# Introduction

**Knack Toolkit Library**

v0.5.0 - pre-release

Knack Toolkit Library, henceforth referred to as **KTL**, is a collection of open-source Javascript utilities that eases Knack application development and add several features that are not easily created from the ground up. Those features that involve using the Knack API are 100% view-based, so your API key is never exposed.

\*\*\* NOTE\*\*\* The code and documentation are currently in a pre-release phase. Thank you for being patient and understanding that everything may not be complete or perfectly organized.

# Overview

## Zero config needed for Basic Features

Right out of the box, without any coding or complex setup, the KTL will provide many nice additions to your app:

* user filters to save your filters to named buttons
* form data persistence that saves as you type, and will load back your data if a page is reloaded after a submit failure or power outage
* sorted menus
* lightly colorized inline-editable fields for easy identification
* special keywords in the view’s title to trigger
  + auto-refresh of tables, details view or other views
  + hidden views
  + hidden titles
  + disable inline editing
* Ctrl+Click on a table header to invert the default sort
* idle timeout watchdog
* spinner timeout watchdog
* numeric pre-validation
* force uppercase on desired fields
* auto-focus on first field of a form or search field in a table
* dropdown selector improvements
* kiosk mode
* debug window for embedded devices

Click the following link if you’re interested to know more about [Advanced Features](#_Advanced_Features).

# No time to read all this now. How about a quick tryout?

If you want to try/use the basic, default setup version of the KTL, all you need to do is copy the content of those two files: [**KTL\_KnackApp\_ACB.js**](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/KTL_KnackApp_ACB.js) and [**KTL.css**](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/KTL.css) to their respective panes in your Builder. If you already have your own code, it will not conflict with the KTL. Just move it between these lines at the end:

//My App code - BEGIN

// .....your code here....

//My App code – END

For your CSS code vs KTL’s, the placement does not matter. But it is recommended to keep each of them grouped together, with clearly identified delimiters.

If you don’t like a feature, don’t worry. It’s possible to turn it off by setting its flag to false in the function **ktl.core.setCfg**, in the **//KTL Setup** section of the **KTL\_KnackApp.js** file.

# Features

The code is organized by specific feature categories, and here's the complete list:

* Bootloader
* Core
* Storage
* Scenes
* Views
* Fields
* Bulk Operations
* User Filters
* Form Persistence
* Accounts
* User Preferences
* iFrame Window
* Debug Window
* Logging
* Windows Messaging
* System Info
* System Colors

In the next section, we’ll go through each one and see what they can do, with the list of all available functions to you as a developer, if ever you’re interested in trying out the more advanced features.

## Bootloader

### Usage

The bootloader is the entry point of all code, including the KTL and your app-specific code. It is very technical, and the average user will not really need to understand it. Basically, it does two things:

### External library loading

First, I wish to say a big *"thank you"* to [**Soluntech**](https://www.soluntech.com/) for their gracious permission to use a portion of their code to manage the dynamic library loading. This is the cornerstone that allowed kickstarting the KTL project. In short, it uses a list of libraries your app will need, and automatically loads them with the LazyLoad function. Again, you won't need to understand how it works since the setup is already done.

### Developing your code locally – aka [CLS mode](#_How_to_use)

Traditionally, i.e. without the Bootloader, all your app code resides in the Builder's Javascript and CSS panes. But if you leverage the Bootloader, you’ll be able to load your Javascript and CSS code from your hard drive instead, at run-time. This means you can now code and save directly on your workstation, without having to copy/paste the code to the Builder every time you want to test a change.

This mode enables you (the developer) to work more efficiently by using your favorite code editor with all its bells and whistles, instead of the basic Builder's Javascript pane editor. You must install **Node.js** (<https://nodejs.org>) on your computer and run the **NodeJS\_FileServer.js** script provided. Then, each time you save your code, all you have to do is refresh the browser to see the changes take effect immediately. In this mode, writing and testing code simply won’t ever get any faster.

Another great advantage is that it opens the possibility of teamwork. Currently, only one developer at a time can edit the code. With the bootloader and Node.js file server, there is no conflict because each developer works with his own "sandboxed" local copy and pulls external changes whenever he/she chooses to do so.

**Furthermore and most importantly**, you can simultaneously work on a production app, running officially released and tested code, while you run development code locally – yes, two different code revisions at the same time, without conflicting! This means that the classic method of going in Knack’s Builder to create a sandbox copy of your app to develop and experiment with new code is not required anymore - or a lot less often. When your app becomes huge, creating a copy can take a whole day sometimes.

### Functions

* Bootloader has no exposed functions.

## Core

### Usage

This contains generic utility functions, used by most other features internally, but also available to your app.

