UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

Breno Rios Ramos

Plugin submission for extensible tools - *piStar Tool* case

RECIFE

2019

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

Breno Rios Ramos

Plugin submission for extensible tools - *piStar Tool* case

Monografia apresentada ao Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Engenharia da Computação, orientada pelo professor Jaelson Freire Brelaz de Castro e co-orientada pelo professor João Henrique Correia Pimentel.

RECIFE

2019

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

Breno Rios Ramos

Plugin submission for extensible tools - *piStar Tool* case

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 03 de Julho de 2019.

Banca Examinadora:

Orientador

____

Jaelson Freire Brelaz de Castro

Doutor

Examinador

____

Kiev Gama

Doutor

I dedicate this work to God and my family
that made it all possible.

SPECIAL THANKS

To professors Jaelson Freire Brelaz de Castro and João Henrique Correia Pimentel for the extremely caring guidance.
To all my colleagues, from the ones who became friends to the others, that made this journey a easier to go through.
Also my family and girlfriend that helped me made it through the bad times both financially and psychologically. Without them this work would never be possible.

“The man who refuses to judge, who neither agrees nor disagrees, who declares that there are no absolutes and believes that he escapes responsibility, is the man responsible for all the blood that is now spilled in the world. Reality is an absolute, existence is an absolute, a speck of dust is an absolute and so is a human life.”

Ayn Rand - Atlas Shrugged

ABSTRACT

An extensible architecture is the one that allows anyone, besides the original author of the application, to add functionalities without the necessity of accessing the source code or the original application creators intervention. It is present in all kinds of software applications nowadays and it is a very popular choice in the development of browsers and IDE´s (Integrated Development Environment). In the case of piStar Tool, there is a necessity of expansion for supporting new extensions for the i* language.

The *piStar Tool* is already a pluggable tool. However, it is still necessary the intervention of the original creator in the source code or the intervention of a plugin creator, cloning the source code repository and creating their own *piStar Tool* version. This work aims to define and implement a plugin submission tool for *piStar Tool*. This is going to be achieved through a preliminary analysis of tools that already support this kind of service, as well as tools that are code repository services. As a result, we are going to have the documentation and implementation of the submission tool, allowing the installation of new extensions using the *piStar Tool* graphical interface.

Keywords: extensible architecture, requirements, piStar Tool, plugin architecture

SUMMARY

1. Introduction 19
1.1. Context 19
1.2. Motivation 19
1.3. Goals 20
1.4. Methodology 20
1.5. Structure 20
2. piStar Tool 21
2.1. Introduction 21
2.2. i* framework 21
2.3. Plugins 21
3. Existing tools 23
3.1. Visual Studio Code 23
3.1.1. Introduction 23
3.1.2. Plugin Creation Process 23
3.1.3. Plugin Submission Process 24
3.1.4. Installation and Usage Process 24
3.1.5. Conclusions 26
3.2. Google Docs 26
3.2.1. Introduction 26
3.2.2. Plugin creation process 27
3.2.3. Plugin submission process 28
3.2.4. Installation and usage process 28
3.2.5. Conclusions 29
3.3. Sketchup 29
3.3.1. Introduction 29
3.3.2. Plugin Creation Process 29
3.3.3. Plugin Submission Process 32
3.3.4. Installation and Usage Process 32
3.3.5. Conclusions 33

3.4. StarUML 34
3.4.1. Introduction 34
3.4.2. Plugin Creation Process 34
3.4.3. Plugin Submission Process 35
3.4.4. Installation and Usage Process 36
3.4.5. Conclusions 36
3.5. Google Chrome 37
3.5.1. Introduction 37
3.5.2. Plugin Creation Process 37
3.5.3. Plugin Submission Process 40
3.5.4. Installation and Usage Process 41
3.5.5. Conclusions 42
3.6. Wordpress 42
3.6.1. Introduction 42
3.6.2. Plugin Creation Process 42
3.6.3. Plugin Submission Process 45
3.6.4. Installation and Usage Process 45
3.6.5. Conclusions 46
3.7. Eclipse 46
3.7.1. Introduction 46
3.7.2. Plugin Creation Process 46
3.7.3. Plugin Submission Process 49
3.7.4. Installation and Usage Process 50
3.7.5. Conclusions 50
3.8. Audacity 51
3.8.1. Introduction 51
3.8.2. Plugin Creation Process 51
3.8.3. Plugin Submission Process 52
3.8.4. Installation and Usage Process 52
3.8.5. Conclusions 53
3.9. *GIMP* (GNU Image Manipulation Program) 53

3.9.1. Introduction 53
3.9.2. Plugin Creation Process 53
3.9.3. Plugin Submission Process 55
3.9.4. Installation and Usage Process 55
3.9.5. Conclusions 56
3.10. Shopify 56
3.10.1. Introduction 56
3.10.2. Plugin Creation Process 56
3.10.3. Plugin Submission Process 59
3.10.4. Installation and Usage Process 59
3.10.5. Conclusions 60
3.11. Conclusions 61
4. p*iStar - Plugin Submission Tool* 62
4.1. Introduction 62
4.2. User roles 62
4.3. User Stories 63
4.4. Architecture 64
4.4.1. *MVC* Pattern with *Node.js* **Error! Bookmark not defined.**
4.5. Database 66
4.5.1. Overview 66
4.5.2. Schema 66
4.5.2.1. Plugins 67
4.5.2.2. Users 67
4.5.2.3. UsersPlugins 68
4.5.2.4. Keywords 68
4.5.2.5. PluginsKeywords 68
4.5.2.6. Reviews 69
4.5.3. Redis 69
4.5.4. Postgres 70
4.6. Views 71
4.6.1. Nunjucks 71

4.6.2. Materialize CSS 72
4.7. Screenshots 73
5. Conclusion 75
5.1. Results 75
5.2. Contributions 75
5.3. Future works 75
5.3.1. Analysis of code repository tools 75
5.3.2. Plugin version manager 76
5.3.3. Feedback system 76
5.3.4. Plugin development code abstraction 76
6. References 77

TABLES LIST

Table 1. Tools comparison result.
Table 2. User Roles
Table 3. Common User Stories
Table 4. Administrator User Stories
Table 5. *API* User Stories
Table 6. List of controllers

FIGURES LIST

Figure 1. Directory structure of a plugin for *piStar Tool*.

Figure 2. File structure provided for a extension called *istar* of type *extension (Typescript)*.
(Source: [13]).

Figure 3. Extensions tab opened with a search bar and its 3 divisions of plugins: Enabled, Recommended and Disabled.

Figure 4. Description panel of a selected plugin

Figure 5. Command bar that lets the user search for any available command. In this example, the *Flutter* plugin provides commands to create a new project with the *Flutter* structure and setup or even upgrade the plugin itself.

Figure 6. *Script Editor* opened in the plugin script. (Source: [18])

Figure 7. Plugin that lets the user search for a location and add a map screenshot in the document.
(Source [18])

Figure 8. *GSuite Marketplace* homepage.

Figure 9 - *MindMup 2* plugin page at *GsuiteMarketplace*.

Figure 10. *root* file structure (Source: [25])

Figure 11. *root* file of *HelloCube* plugin. (Source: [25])

Figure 12 - *main.rb* file of *HelloCube* plugin. (Source: [25])

Figure 13 - *Extension Warehouse* homepage.

Figure 14 - *Extension Manager*.

Figure 15 - File structure of a *StarUML's* plugin

Figure 16 - Plugin that opens a message box. (Source: [30])

Figure 17 - Declaration of a new menu shortcut that runs a certain command (Source: [30])

Figure 18 - File structure of a *StarUML's* plugin

Figure 19 - Popup of a plugin that allows the user to setup some reminders for drinking water. (Source: [34])

Figure 20 - Tooltip appearing when the mouse hovers. It shows the options to run the plugin using

commands. (Source: [34])
Figure 21 - *Omnibox* that allows the user to run a plugin through an specified keyword. (Source: [34])

Figure 22 - *Context Menu* that displays an option to run the plugin. (Source: [34])

Figure 23 - "commands" field in the *manifest.json.* (Source: [34])

Figure 24 - *manifest.json* (Source: [34])

Figure 25 - *popup.html* (Source: [34])

Figure 26 - Part of the code that holds the logic of the plugin (Source: [34])

Figure 27 - *Chrome Web Store homepage.*

Figure 28 - *Evernote Web Cilpper* plugin page

Figure 29 - HTML file (Source: [38])

Figure 30 - Rendered HTML (Source: [38])

Figure 31 - *frontend* (Source: [38])

Figure 32 - *backend part* 1 (Source: [38])

Figure 33 - *backend part* 2 (Source: [38])

Figure 34 - Part of *Wordpress Plugin Directory's* page of *Google Fonts Typography* plugin

Figure 35 - *PDE* New Plugin Project wizard - part 1. (Source: [43])

Figure 36 - *PDE* New Plugin Project wizard - part 2. (Source: [43])

Figure 37 - Blank project structure (Source: [43])

Figure 38 - Plugin that opens a window (Source: [43])

Figure 39 - Blank project structure (Source: [43])

Figure 40 - Packaging wizard provided by *PDE* (Source: [43])

Figure 41 - *Eclipse Marketplace Client.* (Source: [44])

Figure 42 - Example of a plugin code in *Nyquist* (Source: [48])

Figure 43 - Example of a plugin code in *Nyquist* that opens a dialog box and use the user input values to perform its task. (Source: [48])

Figure 44 - Dialog box opened with 3 controls. (Source: [48])

Figure 45 - *query()* function (Source: [51])

Figure 46 - Plugin path in the application Menu. (Source: [51])

Figure 47 - *run()* function (Source: [51])

Figure 48 - Plugin that opens a dialog box showing a message. (Source: [51])

Figure 49 - *React* component that renders a *HTML div* tag with a paragraph containing a message. (Source: [54])

Figure 51 - Form for plugin creation (Source: [54])

Figure 52 - Example plugin running on the *Development Store*. (Source: [54])

Figure 53 - *index.js* using *UI* components from Polaris. (Source: [54])

Figure 54 - Example plugin with the *ResourcePicker* component from *Polaris*. (Source: [54])

Figure 55 - Shopify App Store homepage

Figure 56 - Bitmeli Mercado Libre Importer plugin page

Figure 57 – *piStar Plugin Submission* Tool requests operation.

Figure 58 - *MVC* structure of *piStar - Plugin Submission Tool*.

Figure 59 - Diagram illustrating all *piStar - Plugin Submission Tool* database tables relationships.

Figure 60 - *Plugins* table schema

Figure 61 - *Users* table schema

Figure 62 - *UsersPlugins* table schema

Figure 63 - *Keywords* table schema

Figure 64 - *PluginsKeywords* table schema

Figure 65 - *Reviews* table schema

Figure 66 - Creation of instances from *express-session* and *connect-redis* packages

Figure 67 - Session configuration

Figure 68 - Database configuration

Figure 69 - *Plugin* mapping with *Sequelize*

Figure 70 - *Plugin* migration file

Figure 71 - Value of a key *name* inside an *user* object displayed inside a *h1* tag

Figure 72 - *default.njk* is extended by adding a the main title in *dashboard.njk*.

Figure 73 - *header.html* is included in another file

Figure 74 - *Button* declaration using *Materialize CSS* styles

Figure 75 - *Materialize CSS* button with icon.

Figure 76 - *Login page*

Figure 77 - *Register page*

Figure 78 - *Dashboard page*

Figure 79 - *Submission form page*

Figure 80 - *piStar Tool* screenshot

Figure 81 - *piStar-4Safety Tool* screenshot

ACRONYMS TABLE

Acronym	Meaning	Context
CASE	Computer-Aided Software Engineering	Class of tools that help software engineering activities
CLI	Command-line Interface	-
GCP	Google Cloud Platform	-
NPM	Node Package Manager	It is two things: an online repository for open-source node projects and it is a *CLI* that interacts with the online repository and helps installing and managing those packages in the users machine or project.
MVC	Model-View-Controller	Type of software design pattern
TTL	Time to live	Time to live of a key stored in *Redis* that has a timeout, it has
GCP	Google Cloud Platform	Cloud computing service from Google
.rb	*Ruby* file	*Ruby* file extension
.json	*Javascript Object Notation* file	*Javascript Object Notation* file extension
JSON	*Javascript Object Notation*	*Javascript Object Notation*
API	Application Program Interface	It consists of a set of objects and methods that are used as an access interface between two systems. An *API* allows the user to access only the exposed

		parts of the system.
UI	User Interface	Often relates to the interface that users can interact with.
LADSPA	Linux Audio Developer's Simple Plugin API	API that allows developers to build plugins for audio applications.
LV2	Linux Audio Developer's Simple Plugin API version 2	Successor of LADSPA.
VST	Virtual Studio Technology	Interface that integrates synthesizers and audio effects with digital audio applications.

1. Introduction

This chapter presents the context which this work was developed and its motivation. It also describes the goals and the research methodology used. At the end the structure of the work is defined.

1.1. Context

Software Engineering has the goal to guide development to guarantee certain qualities as formality, abstraction, decomposition, generalization and flexibility. Decomposition and flexibility have become very desired characteristics due to the high demand of new functionalities that have to be implemented in the minimum possible time [1].

An extensible architecture has become very popular in all kinds of applications, precisely for the power of mitigating the problem mentioned above. A lot of them are not just capable of adding some plugins, but are completely formed by them, connected to a common core [2][3].

All applications that allow third-party developers to write plugins to it, have some form of system to distribute it. It can have a centralized system where all plugins are listed in a repository, allowing users to search and even install them [12][20][28][31][35][39][44][53]. Other applications do not have any centralized repository, so the developers are responsible for making it available to the community [46][50].

1.2. Motivation

The *piStar Tool* is a CASE tool that allows the creation of Goal Modelling Diagrams using i* framework that is going to be explained in section 2.2 [64][65]. It is already pluggable, meaning that it is possible to attach another language extension diagram tool [5]. However, it still requires that plugin creators clone the source code and implement their own version of *piStar Tool*, because there is no central repository to gather all extensions in a format that these can be loaded into the tool.

Commonly, more specific extensions are created to model some areas like Safety, Context, Risks, Sustainability and many more listed in [7]. Besides i* language extensions , the tool has the possibility of supporting the creation of other modelling languages diagrams [8]. Given the given large amount of i* language extensions and different languages that use diagrams as tools to modelling, the use of a completely integrated plugin submission tool to a

central repository and the capability of loading these plugins trough *piStar Tool* graphical interface would help the growth and the use of the tool.

1.3. Goals

This work aims to define and implement a plugin submission tool to *piStar Tool* that stores the plugins in a central repository, allowing an even faster expansion of its functionalities, adding support of new languages and extensions of the i* language with the help of the user's community. The submission tool should allow the submission and storage of new plugins that can be loaded in *piStar Tool* through its graphical interface, not requiring any intervention in the source code.

