GenKFI Developers Guide

Jason Browne  
TRD IT

Written: 2014-01-03

Revised: 2014-01-03

Contents

[Page By Page 3](#_Toc376529113)

[Overview 3](#_Toc376529114)

[General Classes and Files 3](#_Toc376529115)

[Master Pages 6](#_Toc376529116)

[Batch Pages 6](#_Toc376529117)

[User Management Pages 6](#_Toc376529118)

[Form Building Pages 7](#_Toc376529119)

[Data Entry Pages 7](#_Toc376529120)

[Javascript Files 7](#_Toc376529121)

[Application Flow 9](#_Toc376529122)

[Overview 9](#_Toc376529123)

[Users Login 9](#_Toc376529124)

[Data Entry 9](#_Toc376529125)

[Building Forms 9](#_Toc376529126)

[Managing Users 9](#_Toc376529127)

[Managing Batches 9](#_Toc376529128)

[Admin vs User 9](#_Toc376529129)

[Database Structure 10](#_Toc376529130)

[Overview 10](#_Toc376529131)

[Batches 10](#_Toc376529132)

[Controls 10](#_Toc376529133)

[Users 10](#_Toc376529134)

[Data 10](#_Toc376529135)

[Known Issues 11](#_Toc376529136)

[Current Issues 11](#_Toc376529137)

[2013-01-01 11](#_Toc376529138)

[Resolved Issues 11](#_Toc376529139)

[2013-01-03 11](#_Toc376529140)

Formatting Conventions

**Bold** = Page or File Name

*Italics* = Function Name, Table Name

Underline = Variable and Class Names, Table Column Names

# Visual Studio Specifics

## Compile Version

Throughout the application, it connects to the database in many places. At each of these connections, the compiler preprocessor directives #if, #else and #endif are used to make the correct connection. If it is being compiled in Debug mode, the regular connection string (Resources.connString) is used. If it being compiled in Release mode, the production connection string is used (Resources.prodConnString). These connection strings are found in the projects Properties under the Resources tab. To access the Resources in the code, using GenKFI.Properties; is included at the top of the file.

## Publishing

When the application is ready for deployment, whether to the test server or to production, the Visual Studio Publishing tool is used. This tool is found under the Build menu. The project currently has two publish settings defined, “toFile-Dev” and “toFile-Release”. Both of these will publish the project to a file system. To publish the application on a new or different machine, the file path may have to be modified. Press the Back button twice on the publisher window to edit the file path. As the names indicate, one setting will publish the Debug version, and one will publish the Release version. Once the application is published, the output files can simply be copied into the appropriate folder on the respective web server, listed below. The TRD Tech Team will have to be the ones to move the files over for a production release.

* Debug: \\trdpipelinedev\wwwroot\GenKFI
* Release: \\trdpipelineprd\wwwroot\GenKFI

# Page By Page

## Overview

This section describes each page in the application individually. It will describe the purpose of the page, and the general flow through the page. Following this is a detailed description of each function. The code-behind page of ASP pages is considered included in a pages description. The pages and files below are ordered by dependence, the following pages, in general, depend on previous pages in the list.

### General Classes and Files

#### godConn.cs

##### General

This file contains the godConn class. This class handles all communications with the database. Usage is outlined in the file. Once an instance is created, it must be connected to the database via the *connect* function. When called without parameters, it will use the connection string stored in the projects Resource properties. Alternatively, a connection string can be supplied, which is done in several of the later pages. When the needed database operations have been completed, the connection is closed with the *disconnect* function.

Once connected, this class has three basic types of operations: transactions, SQL statement processing, and statement processing with parameters.

##### Transactions

When desired, SQL statements can be wrapped by transaction statements. To do this, simply call *startTransaction* before performing any database operations. When the operations re complete, the transaction can either be rolled back, via *rollbackTransaction*, or committed, via *commitTransaction*.

##### Statement Processing

There are three types of statement processing functions: execute, scalar, and select. Execute statements include UPDATE, INSERT, and DELETE, and are passed to the *doExecute* function as a string containing the SQL statement. *doExecute* returns an integer equal to the number of rows affected by the statement. Scalar statements are SQL statements that return exactly one value. These statements are passed to the *doScalar* function as a string. This function returns the result of the statement as an object, which must be cast or parsed to the desired type.

Select statements are processed by two functions, *doSelect* and *readSelect*. *doSelect* returns an integer equal to the number of rows the SELECT statement returned. This allows for better program flow control, allowing the developer to skip reading an empty return by checking if the SELECT returned anything first. Assuming the statement did return rows, they can be retrieved with the *readSelect* function. This function is passed a reference to a DataTable via ‘out’. The function initializes the passed DataTable, and fills it with the rows returned form the most recent call to *doSelect*.

##### Parameterized Statement Processing

Parameterized statements can also be used. To do so, first the *paramCommand* function is called, regardless of the type of statement. Then one of the three overloads of *setParam* is called to set the values of the parameters. The first overload sets one parameter, via the passed name as string, parameter type as SqlDbType, and then the value as an object. The second overload is simply passed an array of SqlParameter objects. The final overload is similar to the first, but the three arguments are arrays.

Once the parameter values have been set, the statement can be executed similar to the non-parameterized functions above, using the functions *doExecutePARAM*, *doScalarPARAM*, *doSelectPARAM* and *readSelectPARAM*.

All functions in this class will throw an exception of anything goes wrong. The message in the exception thrown contains information related to which function was running and the SQL statement that was passed. The exception will also have an *InnerException* set to the actual exception that caused the error.

#### myUserRole.cs

In order to facilitate the impersonation feature (an admin can view the application exactly as another user would), a wrapper of the built-in User and Role features had to be created, the myUserRole class. This class only has three functions, all of which are static. The first, *isImpersonating*, simply checks the Session value ‘impersonating’, and returns true if it is not empty and not null, otherwise false. The second, *UserName*, checks the return value of *isImpersonating*. If it returns true, this function returns the user name of the user being impersonated. If it returns false, the return value will be the current user’s login. The final function, *IsUserInRole*, is passed a string with the name of the role to check (ADMIN, SUPER, USER), and also checks *isImpersonating*. If that function returns true, this returns a Boolean indicating if the impersonated user is in the passed role. Otherwise, the current user is checked.

#### utilClass.cs

This class has several static functions, as well as several public functions. It also has a member variable adminOnlyPages, which is filled on construction.

##### Instantiated Functions

These two functions can only be called from an instance of the class. This is to ensure the ArrayList adminOnlyPages is filled before calling these functions. The first function, *adminOnly*, checks current page against the list of admin only pages, and redirects the user to **error.aspx** if they are not in the one of the roles ‘SUPER’ or ‘ADMIN’. The other function, *errorRedir*, simply redirects the application to **error.aspx** if there is no user name specified.

