ptc

vuforia™ studio

Metadata 303

Creating a Persistent Cart Using ThingWorx

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
Metadata 301 – Adding Pricing Data and a Shopping Cart to a Model
Metadata 302 – Add a Simple ThingWorx Service to Vuforia Studio

Intro

Think of buying something on the internet; most of the time, when you add something to your cart it will stay there until you check out or remove the object. This is the concept of a persistent shopping cart. Up until now, when you had added an item into your cart in your Vuforia Studio experience, it disappeared whenever you restarted your experience. This issue can be resolved with the help of ThingWorx being connected to Vuforia Studio to store part information in a persistent cart, which will be the focus of this exercise.

In order to complete this exercise, you must have completed each part of the Metadata series of projects before this.

The following topics will be covered in this project. Jump to them with their hyperlinks:

There is also an appendix at the end of the document for the completed code of this project.

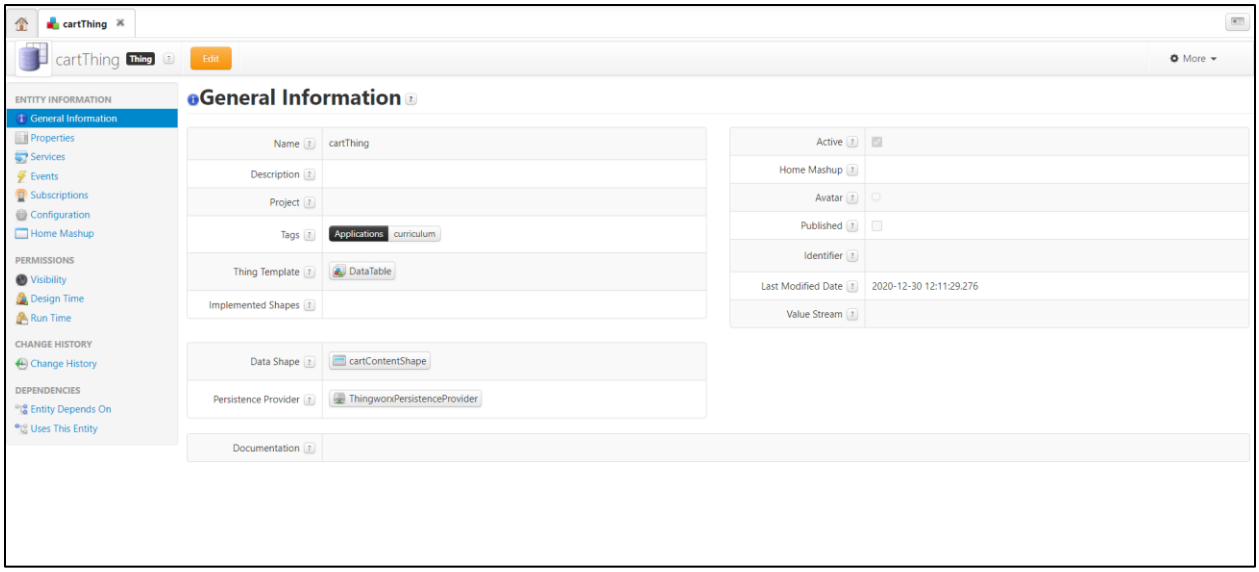
All important notes and UI areas are **Bold**.
All non-code text to be typed is *italicized.*
All code follows `this convention`
All code comments follow `this convention`

303.1 Becoming Familiar with cartThing

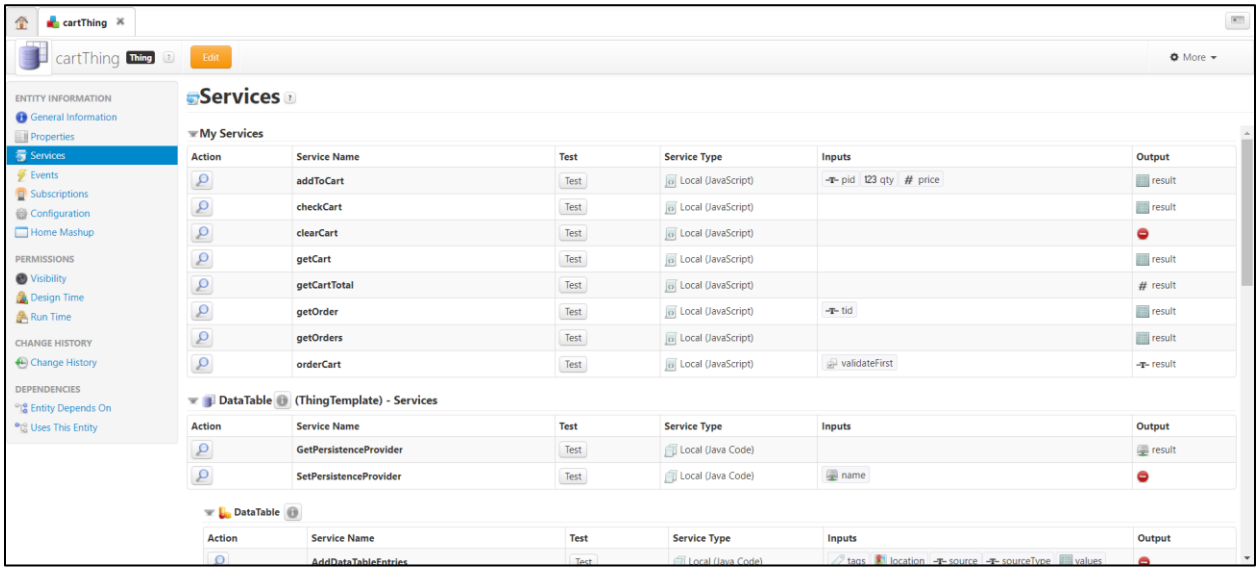
Like in Metadata 302, ThingWorx will be connected to Vuforia Studio. This time, in addition to the **shoppingThing**, a new Thing has been added, **cartThing**. **cartThing** consists of 8 services: addToCart, checkCart, clearCart, getCart, getCartTotal, getOrder, getOrders, and orderCart. Each of these services serves a different functionality, but all come together to create a persistent cart inside ThingWorx.

1. Download the **cartEntity.twx** file that has been included with this section.

2. Follow the instructions for importing and exporting files into ThingWorx Composer from the [PTC Support website].
3. Open **cartThing** once it has been added into your ThingWorx instance.
4. The **General Information** tab will include general information about the Thing. in this case, the **Name, Description, Tags,** and **ThingTemplate** for the Thing are included. **Tags** are used to group or categorize ThingWorx entities and **ThingTemplates** are used to create a new Thing based on a common base and functionality.



5. Open the **Services** tab to view all services associated with cartThing.

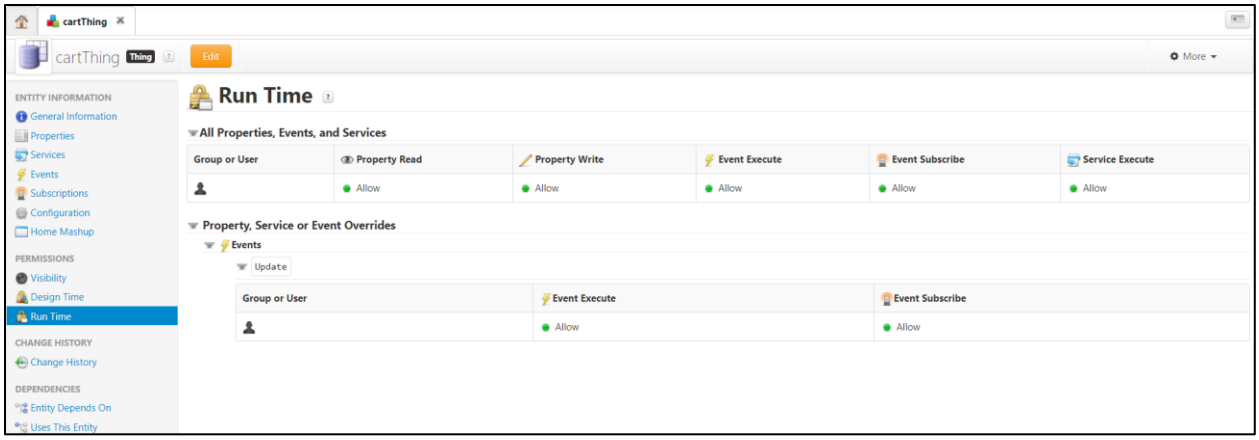


a. The **addToCart** service works much like the `addToCart` function that you created in Vuforia Studio. It intakes the part number (pid), quantity (qty) and price (price) of the part that has been clicked on. Using that input

information, along with user information from ThingWorx, it creates a user ID as a way of designating the person using the service. This user ID (uid) is a unique identifier for each order and is necessary for persistence because it gives the ability to tell the user and their individual sessions apart from others. When a part is added to the cart, its part information is stored inside an entry in a data table, which is then sent into the infotable used to store the cart.

b. Once a cart has been created, the **getCart** service is used to obtain the entries for the cart. It takes the information that was added to the cart and outputs it as an infotable. This will be used to update the cart when changes are made to it in the Vuforia Studio experience.
c. **getCartTotal** works like **getCart**, except instead of returning an entire table of information, its only output is the total price of the items in the cart.
d. When values are input into **addToCart**, there may be mistakes, like in the event of an incorrect price quote for a customer, so the **checkCart** service is used to validate all product information that was input into **addToCart**. If any information is incorrect, **checkCart** will correct that information inside the cart.
e. **clearCart** works exactly like its name suggests, it clears all items from the cart.
f. Like **clearCart**, **orderCart** is also relatively self-explanatory. When invoked, this service will place an order for all the parts that are in the cart. You are given the option of checking a box to validate the information in the cart using **checkCart**. After this service is invoked, a transaction ID (tid) is created with the user's name and number order that it is, and the **purchased** property of the infotable is changed to true to signify that the part in the cart has been bought.
g. **getOrder** allows you to input a transaction ID (tid) and receive all the information about that specific order in a table.
h. **getOrders** skips the tid input and displays all orders that have been made from the cart in a table.

