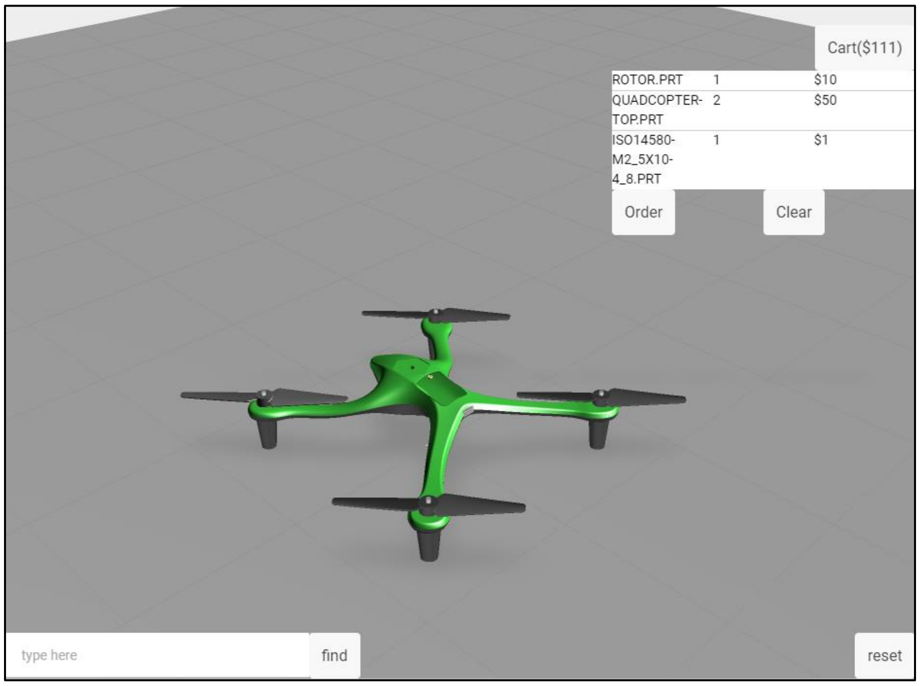
vuforia™ studio

Metadata 301

Adding Pricing Data and a Shopping Cart to a Model

Prerequisites
Completion of the following tutorials:
Metadata 101 – Using Attributes in Creo Illustrate
Metadata 201 – Using JavaScript to Highlight Parts and Create Ionic Popups
Metadata 202 – Using JavaScript to Find Parts



ROTOR.PRT	1	$10
QUADCOPTER-TOP.PRT	2	$50
ISO14580-M2_5X10-4_8.PRT	1	$1

Intro

It’s not uncommon for parts to break during servicing. When parts break, , it can be a lengthy process to order a new part. For example, human error can lead to the wrong part being ordered. Vuforia Studio can help you avoid that lengthy process by providing the ability to not only display part data, but also connect to an ordering system and order parts directly from an AR experience. This project will teach you how to select parts, add them to a cart, and then order them from Vuforia View.

The following topics will be covered in this project.

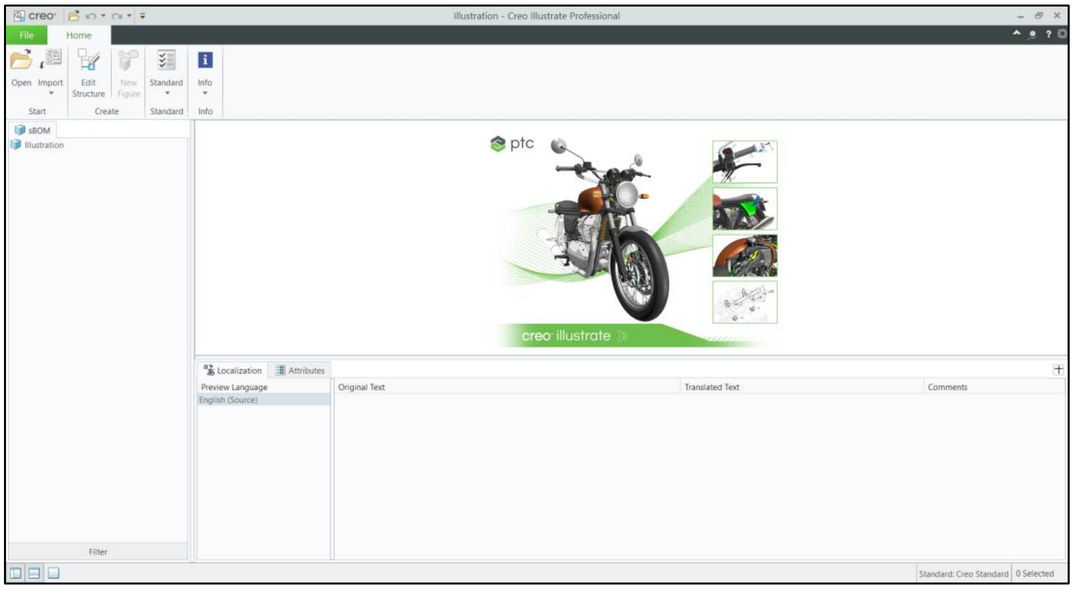
There are also four appendices at the end of the document for the completed code of this project.

All important notes and UI areas are **Bold**.
All non-code text to be typed is *italicized.*
All code follows `this convention`
All code comments follow `this convention`

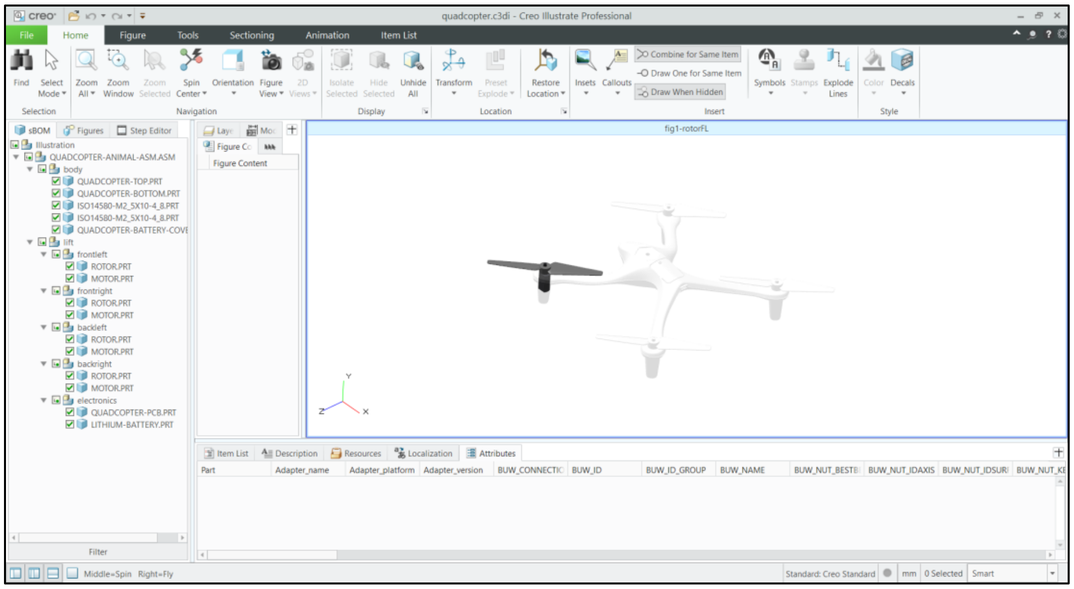
301.1 Add Attributes in Bulk

In **Metadata 101** you added attributes for **partNumber** and **illustration** to the quadcopter model. Those attribute values were populated one by one, but this section will show you how to add the attributes in bulk, which can be much more convenient. You will add pricing data to the models as an attribute that will then be used to create a total when you add items to a cart.

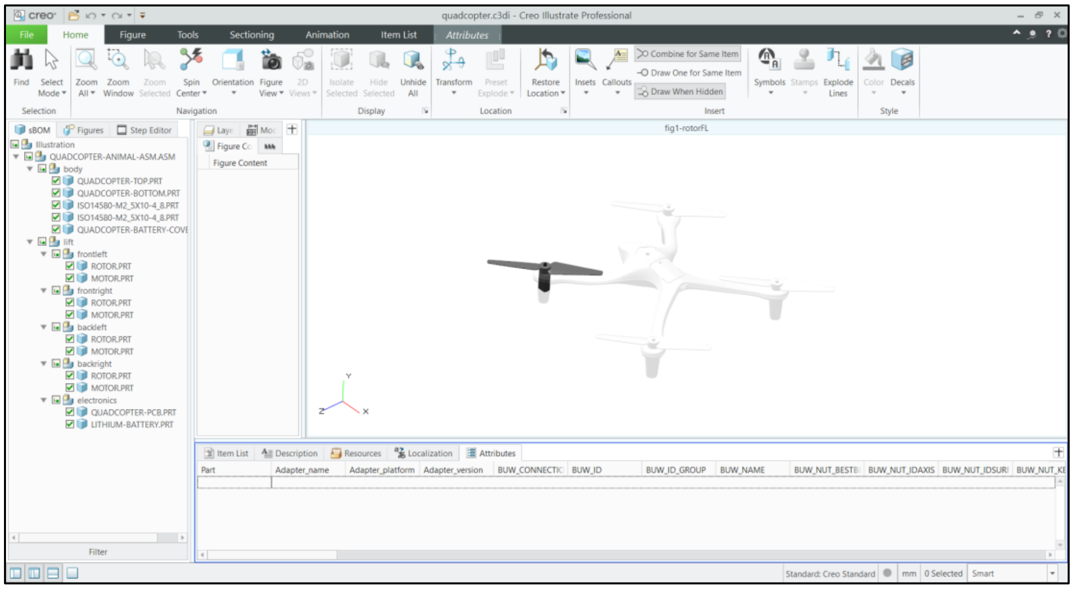
1. Open Creo Illustrate.



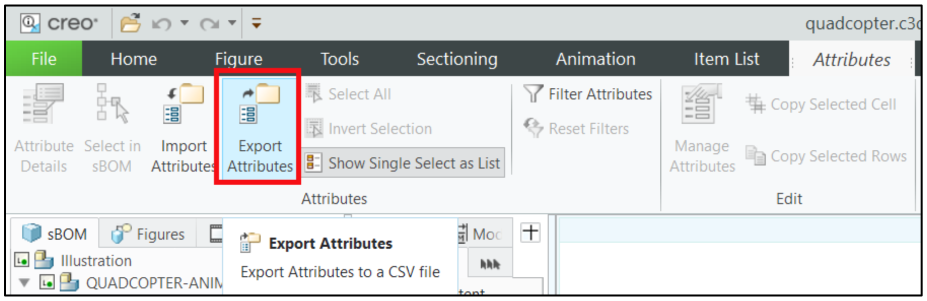
2. From the **File** tab, click **Open** and select the quadcopter.c3di file that was created in **Metadata 101**. If you used the quadcopter101.c3di file that was provided, then open that file.



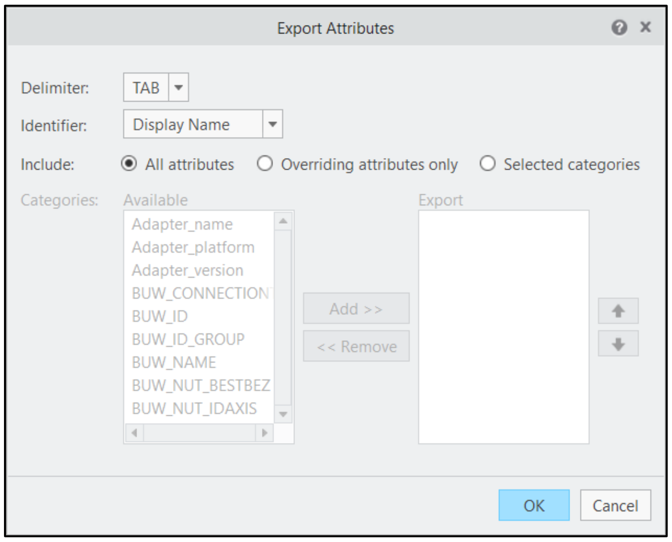
3. Open the Lower Data Panel if it isn’t already open. Open the **Attributes** tab to display the attributes for the selected part.



4. When the **Attributes** tab is opened in the Lower Data Panel, another **Attributes** tab opens in the Ribbon. In the **Attributes** tab at the top of your screen, select **Export Attributes** to prepare the attributes for export. In order to edit the attributes for the model in bulk, they must be exported as a .csv file that can be opened in Excel or a similar application.



a. In the **Export Attributes** window, select a **Delimiter** and **Identifier** from their respective dropdowns. A delimiter is the character that separates the values in the list of attributes. In this example, **TAB** is selected as the delimiter. The **Identifier** is used to determine which attribute is used to identify each part. Select **Display Name** so that each part is designated by the name of its .prt file, as it is displayed in the **sBOM**.

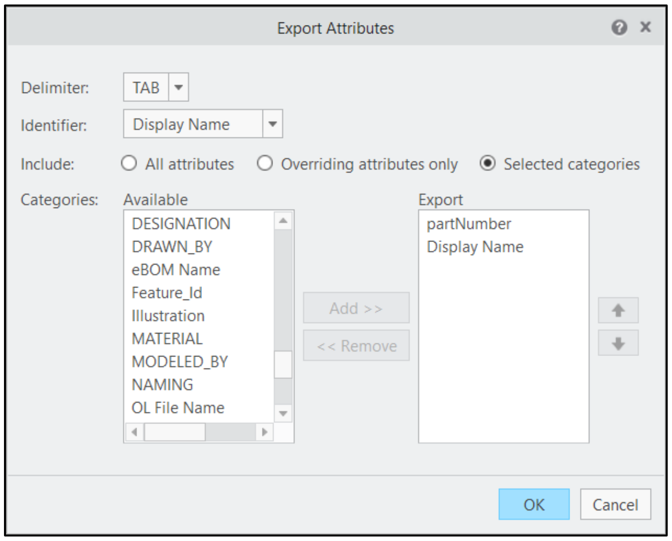

Export Attributes

Delimiter: TAB

Identifier: Display Name

Include: [✓ All attributes] Overriding attributes only Selected categories

b. You must add at least two categories of attributes to the **Export** list to make the file readable. For readability sake, we will only add two attributes, as leaving **All attributes** checked would produce an exported .csv file with as many columns as there are attributes. Select the **Selected categories** option next to **Include:** select the **partNumber** attribute in the **Available** list on the left and select **Add >>** to add that attribute to the **Export** list. You will see that since **Display Name** was chosen as the identifier that it was automatically added to the **Export** list.


Export Attributes

Delimiter: TAB

Identifier: Display Name

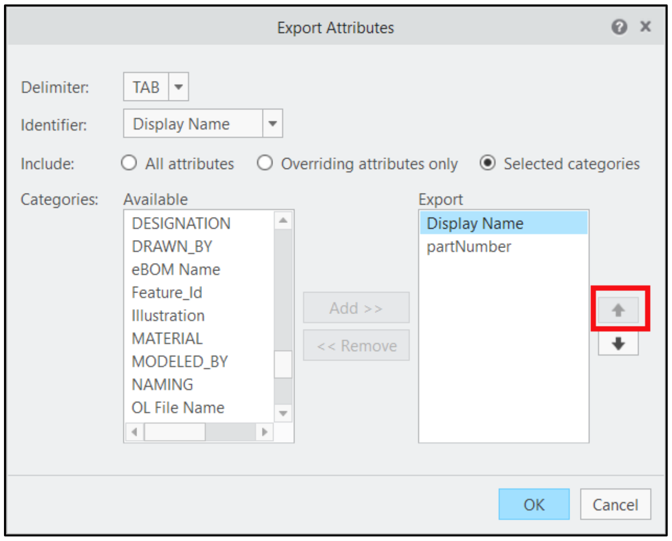
Include: ○ All attributes ○ Overriding attributes only [✓ Selected categories]

Categories:

Available: DESIGNATION, DRAWN_BY, eBOM Name, Feature_Id, Illustration, MATERIAL, MODELED_BY, NAMING, OL File Name

Export: partNumber, Display Name

c. Select **Display Name** in the **Export** list and click the up arrow to move it above **partNumber**.


Export Attributes

Delimiter: TAB

Identifier: Display Name

Include: ○ All attributes ○ Overriding attributes only [✓ Selected categories]

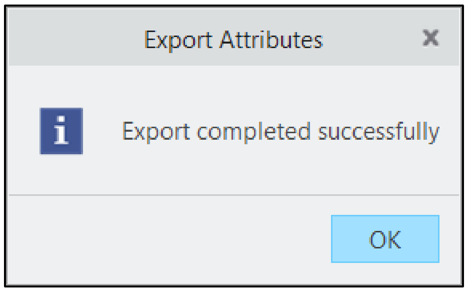
Categories:

Available: DESIGNATION, DRAWN_BY, eBOM Name, Feature_Id, Illustration, MATERIAL, MODELED_BY, NAMING, OL File Name

Export: Display Name, partNumber

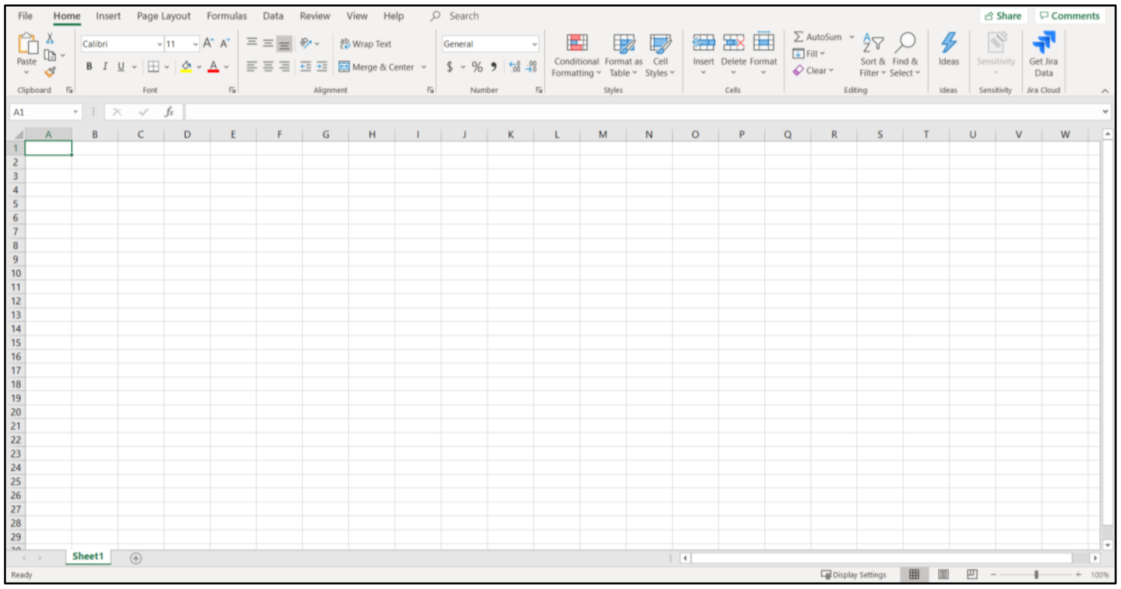
d. Click **OK**, name the file *quadcopter*, and click **Export** to create a .csv file.

e. The below message is displayed when the .csv file has been created successfully.

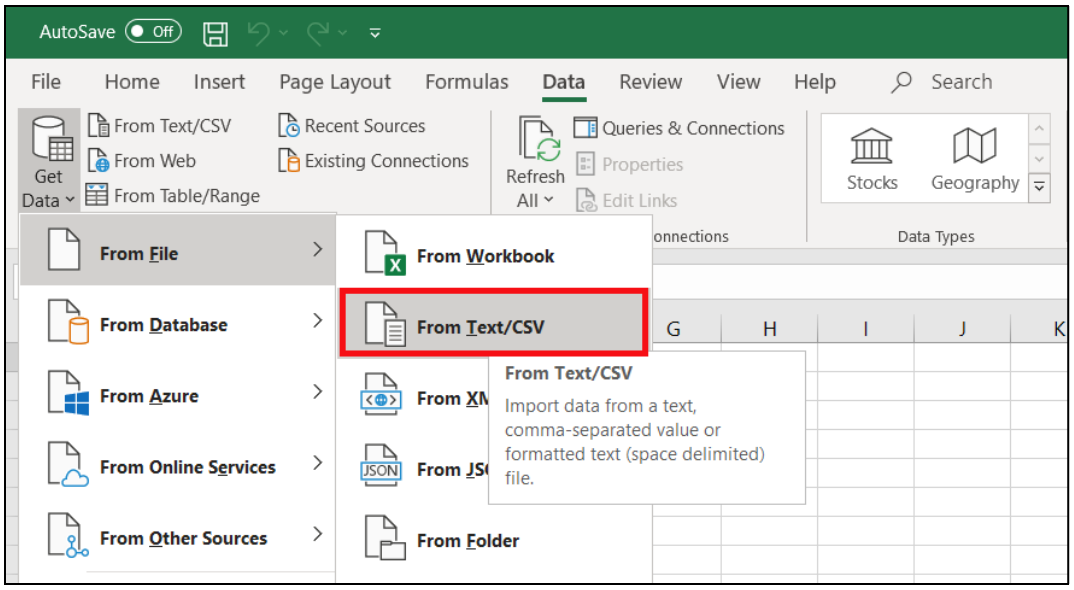

Export Attributes

Export completed successfully

5. Open Microsoft Excel or a similar application. The next few steps will show you how to bulk edit in Excel.
   a. When Excel opens, select **Blank workbook**..

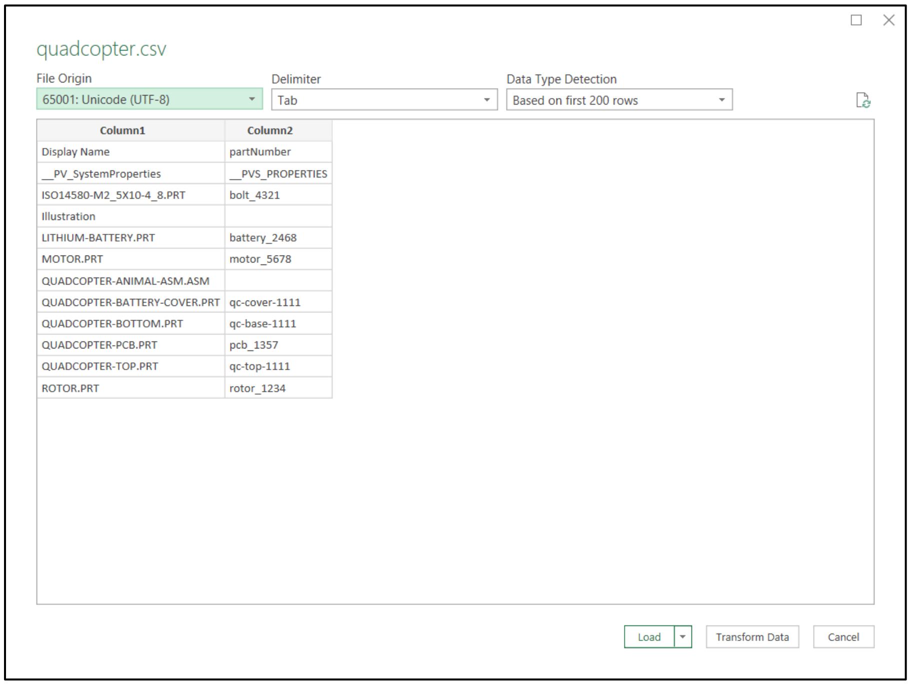


   b. Navigate to the **Data** tab and click **Get Data > From File > From Text/CSV**. This opens a **File Explorer** window.



