**Document Name: Bugzilla Summary**

**Author:** Alex Summerton

**Date:** Nov 18, 2022.

**Description:** This document sets forth a specification/record of a developmental revision of the “TMEIC Project Creation Utility Tool” used to create projects on the TMEIC Metals and MH Bugzilla servers.

**Bugzilla Summary:**

This document serves as a starting point for any person resuming development of the Bugzilla Project Creation Utility Tool from work paused in November of 2022.

Due to unresolved issues at the time, while the project was substantively completed, HTTP exchanges between the application and TMEIC Bugzilla servers could not be serviced due to what is presumed to be a server configuration. This document details the remaining work, as identified at the time, to complete the Project Creation Utility Tool.

**Background:**

Per management request, the previous used Bugzilla Project Creation Utility Tool is being updated. The original tool was written in C# utilizing string uploads based on common gateway interface format. This method requires an additional round of inputs from the user to authenticate the login token and test cookies.

To modernize the application, this project seeks to provide an updated Project Creation Utility tool using the Bugzilla REST API. Moreover, this updated tool will be designed to allow for authentication by either API Key, or by username and password, given the intent expressed in the Bugzilla documentation to depreciate less secure authentication methods in favor of the API Key method. See [§ 6.1.1.3 General -Authentication](https://tools.tmeic.com/mh/docs/en/html/api/core/v1/general.html#authentication). It is worth noting this method is also preferred, as at this time while login tokens can be returned from the server they cannot be used as they immediately expire. This tool will also allow for easier modification of the components included a newly created project, in the even a project contains non-standard components, or for a predefined project, the components list is required to be modified. Finally, the revised tool is designed to interface with both the MH and Metals servers from a single application, rather than being coded to a specific server for a specific application.

**Project Program Language:**

Based on a review of Bugzilla documentation, Bugzilla’s REST API was determined as the target to send procedure calls to 1) create a project and 2) update the project with. The application developed and referenced in this document is written using the Godot Editor, specially 3.5 stable version. While traditionally used for lightweight game development, the Godot Editor and gdot languages were selected as Godot allows for easy development of an interface top down (i.e., developing the GUI first), and more importantly, Godot applications can be designed with an included HTTP Request manager specifically designed to interact with REST APIs. Moreover, applications written in gdot can be easily exported to other platforms besides PC, including iOS and Android.

It is presumed a person with a small amount of programming experience who completed the Godot Tutorial projects (approximately an afternoon’s worth of research) should be able to complete development of this application once the server response issues are resolved. Familiarity with Python, Qt, and HTTP Networking protocol can all assist in understanding this material, though the subsequent explanation aims to provide sufficient document that that only a basic understanding of Godot and programming fundamentals is necessary to complete.

**Challenge:**

Server does not return objects or record behavior.

Due unresolved server issues and time constraints, this project was not completed prior to the end of the author’s tenure at TMEIC in Fall of 2022.

During testing and refinement of the application, it was discovered that the TMEIC Bugzilla MH (and presumptively Metals) server was not configured to respond to REST API calls. Initially, attempts to call the reach the server through the rest API endpoints such as “/rest/version” and “rest/bug/id” failed to return the expected JSON objects and instead produced a 404 error. Contact to IT resolved this issue to the following extent:

* GET requests submitted over x-www-form-urlencoded via browser submission now return the anticipated JSON object when an API Key or username and password are included. This is intended behavior.
* GET requests submitted over x-www-form-urlencoded via browser submission without an API Key nor username as password return a 500 Internal server error. It is unclear it if this intended behavior. Bugzilla REST API documentation indicates a JSON object indicating the error should be returned when an error occurs. See [§5.1.1.1 Basic Information – Errors](https://bugzilla.readthedocs.io/projects/harmony/en/latest/api/core/v1/general.html?highlight=error#basic-information).
* GET requests encoded as JSON submitted via Godot HTML Request module, with an API key or username and password do not return a JSON object as intended. A 500 Internal Server Error in x-www-form-urlencoded encoding is returned. This is not intended behavior. A JSON object should be returned containing the information of the request, provided the relevant fields of the request were properly encoded. *An API Key used in such a request is updated as used on the Bugzilla MH site at the time of the request, indicating the request is received by the server at least in part*.
* GET requests for the server version (/rest/version) do return a JSON object both over browser and via the Godot HTML Request mode. This is the only request that does not require an API Key or username and password. This is intended behavior.
* A GET request submitted over cURL (command line HTTP request program) uploading a JSON object, or x-www-form-urlencoded encoding the request, with or without an API Key or username and password will return a 500 Internal Server error message in x-www-form-urlencoded format. Moreover, if an API Key is used in the transaction, its “last used” time as indicated on the Bugzilla MH server is not updated to the time of the request, meaning it is unclear if the request reaches the server.
* As in the above cases, POST and PUT requests (those necessary to create and update projects) exhibit identical behaviors to requests other than to the /rest/version endpoint, but do not create any change on the server itself. That means a POST or PUT request to add a product, component or bug via Godot with API Key will return a 500 Internal Server Error x-www-form-urlencoded object, the API key will be updated to reflect the time of the request, and no change will occur on the Bugzilla Server. A similar request submitted over cURL returns the same object, but does not update the API Key. The request also does not work is submitted via url encoding as with a get request.

