Tags
Find a Tag
analysis application availability_section best_practices cad change_management changes compare compare_revisions customizations customize database db2 exchange find_available_times gantt_chart gantt_scheduler group memory_footprint modifications modify object_label object_revision operating_system oracle performance platform problem_determination reports reserve reserve_performance revision revisioning single_sign-on snapshot space sql_server sso support system system_performance tags: track_customizations tririga troubleshoot tuning upgrade ux version versioning
Cloud | List

You are in: IBM TRIRIGA > UX Framework > UX App Building > Extending UX

Extending UX

Like | Updated October 10, 2018 by Jay.Manaloto | Tags: *None* Add tags

Edit Page Actions

UX Framework | UX App Building | UX in Classic Tools | UX App Designer Tools | UX Best Practices

See the UX Article 3 "Extending UX" PDF for previous versions of this content. What is UX? The standard definition of "UX" is user experience. But for simplicity, I'll refer to the TRIRIGA UX framework as "UX".

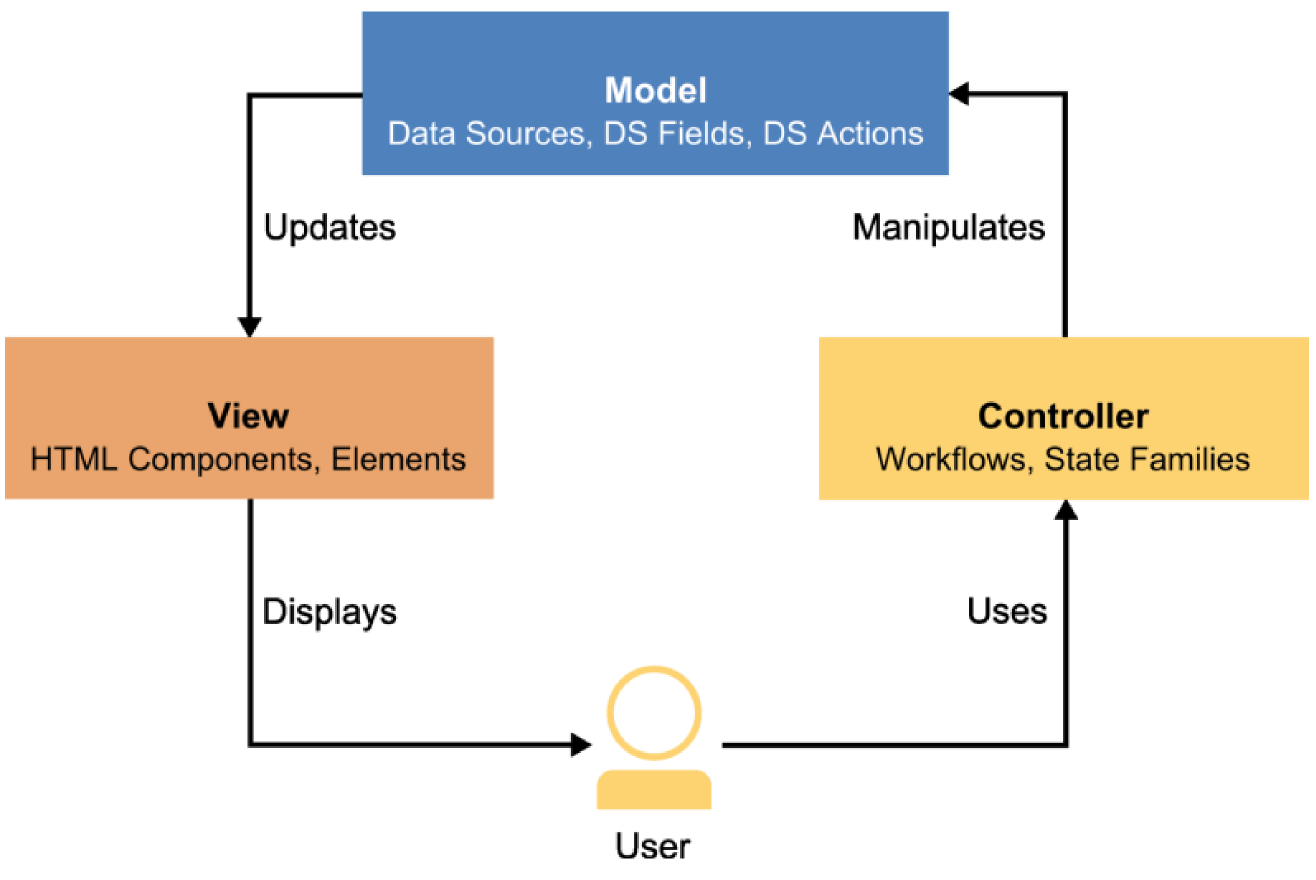
Extending UX: Adding more functionality to your IBM TRIRIGA UX application

STILL HUNGRY? If you're ready for a third serving, I admire your appetite! In my **first two articles**, we explored the concepts and built a simple UX application. This time, we'll extend our simple application with new fields, buttons, dialogs, toasts, and ways to manipulate records.

- What are the UX model and view components?
- What are the basic steps to build a UX application?
- Can we add other types of fields and buttons?
- Can we display dialogs and other messages?
- Can we display and modify existing records?
- Can we explore more advanced functions?
- Still want more?

What are the UX model and view components?

To refresh our memories, if we redraw the basic MVC diagram with our decoupled metadata approach, our UX framework might look like this.


Model
Data Sources, DS Fields, DS Actions
Updates
Manipulates
View
HTML Components, Elements
Controller
Workflows, State Families
Displays
Uses
User

Model components.

If you remember, this is where *you* can define your models in whatever way you see fit to fulfill your business needs. First, you must define your models before you can develop your views.

Each model can be made up of the following components:

- *Data Sources.*
  - *Child Data Sources.*
  - *Related Data Sources.*
- *Data Source Fields.*
- *Data Source Actions.*

View components.

After your models are in place, this is where *you* can design your views in whatever way you require to satisfy your business scenarios. Even better, you're free to design any number of views for each model.

Each view is made up of one or more HTML files. In turn, each HTML file can be made up of the following components:

- *TRIRIGA components.*
- *Custom components.*
- *Polymer elements.*
- *Traditional elements.*

What are the basic steps to build a UX application?

If you also remember, this is where *you* built a simple 3-field 3-button application by (1) defining a **model** with a single data source, (2) defining the view connections to a model-and-view and application, and (3) defining and designing a **view** with a single HTML file.

Here are the basic steps:

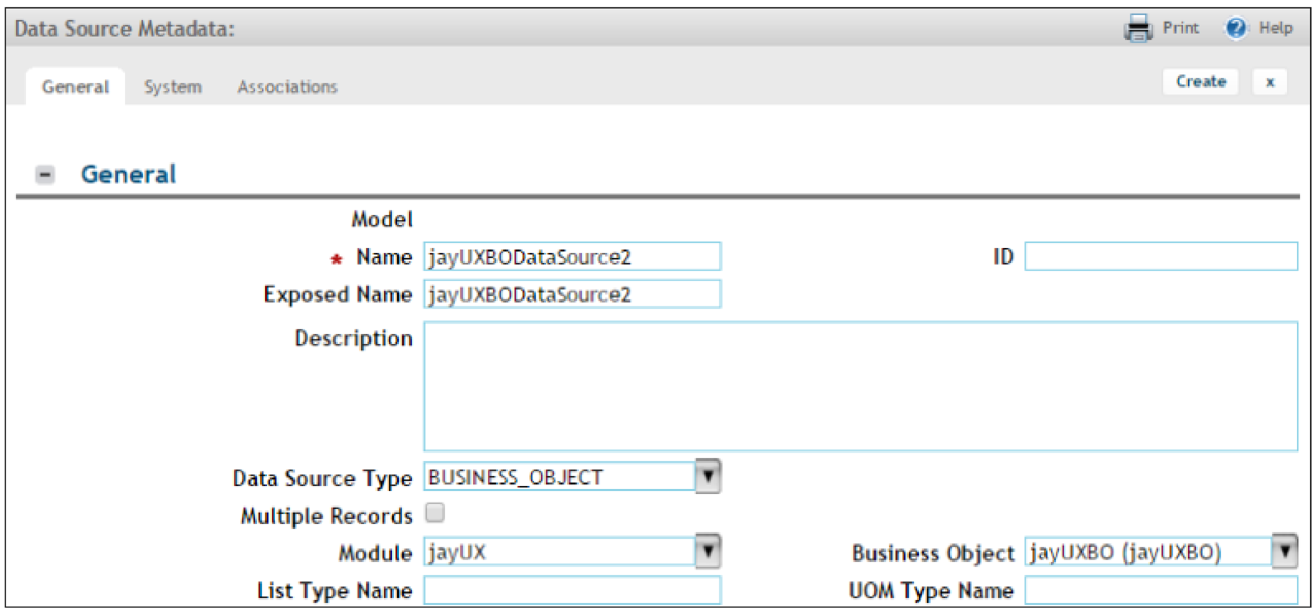
- Define your model.
  - Optional: Add the business object.
  - 1: Add the model.
  - 2: Add the data source.
  - 3: Add a few fields for your data source.
- Define your view connections.
  - 4: Add the view.
  - 5: Add the model-and-view.
  - 6: Add the application for your model-and-view.
- Define your view.
  - 7: Set up the view sync.
  - 8: Add the HTML file for your view.
  - 9: Access the application.
- Design your view.
  - 10: Start the view sync.
  - 11: Add a paragraph element to your HTML file.
  - 12: Add a few field elements to your HTML file.
  - 13: Add a few button elements to your HTML file.

Can we add other types of fields and buttons?

Sure! At this point, you should have a better idea of the application building process. For our exercise, prepare your model with a data source that contains the following field types: **Text** (like **triDescriptionTX**), **Number** (like **triAreaNU**), and **Boolean** (like **triReservableBL**).

In our example, we'll name the model **jayUXBOModel2** and name the data source **jayUXBODataSource2**.

Data Source Metadata.

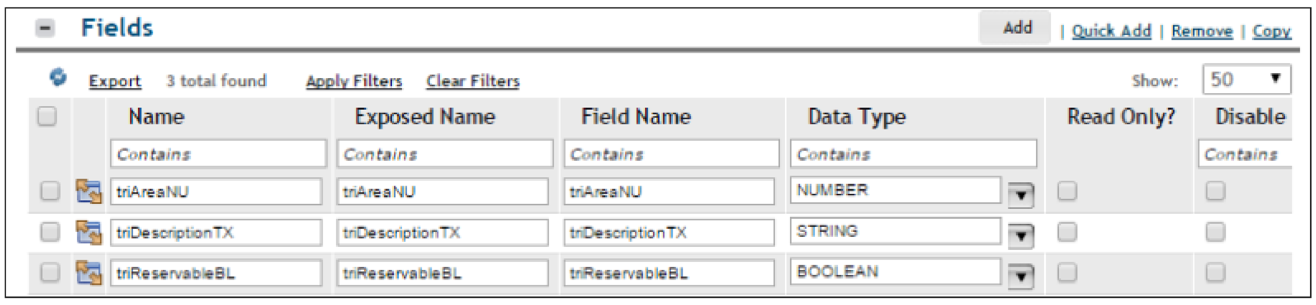


General

Model			
* Name	jayUXBODataSource2	ID	
Exposed Name	jayUXBODataSource2		
Description			
Data Source Type	BUSINESS_OBJECT		
Multiple Records	☐		
Module	jayUX	Business Object	jayUXBO (jayUXBO)
List Type Name		UOM Type Name	

We'll add the fields to the data source by using **Quick Add**.

Data Source Fields.



