ptc

vuforia™ studio

Metadata 302

Add a Simple ThingWorx Service to Vuforia Studio

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

Intro

In Metadata 301, pricing attributes were added to the quadcopter model though bulk adding in Creo Illustrate. Most of the time, though, an attribute like pricing will not be directly baked into model itself and will be stored in a different location. In this case, the price of each part has been added into a table in ThingWorx that can be called into Vuforia Studio. This tutorial will explain how to take data from a ThingWorx experience, add in into Vuforia Studio, and then manipulate it within an existing Studio experience.

The following topics will be covered in this project. Jump to them with their hyperlinks:
Metadata 302.1 Using ThingWorx Composer
Metadata 302.2 Add a Thing to Vuforia Studio
Metadata 302.3 Call getPriceAvailability and Use the serviceInvokeComplete Event Listener

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

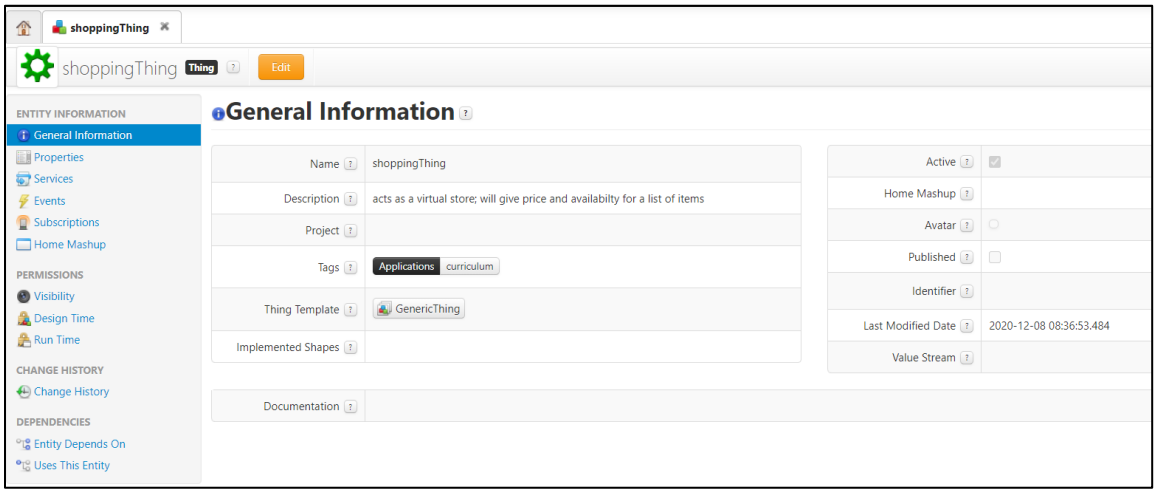
All important notes and UI areas are **Bold**.
All non-code text to be typed is *italicized.*
All code follows `this convention`
All code comments follow `this convention`

302.1 Using ThingWorx Composer

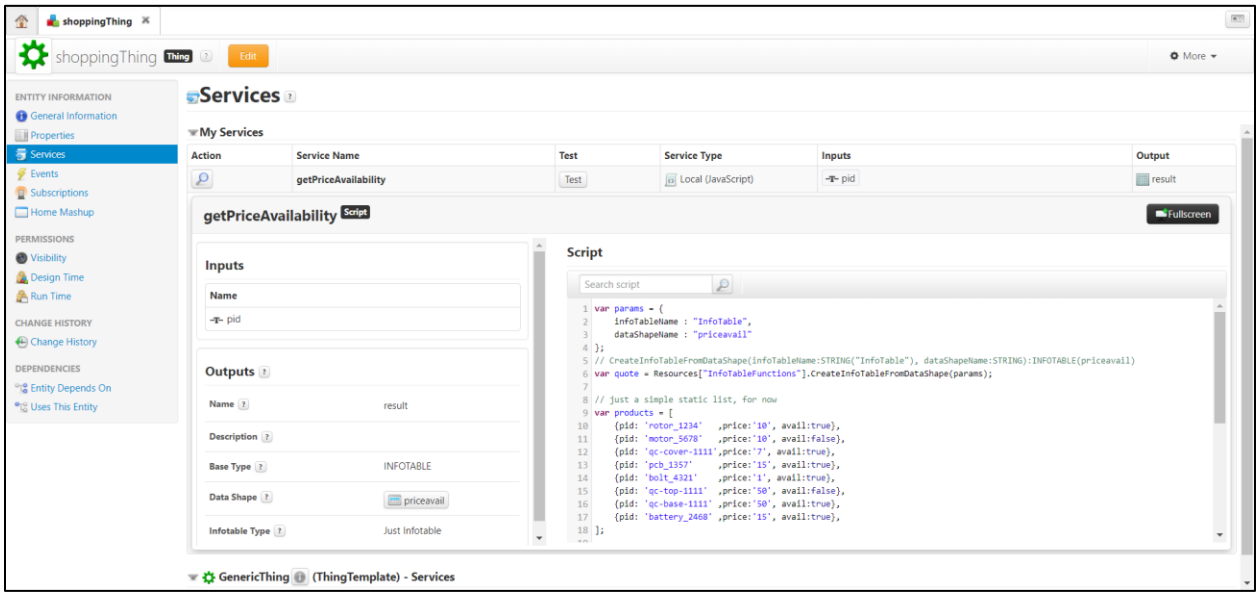
Things are digital representations of physical devices, assets, products, systems, people, or processes that have properties and business logic. In this case, the **shoppingThing** that has been created is representing a digital online parts store. It can be imported into ThingWorx Composer and then later used inside Vuforia Studio

