TRIRIGA Wiki Home
Facilities Management …
Facilities Maintenance
Environmental & Ener…
Real Estate Management
Capital Project Manag…
CAD Integrator-Publis…
IBM TRIRIGA Connect…
IBM TRIRIGA Anywhere
IBM TRIRIGA Applicati…
Release Notes
Media Library
Best Practices
Upgrading
Troubleshooting
UX Framework
UX Articles
UX App Building
Introducing UX
Implementing UX
Extending UX
Implementing UX (…
Extending UX (Pol…
Converting UX to P…
Bundling UX (Poly…
Commanding UX (…
UX Perceptive Apps
UX in Foundation To…
UX App Designer Tools
UX Best Practices
UX in Foundation Docs
UX Component Docs
UX Tips & Tricks
UX Videos
UX Archives
New Page
Index
Members
Trash
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 (Polymer 3)

Extending UX (Polymer 3)

Like | Updated March 28, 2019 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 Polymer website at www.polymer-project.org for more about Polymer 3. See the NPM website at www.npmjs.com for more about Node.js. See the "Extending UX" wiki page for previous Polymer 1 versions of this content.

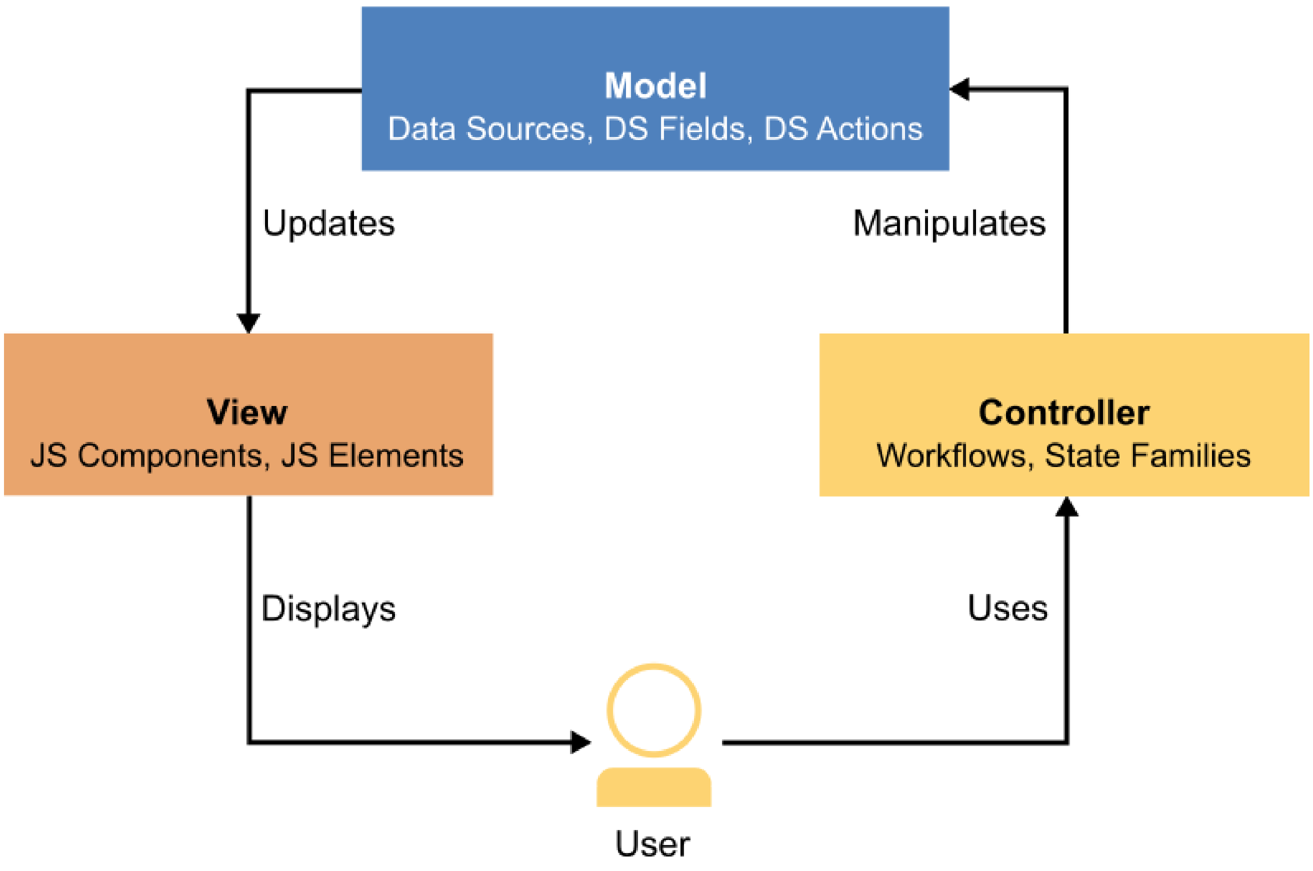
Extending UX (Polymer 3): Adding more functionality to your UX application

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
JS Components, JS 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 JavaScript (JS) files. In turn, each JS file can be made up of the following components:

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

Members

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 main JavaScript (JS) file.

Here are the basic steps:

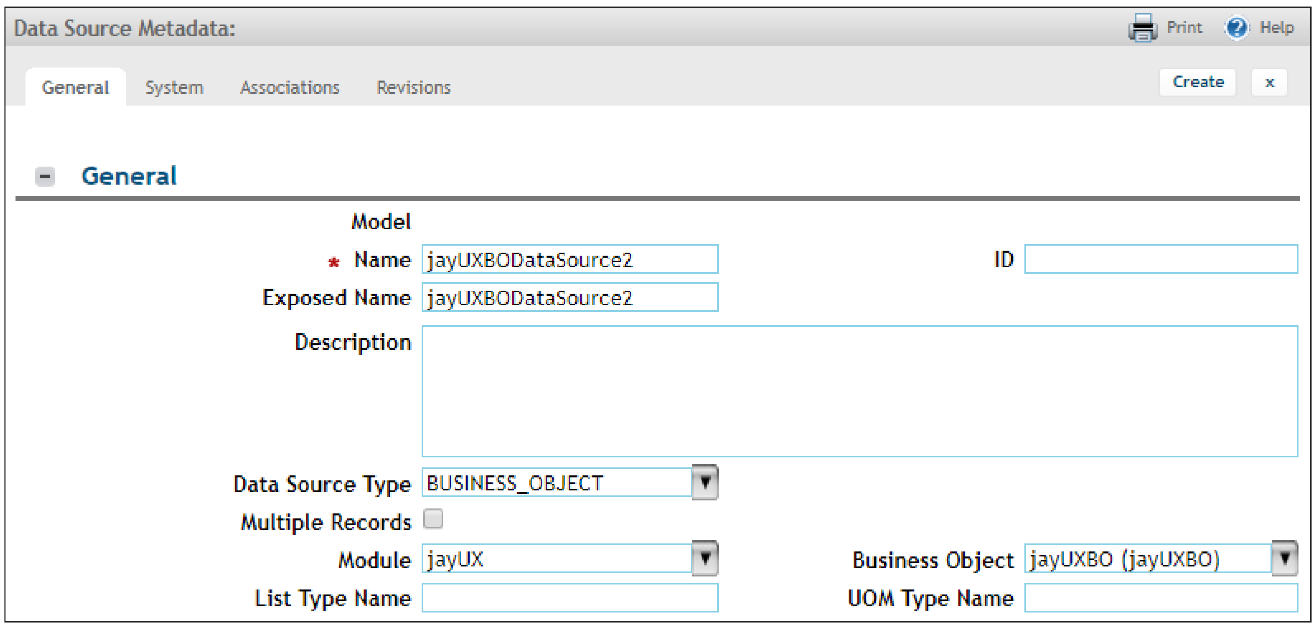
- Define your model.
  - Optional: Add the business object.
  - 1: Add the model.
  - 2: Add the data source.
  - 3: Add a few fields for your data source.
- Define your view connections.
  - 4: Add the view.
  - 5: Add the model-and-view.
  - 6: Add the application for your model-and-view.
- Define your view.
  - 7: Install the NPM and TRIRIGA tools.
  - 8: Add the main JS file for your view.
  - 9: Access the application.
- Design your view.
  - 10: Start the tri-proxy tool.
  - 11: Add a paragraph element to your JS file.
  - 12: Add a few field elements to your JS file.
  - 13: Add a few button elements to your JS 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.



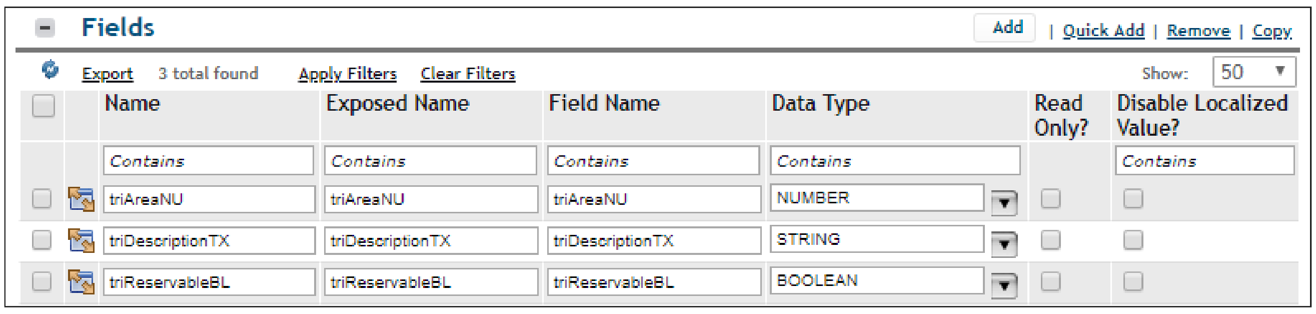
Data Source Metadata:

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.



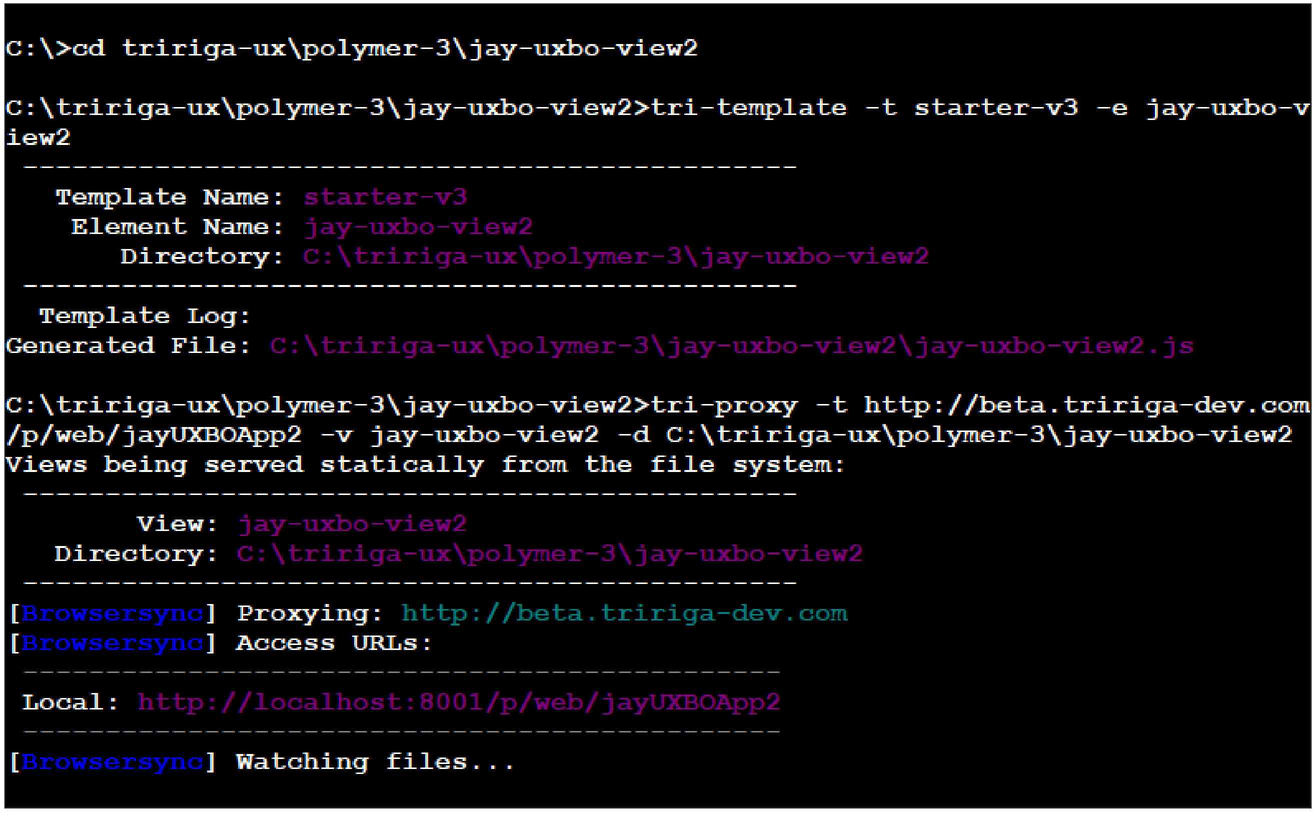
Fields

□	Name	Exposed Name	Field Name	Data Type	Read Only?	Disable Localized Value?
	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 **tri-template** command if needed, and run the **tri-proxy** command to preview your changes. In our example, we'll run **tri-template** with **starter-v3** and **jay-uxbo-view2** to add a new JS view file. Then we'll run **tri-proxy** with **http://beta.tririga-dev.com/p/web/jayUXBOApp2**. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

NPM > tri-template and tri-proxy.



```
C:\>cd tririga-ux\polymer-3\jay-uxbo-view2

C:\tririga-ux\polymer-3\jay-uxbo-view2>tri-template -t starter-v3 -e jay-uxbo-v
iew2
 ----------------------------------------------
   Template Name: starter-v3
    Element Name: jay-uxbo-view2
       Directory: C:\tririga-ux\polymer-3\jay-uxbo-view2
 ----------------------------------------------
  Template Log:
Generated File: C:\tririga-ux\polymer-3\jay-uxbo-view2\jay-uxbo-view2.js

C:\tririga-ux\polymer-3\jay-uxbo-view2>tri-proxy -t http://beta.tririga-dev.com
/p/web/jayUXBOApp2 -v jay-uxbo-view2 -d C:\tririga-ux\polymer-3\jay-uxbo-view2
Views being served statically from the file system:
 ----------------------------------------------
        View: jay-uxbo-view2
   Directory: C:\tririga-ux\polymer-3\jay-uxbo-view2
 ----------------------------------------------
[Browsersync] Proxying: http://beta.tririga-dev.com
[Browsersync] Access URLs:
 -------------------------------------------
 Local: http://localhost:8001/p/web/jayUXBOApp2
 -------------------------------------------
[Browsersync] Watching files...
```

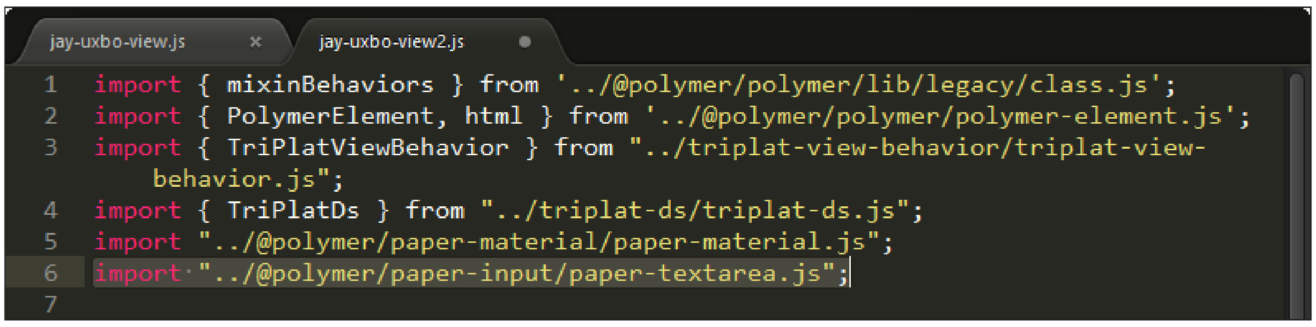
If needed, add the **import** line at the top of your JS file to import the TRIRIGA **triplat-ds** (data source) component and add the **<triplat-ds>** tag to declare it.

Add a text area field to your JS 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 **import** line at the top to import the Polymer element: **import "../@polymer/paper-input/paper-textarea.js";**

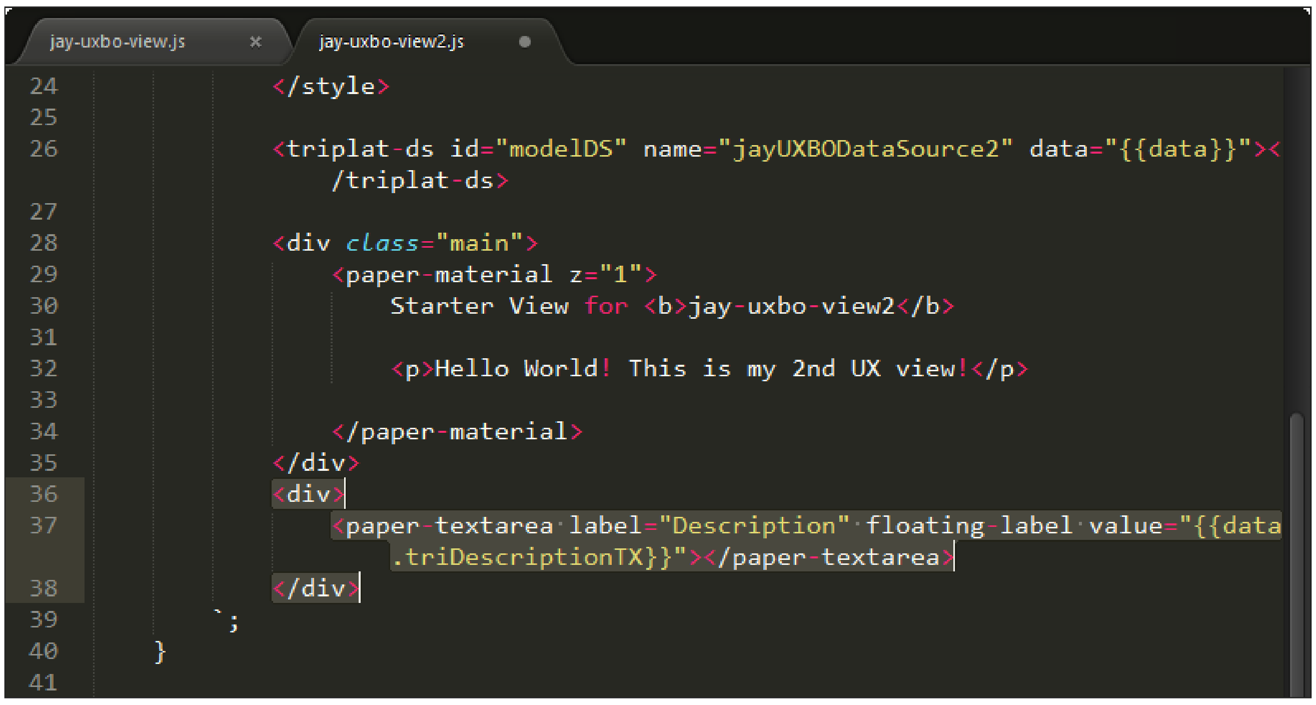
JS File > Import paper-textarea.



```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-textarea.js";

```

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

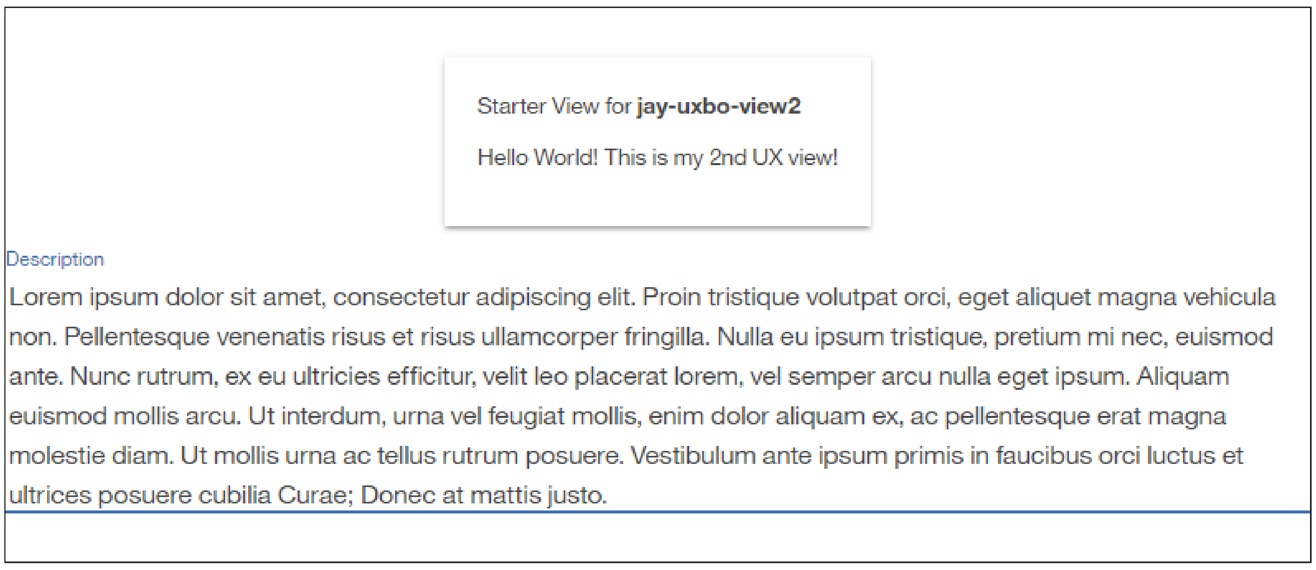
JS File > Declare paper-textarea.



```
            </style>

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

            <div class="main">
                <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>
        `;
    }

```

Save the file and return to 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 > Text Area.



