



# vuforia<sup>TM</sup> studio

## **Metadata 301**

**Adding Pricing Data and a  
Shopping Cart to a Model**

**Copyright © 2020 PTC Inc. and/or Its Subsidiary Companies. All Rights Reserved.**

User and training guides and related documentation from PTC Inc. and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes. Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

**UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN  
RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION.**

PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

**Important Copyright, Trademark, Patent, and Licensing Information:**

See the About Box, or copyright notice, of your PTC software.

**UNITED STATES GOVERNMENT RIGHTS**

PTC software products and software documentation are “commercial items” as that term is defined at 48 C.F.

R. 2.101. Pursuant to Federal Acquisition Regulation (FAR) 12.212 (a)-(b) (Computer Software) (MAY 2014) for civilian agencies or the Defense Federal Acquisition Regulation Supplement (DFARS) at 227.7202-1(a) (Policy) and 227.7202-3 (a) (Rights in commercial computer software or commercial computer software documentation) (FEB 2014) for the Department of Defense, PTC software products and software documentation are provided to the U.S. Government under the PTC commercial license agreement. Use, duplication or disclosure by the U.S. Government is subject solely to the terms and conditions set forth in the applicable PTC software license agreement.

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

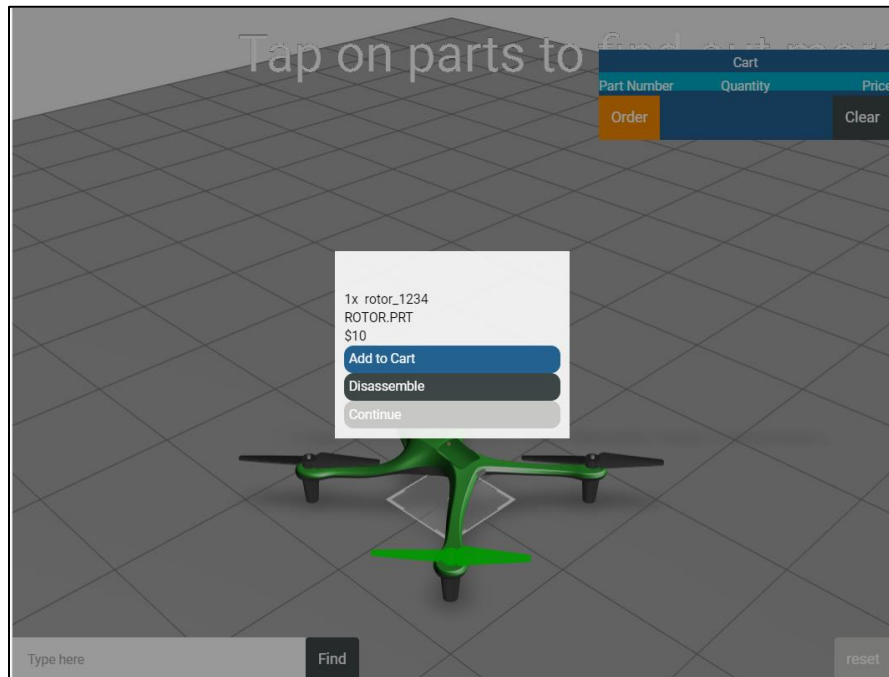
## Prerequisites

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



## 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.

[Metadata 301.1 Bulk Adding Attributes](#)

[Metadata 301.2 Setting Up the Project](#)

[Metadata 301.3 Adding Application Parameters and Editing the Popup with an If Else Statement](#)

[Metadata 301.4 Removing Outdated Functions and Text](#)

[Metadata 301.5 Updating the 2D Interface](#)

[Metadata 301.6 Building the Cart and Popup Functions](#)

[Metadata 301.7 Creating the Popup Buttons](#)

[Metadata 301.8 Styling the Experience](#)

There are also five 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*.

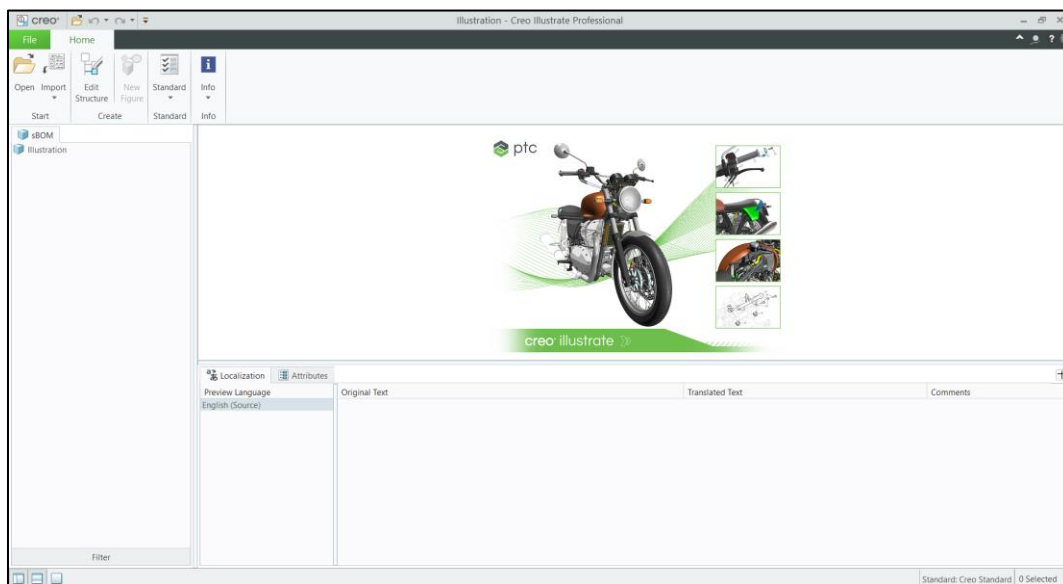
All code follows **this convention**

All code comments follow **this convention**

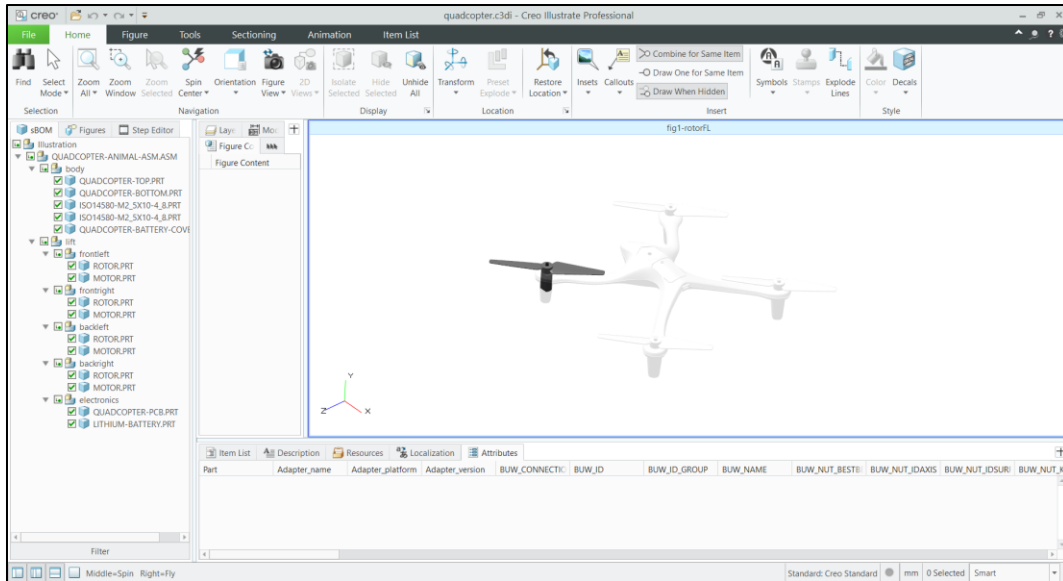
### 301.1 Add Attributes in Bulk

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.

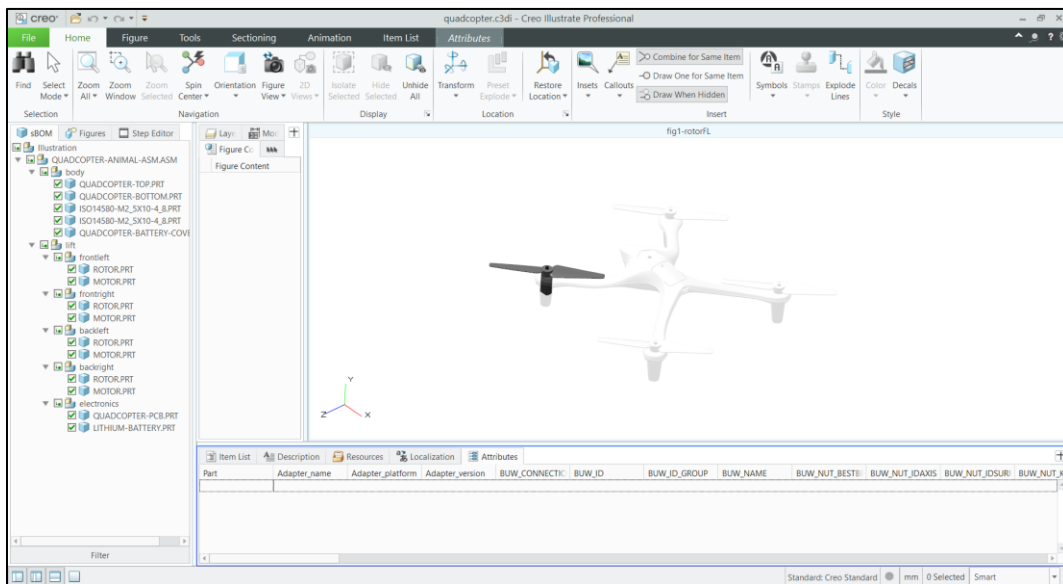
1. Open Creo Illustrate.



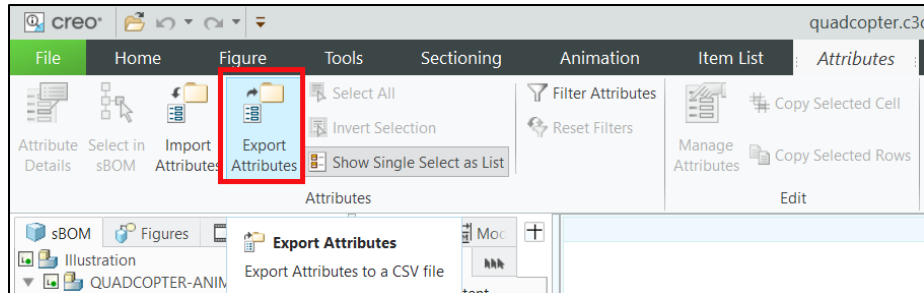
2. From the **File** tab, click **Open** and select the quadcopter.c3di file that was created in **Metadata 101**. If you used the quadcopter101.c3di file that was provided, then open that file.



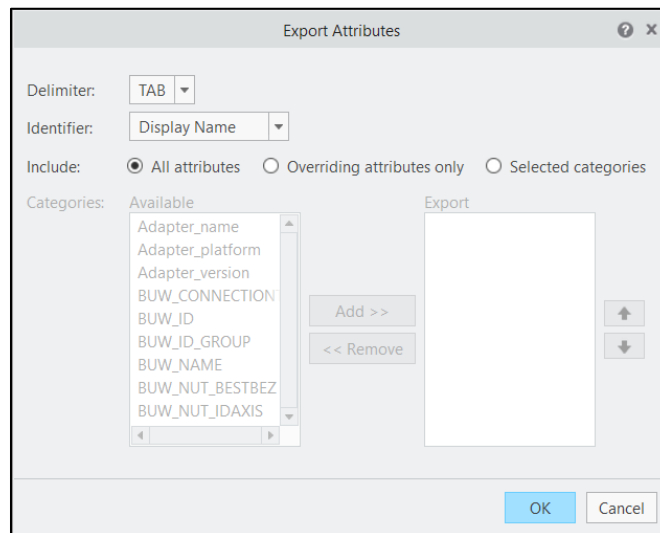
3. Open the Lower Data Panel if it isn't already open. Open the **Attributes** tab to display the attributes for the selected part.



