

Apogee - Getting Started

Apogee - Getting Started

Apogee Quick Start Video

...and answer to the question ***What do you do with Apogee?*** (This is the video on the home page.)



Apogee Quickstart Video

<https://www.youtube.com/watch?v=-yp7rXglEug>

Workspace from video: [Apache Log Example Workspace](#).

Try an exercise for yourself!: [String Translation Table Do-It-Yourself Workspace](#)

Apogee Tour

This video gives a quick tour of how to get around the application, to help you get started.



Apogee Quick Tour

https://www.youtube.com/watch?v=WvvRgn-s_PA

Additional Videos

If you want to learn more about Apogee here are some additional videos. Or, dive in for yourself by going through the tutorials.

Videos:

- [What is Apogee? An Overview](#) - This is a more in depth description of the idea behind Apogee.

Tutorials

Follow the links below for the tutorials, or proceed through them sequentially using the links at the bottom of the page.

Basics

These tutorials cover some basic usage of Apogee.

- [Populating a Data Table](#) - Basics for writing table formulas.
- [Data Views in a Data Table](#) - The main data views for a data table.
- [Defining Functions](#) - Different ways of defining functions.
- [Folders and Scope](#) - This illustrates variable scope with regards to folders.
- ["Immutable" Values Gotcha](#) - Table values are immutable, so sometimes you have to make copies values.
- [Debugging](#) - Using the debugger to help you code.
- [Reusing Code](#) - How to reuse code in other workspaces - sort of like a library
- [The Messenger, and Functional Programming](#) - How to push data to other tables, a seeming cheat on "no side effects"

Mode Components

These tutorials cover using additional types of components in Apogee.

- [Configurable Form Components](#) - How to use *FormDataComponent* and *DynamicForm*.
- [Custom Components](#) - Making a custom control right in your workspace.
- [Folder Functions](#) - Writing a function with tables, rather than lines of code.
- [Roll Your Own Components](#) - Making custom components others can use.

Tutorials - Basics

Populating a Data Table



It might be helpful watching the *Tour of Apogee* video on the [Getting Started](#) page to do this tutorial.

Create a new workspace

In the new workspace, add a data table with a static value: ([example](#))

1. Create a data table name "year". Use the default data view *colorized*.
2. Inside the "data" view of the table place the number *1980*.

Add a data table with a simple formula: ([example](#))

1. Create a data table named "url".
2. Inside the "formula" view, add the single line formula below, creating a URL using a javascript string template literal and the value from our "year" table. In a formula, tables are treated as normal javascript variables.

```
return `https://www.apogeejs.com/web/examples/population/us${year}.json`;
```

Add a data table with an asynchronous formula - web request: ([example](#))

1. Create a data table named "populationData".
2. Inside the formula view, add a single line formula, requesting a json object from the given URL.

```
return apogee.net.jsonRequest(url);
```

The formula for a data table must be synchronous. If you want to do an asynchronous action, like a web request, you can return a Promise object from the formula.

While the promise is pending, a yellow banner is displayed. When the promise resolves, on success the banner will clear. On failure an error will be displayed.

This function in the code above is from the apogee library. It is like the javascript "fetch" function except its promise returns either the body of the request, converted into a json object, or it throws an error.

Modify a data table so it throws an error: (example)

1. On the "url" table, modify the formula so it just throws an error.

```
1  throw new Error("This is an error we generated ourselves!");
2
3  //return `https://www.apogeejs.com/web/examples/population/us${year}.json`
```

In apogee, errors are displayed to the user as a red banner on the table. Additionally, any table that depends on that table will also show a red banner with the message that a table on which it depends has an error.

Modify a data table so it returns INVALID VALUE: (example)

1. On the "url" table, modify the formula so it just returns apogee.util.INVALID_VALUE.

```
1  return apogee.util.INVALID_VALUE;
2
3  //return `https://www.apogeejs.com/web/examples/population/us${year}.json`
```

If you return apogee.util.INVALID_VALUE, the table displays a gray banner. Additionally, any table that depends on that table will also show a gray banner. This can be done to prevent downstream tables from trying to calculate a value if one table does not have a valid value.

The purpose of this feature is so you don't have to do null checking or the equivalent in your functions. This is especially handy when you have a long cascade of tables based on a single value. Apogee does the check for you, and does not execute the code if it depends on a table with an INVALID VALUE.

This does not depend on whether the executed path will use the variable, only if it is present. What this means is you *can't* do an INVALID VALUE check and make a decision based on that. That is a feature that should possibly be added in a future version (along with a way of handling an error in parent table).

Finished!

This concludes the tutorial. From this you should have learned some of the different ways of assigning a value to a data table.

Data Views in a Data Table

Example workspace: [Data View Example Workspace](#)

The workspace linked above contains four tables, with different data views. To set the data view, you select from the "Data View" dropdown on the properties dialog box, either when creating the table or when you edit the properties of the table.

The data view selected does not affect the value or internal mechanics of the table, other than how the value is presented.

- **gridTable** - This is a data table with the view set to *Grid*. It shows the value of the table in a grid, like a spreadsheet. This makes it easy to paste in value from a spreadsheet, or copy values to a spreadsheet.
- **jsonOfGridTable** - This is a data table with a standard JSON view (*colorized*). The formula gives it the same value as *gridTable*. This shows you the JSON structure of the value. Here it is an array of arrays.
- **textDataTable** - This is a data table with the view set to *Text Data*. This is good when you want to work with text and not format it like a JSON.
- **jsonOfGridTable** - This is a data table with a standard JSON view (*colorized*). The formula gives it the same value as *textDataTable*. This shows you the JSON structure of the value. Here it is the text value with start and end quotes, and proper escaping inside.

You can also change the data view of any table by going to "Edit Properties" (click on the icon in the table upper left to get the menu) to see the different views of the data for a given table.

Defining Functions

Example workspace: [Defining Functions Example Workspace](#)

The workspace linked above contains five tables (plus the notes table). Four of them contain formulas, each multiplying the elements of the other table by a factor of two, with alternate ways of defining the needed function.

- **plainValue** - This data table contains a plain static value, an array of numbers.
- **derivedValue** - This table uses the array map function to multiply each element of *plainValue* by two. The function called by *map* is defined inline.
- **alternateDerivedView** - This table also uses the array map function to multiply each element of *plainValue* by two. However, it defines the function called by *map* elsewhere. It puts it in the *private code* section of the table. The view dropdown shows the different sections of the table, including the data, the formula and the private code sections. The private code section allows you to put constants and functions accessed from your formula, to keep the code a little cleaner.
- **anotherAlternateDerivedView** - This table also uses the array map function to multiply each element of *plainValue* by two. However, it uses an external function in the call to the array *map* function. See *testFunction* below.
- **testFunction** - This is an external function, which is used by the data table *anotherAlternateDerivedView*. A function table can be created as you would create a data table, except you select *Add Function* instead of *Add Data Table*. A function table behaves like a normal javascript function when called by other tables.

There is one more way to make a function inside of Apogee, a *folder function*. We will cover that later.

Folders and Scope

Example workspace: [Folders and Scope Example Workspace](#)

The workspace linked above contains several data tables and folders. A folder can be created the same way you create a data table, except you select *Add Folder* instead of *Add Data Table*.

A folder behaves the way you think it would, basically added as a path prefix when accessing a child table. This workspace shows the general scope rules used in Apogee.

There are four folders in this example, the top level folder in the workspace, called *Model*, two child folders inside of *Model*, called *folder1* and *folder2*, and one folder inside of *folder1*, called *folder11*.

Each of these folders contains a table *x* which contains a string telling which folder it is in.

Each folder also contains some test tables, in which the formula *simple* returns a variable named *x*, possibly with a path prefix.

The following rule is used to locate a variable:

- When a variable is access from a formula of a given table, that variable (table) is looked for inside the folder of the table whose formula we are reading.
- If the variable (table) is not found in the local folder, parent folders are checked for the variable, sequentially.
- In order to access a table from a child folder, a prefix on the name must be used giving the folder path to the table.

You can experiment renaming (or deleting) the table called *x* in a given folder and see who this effects the test tables.

"Immutable" Values Gotcha

Example workspace: ["Immutable" Values Example Workspace](#)

In Apogee, the value of a table is immutable, meaning you can't modify it once it is set. Well, you can always edit the workspace, but you can not modify a remote table from within a calculation. This example project illustrates exactly what this means.

The example workspace contains three tables (and the notes table):

- **coordinates** - This holds a manually defined array of coordinate pairs.
- **badModifiedCoordinate** - This table takes one of the coordinate pairs from the *coordinates* table and tries to modify it. This throws an error because modifying the coordinate pair as is done here would also modify the value of the *coordinates* table. The error message says "Cannot assign to read only property..." because the value of the *coordinatestable* is read only once it has been set.
- **goodModifiedCoordinate** - This table does the same thing as the table *badModifiedCoordinate* except it first makes a copy of the data before it modifies it. In this case, modifying the copied data does not attempt to modify the *coordinates* table. The return value from this copy function is not read only, so we can change its value.

The reason Apogee works this way is that once you make a change to the workspace, Apogee internally determines the order in which it must calculate the table values so that every table is calculated before it is used in the formula of another table. If one table could be modified during the calculation of another table, then the determination of the order to calculate tables would be impossible (or at least a lot harder).

Another way to say this rule is that there are "no side effects". We don't want the calculation of one table to affect the values of the other tables. This is a tenet of functional programming, and sure enough, Apogee is a form of functional programming, just like a spreadsheet.

Debugging



<https://www.youtube.com/watch?v=zBinwTBBbiY>

Example workspace: [Debugging Example Workspace](#).