☐		Name	Exposed Name	Field Name	Data Type	Read Only?	Disable
		Contains	Contains	Contains	Contains		Contains
☐		triAreaNU	triAreaNU	triAreaNU	NUMBER	☐	☐
☐		triDescriptionTX	triDescriptionTX	triDescriptionTX	STRING	☐	☐
☐		triReservableBL	triReservableBL	triReservableBL	BOOLEAN	☐	☐

Similarly, we'll name the view **jayUXBOView2** (and **jay-uxbo-view2**), name the model-and-view **jayUXBOModelAndView2**, and name the application **jayUXBOApp2** with the label **Jay UX BO Application 2**.

Next, after you've prepared your model and view connections, open the command prompt in your selected folder, run the **addview** command if needed, and run the **sync** command to sync your changes. In our example, we'll run **addview** with **jay-uxbo-view2** to add a new HTML view file.

WebViewSync > Add View and Sync.

```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar addview -v jay-uxbo-view2 -s

...

C:\tririga_ux\ux_server>java -jar WebViewSync.jar sync -a
Waiting for changes to sync...
```

If needed, add the **<link>** tag at the top to import the TRIRIGA **triplat-ds** (data source) component and add the **<triplat-ds>** tag to declare it.

Add a text area field to your HTML file.

This time, we'll add the Polymer **<paper-textarea>** tag for a multi-line text field based on the material design language by Google. If you have any questions about Polymer, its concepts, or its elements, feel free to check out the Polymer website at www.polymer-project.org.

First, add the **<link>** tag at the top to import the Polymer element: **<link rel="import" href="../paper-input/paper-textarea.html">**

HTML File > Import paper-textarea.

```
<link rel="import" href="../triplat-view-behavior/triplat-view-behavior.html">
<link rel="import" href="../triplat-ds/triplat-ds.html">

<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-textarea.html">
```

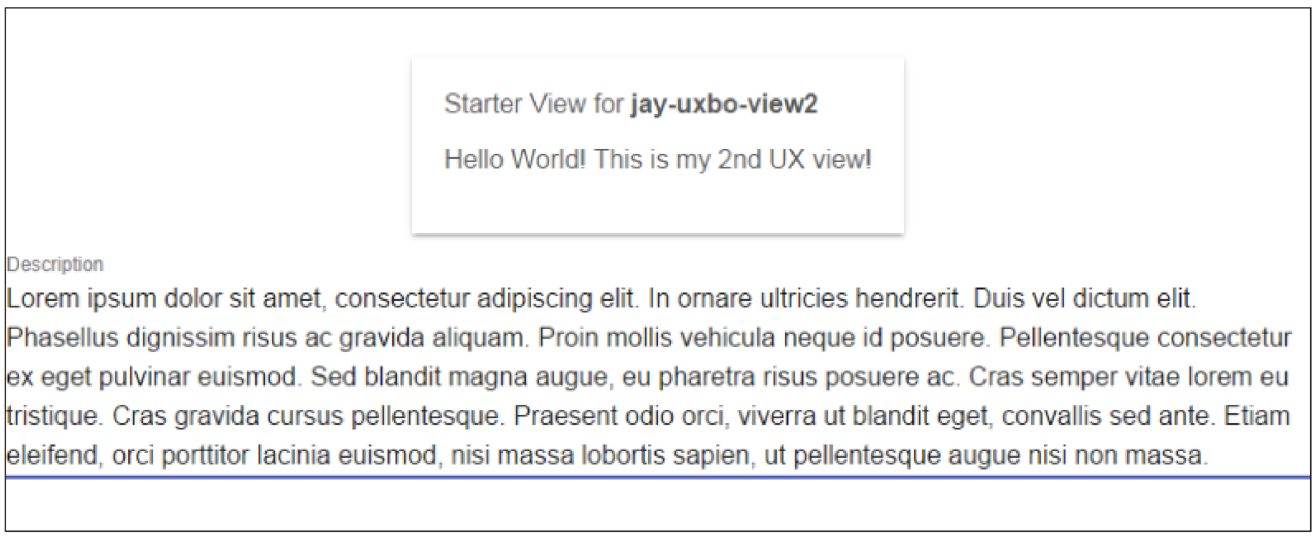
Next, add the **<paper-textarea>** tag: **<paper-textarea label="Description" floating-label value="{{data.triDescriptionTX}}"></paper-textarea>**

HTML File > Declare paper-textarea.

```
    <template>

        <triplat-ds id="model" name="jayUXBODataSource2" data="{{data}}"></triplat-ds>

        <div class="layout horizontal center-justified">
            <paper-material z="1">
                Starter View for <b>jay-uxbo-view2</b>

                <p>Hello World! This is my 2nd UX view!</p>

            </paper-material>
        </div>
        <div>
            <paper-textarea label="Description" floating-label value="{{data.triDescriptionTX}}"></paper-textarea>
        </div>
    </template>
```

Save the file and refresh the UX view. Do you see your field? If you do, why not type a few lines? Notice how the field expands automatically?

UX App > Starter View.


Starter View for **jay-uxbo-view2**

Hello World! This is my 2nd UX view!

Description

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In ornare ultricies hendrerit. Duis vel dictum elit. Phasellus dignissim risus ac gravida aliquam. Proin mollis vehicula neque id posuere. Pellentesque consectetur ex eget pulvinar euismod. Sed blandit magna augue, eu pharetra risus posuere ac. Cras semper vitae lorem eu tristique. Cras gravida cursus pellentesque. Praesent odio orci, viverra ut blandit eget, convallis sed ante. Etiam eleifend, orci porttitor lacinia euismod, nisi massa lobortis sapien, ut pellentesque augue nisi non massa.

Add a number field to your HTML file.

This time, we'll add the Polymer **<paper-input>** tag for a 5-decimal-place number field based on the material design language by Google. Like before, make sure the **sync** command is running in the command prompt.

First, add the **<link>** tag at the top to import the Polymer element: **<link rel="import" href="../paper-input/paper-input.html">**

HTML File > Import paper-input.

```
<link rel="import" href="../triplat-view-behavior/triplat-view-behavior.html">
<link rel="import" href="../triplat-ds/triplat-ds.html">

<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-textarea.html">
<link rel="import" href="../paper-input/paper-input.html">
```

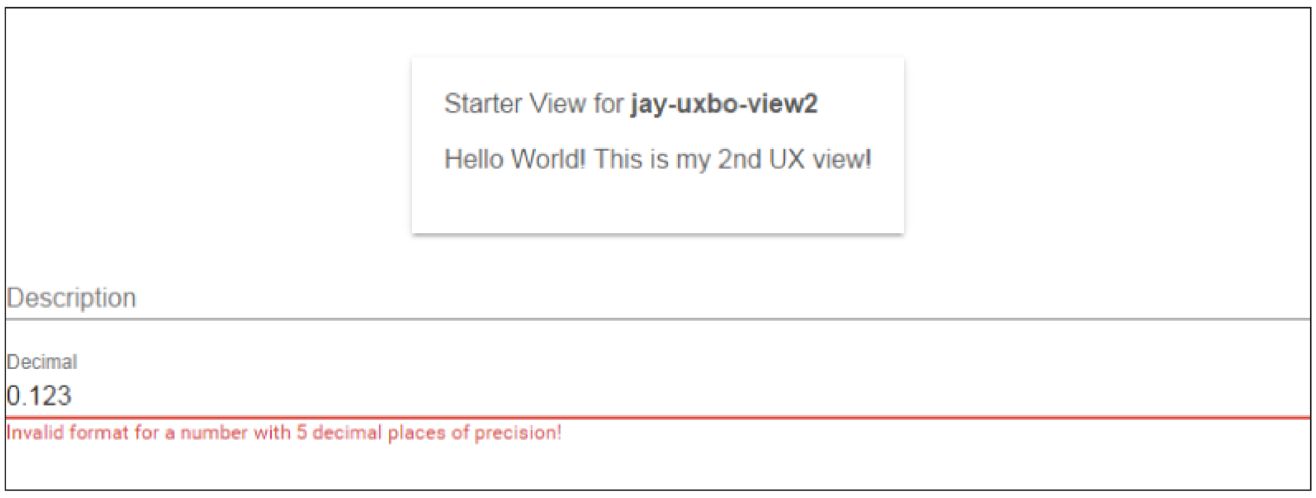
Next, add the **<paper-input>** tag to declare the Polymer element: **<paper-input label="Decimal" floating-label auto-validate pattern="[0-9]*\.[0-9][0-9][0-9][0-9][0-9]" error-message="Invalid format for a number with 5 decimal places of precision!" value="{{data.triAreaNU}}"></paper-input>**

HTML File > Declare paper-input.

```
        <div>
            <paper-textarea label="Description" floating-label value="{{data.triDescriptionTX}}"></paper-textarea>

            <paper-input label="Decimal" floating-label auto-validate pattern="[0-9]*\.[0-9][0-9][0-9][0-9][0-9]" error-message="Invalid format for a number with 5 decimal places of precision!" value="{{data.triAreaNU}}"></paper-input>
        </div>
    </template>
```

Save the file and refresh the UX view. Do you see your field? Why not type a few numbers? Notice how the error message appears when needed?

UX App > Input with Error Message.



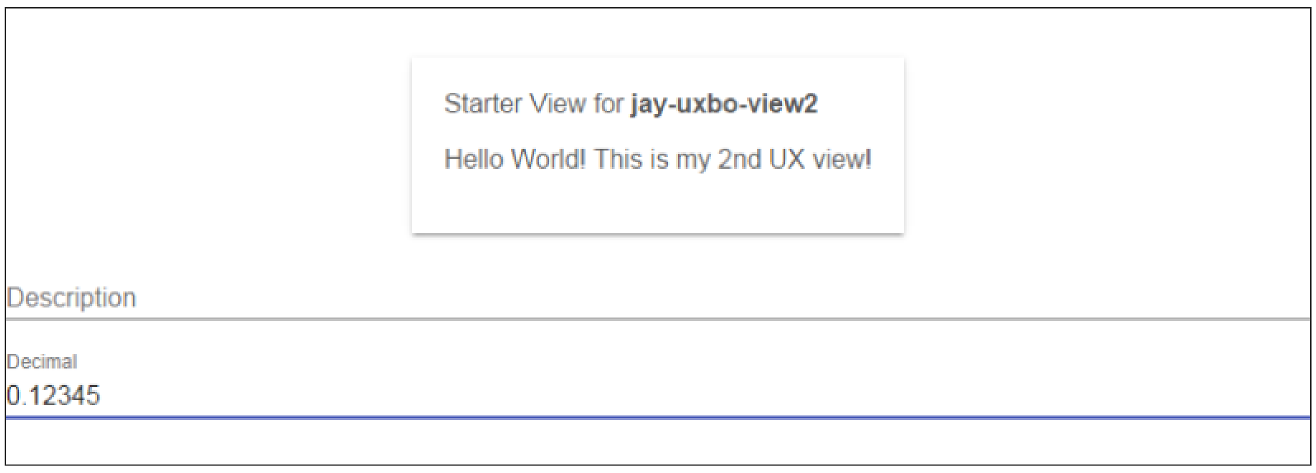
Starter View for **jay-uxbo-view2**

Hello World! This is my 2nd UX view!

Description

Decimal
0.123
Invalid format for a number with 5 decimal places of precision!

UX App > Input without Error Message.



Starter View for **jay-uxbo-view2**

Hello World! This is my 2nd UX view!

Description

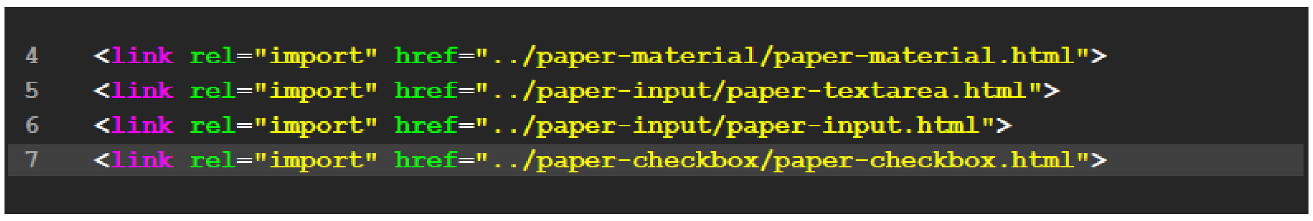
Decimal
0.12345

Add a Boolean button to your HTML file.