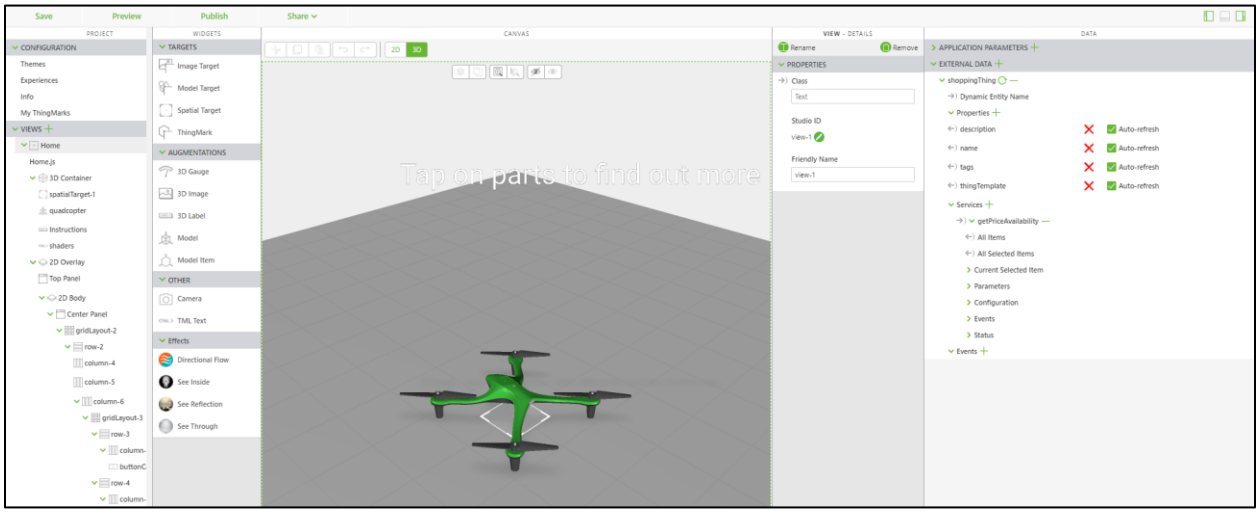
6. Open the **Run Time** tab under **Permissions** and make sure the account for your ThingWorx and Vuforia Studio instances has full access permissions for this Thing. This will enable the service to be called in Vuforia Studio.



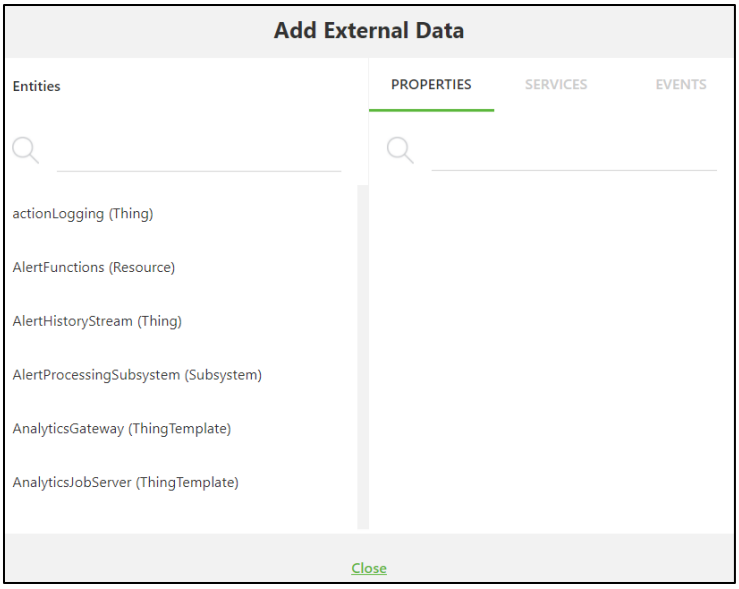
303.2 Add cartThing to Vuforia Studio

In the same manner that you added **shoppingThing** to Vuforia Studio in the previous exercise, **cartThing** will now need to be added so its services can be accessed inside Studio.

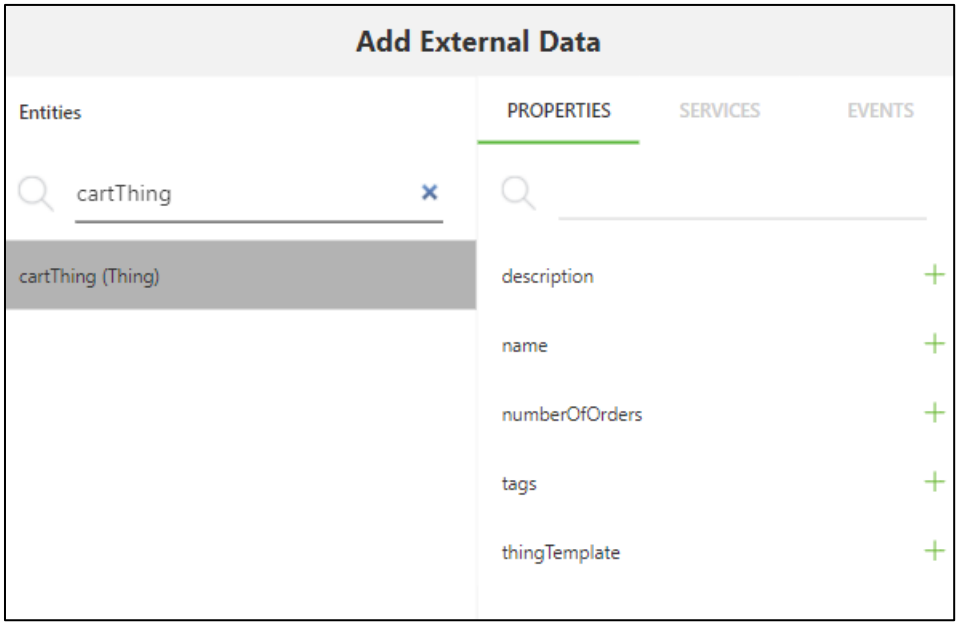
1. In your Vuforia experience, open the **Info** tab and ensure that the **Experience Service** URL is the same for your Vuforia Studio and ThingWorx instances. This is necessary for being able to connect the ThingWorx service into Studio.
2. In the **Home** tab, open the **Data** panel. Open the **External Data** dropdown if it isn't already open. You should see **shoppingThing** in the tab already.



3. Click the green **+** next to **External Data** to open a dialogue box for adding a new Thing.

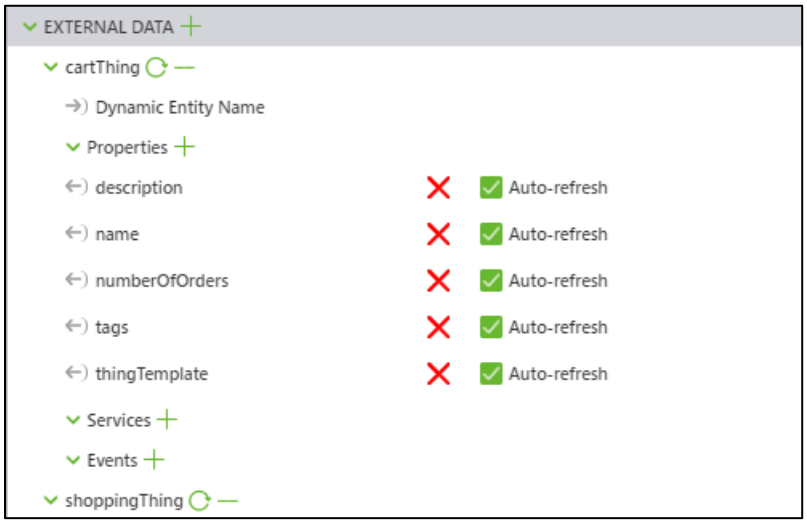


a. Type in *cartThing* into the **Entities** search bar. This will bring up a list of properties for the Thing.

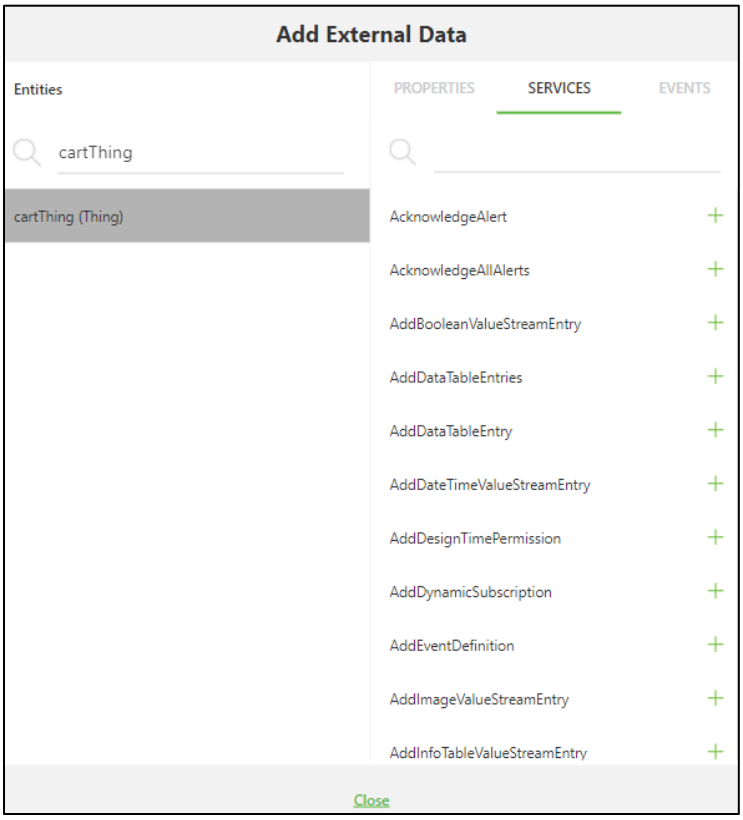


b. Click each of the **+** symbols next to the 5 options listed under **Properties** to add the properties to the Thing in Studio. Click **Close** after all properties have been added. Properties are general information about the Thing. The 4 properties that were also in **shoppingThing** work the same way as before; **name** is the name of the Thing and **description** is a description of what the Thing does. **tags** is a property for organizing things into certain categories and the **thingTemplate** is a template that is provided for generic, base Things to be created easily in ThingWorx. The new

property, **numberOfOrders**, stores an integer that represents the total number of orders that have been made in the system.

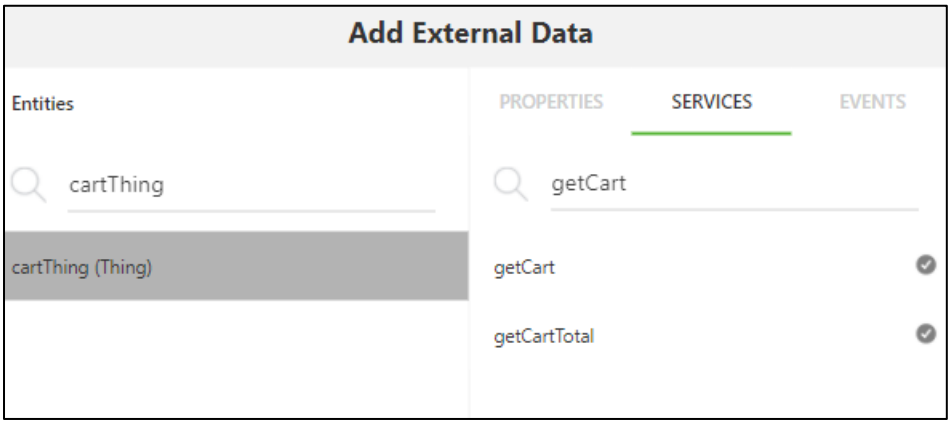


4. In addition to properties, services for **cartThing** need to be added. Click the green **+** next to **Services** to open a dialogue box for adding a new service.