Reusing Code

content TBD!

Example workspace: [Reusing Code](#)

The Messenger, and Functional Programming

In the tutorial [Immutable Values "Gotcha"](#), we stated that the value of a table is immutable once it is set. Specifically, this means you can not modify the value of one table from the formula of another. This is done so there are no side effects, so we can properly determine the order to calculate the tables.

However, there actually *is* a way to modify the value of one table from the formula of another table. And, this actually respects the requirement of no side effects, so that we can determine the proper order of calculation of the tables.

Before we talk about this, we will talk about how you modify the value of a table from *outside* the formula of a table. You use the keyboard. If we could not edit the workspace, you couldn't do anything with the application.

Editing the Workspace

You edit a table by either entering a value explicitly or updating the formula. When you press *save*, Apogee recalculates the workspace. Given the fixed values or formulas for each table, it determines the proper order to evaluate all the formulas. It is during this calculation that there are no side effects.

If you then create another table or edit an existing table, once you press *save* the process will repeat. It is treated as an entirely new system and we recalculate the value of all the tables, again not allowing side effects during the calculation.

The Messenger

There is something called the *messenger*. You can access this from a formula for a table and use it to send a new value to another table. However, it doesn't change the value of the table right away. It waits until the current calculation is over. Then it sets the new value.

This is the same as if you decided to update the workspace yourself with a new value, just as we did in the above section *Editing the Workspace*. In fact, the two processes use the

same code internally. As above, this does not violate the no side effect rule during a calculation.

Examples

Basic Usage Example

Example workspace: [Basic Usage Sample Workspace](#)

Here we show the basic usage of the messenger from a formula. This example contains three tables (and the notes table):

- **valueToCopy** - This holds a simple manually defined value.
- **test** - This table has a formula that calls the messenger, and sends the value from *valueToCopy* to the table called *remoteTable*. It also returns a dummy value for its own value.
- **remoteTable** - This table has no formula. It will receive its value from the messenger. To see this, update the value in the *valueToCopy* table.

User Input Example

Example workspace: [User Input Sample Workspace](#)

This example is one of the most common cases of using the messenger. It uses a control called a *Dynamic Form Table*, which is basically a customizable form that acts as a dialog box. There are other customizable forms too, such as ones that hold a value just like a data table. These form tables are introduced elsewhere.

From the *userInputForm*, view the code by selecting *code* from the *View Dropdown* in the top right corner. The code contains the layout of the form and, at the top, an *onSubmit* callback function. From within this function, the messenger is accessed. It is used to send the value of the form to the table *inputValue*

To see this work, just enter data into the form and press *OK*.

Incidentally, the code that defines the form layout for the *userInputForm* references the table *inputValue*, which is the table we write to when we press *OK*. If you update the data in the table *inputValue* manually, it will be reflected in the form. (But you might get an error. We didn't put much error checking in the form code.)

Iterative Calculation Example

Example workspace: [Iterative Calculation Example Workspace](#)

Generally in functional programming you must do recursive functions rather than iterative functions, which is OK. In Apogee, however, you can choose to do an iterative calculation, using the messenger. Sometimes it is more natural to do this, such as if you are simulating a feedback control system.

In this sample workspace, we calculate the factorial function iteratively. (Of course, this particular example is one that is very natural to do recursively, but we are just doing this as an example.)

This example contains four tables (and the notes table):

- **desiredN** - This holds the value whose factorial you would like to calculate.
- **nMinus1Factorial** - This should be initialized to some known factorial, such as $0!$, where $n=1$.
- **n** - This table should be initialized to the proper value of n associated with the *nMinus1Factorial* table.
- **nFactorial** - This is the only table with a formula. It contains the simple formula $n! = n * (n-1)!$. In addition, it also uses the messenger to iterate the values of the **nMinus1Factorial** and **n** tables.

To try this out, update the value in the *desiredN* table. FYI, the largest factorial that can be calculated here is 171.

You may notice the calculation is very slow, so that you can actually see the values changing. The Apogee calculation itself is not that slow, however, there is a lot of overhead with each iteration of the calculation, mainly from the UI. So, for now at least, doing a lot of iterations has a high performance cost. The messenger works best for things like the previous example, user input.

Tutorials - More Components

Configurable Form Components

This tutorial covers the components *Dynamic Form* and *Form Data Table*. They are components that both use the configurable form widget to define a form that the user can interact with.

A Dynamic Form is a table that is like a non-modal dialog box. Here the programmer configures the layout of the form, which should include a submit button to take action. The programmer defines a handler for the submit button that takes the desired action.

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 model code.

In both of these components the programmer must define the layout for the form. More detail about using the configurable form is available in the programming guide [*Configurable Forms*](#).

To learn about these forms, click on the links below or follow the *Next* link at the bottom of the page.

- [**Dynamic Form Tutorial**](#)
- [**Form Data Table Tutorial**](#)

Dynamic Form

This component displays a form which is dynamically configured by the user. The form acts like a modal dialog box. It is typically used to trigger an action.

Unlike a regular data table, when the user edits the form there is no save bar that appears. There is also no way to save the form content for this component in this component itself. To do that, use the [Form Data Table](#). For the *Dynamic Form*, there should be a submit button included, for which the programmer must define the action.

This component includes the following views:

- Form - This displays the form.
- Code - This should be used to enter a function body returning the form initialization info. For the format of the return value, see the definition of the form initialization data in the *ConfigurableForm* documentation.
- Private - This allows for an private constants or functions needed by the function body initializing this form.
- Notes - This holds any desired notes for the component.

Example Form

The following example describes the workspace: [Example Workspace](#)

The code for this component should return a layout object. The layout object is a javascript/JSON object which gives the list of configurable elements and the initialization of each, defining the form.

The following code defines a form with the following elements:

- Title
- Text Field
- Radio Button Group
- Submit Element

```
myForm Code
1 var defaultValue = {
2   "text1": "Hello There",
3   "radioGroup1": "dog"
4 }
5
6 //we will use the messenger to write the data to the myOutput table
7 var onSubmit = (formValue,formObject) => {
8   apogeeMessenger.dataUpdate("myOutput",formValue);
9 };
10
11 //on cancel we will reset the value of the form to the default
12 var onCancel = (formObject) => {
13   formObject.setValue(defaultValue);
14 };
15
16 return [
17   {
18     "type": "heading",
19     "level": 2,
20     "text": "Test Form"
21   },
22   {
23     "type": "textField",
24     "label": "Test text entry: ",
25     "value": defaultValue.text1,
26     "key": "text1"
27   },
28   {
29     "type": "radioButtonGroup",
30     "label": "Radio 1: ",
31     "groupName": "rg1",
32     "entries": [
33       "antelope",
34       "buffalo",
35       "cat",
36       "dog"
37     ],
38     "value": defaultValue.radioGroup1,
39     "key": "radioGroup1"
40   },
41   {
42     "type": "submit",
43     "submitlabel": "Submit",
44     "cancellabel": "Reset",
45     "onSubmit": onSubmit,
46     "onCancel": onCancel
47   }
48 ];
```

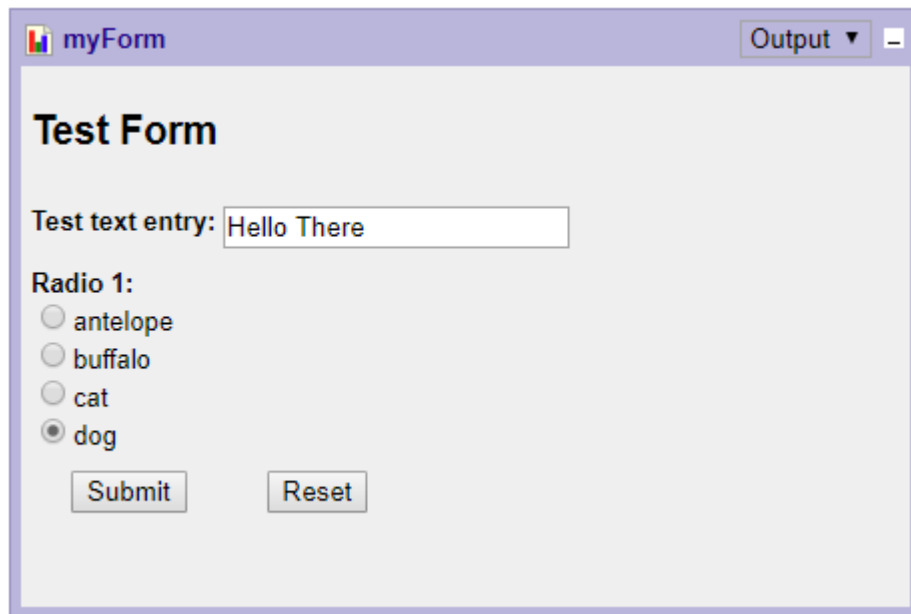
To access this code, view the example workspace

The submit element is given callbacks that define the submit and cancel functions, which we have labeled "Submit" and "Reset". The functions are defined at the top of the code.

The function *onSubmit* uses the **messenger** to send the value of the form to the table *myOutput*.

The function *onCancel* resets the value of the form to the default value, which we defined at the top of the code.

The resulting form looks like this:



The image shows a web browser window with a purple title bar labeled 'myForm'. Inside the window, there is a form titled 'Test Form'. The form contains a text input field labeled 'Test text entry:' with the text 'Hello There' entered. Below this is a section labeled 'Radio 1:' with four radio button options: 'antelope', 'buffalo', 'cat', and 'dog'. The 'dog' option is selected. At the bottom of the form are two buttons: 'Submit' and 'Reset'. In the top right corner of the browser window, there is an 'Output' dropdown menu and a close button.

