VisualVault Customer Project Coding Standards

Objective

The purpose of this document is to establish a pattern of developing scripts and code for customer projects. This will enable readability and efficiency for multiple team members to consume and work on scripts.

Edition Details

|  |  |
| --- | --- |
| Edition 1 | 4/22/2019 |

Contact Us

If you have any suggestions on how this document may be improved, please contact Jason Hatch at [Jason.hatch@visualvault.com](mailto:Jason.hatch@visualvault.com) .

For information on VisualVault, please visit [www.visualvault.com](http://www.visualvault.com).

Copyright Information

This document is for informational purposes only. VisualVault MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT. Complying with all applicable copyright laws is the responsibility of the User. Without limiting the rights under copyright, no part of this document may be reproduced or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise) without written permission from VisualVault. VisualVault may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from VisualVault, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

# Coding Standards

The following are a list of standards and guidelines for how project code should be implemented and used.

**Client-Side Coding Standards**

* Client-side code is for validating data entry and automation of actions where all information is present on the client. This is unsecure code in plain text.
* Calls to web services/outside processes should only occur once in a client-side event cycle to avoid chattiness with the server.
* Client-side logic should not handle heavy lifting business logic that requires API calls to send emails, get form data, etc. This type of logic should be secured and obfuscated in web services.
* For client-side data entry validation functions, put the fields in alphabetical order within the client-side code. If an if statement is used to group sets of fields, put the field names in alphabetical order within the if section of the code.
* Minimize code at the event level. Centralize validation code into Form Template functions and put major logic into the Form Template functions.
* When setting date fields or otherwise handling dates, avoid .toLocaleDateString() and .toLocaleTimeString(), as these methods are not supported in Internet Explorer 11. You may instead build a string to represent the date in order to use the VV.Form.SetFieldValue() function.

Example: Build a string to represent the date. The dateObj may be retrieved from a calendar field, or declared as a new Date().

var dateObj = VV.Form.GetDateObjectFromCalendar('fieldName');

//var dateObj = new Date();

VV.Form.SetFieldValue('nameOfFieldtoSet', (dateObj.getMonth() + 1).toString() + '/' + dateObj.getDate().toString() + '/' + dateObj.getFullYear().toString());

**Server Side/NodeJS Coding Standards**

* A web service on the server side should handle calling multiple server-side resources to bring together information.
* On the server side, when you are using vvClient.forms.getForms in code, assign the name of the form template to a variable in place of the form template id. If you are updating a form or creating a new form record from vvClient.forms type calls, you can also use the name of the form template. It is not necessary to use the GUID of the form template.
  + Create the variable right after the logging statement so it is at the top of the code. Using the name will ease migrating from dev to test and production.
  + The variable that contains the form template name should start with a brief name of the template and end with the string “TemplateID”. Generic names are not helpful in this circumstance to understand the code later.
* A comment section needs to be at the top of the module.exports.main section that looks like the following:

var vvEntities = require("../VVRestApi");

var logger = require('../log');

module.exports.getCredentials = function () {

var options = {};

options.customerAlias = "CUSTOMER ALIAS";

options.databaseAlias = "DATABASE ALIAS";

options.userId = "USER ACCOUNT";

options.password = "PASSWORD";

options.clientId = "DEVELOPER KEY";

options.clientSecret = "DEVELOPER SECRET";

return options;

};

module.exports.main = function (vvClient, response, token) {

/\*

Script Name: NAME OF WEBSERVICE (SHOULD BE ALL ONE WORD)

Customer: CUSTOMER NAME

Purpose: DESCRIPTION OF THE WEB SERVICE.

Parameters: The following represent variables passed into the function: (Following are examples only.)

objUserList - Object with list of users passed to function.

arrStatus - Array with list of statuses for each user.

txtSite - Name of site.

Return Array: The following represents the array of information returned to the calling function. This is a standardized response. Any item in the array at points 2 or above can be used to return multiple items of information.

0 - Status: Unique, Not Unique, Error

1 - Message

2 - objUserInformation - Object containing user information.

Process Pseudocode: The following documents the pseudo code for this process.

Step1: Do this to acquire that.

Step2: Doe that to complete the other.

Date of Dev: 05/07/2017

Last Rev Date: 11/16/2017

Revision Notes:

05/08/2017 - DEVELOPER NAME: Initial creation of the business process.

11/16/2017 - DEVELOPER NAME: Update the query to include submission and waiting approval...

\*/

* For the purpose section of the comment header, put enough description that those reviewing the code might know the actions that are occurring in this code.
* For web services that are reusable, document the Parameters and Returning Array in the header section. See information above. Preface variable names being passed or returned with type of data being passed.
* For web services that are reusable, or in other circumstances where it may be helpful, include a “Process Pseudocode” section in the comment header, after the Parameters section. This can help future developers to understand your code structure.
* Logger.info Standards:
* After the comments section, start every nodejs script with a logger.info statement like:

logger.info('Start of the process NAME OF WEB SERVICE at ' + Date());

* Include a catch block at the end of every nodejs script that includes a logger.info statement like:

logger.info(JSON.stringify(err));

* Throughout the body of your script, rather than using too many logger.info statements where errors occur, which can make your code “chatty,”
  + Throw new errors to the catch block for logging when it is a single error or add to an error log variable when multiple errors should be reported back to the calling function.
  + Use logger.info sparingly throughout the body of your script, where critical events occur.
* After any API call, measure the results and handle the results. Simply continuing without determining what happened is not a good practice.
* When using an API query, such as when using vvClient.forms.getForms(), consider whether your query string could include any apostrophes. This is common in names. String expressions containing an apostrophe must escaped with the apostrophe with either a \ (backslash) or escaped with a ‘ (single quote).
  + This is important to remember when calling LibFormVerifyUniqueRecord, a web service that is frequently used.
  + A reliable way to ensure that all apostrophes are escaped in your string expressions is to use the .replace() method:

var firstName = ffCollection.getFormFieldByName('First Name').value;

var firstNameSearch = firstName.replace(/'/g, "\\'");

var queryParams = {};

queryParams.q = "[First Name] eq '" + firstNameSearch + "'";

* For web services that are scheduled type processes, a response completing the process must respond to VV so that success or issue results are communicated in the logs.
* Comments need to be put throughout the code for others to consume and understand why different actions are being taken or to understand the action that is occurring. Place comments to the right of variables or above a block of code.
* Generic calls to get a form, send an email or other things of this nature must happen server side. Generic functions and business logic like this should not be exposed on the client side in plain text.
* When updating a checkbox with a true or false value, set the value to ‘True’ or ‘False’ rather than ‘true’ or ‘false.’ The value will be saved in a case-sensitive format, and using title case will ensure consistency with manually checked boxes.
* When returning a message from most NodeJS web services called by a form or a reusable function, return as standard an array where item 0 is a status and item 1 is a message. If any other objects need to be returned, put them in an item of the array greater than 1.
* Bubble error messages back through to the client. Not all error messages should be displayed to the client but they should be specific and handled as needed.
* Naming of web services should begin with something about the form in question or Lib if they are reusable web services. Reusable web services should also include the name of the platform area being handled. For example, LibForm, LibDoc, or LibSites. After this, the name should be descriptive of what will occur.
* Any time a change is made, update the Revision Notes section as well as the reason for change and check in changes to Github.
* Changes need to be checked in regularly to Github.