##### Static Functions

These functions are used for general validation and small repeated operations.

Whenever there is an error with the godConn class, it is caught, and the *recordError* function is called. This function is passed the godConn object, and the Exception that was thrown. It then puts an entry into the errMsg table in the database, recording the error message, the time, and the user that was using the application when the error occurred. It also dissects the stack trace, finding the filenames and line numbers of the code leading up to the error. This assists in recreating and debugging the issue that caused the error.

To help ward off invalid data that might cause errors in the database, use *sanitize* on any SQL values. This function will simply double all single quotes in the passed string. This allows the database to accept string values such as “Annie’s”.

In order to make the application usable across different domains with the same user name conventions, the function *trimDomain* can be used to simply trim the domain name off a particular user name.

Because the username is not exactly the same in certain instances (saved in database, different server reporting, manual entry) the function *normalizeUN* is used to convert a user name to a format uniform across this application. It simply converts the domain name to all capitals, and the user name to all lowercase.

There are two functions, *tableExist* and *columnExist*, that will check the connected database for the requested table or column. *tableExist* takes the name of the table, and an already connected godConnobject. *columnExist* takes the table and column names, as well as an already connected godConn object. They both work by simply trying to select something from the provided table and/or column. If the selection works, they return true. If the selection fails, the Exception is caught and checked for keywords that would indicate the requested table/column doesn’t exist. If those keywords are found, they return false. If the selection failed and the keywords are not found, the Exception is thrown back to the calling function.

Very similar to the above functions, *allColumnsExist* will check the rfrControls for a given table and where clause and retrieve all column names that are needed in the data storage table. Note, this function also requires a passed, already open godConn object. Once it has these column names, it checks the table for the needed column (via *columnExists*), and returns a List of the columns that aren’t present in the data table.

#### Global.asax

This class is instantiated when the application is first started or opened. There is a large section of code commented out, which was used to set up the database with users and roles for initial testing. This class does two things. When the application is first started, it checks to see if the needed roles exist, and if not it adds them. It also checks for at least one ADMIN user, and on finding none, adds ‘TRD\_MASTER\jasonb’ as an ADMIN user. This is done to ensure at least one user can access the application for initial setup.

The other function in this class, *Application\_AuthenticateRequest*, is called anytime a user tries to use the application, which automatically tries to authenticate the username with the windows login service. It checks if the user name has been associated with any roles whatsoever. If not, then this is a new user. The new user is then added to the USER role, and assigned access to all the tax programs. More detail can be found in Users Login below.

#### BreadCrumbNode.cs

This is a simple class that holds a URL and a provided name. The name is provided on construction. The URL stored in the class is the page the user is currently on at the time of construction. These values are stored in private member variables. There are public member variables with can only read the values. This class is utilized in the BreadCrumb user control below.

#### BreadCrumb.ascx.cs

This is a user control which renders the previous 5 pages the user has visited. It stores these pages as BreadCrumbNode objects in a private List named URLList. The control has been added to the master page **GenKFI.Master**. As a page is rendered, the user control is also rendered, and the *Page\_Load* function is called. This function checks to make sure the call isn’t a Postback or the master page load, and if so, grabs the title of the page and adds a new BreadCrumbNode to the list via the *Add* function.

The *Add* function has two overloads. The first takes a string as the title of the new BreadCrumbNode, creates an instance, and calls the second overload. The second overload takes a BreadCrumbNode instance. It will check if this URL is already in the list of nodes. If the URL is found, that node is removed from the list. The new node is then added to the end of the list. Lastly, if the list has more than 5 elements, the first node is removed. This serves to keep the most recently visited page at the end of the list, while limiting the length of the list to 5 elements.

The actual list, URLList, is managed by its getter and setter. The setter stores the list in the Session using the key value “KEY\_” + strURLGroup, where strURLGroup is a public property of the control. Similarly, the getter retrieves the list from the Session state, casts it to a List, and returns.

Currently, strURLGroup is unused, and remains blank. It can, however, be set to different values to provide a different group of BreadCrumbNodes, for example, navigating a settings page as a subset of the main application.

#### utils/errView.aspx

This page is simply a display for the rows in the errMsg table in the database. In the *Page\_Load* function it selects all the rows from the errMsg table, along with some formatting that provides the framework for displaying the information as an HTML table. Once it retrieves the DataTable, it is databound to the ASP Repeater on the ASPX page. The Repeater’s ItemTemplate then evaluates the column values, and displays the table. Note also that the *Page\_Load* function does not check the Postback state, as there is no user interaction on this page.

#### utils/editPCodes.aspx

*This page is no longer used, as all users have access to all program codes by default.*

This page displays the available program codes, such as PIT, FID, CIT, etc. On initial load, it loads the program codes, description and removed state from the database table progCodes. It displays the codes in a ListBox, via a function named *loadCodes*. The codes ListBox has AutoPostBack set to true, and its OnSelectedChanged event is bound. The program code is set as the Text of each item in the ListBox, and the primary key from the database is set as the items value.

When a program code is selected, the page posts back calling the ListBox’s OnSelectedChanged function, which simply calls a function called *loadDetails*. This function gets the description and removal state for that primary key from the programCodes table, and displays them in the appropriate controls on the page.

The three buttons, “Add Code”, “Save Edit” and “Remove Code” are fairly self-explanatory. “Add Code” will add a new program code to the related table using the values in the TextBoxes via an INSERT. “Save Edit” will UPDATE the table, filtering by the selected primary key in the ListBox of program codes. “Remove Code” will DELETE from the table, again filtering by the selected primary key. Each of the functions calls *loadCodes* when it is finished, to update the display of the program codes.

*loadCodes* first checks if there is a selected item in the ListBox, and saves it before clearing the ListBox. It then re-fills the ListBox, as described above. It then will check if there was a previous selection, and if it is still in the ListBox after the re-fill. If so, it sets the correct SelectedIndex value, and calls *loadDetails*. All this is done to reflect any and all changes caused when using the “Add”, “Save” and “Remove” buttons.

### Master Pages

#### GenKFI.Master

This page contains the elements, both markup and code-behind, that are used on most of the pages in the application. It contains the script tags including jQuery, jQuery-UI and the drop down check list plug-in. The markup elements included are:

* Image and text in header
* Layout indication
* BreadCrumb control
* Tax Program drop down selector
* List of users with ‘impersonate’ button
* ‘Home’ button
* Footer indicator for use when impersonating

As with master pages, there are two ContentPlaceHolders, one for the headers, and one for the body.