To see this in action, see the example workspace listed at the top of this section.

More Information

For more information about using the configurable form see the programming guide [Configurable Forms](#).

Form Data Table

This component is similar to the Dynamic form except it allows for saving the result of the form in the component.

It includes a similar layout, but here we do not need to include a submit element. When the user interacts with the form, the *save bar* appears, just as when the user edits the JSON value in a data table. When the user presses save, the content of the form is saved.

This component includes the following views:

- Form - This displays the form.
- Layout Code - This should be used to enter a function body returning the form initialization info. For the format, see the definition of the form initialization data in the *Configurable Form* documentation.
- Layout Private - This allows for an private constants or functions needed by the function body initializing this form.
- isValid(formValue) - This view allows for entry of the isValid function for when the user presses *save* from the form.
- isValid Private - This allows for an private constants for functions needed by the isValid function.
- Form Value - This shows the saved data for the form. It can be edited to set the form value, or used to read the return value of the form.
- Notes - This displays any desired notes.

Example Form

The following example describes the workspace: [Example Workspace](#)

We use a similar layout to the one in our [Dynamic Form example](#) except here we omit the submit element. We also can of course omit the submit callback functions. The return value is a pure JSON object in this case.

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

- Title

- Text Field
- Radio Button Group

```
1  [  
2    {  
3      "type": "heading",  
4      "level": 2,  
5      "text": "Test Form"  
6    },  
7    {  
8      "type": "textField",  
9      "label": "Test text entry: ",  
10     "value": "Hello There",  
11     "key": "text1"  
12   },  
13   {  
14     "type": "radioButtonGroup",  
15     "label": "Radio 1: ",  
16     "groupName": "rg1",  
17     "entries": [  
18       "antelope",  
19       "buffalo",  
20       "cat",  
21       "dog"  
22     ],  
23     "value": "dog",  
24     "key": "radioGroup1"  
25   }  
26 ]
```

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

```
1 return [  
2   {  
3     "type": "heading",  
4     "level": 2,  
5     "text": "Test Form"  
6   },  
7   {  
8     "type": "textField",  
9     "label": "Test text entry: ",  
10    "value": "Hello There",  
11    "key": "text1"  
12  },  
13  {  
14    "type": "radioButtonGroup",  
15    "label": "Radio 1: ",  
16    "groupName": "rg1",  
17    "entries": [  
18      "antelope",  
19      "buffalo",  
20      "cat",  
21      "dog"  
22    ],  
23    "value": "dog",  
24    "key": "radioGroup1"  
25  }  
26 ];
```

This creates the following form:

Test Form

Test text entry:

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.

MyForm Form

Edit: Save Cancel

Test Form

Test text entry: Hello There

Radio 1:

- ☐ antelope
- ☒ buffalo
- ☐ cat
- ☐ dog

Pressing *Save*...

MyForm Form

Edit: Save Cancel

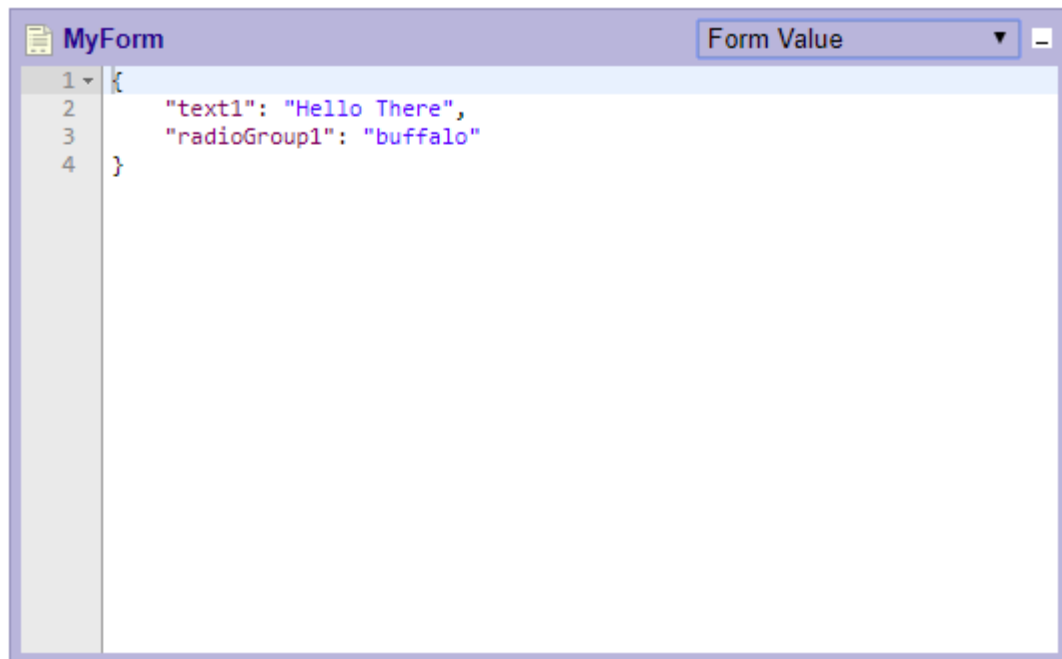
Test Form

Test text entry: Hello There

Radio 1:

- ☐ antelope
- ☒ buffalo
- ☐ cat
- ☐ dog

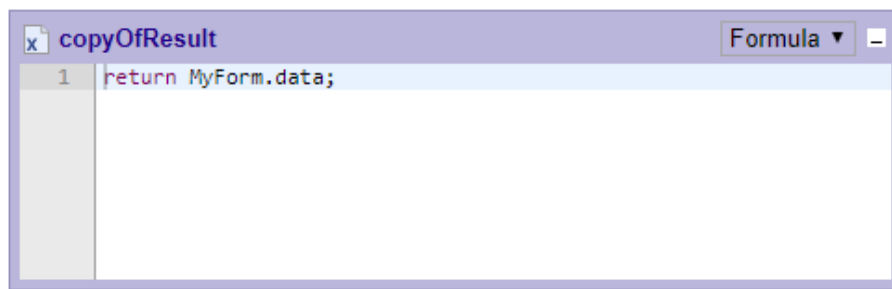
Now if we go to the Form Value view of the Form Data Table, we see what result value is associated with this form content.



Accessing the Form Data From another Table

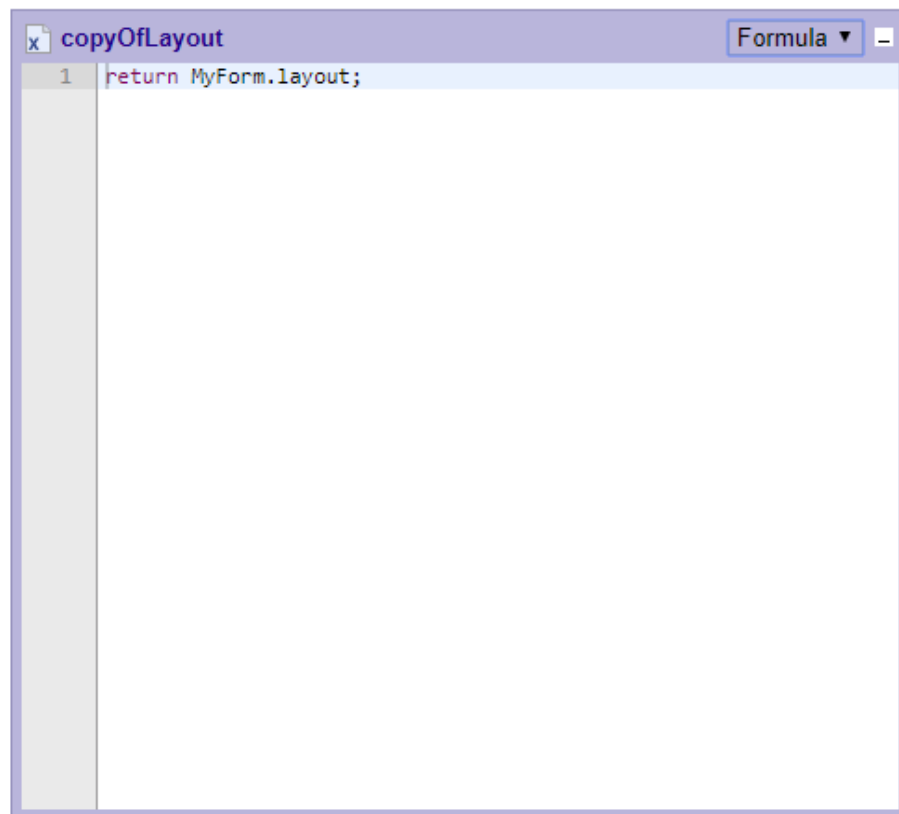
In the workspace, there are two additional tables, *copyOfResult* and *copyOfLayout*. These show how you can externally access the value (*copyOfResult*) and even the layout (*copyOfLayout*) of the Form Data Table.

Here is the code to access the values from the form externally.



The screenshot shows a code editor window with a purple title bar. The title bar contains a small 'x' icon, the text 'copyOfResult', a 'Formula' dropdown menu, and a minus sign. The editor area has a light blue background. A single line of code, 'return MyForm.data;', is highlighted in blue. The line number '1' is visible in the left margin.

```
1 return MyForm.data;
```



The screenshot shows a code editor window with a purple title bar. The title bar contains a small 'x' icon, the text 'copyOfLayout', a 'Formula' dropdown menu, and a minus sign. The editor area has a light blue background. A single line of code, 'return MyForm.layout;', is highlighted in blue. The line number '1' is visible in the left margin.

```
1 return MyForm.layout;
```

And here are the resulting values of these tables.



