Filters API

The filters API is broken into two broad sections.

- The Filters API - this refers to functionality concerned with the filter container object and all of the filters within it (could be one or more filters).
- The Filters Object API - this refers to functionality concerned with an individual filter.

Filters Overview

Filters are used in Yellowfin to restrict the data returned in a report to the exact data that a user is interested in (for example, restricting time periods to just return data for the last quarter, or restricting to a list of countries that a user has responsibility for). You can learn more about filters in general and how to create them in this section.

Dashboard and Report objects will both contain the object *filters*. This object will contain any filters that have been set up as user prompt filters, these are filters that a user can change at run time. Any hardcoded filters on a report are not able to be accessed through the API.

The filter object can be accessed as follows:

For a report:

report.filters

And for a dashboard:

dashboard.filters

This Filters API contains all of the available user prompt filters for a piece of content as well as various methods that allow developers to manipulate the filter. It also can fire a number of events that allows your applications to react to changes within the filters.

How filter values work

A filter will typically relate to a Database column that is to be restricted, such as a specific date field, a dimension field (countries, products) or even a metric (sales, costs, margin). Values are set against a filter and these values are passed through to the Database query in order to restrict the data being returned.

Filter values can have two states - **Staged** (or normal) and **Applied**.

Applied values are the values that are currently being used by the report or dashboards to filter the query result. Anytime the API uses *getAppliedValue* it is referring to these values.

A staged filter value is a value that is set in the Filter but not yet applied to the report output. For example, in the filter “Demographic” when a user selects the value “Adventure” this sets the staged value of that filter to “Adventure”. Anytime the API uses *setValue* or *getValue,* it is referring to the staged value of the filter. This is not reflected in the report or dashboard until the filters are applied.

Some filters will also have *defaultValues*, these are always read only.

Understanding Values

Throughout this reference document and the **FilterObject** reference document, you will see references to different *value* objects.

E.g. valueOne, valueTwo, appliedValueOne.

Yellowfin uses these values to correctly apply the filter values when running a report.

ValueOne will refer to the value of a filter that has a single input option (Equal To, Different From, etc). It will also be used as the lower value in a between operator (Between and Not Between) so when using a between operator valueOne should always be lower than valueTwo.

ValueList is the value used for a list operator (In List, Not In List) and is an array of string or number values.

Differences between report and dashboard filters

From a functionality point of view, there is very little difference between a report filter and a dashboard filter object. A dashboard filter object serves many reports as it will have linking objects that allow many reports to be linked to a single filter.

When building a dashboard in Yellowfin, users will first add reports to the dashboard and then add filters from those reports to the dashboard. This is described here.

A filter added to a dashboard will be allocated its own unique filter id, even though it is derived from a report filter in the first instance. This Dashboard Filter will continue to maintain a reference to the report filter from which it was based. Changes made to report filters will flow through to dashboard filters.

If you have code that references a report filter by UUID it will be accessible in a report filters object by that UUID:

```
report.filters.getFilter('47fe96c2-5101-4b0d-9018-7d12a84d3519');
```

If that report and filter were then added to a dashboard, you would no longer be able to use the exact same code to access the dashboard filters. This is because it is possible for there to be multiple instances of the report and filter active on a dashboard together, so they are all referenced by their dashboard filter uuid.

```
dashboard.filters.getFilter('47fe96c2-5101-4b0d-9018-7d12a84d3519'); //Will return null
```

Property Reference

No properties associated with the Filter object need to be accessed to utilise the API functionality.

Function Reference

getFilter(filterId)

Returns

FilterObject

Description

Fetches a FilterObject for the passed filterId. If there is no matching *filterId* then null will be returned.

Parameters

filterId - (String, Number)

The Name or the UUID of the filter you wish to access.