### Functions

* **setCfg**: This is where you can enable the features you want.
* **getCfg**: To get the config and read the flags.
* **knAPI**: Knack API wrapper with retries and error management.
* **isKiosk**: For support of kiosk mode applications. You decide the trigger conditions for kiosk mode in a callback function.
* **hideSelector**: To move away elements off the screen to hide them or save real-estate.
* **waitSelector**: When you need to wait until an element exists or is visible.
* **waitAndReload**: Waits for a specific delay, then reloads page.
* **switchVersion**: To quickly toggle between production and development versions.
* **enableDragElement**: To add basic drag and drop to an element.
* **splitUrl**: Creates an array containing the path and parameters of the URL.
* **getMenuInfo**: Retrieves the menu and sub-menu items.
* **isHex**: For validation of hexadecimal strings.
* **ipFormatOk**: For IP format validation.
* **getSubstringPosition**: Returns the index of the Nth occurrence of a string within a string.
* **addZero**: Adds leading zeros to 2 or 3-digit numbers, typically for logs alignment.
* **getCurrentDateTime**: Generates a local or UTC date/time string.
* **dateInPast**: Compares the first date to the second one and returns true if it's in the past, ignoring the time component. If a second date is not provided, it uses today.
* **selectElementContents**: Selects all element's text.
* **timedPopup**: Generates a brief, auto-delete popup with status text and color.
* **removeTimedPopup**: To remove the timedPopup.
* **infoPopup**: Similar to timedPopup, but without an expiration delay. Removal must be done manually. Useful for progress indicators.
* **setInfoPopupText**: To indicate general information, status, or progress in infoPopup.
* **removeInfoPopup**: To remove infoPopup.
* **insertAfter**: To insert a node after an existing one, but as sibling, not as a child.
* **setContextMenuPostion**: Upon right-click, ensures that a context menu follows the mouse, but without overflowing outside of window.
* **getObjectIdByName**: Pass the object’s name and returns the object’s ID.
* **getFieldIdByName**: Pass the field name and object ID and will return the field’s ID.
* **getViewIdByTitle:** Pass the view title (and Page URL optionally) and returns the first view ID containing specific text in its title, with optional exact match.
* **sortMenu**: Will sort the menus in alphabetical order when top-menu is opened.
* **sortUList**: Will sort any un-ordered list in alphabetical order.

### A note about knAPI

While Inline editing is mandatory for PUT operations on a table, it may not be desirable to let users modify data manually. You can disable these edits dynamically by adding the view title flag **NO\_INLINE**. This allows the API calls to work properly, while disabling the mouse actions.

## Storage

### Usage

Provides non-volatile storage utilities using the localStorage and cookies objects. It is the cornerstone of several other features.

### Functions

* **hasLocalStorage**: Returns whether or not localStorage is supported.
* **lsSetItem, lsGetItem, lsRemoveItem**: Saves, loads, and deletes text item in app-specific keys.
* **saveUserSetting, loadUserSetting, setCookie, getCookie, deleteCookie, deleteAllCookies**: Same but using cookies.

## Fields

### Usage

Provides field-related features like auto-select all text on focus, convert from text to numeric and enforce numeric validation or uppercase letters.

### Functions

* **setCfg**: Set all callbacks to your app, like keypress event handler and field value changed. Use the textAsNumeric array to specify which fields must be considered as numeric even though you have set them up as Short Text in Knack. This can be very useful in some special use cases. For example, you can dynamically change this to allow a unique Account Role to enter letters, while all others can only enter digits.
* **convertNumToTel**: All numeric fields will automatically be converted to telephone type. This has no negative or perceptible impact for all users, except that it allows mobile devices to switch the keyboard to telephone type for a more convenient numeric layout and also auto-selection of all text upon focus.
* **enforceNumeric**: For all numeric fields, and any specified in textAsNumeric, validation will be performed. If non-numeric values are found, the submit button will be disabled and grayed out, and the field will be colorized with Knack's "pink" error indicator.
* **addButton**: Will add a button to a specified div element. You can specify the label, style, classes and ID, and it will return a button object to which you can attach your event handlers.
* **addCheckbox**: Similar to addButton, but for a checkbox.
* **addRadioButton**: Similar to addButton, but for radio buttons.
* **Barcode reader specific functions**: **addChar, clearBuffer, getBuffer, setUsingBarcode,** and **getUsingBarcode**. Useful in the context of business and industrial projects.
* **addChznBetter**: The chznBetter object is a custom solution that fixes a few problems with the Knack dropdown object. The most annoying being the following: When you have more than 500 items in the list, the dropdown switches mode and displays a search field. But most of the time, when 3 or 4 characters are typed, the last one or two are erased, rendering the selection very tedious. I got so many complaints about this that I decided to code my own solution. As a bonus, you can now customize the delay before the search starts (common to all dropdowns), and for individual dropdowns, the threshold number of characters to type before the search starts. Defaults are 1.5 seconds delay, and 3 characters for short text fields and 4 for textAsNumeric fields. Use the ktl.fields.setCfg function to modify chznBetterSrchDelay and chznBetterThresholds to match your needs.
* **searchChznBetterDropdown**: chznBetter's wrapper to searchDropdown. Mainly used internally, but accessible to your app in case of specific needs.
* **inlineEditChangeStyle**: To dynamically modify of an inline edit cell, typically to make it wider to more text. Not completed, work in progress.
* **onFieldValueChanged**: Callback to your app to process value change events for Dropdowns and Calendars. Driven by processFieldChanged function. More field types will be added eventually.
* **getFieldFromDescription**: Returns an object with the field ID and view ID of a field containing specified text in its description.
* **getFieldDescription**: Returns the description text from the field ID parameter.

