Apogee - Programming Guides

Apogee - Programming Guides

Guides

- **Configurable Forms** - This guide covers configurable forms.
- **Custom Components** - This guide covers using *Custom Components*.
- **Creating Reusable Components** - This guide shows you how to make reusable components.

Configurable Forms

Components Using the Configurable Form

In Apogee there is a configurable form widget that is used in several places to easily define forms for the user. This document gives the reference information for using these configurable forms.

Two examples where these are used are the components *Form Data Table* and *Dynamic Form.* For an example of using these components, see the tutorial Configurable Form Components.

A Form Data Table is similar to a JSON Data Table except the user enters data into a configurable form rather than text editor. The configurable form is defined by the programmer. When the user changes the value, the save bar appears just as if the user changed the value of a data table. When the user saves, the data is saved to an internal data table which can be accessed from the code.

A Dynamic Form is a table that is like a non-modal dialog box. Here the programmer also configures the layout of the form, which should include a submit button to take action. The programmer defines a handler for the save button which can take any action he wants. Unlike with the Form Data Table, there is no data saved in the Dynamic Form.

In either of these components, the programmer must define the layout for the form, which is described in this document.

Configurable Panel and Configurable Elements

The name of the configurable form is actually *ConfigurablePanel*. It includes a list of entries, which are called *ConfigurableElements*. There are several types of configurable elements.

The layout for the form is JSON/javascript object which gives the list of configurable elements and the initialization of each.

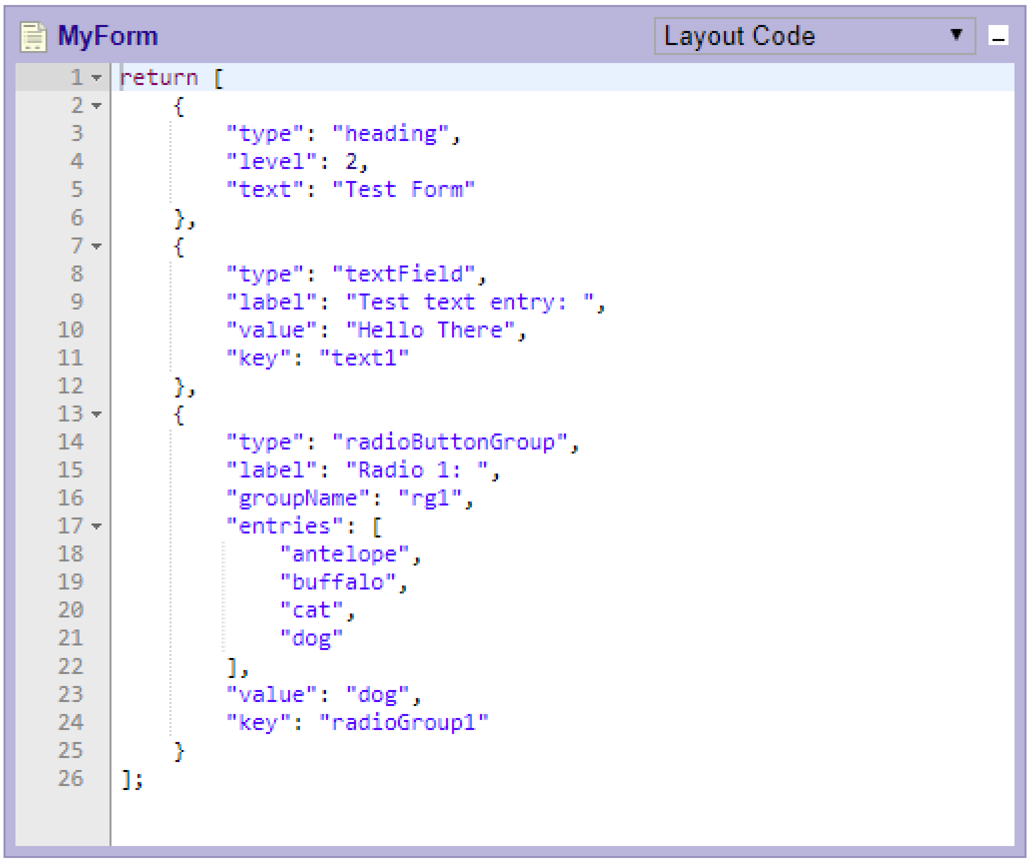
We will show a quick example of using a configurable panel with a Form Data Table.