This time, we'll add the Polymer **<paper-checkbox>** tag for a button that can be either checked or unchecked, based on the material design language by Google. Like before, make sure the **sync** command is running.

First, add the **<link>** tag at the top to import the Polymer element: **<link rel="import" href="../paper-checkbox/paper-checkbox.html">**

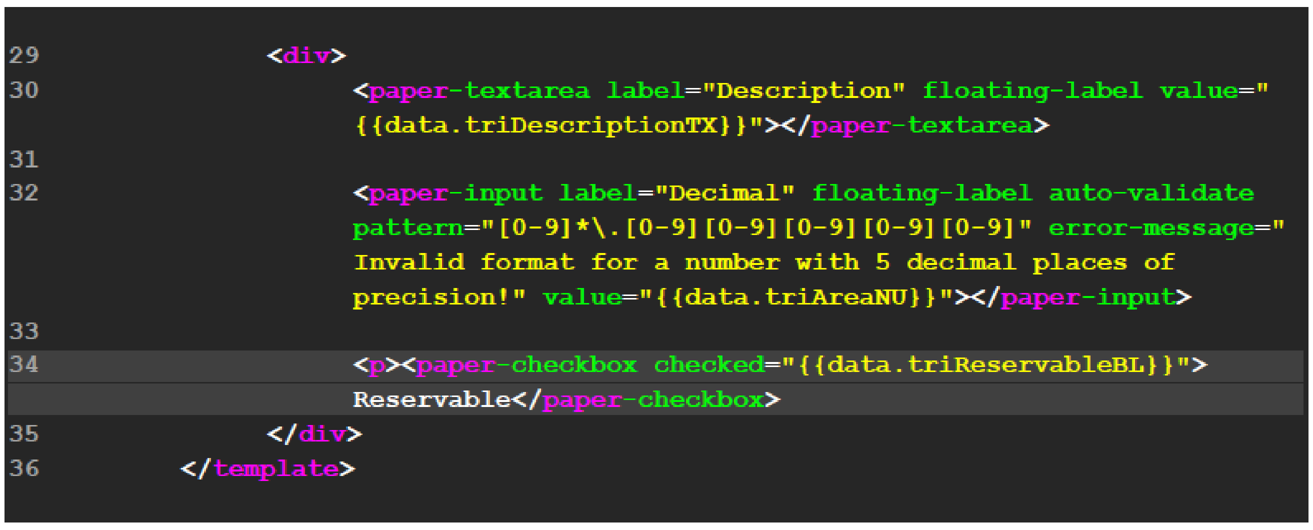
HTML File > Import paper-checkbox.



```
<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-textarea.html">
<link rel="import" href="../paper-input/paper-input.html">
<link rel="import" href="../paper-checkbox/paper-checkbox.html">
```

Next, add the **<paper-checkbox>** tag to declare it: **<paper-checkbox checked="{{data.triReservableBL}}">Reservable</paper-checkbox>**

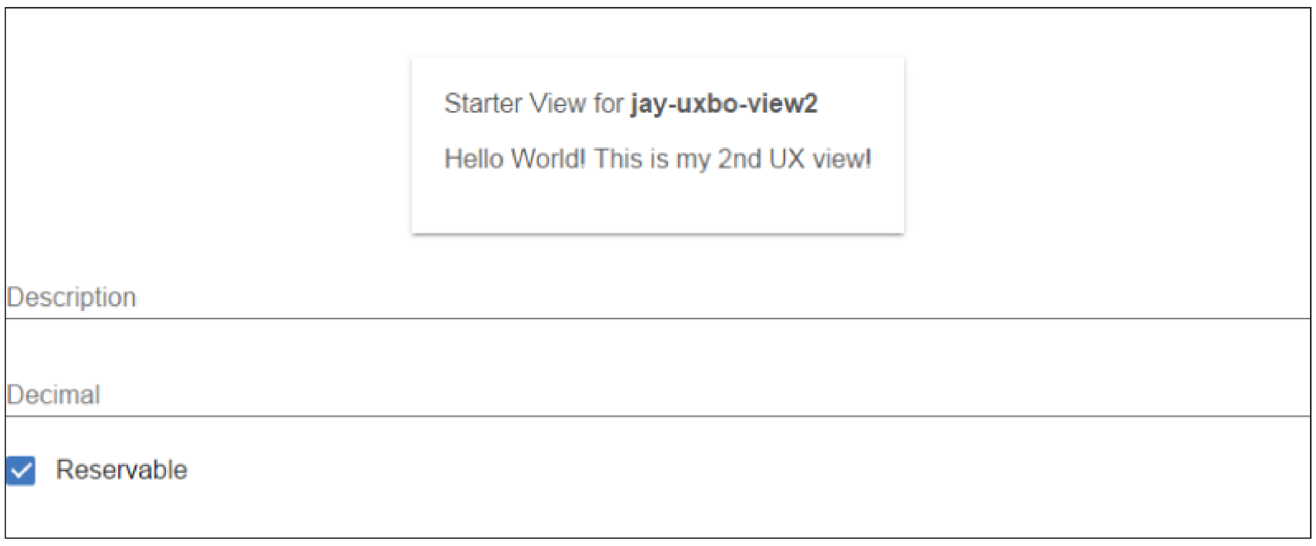
HTML File > Declare paper-checkbox.



```
                <div>
                    <paper-textarea label="Description" floating-label value="
                    {{data.triDescriptionTX}}"></paper-textarea>

                    <paper-input label="Decimal" floating-label auto-validate
                    pattern="[0-9]*\.[0-9][0-9][0-9][0-9][0-9]" error-message="
                    Invalid format for a number with 5 decimal places of
                    precision!" value="{{data.triAreaNU}}"></paper-input>

                    <p><paper-checkbox checked="{{data.triReservableBL}}">
                    Reservable</paper-checkbox>
                </div>
        </template>
```

Save the file and refresh the UX view. Do you see your button? If you do, why not add a couple more **<paper-checkbox>** tags on your own?

UX App > Check Box.



Starter View for **jay-uxbo-view2**

Hello World! This is my 2nd UX view!

Description

Decimal

☑ Reservable

Can we display dialogs and other messages?

Why not? Like before, open the command prompt in your selected folder, run the **addview** command if needed, and run the **sync** command to sync your changes. In our example, we'll keep going with **jay-uxbo-view2**.

Add an action dialog to your HTML file.

This time, we'll add the Polymer **<paper-button>** tag for a button with a ripple effect and Polymer **<paper-dialog>** tag for a popup dialog box, both based on the material design language by Google. For our exercise, this dialog will be triggered from a button, and will offer two button actions.

First, add the **<link>** tags at the top to import both Polymer elements.

HTML File > Import paper-button and paper-dialog.

```
<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-textarea.html">
<link rel="import" href="../paper-input/paper-input.html">
<link rel="import" href="../paper-checkbox/paper-checkbox.html">
<link rel="import" href="../paper-button/paper-button.html">
<link rel="import" href="../paper-dialog/paper-dialog.html">
```

Next, add the **<paper-dialog>** tag beneath the selected **<paper-button>** tag. In this case, the **UX rocks!** button. Then, to call a JavaScript method **ontapHandler** when the button event is caught, wrap the **<section on-tap="ontapHandler">** tag around the **<paper-button>** tag.

HTML File > Declare paper-button and paper-dialog.

```
<template>

    <triplat-ds id="model" name="jayUXBODataSource2" data="{{data}}"></triplat-ds>

    <div class="layout horizontal center-justified">
        <paper-material z="1">
            Starter View for <b>jay-uxbo-view2</b>

            <p>Hello World! This is my 2nd UX view!</p>

            <section on-tap="ontapHandler">
                <paper-button disabled>Disabled</paper-button>
                <br><paper-button raised data-dialog="actions">UX rocks!</paper-button>
                    <paper-dialog id="actions">
                        <h2>Confirm</h2>
                        <p>Are you sure that UX rocks?</p>
                        <div class="buttons">
                        <paper-button raised dialog-dismiss>No</paper-button>
                        <paper-button raised dialog-confirm autofocus>Yes</paper-button>
                        </div>
                    </paper-dialog>
            </section>
        </paper-material>
    </div>
```

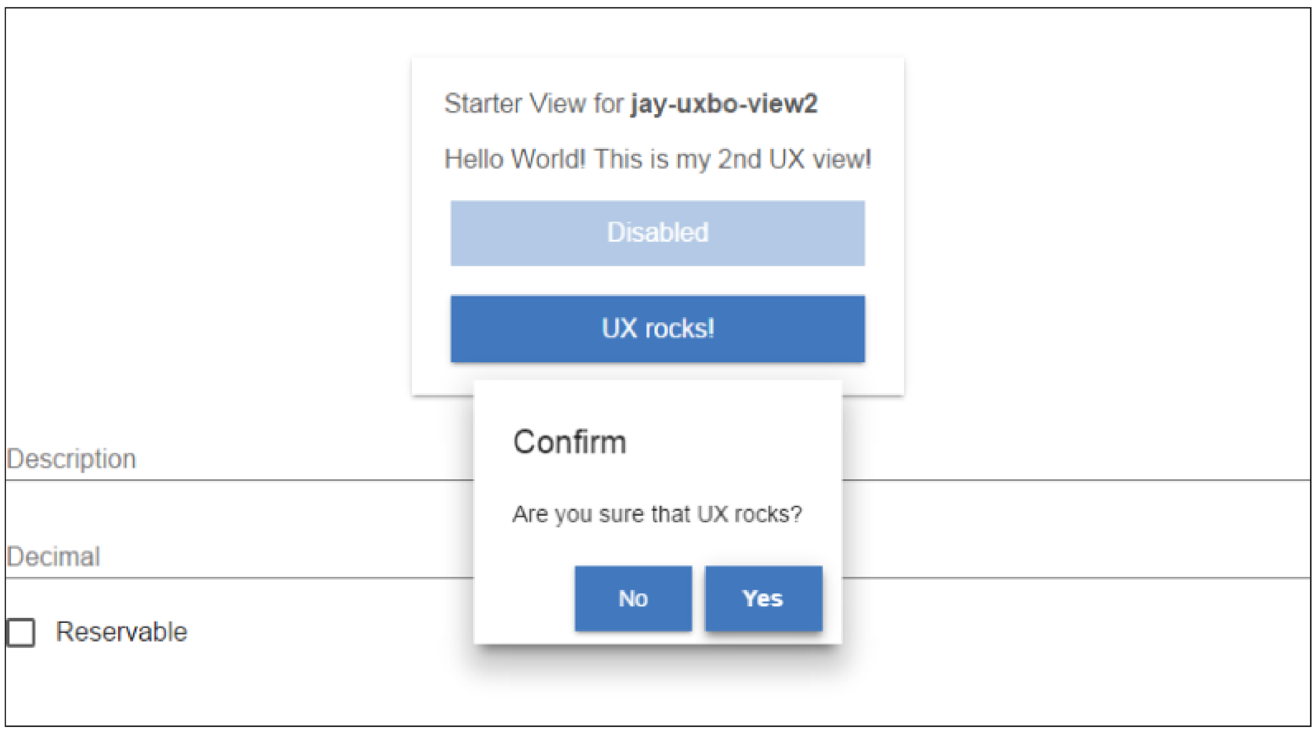
Then, insert the JavaScript method **ontapHandler** within the **<script>** tag.