From Text/CSV
Import data from a text, comma-separated value or formatted text (space delimited) file.

   c. Navigate to and select the quadcopter.csv file from its saved location and then click **Import**.
   d. When the window below appears, click **Load** to load the two data columns into the Excel sheet.



quadcopter.csv

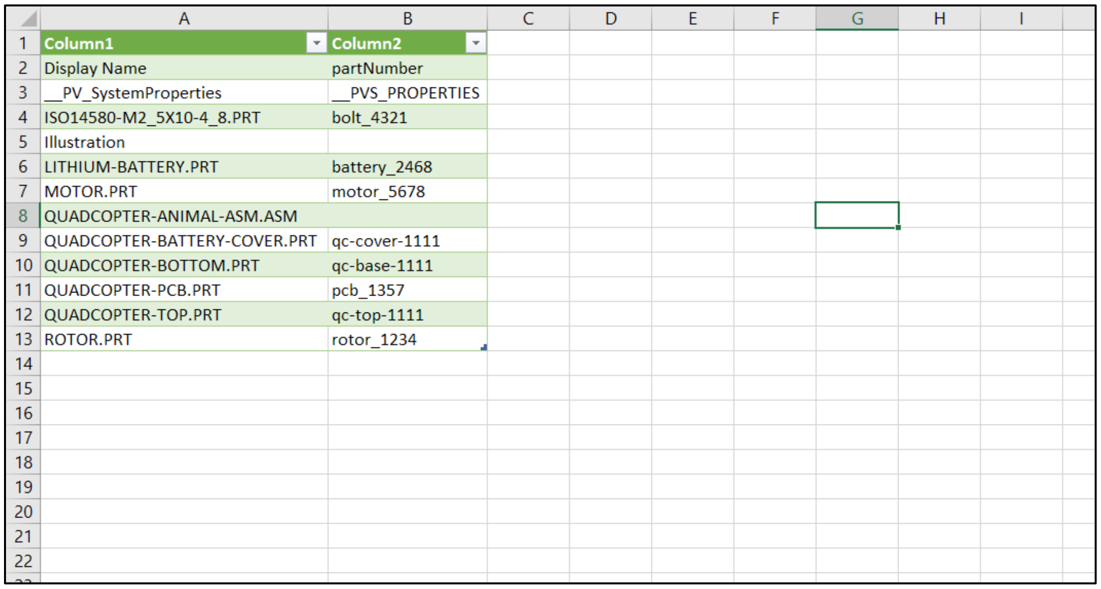
File Origin: 65001: Unicode (UTF-8)

Delimiter: Tab

Data Type Detection: Based on first 200 rows

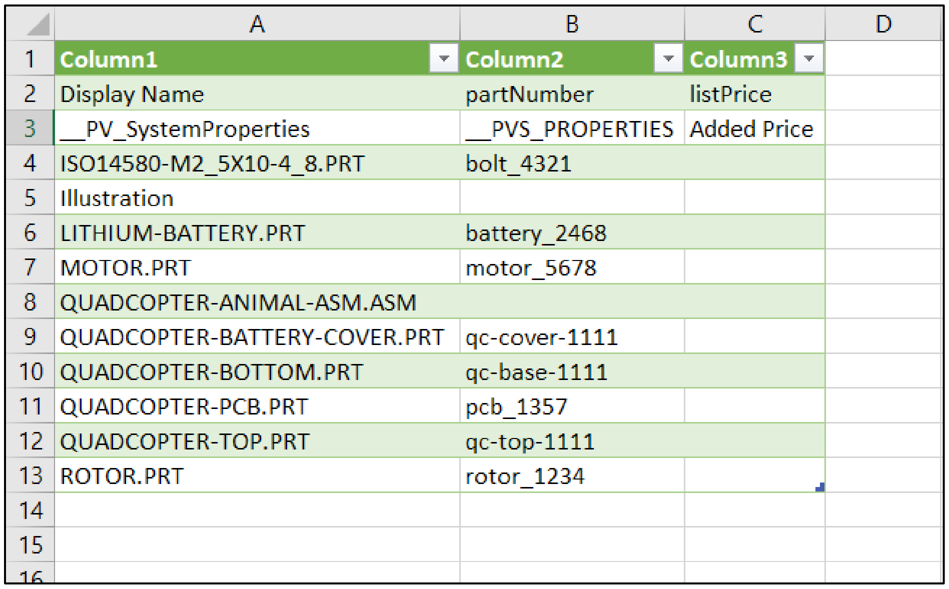
Column1	Column2
Display Name	partNumber
__PV_SystemProperties	__PVS_PROPERTIES
ISO14580-M2_5X10-4_8.PRT	bolt_4321
Illustration	
LITHIUM-BATTERY.PRT	battery_2468
MOTOR.PRT	motor_5678
QUADCOPTER-ANIMAL-ASM.ASM	
QUADCOPTER-BATTERY-COVER.PRT	qc-cover-1111
QUADCOPTER-BOTTOM.PRT	qc-base-1111
QUADCOPTER-PCB.PRT	pcb_1357
QUADCOPTER-TOP.PRT	qc-top-1111
ROTOR.PRT	rotor_1234

e. If your workbook looks similar to the image below, then it has been imported correctly. If not, repeat the steps prior to this.



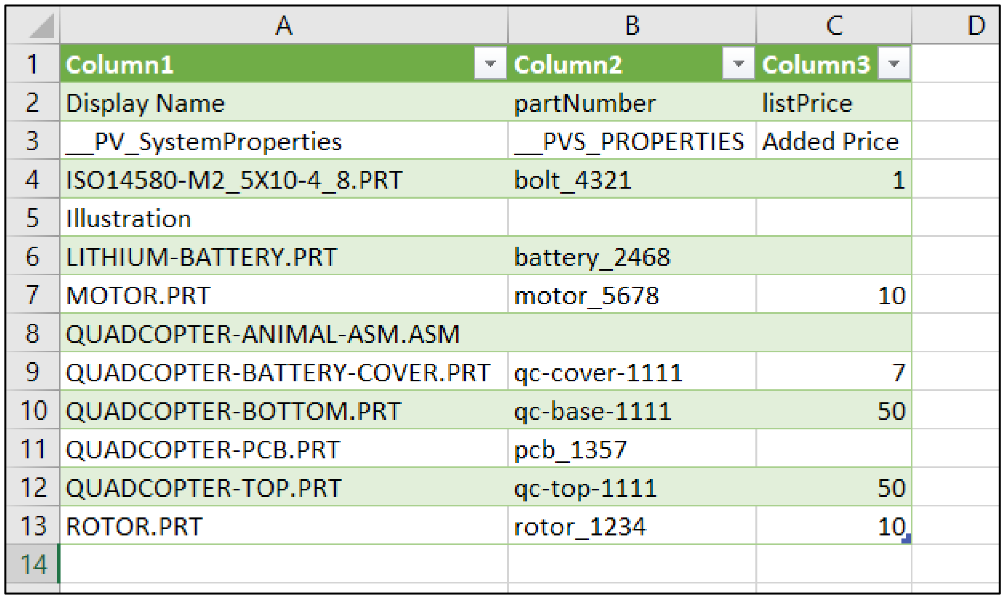
	A	B
1	Column1	Column2
2	Display Name	partNumber
3	__PV_SystemProperties	__PVS_PROPERTIES
4	ISO14580-M2_5X10-4_8.PRT	bolt_4321
5	Illustration	
6	LITHIUM-BATTERY.PRT	battery_2468
7	MOTOR.PRT	motor_5678
8	QUADCOPTER-ANIMAL-ASM.ASM	
9	QUADCOPTER-BATTERY-COVER.PRT	qc-cover-1111
10	QUADCOPTER-BOTTOM.PRT	qc-base-1111
11	QUADCOPTER-PCB.PRT	pcb_1357
12	QUADCOPTER-TOP.PRT	qc-top-1111
13	ROTOR.PRT	rotor_1234

f. In cell **C2**, type *listPrice*. This creates a new attribute name when imported back into Illustrate. In cell **C3**, type *Added Price*. This will be the new attribute category.



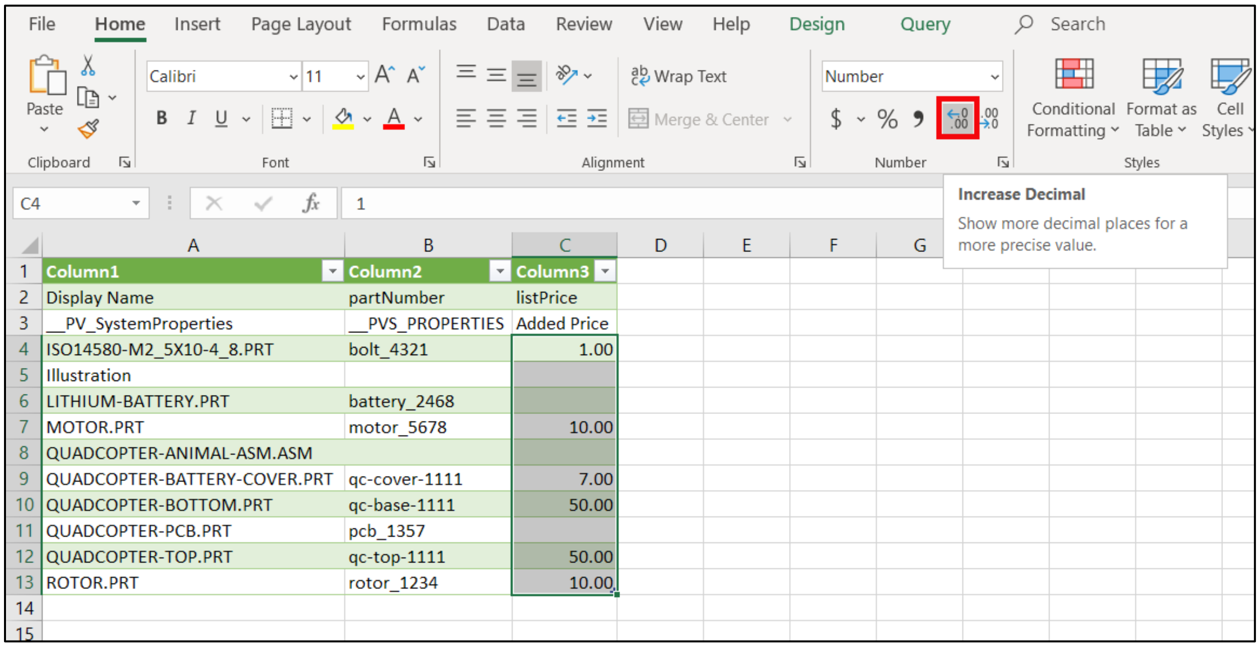
	A	B	C	D
1	Column1	Column2	Column3	
2	Display Name	partNumber	listPrice	
3	__PV_SystemProperties	__PVS_PROPERTIES	Added Price	
4	ISO14580-M2_5X10-4_8.PRT	bolt_4321		
5	Illustration			
6	LITHIUM-BATTERY.PRT	battery_2468		
7	MOTOR.PRT	motor_5678		
8	QUADCOPTER-ANIMAL-ASM.ASM			
9	QUADCOPTER-BATTERY-COVER.PRT	qc-cover-1111		
10	QUADCOPTER-BOTTOM.PRT	qc-base-1111		
11	QUADCOPTER-PCB.PRT	pcb_1357		
12	QUADCOPTER-TOP.PRT	qc-top-1111		
13	ROTOR.PRT	rotor_1234		
14				
15				
16				

g. Add the prices displayed below for the given parts. Illustration and QUADCOPTER-ANIMAL-ASM.ASM do not get prices because they are the assembly portion of the model in the model tree, not actual parts. The electronics combined part is also not listed in this because it is two combined parts, so the pricing information will need to be added manually. Because of that, LITHIUM-BATTERY.PRT and QUADCOPTER-PCB.PRT will not have prices listed.



	A	B	C	D
1	Column1	Column2	Column3	
2	Display Name	partNumber	listPrice	
3	__PV_SystemProperties	__PVS_PROPERTIES	Added Price	
4	ISO14580-M2_5X10-4_8.PRT	bolt_4321	1	
5	Illustration			
6	LITHIUM-BATTERY.PRT	battery_2468		
7	MOTOR.PRT	motor_5678	10	
8	QUADCOPTER-ANIMAL-ASM.ASM			
9	QUADCOPTER-BATTERY-COVER.PRT	qc-cover-1111	7	
10	QUADCOPTER-BOTTOM.PRT	qc-base-1111	50	
11	QUADCOPTER-PCB.PRT	pcb_1357		
12	QUADCOPTER-TOP.PRT	qc-top-1111	50	
13	ROTOR.PRT	rotor_1234	10	
14				

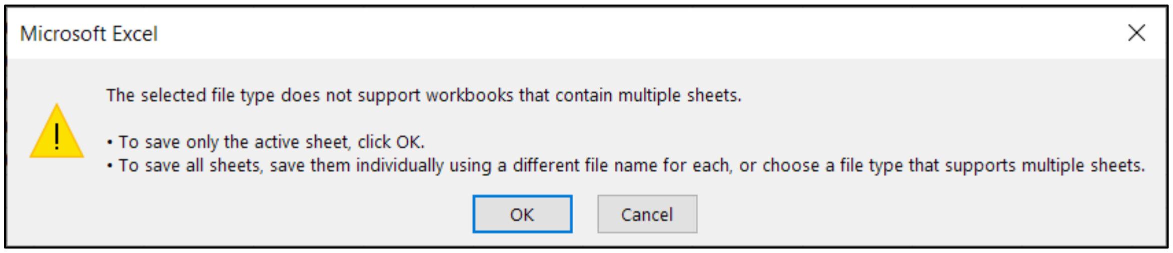
h. Select all **listPrice** values and use the **Increase Decimal** button to add two decimal points to the end of the values. This will ensure that they correspond with how monetary values are written out.



	A	B	C
1	Column1	Column2	Column3
2	Display Name	partNumber	listPrice
3	__PV_SystemProperties	__PVS_PROPERTIES	Added Price
4	ISO14580-M2_5X10-4_8.PRT	bolt_4321	1.00
5	Illustration		
6	LITHIUM-BATTERY.PRT	battery_2468	
7	MOTOR.PRT	motor_5678	10.00
8	QUADCOPTER-ANIMAL-ASM.ASM		
9	QUADCOPTER-BATTERY-COVER.PRT	qc-cover-1111	7.00
10	QUADCOPTER-BOTTOM.PRT	qc-base-1111	50.00
11	QUADCOPTER-PCB.PRT	pcb_1357	
12	QUADCOPTER-TOP.PRT	qc-top-1111	50.00
13	ROTOR.PRT	rotor_1234	10.00

i. Click **File** > **Save As**. Save the file to a desired location as a .csv. Note: You must do a **Save As** and not the *ctrl+S* method of saving to ensure that you save the file as a .csv file. The completed quadcopter_complete.csv file with the completed file will be available in the GitHub folder for this project.

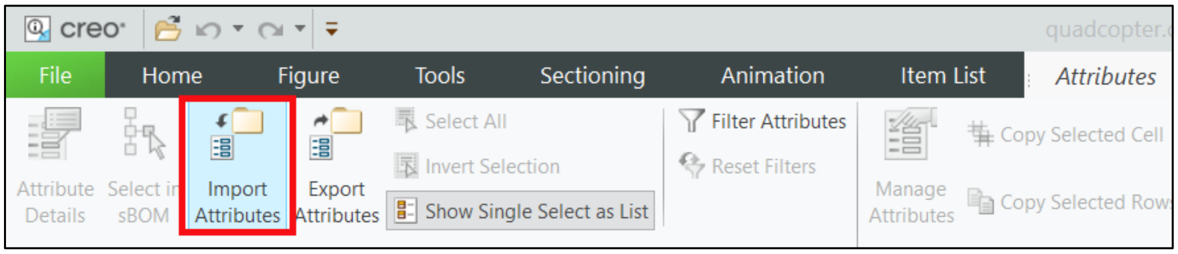
j. In the pop-up that appears, click **OK** since only the active sheet needs to be saved.


Microsoft Excel

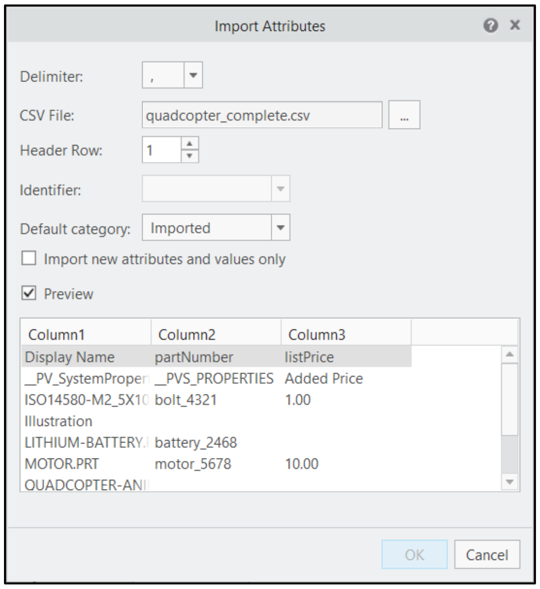
The selected file type does not support workbooks that contain multiple sheets.

• To save only the active sheet, click OK.
• To save all sheets, save them individually using a different file name for each, or choose a file type that supports multiple sheets.

6. Now that this file has been created, navigate back to Creo Illustrate. From the **Attributes** tab, select **Import Attributes**. This is used to bring the **listPrice** attribute that was just created into the quadcopter model.



a. In the **Import Attributes** window click **…** next to **CSV File:.** Navigate to the file you just created, and click **Import**. It should look like the image below.



Import Attributes

Delimiter: ,

CSV File: quadcopter_complete.csv

Header Row: 1

Identifier:

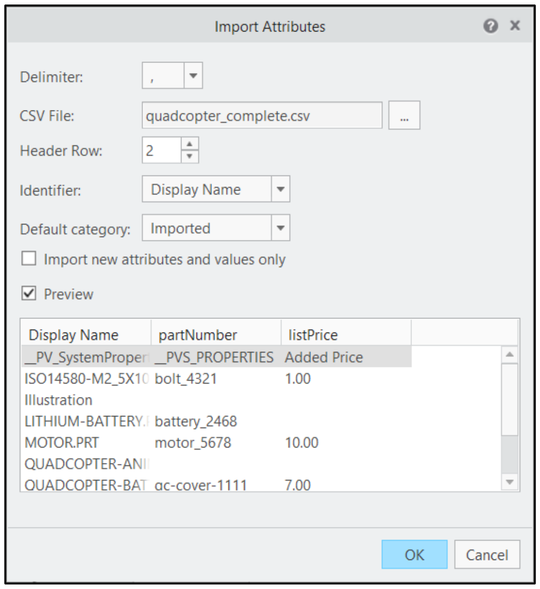
Default category: Imported

☐ Import new attributes and values only

☑ Preview

Column1	Column2	Column3
Display Name	partNumber	listPrice
__PV_SystemProper	__PVS_PROPERTIES	Added Price
ISO14580-M2_5X10	bolt_4321	1.00
Illustration		
LITHIUM-BATTERY.	battery_2468	
MOTOR.PRT	motor_5678	10.00
QUADCOPTER-ANI		

b. Once the data has been imported, you'll notice that the **Column** headers from Excel are still in the import, even though they are not attributes. To resolve this issue, change the value in the box next to **Header Row:** to *2* to set the second row of values as the header row, which will be the name of the attribute.



Import Attributes

Delimiter: ,

CSV File: quadcopter_complete.csv

Header Row: 2

Identifier: Display Name

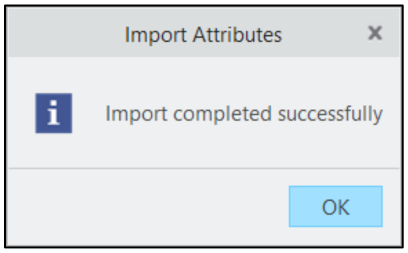
Default category: Imported

☐ Import new attributes and values only

☑ Preview

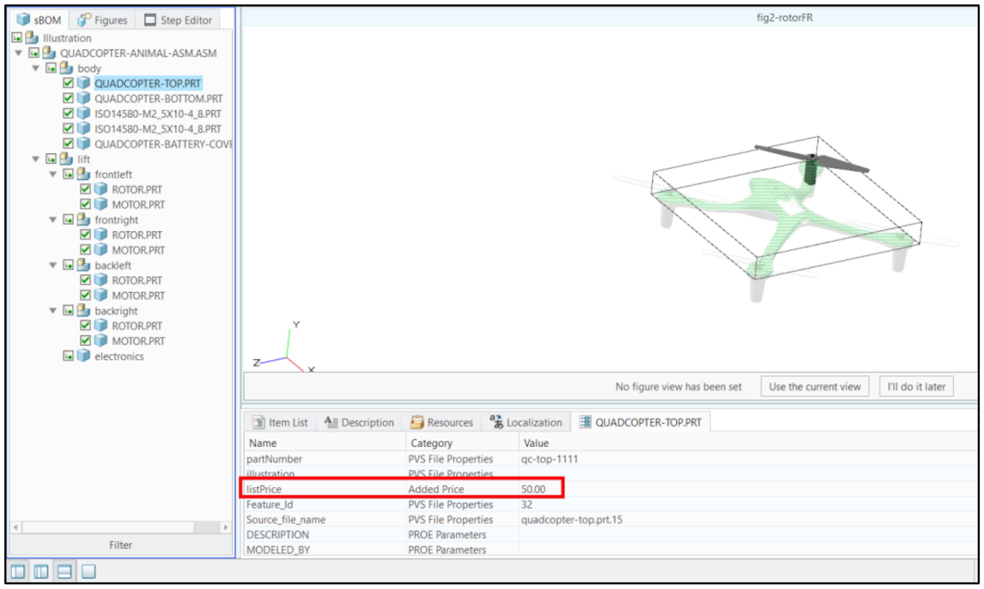
Display Name	partNumber	listPrice
__PV_SystemProper	__PVS_PROPERTIES	Added Price
ISO14580-M2_5X10	bolt_4321	1.00
Illustration		
LITHIUM-BATTERY.	battery_2468	
MOTOR.PRT	motor_5678	10.00
QUADCOPTER-ANI		
QUADCOPTER-BAT	qc-cover-1111	7.00

c. Click **OK** on the success window.