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

Hello World! This is my 2nd UX view!

Description

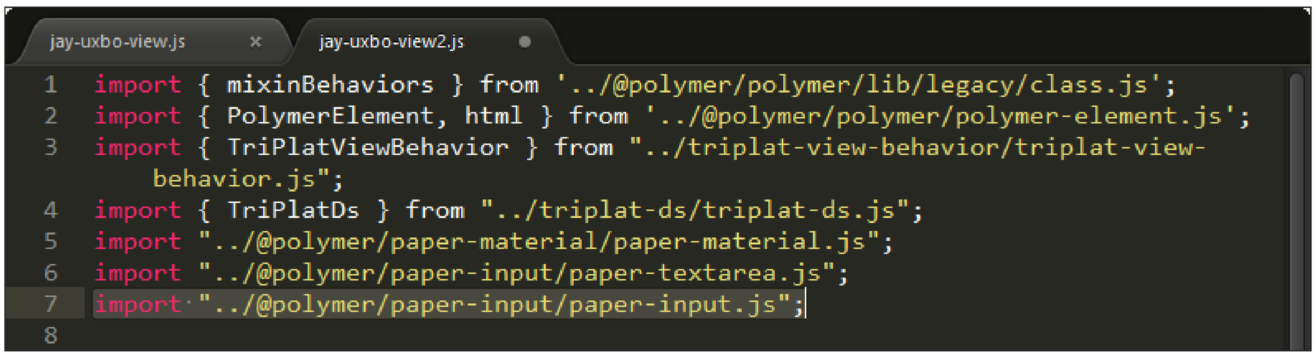
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin tristique volutpat orci, eget aliquet magna vehicula non. Pellentesque venenatis risus et risus ullamcorper fringilla. Nulla eu ipsum tristique, pretium mi nec, euismod ante. Nunc rutrum, ex eu ultricies efficitur, velit leo placerat lorem, vel semper arcu nulla eget ipsum. Aliquam euismod mollis arcu. Ut interdum, urna vel feugiat mollis, enim dolor aliquam ex, ac pellentesque erat magna molestie diam. Ut mollis urna ac tellus rutrum posuere. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Donec at mattis justo.

Add a number field to your JS 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 **tri-proxy** command is running in the command prompt. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

First, add the **import** line at the top to import the Polymer element: **import "../@polymer/paper-input/paper-input.js";**

JS File > Import paper-input.



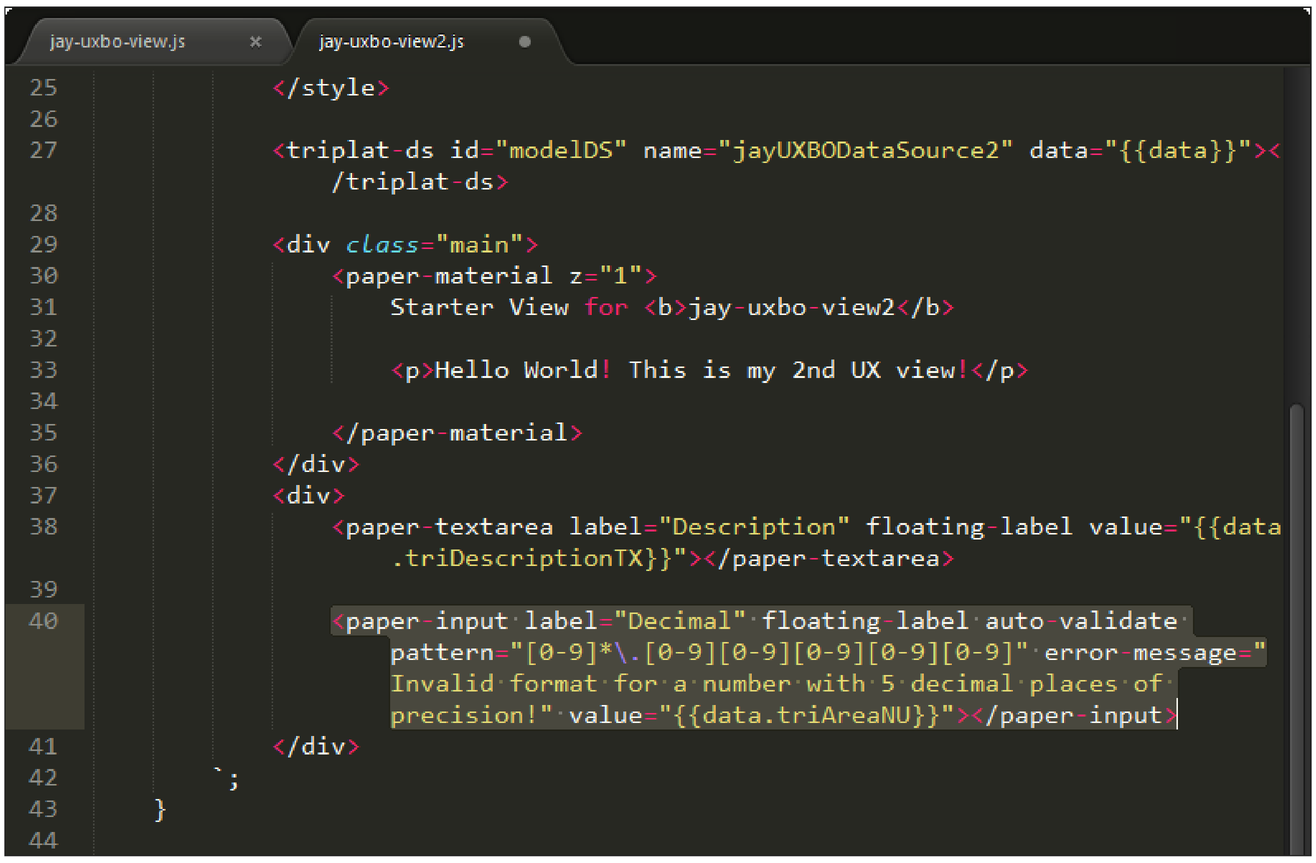
```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-textarea.js";
import "../@polymer/paper-input/paper-input.js";

```

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>**

Notes:

- An alternative to the Polymer **<paper-input>** element is the TRIRIGA **<triplat-paper-input>** component which includes built-in number validation.

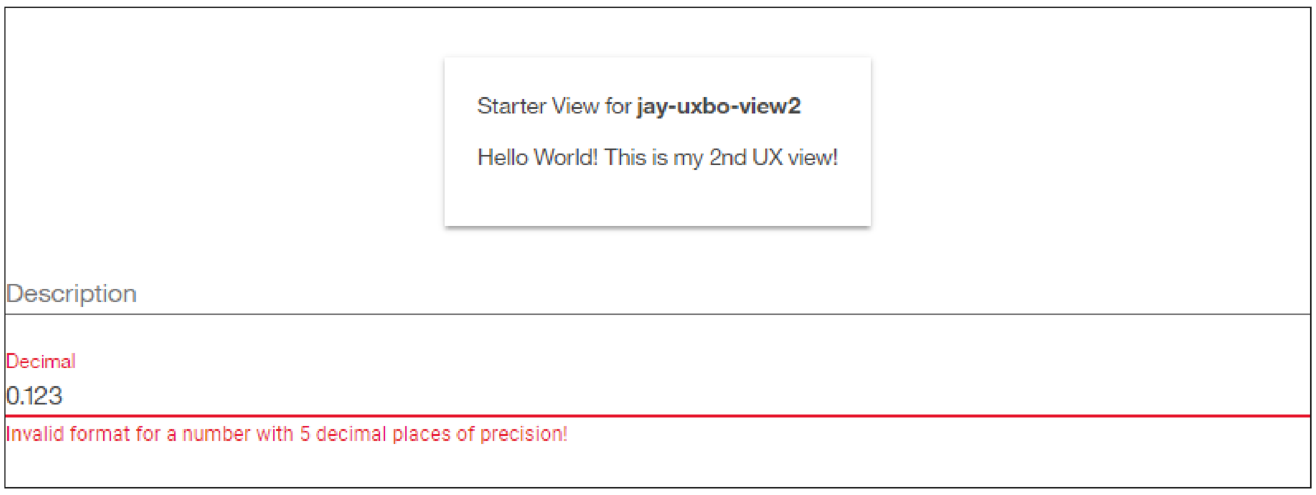
JS File > Declare paper-input.



```
            </style>

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

            <div class="main">
                <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>

                <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>
        `;
    }

```

Save the file and return to 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!

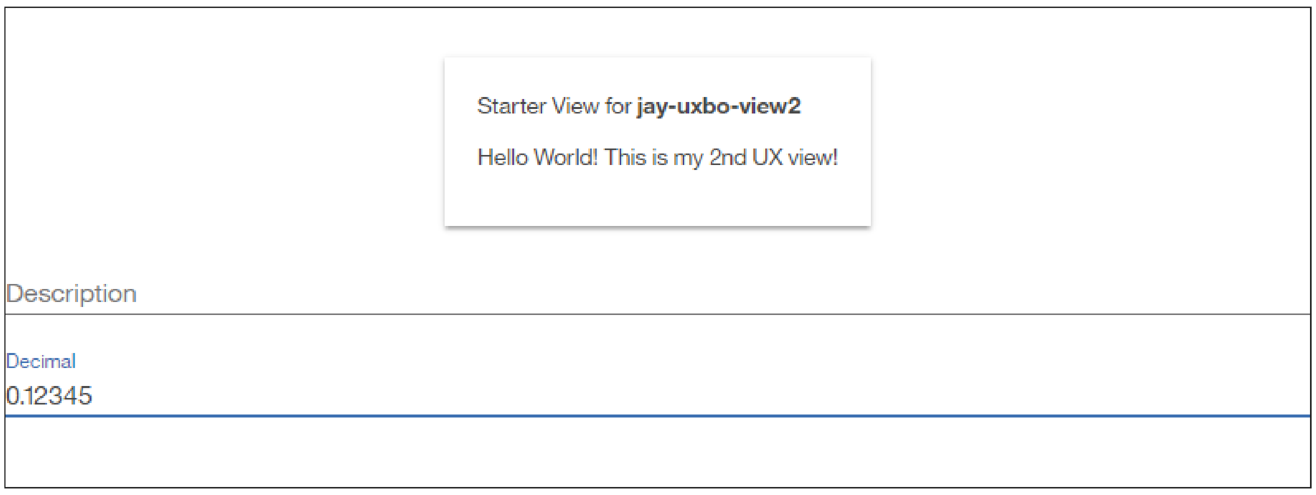
Description

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

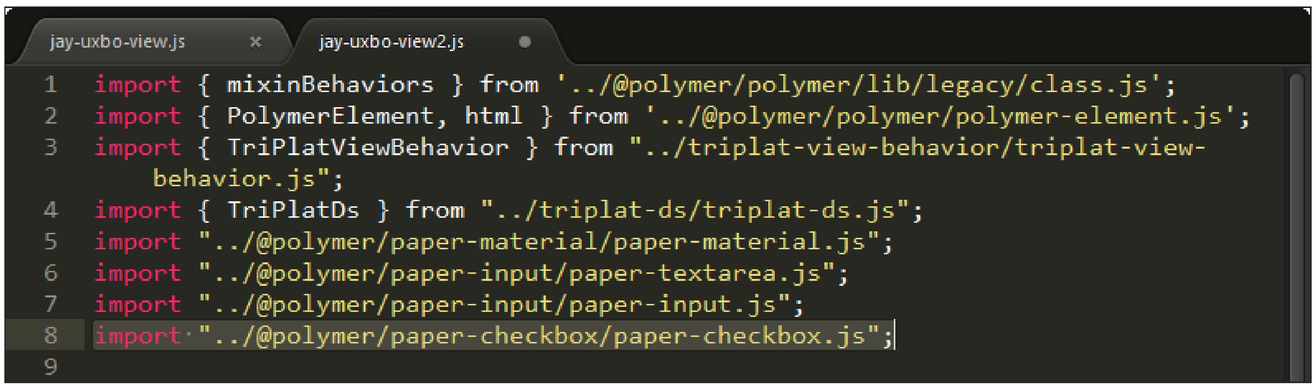
0.12345

Add a Boolean button to your JS 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 **tri-proxy** command is running in the command prompt. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

First, add the **import** line at the top to import the Polymer element: **import "../@polymer/paper-checkbox/paper-checkbox.js";**

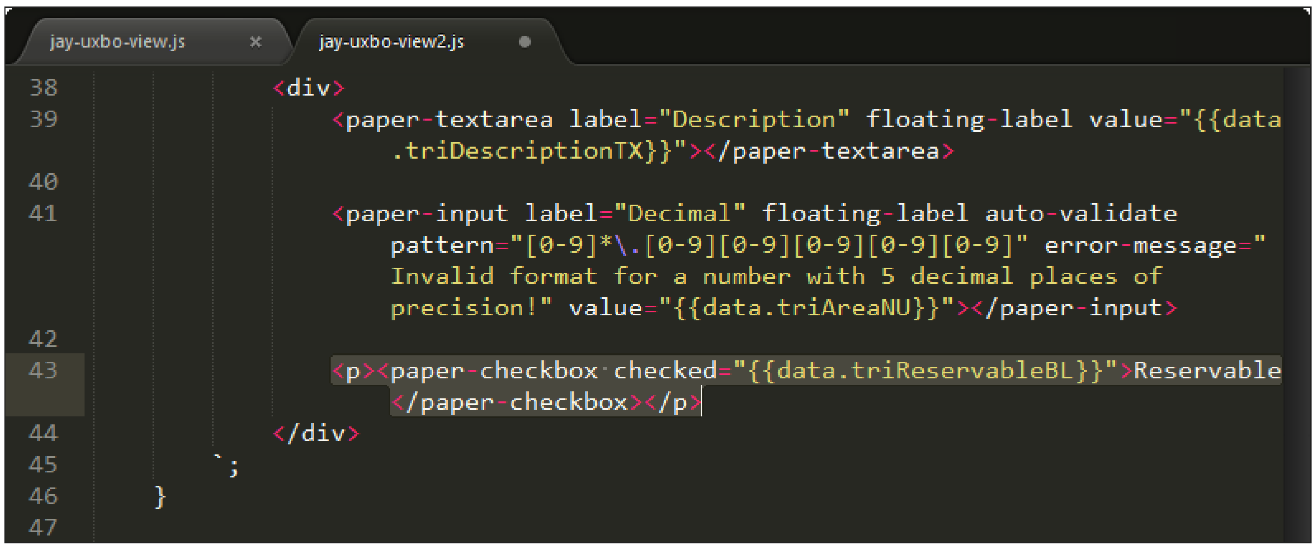
JS File > Import paper-checkbox.



```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-
    behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-textarea.js";
import "../@polymer/paper-input/paper-input.js";
import "../@polymer/paper-checkbox/paper-checkbox.js";

```

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

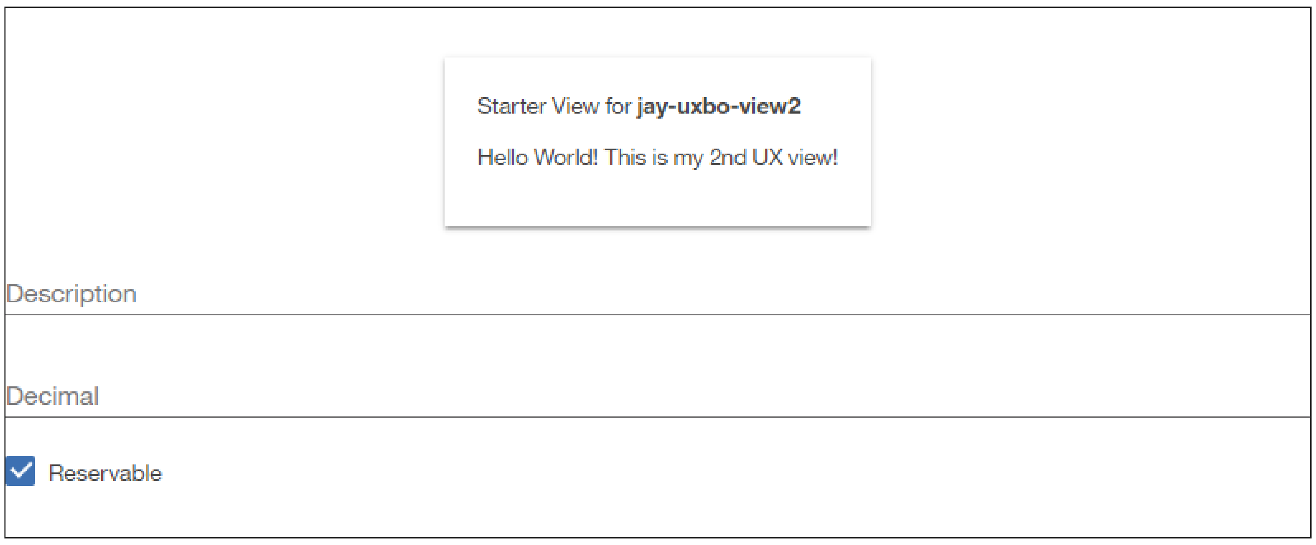
JS File > Declare paper-checkbox.



```
            <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></p>
            </div>
        `;
    }

```

Save the file and return to 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?