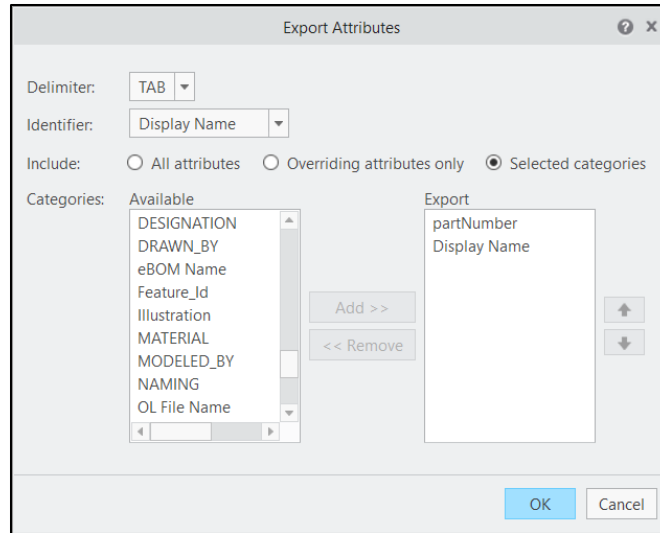
4. When the **Attributes** tab is opened in the Lower Data Panel, another **Attributes** tab opens in the Ribbon. In the **Attributes** tab at the top of your screen, select **Export Attributes** to prepare the attributes for export. In order to edit the attributes for the model in bulk, they must be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



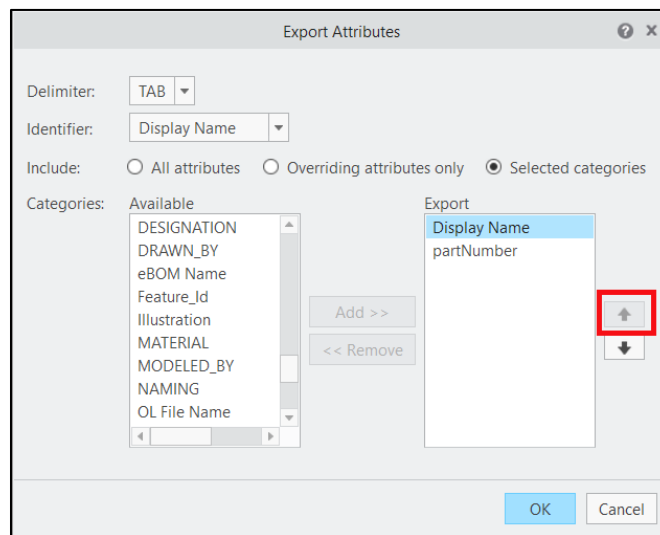
- a. In the **Export Attributes** window, select a **Delimiter** and **Identifier** from their respective dropdowns. A delimiter is the character that separates the values in the list of attributes. In this example, **TAB** is selected as the delimiter. The **Identifier** is used to determine which attribute 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**.



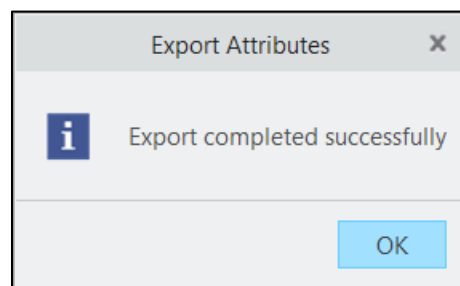
- b. You must add at least two categories of attributes to the **Export** list to make the file readable. For readability sake, we will only add two attributes, as leaving **All attributes** checked would produce an exported .csv file with as many columns as there are attributes. Select the **Selected 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.



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

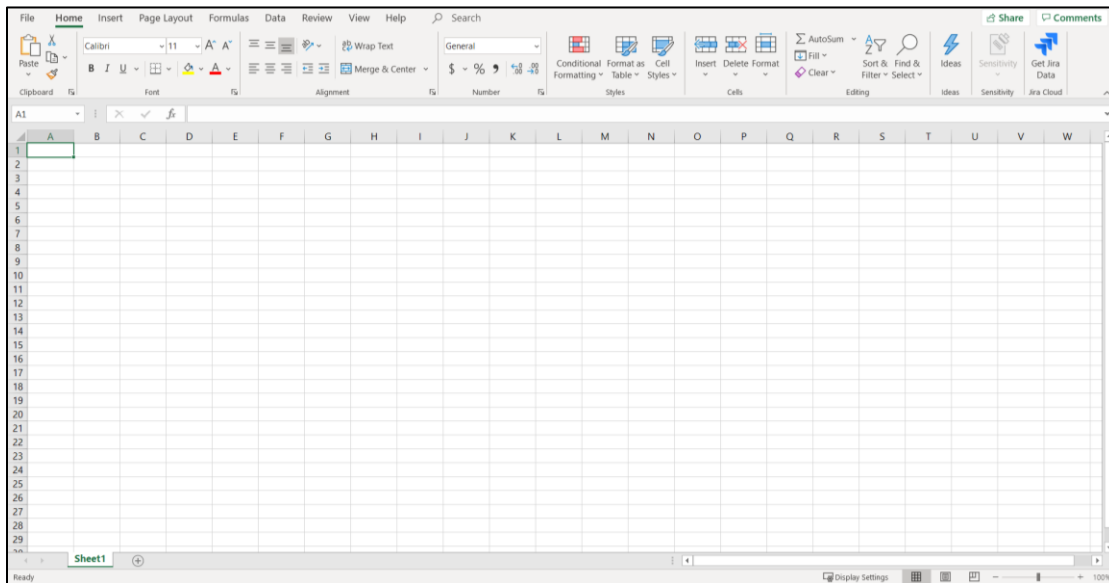


- d. Click **OK**, name the file *quadcopter*, and click **Export** to create a .csv file.
- e. The below message is displayed when the .csv file has been created successfully.

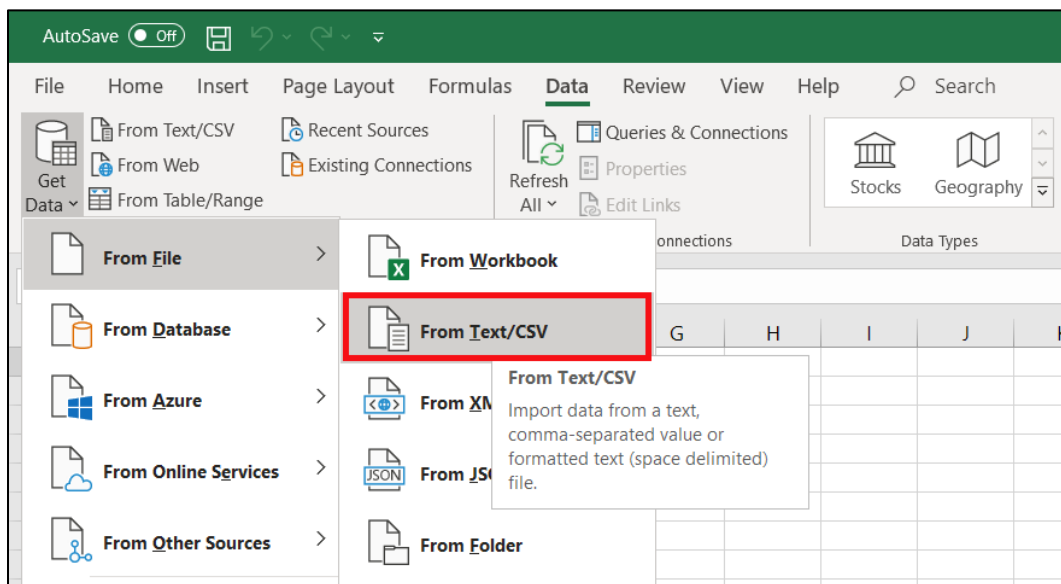




5. Open Microsoft Excel or a similar application. The next few steps will show you how to bulk edit in Excel.
  - a. When Excel opens, select **Blank workbook**..



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



- 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

Load Transform Data Cancel

- 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	C	D	E	F	G	H	I
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							
14									
15									
16									
17									
18									
19									
20									
21									
22									

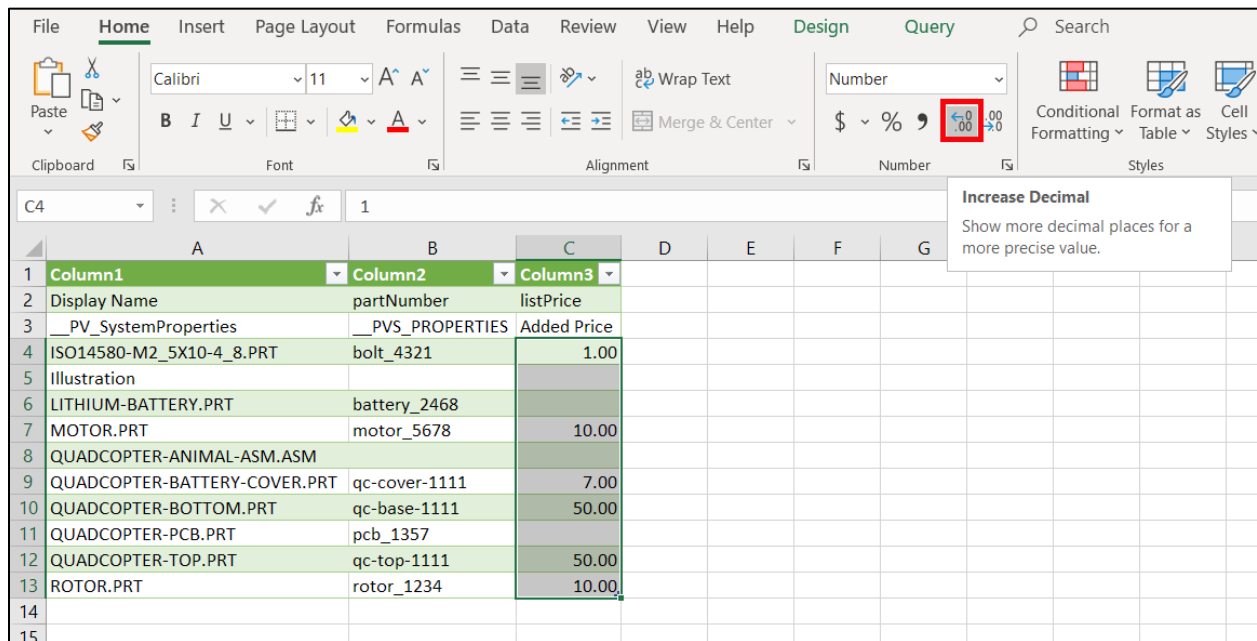
- f. In cell **C2**, type *listPrice*. This creates a new attribute name when imported back into Illustrate. In cell **C3**, type *Added Price*. This will be the new attribute category.

	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				

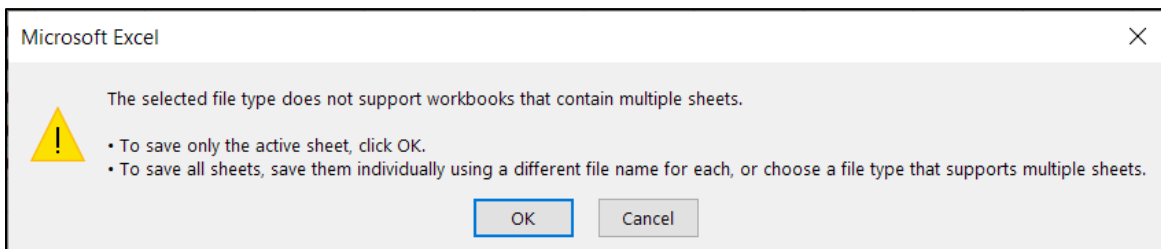
- g. Add the prices displayed below for the given parts. Illustration and QUADCOPTER-ANIMAL-ASM.ASM do not get prices because they are the assembly portion of the model in the model tree, not actual parts. The electronics combined part is also not listed in this because it is two combined parts, so the 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				

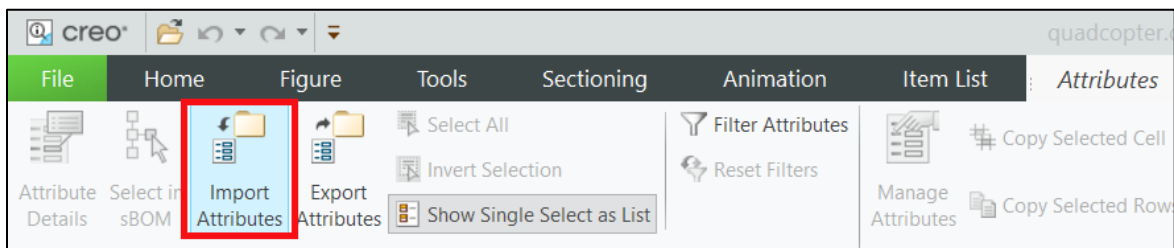
- h. Select all **listPrice** values and use the **Increase Decimal** button to add two decimal points to the end of the values. This will ensure that they correspond with how monetary values are written out.



- 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.



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.

The screenshot shows the 'Import Attributes' dialog box. The 'Header Row' is set to 1. The 'Preview' section shows a table with three columns: Column1, Column2, and Column3. The data rows are as follows:

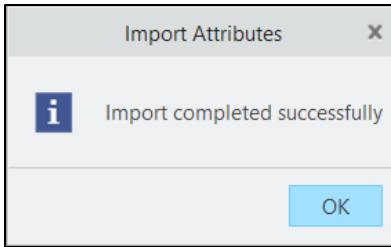
Column1	Column2	Column3
Display Name	partNumber	listPrice
__PV_SystemProperties	__PVS_PROPERTIES	Added Price
ISO14580-M2_5X10	bolt_4321	1.00
Illustration		
LITHIUM-BATTERY	battery_2468	
MOTOR.PRT	motor_5678	10.00
QUADCOPTER-ANI		

- 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.