Examples

Get the filter named “Demographic”

```
let filter = filters.get('Demographic');
console.log(filter.name, filter.uuid); //Output the name of the filter as well as its UUID
```

Get the filter by UUID

```
let filter = filters.get('47fe96c2-5101-4b0d-9018-7d12a84d3519');
console.log(filter.name, filter.uuid); //Output the name of the filter as well as its UUID
```

getAllFilters()

Returns

Object - {String, FilterObject}

Description

Returns an Object that contains all of the user prompt filters for the content that the FiltersAPI is attached to. This object is keyed by Filter UUID.

Notes/Limitations

This function is useful for observing which filters are available in the FiltersAPI. If you are looking for a specific filter it is recommended using *filters.getFilter (filterId)* rather than getAllFilters()[uuid] as *getFilters* can accept a UUID or a name. If you wish to iterate over all filters, we recommend using *filters.forEach (fn)* instead.

forEach(fn)

Returns

Nothing.

Description

Iterates over all the filter objects in a FilterAPI, calls the passed fn with an individual FilterObject being passed to it on each iteration.

Parameters

fn - Function to call for each iteration of the loop.

Example

You might want to build a list of all of the applied values:

```
let appliedFilterValues = [];
filters.forEach(filter => {
        appliedFilterValues.push(Object.assign({
                name: filter.name,
                uuid: filter.uuid,
         }, filter.appliedValues));
```

This would create a list of applied filter values with the filter's name and uuid also included in the object.

hasFilters()

Returns

Boolean

Description

Returns true if there are any User Prompt filters on the content. Returns false otherwise.

Example

Call a custom filter panel generator based on if the filters API has filters or not.

```
if(filters.hasFilters()) {
	generateMyCustomFilterPanel(filters);
}
```

clearFilter(filterId)

Returns

Nothing.

Description

Clears the values and immediately applies the now empty values, from the FilterObject relating to the passed *filterId*. If no filter is found matching that filterId, nothing will happen.

Parameters

filterId - (String)

UUID or Name of the filter you wish to clear.

clear()

Returns

Nothing.

Description

Clears the values and applies the empty filter values in every FilterObject contained within the FiltersAPI.

applyFilters()

Returns

Nothing.

Description

Copies all the staged values to the applied filters object and runs any reports that are affected by this. Triggers the applied event.

Examples

Set Average Age at Camp and Demographic and then apply them:

```
let demographic = filters.getFilter('Demographic');
let ageAtCamp = filters.getFilter('Average Age at Camp');

demographic.setValue(['Adventure']);
ageAtCamp.setValue([15, 35]);

filters.applyFilters();
```

Or apply filters when a button on the page is clicked:

```
document.querySelector('div#applyButton').addEventListener('click', function(e) {
	filters.applyFilters();
});
```

loadFilters()

Returns

Promise.

Description

Reloads filter data from the server. Promise is resolved once the filters have been loaded.

resetFiltersToDefault()

Returns

Nothing.

Description

Resets all filters back to their default values.

Examples

```
filters.resetFiltersToDefault();
```

addEventListener(eventName, callbackFunction)

Returns

Number.

Description

Creates a listener on an event which will call the *callbackFunction* whenever that particular event occurs.

When an event is set up, a unique ID is assigned to it which is returned as the result of this function. This ID can be used by the *removeEventListener* functi on to remove the callback when you are done with it. If you are writing an application that requires loading and unloading reports, it is recommended that you keep track of these listenerIds so that you can remove them when no longer needed.

See the event reference section for details about the events that the API will trigger itself.

It is also possible for developers to trigger their own events (see .trigger()) which can also be listened to using this function.

Example

Create a listener on the changed object and remove it once the event occurs once:

```
let eventListenerId = filters.addEventListener('changed', function(event) {
        console.log('One of my filters changed');
        filters.removeEventListener(eventListenerId);
});
```

removeEventListener(listenerId)

Returns

Nothing.

Description

Removes the callback function associated with the passed listenerId. This will mean that when the event associated with that callback function occurs, that callback will not be fired anymore.

Example

```
let eventListenerId = filters.addEventListener('changed', function(event) {
        console.log('One of my filters changed');
        filters.removeEventListener(eventListenerId);
});
```

trigger(eventName, eventData)

Returns

Nothing.