Import Attributes

Import completed successfully

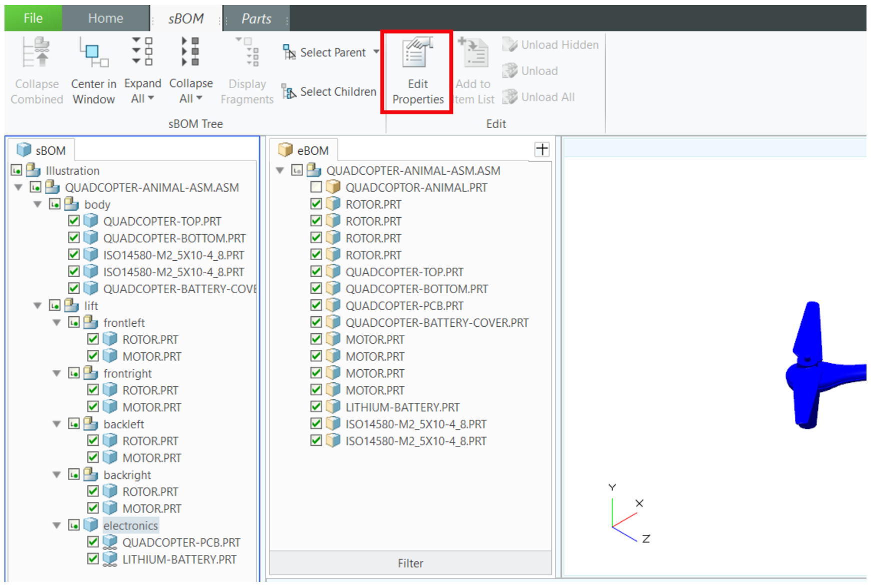
7. Click the parts inside the **sBOM**. The **Attributes** tab in the Lower Data Pane should now have the **listPrice** attribute for each part.



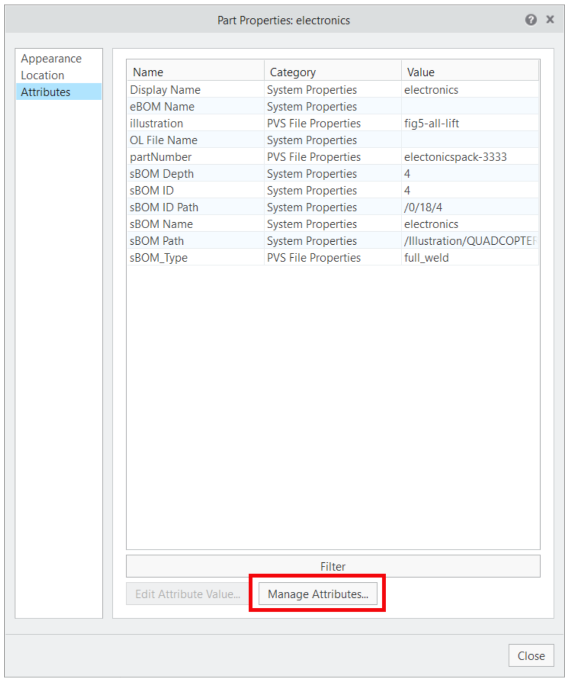
No figure view has been set

Name	Category	Value
partNumber	PVS File Properties	qc-top-1111
illustration	PVS File Properties	
listPrice	Added Price	50.00
Feature_Id	PVS File Properties	32
Source_file_name	PVS File Properties	quadcopter-top.prt.15
DESCRIPTION	PROE Parameters	
MODELED_BY	PROE Parameters	

8. As mentioned before, the **electronics** combined part needs to have its price attribute created manually.
   a. Click the **Edit Structure** button to open the structure editor again. Repeat the same process you used in **Metadata 101** to make the model appear in the **Structure Edit** window.
   b. Like before, select **electronics** in the **sBOM** panel and then select **Edit Properties** in the **sBOM** tab in the Ribbon.

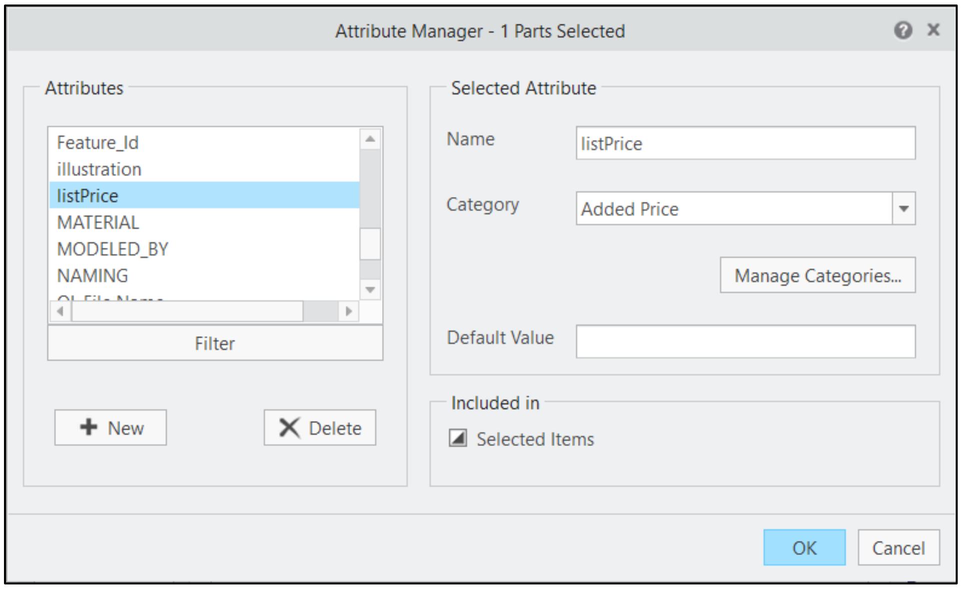


c. In the **Part Properties** menu, open the **Attributes** tab and then click **Manage Attributes**.



Name	Category	Value
Display Name	System Properties	electronics
eBOM Name	System Properties	
illustration	PVS File Properties	fig5-all-lift
OL File Name	System Properties	
partNumber	PVS File Properties	electonicspack-3333
sBOM Depth	System Properties	4
sBOM ID	System Properties	4
sBOM ID Path	System Properties	/0/18/4
sBOM Name	System Properties	electronics
sBOM Path	System Properties	/Illustration/QUADCOPTE
sBOM_Type	PVS File Properties	full_weld

d. In the **Attribute Manager** window that opens, scroll down to the **listPrice** attribute. It is listed as an attribute from when you created it but is not currently associated with this part. This can be determined the way that the **Selected Items** check box is checked off, which is saying that the attribute is on other items, but not this one.


Attribute Manager - 1 Parts Selected

Attributes

Feature_Id
illustration
listPrice
MATERIAL
MODELED_BY
NAMING

Selected Attribute

Name: listPrice

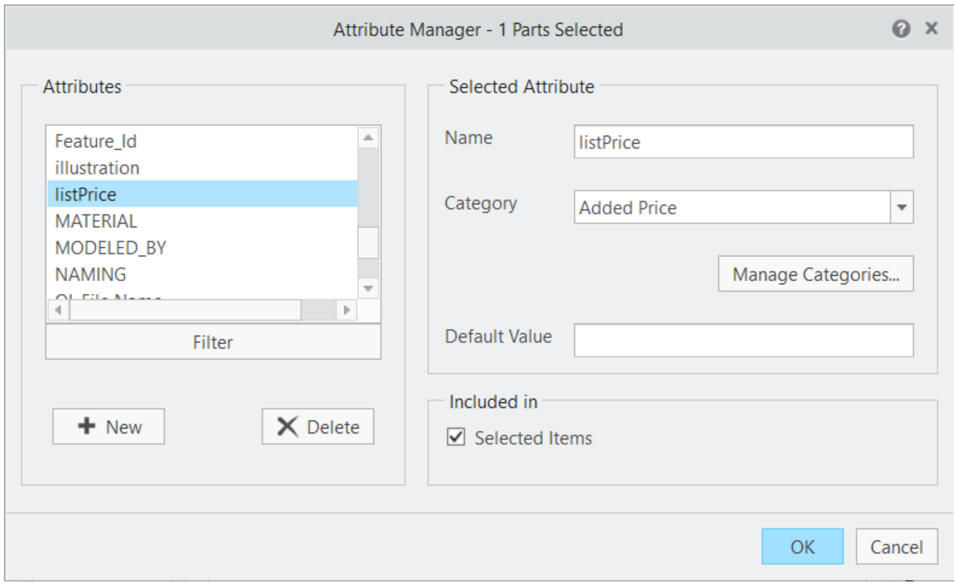
Category: Added Price

Default Value:

Included in

☐ Selected Items

e. Change **Selected Items** to be checked off as shown below and click **OK**. This will add **listPrice** as an attribute for **electronics**. Click **Cancel** to exit this window.


Attribute Manager - 1 Parts Selected

Attributes

Feature_Id
illustration
listPrice
MATERIAL
MODELED_BY
NAMING

Selected Attribute

Name: listPrice

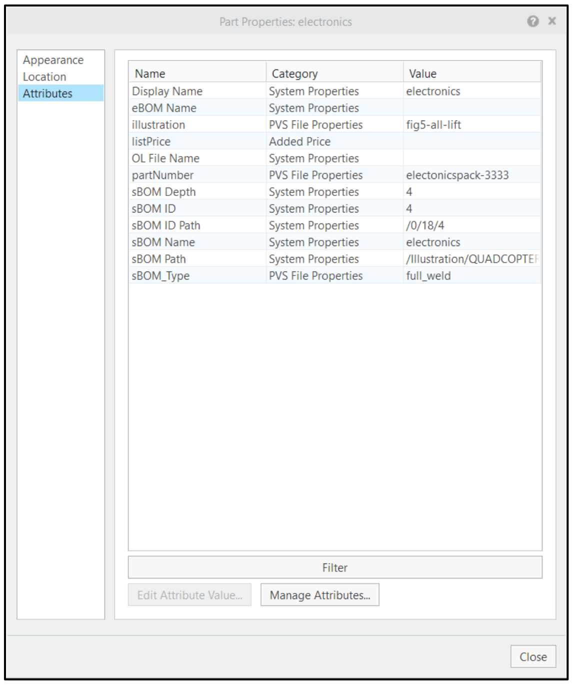
Category: Added Price

Default Value:

Included in

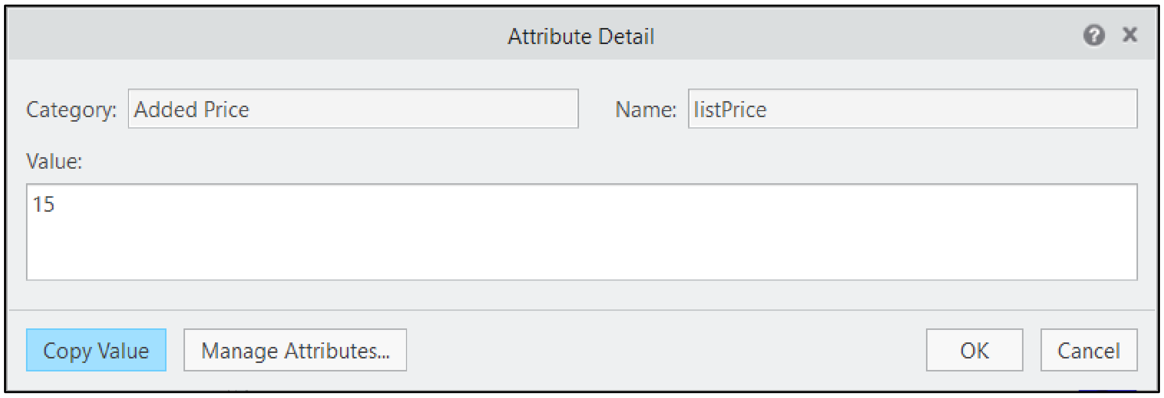
☑ Selected Items

f. Verify that **listPrice** is shown in the **Part Properties** menu and then click **Close**.



Name	Category	Value
Display Name	System Properties	electronics
eBOM Name	System Properties	
illustration	PVS File Properties	fig5-all-lift
listPrice	Added Price	
OL File Name	System Properties	
partNumber	PVS File Properties	electonicspack-3333
sBOM Depth	System Properties	4
sBOM ID	System Properties	4
sBOM ID Path	System Properties	/0/18/4
sBOM Name	System Properties	electronics
sBOM Path	System Properties	/Illustration/QUADCOPTE
sBOM_Type	PVS File Properties	full_weld

g. Once you are back to the **Structure Edit** window, in the **Attributes** (**electronics**) tab at the bottom of your screen, double click the **listPrice** attribute. Type *15* in the **Value** box and click **OK**.



Category: Added Price

Name: listPrice

Value:

15

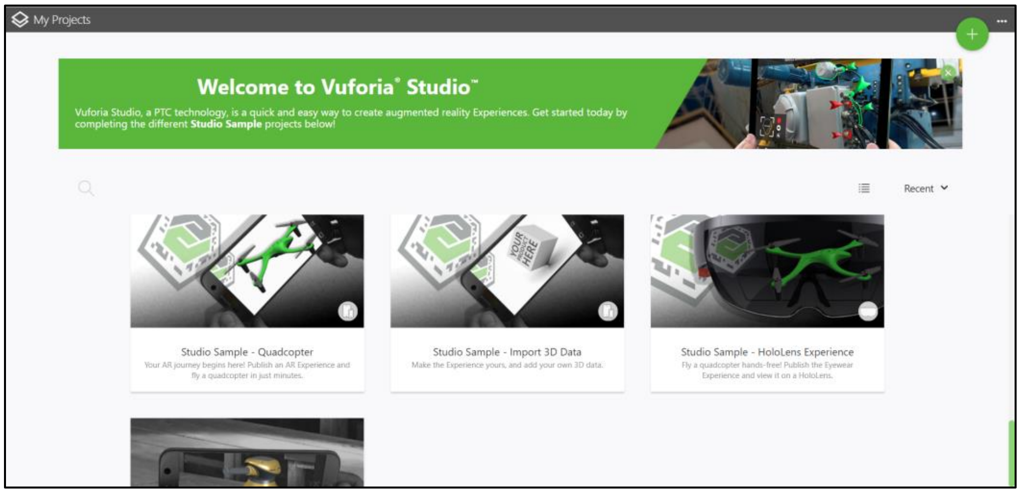
h. Once this is completed, open the **Home** tab in the Ribbon and click **Close Edit** to exit **Structure Edit** mode.

9. Click **Save As** and name the file *quadcopter.c3di*. It's okay to overwrite the first version of quadcopter.c3di that you created in **Metadata 101**, as this new version contains all the information from that section plus the new attributes created. A completed .c3di file named quadcopter301.c3di will be provided.
10. Click **Publish > Publish** and save the file as *quadcopter.pvz*. Similar to the previous step, it's okay to overwrite the previous file. A completed .pvz file named quadcopter301.pvz in the GitHub folder.

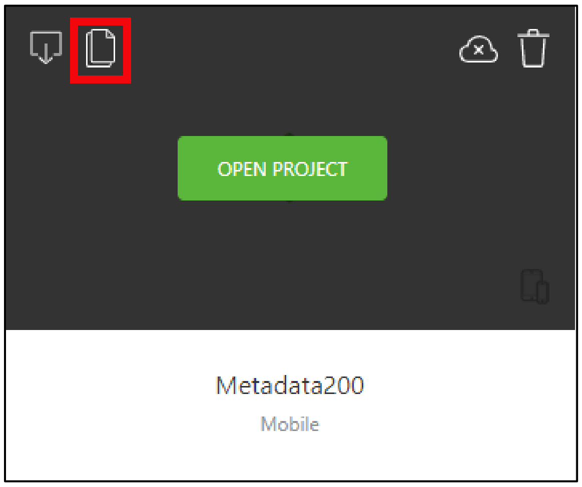
301.2 Set Up the Project

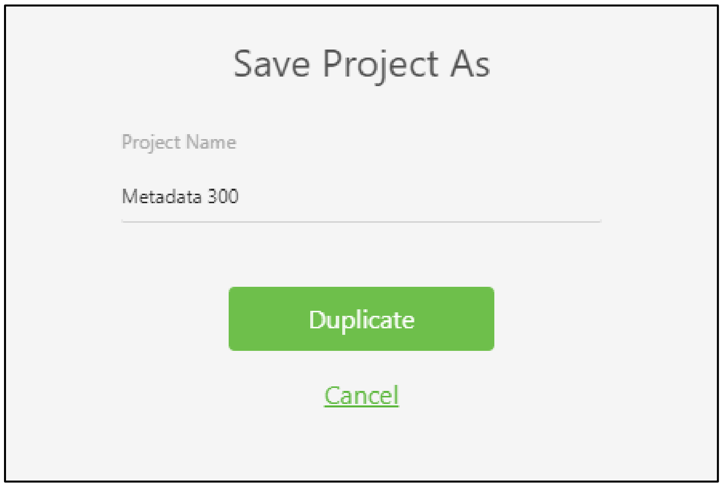
This section will review how to set up the model inside the Vuforia Studio project.

1. Open Vuforia Studio.


My Projects
Welcome to Vuforia® Studio™
Vuforia Studio, a PTC technology, is a quick and easy way to create augmented reality Experiences. Get started today by completing the different Studio Sample projects below!
Recent
Studio Sample - Quadcopter
Your AR journey begins here! Publish an AR Experience and fly a quadcopter in just minutes.
Studio Sample - Import 3D Data
Make the Experience yours, and add your own 3D data.
Studio Sample - HoloLens Experience
Fly a quadcopter hands-free! Publish the Eyewear Experience and view it on a HoloLens.

2. Start by creating a duplicate of the **Metadata 200** experience that was created in the previous tutorials. Click the **Save As** button in the top-left corner of the experience. Name the new experience **Metadata 300** and click **Duplicate**.

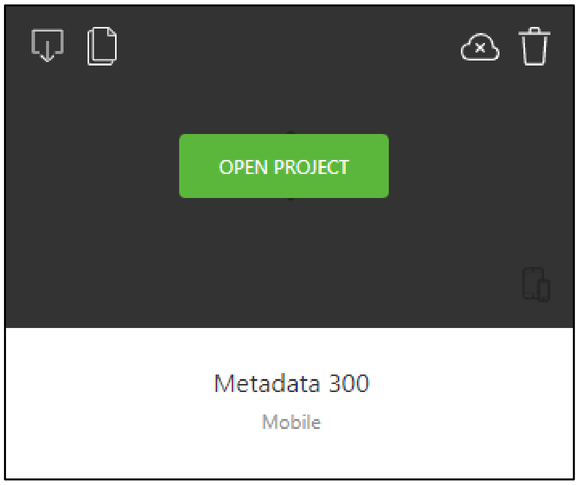



Save Project As

Project Name

Metadata 300

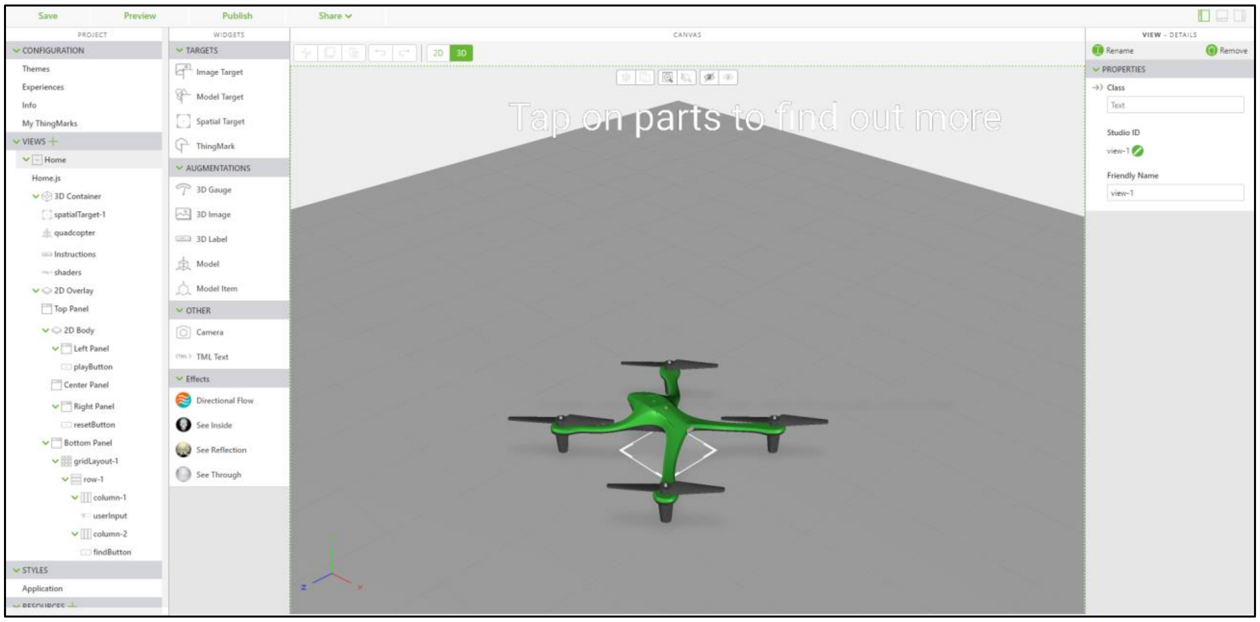
3. A new project named **Metadata 300** appears on your **My Projects** page. Click **Open Project.**



Metadata 300

Mobile

4. The experience should look exactly like the **Metadata 200** experience. If so, then the duplication has been completed successfully.