HTML File > Insert ontapHandler.

```
    <script>
        Polymer({

            is: "jay-uxbo-view2",

            behaviors: [TriPlatViewBehavior],

            ontapHandler: function(e) {
                var button = e.target;
                if (button.innerText=="NO") {
                    console.log("User tapped NO");
                    // Add business logic when user taps NO
                }
                else if (button.innerText=="YES") {
                    console.log("User tapped YES");
                    // Add business logic when user taps YES
                }
                while (!button.hasAttribute('data-dialog') && button !==
                    document.body) {
                    button = button.parentElement;
                }
                if (!button.hasAttribute('data-dialog')) {
                    return;
                }
                var id = button.getAttribute('data-dialog');
                var dialog = document.getElementById(id);
                if (dialog) {
                    dialog.open();
                }
            },

        });
    </script>
```

Save the file and refresh the UX view. Do you see your button? If you do, feel free to click it. Do you see your action dialog? Pretty cool, huh?

UX App > Button and Action Dialog.


Starter View for **jay-uxbo-view2**

Hello World! This is my 2nd UX view!

Are you sure that UX rocks?

Add a toast popup message to your HTML file.

This time, we'll add the Polymer **<paper-toast>** tag for a subtle notification that pops up like toast, based on the material design language by Google. For our exercise, this toast will be triggered from a button, and will show for 7 seconds. Like before, make sure the **sync** command is running.

First, add the **<link>** tag at the top to import the Polymer element: **<link rel="import" href="../paper-toast/paper-toast.html">**

HTML File > Import paper-toast.

```
    <link rel="import" href="../paper-button/paper-button.html">
    <link rel="import" href="../paper-dialog/paper-dialog.html">
    <link rel="import" href="../paper-toast/paper-toast.html">
```

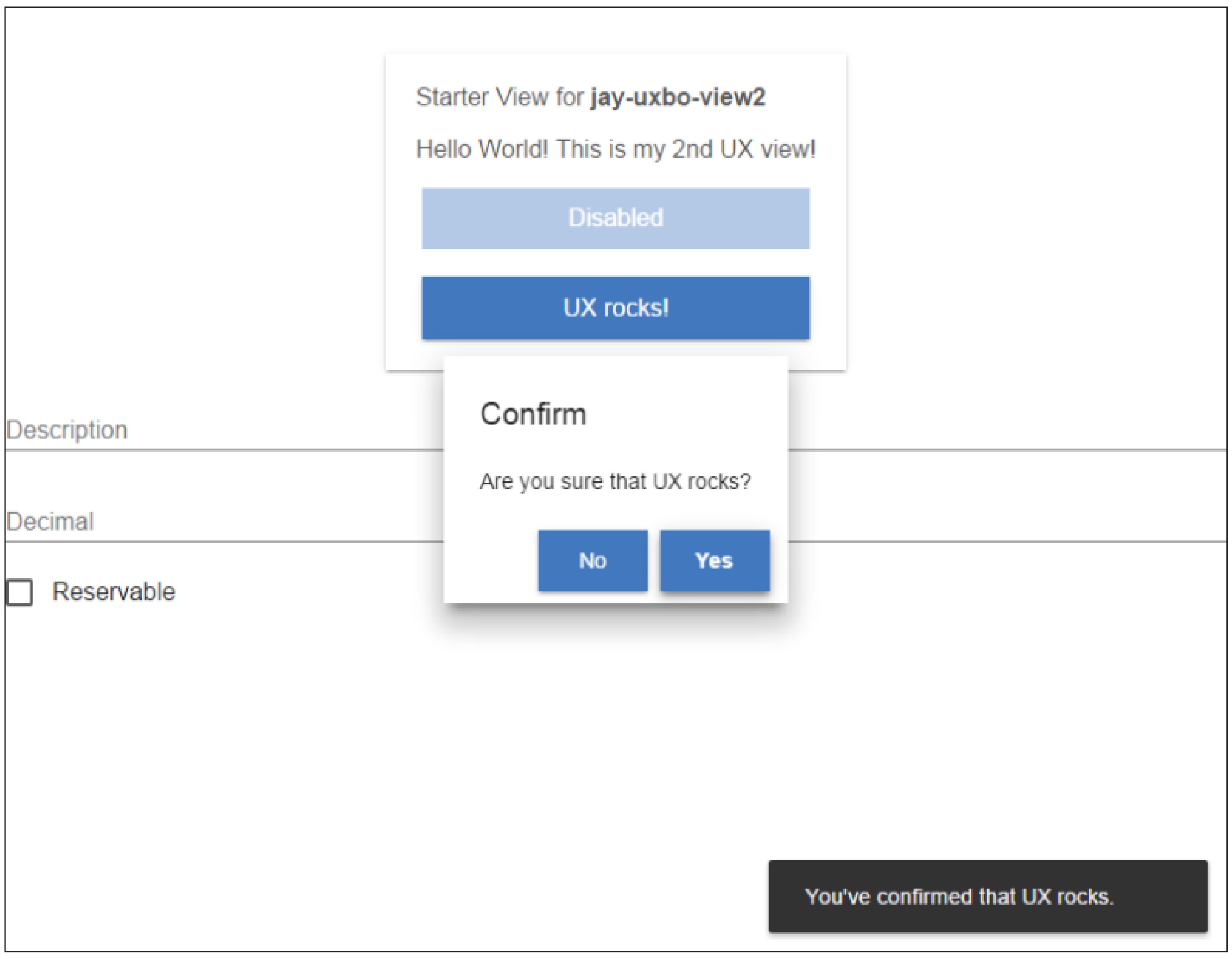
Next, insert **onclick="document.querySelector('#toast1').show()"** within the selected **<paper-button>** tag. In this case, the **Yes** button. Then, add the **<paper-toast>** tag: **<paper-toast id="toast1" text="You've confirmed that UX rocks." duration="7000" style="right:12px; left:initial;"></paper-toast>**

HTML File > Declare paper-toast.

```
        <section on-tap="ontapHandler">
            <paper-button disabled>Disabled</paper-button>
            <br><paper-button raised data-dialog="actions">UX rocks!</paper-button>
                <paper-dialog id="actions">
                    <h2>Confirm</h2>
                    <p>Are you sure that UX rocks?</p>
                    <div class="buttons">
                    <paper-button raised dialog-dismiss>No</paper-button>
                    <paper-button raised dialog-confirm autofocus onclick="document.querySelector('#toast1').show()">Yes</paper-button>
                    </div>
                </paper-dialog>
                <paper-toast id="toast1" text="You've confirmed that UX rocks." duration="7000" style="right:12px; left:initial;"></paper-toast>
        </section>
    </paper-material>
</div>
```

Save the file and refresh the UX view. Feel free to click the **UX rocks!** button, and then the **Yes** button. Do you see your toast? Pretty sweet!

UX App > Toast Popup.

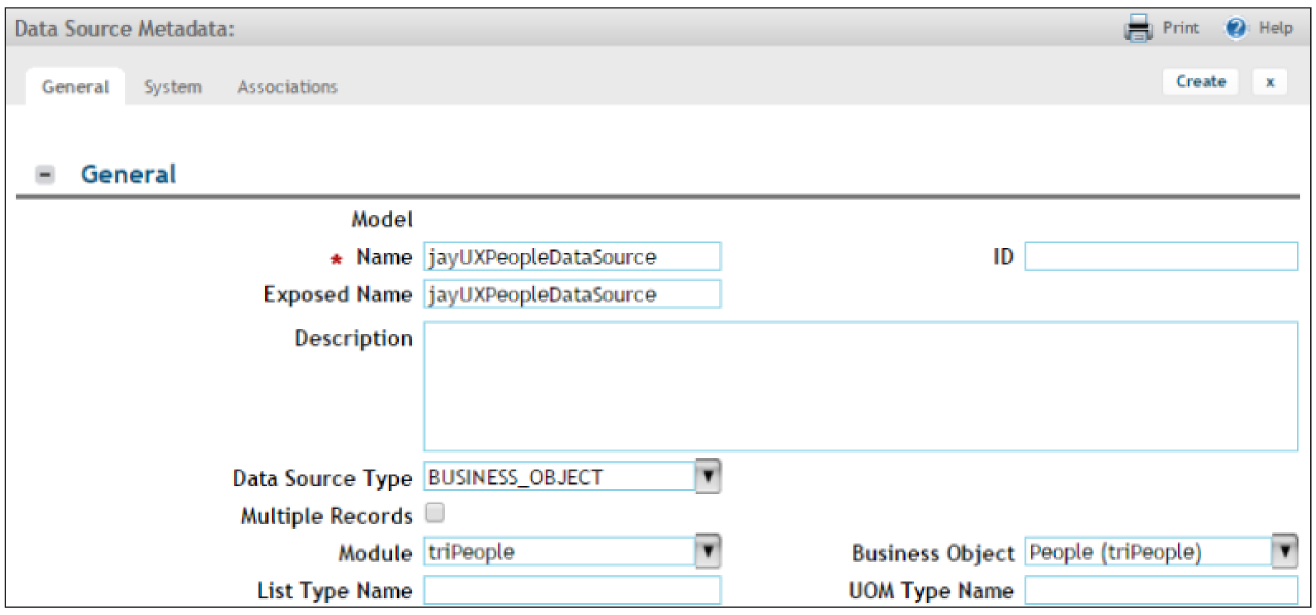

Starter View for **jay-uxbo-view2**
Hello World! This is my 2nd UX view!
Disabled
UX rocks!
Description
Decimal
Reservable
Confirm
Are you sure that UX rocks?
No
Yes
You've confirmed that UX rocks.

Can we display and modify existing records?

Yes, we can! For our next exercise, prepare your model with a data source that (1) has the **BUSINESS_OBJECT** data source type, **triPeople** module, and **triPeople** business object, and that (2) contains the following fields: **triFirstNameTX** (First Name) and **triLastNameTX** (Last Name).

In our example, we'll name the model **jayUXPeopleModel** and name the data source **jayUXPeopleDataSource**.

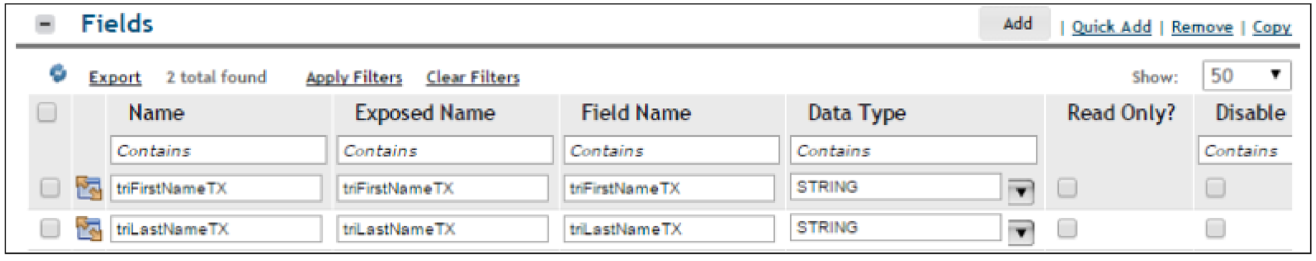
Data Source Metadata.



Model			
* Name	jayUXPeopleDataSource	ID	
Exposed Name	jayUXPeopleDataSource		
Description			
Data Source Type	BUSINESS_OBJECT		
Multiple Records	☐		
Module	triPeople	Business Object	People (triPeople)
List Type Name		UOM Type Name	

We'll add the fields to the data source by using **Quick Add**.

Data Source Fields.