Compound Tables

The FormDataTable is an example of a compound table. This is actually a folder object, to which the user can not add tables. This folder contains three tables:

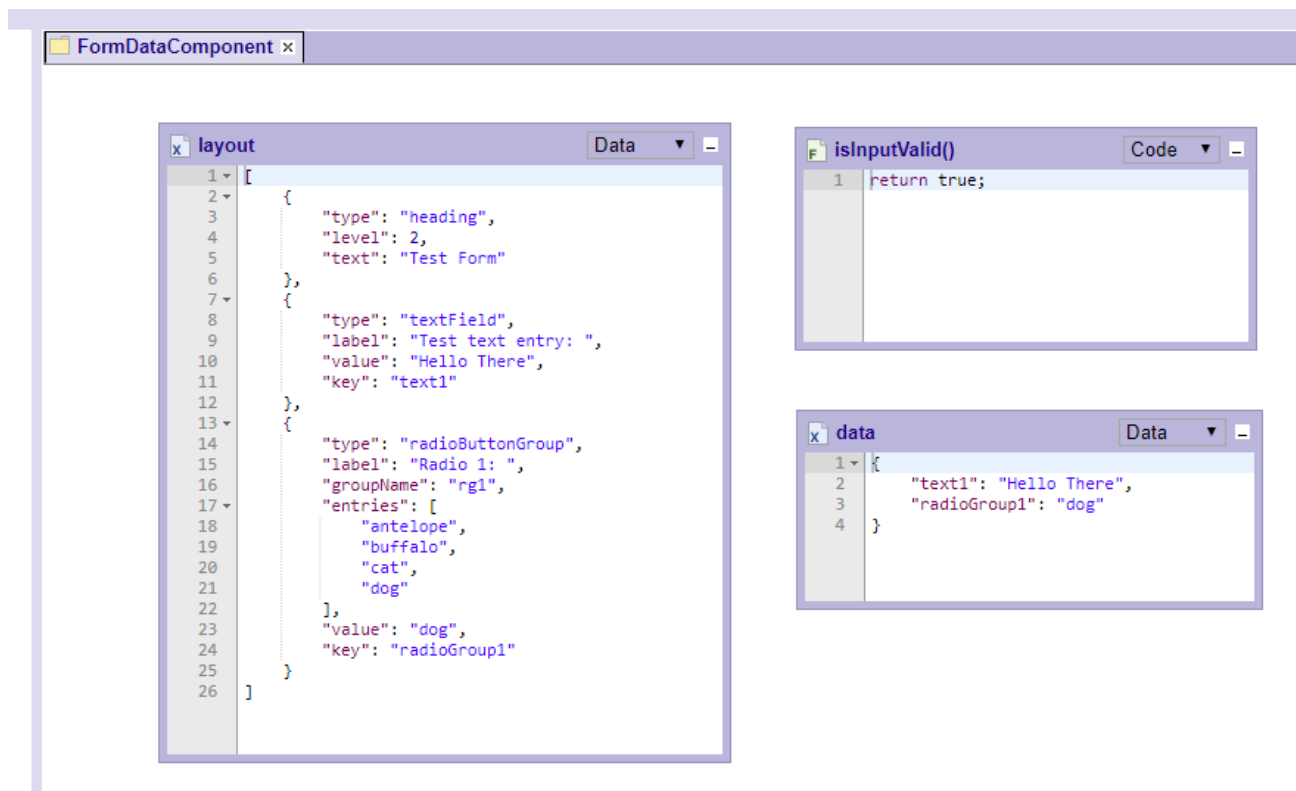
- layout - The value of this table is the layout for the form, or the return value of the layout code function. (Note that this is not a JSON but a javascript object, in general. So the "equivalent" folder below is not really equivalent to the component value in general.)
- data - This table holds the form value. It can be used both to initialize the form and read the form result.
- isValid - This is a function that is called with the argument formValue. When the form is submitted this is called to validate the date. It should return one of the

following values. If a different value is returned, an error message is shown.

- true (boolean value) - This means the data is valid
- error message (string) - This error message will be shown to the user and the form data will not be saved.

However, Apogee does not display this as a folder, it instead appears as a control with the form functionality.

If this were displayed as a folder, it would look something like this:



Compound tables are used in other places as a simple means to put more complex functionality into the components without making more complex underlying tables.

More Information

For more information about using the configurable form see the programming guide [Configurable Forms](#).

External Libraries and Controls

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 modal 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.

UI Generator

Both these components use a *UI Generator* object. This is how we define the user interface for our component. 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 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*.

```
1  /** This sample class gives the format of the UI Generator object
2   * that is used to construct the main display for a custom control.
3   * The user should return an object with the below functions to create
4   * the functionality of the display. All these methods are optional and
5   * any can safely be omitted. As such, a class does not need to be
6   * created, any object can be passed in. If the class is used, an
7   * instance should be returned from the uiGenerator view.
8   * In the methods listed, outputElement is the HTML element that
9   * contains the form. The admin argument is an object that contains
```

```

10  * some utilities. See the documentation for a definition. */
11  var SampleUiGeneratorClass = class {
12
13      /** This can be whatever you want. They user will
14       * return aninstance rather than the class so this is
15       * called by the user himself, if he even chooses to use
16       * a class. OPTIONAL */
17      constructor() {
18      }
19
20      /** This is called when the instance is first compiled into
21       * the control. Note the output element will exist but it
22       * may not be showing. The method onLoad will be called when
23       * the outputElement is loaded into the page. OPTIONAL */
24      init(outputElement,admin) {
25      }
26
27      /** This method is called when the HTML element (outputElement)
28       * is loaded onto the page. OPTIONAL */
29      onLoad(outputElement,admin) {
30      }
31
32      /** This method is called when the HTML element is unloaded
33       * from the page OPTIONAL */
34      onUnload(outputElement,admin) {
35      }
36
37      /** This method is the way of passing data into the component.
38       * The code here can NOT access the other tables because this code
39       * is not part of our model. This object is just UI code.
40       * The data passed is the value returned
41       * from the user input function when the value updates. OPTIONAL */
42      setData(data,outputElement,admin) {
43      }
44
45      /** This method is used when save is pressed on the coponents save too
46       * if applicable. It retrievees an data from the control, such as if th
47       * an edit table. OPTIONAL */
48      getData(outputElement,admin) {
49      }
50
51      /** This method is called when the output element resizes.
52       * OPTIONAL */
53      onResize(outputElement,admin) {
54      }
55
56      /** This method is called before the window is closed. It should
57       * return apogeeapp.app.ViewMode.CLOSE_OK if it is OK to close
58       * this windows. If this function is omitted, it will be assumed
59       * it is OK to close. An alternate return value is
60       * apogeeapp.app.ViewMode.UNSAVED_DATA. OPTIONAL */

```



```

61     isCloseOk(outputElement,admin) {
62         return apogeeapp.app.ViewMode.CLOSE_OK;
63     }
64
65     /** This method is called when the control is being destroyed.
66      * It allows the user to do any needed cleanup. OPTIONAL. */
67     destroy(outputElement,admin) {
68     }
69
70 }

```

The *admin* object passed as an argument has the following utility functions, to be used in the user defined functions above.

```

1  var admin = {
2      /** Returns an instance of the messenger. */
3      getMessenger();
4
5      /** Puts the component in edit mode, bringing up the save bar. */
6      startEditMode();
7
8      /** Ends edit mode for the component, removing the save bar. */
9      endEditMode();
10 }

```

More Information

For more information, see the programming guide for [Custom Components and the HtmlJsDataDisplay](#).

Custom Component

A *Custom Component* is a component where you can make a custom UI.

The *Custom Component* has the following views:

- Display - This shows the HTML element, which the user populates in the other views below.
- Input Code - This is the formula for the table. The value returned from this function will be the data displayed in the user defined display. For example, if the user makes this display a simple text area, this would typically be the text placed in the text area.
- Input Private - This is the private space associated with the input code above.
- HTML - This is HTML that is placed in the HTML element on its creation. Note that IDs on the elements here should be unique for the entire page.
- CSS - This is CSS that is appended to the page. This can be used to style the element. Note that style classes should be unique for the entire page.
- uiGenerator() - This is a function the user codes which should return a UI generator object. That is an object that contains methods shown [here](#) to give functionality to the display for this component.
- Notes - These are notes the user can add for the component.

Reference the [Custom Components](#) page for a reference of what is in the UI Generator object we use to define our user interface.

Example Workspace

This is a simple example of a custom control.

Click here to open the [Example Workspace](#).

This workspace contains three tables:

- initialValue - This is a plain data table containing some random text to be used by our test component.
- testComponent - This is an example custom component, described below.

- `sentData` - This is a plain data table that will receive data from our test component.

The custom component, *testComponent*, has a text area and a button. The text area is initially populated with data from the table *intial/Value*. The button, when pressed, will copy the data from the text area to the table *sentData*.