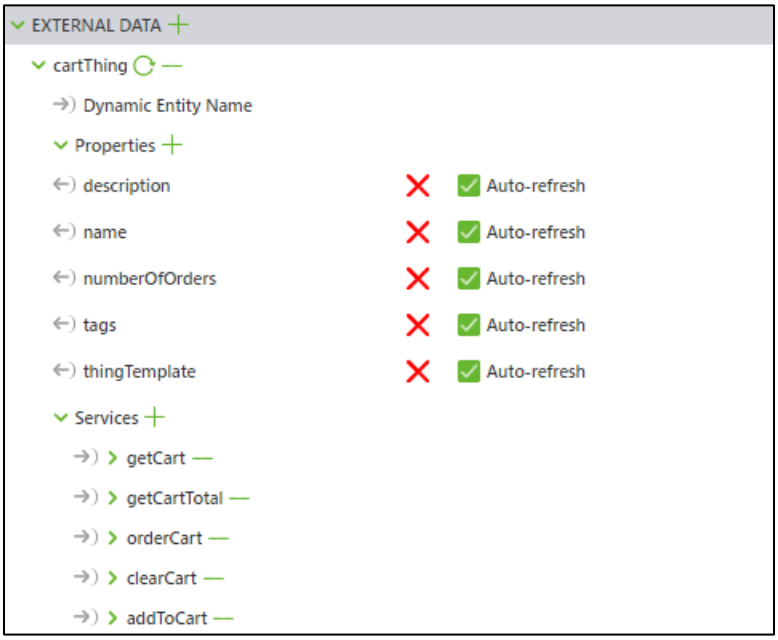


a. To find the correct services to add, type in *getCart* (both *getCart* and *getCartTotal* should appear when you do this), *orderCart*, *clearCart*, and

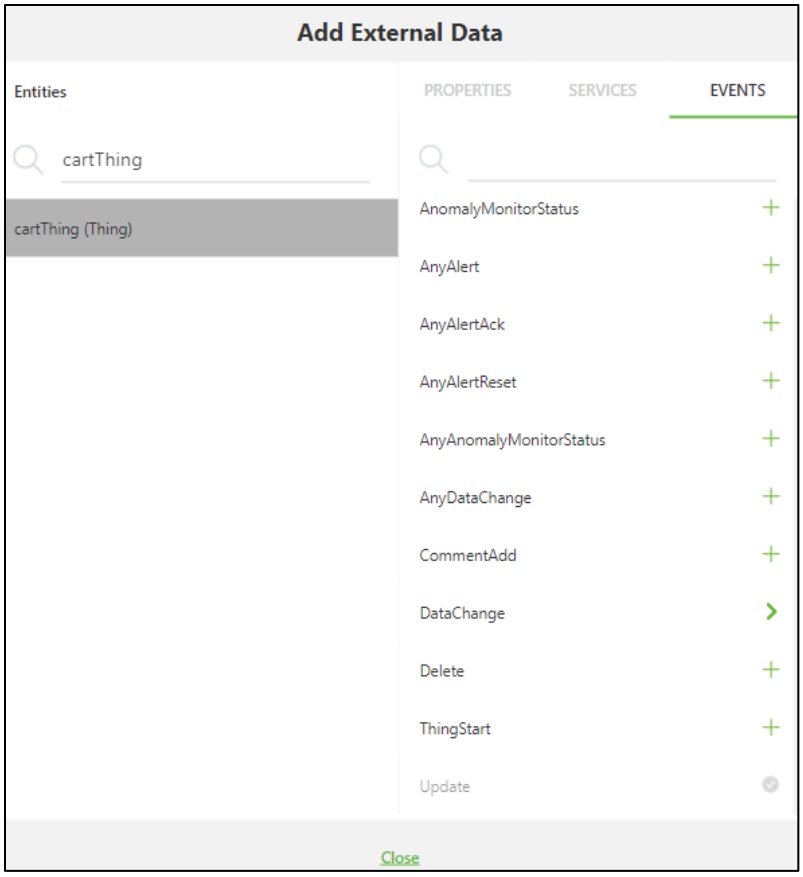
addToCart. Add each service by clicking the green **+** next to its name. Click **Close** once you have added all the services to the experience.



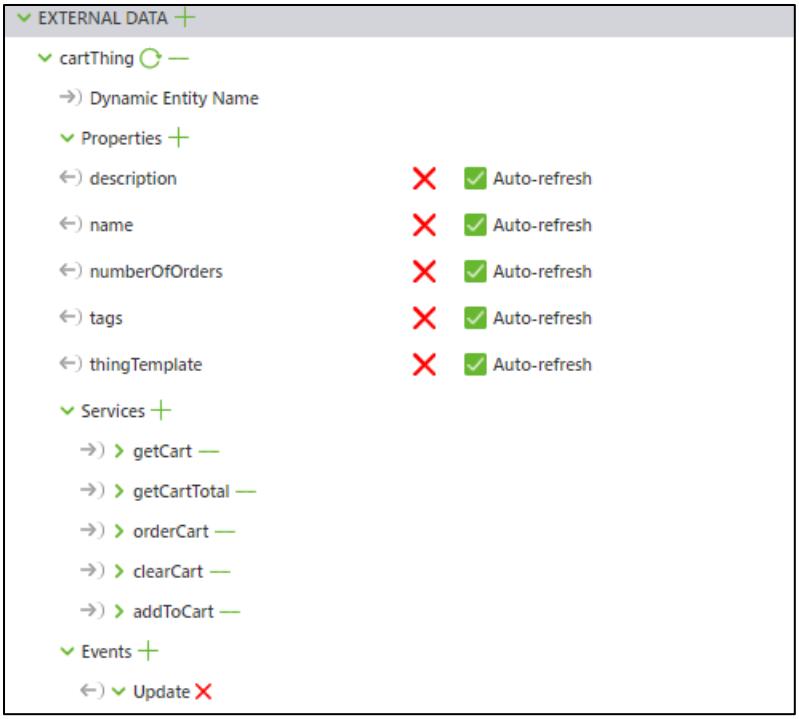
b. If completed correctly, **cartThing** will look like the following in the **External Data** tab (**Services** dropdown bars have been collapsed for visibility in the image).



5. This time, an event will need to be associated with **cartThing** as well. Click on the green **+** next to **Events**. Scroll to the bottom of the list of events and click the green **+** next to **Update**. This **Update** event will be triggered whenever the cart is updated, which you will hear more about shortly.



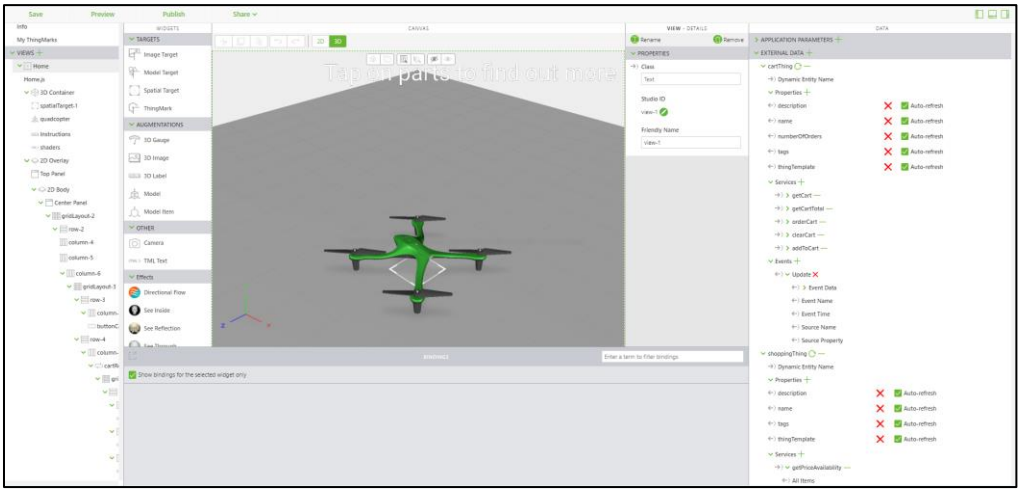
6. Now that the **Update** event has been added, **cartThing** has successfully been added to the experience. It should have 4 properties, 5 services, and 1 event available to it.



303.3 Binding cartThing and Editing the 2D Canvas

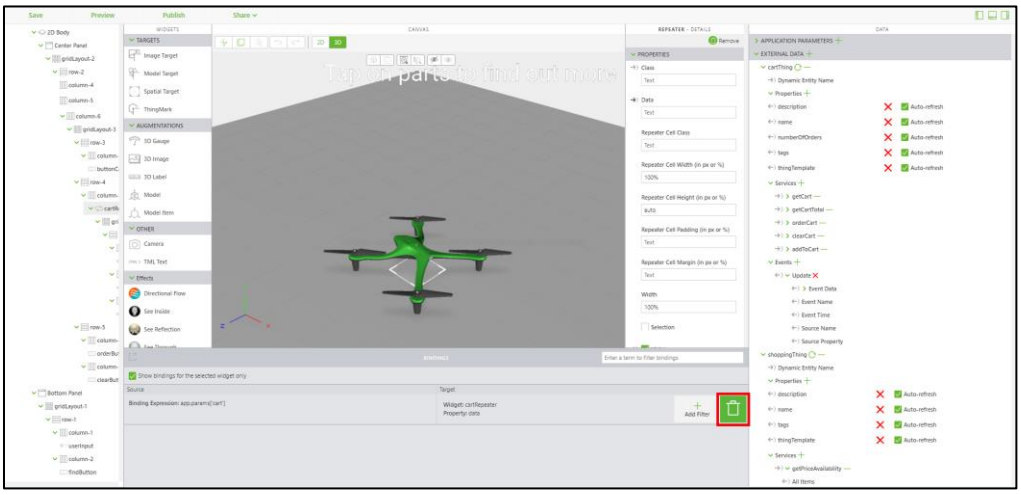
After **cartThing** has been added to the experience, its content needs to be bound to other properties and widgets for it to be effective. This section will walk you through that process, ensuring that ThingWorx and Vuforia Studio are properly connected. Some small changes will also need to be made to the 2D canvas to account for the new method of bringing in information.