1.4. Methodology

These goals can be divided in two parts. One of them is to build the central repository to store all plugins submitted to a central repository trough an integrated submission tool. The other one is to build the interface inside *piStar Tool* to load those plugins. This is going to be achieved based on a comparative study of some tools that already use this extensible approach on its architecture. Each tool will be analysed by the following aspects: the plugin creation process, the plugin submission process, the plugin installation and usage process . This analysis will inspire the description of the first requirements of the submission tool and also the interface to load the plugin inside *piStarTool*

1.5. Structure

Each chapter is going to serve a specific goal. In Chapter 2 is going to be explained a little bit about *piStar Tool* and the *i** framework, focusing on the user needs and its plugin structure. In Chapter 3 is going to be presented a list of tools and a brief analysis of each one. The analysis will consist of five parts: a brief *Introduction* describing what the tool is for, the *Plugin Creation Process*, the *Plugin Submission Process,* the *Installation and Usage Process* and some *Conclusions* about each one of these processes, making observations based on what are the needs of *piStar Tool*. In Chapter 4 is going to be introduced *piStar - Plugin Submission Tool* and presented its *User Roles, User Stories*, its project structure and the technologies used. In Chapter 5 is going to be made some considerations about the contributions that this work brings to the community, the results and the possibilities of future works to expand the tool.

2. piStar Tool

2.1. Introduction

piStar Tool is an open-source goal-oriented modelling tool that runs in the browser, , built with *Javascript*. It is focused in modelling diagrams using *i** 2.0 framework. It provides visual features like *Actors* auto-resizing and collapsing, error explanation, *SVG* and *PNG* exporting and also supports non-english symbols. The *i** 2.0 framework currently has a great set of extensions that could take advantage of *piStar Tool* modelling [7][8]. To accomplish that, the extension authors should clone *piStar Tool* repository and add the elements and rules of the extension to the *language* directory of their copy of the tool [5][6]. This work is not to focus on the tool itself, but it in the features relating to its extensible part. More on the tool can be found in [66].

Figure 80 - *piStar Tool* screenshot.

2.2. i* framework

*i** is a goal-oriented framework proposed by Eric Yu as a modelling technique focused in the early phase of the requirements engineering process. Differently from the other modelling techniques that focus in completeness, consistency and automated verification of requirements, *i** focus on modelling the stakeholders interests and how each one of them can be addressed [67].

2.3. Plugins

There are many extensions already built for the *i** framework. These extensions can be translated into plugins to *piStar Tool*. Figure XX shows the *piStar-4Safety Tool* that is a clone of *piStar Tool* that allows the use of the *i** extension called *iStar4Safety.*

Figure 81 – *piStar-4Safety Tool* screenshot

To build a plugin for *piStar Tool* is necessary to create a folder structure with some *Javascript* files describing rules about the language and about its image representation. The *language* directory is where all the files related to the language elements and rules are stored. It has the following structure:

```
language
  constraints.js
  metamodel.js
  shapes.js
  ui.metamodel.js
  images
```

Figure 1 - Directory structure of a plugin for *piStar Tool*.

- *constraints.js*: object that defines the constraints of the metamodel to be used.
- *metamodel.js*: object that defines the metamodel to be used.
- *shapes.js*: declares the configuration of each shape used in the language.
- *ui.metamodel.js*: function that defines additional information used in the *UI* regarding the elements of the language.
- *images*: directory that stores all the *SVG* files used to represent the elements of the language.

3. Existing tools

This chapter describes and analyzes some of the most popular tools that use the extensible architecture and have some form of plugin submission system. First there will be a brief introduction about the tool and then there is going to be the description of the plugin creation process and the plugin submission process. After that there is going to be a brief analysis about each of the main features.

piStar Tool has a user base with different backgrounds. That is why the tools chosen in this chapter were picked from different areas and with different characteristics. The tools go from code editors, passing to audio and image editors, to complete ecommerce platforms. There are open-source tools and non-open source, as well as web-based and desktop-based. Some of them are built for users without any programming background while others have its user base only programmers. The intention was to get a feel of how things are done in different areas. Each section is entirely based on the initial documentation provided in the official websites, so it can reflect how easily a user who wants to build a plugin can get started.

3.1. Visual Studio Code

3.1.1. Introduction

Visual Studio Code is a cross-platform code editor built by Microsoft that covers the common development cycle of editing, building and debugging. It was chosen to be analyzed because it has an extensible architecture model, allowing the community to add more features to the tool. These extensions can be one of the following types: language packs, extension (*Typescript*), extension (*Javascript*), color themes, language support, code snippets and keymaps [9].

3.1.2. Plugin Creation Process

First we have to make sure to have *Git* and *Node.js* installed. To facilitate the process of creating the extensions project, a *NPM (Node Package Manager)* module called *Yeoman* is used.

Yeoman is a scaffolding tool that provides complete project templates or just useful parts of it. The user must select the extensions type desired and provide additional information depending on the first selection. The generated template provides the folder structure below. [10][11]

Figure 2 - File structure provided for a extension called *istar* of type *extension (Typescript).*

The function of each of main directories and files are described below:

- *istar*: is the root folder named after the extension identifier provided to *Yeoman.*
- *.vscode*: contains .json configuration files for launching and debugging the extension as well as building the task that compiles the chosen language, in the example above, Typescript.
- *.src/test*: contains the test scripts.
- *.src/extension.ts*: is the main file, where the plugin functionality is written. It has a function called *activate*, called when the plugin is activated. The function *deactivate* is called when the plugin is deactivated.
- *vsc-extension-quickstart.md*: contains further explanation about the files, instructions to run tests and links to further information.
- *package.json*: contains all the information provided when creating the scaffold with *Yeoman* , basic scripts and information about the dependencies used. It is worth noting that *Visual Studio Code* uses the following extension ID pattern *<publisher>,<name>.*

3.1.3. Plugin Submission Process

It is possible to test locally the plugin created just copying the root directory to *.vscode/extensions*, that is a directory created in the local system when *Visual Studio Code* is installed. To make the plugin accessible to the community, it can be submitted to *Visual Studio Marketplace* using an specific *CLI (Command-line Interface)* to do it. *Visual Studio Code* uses *Azure DevOps* deal with authentication, hosting and management of the extensions, so the user has to create an *Azure* organization [12]. With the account created, the user must use a *CLI* to submit the plugin.

3.1.4. Installation and Usage Process

There is a Extensions tab on the left side panel, as shown in Figure 3, where the user can search the desired plugins available in *Visual Studio Marketplace,* by keywords.

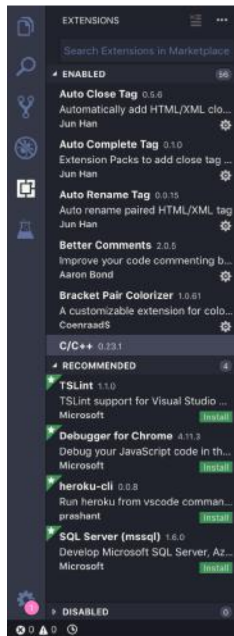

Figure 3 - Extensions tab opened with a search bar and its 3 divisions of plugins: Enabled, Recommended and Disabled.

When a plugin is selected from the list, a description page appears on the right, as shown in Figure 4, showing the name, authors, download count and details about the usage.

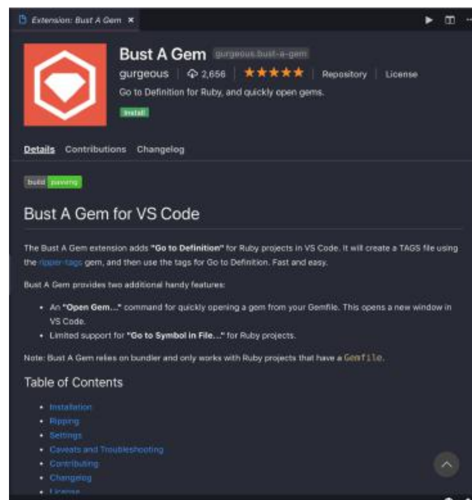

Figure 4 - Description panel of a selected plugin

To install, the user only must click the green button *Install.*

Plugins that are not only language helpers may need some additional step to use. The Figure 5 shows an example of that.

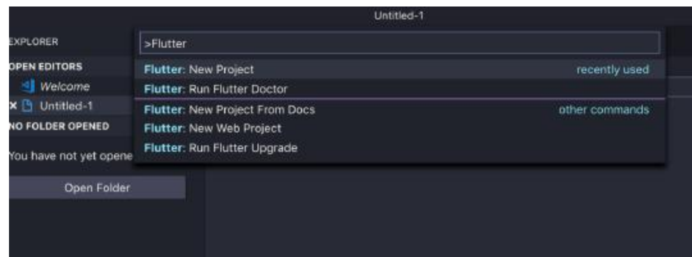

Figure 5 - Command bar that lets the user search for any available command. In this example, the *Flutter* plugin provides commands to create a new project with the *Flutter* structure and setup or even upgrade the plugin itself.

3.1.5. Conclusions

The creation process requires *Typescript* knowledge, so users without an *IT* background will have a bigger barrier to write a new plugin. Its main user is a programmer, so it is not uncomfortable to them to use a *CLI* to package and submit a new plugin. In *piStar Tool*, the main user is not going to always have an *IT* background and therefore not necessarily comfortable using a *CLI*. The installation process is very easy to use and in most cases does not require the reinitialization of the tool, in order to start using the new plugin. So it is valuable to create a submission process that is going to be comfortable to users without *IT* background.

3.2. Google Docs

3.2.1. Introduction

Google Docs is an online based text editor that is part of Google Drive's office applications suite. The main features are: different users can edit the same document at the same time, automatic saving, compatibility with *Microsoft Word* and the capability of extending the application with new plugins. It is worth noting that a lot of the following overview is valid to all the other Google Drive's office applications suite, and there are some peculiarities that are specific to just one of them [14].

3.2.2. Plugin creation process

Google Docs has its own script editor to build plugins using *Javascript, HTML* and *CSS*. First is necessary to be logged in a *Google* account and to create a new *Google Doc*. From within the new document the user can open the *Script Editor* and create a new plugin project.

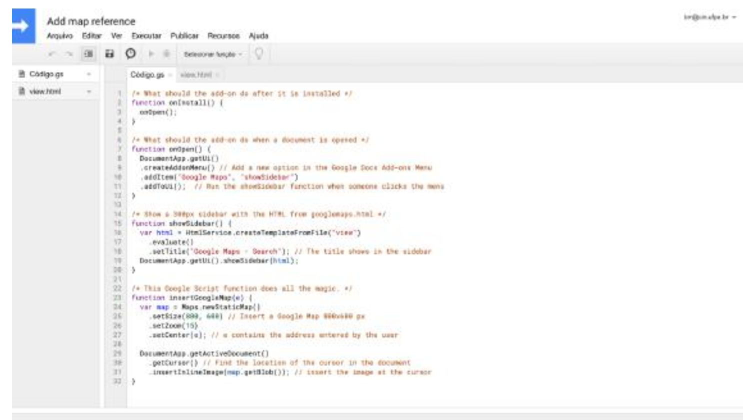

Figure 6 - *Script Editor* opened in the plugin script. (Source: [18])

In this work is going to be used a simple plugin that lets the user search a place in *Google Maps* and add a figure from that region to the current document. There are two main parts of a simple plugin project for *Google Docs*. One of them will use *HTML, CSS* and *Javascript* to deal with the interface of the plugin and its changes. The other one will deal with the specific *triggers* provided by the tool's framework, and they will run when the one of the following events occur: a *doc* is opened, the plugin is installed, or an specified time is reached [15][16][17].

The Figure 7 shows an example of a plugin that allows the user to search for a specific map location and add it to a document, generated by the code in Figure 6.

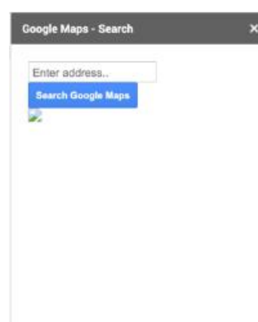

Figure 7 - Plugin that lets the user search for a location and add a map screenshot in the document. (Source [18])

3.2.3. Plugin submission process

All the process is done within the *Apps Script Editor* or in *GCP (Google Cloud Platform)* console. The first step is to create a new version of the add-on. It consists of a static version of the script and a description provided by the user. The next steps are responsible for configuring an *oAuth* consent screen that is prompted for the plugins user, getting *GCP*'s project key, the plugins version number, the localized assets for each language the user wants to publish, the visibility and other basic information [20][21].

3.2.4. Installation and usage process

The publicly published plugins are all available at *GSuite Marketplace*. There is a sidebar that displays some plugin categories and a search bar as shown in figure 8.

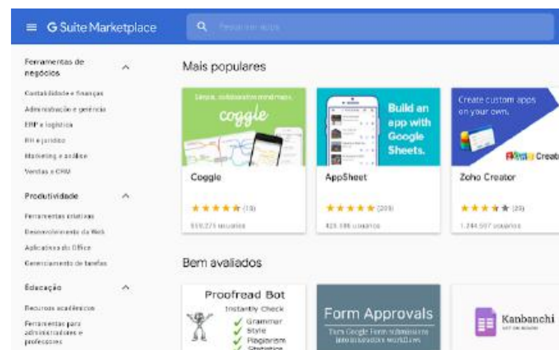

Figure 8 - *GSuite Marketplace homepage*

In the description page is shown some screenshots, reviews the users count and the rating of the selected plugin. To install the plugin it is necessary to press the install button that appears on the top and right side of the page, then a authorization page is prompted asking the user to choose the *Google* account to be used in the installation.

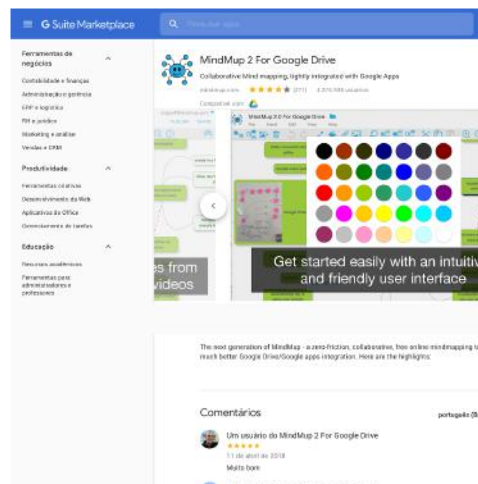

Figure 9 - *MindMup 2* plugin page at *GsuiteMarketplace*

Another screen is shown confirming the installation and showing how to open the new plugin that is available in the *Add-ons* list in the top menu of a document, as shown in Figure 9.

3.2.5. Conclusions

The process of creation of a new plugin requires knowledge in *Javascript, HTML* and *CSS* code, turning this feature almost inaccessible for users with no *IT* background to use. The plugin submission process is not very straightforward and simple, it requires reading of some documentation. The installation process is very intuitive and simple, very similar to other extensible tools, providing a page to search and a install button that makes the plugin available for usage right away. The usage will depend on each plugin functionality.

3.3. Sketchup

3.3.1. Introduction

Sketchup is a 3D modelling software owned by *Trimble Navigation*. Although very well known in Architecture community, it is also used to create models for 3D printing and carpentry. The functionalities of *Sketchup* can be extended with one of the 757 plugins available for it in the *Extension Warehouse* [22].

