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 (Poly...
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 > Implementing UX (Polymer 3)

Implementing UX (Polymer 3)

Like | Updated June 12, 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 "Implementing UX" wiki page for previous Polymer 1 versions of this content.

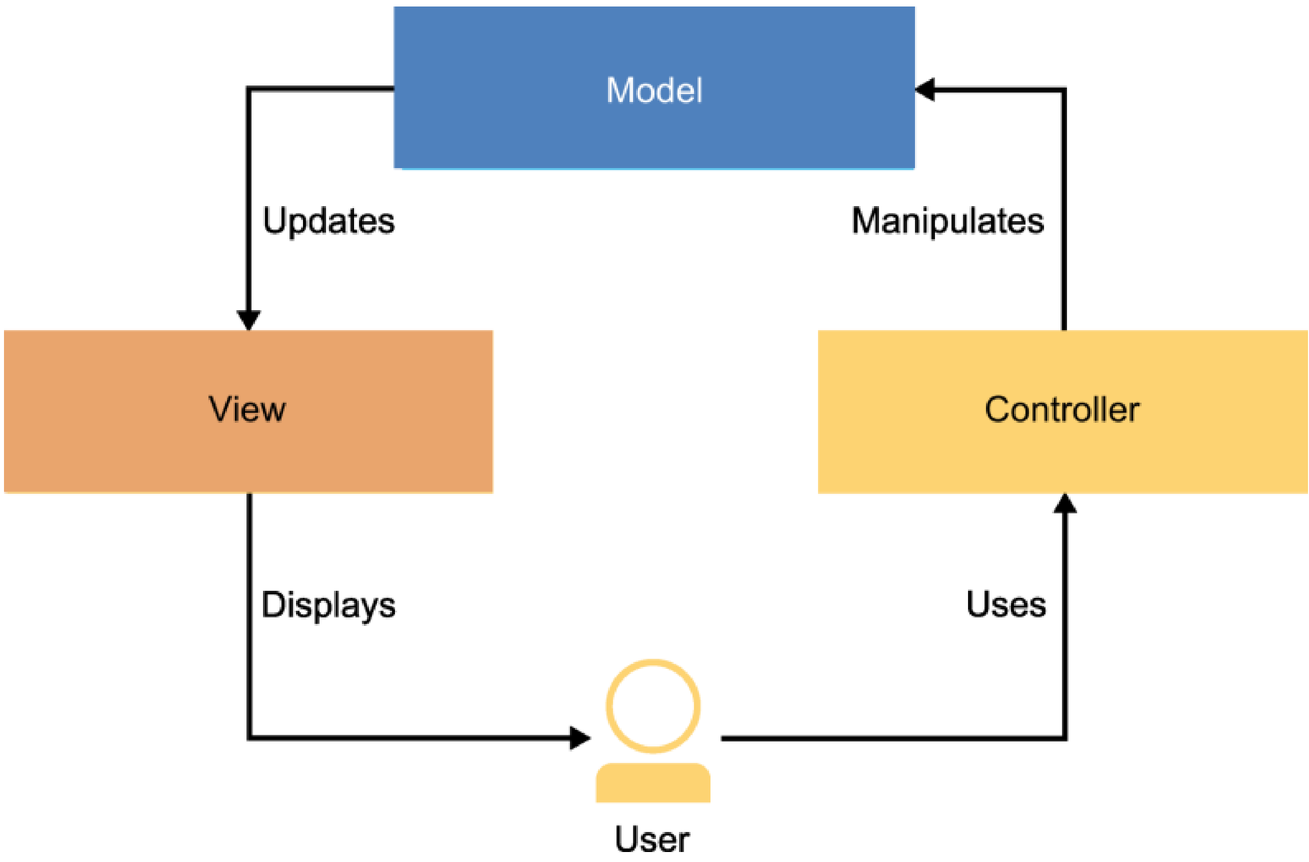
Implementing UX (Polymer 3): Building a simple application in the UX framework

STILL INTERESTED? If you missed my **first article**, I discussed the key concepts and challenges of the UX framework. Don't worry, I'll give a brief summary. But if you've read the article, what's next? This time, we'll get a basic idea of how to build a simple UX application. Sounds good?

- What are the key concepts?
- What are the key challenges?
- What are the new metadata concepts?
- Can we dig deeper into the UX model?
- Can we dig deeper into the UX view?
- Can we build a simple UX application?
- Still want more?

What are the key concepts?

To refresh our memories, UX *"implements an MVC architecture"*. Here's a basic diagram of the typical MVC components and process flows.



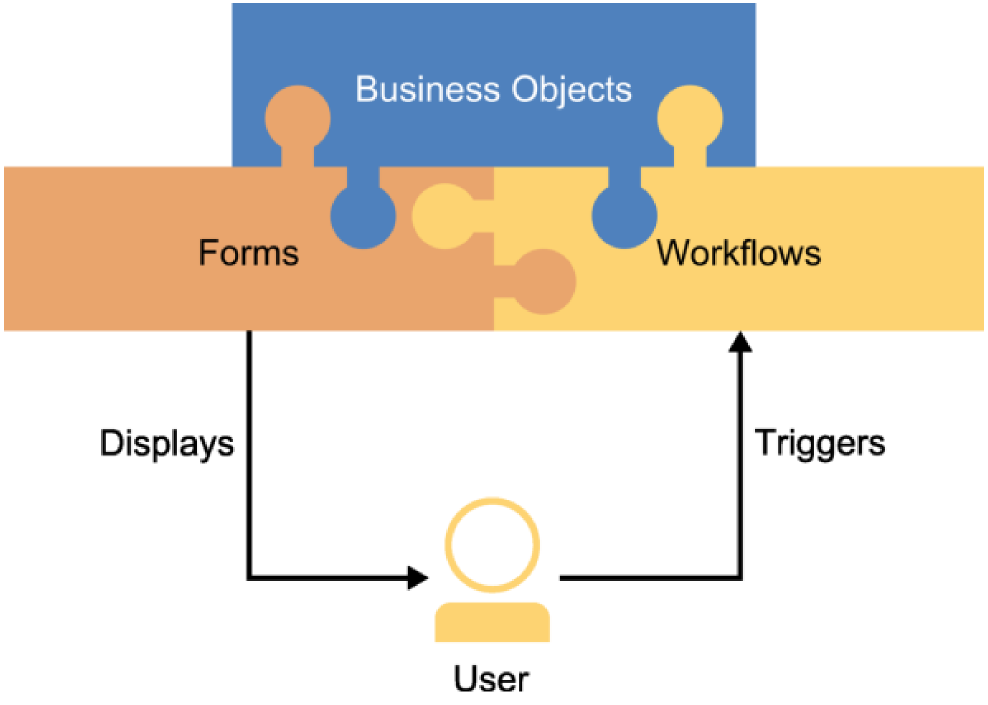
What are the key challenges?

But if you remember, our classic framework doesn't fully apply the *"separation of responsibilities"*. Components are too tightly coupled.

Here are a few examples:

- Records are bound to a single specific form.
- Form sections and fields are tied to BO sections and fields.
- Forms cannot be replaced without breaking workflows.
- The Modify Metadata task in the workflow is tied to a single form.

So, if we redraw the basic MVC diagram with this lack of separation, our TRIRIGA framework might look like this. This is where UX comes in!



At the same time, let's be clear where UX *doesn't* come in.

Because there's no automatic or direct path from key classic concepts to new UX concepts, the term "upgrade" doesn't really apply. In the words of Ryan Koppelman, our (former) manager of TRIRIGA platform development, *"certain concepts do not align, and thus cannot be [directly] upgraded."* So instead, we'll take each concept and compare their approaches.

Comparison of Approaches.

Concept	Classic	UX
Modify Metadata Task	Classic applications typically use workflows with Modify Metadata tasks to hide and show form tabs, sections, and fields, or to change the text or text color of a label.	In UX, there is no concept of a Modify Metadata task. Instead, a variety of layout components, field elements, and action-button elements are available to render a dynamic view.
State Transition Actions	Classic applications typically use state transition actions that call workflows with Modify Metadata tasks to hide and show form tabs, sections, and fields, based on the state of a record.	In UX, there is no concept of a Modify Metadata task. Again, a variety of layout components, field elements, and action-button elements are available to render a dynamic view.
Query Sections	Classic applications typically use query sections to show a collection of records. Query sections can also trigger workflows with Modify Metadata tasks.	In UX, query interactions rely on Query data sources, which can be pulled into table (grid) layout components. Again, there is no concept of a Modify Metadata task.
Query Actions	Section actions and query sections with Find actions are found throughout our classic applications. Find queries also offer an option to add new records.	In UX, there is no concept of section actions or a Find action for query sections. Instead, action-button elements are available to render actions as needed. Also, search interactions rely on Query data sources, which can be pulled into table (grid) layout components, list layout components, or search field elements.
Popup Forms	Popup forms are found throughout our classic applications. Popup forms also display different elements or in different sizes, based on what is selected in the parent form.	In UX, there is no concept of a popup form, which is designed for desktop screens not mobile displays. Instead, different data sources are pulled into their respective components or elements within the same view as needed.
Data Validation	Classic forms rely on Get Temp Record tasks and Modify Metadata tasks to show Attention messages.	In UX, validation relies on in-memory business objects and modal dialogs. This validation approach is significantly different from the classic approach.
Mobile Design	Classic applications were designed for a full desktop experience, not for today's mobile experience with smaller screens and simplified interfaces.	In UX, code that leverages built-in features of Google Polymer elements is “mobile responsive” out-of-the-box. This responsive-design approach is significantly different from the classic approach.

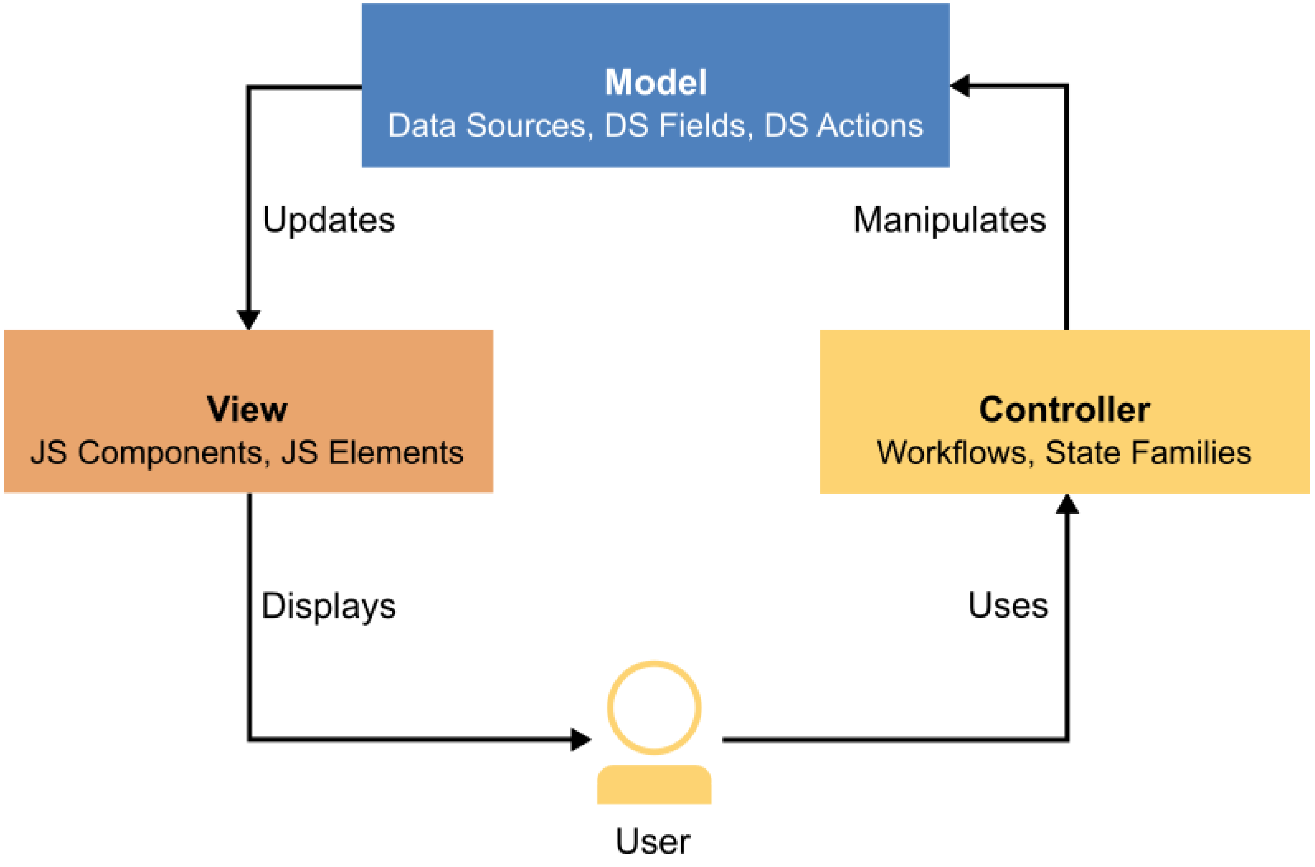
As you can see, while the UX framework tackles the key challenges in decoupling our classic framework into its separate MVC components, it also isn't meant to automatically "upgrade" our classic framework.

As observed by Casey Cantwell, our (former) lead QA engineer on the TRIRIGA platform team, we have *"a unique opportunity to develop a framework for next generation applications."* This innovative freedom is key. With this in mind, let's dig deeper into the UX metadata concepts. Are you ready?

What are the new metadata concepts?

Building on a solid foundation, the UX framework introduces two new metadata concepts: (1) the **model** to retrieve the data and trigger the business logic, and (2) the **view** to render the interfaces or forms. The new renders will be "bolt-on" views that can be quickly added or removed, and will still use our existing application data and workflows.

Once again, if we redraw the basic MVC diagram with our new decoupled metadata approach, our UX framework might look something like this.


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

Can we dig deeper into the UX model?