1. Open the **Bindings** panel. This will allow you to edit property bindings.

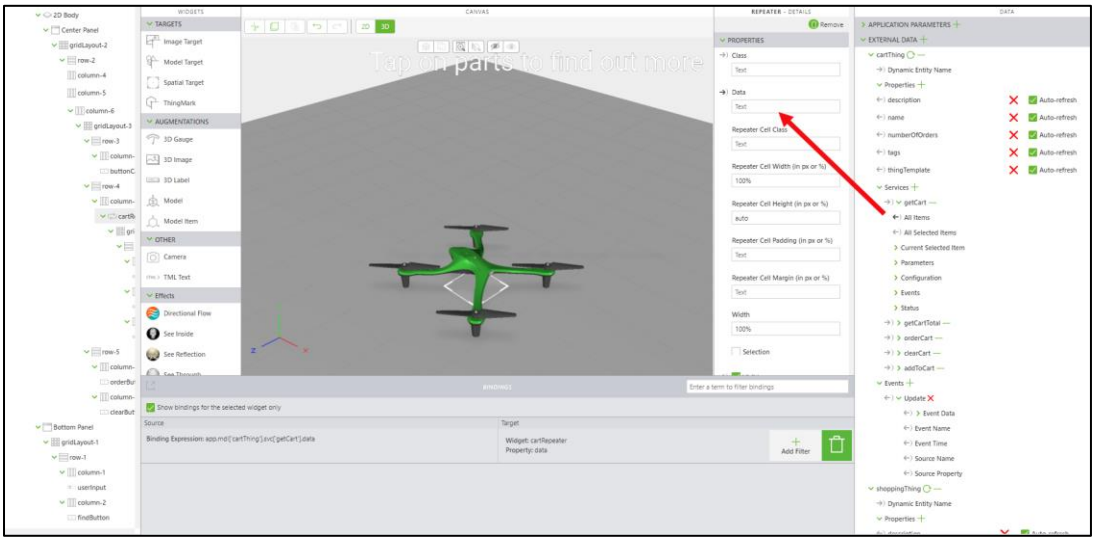


2. With the **Data** panel still open and **cartThing** visible in **External Data**, open the **cartRepeater** widget in **Home**.

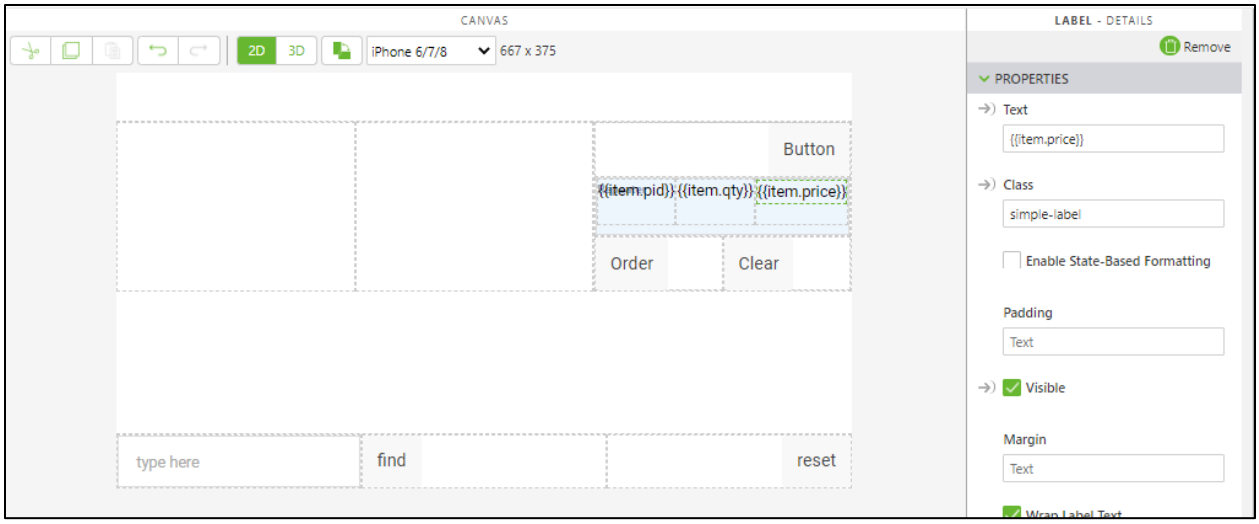
a. From previous activities, the **Data** property of **cartRepeater** has been bound to the **cart** application parameter. Click the green trash can icon in the **Bindings** panel to delete that binding.



b. Expand the dropdown for the **getCart** service. Drag **All Items** for the service onto the **Data** property of **cartRepeater** to bind them together. This binding will send data from **getCart** to the repeater so it can be displayed when the repeater is updated.

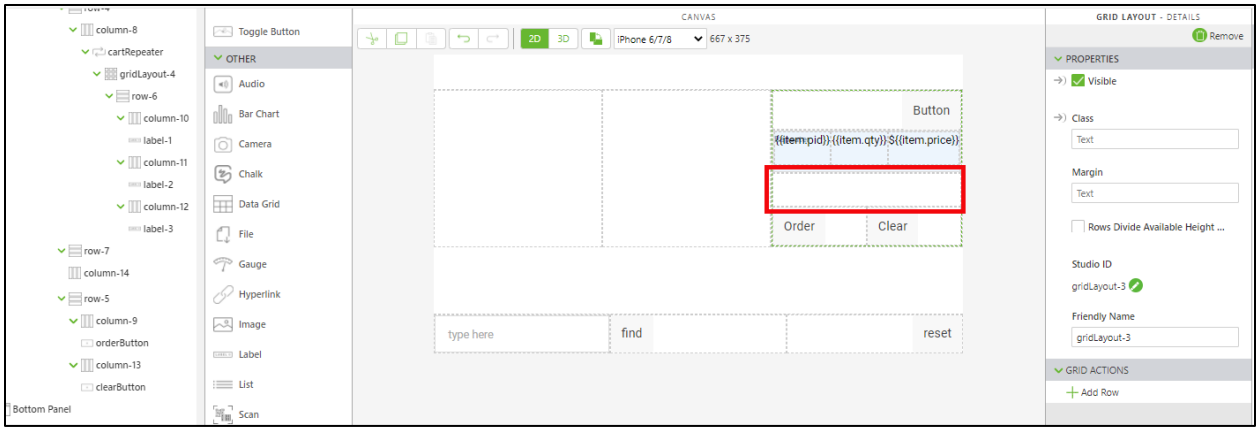


3. After editing the binding for the repeater, click **2D** to open the 2D canvas. Change the **Text** property for the 3 labels inside the repeater from *{{item.tag}}*, *{{item.count}}*, *${{item.prc}}* to *{{item.pid}}*, *{{item.qty}}*, *{{item.price}}*, respectively. For the columns that the qty and price labels are in, change their **Alignment** property to **End**. This change is necessary to match up with the wording of the services inside **cartThing**. The changes will be seen later in the experience when the cart is updated.

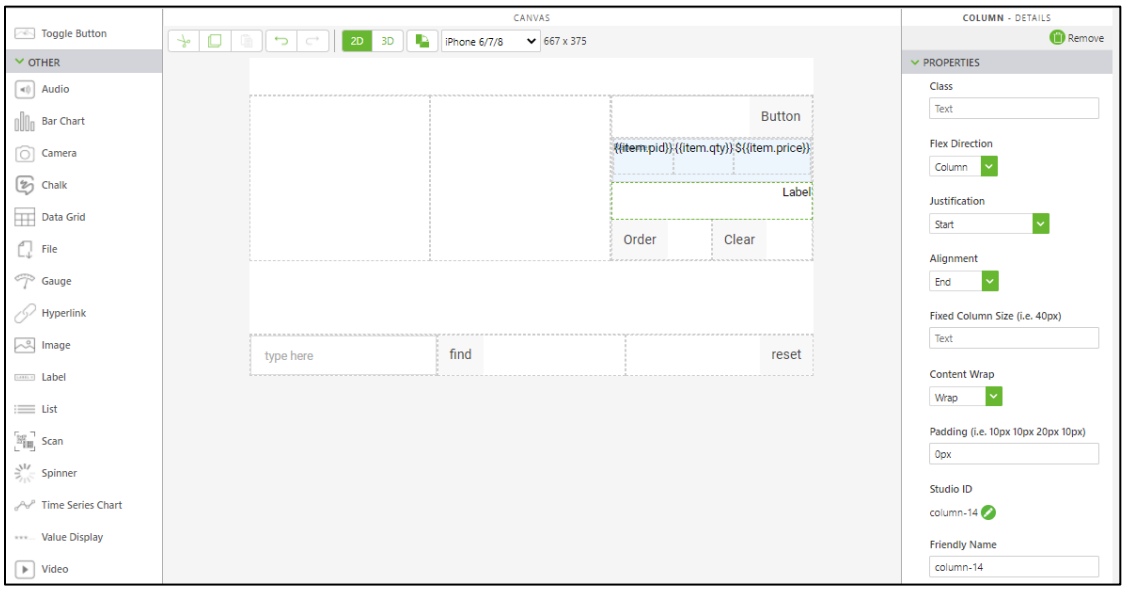


4. Open the **Events** dropdown inside **getCart**. Bind **ServiceInvokeComplete** to the **getCartTotal** service. This binding will ensure that whenever **getCart** is completed to get the contents of the cart, **getCartTotal** will be invoked to update the total of the cart.

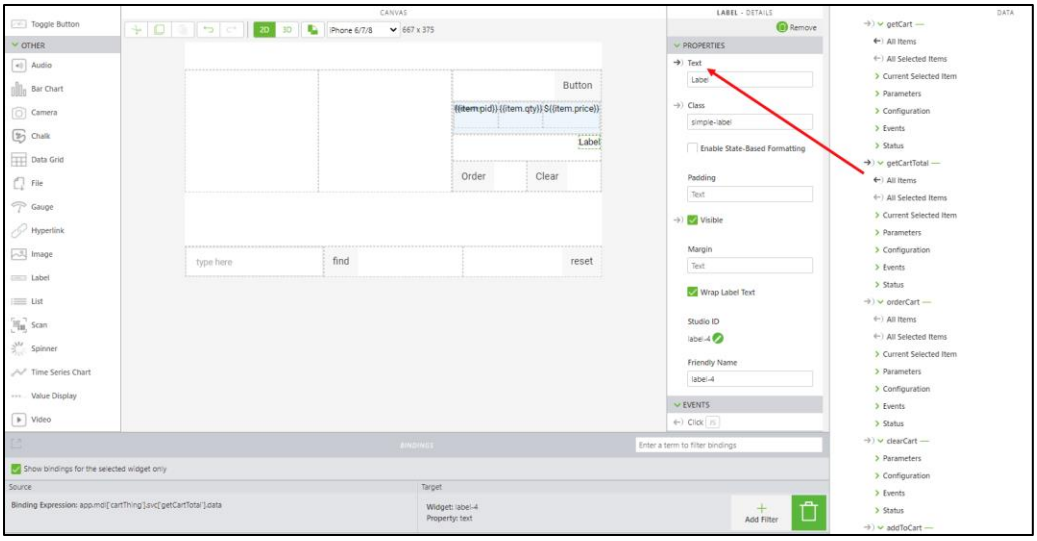
5. Click on any one of the rows inside the **Grid Layout** widget in the right column of the 2D interface and select **Add Row** to add another row to the layout. In the **Home** tree, drag the name for this newly created row above the name for the row that contains the **Order** and **Clear** buttons (for our case, the newly created row was named **row-7** and it was moved above **row-5**)



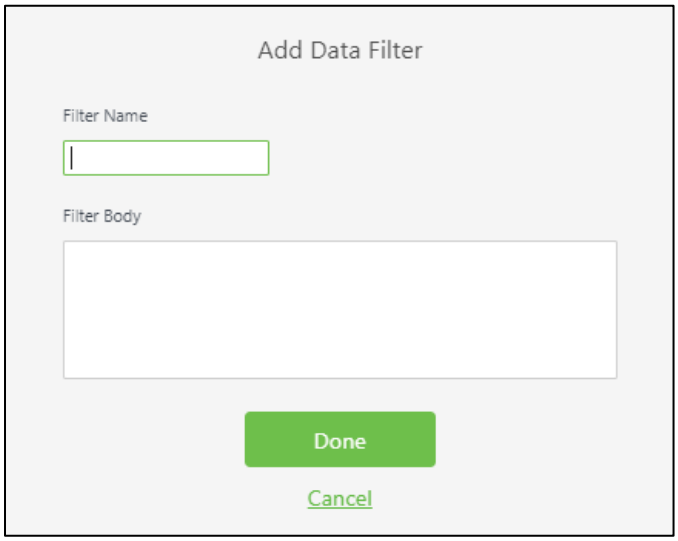
a. Drag a **Label** widget into this newly created row. Click into the column that your label was placed into and change the **Alignment** property dropdown to **End** so the label is right aligned.



