Parameter Helper Options

The PanelCollection API uses several helper objects, that could be used for passing around options, validation or setting up dependencies. These are listed below:

- **GeneralPanelOptions:** Defines general UI display.
- **CssRule:** Defines styling rules.
- **ListOptions:** Defines parameters to be displayed as a list.
- **ParameterValidation:** Covers basic UI validations.
- **ParameterDisplayRule:** Defines whether to show or hide a parameter, based on user input.
- **ParameterValueLoader:** Class used to dynamically load values based on an event.
- **DynamicParameterOptions:** The response object of ParameterValueLoader.
- **FieldObject:** Class representing a field.

GeneralPanelOptions

The **GeneralPanelOptions** class is used to define general UI rendering options at the Parameter Panel and Section levels. Methods of significance are listed below:

Methods	Description
public void setSaveButton(boolean saveButton) public void setSaveButtonOptions (Map<String, Object> opts) public void setSaveText (String saveText)	Setting the **setSaveButton** option to true displays a save button in the panel or section. Display options may be set using **setSaveButtonOptions**. Available options are listed under type BUTTON in the section on Input Types. The text on the button can be customized using **setSaveText**.
public void setExpandable(boolean expandable) public void setExpanded (boolean expandable)	These are used to make a section expandable and render it expanded.
public void setShowName(boolean showTitle)	When this is set to true, the name of the panel or section is displayed at the top of the container.
public void setCssRules (Set<CssRule> cssRules)	This method is used to set custom CSS rules to style a section or panel and everything within its html container.

top

CssRule

This interface is used to define styling rules in various levels of the PanelCollection API. The levels which support it, accept a set of CssRule objects. Yellowfin has an implementation called **CssRuleImpl**, which defines a single CSS Rule, such as:

```
div.styleExampleCell {
          border: none;
          color: #666666;
}
```

Instances will have a selector and one or more of CSS declarations. If a selector isn't defined, Yellowfin will autogenerate one.

The **CssDeclaration** interface describes a single declaration such as:

```
border: none;
```

Yellowfin has an implementation called **CssDeclarationImpl** which accepts a property and value.

```
Parameter inputField = new ParameterImpl();
inputField.setName("Example Param");
inputField.setProperty("PARAM_PROPERTY");
inputField.setInputType(InputType.TEXTBOX);

CssRule cssRule = new CssRuleImpl("input", false);
cssRule.addDeclaration(new CssDeclarationImpl("height", "21px"));
cssRule.addDeclaration(new CssDeclarationImpl("padding", "5px"));
cssRule.addDeclaration(new CssDeclarationImpl("font-size", "16px"));
cssRule.addDeclaration(new CssDeclarationImpl("resize", "none"));
cssRule.addDeclaration(new CssDeclarationImpl("color", "#666666"));
cssRule.addDeclaration(new CssDeclarationImpl("border", "1px solid #e4e4e4"));
Set<CssRule> cssRules = new HashSet<>();
cssRules.add(cssRule);
inputField.setCssRules(cssRules);
```

This created the following CSS rule:

```
input {
          height: 21px;
        padding: 5px;
        font-size: 16px;
        resize: none;
          color: #666666;
          border: 1px solid #e4e4e4;
}
```

top

ListOptions

This class is used to define UI options for when a parameter is rendered as a list. For example, a TEXTBOX parameter could accept several text values, for which it will render a list of textboxes.

Refer to the javadoc for all available options.

top

ParameterValidation

This class is used to define basic UI validation rules. The most useful rule is to check if a Parameter value is empty. Other rules perform relational operations on numeric Parameters.

Refer to the javadoc for all available options.

top

ParameterDisplayRule

There might be a need to hide or show a Parameter based on user input. This can be done using instances of **ParameterDisplayRule** which lets you define those Parameters which a Parameter should be listening to, and specify values to make it appear or hide. See examples below.

Example 1

The below snipped is for showing TABLE_NAME when SOURCE is set to anything other than 0 or null.

```
Parameter p = new ParameterImpl();
p.setName("Table Name");
p.setProperty("TABLE_NAME");
p.InputType(InputType.SELECT);
p.addDisplayRule(new ParameterDisplayRule("AND", "SOURCE", new Object[] { null, 0 }, true, null));
```

This display rule would essentially create the following line of code:

```
if(SOURCE != null && SOURCE != 0) showParameter();
```

Example 2

To make TABLE_NAME show when SOURCE was null or 0, the negative boolean should be changed to false.

```
p.addDisplayRule(new ParameterDisplayRule("AND", "SOURCE", new Object[] { null, 0 }, false, null));
```

This would create something like this:

```
if(SOURCE == null || SOURCE == 0) showParameter();
```

These display rules can be applied to any level of the Panel Collection, even to hide an entire panel or section.