Fields

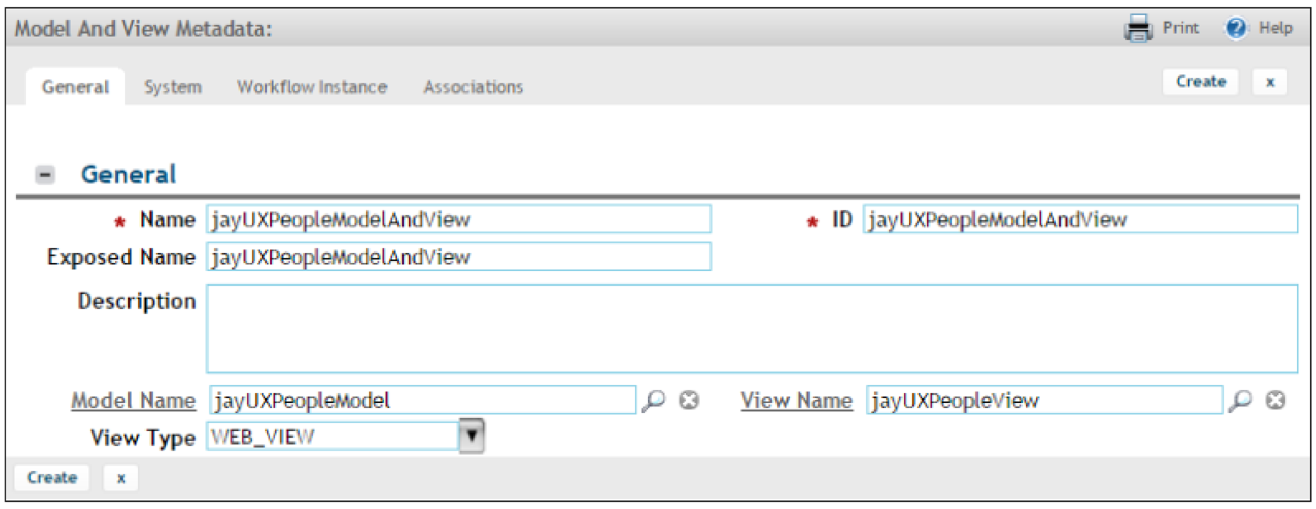
Add | Quick Add | Remove | Copy

Export 2 total found Apply Filters Clear Filters Show: 50

		Name	Exposed Name	Field Name	Data Type	Read Only?	Disable
		Contains	Contains	Contains	Contains		Contains
		triFirstNameTX	triFirstNameTX	triFirstNameTX	STRING		
		triLastNameTX	triLastNameTX	triLastNameTX	STRING		

Similarly, we'll name the view **jayUXPeopleView** (and **jay-ux-people-view**), name the model-and-view **jayUXPeopleModelAndView**, and name the application **jayUXPeopleApp** with the label **Jay UX People Application**.

Model and View Metadata.

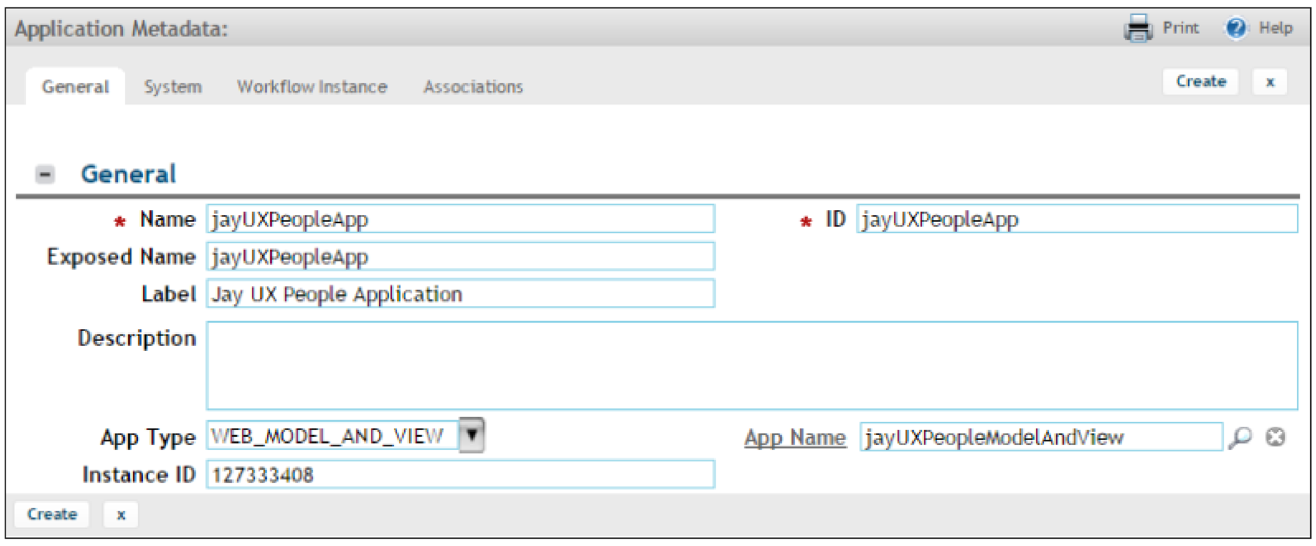

Model And View Metadata:

General

* Name	jayUXPeopleModelAndView	* ID	jayUXPeopleModelAndView
Exposed Name	jayUXPeopleModelAndView		
Description			
Model Name	jayUXPeopleModel	View Name	jayUXPeopleView
View Type	WEB_VIEW		

Since we want to pull data from an existing people record, prepare your application with an **Instance ID** that represents the **specId** of that record. To be clear, this ID isn't the ID that appears in the people form, it's the **specId** that appears in the URL of the people record. For example, a people record with an **ID** of **1000001** might have a **specId=127333408** in its related URL. Enter this **specId** as the **Instance ID** for your application.

Application Metadata.


Application Metadata:

General

* Name	jayUXPeopleApp	* ID	jayUXPeopleApp
Exposed Name	jayUXPeopleApp		
Label	Jay UX People Application		
Description			
App Type	WEB_MODEL_AND_VIEW	App Name	jayUXPeopleModelAndView
Instance ID	127333408		

Next, after you've prepared your model and view connections, open the command prompt in your selected folder, run the **addview** command if needed, and run the **sync** command to sync your changes. In our example, we'll run **addview** with **jay-ux-people-view** to add a new HTML view file.

WebViewSync > Add View and Sync.

```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar addview -v jay-ux-people-view -
s

...

C:\tririga_ux\ux_server>java -jar WebViewSync.jar sync -a
Waiting for changes to sync...
```

If needed, add the **<link>** tag at the top to import the TRIRIGA **triplat-ds** (data source) component and add the **<triplat-ds>** tag to declare it.

Display data from an existing record.

This time, we'll add more Polymer **<paper-input>** tags to grab the instance data **triFirstNameTX** and **triLastNameTX** from the existing people record.

First, add the **<link>** tag at the top to import the Polymer element: **<link rel="import" href="../paper-input/paper-input.html">**

HTML File > Import paper-input.

```
<link rel="import" href="../triplat-view-behavior/triplat-view-
behavior.html">
<link rel="import" href="../triplat-ds/triplat-ds.html">

<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-input.html">
```

Next, add the **<paper-input>** tags to declare the Polymer elements:

<paper-input ... value="{{data.triFirstNameTX}}"></paper-input>

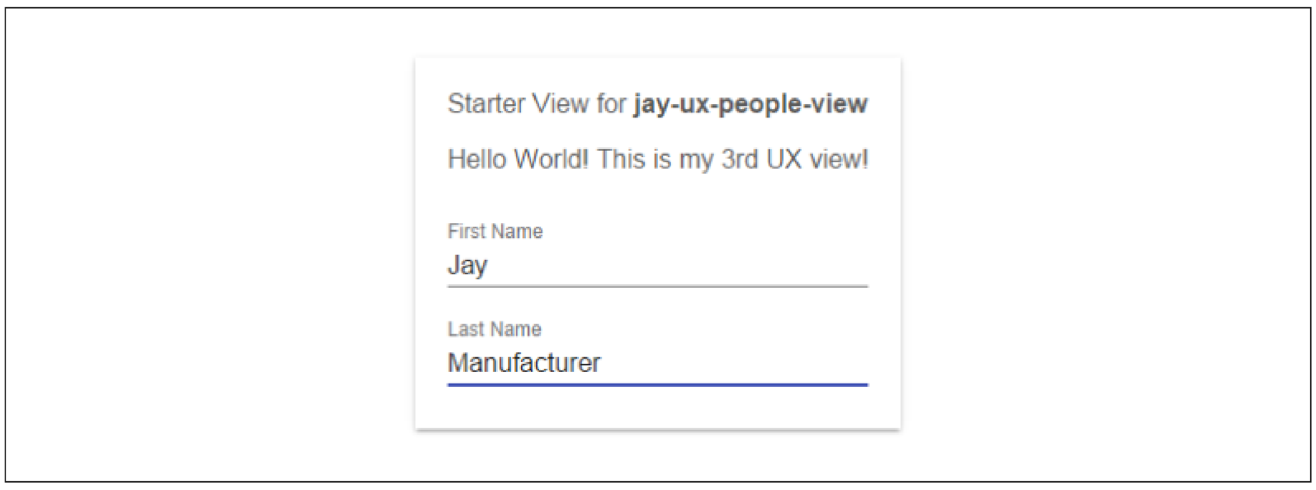
<paper-input ... value="{{data.triLastNameTX}}"></paper-input>

HTML File > Declare paper-input.

```
        <template>

            <triplat-ds id="model" name="jayUXPeopleDataSource" data="
            {{data}}"></triplat-ds>

            <div class="layout horizontal center-justified">
                <paper-material z="1">
                    Starter View for <b>jay-ux-people-view</b>

                    <p>Hello World! This is my 3rd UX view!</p>

                    <paper-input label="First Name" floating-label value="
                    {{data.triFirstNameTX}}"></paper-input>
                    <paper-input label="Last Name" floating-label value="
                    {{data.triLastNameTX}}"></paper-input>

                </paper-material>
            </div>
        </template>
```

Save the file and refresh the UX view. Do you see your fields? Do you see the corresponding instance data from your people record? Nice, huh?

UX App > Input with Instance Data.



Starter View for **jay-ux-people-view**

Hello World! This is my 3rd UX view!

First Name
Jay

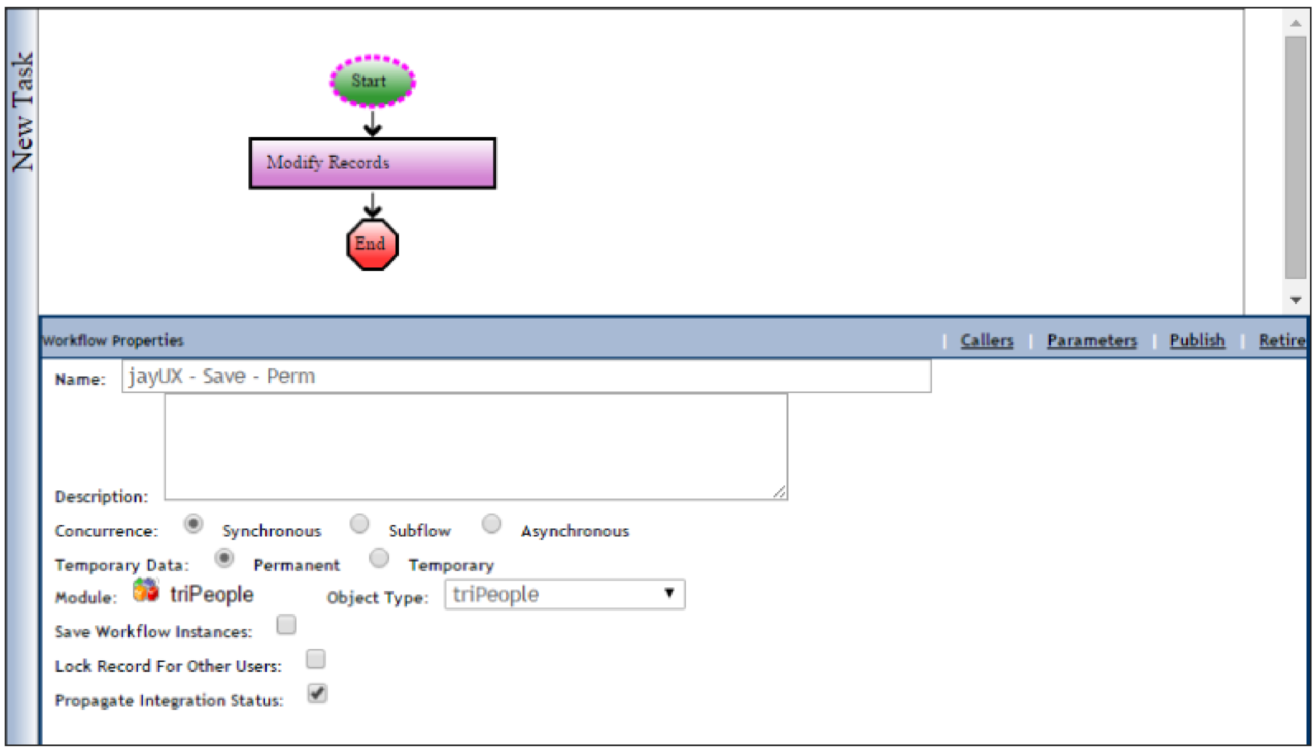
Last Name
Manufacturer

Modify data by triggering a workflow.