### Using field’s Description text box as flags to trigger special behavior

In the Builder, when you edit a field in the schema view, there’s a Description text box, where you can put your own notes, as a developer. Now, this can also be used by the KTL to trigger special behavior. You can add the flag at the end of your description, or on a separate line, as you wish. Here’s the list:

* **TO\_UPPERCASE**: to convert text to uppercase in real-time

## Views

### Usage

Provides view-related features.

### Functions

* **setCfg**: To setup your parameters and callbacks to your app. Callback processViewFlags allows you to process your own special title flags.
* **refreshView**: Robust view refresh function with retries and error handling. Supports most types of views including tables, details, searches, forms, rich text, and menus.
* **refreshViewArray**: Calls refreshView for each view in the array of view ids as parameter, and returns (resolve from promise) only when all are done refreshing.
* **autoRefresh**: You can now add auto refresh to any view without a single line of code. It is done from the Builder, by simply adding AUTOREFRESH=30 at the end of your view's title and it will refresh itself every 30 seconds. Values from 5 (seconds) to 86500 (24 hours) are accepted. Of course, the flag is truncated so only your title remains visible. Also, you can start and stop the process at will by setting the run parameter to true/false. Note that this will apply to all views in scene.
* **addViewId**: Convenient for developers who want to see the view id next to or near the title.
* **addCheckboxesToTable**: Will add checkboxes to a table, including the top one in the header to check all at once. Used by bulk operations.
* **addTimeStampToHeader**: Useful to see when the last refresh date/time occurred and assess that your app is running smoothly.
* **hideField**: Moves a field away from the screen to hide it or save space.
* **searchDropdown**: Searches text in a dropdown or a multiple choices object, with these options: exact match, show popup for outcome. Currently supports 3 different states of the dropdown: single selection, less than 500 and more than 500 entries. Will auto select the found result it it’s an exact match. Otherwise returns all found items and lets you manually choose from the results list. Multiple selections are more complex and will be supported eventually.
* **findInSearchView**: Uses a Search view to find text, with exact match. Very useful to prevent duplicate entries on a connected field, for example, by doing a hidden search on that view before submitting a new connected record.
* **removeTableColumns**: Will hide or remove columns from a table. Pass it an array of field ids, and/or array of columns indexes to remove. Also works with action links, which is useful to remove a Delete action if the logged-in role shouldn't be allowed for example.
* **findFirstExistingField**: Pass a list of field IDs and returns the first found in scene.
* **modifyTableSort**: Inverts the sort order if the data type is Date/Time. In several apps, I found that users always need to click the header twice because they want to see the most recent entries. You can also do a Ctrl+Click to sort it ascending like it is now.
* **submitAndWait**: Pass a form’s view ID and an object containing pairs of field IDs and values. It will fill in the form and submit automatically, then return with a success or failure outcome. If successful, the resulting record is also returned and can be used for further processing.
* **updateSubmitButtonState**: Used to perform real-time form validation, i.e. before Submit is clicked, by enabling or disabling the button based on your criteria. Pass the form’s view ID and it will enable or disable the Submit button. This status extends the existing **validity** property of the button, by adding the **invalidItemObj** object to it. When this object is empty, Submit is enabled, if it contains any key, it will be disabled.
* **ktlProcessViewFlags**: This is an internal function that is not exposed. But worth some additional explaining, nonetheless. It parses the view's title for special flags. Here's the list:
  + AUTOREFRESH=[value] To refresh the view every [value] in seconds
  + HIDDEN\_VIEW To hide the view away from screen, but still existing
  + HIDDEN\_TITLE To hide the view title only to save real-estate
  + ADD\_REFRESH For Kiosk mode only, adds a Refresh button
  + ADD\_BACK For Kiosk mode only, adds a Back button
  + ADD\_DONE For Kiosk mode only, adds a Done button
  + NO\_INLINE Disables inline editing, even if enabled in Builder
  + USER\_FILTERS\_MENU Used in the page to save/load filters to the Cloud
  + USER\_FILTERS\_CODE Used in the page to save/load filters to the Cloud

You can also add your own app-specific flags in the callback function processViewFlags.

## Scenes

### Usage

Provides scene-related features.

