The MasterKey Widget Set developer's guide

Mike Taylor

Introduction

This manual is for people who want to build the widget set from source, develop the widget set's core code, or (more likely) create their own widgets as extensions to the main set.

Those who want to use existing widgets should read The MKWS manual: embedded metasearching with the MasterKey Widget Set instead.

Required development tools

If you are building the widget set, you will need the following Debian packages (or their equivalents on your operating system):

```
$ sudo apt-get install curl git make unzip apache2 pandoc
```

You also need Node.js, but unfortunately the `node-js` package is not available for Debian wheezy. You can either get it from wheezy-backports or download the source from http://nodejs.org/download/ and build it yourself. You need both Node itself and its package manager NPM: `make install` puts them into `/usr/local/bin`.

Concepts

Code structure

The code of the widget set is in four main layers, described here from the bottom up:

1. The core code, which manages the set of widget teams, default options, authentication onto the Service Proxy, and the creation of widgets from HTML elements. This code is in `mkws-core.js`

2. The team code, which manages teams of widgets. This is responsible for the collections of widgets that make up teams, event queues, and handling search-and-retrieval events. This code is in `mkws-team.js`

3. The generic widget code, which handles the creation of widget objects, parsing configuration attributes from their HTML elements, and firing off automatic searches.

4. The code for individual widgets, which is specific to those widgets. It often involves subscribing to events and responding to them by setting the HTML of the widget element, but need not do so. The code for many of the most important widgets is in `mkws-widget-main.js`, but certain other widgets are defined in other files beginning with the prefix `mkws-widget-`.

In addition to this code, there are several source files containing support code:

- `mkws-filter.js` contains support routines implementing the filter-set data structure, which contains information about which filters (e.g. by target, or by facet) are in force.
- `mkws-handlebars.js` contains Handlebars helpers which can be used by the HTML templates.
- `mkws-popup.js` defines a special widget for creating popup windows. These may, but need not, contain other MKWS widgets, forming a popup searching application.

The final component of the source code is the set of Handlebars templates, in the `templates` directory, which are used to emit the HTML of the various widgets' contents. These are compiled into the file `mkws-templates.js`.

Event passing

The primary method of communication between components of the widget set – specifically, between teams and their widgets – is event passing. Widgets subscribe to named events; when something relevant happens (such as the reception of a message from metasearch middleware), the event is published, along with the relevant data. All widgets that subscribed to the event are then notified, and can take appropriate action.

Different kinds of events have different data associated with them. This data is passed when the event is published, and so is made available to the subscribing code.

The possible events, and their associated data, are described below.

Defining new types of widget

Development with MKWS consists primarily of defining new types of widgets. This is done using exactly the same API as the widgets that come as part of the set: they have no privileged access.

You create a new widget type by calling the `mkws.registerWidgetType` function, passing in the widget name and a function. The name is used to recognise HTML elements as being widgets of this type – for example, if you register a `foo` widget, elements like `<div class="mkws-foo">` will become widgets of this type.

The function promotes a bare widget object (which is created by the core widget code and passed in as `this`) into a widget of the appropriate type. MKWS doesn't use classes or explicit prototypes: it just makes objects that have the necessary behaviours. There are *no* behaviours that Widgets are obliged to provide: you can make a doesn't-do-anything-at-all widget if you like:

```
mkws.registerWidgetType('sluggard', function() {});
```

More commonly, widgets will subscribe to one or more events, so that they're notified when something interesting happens. For example, the `log` widget asks to be notified when a `log` event happens, and appends the logged message to its node, as follows:

```
mkws.registerWidgetType('log', function() {
  var that = this;

  this.team.queue("log").subscribe(function(teamName, timestamp, message) {
    $(that.node).append(teamName + ": " + timestamp + message + "<br/>");
  });
});
```

This simple widget illustrates several important points:

- The base widget object (`this`) has several baked-in properties and methods that are available to individual widgets. These include `this.team` (the team that this widget is a part of) and `this.node` (the DOM element of the widget). See below for a full list.
- The team object (`this.team`) also has baked-in properties and methods. These include the `queue` function, which takes an event-name as its argument. See below for a full list.
- You can add functionality to a widget by subscribing it to an event's queue using `this.team.queue("EVENT").subscribe`. The argument is a function which is called whenever the event is published.