Constructors

The class provides overloaded constructors for convenience and setters for each property. The most descriptive constructor is:

public ParameterDisplayRule(String logic, String property, Object[] vals, boolean negative, PropertyLocation location)

Below is an explanation of its attributes:

- **logic** is used to specify the operator when there are multiple “child” display rules within this instance of ParameterDisplayRule. Logic may be AND or OR. If there are three child rules and logic is AND, each rule will be evaluated individually and the results will be combined as : Rule1Result && Rule2Result && Rule3Result
  Logic will be ignored if there are no “child” display rules. If there are child rules, the “parent” will not be evaluated. It will be used only as a container for the child rules.
- **property** specifies the “other” parameter to be inspected by this display rule. Every Parameter instance has an identifier for its value. For example, whatever the user types into a Text Input parameter may be referenced using the identifier “SOURCE” if it is specified as its property.
- **vals** is an array of values for comparison. Each value in the array is compared against the “other” parameter for equality. They are then combined using the OR operator.
- **negative** inverts the result obtained after comparing vals against the “other” parameter.
  If vals = {null, 0}

negative	result
true	property != null && property != 0
false	property == null || property == 0

- **location** is an instance of PropertyLocation and is used to determine where the “other” parameter is located. This is useful if there are parameters with the same property in multiple sections/panels. If this is null, it is assumed the property is in the same location as the parent of this ParameterDisplayRule.

top

ParameterValueLoader

The ParameterValueLoader is an abstract class which can be implemented for dynamically loading values based on an event. Value loaders are attached to a section. Parameters should also be set up to use a value loader. Here are the important things to remember:

- The parameter’s “property” is important as it is used to reference changed values and event data. For example:

```
p.setProperty("view");
```

- An event name needs to be specified for when the parameter changes. The value loader can decide what to do based on this. Example:

```
p.setEvent("viewChanged");
```

- Event Data may be attached to the parameter. This could be any information about the step, such as all of its available fields or the value of some other step option.

```
p.addEventData("fieldsMap", allFieldsMap);
p.addEventData("sourceId", getStepOption("SOURCE_ID"));
```

- The value loader’s response to an event can be returned using the getUpdatedPossibleValues() or generateDynamicParameters() methods. Thes e can make decisions on how to respond to events using the member variables listed below. Yellowfin runs these methods in the following sequence:

```
pvl.getDynamicParameterOptions.setValues(pvl.getUpdatedPossibleValues());
pvl.generateDynamicParameters();
```

> ⚠ Note, that generateDynamicParameters() does not return anything. Implementations of the method are expected to populate the "response" member.

- Dependencies can be set up using value dependencies or events. Both of these options are discussed in detail below.

ValueDependencies

When creating a parameter, one can create a list of "ValueDependencies" that the parameter will be dependent on. These should point to a parameter present somewhere else in the panel collection. Once this is set up, when one of the parameters referenced in the value dependencies is changed, a request containing all the required information will be sent to the server.

Let's assume that we have a parameter for Views and want to make it dependent on a parameter for data sources so that the view list is reloaded whenever the *data source* changes.

```
datasourceParam.setProperty("datasource");
datasourceParam.setEvent("datasourceChanged");

viewParam.setProperty("view");
// Make view dependent on the data source
viewParam.addValueDependency("datasource");
```

A value loader in the data source Parameter's section can populate the available Views for the selected data source whenever it receives the *dataSourceC hanged* event. The response object is sent to the front-end, and Yellowfin populates a list of available Views.

Events

The Events function is similar to value dependencies, except in this case, the changing parameter determines which parameters to reload. Effectively, the direction of the dependency has reversed in this case. Whereas value dependencies are "other" parameters which listen for changes in a parameter, events are used by that parameter to change "other" parameters. This can also be attached to a button if a button click is required.

There are three parts to creating an event trigger:

1. **Event Name:** This is set using the setEvent() method of Parameter. The ParameterValueLoader has a protected member containing all of the events that have been triggered.
2. **Event Data:** This refers to any extra data that the developer can set up when the panel is being generated. This could include adding the step ID or a field's list, for example.
3. **Event Parameters:** The event parameters are other parameters that the event needs data for. Consider a case where there are parameters for region, country, and city, and the city parameter changes. If the value loader needs information about the region and country as well, these will become event parameters. The Parameter API has a number of methods to set up event parameters, but they all add to a list of ValueDependent objects.

