Apogee Custom Components

v1.0.0-p1

Custom Components allow you to make a one-off custom cell from inside an Apogee Workspace. The user can create his own custom cell, complete with a display defined with HTML and javascript.

In Apogee, it is also possible to make your own cell in javascript and import it into the program, and to share it with others. This cell is coded in javascript and must be imported into the workspace as a module. For documentation on these, see documentation for creating reusable cells.

This document uses two example projects. You can get these projects to see them in action at the following URLs:

- https://apogeejs.com/web/gettingStarted/tutorials/more_customCells.json
- https://apogeejs.com/web/gettingStarted/tutorials/more_customDataCells.json

They are also available from the *Getting Started* page on the web site.

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 Cells - 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 Cell - 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.

Coding a Custom Cell

Below we display a sample custom cell in an Apogee workspace.

There are multiple display views.

- **Display** - This is the output of the custom cell that the user will be viewing.
- **Input Code** - This cell is a function that has access to the other workspace cells. This is just like the formula in a data cell, for example. Here we can return a value that we will display inside our custom display.
- **Input Private** - This is used with the *Input Code* view. It is private constants and functions that the *Input Code* view can access.
- **HTML** - This defines HTML for the display. Alternatively the HTML can be defined in javascript in our *uiGenerator*. Notes that there is no isolation from the rest of the page with this HTML so any DOM element IDs should be unique, such as by including the name of the cell in the ID.
- **CSS** - This is CSS to accompany the HTML. Note that there is no isolation from the rest of the page with this CSS so it is recommended you make any class names specific to this element, such as by including the name of the element.
- **uiGenerator()** - This is the javascript code that actually defines the display. See the specific format below.

UI Generator of the *Custom Cell*

This is a javascript object we use to define the user interface for our display. It contains the functions below (all of which are optional).

An important note is that when we code this object in a custom component, in the view marked uiGenerator(), we *do not* have 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.

As we can see below however, we can pass data into the form. The output of the *Input Code* function is passed into the *setData* method below, whenever the output of that function updates.