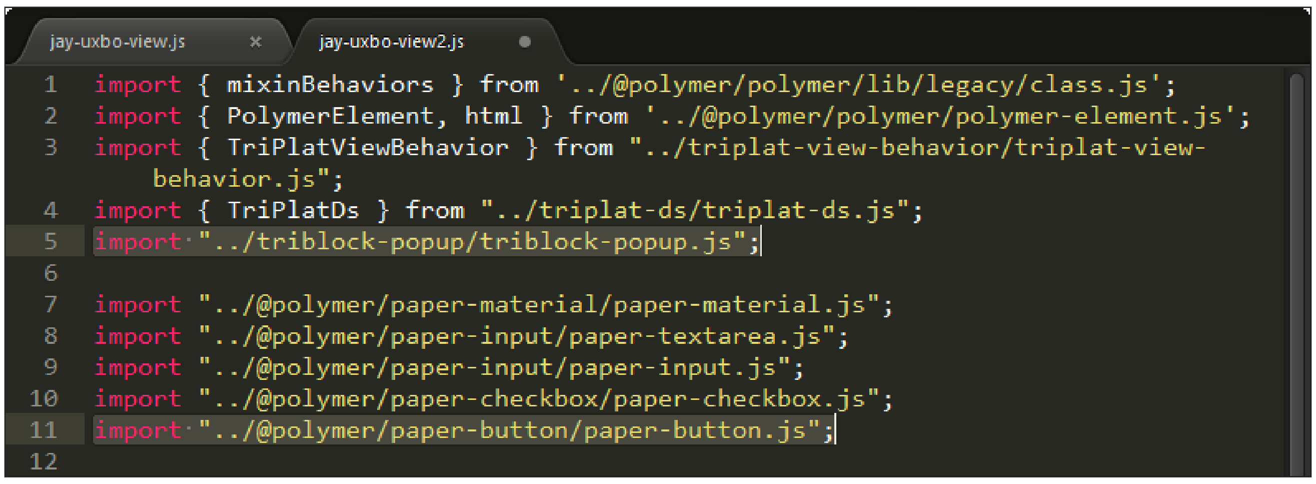
Why not? Like before, open the command prompt in your selected folder, run the **tri-template** command if needed, and run the **tri-proxy** command to preview your changes. In our example, we'll keep going with **jay-uxbo-view2**. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

Add an action dialog to your JS file.

This time, we'll add the Polymer **<paper-button>** tag for a button with a ripple effect and TRIRIGA **<triblock-popup>** 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 **import** lines at the top to import both components: **import "../@polymer/paper-button/paper-button.js";** and **import "../triblock-popup/triblock-popup.js";**

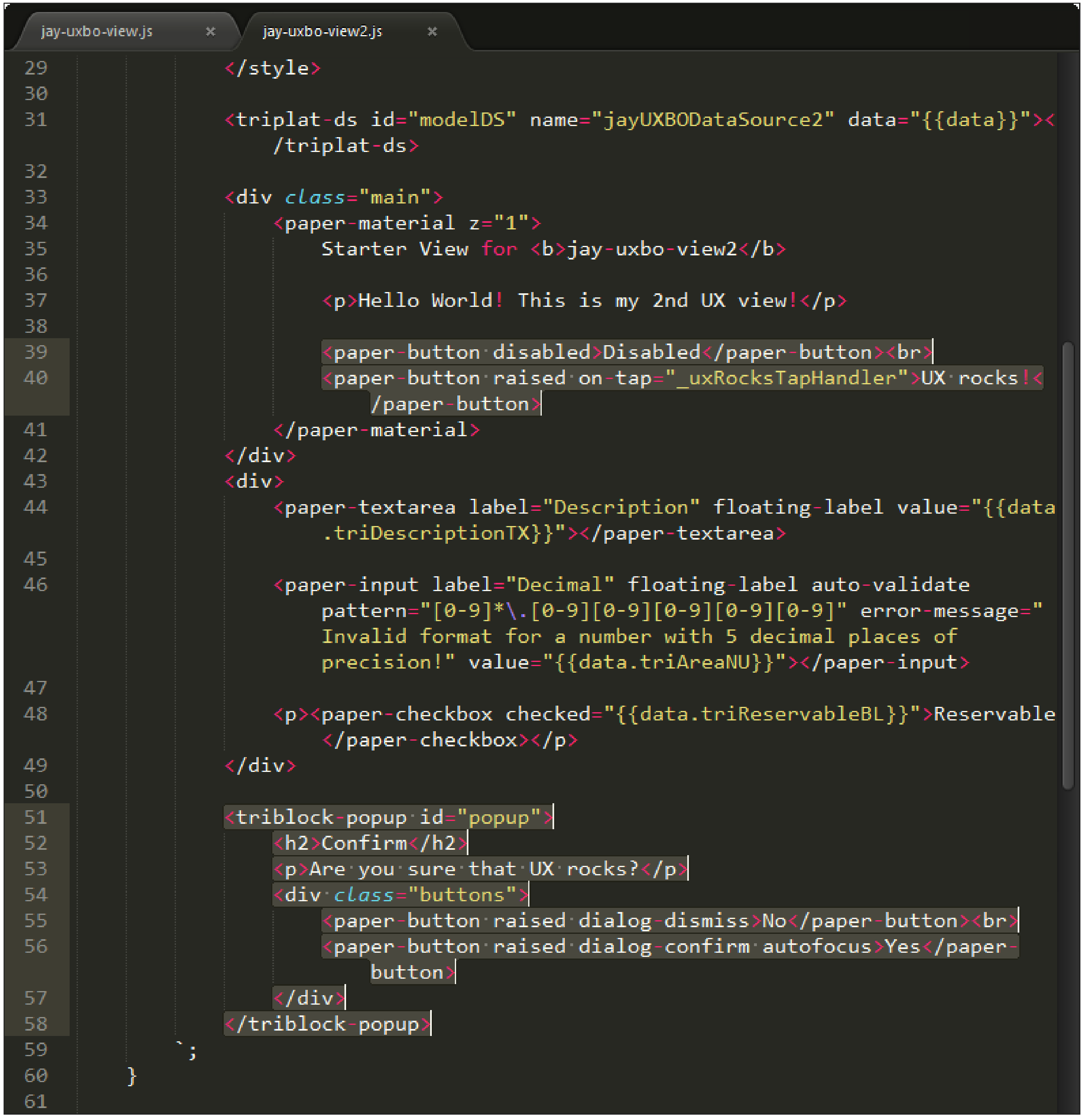
JS File > Import paper-button and triblock-popup.



```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-
    behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../triblock-popup/triblock-popup.js";

import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-textarea.js";
import "../@polymer/paper-input/paper-input.js";
import "../@polymer/paper-checkbox/paper-checkbox.js";
import "../@polymer/paper-button/paper-button.js";

```

Next, add the **<paper-button>** tag and **<triblock-popup>** tag to declare the components. For the **UX rocks!** button, insert the attribute **on-tap="_uxRocksTapHandler"** to call a JavaScript method **_uxRocksTapHandler** when the button event is detected.

JS File > Declare paper-button and triblock-popup.

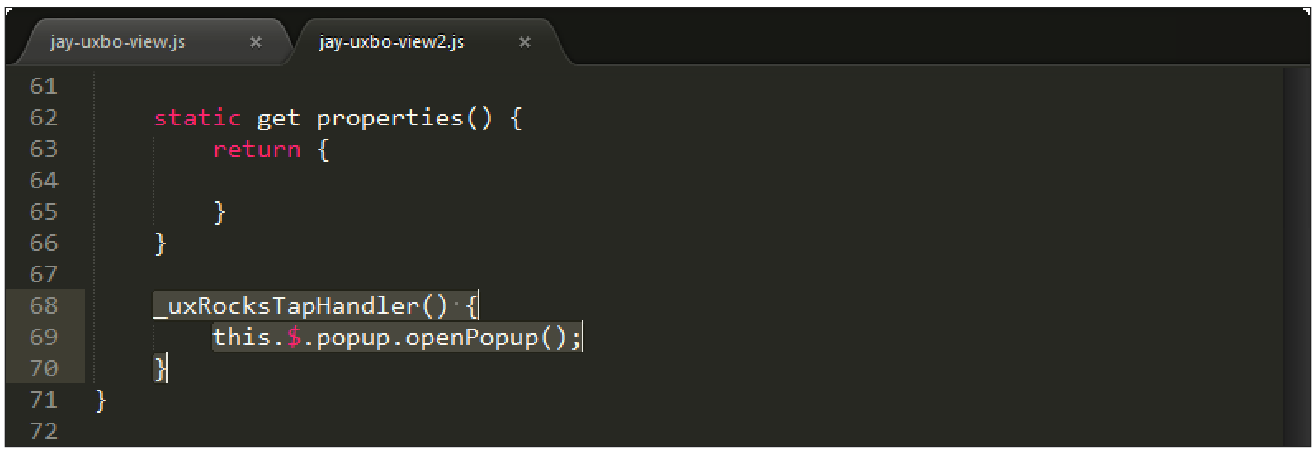


jay-uxbo-view.js ×   jay-uxbo-view2.js ×

```
            </style>

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

            <div class="main">
                <paper-material z="1">
                    Starter View for <b>jay-uxbo-view2</b>

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

                    <paper-button disabled>Disabled</paper-button><br>
                    <paper-button raised on-tap="_uxRocksTapHandler">UX rocks!<
                        /paper-button>
                </paper-material>
            </div>
            <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></p>
            </div>

            <triblock-popup id="popup">
                <h2>Confirm</h2>
                <p>Are you sure that UX rocks?</p>
                <div class="buttons">
                    <paper-button raised dialog-dismiss>No</paper-button><br>
                    <paper-button raised dialog-confirm autofocus>Yes</paper-
                        button>
                </div>
            </triblock-popup>
        `;
    }

```

Then, insert the JavaScript method **_uxRocksTapHandler** at the bottom.

JS File > Insert _uxRocksTapHandler.

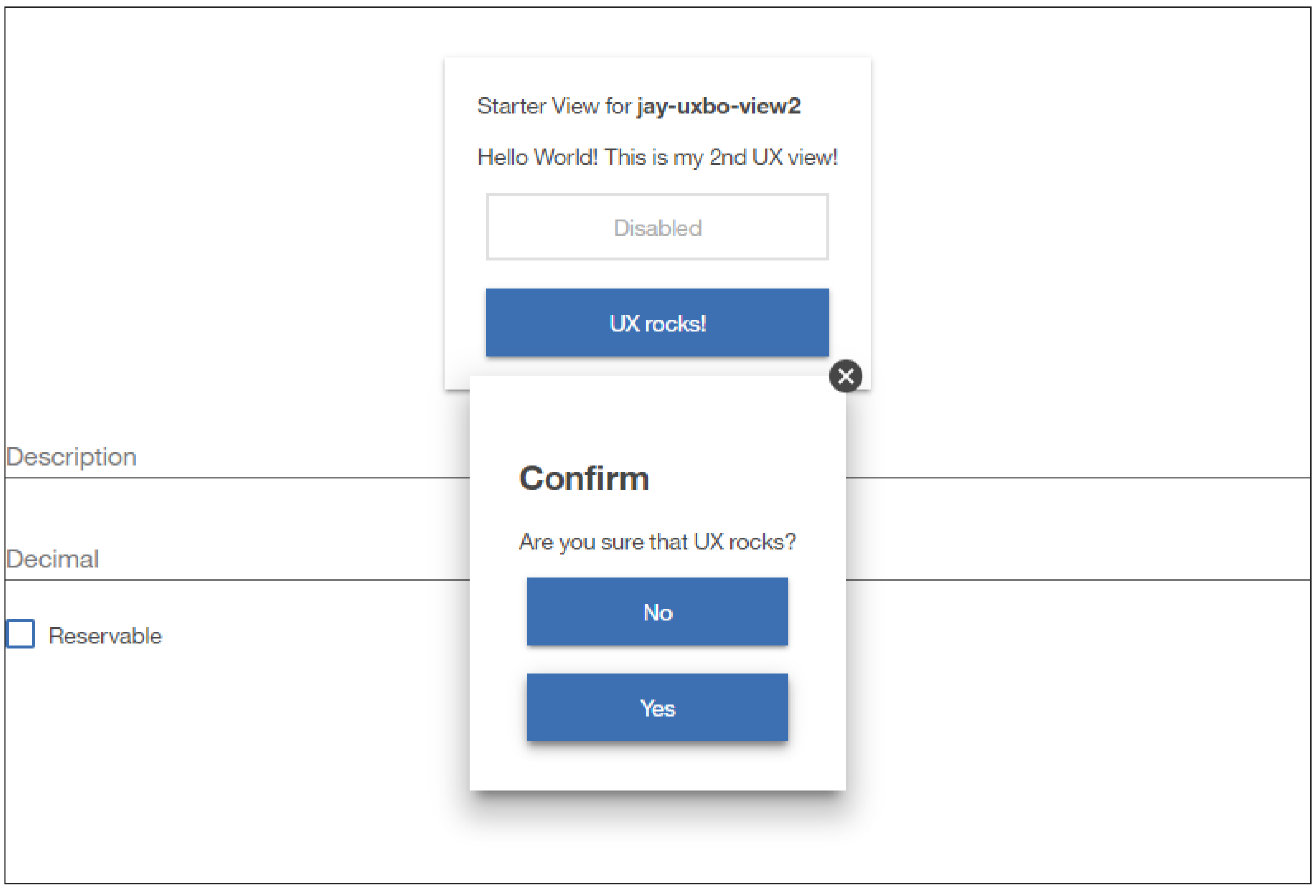


jay-uxbo-view.js ×   jay-uxbo-view2.js ×

```

    static get properties() {
        return {

        }
    }

    _uxRocksTapHandler() {
        this.$.popup.openPopup();
    }
}

```

Save the file and return to 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**

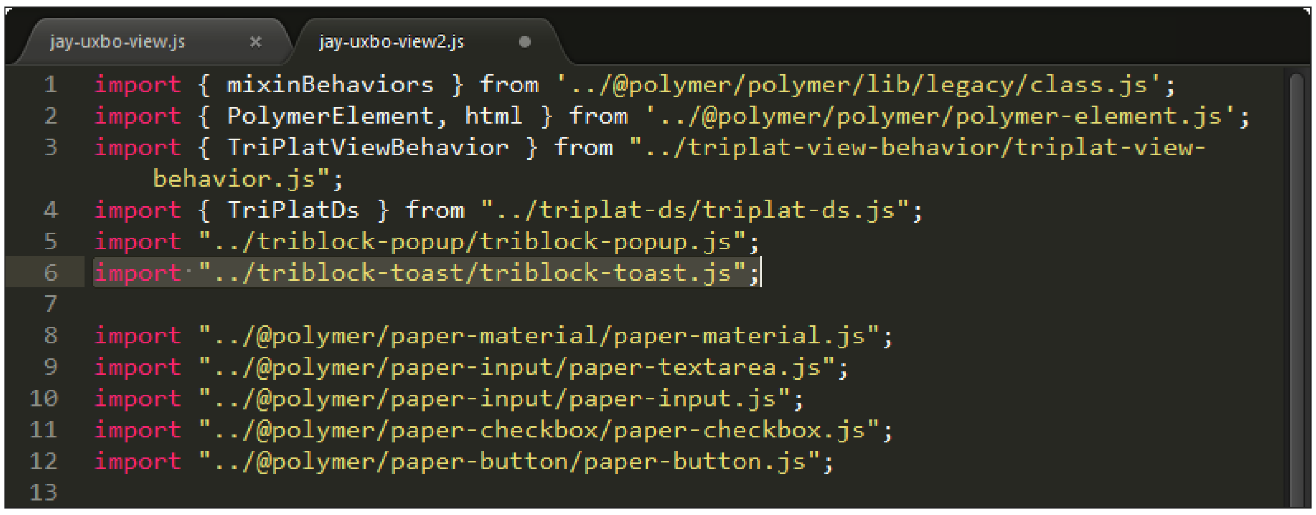
Hello World! This is my 2nd UX view!

Add a toast popup message to your JS file.

This time, we'll add the TRIRIGA **<triblock-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 (or 7000 milliseconds). Like before, make sure the **tri-proxy** command is running in the command prompt. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

First, add the **import** line at the top to import the TRIRIGA component: **import "../triblock-toast/triblock-toast.js";**

JS File > Import triblock-toast.

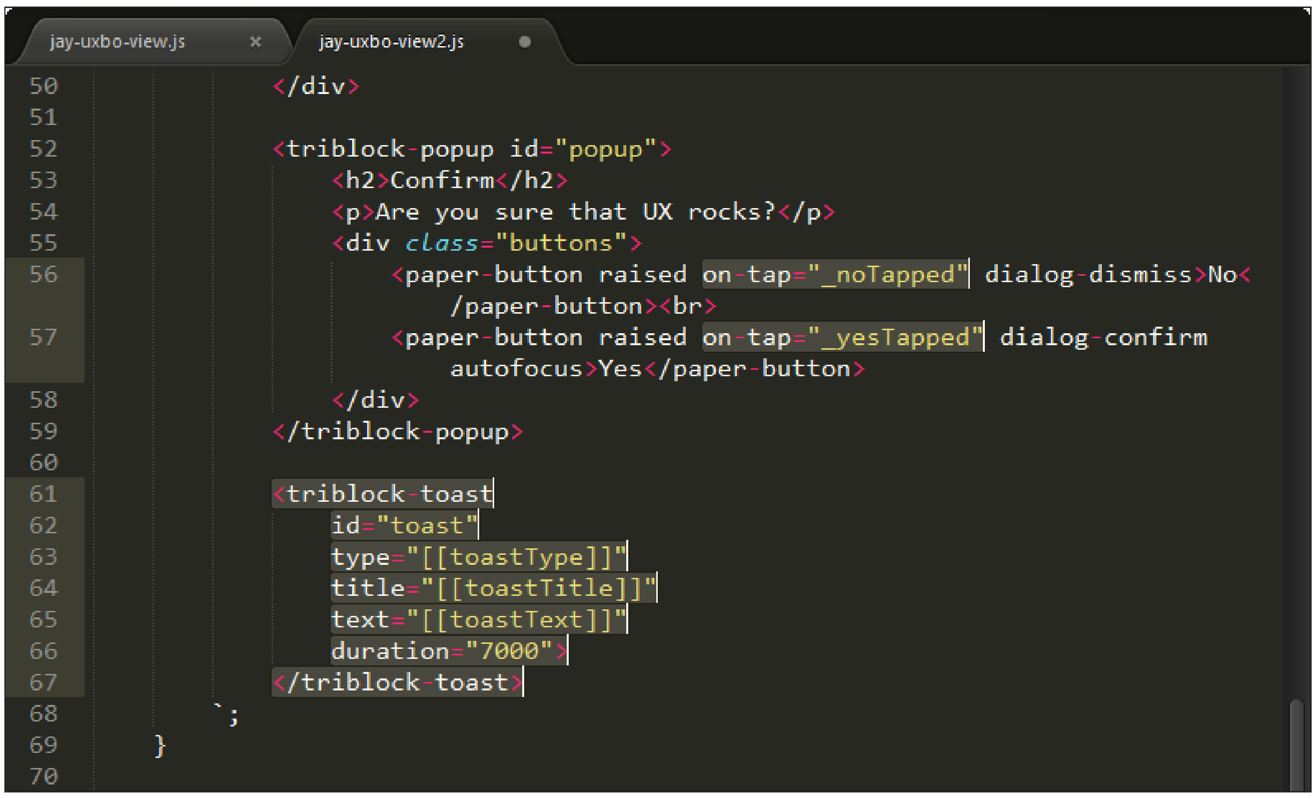


```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../triblock-popup/triblock-popup.js";
import "../triblock-toast/triblock-toast.js";

import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-textarea.js";
import "../@polymer/paper-input/paper-input.js";
import "../@polymer/paper-checkbox/paper-checkbox.js";
import "../@polymer/paper-button/paper-button.js";

```

Next, update the Polymer **<paper-button>** tags as follows. For the **No** button, insert the attribute **on-tap="_noTapped"** to call a JavaScript method **_noTapped** when the button event is detected. Likewise, for the **Yes** button, insert **on-tap="_yesTapped"**.