3.3.2. Plugin Creation Process

A simple *Sketchup* plugin package consists of two *.rb* files. The first one is called the *root* file. The second is the one that describes the plugin functionality itself. The first one is called *main.rb* and makes use of the *Sketchup Ruby API* (*Application Program Interface*). The *API* allows the user to interact with *Sketchup* models and with components of the application. The suggested *root* file structure is shown in Figure 10.

```
module PublisherName
   module ExtensionName
     # ...
   end
 end
```

Figure 10 - *root* file structure (Source: [25])

It encapsulates all the plugins from the same author in its own module. This *module* is recommended and enforced to prevent new plugins to affect already installed ones.

There are three main requirements for a plugin to be accepted in *Extensions Warehouse:*

- Inside the *root* file has to carry a register of a *Sketchup Extension* instance.
- The *root* file should only carry a plugin registration and nothing more.

- Each plugin should be confined in its own *module*, therefore the use of global variables, constants or methods are forbidden as well as any *Sketchup API* and *Ruby API* modifications.

The following plugin is an example available at [25] and the source code is shown in Figure 11. It allows the user to add a simple cube to the current *model*. In the *root* file are loaded *sketchup.rb* and *extensions.rb* files, that are needed to make the plugin *registration*. Inside the *Examples* module it is registered only one plugin called *HelloCube*.

```
require 'sketchup.rb'
require 'extensions.rb'

module Examples
 module HelloCube

   unless file_loaded?(__FILE__)

     ex = SketchupExtension.new('Hello Cube', 'tut_hello_cube/main')

     ex.description = 'SketchUp Ruby API example creating a cube.'
     ex.version     = '1.0.0'
     ex.copyright   = 'Trimble Navigations (c) 2016'
     ex.creator     = 'SketchUp'

     Sketchup.register_extension(ex, true)

     file_loaded(__FILE__)
   end
 end
end
```

Figure 11 - *root* file of *HelloCube* plugin. (Source: [25])

The first line checks if the plugin is already registered, to prevent it to being registered more than one time. After that, a new *SketchupExtension* instance is created. It takes two arguments: the plugin name and the location of the *main.rb* file. In the following lines some other properties of *SketchupExtension* like *description*, *version*, *copyright* and the plugin's *creator* name. The *registration* itself is made in the following line. The *Sketchup.register_extension* method takes as argument a *SketchupExtension* and a boolean that tells *Sketchup* to load the plugin by default if this boolean is set to *true*. If not, the user has to load the plugin manually. The last line of the plugin *module* is needed for the load guard to prevent the *registration* of the same plugin more than one time.

```ruby
require 'sketchup.rb' # 1

module Examples # 1
 module HelloCube # 1

   def self.create_cube # 2

     model = Sketchup.active_model # 3

     model.start_operation('Create Cube', true) # 4

     group = model.active_entities.add_group # 5
     entities = group.entities # 6

     # 7
     points = [
       Geom::Point3d.new(0,   0,   0),
       Geom::Point3d.new(1.m, 0,   0),
       Geom::Point3d.new(1.m, 1.m, 0),
       Geom::Point3d.new(0,   1.m, 0)
     ]

     face = entities.add_face(points) # 8

     face.pushpull(-1.m) # 9

     model.commit_operation # 10
   end


  # 11
   unless file_loaded?(__FILE__)
     menu = UI.menu('Plugins')
     menu.add_item('Create Cube Example') {
       self.create_cube
     }

     file_loaded(__FILE__)
   end

 end
end
```

Figure 12 - *main.rb* file of *HelloCube* plugin. (Source: [25])

In the *main.rb* file, is declared the function that creates the cube in *#2*, shown in Figure 12. Using the *Sketchup API* it is possible to access the current open *model* in *#3*. Each action performed is between the calling of *start_operation()* in *#4* and *commit_operation()* in *#10*. These two functions wrap everything that can be reversed in a single *undo* step. The variable

points in *#7* defines a set of four points that are the vertices used to build the faces of the desired cube in *#8*. Outside *self.create_cube* will be created a new menu item that calls the method when clicked in *#11*.

3.3.3. Plugin Submission Process

The submission requires a *Google ID* and is necessary to be approved to become a developer capable of submitting new plugins. After the approval, the developer must fill out a form with information about the plugin and upload the *.rbz* package. In the form the developer must provide the following information:

- Languages supported by the extension.
- Minimum system requirements.
- Sketchup supported versions.
- Dependencies.
- Documentation.
- Contact for support.

A *.rbz* package is a *.zip* file with a defined structure that contains the *.rb* scripts and possible assets. It is recommended to the documentation to have the following structure:

- A clear guide to get started with menu names and screenshots.
- Usage video.
- Sample files.

A final step before the plugin goes live is the review made by the *Sketchup* review team. They will check if the submission follows the requirements and if it does not, provide some feedback to the developer [26].

3.3.4. Installation and Usage Process

Sketchup provides the *Extension Manager* as tool to install and manage plugins. The plugins available are listed in *Extension Warehouse*. The *Extension Warehouse* homepage displays a list of categories and industries, as Figure 13 shows and only when one of these links are clicked a plugin list is displayed [23].

Figure 13 - *Extension Warehouse* homepage.

The individual plugin page displays a description and some screenshots. It also displays the number of users, reviews and technical information like the size, compatibility and version. To install the plugin, the user must download the plugin package and use *Extension Manager* to install it [27][28].

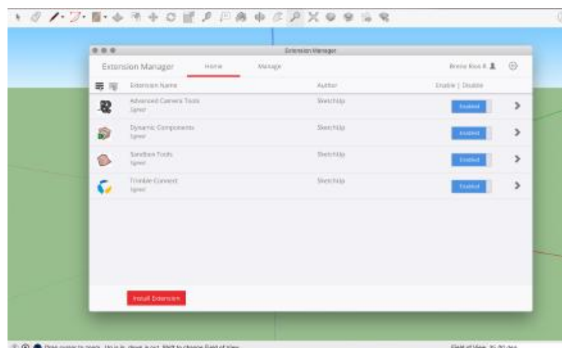

Figure 14 - *Extension Manager*.

3.3.5. Conclusions

The plugin creation requires programming knowledge of *Ruby*. Apart from that the project structure is very simple and easy to understand. The *Sketchup API* and *Ruby API* have dense documentations hence *Sketchup* is a complex graphic software. It is not easy for a user without programming background to start developing plugins for the software. In compensation the plugin submission process is very easy if the plugin package have the right structure and the form have all the required information. The installation and usage process are also very simple and similar to other extensible tools in the market, providing a store to easy navigate through existing plugins and not requiring any programming background to install it. Although

the plugin cannot be installed directly from the store, requiring the user to download a file and load it in *Sketchup* using the *Extension Manager*. (See Figure 14)

3.4. StarUML

3.4.1. Introduction

StarUML is a cross-platform software modelling tool compatible with *UML (Unified Modelling Language)* version 2 standard metamodel and diagrams. Additionally it also supports Entity-Relationship, Data-flow and Flowchart Diagrams [29].

3.4.2. Plugin Creation Process

First the user must create a folder structure as shown in Figure 15. In the *main.js* is the entry point of the plugin: the *init()* function. *StarUML* provides an open *API* to allow the access to the application components. The plugin example in this section opens an alert window when a command is pressed.

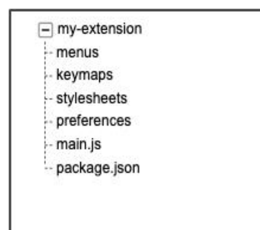

Figure 15 - File structure of a *StarUML's* plugin

The *API* allows a plugin to be executed from the click on a menu item or from a keyboard shortcut. To call the plugin from anyone of those is necessary to register a command. That is accomplished using the instance *app* of *AppContext* class shown in Figure 16. The function *app.commands.register* receives the name of the command and the code block to be executed.

```
function handleShowMessage () {
window.alert('Hello, world!')
}
function init () {
app.commands.register('helloworld:show-message', handleShowMessage)
}
exports.init = init
```

Figure 16 - Plugin that opens a message box. (Source: [30])

To add the same plugin to a menu is necessary to create a *JSON* file, as shown in Figure 17, inside the *menu* folder created in the structure above. The file contains informations like the plugin name, that will appear inside the menu, and the command that it fires when clicked. The same pattern is followed to register a keyboard shortcut inside a *JSON* file in the *keymaps* directory.

```
{
"menu": [
{
"id": "tools",
"submenu": [
{
"label": "Hello World",
"id": "tool.helloworld",
"command": "helloworld:show-message"
}
]
}
]
}
```

Figure 17 - Declaration of a new menu shortcut that runs a certain command (Source: [30])

The elements in a project can be created, accessed and modified using the instance of the current project through *app.project.getProject()*. The *API* documentation provides an extensive documentation on the functions to access each of the elements present in a project and how to add elements to a project [30].

3.4.3. Plugin Submission Process

There are three ways of distributing a plugin for *StarUML*. It is possible to zip the extension folder and unzip in the extensions folder of the application. Another option is to store the extension in a *Github* repository, so users can install from Extension Manager using the *URL* as shown in Figure 18.

The third option is to register the plugin in *Extension Registry* to allow users to install directly from *Extension Manager*. To do that it is necessary to fill out the *package.json* with the following information:

- Plugin name
- Title
- Description
- Homepage
- Issues
- Keywords
- Version
- Author - name, email and *URL*
- License
- Engines

It is also necessary to upload the plugin to *Github* with a *README.md* file containing any useful information and user manual about the extension. With these steps done, the developer must create a release on *Github* and then send an email to the support team of *StarUML* and the plugin will be available in up to two business days [30].

3.4.4. Installation and Usage Process

As seen in section 3.4.3, the installation can be done by unzipping the plugin package inside the extensions folder on the application, using *Extension Manager* with the repository *URL* or directly searching for registered plugins. The user can search and read short descriptions inside the *Extension Manager* window. The usage depends on the plugin installed.

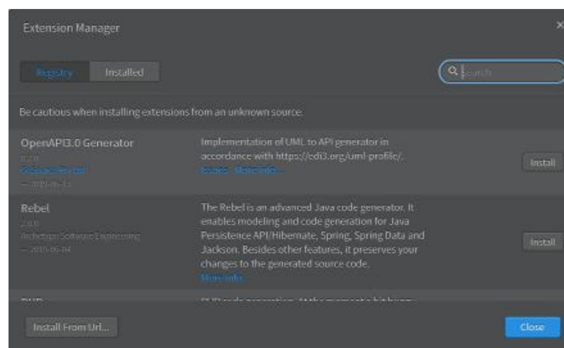

Figure 18 - File structure of a *StarUML's* plugin

3.4.5. Conclusions

The plugin creation process requires programming knowledge in *Javascript*. Also, the documentation of the *API* is extensive. These two facts can put away users without any programming background.

The three options of distribution are a good idea to give the developer more options. All of the three ways are easily done, although users without programming background can feel a little uneasy in the options that require the use of *Github*.

The *Extension Manager* makes the installation process very easy. Also it allows the user to install plugins that did not get through the *Extension Registry* process. It can cause problems if the user tries to install plugins that are not tested, but at the same time cuts one more barrier to developers to distribute their app, cutting out the submission process, which can be a good thing when a developer is just getting started [30][31].

3.5. Google Chrome

3.5.1. Introduction

Google Chrome is a very popular browser launched in 2008 by *Google*. It is also cross-platform and extensible with plugins available in the *Chrome Web Store*.

3.5.2. Plugin Creation Process

As occurs with *Google Docs* plugins, seen on section 2.2 of this work, a *Google Chrome* plugin consists of a package containing *HTML, CSS, Javascript*, images and other necessary files. A *Google Chrome Plugin* can have the following components:

- Manifest
- Background Script
- UI Elements
- Content Script
- Options Page

The Manifest consists of a *.json* file that holds informations like name, version, permissions and important files that it might have to use. The Background Script contains all the listeners that handle the browser events. The interface is handled by the *UI (User Interface)* Elements. The Content Script is only used in plugins that interact with a loaded page, modifying the *DOM (Document Object Model)*. In the Options Page it is possible to customize the plugin if needed.

A plugin *UI* can have the following elements:

- Popup
- Tooltip
- Omnibox
- Context Menu
- Commands

The *Popup* consists in a *HTML* file that is shows up when the user click in its icon placed in the toolbar. It works like a webpage, accepting *HTML, CSS* and *Javascript* with some limitations. (See Figure 19)

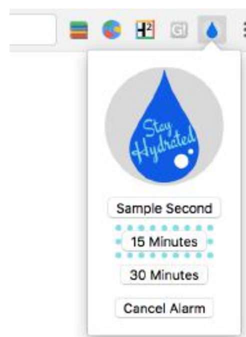

Figure 19 - Popup of a plugin that allows the user to setup some reminders for drinking water. (Source: [34])

The *Tooltip* is used to display some information about the plugins usage when the user hovers the mouse over the icon in the toolbar as in Figure 20. It is also possible to implement internationalization creating specific directories for each language with the translation of the information in the _*locale* folder.

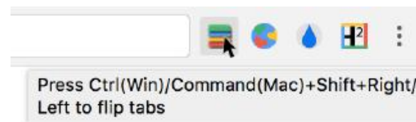

Figure 20 - Tooltip appearing when the mouse hovers. It shows the options to run the plugin using commands. (Source: [34])

The *Omnibox* is a way of invoking the extensions functionality searching for a specific keyword registered in *manifest.json*. (See Figure 21)

Figure 21 - *Omnibox* that allows the user to run a plugin through an specified keyword. (Source: [34])

A plugin can also be invoked in the *Context Menu* that appears when the user clicks with the right button of the mouse in specific areas. It consists in a Menu that displays options based on the context of the place that was clicked, for example, a selection of a text as in Figure 22.

Figure 22 - *Context Menu* that displays an option to run the plugin. (Source: [34])

It is also possible to invoke a plugin through an specific command specified in *manifest.json*. A command is a set of keys that have to be pressed one after the other. (See Figure 23)

```json
{
 "name": "Tab Flipper",

 "commands": {
   "flip-tabs-forward": {
     "suggested_key": {
       "default": "Ctrl+Shift+Right",
       "mac": "Command+Shift+Right"
     },
     "description": "Flip tabs forward"
   },
   "flip-tabs-backwards": {
     "suggested_key": {
       "default": "Ctrl+Shift+Left",
       "mac": "Command+Shift+Left"
     },
     "description": "Flip tabs backwards"
   }
 }

}
```

Figure 23 - "commands" field in the *manifest.json.* (Source: [34])

The example shown in Figure 24, shows a plugin that changes the background of the page in the browser. First there is the *manifest.json* that holds some information about the plugin and sets the *popup.html* file *as the file* where the *UI* is declared. It also specifies the location of icons and the permissions to be used.

```json
{
 "name": "Getting Started Example",
 "version": "1.0",
 "description": "Build an Extension!",
 "permissions": ["declarativeContent","storage"],
 "background": {
   "scripts": ["background.js"],
   "persistent": false
 },
 "page_action": {
   "default_popup": "popup.html",
   "default_icon": {
     "16": "images/get_started16.png",
     "32": "images/get_started32.png",
     "48": "images/get_started48.png",
     "128": "images/get_started128.png"
   }
 },
 "icons": {
   "16": "images/get_started16.png",
   "32": "images/get_started32.png",
   "48": "images/get_started48.png",
   "128": "images/get_started128.png"
 },
 "manifest_version": 2
}
```

Figure 24 - *manifest.json* (Source: [34])

The *popup.html* contains a button and its styling. (See Figure 25)

```
<!DOCTYPE html>
 <html>
   <head>
     <style>
       button {
         height: 30px;
         width: 30px;
         outline: none;
       }
     </style>
   </head>
   <body>
     <button id="changeColor"></button>
   </body>
 </html>
```

Figure 25 - *popup.html* (Source: [34])

After setting up some rules and giving the button a color, it is time to setup the logic of the plugin. The following code adds an event listener that runs a *Javascript* code that changes the color of the page in the browser [32][33][34]. (See Figure 26)

```
let changeColor = document.getElementById('changeColor');

changeColor.onclick = function(element) {
 let color = element.target.value;
 chrome.tabs.query({active: true, currentWindow: true}, function(tabs) {
   chrome.tabs.executeScript(
       tabs[0].id,
       {code: 'document.body.style.backgroundColor = "' + color + '";'});
 });
};
```

Figure 26 - Part of the code that holds the logic of the plugin (Source: [34])

3.5.3. Plugin Submission Process

All plugins for Google Chrome are placed in *Chrome Web Store*. There are basically eight steps in the submission process listed below [35].

