Hestia Marketplace Architecture and Design

Andrew Lalis

Last Updated: May 20, 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.

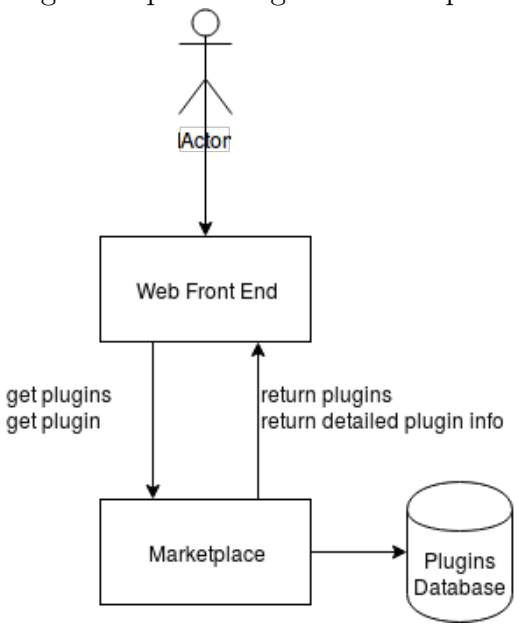
Part II
Architecture and Database Design

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.

Further, because each plugin records the number of votes for it, and each vote is tied to a specific authenticated user, a table is needed to record this information. The **UserVotes** table is described below.

- *user_id* - The unique id of a user.
- *plugin_id* - The unique id of a plugin for which a user has voted.
- *vote* - Either 1 or -1, depending on if the vote is an *up_goat* or *down_goat*.

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*. Some options can be specified to customize the response content, and it is highly encouraged to make use of these, so as to avoid stressing the server by requesting almost the entire contents of the database.

4.1 Syntax

`GET <URL>/plugins`

To constrain the results, it is possible to specify some options via a JSON object, such as the example below.

```
{
        "count": 10,
        "author_id": "fhlufkeh74i7y34fh7oh",
        "title_contains": "example"
        "tags": [
                "cats",
                "light"
        ],
        "sort": {
                "by": "up_goats",
                "order": "ASC"
```

```
        }
}
```

More specifically, a **count** can limit the query to some number of plugins. **tags** is used to filter the query to only contain plugins whose tags contain all those listed inside that list. **title_contains** will filter plugins so that only those whose title contains the supplied text will be returned. **author_id** can be used to only return plugins by a specified author.

It is also possible to request that the returned data be sorted, and this can be done with the **sort** key, where the **by** keyword specifies which data category to sort by. The categories are simply any attributes listed in the **Plugins** table above. Additionally, **order** can be specified to sort the data either ascendingly or descendingly. When this is omitted, data is sorted ascendingly by default.

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,
                "tags": [
                        "cats",
                        "light",
                        "dumb"
                ]
        },
        {
                "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,
                "tags": []
        }
]
```

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

6.2.1 Success

If the plugin specified by the given id exists, then the server will return a response consisting of a `200 Ok`, along with header `Content-Type: application/zip`, which specifies that the server has sent a `.zip` file in the response.

6.2.2 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.

6.2.3 Server Error

In the event of a server error, despite a possibly valid request, `500 Internal Error` will be returned.

7 POST a new plugin

To create a new plugin, one must send a POST request with the necessary data.

7.1 Syntax

`POST <URL>/plugin`; Where the header `Content-Type` is set to `application/json`, and the content of the POST contains data with the same structure as shown below.

```
{
        "name": "Example Plugin 1",
        "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.",
        "tags": [
                "light",
                "rgb",
                "cat"
        ]
}
```

7.2 Responses

7.2.1 Success

If the server is able to successfully create the plugin without any constraint conflicts or other issues, a `201 Created` message will be returned, as well as the id of the newly created plugin in a json object as shown below. Once the plugin has been created, one must then use a PUT request to upload the zipped plugin files to the server, using the received id.