Presently, it is presumed a server configuration is not correct, and it prohibiting the server from properly servicing JSON encoded requests, though in the case of the Godot HTTP Request module, such requests are being parsed to the extent the API Key is flagged as used.

Without resolution of this issue, it is anticipated the application cannot be completed. Moreover, with the continual updates to the Bugzilla

**Features:**

The following features are discussed in relation to the original project, and should be viewed as additional options to the design to improve utility:

*Custom Predefined Components:*

A CSV file, “components.csv” is included with the project, and is meant to be kept in the location with the completed application and accessible externally without need to modify the app. This file contains a header row which should not be modified, and subsequent entries, starting with id 1, for each value should be filled. Each entry is meant to be a possible component to add to a TMEIC Project. The third value is the name of the component, as it will appear in the Project on the Bugzilla database, and the forth value is its description in the same manner. The second value of an entry should be an identifiable shortening of the component name, no longer than 8 characters, and compatible with the CSV format. During execution of the application, a person wishing to create a project with custom defined components will be presented with a list of component shortnames, where hovering over the shortname will present the full name, and a longer hover will replace the hover text of the full name with the component description.

*Presets:*

Coded into the application are radio buttons which provide a present selection of components. The exact component listing of a preset is determined by a CSV file “predef\_proj.CSV” This file is configured as a CSV where each row is a predefined project, with the first item in the row (0th index) being the short name of the project, and every subsequent entry being an id, draw from the components.CSV file, to be included in this project. If an ID is duplicated, or does not exist in the components CSV prior to loading the application, the duplicated or non-existance ID will not be added to the dictionary the application constructs to store the components of the predefined dictionary.

*Project Retrieval:*

In overwatch.gd, the project name is subdivided into three string fields, “project\_num”, “client\_code”, and “project\_name.” When creating a project, these three strings will be concatenated to construct the

Project numbers are meant to be exclusive, and while careful management of the value entered into the “project\_num” field can avoid accidental duplication of a project number, the “project\_num” field may be used to automate a comparison check to ensure no project has a duplicate number.

When creating a project, first suppy a request to the “/rest/product\_accessable”, which will return a JSON object containing an array of the ids for all objects for which the project creator credentials may have access to. GDOT JSON.parse() may then be used to convert this to a 1 entry dictonary, where at key “ids” an array containing the ids of all products (different from the 5 digit code assigned as part of the name) is stored. Iterative access to the array using each entry as an argument to the Get Product access point “/rest/product/(id)” will return a JSON object containing the product (project) details. As above, this object can be parsed into a key:array dictionary where key is “products” and the array is a 1 entry array further containing dictionaries with the various parameters of the product. From this array, the last entry is a dictionary “name”:string, where the string is the product (project name), including the internal TMEIC project number. Comparing the leading characters of this string to the string entered into the “project\_num” field can then be used to ensure the project creation utility does not accidentally create two projects with the same number.

*App Architecture:*

Godot uses an architecture where individual graphical and functional elements are specified as “Nodes,” which are classes inheriting from more abstract parent classes. Different nodes serve different purposes and have different effects. For example, a button is a node, inheriting indirectly from a parent class “control node,” and a HBoxContainer is a node also inheriting indirectly from the control node parent class. However, the button node is plainly visible and interactable with from the interface, whereas an HBoxContainer node is only indirectly visible in the way in which it positions other nodes related to it. In the hierarchy of design, there is a root node, which may have 1 more child nodes attached, each child node having any number of further children attached, so on and so forth, in a tree pattern. Each node may have a single script attached to it, which can be used to extend the functionality of the node beyond what the default configuration of the node is capable of.

In this design, it has been the objective to minimize the number of scripts while also keeping scripts local to the nodes they primarily act upon. Most information used in the creation utility is stored in a text field which is local to the specific node creating the graphical element. The “main” script, attached to the root node “overwatch” can retrieve these values from these text fields, since these text fields are children.

The remaining scripts used in this application are:

Popuppanel.gd: This script simply controls whether the panel presenting selectable components is visible or not. Its methods receive signals from the open button on the main form, and a close button on the popup panel.