- Create the package: a package for a *Google Chrome's* plugin consists in a *ZIP* file with at least the *manifest.json* file. The presence of other files will depend on the plugin functionality.
- Create a developer account: is suggested the creation of a new account specific for the app.
- Upload the app: the package upload is done through the *Chrome Developer Dashboard*. The system will check the validity of the *ZIP* file and of the *manifest.json* file. If the app needs to use its *app ID* or *OAuth token*, the upload must be done before finishing to app.

- Pick a payments system: only for paid plugins or developers that want to use *Chrome Web Store Payments*.
- Provide store content: the developer must provide a detailed description of the plugin, an icon for the store and at least one screenshot in specific sizes, the category and the language that the plugin uses.
- Pay the signup fee: it is a one time only payment that must be done before publishing the plugin.
- Publish the app: it is possible to publish for test groups first and then publish to the entire community. Once the plugin is published to the world it appears the search result on *Chrome Web Store*. It is also possible to control how fast the app can reach new users setting up a "*max deploy percentage*".

3.5.4. Installation and Usage Process

To install a plugin it must be found in *Chrome Web Store*. The store is divided in two types: the extension store and theme store. In is possible to see recommended plugins, added recently, featured and browse by category. A search bar is also available as well as some filters like classification and resources. (See Figure 27)

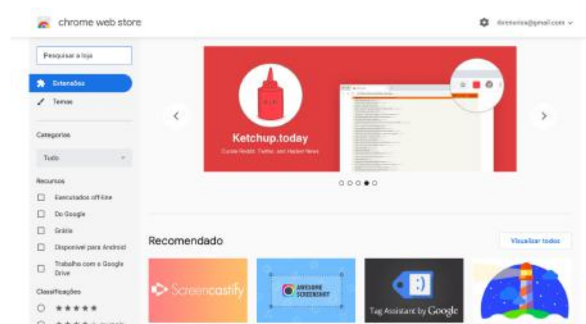

Figure 27 - *Chrome Web Store homepage.*

Each plugin has its own page that is loaded when it is selected. In the page is available the plugins description, homepage, update and version information as well as user comments, support contact, suggestions of other related plugins, classification, number of users and its categories.

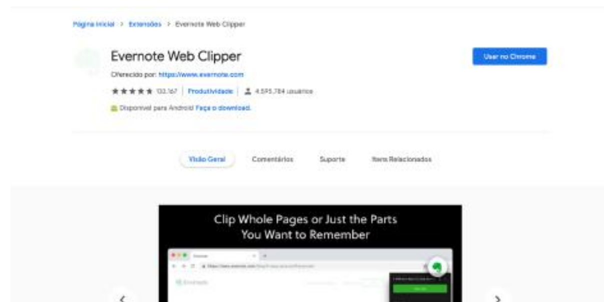

Figure 28 - *Evernote Web Cilpper* plugin page

To install it on the browser the user must click in the button *Add to Chrome*, as shown in Figure 28. Before installation, the browser shows a popup listing the functionality of the plugin that needs permission of the user. If the user accepts and clicks *Add* extension the icon shows up immediately in the tooltip [36].

3.5.5. Conclusions

The plugin creation process requires the use of *HTML*, *CSS* and *Javascript*, so users without a programming background have a great barrier to overcome. For a user with programming background the process is also not very easy due to the very extensive documentation of the library used to develop for *Google Chrome*.

The submission process is fairly easy compared to *Google Doc*s process, and does not require the use of any *CLI* or specific tools. There is also a required fee for the developers to submit plugins, a fact that can put away some of them. Also the installation and usage are very simple and quick, very similar to other applications with a store and a automatic installation.

3.6. Wordpress

3.6.1. Introduction

Wordpress is an open source and web-based *CMS (Content Management System)* that is embedded in 34% of all websites of the world. It makes easy for anyone to build a website and start blogging. The plugins built for *Wordpress* range from ecommerce platforms to *SEO (Search Engine Optimization)* enhancers [37].

3.6.2. Plugin Creation Process

A *Wordpress* plugin can consist of one *PHP* file. It can also include some assets like images, *CSS* and *Javascript* files. The example of plugin used in this section lets a user select a post title and retrieves the number of posts with the same title. The HTML file that will be used is shown in Figure 29 and its result in Figure 30.

```
</pre>
<form id="radioform">
 <table>
  <tbody>
   <tr>
    <td><input class="pref" checked="checked" name="book" type="radio" value="Sycamore Row" />Sycamore
Row</td>
    <td>John Grisham</td>
   </tr>
   <tr>
    <td><input class="pref" name="book" type="radio" value="Dark Witch" />Dark Witch</td>
    <td>Nora Roberts</td>
   </tr>
  </tbody>
 </table>
</form>
<pre>
```

Figure 29 - HTML file (Source: [38])

Sycamore Row	John Grisham
Dark Witch	Nora Roberts

Figure 30 - Rendered HTML (Source: [38])

To access *Wordpress* elements is used the *Heartbeat API*, that is a server polling *API* that is used to handle frontend updated. It works by setting up a time interval to run and get data to the server and wait the response. In Figure 31 there is declared a *jQuery* function that runs when the page loads. On the second line *jQuery*, now represented by *$* symbol, sets a listener to an event of change inside the class *pref*. That means that everytime any element inside a tag with a class equals to *pref* changes, the given function will run.

```
jQuery(document).ready(function($) {
    $(".pref").change(function() {
        var this2 = this;
        $.post(my_ajax_obj.ajax_url, {
           _ajax_nonce: my_ajax_obj.nonce,
            action: "my_tag_count",
            title: this.value
        }, function(data) {
            this2.nextSibling.remove();
            $(this2).after(data);
        });
    });
});
```

Figure 31 - *frontend part* (Source: [38])

Everytime a change happens inside an element with a class *.pref*, the function *$.post* will be called. This function receives as parameters the *URL* of the request, an object containing a string that

represents the action that must be triggered in the server, the data that must be sent and a callback function that runs when the response is retrieved. In this case the data is the selected value passed to the *title* parameter, and the action is "my_tag_count" passed to the *action* parameter. If the action that will be triggered in the server changes in anyway the database, it is also needed a parameter *_ajax_nonce* with a unique hexadecimal serial number that identifies an instance of a form. Both the request *URL* and and the *_ajax_nonce* parameter are set by the *PHP* script and are available in a global object.

On the server side there will be two parts: the one who serves the *jQuery* script in Figure 31 with the needed values, shown in Figure 32, and the one who handles the requests shown in Figure 33. The first part is represented by the function *my_enqueue*. Inside this function is called *wp_enqueue_script* that creates a meta link to the script in the page section. It is not possible to add a meta link without the use of *wp_enqueue_script* because the header is not available to direct access. It takes three parameters: an arbitrary tag to refer to the script file, the complete *URL* to the script and an array listing the other scripts that the script depends on, that in this case is *jquery*.

In *wp_localize_script* is where the localizing process is done. Localizing is the process of creating the global *jQuery* object that is used in the script of Figure 31. This function takes three parameters: the first is the arbitrary tag that refers to the script file that was set in *my_enqueue*, the name of the global object and an array with the objects that the object contains. The elements or the array are the url that points to the *PHP* script and the *$title_nonce* using *wp_create_nonce()* that generates the form identifier mentioned earlier.

```
<?php add_action('admin_enqueue_scripts', 'my_enqueue');
function my_enqueue($hook) {
    if( 'myplugin_settings.php' != $hook) return;
    wp_enqueue_script( 'ajax-script',
        plugins_url( '/js/myjquery.js', __FILE__ ),
        array('jquery')
    );
    $title_nonce = wp_create_nonce('title_example');
    wp_localize_script('ajax-script', 'my_ajax_obj', array(
        'ajax_url' => admin_url( 'admin-ajax.php' ),
        'nonce'   => $title_nonce,
    ));
}
```

Figure 32 - *backend part* 1(Source: [38])

```
add_action('wp_ajax_my_tag_count', 'my_ajax_handler');
function my_ajax_handler() {
    check_ajax_referer('title_example');
    update_user_meta( get_current_user_id(), 'title_preference', $_POST['title']);
    $args = array(
       'tag' => $_POST['title'],
    );
    $the_query = new WP_Query( $args );
    echo $_POST['title'].' ('.$the_query->post_count.') ';
    wp_die(); // all ajax handlers should die when finished
}
```

Figure 33 - *backend part 2* (Source: [38])

The second part of the server side is going to handle the received requests. In the example of Figure 33, the handler called *my_ajax_handler()* first checks the *nonce* using *check_ajax_referer()*. If the *nonce* does not check out the function just calls *wp_die()* that finished the execution. The function *update_user_meta()* to save the user selection, represented by *$_POS['title']*, in user meta. The rest of the handler is going to create the response and send it back, first creating the data that must be sent back in *$args* and then actually sending *$the_query* back [38].

3.6.3. Plugin Submission Process

To submit a plugin for the store the developer must have an account in *Wordpress.org*. The developer must also provide a plugin name and a documentation describing what the plugin does, installation instructions and informations about support. Some instructions and a template for this *README.txt* documentation file is provided by *Worpress*. A form must be filled out by the developer with the following information:

- Plugin name
- Plugin description
- Plugin *URL* - a link to the *ZIP* file that contains the complete version of the plugin and includes the *README.txt*.

After the submission of this form, the developer will receive an email by *Wordpress* within 14 business days. These email is going to contain the result of the review. If approved, a link to a *SVN* (*Apache Subversion*) repository will be provided, so that the developer can host the plugin there. After that the plugin will be available in the store [39][40].

3.6.4. Installation and Usage Process

Wordpress provide *Wordpress Plugin Directory* so that users can search for the available plugins.

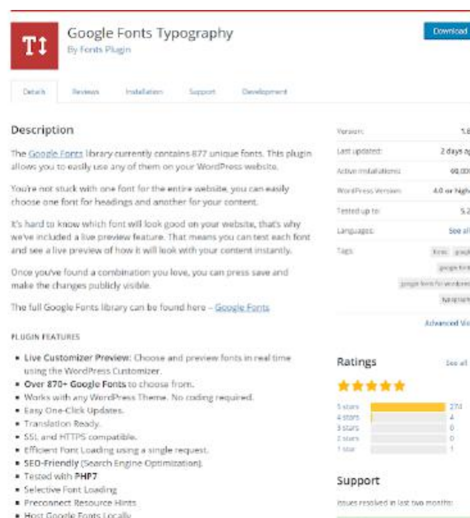

Figure 34 - Part of *Wordpress Plugin Directory's* page of *Google Fonts Typography* plugin

The directory shows a list of plugins divided by category. Each individual plugin page contains the following information as shown in Figure 34:

- Description
- Screenshots
- Authors and contributors
- Technical information (version, last updated, active installations and more)
- Tags
- Rating and Reviews
- Support information

The plugin can be installed directly from the *Wordpress Plugin Directory*. There is a possibility of downloading any plugin *ZIP* file and upload it in *Wordpress* or using *FTP (File Transfer Protocol)* [41].

3.6.5. Conclusions

The plugin creation process makes it harder to users without programming background to start write plugins, because it requires knowledge of *Javascript*, some of its frameworks and *PHP*. It also requires the reading of an extensive documentation. The plugin submission is fairly simple, requiring the filing out of a small form. It has the downside of having its review process done within 14 business days. Although the process is not necessary to test the plugin.

The installation goes beyond other plugin stores of other applications and offers three different methods. Two of them does not require the submission of the plugin for review, which is good for developers that are just starting out and want to test the plugin.

3.7. Eclipse

3.7.1. Introduction

Eclipse is an *IDE* built by the *Eclipse Foundation*, widely used for *Java* development, but with support for other languages. It is built with focus on extensibility, providing a basic workspace and relying most of its features to plugins [42].

3.7.2. Plugin Creation Process

The application provides a set of tools for plugin development called *PDE* (*Plug-in Development Environment)*. It helps the developer from development to deployment phase providing editors, wizards views and a launcher.

A plugin for *Eclipse* consists of a manifest file and, in most cases, the source code written in *Java*. There are plugins that do need any *Java* code such as documentation plugins. This section is going to go through some steps of the development of a Hello World plugin described in []. A new project can be created through *PDE*'s *New Plugin Project* wizard as shown in Figure 35. The developer must create

a unique plugin id, the type of project, the directory for the *Java* builder output, the runtime library and the directory of the source folder.

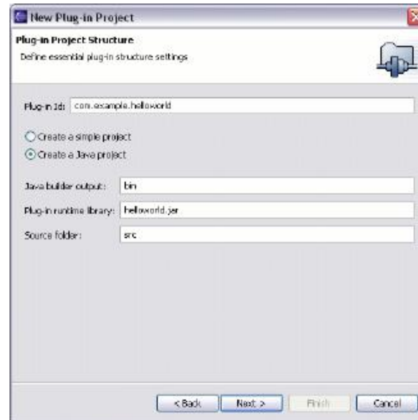

Figure 35 - *PDE* New Plugin Project wizard - part 1. (Source: [43])

In the second part of the wizard, the developer can choose to use a code generator wizard to create a project starting from a template, or to start from a blank one, as shown in Figure 36. In the example analyzed in this section a blank project will be created.

Figure 36 - *PDE* New Plugin Project wizard - part 2. (Source: [43])

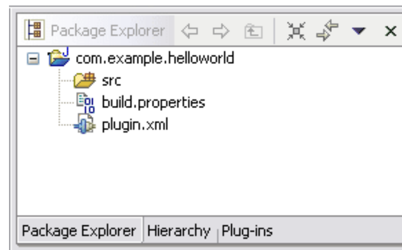

Figure 37 - Blank project structure (Source: [43])

The *plugin.xml* file, shown in Figure 37, describes the plugin content. It contains the following information:

- Basic information like unique identifier, version and name.
- Dependencies section: describes the plugin needs to compile.
- Extensions section: lists the functionalities that the plugin adds to the platform.
- Runtime section: lists the libraries that the plugin code is packaged.
- Extension Points: lists points where other plugins can use to extend its functionality.