Programming the Test Component

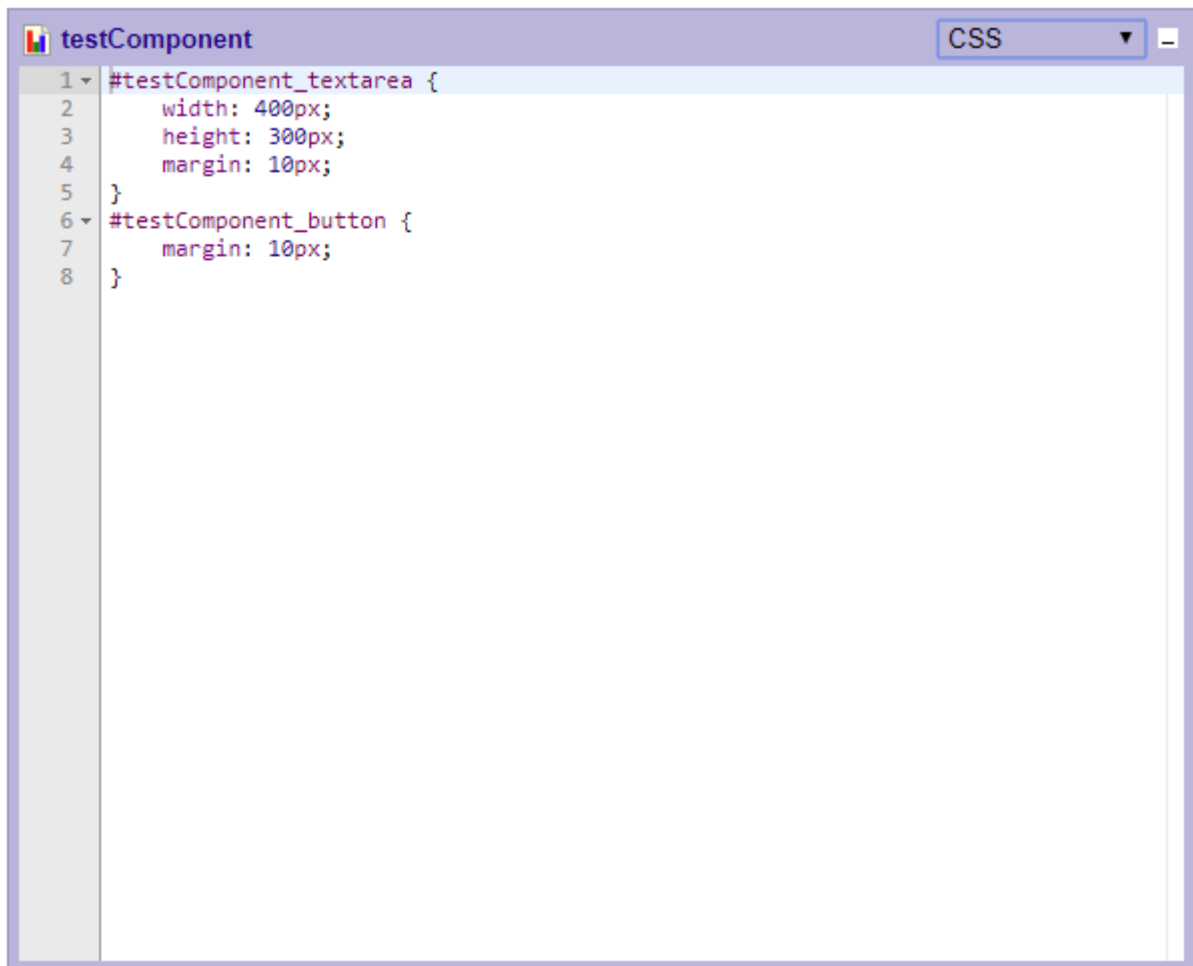
The following screen shots show the code added to the custom component, *testComponent*.

This is the HTML for our component. It includes a text area and a button.



```
testComponent HTML
1 <textarea id="testComponent_textarea"></textarea><br>
2 <button id="testComponent_button">SEND</button>
```

This is the CSS to style our HTML. For simplicity we are just making the text area fixed in size. (Note that is we want to react to size changes of the component, we could have defined an `onResize` method.)



The main code for the example is the UI Generator. This is just an object that has some callback functions in it. It is convenient to define this as a class and then return an instance of it, as we did below.

In the `onload` method, we loaded the text area and the button from the HTML. We also add a click handler to the button. This takes the data in the text area and sends it to the *sentData* table using the messenger, which we can obtain from the admin object passed in.

An important point here is that we can NOT access other table from this code. This is not code in our model. This is just UI code. We can however get data from our model by defining the *setData* function.

The *setData* function receives the output of the "formula", or the function we defined in the *input code* view for this component. The input code DOES have access to the model. We show that code below.

There is one other notable item here in the constructor. We added a call to `__customControlDebugHook()`. This doesn't have any action. It references a function in our static code, in the source file called "debugHook.js". We put it there so we can set a breakpoint in our debugger to debug our code. This is optional.



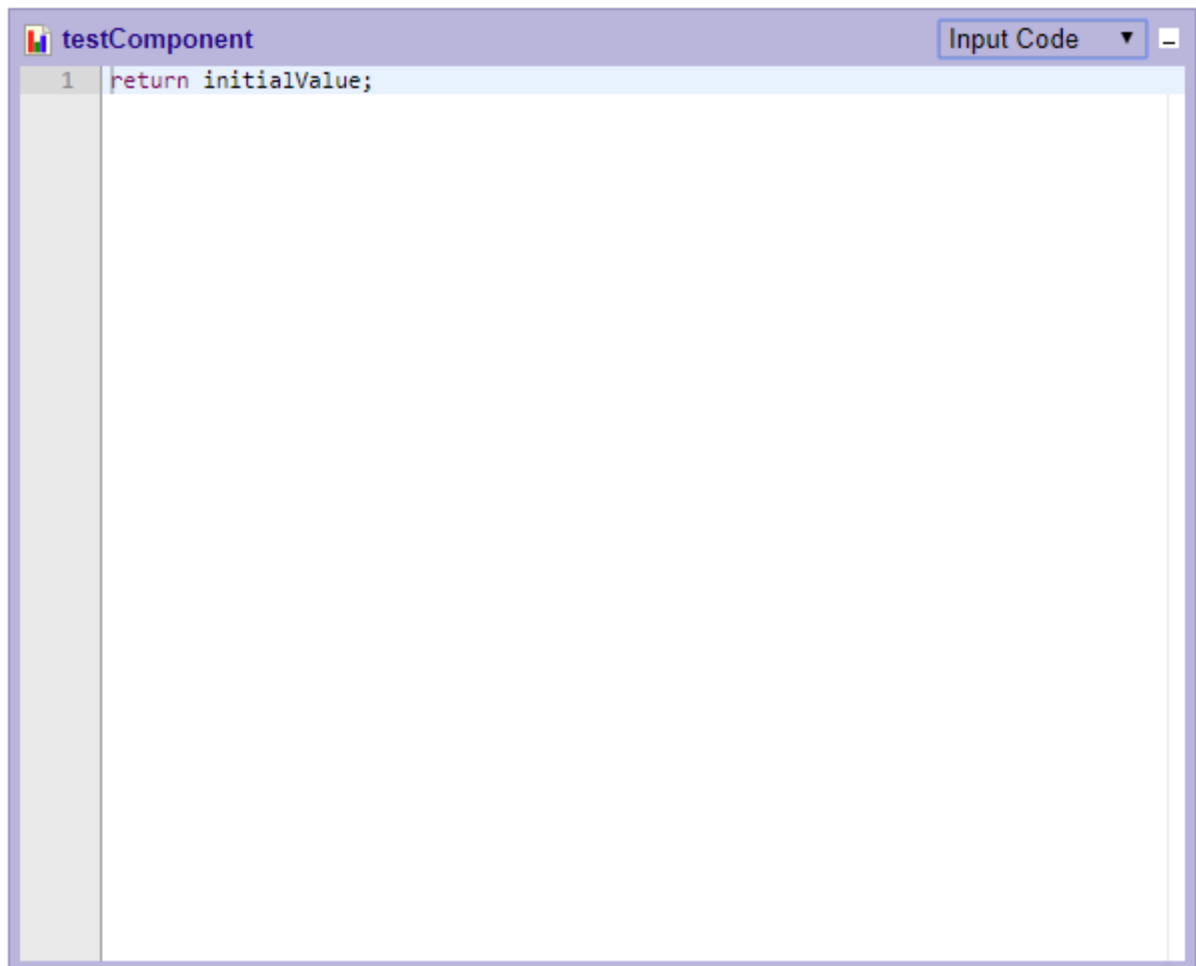
```
testComponent
uiGenerator() ▼

1 var SampleUiGeneratorClass = class {
2
3   constructor() {
4     this.button = null;
5     this.textarea = null;
6
7     //this is added just to help us debug
8     __customControlDebugHook();
9   }
10
11  onLoad(outputElement, admin) {
12    this.button = document.getElementById("testComponent_button");
13    this.textarea = document.getElementById("testComponent_textarea");
14
15    this.button.onclick = () => {
16      var data = this.textarea.value;
17      admin.getMessenger().dataUpdate("sentData",data);
18    }
19  }
20
21  setData(data, outputElement, admin) {
22    if(this.textarea) {
23      this.textarea.value = data;
24    }
25  }
26 }
27
28 return new SampleUiGeneratorClass();
```