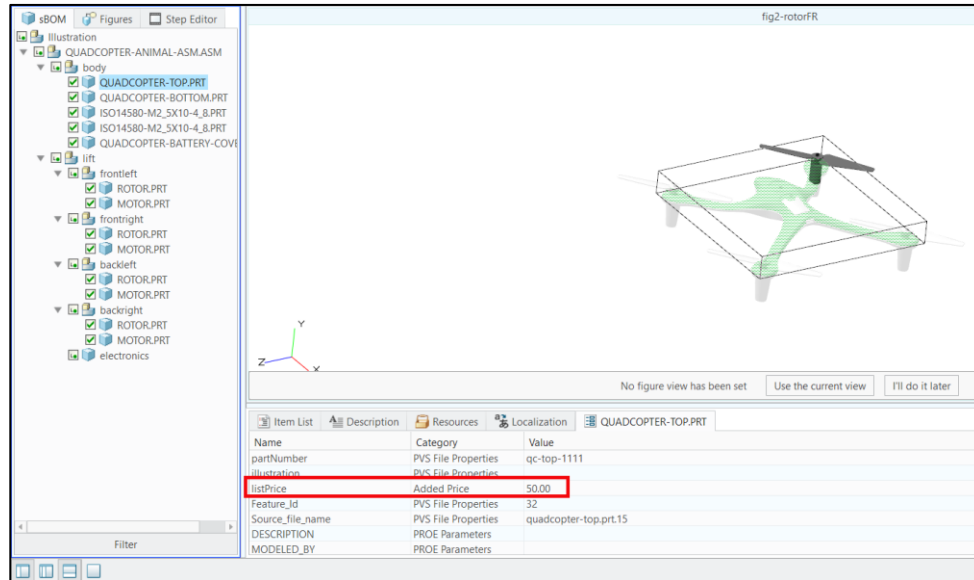
The screenshot shows the 'Import Attributes' dialog box with 'Header Row' set to 2 and 'Identifier' set to 'Display Name'. The 'Preview' section shows a table with three columns: Display Name, partNumber, and listPrice. The data rows are as follows:

Display Name	partNumber	listPrice
__PV_SystemProperties	__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	ac-cover-1111	7.00

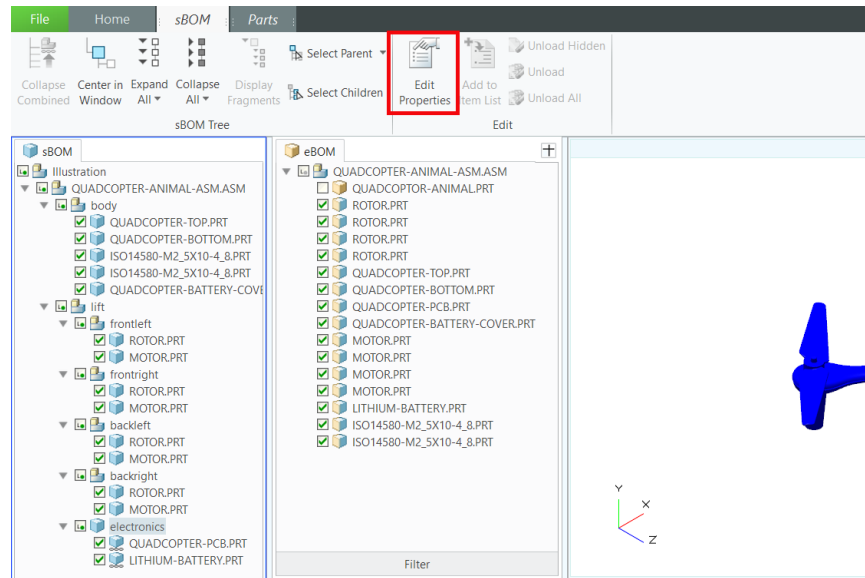
- c. Click **OK** on the success window.



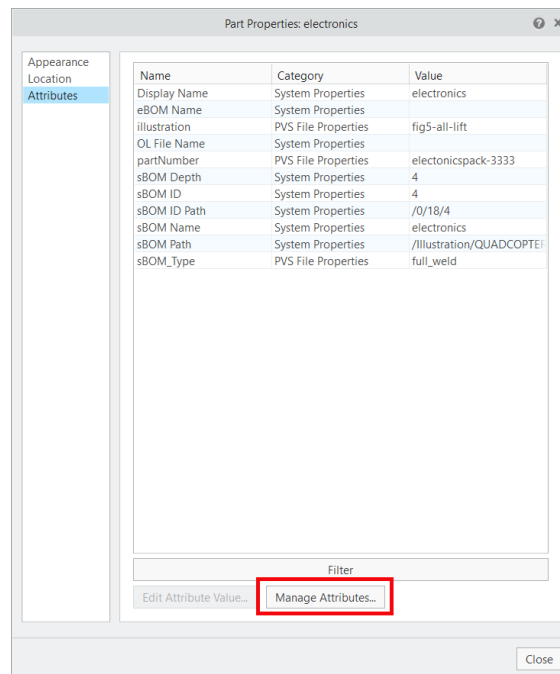
7. Click the parts inside the **sBOM**. The **Attributes** tab in the Lower Data Pane should now have the **listPrice** attribute for each part.



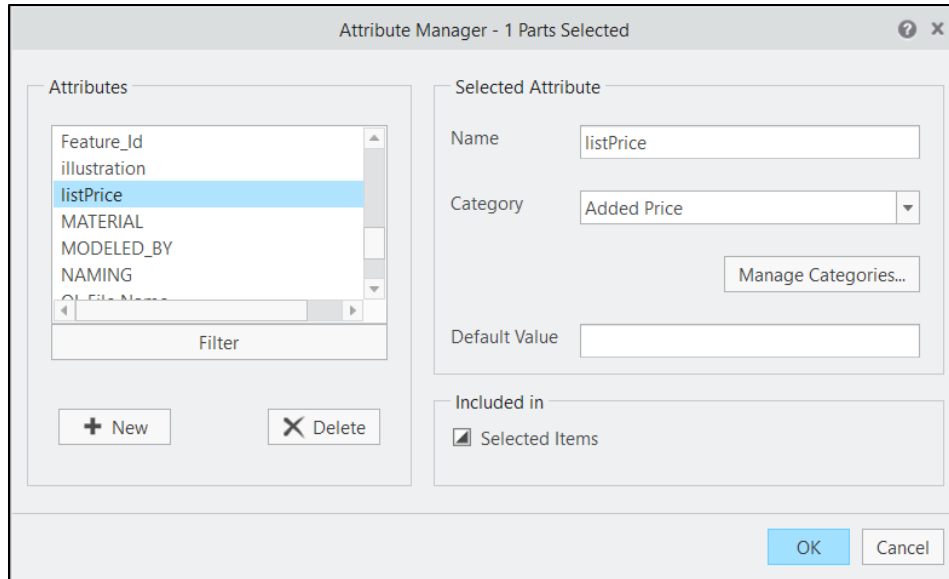
8. As mentioned before, the **electronics** combined part needs to have its price attribute created manually.
  - a. Click the **Edit Structure** button to open the structure editor again. Repeat the same process you used in **Metadata 101** to make the model appear in the **Structure Edit** window.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



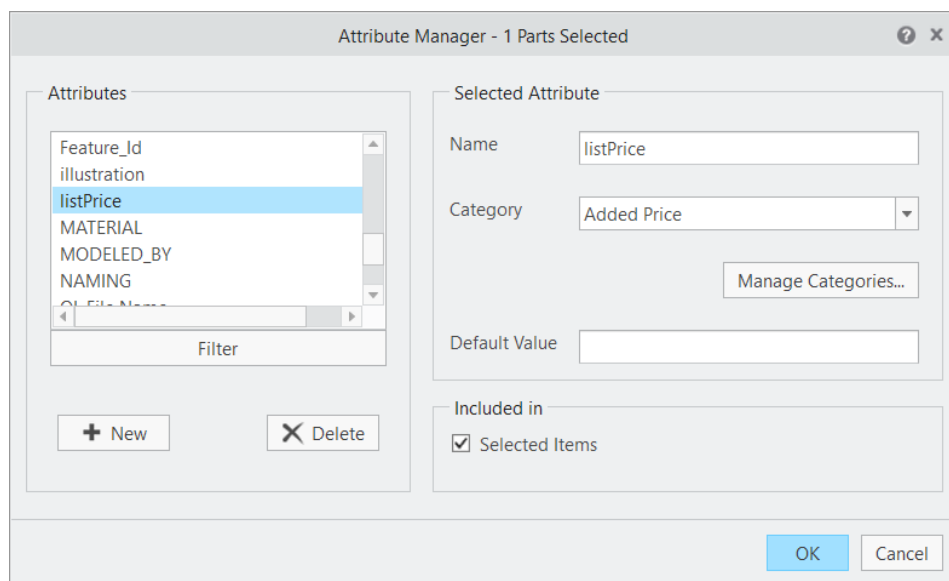
- c. In the **Part Properties** menu, open the **Attributes** tab and then click **Manage Attributes**.



- 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.

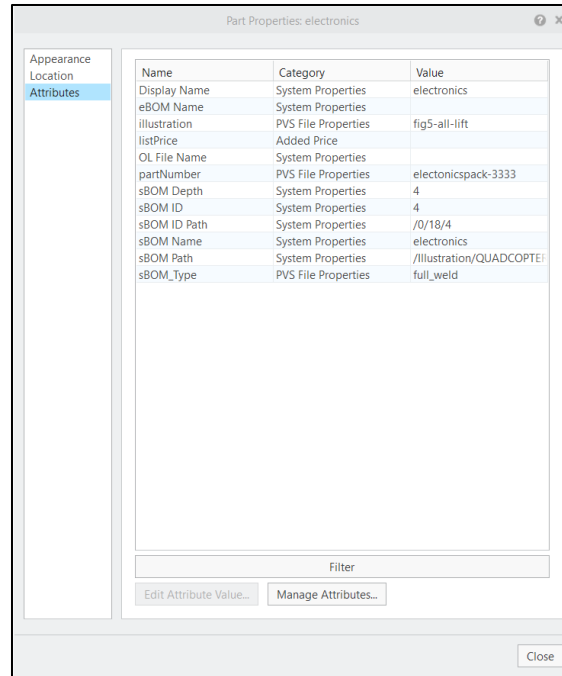


- e. Change **Selected Items** to be checked off as shown below and click **OK**. This will add **listPrice** as an attribute for **electronics**. Click **Cancel** to exit this window.



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





- 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**.

Attribute Detail

Category:  Name:

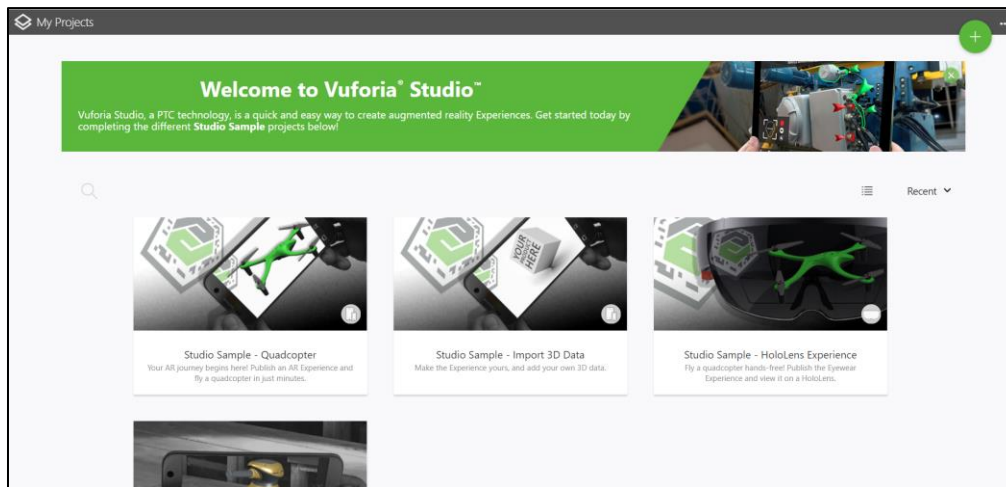
Value:

- 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.

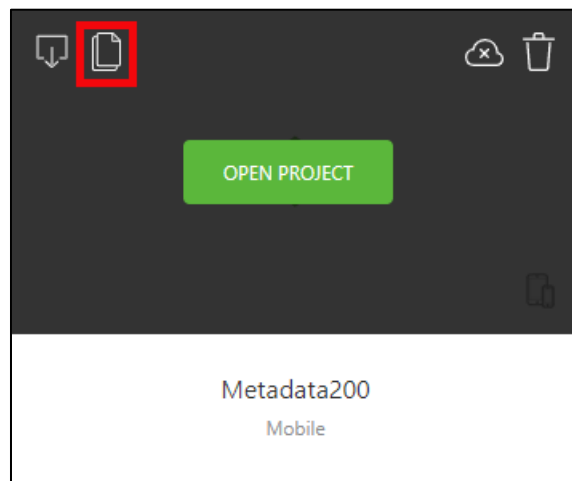
## 301.2 Set Up the Project

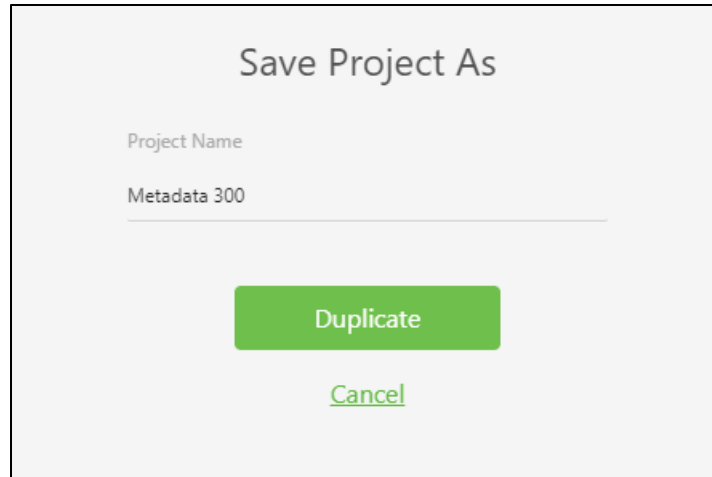
This section will review how to set up the model inside the Vuforia Studio project.