In Figure 38 is the code of a plugin that opens a window titled *First plug-in* containing a *Hello, world!* message. The result can be seen in Figure 39.

```
package com.example.helloworld;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActivationDelegate;

public class HelloWorldAction implements IWorkbenchWindowActionDelegate {
 private IWorkbenchWindow window;

 public HelloWorldAction() {

 }

 public void run(IAction action) {
   MessageDialog.openInformation(
     window.getShell(),
     "First plug-in",
     "Hello, world!"
   );
 }

 public void selectionChanged(IAction action, ISelection selection) {

 }

 public void dispose() {

 }

 public void init(IWorkbenchWindow window) {
   this.window = window;
 }
}
```

Figure 38 - Plugin that opens a window (Source: [43])

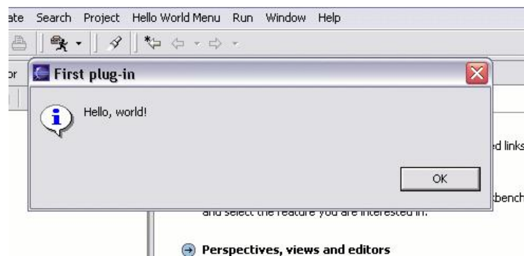

Figure 39 - Blank project structure (Source: [43])

3.7.3. Plugin Submission Process

Before the deployment, a package of the plugin must be created. A package of a plugin for *Eclipse* consists of a *ZIP* file. *PDE* also provides a wizard to guide the packaging process as shown in Figure 40. The process is guided by information about the folders that should be included in the package. Those informations are provided by a configuration file called *build.properties*.

Figure 40 - Packaging wizard provided by *PDE* (Source: [43])

To publish a new plugin in *Eclipse Marketplace* is necessary an account. Once logged in in the *Marketplace* the user must create a new listing and fill out a form with the following fields:

- Solution name
- Organization name
- URL
- Logo

- Description
- Development status
- License type
- Support URL
- Categories that the solution fits in.
- Solution type (*Markets*)
- Tags
- Solution version
- Screenshot
- Ownership (authors)

After the submission the solution must pass through a review that should determine if it is eligible to be published or not [45].

3.7.4. Installation and Usage Process

A user can install a recently developed plugin to test without the submission to *Eclipse Marketplace*. To do this it is necessary to unzip the plugin package directly into the *plugins* folder of the application. To install from *Eclipse Marketplace*, it is necessary to install the *Marketplace* client, shown in Figure 41, using the *Install new software* option in the help menu. The client allows the user to search all available and submitted plugins by name, categories, also listing the recent and most popular submissions [44].

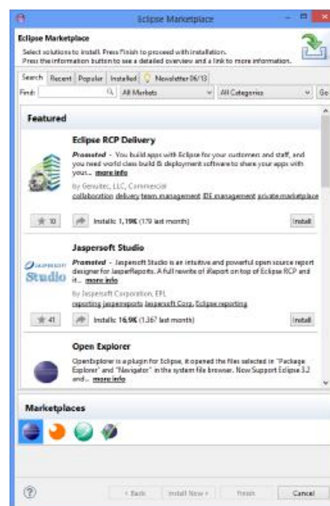

Figure 41 - *Eclipse Marketplace Client.* (Source: [44])

3.7.5. Conclusions

Eclipse plugin development still requires knowledge about *Java* programming knowledge, letting out any user without programming background. A very interesting thing about the plugins generated for *Eclipse* are the *Extension Points* that can be provided so that other developers build upon an existing plugin.

To submit a plugin to the *Eclipse Marketplace* the user must fill out a form and provide the generated package provided by the wizard mentioned above. It needs a review period that is around 24 hours. The review can ensure the quality of the plugin, but at the same time creates one more barrier to the user.

The installation of the plugin is very easy done using the *Marketplace* client, although the installation of the client is not usual. The usage will depend on the plugin installed.

3.8. Audacity

3.8.1. Introduction

Audacity is an open-source and cross-platform multi-track audio editor. Among other common audio editor features, it is an extensible application, allowing the use of audio effect plugins with some *API'* like LADSPA, LV2, *Nyquist* and *VST* [46].

3.8.2. Plugin Creation Process

As mentioned above, *Audacity* accepts a range of different technologies to build plugins. This section is going to focus on the development of plugins using *Nyquist*, that is a section also present in Audacity Manual.

Nyquist is a programming language for audio synthesis and analysis that supports *MIDI*, audio recording and playback, file I/O, object-oriented programming, profiling and debugging. A plugin only consists in a text file with a *.ny* extension and with code written in *Nyquist*. (See Figure 42)

```
;nyquist plug-in
;version 4
;type process
;name "Fade In"
(mult (ramp) *track*)
```

Figure 42 - Example of a plugin code in *Nyquist* (Source: [48])

The first and second line of the code are required in this order. They declare that it is a *Nyquist* plugin and its version respectively. The third line declares the type of plugin that can be one of the four: *generate, process, analyze, tool*. The type will define in which menu of the application the plugin will appear. In the list below are the differences between each one.

- *generate*: for plugins that will generate new audio.
- *process*: for plugins that modify existing audio. It is placed in *Effects* menu.
- *analyze*: for plugins that only process audio without modifying it.
- *tool:* plugins that do not fit in any of the categories above.

The setup of a type is important because, for example, for plugins in of the *process* or *analyze* type, *Audacity* let the selected piece of audio selected available for *Nyquist* in a

environment variable. The commands in the file are executed in order and must return a value. This value replaces whatever is selected by the user.

```
;control decay "Decay amount" int "dB" 6 0 24
;control delay "Delay time" float "seconds" 0.5 0.0 5.0
;control count "Number of echos" int "times" 5 1 30

(defun delays (sig decay delay count)
    (if (= count 0)
        (cue sig)
        (sim (cue sig)
             (loud decay (at delay (delays sig decay delay (- count 1)))))))

  (stretch-abs 1 (delays *track* (- 0 decay) delay count))
```

Figure 43 - Example of a plugin code in *Nyquist* that opens a dialog box and use the user input values to perform its task. (Source: [48])

A *process* or *generate* type plugin can open a dialog box, for example, that allow the user to enter input some parameters to setup the plugin execution, as shown in Figure 43 with the result in Figure 44. Another example can be a plugin of type *analyze* returning a list of tuples with pairs of time and labels that *Audacity* uses to create a track label in each of these time positions.

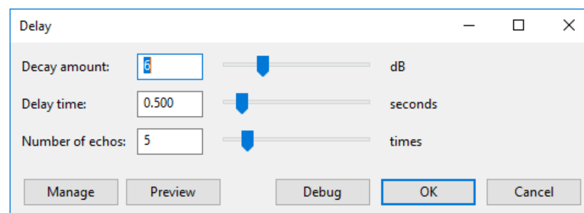

Figure 44 - Dialog box opened with 3 controls. (Source: [48])

3.8.3. Plugin Submission Process

Audacity does not have a central repository or store that aggregates all the plugins, therefore the submission process does not exist. In [46] there is a list of plugins that already are shipped with *Audacity* and a list of authors that contributed with it. To have a plugin listed its necessary to register on *Audacity Forum* and post the plugin for review. The plugin must follow the recommendations listed in a forum post called *Convention for Nyquist Plugins*. As it is a discussion, the recommendations are not listed in a formal way. Each user that wants to contribute can just make a comment [48] [49].

3.8.4. Installation and Usage Process

The installation process will depend on the *API* or language that the plugin was built. In the case of *Nyquist* plugins, the user can add the plugin directly in a specific directory or use a

tool called *Nyquist Plugin Installer*. After the installation the plugin will appear in the menu specific for its type ready to use [48].

3.8.5. Conclusions

The creation of the plugin requires the use of a less common language called *Nyquist*. Although it is possible to write a plugin in under ten lines of code, it would be a barrier for users without a programming background.

A plugin submission can be very easily done, since there is no central place that gathers all the available plugins. To have a plugin listed in the website, it would have to pass a review done by the community. This decentralization in the distribution makes it a little harder for a user to find a plugin.

The *Nyquist Plugin Installer* turns the installation process very easy, but it is not as easy as other applications that let the user search and install from the store page of the plugin. The usage functions is very similar to other applications, letting the plugin available in a menu that is separated by category, making it easy to be found by the user.

3.9. *GIMP* (GNU Image Manipulation Program)

3.9.1. Introduction

GIMP is a cross-platform image editor. It is possible to get its source code and modify creating unique variations of the application, since it is an open-source tool. The editor delivers common tools available in other applications of the same type. Selection Tools like *Lasso* and *Magic Wand*, Paint tools like *Brushes* and *Gradient* and also basic Transformation tools like *Crop*, *Align* and *Scale*, are features present in other famous editors on the market [50].

3.9.2. Plugin Creation Process

GIMP has a main interface that connect a plugin to the application core called *PDB (Procedural Database)*. It manages all the communication between those two parts. All the code is written in *C*. To build a plugin is necessary to use the *libgimp* library and a *CLI* tool called *gimptool*. A plugin generally serves for one of the following three purposes:

- Modify image data: receive image, modify and return the modified version.
- Generate an image.
- Process image without modifying.

Basically a plugin has four lifecycle functions. The *init()* and *quit()* functions are called respectively when *GIMP* application is open and when it is going to close. The *query()* function is called every time a plugin changes. The *run()* function is where the plugin performs its desired task. It is called when the plugin is asked to run.

The example below demonstrates a simple plugin that only open a window containing a message. In Figure 45, inside the *query()* function, the variable of type *GimpParamDef* holds the type,

name and a description of the parameter. After that the function *gimp_install_procedure()* is called to declare some information of the procedure like its name and description. In the same function is worth noticing that *GIMP_PLUGIN* is a constant that declares that the procedure to be external, that means that it will be not executed inside the *GIMP* core, therefore acting as a plugin. The last function called registers the plugin in the top Menu of the application, specifying the path *Filters/Misc* as shown in Figure 46.

```
static void query (void)
       {
         static GimpParamDef args[] = {
           {
             GIMP_PDB_INT32,
             "run-mode",
             "Run mode"
           },
           {
             GIMP_PDB_IMAGE,
             "image",
             "Input image"
           },
           {
             GIMP_PDB_DRAWABLE,
             "drawable",
             "Input drawable"
           }
         };

         gimp_install_procedure (
           "plug-in-hello",
           "Hello, world!",
           "Displays \"Hello, world!\" in a dialog",
           "David Neary",
           "Copyright David Neary",
           "2004",
           "_Hello world...",
           "RGB*, GRAY*",
           GIMP_PLUGIN,
           G_N_ELEMENTS (args), 0,
           args, NULL);

           gimp_plugin_menu_register ("plug-in-hello",
                                      "/Filters/Misc");
       }
```

Figure 45 - *query()* function (Source: [51])

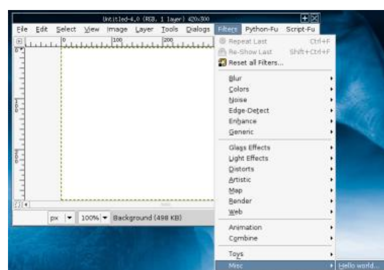

Figure 46 - Plugin path in the application Menu. (Source: [51])

The next necessary function is *run()*, shown in Figure 47. The plugin can run in one of the three available modes, called *run_mode* declared by the following constants: *GIMP_RUN_INTERACTIVE*, *GIMP_RUN_NONINTERACTIVE* and *GIMP_RUN_WITH_LAST_VALS*. The first one let the user use an options dialog opened, when the plugin is selected from the Menu, to configure the necessary parameters before running the plugin effectively. The second and the third do not open any dialog window and run directly from a script or a batch. The result of calling this simple plugin is shown in Figure 48 [51].

```
static void run (const gchar      *name,
          gint              nparams,
          const GimpParam  *param,
          gint             *nreturn_vals,
          GimpParam       **return_vals)
     {
       static GimpParam  values[1];
       GimpPDBStatusType status = GIMP_PDB_SUCCESS;
       GimpRunMode       run_mode;

       *nreturn_vals = 1;
       *return_vals  = values;

       values[0].type = GIMP_PDB_STATUS;
       values[0].data.d_status = status;

       run_mode = param[0].data.d_int32;

       if (run_mode != GIMP_RUN_NONINTERACTIVE)
         g_message("Hello, world!\n");
     }
```

Figure 47 - *run()* function (Source: [51])

Figure 48 - Plugin that opens a dialog box showing a message. (Source: [51])

3.9.3. Plugin Submission Process

GIMP does not have a repository that centralizes all available plugins. Therefore there is no need of submission. To share a plugin with the community the author just have to make it available for download anywhere on the internet.

3.9.4. Installation and Usage Process

As *GIMP* does not have a central repository, it is necessary to rely on website lists of some plugins and search engines to find them. Once the desired plugin is found, is necessary to check the form it is distributed. Some plugins come with an installer, making the installing process free of any necessity

of knowledge about using a *CLI* or any programming language. Others require the user to copy the downloaded script to the plugins folder of *GIMP*. If the script is in a programming language other than *C*, the user has to make sure that the support for that language is already installed in *GIMP*.

The usage of each plugin depends on its *run_mode* and where the author chose to register it on the application Menu. Some plugins may open an options dialog to provide some additional configuration to the user before effectively running the script and others run directly when selected in the Menu [51].

3.9.5. Conclusions

GIMP has a great barrier for authors without any programming background. It requires knowledge on *C* programming language. Since it is an image editor, it also requires additional knowledge about computer graphics to navigate through its documentation. The plugin submission process is inexistent, given that there is no central repository. The installation has no standard, it depends on the plugin provider. Some can provide an installer to facilitate the process, but others may require an *IT* background what creates a barrier for the majority of users.

3.10. Shopify

3.10.1. Introduction

Shopify is an online ecommerce platform with more than 800k stores built with it. It delivers a complete solution for building a store online very fast and without prior programming background. *Shopify* also is an extensible application, allowing developers to add features that help every aspect of a store like marketing, finance, design, shipping and many more. It is an interesting tool to analyze because it is aimed to users without any programming background. [53]

3.10.2. Plugin Creation Process

This section is going to focus on a specific type of plugin that is embedded on the *Shopify Admin* interface. *Shopify Admin* is the interface that store owners use to control all of the operation, see statistics, add products, manage finance among other things.