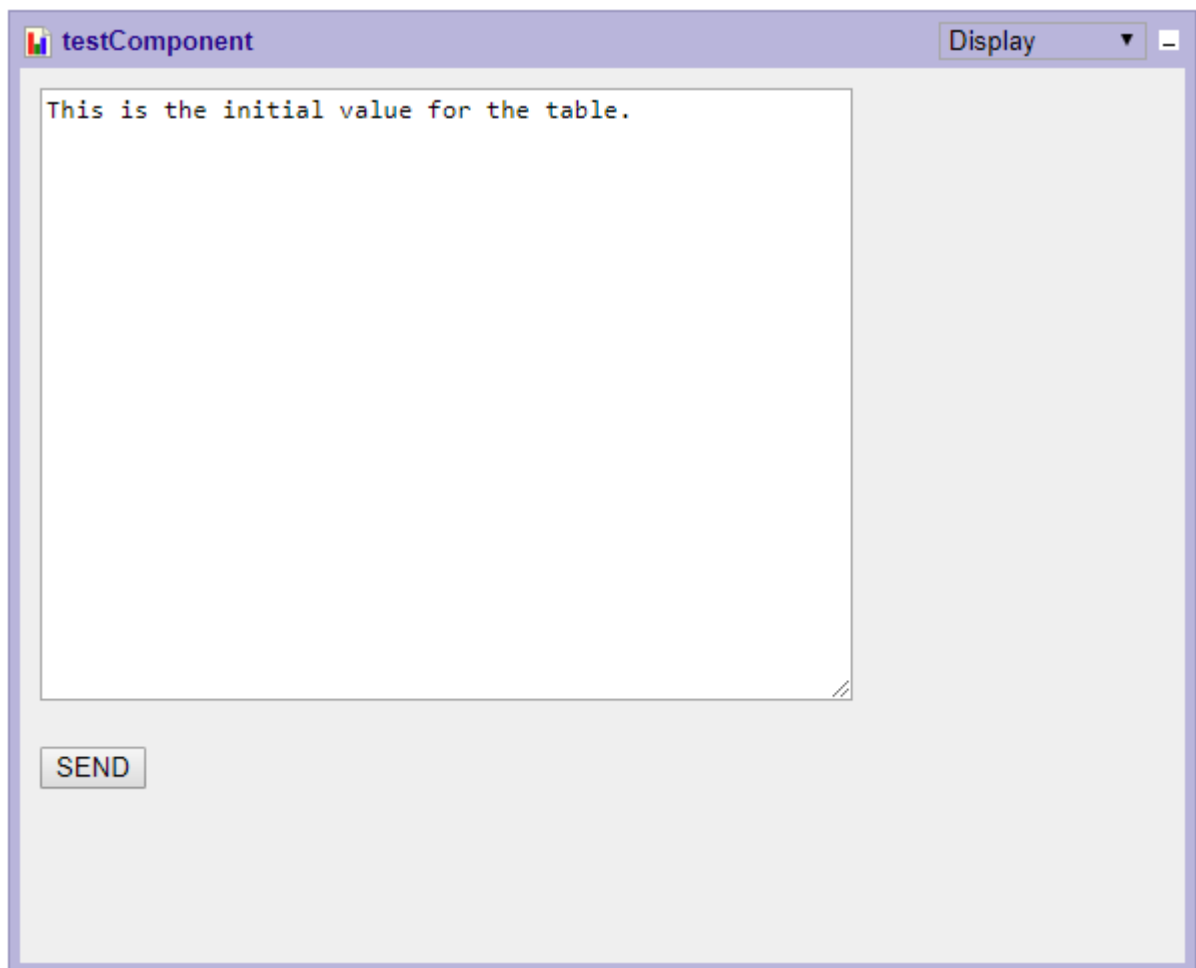
Below is the *input code*. This is equivalent to the formula in a data table. It is a function that returns a value that will be passed into our UI generator code with the *setData* function. This code does have access to our model, as we see here, and it is the only way to pass data from our model to our UI.

In this case we just return the value in *initial/ValueTable*. In our model we will use this to set the value in our test area. For one thing, it sets the initial value of the table.

The input code will execute any time we update the tables it depends on. In this case that means if we change the value in the *initial/Value* table, it will overwrite the data in the text area. (That may not be the behavior we want. But this is just an illustration of how the component works. It can be coded differently.)



Here is our result for the *Display* view. We have the simple for as we described.



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, then the display is only created once and then saved, so there is no need to worry about saving state between the 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.

Custom Data Component

A *Custom Data Component* is very similar to the Custom Component except it stores data created in our UI.

The *Custom Data Component* has the following views. This is the same as *Custom Component* except with one addition, the *Data Value* view.

- Display - This shows the HTML element which the user populates in some of the other views below.
- Data Value - This is where we can view the data stored from this control.
- Input Code - This is the formula for the table. The value returned from this function will be included the data displayed in the user defined display. For example, if the user makes this display a simple text area, this would typically be the text placed in the text area. (Note that unlike the *Custom Component*, this is not the only data passed in to *setData*. See below.)
- Input Private - This is the private space associated with the input code above.
- HTML - This is HTML that is placed in the HTML element on its creation. Note that IDs on the elements here should be unique for the entire page.
- CSS - This is CSS that is appended to the page. This can be used to style the element. Note that style classes should be unique for the entire page.
- *uiGenerator()* - This is a function the user codes which should return a UI generator object. That is an object that contains methods shown [here](#) to give functionality to the display for this component.
- Notes - These are notes the user can add for the component.

Reference the [Custom Components](#) page for a reference of what is in the UI Generator object we use to define our user interface.

Example Workspace

We will modify our *Custom Component* example only slightly to demonstrate the *Custom Data Component*.

Click here to open the [Example Workspace](#).

This workspace contains three tables, shown below.

- `initialValue` - This is a plain data table containing some random text to be used by our test component.
- `testDataComponent` - This is an example custom data component, described below.
- `copyOfSavedData` - This is a plain data table which shows the data saved on `testDataComponent`, illustrating how to access it remotely.

The custom data component, *testDataComponent*, keeps the text area from the previous example, but it removes the button.

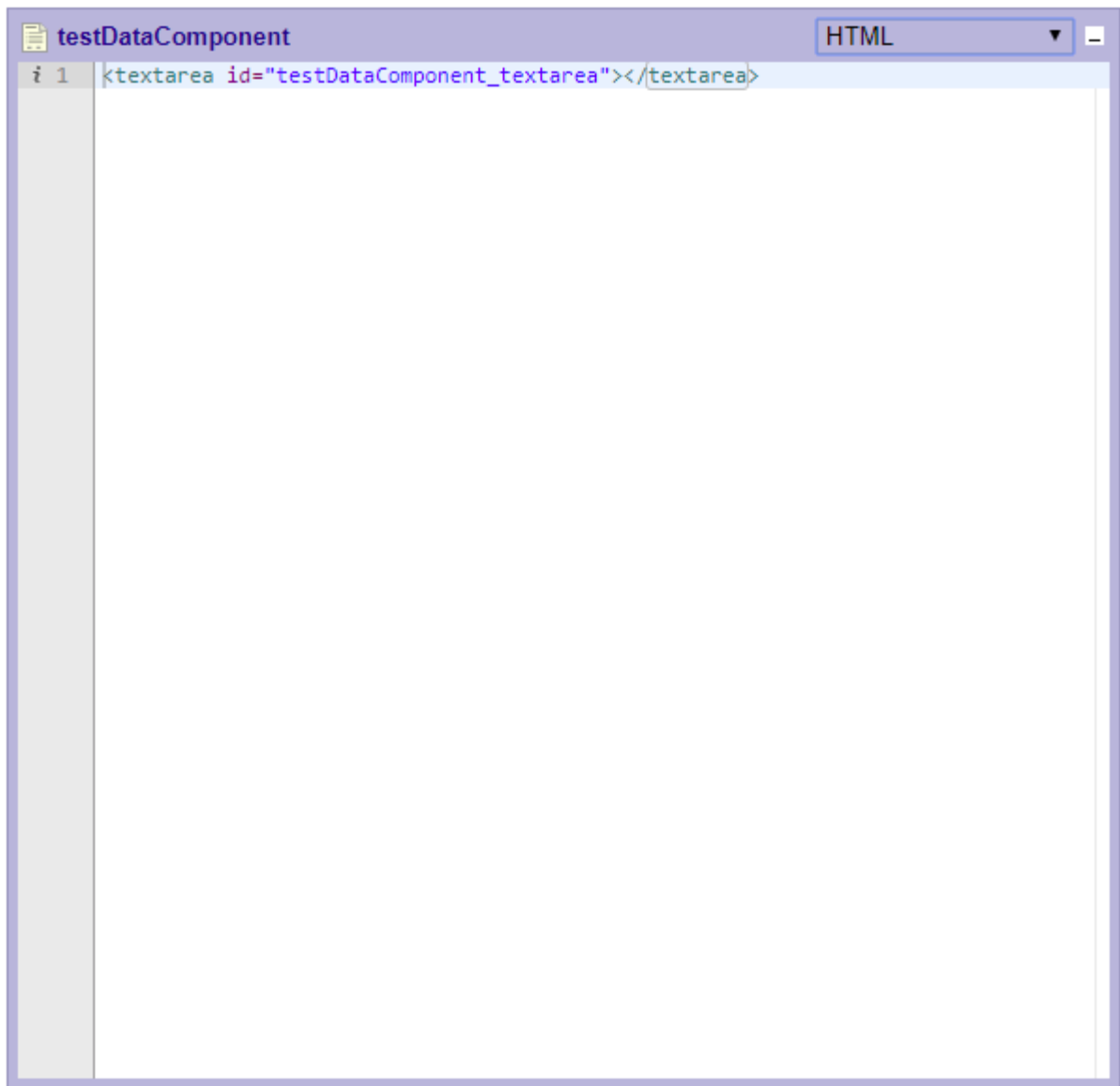
In the custom data component, the save bar can be used instead of adding our own button. When the save button is pressed, the component will automatically save the data, obtained from the `getData` method we define, and stores it. If the cancel button is pressed, the previous data is restored.

To use the save button, we must trigger it. This can be done, for example, when we start to edit our data in the form we create.