### Functions

* **setCfg**: To set up your parameters and callbacks to your app.
* **getCfg**: To read the idle watchdog delay value.
* **autoFocus**: Callback to your app's handler of autoFocus. By default, Knack does not set the focus to a field. But this enables you to choose when and how to do it – your way.
* **renderViews**: Renders all views in the current scene.
* **addKioskButtons**: In kiosk mode, most of the time there is no mouse or keyboard. This enables adding specific buttons, typically for navigation: Back, Done, Refresh. I've also added Work Shift and Messaging buttons, if ever you need them (more information provided upon request).
* **spinnerWatchdog**: This is a timer that checks if the app is in a waiting state. If the spinner takes more than a specified amount of time (default is 30s), you can gain back control, typically by reloading the page. Most of the time, this solves the "infinite waiting" problem after a Submit or any page load/refresh, especially for kiosks without a keyboard, where users would otherwise have to reboot the device. After quite a bit of experimentation, I was surprised to observe that a submitted form was indeed sent successfully, but it was the screen refresh that never came back. This forced refresh solved the problem at least 90% of the time.
* **isSpinnerWdRunning**: Returns true if page is busy and spinner is shown.
* **flashBackground**: Simple attention getter, useful on small devices monitored from a distant area, to show status like success or failure.
* **resetIdleWatchdog**: The idle watchdog is an “inactivity timer”. Each time a mouse click/move or a key press is detected, this is called. After a given amount of time without activity, the idleWatchDogTimeout callback (below) in your app allows a specific action to take place.
* **idleWatchDogTimeout**: The idle callback to your app, typically for reloading the page or logging out the user.
* **findViewWithTitle**: Searches through each view in the current scene and returns the first view ID containing specific text in its title, with optional exact match.
* **scrollToTop**: Scrolls the page all the way up.
* **addVersionNumber**: Adds the app and optionally the KTL version numbers on the page.
* **isiFrameWnd**: returns whether the window is the top-level app, or the hidden child utility page called iFrameWnd.
* **onSceneRender**: Callback to your app's handler of a “knack-scene-render.any” event.

## Form Persistence

### Usage

When user types-in data in a form, values are saved to localStorage and restored in case of power outage, accidental refresh, loss of network or other mishaps. **Data is erased** when the form is **submitted** **successfully,** or user **navigates away** from page.

### Functions

* **setCfg**: To define scenes and fields to exclude, i.e. that are never saved.

## User Filters

### Usage

When "Add filters" is used in tables, it is possible to save each one you create to a named button by clicking on the yellow diskette save icon. Your filters are saved in localStorage but can be saved/restored to/from Knack for backup or migration to other devices, provided some additional setup. See the [User Filters setup procedure](#_User_Filters).

Each active filter will not only remember its filter parameters, but also the column selected for sorting, the sort order, and the searched text. You can rename and delete buttons at will, and you can also drag and drop the buttons to re-order them at your convenience. The button colors will have matching variations based on the app's theme. Each scene remembers the last active filter when you go back to it.

The User Filters feature is enabled by default, but you can disable it by setting the userFilters flag to false in the ktl.core.setCfg function.

### Functions

* **setCfg**: When User Filters are enabled with the main flag, it is possible to use the allowUserFilters callback to your app to disable it based on specific conditions. Ex: Kiosk mode devices usually don’t have filters enabled, while all others do.

## Bulk Operations

### Usage

There are two types of bulk operations: Bulk Edit and Bulk Delete. As their names imply, they provide the ability to perform several record modifications or delete operations in batches. Both work with table views and have a global flag to enable each of them separately.

### Bulk Edit

To use this feature, you must:

1. Enable the bulkEdit flag in the ktl.core.setCfg function
2. Create an account role named "Bulk Edit" and assign it diligently to very trusty and liable users.
3. For each applicable table, enable Inline editing and be sure to disable all the fields that should be protected against unintended modifications.

These field types are supported: all text fields, connected fields, date time picker, Yes/No, multiple choices (single selection and radio buttons only at this time).

Usage: In the table, select all the checkboxes for the records to be modified. Then click on any cell to edit its value (inline). After submitting the change, a prompt will ask you if the value should also apply to all selected records. Click yes to apply to all. A confirmation message will pop up after completion.

\*\* Important note\*\* the table's sort may cause your changes to disappear due to being out of scope. This is normal. You can prevent this by first choosing a sort that will not cause this, or filtering as much as possible to show a very restricted number of records, but still including the ones you need. Experimenting with only a few records at a time (less than 10) is recommended as a starting point. If you've made an error, the process can be interrupted (but not undone) at any time be pressing F5 to reload the page.

### Bulk Delete

To use this feature, you must:

1. Enable the bulkDelete flag in the ktl.core.setCfg function
2. Create an account role named "Bulk Delete" and assign it diligently to very trusty and liable users.
3. For each applicable table, a Delete action link must be added.

You will see two buttons appear:

* **Delete Selected**: Is enabled when at least one record is selected
* **Delete All**: Is enabled when "Add filters" is used. The checkboxes are ignored, and the process will keep deleting records until none is left, flipping through pages automatically.

If you've made an error, the process can be interrupted (but not undone) at any time be pressing F5 to reload the page.

### Functions

* **deleteRecords**: Used internally by bulk delete to delete an array of records but may be used elsewhere by your app if ever needed.

## Account

### Usage

Provides features for the currently logged-in account.

### Functions

* **isDeveloper**: Check if the list of role names contains "Developer"
* **isLoggedIn**: Returns false if Knack.getUserAttributes() is not "No user found" (not logged-in).

## User Preferences

### Usage

Provides various settings for the currently logged-in account. Some are built-in, and more can be added by your app. You can control which settings can be modified by the user and they can access them in the Account Settings page. See the [User Preferences setup procedure](#_User_Preferences).

### Functions

* **setCfg**: To set up your parameters and callbacks to your app. The allowShowPrefs() callback is where you can control what preferences you give access to specific roles. Typically, this is used to give access to more advanced flags to developers. The applyUserPrefs callback is where you can process your own custom preferences.
* **getUserPrefs**: Reads the user preferences from localStorage.