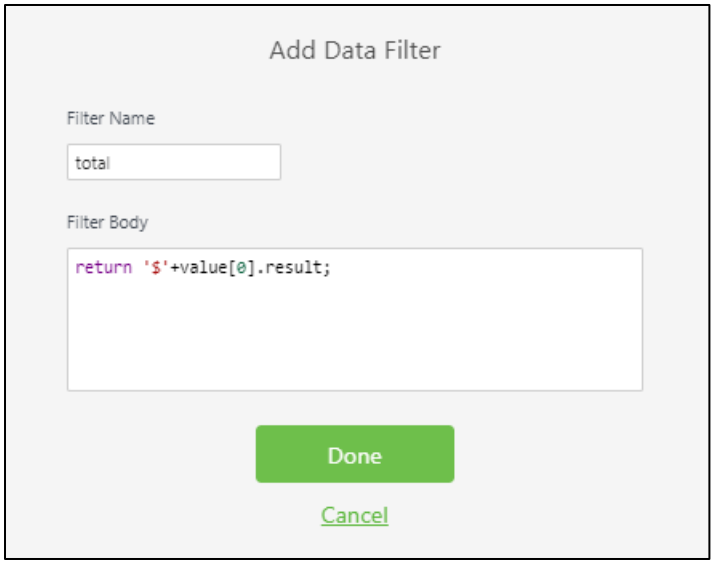
b. Bind **All Items** from the **getCartTotal** service to the **Text** property of the label that was just placed. This will be used to display the total amount of the cart below all the items, as opposed to how it was set up previously with the total being in the **buttonCart** button above the cart.



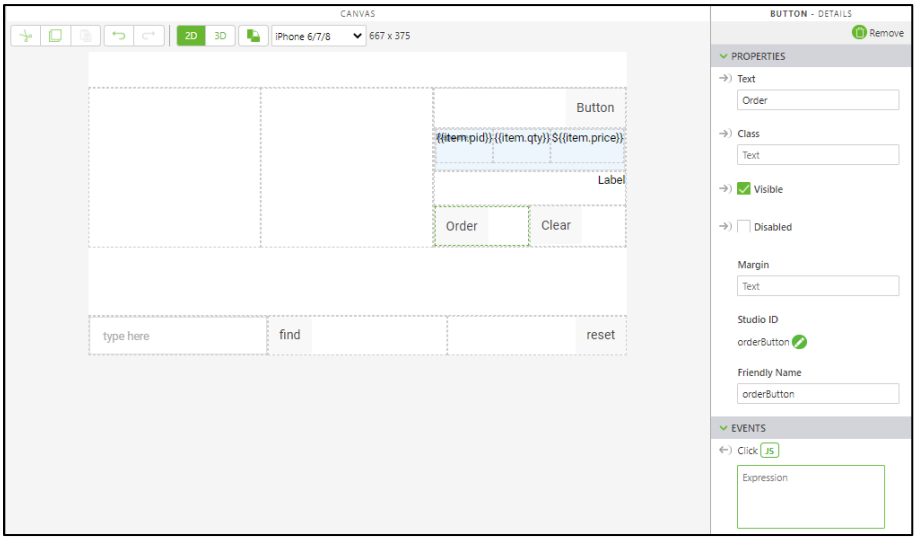
c. A filter is going to need to be added so that when the total is displayed in the label, it has a $ in front of it. A filter is used instead of typing out the $ in the **Text** property because this information is coming from a ThingWorx service. In the **Target** box for the binding between **getCartTotal** and **Text** for the label, click **Add Filter**. This will open a dialogue box for editing information about a filter.



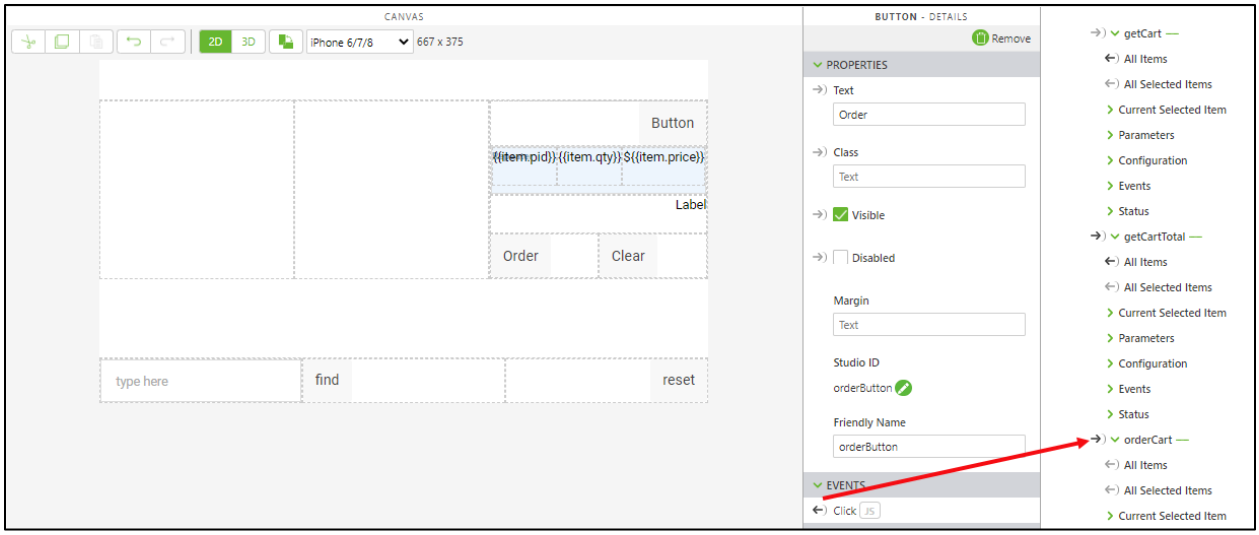
d. Type *total* in the **Filter Name** box. In the **Filter Body** box, add the following line of code: `return '$'+value[0].result;`. this filter will take the value that is output from **getCartTotal** and add a $ in front of it. Click **Done** when you have completed this step.



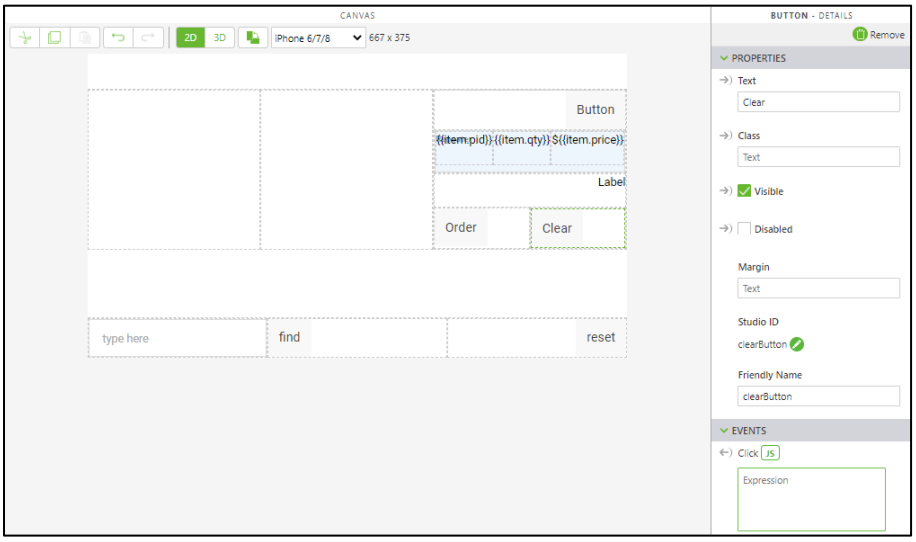
6. Open the properties for the **Order** button.
    a. Previously, you had created a function for ordering and called that function in the **JS** section of the **Click** event. Remove the text from the **JS** section so it is blank, since the `orderCart` function will be brought in from a ThingWorx service instead of a function in **Home.js**.



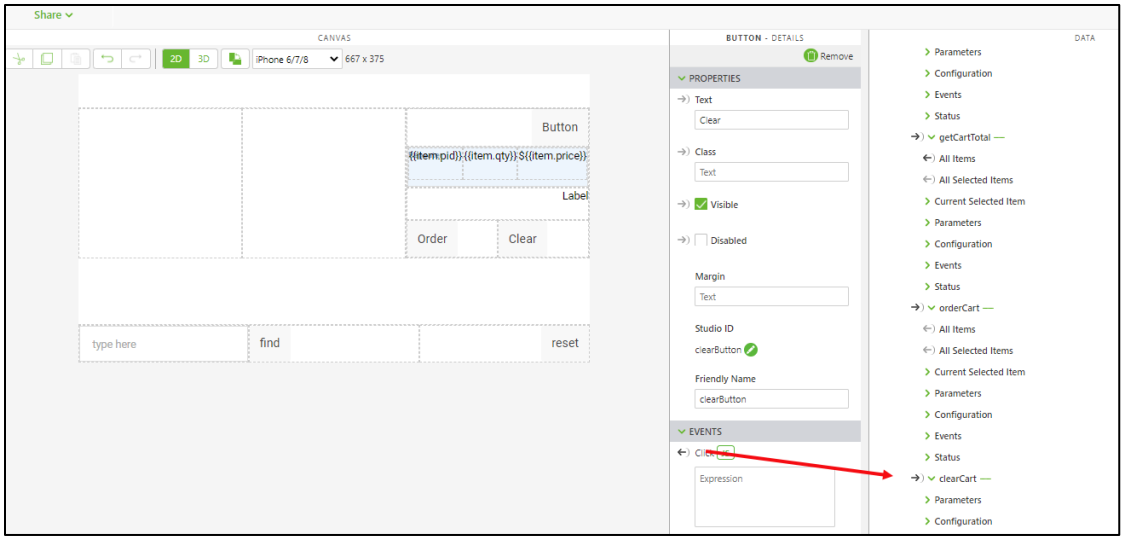
    b. Bind the **Click** event for the button to the **orderCart** service that was added. Now, when the **Order** button is clicked on, the **orderCart** service will be invoked in ThingWorx and order the cart.