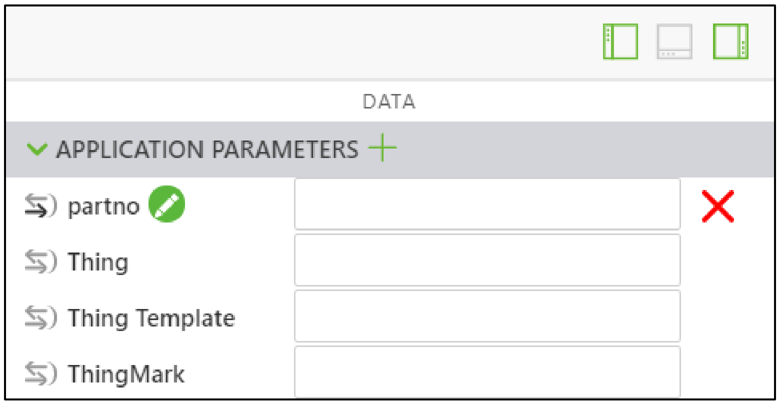


5. The first thing you will update is the quadcopter.pvz model. The new model includes the listPrice attribute that was added in section **301.1**. This model can be added by opening the **quadcopter** model widget and clicking the **+** next to **Resource**. Navigate to and select the updated model and click **Add.**

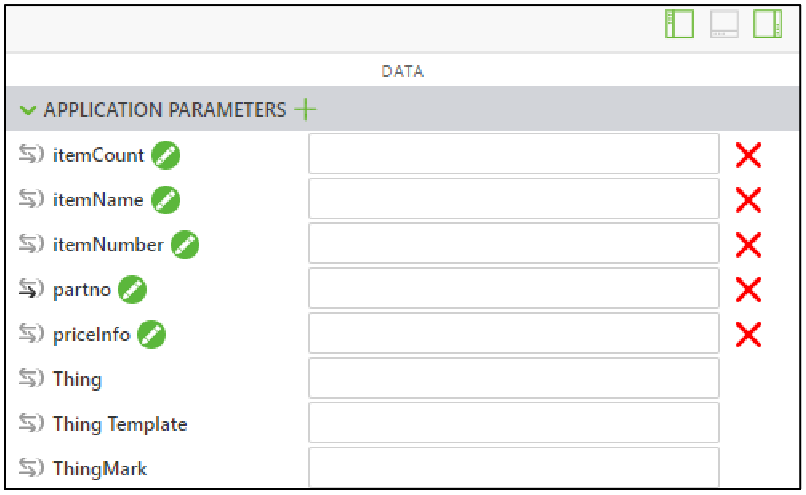
301.3 Add Application Parameters and Edit the Popup with an If Else Statement

In addition to reusing much of the code from the **Metadata 200** experience, you will add some new code and make changes to the 3D interface. This section focuses on adding new application parameters for the model and editing the popup so that its appearance changes based on which part is selected.

1. Before adding the new JavaScript, some new application parameters need to be created. The application parameters will be used for binding attributes of the quadcopter model to parameters inside Studio that can be used for data binding.
   a. Open the **Data** pane on the right side of the screen, and expand the list of **Application Parameters**.

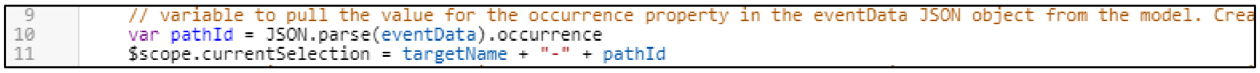


b. Use the green **+** next to the **Application Parameters** to add the following application parameters: **itemCount**, **itemName**, **itemNumber**, and **priceInfo**. **itemCount** is used to display the amount of a given part that is being selected, **itemName** is used to store the name of the selected part, **itemNumber** will be the part number, and **priceInfo** is the price of the part. Do not enter any data into the box next to each of these parameters; the data will be added from the attributes of the model. In a later activity, this data will be retrieved from an outside source.



2. Open **Home.js**. Below the `pathId` variable, create a variable in the `$scope` object named `currentSelection`. This will make the variable accessible throughout the code in different functions. This will also be used later when additional functions are added.

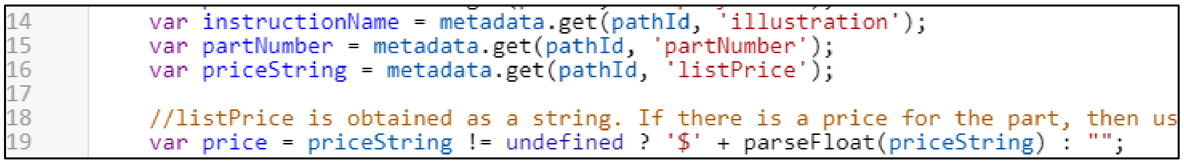
```
$scope.currentSelection = targetName + "-" + pathId
```



```
    // variable to pull the value for the occurrence property in the eventData JSON object from the model. Crea
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId
```

3. Below the `partName`, `instructionName`, and `partNumber` variables, add two new variables: `priceString` and `price`. `priceString` will use `metadata.get` to retrieve the `listPrice` attribute from the model. `price` uses a conditional operator to change that value from a `string` data type to a `float` so it can be registered as a number if there is a price associated with the part, and if not, then it becomes an empty string.

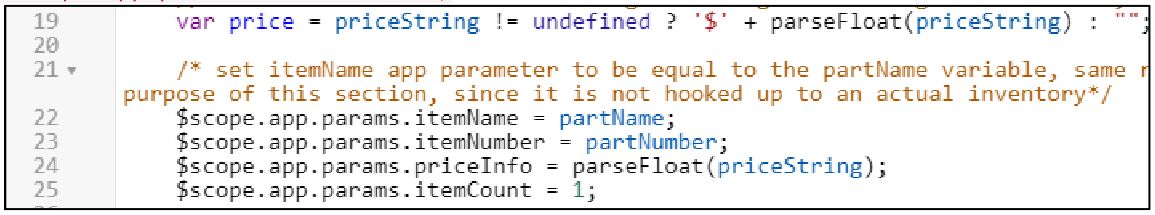
```
var priceString = metadata.get(pathId, 'listPrice');

    //listPrice is obtained as a string. If there is a price for the part, then use
parseFloat to turn the string into a float. If there is not a defined price, set price to
''
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";
```



```
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber = metadata.get(pathId, 'partNumber');
    var priceString = metadata.get(pathId, 'listPrice');

    //listPrice is obtained as a string. If there is a price for the part, then us
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";
```

4. Next, initialize the application parameters created in Step 1 of this section so that they are equal to some of the variables that you just created. These will be used when adding parts to the cart, which will be created in a later section. For this portion of the project, the inventory of each part, the `itemCount`, is going to be `1` since there is no external data attached to it. `priceInfo` is slightly different from `price`, since it does not have a `$` added to the variable like `price` has because it is used for adding the total of the cart in a later section.

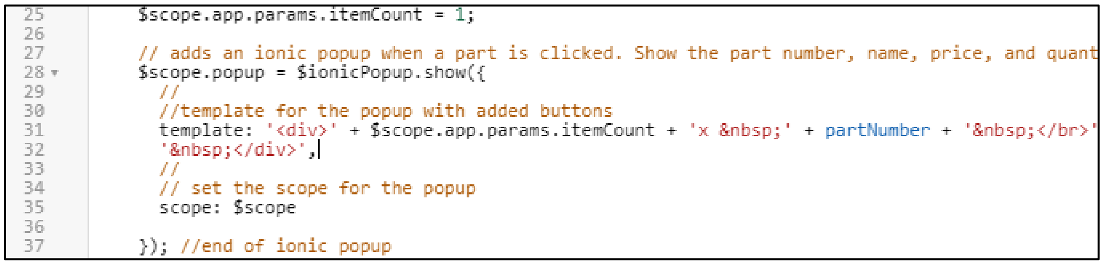
```
/* set itemName app parameter to be equal to the partName variable, same relationship
with itemNumber and partNumber and priceInfo and price. Set the itemCount to 1 for the
purpose of this section, since it is not hooked up to an actual inventory*/
    $scope.app.params.itemName = partName;
    $scope.app.params.itemNumber = partNumber;
    $scope.app.params.priceInfo = parseFloat(priceString);
    $scope.app.params.itemCount = 1;
```



```
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";

    /* set itemName app parameter to be equal to the partName variable, same r
purpose of this section, since it is not hooked up to an actual inventory*/
    $scope.app.params.itemName = partName;
    $scope.app.params.itemNumber = partNumber;
    $scope.app.params.priceInfo = parseFloat(priceString);
    $scope.app.params.itemCount = 1;
```

5. The Ionic popup must be edited from its former state to accommodate additional information from this section. More model information and functional buttons will be added to the popups. The buttons you will add are **Add to Cart**, which adds the selected part to the cart that is going to be created later, **Continue**, which closes out of the popup without completing any action, and **Disassemble**. The **Disassemble** button will take the place of the **playButton** from the 2D interface and allow you to activate the sequence from the popup. This new code will take the place of the old popup that was created.
    a. Add the `itemCount` and `price` variables to the popup.

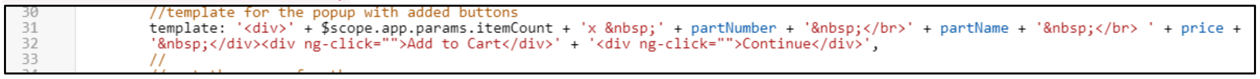
```
   // adds an ionic popup when a part is clicked. Show the part number, name, price, and
quantity of the selected object. &nbsp;</br> adds a line break between the two variables
    $scope.popup = $ionicPopup.show({
      //
      // set the template for the popup
      template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
      '&nbsp;</div>',
      //
      // set the scope for the popup
      scope: $scope

    }); //end of ionic popup
```



```
    $scope.app.params.itemCount = 1;

    // adds an ionic popup when a part is clicked. Show the part number, name, price, and quant
    $scope.popup = $ionicPopup.show({
      //
      //template for the popup with added buttons
      template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>'
      '&nbsp;</div>',
      //
      // set the scope for the popup
      scope: $scope

    }); //end of ionic popup
```

b. Add buttons for **Add to Cart** and **Continue** to the popup. These will be tied to functions that are activated by clicking the buttons later.

```
//template for the popup with added buttons
      template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
      '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Continue</div>',
```



```
      //template for the popup with added buttons
      template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
      '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-click="">Continue</div>',
      //
```

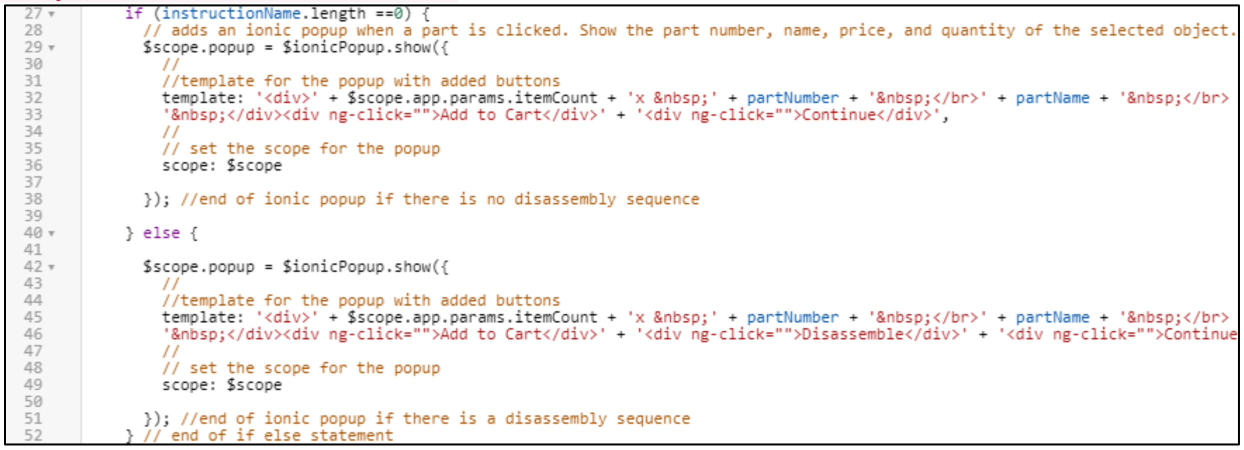
c. Instead of binding the disassembly sequence to a button on the 2D interface, it is going to be bound to a button in the popup. Since only certain parts have disassembly sequences attributed to them, an `if else` statement needs to be created to determine which version of the popup will appear, the one with all three buttons, or only the **Add to Cart** and **Continue** buttons. Both portions of this `if else` statement are based upon the length of the `instructionName` variable. Each button in the popup will have a function added to it later in this tutorial.

   i. For the first condition, if the length of the `instructionName` variable is 0, it means that the selected part does not have an associated disassembly sequence, which therefore means it does not need a **Disassemble** button in the popup. It will only have the **Add to Cart** and **Continue** buttons.

```
  if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price,
and quantity of the selected object. &nbsp;</br> adds a line break between the two
variables
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence
```

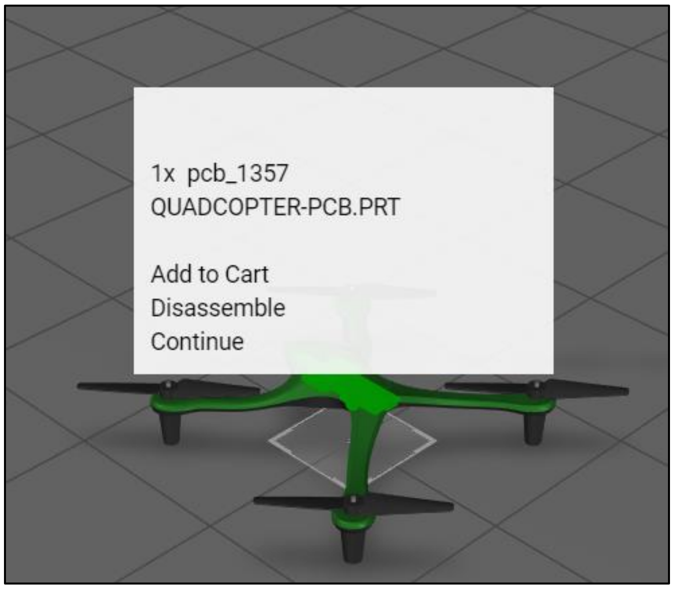
   ii. In the second condition that uses the `else` statement when `instructionName` isn’t equal to 0, there is an associated disassembly sequence, and the **Add to Cart** and **Continue** buttons are available along with a newly added **Disassemble** button to the popup.

```
    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Disassemble</div>' + '<div ng-click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement
```

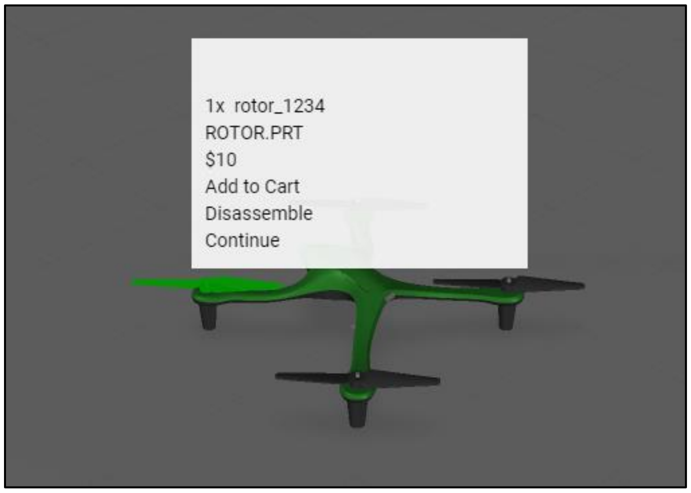


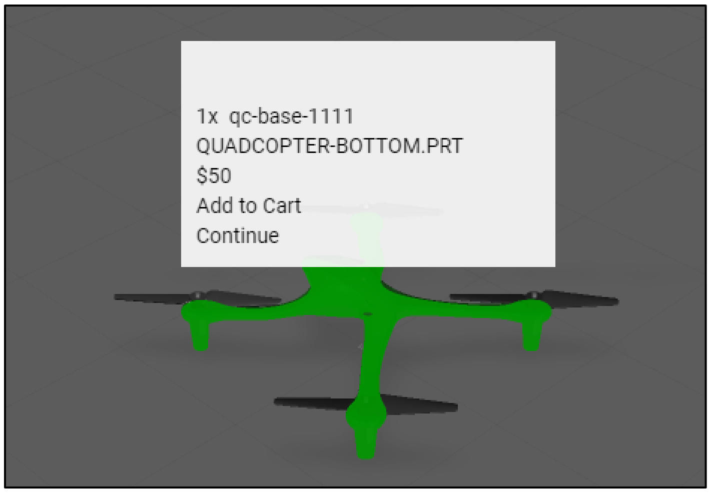
```
    if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price, and quantity of the selected object.
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br>
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br>
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-click="">Disassemble</div>' + '<div ng-click="">Continue

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement
```

d. Click on the PCB. Notice that because it is a now part of a combined part that a price does not appear in the popup.



e. Click on the front-left rotor and compare it to the first image below. Click on the base of the rotor and compare it to the second image below. If there is a **Disassemble** button when the rotor is clicked and there is not one when the base is clicked, then this logic has been added successfully. Visit Appendix 1 to view the full code for this section.





301.4 Remove Outdated Functions and Text

Some functions from **Metadata 200** must be updated in this new experience to add more functionality to the popover or remove outdated portions of the code. Removing outdated code is good practice for keeping clean code.

1. There will be functions called when the buttons in the popup are clicked, but first, there are functions that need to be removed and replaced inside the experience.
   a. Remove the `closePopup` function and the `$timeout` service calling the `closePopup` function. This can be removed because unlike the first experience where the popup disappears on its own, the popup is only going to disappear when prompted by a popup button.

   b. Remove the `playit` function that was tied to the **playButton** in the experience. This function is going to be automated by the **Disassemble** button in the popup.
   c. Inside the `findMeta` function, remove the following lines of code and associated comments. They reference the **playButton** widget, which will be deleted from the 2D interface in the next section.

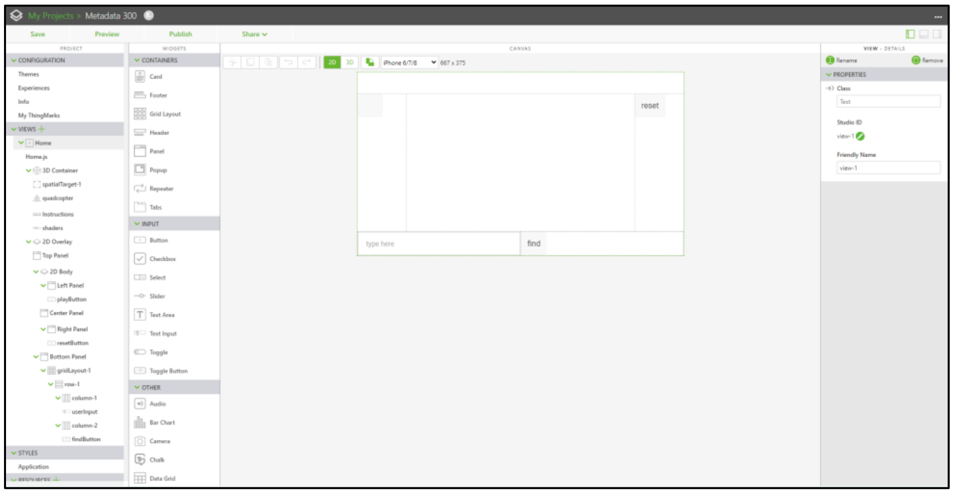
```
$scope.view.wdg.playButton.text = '';
$scope.view.wdg.playButton.toPlay = undefined;
```

2. Visit Appendix 2 for the updated code.

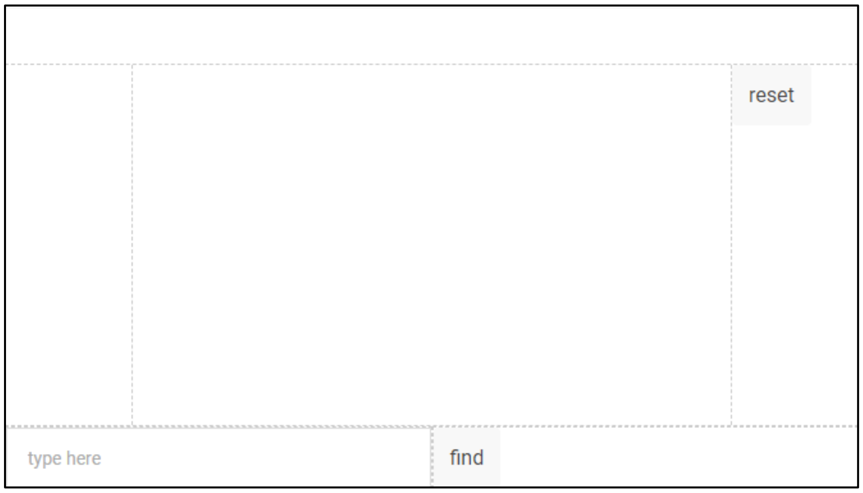
301.5 Update the 2D Interface

The 2D canvas from **Metadata 200** must be updated to fit the needs of the new popup and the cart. Buttons will be added for ordering and clearing the cart, along with a **repeater** for listing all the items in the cart.

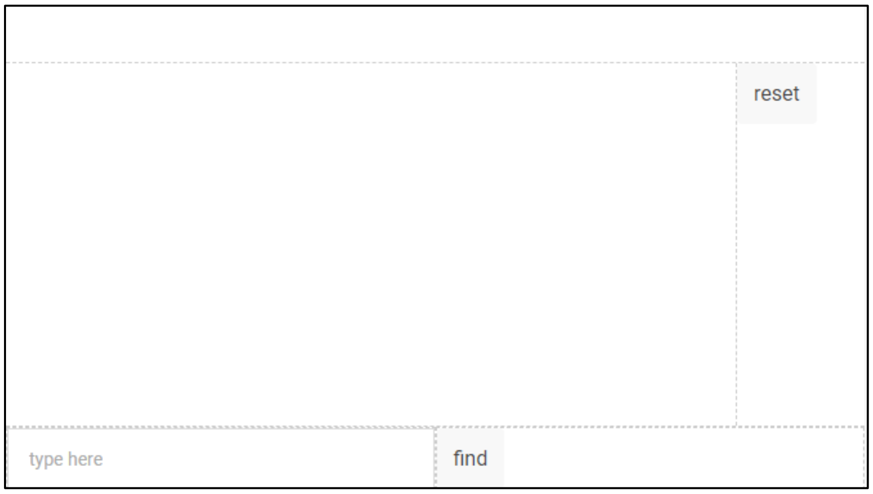
1. Open the 2D canvas.