1. Download the **ShoppingEntities.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 **shoppingThing** 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.



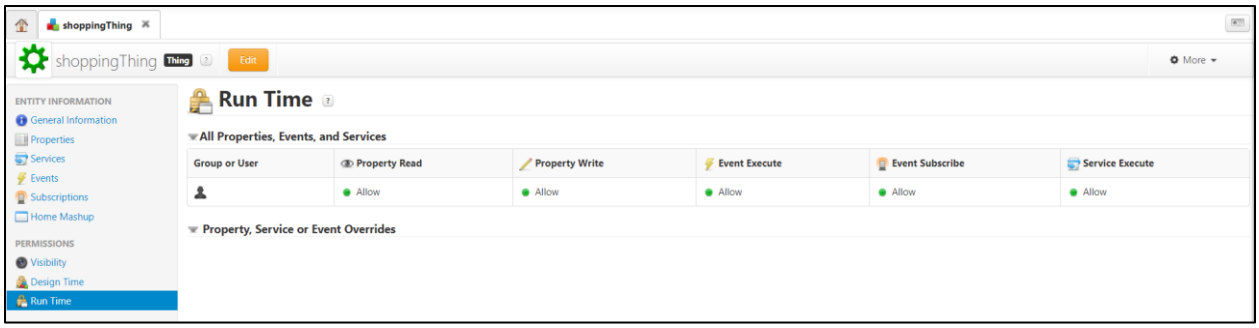
5. Open the **Services** tab under **Entity Information.** Services are functions that a Thing can perform. In this case, the **getPriceAvailability** service is associated with the **shoppingThing** Thing. This service takes an input part number for a selected part and outputs a set of values associated with an object that has the same part number used to represent each part in the quadcopter mode.



   a. Under **Inputs** is an input named **pid**. **pid** is a text string that will be associated with the part number of a part that is clicked on in the Vuforia Studio experience.
   b. In the **Outputs** section of the page is an output named **result**. This variable is an infotable, which is a datatable of values stored in ThingWorx.

c. The **Script** is where all code is written that will be triggered when the service is run. In this case the script starts by creating an empty infotable named `quote`. An array called `products` is then created, which has objects for each of the parts that include their part number (`pid`) and price as strings, and a Boolean called availability, which tells you if a part is available or not. An object called `newEntry` is created which will store the availability, price, and part number of the selected part. A `for` loop is added to the script to run through the `products` array, match the input `pid` to the `pid` property of one of the parts in the `products` array, and then update the `newEntry` object with the availability and price of the object. The information in the `newEntry` object is then added as a new row in the `quote` table, which is output as the `result` variable back into Vuforia Studio.