The following JSON gives a sample layout that includes the following elements:

- Title
- Text Field
- Radio Button Group

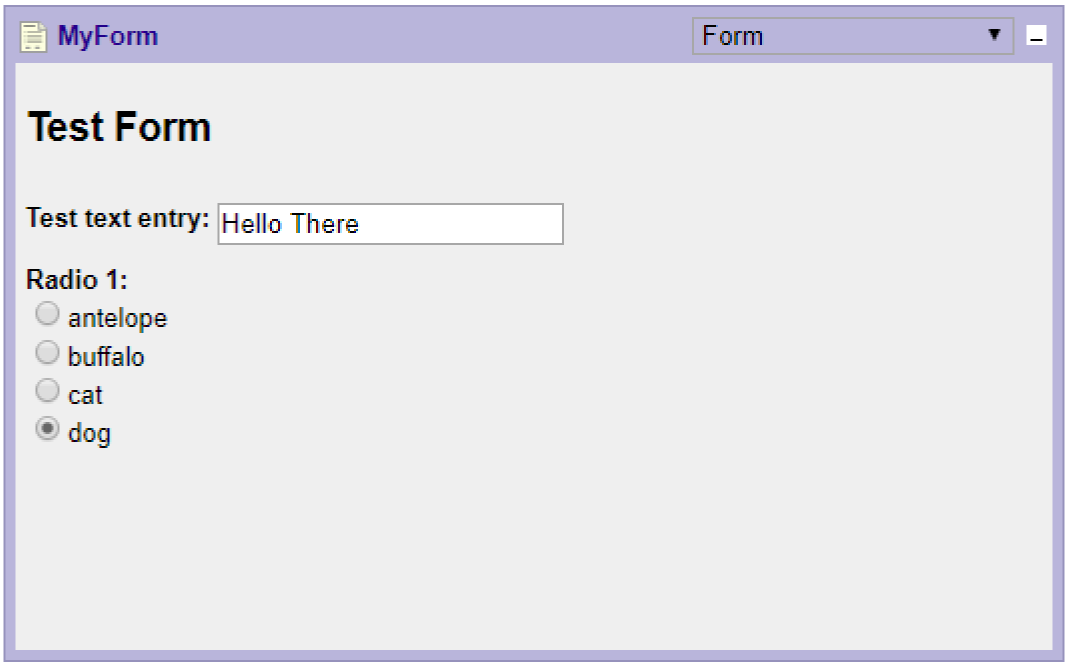
```json
[
  {
    "type": "heading",
    "level": 2,
    "text": "Test Form"
  },
  {
    "type": "textField",
    "label": "Test text entry: ",
    "value": "Hello There",
    "key": "text1"
  },
  {
    "type": "radioButtonGroup",
    "label": "Radio 1: ",
    "groupName": "rg1",
    "entries": [
      "antelope",
      "buffalo",
      "cat",
      "dog"
    ],
    "value": "dog",
    "key": "radioGroup1"
  }
]
```

Using this layout JSON in a *Form Data Table,* we return this value from the layout code as follows:



```
return [
    {
        "type": "heading",
        "level": 2,
        "text": "Test Form"
    },
    {
        "type": "textField",
        "label": "Test text entry: ",
        "value": "Hello There",
        "key": "text1"
    },
    {
        "type": "radioButtonGroup",
        "label": "Radio 1: ",
        "groupName": "rg1",
        "entries": [
            "antelope",
            "buffalo",
            "cat",
            "dog"
        ],
        "value": "dog",
        "key": "radioGroup1"
    }
];
```

This creates the following form:

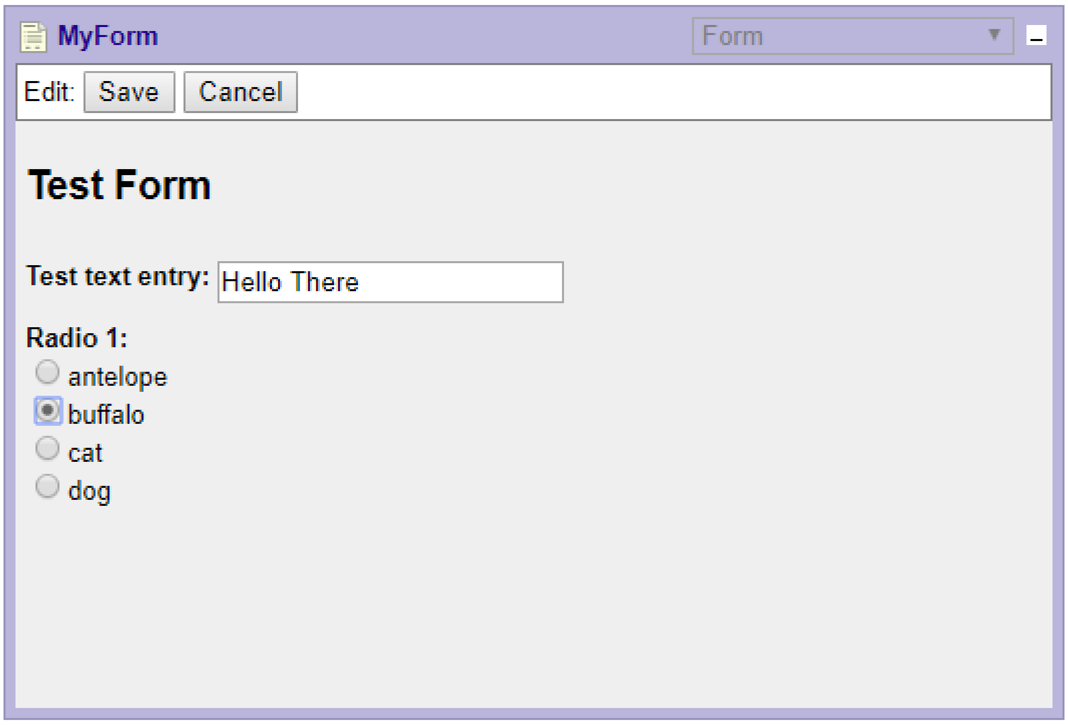


Test Form

Test text entry: Hello There

Radio 1:
( ) antelope
( ) buffalo
( ) cat
(●) dog

If we change the radio buttons, we get the *save bar*, just as if we edited the value of a plain data table.



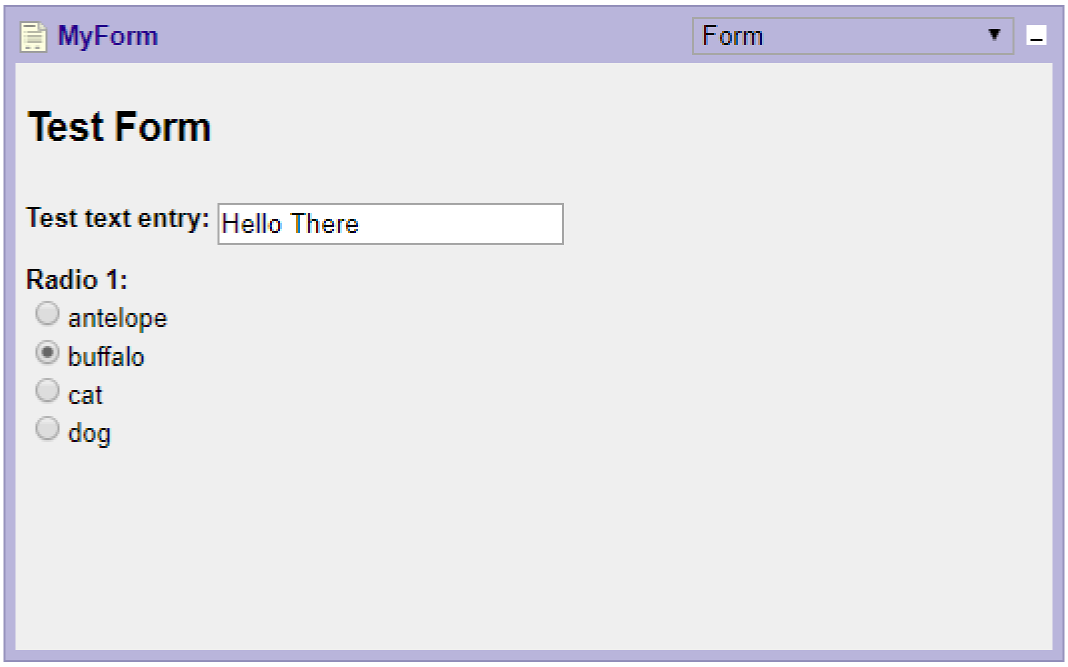
Test Form

Test text entry: Hello There

Radio 1:
antelope
[✓ buffalo]
cat
dog

Here is the Form Data Component after changing the value.

Pressing *Save*...