For our next exercise, prepare a workflow with **Synchronous** concurrence, **Permanent** data, the **triPeople** module, and the **triPeople** business object. Add a **Modify Records** task that will map to and from the **triPeople** business object. Edit the map to modify the **triNickNameTX** (Nick Name).

In our example, we'll name the workflow **jayUX - Save - Perm** and modify the **triNickNameTX** to **Jayman**. Then **Publish** the workflow.

Workflow > Modify Records Task.



Name: jayUX - Save - Perm

Description:

Concurrence: [✓ Synchronous] Subflow Asynchronous

Temporary Data: [✓ Permanent] Temporary

Module: triPeople Object Type: triPeople

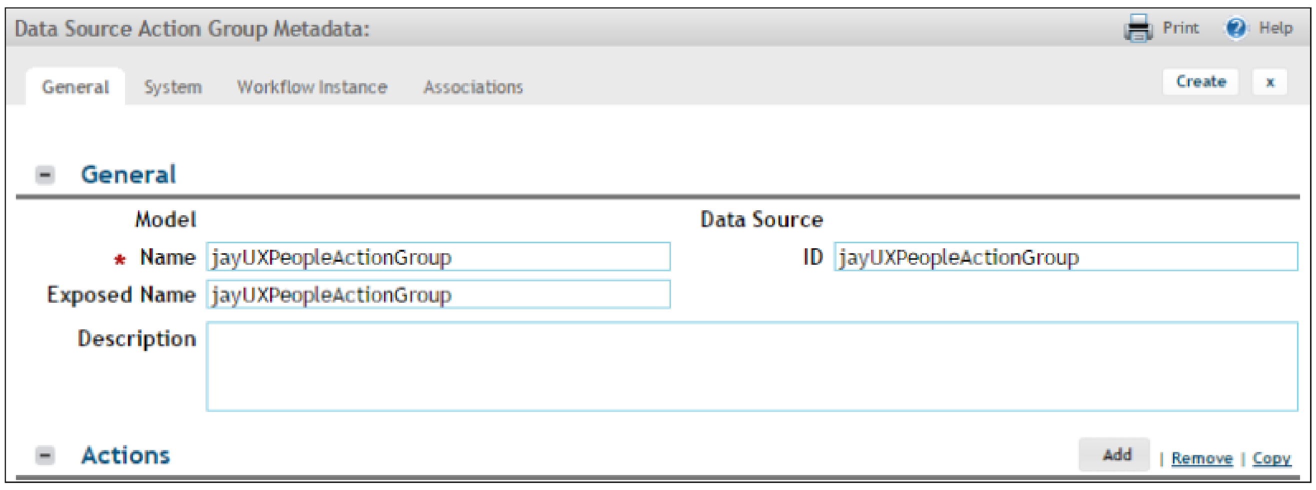
Save Workflow Instances: ☐

Lock Record For Other Users: ☐

Propagate Integration Status: ✓

Next, return to your data source from the last exercise, and (1) add the field **triNickNameTX**. Then, (2) add an action group, and (3) add an action that has the **WORKFLOW** action type, **triPeople** module, **triPeople** business object, and workflow name that you defined earlier.

Data Source Action Group Metadata.



Model

* Name: jayUXPeopleActionGroup

Exposed Name: jayUXPeopleActionGroup

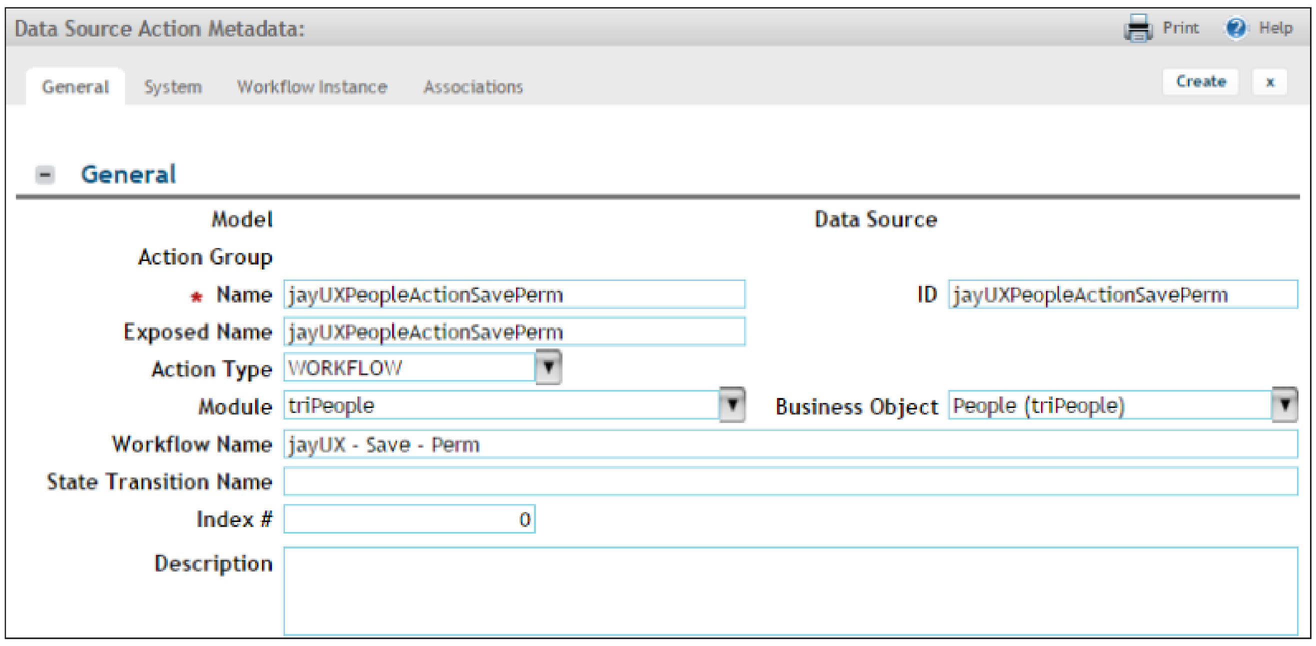
Description:

Data Source

ID: jayUXPeopleActionGroup

In our example, we'll name the action group **jayUXPeopleActionGroup**, name the action **jayUXPeopleActionSavePerm**, and name the workflow **jayUX - Save - Perm**. This hooks up your workflow into your data source. Finally, **Save & Close** your action, action group, and data source.

Data Source Action Metadata.



General

Model		Data Source	
Action Group			
* Name	jayUXPeopleActionSavePerm	ID	jayUXPeopleActionSavePerm
Exposed Name	jayUXPeopleActionSavePerm		
Action Type	WORKFLOW		
Module	triPeople	Business Object	People (triPeople)
Workflow Name	jayUX - Save - Perm		
State Transition Name			
Index #	0		
Description			

This time, we'll add one more Polymer **<paper-input>** tag to hold the instance data **triNickNameTX** and Polymer **<paper-button>** tag to trigger a workflow that modifies the **triNickNameTX** from null to **Jayman**. Like before, make sure the **sync** command is running in the command prompt.

First, add the **<link>** tag at the top to import the Polymer element: **<link rel="import" href="../paper-button/paper-button.html">**

HTML File > Import paper-button.

```
<link rel="import" href="../triplat-view-behavior/triplat-view-behavior.html">
<link rel="import" href="../triplat-ds/triplat-ds.html">

<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-input.html">
<link rel="import" href="../paper-button/paper-button.html">
```

Next, add the **<paper-input>** tag to declare the Polymer element: **<paper-input ... value="{{data.triNickNameTX}}"></paper-input>**

Then, add the **<paper-button>** tag to declare it and call a JavaScript method **updateActionHandler** when the button is tapped: **<paper-button raised on-tap="updateActionHandler">Trigger now!</paper-button>**

HTML File > Declare paper-input and paper-button.

```
                <paper-material z="1">
                        Starter View for <b>jay-ux-people-view</b>

                        <p>Hello World! This is my 3rd UX view!</p>

                        <paper-input label="First Name" floating-label value="
                        {{data.triFirstNameTX}}"></paper-input>
                        <paper-input label="Last Name" floating-label value="
                        {{data.triLastNameTX}}"></paper-input>
                        <paper-input label="Nick Name" floating-label value="
                        {{data.triNickNameTX}}"></paper-input>

                        <paper-button raised on-tap="updateActionHandler">
                        Trigger now!</paper-button>

                </paper-material>
```

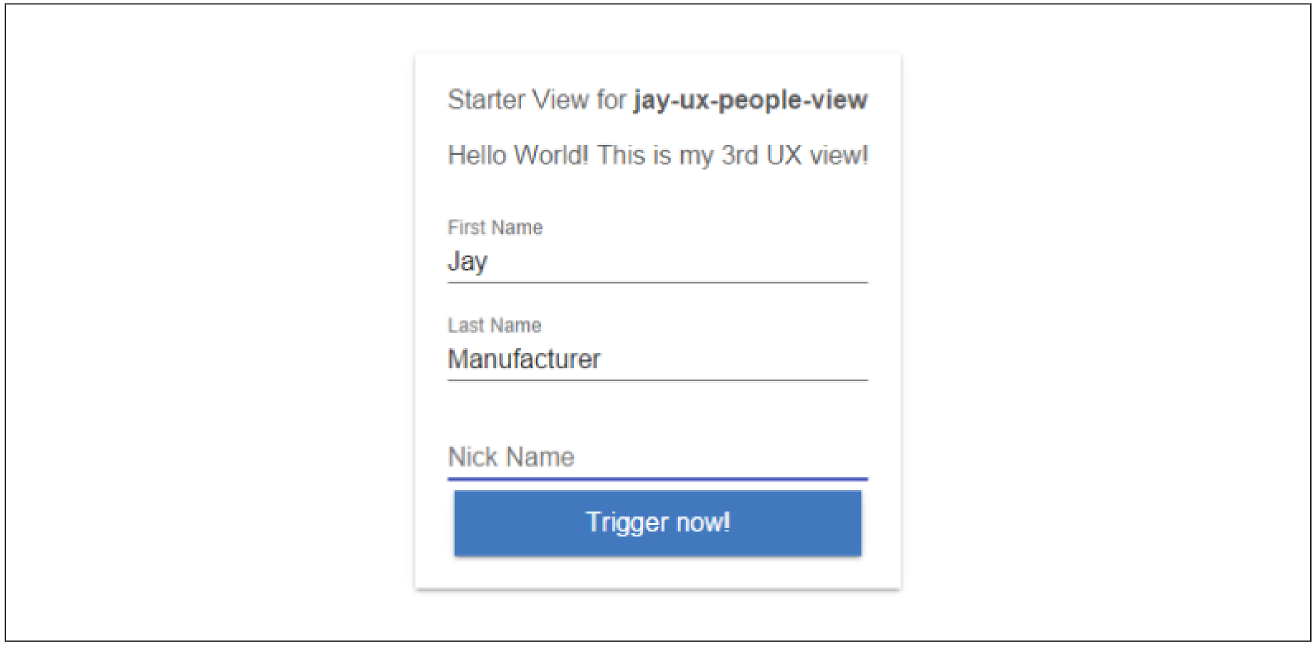
Insert the JavaScript method **updateActionHandler** within the **<script>** tag.