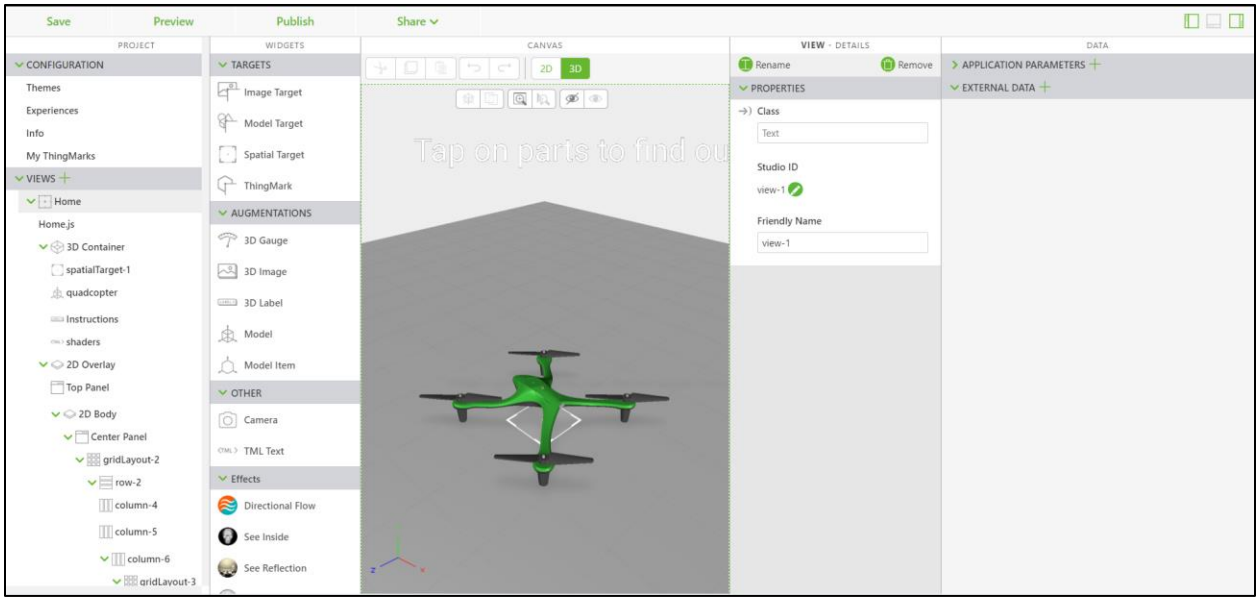
6. Open the **Run Time** tab under **Permissions** and make sure the account for your ThingWorx and Vuforia Studio instances has full access permissions for this Thing. This will enable the service to be called in Vuforia Studio.



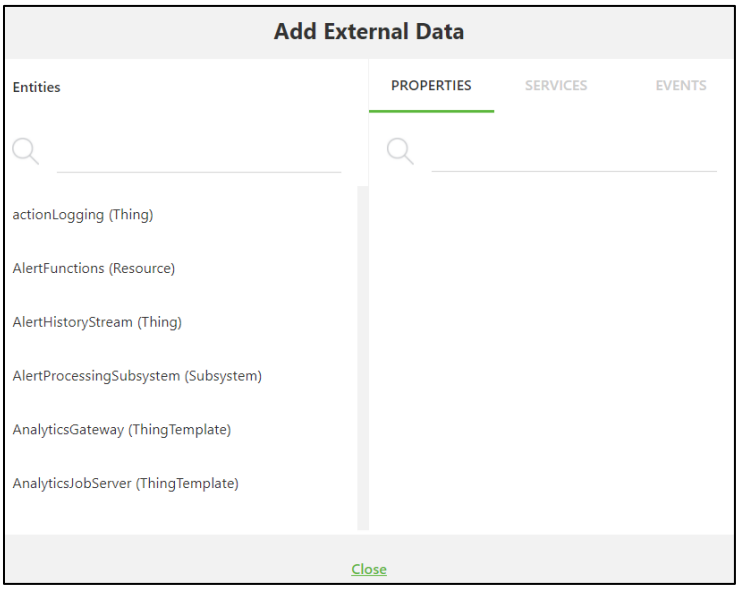
302.2 Add a Thing to Vuforia Studio

Once a Thing has been created in ThingWorx, it needs to be imported into Vuforia Studio for anything associated with it to be called inside Studio.