```
{
        "id": "feo7h4fohfo7hf47ih"
}
```

Note that after some time, if a created plugin has not had some files added to it, that plugin will be removed automatically. Therefore, the files should be uploaded immediately after the success of this request is returned.

7.2.2 Constraint Violation

Due to the nature of entering data into a database, certain values must not be null, certain values must be unique, etc. If the content of the POST request does not satisfy the constraints set forth for the database, the server will return `400 Bad Request`.

7.2.3 Authorization Error

If the sender of the request is not authorized via Auth0, then the server will return `401 Unauthorized`, as it is necessary for a user to be logged in to create a plugin so it can be traced to that user.

7.2.4 Server Error

If for some reason the server encounters an error beyond what was defined above, it will return `500 Internal Error`.

8 PUT .zip file to a plugin

Immediately after creating a plugin, one should use this request to upload a `.zip` file to the server, with the same authenticated user as the author of the plugin. Nobody else will be allowed to upload to the plugin except for the author.

This method is also used to update the files of an existing plugin, which will overwrite any existing files.

8.1 Syntax

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

The content of the request contains the `.zip` file, which itself contains the contents of the plugin.

8.2 Responses

8.2.1 Success

If the server is able to successfully receive the zipped files and store it, the server will return `200 Ok`.

8.2.2 Authorization Error

If the sender of the request is not authorized via Auth0, and the author of the plugin to which they are attempting to upload the files, a `401 Unauthorized` error will be returned.

8.2.3 No Files

Because the whole point of this request is to upload a `.zip` file, if none is provided, the server will return a `400 Bad Request` error.

8.2.4 Plugin Not Found

If the id provided does not refer to any existing plugin, then `404 Not Found` will be returned.

8.2.5 Server Error

Beyond all the possible user errors written above, if the server fails on its own accord, then it will return a `500 Internal Error`, at which point the user can assume that their files have not been saved on the server.

9 DELETE a plugin

This will irreversibly delete both a plugin and any zipped files associated with it. This action must be performed by either an administrator or the plugin owner.

9.1 Syntax

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

9.2 Responses

9.2.1 Success

If the resource is deleted successfully, as well as the associated files, then the server will return `205 Reset Content`, as it will be necessary for the user to refresh their content view now that the plugin has been deleted.

9.2.2 Authorization Error

If the user is not both authenticated, and the author of the plugin, then they are forbidden from deleting the plugin. The server will return `401 Unauthorized`.

9.2.3 Plugin Not Found

If the id provided does not refer to any existing plugin, then `404 Not Found` will be returned.

9.2.4 Server Error

If for some reason, the server is not able to delete both the plugin in the database and the associated files, then it will return `500 Internal Error`.

10 PATCH a plugin's metadata

Using this method, it is possible to update parts of the plugin's metadata, if the user is authenticated as the author of the plugin.

10.1 Syntax

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

The new data can be sent in the same format as when POSTing a new plugin, although here it is permissable to omit fields which do not need to be updated. Therefore, the following example from the POST request is applicable here too.

```
{
        "name": "Example Plugin 1",
        "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.",
        "tags": [
```

```
                "light",
                "rgb",
                "cat"
        ]
}
```

10.2 Responses

10.2.1 Success

If the server can successfully use the data which is sent to update the plugin's metadata, then the server will return a `205 Reset Content` code, as the user's view should be updated to reflect the successful changes.

10.2.2 Authorization Error

If the user is not both authenticated, and the author of the plugin, then they are forbidden from updating the metadata of the plugin. The server will therefore return `401 Unauthorized`.

10.2.3 Plugin Not Found

If the id provided does not refer to any existing plugin, then `404 Not Found` will be returned.

10.2.4 Server Error

If for some reason, the server is not able to update the metadata and fails internally, it will return a `500 Internal Error`.

11 POST to vote for a plugin

Use this method to allow a user to cast a vote for a particular plugin. The vote will be traced to their authenticated user id, so that the vote can be changed at any time simply by calling this method again.

11.1 Syntax

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

The content of the post request must then follow the structure shown below.

```
{
        "vote": 1
}
```

Note that the value of the key `"vote"` may be either 1 (*up_goat*), 0 (*reset vote*), or -1 (*down_goat*). Whatever value is sent will override any previous vote by that user.

11.2 Responses

11.2.1 Success

If the user's vote is successfully cast, then the server will return `205 Reset Content`, indicating that the user's document should reflect the change in vote.

11.2.2 Authorization Error

Because only authorized users are allowed to vote, those who are not will receive a `401 Unauthorized` response when attempting to send this request.

11.2.3 Plugin Not Found

If the id provided does not refer to any existing plugin, then `404 Not Found` is returned.

11.2.4 Server Error

If it is not possible for the server to register the vote, then it will return `500 Internal Error`, and the user should interpret this as that no change has been made to the votes.