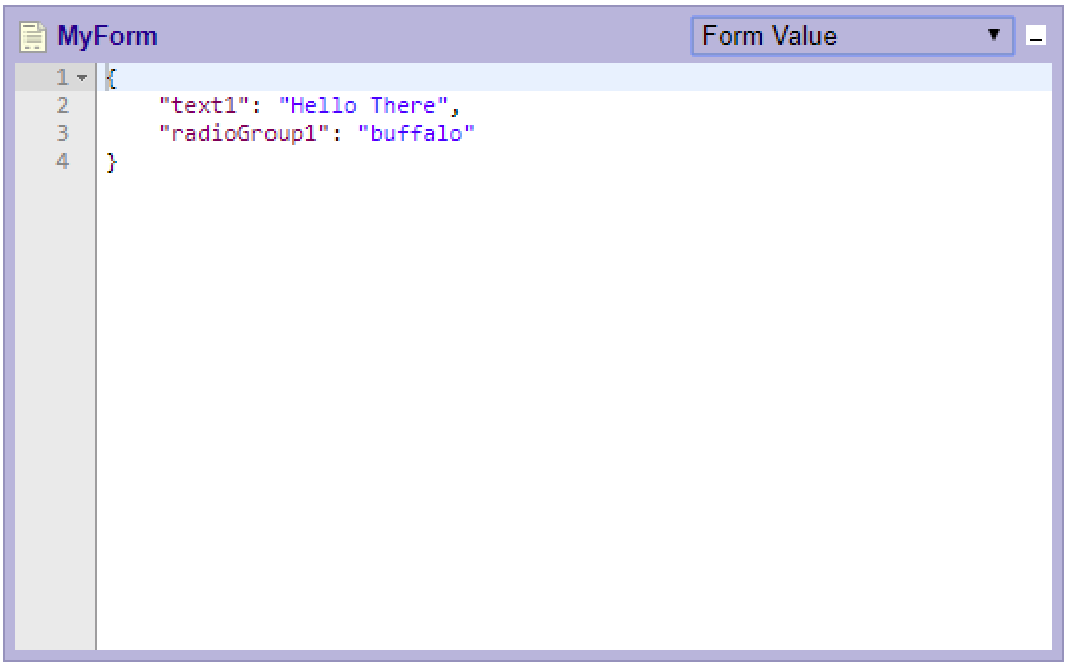
Test Form

Test text entry: Hello There

Radio 1:
antelope
[✓ buffalo]
cat
dog

Here is the form after pressing the save button.

Now if we go to the Form Value view of the Form Data Component, we see what result value is associated with this form content.



```
{
    "text1": "Hello There",
    "radioGroup1": "buffalo"
}
```

This is the form value that is returned for the given form content.

Element Types

The section gives the available types of elements and the configuration rules for each. There are also some additional configuration options listed below in the section Shared Configuration Options.

CheckboxElement

This is a simple, singular check box.

Init Data:

- type - "checkbox"
- label - The label for the checkbox (optional)
- value - initial value: true/false (optional)
- key - The key used to report this value in the form result
- onChange(elementValue,formObject) - handler called when the value of this checkbox changes (optional)

CheckboxGroupElement

This is a group of checkboxes.

Init Data:

- type - "checkboxGroup"
- label - The label for the checkbox group (optional)
- entries - array of values, one for each checkbox (optional)
- value - a list containing the values for the checked checkboxes
- key - The key used to report this value in the form result
- horizontal - if this is set to true the checkboxes will be arranged horizontally, to the extent this is possible. Otherwise they will be placed one to each line. (optional)
- onChange(elementValue,formObject) - handler called when the value of this checkbox changes (optional)

DropdownElement

This is a dropdown element.

Init Data:

- type - "dropdown"
- label - The label for the dropdown (optional)
- entries - array of values in the dropdown
- value - this initial value for the dropdown (optional)
- key - The key used to report this value in the form result
- onChange(elementValue,formObject) - handler called when the value of this checkbox changes (optional)

HtmlDisplayElement

This is an element that displays arbitrary HTML.

Init Data:

- type - "htmlDisplay"
- html - This is the HTML string

HeadingElement

This is a heading element.

Init Data:

- type - "heading"
- text - This is the text for the heading
- level - This gives the weight of the heading, like the weight in an HTML heading.

InvisibleElement

This is an element that holds a value but is non-interactive and not visible.

Init Data:

- type - "invisible"
- value - The value for the element
- key - The key for the element

PanelElement

This is an panel that contains a list of child elements.

InitData

- type - "panel"
- formData - This is the init data for the form, in the same format as the init data for the parent panel.
- onChange - This is a handler for when the element changes. It differs from the onChange handler for the form in the panel only in that the panel form onChange function includes the panel entry as an argument. The Element on onChange function includes the parent panel as an argument, just as is done on the other elements.
- key - The key for the element

RadioGroupElement

This is a group of radio buttons.

Init Data:

- type - "radioButtonGroup"

- groupName - the HTML radio button group name
- label - The label for the radio button group (optional)
- entries - array of values, one for each radio button
- value - the value of the initially checked button (optional)
- key - The key used to report this value in the form result
- horizontal - if this is set to true the checkboxes will be arranged horizontally, to the extent this is possible. Otherwise they will be placed one to each line. (optional)
- onChange(elementValue,formObject) - handler called when the value of this checkbox changes (optional)

SpacerElement

This is an element to add some vertical space to the form.

Init Data:

- type - "spacer"
- height - pixel height for the space (optional - current default = 15px)

TextFieldElement

This is a text field.

Init Data:

- type - "textField"
- label - The label for the text field (optional)
- value - the initial value (optional)
- key - The key used to report this value in the form result
- password - if this is true, the field is a password field (optional)
- size - the character width of the text field. (optional)
- onChange(elementValue,formObject) - handler called when the value of this element changes (optional)
- onChangeCompleted(elementValue,formObject) - handler called when the element loses focus and was changed (optional)

TextareaElement

This is a text area.

Init Data:

- type - "textarea"
- label - The label for the text field (optional)
- value - the initial value (optional)
- key - The key used to report this value in the form result
- rows - The number of rows in the text area. (optional)
- cols - The number of columns in the text area. (optional)
- onChange(elementValue,formObject) - handler called when the value of this element changes (optional)
- onChangeCompleted(elementValue,formObject) - handler called when the element loses focus and was changed (optional)

Custom Element

This is a custom element. It is made by explicitly constructing a configurable element so it has the proper functionality. (This is done because at the time there is no support for "class" in the code. If this is added you can pass in a constructor that extends ConfigurableElement).