Of course! As I just mentioned, the model is used *"to retrieve the data and trigger the business logic"*. To be clear, 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.*

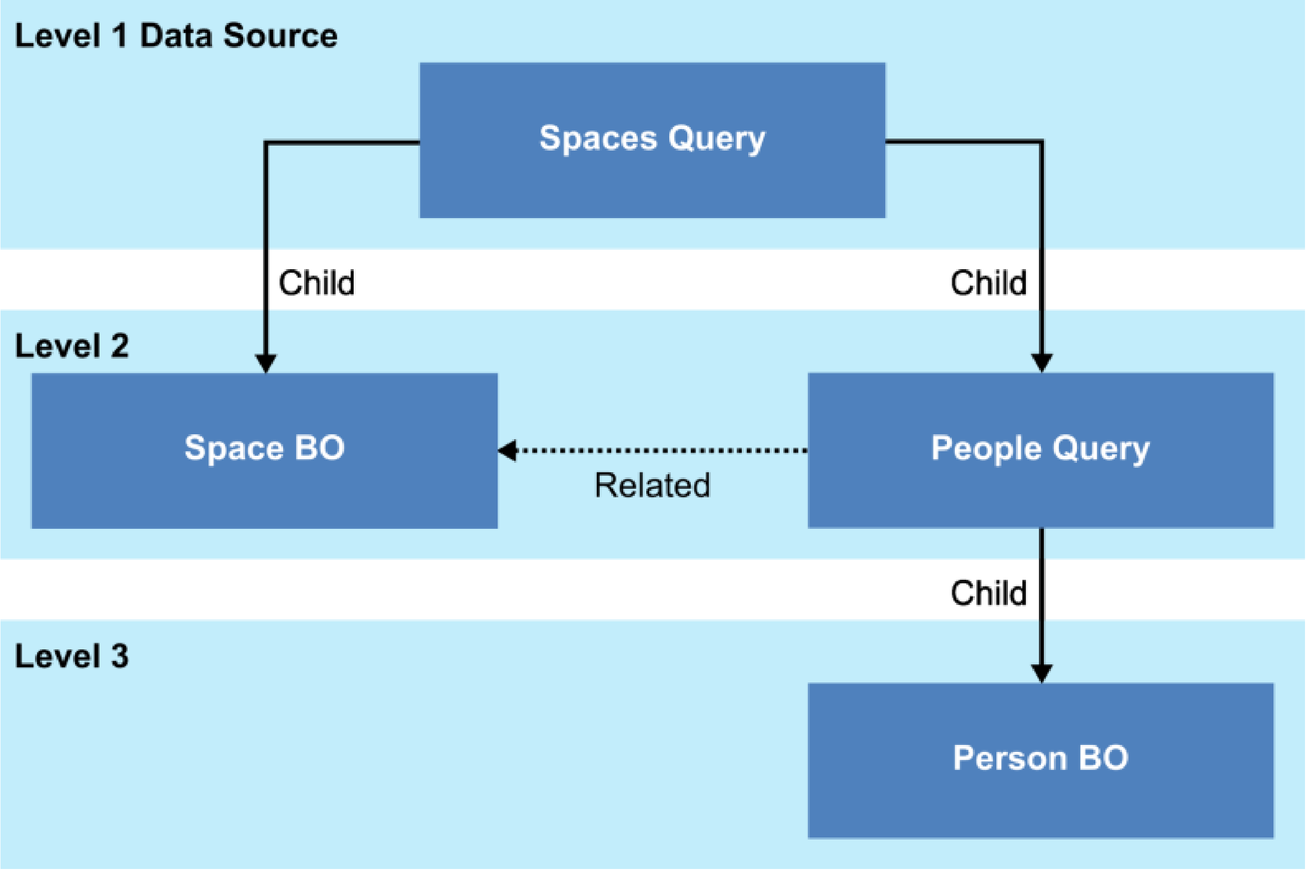
Before we look at some screenshots, here are some longer descriptions.

Component Descriptions.

Component	Description
Data Sources	You can define data sources, child data sources, and related data sources to pull together all of the data needed for a model. A data source can be one of several types: - **Business Object:** This type identifies a single record. Traditional scenarios include persistent Create, Update, and Delete interactions. - **Current User:** This type identifies a single user. Traditional scenarios include language, time-zone, and date-time interactions. - **In-Memory Business Object:** This type stores data in a non-persistent scenario. - **List:** This type identifies a collection of values. Traditional scenarios include list-value interactions. - **Query:** This type identifies a collection of records. Traditional scenarios include table (grid), list, and search interactions. - **Resource Calendar:** This type identifies an array of calendar events for resources. - **Security Information:** This type delivers permission information of the current user to the UX application. - **Smart Section:** This type identifies an associated business record. Traditional scenarios include smart section interactions. For future use. - **UOM:** This type identifies a collection of units of measure. Traditional scenarios include area, length, and currency interactions. - **Work Planner:** This type identifies the work availability for a set of people. It uses the person's calendar to calculate the available hours, while it uses a query to calculate the planned hours. This is intended for Polymer 3 apps only. If you have any questions about these data source types, feel free to check out the **Application Building for the IBM TRIRIGA Application Platform 3** user guide.
Child Data Sources	Child data sources are not required, but can also be powerful in shaping the user experience. They are identical to other data sources, but they operate as children at a lower level beneath their parent data source. In fact, you can add several levels to build a hierarchy of data sources. To illustrate, let's say that you defined Spaces Query as your first-level data source. Then you might define Space BO and People Query as second-level child data sources, where the Space BO would be a related (contextual) data source for People Query. Lastly, you might also define Person BO as a third-level child data source of People Query. - **Level 1 Data Source:** Spaces Query with 2 children. - **Level 2 Data Source:** Space BO. - **Level 2 Data Source:** People Query with related Space BO and with 1 child. - **Level 3 Data Source:** Person BO. With this hierarchy, a user can (1) see a list of spaces, (2) drill into a single space and see people assigned to that space, and (3) drill into a single person record. In our classic framework, this scenario could only be achieved by using many workflows to set variables. In our UX framework, we can achieve this with zero workflows!
Related Data Sources	Related (or contextual) data sources are not required either. But they can be just as powerful in filtering the results of one data source, based on the context of another data source. Imagine that! To illustrate, let's say that you defined Work Task as one data source. Then you might define Responsible Organization as another data source with the related data source of Work Task. You might also define Manager of Organization as yet another data source with the related data source of Responsible Organization. - **Data Source:** Work Task. - **Data Source:** Responsible Organization with related Work Task.

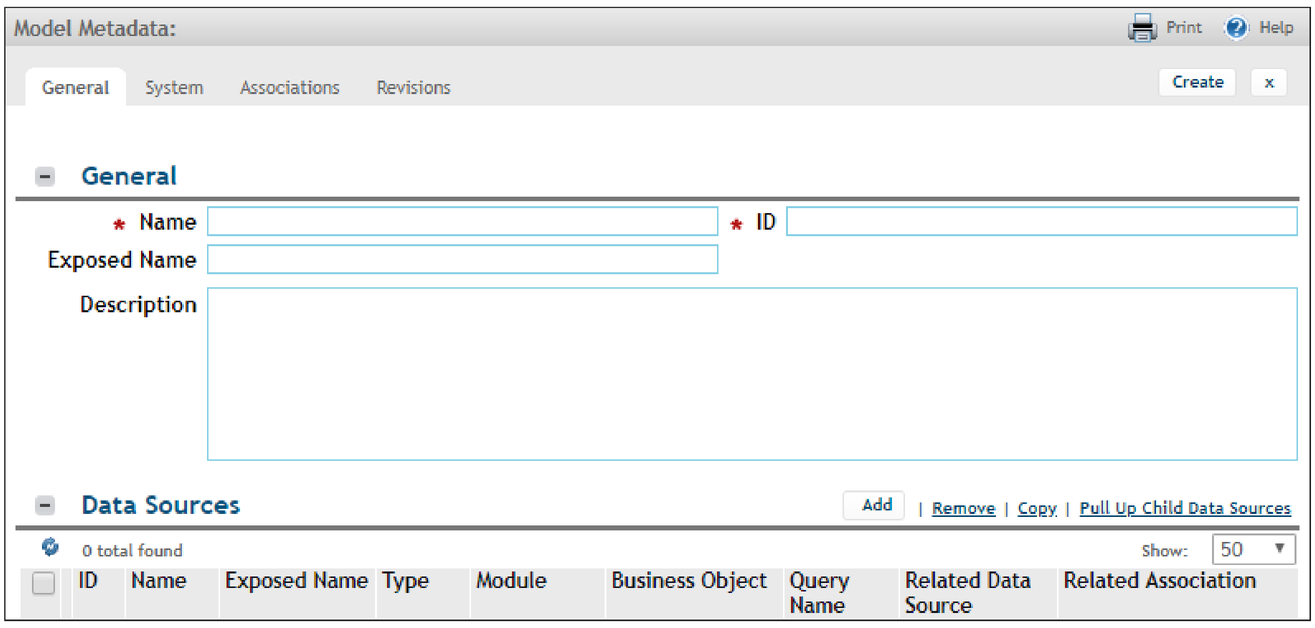
	- **Data Source:** Manager of Organization with related Responsible Organization.
Data Source Fields	Each data source must define at least one field. Each field corresponds to a field in the data source type. To illustrate, let's say that you defined a data source with a Business Object type. Then each field in your data source references a corresponding field in the business object.
Data Source Actions	With data source actions, you define which business rules or workflow logic can be triggered by your data source. For convenience, your actions can also be grouped together into action groups.

Here's a basic diagram of the data source hierarchy and its relationships.



Next, here's an example of a blank model metadata form, where you define your model and add its data sources. In case you're wondering, while new UX applications will use MVC **views**, the UX metadata will use traditional **forms** until our UX framework matures. So stay tuned!

Model Metadata.



General

* Name	* ID
Exposed Name	
Description	

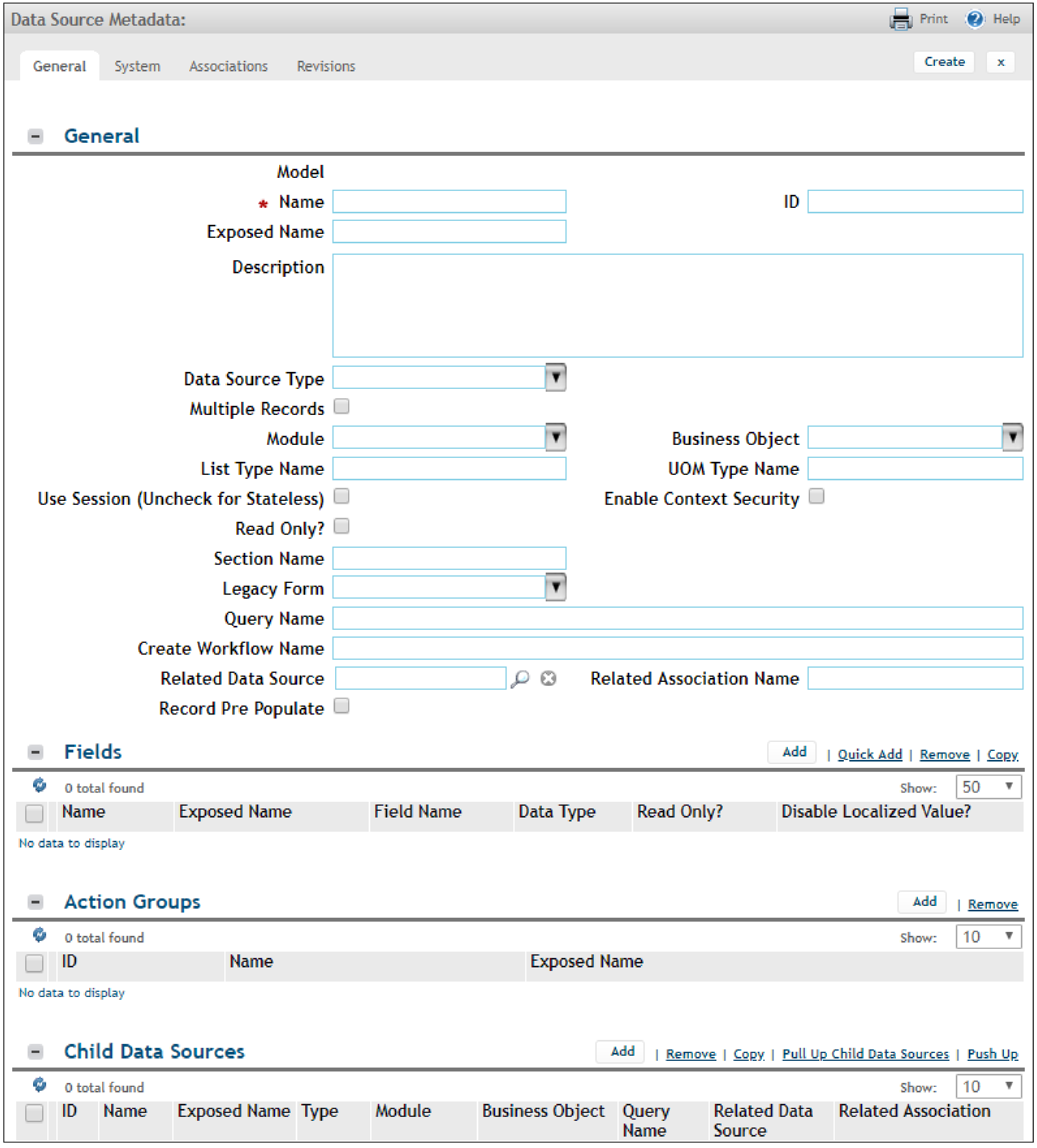
Data Sources

0 total found Show: 50

□	ID	Name	Exposed Name	Type	Module	Business Object	Query Name	Related Data Source	Related Association

Here's an example of a blank data source metadata form, where you define your data source and add its fields, actions, and child data sources.

Data Source Metadata.



General

Model

* Name ID

Exposed Name

Description

Data Source Type

Multiple Records □

Module Business Object

List Type Name UOM Type Name

Use Session (Uncheck for Stateless) □ Enable Context Security □

Read Only? □

Section Name

Legacy Form

Query Name

Create Workflow Name

Related Data Source Related Association Name

Record Pre Populate □

Fields Add | Quick Add | Remove | Copy

0 total found Show: 50

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

No data to display

Action Groups Add | Remove

0 total found Show: 10

□	ID	Name	Exposed Name

No data to display

Child Data Sources Add | Remove | Copy | Pull Up Child Data Sources | Push Up

0 total found Show: 10

□	ID	Name	Exposed Name	Type	Module	Business Object	Query Name	Related Data Source	Related Association

Can we dig deeper into the UX view?