HTML File > Insert updateActionHandler.

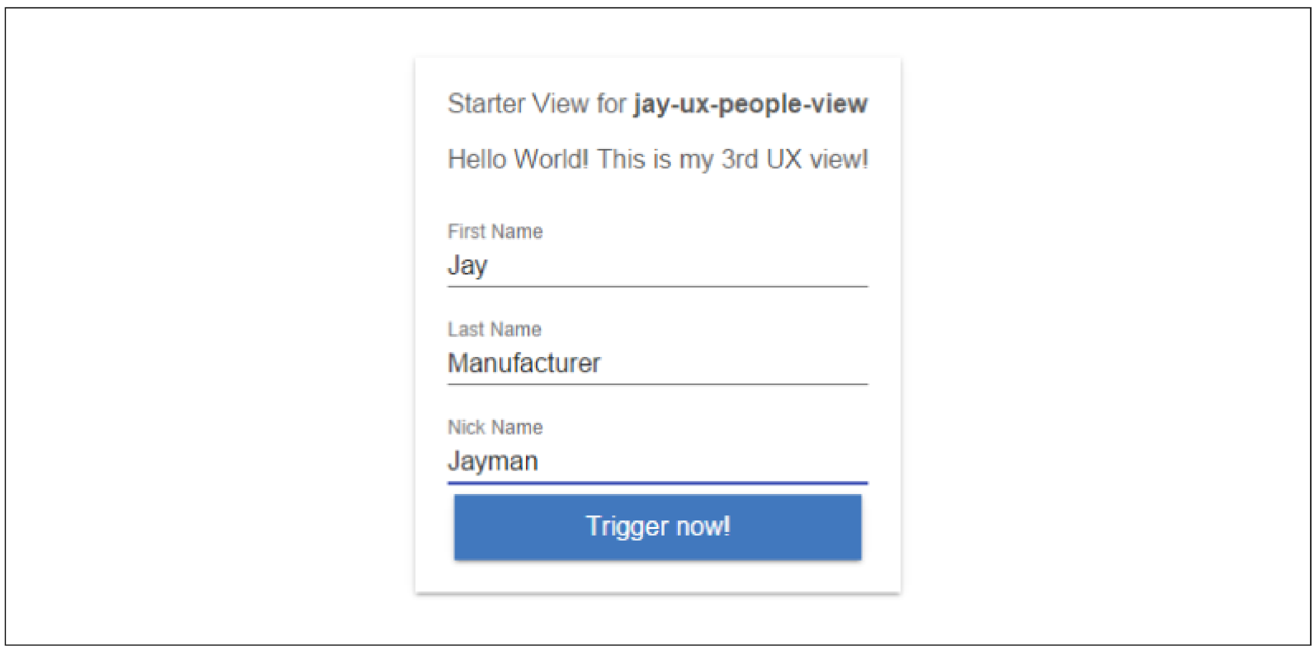
```
<script>
        Polymer({

                is: "jay-ux-people-view",

                behaviors: [TriPlatViewBehavior],

                updateActionHandler: function() {
                        this.$.model.updateRecord(127333408,TriPlatDs.RefreshType.
                        BOTH,"jayUXPeopleActionGroup","
                        jayUXPeopleActionSavePerm");
                },

        });
</script>
```

Save the file and refresh the UX view. Click the **Trigger now!** button and refresh again. Do you see your modified instance data? Pretty nice!

UX App > Input with No Nick Name.


Starter View for jay-ux-people-view
Hello World! This is my 3rd UX view!
First Name
Jay
Last Name
Manufacturer
Nick Name
Trigger now!

UX App > Input with Modified Nick Name.


Starter View for jay-ux-people-view
Hello World! This is my 3rd UX view!
First Name
Jay
Last Name
Manufacturer
Nick Name
Jayman
Trigger now!

Can we explore more advanced functions?

Maybe a few more? At this point, you should have an even better idea of how UX applications are built and how UX views are designed with Polymer components. But what about designing a view with your own customized reusable component? Since you're already here, why not?

<u>Start a card view as a custom component.</u>

For our next exercise, we'll prepare a new HTML file in the same folder as your existing HTML file. ***But don't move it into the same folder yet!*** First, edit the new HTML file in a different folder to begin with the simplest running design. This new HTML file will start your custom component.

In our example, we'll name the new HTML file **jay-ux-people-card.html** for our card-view custom component, and edit its HTML to look like this.

HTML File > Custom Component.

```
<dom-module id="jay-ux-people-card">
     <template>

          <p>This is my card-view custom component.</p>

     </template>
</dom-module>
<script>
     Polymer({

          is: "jay-ux-people-card",

     });
</script>
```

Next, after you've prepared your HTML file, open the command prompt in your selected folder, and run the **sync** command to sync your changes.

WebViewSync > Sync.

```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar sync -a
Waiting for changes to sync...
```

This time, you're ready to move your new HTML file. After we move this file, and hook it up into your main view, we can expand the component.

In our example, the **C:\tririga_ux\ux_server** folder contains the **jay-ux-people-view** folder, which contains the existing **jay-ux-people-view.html** file from the last exercise. Go ahead and move the new HTML file **jay-ux-people-card.html** beside the existing HTML file in the same folder.

Notice how the **WebViewSync** tool detects and pushes the new HTML file.

WebViewSync > Push.

```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar sync -a
Waiting for changes to sync...
Signing On To TRIRIGA [success]
  [2015-09-02 20:30:29] [push]     /jay-ux-people-card.html
  [2015-09-02 20:30:29] [push]     /jay-ux-people-card.html
     [ok]
```

Add a card view component to your HTML file.

This time, we'll add a custom tag to our existing HTML file so it can grab the new custom component. In our example, we'll add a custom **<jay-ux-people-card>** tag to our existing **jay-ux-people-view.html** file.

First, add the **<link>** tag at the top to import the custom component: **<link rel="import" href="jay-ux-people-card.html">**

HTML File > Import Custom Component.

```
<link rel="import" href="../triplat-view-behavior/triplat-view-behavior.html">
<link rel="import" href="../triplat-ds/triplat-ds.html">

<link rel="import" href="../paper-material/paper-material.html">
<link rel="import" href="../paper-input/paper-input.html">
<link rel="import" href="../paper-button/paper-button.html">
<link rel="import" href="jay-ux-people-card.html">
```

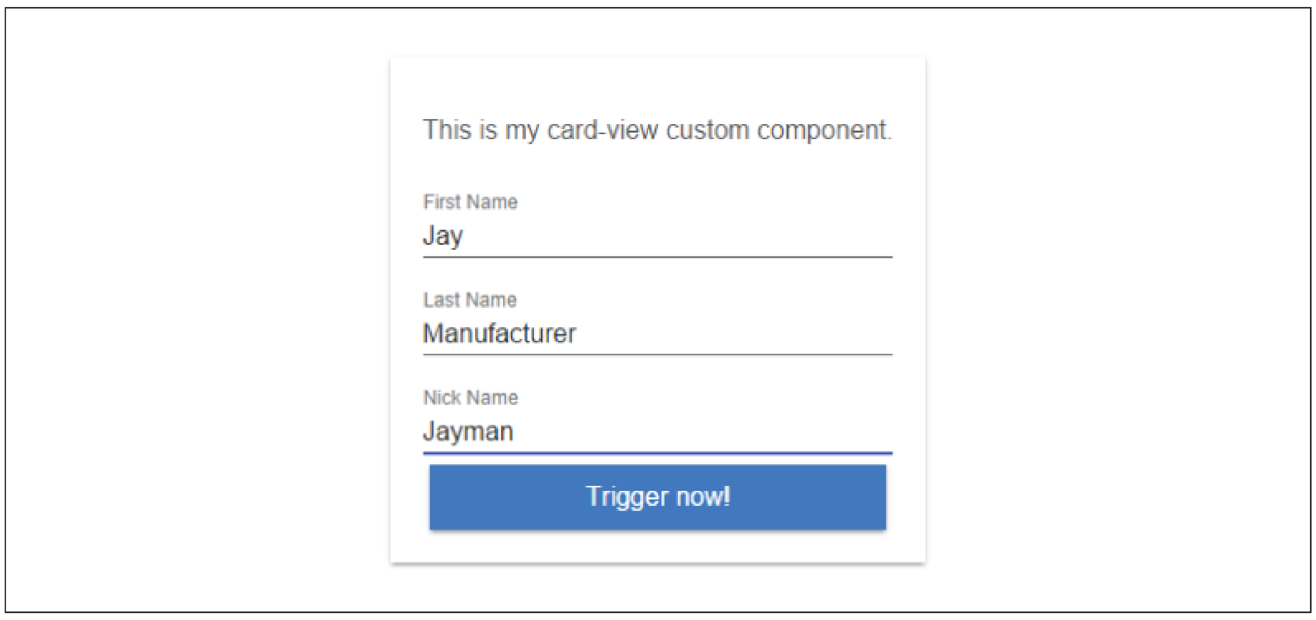
Next, add the **<jay-ux-people-card>** tag to declare the custom component: **<jay-ux-people-card></jay-ux-people-card>**

HTML File > Declare Custom Component.

```
        <paper-material z="1">
            <!-- Starter View for <b>jay-ux-people-view</b>

            <p>Hello World! This is my 3rd UX view!</p> -->

            <jay-ux-people-card></jay-ux-people-card>

            <paper-input label="First Name" floating-label value="{{data.triFirstNameTX}}"></paper-input>
            <paper-input label="Last Name" floating-label value="{{data.triLastNameTX}}"></paper-input>
            <paper-input label="Nick Name" floating-label value="{{data.triNickNameTX}}"></paper-input>

            <paper-button raised on-tap="updateActionHandler">Trigger now!</paper-button>

        </paper-material>
```

Save the file and refresh the UX view. Do you see your new custom component? Now that it's hooked up, we can expand it! Sweet, huh?

UX App > Custom Component.



This is my card-view custom component.

First Name: Jay

Last Name: Manufacturer

Nick Name: Jayman

Expand your card view component.

This time, we'll add a simple **peopleData** property to our component **jay-ux-people-card.html** to show that we can use component properties. Like before, we'll also add instance data **triFirstNameTX** and **triLastNameTX** from an existing people record, the same record from previous exercises.

First, insert the **peopleData** property within the **<script>** tag.

HTML File > Insert peopleData.

```
<script>
     Polymer({

          is: "jay-ux-people-card",

          properties: {

               peopleData: {
                    type: Object,
               },
          }

     });
</script>
```

Next, replace the placeholder text with **peopleData** values and a profile image. Ideally, the image data should be pulled from an existing people record. But for now, we'll add the profile image **jay-profile.png** locally. Go ahead and move the profile image to the same **jay-ux-people-view** folder.

HTML File > Add peopleData Values and Profile Image.

```
<dom-module id="jay-ux-people-card">
     <template>

          <!-- <p>This is my card-view custom component.</p> -->

          <div class="layout vertical center">
               <img src="jay-profile.png">
               <h2><span>{{peopleData.triFirstNameTX}}</span>
               <span>{{peopleData.triLastNameTX}}</span></h2>
          </div>

     </template>
</dom-module>
```

At this point, how do you pass the model **data** values from your main view to the **peopleData** values in your custom component? Easy! In our example, we'll return to our main **jay-ux-people-view.html** file, and add the attribute **people-data="{{data}}"** to our custom **<jay-ux-people-card>** tag. This will assign **data** values to the **peopleData** property.