- As with so much JavaScript programming, the value of the special variable `this` is lost inside the `subscribe` callback function, so it must be saved if it's to be used inside that callback (typically as a local variable named `that`).

Widget specialisation (inheritance)

Many widgets are simple specialisations of existing widgets. For example, the `images` widget is the same as the `records` widget except that it defaults to using the `images` template for displaying its result list. It's defined as follows:

```
mkws.registerWidgetType('images', function() {
  mkws.promotionFunction('records').call(this);
  if (!this.config.template) this.config.template = 'images';
});
```

Remember that when a promotion function is called, it's passed a base widget object that's not specialised for any particular task. To make a specialised widget, you first promote that base widget into the type that you want to specialise from – in this case, `Records` – using the promotion function that's been registered for that type.

Once this has been done, the specialisations can be introduced. In this case, it's a very simple matter of changing the `template` configuration setting to `'images'` unless it's already been given an explicit value. (That would occur if the HTML used an element like `<div class="mkws-images" template="my-images">` to use a customised template.)

Reference Guide

Widget properties and methods

The following properties and methods exist in the bare widget object that is passed into `registerWidgetType`'s callback function, and can be used by the derived widget.

- `String this.type` – A string containing the type of the widget (`search`, `switch`, etc.)
- `Team this.team` – The team object to which this widget belongs. The team has several additional important properties and methods, described below.

- `DOMElement this.node` – The DOM element of the widget. Most often used for inserting HTML into the widget element.
- `Hash this.config` – A table of configuration values for the widget. This table inherits missing values from the team's configuration, which in turn inherits from the top-level MKWS configuration, which inherits from the default configuration. Instances of widgets in HTML can set configuration items as HTML attributes: for example, the HTML element `<div class="mkwsRecords" maxrecs="10">` creates a widget for which `this.config.maxrecs` is set to 10.
- `String this.toString()` – A function returning a string that briefly names this widget. Can be useful in logging.
- `Void this.log(string)` – A function to log a string for debugging purposes. The string is written on the browser console, and also published to any subcribers to the `log` event.
- `String this.value()` – A function returning the value of the widget's HTML element.
- `VOID autosearch()` – Registers that this kind of widget is one that requires an automatic search to be run for it if an `autosearch` attribute is provided on the HTML element. This is appropriate for widgets such as `Records` and `Facet` that display some part of a search result.
- `subwidget(type, overrides, defaults)` – Returns the HTML of a subwidget of the specified type, which can then be inserted into the widget using the `this.node.html` function. The subwidget is given the same attributes at the parent widget that invokes this function, except where overrides are passed in. If defaults are also provided, then these are used when the parent widget provides no values. Both the `overrides` and `defaults` arguments are hashes: the latter is optional. This can be used to assemble compound widgets containing several subwidgets.

In addition to these properties and methods of the bare widget object, some kinds of specific widget add other properties of their own. For example, the `builder` widget uses a `callback` property as the function that it uses to publish the widget definition that it constructs. This defaults to the builtin function `alert`, but can be overridden by derived widgets such as `console-builder`.

Team methods

Since the team object is supposed to be opaque to widgets, all access is via the following API methods rather than direct access to properties.

- `String team.name()`

- `Bool team.submitted()`
- `Num team.perpage()`
- `Num team.totalRecordCount()`
- `Num team.currentPage();`
- `String team.currentRecordId()`
- `String team.currentRecordData()`

These are all simple accessor functions that provide the ability to read properties of the team. `submitted` is initially false, then becomes true when the first search is submitted (manually or automatically).