Then, add the **<triblock-toast>** tag to declare the TRIRIGA component: **<triblock-toast id="toast" type="[[toastType]]" title="[[toastTitle]]" text="[[toastText]]" duration="7000"></triblock-toast>**

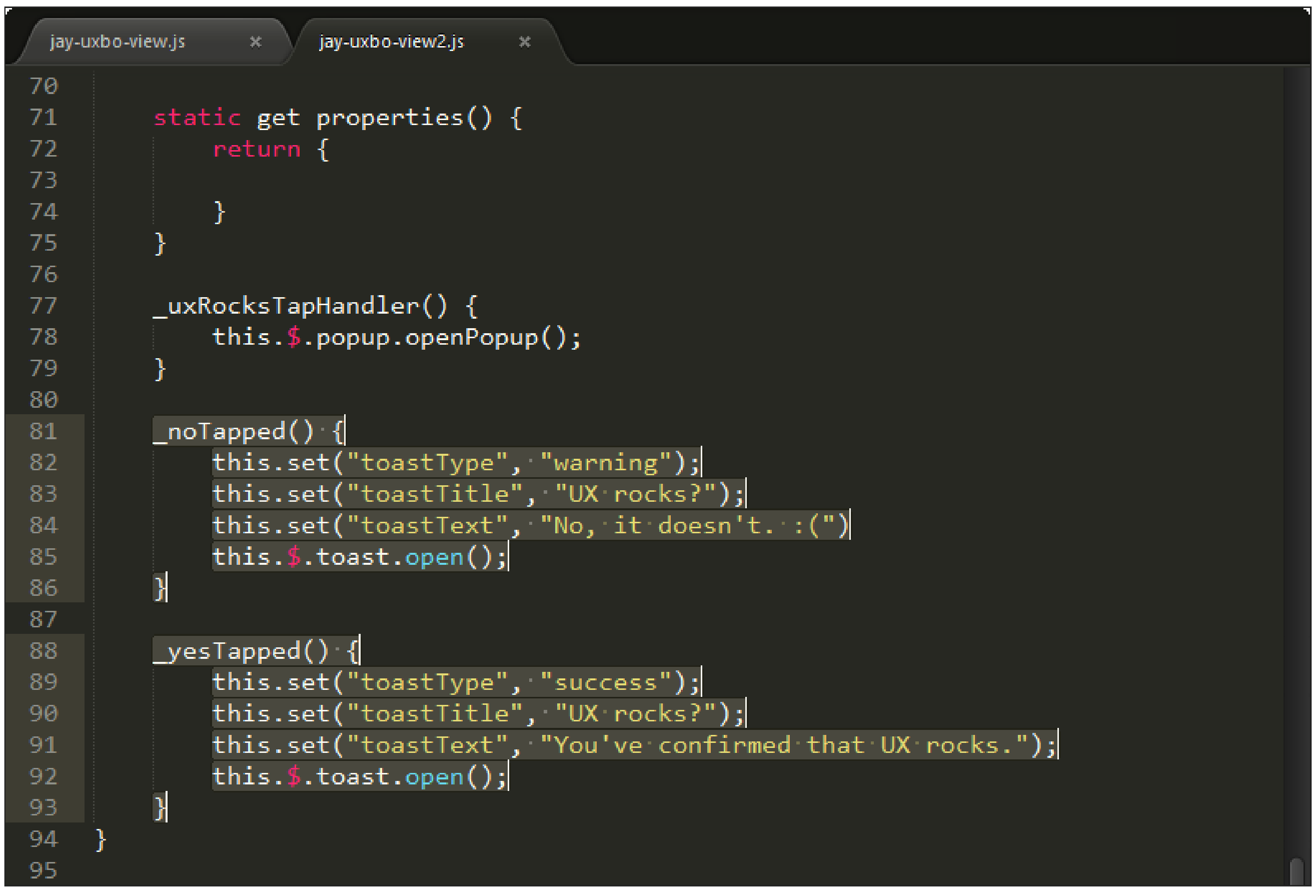
JS File > Update paper-button and declare triblock-toast.



```
            </div>

            <triblock-popup id="popup">
                <h2>Confirm</h2>
                <p>Are you sure that UX rocks?</p>
                <div class="buttons">
                    <paper-button raised on-tap="_noTapped" dialog-dismiss>No</paper-button><br>
                    <paper-button raised on-tap="_yesTapped" dialog-confirm autofocus>Yes</paper-button>
                </div>
            </triblock-popup>

            <triblock-toast
                id="toast"
                type="[[toastType]]"
                title="[[toastTitle]]"
                text="[[toastText]]"
                duration="7000">
            </triblock-toast>
        `;
    }

```

Then, insert the JavaScript methods **_noTapped** and **_yesTapped** at the bottom.

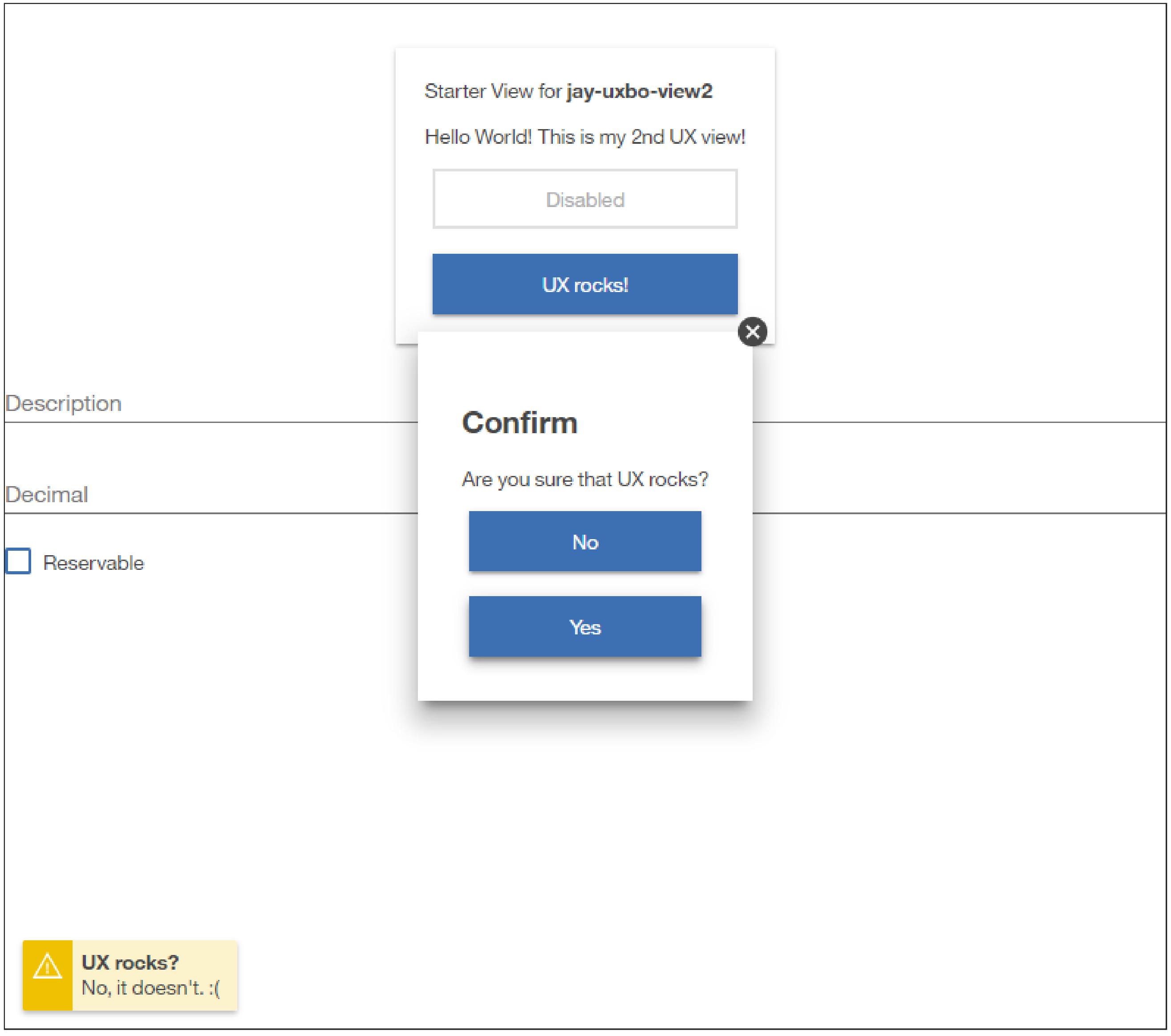
JS File > Insert _noTapped and _yesTapped.



```

    static get properties() {
        return {

        }
    }

    _uxRocksTapHandler() {
        this.$.popup.openPopup();
    }

    _noTapped() {
        this.set("toastType", "warning");
        this.set("toastTitle", "UX rocks?");
        this.set("toastText", "No, it doesn't.  :(")
        this.$.toast.open();
    }

    _yesTapped() {
        this.set("toastType", "success");
        this.set("toastTitle", "UX rocks?");
        this.set("toastText", "You've confirmed that UX rocks.");
        this.$.toast.open();
    }
}

```

Save the file and return to the UX view. Feel free to click the **UX rocks!** button, and then the **No** and **Yes** buttons. Do you see your toasts? Pretty sweet!

UX App > Toast Popup with Warning Style.



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

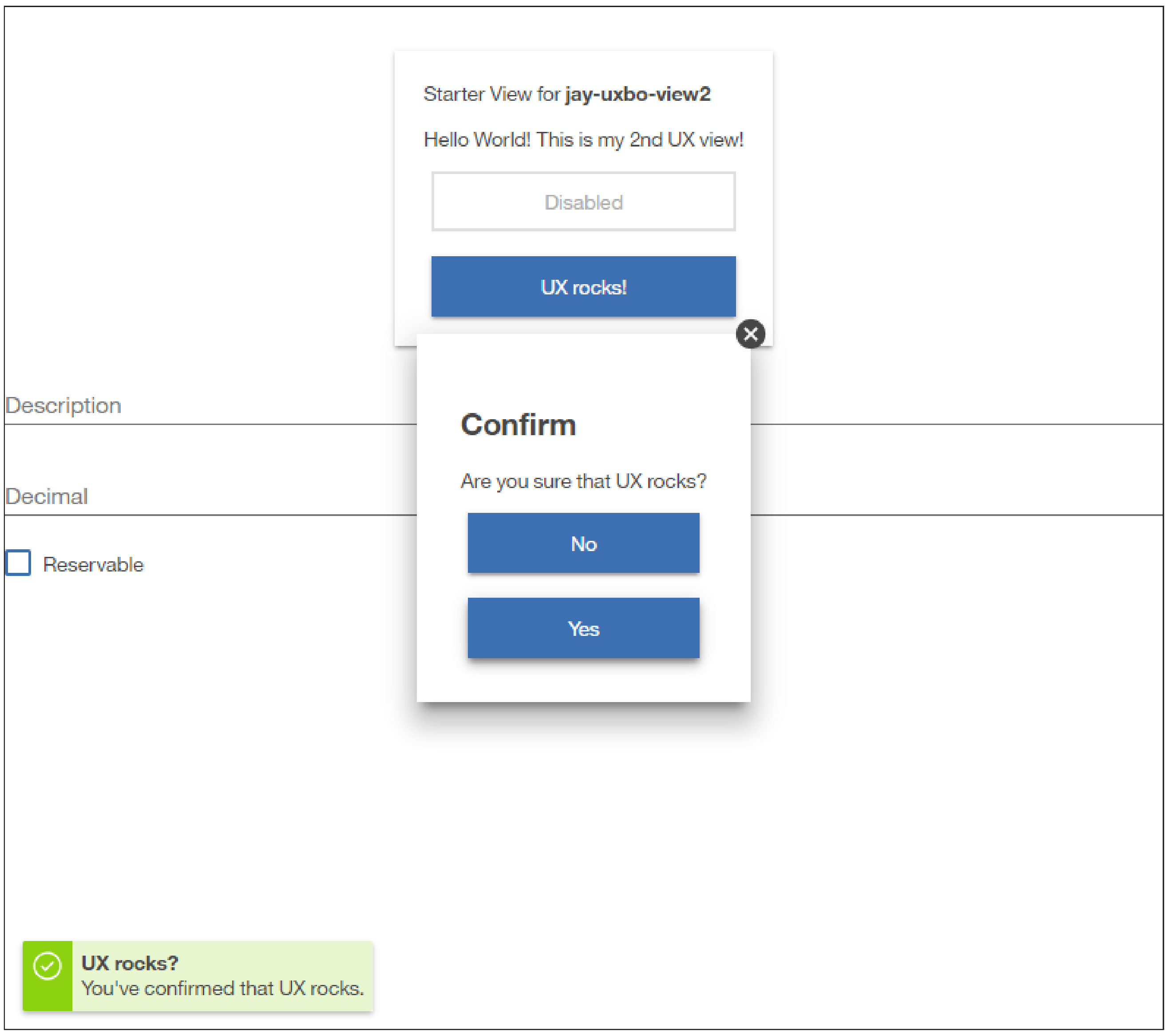
Hello World! This is my 2nd UX view!

Confirm

Are you sure that UX rocks?

UX rocks?
No, it doesn't. :(

UX App > Toast Popup with Success Style.



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

Hello World! This is my 2nd UX view!

Confirm

Are you sure that UX rocks?

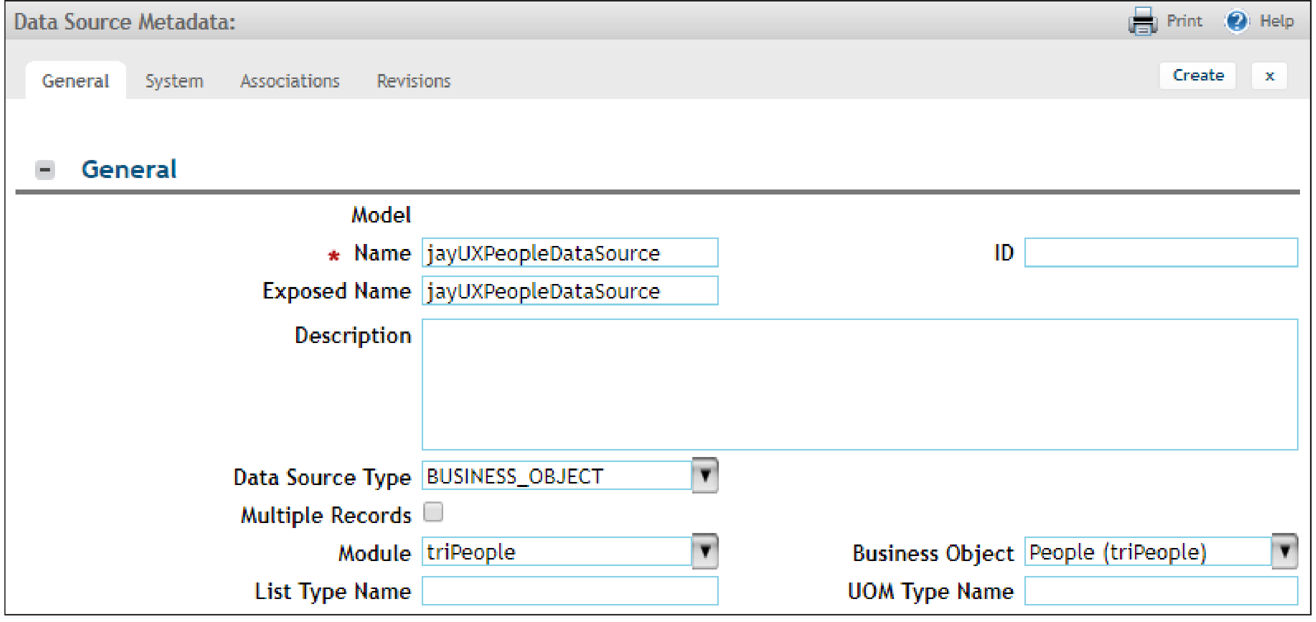
UX rocks?
You've confirmed that UX rocks.

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.

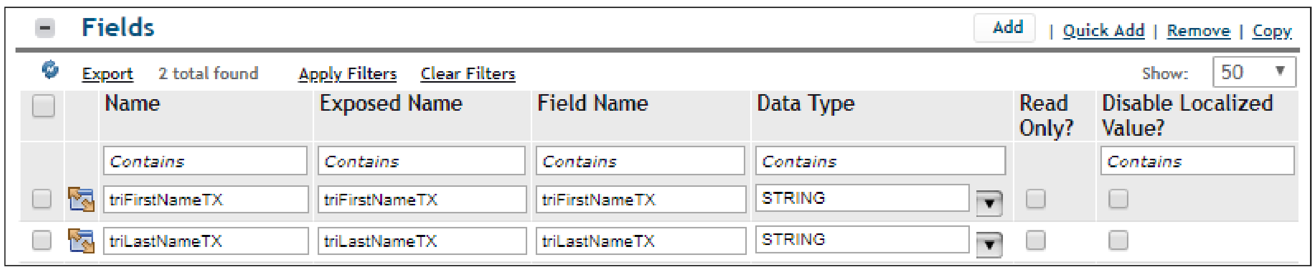


General

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**.

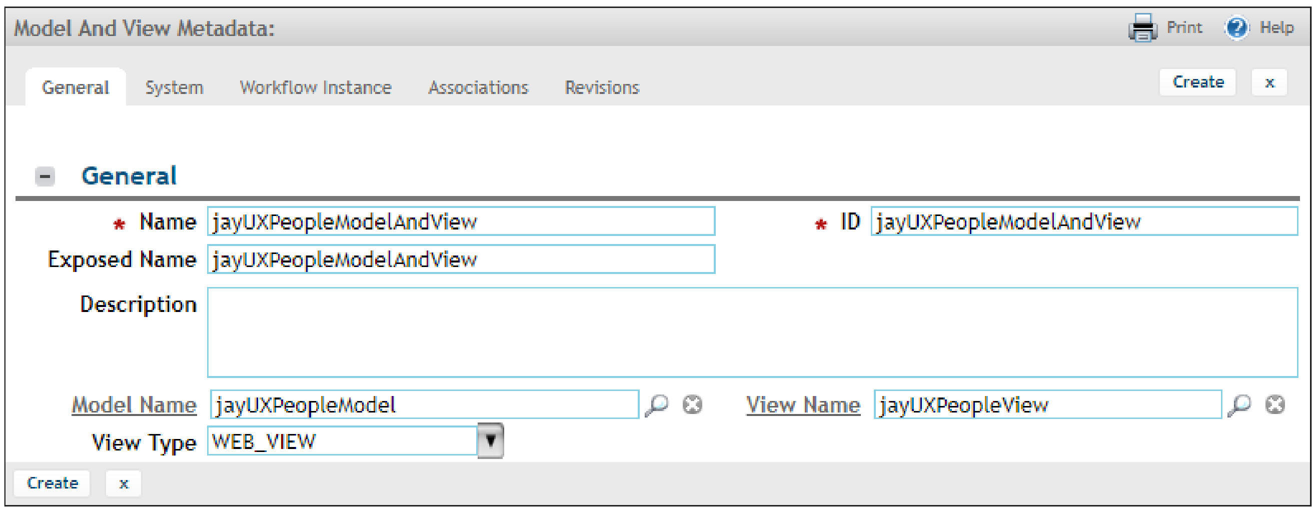
Data Source Fields.



□	Name	Exposed Name	Field Name	Data Type	Read Only?	Disable Localized Value?
	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.