Init Data:

- type - "custom"
- key - The key used to report this value in the form result
- builderFunction - This is a function which converts a base ConfigurableElement into the custom element.

SubmitElement

This element provides a submit button and a cancel button, to control the panel

Init Data:

- type - "submit"
- key - The key used to identify the element
- submitLabel - a label for the submit button. Default is "OK"

- cancelLabel - a label for the second button. Default is "Cancel"
- submitDisabled - this enables or disables the submit button (optional)
- cancelDisabled - this enables or disables the cancel button (optional)
- onSubmit(formValue,formObject) - handler called when the value of this element changes (optional)
- onCancel(formObject) - handler called when the element loses focus and was changed (optional)

Additional Element Functions

- submitDisable(boolean) - this enables or disables the submit button
- cancelDisable(boolean) - this enables or disables the cancel button

Shared Configuration Options

In addition to the above listed initialization options, there are some additional options common to all elements.

- state - The state for the form element. The following states are available:
    - apogeeapp.ui.ConfigurableElement.STATE_NORMAL - The element is visible and not disabled.
    - apogeeapp.ui.ConfigurableElement.STATE_DISABLED - The element is visible but disabled.
    - apogeeapp.ui.ConfigurableElement.STATE_HIDDEN - The element is not visible. However, any value it holds is included in the value of the form.
    - apogeeapp.ui.ConfigurableElement.STATE_INACTIVE - The element is not visible and the element value is NOT included in the value of the form.
- selector - This can be used to display an element based on a parent selection. It is most often used for panels but can be used for any element. This is an object with the following fields.
    - parentKey - The key for the parent. The parent must be in the same panel and be before the child.
    - parentValue - This is the value of the parent that triggers display of the child.
    - keepActiveOnHide - If this is true the form value will return the value of any element wether it is hidden or not. This is how a tab would typically work. If the value is false or undefined then only the visible panel is included in the form value.

- onChange(elementValue,formObject) - A standard event for elements. It is further described under each element.

Using the Selector

We will modify the example form above to add an option "custom" to the list of animals, and an additional text field to input the type of the custom animal. We will use the selector on the custom animal type field so that it is only visible is "custom" is selected from the list of animals.

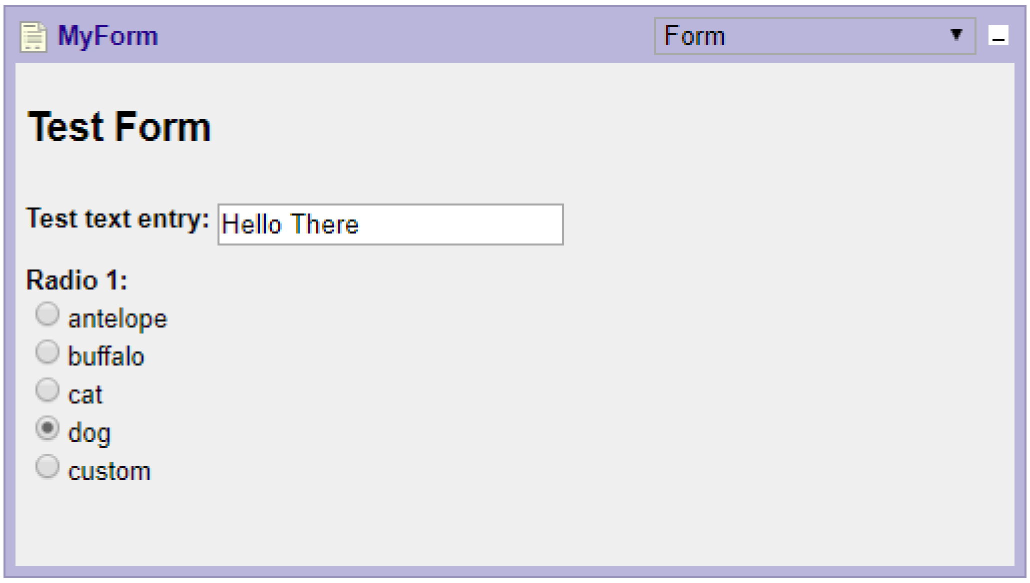
```
[
  {
    "type": "heading",
    "level": 2,
    "text": "Test Form"
  },
  {
    "type": "textField",
    "label": "Test text entry: ",
    "value": "Hello There",
    "key": "text1"
  },
  {
    "type": "radioButtonGroup",
    "label": "Radio 1: ",
    "groupName": "rg1",
    "entries": [
      "antelope",
      "buffalo",
      "cat",
      "dog",
      "custom"
    ],
    "value": "dog",
    "key": "radioGroup1"
  },
  {
    "type": "textField",
    "label": "Custom Animal: ",
    "key": "customAnimal",
    "selector": {
        "parentKey": "radioGroup1",
        "parentValue": "custom",
        "keepActiveOnHide": false
    }
  }
];
```

In the selector on the new text field, the *parentKey* is "radioGroup1", which is the element we want to control the visibility of our new text field.