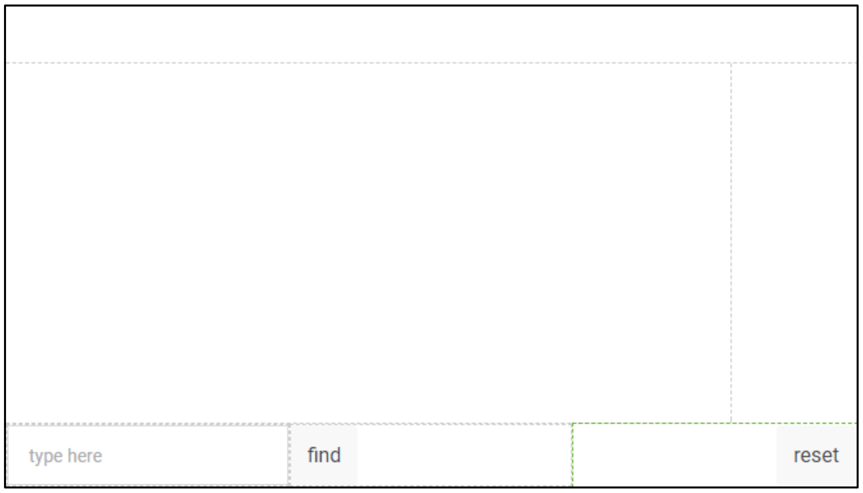
2. Click on the **playButton**. In the **Properties** pane, select **Remove** to remove it from the canvas. The function of this button will now be triggered by the popup.



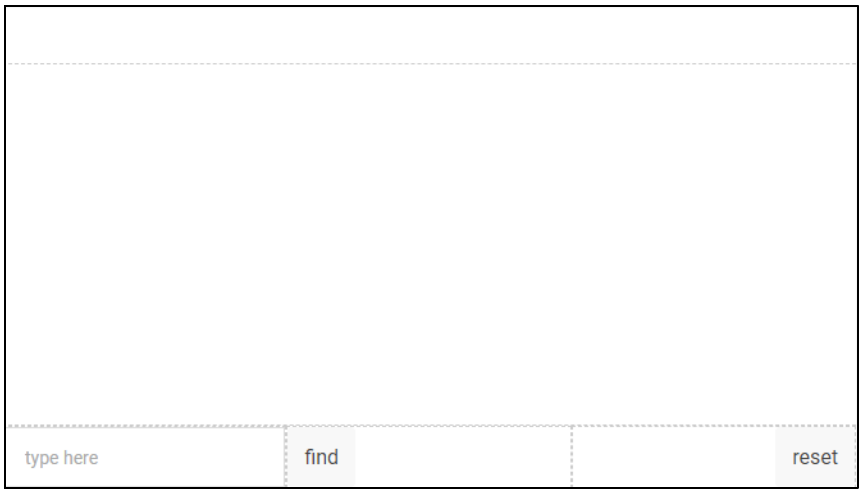
3. Click on the **left panel** in the **Views** tab. **Remove** the panel.



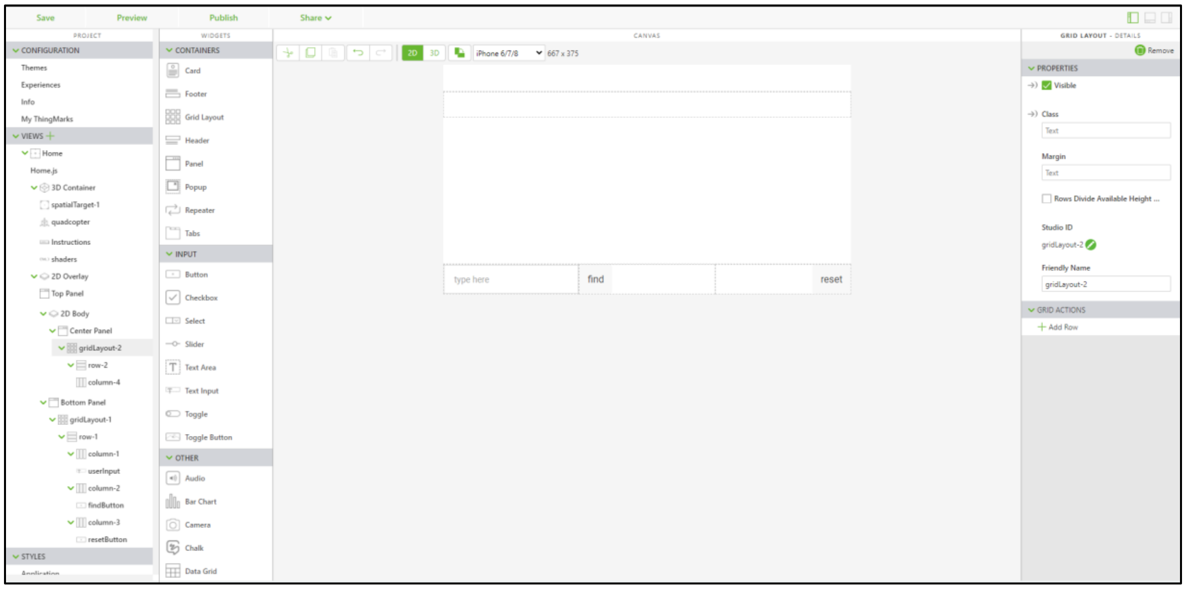
4. In **row-1** of **gridLayout-1** in the bottom panel**,** click **Add Column** to add a column to the bottom row. Drag the **resetButton** widget from its original location and drop it into the new column. In the properties for the new column, set the **Alignment** dropdown to **End** so the layout looks like the image below.



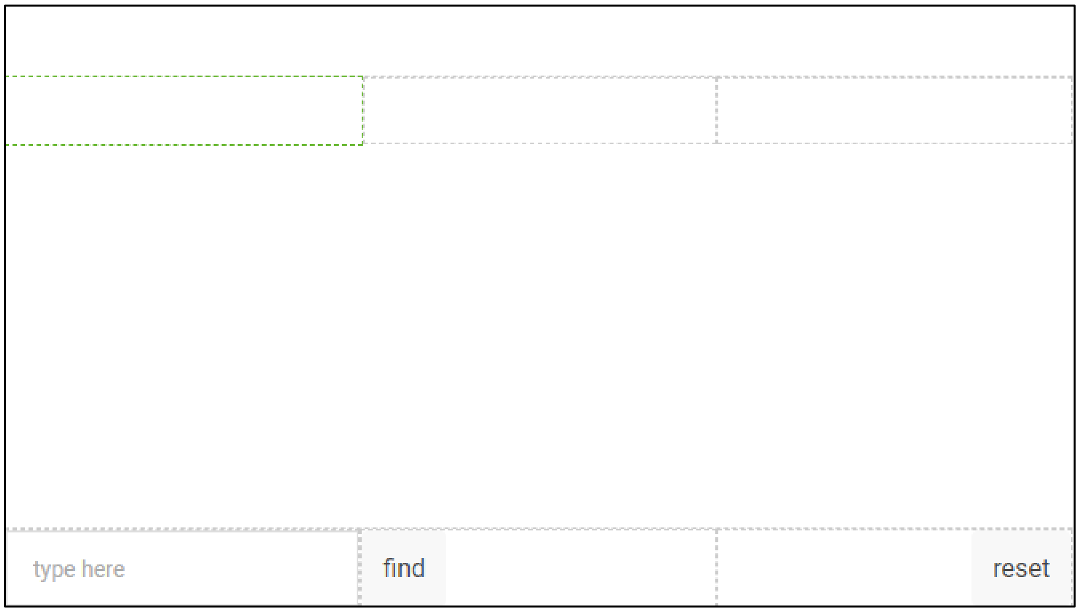
5. **Remove** the right panel**.**



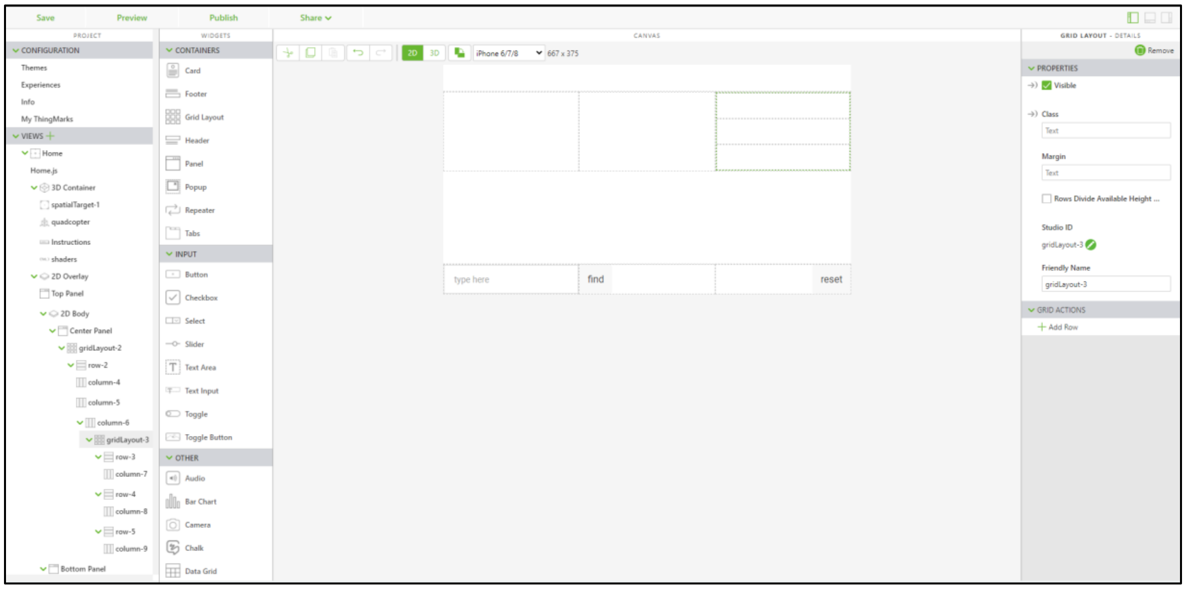
6. Now that parts of the layout from the old experience have been removed, new 2D widgets will be added to the experience.
    a. Drag a **Grid Layout** widget onto the **Center Panel**. This will be used to split this panel into columns of equal width.



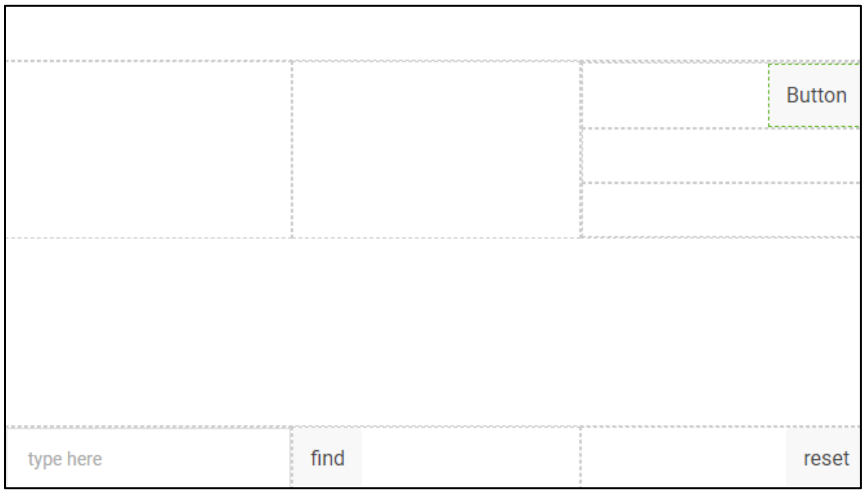
b. Click into **column-4**, the newly created column in the **Grid Layout.** Under **Grid Actions,** click **Add Column** twice. This will split the **Grid Layout** into three equal columns.



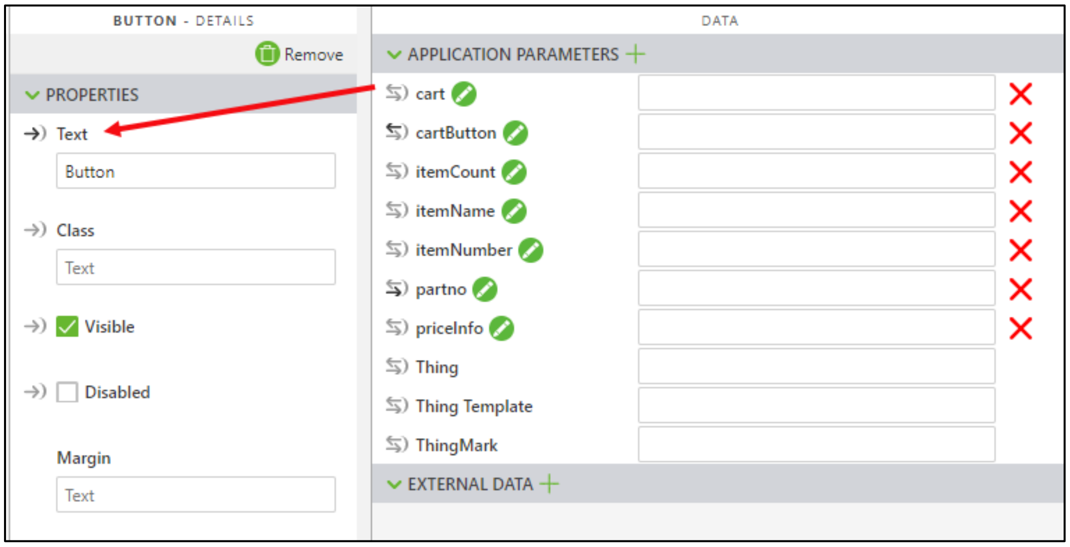
c. Drag another **Grid Layout** widget into the **right** column, **column-6**. In the **Grid Layout** properties, click **Add Row** twice so that **column-6** now has three rows.



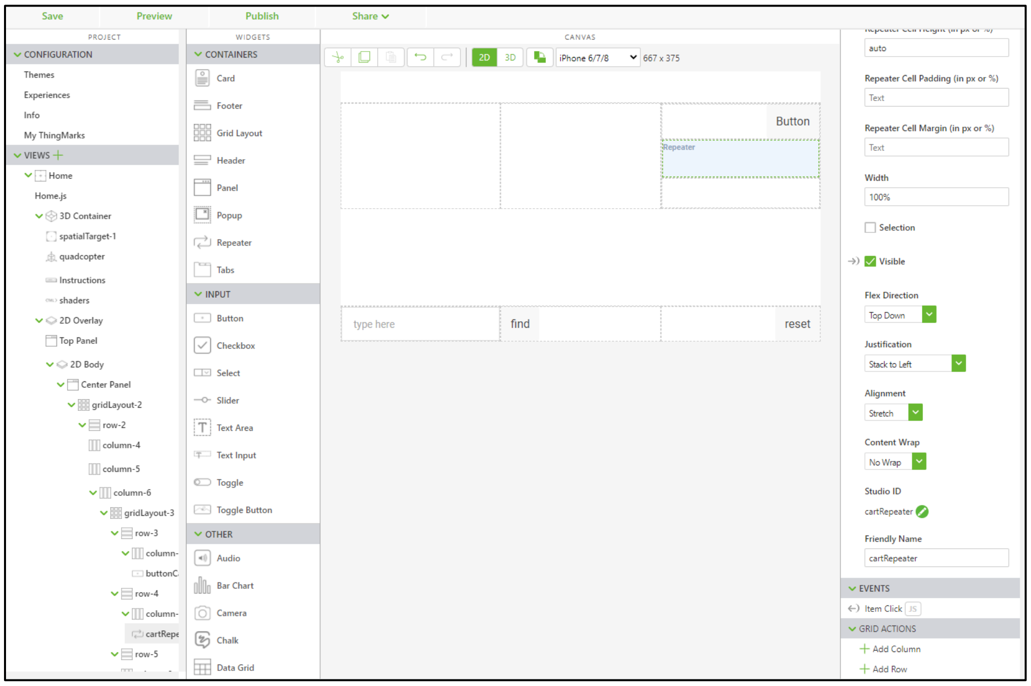
d. Drag a **Button** widget into the **top** row of the new grid layout. Change the **Studio ID** of this button to be *buttonCart*. Click into **column-7**, or whatever the name is for the column that the button was just placed into, and change the **Alignment** property to **End** to place the button on the right side of the row. This button will display the quantity of items in the cart when it is populated.



e. Open the **Data Panel**. Add a new application parameter called **cartButton**. Open the properties for **buttonCart** and drag the binding arrows from **cartButton** to the **Text** property in **buttonCart**. This binding will allow the text for this button to be changed using code in **Home.js**.

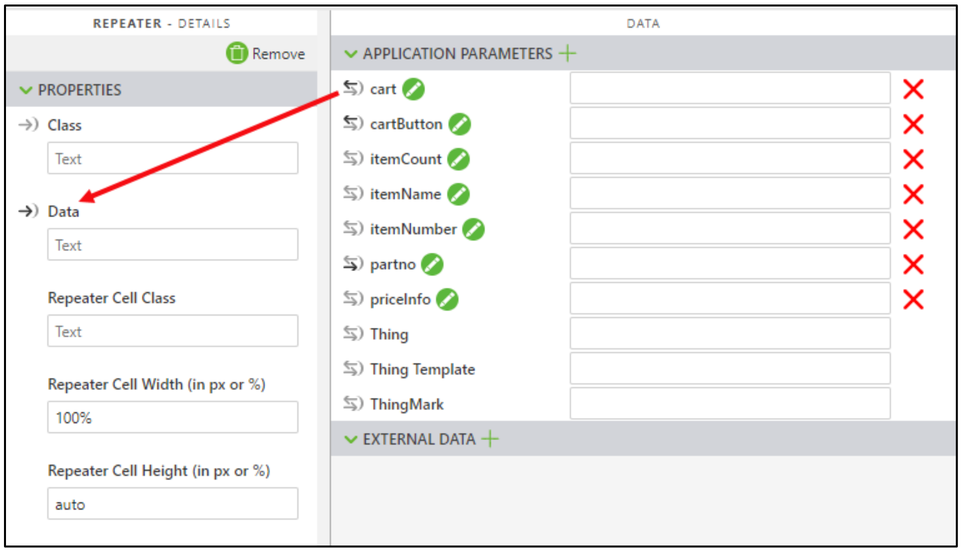


f. In the row below the **buttonCart** button, add a **Repeater** widget. The **Repeater** widget allows data to be displayed in a desired format as many times as required. It will be used for logging information about the parts that are added to the cart through the ordering system and will grow as more parts are added. Change the **Studio ID** for this widget to *cartRepeater*.



Repeater Cell Height (in px or %): auto
Repeater Cell Padding (in px or %): Text
Repeater Cell Margin (in px or %): Text
Width: 100%
Selection
Visible ✓
Flex Direction: Top Down
Justification: Stack to Left
Alignment: Stretch
Content Wrap: No Wrap
Studio ID: cartRepeater
Friendly Name: cartRepeater

g. Open the **Data** panel. Create a new application parameter named **cart**. Use the binding arrows to drag the binding for **cart** onto the **Data** property of **cartRepeater**. This will link the data that is added to the cart based on part selections to the repeater.



REPEATER - DETAILS

Remove

PROPERTIES

Class: Text

Data: Text

Repeater Cell Class: Text

Repeater Cell Width (in px or %): 100%

Repeater Cell Height (in px or %): auto

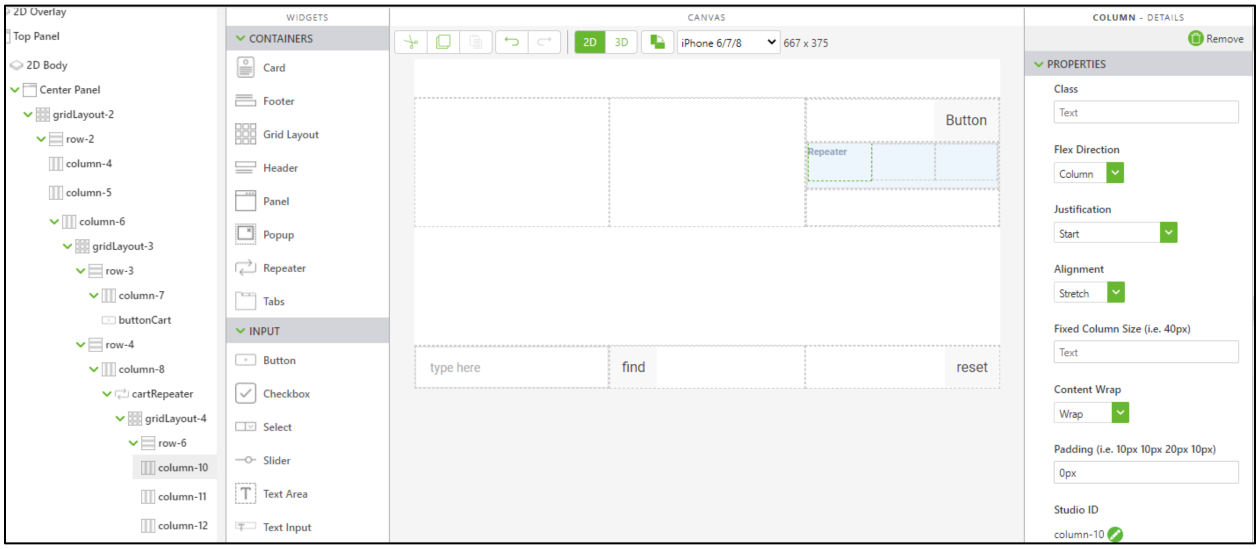
DATA

APPLICATION PARAMETERS

- cart
- cartButton
- itemCount
- itemName
- itemNumber
- partno
- priceInfo
- Thing
- Thing Template
- ThingMark

EXTERNAL DATA

h. Add a **Grid Layout** widget onto the **cartRepeater**. Click into the new column that was created for the grid layout and click **Add Column** twice so the repeater is split into 3 columns.