Since the master pages *OnInit* is called before the pages *Page\_Load*, and the masters *Page\_Load* is called afterwards, we utilize the masters *OnInit* for some initialization. This event simply SELECTs the program codes from the database, and stores them in a Session value, *tpSelHID*, and a hidden control of the same name if that Session value is null. This is the value that indicates which tax programs the user has selected. If the Session value is null, then it defaults to having all tax programs selected.

Several more forms of initialization are done in the masters *Page\_Load* event. (Note: At this point the pages *Page\_Load* event has already been called and finished.) If a home link has been set by the page, it is set as the link for the home button. If no layout has been set, it defaults to portrait. If this is not a postback, the tax program list is filled via the *fillTPList* function. This function not only fills the drop down with the tax programs, but checks the Session value *tpSelHID* (set in *OnInit* above) for selected tax programs, and sets the selections appropriately.

It then also checks if the user is in the ADMIN role or not. If not, the impersonate dropdown list and button are hidden, restricting the user. If so, the user list is filled. Then the Session value *impersonating* is checked. If that is not empty or null, it selects the user in the impersonate dropdown list, and shows the footer at the bottom of the page. The footer contains a button that resets the impersonating status by clearing the *impersonating* Session value. Similarly, the impersonate button sets the Session value, displays the footer, and reloads the page to force the application to behave as the impersonated user would see it.

On reloading the page when impersonating, the *Page\_Load* function does a very important check. This check succeeds if an ADMIN is impersonating a USER, if the user is in the USER role, or the Session value *keyMode* is set. The function proceeds to check the database tables for valid batches and images assigned to the user. If there are batches, it then proceeds to check the batch queue (table tblKeQueue) for the username and any intermediate sequence values. This is to check if the user had left off in the middle of a batch, rather than just finding the next batch assigned to the user. It then proceeds to build a query string with all the relevant information, and redirects to the **KeyEntry.aspx** page. This effectively forces the users to simply start keying, without having to find their batch and sequence numbers. If no batches are found to be assigned to the current user, they are redirected to the **Defaults.aspx** page, and shown a message that they need to contact their supervisor to queue more batches.

#### forms.Master

This is the Master Page for the three pages, the form builder, form tester and the actual key entry page. This page has the panels for the image and the HTML controls, several java scripts that are used to control data flow and managing the image, as well as hidden fields that assist with data flow across forms.

In the code behind for this page, there are two main functions that are of importance: *registerAutoJavaScripts* and *getControlsTable*.

*registerAutoJavaScripts* is used by **KeyEntry.aspx** to register the editable JavaScript templates when loading a form for entry. (Read more about the auto JavaScript templates in Application Flow below.) This function takes three parameters: an instance of the godConn connection class, a control header ID, and a name index. The connection must already be open before passing it to this function.

The control header ID identifies exactly which form is needed. This is used to retrieve the control names, for use in applying the JavaScript templates to the actual controls.

The name index is used for repeated forms. When a form is repeating, it can repeat the same HTML template several times. Each time the HTML form is repeated, an increasing number is appended to the control names. That number is what is passed as the name index, so the JavaScript that is written to the page can be linked to the correct controls.

The retrieval of the templates (stored in *rfrJavaAutoFunctions*) is combined via several UNIONs with retrieval of actual uses of those templates (the actual uses are when a template is applied to a specific control on a form, stored in *tblAutoJavaComplete*). The UNIONs are necessary because there are several use cases combining an actual control name, the previous value of a function, or a constant. Once all these retrieved, the actual JavaScript is constructed, replacing placeholders in the templates with the names of the controls for this form. The final JavaScript is wrapped in a jQuery *$(document).ready* function, and registered on the page with the *RegisterClientStriptBlock*.

The other function used by**KeyEntry.aspx** is the *getControlsTable*. This function takes 10 parameters, but only 3 are vital (the rest have defaults set). As with *registerAutoJavaScripts*, this function takes an instance of the godConn connection class. This connection must be opened before passing it to this function. Also passed into this function is the ID of the submit button on the page that will be using the resulting HTML table. The last vital parameter is the control header ID of the desired form. (Read more about the form setup in the Database Structure section.) The rest of the parameters relate to fine tuning the movement of the image and marker, but are not required.

The function, using the control header ID, retrieves all the stored controls from the database for that ID. It then uses a template string for a generic HTML input element, and replaces some values with the options for each control retrieved from the database. The returned HTML string contains all the needed input elements, neatly placed into a HTML table structure.

### Generic Pages

#### Default.aspx

This page is merely a landing page, with links to the various functional pages within the application. Depending on the role of the user, some or all of the links in the markup are removed. An instance of utilClass is made to double check if the user is allowed on this page, via the *adminOnly* function.

The button “Key Mode” will set the Session value *keyMode*, and refresh the page. This effectively allows an ADMIN user to assign batches to themselves, and key as a normal user would.

The link editLNK first checks on how many tax programs are selected (via the Session value *tpSelHID*). If only one tax program is selected, the user is redirected to the **existForms.aspx** page, where the process for adding and editing pages and forms for each tax year begins.

#### layoutStart.aspx

This is a very simply page with two ImageButton controls, and merely sets the Session value *layout* to either ‘portrait’ or ‘landscape’, and redirects back to **Defaults.aspx**. This Session value is used when rendering the form pages, both through the page itself and the **forms.Master** page.

### Batch Pages

#### utils/unassignedBatches.aspx

Batches of scanned returns cannot be keyed until they are assigned to a user. This page lists any and all batches that are not currently assigned. A batch is said to be unassigned if the value of *tblBatchInfo.fstrBatchUser* begins with the string ‘BATCH’. The page lists three tables of unassigned batches containing different levels of information. All the tables are filtered by the selected tax programs in the upper right. All information is SELECTed from the table *tblBatchInfo*.

The first table is a simply summary of unassigned batches. A simple SELECT is done, counting the number of batches, and summing the number of keyed and unkeyed returns. These values are then set on three Literal controls on the page.

The second table is slightly more detailed that the first, grouping unassigned batches by tax program and batch source. The same numbers are displayed, number of batches and number of returns, along with the tax year, batch source, and tax program. The resulting DataTable, returned from an instance of the godConn class, is DataBound to a Repeater on the page, detailRPT. The Repeater evaluates the column values in the ItemTemplate to create the table.

The third table offers the most detail of unkeyed batches. All unassigned batches are selected and listed. The rendering of the table is done using a Repeater and a bound DataTable, just as with the second table. The BatchID of each row is rendered as a link to **BatchModify.aspx** with the appropriate QueryString to identify the batch.