1. Open Vuforia Studio.



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

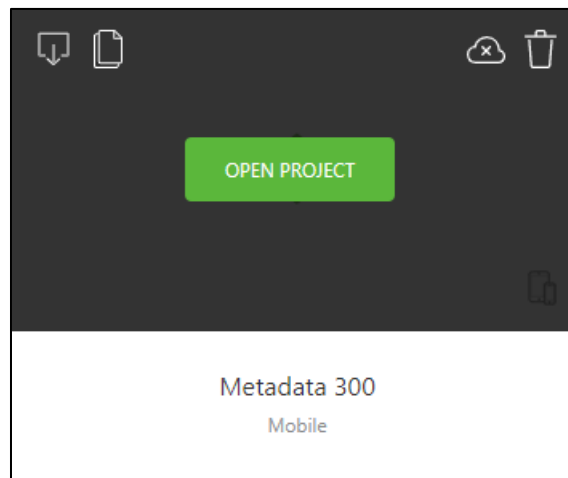
Metadata 300

Duplicate

[Cancel](#)

This is a light gray dialog box titled "Save Project As". It contains a label "Project Name" above a text input field that contains the text "Metadata 300". Below the input field, there is a green button labeled "Duplicate" and a green text link labeled "Cancel".

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



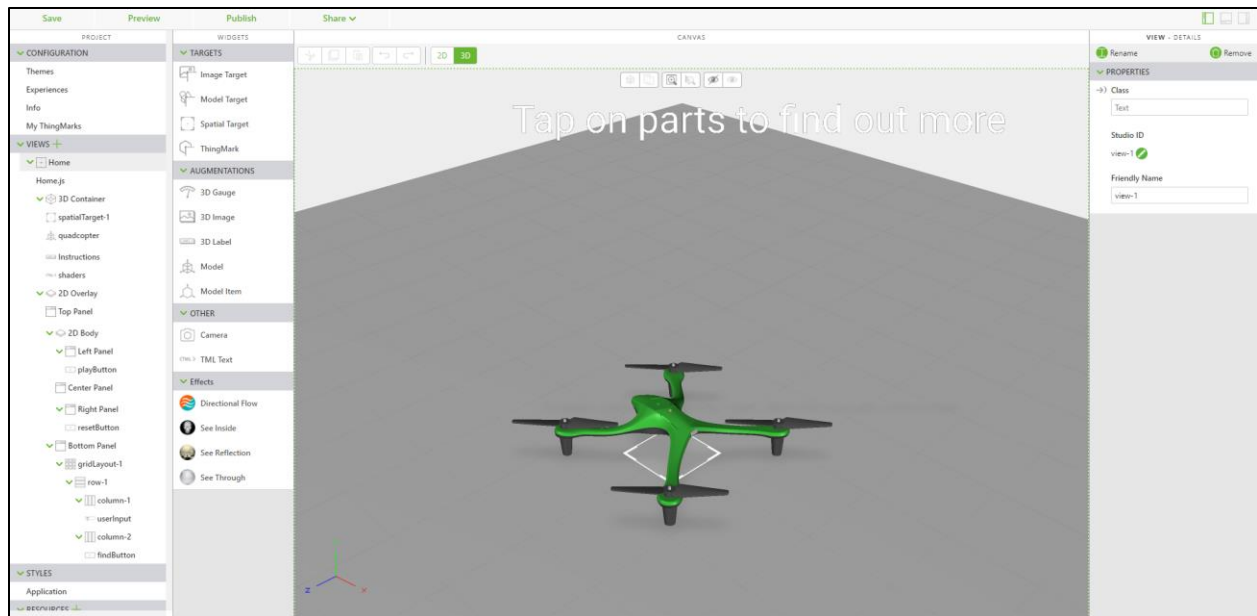
OPEN PROJECT

Metadata 300

Mobile

This is a project card with a dark gray header and a white body. The header contains a green button labeled "OPEN PROJECT". The body contains the text "Metadata 300" and "Mobile" below it. The card also features several icons: a download icon, a document icon, a cloud with an 'x' icon, and a trash icon in the top right corner, and a document icon in the bottom right corner.

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

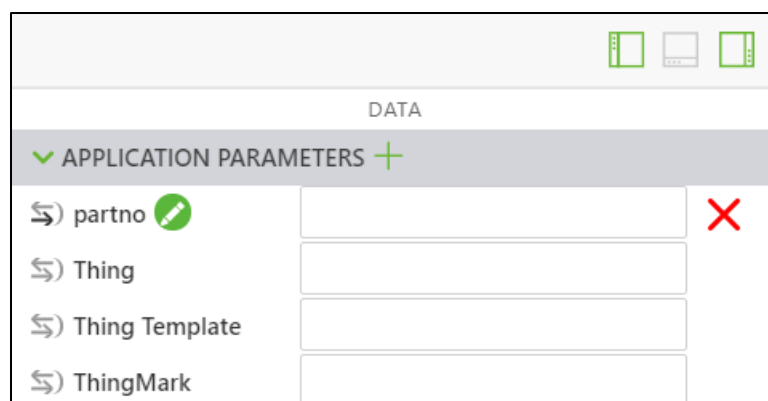


5. The first thing you will update is the quadcopter.pvz model. The new model includes the listPrice attribute that was added in section **301.1**. This model can be added by opening the **quadcopter** model widget and clicking the **+** next to **Resource**. Navigate to and select the updated model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

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.

DATA		
✓ APPLICATION PARAMETERS +		
\$ itemCount	<input type="text"/>	✓ ✗
\$ itemName	<input type="text"/>	✓ ✗
\$ itemNumber	<input type="text"/>	✓ ✗
\$ partno	<input type="text"/>	✓ ✗
\$ priceInfo	<input type="text"/>	✓ ✗
\$ Thing	<input type="text"/>	✓
\$ Thing Template	<input type="text"/>	✓
\$ ThingMark	<input type="text"/>	✓

- Below the `partName`, `instructionName`, and `partNumber` variables in **Home.js**, 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 `priceString` 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 ? '&nbsp;</br>$' + parseFloat(priceString)  
: "";
```

```
22 var priceString = metadata.get(pathId, 'listPrice');  
23  
24 //  
25 //listPrice is obtained as a string. If there is a price for the part, then use parseFloat to turn the string into a f  
26 var price = priceString != undefined ? '&nbsp;</br>$' + parseFloat(priceString)  
27 : "";  
28
```

- Next, below those new variables, 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;
```

```
26     var price = priceString != undefined ? '&nbsp;<br>' + parseFloat(priceString)  
27         : '';  
28  
29  
30     //  
31     // Set the itemCount to 1 for the purpose of this section, since it is not hooked up to an actual inventory.  
32     $scope.app.params.itemName = partName;  
33     $scope.app.params.itemNumber = partNumber;  
34     $scope.app.params.priceInfo = parseFloat(priceString);  
35     $scope.app.params.itemCount = 1;
```

4. All the initial variables and application parameters have been created. Visit [Appendix 1](#) for 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.

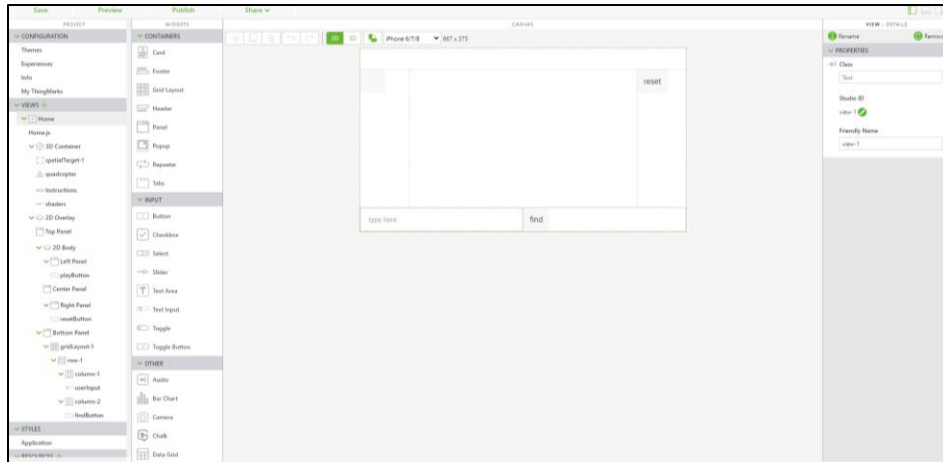
```
$scope.view.wdg.playButton.text = '';  
$scope.view.wdg.playButton.toPlay = undefined;
```

2. Visit [Appendix 2](#) for the updated code.

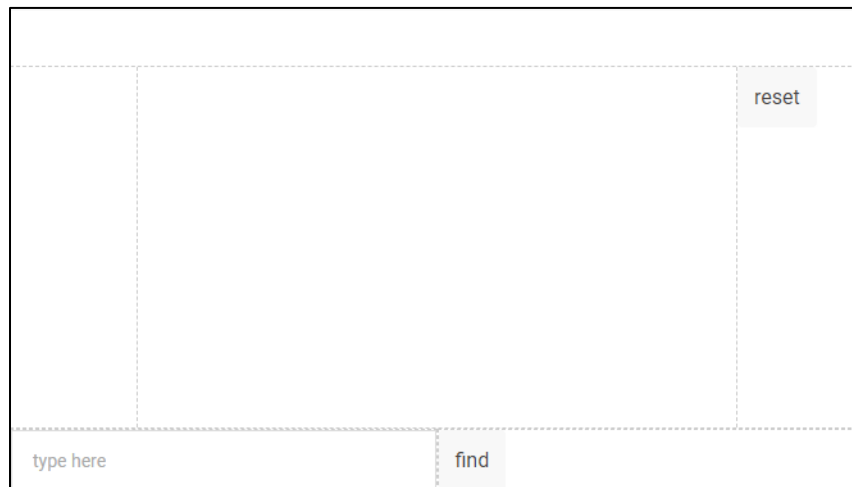
### 301.5 Update the 2D Interface

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 labels and a **repeater** for listing all the items in the cart.

1. Open the 2D canvas.



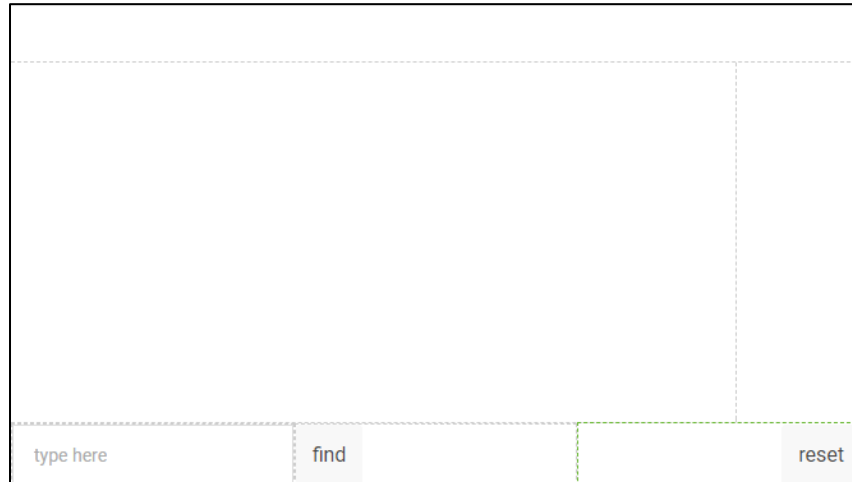
- Click on the **playButton**. In the **Properties** pane, select **Remove** to remove it from the canvas. The function of this button will now be triggered by the popup.



- 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.

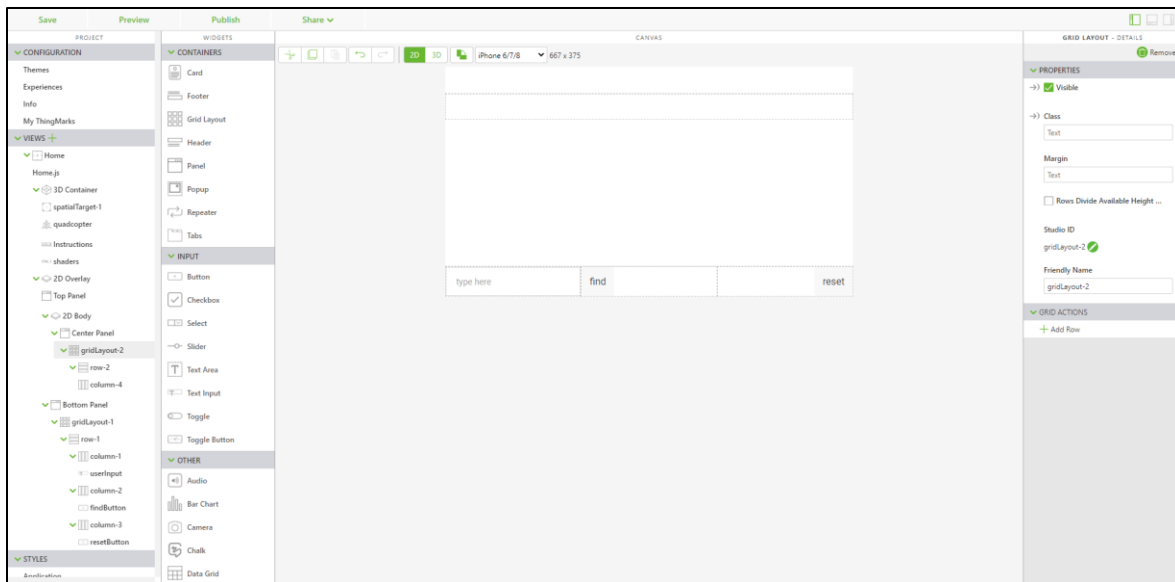


5. **Remove** the right panel.