2D Overlay
Top Panel
2D Body
Center Panel
gridLayout-2
row-2
column-4
column-5
column-6
gridLayout-3
row-3
column-7
buttonCart
row-4
column-8
cartRepeater
gridLayout-4
row-6
column-10
column-11
column-12

WIDGETS

CONTAINERS: Card, Footer, Grid Layout, Header, Panel, Popup, Repeater, Tabs

INPUT: Button, Checkbox, Select, Slider, Text Area, Text Input

CANVAS

2D 3D iPhone 6/7/8 667 x 375

Button

Repeater

type here find reset

COLUMN - DETAILS

Remove

PROPERTIES

Class: Text

Flex Direction: Column

Justification: Start

Alignment: Stretch

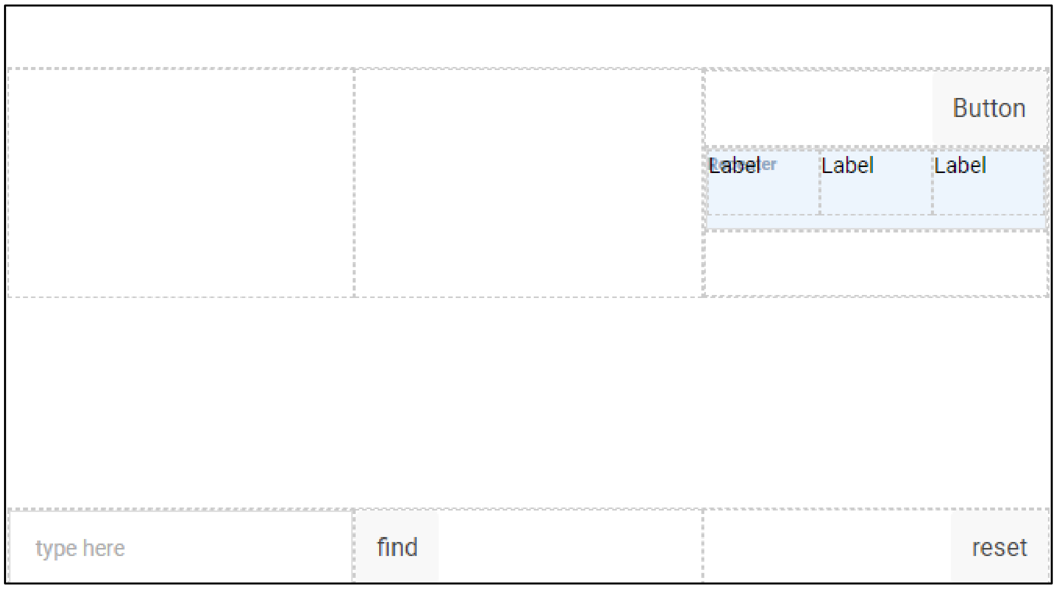
Fixed Column Size (i.e. 40px): Text

Content Wrap: Wrap

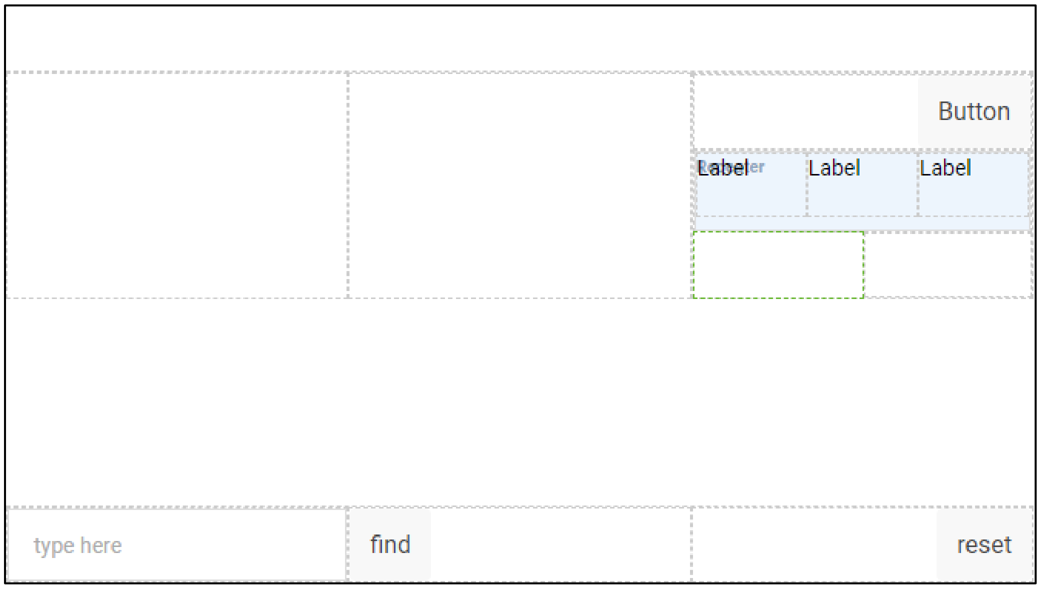
Padding (i.e. 10px 10px 20px 10px): 0px

Studio ID: column-10

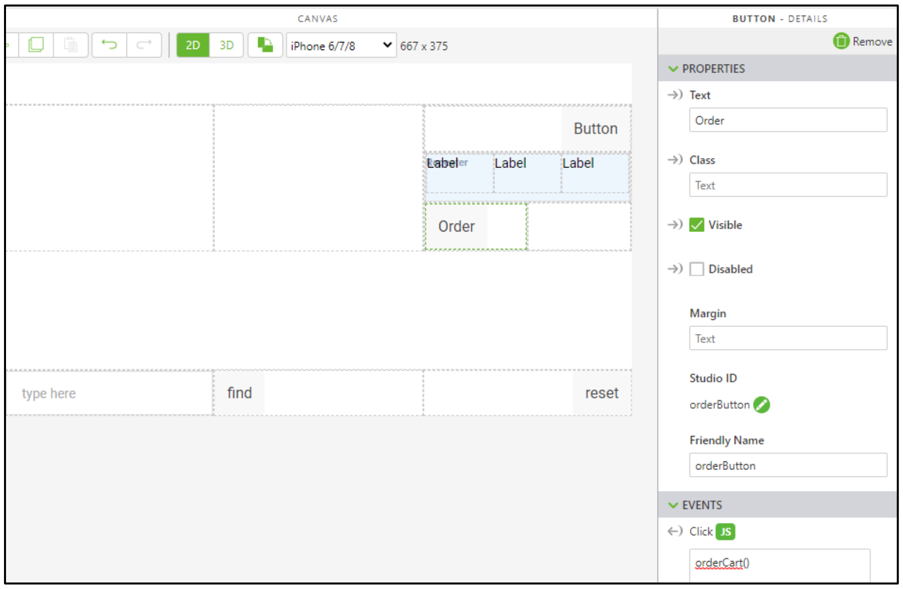
i. Add a **Label** widget into each of the newly created columns. These labels will be edited in a later portion of the project.



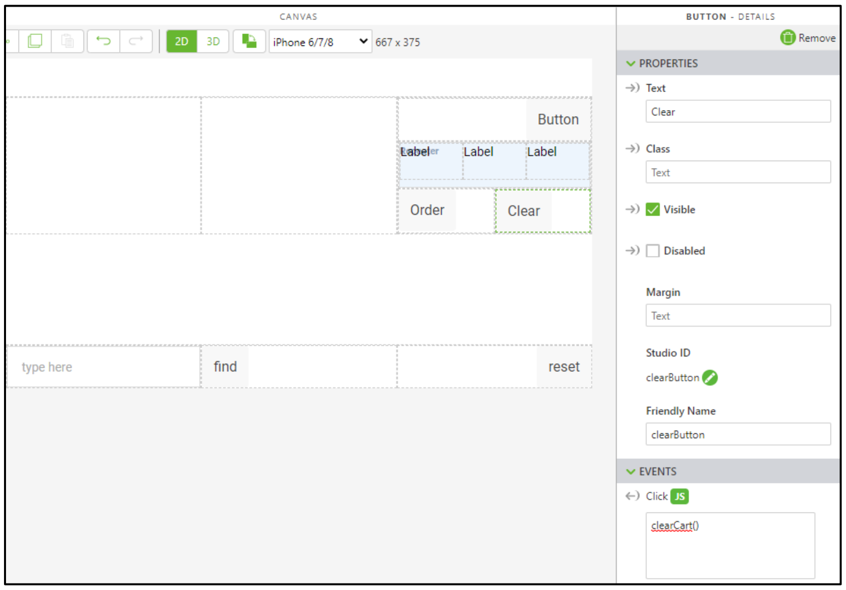
j. Click into the third row of this section and select **Add Column** to add another column.



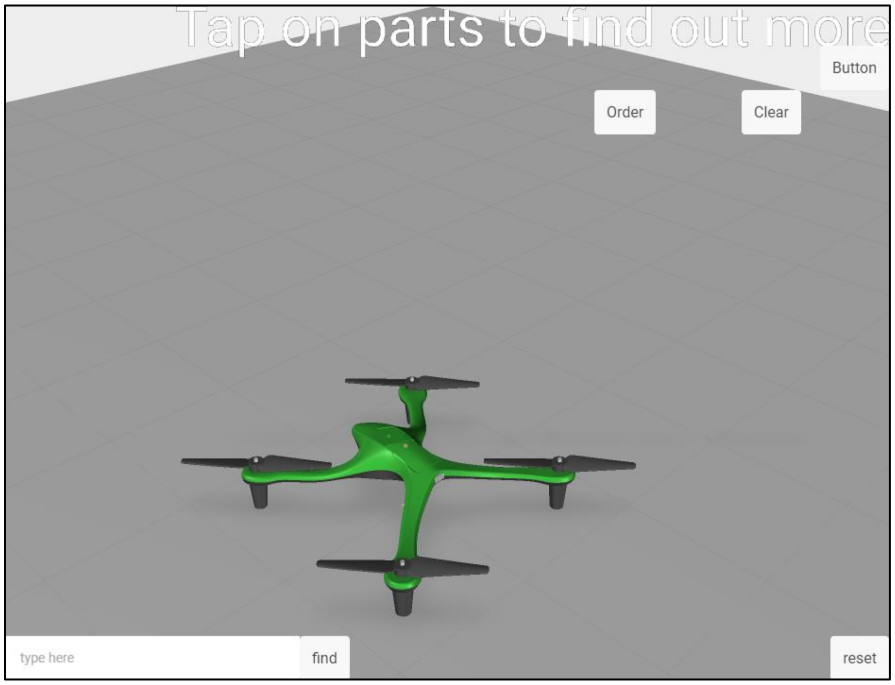
k. Add a **Button** widget to the left column. Change the **Text** property to *Order* and the **Studio ID** to *orderButton***.** Open the **JS** box for the **Click** event and type in *orderCart()*. This will call a function that will be added later in the project for ordering the parts in the cart.



l. In the right-hand column of the same row, add another **Button** widget. This time, Change the **Text** property to *Clear*, the **Studio ID** to *clearButton*, and in the **JS** section of the **Click** event type *clearCart()*. This button will be used to clear all listed items from the cart.



a. Save the experience and click **Preview**. If it looks like the image below, then the 2D layout has been created correctly.

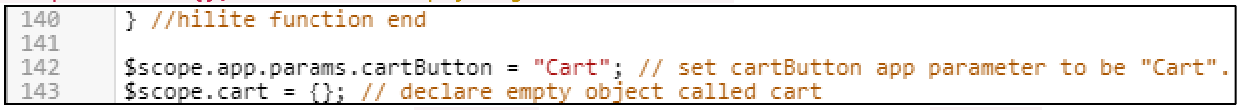


301.6 Build the Cart and Popup Functions

When you click on a part, you are given the option to add the part to a cart, view a disassembly sequence, or continue inspecting the model. The cart contains information that is stored in the **cartRepeater** widget and grows as parts are added to the order. Functions are written both on their own and inside the popup to build the cart and interact with the model.

1. Open **Home.js**. In order to be able to interact with the model through button clicks in the popup, the following functions must be created. These functions will each be tied to a button.
    a. The `addToCart` function, which will add the selected parts to the cart, will start off as its own standalone function.
        i. Initialize the `cartButton` application parameter to just say `Cart`. This will be the beginning text in the **buttonCart** button and will change as items are added to the cart. Also, initialize an empty object named `cart`**.**

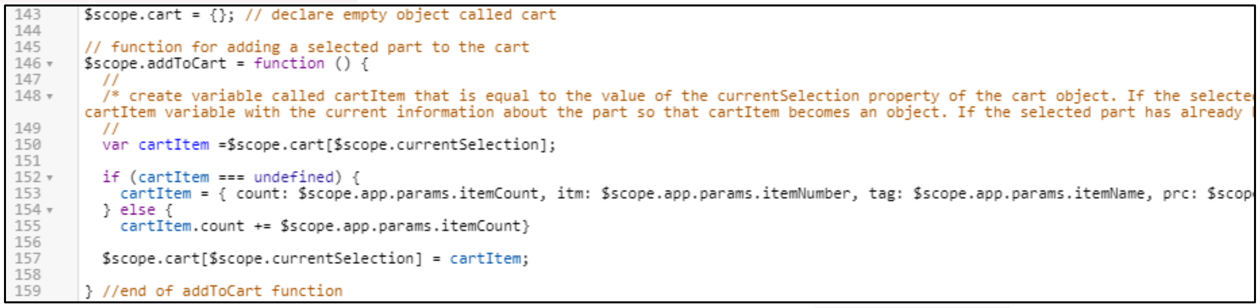
```
$scope.app.params.cartButton = "Cart"; // set cartButton app parameter to be "Cart". This will bind to the Text property for the buttonCart button
$scope.cart = {}; // declare empty object called cart
```



```
} //hilite function end

$scope.app.params.cartButton = "Cart"; // set cartButton app parameter to be "Cart".
$scope.cart = {}; // declare empty object called cart
```

ii. Start the `addToCart` function. A variable called `cartItem` will be created and initialized to be the value of the property that is equal to the part that is currently selected. The first time that a part is selected, this property will be undefined because the cart was initialized as an empty object. If the current selection has not yet been added to the `cart` object, aka `undefined`, then `cartItem` will become populated with the `itemCount`, `itemNumber`, `itemName`, and `priceInfo` application parameters for the selected part. All these application parameters will be set as property values inside the `cartItem` object. If the item has already been added to the cart once, then the only property in `cartItem` that will change is the `itemCount` parameter, which will increase by 1, since another instance of that same part has been added to the cart. After this `if` statement, the `cartItem` object and its properties will be added to the `cart` object. This process helps differentiate between the different parts that are added to the cart.

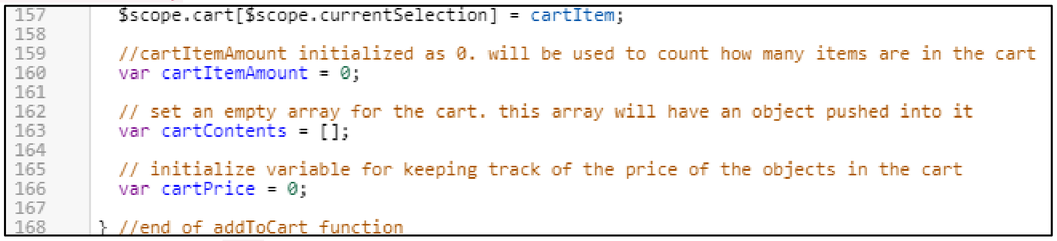
```
// function for adding a selected part to the cart
$scope.addToCart = function () {
  //
  /* create variable called cartItem that is equal to the value of the currentSelection
property of the cart object. If the selected part hasn't been added to the cart yet, then
the cartItem variable will be undefined and populate the cartItem variable with the
current information about the part so that cartItem becomes an object. If the selected
part has already been added, then the count property of cartItem will increase by the
item count*/
  //
  var cartItem =$scope.cart[$scope.currentSelection];

  if (cartItem === undefined) {
       cartItem = { count: $scope.app.params.itemCount, itm:
$scope.app.params.itemNumber, tag: $scope.app.params.itemName, prc:
$scope.app.params.priceInfo }
  } else {
    cartItem.count += $scope.app.params.itemCount}

  $scope.cart[$scope.currentSelection] = cartItem;

} //end of addToCart function
```



```
$scope.cart = {}; // declare empty object called cart

// function for adding a selected part to the cart
$scope.addToCart = function () {
  //
  /* create variable called cartItem that is equal to the value of the currentSelection property of the cart object. If the selecte
cartItem variable with the current information about the part so that cartItem becomes an object. If the selected part has already
  //
  var cartItem =$scope.cart[$scope.currentSelection];

  if (cartItem === undefined) {
    cartItem = { count: $scope.app.params.itemCount, itm: $scope.app.params.itemNumber, tag: $scope.app.params.itemName, prc: $scop
  } else {
    cartItem.count += $scope.app.params.itemCount}

  $scope.cart[$scope.currentSelection] = cartItem;

} //end of addToCart function
```

iii. Initialize more variables for the function. `cartItemAmount` will be used to count the number of items in the cart, `cartContents` will be initialized as an empty array that will hold the contents of the cart, and `cartPrice` will be the total price of the objects in the cart

```
//cartItemAmount initialized as 0. will be used to count how many items are in the cart
  var cartItemAmount = 0;
  // set an empty array for the cart. this array will have an object pushed into it
  var cartContents = [];

  // initialize variable for keeping track of the price of the objects in the cart
  var cartPrice = 0;
```

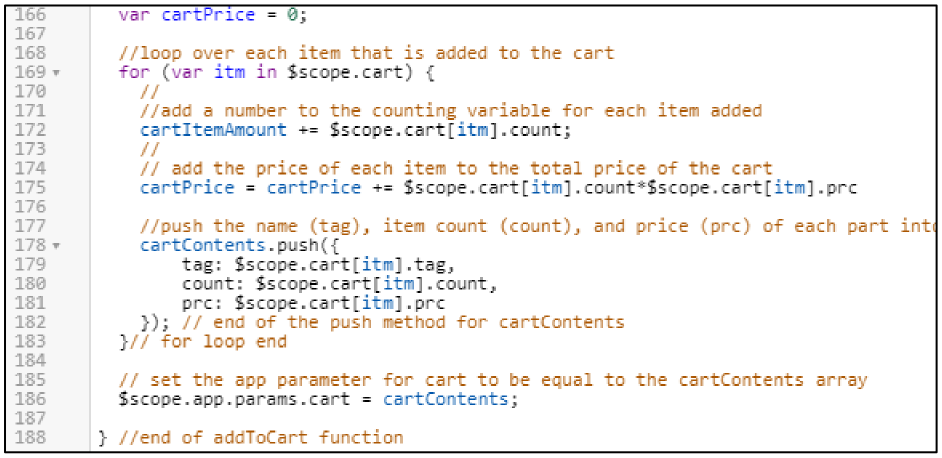


```
    $scope.cart[$scope.currentSelection] = cartItem;

    //cartItemAmount initialized as 0. will be used to count how many items are in the cart
    var cartItemAmount = 0;

    // set an empty array for the cart. this array will have an object pushed into it
    var cartContents = [];

    // initialize variable for keeping track of the price of the objects in the cart
    var cartPrice = 0;

} //end of addToCart function
```

iv. A `for` loop will be used to loop over each item that is in the cart to check for the following conditions. `itm` will be the counting variable for the loop. In this loop, the `count` property for the object corresponding to the selected part will be increased for each time that another quantity of the part is added to the cart. In addition to this, `cartPrice`, the price variable, will increase based on the `prc` property in the `cart` for the selected item. When an item is added to the cart, the price will increase by the price of the part multiplied by the quantity of parts added. Using the `.push` method, the name (`tag`), quantity (`count`), and price (`prc`) of the selected object will be pushed into the `cartContents` array. The `cartContents` array will then be set to be equal to the `cart` application parameter.

    1. This loop is used to populate the **repeater** widget that was added earlier and bound to the **cart** application parameter. The repeater will display each row of the `cart` array on the screen and whenever the data changes inside of `cart`, the repeater will reevaluate its contents and change the display.

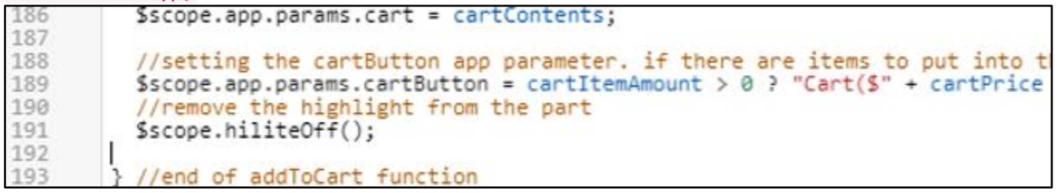
```
  //loop over each item that is added to the cart
  for (var itm in $scope.cart) {
    //
    //add a number to the counting variable for each item added
    cartItemAmount += $scope.cart[itm].count;
    //
    // add the price of each item to the total price of the cart
    cartPrice = cartPrice += $scope.cart[itm].count*$scope.cart[itm].prc

    //push the name (tag), item count (count), and price (prc) of each part into the
repeater for the cart
    cartContents.push({
        tag: $scope.cart[itm].tag,
        count: $scope.cart[itm].count,
        prc: $scope.cart[itm].prc
    }); // end of the push method for cartContents
  }// for loop end

  // set the app parameter for cart to be equal to the cartContents array
  $scope.app.params.cart = cartContents;
```



```
    var cartPrice = 0;

    //loop over each item that is added to the cart
    for (var itm in $scope.cart) {
      //
      //add a number to the counting variable for each item added
      cartItemAmount += $scope.cart[itm].count;
      //
      // add the price of each item to the total price of the cart
      cartPrice = cartPrice += $scope.cart[itm].count*$scope.cart[itm].prc

      //push the name (tag), item count (count), and price (prc) of each part int
      cartContents.push({
          tag: $scope.cart[itm].tag,
          count: $scope.cart[itm].count,
          prc: $scope.cart[itm].prc
      }); // end of the push method for cartContents
    }// for loop end

    // set the app parameter for cart to be equal to the cartContents array
    $scope.app.params.cart = cartContents;

} //end of addToCart function
```

v. The last portion of the `addToCart` function is editing the text for the **buttonCart** button by adding a value to the **cartButton** application parameter and removing the highlight from the selected part.

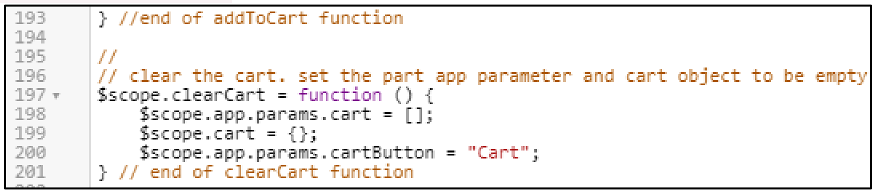
```
  //setting the cartButton app parameter. if there are items to put into the cart (true), the text of the cart button should be cart(total cost of cart). If false, just keep the button text as cart
  $scope.app.params.cartButton = cartItemAmount > 0 ? "Cart($" + cartPrice + ")" : "Cart";
  //remove the highlight from the part
  $scope.hiliteOff();
```



```
    $scope.app.params.cart = cartContents;

    //setting the cartButton app parameter. if there are items to put into t
    $scope.app.params.cartButton = cartItemAmount > 0 ? "Cart($" + cartPrice
    //remove the highlight from the part
    $scope.hiliteOff();

} //end of addToCart function
```

b. A function needs to be created to clear the cart when the **clearButton** button is selected. This function sets the application parameter for **cart** to be an empty array, sets the `cart` object to be empty, and changes the **Text** property for the **buttonCart** button to be set back to **Cart** without an item count using the **cartButton** application parameter**.**

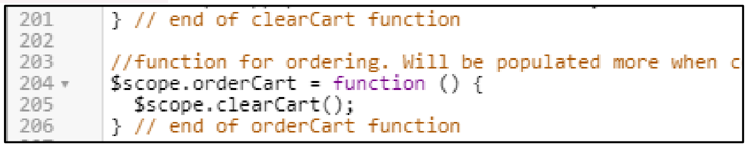
```
//
// clear the cart. set the part app parameter and cart object to be empty. change the text on the cart button back to just Cart
$scope.clearCart = function () {
    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartButton = "Cart";
} // end of clearCart function
```



```
} //end of addToCart function

//
// clear the cart. set the part app parameter and cart object to be empty
$scope.clearCart = function () {
    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartButton = "Cart";
} // end of clearCart function
```

c. When a ThingWorx connection is added to the experience in a later section, users will be able to use the **orderButton** button to place a part order. For now, the only functionality that needs to be added to the button is that it will clear the cart when pressed to simulate an order being sent.