The third table also contains a checkbox for each row. The value of each checkbox is rendered in the Repeater, and consists of the batchID, batchDate and batchSource. These values are read in the click handler of the *assignBTN* button. Even though the checkboxes are standard HTML inputs, they can be read by the server-side code in the *Request.Form* object by name. Since all the checkboxes have the same name, the Form object returns an array of the selected checkboxes. The button handler checks the user name dropdown, and UPDATEs *tblBatchInfo* with the selected username, filtering by the values in each checked checkbox. The page is then reloaded, and the tables rebuilt. This assigns the batches to the selected user, and the batches will then come up for the user to key when they next load the application.

#### utils/reassignBatches.aspx

#### utils/OpenBatches.aspx

#### utils/closedBatches.aspx

#### utils/unbalanced.aspx

#### utils/batchInquire.aspx

#### utils/BatchModify.aspx

### User Management Pages

#### utils/manageUserRoles.aspx

*The functionality of assigning particular tax programs to a user is no longer used. Users are, by default, assigned all tax programs. Currently the expiration feature does nothing, and may never be implemented.*

This page serve to list all users, assign users to roles, and assign tax programs to users. The ListBox is filled on first load, simply by looping over all roles, and then looping over all users in each role. The Roles RadioButtonList is filled similarly, and the tax programs ListBox is filled off the progCodes table.

Expired users are listed with an asterisk next to their name. The button “(Un)Expire User” simply sets a date in the userExpires table. Un-expiration sets the date to ‘9999-12-31’. The displayed list can be filtered by expired status, role and/or tax program, which sets filters on the original select statement as would be expected.

The users ListBox is AutoPostback, such that when a user is selected in the list, that users role is checked against the RoleManager and selected in the radio list, as well as that users assigned tax programs (from userProgCodes) and selected in the list. Changes can then be made to the user. Changes to a user are saved via the “Update User” button, which simply updates the appropriate tables and values.

At the bottom of the page is a button to edit the program codes. This button first displays a large div to block out the page and prevent changes. It then brings up the **editPCodes.aspx** page in a modal child window. This blocks the JavaScript until the child window is closed. Once it is, the page resubmits itself with an explicit PostBack value, “codes”. While reloading like this, it is not an actual PostBack, so when the PostBack value, the page proceeds to reinitialize as if it were a first load. This will then reflect any changes to the program codes and tax programs in the list.

#### utils/userStats.aspx

This page is designed to give an idea of the volume of batches being keyed. It can be filtered by a date range, by user and by the list of tax programs in the upper right drop down list. The displayed list is broken down by tax program and user, as well as a total number of batches keyed, both for user and overall.

Loading this table is done in two parts. First the main data table for a tax program must be found. This is done by joining the following tables: rfrControlHeaders, rfrReturnHeaders, tblBatchImage and tblBatchInfo. The SQL can be found in the code. The statement will return all data tables related to a tax program.

The actual data table is in itself built in two parts. First a series of SELECT statements are UNIONed together. Each of these statements selects from a data table, filtered by the above conditions, and grouped by fstrWho. The grouping column is a record of who last touched that row, by either keying in a form or updating an existing, already keyed record. This union provides a table of the number of batches keyed by each user in each of the selected tax programs. This table is then SELECTed against, this time with grouping on fstrWho and the tax program with ROLLUP. This effectively returns the same rows in the composite table, as well as getting summations for each tax program and user, as well as a total of all the batches keyed.

The resulting DataTable is then databound to the repeater on the frontend. Via the OnItemDataBound event, the CommandArgument of the LinkButtons in the ItemTemplate is set with the username of that row. This is used when clicking on the link to select the username in the dropdown list.

When a username is selected in the dropdown list, either by clicking on a link on the initial overview list (as described above) or by the user explicitly selecting a user and submitting, the displayed table is constructed slightly differently. The references to fstrWho are replaced by either OpenedDate or fdtmWhen. This change then gives a table that displays a listing of a particular users keyed batches, grouped and organized by date entered.

In both cases, if no rows are returned the Repeater is hidden, and noUserPNL is displayed, with a message that no data was found.

### Form Building Pages

#### utils/formBuilder/existForms.aspx

#### utils/formBuilder/blockBatch.aspx

#### noFront/newBlock.ashx

#### noFront/remBlock.ashx

#### utils/formBuilder/formBuild.aspx

#### utils/formBuilder/formTest.aspx

#### utils/formBuilder/editJSauto.aspx

Many forms in most tax programs have fields that require calculations. In order to save keyers time, these fields can be calculated automatically. While the actual calculations are defined on **formBuild.aspx**, the templates for those calculations are entered on this page. The behind-the-scenes of this page is fairly straight forward. Each control is related to a column in the database table *rfrJavaAutoFunctions*. The “Add”, “Save” and “Remove” buttons simply INSERT, UPDATE or DELETE rows from this table, respectively, filtering to the selected value in *functionLST*.

These auto-calculation functions are defined as template stubs. A stub can have up to two value placeholders, and up to two name placeholders. The stub function itself can perform calculations on the value placeholders, or use any JavaScript or jQuery functions on the name placeholders. More detailed instructions are on the page itself.

There are two useful functions used to carry data across pages, P*UTVAL* and *GETVAL*. *PUTVAL* merely calls a function defined on **KeyEntry.aspx** named *d\_PUT*. This function takes a value and stores it in a numbered hidden dummy control, on the same page. The value and the number of they dummy control are passed as parameters. The values in these dummy controls are preserved across PostBacks, and thus can be retrieved in later pages of a return. They are retrieved via the *GETVAL* stub, which calls the *d\_GET* function. This function takes only one parameter, the number of the dummy control containing the desired value.

#### utils/formBuilder/needSql.aspx

This page is merely a display for the Session value *strSQL*. This value is filled by the **formBuild.aspx** page, and is an SQL statement of what needs to be changed or added to the data tables in the database. This can be either a statement to add or alter a data column to save the keyed data, or a statement to facilitate the functionality of the application, required columns, their data types, etc.

### Data Entry Pages

#### utils/entry/KeyEntry.aspx

#### noFront/imageLoad.aspx

Because most modern web browsers have a security ‘feature’ that prevents them from displaying images stored outside the root folder of a website, and the scanned images of a return are stored on a different networked server, a workaround was needed. This is that workaround, based on an article on aspsnippets.com[[1]](#footnote-1).

While the browser cannot display anything outside the sites root folder, the server has access to the network, and thus the images. Any page that uses this image loader simply use an *asp:Image* in the markup. In the code-behind, the Session value *kestrImagePath* is set to the network path of the desired image, and the *ImageUrl* property of the image is set to this page.