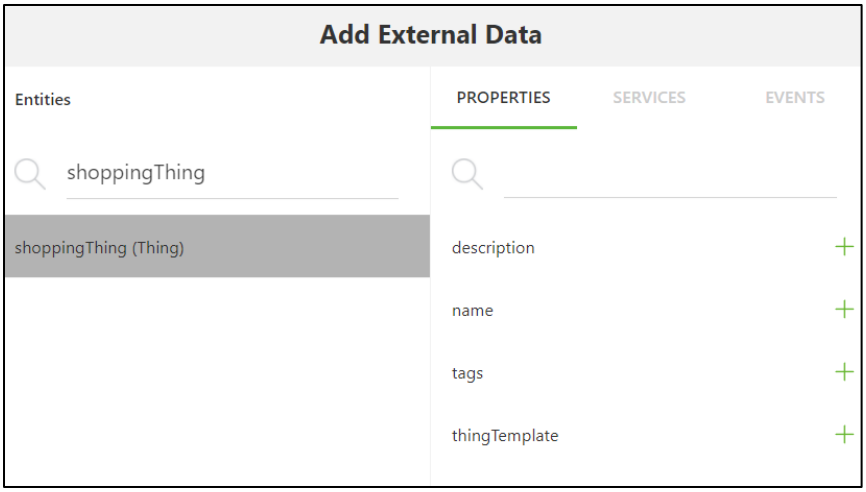
1. In your Vuforia experience, open the **Info** tab and ensure that the **Experience Service** URL is the same for your Vuforia Studio and ThingWorx instances. This is necessary for being able to connect the ThingWorx service into Studio.
2. In the **Home** tab, open the **Data** panel.



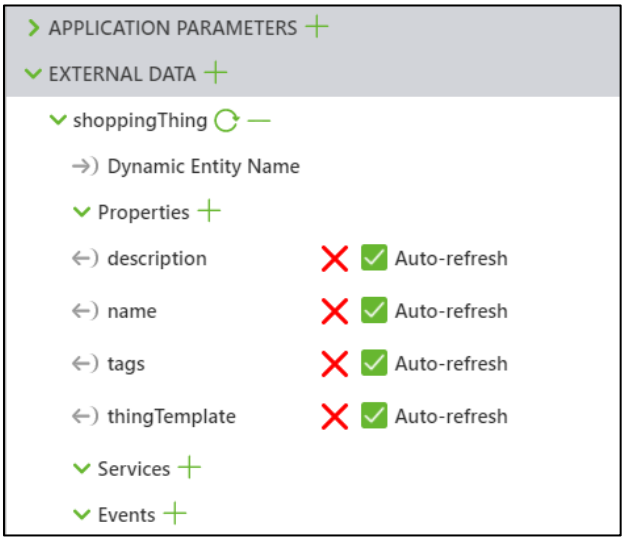
3. Click the green **+** next to **External Data**. This will open a window for an entity from ThingWorx to be added.



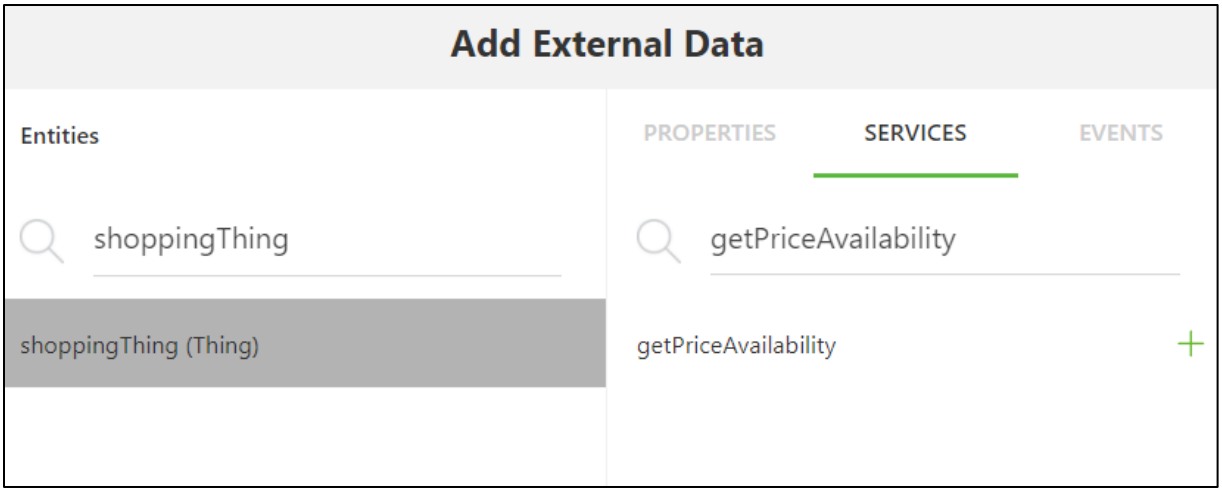
4. Type in *shoppingThing* into the **Entities** search bar. This will bring up a list of properties for the Thing.