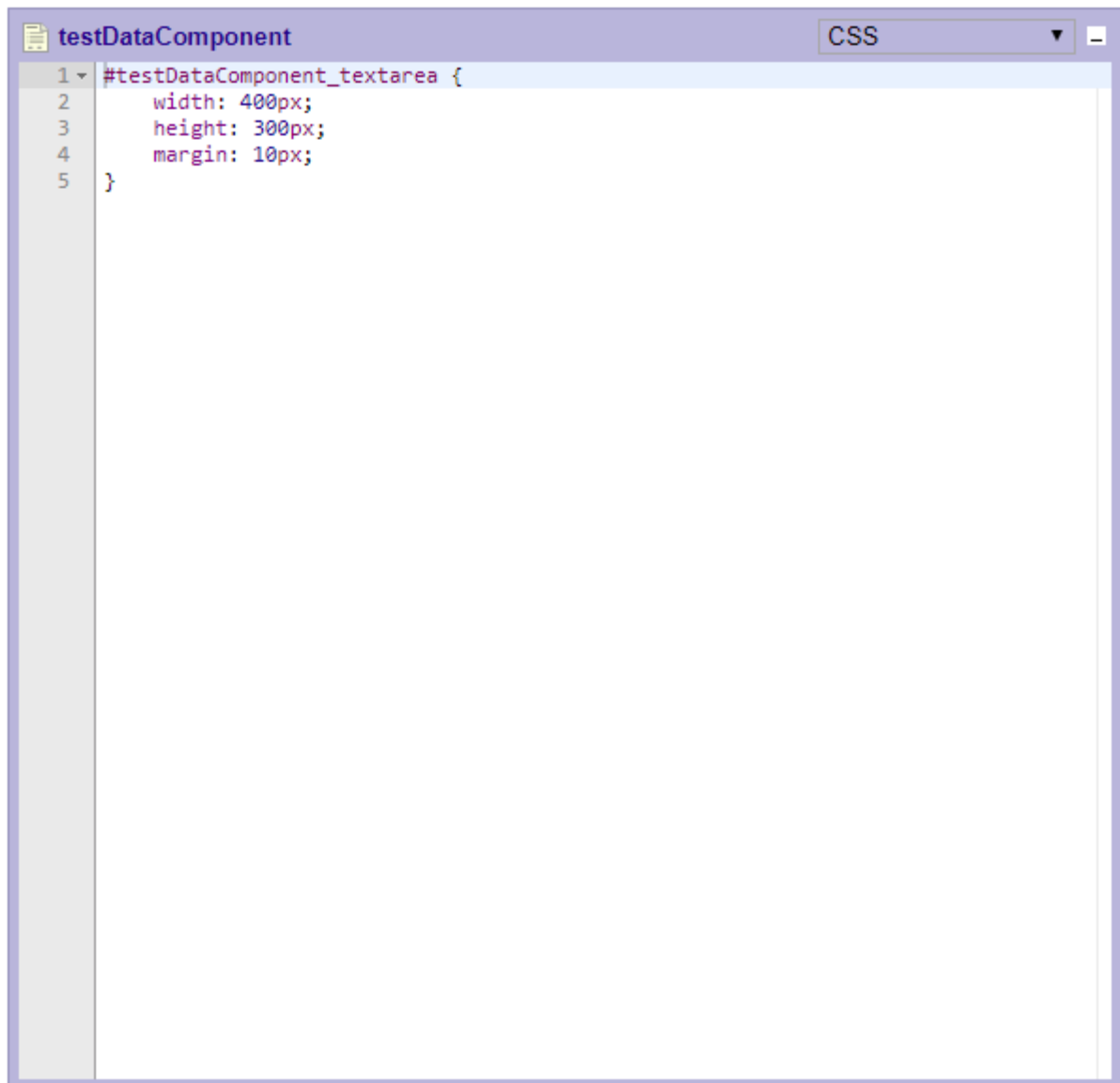
Programming the Test Component

The following screen shots show the code added to the custom component, *testDataComponent*.

This is the HTML for our component. It includes just a text area.



This is the CSS to style our HTML. For simplicity we are just making the text area fixed in size. (Note that if we want to react to size changes of the component, we could have defined an `onResize` method.)



The main code for the example is the UI Generator. This is just an object that has some callback functions in it. It is convenient to define this as a class and then return an instance of it, as we did below.

In the onload method, we loaded the text area from the HTML. We also add an event handler to start edit mode (bringing up the save bar) if we edit the data.

An important point here is that we can NOT access other table from this code. This is not code in our model. This is just UI code. We can however get data from our model by defining the *setData* function.

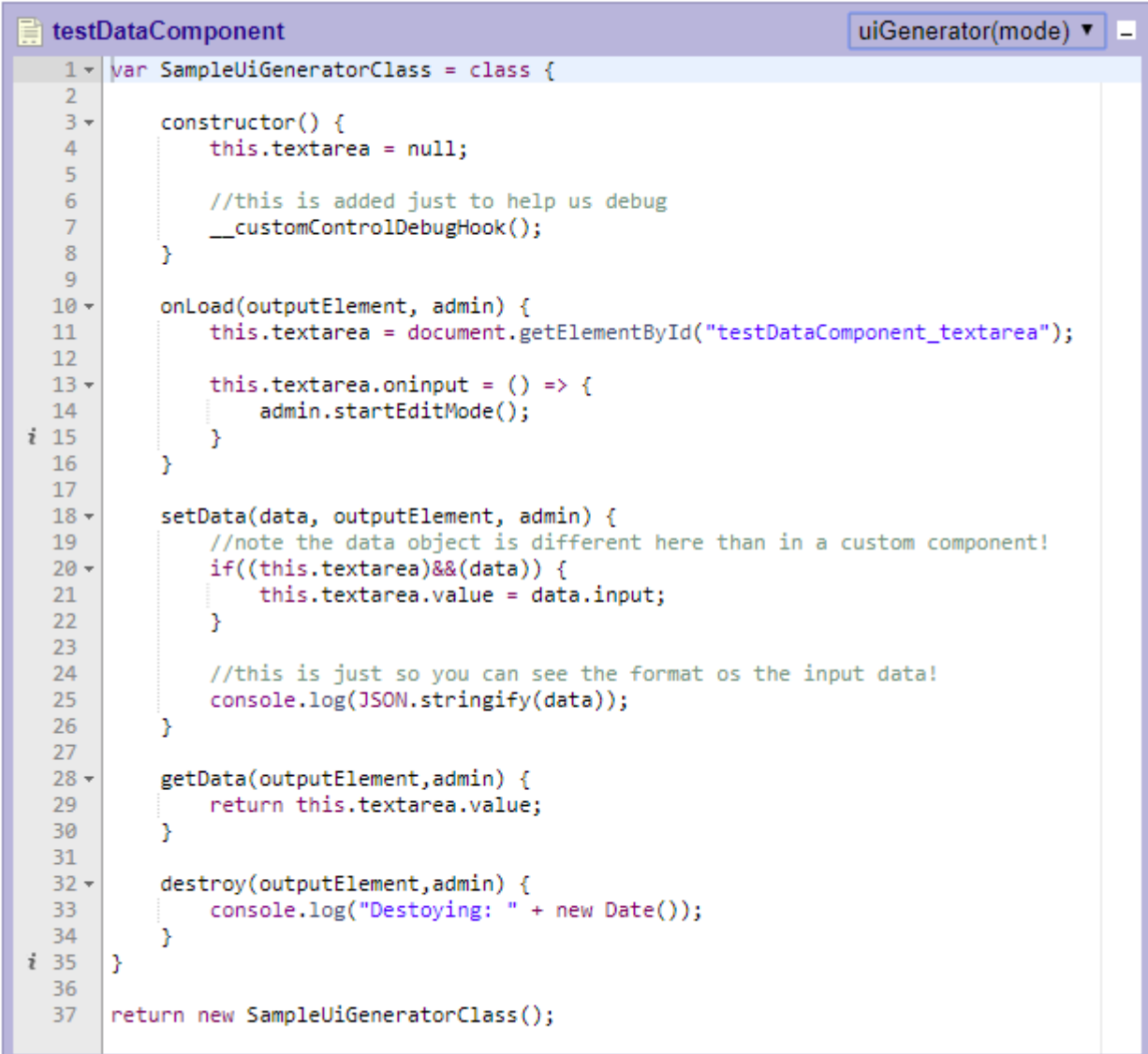
The *setData* function here is slightly different from the Custom Component. In the Custom Component, the argument *data* is the output of the "formula", or the function we defined in

the *input code* view. Here in the Custom Data Component it is different. It is an object containing the result of the input code and also the value of the stored data for this table.

There is a call to the console log to show what the data looks like. You can see the result by viewing the javascript console for your browser.

We have also added a *getData* function, which was not relevant for the Custom Component. This returns the data in the text area. It is used when we press the save button on the *save bar*.