## iFrame Window

Referred to as the **iFrameWnd**, it's a hidden utility page that is dynamically created at the bottom of the main app page. It contains various views to implement system status, user preferences, remote SW updates and logging features. You may even add your own views if you need any. The idea is to be at two places at the same time: The main app page that changes as the user navigates around, and that invisible iFrameWnd that stays with us to serve various functions in the background. When the user logs-in, the authentication token is conveniently shared with the iFrameWnd, allowing us to log-in seamlessly and do API calls. If desired, it is possible to exchange information between both windows using the powerful [wndMsg](#_Windows_Messaging) feature.

### Usage

* It is used to monitor the current SW version on all devices, perform remote SW updates, send UTC timestamps called *heartbeats* from devices to the system to assess sanity/presence.
* The user preferences are also read/modified here, including various debug flags and the work shift.
* A logging table is used to send all logs to Knack via an API call. It contains the 5 most recent logs with a unique identifier (Log ID) to confirm the transaction.
* To enable and configure the iFrameWnd feature, see the [iFrameWnd setup procedure](#_iFrameWnd).

### Functions

* **setCfg**: Called when the iFrameWnd is ready.
* **getCfg**: Returns the iFrameWnd config about field and view IDs.
* **showIFrame**: To show or hide the iFrameWnd.
* **getiFrameWnd**: Returns the iFrameWnd object. Mainly used by sendAppMsg but also available to your app for any use.

## Debug Window

### Usage

Provides a window to see local logs on mobile devices where we don't have the luxury of a console log output. Useful for simple tracing/debugging without the complexity of USB tethering and the learning curve that comes with all the tools. Works on all device types (not just mobile), and the window can be moved around. The logs are stored in a ring buffer of 100 elements.

### Functions

* **lsLog**: Adds a log to localStorage, with timestamp to millisecond resolution. These logs can be shown in the debugWnd when visible, and optionally, in the console.log if you have one.
* **showDebugWnd**: Show or hide the debugWnd.

## Logging

### Usage

Provides comprehensive logging functionalities for just about anything you want to monitor, such as user activity, navigation, system status, errors, or simply traces for development and debugging. All logs are connected to a given account.

To use this feature, you must set the iFrameWnd and all desired logging flags to true in the ktl.core.setCfg function, then follow the [Account Logs setup procedure](#_Account_Logging).

All logs are aways saved in localStorage, with their timestamp. This is to prevent losing any of them in case of power outage or browser crash.

Then, at certain intervals, the logs are inserted to the Account Logs object with an API call, and upon confirmation, they are erased from localStorage.

To minimize record consumption and API calls usage, navigation logs are agglomerated over an hour and sent only once as a single stringified object. A custom viewer then disassembles them for display in chronological order.

The logging categories are: User Login, Navigation, Activity (count of keypresses and mouse clicks), Critical Events, App Errors, Server Errors, Warnings, Info and Debug.

### Functions

* **setCfg**: Allows setting a callback logCategoryAllowed() that returns whether or not a category should be logged, based on specific conditions.
* **clog**: Provides an enhanced version of console.log(), with custom color and bold font text.
* **objSnapshot**: Converts an object to a string and back to an object. This is used to *freeze* the content of an object in time.
* **addLog**: Adds a log to the localStorage for deferred processing. All log categories are not created equal. Here's how each work:
  + Critical: Sent to Knack within 1 minute. An email is also sent to the Sysop.
  + Login, Info, Debug, Warning, App Error, Server Error: Sent to Knack within 1 minute.
  + Activity, Navigation: Data is accumulated in an object in localStorage, then sent as a single bundle to Knack every hour to reduce record usage and API calls.
* **getLogArrayAge**: Used internally by iFrameWnd and returns the oldest log's date/time from array within a resolution of 1 minute.
* **monitorActivity**: Entry point that starts the user activity logging. Every 5 seconds, the mouse clicks and key presses counters are updated in localStorage, and counters from all opened pages and tabs are merged (added) together.
* **resetActivityCtr**: Resets mouse and keyboard activity counters.
* **updateActivity**: Updates the keyboard and mouse activity counters in localStorage. Mainly used by KTL internally, but available to your app, for specific use.

## Windows Messaging

Provides a framework to exchange data between windows. It uses a queue and supports automatic retries and error handling. The windows can be app window, the iFrameWnd, or any other window that the app creates and needs to communicate with. For example, this is how your app can implement a heartbeat message that notifies Knack about your account (or device) being online and running properly.

### Functions

* **setCfg:** Used to set callbacks to your app.
  + **processFailedMessages** to handle messages that were never acknowledged.
  + **processAppMsg** to implement your own messages.
  + **processServerErrors** to implement your own processing of server errors.
  + **sendAppMsg**: Experimental feature still under development. Will be used to exchange messages across different Knack apps.
* **send**: To send a msg to a specific window. May contain a payload or not.
* **removeAllMsgOfType**: Cleans up the msg queue of all those of a specific type.

## System Info

Retrieves information about the operating system, browser, device model, processor, whether or not we are on a mobile device, and public IP address.

This is also where the SW Update broadcast takes place.

