# Hestia Marketplace Architecture and Design

Andrew Lalis

Last Updated: May 19, 2018

## Contents

# Part I
# Introduction and Background

The Hestia Web App is a method by which users can interact with their local Hestia *controllers*, to manipulate certain *devices* in their home. In order for the controller to be able to send the proper commands to such devices, it is imperative that the user install the device's required *plugin*.

A plugin is a collection of python scripts and configuration files that enables the controller to interact with any IP-enabled device on its local network. To make it easier for users to ensure their devices work without hassle, the *Plugin Marketplace* will provide access to a database of plugins. The marketplace will provide a REST API to let the Web App query plugins and to let plugin developers upload their work.

From the developer's perspective, a plugin is much more than simply a file folder, and as such, information such as the date of creation, author, and a description need to also be stored so that users can easily understand the contents and purpose of a plugin without needing to dissect any source code.

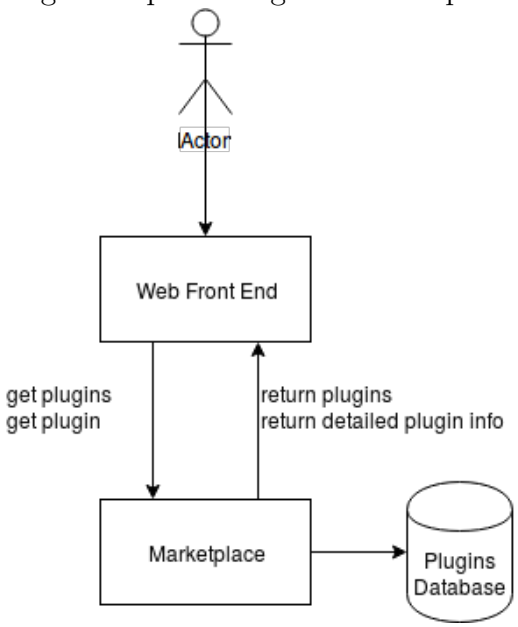
# Part II
# Architecture and Database Design

The main focus of this section will be on the tables needed to store all the information relevant to a plugin. This will consist of a list of attributes that describe a plugin, and any needed auxiliary tables for storing other information in a way that wastes as little space as possible.

## 1 General Architecture

In the most general sense, the purpose of the plugin marketplace is to accept requests for plugins, their metadata, or their files. Therefore, the marketplace is designed around the use of REST endpoints to allow users to interact with a privately managed database.

Figure 1: A diagram representing the marketplace architecture.




# 2 Tables

As mentioned before, the most important table in the database is the **Plugins** table, whose attributes are defined in detail in the following list.

- *id* - A globally unique identification string which can be used to reference this plugin.
- *name* - The name of the plugin, as it will appear in lists when users wish to browse the list of plugins.
- *author_id* - The unique authenticated id of the user who uploaded the plugin.
- *created_date* - The date on which the plugin was first uploaded.
- *last_edited_date* - The date of the most recent update to the plugin.
- *version* - A string of numbers representing the current version of the plugin. This is up to the author's discretion.
- *description_short* - A short and concise description, less than 100 characters, which can be displayed in menus to give users a hint as to what the plugin does.

- *description_long* - A much more detailed description of the plugin, which provides detailed instructions for use of the plugin, or other relevant details that the author thinks the users should know.
- *up_goats* - Up-Goats represent the number of users who cast a positive vote for the plugin, meaning that they believe it is valuable and useful.
- *down_goats* - Down-Goats are the opposite of Up-Goats, and count the number of users who voted negatively for the plugin.

Additionally, some other tables are needed to provide auxiliary information to make the user experience more enjoyable. These are defined below.

The **Tags** table holds many tags, which are can be applied to many plugins to group them into related categories.

- *name* - A unique string which represents a tag, which can be applied to a plugin to give it some extra meaning.
- *description* - A short description of the tag, or an embellishment of its meaning.

In order to manifest the *many-to-many* relationship between tags and plugins, a third table, **PluginTags**, is needed to pair the two together.

- *plugin_id* - The global id of the plugin paired with a tag.
- *tag_name* - The unique name of a tag which is paired to the above plugin.

## 3 File Storage

Because the core of each plugin is a collection of files, these will need to be stored in or alongside the database. In order to keep the files synchronized with the contents of the actual relational database, a **plugins** folder will contain many `.zip` files, each named according the the globally unique id number of the plugin they represent.

# Part III
# REST API

This section contains the API documentation for the HTTP methods used to interact with the server, and defines the syntax for making requests, as

well as what responses should be expected when requests are made, valid or not.

# 4 GET plugins

Returns some basic information about all available plugins, while omitting more specific information like *description_long*.

## 4.1 Syntax

`GET <URL>/plugins`

## 4.2 Responses

### 4.2.1 Success

In the likely event that the server can retrieve the list of plugins, it will return a `JSON` object containing all the plugins' metadata. The structure of that object is defined below.

```
[
        {
                "id": "uniqueid1",
                "name": "Example Plugin 1",
                "author_id": "f83j9pjfpofjpwj4fjf",
                "created_date": "01-01-1970",
                "last_edited_date": "19-05-2018",
                "version": "0.1.0rc3",
                "description_short": "A plugin designed to do
                    nothing.",
                "up_goats": 45,
                "down_goats": 24
        },
        {
                "id": "uniqueid2",
                "name": "Example Plugin 2",
                "author_id": "f83j9pjfpofjpwj4fjf",
                "created_date": "01-03-1945",
                "last_edited_date": "19-05-2018",
                "version": "2.5",
                "description_short": "Another plugin with
                   some metadata.",
                "up_goats": 3,
                "down_goats": 54
        }
]
```

### 4.2.2 Server Error

Because no parameters are required, any error, given a proper `URL`, is the fault of the server, and as such, a `500 Internal Error` message will be returned.

# 5 GET a plugin's metadata

Returns more detailed information about a specific plugin, including the *description_long*.

## 5.1 Syntax

`GET <URL>/plugin/<PLUGIN_ID>`; Where `PLUGIN_ID` is the unique id of the plugin.

## 5.2 Responses

### 5.2.1 Success

If the server successfully finds a plugin with the id specified in the URL, then it will return a JSON object with the structure shown below.

```
{
        "id": "uniqueid1",
        "name": "Example Plugin 1",
        "author_id": "f83j9pjfpofjpwj4fjf",
        "created_date": "01-01-1970",
        "last_edited_date": "19-05-2018",
        "version": "0.1.0rc3",
        "description_short": "A plugin designed to do nothing
           .",
        "description_long": "This is a much longer
           description that is supposed to only be sent when
           someone really wants to know more about this
           specific plugin."
        "up_goats": 45,
        "down_goats": 24
}
```

### 5.2.2 Plugin Not Found

If the id specified in the URL does not refer to any existing plugin, then a `404 Not Found` error message will be returned.

### 5.2.3 Server Error

If the server fails to retrieve information for a plugin which may or may not exist, or if the server fails in some other way, on its own fault, it will return `500 Internal Error`.

# 6 GET a plugin's zipped files

Returns the `.zip` file associated with a plugin whose id is specified in the request.

## 6.1 Syntax

`GET <URL>/plugin/<PLUGIN_ID>/files`; Where `PLUGIN_ID` is the unique id of the plugin.

## 6.2 Responses