The *parentValue* is "custom". This means the new text field will be visible only if *radioGroup1* holds the value "custom".

Lastly, we have included the field *keepActiveOnHide*. Since this is false, when the *radioGroup1* is not "custom", the value of this field does not appear in the form value.

Here is the form when we do not select "custom".



MyForm

Form ▼

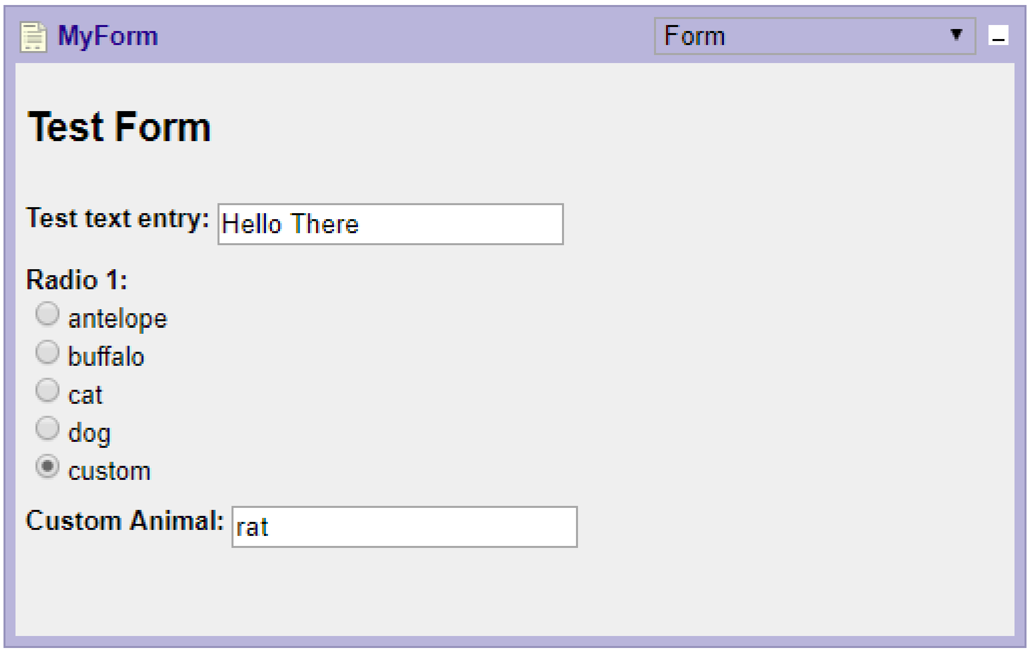
Test Form

Test text entry: Hello There

Radio 1:

- antelope
- buffalo
- cat
- [✓ dog]
- custom

And here is the form when we do select "custom".



Test Form

Test text entry: Hello There

Radio 1:
( ) antelope
( ) buffalo
( ) cat
( ) dog
[✓ custom]

Custom Animal: rat

Additional Reference

In more advanced usage of forms, such as if you are incorporating one in a custom form.

Configurable Form Panel

Methods

- constructor(optionalContainerClassName) - Creates the form, passing the initialization value. Optionally, a container CSS class can be specified.
- configureForm(formInitData) - This method constructs the form inside the panel. It can be called as many times as desired to reconfigure the panel.
  - formInitData format - This is a javascript/json object with the following fields:
    - layout - (REQUIRED) This is an array of initialization elements defining the elements in the form.
    - onSubmit - (OPTIONAL) This is a javscript/json object that adds a submit button to the form. The submit button may also be added as an individual layout element. The fields in this object are:
      - onSubmit - This function is a handler for the submit action. If it is not included there is no submit button.

        - onCancel - This function is a handler for the cancel action. if it is not included there is no cancel button.
        - submitLabel - This is an optional label for the submit button.
        - cancelLabel - This is an optional label for the cancel button.
    - onChange - (OPTIONAL) This handler is called whenever the form changes. The arguments are (formValue,formObject).
    - disabled - (OPTIONAL) If this field is set to true, the form will be disabled.
- getEntry(key) - This gets a Form Element, by key, from the panel.
- getValue() - This gets the value for the form.
- setValue(formValue) - This sets the value for the form.
- getElement() - This returns the DOM element for the panel.
- getChildEntries() - This returns an array of form elements.
- addSubmit(onSubmit,onCancel,optionalSubmitLable,optionalCancelLabel) - This is an additional way of adding a submit entry to a form, besides using the initData for the panel.
- addOnChange(onChange) - This is called if the form changes value. The onChange function is passed the arguments (formValue, formObject)
- setDisabled(isDisabled) - This enables or disables the form.