Here is sample code. We will explay the contents after we also show an example of a *Custom Data Cell*.

```javascript
//We must return a javascript object with the given functions below. Here we
//choose to create that using a class, which we immediately instantiate.
var SampleUiGeneratorClass = class {

    //Standard constructor
    constructor() {
        this.button = null;
        this.textarea = null;

        //this is added just to help us debug
        __customControlDebugHook();
    }

    //We can do any added initialization here
    //init() {
    //}

    //This will be called when the output element is loaded into the DOM
    onLoad(outputElement, admin) {
        this.button = document.getElementById("testComponent_button");
        this.textarea = document.getElementById("testComponent_textarea");

        this.button.onclick = () => {
            var data = this.textarea.value;
            admin.getCommandMessenger().dataCommand("sentData",data);
        }
    }

    //This will be called when the output element is removed from the DOM
    //onUnload(outputEelement, admin) {
    //}

    //This is called with the data value from the "Input Code" section.
    //It is first called _after_ onLoad.
    setData(data, outputElement, admin) {
        if(this.textarea) {
            this.textarea.value = data;
```

```
        }
    }

    //This is called to check if closing the cell should be delayed
    //isCloseOk(outputElement, admin) {
    //}

    //This is called when the component is destroyed
    //destroy() {
    //}

}

return new SampleUiGeneratorClass();
```
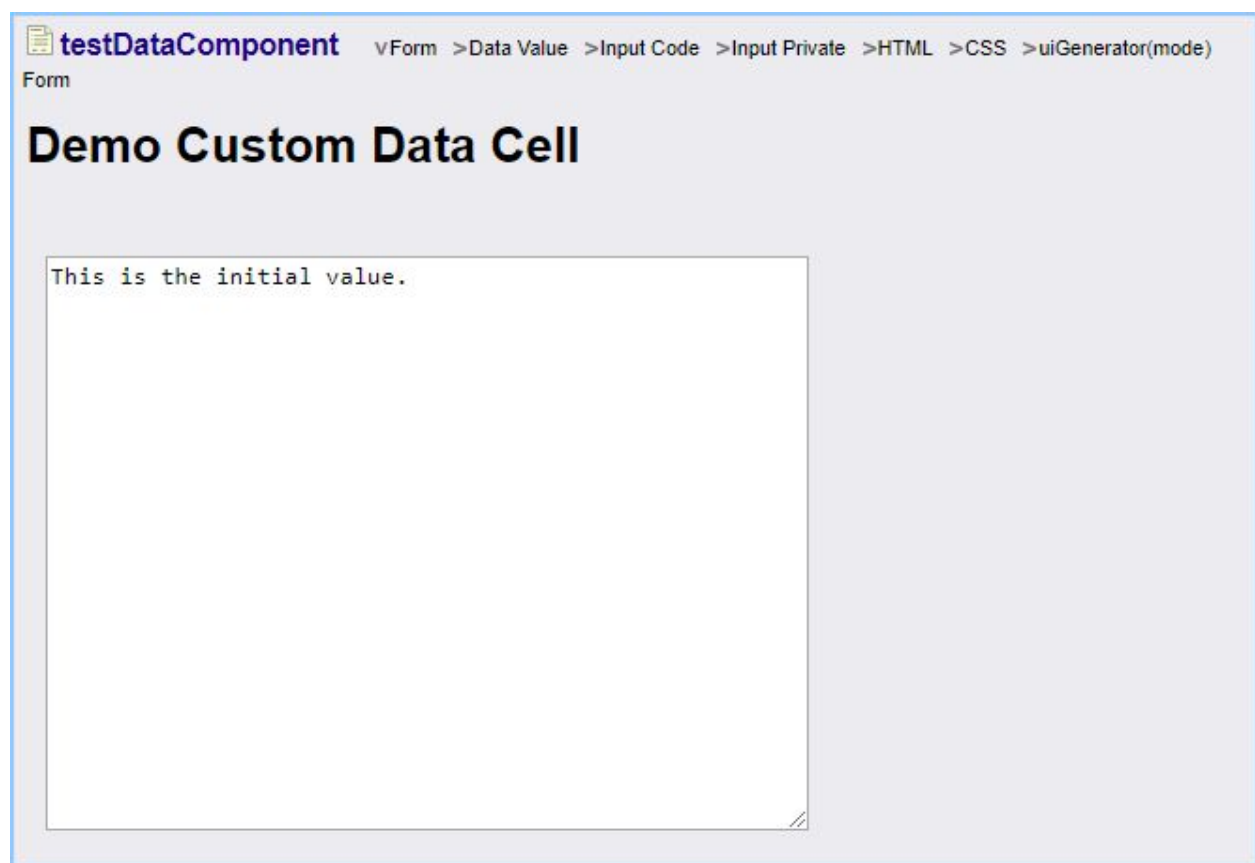
Coding a Custom Data Cell

Here we display a custom data cell.

This is similar to the *Custom Cell* but it adds one more data view, the *Data Value*, and it treats the *Input Code* view differently.

- **Data Value** - In the *Custom Data Cell* the stored value for the cell is held here. We can view the stored value, or we can write to this field and it will modify the display.
- **Input Code** - Just like this the *Custom Cell*, this is a function that has access to the other workspace cells. In this case the result is not the value displayed in our form but rather it is a value used to construct the form. (The *Data Value* shown above is the value that is displayed in the form.)