Shopify delivers a product component library called *Polaris* that help developers to make the plugin integrate seamlessly into *Shopify Admin* interface. *Polaris* is a library of components built to use with *React*, a *Javascript* library for user interfaces development. There is no template specific to a *Shopify* plugin provided, only a general scaffolding of a *React* application generated by the use of *Next.js*. A simple page can be created just by adding a *React* component to *index.js* file as shown in Figure 49.

```
const Index = () => (
 <div>
   <p>Sample app using React and Next.js</p>
 </div>
);

export default Index;
```

Figure 49 - *React* component that renders a *HTML div* tag with a paragraph containing a message. (Source: [54])

To run this simple plugin inside *Shopify Admin* it is necessary to expose the environment to a public address. Figure 50 shows the setup to use a tool called *ngrok*, a tool that is able to expose a local server securely to the public internet.

```
ngrok http 3000
```

Figure 50 - *ngrok* command to create a public interface using port 3000 (Source: [54])

To use the plugin exposed by the public interface created, the developer must create a new application inside its *Shopify* developers account. The developer must provide an application name, the public *URL* and whitelisted redirection *URL's* provided by *ngrok*. This creation process will create a *Shopify API key* and a *Shopify API secret key.* It is recommended to store them in the *.env* and *process.env* to make them available.

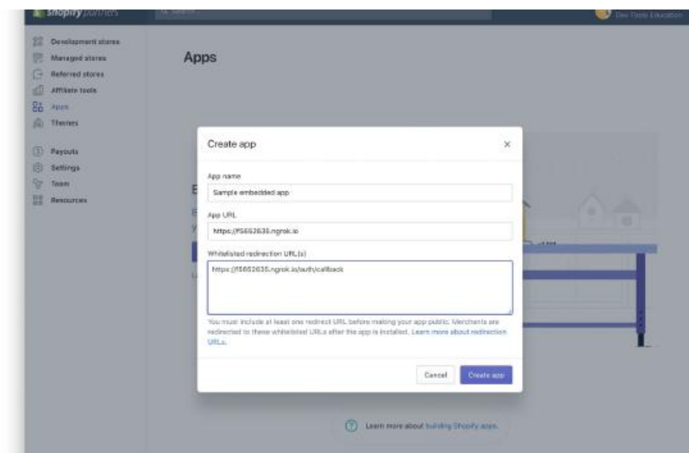

Figure 51 - Form for plugin creation (Source: [54])

It is also necessary to use an authentication system before creating an app as shown in Figure 51. In this example will be used *OAuth* setup in a *Node.js* server that is going to run on port 3000. After this everything is setup to authenticate and test the plugin inside *Shopify Admin*.

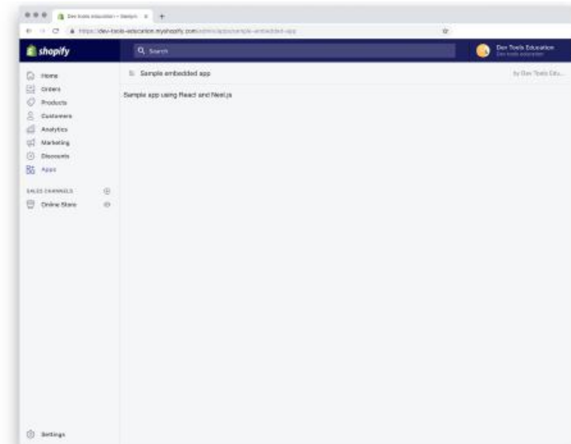

Figure 52 - Example plugin running on the *Development Store*. (Source: [54])

Figure 51 shows a plugin that only displays a message on the screen inside the *Shopify Admin*. For the plugin to make use of *Shopify UI* components is used a the *Polaris* library, as shown in Figure 53.

```
import { EmptyState, Layout, Page, ResourcePicker } from '@shopify/polaris';

const img = 'https://cdn.shopify.com/s/files/1/0757/9955/files/empty-state.svg';

class Index extends React.Component {
 state = { open: false };
 render() {
   return (
     <Page
       primaryAction={{
         content: 'Select products',
       }}
     >
       <ResourcePicker
         resourceType="Product"
         showVariants={false}
         open={this.state.open}
         onSelection={(resources) => this.handleSelection(resources)}
         onCancel={() => this.setState({ open: false })}
       />
       <Layout>
         <EmptyState
           heading="Select products to start"
           action={{
             content: 'Select products',
             onAction: () => this.setState({ open: true }),
           }}
           image={img}
         >
           <p>Select products and change their price temporarily</p>
         </EmptyState>
       </Layout>
     </Page >
   );
 }
 handleSelection = (resources) => {
   this.setState({ open: false })
   console.log(resources)
 };
}

export default Index;
```

Figure 53 - *index.js* using *UI* components from Polaris. (Source: [54])

The code in Figure 53 uses an *EmptyState* component that displays a view with explanations on how to start using the plugin. This component contains a button that when clicked opens a *Polaris* component called *ResourcePicker*, resulting in the screen on Figure 53.

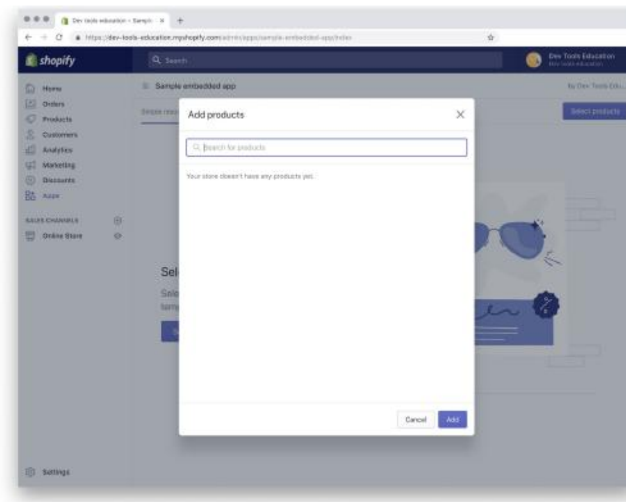

Figure 54 - Example plugin with the *ResourcePicker* component from *Polaris*. (Source: [54])

3.10.3. Plugin Submission Process

The submission is done through *Shopify*'s platform. As shown in section 2.10.4 to test plugin the developer must create it inside its developer account, and this plugin created can be updated and published. The developer must be logged in to create a new *Shopify App Store listing*. To create the listing submission, the developer must fill out a form providing a description highlighting the features of the plugin, chose a category, set pricing information and specify which merchants can use the plugin.

3.10.4. Installation and Usage Process

The installation is done directly from *Shopify App Store*. The store homepage lists plugins dividing by category and popularity. The user can also search by name, keyword or category using the search bar. (See Figure 55)

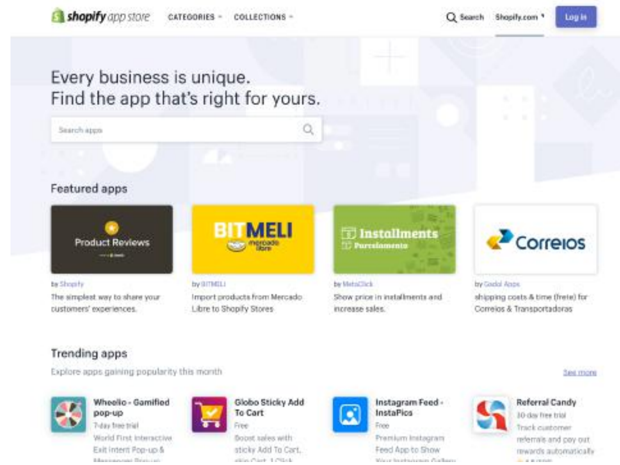

Figure 55 - Shopify App Store homepage

Figure 56 shows a plugin page that provides a description and screenshots of the app. It also displays information about pricing for paid plugins. To add a plugin the user must only click the Add button.

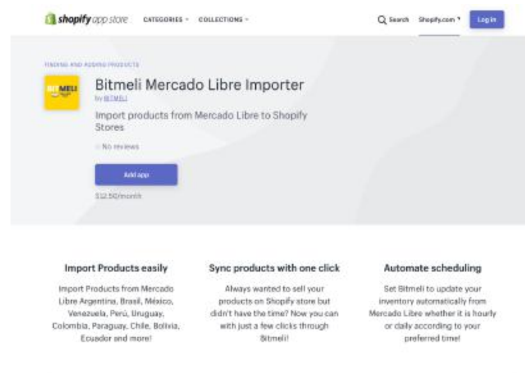

Figure 56 - Bitmeli Mercado Libre Importer plugin page

3.10.5. Conclusions

The creation process requires the knowledge of the three main areas of software development: front-end using *React*, back-end using *Node.js* to build a server and a little bit of *devops* in the use of *ngrok* to expose the local server to the public internet. Even users with a development background can have difficulties in the development if their programming background does not cover front-end reactive programming and back-end.

The submission process is done through *Shopify's* platform and does not require any specific *CLI* tool, which is good for users without a programming background. The installation is done directly from the *Shopify*'s *App Store* and the usage depends on the type of plugin.

3.11. Conclusions

As seen in the analysis in the previous sections and in Table XX, some common features emerge. All tools require some level of programming background, which is a requirement that can left out some users without this background. All tools that have a central repository require the developer to register in the platform. The majority of the tools have a central repository to store the plugins, allowing the users to search and discover new ones. For any developer to submit a plugin, it is required the filling out of a form with some general information about the extension and about the author. The developer is able to see the submitted plugins and change some information, either requiring the submission of a new version or changing it directly. The installation can be done directly from the repository, if the tool provides one, or it can require the user to download the package and load it manually into the tool.

Table 1 – Tools comparison result

	Visual Studio Code	**Googl e Docs**	**Sketchup**	**StarUML**	**Google Chrome**	**Wordpress**	**Eclipse**	**Audacity**	**GIMP**	**Shopify**
Plugin development process without programming background required	No	No	No	No	No	No	No	No	No	No
Has central repository	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Easy submission process through graphical interface	Yes	Not so much	Yes	Yes	Not so much	Yes	Yes	No	No	Yes
Easy installation through graphical interface	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	It depends on the plugin	Yes

For *piStar - Plugin Submission Tool* these common features were chosen. Some other features that appeared in only one or a few tools are going to be listed in Chapter 5 as suggestions for future development.

4. p*iStar - Plugin Submission Tool*

4.1. Introduction

The project of a plugin submission application for *piStar Tool* was planned to provide the features given by state of the art applications like the ones analysed in Chapter 3. pi*Star - Plugin Submission Tool* that has the goal to make *piStar Tool* easier to extend, allowing any *i** extension developer to make its extension available for all the community in a centralized way.

The developers of an extension must have an account in *piStar - Plugin Submission Tool* to send new plugins. To send a new plugin the developer is required to fill out a form with the following information:

- Plugin name
- Keywords - one of the ways of representing the plugin subject.
- Authors - emails used to register in the platform of each author.
- Short description - the plugin in a sentence.
- Long description - a longer description that describes to the user each functionality of the extension and its rules.
- References - books/articles/websites where the theoretical work present in the plugin can be found.
- Homepage link - link to plugin homepage if it exists.
- Category - category that the plugin fits in [6].
- Plugin package - *ZIP* file containing the package with a folder structure following the rules described in Chapter 2.

The developers can see a list of submitted plugins that he is cited as an author. An author can also delete a submission to make the plugin unavailable.

The following sections are going to start describing the application requirements through *User Roles* and *User Stories*. After that, the architecture and technologies used in the development are going to be explained to make it more accessible for any contributors that might want to build more features into the application.

4.2. User roles

Table 2 – User Roles

Administrator	**Plugin Developer**	***API***
It is the *Administrator* of the system and needs additional privileges. An *Administrator* has the power to *Edit* any information of any plugin submitted or even *Delete* the whole plugin.	It is divided in two types of users: the ones with programming background and the ones without any knowledge in this area. It helps to make the tool comfortable to use to those without any programming background.	It is the *API* of the system and is responsible for making the data collected available to be used in the client.

4.3. User Stories

As seen in the section 4.2, users were divided in three categories: *Administrator*, *Plugin Developer* and *API*. The list below is going to describe their *User Stories*. First is going to be listed the common *User Stories* for all plugin developers and administrators and then the stories of the *API*.

Table 3 – Common User Stories

Common User Stories for Plugin Developers and Administrators
1.Register: As an **USER** I want to be able to **create a new account** so that I can access the system and perform the actions related to my role.
2.Login: As an **USER** I want to be able to **login** in the system so that I can access the system and perform the actions related to my role.
3.Logout: As an **USER** I want to be able to **logout from my account** so that another user can login from my machine.
4.Reset Password: As an **USER** I want to be able to **reset my password** so that I can change my password.
5.Submit plugins: As an **USER** I want to be able to ***submit plugins easily*** so that I can share my plugins without any contact with developer specific tools that I am not comfortable to use.
6.Edit plugins: As an **USER** I want to be able to ***edit information about my plugins*** so that I can correct any possible mistakes.
7.List of plugins: As an **USER** I want to be able to **see a list of my plugins** so that I can perform actions related to plugins according to my role.

Table 4 – Administrator User Stories

Administrator User Stories
1. List of all plugins: As an **ADMINISTRATOR** I want to be able to **see a list of all submitted plugins** so that I can perform the actions according to my role.
2.Reset password of any user: As an **ADMINISTRATOR** I want to be able to **reset the password of any user** so that I can solve any user related problems.
3.Edit submitted plugins data: As an **ADMINISTRATOR** I want to be able to ***edit any information of a submitted plugin*** so that I can correct any possible errors or add any missing data.
4.Turn any user into admin: As an **ADMINISTRATOR** I want to be able to **turn any user into an *Admin*** so that I any chosen by me can have the same privileges I have.
5.Change status of submitted plugins: As an **ADMINISTRATOR** I want to be able to ***change the status of any submitted plugin*** so that I can mark it as *published* or *unpublished* when necessary.

7.Search all users: As an **ADMINISTRATOR** I want to be able to ***search by email any registered users*** so that I can find the user I need to perform any action according to my role.
8.Edit user's data: As an **ADMINISTRATOR** I want to be able to ***edit any information about any user*** so that I can correct any possible errors in registration upon request.
9.Delete users: As an **ADMINISTRATOR** I want to be able to ***delete any user*** so that I can remove any user that asks to be removed or have to removed for violating any *Usage Policy*.

Table 5 – *API* User Stories

***API* User Stories**
1. List of all plugins: As an **API** I want to be able to **retrieve a list of all published plugins** so that I can fulfill the client's request.
2.Get data of specific plugin: As an **API** I want to be able to **get an specific plugin** so that I can retrieve all the metadata associated to it to the client who made the requisition.
3.Get rating/review data of specific plugin: As an **API** I want to be able to **get an rating/review information of an specified plugin** so that I can retrieve all this data to the client who made the requisition.

4.4. Architecture

4.4.1. *Introduction*

In Figure 57 is shown an overview of the submission tool operation. *piStar – Plugin Submission Tool* has two responsibilities: to receive the submission of new plugins sent by the *Plugin Developer* and also to make available to the client, *piStar Tool,* all developers and plugins data through the *API* acting as a central repository. A user of *piStar Tool* needs to be able to see a list of all available plugins and to load the elements specific to one of them. *piStar Tool* requests a list of the available plugins to the *API* and it receives a response containing a list with ids and names of all available plugins. If a user of *piStar Tool* selects one of the plugins in the list, *piStar Tool* makes a request to the *API* passing the id of the desired plugin and it receives a response with all the plugin data.