When the page is loaded, this page’s *Page\_Load­* is called, which checks the Session value. If it is not empty, it opens that path with a *FileStream* and *BinaryReader*. It reads the entire image, and writes it to the *Response* using *BinaryWrite*. The content type of the *Response* is set to “image”, and a header is added. This serves to work around the browser’s failing by serving the image directly to the browser, appearing as though it came from this page.

This page can also be used by placing the image file path in the *QueryString* as *fName*.

#### noFront/doLookup.ashx

This service takes an *id* and an *idtype* in the QueryString, and returns demographic information for that ID and, if existent, the demographic information of the ID of the joint filer in a JSON object. The format of the JSON return is below:

{ "primary": {

"fstrListFormatName":"",

"fstrFirstName":"",

"fstrMiddleName":"",

"fstrFamilyName":"",

"fstrSuffix":"",

"fstrStreet":"",

"fstrCity":"",

"fstrState":"",

"fstrZip":"",

"fstrCountry":"",

"fstrID":"",

"flngKey":0,

"flngJointID":0,

"fromTbl":""

},

"secondary": {

"fstrListFormatName":"",

"fstrFirstName":"",

"fstrMiddleName":"",

"fstrFamilyName":"",

"fstrSuffix":"",

"fstrStreet":"",

"fstrCity":"",

"fstrState":"",

"fstrZip":"",

"fstrCountry":"",

"fstrID":"",

"flngKey":0,

"flngJointID":0,

"fromTbl":""

}

}

The return value is a serialized version of a nested class. The nested class’ only purpose is to hold the data. This service uses a different connection string, and checks several tables for both the primary key, and the joint key (if existent). It does this in a function, which can check either the table *tblAccountInfo* or *tblCustomerInfo*. Initially, for the passed ID, it checks both tables. If the ID is not found on *tblAccountInfo*, it checks the customer table. If a joint key is found, it must be converted back to an ID via the *tblId* table. This data is returned from the initial call to the function.

If a joint ID is returned, the function is called again with that ID, and additionally with the table that the ID was found, account or customer. This ensures that the joint ID’s demographic information will be pulled from the same table. The returned data, in their respective instances of the nested class, are then serialized, and the larger JSON object is constructed and returned.

#### noFront/rateTableLookup.ashx

In the database, there is a table called *rfrTaxTable*. It lists the tax rates for a given tax year, filing status, and taxable income range (*fcurLower* < taxable income <= *fcurUpper*). This service simply looks up the tax rate based on those values, which are provided via a post of a JSON object defining the values *taxYear*, *status*, and *taxable*. After SELECTing the data from the database, the tax rate is returned via the *Response.Write* function. If the SELECT doesn’t return anything (i.e. the taxable income was outside the range defined in the table) the service returns an empty string.

### JavaScript Files

#### scripts/divMove.js

Contains the functions used in the image manipulation (moving the offset, rotation) as well as the function to move the image, marker div and control scrolling. The movement of the image, marker, and control div is controlled by values stored in the database, and passed to the functions *CtlMove* and *moveStuff* respectively. The *moveStuff* function contains logic to check if the marker or image would be moved out of view, based on *$(window)* values, and adjusts the values accordingly. This is to facilitate the different layout options (selected via **layoutStart.aspx**, defaults to ‘portrait’).

#### scripts/formattingDB.js

This file contains all the formatting functions referenced by the database table *rfrFormatTypes*, which is in turn referenced by the table *rfrControls*. All formatting is passed through the *doFormat* function. This function first checks a passed flag to determine if a lookup of the taxpayer information is needed (via **doLookup.ashx**). When that completes, it calls passed formatting function by name, with the passed object (usually *this*). The respective formatting functions are matched to data and formatting types, and process the data in the passed object via a series of regular expressions. If the data is validated, it is then formatted to look ‘pretty’, and put back into the control.

#### scripts/jquery-1.8.2.js

jQuery v.1.8.2, used across the site and in dynamic functions.

#### scripts/jquery-1.8.2.min.js

Minified version of jquery-1.8.2.js.

#### scripts/jquery-ui-1.10.3.custom.min.js

Unused.

#### scripts/jquery-ui.js

Needed for the accordion on the form building page **formBuild.aspx**.

#### scripts/jquery-ui.min.js

Minified version of jquery-ui.js.

#### scripts/jQueryRotate.2.2.js

A jQuery plugin needed for the image rotation functions. Used in the function *fncImageRotate* in the script file **divMove.js**.

#### scripts/moment.min.js

The usual JavaScript data functions weren’t returning the correct years for dates entered in MM/DD/YY format. This module provides better parsing of date strings. It is used in the dynamic function *checkDate*, stored in the database table *rfrJavaAutoFunctions*. This function is used on date fields on a form to validate they are within ±1 year from the current year, since the keyers don’t always notice what year is on the return.

#### scripts/ui.dropdownchecklist.js

Needed for the dropdown checklist of tax programs on **GenKFI.Master**. Setup includes the Check All item as the first item, and an *onComplete* function. The *onComplete* function loops over all the values in the selector, and builds a semicolon separated list of selected options. This list is stored in *tpSelHID*, a hidden control used to pass the selected values back to the server for the pages that need to filter their actions based on selected tax programs.

# Application Flow

## Overview

This section describes the flow of the application, for each of various functionalities of the application. Included is a list of values that are carried through the flow of the application, whether in the Session state or the QueryString.

### Tax Programs

In the upper left of **GenKFI.Master** is a dropdown list of all the configured tax programs. This list, an *asp:ListBox*, is filled with entries from the table *progCodes*. The checkboxes are added via a third party jQuery plugin called *dropdownchecklist[[2]](#footnote-2)*. The *onComplete* function of the plugin fires when the dropdownlist is closed. Utilizing this function, when the user has closed the dropdown, the selection are looped over, and the selected ones are added to a semicolon separated list. This list is then put into a hidden ASP control, which can then be read by the application. (The default value for both the list and the hidden control is PIT, since it is very unlikely that the PIT program would be removed from GenKFI.) Finally, when the list is closed the *\_\_doPostBack* function is called to force the application to refresh the page.

When a page first loads, it reads the value in the hidden control, and filters what the page displays. On pages that display batch information, the displayed batches are only those that are of the selected type(s). When going into the forms editor, the application will warn the user if more than one tax program is selected. The forms editor will allow editing of only one selected tax program.

### Users Login

Users don’t necessarily log into the application. When a user visits the site, their authenticated Windows credentials are passed along to the web server. The web server then checks if the user name accessing the site has been assigned a role. It does this in the function *Application\_AuthenticateRequest* located in the **Global.asax.cs** file. In this file are functions that are executed on specific events in the lifecycle of the application, regardless of page visited or other interaction by the user.