7. Like the **Order** button, the same action will be taken with the **Clear** button.
    a. Open the **Clear** button and remove the text in the **JS** section of the **Click** event since the **clearCart** service is going to be used.



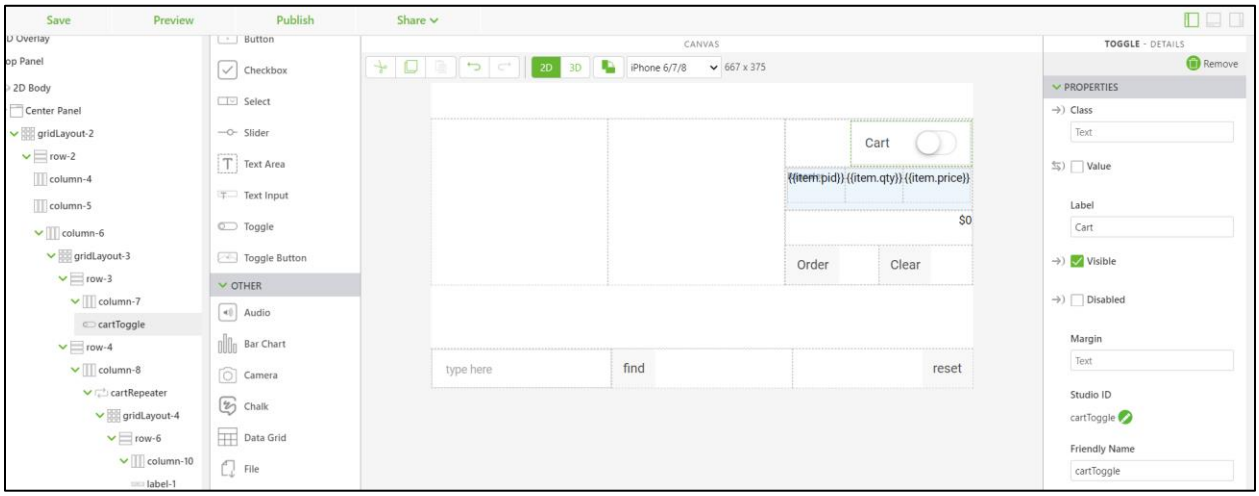
    b. Bind the **Click** event to the **clearCart** service. Clicking the **Clear** button in the experience will now clear all items from the cart.



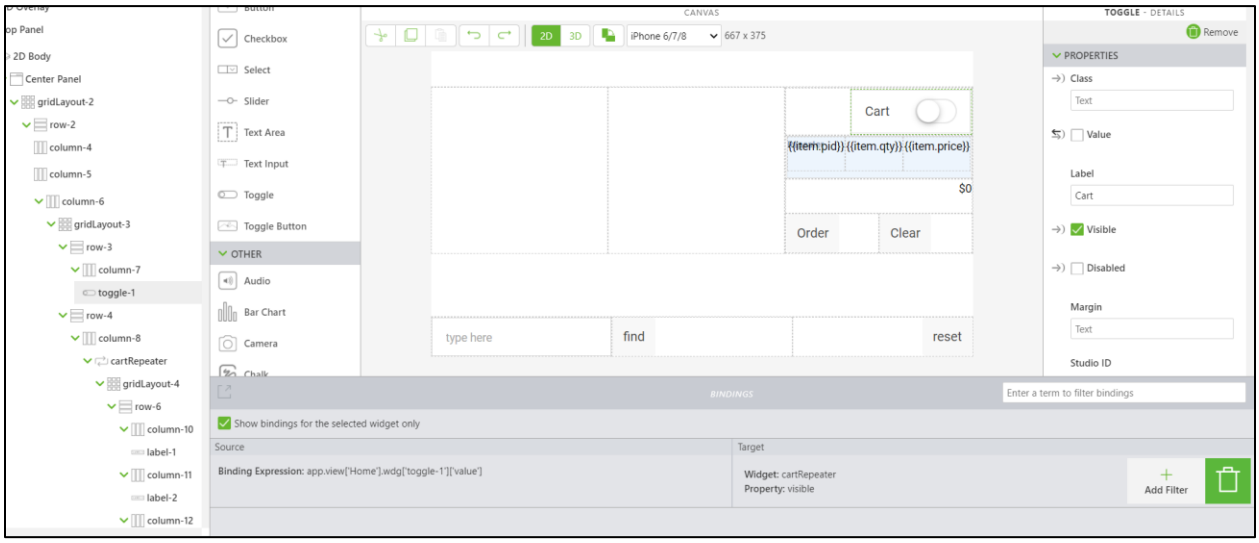
8. As the final binding, bind the **Update** event for **cartThing** to **getCart**. By creating this binding, whenever an update is made to the cart, whether it be an item added or the cart ordered/cleared, **getCart** will be triggered to update the displayed cart in the experience.

9. For ease of visibility, a **Toggle** widget will be added to the 2D canvas of experience in the place of the **buttonCart** button widget.
    a. Remove the **buttonCart** button.

b. Add a **Toggle** widget to the column where that button previously was. Change its **Label** property to say *Cart* and its **Studio ID** to *cartToggle*.



c. Drag the binding for the **Value** property onto the **cartRepeater** widget. When prompted, choose to bind the **Value** property to the **Visible** property of the repeater. This will allow the toggle to turn the visibility of the repeater on and off at your discretion.



303.4 Invoking the addToCart Service, Accounting for Welded Parts, and Cleaning Up the Code

Once the bindings have been created, the last step to creating this persistent cart is calling the **addToCart** service using the popup that appears when a part is selected. This is what will lead to the part being added to the cart inside Thingworx. This section will also make a callback to Metadata 101, when you combined the battery and PCB into the electronics part. Additionally, code that is not needed anymore will be deleted to keep the code as clean as possible.

1. Open **Home.js**.
2. In order to clean up some code, the `priceInfo` application parameter will be deleted from the code. It is not needed anymore since the total of the cart will be calculated in ThingWorx.
3. With `priceInfo` not part of the code anymore, the price variable needs to be changed to `$scope.price` to make it accessible throughout the code. Additionally, the `'$'` can be removed from the conditional statement, as that will be added in the text of the popup itself and in the label filter.

```
// price is going to be the variable that is referenced in the popup, while the app parameter priceInfo will be
$scope.price = rowData.avail === true ? rowData.price : 'UNAVAILABLE';

// create a variable to bring the $scope.meta object into this event listener as a local object
```

   a. Inside the template for the popup, change the `price` input for `setTemplate` to be `$scope.price` to account for its new scope.

```
template: $scope.setTemplate(meta, $scope.price),
```

```
//call the function for setting the template
template: $scope.setTemplate(meta, $scope.price),
```

   b. Navigate to the `setTemplate` function. In the `addTo` variable, add a `'$'` to the variable so the popup will display the dollar sign before `price`.

```
var addTo = price != 'UNAVAILABLE' ? '$' + price + ' </div><div class="btnadd" ng-
click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' :
  price ;
```

```
// if price != unavailable, define an add to cart button and have the price displayed in the popup, if it is una
var addTo = price != 'UNAVAILABLE' ? '$' + price + ' </div><div class="btnadd" ng-click="hiliteOff();popup.
price ;
```

4. Scroll down to your `addToCart` function.
   a. The **addToCart** service has the same functionality as the `addToCart` function, so remove the entire body of the function so it just looks like the code below.

```
// function for adding a selected part to the cart
$scope.addToCart = function () {

} //end of addToCart function
```

```
// function for adding a selected part to the cart
$scope.addToCart = function () {

} //end of addToCart function
```

   b. Use the triggerDataService function to call the addToCart service from ThingWorx. Use the variables below for the pid, qty, and price inputs
5. Remove the `orderCart` and `clearCart` functions entirely, since those have been transitioned to be services in ThingWorx.
6. Also, remove the code for creating the `cartButton` application parameter and the `cart` object, as those are not necessary anymore.
7. In order to access the pricing data for the electronics combined part, logic needs to be implemented to bring the occurrence data up a level in its structure. This will allow for data about the electronics part to appear in the popup when you click on either the PCB or battery.
   a. Add this code above `$scope.meta`.

b. The code below works in the following way; a variable named `welding` is created that triggers the `while` loop to occur. Inside the `while` loop, a variable named `sbominfo` is created, which is set with the value of the sBOM_Welded attribute. If `sbominfo` has information in it and the sBOM_Welded attribute had a value of true, then the index of the occurrence will be rolled back a level to the parent part of the selected parts. In other words, since electronics is considered to be the parent part of the PCB and battery, whenever the PCB or battery is selected, the occurrence path (`pathId`) will become the path of the electronics part. If the selected part does not have the sBOM_Welded attribute, then `welding` is set to `false` and the `while` loop ends.

```
    // set welding = true until it is turned false
    var welding = true
    // while welding is true
    while (welding) {

      // create a variable named sbominfo which will store the value of the sBOM_Welded
attribute
      var sbominfo = metadata.get(pathId, 'sBOM_Welded');

      // if sbominfo has information and its value is true
      if (sbominfo != undefined && sbominfo === 'true') {

        // trace back the index of the occurrence path up a level
        var child = pathId.lastIndexOf('/');
        // if the index of the occurrence path at its current position is equal to 0,
return the pathId of the level above
        if (child === 0)
          return;

        pathId = pathId.substr(0, child);

        //else, set welding to false
      } else {
        welding = false;
      } // end of sbominfo if statement
    } //end of welding while loop
```

```
12
13      // set welding = true until it is turned false
14      var welding = true
15      // while welding is true
16      while (welding) {
17
18        // create a variable named sbominfo which will store the value of the sBOM_Welded attribute
19        var sbominfo = metadata.get(pathId, 'sBOM_Welded');
20
21        // if sbominfo has information and its value is true
22        if (sbominfo != undefined && sbominfo === 'true') {
23
24          // trace back the index of the occurrence path up a level
25          var child = pathId.lastIndexOf('/');
26          // if the index of the occurrence path at its current position is equal to 0, return the
27          if (child === 0)
28            return;
29
30          pathId = pathId.substr(0, child);
31
32          //else, set welding to false
33        } else {
34          welding = false;
35        } // end of sbominfo if statement
36      } //end of welding while loop
37
38      //create an object with the properties below that are based on attribute names from Creo Illu
    for this occurrence.
39      $scope.meta = {
```