- `Array team.filters()` – Another accessor function, providing access to the array of prevailing filters (which narrow the search results by means of Pazpar2 filters and limits). This is really too complicated an object for the widgets to be given access to, but it's convenient to do it this way. If you have a reason for using this, see the `Navi` widget, which is the only place it's used.
- `Bool team.targetFiltered(targetId)` – Indicates whether the specified target has been filtered by selection as a facet. This is used only by the `Facet` widget, and there is probably no reason for you to use it.
- `Hash team.config()` – Access to the team's configuration settings. There is rarely a need to use this: the settings that haven't been overridden are accessible via `this.config`.
- `Void team.set_sortOrder(string)`, `Void team.set_perpage(number)` – "Setter" functions for the team's `sortOrder` and `perpage` functions. Unlikely to be needed outside of the `Sort` and `Perpage` widgets.
- `Queue team.queue(eventName)` – Returns the queue associated with the named event: this can be used to subscribe to the event (or more rarely to publish it). See the section on events, below.
- `Void team.newSearch(widget, query, sortOrder, maxrecs, perpage, limit, targets, targetfilter)` – Starts a new search with the specified parameters, taking settings from the nominated widget's configuration when not specified explicitly. All but the query may be omitted. The meanings of the parameters are those of the same-named configuration settings described in the user's manual.
- `Void team.reShow()` – Using the existing search, re-shows the result records after a change in sort-order, per-page count, etc.

- `String team.recordElementId(recordId)` – Utility function for converting a record identifer (returned from Pazpar2) into a version suitable for use as an HTML element ID.
- `String team.renderDetails(recordData)` – Utility function returns an HTML rendering of the record represented by the specified data.
- `Template team.loadTemplate(templateName)` – Loads (or retrieves from cache) the named Handlebars template, and returns it in a form that can be invoked as a function, passed a data-set.

Some of these methods are arguably too low-level and should not be exposed; others should probably be widget-level methods. The present infelicities should be fixed in future releases, but backwards compatibility with the present API will be maintained for at least one complete major-release cycle.

Events

The following events are generated by the widget-set code:

- `authenticated` (authName, realm) – When authentication is completed successfully, this event is published to *all* teams. Two parameters are passed: the human-readable name of the library that has been authenticated onto, and the corresponding machine-readable library ID.
- `ready` – Published to *all* teams when they are ready to search. No parameters are passed. This event is used to implement automatic searching, and should probably not be used by application code.
- `stat` (data) – Published to a team when a `stat` response is received from Pazpar2. The data from the response is passed as a parameter.
- `firstrecords` (hitcount) – Published to a team when the first set of records is found by a search. The number of records found (so far) is passed as the parameter.
- `complete` (hitcount) – Published to a team when a search is complete, and no more records will be found (i.e. all targets have either responded or failed with an error). The final number of records found is passed as the parameter.
- `targets` (data) – Published to a team when a `bytarget` response is received from Pazpar2. The data from the response is passed as a parameter.
- `facets` (data) – Published to a team when a `term` response is received from Pazpar2. The data from the response is passed as a parameter.

- `pager` (data) – Published to a team when a `show` response is received from Pazpar2. The data from the response is passed as a parameter. This event is used to update the pager, showing how many records have been found, which page is being displayed, etc.
- `records` (data) – Also published to a team when a `show` response is received from Pazpar2. The data from the response is passed as a parameter. This event is used to update the displayed records.
- `record` (data) – Published to a team when a `record` response is received from Pazpar2 (i.e. the full data for a single record). The data from the response is passed as a parameter.
- `navi` – Published to a team when a new search is about to be submitted. This is a signal that the navigation area, showing which filters are in effect, should be updated. No parameter is passed: the event handler should consult `team.filters` to see what the prevailing set is.
- `log` (teamName, timestamp, message) – Published to a team when a message is about to be logged to the console for debugging. Three arguments are passed: the name of the team that generated the log message, a timestamp string, and the message itself. Note that this event is *not* published when the widget-set core code generates a log message – only when a team or a widget does.