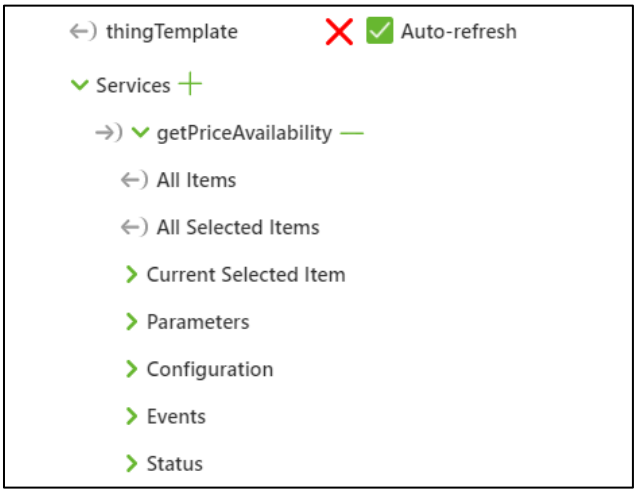
5. Click each of the **+** symbols next to the 4 options listed under **Properties** to add the properties to the Thing in Studio. Click **Close** after all properties have been added. Properties are general information about the Thing. **name** is the name of the Thing and **description** is a description 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.



6. After adding properties for the thing, a service needs to be added as well. The service is the code that will be executed to call the information for the model from ThingWorx. Click the **+** next to **Services**. In the window that appears, type *getPriceAvailability***.**



7. Click the green **+** next to **getPriceAvailability** to add the service to the Vuforia Studio experience and then click **Close**. The service is now available to be called upon by either bindings or JavaScript code in **Home.js**.



302.3 Call getPriceAvailability and Use the serviceInvokeComplete Event Listener

Once the **getPriceAvailability** service has been added to Studio, it needs to be called within the experience to make the data accessible for manipulation. In order to do this, you need to change the code from Metadata 301. Additionally, a new event listener will need to be added to the code to make the data from this service accessible by the experience.

1. Open **Home.js.**
2. In the `userpick` event listener, make the following changes:
    a. Delete the `priceString` variable, the data will be accessed from the **shoppingThing** in ThingWorx instead of the attributes for the model.

b. Create an object named `meta` and set it to the scope of the application so that it can be accessed by different functions in the script. Change the `partName`, `instructionName`, and `partNumber` variables to be properties of the `meta` object.

```
//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
```

```
10      var pathId = JSON.parse(eventData).occurrence
11      $scope.currentSelection = targetName + "-" + pathId
12
13      //create an object with the properties below that are based
14      $scope.meta = {
15        partName: metadata.get(pathId, 'Display Name'),
16        instructionName : metadata.get(pathId, 'illustration'),
17        partNumber: metadata.get(pathId, 'partNumber'),
18      } //scope.meta object
```

c. Delete the `price` variable and `priceInfo` application parameter since they were directly dependent on the `priceString` variable. Edit the definition of the `itemName` and `itemNumber` application parameters to stay consistent with the new syntax for calling `partName` and `partNumber` from the `meta` object. Create a new variable named `target`, scoped to the application, that will be set to the `targetName` variable from the `userpick` event.

```
// 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
```

```
18      } //$scope.meta object
19
20      // set itemName app parameter to be equal to the partName variable, same relationship wi
21      $scope.app.params.itemName = $scope.meta.partName;
22      $scope.app.params.itemNumber = $scope.meta.partNumber;
23      $scope.app.params.itemCount = 1;
24
25      $scope.target = targetName
```

d. Add the following code for triggering the `getPriceAvailability` service inside the `shoppingThing`. This code follows a format of twx.app.fn.triggerDataService('[TWX Entity]', '[TWX Service]', 'parameter'). The `triggerDataService` function works as a way of calling a service provided by ThingWorx into Vuforia Studio, where the ThingWorx Entity and Service are called, along with any input parameters that the service takes. In our case, the thing being called is the `shoppingThing` that holds all part information, the service being called is `getPriceAvailability`, and the parameter called is `pid`, which is a parameter for the part number of the selected part. The output of this service is called using an

asynchronous callback, which calls the `serviceInvokeComplete` event listener, which will be created later in this section