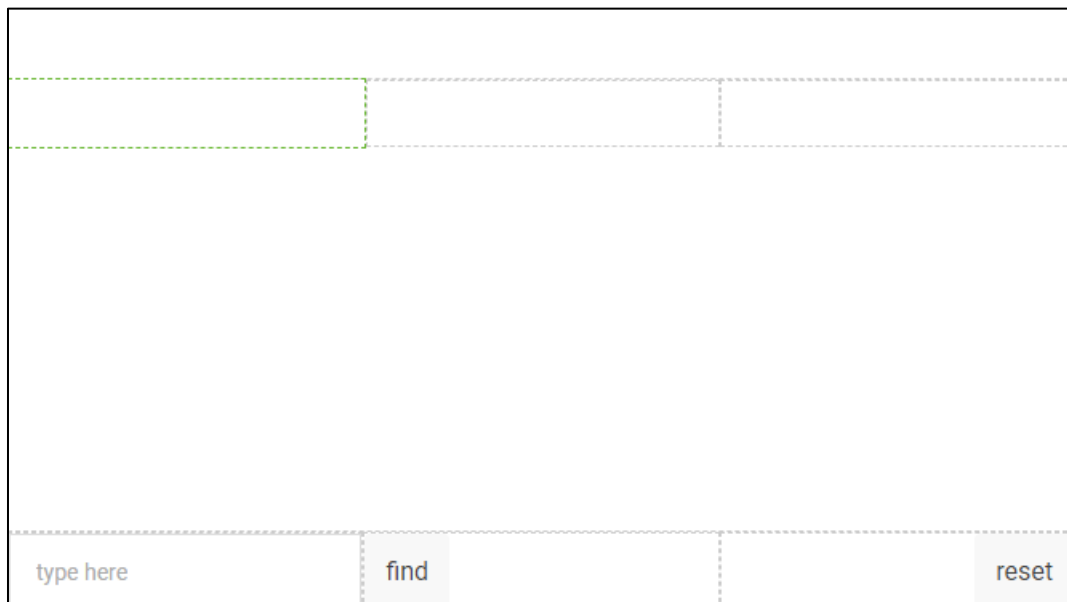


6. Now that parts of the layout from the old experience have been removed, new 2D widgets will be added to the experience.
  - a. Drag a **Grid Layout** widget onto the **Center Panel**. This will be used to split this panel into columns of equal width.

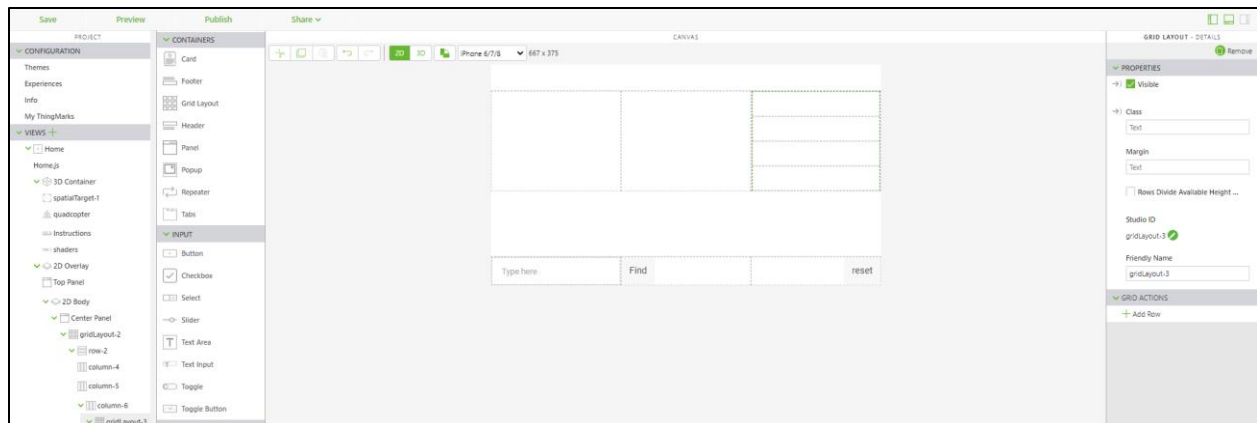




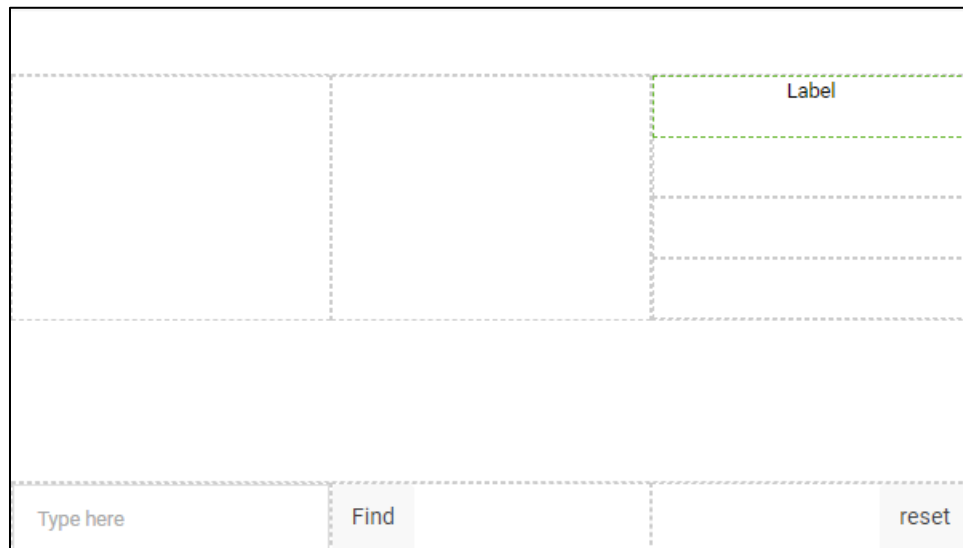
- b. Click into **column-4**, the newly created column in the **Grid Layout**. Under **Grid Actions**, click **Add Column** twice. This will split the **Grid Layout** into three equal columns.



- c. Drag another **Grid Layout** widget into the **right** column, **column-6**. In the **Grid Layout** properties, click **Add Row** three times so that **gridLayout-3** now has four rows.



- d. Drag a **Label** widget into the top row of the new grid layout. Change the **Studio ID** of this label to be *labelCart*. Click into **column-7**, or whatever the name is for the column that the label was just placed into, and change the **Alignment** property to **Center** to place the label in the middle of the row. This label will display the total price of the items in the cart when it is populated.



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

Remove

PROPERTIES

Text

Label

Class

simple-label

Enable State-Based Formatting

Padding

Text

Visible

Margin

Text

Wrap Label Text

Studio ID

label/Cart

Friendly Name

label/Cart

DATA

APPLICATION PARAMETERS

cartLabel

itemCount

itemName

itemNumber

partno

priceInfo

Thing

Thing Template

ThingMark

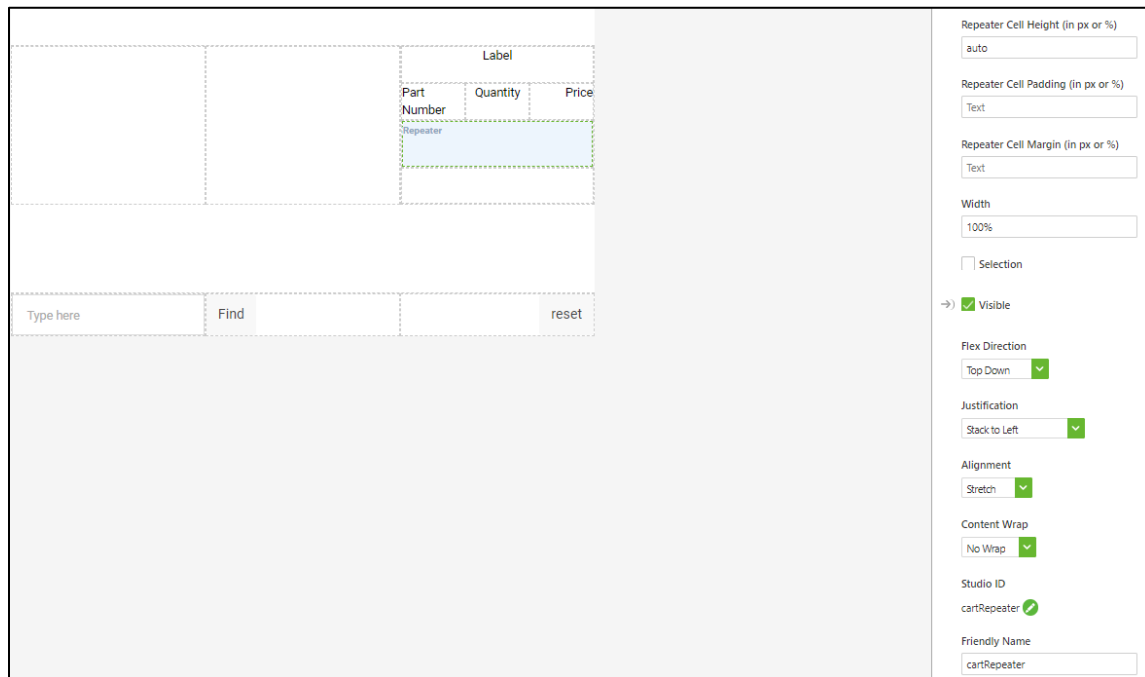
EXTERNAL DATA

- f. Divide the second row into three columns. Place a **Label** widget into each column.
- In the first column, change the **Text** property of the label to *Part Name*. Change the **Alignment** property for the column to **Start**.
  - For the middle label, change the **Text** property to **Quantity**. Change the **Alignment** property for the column to **Center**.
  - In the third column, change the **Text** property of the label to *Price*. Change the **Alignment** property for the column to **End**.

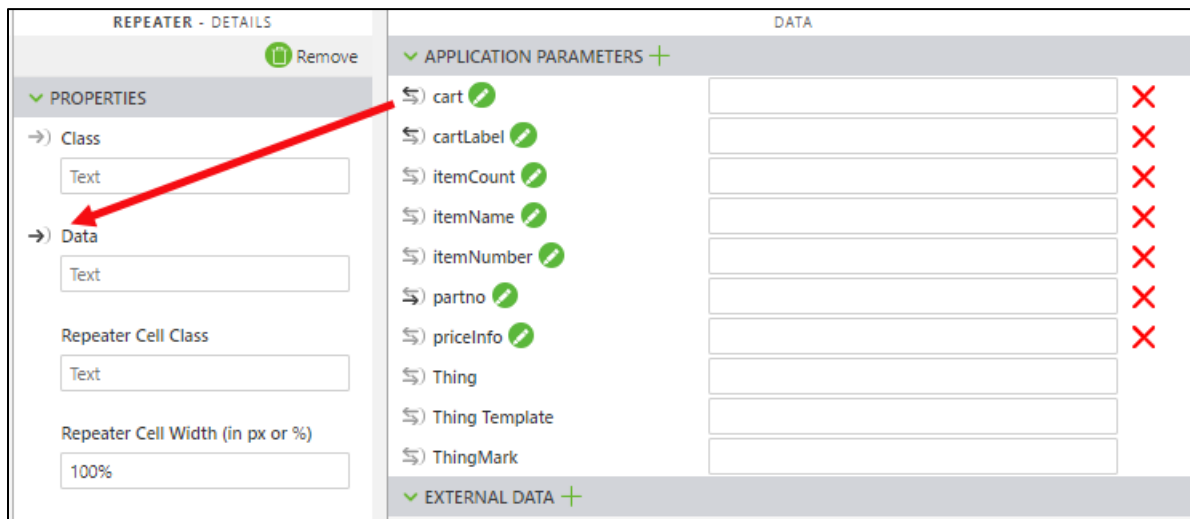
Label		
Part Number	Quantity	Price
Type here	Find	reset

- g. In the third row, 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*.



- h. 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.



- i. 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.

		Label		
		Part Number	Quantity	Price
		Repeater		
Type here	Find	reset		

- j. Add a **Label** widget into each of the newly created columns. Like with the labels in the row above them, set the **Alignment** of the columns to be **Start**, **Center**, and **End**, respectively. These labels will be further edited in a later portion of the project.

		Label		
		Part Number	Quantity	Price
		Label	Label	Label
Type here	Find	reset		

- k. Click into the last row of this section and select **Add Column** to add another column.

		Label		
		Part Number	Quantity	Price
		Label	Label	Label
Type here	Find	reset		

- I. 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 button will be bound to a ThingWorx service for ordering the contents of the cart in Metadata 303.