Sure! As I mentioned earlier, the view is used to *"render the interfaces or forms"*. 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.*

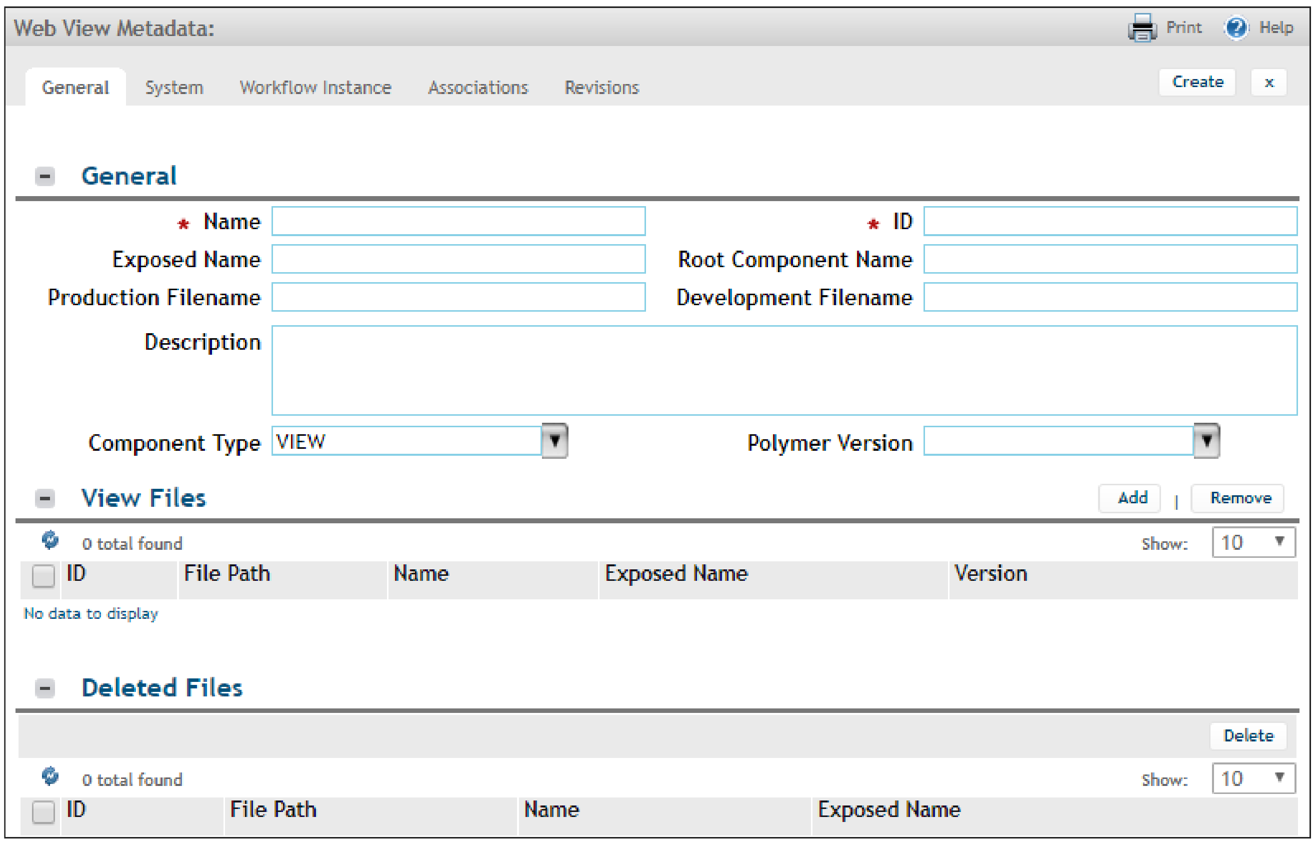
Before we peek at a few screenshots, here are some deeper descriptions.

Component Descriptions.

Component	Description
TRIRIGA Components	You can add Polymer-based components provided by TRIRIGA to assemble all of the necessary data and metadata, or enable field-level interactions or information, in a rendered view. These TRIRIGA components include a TRIRIGA graphic and TRIRIGA search field. Example tags include **<triplat-ds>**, **<triplat-graphic>**, **<triplat-search-input>**, and **<triblock-open-page>**. To access the full list of TRIRIGA components and their related documentation, enter the following URL address: **http://[hostname:port][/context_path]/p/web/doc**, where **[hostname:port]** and **[/context_path]** are the specific values for your TRIRIGA environment. For example, just add **/p/web/doc**: **http://localhost:9080/dev/p/web/doc**
Custom Components	You can add Polymer-based components customized by yourself to enable field-level interactions or information in a rendered view. These components might include a custom search field or custom people card view. Example tags might include **<custom-search-input>**, **<my-paper-button>**, and **<jay-ux-people-card>**.
Polymer Elements	Not only can you add components provided by TRIRIGA or customized by yourself, you can also add elements provided by the Polymer library to provide field-level interactions or information in a rendered view. These Polymer elements include a check box, data field, number field, search field, and text field. • **Iron Elements:** This type represents the core elements that don't express a specific visual design style or language. • **Paper Elements:** This type expresses the material design language by Google. Examples include **<paper-material>**, **<paper-input>**, and **<paper-button>**. • **Other Elements:** Other types like *Neon* elements represent animation, and additional functions. 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.
Traditional Elements	You can also add traditional HTML elements such as containers, headings, or paragraphs. In addition, you can apply CSS styles to these traditional HTML elements as well as TRIRIGA elements and Polymer elements. Example tags include **<div>**, **<h1>**, and **<p>**.

Next, here's an example of a blank view metadata form, where you define your view and add its JavaScript (JS) files. Later, we'll learn to add JS files.

Web View Metadata.



Web View Metadata:

Print | Help

General | System | Workflow Instance | Associations | Revisions

Create | x

General

* Name		* ID	
Exposed Name		Root Component Name	
Production Filename		Development Filename	
Description			
Component Type	VIEW	Polymer Version	

View Files

Add | Remove

0 total found

Show: 10

□ ID	File Path	Name	Exposed Name	Version

No data to display

Deleted Files

Delete

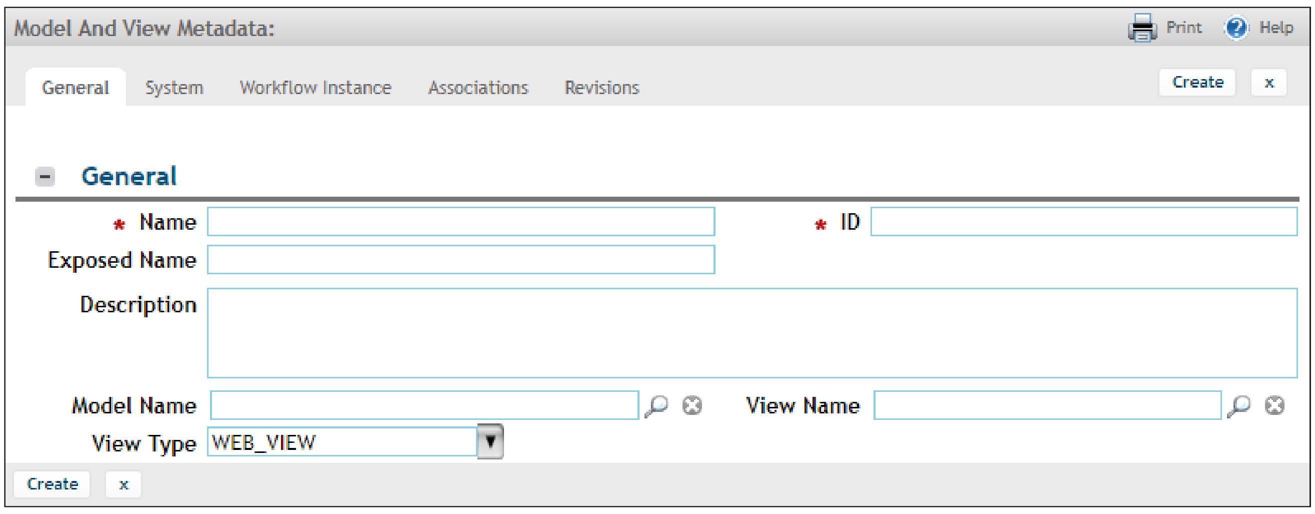
0 total found

Show: 10

□ ID	File Path	Name	Exposed Name

Here's an example of a blank model-and-view metadata form, where you tie your view to a model, and define your view type. More about this later.

Model and View Metadata.



Model And View Metadata:

Print | Help

General | System | Workflow Instance | Associations | Revisions

Create | x

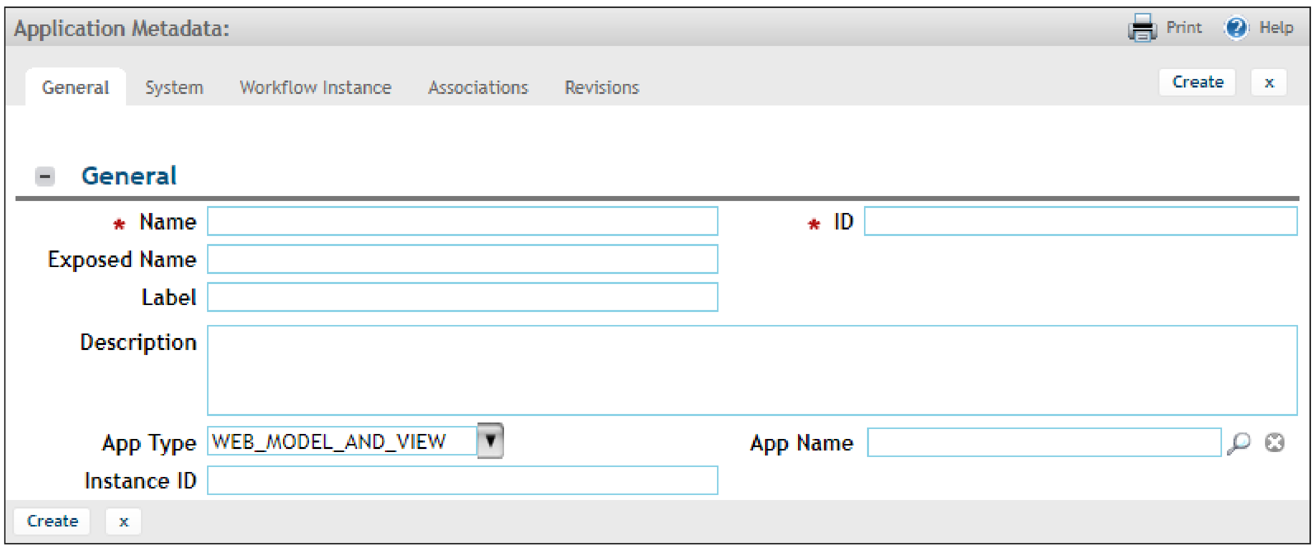
General

* Name		* ID	
Exposed Name			
Description			
Model Name		View Name	
View Type	WEB_VIEW		

Create | x

Finally, here's an example of a blank application metadata form, where you define your application, app type, and app (source) name, such as a model-and-view. Why will UX use an "extra" metadata layer to connect the model-and-view to the application? *Flexibility.* This extra layer allows the application to pull data from either a UX or non-UX source if needed.

Application Metadata.



Application Metadata:

Print | Help

General | System | Workflow Instance | Associations | Revisions

Create | x

General

* Name		* ID	
Exposed Name			
Label			
Description			
App Type	WEB_MODEL_AND_VIEW	App Name	
Instance ID			

Create | x

Can we build a simple UX application?

Yes, I think we can! After all, this is what you were waiting for, right? At this point, you should have a better idea of the concepts and components.

For our example, we'll build 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, then (3) defining and designing a **view** with a single main JavaScript (JS) file. Sounds easy, huh?

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.

Define your model.

Before you begin.

In your web browser's address bar, enter the following URL address: **http://[hostname:port][/context_path]**, where **[hostname:port]** and **[/context_path]** are the specific values for your TRIRIGA environment. For example, if you're building the app locally: **http://localhost:9080/dev**

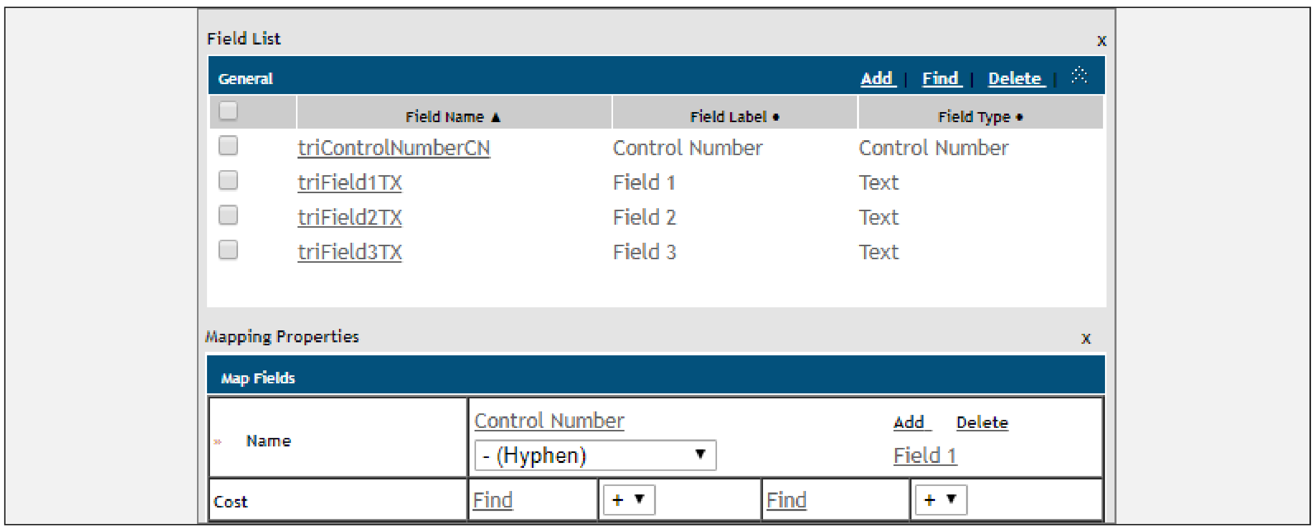
In **Step 7**, I'll ask you to contact your IBM TRIRIGA representative if you cannot access the download location of the **Node.js Package Manager (NPM)** tool. So be prepared for that.

Optional Step: Add the business object.

If you're comfortable with using an existing business object, that's great! You can skip this step. But if you feel safer with a test BO, that's cool too.

From the navigation bar, select **Tools > Builder Tools > Data Modeler**. Add your new module and BO with a prefix that's easy to identify. For our example, we'll add the **jayUX** module and **jayUXBO** business object. Add 3 fields to your BO and update the BO mapping. Then **Publish BO**.

Data Modeler.