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

```
25      $scope.target = targetName
26
27      // call the getPriceAvailability ThingWorx service based on partNumber
28      twx.app.fn.triggerDataService('shoppingThing', 'getPriceAvailability', {pid: $scope.meta.partNumber})
```

e. The `if` statement for the popups below is going to be moved out of the `userpick` event listener and moved into the `serviceInvokeComplete` event listener. Copy and paste your code for the brackets to end the `PTC Metadata API` and `userpick` event listeners and catch and log error codes from after the `if` statement that is currently in the code for the popup and move it to below where the service trigger is. Make sure to delete the code that you just copied from below the `disassemble` function for the popup, or else you will have extra brackets.

```
    }) //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
```

```
28        twx.app.fn.triggerDataService('shoppingThing', 'getPriceAvailability', {pid: $scope
29
30        }) //end brackets for PTC API and .then
31      //
32      //catch statement if the promise of having a part with metadata is not met
33      .catch((err) => { console.log('metadata extraction failed with reason : ' + err) })
34
35  }) //end brackets for userpick function. Will continue to move throughout code
36
37  if (instructionName.length ==0) {
38        // adds an ionic popup when a part is clicked. Show the part number, name, price
```

3. With the `userpick` event listener updated, now it is time to create the `serviceInvokeComplete` event listener. This event listener will only be activated once the `getPriceAvailability` service has been completed.
    a. Add the following code for the event listener above the `if` statement for the popup. **Note**: there will be an X in a red circle next to this line because the end bracket for it has not been created yet. This is to be expected.

```
$scope.$on('getPriceAvailability.serviceInvokeComplete', function(evt) {
```

```
37  $scope.$on('getPriceAvailability.serviceInvokeComplete', function(evt) {
38    if (instructionName.length ==0) {
39      // adds an ionic popup when a part is clicked. Show the part number, n
```

    b. Create some space in between the `serviceInvokeComplete` event listener your `if` statement for the popup. The process for calling the popup is going to change now that data is being added dynamically from ThingWorx. Below the `serviceInvokeComplete` event listener, create a

variable named `rowData`, which will call upon the current row of the infotable that has been created with the `getPriceAvailability` service.

```
  //
  // variable holding all data for the current row in the infotable
  var rowData = twx.app.mdl['shoppingThing'].svc['getPriceAvailability'].data.current
```

```
$scope.$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

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

c. Next, add in scripting to determine what the price of the selected object is. The `price` variable created will be set to either a string with the price of the part in the selected row with a `$` in front of it or `'UNAVAILABLE'` if the part is unavailable. This has been created using a conditional operator which validates that the part is available before checking on the price of the part. This could also be done using an `if` statement with the same conditions. Similar logic is used to create the **priceInfo** application parameter, which is used for adding up prices in the cart.

```
  // 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
  var price = rowData.avail === true ? '$' + rowData.price : 'UNAVAILABLE';
  $scope.app.params.priceInfo = rowData.avail === true ? parseFloat(rowData.price) : undefined
```

```
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 pric
var price = rowData.avail === true ? '$' + rowData.price : 'UNAVAILABLE';
$scope.app.params.priceInfo = rowData.avail === true ? parseFloat(rowData.price) : undefined
```

d. Create a variable named `meta` which will be used to bring the `$scope.meta` object into the event listener as a local object to allow for easy access to its values.

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

```
  $scope.app.params.priceInfo = rowData.avail ===

  // create a variable to bring the $scope.meta ob
  let meta = $scope.meta
```

e. Since additional information is going to be added into the popup to account for determining whether a part is available or not based on information from **shoppingThing**, the template for the popup is now going to be set using a function that will be created later in the activity. Delete the conditions and endpoints for the if statement that you created earlier and outdent your code for calling the popup.

```
47    let meta = $scope.meta
48
49    // adds an ionic popup when a part is clicked. Show the part number, name, price, and quantity
50    $scope.popup = $ionicPopup.show({
51      //
52      //template for the popup with added buttons
53      template: '<div>' + $scope.app.params.itemCount + 'x  ' + partNumber + ' </br>' +
54      ' </div><div class="btnadd" ng-click="hiliteOff();popup.close();addToCart();">Add to Ca
55      //
56      // set the scope for the popup
57      scope: $scope
58
59    }); //end of ionic popup if there is no disassembly sequence
60
61    $scope.popup = $ionicPopup.show({
62      //
63      //template for the popup with added buttons
64      template: '<div>' + $scope.app.params.itemCount + 'x  ' + partNumber + ' </br>' +
65      ' </div><div class= "btnadd" ng-click="hiliteOff();popup.close();addToCart();">Add to C
66      //
67      // set the scope for the popup
68      scope: $scope
69
70    }); //end of ionic popup if there is a disassembly sequence
```

f. Delete the code for the popup for what would have previously been the `else` case for the `if` statement. Additionally, delete the value for the `template` property, this will be replaced in the next step. There will be error indicators next to these lines of code, but they will be resolved when the `template` value is populated.

