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 | 1/8/2020 |

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.

# Working in Projects Efficiently

The purpose of this section of the document is to communicate and recommend how developers can work most efficiently when implementing VisualVault. The following are suggested guidelines to help developers acquire momentum and provide higher levels of quality.

Before Starting Configuration and Development

* Start by reading and understanding the specifications and design documents. Focus first on the high level/system drawings and explanations. Then the sections of these documents that relate to your tasks. Understand how the data and workflow work together to support the solution.
* Understand how data flows into your assigned tasks/process.
* Identify happy and unhappy paths from the design document or that are implied in the system.
* Identify a logical order for developing your tasks if that has not been dictated to you.
* Setup test data that flow into your task/process that represent happy and unhappy process paths if configuration has been completed. (Under most circumstances, dependencies should be scheduled to be built before they are needed.)

Configuration and Development

* Begin configuration and development of a single task selecting the tasks that logically should be first. Move it to an In-Process state in the project management software.
* When developing code, use the test data of happy and unhappy paths to make sure the code functions properly under multiple conditions. Make sure error handling is correct. To accomplish this, you may need to key in test data to a form to force a certain path. You may also temporarily hard code a variable to force a certain path in your code.
* User the debugger to debug your code when you run into issues.
  + When debugging client-side script, press F12 in the browser, search for your script and set a break point. When the breakpoint is reached, highlight over the variable to identify current information in the variable.
  + When debugging server-side scripts, use VisualStudio Codes debugging tools to walk through your code and processes.
* If you run into an issue with the technology, try to resolve the issue for 30 minutes to no more than an hour. Use tools like user manuals, specification documents, code examples from GitHub, etc. to try and solve your problem. Then ask someone for help via Slack, email or in person. Avoid spending too much time to resolve issues that others know how to resolve.
* If you don’t have all the information you need to finish a task, then look at customer provided documents in the project repository or notes in the Zachman framework for this information. If the information is not present or clear, ask the business analyst or solution architect where you can find this information. In the end, if the information is not present, the project manager and solution architect need to work with the customer to acquire appropriate information.
* When a task is completed, mark as complete and move to the next task.
* After completing all configuration and coding tasks, conduct unit testing of an entire process. Make sure the workflow functions, code executes and business rules are enforced using specific users or states of the process.

**Note**: By following the above points, you will gain momentum in working through tasks, testing will occur along the way and the process will be tested in the context of each user who interacts during the workflow of a process.

# Template Design Standards

The following is a list of standards and guidelines for designing VisualVault iForm templates.

* Every template should have an Admin Override container.
  + This container should have only two controls in it: a checkbox named “Admin Override” and a button for Admin Save.
  + The Admin Save button should call the global template function VV.Form.Global.AdminSave();
  + Only VaultAccess users should have visibility permissions to the Admin Override container.
* A close button should be available at the top and bottom of each form. Close buttons should call the global template function VV.Form.Global.CloseAndUnlockForm();
* When working with Groups and Conditions, a given control or container should not be placed in two different groups. Each form element that is in a group should be part of only one group.
* The SaveButton Form Control should always be hidden.
* The TabControl Form Control should never be hidden. Access to form tabs should be controlled in the Menu tab of the Template Details screen.

# 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. Items that should be at the event level are as follows:
  + Confirmation messages.
  + Calls to validate data entry.
  + Messages to handle data entry errors.
  + Formatting of phone numbers, zip codes or other similar scenarios.
* 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());

* When using JavaScript functions in client-side scripts, make sure to use functions that are compatible with Internet Explorer 11 or greater, Chrome, Firefox, Microsoft Edge or Safari. Using unsupported functions will cause solutions to break or be unreliable in different browsers. Here are examples of commands that we should not use:

**DO NOT USE:**

* + “let”, “await” or “async”.
  + const absValue = (number) => {
  + backtick literals: (`${successMessage} The form will be saved when the yellow banner appears.`)

These commands in code will not work with Internet Explorer 11. It is a developer’s responsibility to check for compatibility against the MDN documentation.

## 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.
* When declaring variables, only use const in web service scripts to declare configuration type variables that will never change in your code. At all other times use let or var. This will prevent errors from occurring if someone inadvertently attempts to assign values to a const variable after it has been declared.
* 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. For example, if a “not unique” error returns from a verifyUnique call, pass back a user-friendly error that will tell the user know which fields need to be updated.
* 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.

# Implementing Specifications

* The design document serves as the instructions for putting together a given solution.
* It is important to be familiar with the structure of the design document so that you can effectively look up information related to a task that has been assigned to you as a developer.

## Developing to the Specifications

* Discovery is completed on-site with the customer.
* The solution architect, the project manager, and a team including at least one business analyst is typically involved in discovery.
* The design document is written to capture the solution architect’s complete concept of the implementation approach.
* The architect is the person who holds and drives the vision of the end product.
* As a developer assigned to a task, you must use the design document to understand why you are being asked to implement an item in a specific way.
* Taking time to understanding the bigger picture by reading the design document will ensure that both the letter and the spirit of the specifications are implemented.

## Professional Recommendations

* If you understand the “why” of your task, you will likely find opportunities for improvement, whether that be efficiency of processes, enhanced user experience, or something else.
* Your feedback is valuable and encouraged. The business analyst or solution architect is the appropriate person to contact.
* The business analyst and solution architect will consider your suggestion with the full context of the entire solution. If accepted, the solution architect will ensure the change is communicated to the rest of the project team, communicated to the customer if necessary, and that supporting documentation is updated.

## Scope Creep and Other Unfortunate Realities

* Not every professional recommendation or suggestion may be implemented.
* It is expected that the solution architect will be open to suggestions from team members, that each suggestion will be considered fully, and that if a suggestion is not accepted, the architect will communicate why.

## Bug Fixes and Corrections

* Refer to the specification to identify what the configuration should be for the process.
* If there are major discrepancies in the implementation versus the specification, communicate with business analysts or solution architects to identify that you have the most recent information.
* As you fix the issue keep track of the changes you make. Keep track of the following:
  + Record every field where you add or change the properties of the field.
  + Record every group and condition you add or change with form groups and conditions. Communicate group names, any controls you added/removed with the group, and any conditions that were added/removed.
  + Record where you added any event scripts.
  + Record the name of any form template or global script you changed or created.
  + Record the name of any web service you changed or created.
  + Communicate the above changes in the JIRA ticket to the QA Team. More detail is required when working in the form template because those changes are manually migrated to Sandbox or Production environments. Only names of scripts are needed because the entire script is moved when doing migrations.
* Test your fixes as every possible user through the complete workflow and regression test to make sure previous use cases still work.

# Site Configuration Standards

When setting up a new VisualVault site, standard configuration steps should be followed.

## Users

User accounts that serve a programmatic purpose should be named as follows:

* Customer.api – used for web services only.
* Customer.config – used for configuration accounts. The data center team will typically set up this user.
* Customer.purpose.import – used for configuring imports that have security in a specific area of the system. Example: customer.files.import or customer.emails.import
* Customer.ocr – used for OCR. The VisualVault technical resource that is configuring OCR will typically set up this user.
* Customer.purpose – used for any other solution-specific programmatic purpose with security in a specific area of the system.