMan

class hookman.hooks.**HookMan**(**, specs, plugin_dirs*)

Main class of HookMan, this class holds all the information related to the plugins

get_hook_caller(*ignored_plugins=()*)

Return a HookCaller class that holds all references for the functions implemented on the plugins.

When informed, the *ignored_plugins* must be a list with the names of the plugins (same as shared_lib_name) instead of the plugin caption.

get_plugins_available(*ignored_plugins=()*)

Return a list of *PluginInfo* that are available on `plugins_dirs`

Optionally you can pass a list of plugins that should be ignored. When informed, the *ignored_plugins* must be a list with the names of the plugins (same as shared_lib_name) instead of the plugin caption.

The *PluginInfo* is a object that holds all information related to the plugin.

Return type
`Optional[List[PluginInfo]]`

install_plugin(*plugin_file_path*, *dest_path*)

Extract the content of the zip file into dest_path. If the installation occurs successfully the name of the installed plugin will be returned.

The following checks will be executed to validate the consistency of the inputs:

1. The destination Path should be one of the paths informed during the initialization of HookMan (plugins_dirs field).
2. The plugins_dirs cannot have two plugins with the same name.

Plugin_file_path
The Path for the `.hmplugin`

Dest_path
The destination to where the plugin should be placed.

Return type
`str`

`remove_plugin`(*caption*)

This method receives the name of the plugin as input, and will remove completely the plugin from `plugin_dirs`.

Caption
Name of the plugin to be removed

2.6.2 HookSpecs

`class` hookman.hooks.`HookSpecs`(*, *project_name*, *version*, *pyd_name=None*, *hooks*, *extra_includes=()*)

A class that holds the specification of the hooks, currently the following specification are available:

Parameters

- **`project_name`** – This field will be used to identify the project and to name the hook functions. This is usually a project name in a user-friendly format, such as "My Project".
- **`version`** (*`str`*) – The current version of the spec, when a new hook is created or modified this version should be changed.
- **`pyd_name`** (*`str`*) – Base name of the shared library for the bindings for the HookCaller class. If None, no bindings will be generated.
- **`hooks`** (*`List[function]`*) – A list with the hooks available for the project, each hook is a python function with type annotations.
- **`extra_includes`** (*`List[str]`*) – Extra #include directives that will be added to the generated HookCaller.hpp file.

2.6.3 PluginInfo

`class` hookman.plugin_config.`PluginInfo`(*yaml_location*, *hooks_available*)

Class that holds all information related to the plugin with some auxiliary methods

`classmethod is_implemented_on_plugin`(*plugin_dll*, *hook_name*)

Check if the given function name is available on the plugin_dll informed

---

Note: The hook_name should be the full name of the hook

Ex.: {project}_{version}_{hook_name} -> hookman_v4_friction_factor

Return type
`bool`

`classmethod validate_plugin_file`(*plugin_file_zip*)

Check if the given plugin_file is valid, currently the only check that this method do is to verify if the id is available

2.6.4 HookManGenerator

class hookman.hookman_generator.**HookManGenerator**(*hook_spec_file_path*)

Class to assist in the process of creating necessary files for the hookman

generate_hook_specs_header(*plugin_id*, *dst_path*)

Generates the "hook_specs.h" file which is consumed by plugins to implement the hooks.

Parameters

- **plugin_id** (str) – short name of the generated shared library
- **dst_path** (Union[str, Path]) – directory where to generate the file.

generate_plugin_package(*package_name*, *plugin_dir*, *dst_path=None*, *extras_defaults=None*, *package_name_suffix=None*)

Creates a .hmplugin file using the name provided on package_name argument. The file *.hmplugin* will be created with the content from the folder assets and artifacts.

In order to successfully creates a plugin, at least the following files should be present:

- plugin.yml
- shared library (.ddl or .so)
- readme.md

Per default, the package will be created inside the folder plugin_dir, however it's possible to give another path filling the dst argument

Parameters

- **extras_defaults** (*Dict[str,str]*) – (key, value) entries to be added to "extras" if not defined by the original input yaml.
- **package_name_suffix** (*Optional[str]*) – If not *None* this string is inserted after the plugin version in the filename.

generate_plugin_template(*caption*, *plugin_id*, *author_email*, *author_name*, *dst_path*, *extra_includes=None*, *extra_body_lines=None*, *exclude_hooks=None*, *extras=None*)

Generate a template with the necessary files and structure to create a plugin

Parameters
caption (*str*) – the user-friendly name of the plugin, for example "Hydrates".

A folder with the same name as the plugin_id argument will be created, with the following files:

<plugin_folder>

- CMakeLists.txt
- compile.py

assets/

- plugin.yaml
- README.md

src/

- hook_specs.h

- {plugin_id}.cpp
- CMakeLists.txt

Parameters

- **extra_includes** (Optional[List[str]]) – Extras include to be added on {plugin_id}.cpp as "default", as an example is the includes for a SDK.
- **extra_body_lines** (Optional[List[str]]) – Extras lines to be added on {plugin_id}.cpp on the body, used for default implementations of hooks
- **exclude_hooks** (Optional[List[str]]) – List of hooks, that will not be inserted on the {plugin_id}.cpp

generate_project_files(*dst_path*)

Generate the following files on the dst_path: - HookCaller.hpp - HookCallerPython.cpp

2.6.5 Exception

exception hookman.exceptions.**ArtifactsDirNotFoundError**

Exception raised when the artifacts folder it's not found on the root of the plugin folder

exception hookman.exceptions.**AssetsDirNotFoundError**

Exception raised when the assets folder it's not found on the root of the plugin folder

exception hookman.exceptions.**HookmanError**

Base class for all hookman exceptions.

exception hookman.exceptions.**InvalidDestinationPathError**

Exception raised when the destination path to install the plugin is not one of the paths used by HookMan to find plugins already installed.

exception hookman.exceptions.**PluginAlreadyInstalledError**

Exception raise when a folder with the same name of the plugin already is placed on the destination folder informed.

exception hookman.exceptions.**SharedLibraryNotFoundError**

Exception raise when the file informed doesn't contain a correct shared library Ex.: The user informed a linux plugin on a Windows application.

PYTHON MODULE INDEX

h

INDEX