		Label		
		Part Number	Quantity	Price
		Label	Label	Label
		Order		
Type here	Find	reset		

PROPERTIES

→ Text  
Order

→ Class  
Text

→ ☒ Visible

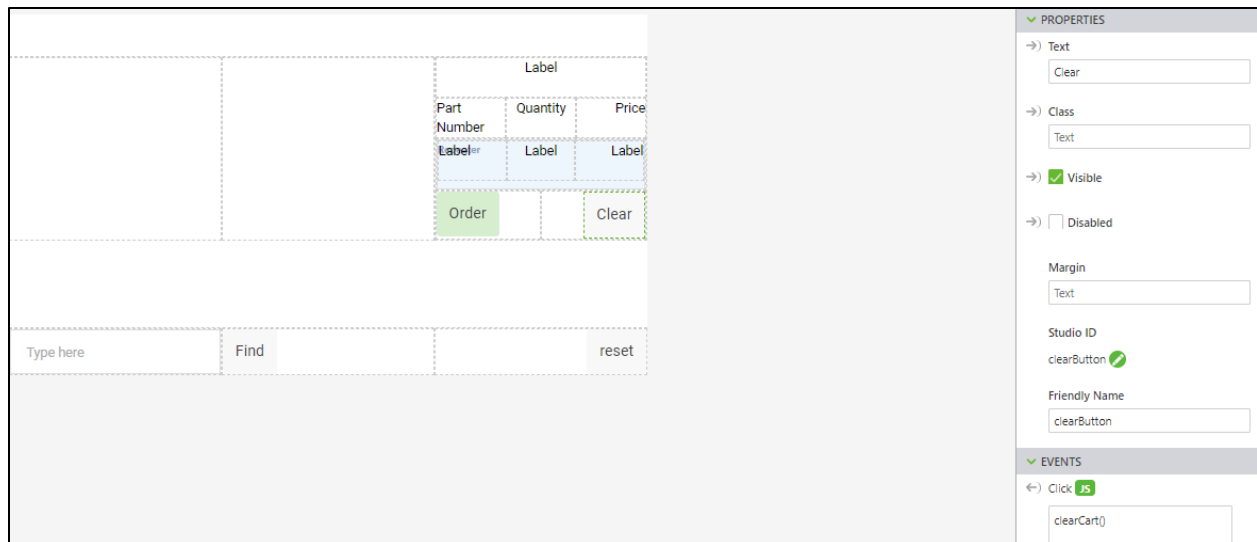
→ ☐ Disabled

Margin  
Text

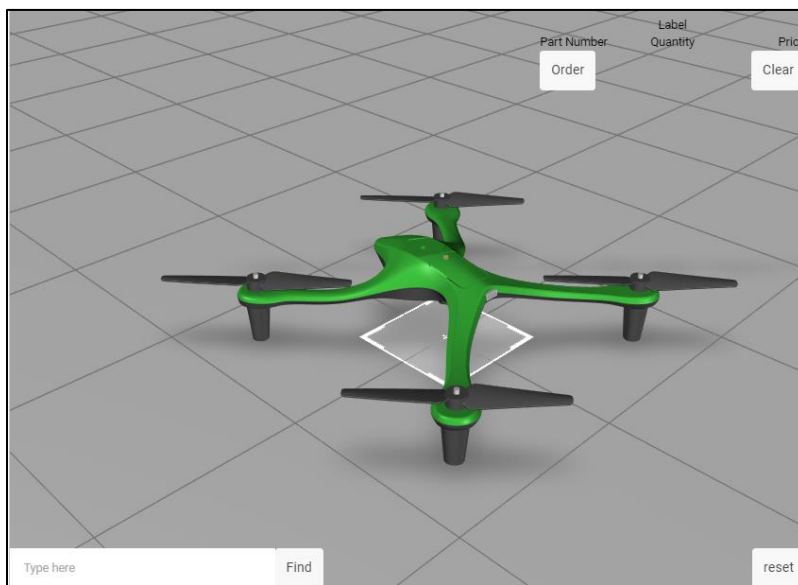
Studio ID  
orderButton

Friendly Name  
orderButton

- m. 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. Change the **Alignment** property of this column to **End**.



- n. 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

A major part of this tutorial is adding functionality to the popup so that when you click on a part, there are clickable buttons to add the part to a cart, view a disassembly sequence, or continue inspecting the model. Before they can actually be added to the popup, though, these buttons need to have functions that can be tied to them. Functions will be created in this section for adding a part to the cart and starting the disassembly sequence. The cart contains information that is stored in the **cartRepeater** widget and grows as parts are added to the order.

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 be created as a standalone function.
    - i. Below the **hilite** function, initialize the **cartLabel** application parameter to just say **Cart**. This will be the beginning text in the **labelCart** label and will change as items are added to the cart. Also, initialize an empty object named **cart**.

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

```
152 } //hilite function end
153
154 $scope.app.params.cartLabel = "Cart"; // set cartLabel app parameter to be "Cart". This will bind to the Text proper
155 $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 in **cart** 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

```

```

155 $scope.cart = {}; // declare empty object called cart
156
157 //
158 // function for adding a selected part to the cart
159 $scope.addToCart = function () {
160
161     //
162     // create variable called cartItem that is equal to the value of the currentSelection property of the cart object.
163     // If the selected part hasn't been added to the cart yet, then the cartItem variable will be undefined and populate the
164     // information about the part so that cartItem becomes an object. If the selected part has already been added, then the
165     var cartItem = $scope.cart[$scope.currentSelection];
166
167     if (cartItem === undefined) {
168         cartItem = { count: $scope.app.params.itemCount,
169                     itm: $scope.app.params.itemNumber,
170                     tag: $scope.app.params.itemName,
171                     prc: $scope.app.params.priceInfo };
172     } else {
173         cartItem.count += $scope.app.params.itemCount
174     }
175
176     $scope.cart[$scope.currentSelection] = cartItem;
177
178 } // 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;

```

```

176 $scope.cart[$scope.currentSelection] = cartItem;
177
178 //
179 //cartItemAmount initialized as 0. will be used to count how many items are in the cart
180 var cartItemAmount = 0;
181
182 //
183 // set an empty array for the cart. this array will have an object pushed into it
184 var cartContents = [];
185
186 //
187 // initialize variable for keeping track of the price of the objects in the cart
188 var cartPrice = 0;
189
190 } // 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 instance 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;

188     var cartPrice = 0;
189
190     //
191     //loop over each item that is added to the cart
192     for (var itm in $scope.cart) {
193
194         //
195         //add a number to the counting variable for each item added
196         cartItemAmount += $scope.cart[itm].count;
197
198         //
199         // add the price of each item to the total price of the cart
200         cartPrice = cartPrice += $scope.cart[itm].count*$scope.cart[itm].prc
201
202         //
203         //push the name (tag), item count (count), and price (prc) of each part into the repeater for the cart
204         cartContents.push({
205             tag : $scope.cart[itm].tag,
206             count: $scope.cart[itm].count,
207             prc : $scope.cart[itm].prc
208         }); // end of the push method for cartContents
209
210     } // for loop end
211
212     //
213     // set the app parameter for cart to be equal to the cartContents array
214     $scope.app.params.cart = cartContents;
215
216 }
```

- v. The last portion of the `addToCart` function is editing the text for the **labelCart** label by adding a value to the **cartLabel** application parameter.

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

```
214 $scope.app.params.cart = cartContents;
215
216 //
217 //setting the cartLabel app parameter. if there are items to put into the cart (true), the text of
218 $scope.app.params.cartLabel = cartItemAmount > 0 ? "Cart($ " + cartPrice + ")"
219 : "Cart";
220
221 } // end of addToCart function
```

- b. A new 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 **labelCart** button to be set back to **Cart** without a price total using the **cartLabel** application parameter.

```
//
// clear the cart. set the part app parameter and cart object to be empty. change the
text on the cart label back to just Cart
$scope.clearCart = function () {

    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartLabel = "Cart";

} // end of clearCart function
```

```
221 } // end of addToCart function
222
223 //
224 // clear the cart. set the part app parameter and cart object to be empty. change
225 $scope.clearCart = function () {
226
227     $scope.app.params.cart = [];
228     $scope.cart = {};
229     $scope.app.params.cartLabel = "Cart";
230
231 } // end of clearCart function
```

- c. 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 below where the shader is added to the part.

```
//
//function for removing the highlight
$scope.hiliteOff = function() {

    $scope.hilite[$scope.currentSelection], false)

}; // end of hiliteOff function
```

```

45 //highlight the chosen item and set the shader to true
46 $scope.hilite([$scope.currentSelection], true);
47
48 //
49 //function for removing the highlight
50 $scope.hiliteOff = function() {
51     $scope.hilite([$scope.currentSelection], false)
52
53 }; // end of hiliteOff function
54

```

- d. Finally, inside the PTC API, create a function called **disassemble** below the function that you just created. 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: '1-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

```

```

54 }; // end of hiliteOff function
55
56 //
57 // function to be bound to the Disassemble button in the popup
58 $scope.disassemble = function () {
59
60     //
61     // set an object that targets the model and its instruction property
62     var modelObject = {      model: targetName,
63                             instruction: '1-Creo 3D - ' + instructionName + '.pvi' };
64
65     //
66     // set the sequence for the quadcopter to be the name of the associated instruction
67     $scope.view.wdg.quadcopter.sequence = modelObject.instruction
68
69 } //disassemble function end

```

- e. The completed code for this section is available in [Appendix 3](#).

### 301.7 Creating Popup Buttons

Now that the addToCart and disassemble functions have been created, they will be added into the popup to make them interactive. The template for the popup will also be changed to account for the differences between parts

1. The **Add to Cart** button will be added into the **price** variable and will have the **addToCart** function attached to it. It is added to the price variable instead of the popup because the button will only appear if there is a price associated with the part. The **.close** method and **hiliteOff** function are also tied to the button to close the popup and remove the shader once the button has been clicked.

```

var price = priceString != undefined ? '&nbsp;<br>$' + parseFloat(priceString) +
'&nbsp;</div><div ng-click="hiliteOff(); popup.close();addToCart();"> Add to Cart</div>'
: "</div>";

```

```

25 //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
26 var price = priceString != undefined ? '&nbsp;<br>$' + parseFloat(priceString) + '&nbsp;</div><div ng-click="hiliteOff(); popup.close();addToCart();"
27 : "</div>";

```

2. The template for the popup will now be edited to add in the `itemCount` application parameter, the price of the part, and a **Continue** button.
  - a. Add the `itemCount` application parameter and `price` variable to the popup template as shown below.

```
template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
                                             '&nbsp;<br>' + partName +
                                             price +
'&nbsp;</div>';
```

```
39 *   var popup = $ionicPopup.show({
40     template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
41                                             '&nbsp;<br>' + partName +
42                                             price + '&nbsp;</div>',
43     scope: $scope
44   }); //end of ionic popup
```

- b. A **Continue** button will be added directly to the template of the popup. Clicking this button will simply close out of the popup and not perform any other action. It activates the `hiliteoff` function and `.close` method when clicked. The popup will now have buttons for adding a part to the cart and closing the popup.

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

```
39 *   var popup = $ionicPopup.show({
40     //template for the popup with added buttons
41     template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
42                                             '&nbsp;<br>' + partName +
43                                             price +
44                                             '<div ng-click="hiliteOff();popup.close();">Continue</div>',
45     scope: $scope
46   }); //end of ionic popup
```

3. 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 **Add to Cart**, **Disassembly**, and **Continue** 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.
  - a. 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, which is the current state of your popup. Add the new line of code above where the popup is called and change the scope of the `popup` variable so that it can be accessed outside the `if` statement. Indent your code so it lines up correctly inside the `if` statement.

```
if (instructionName.length == 0) {
```

```

//
// adds an ionic popup when a part is clicked. Show the quantity, part number,
name, and price 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 +
        price +
        '<div ng-

```

```

click="hiliteOff();popup.close();">Continue</div>',

```

```

    scope: $scope
}); //end of ionic popup

```

```

}

```

```

35 $scope.app.params.itemCount = 1;
36
37 if (instructionName.length == 0) {
38
39     //
40     // adds an ionic popup when a part is clicked. Show the quantity, part number, name, and price of the selected object
41     $scope.popup = $ionicPopup.show({
42
43         //
44         //template for the popup with added buttons
45         template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
46                 '&nbsp;<br>' + partName +
47                 price +
48                 '<div ng-click="hiliteOff();popup.close();">Continue</div>',
49
50         scope: $scope
51     }); //end of ionic popup
52
53 }

```

- b. The **else** condition will call a popup for the case when there is a disassembly sequence associated with the part. This will call a popup with all three possible buttons; **Add to Cart**, **Disassemble**, and **Continue**. The **Disassemble** button not only calls the **disassemble** function, but also **hiliteOff** and **popup.close**.