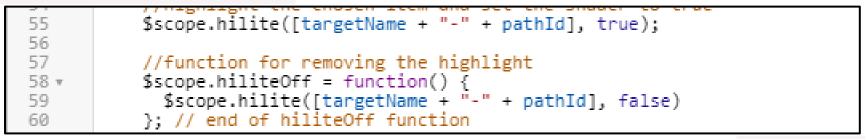
```
//function for ordering. Will be populated more when connected to ThingWorx
$scope.orderCart = function () {
  $scope.clearCart();
} // end of orderCart function
```



```
} // end of clearCart function

//function for ordering. Will be populated more when c
$scope.orderCart = function () {
  $scope.clearCart();
} // end of orderCart function
```

d. In order to make sure that the shader is removed from the model when the popup closes, a function called `hiliteOff` will be created inside the PTC API.

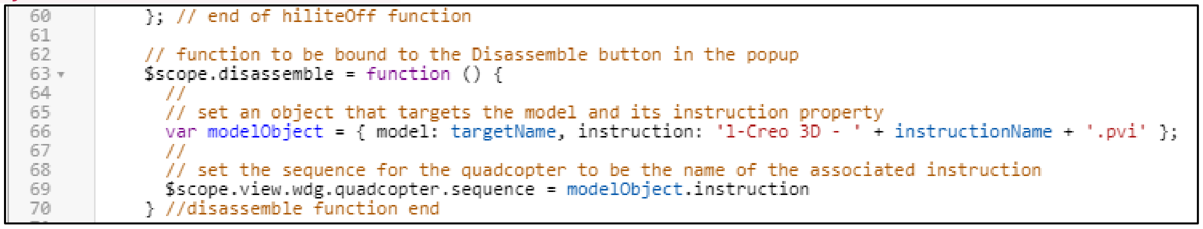
```
    //function for removing the highlight
    $scope.hiliteOff = function() {
      $scope.hilite([targetName + "-" + pathId], false)
    }; // end of hiliteOff function
```



```
    $scope.hilite([targetName + "-" + pathId], true);

    //function for removing the highlight
    $scope.hiliteOff = function() {
      $scope.hilite([targetName + "-" + pathId], false)
    }; // end of hiliteOff function
```

e. Finally, inside the PTC API, create a function called `disassemble`. This will be used to add the functionality to the **Disassemble** button on the popup. It uses the same logic that was used with the **playButton** and triggers the `sequenceloaded` listener event when the sequence of the model is set.

```
    // function to be bound to the Disassemble button in the popup
    $scope.disassemble = function () {
      //
      // set an object that targets the model and its instruction property
      var modelObject = { model: targetName, instruction: 'l-Creo 3D - ' +
instructionName + '.pvi' };
      //
      // set the sequence for the quadcopter to be the name of the associated instruction
      $scope.view.wdg.quadcopter.sequence = modelObject.instruction
    } //disassemble function end
```



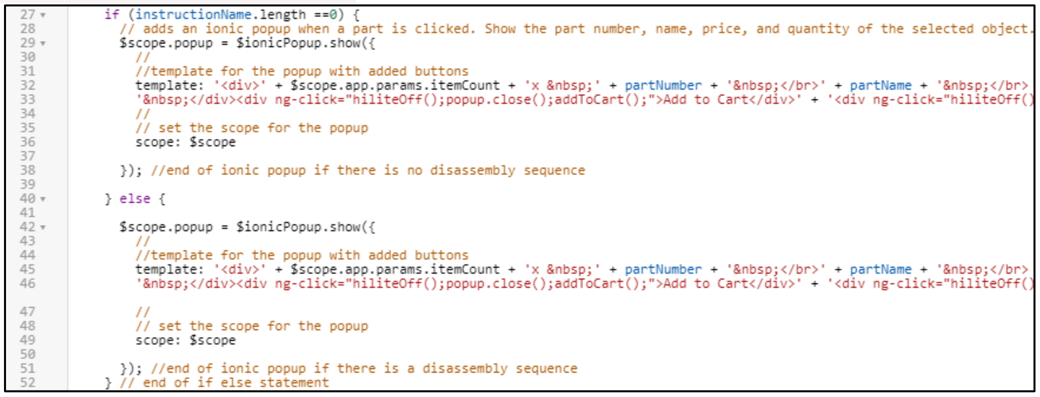
```
    }; // end of hiliteOff function

    // function to be bound to the Disassemble button in the popup
    $scope.disassemble = function () {
      //
      // set an object that targets the model and its instruction property
      var modelObject = { model: targetName, instruction: 'l-Creo 3D - ' + instructionName + '.pvi' };
      //
      // set the sequence for the quadcopter to be the name of the associated instruction
      $scope.view.wdg.quadcopter.sequence = modelObject.instruction
    } //disassemble function end
```

f. The completed code for this section is available in Appendix 3.

301.7 Finalizing the UI

1. Now that functions have been created, they will be added to the `template` property of the popup to make them interactive. The **Add to Cart** button will have the `addToCart` function attached to it and the `disassemble` function will be attached to **Disassemble** button. The `.close` method and `hiliteOff` functions are applied to each button to close the popup and remove the shader once it has been clicked.

```
if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price, 
and quantity of the selected object. &nbsp;</br> adds a line break between the two 
variables
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + 
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price + 
        '&nbsp;</div><div ng-click="hiliteOff();popup.close();addToCart();">Add to 
Cart</div>' + '<div ng-click="hiliteOff();popup.close()">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + 
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price + 
        '&nbsp;</div><div ng-click="hiliteOff();popup.close();addToCart();">Add to 
Cart</div>' + '<div ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' + '<div ng-
click="hiliteOff();popup.close();">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement
```



```
      if (instructionName.length ==0) {
        // adds an ionic popup when a part is clicked. Show the part number, name, price, and quantity of the selected object.
        $scope.popup = $ionicPopup.show({
          //
          //template for the popup with added buttons
          template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br>
          '&nbsp;</div><div ng-click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div ng-click="hiliteOff()
          //
          // set the scope for the popup
          scope: $scope

        }); //end of ionic popup if there is no disassembly sequence

      } else {

        $scope.popup = $ionicPopup.show({
          //
          //template for the popup with added buttons
          template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br>
          '&nbsp;</div><div ng-click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div ng-click="hiliteOff()

          //
          // set the scope for the popup
          scope: $scope

        }); //end of ionic popup if there is a disassembly sequence
      } // end of if else statement
```

2. Open the **Application** tab under **Styles**. This is where the CSS code for styling the popup will be added.

a. Add the code below to style the **Add to Cart** button. This will be called in the popup.

```
.btnadd {
  background: #3498db;
  background-image: -webkit-linear-gradient(top, #3498db, #2980b9);
  background-image: -moz-linear-gradient(top, #3498db, #2980b9);
  background-image: -ms-linear-gradient(top, #3498db, #2980b9);
  background-image: -o-linear-gradient(top, #3498db, #2980b9);
  background-image: linear-gradient(to bottom, #3498db, #2980b9);
  -webkit-border-radius: 10;
  -moz-border-radius: 10;
  border-radius: 10px;
  font-family: Arial;
  color: #ffffff;
  font-size: 14px;
  padding: 5px 10px 5px 10px;
  text-decoration: none;
}
```

b. Style the **Continue** button. This will be called within the popup.

```
.btncontinue {
  background: #e0e0e0;
  background-image: -webkit-linear-gradient(top, #e0e0e0, #6b6b6b);
  background-image: -moz-linear-gradient(top, #e0e0e0, #6b6b6b);
  background-image: -ms-linear-gradient(top, #e0e0e0, #6b6b6b);
  background-image: -o-linear-gradient(top, #e0e0e0, #6b6b6b);
  background-image: linear-gradient(to bottom, #e0e0e0, #6b6b6b);
  -webkit-border-radius: 10;
  -moz-border-radius: 10;
  border-radius: 10px;
  font-family: Arial;
  color: #ffffff;
  font-size: 14px;
  padding: 5px 10px 5px 10px;
  text-decoration: none;
}
```

c. Finally, style the **Disassemble** button. This will be called within the popup.

```
.btndisassemble {
  background: #91a3b0;
  background-image: -webkit-linear-gradient(top, #91a3b0, #6b6b6b);
  background-image: -moz-linear-gradient(top, #91a3b0, #6b6b6b);
  background-image: -ms-linear-gradient(top, #91a3b0, #6b6b6b);
  background-image: -o-linear-gradient(top, #91a3b0, #6b6b6b);
  background-image: linear-gradient(to bottom, #91a3b0, #6b6b6b);
  -webkit-border-radius: 10;
  -moz-border-radius: 10;
  border-radius: 10px;
  font-family: Arial;
  color: #ffffff;
  font-size: 14px;
  padding: 5px 10px 5px 10px;
  text-decoration: none;
}
```

3. To add the CSS styling to the popup, edit the code for the popup to be like the one below. The **Add to Cart** button will be styled using the `btnadd` styling, the

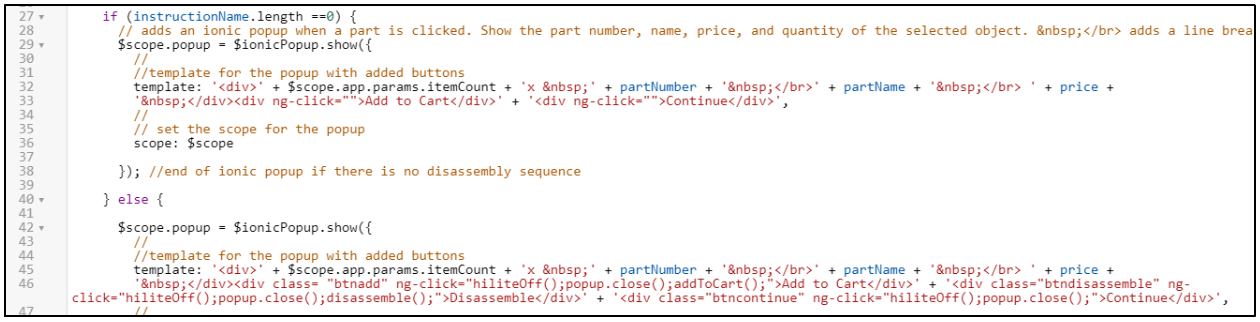
Continue button will use the `btncontinue` styling, and the **Disassemble** button will use the `btndisassemble` styling.

a. Use the code below for the `if` portion of the statement

```
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div class="btnadd" ng-
click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div
class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
```

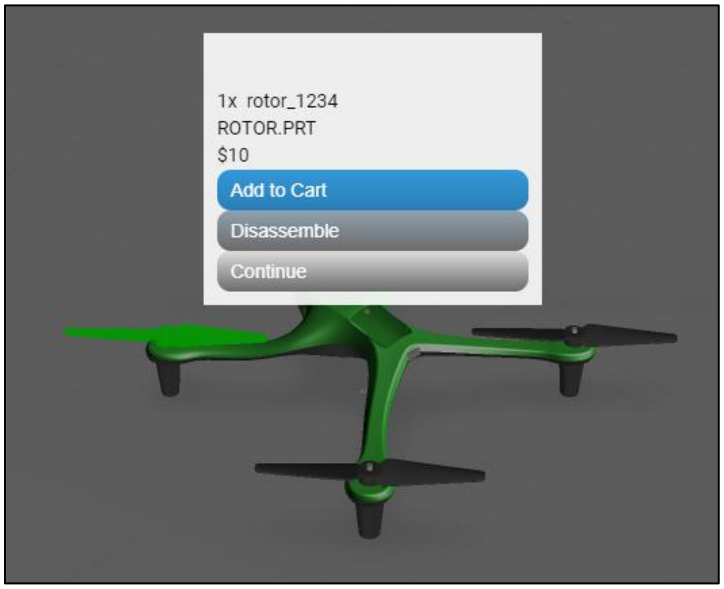
b. Use the code below for the `else` portion of the statement

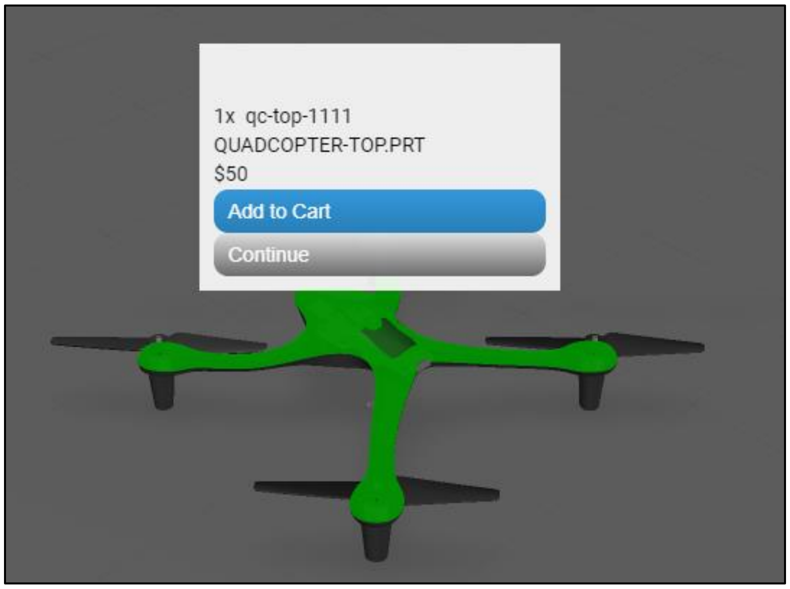
```
template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>'
+ partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div class= "btnadd" ng-
click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div
class="btndisassemble" ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' + '<div
class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
```



```
    if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price, and quantity of the selected object. &nbsp;</br> adds a line brea
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber + '&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div class= "btnadd" ng-click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div class="btndisassemble" ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' + '<div class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
        //
```

4. Open **Preview** to see the result of adding CSS styling to create buttons inside the popup

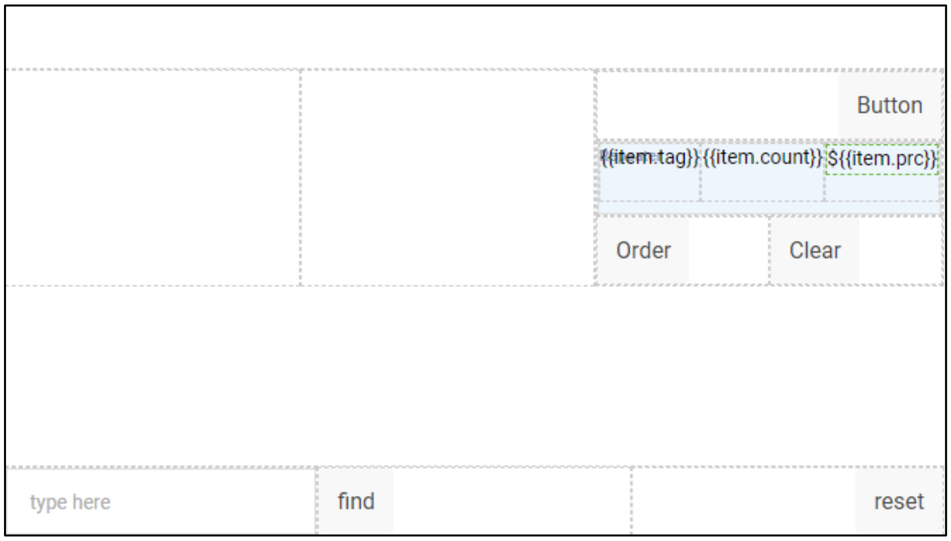




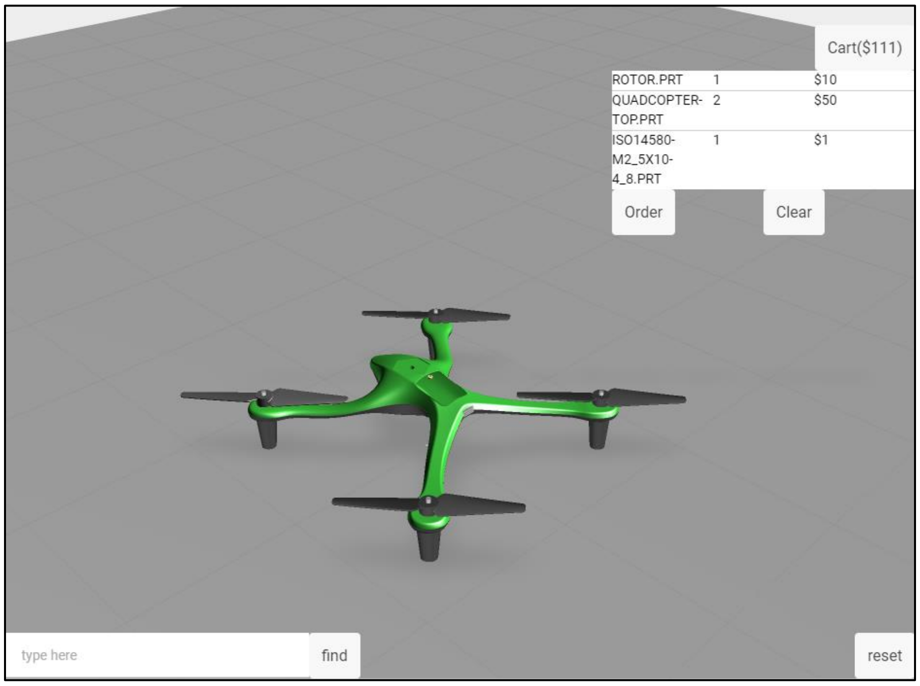
5. The last portion of this project is to add the cart information to the **cartRepeater** that was created earlier. To do this, navigate back to the **Home** tab. Edit the **Text** properties of the three labels that were added earlier to be as follows:
   a. *{{item.tag}}*
   b. *{{item.count}}*
   c. *${{item.prc}}*

   This will call upon the object that was created for the cart in the section of code shown below for the `cartItem` variable, which was later set to be equal to the **cart** application parameter. The labels tie the repeater back to these properties of **cart**. These will list the name, quantity, and price of each part that is selected.

```
if (cartItem === undefined) {
       cartItem = { count: $scope.app.params.itemCount, itm:
$scope.app.params.itemNumber, tag: $scope.app.params.itemName, prc:
$scope.app.params.priceInfo }
  } else {
    cartItem.count += $scope.app.params.itemCount}
```



6. Open the **Experiences** tab and change the value in the **ThingMark Association** dropdown to be **None** since a spatial target is being used for this experience. Open **Preview** to view the experience. Click on a few of the parts so that the popup opens. Click **Add to Cart** to add a part to the cart. Notice that a list of the selected parts appears. This is because of the repeater, which allows the list to grow while being displayed. If everything appears like the image below, then this project has been completed successfully. All code up to this section will be in Appendix 4.