Configurable Form Elements

Methods

The following functions are available on each form element:

- getKey() - returns the key for this element
- getState() - returns the state for this element
    - the form value. (In all other states the value is included)
- setState(state) - sets the state for this element
- setValue(value) - sets the value for this element
- getElement() - returns the DOM element for this for element
- getForm() - returns the parent form for this element

Custom Components and The HtmlJsDataDisplay

A *Custom Component* is a control that allows the user to create his own custom component, complete with a display defined with HTML and javascript.

For a tutorial on this type of component, click here.

In Apogee, it is also possible to make your own component in javascript and import it into the program, and to share it with others. This component is coded in javascript and must be imported into the workspace as a module. For documentation on these, go to Creating Reusable Components.

The Custom Components

You can define a custom component by writing your own user interface code in HTML and javascript right in the workspace. There are two types of custom components:

- **Custom Component** - This is a simple custom component that has a programmable display. This is convenient for things like output elements like charts or for action elements like a dialog.
- **Custom Data Component** - This is similar to the custom component except it stores a data value. Interacting with the programmable display shows the save bar. Pressing save updates the stored value in the component. This is convenient for creating an object like a plain data table but with a custom display.

Both of these components use the HtmlJsDataDisplay. You don't need to know about data displays to write custom components. This is mentioned here only because this page also serves as a reference for people using this data display for writing reusable components.

UI Generator

For our these components, and for the HtmlJsDataDisplay, we use a *UI Generator* object. This is how we define the user interface for our display. It contains the functions below (all of which are optional). This can be used as a reference for the tutorials on the different custom components that follow.

An important note is that when we code this object in a custom component, in the view marked *uiGenerator()*, we can not access the other tables as we can from most of the other code views. That is because this code is not part of our model. This is just UI code.

We can however pass data into the UI. This will be done with the *setData* method on this object. The data passed to this function will be the output of the function we define in the view called input code, which is the equivalent to the formula in a plain data table.

```
/** This sample class gives the format of the UI Generator object
 * that is used to construct the main display for a custom control.
 * The user should return an object with these functions to implement
 * the functionality of the display. All these methods are optional and
 * any can safely be omitted. As such, a class does not need to be
 * created, any object be passed in. If the class is used, an
 * instance should be returned from the uiGenerator view.
 * In the methods listed, outputElement is the HTML element that
 * contains the form. The admin argument is an object that contains
 * some utilities. See the documentation for a definition. */
var SampleUiGeneratorClass = class {

    /** This can have whatever you want in it. They user will
     * return an an instance rather than the class so this is
     * called by the user himself, if he even shooses to use
     * a class. OPTIONAL */
    constructor() {
    }

    /** This is called when the instance is first compiled into
     * the control. Note the output element will exist but it
     * may not be showing. The method onLoad will be called when
     * the outputElement is loaded into the page. OPTIONAL */
    init(outputElement,admin) {
    }

    /** This method is called when the HTML element (outputElement)
     * is loaded onto the page. OPTIONAL */
    onLoad(outputElement,admin) {
    }

    /** This method is called when the HTML element is unloaded
```

```
     * from the page OPTIONAL */
    onUnload(outputElement,admin) {
    }

    /** This method is the way of passing data into the component.
     * The code here can NOT access the other tables because this code
     * is not part of our model. This object is just UI code.
     * The data passed is the value returned
     * from the user input function when the value updates. OPTIONAL */
    setData(data,outputElement,admin) {
    }

    /** This method is used when save is pressed on the coponents save too
     * if applicable. It retreives an data from the control, such as if th
     * an edit table. OPTIONAL */
    getData(outputElement,admin) {
    }

    /** This method is called id the output element resizes.
     * OPTIONAL */
    onResize(outputElement,admin) {
    }

    /** This method is called before the window is closed. It should
     * return apogeeapp.app.ViewMode.CLOSE_OK if it is OK to close
     * this windows. If this function is omitted, it will be assumed
     * it is OK to close. An alternate return value is
     * apogeeapp.app.ViewMode.UNSAVED_DATA. OPTIONAL */
    isCloseOk(outputElement,admin) {
        return apogeeapp.app.ViewMode.CLOSE_OK;
    }

    /** This method is called when the control is being destroyed.
     * It allows the user to do any needed cleanup. OPTIONAL. */
    destroy(outputElement,admin) {
    }

}
```

Output Element

In the functions above, one argument passed is the outputElement. This is an HTML DOM element in which we should place content for the display.

Admin Object

Another argument passed in the functions above is the admin object. This contains the following utility functions, to be used in the user defined functions above.

```
var admin = {
    /** Returns an instance of the messenger. */
    getMessenger();

    /** Puts the component in edit mode, bringing up the save bar. */
    startEditMode();

    /** Ends edit mode for the component, removing the save bar. */
    endEditMode();
}
```

Constructor

The constructor is not actually used by Apogee. In fact, this doesn't have to be a class at all. It can just be a javascript object with the necessary functions included in it. It is however convenient to write it as a class. The class itself is not passed into Apogee but rather an instance of it is passed. So in this case the constructor will just be called by the user himself.