Component\_Selection.gd: This script builds the selection components for the popup, allowing a user to configure a custom selection of components to add to a newly created project. As with popup panel, this script receives signals from the open and close buttons. Upon pressing the open button, the script searches the dictionary stored in the list\_of\_components.gd node, and preselect those components. The build components function is currently coupled to the overwatch script, and could be decoupled by moving the CSV list of components variables and their converted dictionary to the ready() function of this script. Currently build\_components\_panel(), the function which sets up the available components from the components.CSV file, is called during the ready() function of overwatch, which is after the instantiation of each child node to overwatch, and the completion of ready() functions for each of those nodes. (Godot builds a node tree and recursively calls ready on each child as it does so, resulting in resolving the ready() functions of the last added nodes at the child-most level). This currently is done so that the CSV file is fully loaded and parsed into a dictionary before constructing the panel. Each item on the panel is a child of the popup panel, dynamically added, and thus indexed 0-# of components -1. Each of these items is instantiated as a component\_chec.tscn item, which has its own scripting for storing the id, shortname, full name, and description from the CSV file. Behavior of these children nodes is internalized, and the Component\_Selection.gd script handles retrieving relevant information from these children when updating which components are presently selected. When the close button for the panel is pressed, the singnal emitted with invoke the on\_clone\_panel\_pressed() function, which will clear overwatch’s tracking of currently selected components, and then repopulate that tracking with each child which is selected. This function then calls the updateListing() function of the components list (later described) to update the window of selected components.

List\_of\_components.gd: This script contains a single function updateListing(), which when called clears its current display text, retrieves the id’s of all currently included components in overwatch’s dictionary of currently included components, and then adds the full name of each component to the text box to which the script it attached on a separate line by referencing the retrieved keys against the dictionary containing all components.

Project\_type.gd: This script solely serves to establish the radio buttons on the form as a mutually exclusive set of radio buttons (that is only one may be pressed at a time). It is noted here that the children buttons of the Project Type Container each emit a signal, on pressing, to overwatch.gd instructing overwatch to load a predefined selection of components as the current components, and instruct the list of components to update.

REST\_to\_Bugzilla: This module is to be completed, (see flow chart later), and handles the processing and HTTP requests to utilize the Bugzilla REST API. It is structure that, upon receiving a begin signal from overwatch, will make several requests, parse the resultant data, iteratively retrieve configuration data, construct the necessary JSON objects for submission for the new project, and transmit the objects. It is advise that when attempting to complete this module, one refers to the Bugzilla REST API documentation to understand the JSON objects anticipated to be retrieved and transmitted. Integration of this .gd file should be as a node parented to the overwatch node, the internal testing button in the scene removed, and references to the button in the script be redirected to the project creation button located at the bottom of the form.

URI\_to\_bugzilla: this .tscn file and its accompanying .gd script should not be used. It was an earlier attempt to use the HTTP Client node underlying a HTTP Request node to manually configure HTTP requests (other than REST requests) to the Bugzilla server. At this time, since the issue is apparently server side, it is recommended resolving the server size issue to enable to application to be competed.

**TODOs:**

1. Resolve server issue: Without resolving the issue regarding JSON returns in the TMEIC Bugzilla server, this program will remain a non-starter. While it is hypothetically possible to do PUT/POST requests via the REST API, and hope the requests are handled without a proper return confirmation, testing with properly structured requests to the REST endpoints with appropriate POST requests (I.E. attempting to create a new project) has shown the server is not processing these requests, as no new project is created.
2. Complete scraping algorithm: The algorithm for converting text field contents to the JSON object is incomplete at this time. Completion of this algorithm will involve developing a method called on project creation that will 1) find all Nodes containing relevant information for project creation, 2) retrieving said information and storing into Overwatch, 3) organizing said information into a nested dictionary commensurate with the
3. Complete Upload method: Project creation via REST API will still need to be implemented and ca be added based on the scraped data from the fields and component dictionary configuration. Due to the way the REST API for Bugzilla is created, a 2 stage process should be employed. Reviewing [§ 6.1.12.3 Products](https://bugzilla.readthedocs.io/en/latest/api/core/v1/product.html) & [§ 6.1.8 Components](https://bugzilla.readthedocs.io/en/latest/api/core/v1/component.html) for the create product and create components endpoints shows when creating a new product (project in the case of the TMEIC Server), a product can be created without a first component, as opposed to the CGI method employed in the previous tool. Based on the data scarped from the Project Name, ID, Client, and description fields, the name & description objects of a dictionary, to be converted to JSON, should be set. It is recommended that product version be configured to a default of 1.0, though it may be desirable to add a control field for this to the Kabuto application. The Product JSON can then be submitted to the /rest/product endpoint via post, to add the new project, which will return a JSON containing the id of the newly created project. It is important to save the name of project to which the upload is made, or ideally, use the newly returned id to retrieve the name to which the project has been saved, for use in the next stage.

In the second stage of uploading to the Bugzilla server, the dictionary current\_components\_dict should be iterated through to retrieve component names and description, and each component name and description begin added to a dictionary converted to JSON object as described in § 6.1.8.1 Create component, and assigned an appropriate default owner. These objects should hen be posted to the /rest/component end point as created to add the new components to the newly created project. An id for the new component will be returned with each successful call, and receipt of this id can be used to confirm successful addition of the new components to the project.

1. Complete server selection: While a selector button has been added for changing the target server for project creation, scripting has not be added to target the appropriate server based on the selection. Scripting should be added to change this over, as well as update the components lists to a default components configuration dictionary as appropriate when the server is changed over.