ROTOR.PRT	1	$10
QUADCOPTER-TOP.PRT	2	$50
ISO14580-M2_5X10-4_8.PRT	1	$1

Appendix 1: Section 301.3 Code

```
// $scope, $element, $attrs, $injector, $sce, $timeout, $http, $ionicPopup, and
$ionicpopup services are available
```

```
$scope.$on('userpick', function (event, targetName, targetType, eventData) {
  //
  //Look at model and see if it has metadata. If it does, then execute the below code and
create an object called metadata
  PTC.Metadata.fromId(targetName)
      .then((metadata) => {
    //
    // variable to pull the value for the occurrence property in the eventData JSON
object from the model
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId
    // create variables based on attribute names from Creo Illustrate for this model. use
metadata.get to obtain the data from the JSON properties for this occurrence.
    var partName = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber = metadata.get(pathId, 'partNumber');
    var priceString = metadata.get(pathId, 'listPrice');

    //listPrice is obtained as a string. to change the string into an float, use
parseFloat
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";

    /* set itemName app parameter to be equal to the partName variable, same relationship
with itemNumber and partNumber and priceInfo and price. Set the itemCount to 1 for the
purpose of this section, since it is not hooked up to an             actual inventory*/
    $scope.app.params.itemName = partName;
    $scope.app.params.itemNumber = partNumber;
    $scope.app.params.priceInfo = parseFloat(priceString);
    $scope.app.params.itemCount = 1;

    if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price,
and quantity of the selected object. &nbsp;</br> adds a line break between the two
variables
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Disassemble</div>' + '<div ng-click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement

    //highlight the chosen item and set the shader to true
    $scope.hilite([targetName + "-" + pathId], true);

    // create a function to close the popup and turn off shading. popup is the popup,
refitems is the input for the part(s) that is being highlighted
    var closePopup = function (popup, refitems) {
        //
        //The function returns a method for removing the popup from the screen and turns
off the shader
        return function () {
          //
          //using the input parts, set the hilite function to be false, removing the
shading
          $scope.hilite(refitems, false)
          //
          //apply the .close method, which removes a certain section of a selected
object, to the popup variable
          popup.close()
          //
          //change the Text property of the playButton to the instructionName variable,
which was created from the JSON data of the model
          $scope.view.wdg.playButton.text = instructionName;
          //
          /* create an object for the playButton called toPlay. This object will have
properties of model, which will be the name of the object that is clicked on and
instruction,
          which will add the proper syntax for calling a sequence, based off the
instructionName variable, into Studio*/
          $scope.view.wdg.playButton.toPlay = { model: targetName, instruction: 'l-Creo
3D - ' + instructionName + '.pvi' };
          //
        } //return end
    } // closepopup function end
    //call the $timeout service which will call the function for closing the popup and
removing the shader after 3 seconds (3000 ms)
    $timeout(closePopup(popup, [targetName + "-" + pathId]), 3000);

  }) //end brackets for PTC API and .then
  //
  //catch statement if the promise of having a part with metadata is not met
  .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })

}) //end brackets for userpick function. Will continue to move throughout code

//function for using the userInput text box to search for parts
$scope.findMeta = function () {

  //reset the text property of the play button to be blank
  $scope.view.wdg.playButton.text = '';
  //
  //set the toPlay object for the play button to be undefined
  $scope.view.wdg.playButton.toPlay = undefined;
  //
  //set a variable for comparing the user input to the value of the partno application
parameter
  var searchNum = $scope.app.params.partno;

  //
  // instead of using metadata from just the picked part, use metadata from the whole
model. If resolved, proceed
  PTC.Metadata.fromId('quadcopter')
    .then((metadata) => {
        //
        // set a variable named options. this variable will become an array of ID paths
that fit the input text.
        // 'like' will look for a partial text match to what is typed in. use 'same' to
get an exact match
        var options = metadata.find('partNumber').like(searchNum).getSelected();

        //
        // if the text input leads to a part number so that there is an entry in the
options array
        if (options != undefined && options.length > 0) {
            //
            // set an empty array called ID. This array will house the parts that contain
the entered part number
            var identifiers = []
            //
            // for each entry in the options array, push that value with 'quadcopter-' at
the beginning into the ID array
            options.forEach(function (i) {
                identifiers.push('quadcopter-' + i)
            }) //end forEach

            //
            // highlight each object in the identifiers array with the shader
            $scope.hilite(identifiers, true)

            //
            // function for removing the highlight
            var removeHilite = function (refitems) {
                //
                // return the hilite function with a value of false to the given part(s)
                return function () {
                    $scope.hilite(refitems, false)
                } // end of return function
            } // end of turning off hilite

            //
            // remove the highlight of the selected part(s) after 3000 ms
            $timeout(removeHilite(identifiers), 3000)

          } //end if statement
    }) // end .then

      //catch statement if the promise of having a part with metadata is not met
      .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })
} // end findMeta function
```

```
//create the playit function to bind a sequence for the model to the play button
$scope.playit = function () {
    //
    // if there is information in the created toPlay object to say that there is an
illustration attribute for the part
    if ($scope.view.wdg.playButton.toPlay != undefined)
        //
        // set the sequence property for the quadcopter model to be equal to the value of
the instruction property of the toPlay object
        $scope.view.wdg.quadcopter.sequence =
$scope.view.wdg.playButton.toPlay.instruction;
}// playit function end

//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function (event) {
    //
    // call a widget service to trigger the quadcopter model to play all steps for the
given sequence
    twx.app.fn.triggerWidgetService('quadcopter', 'playAll');
}); //serviceloaded event function end

//resetit function
$scope.resetit = function () {
    //
    //set the sequence property of the quadcopter model to blank
    $scope.view.wdg.quadcopter.sequence = ''
}//resetit function end

//highlight function. Inputs are the selected part and a boolean for hilite
$scope.hilite = function (items, hilite) {
    //
    //iterate over each item that is used as an imported variable for the function using
.forEach to look at each value that comes in the items input
    items.forEach(function (item) {
        //
        //set the properties of the TML 3D Renderer to highlight the selected item using
a TML Text shader. "green" is the name of the script for the TML Text.
        tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden:
false, opacity: 0.9, phantom: false, decal: true }
            : { shader: "Default", hidden: false, opacity: 1.0, phantom: false, decal:
false });
    }) //foreach function end
} //hilite function end
```

Appendix 2: Section 301.4 Code

```
// $scope, $element, $attrs, $injector, $sce, $timeout, $http, $ionicPopup, and
$ionicpopup services are available

$scope.$on('userpick', function (event, targetName, targetType, eventData) {
  //
  //Look at model and see if it has metadata. If it does, then execute the below code and
create an object called metadata
  PTC.Metadata.fromId(targetName)
      .then((metadata) => {
    //
    // variable to pull the value for the occurrence property in the eventData JSON
object from the model
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId
    // create variables based on attribute names from Creo Illustrate for this model. use
metadata.get to obtain the data from the JSON properties for this occurrence.
    var partName = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber = metadata.get(pathId, 'partNumber');
    var priceString = metadata.get(pathId, 'listPrice');

    //listPrice is obtained as a string. to change the string into an float, use
parseFloat
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";

    /* set itemName app parameter to be equal to the partName variable, same relationship
with itemNumber and partNumber and priceInfo and price. Set the itemCount to 1 for the
purpose of this section, since it is not hooked up to an             actual inventory*/
    $scope.app.params.itemName = partName;
    $scope.app.params.itemNumber = partNumber;
    $scope.app.params.priceInfo = parseFloat(priceString);
    $scope.app.params.itemCount = 1;

    if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price,
and quantity of the selected object. &nbsp;</br> adds a line break between the two
variables
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Disassemble</div>' + '<div ng-click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement

    //highlight the chosen item and set the shader to true
    $scope.hilite([targetName + "-" + pathId], true);

  }) //end brackets for PTC API and .then
  //
  //catch statement if the promise of having a part with metadata is not met
  .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })

}) //end brackets for userpick function. Will continue to move throughout code

//function for using the userInput text box to search for parts
$scope.findMeta = function () {
  //
  //set a variable for comparing the user input to the value of the partno application
parameter
  var searchNum = $scope.app.params.partno;

  //
  // instead of using metadata from just the picked part, use metadata from the whole
model. If resolved, proceed
  PTC.Metadata.fromId('quadcopter')
    .then((metadata) => {
        //
        // set a variable named options. this variable will become an array of ID paths
that fit the input text.
        // 'like' will look for a partial text match to what is typed in. use 'same' to
get an exact match
        var options = metadata.find('partNumber').like(searchNum).getSelected();

        //
        // if the text input leads to a part number so that there is an entry in the
options array
        if (options != undefined && options.length > 0) {
            //
            // set an empty array called ID. This array will house the parts that contain
the entered part number
            var identifiers = []
            //
            // for each entry in the options array, push that value with 'quadcopter-' at
the beginning into the ID array
            options.forEach(function (i) {
                identifiers.push('quadcopter-' + i)
            }) //end forEach

            //
            // highlight each object in the identifiers array with the shader
            $scope.hilite(identifiers, true)

            //
            // function for removing the highlight
            var removeHilite = function (refitems) {
                //
                // return the hilite function with a value of false to the given part(s)
                return function () {
                    $scope.hilite(refitems, false)
                } // end of return function
            } // end of turning off hilite

            //
            // remove the highlight of the selected part(s) after 3000 ms
            $timeout(removeHilite(identifiers), 3000)

          } //end if statement
    }) // end .then

      //catch statement if the promise of having a part with metadata is not met
      .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })
} // end findMeta function

//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function (event) {
    //
    // call a widget service to trigger the quadcopter model to play all steps for the
given sequence
    twx.app.fn.triggerWidgetService('quadcopter', 'playAll');
}); //serviceloaded event function end

//resetit function
$scope.resetit = function () {
    //
    //set the sequence property of the quadcopter model to blank
    $scope.view.wdg.quadcopter.sequence = ''
}//resetit function end

//highlight function. Inputs are the selected part and a boolean for hilite
$scope.hilite = function (items, hilite) {
    //
    //iterate over each item that is used as an imported variable for the function using
.forEach to look at each value that comes in the items input
    items.forEach(function (item) {
        //
        //set the properties of the TML 3D Renderer to highlight the selected item using
a TML Text shader. "green" is the name of the script for the TML Text.
        tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden:
false, opacity: 0.9, phantom: false, decal: true }
            : { shader: "Default", hidden: false, opacity: 1.0, phantom: false, decal:
false });
    }) //foreach function end
} //hilite function end
```

Appendix 3: Section 301.6 Code

```
// $scope, $element, $attrs, $injector, $sce, $timeout, $http, $ionicPopup, and
$ionicpopup services are available

$scope.$on('userpick', function (event, targetName, targetType, eventData) {
  //
  //Look at model and see if it has metadata. If it does, then execute the below code and
create an object called metadata
  PTC.Metadata.fromId(targetName)
      .then((metadata) => {
    //
    // variable to pull the value for the occurrence property in the eventData JSON
object from the model
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId
    // create variables based on attribute names from Creo Illustrate for this model. use
metadata.get to obtain the data from the JSON properties for this occurrence.
    var partName = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber = metadata.get(pathId, 'partNumber');
    var priceString = metadata.get(pathId, 'listPrice');

    //listPrice is obtained as a string. to change the string into an float, use
parseFloat
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";

    /* set itemName app parameter to be equal to the partName variable, same relationship
with itemNumber and partNumber and priceInfo and price. Set the itemCount to 1 for the
purpose of this section, since it is not hooked up to an             actual inventory*/
    $scope.app.params.itemName = partName;
    $scope.app.params.itemNumber = partNumber;
    $scope.app.params.priceInfo = parseFloat(priceString);
    $scope.app.params.itemCount = 1;

    if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price,
and quantity of the selected object. &nbsp;</br> adds a line break between the two
variables
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div ng-click="">Add to Cart</div>' + '<div ng-
click="">Disassemble</div>' + '<div ng-click="">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement

    //highlight the chosen item and set the shader to true
    $scope.hilite([targetName + "-" + pathId], true);

    //function for removing the highlight
    $scope.hiliteOff = function() {
      $scope.hilite([targetName + "-" + pathId], false)
    }; // end of hiliteOff function

    // function to be bound to the Disassemble button in the popup
    $scope.disassemble = function () {
      //
      // set an object that targets the model and its instruction property
      var modelObject = { model: targetName, instruction: 'l-Creo 3D - ' +
instructionName + '.pvi' };
      //
      // set the sequence for the quadcopter to be the name of the associated instruction
      $scope.view.wdg.quadcopter.sequence = modelObject.instruction
    } //disassemble function end

  }) //end brackets for PTC API and .then
  //
  //catch statement if the promise of having a part with metadata is not met
  .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })

}) //end brackets for userpick function. Will continue to move throughout code

//function for using the userInput text box to search for parts
$scope.findMeta = function () {
  //
  //set a variable for comparing the user input to the value of the partno application
parameter
  var searchNum = $scope.app.params.partno;

  //
  // instead of using metadata from just the picked part, use metadata from the whole
model. If resolved, proceed
  PTC.Metadata.fromId('quadcopter')
    .then((metadata) => {
        //
        // set a variable named options. this variable will become an array of ID paths
that fit the input text.
        // 'like' will look for a partial text match to what is typed in. use 'same' to
get an exact match
        var options = metadata.find('partNumber').like(searchNum).getSelected();

        //
        // if the text input leads to a part number so that there is an entry in the
options array
        if (options != undefined && options.length > 0) {
            //
            // set an empty array called ID. This array will house the parts that contain
the entered part number
            var identifiers = []
            //
            // for each entry in the options array, push that value with 'quadcopter-' at
the beginning into the ID array
            options.forEach(function (i) {
                identifiers.push('quadcopter-' + i)
            }) //end forEach

            //
            // highlight each object in the identifiers array with the shader
            $scope.hilite(identifiers, true)

            //
            // function for removing the highlight
            var removeHilite = function (refitems) {
                //
                // return the hilite function with a value of false to the given part(s)
                return function () {
                    $scope.hilite(refitems, false)
                } // end of return function
            } // end of turning off hilite

            //
            // remove the highlight of the selected part(s) after 3000 ms
            $timeout(removeHilite(identifiers), 3000)

          } //end if statement
    }) // end .then

      //catch statement if the promise of having a part with metadata is not met
      .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })
} // end findMeta function

//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function (event) {
    //
    // call a widget service to trigger the quadcopter model to play all steps for the
given sequence
    twx.app.fn.triggerWidgetService('quadcopter', 'playAll');
}); //serviceloaded event function end

//resetit function
$scope.resetit = function () {
    //
    //set the sequence property of the quadcopter model to blank
    $scope.view.wdg.quadcopter.sequence = ''
}//resetit function end

//highlight function. Inputs are the selected part and a boolean for hilite
$scope.hilite = function (items, hilite) {
    //
    //iterate over each item that is used as an imported variable for the function using
.forEach to look at each value that comes in the items input
    items.forEach(function (item) {
        //
        //set the properties of the TML 3D Renderer to highlight the selected item using
a TML Text shader. "green" is the name of the script for the TML Text.
        tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden:
false, opacity: 0.9, phantom: false, decal: true }
            : { shader: "Default", hidden: false, opacity: 1.0, phantom: false, decal:
false });
    }) //foreach function end
} //hilite function end

$scope.app.params.cartButton = "Cart"; // set cartButton app parameter to be "Cart". This
will bind to the Text property for the buttonCart button
$scope.cart = {}; // declare empty object called cart

// function for adding a selected part to the cart
$scope.addToCart = function () {
  //
  /* create variable called cartItem that is equal to the value of the currentSelection
property of the cart object. If the selected part hasn't been added to the cart yet, then
the cartItem variable will be undefined and populate the cartItem variable with the
current information about the part so that cartItem becomes an object. If the selected
```

```
part has already been added, then the count property of cartItem will increase by the
item count*/
  //
  var cartItem =$scope.cart[$scope.currentSelection];

  if (cartItem === undefined) {
       cartItem = { count: $scope.app.params.itemCount, itm:
$scope.app.params.itemNumber, tag: $scope.app.params.itemName, prc:
$scope.app.params.priceInfo }
  } else {
    cartItem.count += $scope.app.params.itemCount}

  $scope.cart[$scope.currentSelection] = cartItem;

//cartItemAmount initialized as 0. will be used to count how many items are in the cart
  var cartItemAmount = 0;

  // set an empty array for the cart. this array will have an object pushed into it
  var cartContents = [];

  // initialize variable for keeping track of the price of the objects in the cart
  var cartPrice = 0;

  //loop over each item that is added to the cart
  for (var itm in $scope.cart) {
    //
    //add a number to the counting variable for each item added
    cartItemAmount += $scope.cart[itm].count;
    //
    // add the price of each item to the total price of the cart
    cartPrice = cartPrice += $scope.cart[itm].count*$scope.cart[itm].prc

    //push the name (tag), item count (count), and price (prc) of each part into the
repeater for the cart
    cartContents.push({
        tag: $scope.cart[itm].tag,
        count: $scope.cart[itm].count,
        prc: $scope.cart[itm].prc
    }); // end of the push method for cartContents
  }// for loop end

  // set the app parameter for cart to be equal to the cartContents array
  $scope.app.params.cart = cartContents;

  //setting the cartButton app parameter. if there are items to put into the cart (true),
the text of the cart button should be cart(total cost of cart). If false, just keep the
button text as cart
  $scope.app.params.cartButton = cartItemAmount > 0 ? "Cart($" + cartPrice + ")" :
"Cart";
  //remove the highlight from the part
  $scope.hiliteOff();

} //end of addToCart function

//
// clear the cart. set the part app parameter and cart object to be empty. change the
text on the cart button back to just Cart
$scope.clearCart = function () {
    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartButton = "Cart";
} // end of clearCart function

//function for ordering. Will be populated more when connected to ThingWorx
$scope.orderCart = function () {
  $scope.clearCart();
} // end of orderCart function
```

Appendix 4: Section 301.7 Code

```
// $scope, $element, $attrs, $injector, $sce, $timeout, $http, $ionicPopup, and
$ionicpopup services are available

$scope.$on('userpick', function (event, targetName, targetType, eventData) {
  //
  //Look at model and see if it has metadata. If it does, then execute the below code and
create an object called metadata
  PTC.Metadata.fromId(targetName)
      .then((metadata) => {
    //
    // variable to pull the value for the occurrence property in the eventData JSON
object from the model
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId
    // create variables based on attribute names from Creo Illustrate for this model. use
metadata.get to obtain the data from the JSON properties for this occurrence.
    var partName = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber = metadata.get(pathId, 'partNumber');
    var priceString = metadata.get(pathId, 'listPrice');

    //listPrice is obtained as a string. to change the string into an float, use
parseFloat
    var price = priceString != undefined ? '$' + parseFloat(priceString) : "";

    /* set itemName app parameter to be equal to the partName variable, same relationship
with itemNumber and partNumber and priceInfo and price. Set the itemCount to 1 for the
purpose of this section, since it is not hooked up to an             actual inventory*/
    $scope.app.params.itemName = partName;
    $scope.app.params.itemNumber = partNumber;
    $scope.app.params.priceInfo = parseFloat(priceString);
    $scope.app.params.itemCount = 1;

if (instructionName.length ==0) {
      // adds an ionic popup when a part is clicked. Show the part number, name, price,
and quantity of the selected object. &nbsp;</br> adds a line break between the two
variables
      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div class="btnadd" ng-
click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div
class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is no disassembly sequence

    } else {

      $scope.popup = $ionicPopup.show({
        //
        //template for the popup with added buttons
        template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
'&nbsp;</br>' + partName + '&nbsp;</br> ' + price +
        '&nbsp;</div><div class= "btnadd" ng-
click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' + '<div
class="btndisassemble" ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' + '<div
class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
        //
        // set the scope for the popup
        scope: $scope

      }); //end of ionic popup if there is a disassembly sequence
    } // end of if else statement

    //highlight the chosen item and set the shader to true
    $scope.hilite([targetName + "-" + pathId], true);

    //function for removing the highlight
    $scope.hiliteOff = function() {
      $scope.hilite([targetName + "-" + pathId], false)
    }; // end of hiliteOff function

    // function to be bound to the Disassemble button in the popup
    $scope.disassemble = function () {
      //
      // set an object that targets the model and its instruction property
      var modelObject = { model: targetName, instruction: 'l-Creo 3D - ' +
instructionName + '.pvi' };
      //
      // set the sequence for the quadcopter to be the name of the associated instruction
      $scope.view.wdg.quadcopter.sequence = modelObject.instruction
    } //disassemble function end

  }) //end brackets for PTC API and .then
  //
  //catch statement if the promise of having a part with metadata is not met
  .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })

}) //end brackets for userpick function. Will continue to move throughout code

//function for using the userInput text box to search for parts
$scope.findMeta = function () {
  //
  //set a variable for comparing the user input to the value of the partno application
parameter
  var searchNum = $scope.app.params.partno;

  //
  // instead of using metadata from just the picked part, use metadata from the whole
model. If resolved, proceed
  PTC.Metadata.fromId('quadcopter')
    .then((metadata) => {
        //
        // set a variable named options. this variable will become an array of ID paths
that fit the input text.
        // 'like' will look for a partial text match to what is typed in. use 'same' to
get an exact match
        var options = metadata.find('partNumber').like(searchNum).getSelected();

        //
        // if the text input leads to a part number so that there is an entry in the
options array
        if (options != undefined && options.length > 0) {
            //
            // set an empty array called ID. This array will house the parts that contain
the entered part number
            var identifiers = []
            //
            // for each entry in the options array, push that value with 'quadcopter-' at
the beginning into the ID array
            options.forEach(function (i) {
                identifiers.push('quadcopter-' + i)
            }) //end forEach

            //
            // highlight each object in the identifiers array with the shader
            $scope.hilite(identifiers, true)

            //
            // function for removing the highlight
            var removeHilite = function (refitems) {
                //
                // return the hilite function with a value of false to the given part(s)
                return function () {
                    $scope.hilite(refitems, false)
                } // end of return function
            } // end of turning off hilite

            //
            // remove the highlight of the selected part(s) after 3000 ms
            $timeout(removeHilite(identifiers), 3000)

          } //end if statement
    }) // end .then

      //catch statement if the promise of having a part with metadata is not met
      .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })
} // end findMeta function

//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function (event) {
    //
    // call a widget service to trigger the quadcopter model to play all steps for the
given sequence
    twx.app.fn.triggerWidgetService('quadcopter', 'playAll');
}); //serviceloaded event function end
```

```
//resetit function
$scope.resetit = function () {
    //
    //set the sequence property of the quadcopter model to blank
    $scope.view.wdg.quadcopter.sequence = ''
}//resetit function end

//highlight function. Inputs are the selected part and a boolean for hilite
$scope.hilite = function (items, hilite) {
    //
    //iterate over each item that is used as an imported variable for the function using
.forEach to look at each value that comes in the items input
    items.forEach(function (item) {
        //
        //set the properties of the TML 3D Renderer to highlight the selected item using
a TML Text shader. "green" is the name of the script for the TML Text.
        tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden:
false, opacity: 0.9, phantom: false, decal: true }
            : { shader: "Default", hidden: false, opacity: 1.0, phantom: false, decal:
false });
    }) //foreach function end
} //hilite function end

$scope.app.params.cartButton = "Cart"; // set cartButton app parameter to be "Cart". This
will bind to the Text property for the buttonCart button
$scope.cart = {}; // declare empty object called cart

// function for adding a selected part to the cart
$scope.addToCart = function () {
  //
  /* create variable called cartItem that is equal to the value of the currentSelection
property of the cart object. If the selected part hasn't been added to the cart yet, then
the cartItem variable will be undefined and populate the cartItem variable with the
current information about the part so that cartItem becomes an object. If the selected
part has already been added, then the count property of cartItem will increase by the
item count*/
  //
  var cartItem =$scope.cart[$scope.currentSelection];

  if (cartItem === undefined) {
       cartItem = { count: $scope.app.params.itemCount, itm:
$scope.app.params.itemNumber, tag: $scope.app.params.itemName, prc:
$scope.app.params.priceInfo }
  } else {
    cartItem.count += $scope.app.params.itemCount}

  $scope.cart[$scope.currentSelection] = cartItem;

//cartItemAmount initialized as 0. will be used to count how many items are in the cart
  var cartItemAmount = 0;

  // set an empty array for the cart. this array will have an object pushed into it
  var cartContents = [];

  // initialize variable for keeping track of the price of the objects in the cart
  var cartPrice = 0;
  //loop over each item that is added to the cart
  for (var itm in $scope.cart) {
    //
    //add a number to the counting variable for each item added
    cartItemAmount += $scope.cart[itm].count;
    //
    // add the price of each item to the total price of the cart
    cartPrice = cartPrice += $scope.cart[itm].count*$scope.cart[itm].prc

    //push the name (tag), item count (count), and price (prc) of each part into the
repeater for the cart
    cartContents.push({
        tag: $scope.cart[itm].tag,
        count: $scope.cart[itm].count,
        prc: $scope.cart[itm].prc
    }); // end of the push method for cartContents
  }// for loop end

  // set the app parameter for cart to be equal to the cartContents array
  $scope.app.params.cart = cartContents;

  //setting the cartButton app parameter. if there are items to put into the cart (true),
the text of the cart button should be cart(total cost of cart). If false, just keep the
button text as cart
  $scope.app.params.cartButton = cartItemAmount > 0 ? "Cart($" + cartPrice + ")" :
"Cart";
  //remove the highlight from the part
  $scope.hiliteOff();

} //end of addToCart function

//
// clear the cart. set the part app parameter and cart object to be empty. change the
text on the cart button back to just Cart
$scope.clearCart = function () {
    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartButton = "Cart";
} // end of clearCart function

//function for ordering. Will be populated more when connected to ThingWorx
$scope.orderCart = function () {
  $scope.clearCart();
} // end of orderCart function
```