### Functions

* **getSysInfo**: Returns an object with the above-mentioned properties.

## System Colors

Retrieves information about Knack's colors and generates a few variations for KTL features.

### Functions

* **getSystemColors**: Get the sysColors object.
* **rgbToHsl**, **hslToRgb**, **rgbToHsv**, **hsvToRgb**, **hexToRgb**: Various color conversion utilities.

# How to use KTL

There are two methods to use KTL: **ACB** and **CLS** modes. If you're in a hurry to test it, and experiment with all the features, you should go for the ACB mode. [See this section for a quick tryout](#_No_time_for).

## ACB Mode – “All Code in Builder”

This is the traditional mode that we're all used to, i.e. when all the code resides in the Builder's Javascript and CSS panes.

To use this mode, you have two options:

1. Use the default, basic, ready-to-use setup [here](#_No_time_for)
2. Use your custom app code and generate the ACB file yourself. This is described in the following section.

### How to generate your own ACB file

First, you’ll need to install **Node.js** (<https://nodejs.org>) on your workstation. Just the basic installation, no optional component is needed.

Then, the custom ACB file can be generated using the [NodeJS\_ACB\_MergeFiles.js](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/NodeJS/NodeJS_ACB_MergeFiles.js) merge utility provided in the NodeJS folder.

This script can be invoked from the batch file provided: [Merge\_ACB.bat](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/Merge_ACB.bat) (for Windows only) You can also run it manually in a command prompt (shell). See batch file for parameters.

These three files will be merged together:

1. [KTL\_Bootloader.js](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/KTL_Bootloader.js)
2. [KTL.js](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/KTL.js)
3. [KTL\_KnackApp.js](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/KTL_KnackApp.js)

The output file is [KTL\_KnackApp\_ACB](https://github.com/cortexrd/Knack-Toolkit-Library/blob/master/KTL_KnackApp_ACB.js).

\*Note about **KTL\_KnackApp.js**: throughout the document, we’ll refer to the app code file as this one, but you can substitute it to anything that would better match your app name. As long as you modify the merge utilities accordingly. See the **-filename** parameter in the batch file.

Open the KTL\_KnackApp\_ACB.js file, copy its content to your Javascript pane in the Builder and save.

Open the KTL.css file, copy its content to your CSS pane in the Builder and save.

\*Note that eventually, when we reach the first official release, these two files will be hosted on my Cortex R&D’s CDN, and all these copy operations won’t be required anymore.

### Pros

* Easier and faster setup, no need to install anything for default ACB.
* Other users can always see your changes.
* You can test your code on any device, not limited to your workstation.

### Cons

* Slower than CLS mode and more tedious to work, due to the redundant merge/copy/paste/save sequence required each time you make a change.
* Can be risky if used in production (when the app is being used in a live and consequential context) since your development code always takes effect immediately. You must have good coding experience and know exactly what you're doing.

### Setup

You will need to modify the KTL\_KnackApp.js file to match your needs if you want to go beyond the basic default setup. [Follow the procedure here](#_Setup_of_KTL_KnackApp.js).

## CLS Mode – “Code on Local Server”

This mode provides much faster code-save-test cycles and is typically used when you have frequent code changes, and where you don't need to show your results to others until a milestone is reached. It requires the installation of Node.js as a basic local file server that the Bootloader uses to fetch the KTL files and your app's code. The Builder's Javascript pane only needs to contain the Bootloader (~240 lines of code!), though it can also have the full ACB code without conflicting.

### Multi-Developers Collaboration

With the CLS mode, it is now possible to have many developers write code independently on the same app. Of course, for other developers to see your changes, they need to pull/merge your new code with theirs, and vice-versa for you to see their changes. GitHub is excellent at that.

### Hybrid Mode for Production and Development at same time!

Traditionally, Knack developers have to create a temporary copy of their production app to experiment freely without fearing serious consequences or disruption. While this is still desirable in many cases, you now have another option: **Hybrid Mode**. Thanks to the Bootloader, a hybrid setup is possible with both the ACB and CLS modes at same time. This enables you to run development code in a production environment without users being affected by it.

What happens is that the Bootloader will use the stable and released code from the ACB in the Javascript pane by default for remote users. But if it detects a development flag in your localStorage, it will switch you to the CLS code on your workstation.

With Hybrid Mode, it is also possible to switch back and forth between the ACB and CLS modes instantly. See [Switching Modes](#_Switching_Modes).

### Pros

* Allows very fast "code-save-test" cycles.
* Allows multi-developer collaboration without conflict.
* Allows Hybrid Mode for dev and prod code running at same time.

### Cons

* Requires a one-time Node.js installation and setup.
* Other users or clients can't see the updates until you merge all code and switch to the ACB Mode.
* You can't test on devices other than your workstation, running the file server.

### Setup

Install **Node.js** (<https://nodejs.org>) on your workstation. Just the basic installation, no optional component is needed.

Validate installation by typing **node -v** in a command prompt or terminal window. You should see the version number displayed.