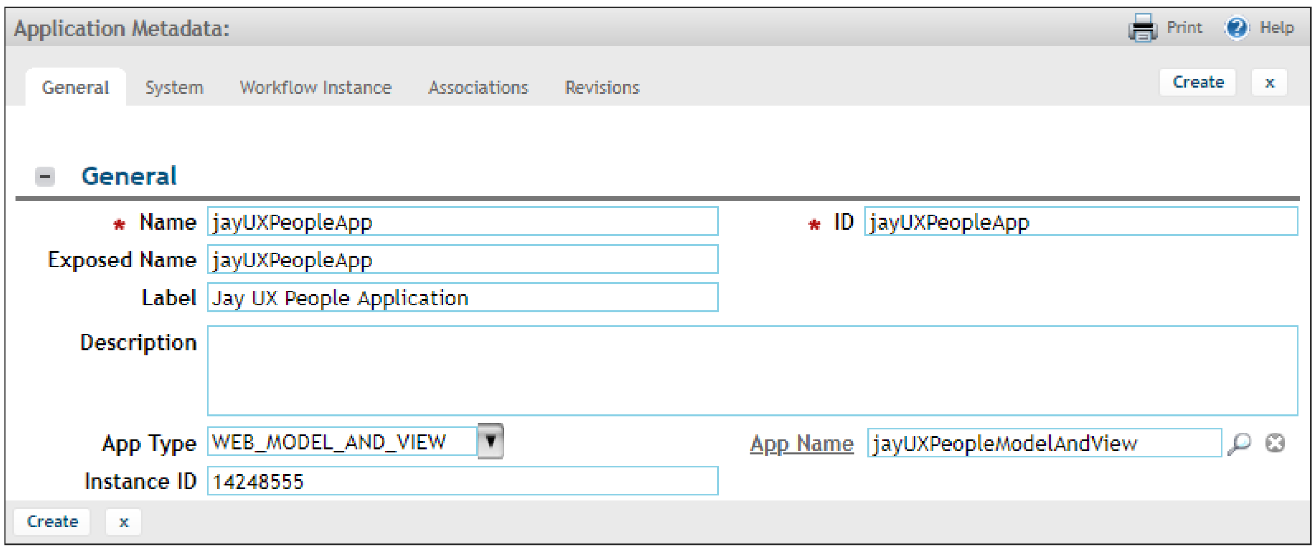


General

* 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=14248555** in its related URL. Enter your specific **specId** as the **Instance ID** for your application.

Application Metadata.

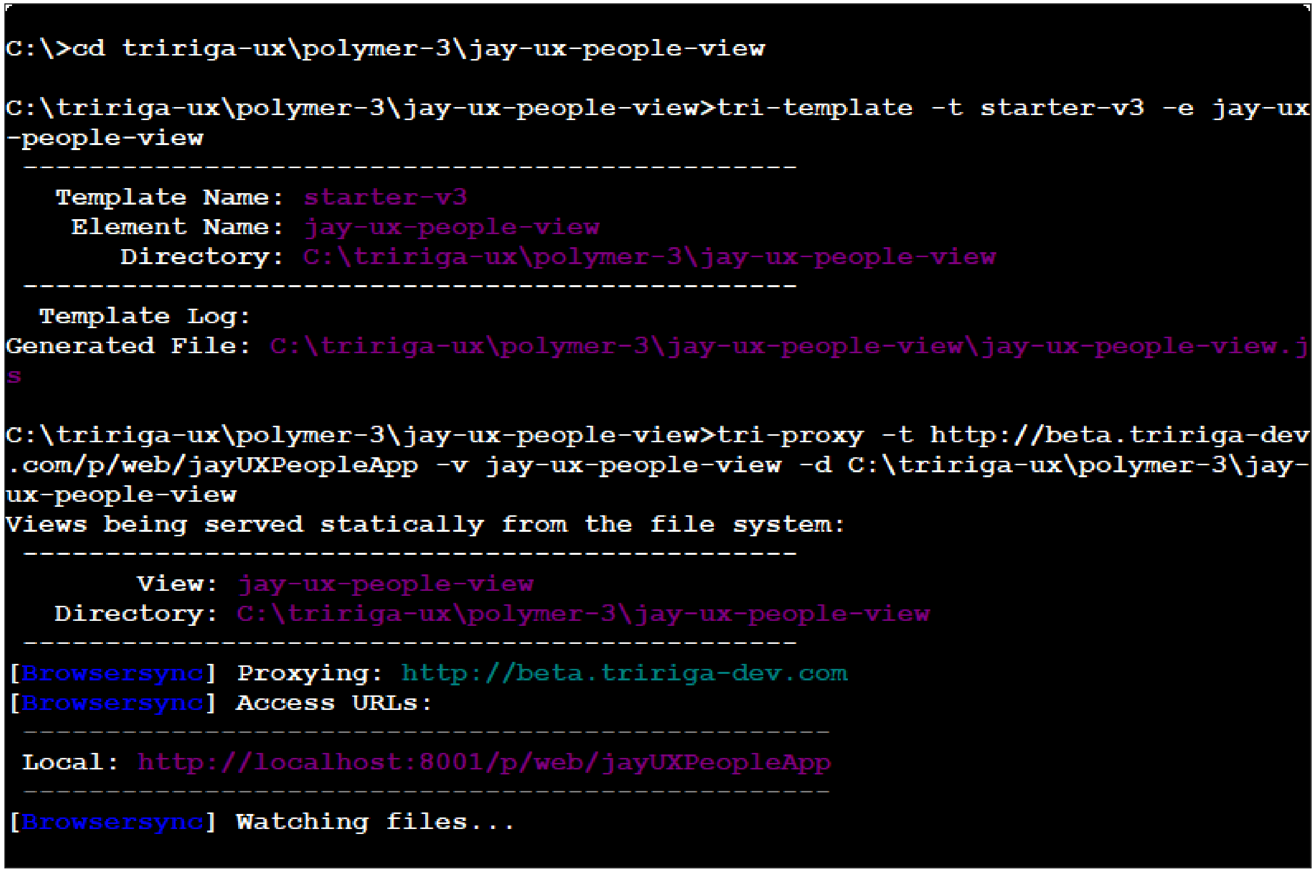


General

* Name	jayUXPeopleApp
* ID	jayUXPeopleApp
Exposed Name	jayUXPeopleApp
Label	Jay UX People Application
Description	
App Type	WEB_MODEL_AND_VIEW
App Name	jayUXPeopleModelAndView
Instance ID	14248555

Next, after you've prepared your model and view connections, open the command prompt in your selected folder, run the **tri-template** command if needed, and run the **tri-proxy** command to preview your changes. In our example, we'll run **tri-template** with **starter-v3** and **jay-ux-people-view** to add a new JS view file. Then we'll run **tri-proxy** with **http://beta.tririga-dev.com/p/web/jayUXPeopleApp**. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

NPM > tri-template and tri-proxy.



```
C:\>cd tririga-ux\polymer-3\jay-ux-people-view

C:\tririga-ux\polymer-3\jay-ux-people-view>tri-template -t starter-v3 -e jay-ux
-people-view
 -----------------------------------------------
   Template Name: starter-v3
    Element Name: jay-ux-people-view
       Directory: C:\tririga-ux\polymer-3\jay-ux-people-view
 -----------------------------------------------
  Template Log:
Generated File: C:\tririga-ux\polymer-3\jay-ux-people-view\jay-ux-people-view.j
s

C:\tririga-ux\polymer-3\jay-ux-people-view>tri-proxy -t http://beta.tririga-dev
.com/p/web/jayUXPeopleApp -v jay-ux-people-view -d C:\tririga-ux\polymer-3\jay-
ux-people-view
Views being served statically from the file system:
 -----------------------------------------------
        View: jay-ux-people-view
   Directory: C:\tririga-ux\polymer-3\jay-ux-people-view
 -----------------------------------------------
[Browsersync] Proxying: http://beta.tririga-dev.com
[Browsersync] Access URLs:
 -------------------------------------------------
 Local: http://localhost:8001/p/web/jayUXPeopleApp
 -------------------------------------------------
[Browsersync] Watching files...
```

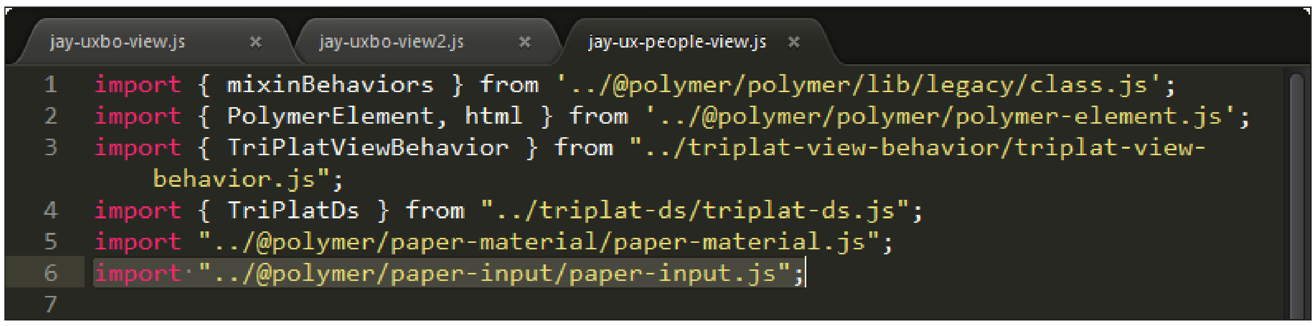
If needed, add the **import** line at the top of your JS file to import the TRIRIGA **triplat-ds** (data source) component and add the **<triplat-ds>** tag to declare it.

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 **import** line at the top to import the Polymer element: **import "../@polymer/paper-input/paper-input.js";**

JS File > Import paper-input.



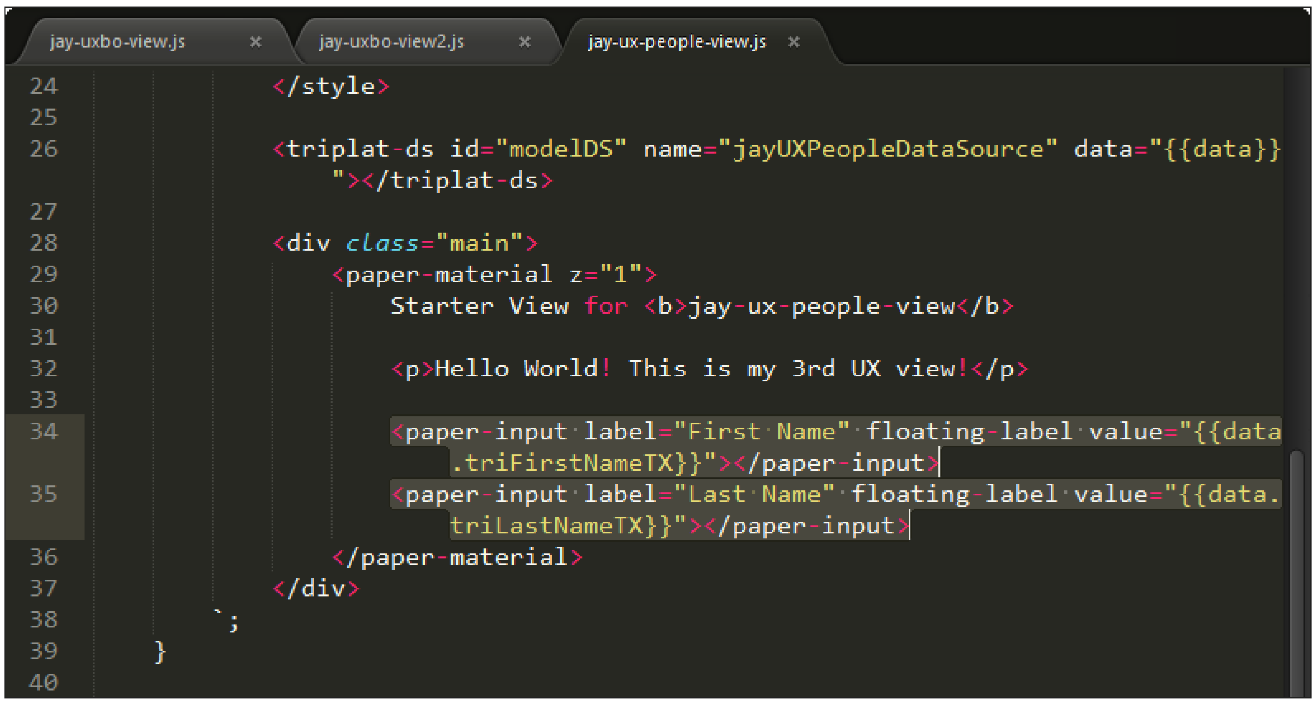
```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-input.js";
```

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

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

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

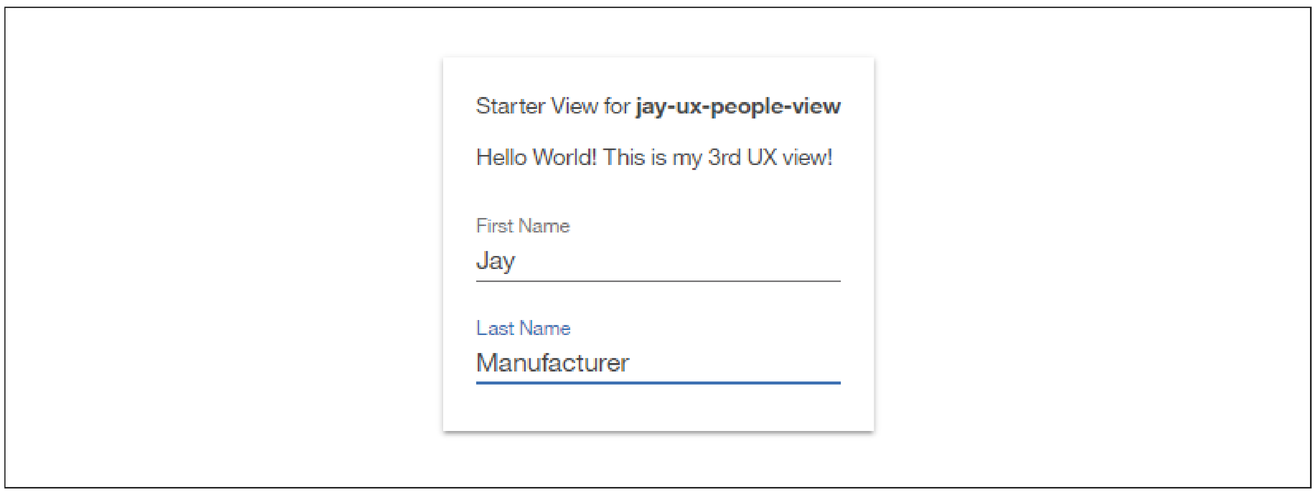
JS File > Declare paper-input.



```
            </style>

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

            <div class="main">
                <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>
        `;
    }

```

Save the file and return to 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

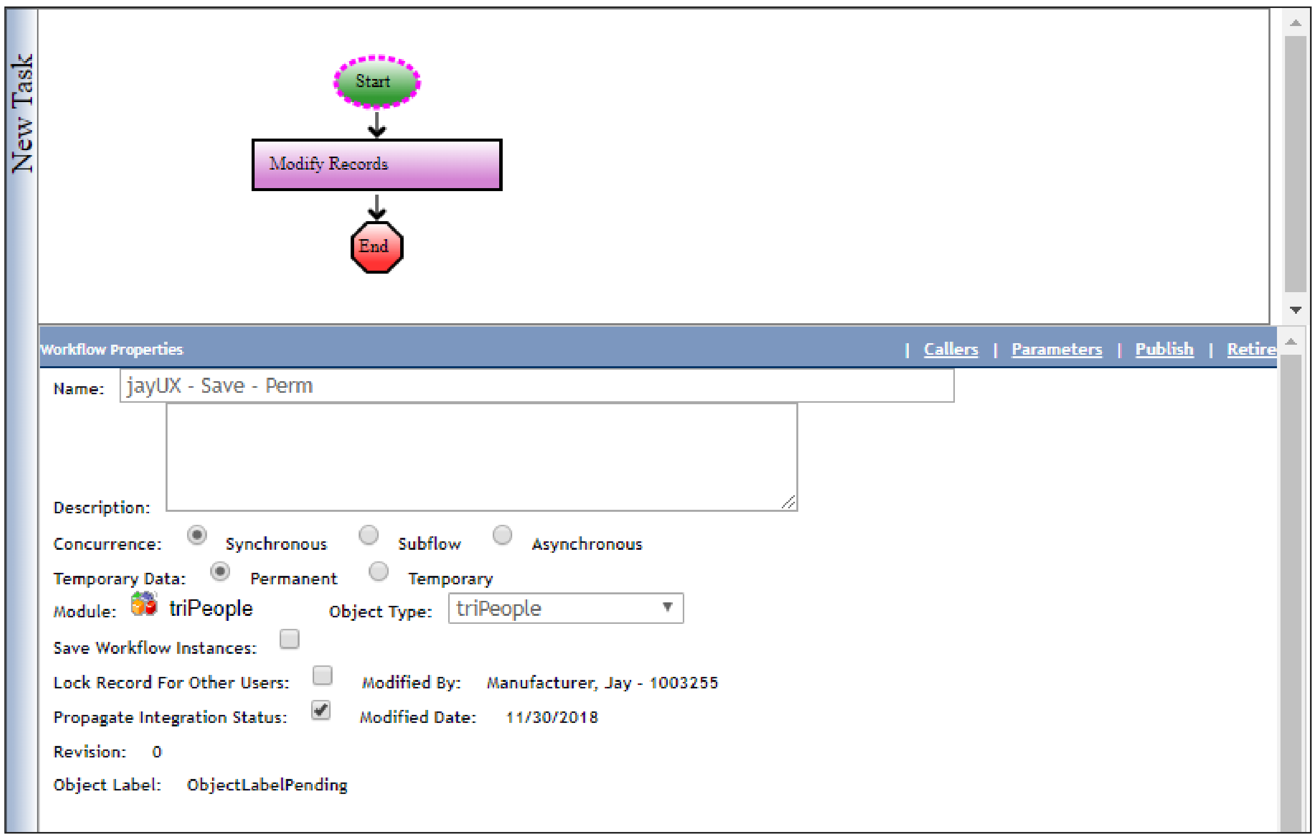
Last Name
Manufacturer

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: □ Modified By: Manufacturer, Jay - 1003255

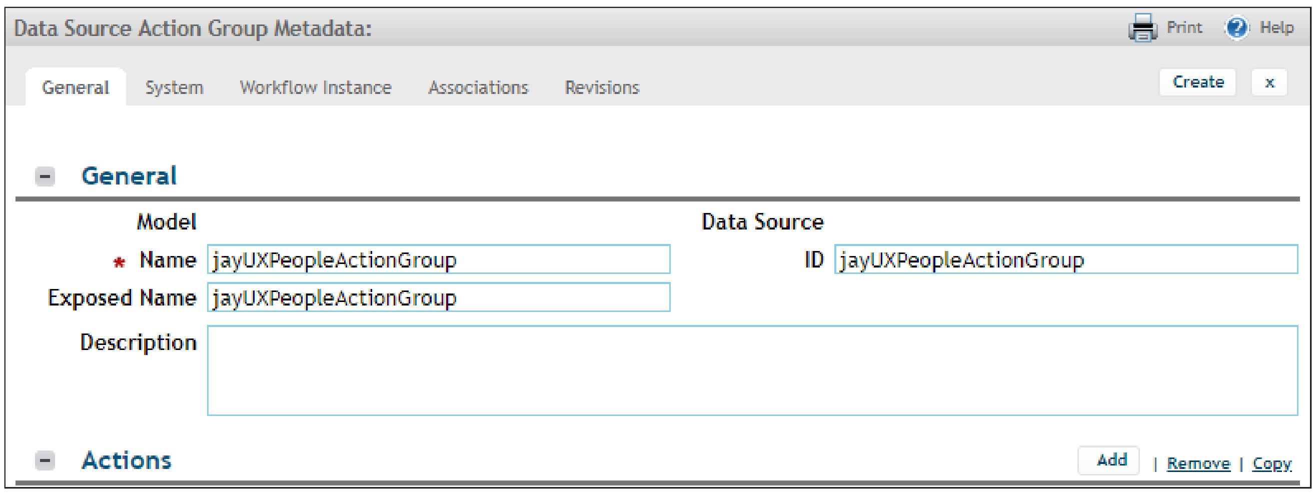
Propagate Integration Status: ☑ Modified Date: 11/30/2018

Revision: 0

Object Label: ObjectLabelPending

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.