Field List

General — Add | Find | Delete

☐	Field Name ▲	Field Label ●	Field Type ●
☐	triControlNumberCN	Control Number	Control Number
☐	triField1TX	Field 1	Text
☐	triField2TX	Field 2	Text
☐	triField3TX	Field 3	Text

Mapping Properties

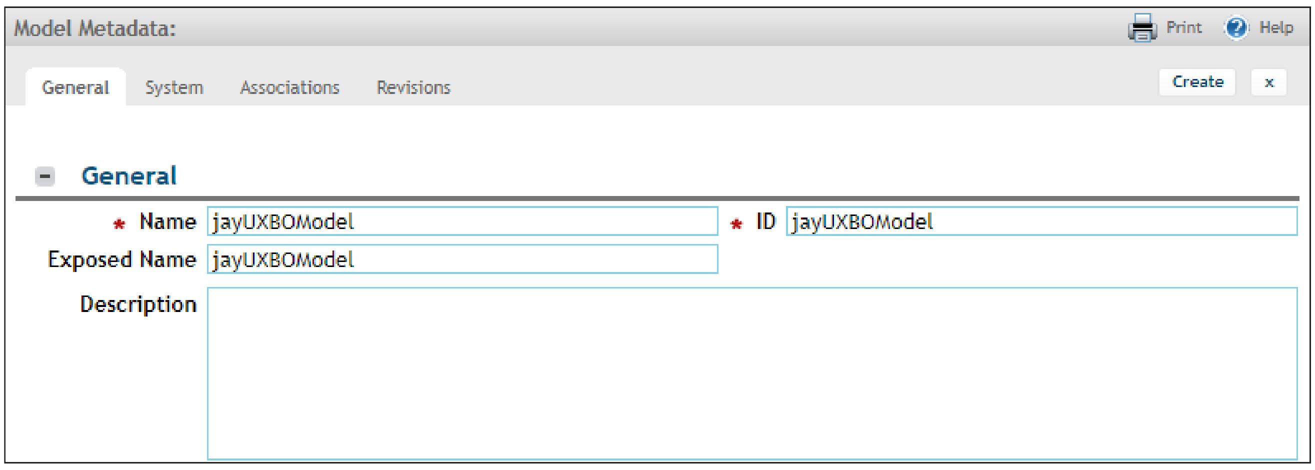
Map Fields

Name	Control Number - (Hyphen) ▼		Add Delete Field 1	
Cost	Find	+ ▼	Find	+ ▼

Step 1: Add the model.

From the navigation bar, select **Tools > [Tools Portal] > Model Designer**. Click **Add**. Enter the name, exposed name, and ID of your model. The exposed name should be a browser-friendly string. For our example, we'll type **jayUXBOModel** and skip the description. Then click **Create**.

Model Metadata.



General

* Name: jayUXBOModel

* ID: jayUXBOModel

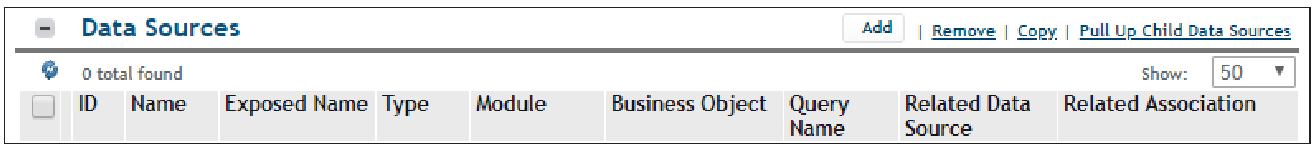
Exposed Name: jayUXBOModel

Description:

Step 2: Add the data source.

Next, in the Data Sources section of your model, click **Add**.

Model Metadata > Data Sources.

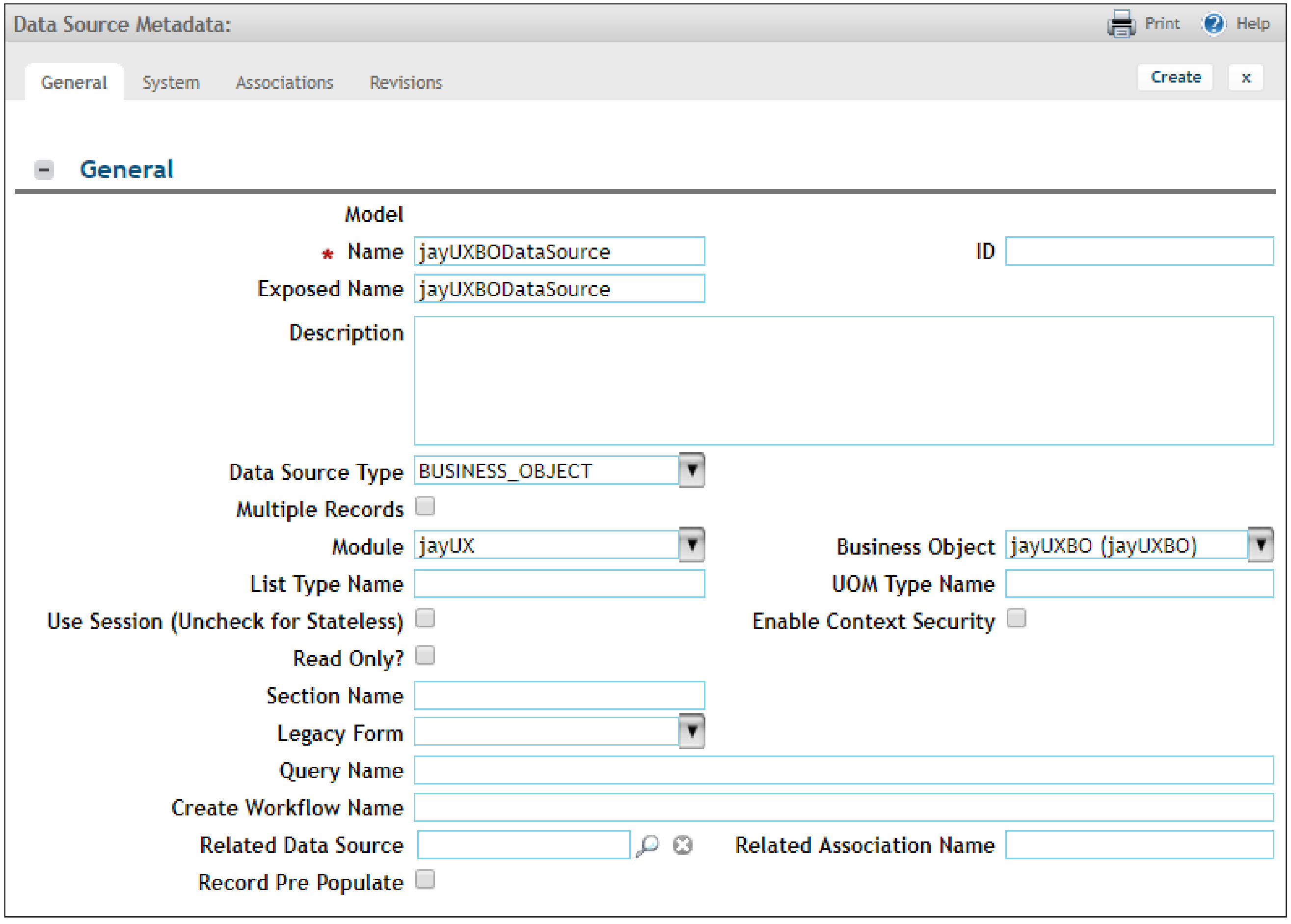


0 total found

☐	ID	Name	Exposed Name	Type	Module	Business Object	Query Name	Related Data Source	Related Association

Enter the name and exposed name of your data source. Since we want to pull data from a record, select **BUSINESS_OBJECT** for the data source type. For our example, we'll type **jayUXBODataSource** and choose the **jayUX** module and **jayUXBO** business object. Then click **Create**.

Data Source Metadata.



General

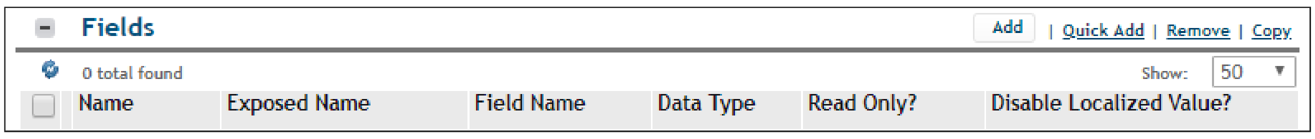
Model

* Name: jayUXBODataSource
ID:
Exposed Name: jayUXBODataSource
Description:
Data Source Type: BUSINESS_OBJECT
Multiple Records □
Module: jayUX
Business Object: jayUXBO (jayUXBO)
List Type Name:
UOM Type Name:
Use Session (Uncheck for Stateless) □
Enable Context Security □
Read Only? □
Section Name:
Legacy Form:
Query Name:
Create Workflow Name:
Related Data Source:
Related Association Name:
Record Pre Populate □

Step 3: Add a few fields for your data source.

Next, in the Fields section of your data source, click **Quick Add**.

Data Source Metadata > Fields.



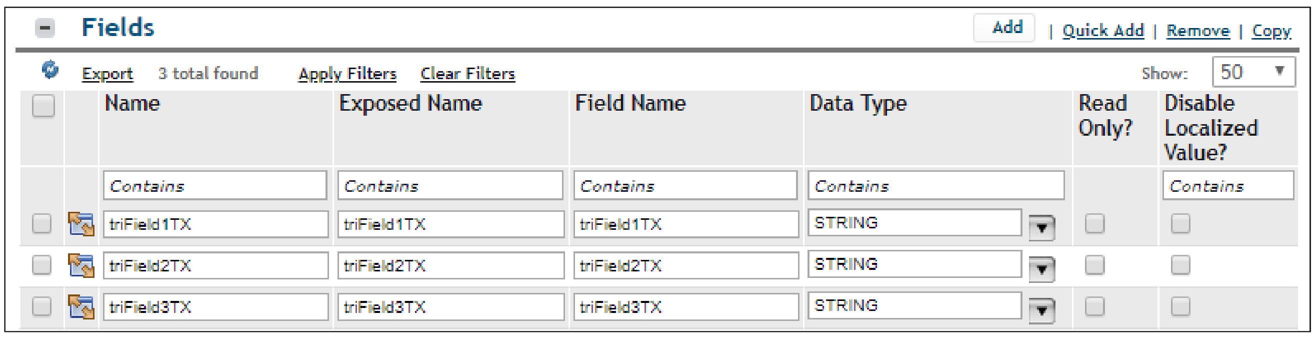
0 total found

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

Enter the name, exposed name, and field name of your data source field. Again, the exposed name should be a browser-friendly string. But the field name should match the field in your data source, like your BO. Be aware that the format of the field name depends on your data source type.

Repeat this for each field that you defined in your test BO or existing BO. For our example, we'll type **triField1TX**, **triField2TX**, and **triField3TX** for the first, second, and third field, respectively. Then click **Save**.

Data Source Metadata > Fields.



3 total found

□	Name	Exposed Name	Field Name	Data Type	Read Only?	Disable Localized Value?
	Contains	Contains	Contains	Contains		Contains
□	triField1TX	triField1TX	triField1TX	STRING	□	□
□	triField2TX	triField2TX	triField2TX	STRING	□	□
□	triField3TX	triField3TX	triField3TX	STRING	□	□

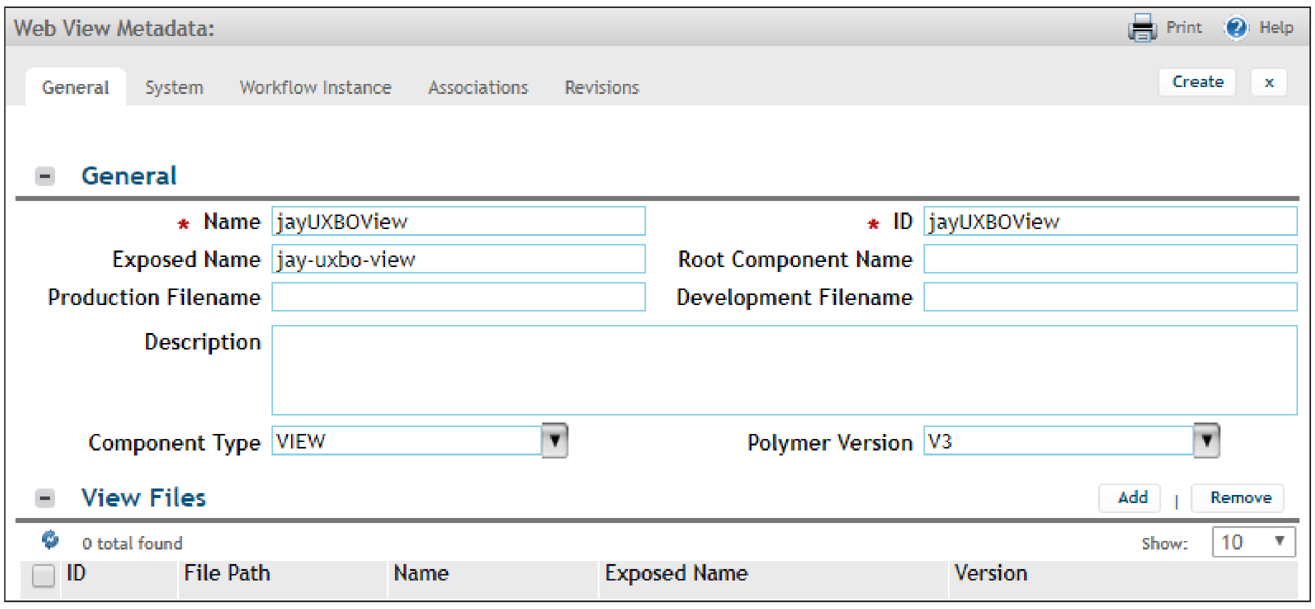
Finally, **Save & Close** your model components -- data source and model. Guess what? We're done with the first part. *You've defined your first model!* Ready to move on to the next part?

Define your view connections.

Step 4: Add the view.

From the navigation bar, select **Tools > [Tools Portal] > Web View Designer**. Click **Add**. Enter the name, exposed name, and ID of your view. The exposed name should include a dash (-). For our example, we'll type **jayUXBOView** for the name and ID, type **jay-uxbo-view** for the exposed name, and skip the description. Choose the **VIEW** component type and **V3** version of Polymer. Click **Create**. Then **Save & Close**.

Web View Metadata.