The role check is handled automatically by the built in ASP .NET Role Manager. If the user name has no role, it is put into the basic USER role, and the default page is displayed. If the user does have a role, when the default page is displayed, this role is checked, and in the case of SUPER or ADMIN roles, extra links and functionality are displayed. In order of increasing access, the roles are USER, SUPER, ADMIN, DEV.

### Data Entry

### Building Forms

### Managing Users

In order of increasing access, the roles are USER, SUPER, ADMIN, DEV. Users in roles above the basic USER role have access to the user management page, **manageUserRoles2.aspx**. This page allows users to ‘expire’ a user, which just revokes a user’s access to any of the pages in the application. This expiration is set via a DateTime column in the database table *userExpires*. Whenever the *utilClass.adminOnly* function is called (which it is at the beginning of most pages) there is a check against this DateTime to the current date and time. If the expires date is in the past, the user is redirected (every time) to an error page, and is shown a message that their access has been restricted.

On the user management page, access levels can also be changed. However, a user can never grant higher access than they themselves have. So, for example, a SUPER role user can never grant ADMIN access to another user. Note that with this setup, only DEV users can grant other users the DEV role. DEV access was originally granted by hand by editing the Role Membership tables directly. For more information on ASP .NET Roles Management, see <https://msdn.microsoft.com/en-us/library/ff647401.aspx>.

Originally, users were also restricted to which tax programs they could view and key, which would have been managed via the database table *userProgCodes*. However, it was ultimately decided that all users should be allowed to see all tax programs, and this table remains unused. The page **manageUserRoles.aspx** was originally written with this capability in mind, and currently remains unused.

### Managing Batches

When returns are scanned, they are scanned in batches, usually 25 returns to a batch. Batches can be smaller however, as when a batch is formed from rejected 2D bar codes. Originally, when a batch is scanned and extracted into GenKFI to be keyed, it isn’t assigned to anyone. These new batches are listed under the **UnAssignedBatches.aspx** page. The batches are pulled from *tblBatchInfo*, and displayed here. At the top of the page is a total count of the batches, unkeyed returns (or Header Count), and the completed or keyed returns (Item Count). Below that is a slightly more detailed breakdown of the batches into type/tax program and tax year, again with the same totals. These are all simply pulled from *tblBatchInfo*, with various GROUP BY conditions. For more information on this table and the batch information structure, see the Batches section under Database Structure.

Below this is a table of the actual batches that need to be assigned. Each row in the table is one batch, listing various information about the batch. The leftmost cell of this table contains a checkbox, whose value is set to identifying keys for a specific batch. All the checkboxes have the same name. The user can check which ones should be assigned to a user. When the ‘assign’ button is clicked, the application checks the form via Request.Form with the IDs of the checkboxes. Because the checkboxes have the same name, this results in a comma separated list of the values of the checked checkboxes. The batch keys are extracted, and the fstrBatchUser is set in *tblBatchInfo* to indicate who is assigned this batch.

A similar functionality exists for open, assigned batches (**OpenBatches.aspx**), closed batches (**ClosedBatches.aspx**), and the page to reassign batches (**ReassignBatches.aspx**). On all these pages, the same information about the batches is displayed, with the same keys linked to the checkboxes. Then, depending on the pages’ function, a flag is set or toggled in *tblBatchInfo*. In **OpenBatches.aspx**, fstrBatchStatus is set to ‘C’ for closed. In **ClosedBatches.aspx**, the same fstrBatchStatus, is set to ‘O’ for open. Closed batches are imported into GenTax nightly. On **ReassignBatches.aspx**, the user selects a first user, and the page loads all the batches assigned to them by filtering on fstrBatchUser. The user then selects a second user to assign the batches to, and the column fstrBatchUser is set to the second user’s user name.

### Admin vs User

Below are the pages each role can access. Note that while **imgLoad.aspx** is never visited directly, it is used as the source URL of an *asp:Image* control.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **USER** | **SUPER** | **ADMIN** | **DEV** | **Pages** |
| X | X | X | X | Default.aspx |
| X | X | X | X | noFront\imageLoad.aspx |
| X | X | X | X | utils\entry\KeyEntry.aspx |
|  | X | X | X | utils\BatchModify.aspx |
|  | X | X | X | utils\manageUserRoles2.aspx |
|  | X | X | X | utils\unassignedBatches.aspx |
|  | X | X | X | utils\reassignBatches.aspx |
|  | X | X | X | utils\batchInquire.aspx |
|  | X | X | X | utils\closedBatches.aspx |
|  | X | X | X | utils\userStats.aspx |
|  |  |  | X | utils\formBuilder\blockBatch.aspx |
|  |  |  | X | utils\formBuilder\editJSauto.aspx |
|  |  |  | X | utils\formBuilder\existForms.aspx |
|  |  |  | X | utils\formBuilder\formBuild.aspx |
|  |  |  | X | utils\formBuilder\formTest.aspx |
|  |  |  | X | utils\formBuilder\needSql.aspx |
|  |  |  | X | utils\errView.aspx |
|  |  |  | X | utils\autoInstall.aspx |
|  |  |  | X | utils\preAutoInstall.aspx |

These permissions are checked in *utilClass.adminOnly*. This function is called in the *Page\_Load* of the relevant pages. The title of the page is passed to the *adminOnly* function, which is then checked against a list of pages only SUPER or higher can access. This list is filled in the constructor of the utilClass.

Additionally, in a previous version users in the ADMIN role were able to ‘impersonate’ another user. This was designed to be used in debugging, allowing an ADMIN user to see exactly what another user would see, allowing them to attempt to reproduce an error. It was later decided that this functionality wasn’t needed, and the controls were removed from the **GenKFI.Master** page. The code and the controls are still in the project however, and the visibility only needs to be set back to true, if it is decided at a later date that this functionality is again wanted.

### Installation of New Tax Programs

AS OF 2015-08-04 THIS IS AN EXPERIMENTAL FEATURE THAT HAS NOT BEEN THOROUGHLY TESTED.

New tax programs or tax years are first developed in the test environment. This prevents mistakes from showing up in production. However, once a new tax program or tax year is ready for production, there are many database rows, and perhaps new tables, that need to be copied from the test database to the production database. This tool is meant to help with that transition. Note, only the DEV role has access to this page.

In both databases, there is a table called *rfrGkInstall*. In the **notes** folder in the GenKFI project folder there is a file called **autoInstallMappings.docx**. From a high level overview, this tool copies the relevant configuration rows from one tax program/year into the *rfrGkInstall* table. Since different required tables will probably have differently named columns, the Mapping document outlines which columns from which tables are in which columns of the install table.