Data Source Action Group Metadata:

General

Model

* Name: jayUXPeopleActionGroup

Exposed Name: jayUXPeopleActionGroup

Description:

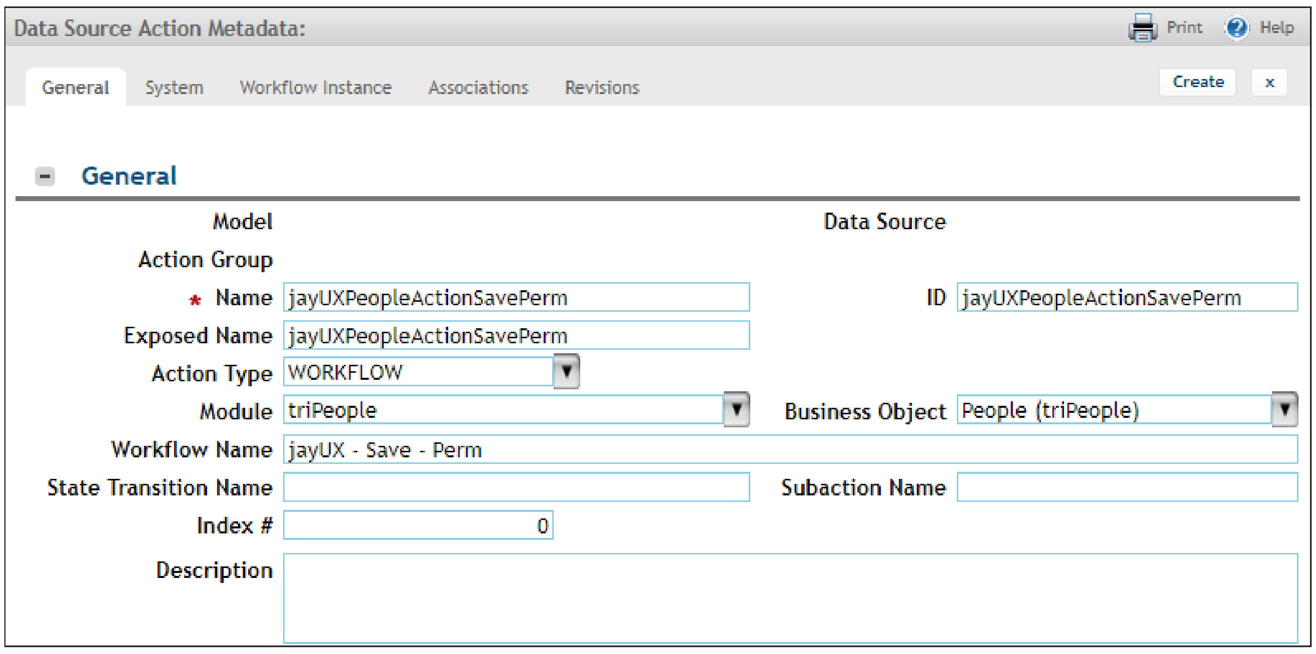
Data Source

ID: jayUXPeopleActionGroup

Actions

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.



Data Source Action Metadata:

General

Model

Action Group

* Name: jayUXPeopleActionSavePerm

Exposed Name: jayUXPeopleActionSavePerm

Action Type: WORKFLOW

Module: triPeople

Workflow Name: jayUX - Save - Perm

State Transition Name:

Index #: 0

Description:

Data Source

ID: jayUXPeopleActionSavePerm

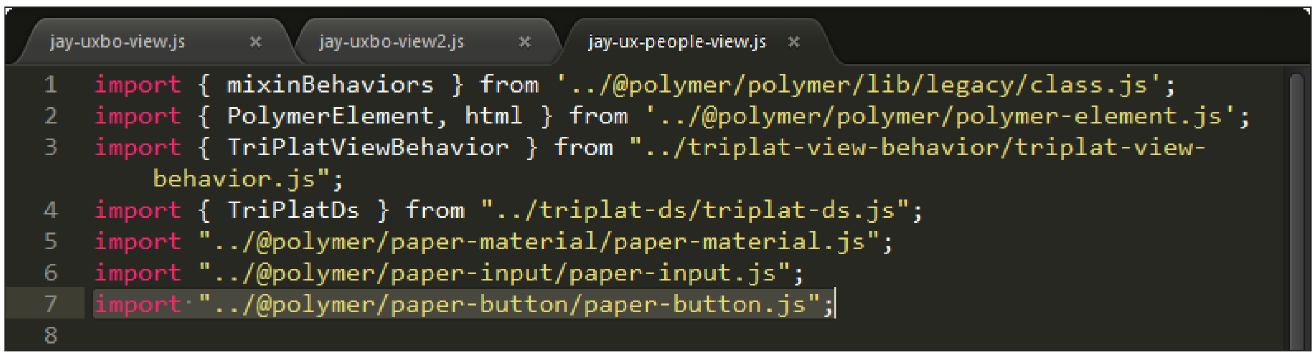
Business Object: People (triPeople)

Subaction Name:

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 **tri-proxy** command is running in the command prompt. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

First, add the **import** line at the top to import the Polymer element: **import "../@polymer/paper-button/paper-button.js";**

JS File > Import paper-button.

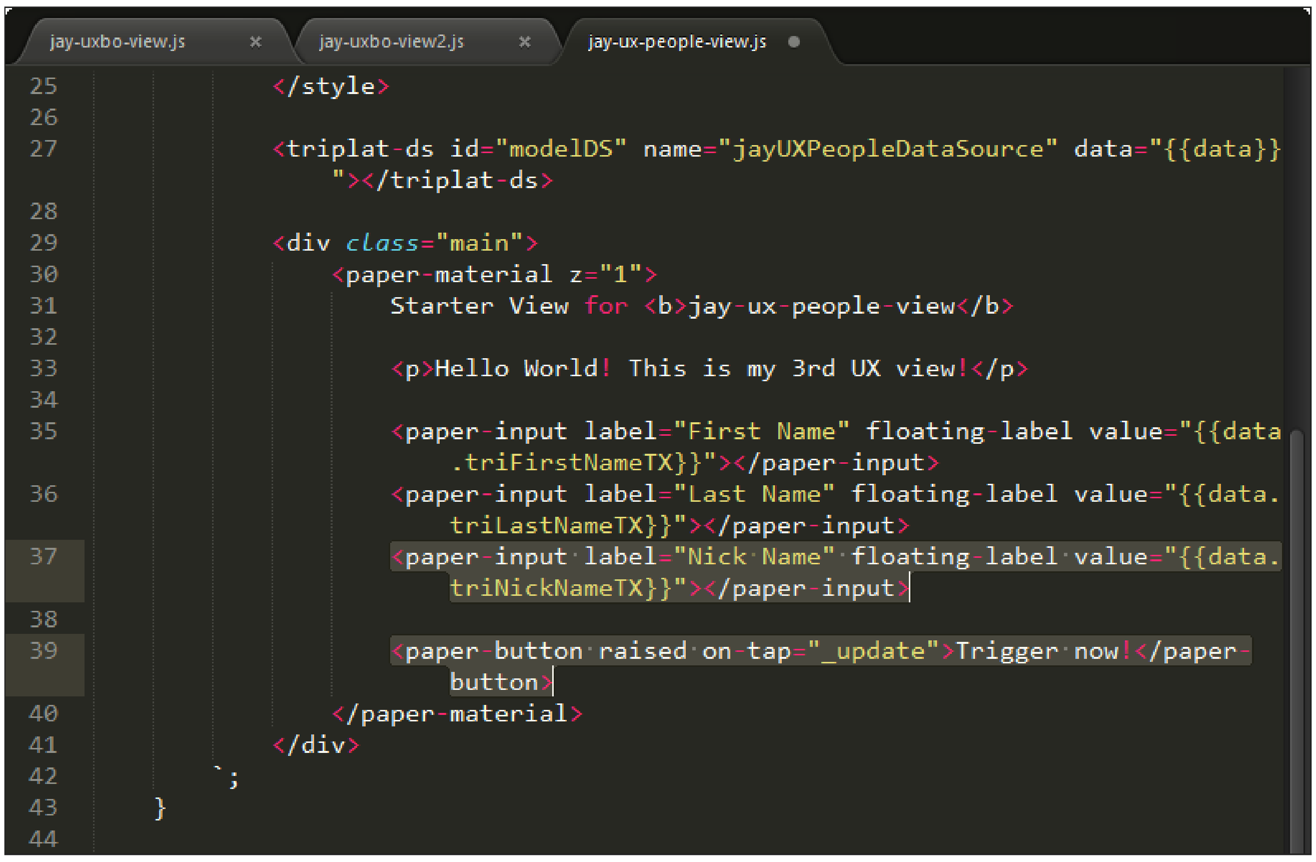


```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-input.js";
import "../@polymer/paper-button/paper-button.js";

```

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

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

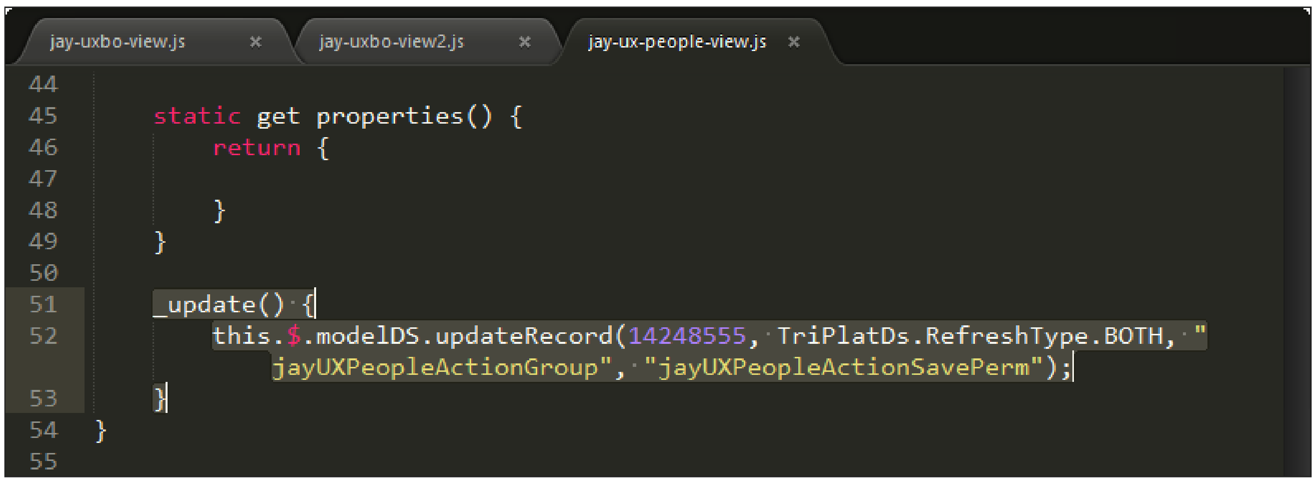
JS File > Declare paper-input and paper-button.



```
            </style>

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

            <div class="main">
                <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="_update">Trigger now!</paper-button>
                </paper-material>
            </div>
        `;
    }

```

Then, insert the JavaScript method **_update** at the bottom with your specific **specId**.

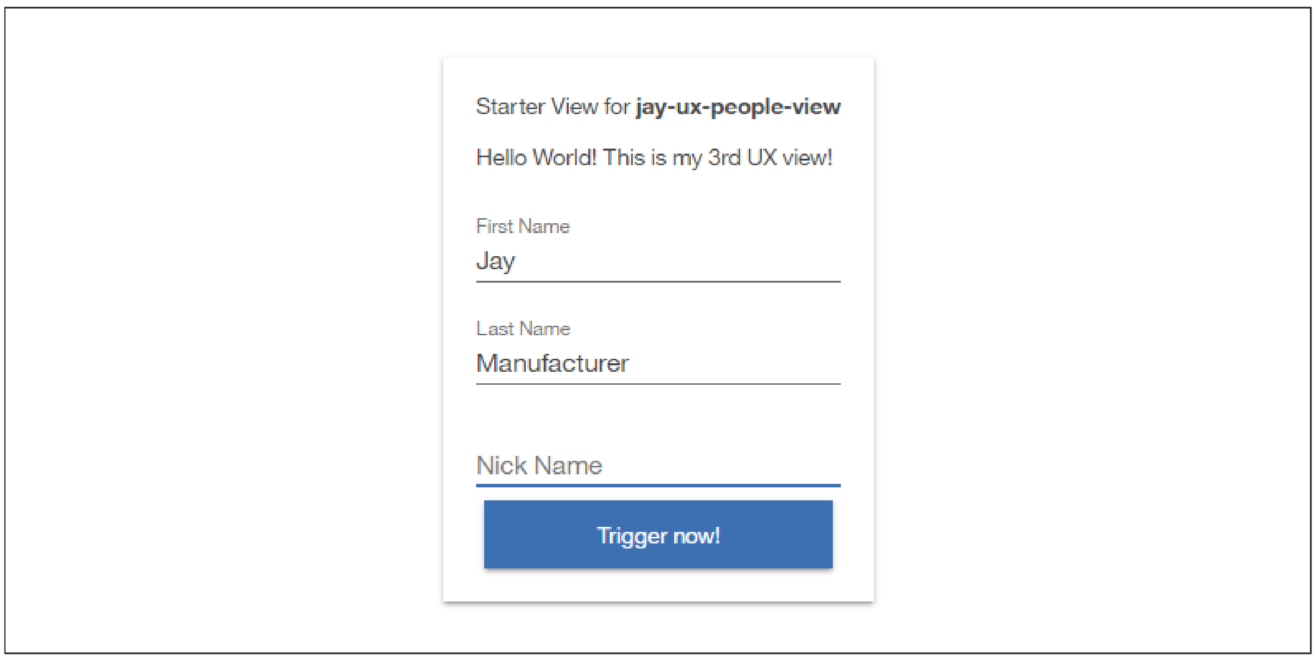
JS File > Insert _update.



```

    static get properties() {
        return {

        }
    }

    _update() {
        this.$.modelDS.updateRecord(14248555, TriPlatDs.RefreshType.BOTH, "jayUXPeopleActionGroup", "jayUXPeopleActionSavePerm");
    }
}

```

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

UX App > Input with No Nick Name.



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

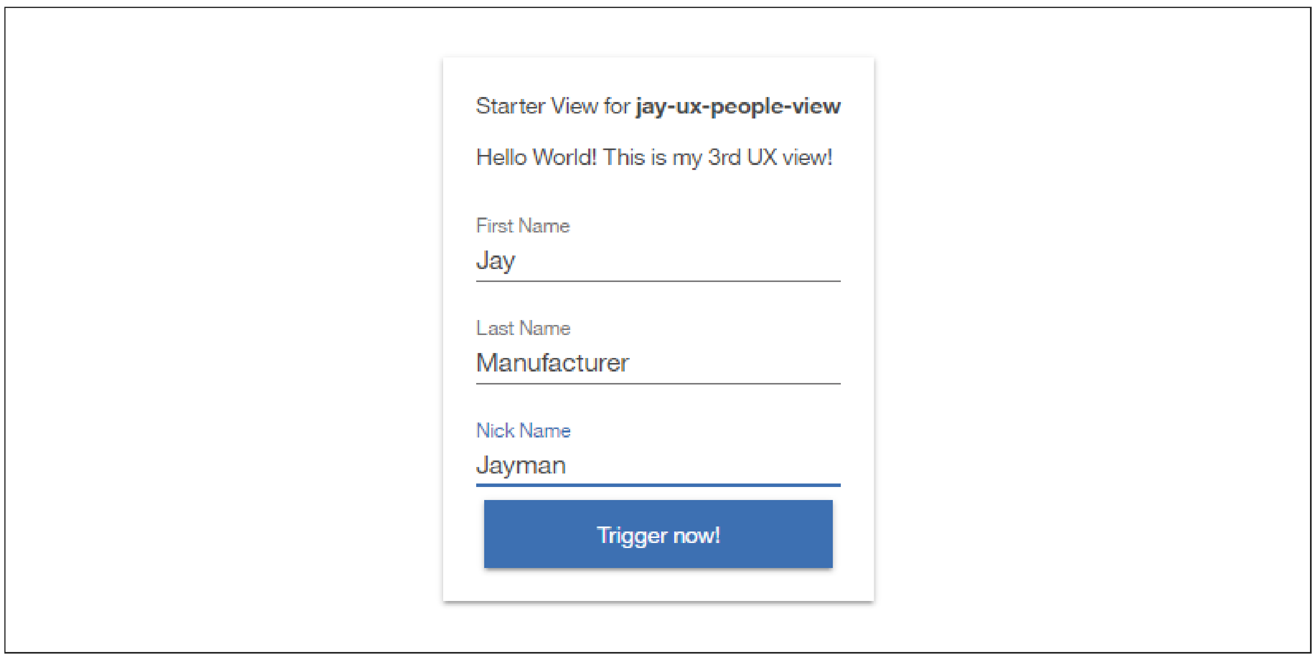
Hello World! This is my 3rd UX view!

First Name
Jay

Last Name
Manufacturer

Nick Name

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

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?

Start a card view as a custom component.

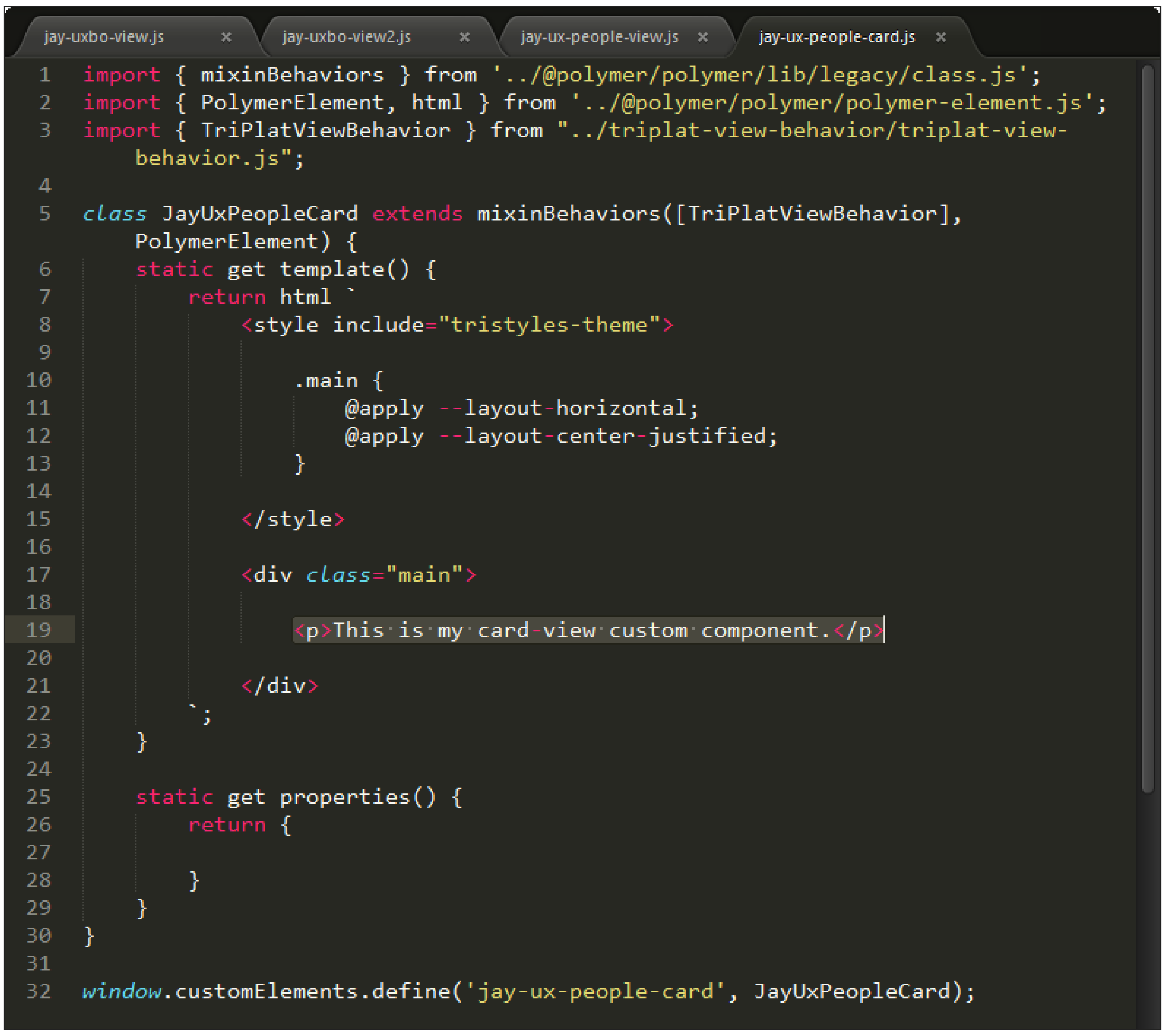
For our next exercise, we'll prepare a new JS file in the same folder as your existing JS file. ***But don't move it into the same folder yet!*** First, edit the new JS file in a different folder to begin with the simplest running design. This new JS file will start your custom component.