General

* Name: jayUXBOView
* ID: jayUXBOView
Exposed Name: jay-uxbo-view
Root Component Name:
Production Filename:
Development Filename:
Description:
Component Type: VIEW
Polymer Version: V3

View Files

0 total found

□	ID	File Path	Name	Exposed Name	Version

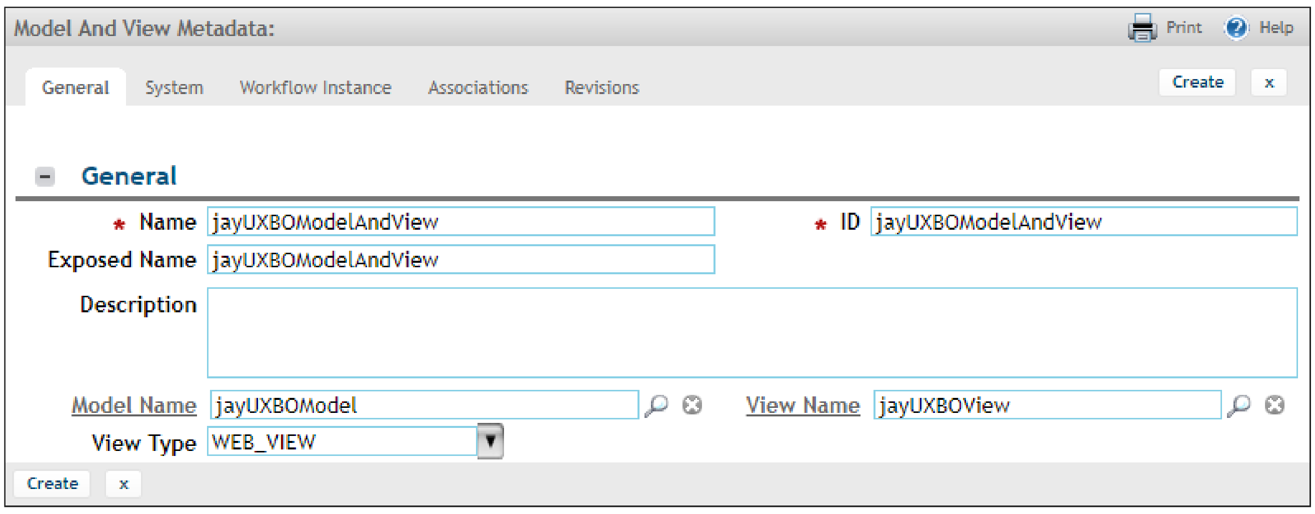
Why do we need a dash in the exposed name? *In Polymer, custom element names must always contain a dash (-).* This distinguishes custom elements from regular elements but also ensures forward compatibility when new tags are introduced. So, later in our example, when you design your JS view, your metadata will already reflect that dash.

Why are we skipping the View Files section? We're saving this part for later! So, for now, let's define the rest of the connections.

Step 5: Add the model-and-view.

From the navigation bar, select **Tools > [Tools Portal] > Model and View Designer**. Click **Add**. Enter the name, exposed name, and ID of your model-and-view. For our example, we'll type **jayUXBOModelAndView**. Enter the names of the model and the view that you defined earlier. For the view type, select **WEB_VIEW**. Click **Create**. Then **Save & Close**.

Model and View Metadata.



General

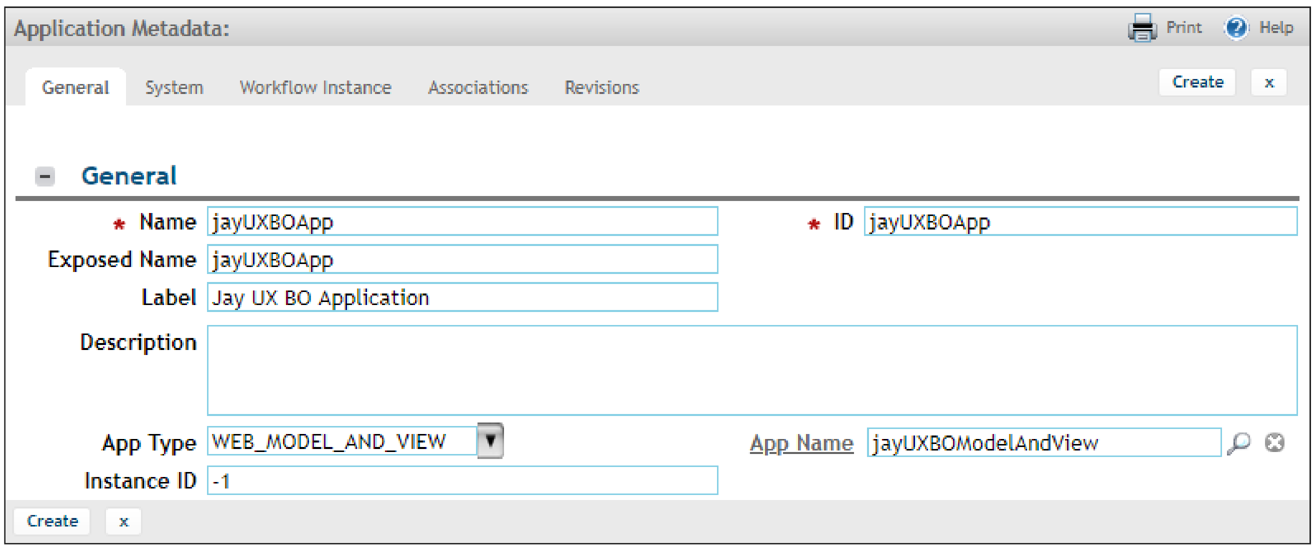
* Name	jayUXBOModelAndView	* ID	jayUXBOModelAndView
Exposed Name	jayUXBOModelAndView		
Description			
Model Name	jayUXBOModel	View Name	jayUXBOView
View Type	WEB_VIEW		

Step 6: Add the application for your model-and-view.

From the navigation bar, select **Tools > [Tools Portal] > Application Designer**. Click **Add**. Enter the name, exposed name, and ID of your application. For our example, we'll type **jayUXBOApp**. For the label, type **Jay UX BO Application**.

For the app type, select **WEB_MODEL_AND_VIEW**. For the app (source) name, enter the name of the model-and-view that you defined earlier. For the instance ID, type **-1** to generate a new record when the application is opened. Click **Create**. Then **Save & Close**.

Application Metadata.



General

* Name	jayUXBOApp	* ID	jayUXBOApp
Exposed Name	jayUXBOApp		
Label	Jay UX BO Application		
Description			
App Type	WEB_MODEL_AND_VIEW	App Name	jayUXBOModelAndView
Instance ID	-1		

Guess what? We're done with the second part. *You've defined your view connections!* Ready to move on to the next part?

Define your view.

Step 7: Install the NPM and TRIRIGA tools.

Contact your IBM TRIRIGA representative or business partner if you cannot access the download location of the Node.js Package Manager (NPM) tool. This NPM tool is used to install several TRIRIGA tools which allow you to populate the JavaScript (JS) files in your view metadata, preview your JS changes, and sync (deploy) your JS changes with the JS files in your TRIRIGA environment. Be aware that these tools are **not** officially supported at this time.

Download and install the Node/NPM file. For example: **node-v8.12.0-x64.msi**.

Next, open your command prompt. Run the following NPM commands to install the following TRIRIGA tools.

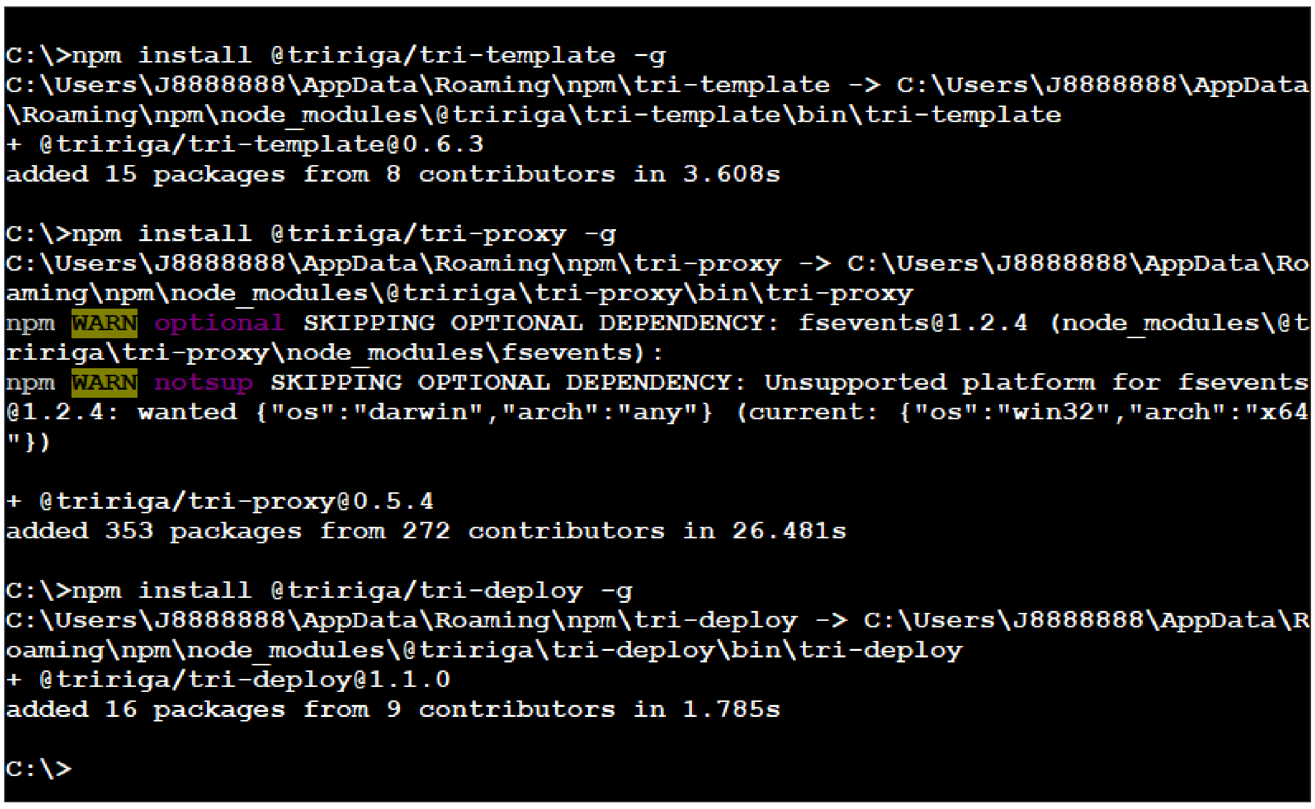
If you see any NPM-related warnings (optional, unsupported, or deprecated), you can ignore them.

TRIRIGA Tools.

Tool	Description
tri-template	**npm install @tririga/tri-template -g** This command installs the **tri-template** tool. **tri-template** This is a simple tool that generates UX view skeletons from available templates. This tool resembles the WebViewSync **addview -s** starter view command. If you're curious, feel free to check out the tri-template options and details.
tri-proxy	**npm install @tririga/tri-proxy -g** This command installs the **tri-proxy** tool. **tri-proxy** This is a simple tool that serves UX views from your local file system and proxies all other view files and calls to a TRIRIGA server. This tool resembles the WebViewSync **sync -a** command, but provides a continuous preview (after each file save) **without** permanent changes. If you're curious, feel free to check out the tri-proxy options and details.
tri-deploy	**npm install @tririga/tri-deploy -g** This command installs the **tri-deploy** tool. **tri-deploy** This is a simple tool that deploys UX views to a TRIRIGA server. It updates the UX view files on the server with the files from the specified local directory, and deletes any files on the server that does not exist in the local directory. This tool resembles the WebViewSync **push** or **sync -a** command, but provides a **one-time** action (not continuous) with permanent changes. If you're curious, feel free to check out the tri-deploy options and details.
tri-pull	**npm install @tririga/tri-pull -g** This command installs the **tri-pull** tool. **tri-pull** This is a simple tool that pulls UX views from a TRIRIGA server. It updates the UX view files in the executed local directory with the files from the server. This tool

	resembles the WebViewSync **pull** command. If you're curious, feel free to check out the tri-pull options and details.

NPM > Install TRIRIGA Tools.



```
C:\>npm install @tririga/tri-template -g
C:\Users\J8888888\AppData\Roaming\npm\tri-template -> C:\Users\J8888888\AppData
\Roaming\npm\node_modules\@tririga\tri-template\bin\tri-template
+ @tririga/tri-template@0.6.3
added 15 packages from 8 contributors in 3.608s

C:\>npm install @tririga/tri-proxy -g
C:\Users\J8888888\AppData\Roaming\npm\tri-proxy -> C:\Users\J8888888\AppData\Ro
aming\npm\node_modules\@tririga\tri-proxy\bin\tri-proxy
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\@t
ririga\tri-proxy\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents
@1.2.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64
"})

+ @tririga/tri-proxy@0.5.4
added 353 packages from 272 contributors in 26.481s

C:\>npm install @tririga/tri-deploy -g
C:\Users\J8888888\AppData\Roaming\npm\tri-deploy -> C:\Users\J8888888\AppData\R
oaming\npm\node_modules\@tririga\tri-deploy\bin\tri-deploy
+ @tririga/tri-deploy@1.1.0
added 16 packages from 9 contributors in 1.785s

C:\>
```

Step 8: Add the main JS file for your view.

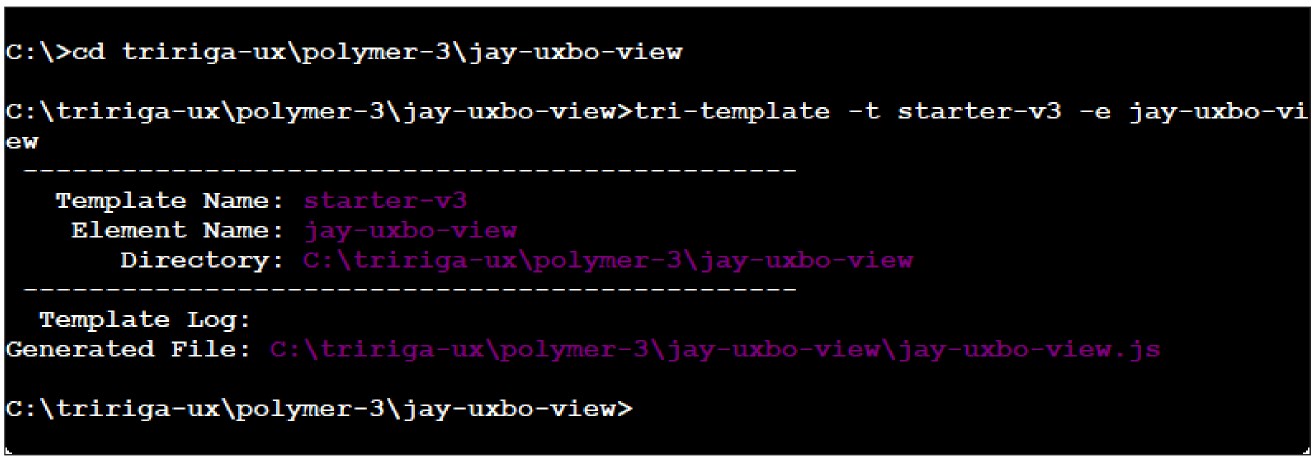
After you've installed the tools, it's time to add your main JS view file. If you remember, we skipped the View Files section of your view metadata. Now we'll auto-populate it.

First, manually create a folder with the same name as your view name. For our example, in the folder **C:\tririga-ux\polymer-3\**, we'll create the new folder name **jay-uxbo-view**. Next, in your command prompt, change directory to this new folder.

To go ahead and add your main JS view file, run the following **tri-template** command: **tri-template -t template-name -e view-exposed-name** where **-t** applies a starter template, **template-name** is one of the available starter templates, **-e** generates your starter view file, and **view-exposed-name** is the exposed name of your view (with the dash).