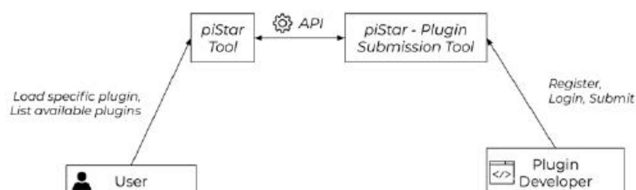

Figure 57 – *piStar Plugin Submission* Tool requests operation.

A *Plugin Developer* must be able to submit a plugin and see a list of submitted plugins. To achieve the first task, it must fill out a form containing some information about the plugin being submitted and the authors. The information required in the form is listed in section 4.1 and the Figure 79 shows how the implementation of the form. Figure 78 shows the dashboard that lists all submitted plugins by the developer.

The controller for *piStar - Submission Tool* was built using *Node.js*. *Node.js* is a *Javascript* runtime used to build network applications. For a better application structure, *MVC* design pattern was used. One of the main advantages of *MVC* pattern is the separation of concerns, because it makes the application structured with each part of it responsible for a specific task. A common problem that happens with *MVC* is to have a bloated *Controller*, that concentrates everything that is not a *Model* or a *View*. Since *piStar - Submission Tool* is a relatively small application, given that it has only one main purpose, that is to submit new plugins, the *MVC* possible problems do not appear [55].

Figure 58 - *MVC* structure of *piStar - Plugin Submission Tool*.

The code of *piStar - Plugin Submission Tool* is structured following *MVC* principles. In the controller's folder there are 4 of them. Each controller was created as a *Javascript* class and are instantiated as needed in the *routes.js* file. The responsibility of each controller is shown in Table 6.

Table 6 – *List of controllers*

Dashboard Controller	**Session Controller**	**Submission Controller**	**User Controller**
Controls the main page of the application. In Dashboard the page shows a list of plugins submitted by the user that is logged in, a button to make a new plugin submission and a button for the user to logout of the account.	Controls the session of a logged in user. Is used in the Login page to create a new session with the user credentials informed by the user.	Controls the submission of a new plugin. In the Submission page is showed a form with informations about the authors and the new plugin that the plugin developer must fill out.	Controls the registration of a user. Is used in the Signup page when a registration is completed.

In models folder there are six files describing a model for each table in the database as shown in Figure 59. An example of the code of each file is shown in Figure 69. The views folder contains mainly four files describing the layout of the pages that are controlled by the four controllers listed above.

4.5. Database

4.5.1. Overview

In this section is going to be presented some topics about the database used in the tool. First is *schemas* used in the database. Then is going to be presented a brief introduction of the two databases used to build the tool: *Redis* and *Postgres*. The image representation of each table of the *schema* was built using the online tool *dbdiagram.io* [56]. In Figure XX is shown an diagram of all the database tables and relationships between them represented by the blue lines connecting the tables. The number 1 or the symbol * appear representing the *one-to-many* relationship. As an example, a *user* can have many entries in *users_plugins* table. The *many-to-many* relationship is illustrated by the tables that carry the names of two tables that it uses as a foreign reference. As an example, a *user* can be the author of many *plugins*, as well as a *plugin* can have many users as authors. That relationship is made using the *users_plugins* table. The same occurs with the *plugins_keywords* table.

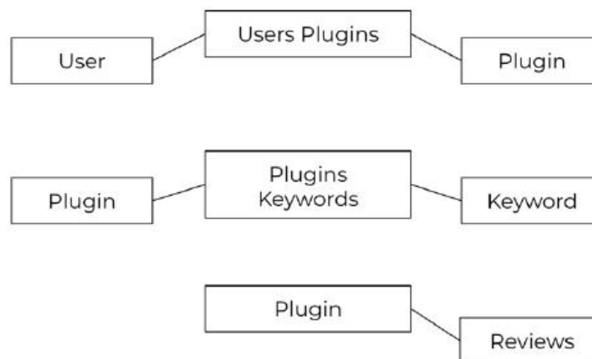

Figure 59 - Diagram illustrating all *piStar - Plugin Submission Tool* database tables relationships.

4.5.2. Schema

It is necessary to make a few explanations about the notation used by the tool that generated the diagrams in the following sections. The left column shows the name of the table column and the right column shows the type of the data stored in the respective column. Where it shows *SERIAL* as a type, it means that the data stored will be an integer the increments automatically by one. Where it shows *character* as a type, it means that the data stored will be a *string*. Where it shows *timestamp* as a type, it means that the data stored will be a time in the *ISO 8601* format. All tables have three fields in common: *id*, *created_at* and *updated_at*. All of them are created automatically by *Postgres*.

4.5.2.1. Plugins

plugins	
id	SERIAL
name	character
short_description	character
long_description	character
homepage_link	character
category	character
status	character
use_count	integer
reference	character
svgs	character
constraints	character
metamodel	character
shapes	character
uimetamodel	character
created_at	timestamp
updated_at	timestamp

Figure 60 - *Plugins* table schema

The *Plugins* table, shown in Figure 60, contain the fields required in the submission form and some additional ones. In the submission form the user provides the name, a short description, a long description, a homepage link, a category for the plugin and its references. The authors and keywords are stored using another table to describe a *many-to-many* relationship as explained in section 4.5.1. Inside the *Plugin* table is also stored the data of the plugin package uploaded by the user. The package is a *ZIP* file of the language folder structured as shown in section 2.2. Each field is named after the file that it stores. The only exception is the *svgs* field that stores all the SVG files contained in the folder *images*.

4.5.2.2. Users

users	
id	SERIAL
full_name	character
institution_name	character
role	character
password_hash	character
email	character
is_business	boolean
is_university	boolean
created_at	timestamp
updated_at	timestamp
is_admin	boolean

Figure 61 - *Users* table schema

The *Users* table, shown in Figure 61, contains exactly the fields required in the form showed in *Signup* page plus the three fields automatically added by *Postgres*. The *full_name, institution_name, role, password_hash* and *email* fields are all stored as strings. The *is_business, is_university* and *is_admin* are booleans respectively representing if the user is from a business or a university, with the possibility of being both and if the user is an administrator. It is worth noticing that the field *password_hash* stores an encrypted version of password typed by the user in the form.

4.5.2.3. UsersPlugins

users_plugins	
id	SERIAL
date	timestamp
user_id	integer
plugin_id	integer
created_at	timestamp
updated_at	timestamp

Figure 62 - *UsersPlugins* table schema

The *UsersPlugins,* shown in Figure 62, is a table that allows the *many-to-many* relationship between the tables *Users* and *Plugins*. Beyond the automatically added fields, it only contains two fields to store the two foreign keys from the two tables in the relationship.

4.5.2.4. Keywords

keywords	
id	SERIAL
title	character
created_at	timestamp
updated_at	timestamp

Figure 63 - *Keywords* table schema

The *Keywords* table*,* shown in Figure 63, only have one column named *title*. When the user make a new submission, is passed a list of keywords separated by commas. Each of these keywords are stored in a new entry in the table inside the column *title*.

4.5.2.5. PluginsKeywords

plugins_keywords	
id	SERIAL
keyword_id	integer
plugin_id	integer
created_at	timestamp
updated_at	timestamp

Figure 64 - *PluginsKeywords* table schema

The *Keywords* table*,* shown in Figure 64, is another table that allows the *many-to-many* relationship. In this case it is between the tables *Plugins* and *Keywords*. It also only contains two fields to store the two foreign keys from the two tables in the relationship, beyond the automatically added

fields. This relationship is going to be useful with a feature that allows the users to search for plugins by keyword.

4.5.2.6. Reviews

reviews	
id	SERIAL
title	character
description	character
rating	integer
plugin_id	integer
created_at	timestamp
updated_at	timestamp

Figure 65 - *Reviews* table schema

The *Reviews* table, shown in Figure 65, stores a *title, a description* and a *rating* to a given plugin identified by *plugin_id* fields. This table has a *one-to-many* relationship with the *Plugins* table, meaning that a review can belong to only one plugin.

4.5.3. Redis

Redis is an in-memory data structure store used as a database. Because it is an in-memory data structure, it has a better performance than other databases. Generally, it is used for data that needs high availability and does not require much space. There are two main scenarios that *Redis* is recommended: Cache storage and Session storage. That is why it was chosen to handle the login sessions inside *piStar - Plugin Submission Tool.* With *Redis* is possible to implement "*sticky sessions*" across many different servers. It allows for a user to preserve its session, while inside the *TTL (Time To Leave)* timebox, even if the connection is made with another server different from the first one. In *piStar - Submission Tool Redis* is used for session storage. The setup only require a few lines of code using two main dependencies: *express-session* and *connect-redis* [57][58][59].

Figure 66 shows the creation of instances of both packages. The object *session* is used to initialize the *RedisStore* object. The configuration of *Redis Store* is done in the *middleware* that is called when the application initializes. Figure 67 shows the configuration of new store with *RedisStore*. It takes three parameters: the host and the port where the *Redis* server is running and optionally takes the Time To Leave parameter that sets a time in milliseconds to expire the session.

```
const session = require('express-session')
const RedisStore = require('connect-redis')(session)
```

Figure 66 - Creation of instances from *express-session* and *connect-redis* packages

```
this.express.use(
    session({
      store: new RedisStore({
        host: 'localhost',
        port: 6379,
        ttl: 1800
      }),
      name: 'root',
      secret: 'piStarPluginSecret',
      resave: true,
      saveUninitialized: true
    })
  )
```

Figure 67 - Session configuration

4.5.4. Postgres

Postgres is an open source object-relational database. It is a project with more than 30 years of development, making it a reliable choice for a database solution. Beyond the fact that is a established technology with decades of development, *Postgres* was chosen for being a relational database, ideal for applications with models with many relations.

Postgres extends *SQL* with features that provide safety, reliability, concurrency, extensibility and more. To use *Postgres with Node.js*, *piStar - Plugin Submission Tool* use *Sequelize*, an *ORM (Object-Relational Mapper)* that has the task to map the objects (or models) in *Javascript* to the tables in the database [60][61]. Before start building any models it is necessary to declare the information to access the database. (See Figure 68)

```
module.exports = {
 dialect: 'postgres',
 host: '127.0.0.1',
 username: 'docker',
 password: 'docker',
 database: 'piStarPluginSub',
 operatorAliases: false,
 define: {
   timestamps: true,
   underscored: true,
   underscoredAll: true
 }
}
```

Figure 68 - Database configuration

The first step is to build an object using *Sequelize* like Figure 69 shows. *Sequelize* adds a *createdAt, updatedAt* and an unique *id* automatically, so it is not necessary to declare it explicitly.

```
module.exports = (sequelize, DataTypes) => {
 const Plugin = sequelize.define('Plugin', {
   name: DataTypes.STRING,
   short_description: DataTypes.STRING,
   long_description: DataTypes.STRING,
   homepage_link: DataTypes.STRING,
   category: DataTypes.STRING,
   use_count: DataTypes.INTEGER
   status: DataTypes.STRING,
   reference: DataTypes.STRING,
   svgs: DataTypes.STRING,
   constraints: DataTypes.STRING,
   metamodel: DataTypes.STRING,
   shapes: DataTypes.STRING,
   uimetamodel: DataTypes.STRING
 })

 return Plugin
}
```

Figure 69 - *Plugin* mapping with *Sequelize*

To track changes in the database it is used *migrations*. It helps to revert the changes if needed at any time. Figure 70 show a *migration* file for the *Plugin* table.

```
'use strict'

module.exports = {
 up: (queryInterface, Sequelize) => {
   return queryInterface.createTable('plugins', {
     id: {
       allowNull: false,
       autoIncrement: true,
       primaryKey: true,
       type: Sequelize.INTEGER
     },
     name: {
       type: Sequelize.STRING,
       allowNull: false
     },
     short_description: {
       type: Sequelize.STRING,
       allowNull: false
     },
     long_description: {
       type: Sequelize.STRING,
       allowNull: false
     },
     homepage_link: {
       type: Sequelize.STRING,
       allowNull: false
     },
     category: {
       allowNull: false,
       type: Sequelize.STRING
     },
     status: {
       allowNull: false,
       type: Sequelize.STRING,
       default: 'Not Published'
     }, …
```

Figure 70 - *Plugin* migration file

4.6. Views

4.6.1. Nunjucks

Nunjucks is a *Javascript* templating language that enhances the capabilities of a simple *HTML* file. It add features like the use of variables, filters, template inheritance, conditionals, synchronous and asynchronous loops and more. The most used features used in *piStar - Submission Tool* are exemplified in the figures below. Figure 71 shows an example where an *HTML h1* tag displays a variable in the context of a given template.

```
<div class="col s12">
   <h1>Hello, {{ user.full_name }}</h1>
   <h4>Here are your submitted plugins</h4>
 </div>
```

Figure 71 - Value of a key *name* inside an *user* object displayed inside a *h1* tag

Template inheritance helps in the organization of code, allowing the developer write code in separate files. In Figure 72 a new *HTML* file is extending another one previous created.

```
{% extends "_layouts/default.njk" %}
<div class="col s12">
   <h1>Hello, {{ user.full_name }}</h1>
   <h4>Here are your submitted plugins</h4>
 </div>
```

Figure 72 - *default.njk* is extended by adding a the main title in *dashboard.njk*.

To help the developer write less code by reusing it, an *include* block is used. In Figure 73, a *HTML* file starts by using the contents of another file.

```
{% include "header.html" %}
<body>
<div>
<h1>My page</h1>
</div>
</body>
```

Figure 73 - *header.html* is included in another file

4.6.2. Materialize CSS

Materialize CSS is a *CSS* framework based on *Material Design*, a visual language built by *Google*. It brings a variety of stylized web elements. Besides the use simplicity, a great advantage is that this visual language is spread all over the web. That brings a important sense of familiarity with the elements of the interface and a intuitive use. Figure 74 shows an example of a submit button that contains an icon. The use of the styles from *Materialize CSS* framework are done through the *class* attribute. The result is shown in Figure 75.

```
<button class="btn waves-effect waves-light" type="submit" name="action">Submit
    <i class="material-icons right">send</i>
  </button>
```

Figure 74 - *Button* declaration using *Materialize CSS* styles

Figure 75 - *Materialize CSS* button with icon.