The data from each required table can be generated individually, to reduce the risk of error. Alternatively, once it is verified that this tool is working correctly, all tables can be pulled into the installer table at once. Once all the required and relevant data has been put into the installer table, the data can be copied out and inserted into the production installer table.

The process is then reversed, pulling data out of the installer table and inserting it into the required tables in production. Because auto-generated keys cannot be guaranteed to be the same between test and production, anywhere an auto-generated key exists, the generated value is saved. Anywhere that key is referenced (e.g. foreign keys), the new value will be used.

# Database Structure

## Overview

Below are descriptions of the database structures relating to major areas of the application. Each section lists both the involved tables and relations between those tables, along with notable columns.

### Users

The users in GenKFI are handled mostly by the automated Role Manager via ASP .NET. There are however, a few custom tables.

Since the application relies on integrated windows authentication, there is no need to use the ASP .NET Membership Manager. The tables used by the Role Manager are *aspnet\_Users*, *aspnet\_Roles*, and *aspnet\_UsersInRoles*. The Users and Roles tables provide a GUID for each user name and role name. The UsersInRoles table simply links these GUIDs together, linking each user with one or more roles. This is all handled automatically by the Roles Provider.

The custom tables also use the generated GUIDs for the users. The first, *userExpires* keeps track of when a user is ‘expired’ (see ‘Managing Users’ above). When a user is ‘expired’, the expires column is set to the current date time.

The second table, *userProgCodes*, is no longer used. Its intended purpose was to limit what tax programs a user could see, and by extension which tax programs a user would be able to key.

**Batch Table Columns**

**tblBatchImage**

fstrTaxProgram

fstrBatchSource

fstrBatchID

flngSequence

fdtmScanDate

fstrImageDocID

fstrID

fstrID2

fstrID3

fstrDLN

fstrFormName

fstrDir

fstrFile

fstrImagePath

fstrBatchName

flngPosInBatch

flngImgWnd

flngCamera

fstrBarData

flngLevelID

fstrFormCode

fstrFormOCR

fstrFormIdentified

fstrFormConfirmed

fstrVendorCode

flngImageWidth

flngImageHeight

flngImageSize

flngImageDPI

fblnImageFolderMissing

fblnImageFileMissing

fblnImageBlankSizeCheck

fblnImageBlankSizeCompare

fblnImageBlankMiniCheck

fblnImageBlankUserCheck

fdtmImageCheckedOut

fdtmImageCheckedIn

fstrStatus

fstrWho

fdtmWhen

fblnPtdAddition

**tblBatchInfo**

fstrBatchID

fdtmBatchDate

fstrTaxProgram

fstrBatchSource

fstrBatchType

fstrForm

fstrDLN

fintBatchCount

fcurBatchAmount

fintHeaderCount

fcurHeaderAmount

fintRejectCount

fcurRejectAmount

fstrBatchUploaded

fdtmBatchUploaded

fstrPaymentUploaded

fdtmPaymentUploaded

fdtmBatchCreated

fdtmReceivedDate

fdtmDepositDate

fdtmPostmarkDate

fstrBatchStatus

fstrBatchUser

fstrTimelyBatch

fstrImageUploaded

fdtmImageUploaded

fstrWho

fdtmWhen

fdtmAssigned

### Batches

The scanned batches are handled via two tables: *tblBatchInfo* and *tblBatchImage*. They are filled via a nightly job in GenTax, which runs the BO NM\_raGmi. A listing of all the columns in each of these tables is included above.

The first of these table *tblBatchInfo* acts like a header for the entire batch. Most of this data is filled by NM\_raGmi and needn’t be worried over. The columns that are of interest are:

* fstrBatchID
  + This column is that is used to trach the batch through the system, from who is assigned the batch to tracking the stored data in the data tables (below)
* fstrBatchSource
  + This is the source of the batch. The source can be ‘GENKFI’, in which case it was scanned specifically for GenKFI, or ‘2D’ indicating it has a 2D barcode that failed to be read.
* fstrTaxProgram, fstrForm
  + These columns together indicate the tax program and the tax year. This is used when building the key entry page by identifying exactly which control definitions to use (for more information see the ‘Controls” section below).
* fstrBatchUploaded
  + This is an indicator if the data from this batch has been loaded into GenTax. That process runs as a nightly job, which runs the BO NM\_raGre. Its values can be either ‘Y’ or ‘N’.
* fintHeaderCount, fintBatchCount
  + These columns are counts of the number of returns scanned in the batch and the number of returns that have been keyed, respectively. The number of returns keyed is updated as users enter complete returns. They are mostly for informational uses.
* fstrBatchStatus
  + This is the status of a batch. Its default starting value is ‘O’, indicating open batches. This is the used in the WHERE clause on the **OpenBatches.aspx** page. Once a batch has been keyed, an admin or supervisor can close the batch on that same page. When that is done, this column is set to ‘C’. The nightly job that loads the data into GenTax sees this status, and knows that batch is ready to be loaded. Sometimes a batch needs to be deleted and rescanned. When this happens, the batch isn’t really deleted from the system. This column is simply set to ‘D’, for deleted. Then we still have the information, but the batch is ignored by the system.
* fstrBatchUser, fdtmAssigned
  + When a batch is ready to be keyed, a supervisor will assign it to a user. These columns track batch assignment. This is just the user’s windows logon and the date and time it was assigned. *fstrBatchUser* is also used when a user begins keying to find which batches they should be presented with.

The second table involved with batches is *tblBatchImage*. Where *tblBatchInfo* acted like a header for the batch, this table contains information on the actual forms scanned, including the file names and paths to the actual images. Many of the columns in this table are also related to the queue and the actual keying process, so the discussion here will relate only to the information needed to identify a batch item.

* fstrBatchID, fstrTaxProgram
  + These are the same as in *tblBatchInfo*. They are used to link the actual image scans back to the header info. This provides a way for the application to know where to store any keyed data.
* flngSequence
  + Actual returns in a batch are tracked with this sequence number. Where the batch ID helps keep batches together, this sequence number helps keep individual returns together.
* fstrFormName
  + Based on a barcode on each scanned form, the nightly job that loads batches into GenKFI flags each image. When keying, this information is used to determine which digital form to load and present to the user.
* fstrDir, fstrFile, fstrImagePath
  + These indicate exactly where on the network the actual image files can be found. *fstrImagePath* is the full path, including the filename.
* fstrDLN
  + Every image scanned gets assigned a unique DLN number. This is used in GenKFI to distinguish scans of repeated forms. For example, PIT returns can list multiple dependents on an additional page, possibly more than one page. By the barcode, the system would only see one form, but by also using the DLN number, the two separate forms can be distinguished.