For our example, we'll use the template name **starter-v3** and the view name **jay-uxbo-view**.

NPM > tri-template > Add View.



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

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

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

After your JS file is added, you'll see that the **C:\tririga-ux\polymer-3\jay-uxbo-view** folder now contains the **jay-uxbo-view.js** file that you started.

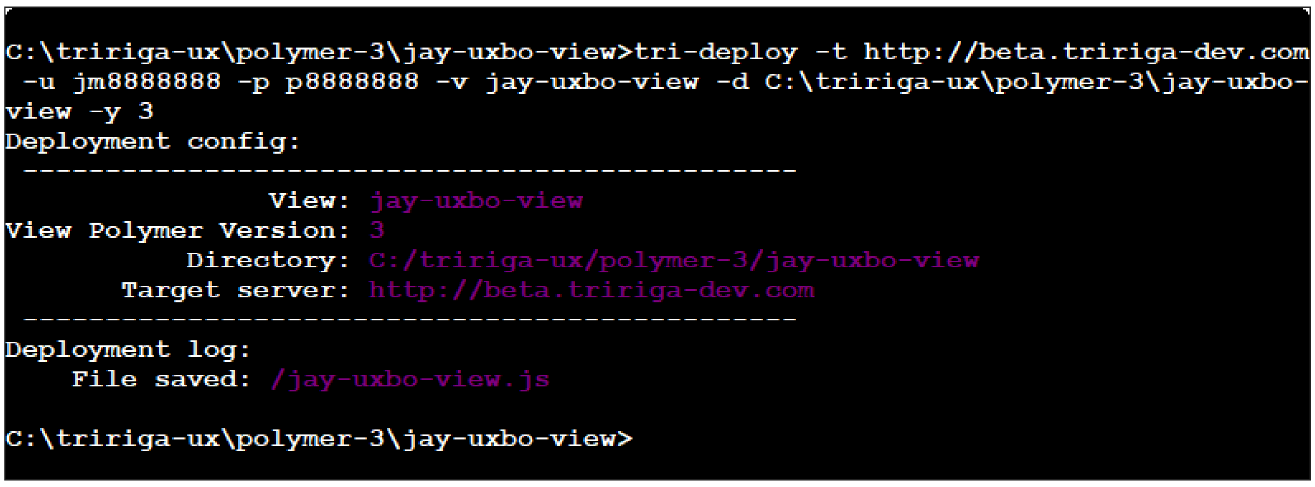
To deploy (or push) your view file to the server, run the following **tri-deploy** command: **tri-deploy -t http://[hostname:port][/context_path] -u username -p password -v view-exposed-name -d directory-path -y 3** where **-t** targets the server URL, **-u** applies your TRIRIGA username, **-p** applies your TRIRIGA password, **-v** deploys your view, **view-exposed-name** is the exposed name of your view (with the dash), **-d** applies your local directory, **directory-path** is the full local directory path of your view, **-y** applies the Polymer version, and **3** is Polymer 3.

For our example, we'll use **http://beta.tririga-dev.com** (no context), the view name **jay-uxbo-view**, and the full local directory path **C:\tririga-ux\polymer-3\jay-uxbo-view**.

Notes:

- If your environment implements a firewall, the **tri-deploy** command must include 2 additional options and their values, where **--basicuser** applies the username for basic authentication, and **--basicpassword** applies the password for basic authentication. Feel free to check out the tri-deploy options and details.

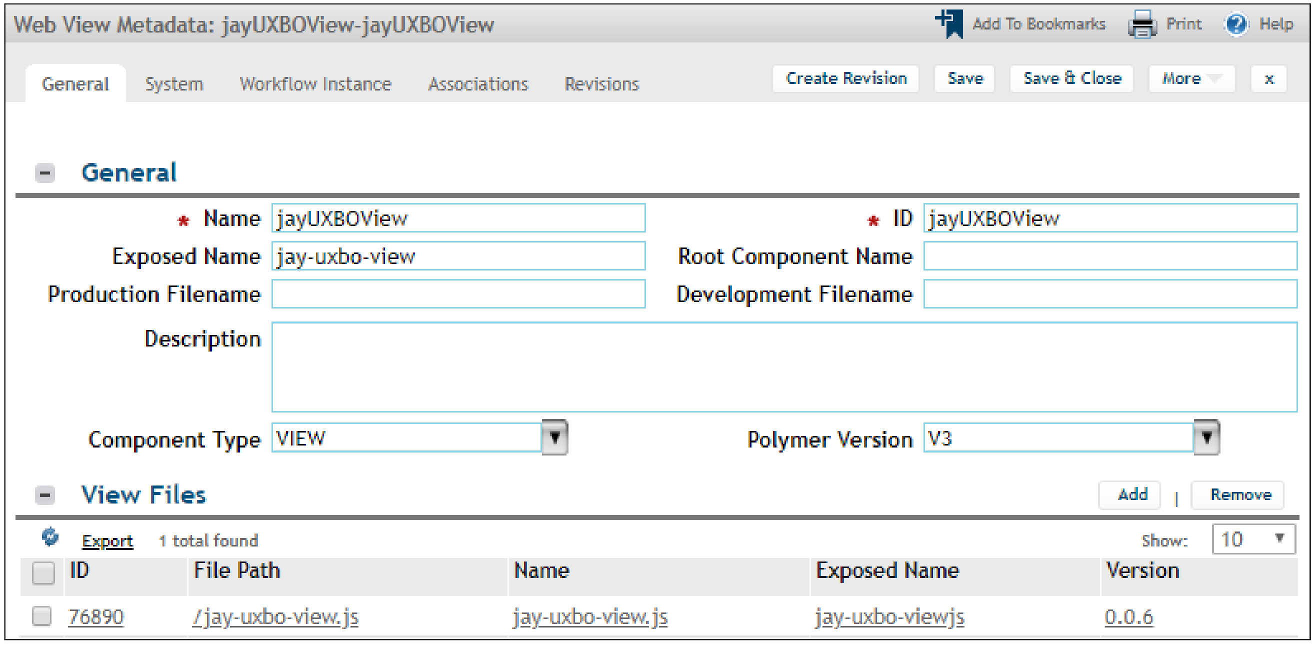
NPM > tri-deploy > Deploy View.



```
C:\tririga-ux\polymer-3\jay-uxbo-view>tri-deploy -t http://beta.tririga-dev.com
 -u jm8888888 -p p8888888 -v jay-uxbo-view -d C:\tririga-ux\polymer-3\jay-uxbo-
view -y 3
Deployment config:
 -----------------------------------------------
                View: jay-uxbo-view
View Polymer Version: 3
           Directory: C:/tririga-ux/polymer-3/jay-uxbo-view
       Target server: http://beta.tririga-dev.com
 -----------------------------------------------
Deployment log:
    File saved: /jay-uxbo-view.js

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

Next, to verify the view metadata, return to **Tools > [Tools Portal] > Web View Designer** and the **jayUXBOView** view. You'll see that the View Files section is now populated with the **jay-uxbo-view.js** view file metadata.

Web View Metadata > View Files.



Web View Metadata: jayUXBOView-jayUXBOView

General

* Name	jayUXBOView	* ID	jayUXBOView
Exposed Name	jay-uxbo-view	Root Component Name	
Production Filename		Development Filename	
Description			
Component Type	VIEW	Polymer Version	V3

View Files

1 total found Show: 10

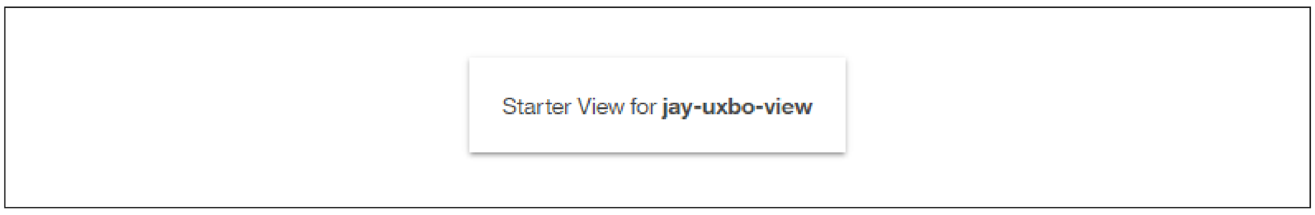
□	ID	File Path	Name	Exposed Name	Version
□	76890	/jay-uxbo-view.js	jay-uxbo-view.js	jay-uxbo-viewjs	0.0.6

Step 9: Access the application.

In your web browser's address bar, enter the following URL address: **http://[hostname:port][/context_path]/p/web/[yourApp]**, where **[hostname:port]** and **[/context_path]** are the specific values for your TRIRIGA environment, and **[yourApp]** is the exposed name of your application. For example, **http://beta.tririga-dev.com/p/web/jayUXBOApp**

If you can see the starter view, that's great! *You've defined your first view and accessed your first application!* Ready to move on to the best part?

UX App > Starter View.



Starter View for **jay-uxbo-view**

Design your view.

Step 10: Start the tri-proxy tool.

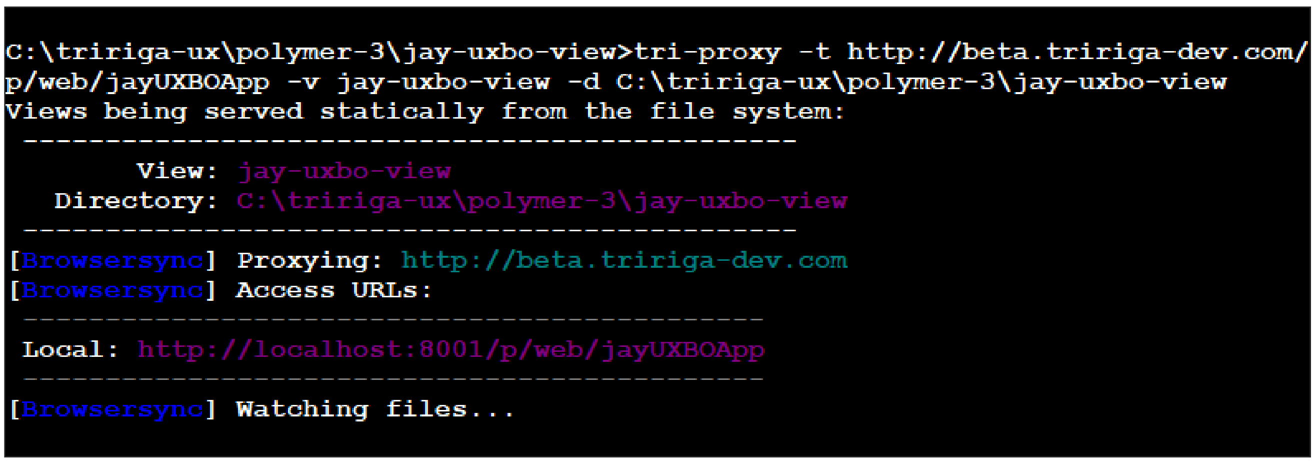
To start the "listening" process so your local JS changes are previewed continuously (but not deployed or synced permanently) with your TRIRIGA environment, return to the command prompt in the same folder as before. Run the following **tri-proxy** command: **tri-proxy -t http://[hostname:port][/context_path]/p/web/[yourApp] -v view-exposed-name -d directory-path** where **-v** serves your local view, **view-exposed-name** is the exposed name of your view, **-d** listens to your local directory for any file changes (or saves), and **directory-path** is the full local directory path of your view.

For our example, we'll use **http://beta.tririga-dev.com/p/web/jayUXBOApp**, the view name **jay-uxbo-view**, and the full local directory path **C:\tririga-ux\polymer-3\jay-uxbo-view**.

Notice that a new browser window opens your preview. In our example, it opens **http://localhost:8001/p/web/jayUXBOApp**.

When you see the message "**Watching files...**", it's time to design your view! But be careful not to close the command prompt.

NPM > tri-proxy > Preview View.



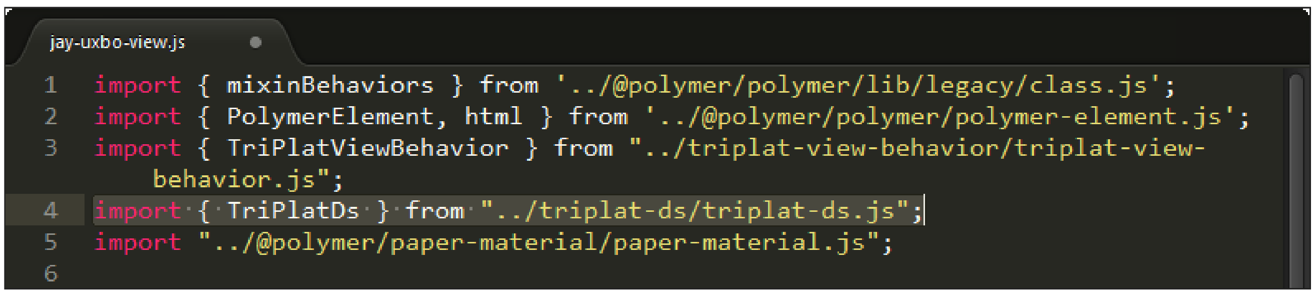
```
C:\tririga-ux\polymer-3\jay-uxbo-view>tri-proxy -t http://beta.tririga-dev.com/
p/web/jayUXBOApp -v jay-uxbo-view -d C:\tririga-ux\polymer-3\jay-uxbo-view
Views being served statically from the file system:
 ----------------------------------------------
        View: jay-uxbo-view
   Directory: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

Step 11: Add a paragraph element to your JS file.

In the new view folder that contains the new starter file that you added, open the JS file with the HTML/JS editor of your choice. In our example, we'll open **jay-uxbo-view.js**. For now, we'll skip the HTML/JS introductions and dive into editing the starter view.

First, add the **import** line at the top of your JS file to import the TRIRIGA **triplat-ds** (data source) component: **import { TriPlatDs } from "../triplat-ds/triplat-ds.js";**

JS File > Import triplat-ds.