```

} else {

```

```

//
// adds an ionic popup when a part is clicked. Show the quantity, part number,
name, and price 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 +
        price +

```

```

'<div ng-click="hiliteOff();popup.close();disassemble();">Disassemble</div>' +
        '<div ng-
click="hiliteOff();popup.close();">Continue</div>',

```

```

    scope: $scope

```

```
}); //end of ionic popup if there is a disassembly sequence associated with it
```

```
} // end of if statement
```

```
51     }); //end of ionic popup
52
53   } else {
54
55     //
56     // adds an ionic popup when a part is clicked. Show the quantity, part number, name, and price of the selected object. &n
57     $scope.popup = $ionicPopup.show({
58
59       //
60       //template for the popup with added buttons
61       template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;' + partNumber +
62                '&nbsp;' + partName +
63                '<div ng-click="highlightOff();popup.close();disassemble();">Disassemble</d
64                '<div ng-click="highlightOff();popup.close();">Continue</div>',
65                price +
66
67       scope: $scope
68     }); //end of ionic popup if there is a disassembly sequence associated with it
69
70   } // end of if statement
```

4. Click **Save** and open **Preview**.

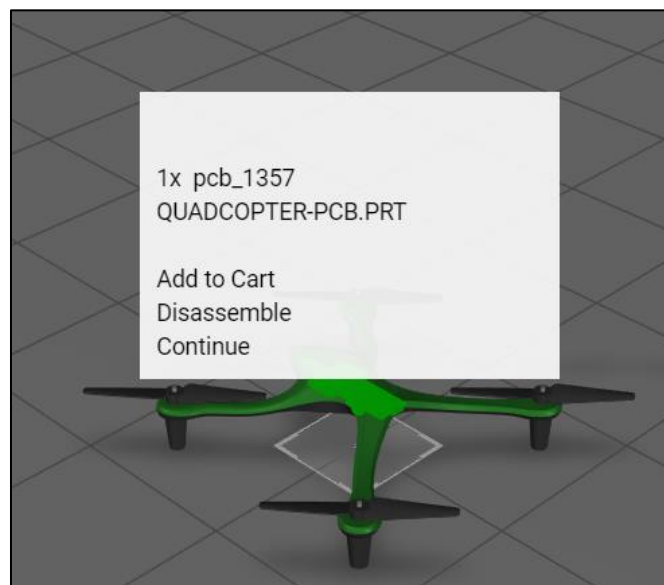
- a. Click on any of the rotors to see the changes that have been made to the popup for when there are three buttons available. Though they have not been styled yet, each button can be clicked. Try each button to see how they interact with the model.



- b. Now click on the base of the quadcopter to see the difference in the popup when there is not a disassembly sequence associated with a part.



- c. Finally, click on the PCB. You will notice that when you click on the PCB that a price does not appear, and you are not given the option to add the part to your cart. This is because the PCB and battery were combined into a single part earlier and the individual PCB does not have a price associated with it.



- d. If all three different versions of the popup appear successfully, you have completed this section. The complete code is available in [Appendix 4](#).

### 301.8 Styling the Experience

As a final step for this tutorial, CSS styling will be added to the project to add color to the buttons and cart.

1. Open the **Applications** tab under **Styles**. This is where the CSS code will be written.
  - a. Add the following CSS classes. These classes will be tied to parts of the experience in the next few steps.



```
.btnadd {  
  background: #236192;  
  color: #ffffff;  
  border-radius: 10px;  
  padding: 5px 5px 5px 5px;  
}
```

```
.btncontinue {  
  background: #c8c9c7;  
  color: #ffffff;  
  border-radius: 10px;  
  padding: 5px 5px 5px 5px;  
}
```

```
.btndisassemble {  
  background: #3d4647;  
  color: #ffffff;  
  border-radius: 10px;  
  padding: 5px 5px 5px 5px;  
}
```

```
.btnfind {  
  background: #3d4647;  
  color: #ffffff;  
}
```

```
.btnreset {  
  background: #c8c9c7;  
  color: #ffffff;  
}
```

```
.cart {  
  background: #236192;  
  color: #ffffff;  
}
```

```
.cartlabels {  
  background: #00acc8;  
  color: #ffffff;  
}
```

```
.btnclear {  
  background: #3d4647;  
  color: #ffffff;  
  border-radius: 0px;  
}
```

```
.btnorder {  
  background: #f38800;  
  color: #ffffff;  
  border-radius: 0px;  
}
```

```
.repeater {  
  background: #c8c9c7;  
  color: #236192;  
}
```

```

17 ▾ .btnadd {
18     background: #236192;
19     color: #ffffff;
20     border-radius: 10px;
21     padding: 5px 5px 5px 5px;
22 }
23
24 ▾ .btncontinue {
25     background: #c8c9c7;
26     color: #ffffff;
27     border-radius: 10px;
28     padding: 5px 5px 5px 5px;
29 }
30
31 ▾ .btndisassemble {
32     background: #3d4647;
33     color: #ffffff;
34     border-radius: 10px;
35     padding: 5px 5px 5px 5px;
36 }
37
38 ▾ .btnfind {
39     background: #3d4647;
40     color: #ffffff;
41 }
42
43 ▾ .btnreset {
44     background: #c8c9c7;
45     color: #ffffff;
46 }
47
48 ▾ .cart {
49     background: #236192;
50     color: #ffffff;
51 }
52
53 ▾ .cartlabels {
54     background: #00acc8;
55     color: #ffffff;
56 }
57
58 ▾ .btnclear {
59     background: #3d4647;
60     color: #ffffff;
61     border-radius: 0px;
62 }
63
64 ▾ .btnorder {
65     background: #f38800;
66     color: #ffffff;
67     border-radius: 0px;
68 }
69
70 ▾ .repeater {
71     background: #c8c9c7;
72     color: #236192;
73 }

```

2. Now that the CSS classes have been added in, it is time to add them to the 2D interface.
  - a. Navigate back to the **Home** tab and open the 2D canvas. Change the **Text** property for each label to the following:
    - i. `{{item.tag}}`
    - ii. `{{item.count}}`
    - iii. `${{item.prc}}`

Label		
Part Number	Quantity	Price
{{item.tag}}	{{item.count}}	\${{item.prc}}
Order		Clear
Type here	Find	reset

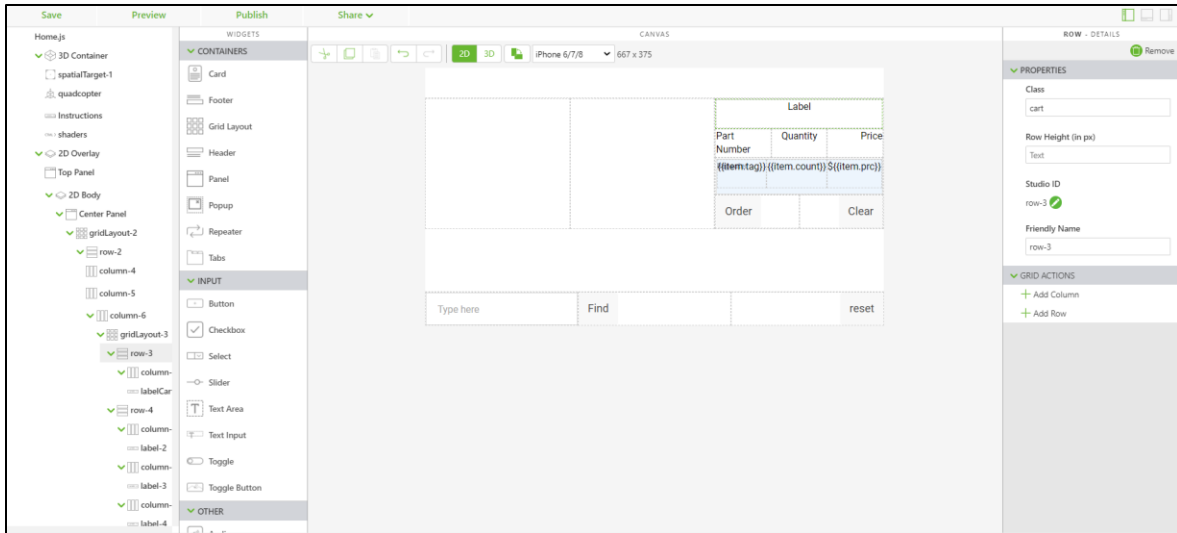
- iv. Adding this text calls upon the properties of the `cartItem` object that was created in the below section of code. The contents of this object were set equal to the `cart` application parameter and the labels tie the repeater back to the properties of `cart`.

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

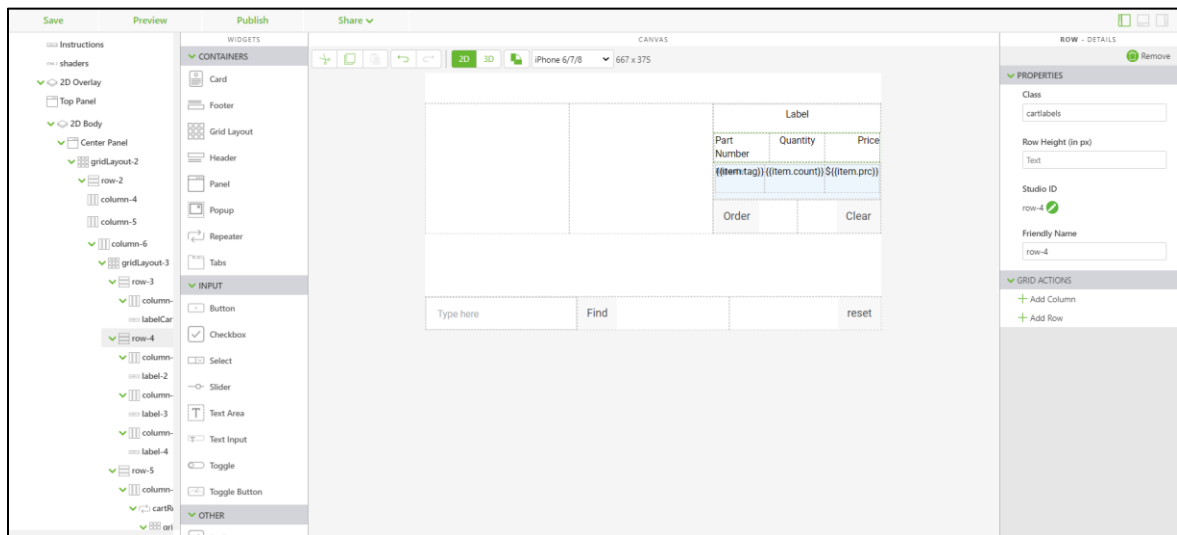
if (cartItem === undefined) {
    cartItem = { count: $scope.app.params.itemCount,
        itm: $scope.app.params.itemNumber,
        tag: $scope.app.params.itemName,
        prc: $scope.app.params.priceInfo }
} else {
    cartItem.count += $scope.app.params.itemCount
}

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

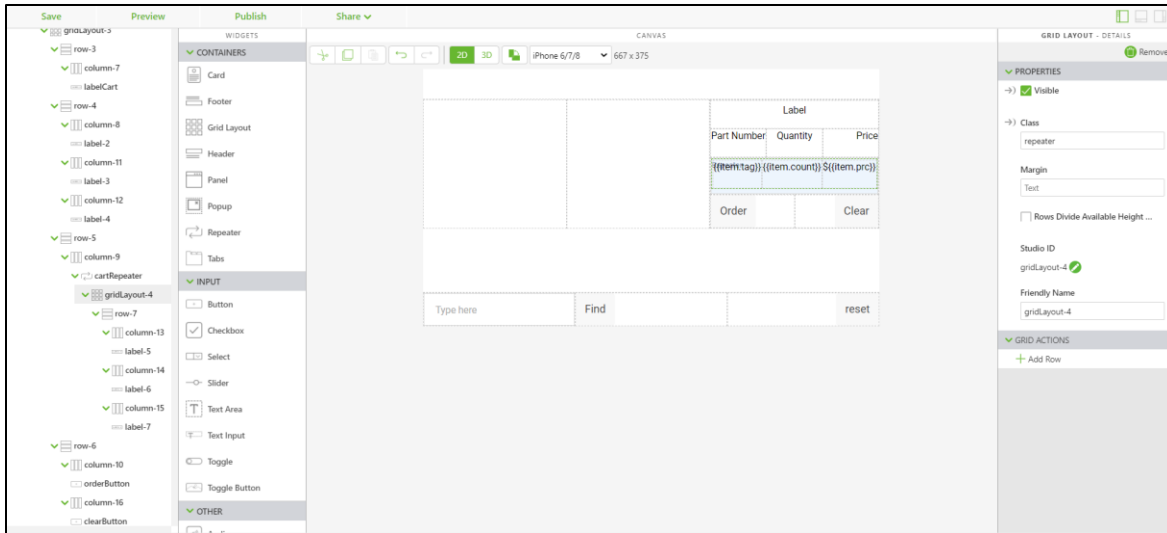
- b. Click on the first row in the grid layout with the **labelCart** widget in it. Change the **Class** property of the row to `cart`. This will add a blue background to that column that will act as a header for the cart. Open the column of that same row and change the **Padding** property to `5px` to add padding around the row to separate it from other rows.



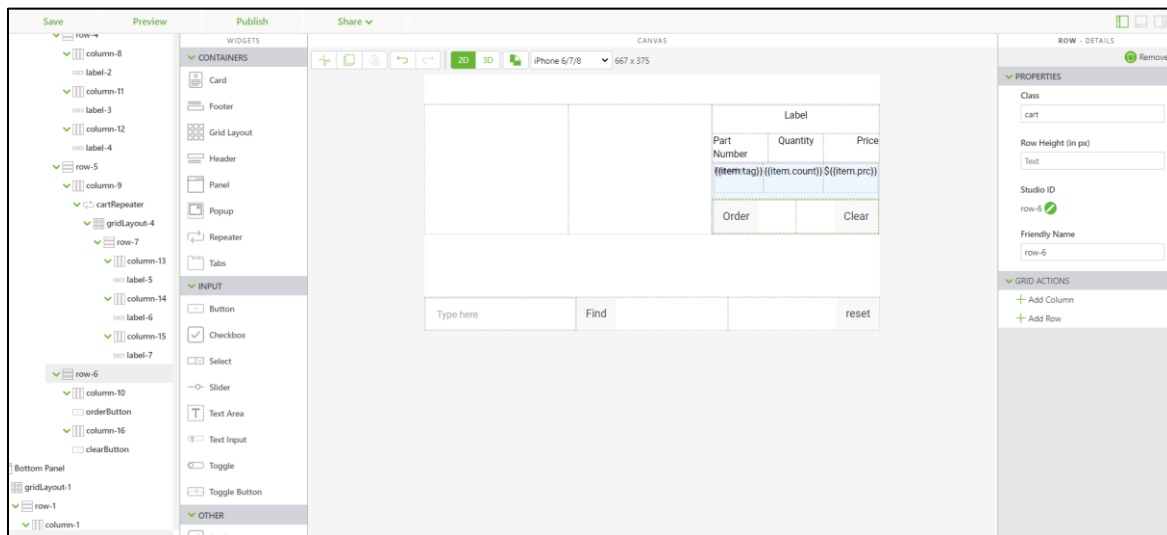
- c. Click the row below it and change the **Class** property to *cartlabels*. This will add a light blue background to this row and will be used as headers for each cart column.



- d. To add styling to the contents of the repeater, open the Grid Layout widget that you added after you added the repeater. Change its **Class** property to *repeater*. This will add a light gray background with blue lettering to the contents of the repeater.



- e. Like with the first row that you edited, click the last row in the layout and change its **Class** property to **cart**. It will follow the same styling convention as the row with the **labelCart** widget in it.



- f. Edit the **Class** property of each of the Button widgets with their corresponding CSS class:
- orderButton** > *btnorder*
  - clearButton** > *btnclear*
  - findButton** > *btnfind*
  - resetButton** > *btnreset*
- g. Once you have added those classes to the appropriate buttons, click **Save** and then **Preview**. Click on a part and add it to the cart. Your experience should now look like the one below.



3. The final step for styling the experience is to add CSS classes to the buttons in the popup. Unlike the buttons in the 2D interface, to edit the buttons in the popup you need to edit the template of the popup. Open **Home.js**.

- a. In the **price** variable, **class="btnadd"** needs to be added to the popup template to style the **Add to Cart** button.