1. In a code editor, create a new file named *AppName*.js. Using the actual name of your app is required for the Bootloader to recognize it.
2. Add the code from file KTL\_KnackApp.js.
3. Copy your existing code from the Builder at the end, where you see***//My App code*** (between the begin and end markers). Save *AppName*.js.
4. In the Builder's Javascript pane
   1. Wipe all code
   2. Add the code from file KTL\_Bootloader.js
5. Copy the KTL.js and KTL.css files to the folder as per recommendation below (Lib\KTL).
6. Back in *AppName*.js
   1. Locate the **//App constants** section and add any const (scenes, views, field IDs) that KTL may need.
   2. Locate the **//KTL Setup** section and go through all the flags and settings to match your needs.
   3. Locate the **//KTL callbacks to your App** section and go through each function, adapting them to match your needs.
   4. Locate the **//Setup default preferences** section and go through all the flags and settings to match your needs.
7. Open command prompt or a terminal window, go to the **code** folder (see folder structure below) and launch **node NodeJS\_FileServer.js**. Each time you refresh your app's page, you will see logs showing the path and file name requested.
8. Open a browser to your Knack app.
9. Check console logs to see if all is looking good.

### Folder Structure

The following structure is recommended, to keep each apps' code separated, a single set of shared libraries, and everything easy to maintain with a revision control tool like GIT.

.code\MyKnackApps\App1\App1.js

.code\MyKnackApps\App2\App2.js

.code\MyKnackApps\App3\App3.js

.code\Lib\KTL\KTL\_Bootloader.js

.code\Lib\KTL\KTL.js

.code\Lib\KTL\KTL.css

.code\Lib\KTL\KTL\_KnackApp.js

.code\Lib\KTL\NodeJS\NodeJS\_ACB\_MergeFiles.js

.code\Lib\KTL\NodeJS\NodeJS\_FileServer.js

.code\Lib\SomeOtherCoolLib\CoolCode.js

## Switching Modes

Once you’ve mastered both modes, you’ll typically spend 95% of the time in CLS mode for its efficiency and speed, and 5% in ACB mode to show updates to your client.

Switching modes can be done two ways:

1. If you have the showAppInfo flag enabled, it will add the version info on the top-right of the screen. Clicking on it will show a prompt with this: *Which version to run, "prod" or "dev"?* Type in the desired mode and click ok. Note that this is possible only for accounts having the “Developer” role.
2. Add a key to the localStorage for your app with the name followed by **\_dev** like this: **KTL\_KnackApp\_dev**. Leave the value empty since it is not used. Refresh the page and you’ll see the version now shown with bright yellow/red attention getter that indicates you’re in CLS development mode.

# Editing KTL\_KnackApp.js

1. Open the **KTL\_KnackApp.js** file in your favorite editor.
2. Locate the **//App constants** section and add any const (scenes, views, field IDs) that KTL may need. If not sure, just ignore for now.
3. Locate the **//KTL Setup** section and go through all the flags and settings to match your needs.
4. Locate the **//KTL callbacks to your App** section and go through each function, adapting them to match your needs.
5. Locate the **//Setup default preferences** section and go through all the flags and settings to match your needs.
6. In the CSS pane, add the CSS code from file KTL.css to yours.
7. Open a browser to your Knack app.
8. Check console logs to see if all is looking good.

\*Note: If you’re using the ACB mode, you should never edit the generated ACB file directly. Always edit the KTL\_KnackApp.js file and merge again.

# Advanced Features

These features are considered "advanced" in the sense that they require additional setup. Also, some of them can provide communication between various parts of your app, thus leveraging quite powerful administration features.

Namely:

1. iFrameWnd
   1. Heartbeat Monitoring
   2. User Preferences
   3. Account Logging
   4. Remote SW Updates
   5. App Filters (in progress)
2. Bulk Operations
3. User Filters Upload and Download (save/restore)

## Setup

### Invisible Menu

This shall be your default place for any future utility hidden pages. For now, the iFrameWnd page will be its first resident.

1. Create a menu named Invisible Menu.
2. In settings, uncheck Include this page in the Page Menu.

### iFrameWnd

To use this feature, you must set the iFrameWnd flag to true in the ktl.core.setCfg function.

Create a new Login Page and give permission to all users. Set Page Name to: **iFrameWnd**. Its URL should automatically be set to **iframewnd**. This page will be the placeholder for the next features. For now, leave it blank as we need to create a few objects first. Now, go back to the Invisible Menu and assign the iFrameWnd to it.

#### Heartbeat Monitoring and SW Update

If you want to add Heartbeat Monitoring to your app to assess an account's presence and generate alarms, or perform remote SW updates, follow this procedure:

1. In the Accounts object, add these fields:
   1. **SW Version**: Type: Short text
   2. **UTC HB**: Type: Date/Time, Date Format: mm/dd/yyyy, Default Date: none, Time Format: military, Default Time: none.
   3. **Time Zone**: Type: Number, no decimals.
   4. **LOC HB**: Type: Equation, Equation Type: Date, Date Type: hours, Result Type: Date, Equation Editor: {UTC HB}+{Time Zone}, Date Format: mm/dd/yyyy, Time Format: military
   5. **Online**: Type: Yes/No, Default No, Input: Checkbox
   6. **UTC Last Activity**: Type: Date/Time, Date Format: mm/dd/yyyy, Time Format: military
2. Create a new object called **App Settings** with these fields:
   1. **Item**: Type: Short Text, set as object’s Display Field and Sort in Alphabetic order
   2. **Value**: Type: Short Text
   3. **Date/Time**: Type: Date/Time, Date Format: mm/dd/yyyy, Time Format: military
3. In the iFrameWnd page created above, add a Form view that updates the currently logged-in account. Once the view is added, remove all fields, then add on a first line: SW Version, UTC HB and LOC HB (set as read-only). Then on a second line: Online, UTC Last Activity and Time Zone. Set the view title to **Heartbeat**. In the form’s Submit rules, enable auto-reload and set the Confirmation message to “Heartbeat sent successfully.”
4. In the ktl.core.setCfg function, set the iFrameWnd flag to true.
5. Refresh the app and you should see the heartbeat being submitted every minute and online being set to Yes.

#### User Preferences

If you want to add User Preferences to your app, there are some already built-in, and you can also add your own. Follow this procedure:

1. In the Accounts object, add a Paragraph Text field named **User Prefs**.
2. In the iFrameWnd page, add a view: Type: Details, For: Logged-in Account. Once the view is added, remove all fields, then add User Prefs. Set the view title to **Current User Prefs AUTOREFRESH=10**.
3. Add a Form view that updates the currently logged-in account. Once the view is added, remove all fields, then add User Prefs. Set the view title to **Update User Prefs**. Enable the form's auto-reload in the Submit rule.
4. Align both views on the same row to save space.
5. Go to User Pages (at the bottom of the Pages view) and edit the Account Settings page.
6. Add a menu named **My Settings** and move it to the top of the page.
7. Add a link to a new page named **My Preferences** and enter to edit that page.
8. Add a Form view that updates the currently logged-in account. Once the view is added, remove all fields, then add User Prefs. Set the view title to **My Preferences**.

#### Account Logging

If your app requires Account Logging, follow this procedure:

1. Create an object named Account Logs and add these fields:
   1. **Account**: Type: Connection to Accounts, all settings at default.
   2. **Date/Time**: Type: Date/Time, Date Format: mm/dd/yyyy, Default Date: Current Date, Time Format: military, Default Time: Current Time.
   3. **Log Type**: Type: Short Text
   4. **Details**: Type: Paragraph Text
   5. **Log Id**: Type: Short Text
   6. **Log Nb**: Type: Auto-Increment
2. In the iFrameWnd, add a view: Type: Table, For: Account Logs, connected to the logged-in Account.
   1. Once the view is added, remove all fields, then add Log Type, Date/Time, Details, Log ID and an Custom Email action with these settings, as from the screen capture **KTL Account Logs Email Settings.jpg**
   2. Set the view title to **ACCOUNT\_LOGS AUTOREFRESH=30**, disable keyword search, enable Inline editing, 10 records at a time, no filter,
   3. Sort by Log Nb: high to low, limit to 5 records.
3. In your app, locate the function ktl.iFrameWnd.setCfg and set all required fields to match those in your Account Logs object.

#### Bulk Operations

If your app requires Bulk Edit and Bulk Delete operations, follow the procedure described in this section: Bulk Operations.

#### User Filters

In addition to be able to create named buttons for the User Filters that are save in localStorage, it is possible to upload the settings to Knack and download them back wherever and whenever needed. This can be seen as a backup method, or also to migrate them to other devices. Note that if you migrate filters from one app to another, typically a temporary development copy, some filters will not work due to the record IDs that have changed for connected fields. This is a normal behavior, and the only way to fix this is to redo their settings and save back to the same button name.

To support upload and download, follow this procedure:

1. Create an object named Filters and add these fields:
   1. **Account**: Type: Connection to Accounts, all settings at default.
   2. **Date/Time**: Type: Date/Time, Date Format: mm/dd/yyyy, Default Date: Current Date, Time Format: military, Default Time: Current Time.
   3. **Filters Code**: Type: Paragraph Text
2. Go to User Pages and edit the Account Settings page.
3. If not already done from the User Preferences procedure above, add a menu named My Settings and move it at the top of the page.
4. Add a link to a new page named Filters Backup and Restore, and enter to edit that page.
5. Add a menu named My Filters and add two links to a URL named Upload and Download. Both URLs are: **javascript:void(0);** This unusual URL is actually a method that means "do nothing" and let the KTL handle click events with special processing.
6. Add a table that displays Filters connected to the logged-in account, with two fields: Date/Time and Filters Code.
7. Enable Inline Editing, remove Title, disable filtering and search, 10 records at a time.
8. In your app, locate the function ktl.userFilters.setCfg and set all required fields to match those in your Account object

# Future Improvements

* Use JSDoc to have an adequate auto-generated and detailed API documentation, for each function with parameter description, etc.
* Geofencing and other map-based features, with geo-based events and Google Maps integration.
* The sky's the limit! Let's see what we can come up with…

# Conslusion

That's about it for now, thanks for reading this and testing the library. Hope you enjoy it as much as I did writing it.

All code and documentation written by:

Normand Defayette

[nd@ctrnd.com](mailto:nd@ctrnd.com)

Cortex R&D Inc.

Blainville, Québec, Canada