```
47    let meta = $scope.meta
48
49    // adds an ionic popup when a part is clicked. Show the part number, name, price, and quantity of the selected
50    $scope.popup = $ionicPopup.show({
51      //
52      //template for the popup with added buttons
53      template:
54      //
55      // set the scope for the popup
56      scope: $scope
57
58    }); //end of ionic popup
```

g. The `template` property should have a value that is equal to the calling of the `setTemplate` function, which will be created later, with inputs of the `meta` object and the `price` variable for the selected part. The code for calling the popup should now match with the code provided below.

```
// adds an ionic popup when a part is clicked
$scope.popup = $ionicPopup.show({
  //
  //call the function for setting the template
  template: $scope.setTemplate(meta, price),
  //
  // set the scope for the popup
  scope: $scope

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

```
47    let meta = $scope.meta
48
49    // adds an ionic popup when a part is clicked
50    $scope.popup = $ionicPopup.show({
51      //
52      //call the function for setting the template
53      template: $scope.setTemplate(meta, price),
54      //
55      // set the scope for the popup
56      scope: $scope
57
58    }); //end of ionic popup
```

h. In the `hiliteOff` function and the calling of the `hilite` function inside the `serviceInvokeComplete` event listener, the input for the function should be changed to replace `targetName + "-" + pathId` with `$scope.currentSelection`.

```
  //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
```

```
58      }); //end of ionic popup
59
60      //highlight the chosen item and set the shader to true
61      $scope.hilite([$scope.currentSelection], true);
62
63      //function for removing the highlight
64      $scope.hiliteOff = function() {
65        $scope.hilite([$scope.currentSelection], false)
66      }; // end of hiliteOff function
```

i. For the `disassemble` function, the `model` and `instruction` properties in the `modelObject` for setting the animation sequence of the model need to be updated. This is where the `$scope.target` variable that was created earlier will be accessed, and `instructionName` will be updated once again.

```
  // 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
```

```
66      }; // end of hiliteOff function
67
68      // function to be bound to the Disassemble button in the popup
69      $scope.disassemble = function () {
70        //
71        // set an object that targets the model and its instruction property
72        var modelObject = { model: $scope.target, instruction: 'l-Creo 3D - ' + meta.instructionName + '.pvi' };
73        //
74        // set the sequence for the quadcopter to be the name of the associated instruction
75        $scope.view.wdg.quadcopter.sequence = modelObject.instruction
76      } //disassemble function end
```

4. The final step to updating your experience is to create the `setTemplate` function that was previously mentioned. This function will be used to set the `template` variable that will be used in the popover to designate the information that is available. This will include the logic for determining if there is a disassembly sequence associated with a part and if it is available in **shoppingThing**.
    a. Below the `orderCart` function, create a function named `setTemplate` with inputs of `meta` and `price`.

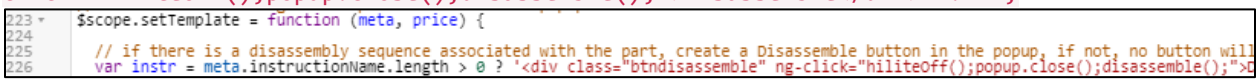
```
// function for setting the template for the Ionic popup
$scope.setTemplate = function (meta, price) {

}
```

    b. The first information added to the function will be logic for determining if there is a disassembly sequence associated with the selected part. The `instr` variable uses a conditional operator to see if the `instructionName` property of `meta` is populated or not. If there is an associated sequence, then there will be a button created that can be clicked to trigger the

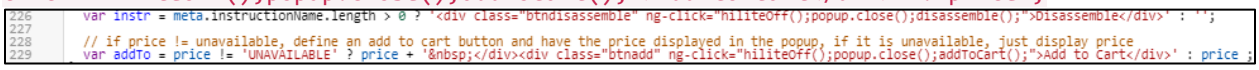
`disassemble` function, turn off the shader, and close the popup. If there is not an associated sequence, `instr` will become an empty string.

```
  // 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>' : '';
```



c. Additionally, logic needs to be created to determine if a part is available or not. The addTo variable is created to be the result of a conditional operator determining if a part has an associated price with it or not based on the information in shoppingThing. If it does, then the price is displayed in the popup along with a clickable Add to Cart button that triggers the addToCart function, if not, then only the price is added to the popup.

```
  // 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 ;
```