4.7. Screenshots

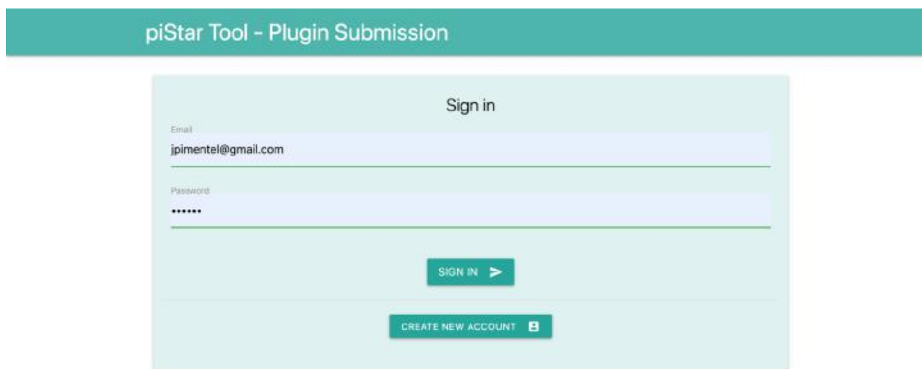

Figure 76 - *Login page*

On *Login* page, the user can enter its credentials to login in the tool or, if not registered yet, solve this by clicking in the *Create New Account* button. (See Figure 76)

To create a new account the user is going to be sent to the *Signup* page. The registration form will require the user's full name, the type of institution it comes from, the institution name, its role, department, email and a password. (See Figure 77)

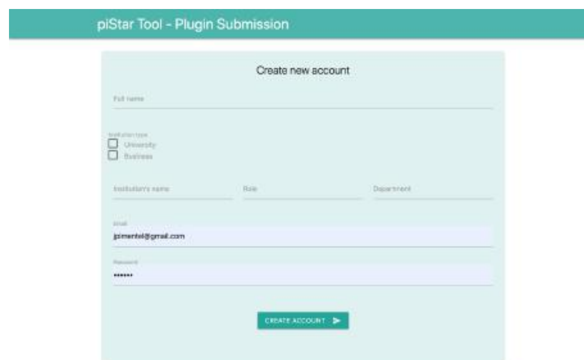

Figure 77 - *Signup page*

Once registered, the user is going to be redirected to the *Login* page so it can login. If the credentials are valid, the *Dashboard* page is going to be loaded. It will display a list of all the plugins submitted by the user or an empty list if the user has not send any yet. (See Figure 78)

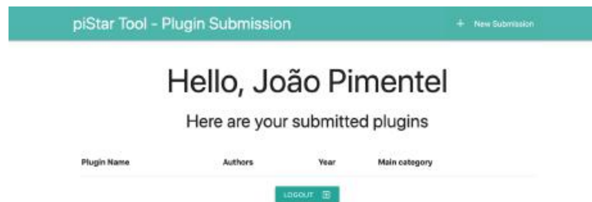

Figure 78 - *Dashboard page*

Figure 79 - *Submission form page*

To submit a new plugin, the user can click on the + *New Submission* button at the top in the right side of the *Dashboard* page. After that, the *Submission* page will be displayed. The form shown in Figure 79 asks some information about the plugin and the authors. It is required a plugin name, some keywords separated by commas, the emails of the authors of the plugin, a short and a long description, references related to the plugin, a homepage link, a category based on the list in [6] and the plugin package that the user can choose from the disk. It is worth to notice that all the authors must be registered in the *piStar - Plugin Submission Tool* and provide the same emails used to create the account when submitting a new plugin.

5. Conclusion

5.1. Results

This work started with an overview of the plugin structure of *piStar Tool* in section 2.2 and with the motivation of taking its extensibility to another level, providing a tool through which plugin developers could submit and share their plugins with the community. To achieve that, an analysis of ten popular extensible tools in different fields of knowledge was made. The analysis captured common requirements in three different perspectives: the plugin creation process, the submission process and the installation and usage process. Some of these common requirements were chosen to be part of *piStar - Plugin Submission Tool*. After that, the solution was presented, beginning with a brief overview of the tool and going through its requirements using *User Stories*.

As *piStar - Plugin Submission Tool* is an open-source tool, and therefore with the possibility of contributions made by third-parties, it was made necessary to make an overview on the architecture and on the technologies used to build the tool. This overview made in section 4.4 begins describing the *MVC* pattern used by *Node.js* and how the project directory structure reflects that. After that, the database modelling and technologies were explained, listing each table, its relationships and how it was made in code using *Redis* and *Postgres*. The next section went through the technologies used to build the visual part of the application using *Nunjucks* and *Materialize CSS*. To finish the chapter, screenshots of all the pages of the application were presented with a use case description.

Until the conclusion of this work the following requirements listed in section 4.3 were delivered:

- Common to Administrator and Plugin Developers: 1, 2, 3, 5 and 7. (See Table 3)
- *API*: 1, 2 and 3. (See Table 5)

5.2. Contributions

An extensive tool can build a very large user base because of this particular feature. It is the case with the tools shown in Chapter 3. Although *piStar Tool* is already extensible, it required some effort to develop, test and share with the community new plugins. With *piStar - Plugin Submission Tool*, testing and sharing are made easier. A developer can make its language extension available to all the users of *piStar Tool*. This helps to get more feedback from users with different backgrounds, since it is not required to have a programming experience or to clone and modify the source code to use a plugin.

5.3. Future works

Beyond the requirements described in section 4.3, some requirements, either captured with the analysis made in Chapter 3 or captured by an intrinsic necessity of the tool and its user base, are going to be listed in this section as an inspiration for future work to develop even more *piStar - Plugin Submission Tool*.

5.3.1. Analysis of code repository tools

As done in Chapter 3 with the extensible tools, a code repository tools analysis would be beneficial to the improvement of *piStar – Plugin Submission Tool*. Code repositories are important not only to store source code, but they can collect useful information about the development process [4]. In the current level of implementation of the *piStar – Plugin Submission Tool* this type of information is

not tracked yet, mainly because there is any plugin version manager like described in the section 5.3.2 below.

5.3.2. Plugin version manager

To update an extension, it is required to delete the submitted version or to make a submission as if it is a new extension. It can lead to the unnecessary display of older versions, what can be confusing to the user. A plugin version manager could avoid that problem and help the developer to keep control of each version and also help to keep track of information about the development process.

5.3.3. Feedback system

A feedback can be provided to the developer by using its email or by visiting the homepage link of the plugin, if a support contact is provided. The necessity to search for a way to send feedback can make the developer loose valuable information that users could provide to help improve the extension. A review and rating system could help that by bringing allowing users to send feedback right on *piStar Tool* interface.

5.3.4. Plugin development code abstraction

piStar Tool has a heterogeneous user base, meaning that not all users have a programming background. Although that with a few changes on the plugin template, an extension can be made available to use with *piStar Tool*, it still requires some code editing. *piStar - Plugin Submission Tool* could allow developers to make those changes in a more abstract way, just filling out a form that could generate the plugin with the specified extension in the end.

6. References

[1] PRIKLADNICKI, RAFAEL "Problemas, Desafios e Abordagens do Processo de Desenvolvimento de Software." PUCRS.

[2] MCVEIGH, ANDREW "A Rigorous, Architectural Approach to Extensible Applications." Imperial College London - Department of Computing

[3] BIRSAN, DORIAN "On Plug-ins and Extensible Architectures." ACME Queue Volume 3, Issue 2, March 18th, 2005.

[4] Alberto Sillitti, Giancarlo Succi, Tullio Vernazza . Analysis of Source Code Repositories. DIST – Università di Genova, Libera Università di Bolzano.

[5] PIMENTEL, JOÃO "piStar Tool - A Pluggable Online Tool For Goal Modeling."

[6] piStar Tool repository. Available at: <https://github.com/jhcp/piStar>. Access date: June 19, 2019.

[7] GONÇALVES, ENYO "A Systematic Literature Review of iStar Extensions",

[8] iStar Extension Repository. Available at: <https://istarextensions.cin.ufpe.br/catalogue/publication/list?q=&f=92>. Access date: March 22, 2019.

[9] Visual Studio Code Repository. Available at: <https://code.visualstudio.com/>. Access date: May 20, 2019.

[10] Visual Studio Code: Your First Extension. Available at: <https://code.visualstudio.com/api/get-started/your-first-extension>. Access date: May 20, 2019.

[11] Visual Studio Code: Yeoman. Available at: <https://yeoman.io/>. Access date: May 20, 2019.

[12] Visual Studio Code: Publishing Extension. Available at: <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>. Access date: May 20, 2019.

[13] Visual Studio Code: Extension Anatomy. Available at: <https://code.visualstudio.com/api/get-started/extension-anatomy>. Access date: May 20, 2019.[14] Google Docs: About. Available at: <https://www.google.com/docs/about//>. Access date: May 22, 2019.

[15] Google Docs: Quickstart - Add-on for Google Docs. Available at: <https://developers.google.com/gsuite/add-ons/editors/docs/quickstart/translate>. Access date: May 22, 2019.

[16] Google Docs: Apps Script - Overview. Available at: <https://developers.google.com/apps-script/overview>. Access date: May 23, 2019.

[17] Google Docs: Triggers. Available at: <https://developers.google.com/apps-script/guides/triggers/>. Access date: May 23, 2019.

[18] Google Docs: Tutorial - Google Maps plugin - Overview. Available at: <https://www.labnol.org/internet/write-google-docs-addon/28446/>. Access date: May 23, 2019.

[19] Google Docs: Creating a version. Available at: <https://developers.google.com/apps-script/guides/versions#creating_a_version>. Access date: May 28, 2019.

[20] Google Docs: Publishing editor add-ons - Overview. Available at: <https://developers.google.com/gsuite/add-ons/how-tos/publishing-editor-addons>. Access date: May 23, 2019.

[21] Google Docs: oAuth consent screen. Available at: <https://developers.google.com/apps-script/guides/cloud-platform-projects#completing_the_oauth_consent_screen>. Access date: May 23, 2019.

[22] Sketchup. Available at: <https://www.sketchup.com/>. Access date: May 28, 2019.

[23] Sketchup: Extension Warehouse. Available at: <https://extensions.sketchup.com/>. Access date: May 28, 2019.

[24] Sketchup Ruby API. Available at: <https://ruby.sketchup.com/>. Access date: June 4, 2019.

[25] Sketchup: Hello Cube tutorial. Available at: <https://github.com/SketchUp/sketchup-ruby-api-tutorials/tree/master/tutorials/01_hello_cube>. Access date: June 5, 2019.

[26] Sketchup: Developer Center Guidelines. Available at: <https://extensions.sketchup.com/developer_center/ew_developer>. Access date: June 5, 2019.

[27] Sketchup: Installing plugins. Available at: <https://help.sketchup.com/en/installing-ruby-plugins-extensions>. Access date: June 4, 2019.

[28] Sketchup: Managing Extensions. Available at: <https://help.sketchup.com/en/extension-warehouse/managing-extensions>. Access date: June 18, 2019.

[29] StarUML. Available at: <http://staruml.io/>. Access date: June 14, 2019.

[30] StarUML: Getting Started. Available at: <https://docs.staruml.io/developing-extensions/getting-started>. Access date: June 14, 2019.

[31] StarUML: Getting Started: Marketplace Client. Available at: <https://docs.staruml.io/developing-extensions/getting-started>. Access date: June 14, 2019.

[32] Google Chrome: Extensions - Get Started. Available at: <https://developer.chrome.com/extensions/getstarted>. Access date: May 26, 2019.

[33] Google Chrome: Extensions - Overview. Available at: <https://developer.chrome.com/extensions/overview>. Access date: May 26, 2019.

[34] Google Chrome: User Interface. Available at: <https://developer.chrome.com/extensions/user_interface>. Access date: June 7, 2019.

[35] Google Chrome: Publish in the Chrome Web Store. Available at: <https://developer.chrome.com/webstore/publish>. Access date: June 10, 2019.

[36] Google Chrome: Install and manage extensions. Available at: <https://support.google.com/chrome_webstore/answer/2664769?hl=en>. Access date: June 10, 2019.

[37] Wordpress. Available at: <https://wordpress.org/>. Access date: May 20, 2019.

[38] Wordpress: Plugin Handbook. Available at: <https://developer.wordpress.org/plugins/javascript/>. Access date: May 20, 2019.

[39] Wordpress: Add your plugin. Available at: <https://wordpress.org/plugins/developers/add/>. Access date: June 12, 2019.

[40] Wordpress: Planning your Plugin. Available at: <https://developer.wordpress.org/plugins/wordpress-org/planning-your-plugin/>. Access date: June 18, 2019.

[41] Wordpress: Plugin Directory. Available at: <https://wordpress.org/plugins/>. Access date: May 20, 2019.

[42] Eclipse. Available at: <https://www.eclipse.org/eclipseide/>. Access date: June 14, 2019.

[43] Eclipse: *PDE* does Plugins. Available at: <https://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>. Access date: June 14, 2019.

[44] Eclipse: Marketplace Client. Available at: <https://www.eclipse.org/mpc/>. Access date: June 14, 2019.

[45] Eclipse: Marketplace Quickstart. Available at: <https://marketplace.eclipse.org/quickstart>. Access date: June 14, 2019.

[46] Audacity: Features. Available at: <https://www.audacityteam.org/about/features/>. Access date: June 10, 2019.

[47] Audacity: Nyquist - Overview. Available at: <https://manual.audacityteam.org/man/nyquist.html#program>. Access date: June 10, 2019.

[48] Audacity: Creating Nyquist plugins. Available at: <https://manual.audacityteam.org/man/creating_nyquist_plug_ins.html#Creating_Nyquist_Plug-ins>. Access date: June 10, 2019.

[49] Audacity: Conventions for Nyquist Plugins. Available at: <https://forum.audacityteam.org/viewtopic.php?f=39&t=42106>. Access date: June 10, 2019.

[50] GIMP. Available at: <https://www.gimp.org/>. Access date: June 6, 2019.

[51] GIMP: Installing plugins. Available at: <https://docs.gimp.org/2.10/en/gimp-scripting.html#gimp-plugins-install>. Access date: June 6, 2019.

[52] GIMP: Writing a plugin. Available at: <https://developer.gimp.org/writing-a-plug-in/1/index.html>. Access date: June 6, 2019.

[53] Shopify. Available at: <https://www.shopify.com/>. Access date: June 6, 2019.

[54] Shopify: Build a plugin with Node.js and React. Available at: <https://developers.shopify.com/tutorials/build-a-shopify-app-with-node-and-react>. Access date: June 6, 2019.

[55] About NodeJS. Available at: <https://nodejs.org/en/about/>. Access date: May 26, 2019.

[56] dbdiagram.io. Available at: <https://dbdiagram.io/home>. Access date: May 26, 2019.

[57] Redis. Available at: <https://redis.io>. Access date: May 27, 2019.

[58] Redis: connect-redis. Available at: <https://github.com/tj/connect-redis>. Access date: May 27, 2019.

[59] Redis: 15 reasons to use Redis as an Application Cache. Available at: <https://redislabs.com/wp-content/uploads/2016/03/15-Reasons-Caching-is-best-with-Redis-RedisLabs-1.pdf>. Access date: May 27, 2019.

[60] Postgres. Available at: <https://www.postgresql.org/>. Access date: May 27, 2019.

[61] Sequelize. Available at: <http://docs.sequelizejs.com/>. Access date: May 27, 2019.

[62] Nunjucks. Available at: <https://mozilla.github.io/nunjucks/>. Access date: May 27, 2019.

[63] Materialize CSS. Available at: <https://materializecss.com/>. Access date: May 27, 2019.

[64] Dalpiaz, F., Franch, X. and Horkoff, J. (2016) iStar 2.0 language guide. arXiv preprint arXiv:1605.07767.

[65] Eric, S. Y. (1995). Modelling strategic relationships for process reengineering. PhD Thesis. Department of Computer Science, University of Toronto.

[66] piStar Tool. Available at: <https://www.cin.ufpe.br/~jhcp/pistar/>. Access date: June 27, 2019

[67] Eric S. K Yu "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering" Faculty of Information Studies, University of Toronto Toronto, Ontario, Canada M5S 3G6