```
var price = priceString != undefined ? '&nbsp;<br>$' + parseFloat(priceString) +
'&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>";
25 //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
26 var price = priceString != undefined ? '&nbsp;<br>$' + parseFloat(priceString) + '&nbsp;</div><div class="btnadd" ng-click="hiliteOff(); popup.close();
27 : "<div class="btncontinue" ng-click="hiliteOff(); popup.close();">Continue</div>";
```

- b. For the case in the if else statement when there is not a disassembly sequence, the **btncontinue** class will be added to the popup template for styling the **Continue** button.

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

```
37 if (instructionName.length == 0) {
38
39 //
40 // adds an ionic popup when a part is clicked. Show the quantity, part number, name, and price of the selected object. &nbsp;<br> adds
41 $scope.popup = $ionicPopup.show({
42
43 //
44 //template for the popup with added buttons
45 template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;<br>' + partNumber +
46 '&nbsp;</div>' + partName +
47 price +
48 '<div class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
49
```

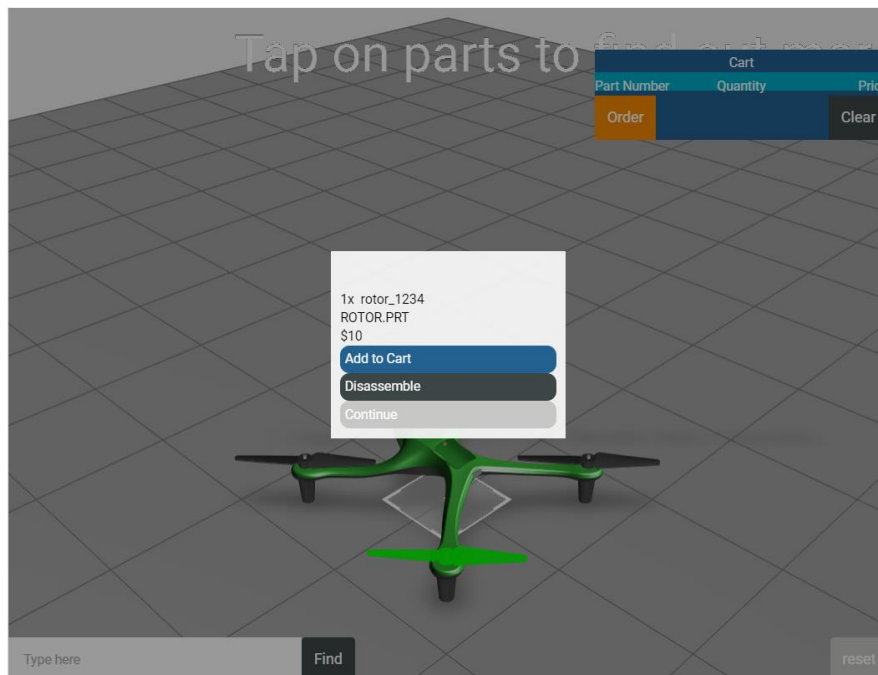
- c. When there is both a **Continue** and **Disassemble** button in the popup in the else portion of the if else statement, classes for **btncontinue** and **btndisassemble** will need to be added to the template.

```
template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;<br>' + partNumber +
'&nbsp;</div>' + partName +
price +
```

```
'<div class="btndisassemble" ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' +
'<div class="btncontinue" ng-
click="hiliteOff();popup.close();">Continue</div>',
```

```
52 *
53 * } else {
54 *
55 * // adds an ionic popup when a part is clicked. Show the quantity, part number, name, and price of the selected object. &nbsp;&nbsp;<br> adds a line break bet
56 * $scope.popup = $ionicPopup.show({
57 *
58 * //
59 * //template for the popup with added buttons
60 * template: '<div>' + $scope.app.params.itemCount + 'x &nbsp;&nbsp;' + partNumber +
61 * '&nbsp;&nbsp;<br>' + partName +
62 * 'price' +
63 * '<div class="btndisassemble" ng-click="hiliteOff();popup.close();disassemble();">Disassemble</div>' +
64 * '<div class="btncontinue" ng-click="hiliteOff();popup.close();">Continue</div>',
65 *
66 *
```

4. All the CSS styling has now been added to the experience. Click **Save** and then **Preview** to view how your experience has changed. If your popup buttons look like the ones in the image, then this section has been completed correctly. Try changing the values in the **Application** tab to see how the styling on the experience can change. The full code for this section is available in [Appendix 5](#).



## Appendix 1: Section 301.3 Code

```
//
// triggered when user clicks on object in the scene
$scope.$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. Create variable for the currently selected part
```

```

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. 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 ? '&nbsp;<br>$' + 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 quantity, part number,
name, and price 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 +
                                                price +
                                                '<div ng-

click="hiliteOff();popup.close();">Continue</div>',

scope: $scope
}); //end of ionic popup

} else {

//
// adds an ionic popup when a part is clicked. Show the quantity, part number,
name, and price 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 +
price +

'<div ng-click="hiliteOff();popup.close();disassemble();">Disassemble</div>' +
'<div ng-
click="hiliteOff();popup.close();">Continue</div>',

    scope: $scope
}); //end of ionic popup if there is a disassembly sequence associated with it

} // end of if statement

//
//highlight the chosen item and set the shader to true
$scope.hilite([$scope.currentSelection], 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 after
3 seconds (3000 ms)
$timeout(closePopup(popup, [$scope.currentSelection]), 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 identifiers. 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

//
// highlighting 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 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

```

//
// triggered when user clicks on object in the scene
$scope.$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. Create variable for the currently selected part
    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. 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 ? '&nbsp;<br>$' + 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;

//
// adds an ionic popup when a part is clicked. Show the part number and name of the
selected object. &nbsp;</br> adds a line break between the two variables
var popup = $ionicPopup.show({
  template: '<div>' + partNumber + '&nbsp;</br>' + partName + '</div>',
  scope: $scope
}); //end of ionic popup

//
//highlight the chosen item and set the shader to true
$scope.hilite([$scope.currentSelection], true);

//
//function for removing the highlight
$scope.hiliteOff = function() {

  $scope.hilite([$scope.currentSelection], 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: '1-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 identifiers. 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

//
// highlighting 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 end

} //hilite function end

$scope.app.params.cartLabel = "Cart"; // set cartLabel app parameter to be "Cart". This
will bind to the Text property for the labelCart label
$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 cartLabel app parameter. if there are items to put into the cart (true),
the text of the cart label should be cart(total cost of cart). If false, just keep the
label text as cart
$scope.app.params.cartLabel = cartItemAmount > 0 ? "Cart($ + cartPrice + ")"

```

```

"Cart";

} // end of addToCart function

//
// clear the cart. set the part app parameter and cart object to be empty. change the
text on the cart label back to just Cart
$scope.clearCart = function () {

    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartLabel = "Cart";

} // end of clearCart function

```

## Appendix 4: Section 301.7 Code

```

//
// triggered when user clicks on object in the scene
$scope.$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. Create variable for the currently selected part
        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. 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 ? '&nbsp;<br>$' + parseFloat(priceString) +
'&nbsp;</div><div ng-click="hiliteOff(); popup.close();addToCart();"> Add to Cart</div>'
        : "";

        //
        // 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 quantity, part number,
    name, and price 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 +
                                                         price +
                                                         '<div ng-
click="hiliteOff();popup.close();">Continue</div>',

        scope: $scope
    }); //end of ionic popup

} else {

    //
    // adds an ionic popup when a part is clicked. Show the quantity, part number,
    name, and price 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 +
                                                         price +

        '<div ng-click="hiliteOff();popup.close();disassemble();">Disassemble</div>' +
                                                         '<div ng-
click="hiliteOff();popup.close();">Continue</div>',

        scope: $scope
    }); //end of ionic popup if there is a disassembly sequence associated with it

} // end of if statement

//
//highlight the chosen item and set the shader to true
$scope.hilite([$scope.currentSelection], true);

//
//function for removing the highlight
$scope.hiliteOff = function() {

    $scope.hilite([$scope.currentSelection], 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: '1-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 identifiers. 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

    //
    // highlighting 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 end

} //hilite function end

$scope.app.params.cartLabel = "Cart"; // set cartLabel app parameter to be "Cart". This
will bind to the Text property for the labelCart label
$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 cartLabel app parameter. if there are items to put into the cart (true),
the text of the cart label should be cart(total cost of cart). If false, just keep the
label text as cart
$scope.app.params.cartLabel = cartItemAmount > 0 ? "Cart($ " + cartPrice + " )"
:
"Cart";

} // end of addToCart function

//
// clear the cart. set the part app parameter and cart object to be empty. change the
text on the cart label back to just Cart
$scope.clearCart = function () {

    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartLabel = "Cart";

} // end of clearCart function

```

## Appendix 5: Section 301.8 Code

```

//
// triggered when user clicks on object in the scene
$scope.$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. Create variable for the currently selected part
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. 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 ? '&nbsp;</br>$' + parseFloat(priceString) +
'&nbsp;</div><div class="btnadd" ng-click="hiliteOff(); popup.close();addToCart();"> Add
to Cart</div>'
: "";

//
// 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 quantity, part number,
name, and price 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;</div>' + partNumber +
'&nbsp;</br>' + partName +
price +
'<div class="btncontinue" ng-
click="hiliteOff();popup.close();">Continue</div>',

scope: $scope
}); //end of ionic popup

} else {

//
// adds an ionic popup when a part is clicked. Show the quantity, part number,
name, and price 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 +
                                                    price +

    '<div class="btndisassemble" ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' +
                                                    '<div class="btncontinue" ng-
click="hiliteOff();popup.close();">Continue</div>',

    scope: $scope
}); //end of ionic popup if there is a disassembly sequence associated with it

} // end of if statement

//
//highlight the chosen item and set the shader to true
$scope.hilite([$scope.currentSelection], true);

//
//function for removing the highlight
$scope.hiliteOff = function() {

    $scope.hilite([$scope.currentSelection], 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: '1-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 identifiers. 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

    //
    // highlighting 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 end

    } //hilite function end

    $scope.app.params.cartLabel = "Cart"; // set cartLabel app parameter to be "Cart". This
    will bind to the Text property for the labelCart label
    $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 cartLabel app parameter. if there are items to put into the cart (true),
the text of the cart label should be cart(total cost of cart). If false, just keep the
label text as cart
$scope.app.params.cartLabel = cartItemAmount > 0 ? "Cart($" + cartPrice + ")"
:
"Cart";

} // end of addToCart function

//
// clear the cart. set the part app parameter and cart object to be empty. change the
text on the cart label back to just Cart
$scope.clearCart = function () {

    $scope.app.params.cart = [];
    $scope.cart = {};
    $scope.app.params.cartLabel = "Cart";

} // end of clearCart function
```