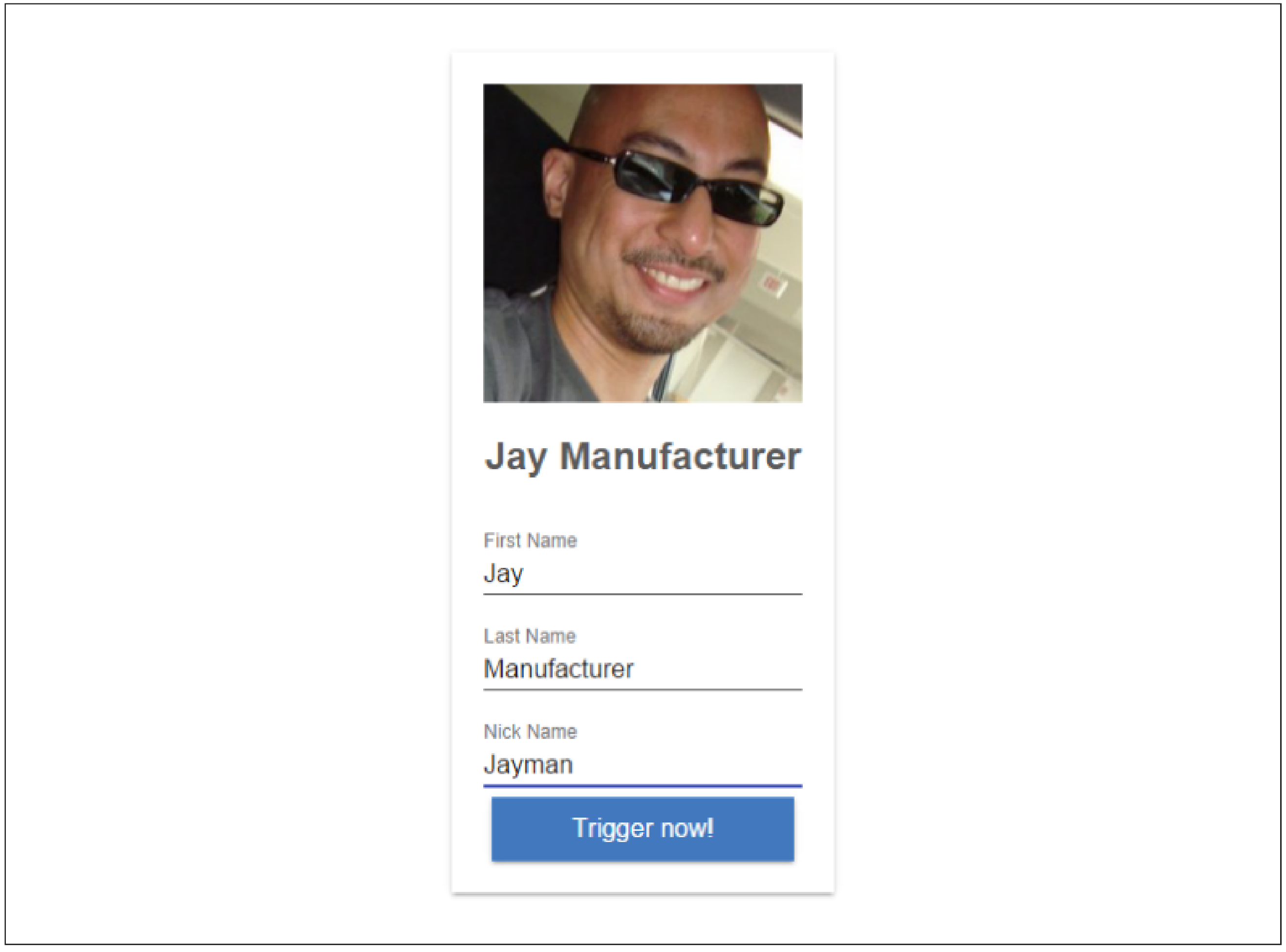
HTML File > Add people-data Attribute.

```
               <paper-material z="1">
                    <!-- Starter View for <b>jay-ux-people-view</b>

                    <p>Hello World! This is my 3rd UX view!</p> -->

                    <jay-ux-people-card people-data="{{data}}"></jay-ux-people-card>

                    <paper-input label="First Name" floating-label value="{{data.triFirstNameTX}}"></paper-input>
                    <paper-input label="Last Name" floating-label value="{{data.triLastNameTX}}"></paper-input>
                    <paper-input label="Nick Name" floating-label value="{{data.triNickNameTX}}"></paper-input>

                    <paper-button raised on-tap="updateActionHandler">Trigger now!</paper-button>

               </paper-material>
```

Why do we need a dash in the HTML tag attribute **people-data** instead of simply using **peopleData** like the component property? *Well, Polymer maps any attribute name with dashes to the corresponding property name by automatically converting attribute* **dash-case** *to property* **camelCase**.

Save the file and refresh the UX view. Do you see your expanded custom component? Now it's time to replace the local profile image!

UX App > Expanded Custom Component.



Display image data from an existing record.

Before we forget, return to your existing model and data source from the last exercise, and add the field **triImageIM** (Image).

If you remember, we added the profile image **jay-profile.png** locally. This time, we'll replace it and add the TRIRIGA **<triplat-image>** tag to our component **jay-ux-people-card.html** to grab the instance data **triImageIM** from an existing people record, the same record from previous exercises.

First, add the **<link>** tag at the top to import the TRIRIGA element: **<link rel="import" href="../triplat-image/triplat-image.html">**

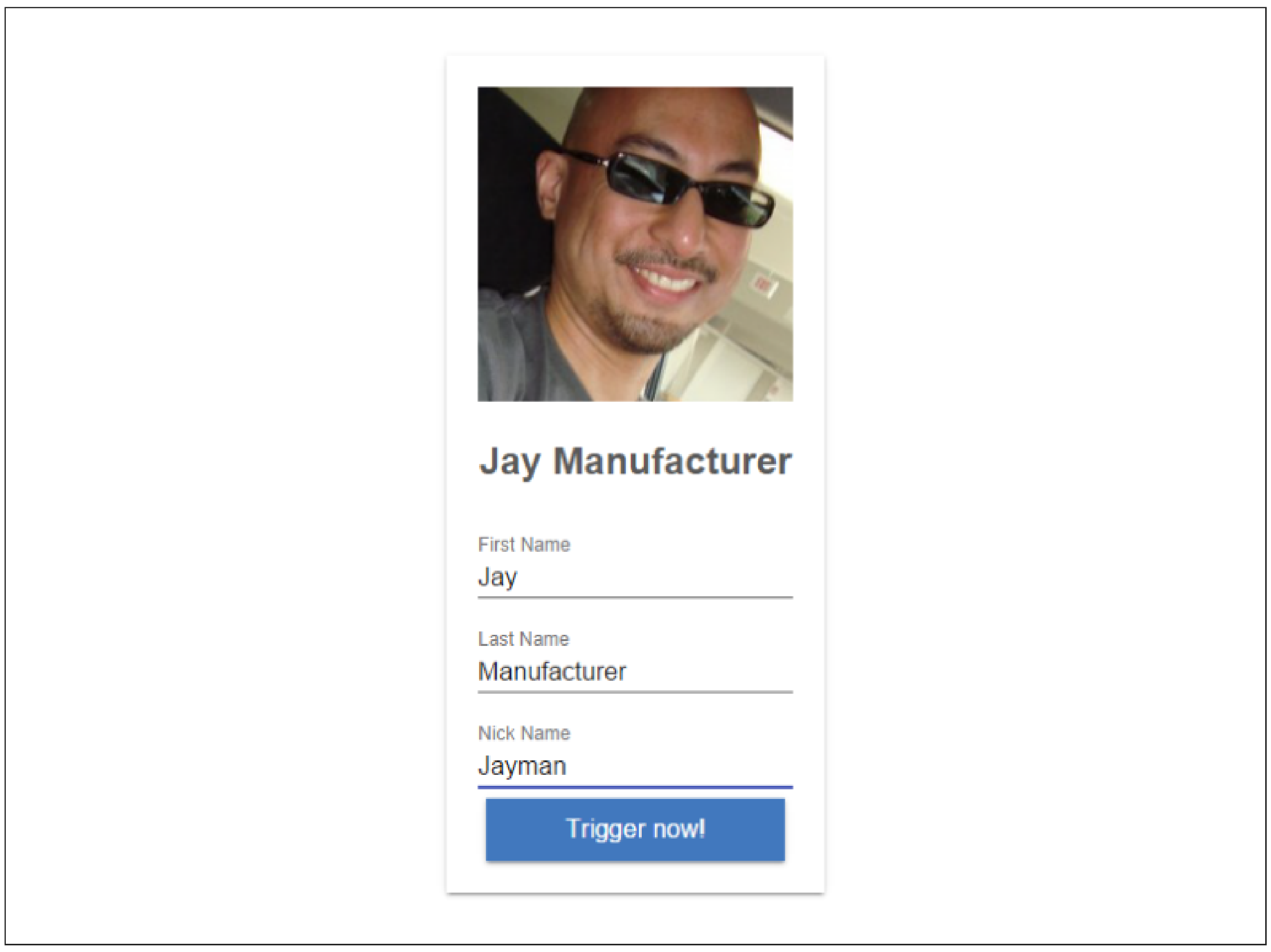
Next, add the **<triplat-image>** tag to declare the TRIRIGA element: **<triplat-image src="{{peopleData.triImageIM}}"></triplat-image>**

HTML File > Import and Declare triplat-image.

```
<link rel="import" href="../triplat-image/triplat-image.html">

<dom-module id="jay-ux-people-card">
     <template>

          <!-- <p>This is my card-view custom component.</p> -->

          <div class="layout vertical center">
               <triplat-image src="{{peopleData.triImageIM}}"></triplat-
               image>
               <h2><span>{{peopleData.triFirstNameTX}}</span>
               <span>{{peopleData.triLastNameTX}}</span></h2>
          </div>

     </template>
</dom-module>
```

Save the file and refresh the UX view. Do you see your completed custom component? Now you're ready to reuse it in other applications!

UX App > Completed Custom Component.



Still want more?

If you have any questions about UX that weren't answered in this article, feel free to reach out to your IBM TRIRIGA representative or business partner. In the meantime, here are some more background questions and answers from my previous articles that might help to fill in the gaps or give you a better idea of what we're trying to do. In any case, stay tuned!

Background Q & A.

Question	Answer
How do we build applications in the new MVC framework?	While we can't promise any specific dates, we plan to develop a new "model designer" or "model builder" metadata construct that supports the model. Similarly, we plan to develop a new "view designer" or "view builder" metadata construct that supports the view. Fortunately, we don't need to develop a new metadata construct for the controller since existing workflows and state families can already serve this function. Meanwhile, if we store the new platform metadata as records that can be accessed through forms, we can more quickly react to business requirements and add features.
How do we simplify the interface or view?	Our existing technology ties forms to "things" like people and locations. So why not change the pattern so that views are tied to "actions" like creating and submitting requests? This change could be accomplished by designing views that are specific to a user role. Then we could still reuse our existing business objects and workflows to support the new role-based interfaces.
What happens to our existing customers?	Because the new views will be "bolt-on" interfaces that are "bolted onto" existing applications, customers who don't choose the new MVC framework won't be affected. But for customers who choose the new framework, results could vary depending on how new role-based interfaces are applied and how much the application is customized. Fortunately, a flexible MVC model would offer customers a more efficient customization and upgrade strategy. For example, customers could add their own business objects instead of adding fields to our shipped business objects. This scenario would be easier to track during upgrade.

Next >