In our example, we'll name the new JS file **jay-ux-people-card.js** for our card-view custom component, and edit its JS to look like the following code, very similar to that of a regular starter view.

JS File > Custom Component.

Notes:

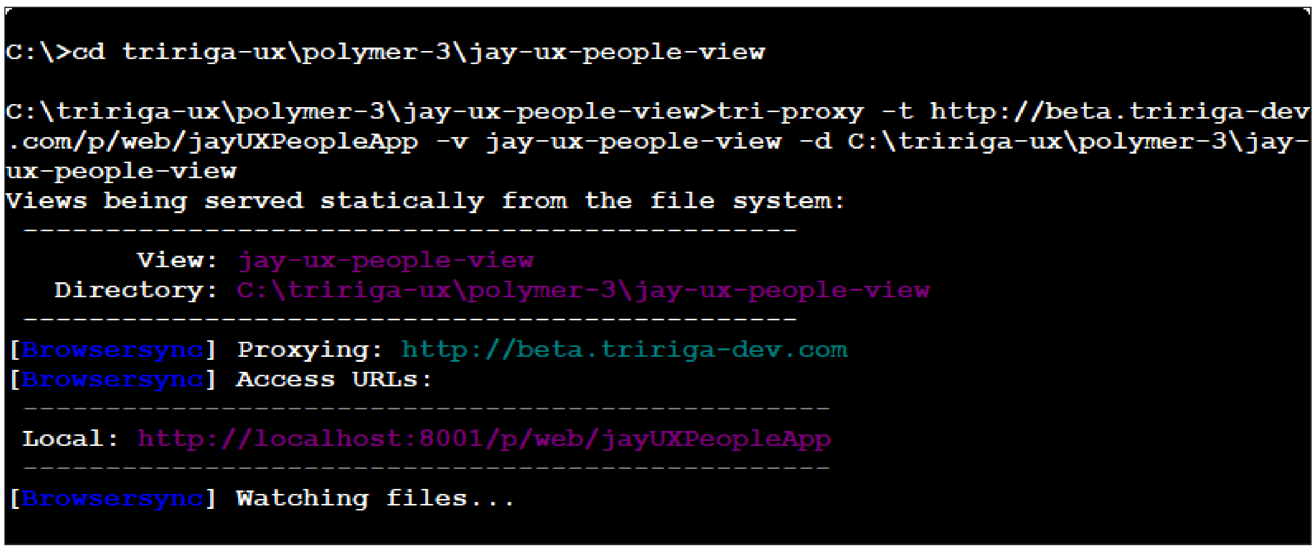
- For custom components, the **mixinBehaviors import** (line 1) and the **TriPlatViewBehavior import** (line 3) are not required, but are shown for demonstration purposes. A future release might make this situation more strict. In such a case, the **JayUxPeopleCard class** (line 5) can be simplified to: **class JayUxPeopleCard extends PolymerElement {**



```
1   import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
2   import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
3   import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-
        behavior.js";
4
5   class JayUxPeopleCard extends mixinBehaviors([TriPlatViewBehavior],
        PolymerElement) {
6       static get template() {
7           return html `
8               <style include="tristyles-theme">
9
10                  .main {
11                      @apply --layout-horizontal;
12                      @apply --layout-center-justified;
13                  }
14
15              </style>
16
17              <div class="main">
18
19                  <p>This is my card-view custom component.</p>
20
21              </div>
22          `;
23      }
24
25      static get properties() {
26          return {
27
28          }
29      }
30  }
31
32  window.customElements.define('jay-ux-people-card', JayUxPeopleCard);
```

Next, after you've prepared your JS file, open the command prompt in your selected folder, and run the **tri-proxy** command to preview your changes. In our example, we'll go to the same previous **jay-ux-people-view** folder. Then we'll run **tri-proxy** with the same previous **http://beta.tririga-dev.com/p/web/jayUXPeopleApp**. (At any time, you can also run **tri-deploy** to push your updated view file to the server.)

NPM > tri-proxy > Preview View.

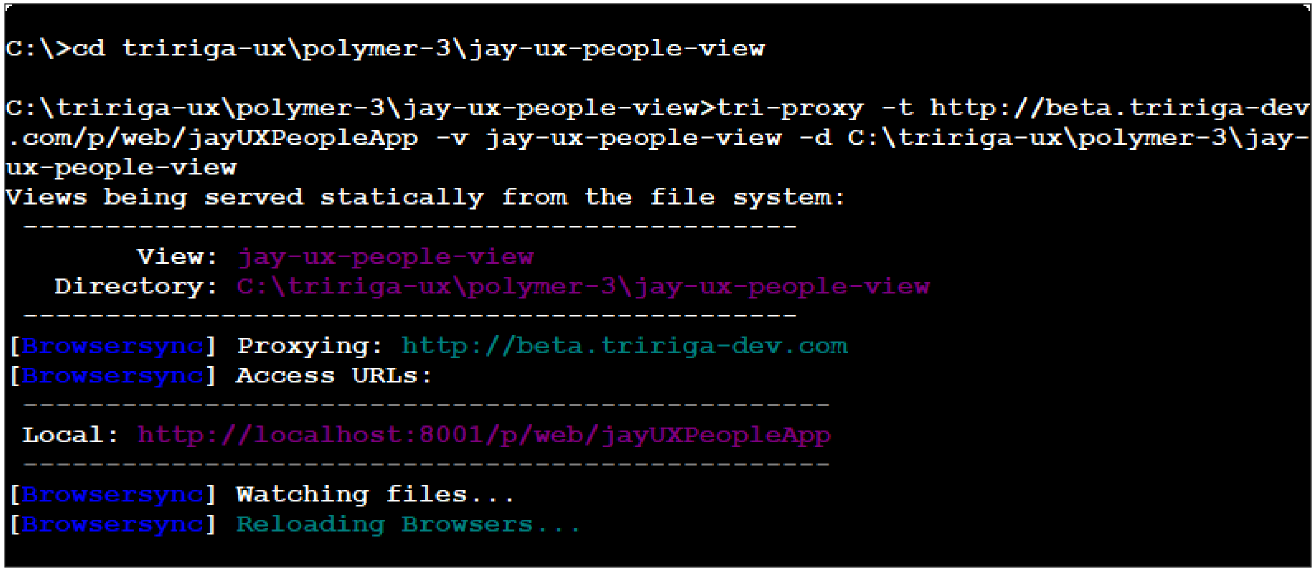


```
C:\>cd tririga-ux\polymer-3\jay-ux-people-view

C:\tririga-ux\polymer-3\jay-ux-people-view>tri-proxy -t http://beta.tririga-dev
.com/p/web/jayUXPeopleApp -v jay-ux-people-view -d C:\tririga-ux\polymer-3\jay-
ux-people-view
Views being served statically from the file system:
 ----------------------------------------------------
       View: jay-ux-people-view
  Directory: C:\tririga-ux\polymer-3\jay-ux-people-view
 ----------------------------------------------------
[Browsersync] Proxying: http://beta.tririga-dev.com
[Browsersync] Access URLs:
 ------------------------------------------------------
 Local: http://localhost:8001/p/web/jayUXPeopleApp
 ------------------------------------------------------
[Browsersync] Watching files...
```

This time, you're ready to move your new JS 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\polymer-3** folder contains the **jay-ux-people-view** folder, which contains the existing **jay-ux-people-view.js** file from the last exercise. Go ahead and move the new JS file **jay-ux-people-card.js** beside the existing JS file in the same folder. The next time you save the moved JS file, notice how the **tri-proxy** tool detects it and reloads your preview.

NPM > tri-proxy > Reload View.



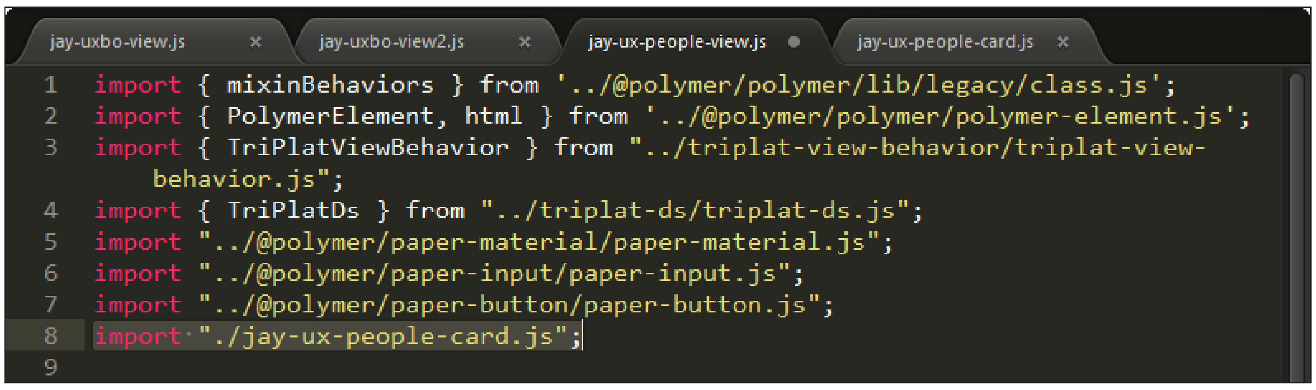
```
C:\>cd tririga-ux\polymer-3\jay-ux-people-view

C:\tririga-ux\polymer-3\jay-ux-people-view>tri-proxy -t http://beta.tririga-dev
.com/p/web/jayUXPeopleApp -v jay-ux-people-view -d C:\tririga-ux\polymer-3\jay-
ux-people-view
Views being served statically from the file system:
 -----------------------------------------------
        View: jay-ux-people-view
   Directory: C:\tririga-ux\polymer-3\jay-ux-people-view
 -----------------------------------------------
[Browsersync] Proxying: http://beta.tririga-dev.com
[Browsersync] Access URLs:
 -------------------------------------------------
 Local: http://localhost:8001/p/web/jayUXPeopleApp
 -------------------------------------------------
[Browsersync] Watching files...
[Browsersync] Reloading Browsers...
```

Add a card view component to your JS file.

This time, we'll add a custom tag to our existing JS 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.js** file.

First, add the **import** line at the top to import the custom component: **import "./jay-ux-people-card.js";**

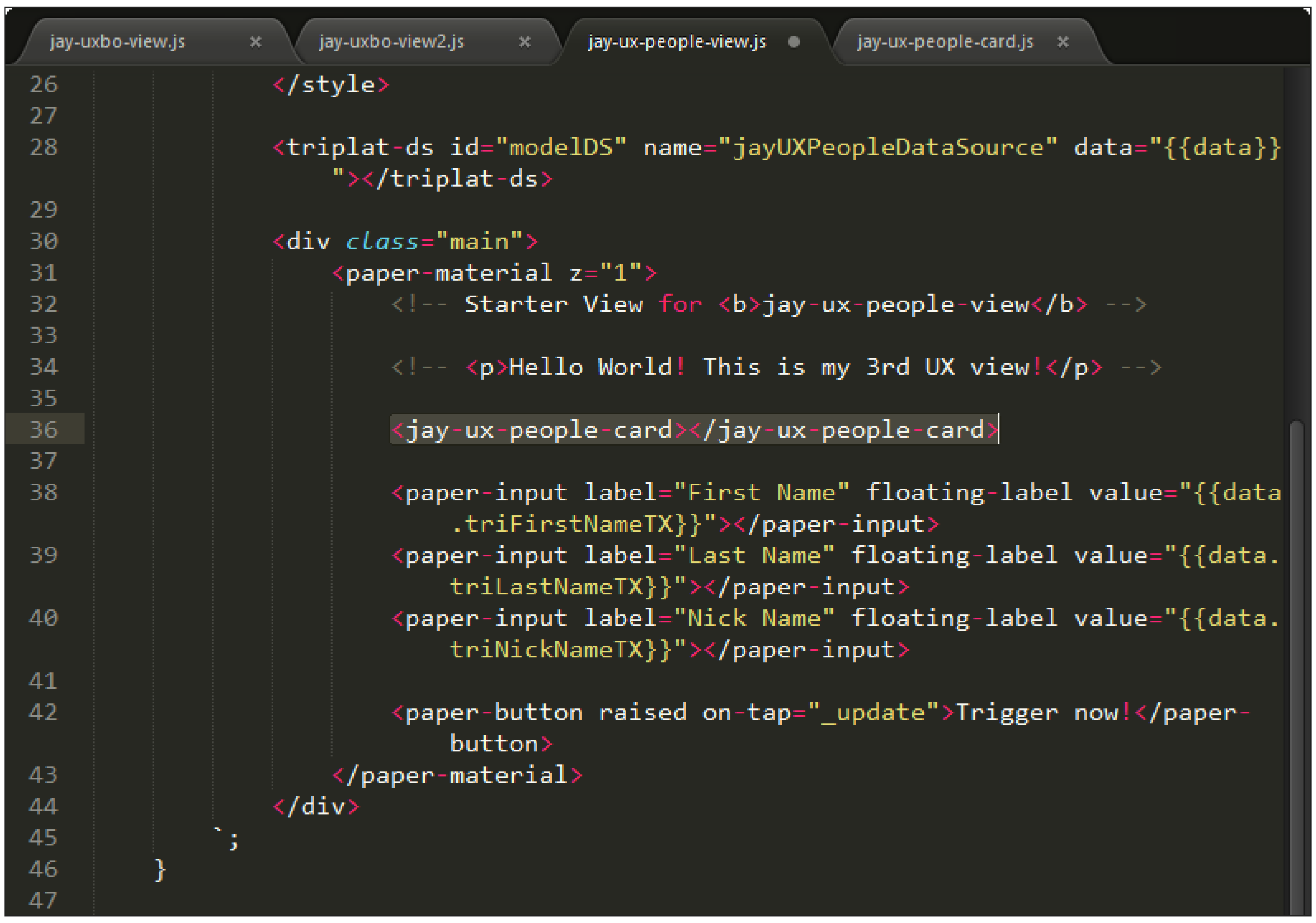
JS File > Import Custom Component.



```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import { TriPlatDs } from "../triplat-ds/triplat-ds.js";
import "../@polymer/paper-material/paper-material.js";
import "../@polymer/paper-input/paper-input.js";
import "../@polymer/paper-button/paper-button.js";
import "./jay-ux-people-card.js";

```

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

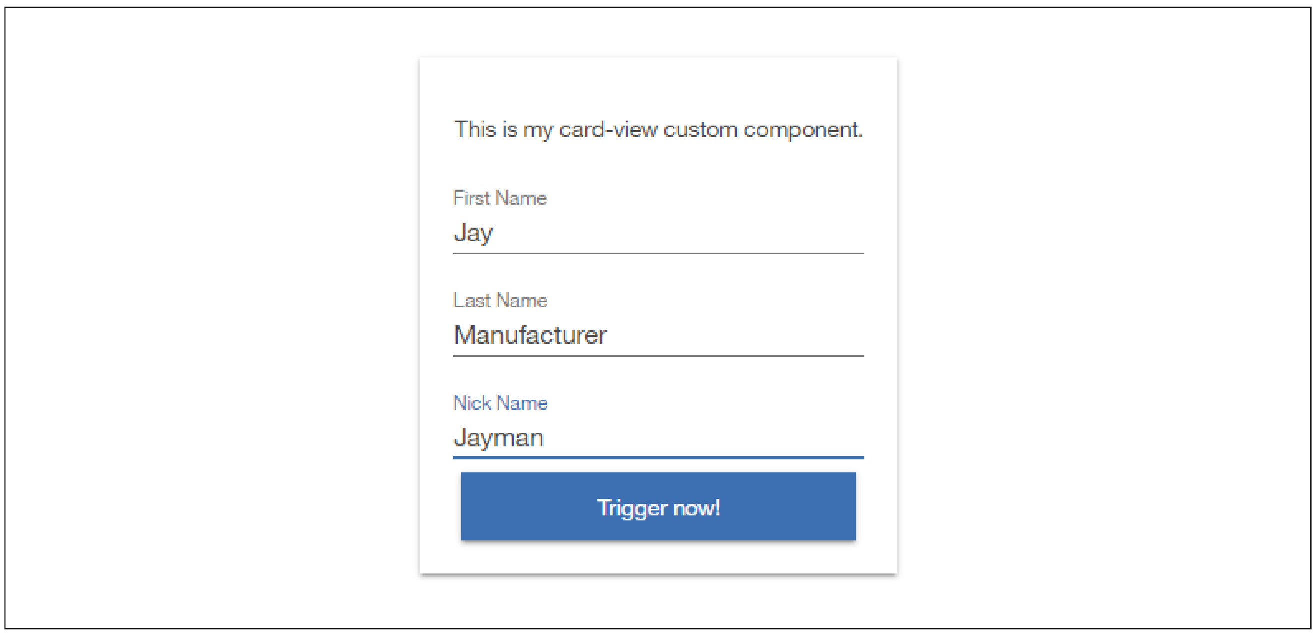
JS File > Declare Custom Component.



```
            </style>

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

            <div class="main">
                <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="_update">Trigger now!</paper-button>
                </paper-material>
            </div>
        `;
    }

```

Save the file and return to 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

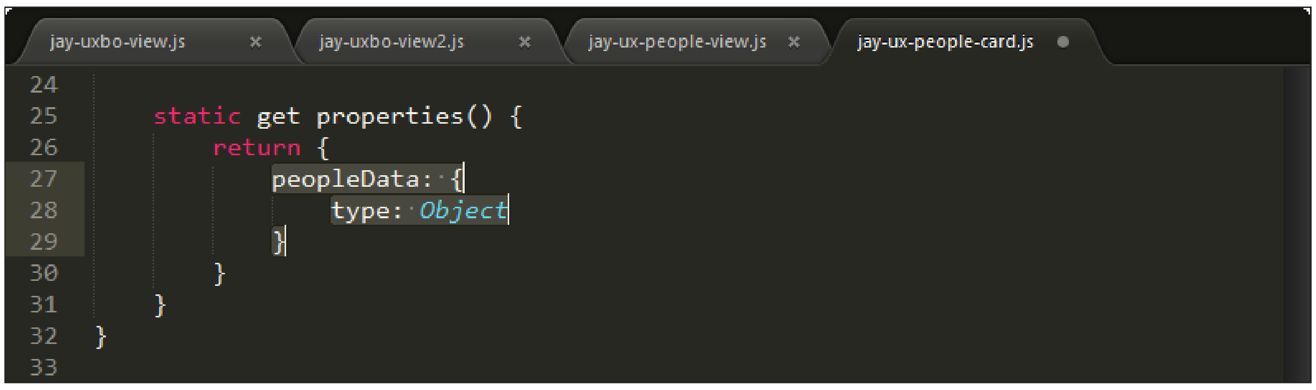
Nick Name
Jayman

Expand your card view component.