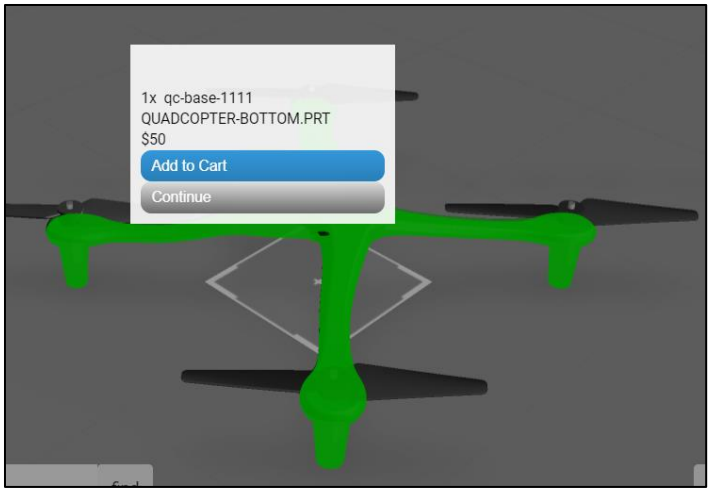


d. After the buttons available for the popup have been determined, the `template` variable will be created like the one that you have made for previous popups. The popup will display the quantity, part number, part name, and price of the selected part, along with the buttons that apply to the part. The Continue button will once again be added for closing the popup. A return statement will be added for the `template` variable, so it is output from the function when it is run. This will complete the `setTemplate` function.

```
  // 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
```

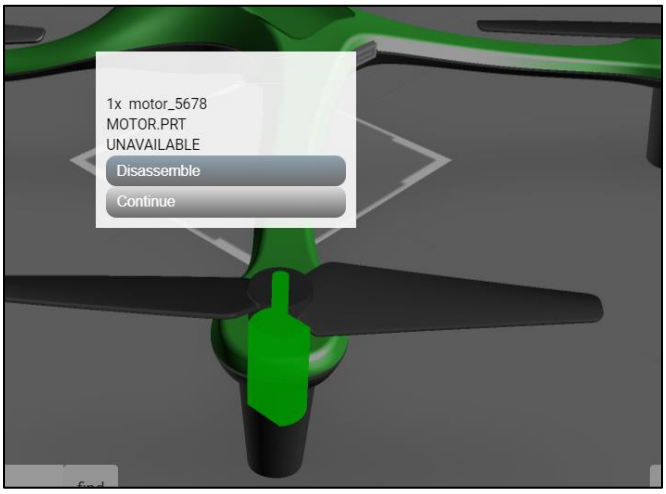
5. Click **Save** and open **Preview**
   a. Click the base of the quadcopter to view the popup option where a part is available but doesn't have a disassembly sequence associated with it. Click **Continue** to close the popup.
      i. **Note**: If the **Continue** button is not clicked before clicking on another part, the popups will stack on top of one another instead of the first one closing.



b. Select any of the rotors to see the popup when a part has a disassembly sequence associated with it and is available.

c. Select the top of the quadcopter to see the popup when a part doesn't have a disassembly sequence associated with it and is unavailable.

i. Select any of the rotors to see the popup when a part has a disassembly sequence associated with it but is unavailable.



6. This section is now complete. The code for everything else in this section is the same as it was in Metadata 301. You should be able to click on a part, have the popup appear and be given one of the four options above for a popup, and then if applicable, add the part to the cart. Part numbers can also still be looked up using the **userInput** widget. Check out Appendix 1 for the full code for this section.

Appendix 1: Section 302.3 Code

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

$scope.$on('userpick', function (event, targetName, targetType, eventData) {
  //
  //Look at model and see if it has metadata. If it does, then execute the below code and
create an object called metadata
  PTC.Metadata.fromId(targetName)
      .then((metadata) => {
    //
    // variable to pull the value for the occurrence property in the eventData JSON
object from the model. Create variable for the currently selected part
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId

    //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
  var price = rowData.avail === true ? '$' + rowData.price : 'UNAVAILABLE';
  $scope.app.params.priceInfo = rowData.avail === true ? parseFloat(rowData.price) :
undefined

  // 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, 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

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

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

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

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

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

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

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

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

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

```
    }); // end of the push method for cartContents
  }// for loop end

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

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

} //end of addToCart function

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

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

// 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
}
```