### The Queue

The Queue is a system for determining how to present returns to the user to be keyed. It works on several levels, each of which will be addressed. The Queue determines what order batches and returns are presented, it keeps track of where a user is in a particular return, as well as allowing them to return to the same point they left off, should something happen to their connection or browser. It is also used in the ‘key mode’ function, which takes users to the next batch or return in their assigned queue without having to click through the batch and sequence links.

The Queue works via the interaction of data on four tables in the database: *tblBatchInfo*, *tblBatchImage*, *tblKeQueue* and *tblKeQMast*. As each aspect of the Queue is discussed, relevant tables and columns will be mentioned.

### Controls

The information for the building the dynamic HTML forms is spread across 3 tables, *rfrReturnHeaders*, *rfrControlHeaders*, and *rfrControls*. It is structured this way to prevent needlessly duplicating information. For example, every HTML control for the PIT, Page 1 form will be associated with that form, and furthermore with the PIT tax program. There is no need to repeat this information for every control, when we can split it out to its own table.

The first of these tables, *rfrReturnHeaders*, is the high level indication of what tax programs and years have controls. This table has a unique ID for each row, and contains the file type, the tax program and the tax year.

Each page of a form may have multiple blocks of HTML controls for keying data. See for instance the PIT, Page 1 form. It has three different sections of differing data: the taxpayers’ information, dependent information, and income information. Each of these sections in GenKFI is broken into separate keying blocks. The information for each of those blocks is stored in the second controls table, *rfrControlHeaders*. As with *rfrReturnHeaders*, this table has a unique row ID for every row. This table also has a row, flngReturnHeadersIDFK, which references *rfrReturnHeaders*, indicating which tax program each block is for. This column acts as a foreign key (thus the FK), but isn’t actually set as one. This table also contains a column for the tax form, which is referenced with the fstrFormName in *tblBatchImage* to match the control blocks to the correct images.

Below are additional noteworthy columns in this table.

* fstrTable
  + This is the table where information keyed on this control block will be stored
* fstrTemplateImage
  + This is an image of a blank form that is used in the GenKFI form builder tool.
* fstrFormType
  + This is the type of form. This determines how the form is rendered to the user. Possible values are:
    - ‘N’, a normal form
    - ‘R’, a repeating form (i.e. a form with many possible lines)
    - ‘P’, a form that could have multiple pages of the same paper form.

One thing to note with the form blocks, is that the block actually used to render the controls to the user is block 1. Block 0 is reserved as a header block, and is used in the GenKFI form builder tool. Thus even if there is only one block of controls for a given paper form, there will be two entries for that form in this table.

Lastly is the table *rfrControls*. This table stores specific information for each HTML control. As with the other tables, there is a unique ID column for each row. Listed below are relevant columns. Columns not listed either aren’t used, or are legacy leftovers, with the data they store moved into one of the tables previously discussed.

* flngControlHeadersID
  + This is a reference back to *rfrControlHeaders*, indicating which block this control is associated with. Again, this acts like a foreign key, but isn’t actually configured as such.
* fstrDocName
  + This is the name of the field where the information is stored in GenTax.
* fstrInputID
  + This is the HTML ID of the rendered control.
* flngOrder
  + This simply gives something to keep the rendered controls in the correct order.
* fstrColumn
  + While the table was specified in *rfrControlHeaders*, the actual column to store the keyed information is saved here.
* flngDispLength, flngMaxLength
  + These are set on each rendered control to specify the display length and the max length of keyed data. The max length prevents the user from entered too many characters, while the display length is sometimes larger to accommodate for formatting.
* fblnHidden
  + If this is set, the HTML control is rendered as a <hidden> control. This is useful when GenTax requires a field that isn’t on the paper forms, and is usually set from a default value on the data table when the user first enters this block (see the section on the Queue).
* fblnRequired
  + If set, the user must enter information in this control before they are allowed to move on to the next block.
* fblnVertSkip
  + If set, this control isn’t rendered on a new row in the final output table, but rather in the same <tr> element as a new <td> element.
* fstrLabel
  + The rendered label placed next to the HTML control.
* flngCtrlMove
  + This value indicates how far down to scroll the panel containing the HTML controls. This is useful when there are many controls in a single block, and allows the controls to remain visible without the user having to do it manually.
* flngImgMoveX, flngImgMoveY
  + These values are the X and Y position to move the image. This keeps the data in the paper form lined up with the associated HTML controls, showing the user exactly what to enter.
* flngMarkerMoveX, flngMarkerMoveY
  + These values are the X and Y position to move the marker. This further clarifies exactly which data on the paper form the user should capture.
* fstrNextCtrl
  + When a user presses enter on a rendered control, this indicates which control should next receive focus. This value is one of the values in fstrInputID. If this is the last control in a block, this should be ‘”btnSub” which will move the focus to the Submit button.
* fstrTranslate
  + Since radio button controls aren’t used in the GenKFI keying process (it would take too much of the keyers time to find and click a radio button), selections and checkboxes on a paper form are translated to a single character text box, where the label indicates a number for each option on the paper form. Based on that number, the system translates it back to the selection based on this column. This value is a comma separated list of the selection options.
* flngFormatIDFK
  + This value references the unique key in *rfrFormatTypes*. That table is a list of formatting functions (written in JavaScript) that are included on the **KeyEntry.aspx** page. When a control loses focus, the referenced function is called, and the entered data is formatted and, in some cases, validated.

### Data

Each form in a tax program is associated with a table in the database. Multiple forms could possible use the same data tables, but column names would have to be carefully checked so data is not overwritten. When designing forms, each form is given a table name. When defining controls, each control has a column name. Together, these tell the system where to store the data. This is a fairly straightforward association, making sure every control on every form has a column.

There are a few things, however, that much be in every data table. To be able to track which data came from what return, every data table must have the following columns: fstrBatchID, fdtmBatchDate, and flngSequence. These are the same values as those from *tblBatchInfo* that are associated with this return. Additionally, if the form is a repeater form, three more columns are required: flngRepNum, fblnFilled, and fstrTaxForm. These are for the system to track how many repeated lines have been entered and keyed. Finally, if a form is a multi-page form, fstrMPDLN must also be included. This is used to track the image itself, since there may be many images of the same form.

Lastly, every data table needs fstrWho and fdtmWhen.

1. http://www.aspsnippets.com/Articles/Displaying-images-that-are-stored-outside-the-Website-Root-Folder.aspx [↑](#footnote-ref-1)
2. http://dropdownchecklist.sourceforge.net/ [↑](#footnote-ref-2)