This time, we'll add a simple **peopleData** property to our component **jay-ux-people-card.js** 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 **static** function.

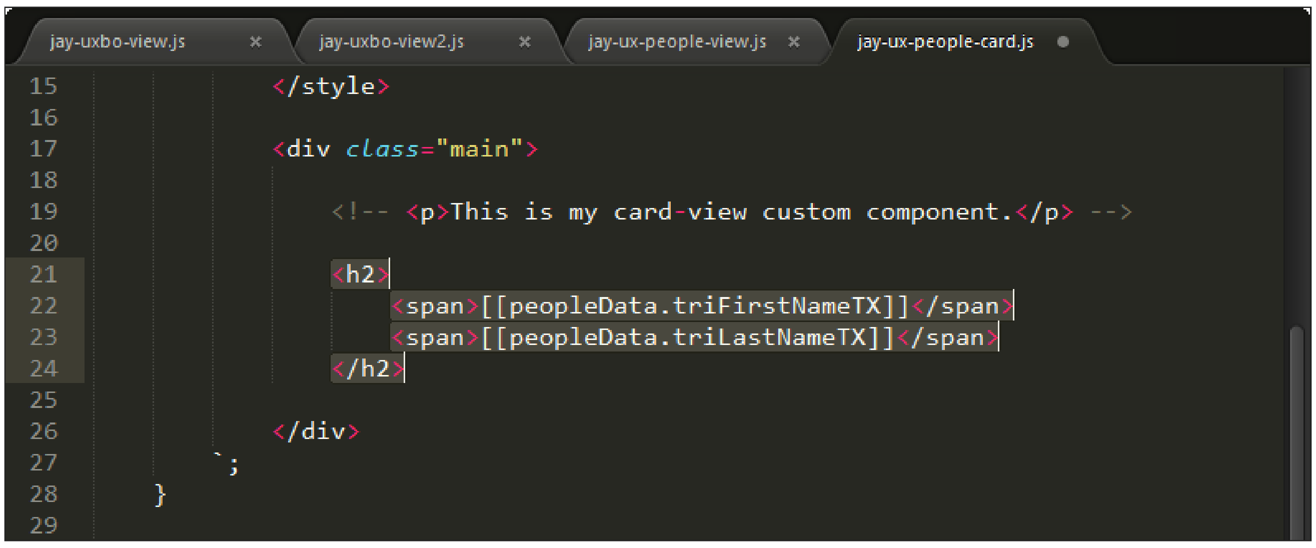
JS File > Insert peopleData.



```
    static get properties() {
        return {
            peopleData: {
                type: Object
            }
        }
    }
}
```

Next, replace the placeholder text with **peopleData** values.

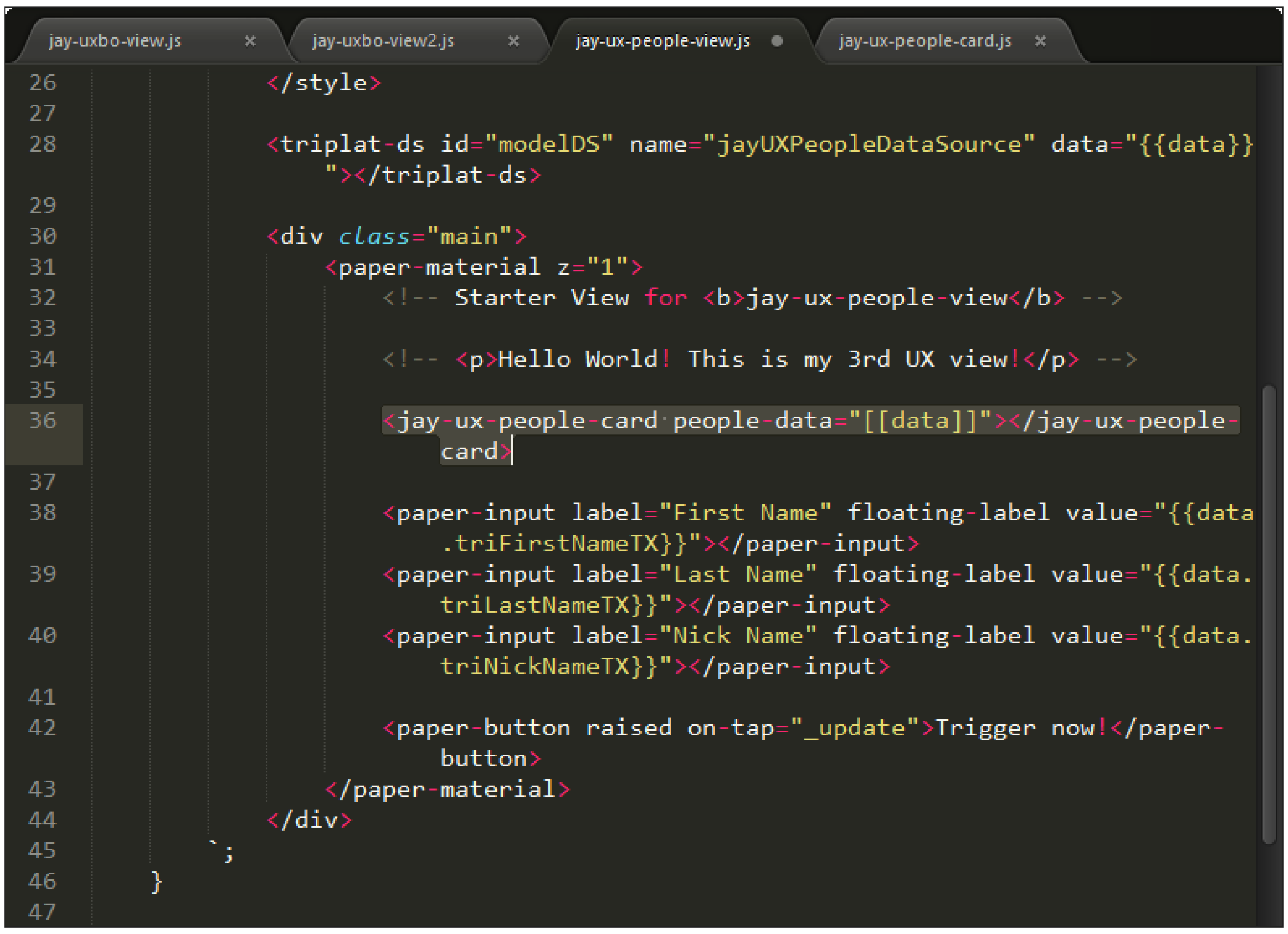
JS File > Add peopleData Values.



```
            </style>

            <div class="main">

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

                <h2>
                    <span>[[peopleData.triFirstNameTX]]</span>
                    <span>[[peopleData.triLastNameTX]]</span>
                </h2>

            </div>
        `;
    }

```

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.js** 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.

JS File > Add people-data Attribute.

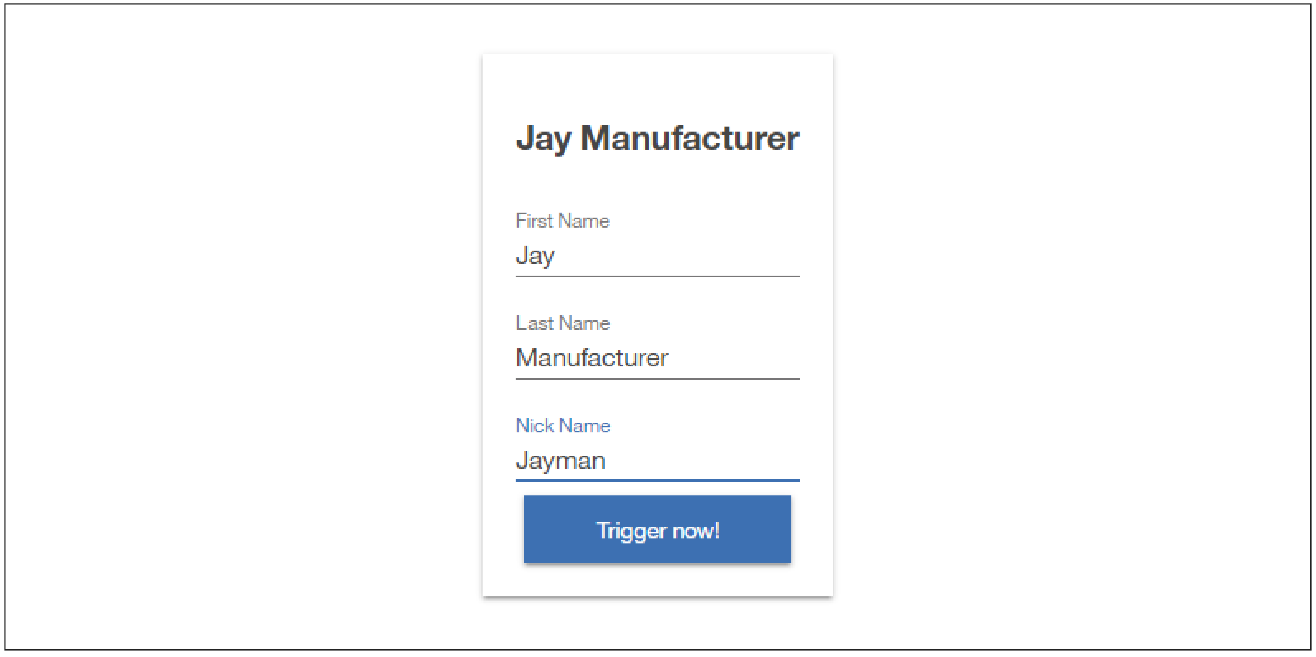


```
            </style>

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

            <div class="main">
                <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="_update">Trigger now!</paper-button>
                </paper-material>
            </div>
        `;
    }

```

Why do we need a dash in the 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 return to the UX view. Do you see your expanded custom component? Now it's time to display a profile image!

UX App > Expanded Custom Component.



Jay Manufacturer

First Name
Jay

Last Name
Manufacturer

Nick Name
Jayman

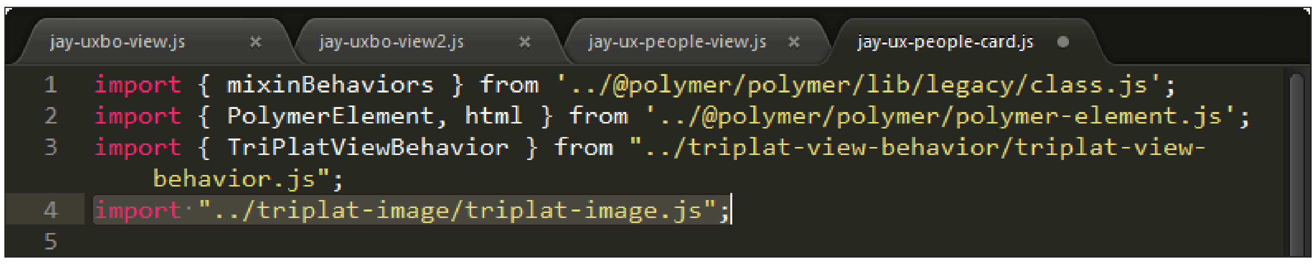
Display image data from an existing record.

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

This time, we'll add the TRIRIGA **<triplat-image>** tag to our component **jay-ux-people-card.js** to grab the instance data **triImageIM** from an existing people record, the same record from previous exercises.

First, add the **import** line at the top to import the TRIRIGA element: **import "../triplat-image/triplat-image.js";**

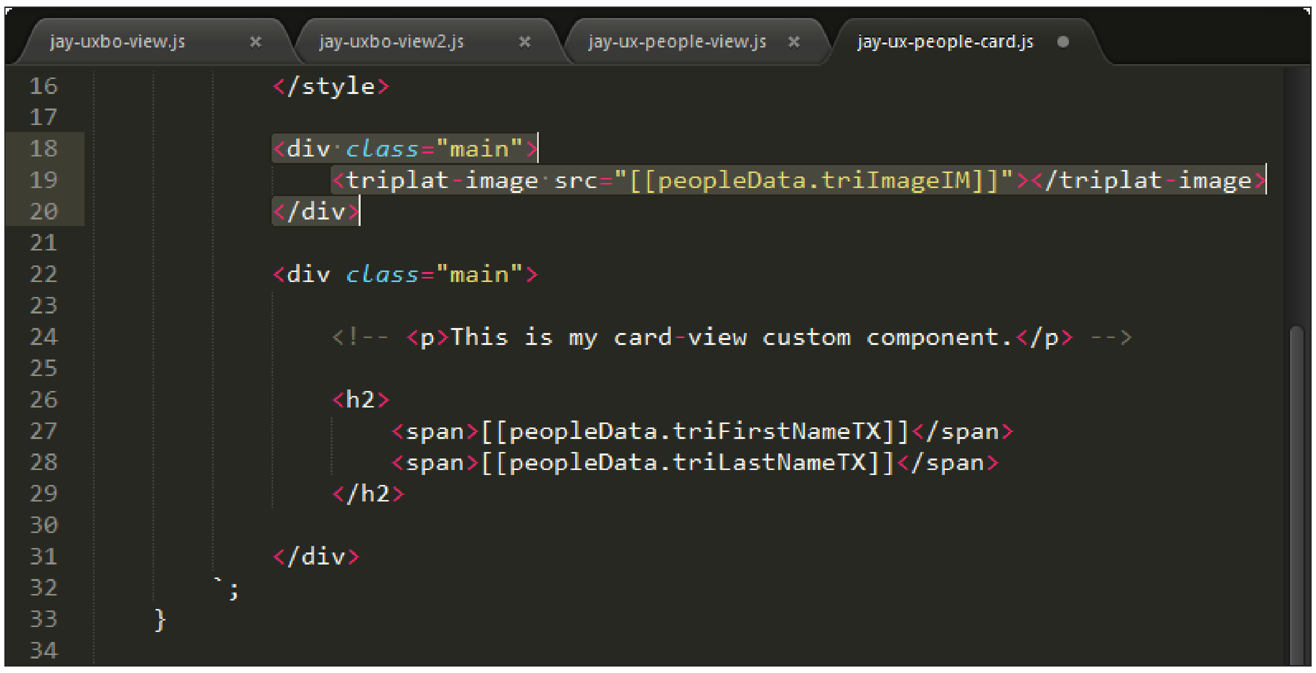
JS File > Import triplat-image.



```
import { mixinBehaviors } from '../@polymer/polymer/lib/legacy/class.js';
import { PolymerElement, html } from '../@polymer/polymer/polymer-element.js';
import { TriPlatViewBehavior } from "../triplat-view-behavior/triplat-view-behavior.js";
import "../triplat-image/triplat-image.js";

```

Next, add the **<div>** tag if needed, and add the **<triplat-image>** tag to declare the TRIRIGA element: **<triplat-image src="[[peopleData.triImageIM]]"></triplat-image>**

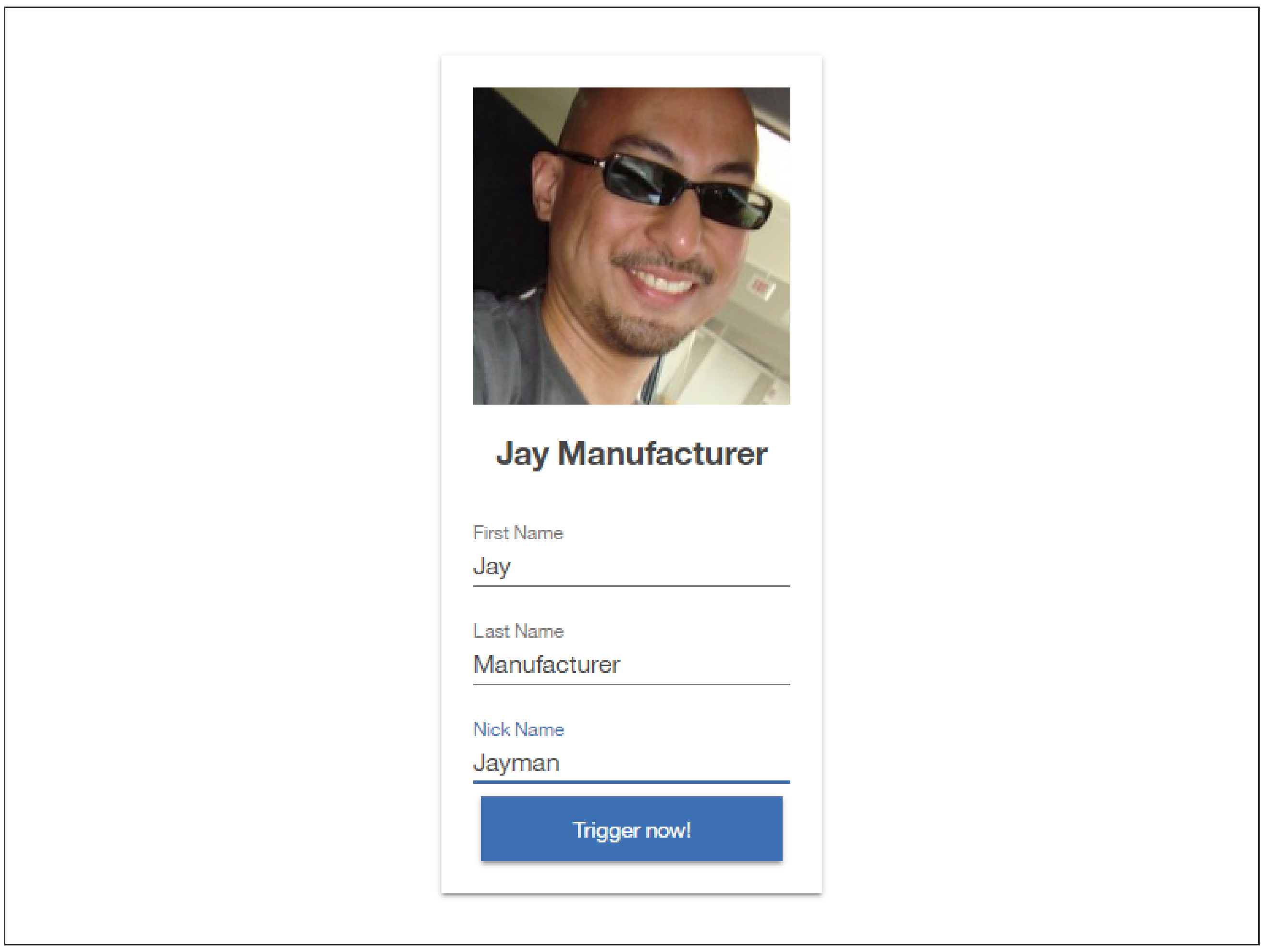
JS File > Declare triplat-image.



```
            </style>

            <div class="main">
                <triplat-image src="[[peopleData.triImageIM]]"></triplat-image>
            </div>

            <div class="main">

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

                <h2>
                    <span>[[peopleData.triFirstNameTX]]</span>
                    <span>[[peopleData.triLastNameTX]]</span>
                </h2>

            </div>
        `;
    }

```

Save the file and return to 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.



Jay Manufacturer

First Name
Jay

Last Name
Manufacturer

Nick Name
Jayman

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. Or if you want, I'll go ask the team.

Next >

Comments (0) | Versions (3) | Attachments (52) | About

There are no comments.

Add a comment

Feed for this page | Feed for these comments