There is one other notable item here in the constructor. We added a call to `__customControlDebugHook()`. This doesn't have any action. It references a function in our static code, in the source file called "debugHook.js". We put it there so we can set a breakpoint in our debugger to debug our code. This is optional.



```
1 var SampleUiGeneratorClass = class {
2
3   constructor() {
4     this.textarea = null;
5
6     //this is added just to help us debug
7     __customControlDebugHook();
8   }
9
10  onLoad(outputElement, admin) {
11    this.textarea = document.getElementById("testDataComponent_textarea");
12
13    this.textarea.oninput = () => {
14      admin.startEditMode();
15    }
16  }
17
18  setData(data, outputElement, admin) {
19    //note the data object is different here than in a custom component!
20    if((this.textarea)&&(data)) {
21      this.textarea.value = data.input;
22    }
23
24    //this is just so you can see the format os the input data!
25    console.log(JSON.stringify(data));
26  }
27
28  getData(outputElement,admin) {
29    return this.textarea.value;
30  }
31
32  destroy(outputElement,admin) {
33    console.log("Destroying: " + new Date());
34  }
35 }
36
37 return new SampleUiGeneratorClass();
```

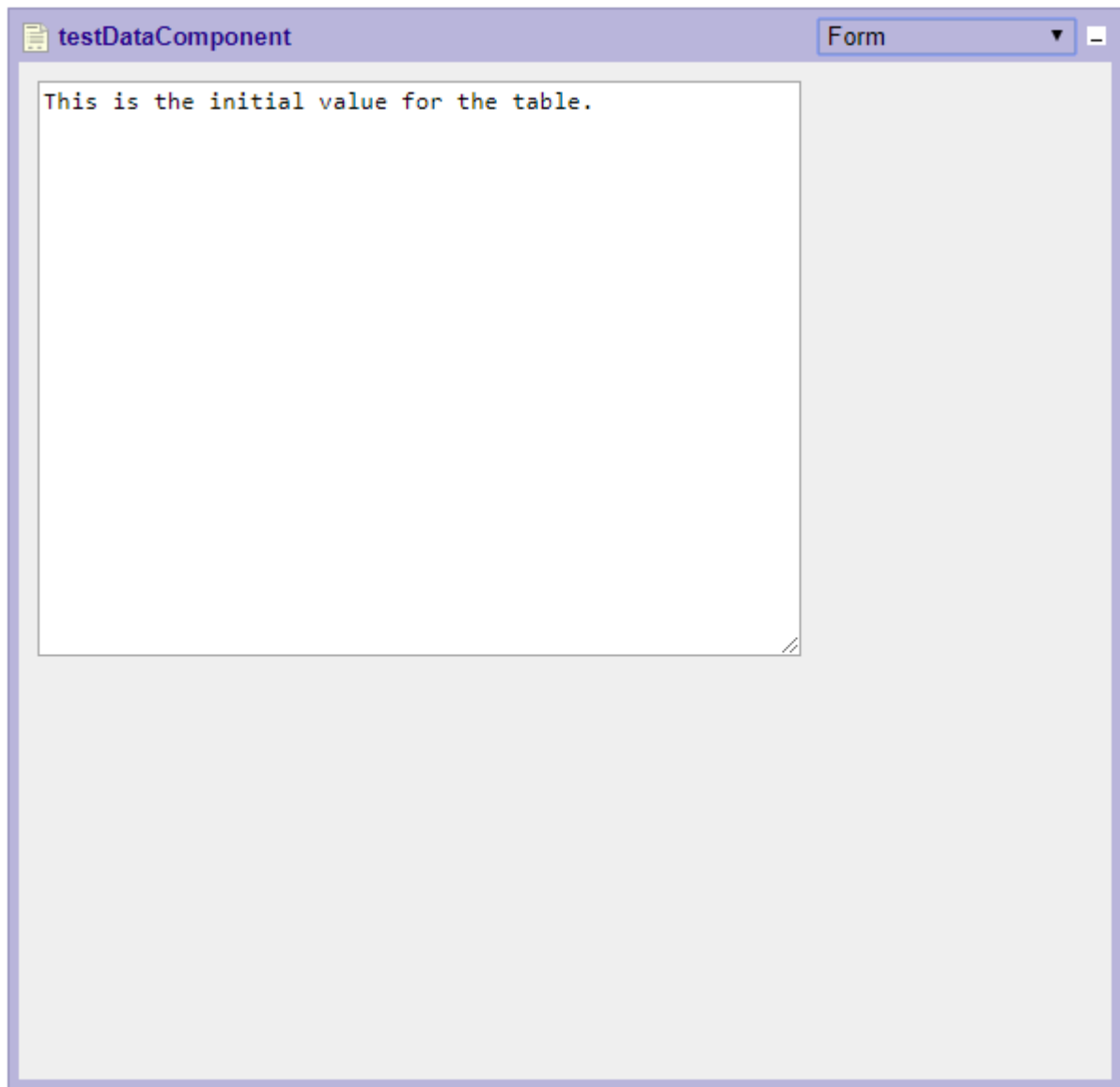
Below is the *input code*. This is equivalent to the formula in a data table. It is a function that returns a value that will be passed into our UI generator code with the *setData* function. Note however that this is slightly different than in the Custom Component, as mentioned above. Here in the *Custom Data Component*, the data argument passed into the *setData* function includes the result of our *input code* and the value that has currently been saved for this component.

In this case we just return the value in the initial value table. In our model we will use this to set the value in our test area. For one thing, it sets the initial value of the table.

The input code will execute any time we update the tables it depends on. In this case that means if we change the value in the *initialValue* table, it will overwrite the data in the text area. (That may not be the behavior we want. But this is just an illustration of how the component works. It can be coded differently.)

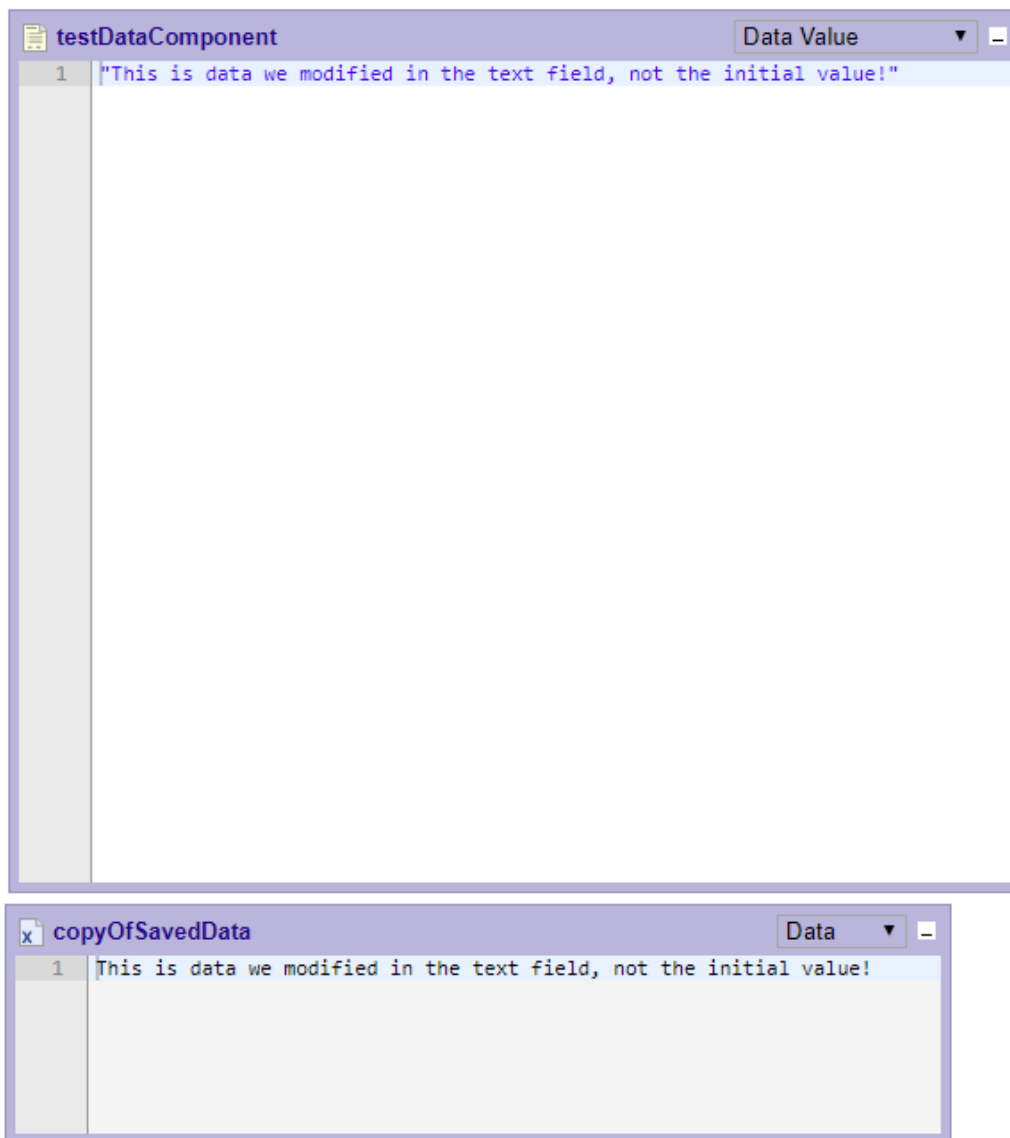


Here is our result for the *Display* view. We have the simple form as we described.



There is one more view of interest to our example, *Data Value*. Below we show our value in this view, after we have edited the text in the text area and pressed save. This is where the data is stored.

Also, this is the data we referenced in our table *copyOfSavedData*.



The following code is in the *Formula* for *copyOfSavedData* to access this value from the *testDataComponent*.

```
return testDataComponent.data;
```

Compound Component

Just like the **Form Data Table**, the custom component is a compound component, meaning it is really a folder containing multiple tables. In this case, it is the following tables:

- input - This is the input data we use for the component. It is the result of the function we defined in the *input code* view.
- data - This is the stored value for the component.

Folder Functions

Roll Your Own Components

Additional Apogee Videos

What is Apogee? An Overview



Apogee Programming Tool 2/8/18

https://www.youtube.com/watch?v=_aBkp-kvc0c

Workspace from video: [Population Example Workspace](#)

Additional Example Workspaces

The Grid and Spreadsheets

String Translation Workspace

Combined Example

OSM GeoJSON Render Workspace