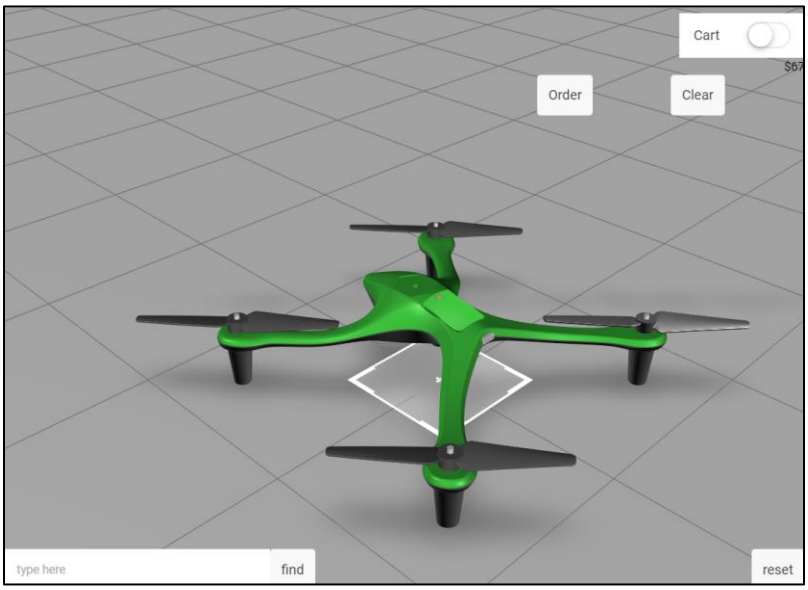
c. To see how this works, click **Save** and then **Preview**. Click on the end of the PCB. Notice that the popup now has the part name, display name, and price of the electronics combined part, as opposed to before when there was not a price displayed and the information for the PCB appeared in the popup.

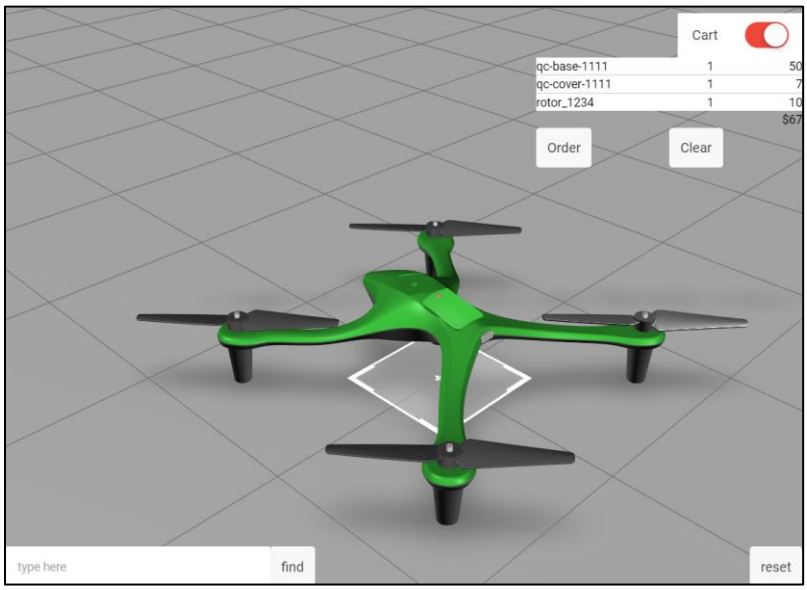
8. The full code for this section can be found in Appendix 1.

303.5 Using the Experience

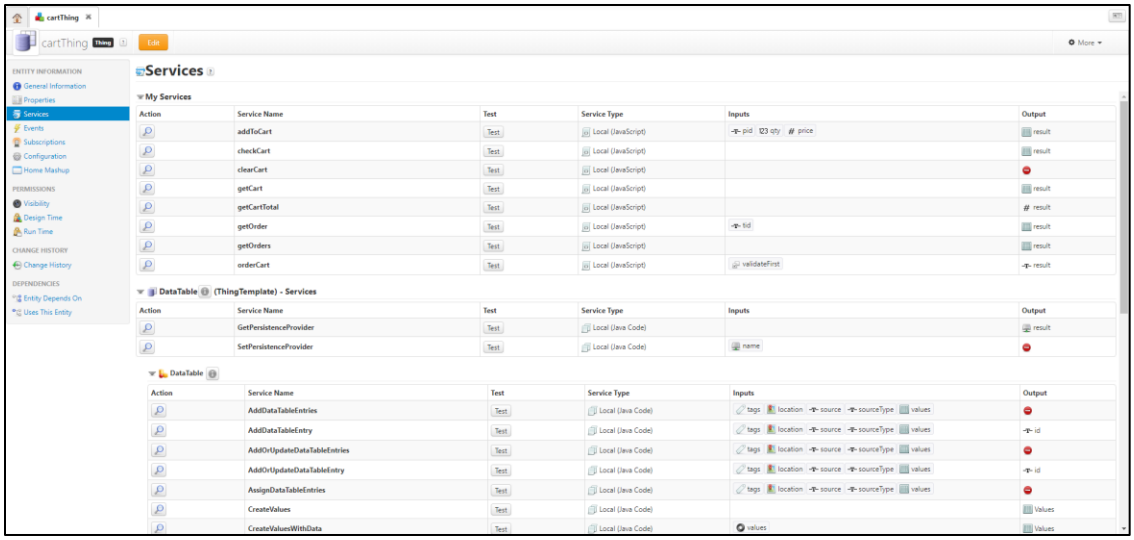
Now that all the code has been cleaned up and the Vuforia experience is connected to ThingWorx, you will be able to see how the two systems work together.

1. Open **Preview** and add a few parts to your cart. Notice that the label for the total of the cart changes, but the cart does not appear. Toggle the **cartToggle** widget on to observe how the repeater adds the parts.

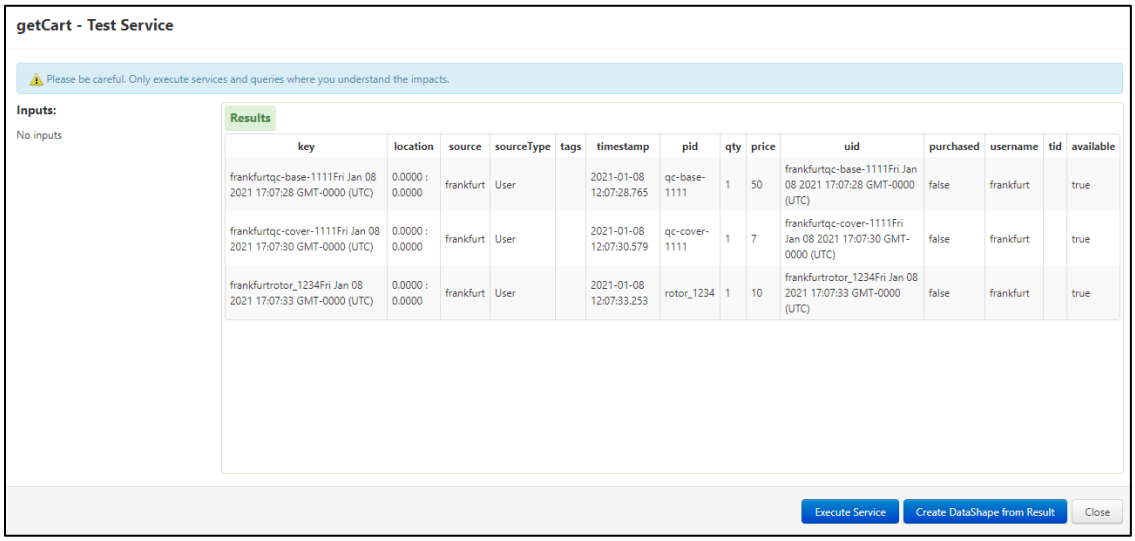




2. Open your ThingWorx instance in another tab and navigate back to the **Services** tab for **cartThing**.

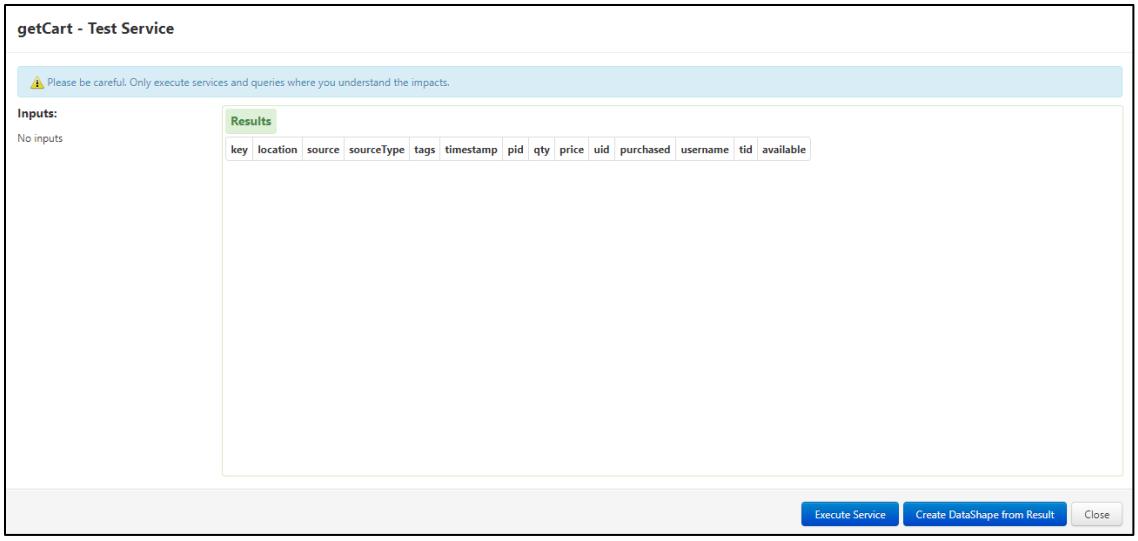


a. Click the **Test** button for **getCart**. This will allow you to run the service inside ThingWorx and view the current cart. In the window that appears, click **Execute Service** to make the cart appear. Compare your cart in ThingWorx to the repeater in Studio; notice that the two have the same items in them. Notice that **purchased** is false for all items, this is because they are in the cart, but they have not been purchased yet.