Description

Triggers an event on the FiltersAPI and calls any listener functions that have been created for that event. This can be used to trigger custom events that you may have set up for the application.

Example

When using a custom filter input we could trigger a filter click event, so that another part of the application could react to that.

```
//Add a 'userClick' listener to the filter object, which we will set up a trigger for later on.
filters.addEventListener('userClick', function(event) {
        console.log('A user clicked on the element ' + event.element + ' which is tied to this filter');
});
//Get the custom filter list from the DOM and create a click listener on that which will trigger userClicked events on the filter
let myCustomFilterList = document.querySelector('div#customFilterList')
myCustomFilterList .addEventListener('click', function(e) {
        filters.trigger('userClicked', { element: e.currentTarget } );
});
```

Event Reference

There are a number of events related to actions a user takes or functions that are called, that will automatically trigger within the FilterAPI. All of these events will contain the following structure:

- **eventName** - String - The name of the event that has been triggered.
- **filterEvents** - Array{Object} - An array of events that comprised this filter list change event. The object will contain at least the following.
    - **uuid** - UUID of the filter that changed.
    - **filter** - FilterObject that caused this event to trigger.
    - Certain other filter events can contain more information. You can see the documentation in FilterObject for more in depth details on the Event object for each event.

changed

Description

Triggered when any of the FilterObjects within the FiltersAPI changes its staged values.

Parameters

Event - Object

Contains

- eventName - “changed”
- filterEvents - Array{Object} - Array of change event objects that have contributed to the filtersAPI triggering the changed event.

Example

```
let filters = report.filters;

filters.addEventListener('changed', function(event) {
        console.log(event.eventName); //'changed'
        console.log(event.filterEvents); //[{ uuid: 'a-filter-uuid', filter: FilterObject, changed: Object, previous Object }, {....}]
});

filters.setFilterValue('Demographic', ['Adventure']);
```

applied

Description

Triggered when any of the filters within the FiltersAPI has different values applied to it. This happens when the user hits the apply button, or the applyFilters function is called.

Parameters

Event - Object

Contains

- eventName - “applied”
- filterEvents - Array{Object} - Array of change event objects that have contributed to the filtersAPI triggering the applied event

Example

```
let filters = report.filters;

filters.addEventListener('applied', function(event) {
        console.log(event.eventName); //'applied'
        console.log(event.filterEvents); //[{ uuid: 'a-filter-uuid', filter: FilterObject, changed: Object, previous Object }, {....}]
});

filters.setFilterValue('Demographic', ['Adventure']);

filters.applyFilters();
```

reset

Description

Triggered when any of the FilterObjects within the FiltersAPI are reset.

Parameters

Event - Object

Contains

- eventName - “reset”
- filterEvents - Array{Object} - Array of reset event objects that have contributed to the FiltersAPI triggering the reset event.

Example

```
let filters = report.filters;

filters.addEventListener('reset', function(event) {
        console.log(event.eventName); //'changed'
        console.log(event.filterEvents); //[{ uuid: 'a-filter-uuid', filter: FilterObject}, {....}]
});

filters.setFilterValue('Demographic', ['Adventure']);
filters.resetFiltersToDefault();

filters.getFilter('Demographic').reset(); //Trigger reset on an individual filter
```

cleared

Description

Triggered when any of the FilterObjects within the FiltersAPI are cleared.

Parameters

Event - Object

Contains:

- eventName - “cleared”
- filterEvents - Array{Object} - Array of cleared event objects that have contributed to the filtersAPI triggering the cleared event.

Example

```
let filters = report.filters;

filters.addEventListener('reset', function(event) {
        console.log(event.eventName); //'changed'
        console.log(event.filterEvents); //[{ uuid: 'a-filter-uuid', filter: FilterObject}, {....}]
});

filters.setFilterValue('Demographic', ['Adventure']);

filters.applyFilters();
```