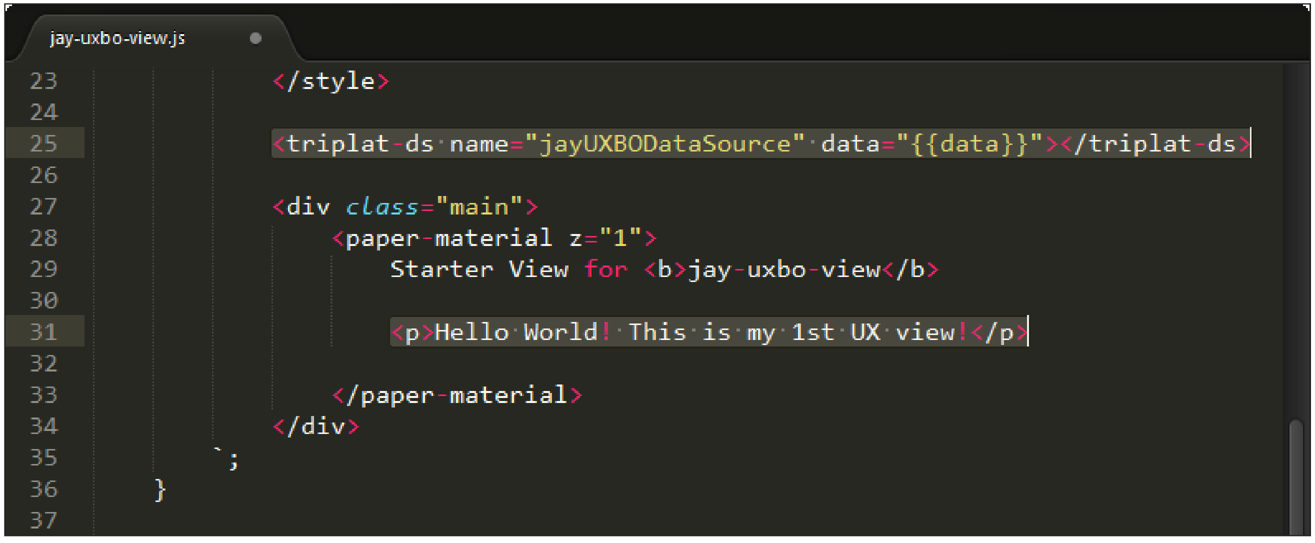
jay-uxbo-view.js

```
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";

```

Next, add the **<triplat-ds>** tag to declare the TRIRIGA **triplat-ds** component: **<triplat-ds name="jayUXBODataSource" data="{{data}}"></triplat-ds>** where **name="jayUXBODataSource"** points to your defined data source.

Now, it's time to add the traditional **<p>** tag for the paragraph element. Let's type: **<p>Hello World! This is my 1st UX view!</p>**

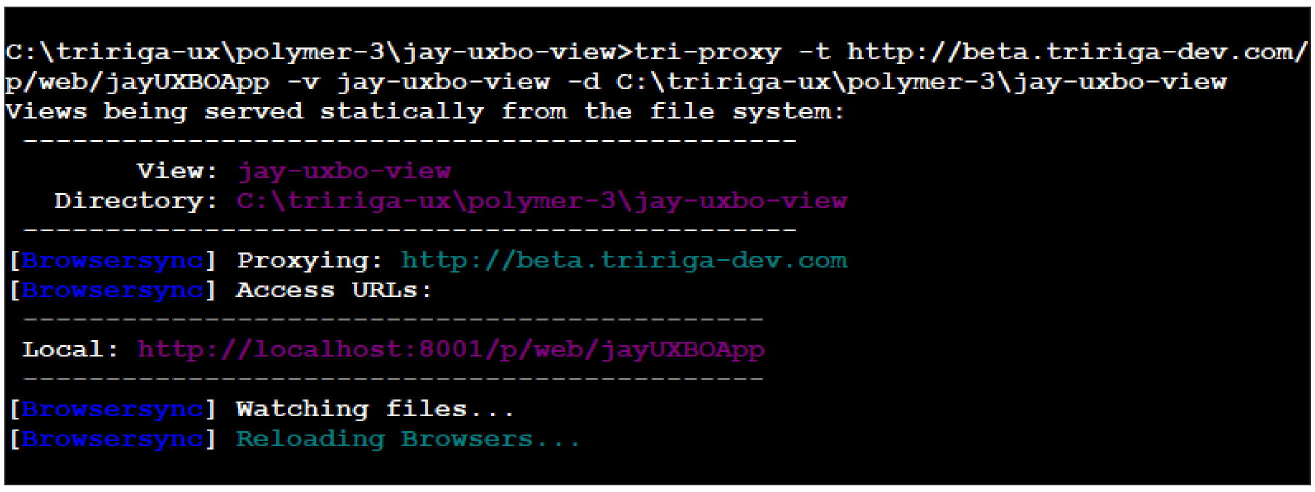
JS File > Declare triplat-ds.



```
      </style>

      <triplat-ds name="jayUXBODataSource" data="{{data}}"></triplat-ds>

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

          <p>Hello World! This is my 1st UX view!</p>

        </paper-material>
      </div>
    `;
  }

```

When you save the file, return to the command prompt. You'll see the message "**Reloading Browsers..**." indicating that the changed **jay-uxbo-view.js** is reloaded into your preview. In other words, each save will reload your preview.

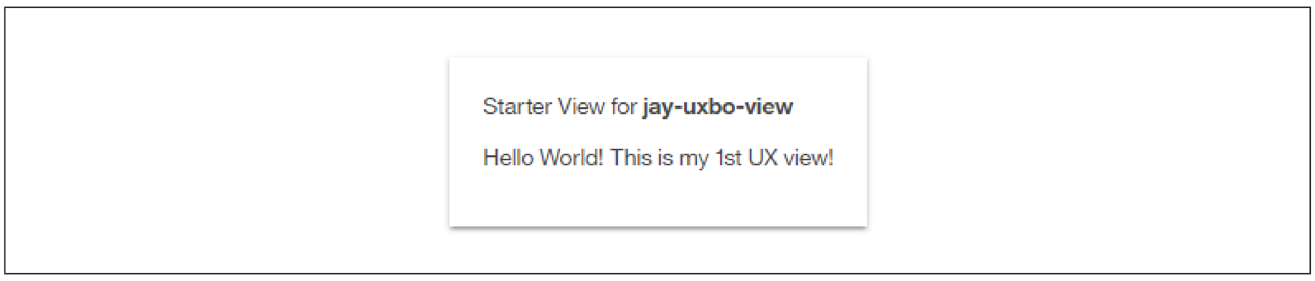
NPM > tri-proxy > Reload View.



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

Next, to verify the change, return to the UX view. Do you see your change? *You've added your first element!* Ready for more?

UX App > Preview Starter View.



Starter View for **jay-uxbo-view**

Hello World! This is my 1st UX view!

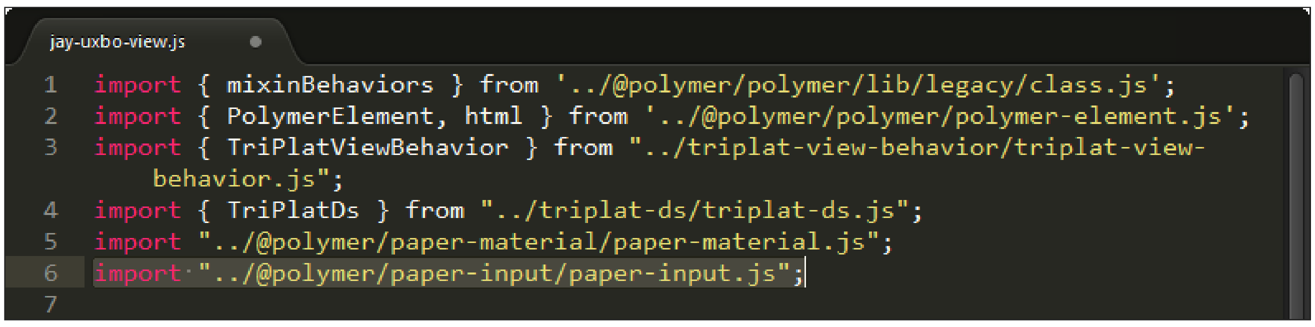
Step 12: Add a few field elements to your JS file.

This time, we'll add the Polymer **<paper-input>** tag for a single-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**.

If you remember, we added several fields to your data source. Now we'll use the **<paper-input>** tag to create a data-binding relationship to each data source field **triField1TX**, **triField2TX**, and **triField3TX**. 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";**

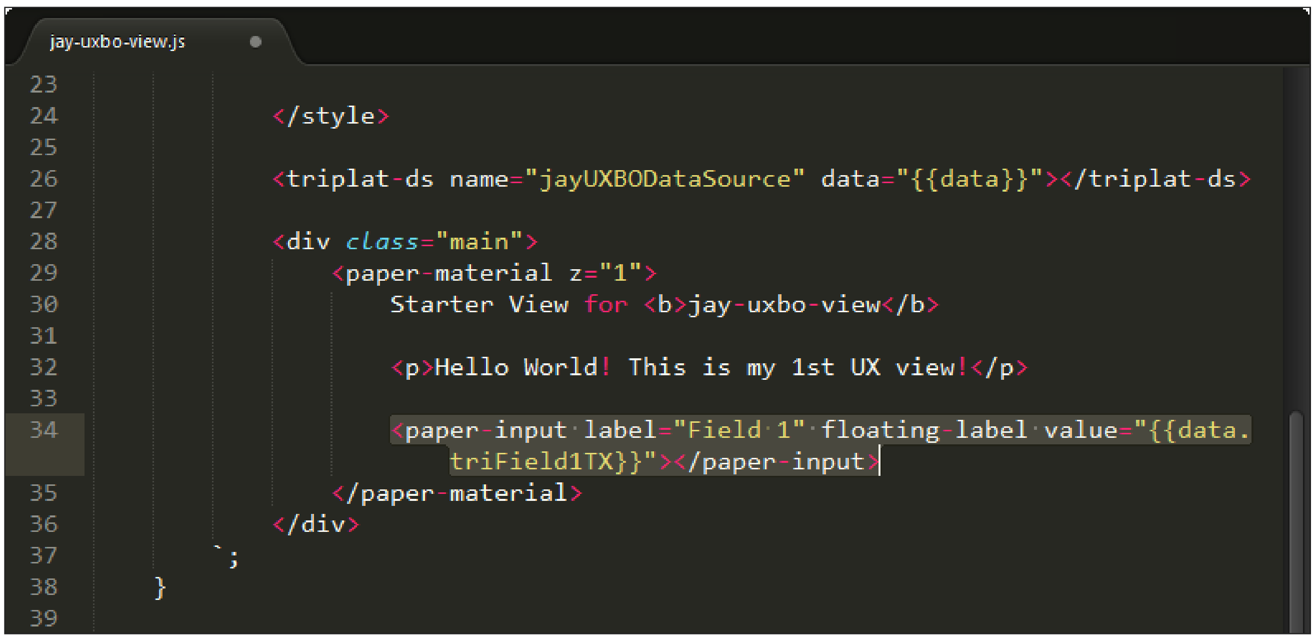
JS File > Import paper-input.



```
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-input.js";

```

Next, add the **<paper-input>** tag to declare the element: **<paper-input label="Field 1" floating-label value="{{data.triField1TX}}"></paper-input>**

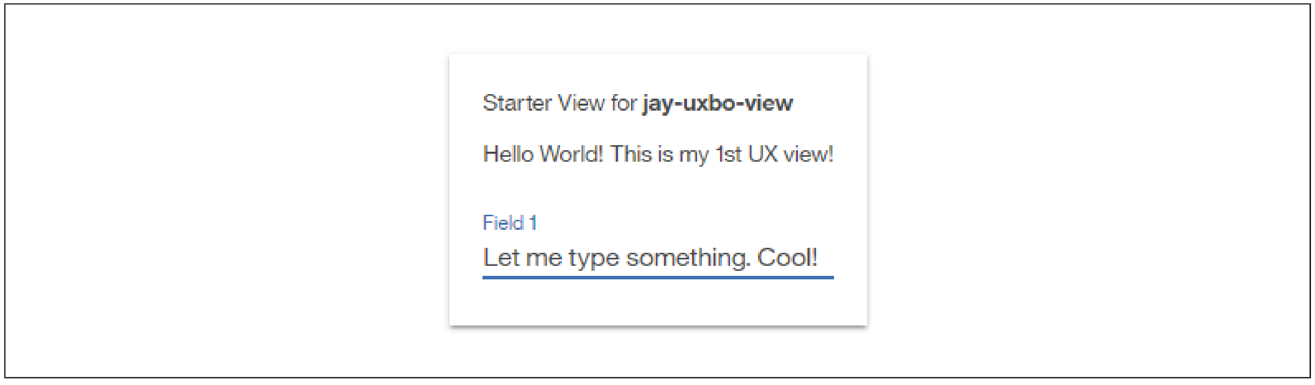
JS File > Declare paper-input.



```

      </style>

      <triplat-ds name="jayUXBODataSource" data="{{data}}"></triplat-ds>

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

          <p>Hello World! This is my 1st UX view!</p>

          <paper-input label="Field 1" floating-label value="{{data.triField1TX}}"></paper-input>
        </paper-material>
      </div>
    `;
  }

```

Save the file and return to the UX view. Do you see your field? If you do, that's cool! Why not add a couple more **<paper-input>** tags on your own?

UX App > Preview Starter View.



Starter View for **jay-uxbo-view**

Hello World! This is my 1st UX view!

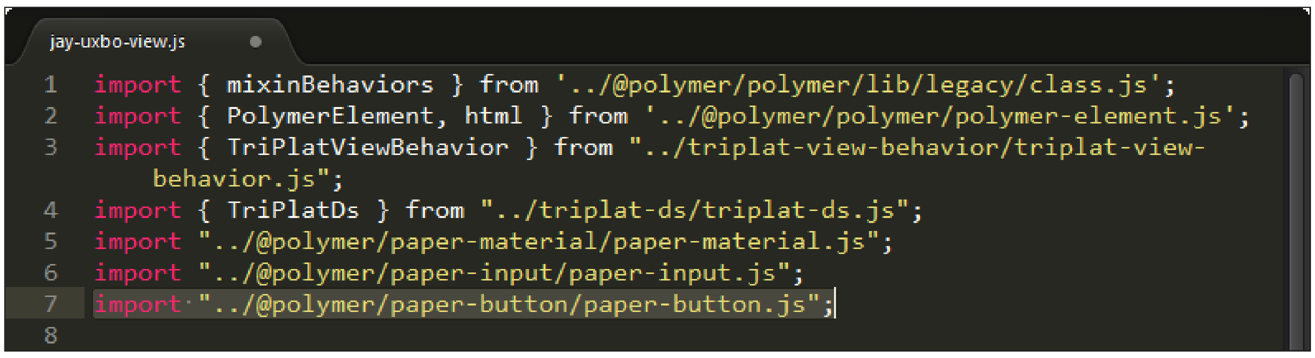
Field 1
Let me type something. Cool!

Step 13: Add a few button elements to your JS file.