Event Data and the current values of the parameters specified in Event Parameters will be combined into a single EventData object.

```
cityParam.setProperty("city");
cityParam.setEvent("cityChanged");
cityParam.addEventData("fieldUUIDS", fieldUuids);

p.addEventParameter("country", new PropertyLocation("MyPanel", "addressSection"));
p.addEventParameter("region", new PropertyLocation("MyPanel", "addressSection"));
```

Member Variables

The value loader class has a number of important member variables which hold information about events:

Member Variable	Description
protected Map<String, Object> changedValues	Member changedValues has the parameters and their changed values. The keys in the map should be the “property” attribute of the parameter which triggered the change event.

protected Map<String, Object> eventData	This member is keyed by the “property” attribute of parameters which are listening for the change event of another parameter. The value is whatever data the developer decided should be sent back on receiving the change event.
	
	Let's assume an example in which a developer were to create the data source and view parameters, where the view is dependent on the data source. If a user changed the value of the data source to 1, changedValues would look like the following:
	
	`{ "datasource": 1 }`
	
	And here's what eventData would look like based on our example:
	
	`{ "view": objects set by the developer }`

protected List<String> events	This holds all events received by the value loader. `if(events.contains("cityChanged")) { // do stuff }`
protected DynamicParameterOptions response	Dynamic Parameter Options is the response object of the ParameterValueLoader. It should contain everything the developer wants to show on the front-end. DynamicParameterOptions is discussed in detail below.

Methods

There are two methods which can be implemented. However, only one of them would need to be implemented, depending on what the developer wants to do. They both serve nearly the same purpose.

Method	Description
public void generateDynamicParameters()	This method is expected to populate the response member variable. See the section on DynamicParameterOptions for more information. This is the preferred method, and is more flexible.
public Map<String, List<CustomValue<?>>> getUpdatedPossibleValues()	The method is expected to return a map of “possible values” for every affected parameter. For each entry in the map, the key should be a parameter’s “property” attribute. The value should be a list of CustomValue objects containing the “possible” values. This method may be deprecated in the future.

There are some other helper methods for reading files and large text, listed below.

Helper method	Description

public final byte[] getFile (Integer fileId)	This method can be used to read a file written by a FileUploadParameter. An event must be set in the parameter. It could also be set using the constructor of FileUploadParameter. The value loader is triggered with this event after the file is uploaded and saved into the Yellowfin config database. This may be used to load/remove parameters based on data in the file.
public final String getText (Integer textId)	This is similar to getFile, except that it reads CLOBs stored in the Yellowfin config database.

top

DynamicParameterOptions

Dynamic Parameter Options is the response object of the ParameterValueLoader. It should contain everything the developer wants to present on the front-end. There are six things that a developer can add to this:

Dynamic Parameter Option	Description
public void addPanel(String dynamicKey, ParameterPanel panelObject)	If one of the triggered events requires a new panel to be added to the menu, it can be added here. The front-end will then convert the panel definition into a user interface. The dynamicKey will be used to remove any objects currently using that key, as well as tagging this Panel with that key so it can be removed in the future.
public void addSection(String dKey, String panelKey, ParameterSection sec)	This allows developers to add a section to a panel that should already exist as part of the client side panel collection. dKey is the dynamic key which will be used to remove any objects currently using that key, as well as tagging this Section with that key so it can be removed in the future. The panelKey should be the key to add this section to.
public void addParameter (String dKey, String panelKey, String secKey, Parameter parameter)	This allows developers to add a parameter to an already existing section/panel combination. dKey is the dynamic key which will be used to remove any objects currently using that key, as well as tagging this parameter with that key so it can be removed in the future. The panelKey and secKey should be keys of the Panel and Section to add this parameter to.
public void addValue(String property, CustomValue<?> value)	Adds a single option that a user can select, to the DynamicParameterOptions. The property corresponds to the parameter which displays the value.
public void setValues (Map<String, List<CustomValue<?>>)	Overwrites any values that are currently set in the DynamicParameterOptions.
public void addKeyToRemove (String key)	Adds a key of a Parameter/Section/Panel to remove from the client-side panel collection. Any parameter, section or panel that is added using their respective methods will also add their keys to be removed.

top

FieldObject

This is a class which represents a field. Instances of this class are used in UI widgets for matching fields.

public FieldObject(String id, String keyType, String name, FieldObjectDataType dataType)

Where,

- **id** is used to uniquely identify the field.
- **keyType** may be set to “primary” to identify that this a primary key field.
- **name** is the display name of the field.
- **dataType** is the type of the field, as defined in enum *FieldObjectDataType*. An overloaded constructor accepts a string data type, but it must match with an enum element.

top