Init

This function is called as soon as the UI generator object is passed into Apogee. It can be used to do any one time initialization needed. Note however that the *outputElement* may not be loaded on the page. If having the element loaded is a requirement, you can use the *onLoad* function.

onLoad

This method is called when the *outputElement* is loaded onto the web page. This should happen when the display is first created and it can happen subsequently if the *outputElement* is removed and reloaded onto the page, such as when you change views in the component.

OnUnload

This method is called when the *outputElement* is unloaded from the page.

setData

This method is called whenever there is new input data for the component.

In the *Custom Component*, the data passed is the return value of the function defined in the *input code* view. This function will be called each time any dependency in this function is called.

In the *Custom Data Component*, the data passed is the total value of the component, including both the return value of the *input code* view function and the stored data for this object. This is the same as the total value of the component, consisting of a JSON object with the following fields:

- input - This is the return value of the function defined in the *input code* view.
- data - This is the saved value associated with this component.

In other uses of the *HtmlJsDataDisplay*, the programmer passes callbacks in when instantiating the *HtmlJsDataDisplay*. See the *HtmlJsDataDisplay* constructor arguments listed below.

getData

This method is called when the end user presses *Save* in edit mode.

In the *Custom Component*, this is not applicable.

In the *Custom Data Component*, the function should return whatever data is desired for this table.

In other uses of the *HtmlJsDataDisplay*, the programmer passes callbacks in when instantiating the *HtmlJsDataDisplay*. See the callbacks listed for the *HtmlJsDataDisplay* constructor.

onResize

This method is called whenever the end user changes the size of the component.

isCloseOk

This method is called before the component is closed, to see if it is OK to close the component. The return values are:

- apogeeapp.app.ViewMode.UNSAVED_DATA
- apogeeapp.app.ViewMode.CLOSE_OK

destroy

This method is called when the display will be destroyed. This allows the user to cleanup and resources that should be destroyed.

HtmlJsDataDisplay Constructor

When using the *HtmlJsDataDisplay* in a reusable component, the constructor has the following signature. (Note this is not relevant for the *Custom Component* or the *Custom Data Component*.)

```
constructor(viewMode,callbacks,member,html,resource)
```

It has the following arguments:

- viewMode - This is the instance of the view mode that shows this data display. It will be passed directly to the *DataDisplay* base class from which the *HtmlJsDataDisplay* inherits.
- callbacks - These are the callbacks needed for the data display. It will be passed directly to the DataDisplay base class from which the HtmlJsDataDisplay inherits. The following callbacks should be included on this javascript object:
    - getData() - This gets the data that should be loaded into the display.
    - getEditOk() - This method returns true if edit mode is allowed for this component
    - saveData(formValue) - This is called with a the value from the display (form). It should save that value.

- member - This is the member object associated with the component.
- html - This is the HTML content with which to initialize display element content.
- resource - This is the UI generator object for the display.

Additional Content

Besides the UI generator object, the custom components and the HtmlJsDataDisplay take some additional content.

HTML

The HTML is loaded into the output element on creation of the output element. Make sure any IDs in the HTML elements are unique on the page.

CSS

The CSS is used by the HTML content in the element, though it is simply added to the page in the header. As such, make sure the class names used in the CSS are unique on the page.

Input Code and Input Private

For the custom components, the programmer provides a function to give an input value to the component.

In the UI code, there is no access to the model and the other tables in it. We instead have an input function we can define. This can access the other tables. The result of this function is passed to the UI code in the *setData* function of the UI generator.

Destroy On Hide Option

When we create a custom component, or when we edit the properties of the component, there is an option *Destroy on Hide*. The default value is false.

If we set this to true, any time we hide the component display we will destroy it. It will be reconstructed when we show it again. This is done in most of the native components to save resources. However it may take a little extra care in defining the component. If the value is set to false, which is the default, than the display is only created once and then saved, so there is no need to worry about saving state between when the display is destroyed and recreated.

If you do want to tear down the display created when it is not shown, this flag can be set to true. It is typically not needed however.

HtmlJsDataDisplay Options

If we are using a HtmlJsDataDisplay in defining a reusable component, then the data display is part of a View Mode (which is what we are accessing in the view drop down). There are more options here for this same destroy concept.We have the following destroy options:

- apogeeapp.app.ViewMode.DISPLAY_DESTROY_FLAG_NEVER - Do not destroy the view mode.
- apogeeapp.app.ViewMode.DISPLAY_DESTROY_FLAG_INACTIVE - Destroy the display when the view mode is inactive.
- apogeeapp.app.ViewMode.DISPLAY_DESTROY_FLAG_INACTIVE_AND_MINIMIZED - Destroy the display when the view mode is inactive or when the component window is minimized.

Creating Reusable Components

WRITE THIS!