This time, we'll add the Polymer **<paper-button>** tag for a button with a ripple effect 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-button/paper-button.js";**

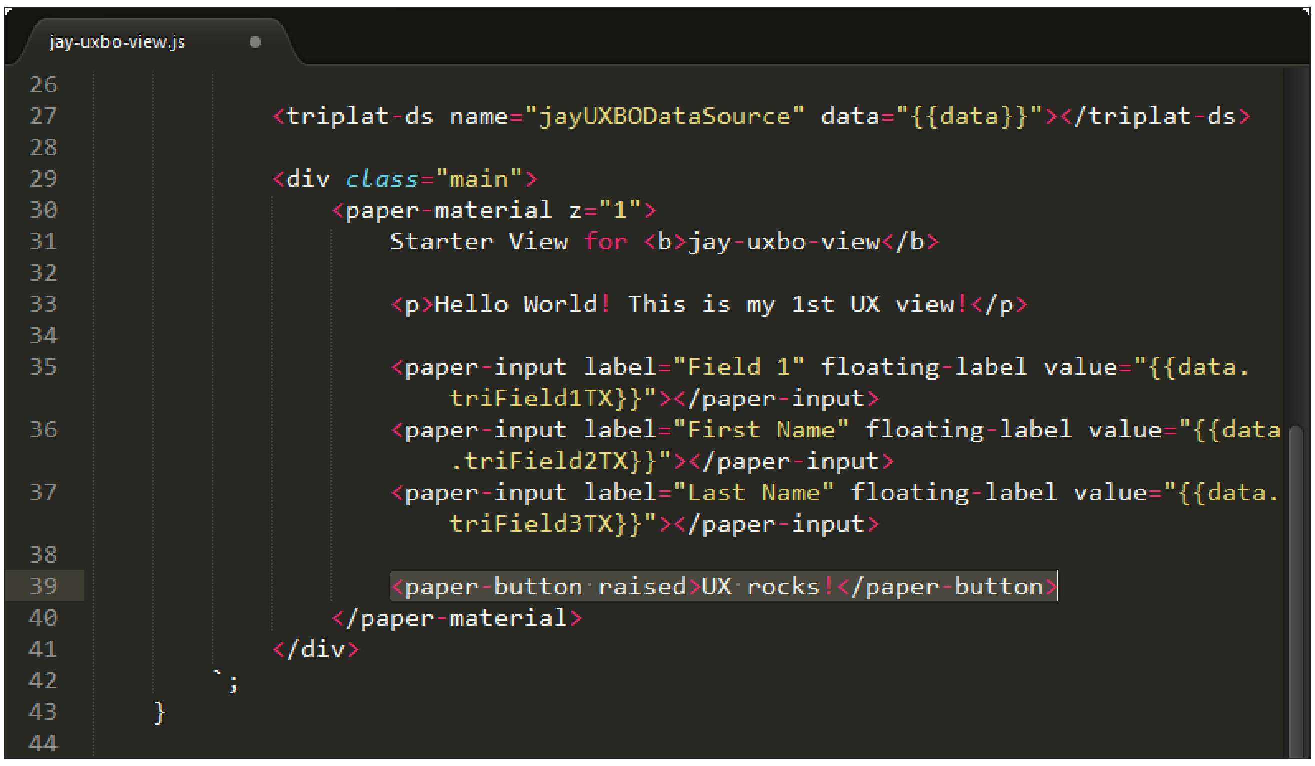
JS File > Import paper-button.



```
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-button>** tag to declare the Polymer element: **<paper-button raised>UX rocks!</paper-button>** where **raised** adds a shadow.

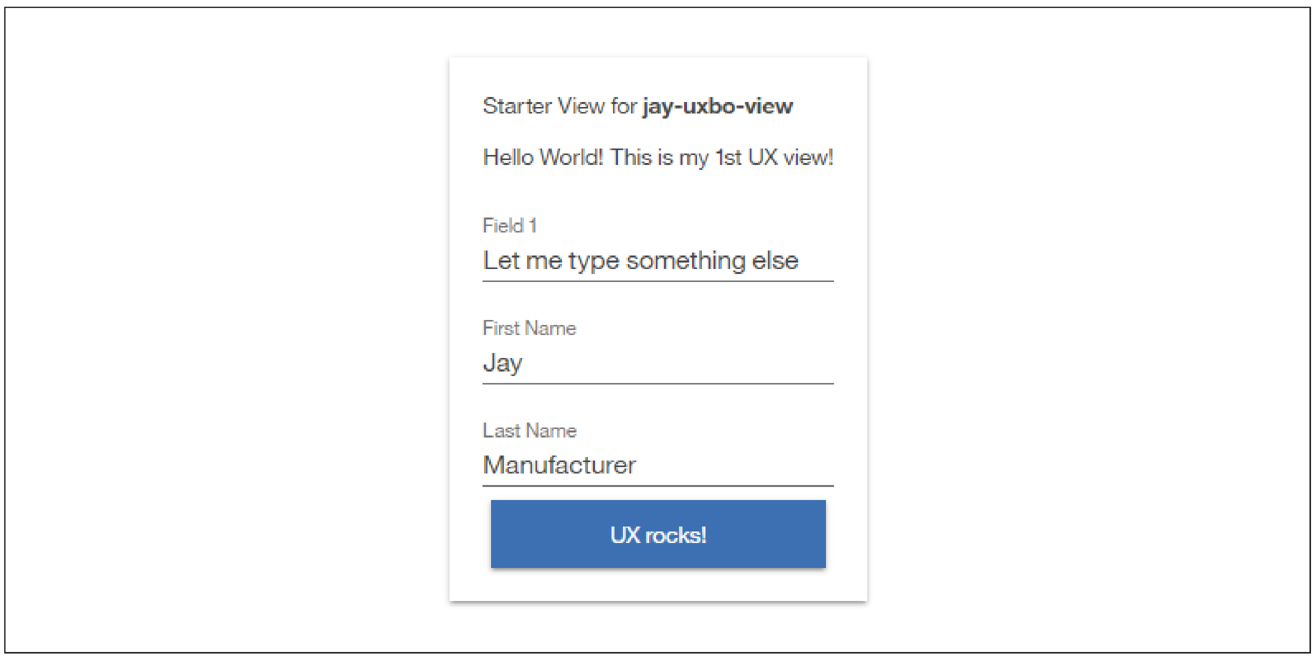
JS File > Declare paper-button.



```

            <triplat-ds name="jayUXBODataSource" data="{{data}}"></triplat-ds>

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

                    <p>Hello World! This is my 1st UX view!</p>

                    <paper-input label="Field 1" floating-label value="{{data.triField1TX}}"></paper-input>
                    <paper-input label="First Name" floating-label value="{{data.triField2TX}}"></paper-input>
                    <paper-input label="Last Name" floating-label value="{{data.triField3TX}}"></paper-input>

                    <paper-button raised>UX rocks!</paper-button>
                </paper-material>
            </div>
        `;
    }

```

Save the file and return to the UX view. Do you see your button? If you do, feel free to add a couple more **<paper-button>** tags on your own!

UX App > Preview Starter View.



Starter View for **jay-uxbo-view**

Hello World! This is my 1st UX view!

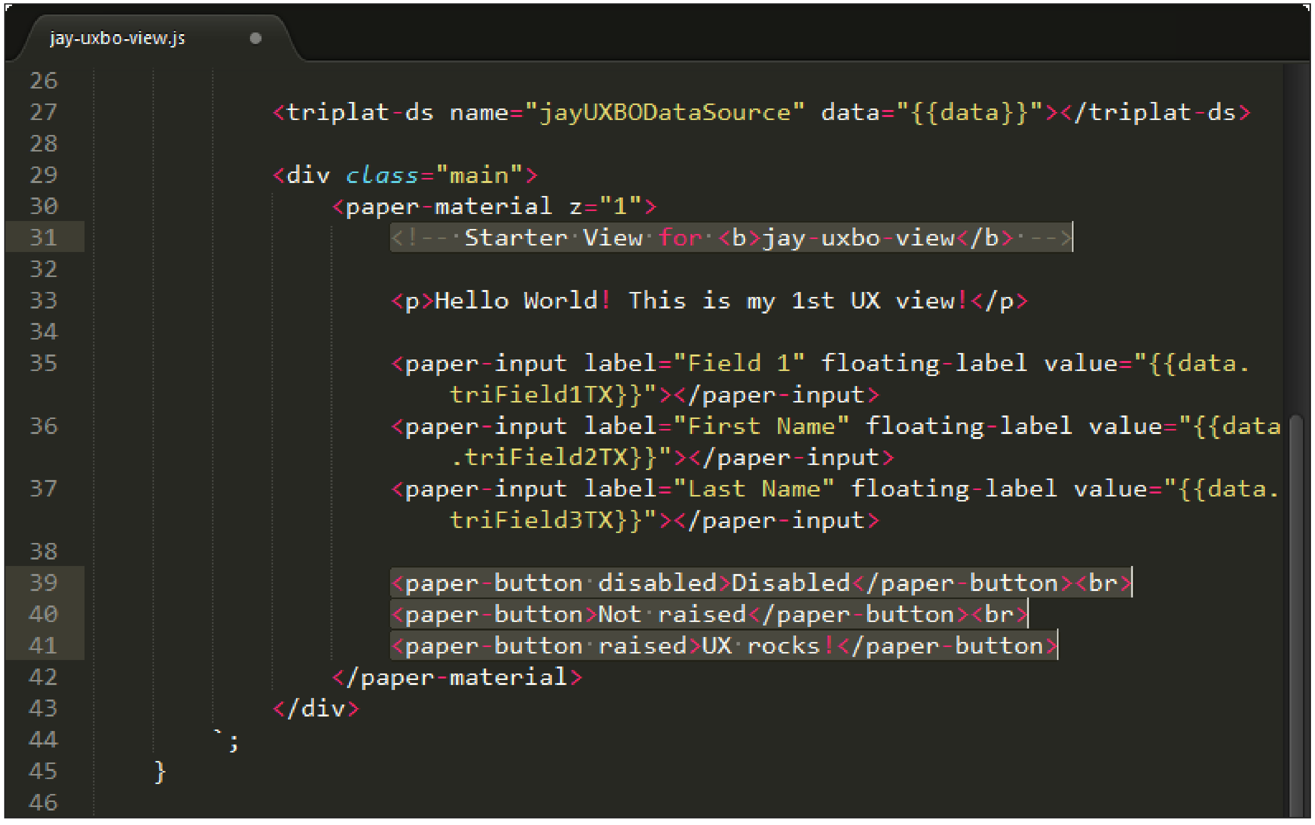
Field 1
Let me type something else

First Name
Jay

Last Name
Manufacturer

With some creativity, you can try other Polymer **<paper-button>** attributes.

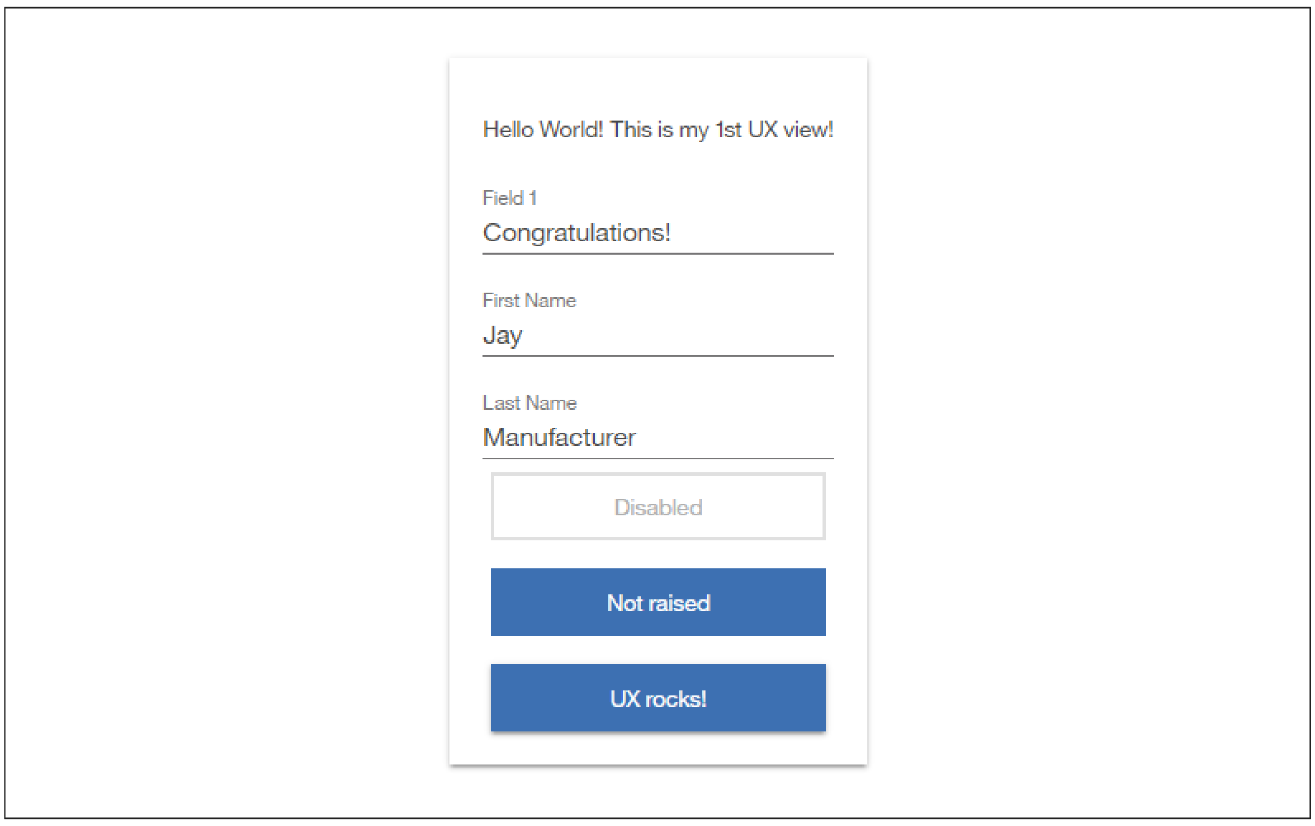
JS File > Declare paper-button.



```

            <triplat-ds name="jayUXBODataSource" data="{{data}}"></triplat-ds>

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

                    <p>Hello World! This is my 1st UX view!</p>

                    <paper-input label="Field 1" floating-label value="{{data.triField1TX}}"></paper-input>
                    <paper-input label="First Name" floating-label value="{{data.triField2TX}}"></paper-input>
                    <paper-input label="Last Name" floating-label value="{{data.triField3TX}}"></paper-input>

                    <paper-button disabled>Disabled</paper-button><br>
                    <paper-button>Not raised</paper-button><br>
                    <paper-button raised>UX rocks!</paper-button>
                </paper-material>
            </div>
        `;
    }

```

Guess what? We're done. As planned, we built a simple 3-field 3-button application. *Congratulations! You've built your first UX application!* (If you want, you can also run **tri-deploy** to push your updated view file to the server.)

UX App > Preview Starter View.



Hello World! This is my 1st UX view!

Field 1
Congratulations!

First Name
Jay

Last Name
Manufacturer

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 (8) | Attachments (36) | About

There are no comments.

Add a comment

Feed for this page | Feed for these comments