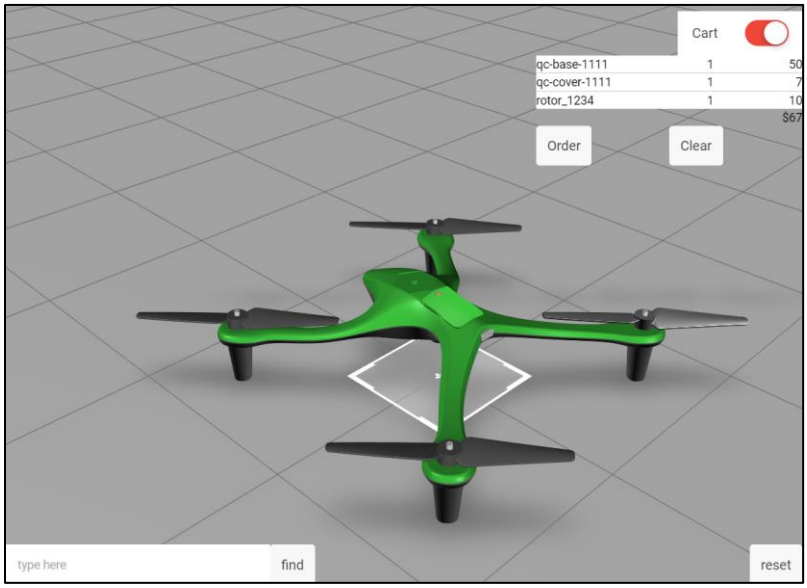
b. Click **Close** to exit the window.

3. Go back to the Studio preview tab and click the **Clear** button. You should see the repeater be cleared and the total of the cart go to **$0**.
    a. In ThingWorx, test **getCart** again. The cart should now appear as empty.

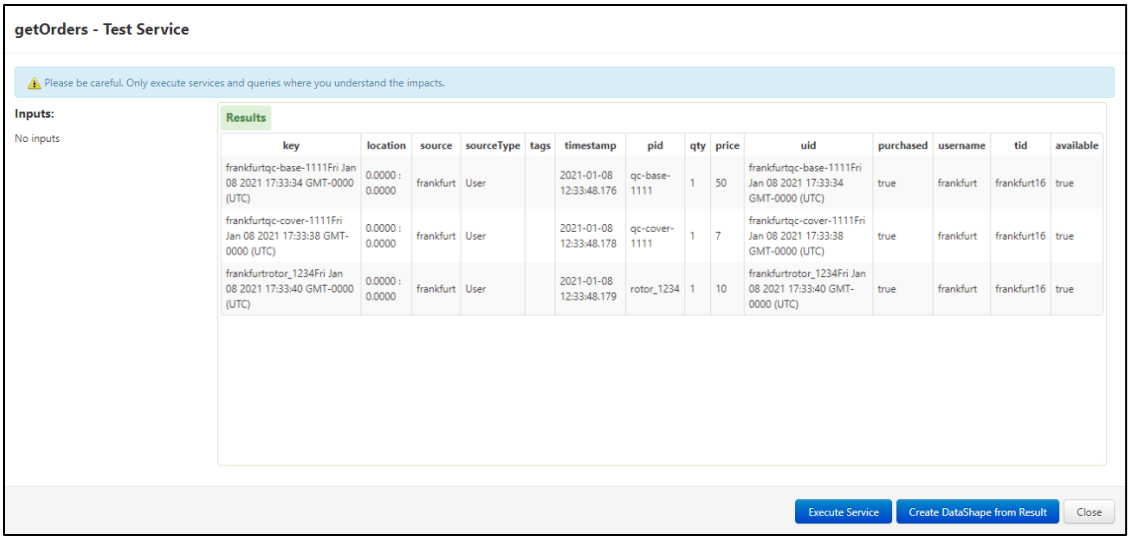


b. Click **Close** to exit the window.

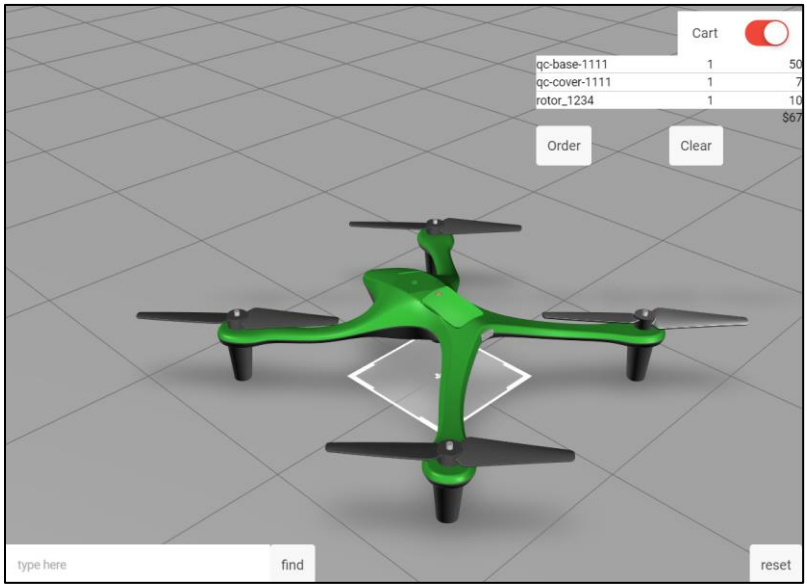
4. In Studio, add the same items back to the cart.



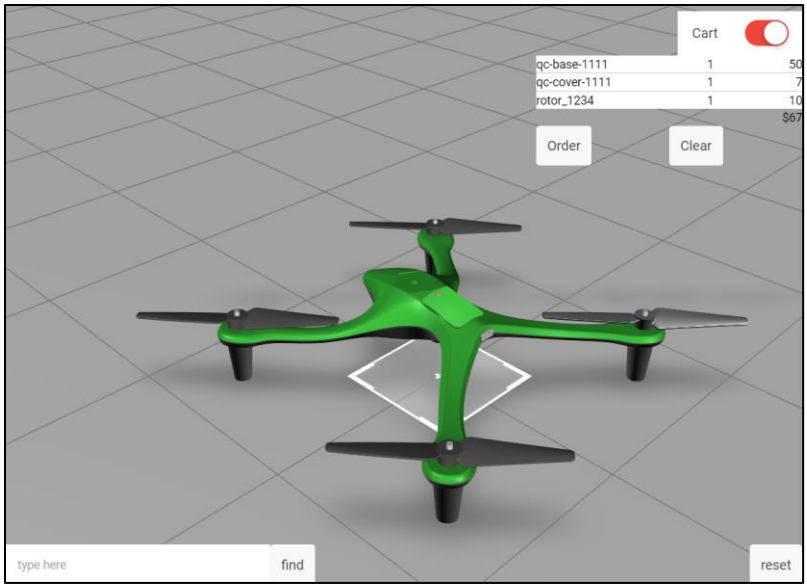
5. This time click the **Order** button. You should see the cart clear once again.
6. In ThingWorx, click the **Test** button next to the **getOrders** service. Click **Execute Service** in the window that appears. You will see your order appear in the list. If you have made multiple orders, then multiple orders will be listed. Notice that the **purchased** property is now set to **true** because you purchased the item out of the cart, the there is a value in the **tid** column for the transaction with your login number and the order number.



7. Continue to test out the services. In Studio, add the same parts that you added before to the cart.



   a. This time, after doing that, close out of Vuforia Studio and then open it back up again. Open the preview of the experience and add a new part. You will see that since you did not clear your cart or order the parts before closing the window that they stayed in the cart. This is exactly how persistence works.



8. If the persistent cart is working, then you have completed this tutorial and our Metadata series!

Appendix 1: Section 303.4 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. Create variable for the currently selected part
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId

    // set welding = true until it is turned false
    var welding = true
    // while welding is true
    while (welding) {

      // create a variable named sbominfo which will store the value of the sBOM_Welded
attribute
      var sbominfo = metadata.get(pathId, 'sBOM_Welded');

      // if sbominfo has information and its value is true
      if (sbominfo != undefined && sbominfo === 'true') {

        // trace back the index of the occurrence path up a level
        var child = pathId.lastIndexOf('/');
        // if the index of the occurrence path at its current position is equal to 0,
return the pathId of the level above
        if (child === 0)
          return;

        pathId = pathId.substr(0, child);

        //else, set welding to false
      } else {
        welding = false;
      } // end of sbominfo if statement
    } //end of welding while loop

    //create an object with the properties below that are based on attribute names from
Creo Illustrate for this model. use metadata.get to obtain the data from the JSON
properties for this occurrence.
    $scope.meta = {
      partName: metadata.get(pathId, 'Display Name'),
      instructionName : metadata.get(pathId, 'illustration'),
      partNumber: metadata.get(pathId, 'partNumber'),
    } //scope.meta object

    // set itemName app parameter to be equal to the partName variable, same relationship
with itemNumber and partNumber. Set the itemCount to 1 for the purpose of this section.
    $scope.app.params.itemName = $scope.meta.partName;
    $scope.app.params.itemNumber = $scope.meta.partNumber;
    $scope.app.params.itemCount = 1;
```

```
    $scope.target = targetName

    // call the getPriceAvailability ThingWorx service based on partNumber
    twx.app.fn.triggerDataService('shoppingThing', 'getPriceAvailability', {pid:
$scope.meta.partNumber})

    }) //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

$scope.$on('getPriceAvailability.serviceInvokeComplete', function(evt) {
  //
  // variable holding all data for the current row in the infotable
  var rowData = twx.app.mdl['shoppingThing'].svc['getPriceAvailability'].data.current

  // price is going to be the variable that is referenced in the popup, while the app
parameter priceInfo will be used for adding the total in the cart
  $scope.price = rowData.avail === true ?  rowData.price : 'UNAVAILABLE';

  // create a variable to bring the $scope.meta object into this event listener as a
local object
  let meta = $scope.meta

  // adds an ionic popup when a part is clicked
  $scope.popup = $ionicPopup.show({
    //
    //call the function for setting the template
    template: $scope.setTemplate(meta, $scope.price),
    //
    // set the scope for the popup
    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: $scope.target, instruction: 'l-Creo 3D - ' +
meta.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
}) // getPriceAvailability end
```

```
//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

// function for adding a selected part to the cart
$scope.addToCart = function () {
  // call addToCart service from ThingWorx
  twx.app.fn.triggerDataService('cartThing', 'addToCart', {pid: $scope.meta.partNumber,
qty: 1, price: $scope.price});
} //end of addToCart function

// function for setting the template for the Ionic popup
$scope.setTemplate = function (meta, price) {
  // if there is a disassembly sequence associated with the part, create a Disassemble
button in the popup, if not, no button will appear
  var instr = meta.instructionName.length > 0 ? '<div class="btndisassemble" ng-
click="hiliteOff();popup.close();disassemble();">Disassemble</div>' :
  '';

  // if price != unavailable, define an add to cart button and have the price displayed
in the popup, if it is unavailable, just display price
  var addTo = price != 'UNAVAILABLE' ? '$' + price + ' </div><div class="btnadd" ng-
click="hiliteOff();popup.close();addToCart();">Add to Cart</div>' :
  price ;

  // build the template for the popup
  var template = '<div>' + $scope.app.params.itemCount + 'x  ' + meta.partNumber +
' </br>' +
                 meta.partName + ' </br>' +
                 addTo  +
```

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

  // return the template variable when this function is called
  return template
}
```