UI Generator of the *Custom Data Cell*

And here is the corresponding uiGenerator code. Again this is very similar to the *Custom Cell* but it uses a few additional things, the methods *getData* and *setDisplayData*.

```javascript
//We must return a javascript object with the given functions below. Here we
//choose to create that using a class, which we immediately instantiate.
var SampleUiGeneratorClass = class {

    //Standard constructor
    constructor() {
        this.title = null;
        this.textarea = null;

        //this is added just to help us debug
        __customControlDebugHook();
    }

    //We can do any added initialization here
    //init() {
    //}

    //This will be called when the output element is loaded into the DOM
    onLoad(outputElement, admin) {
        this.title = document.getElementById("testDataComponent_heading");
        this.textarea = document.getElementById("testDataComponent_textarea");

        //we need to trigger a start to edit mode
        //the admin input argument provides the function to start edit mode, which is
        //the same mode used for other edit cells like the data cell.
        this.textarea.oninput = () => admin.startEditMode();
    }

    //This will be called when the output element is removed from the DOM
    //onUnload(outputEelement, admin) {
    //}
```

```
        //This function passes in the data to create the display, which is the
        //data we creared with the input code. In this case we
        //are just setting the title. Note that this is distinct from the
        //data value of the display, which is set in setData below.
        setDisplayData(data) {
            this.title.innerHTML = data;
        }

        //This is the data content of the display. It is set from the data
        //shown in the Data Value view.
        //It is first called _after_ onLoad.
        setData(data, outputElement, admin) {
            if(this.textarea) {
                this.textarea.value = data;
            }
        }

        //This is called to retrieve data from the cell, if we are using the
        //built in edit capabilities
        getData(outputElement, admin) {
            if(this.textarea) {
                return this.textarea.value;
            }
            else {
                return "";
            }
        }

        //This is called to check if closing the cell should be delayed
        //isCloseOk(outputElement, admin) {
        //}

        //This is called when the component is destroyed
        //destroy() {
        //}

}

return new SampleUiGeneratorClass();
```

UiGenerator

The *Custom Cell* and *Custom Data Cell* both use the same output display, the
HtmlJsDataDisplay. The UiGenerator should provide the methods listed below.

It is used slightly differently in the case of the *Custom Cell* and *Custom Data Cell*, as described
below.

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(outputElement,admin)

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(outputElement,admin)

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(outputElement,admin)

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

setDisplayData(displayData)

Here data is passed in which is used in constructing the display.

Of these two cells, this is used only in the Custom Data Cell. It takes the output of the Input Code function.

setData(data,outputElement,admin)

This method is called whenever there is new input data for the content of the cell. If we were making an editor, this would be the data that is being edited.

This is used slightly differently for the *Custom Cell* and the *Custom Data Cell*.

In the *Custom Cell* this is the output of the *Input Code* function.

In the *Custom Data Cell* this is the value stored in the *Data Value* view. This is the stored value for the cell, which can also be accessed externally.\

getData(outputElement,admin)

This method is used for the built in edit logic. It is called when the user presses the save button. This is used only for the *Custom Data Cell*.

When this function is called for the *Custom Data Cell*, it should return the value stored in the display. For example if the cell were an editor it would return the value being edited.

In the *Custom Data Cell*, this is then used as the *Data Value*.

isCloseOk(outputElement,admin)

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(outputElement,admin)

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

UiGenerator Input Arguments

There are two input arguments that appear repeated in the UiGenerator functions.
- **outputElement** - This is the HTML element that contains the display.
- **admin** - This is a javascript object that provides access to some needed functionality. It includes the following methods:
  - **getCommandManager()** - This returns an instance of the Command Messenger This can be used to send data to other cells. See below.
  - **startEditMode()** - This is used to start the built-in edit mode. This is typically used in Custom Data Cell and not Custom Cell.
  - **endEditMode()** - This is used to end the built-in edit mode. This is typically used in Custom Data Cell and not Custom Cell.

Command Messenger

The Command Messenger is essentially the same thing as the Apogee Messenger. They both can send data to other cells. The difference is that the Command Messenger is used for UI actions taken by the user. The Apogee Messenger is used from within a formula for a cell.

The command Messenger has the following functions:

`commandMessenger.dataCommand(relativeCellName,dataValue)` - This sends the given data value to the cell with the given name, as would be entered as a variable in the current code.

Note that this works just like a user entering data from the user interface. It is an asynchronous action and does not change other cell values while a current calculation is in process. See documentation on the apogee messenger for more information.

`commandMessenger.compoundDataCommand(commandArray)` - This does the same thing as the command messenger except it sets multiple values. The command array has the following format:

`[[relativeCellName1,dataValue1],[relativeCellName2,dataValue2]...]`

Destroy On Hide Option

If you go to "Edit Properties" for a *Custom Cell* or *Custom Data Cell*, there is a checkbox marked "Destroy on Hide". The default value is false.

This checkbox controls whether or not the HTML display is destroyed if the component is hidden. This can usually be ignored. However, if there is a reason you want to destroy the HTML elements, such as if it is expensive to keep alive and it will typically be hidden, or if you want to keep it alive, such as if it is very costly to construct, then you can use this checkbox.