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

Implementing UX

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

Edit Page Actions

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

See the UX Article 2 "Implementing UX" PDF for previous versions of this content. What is UX? The standard definition of "UX" is user experience. But for simplicity, I'll refer to the TRIRIGA UX framework as "UX".

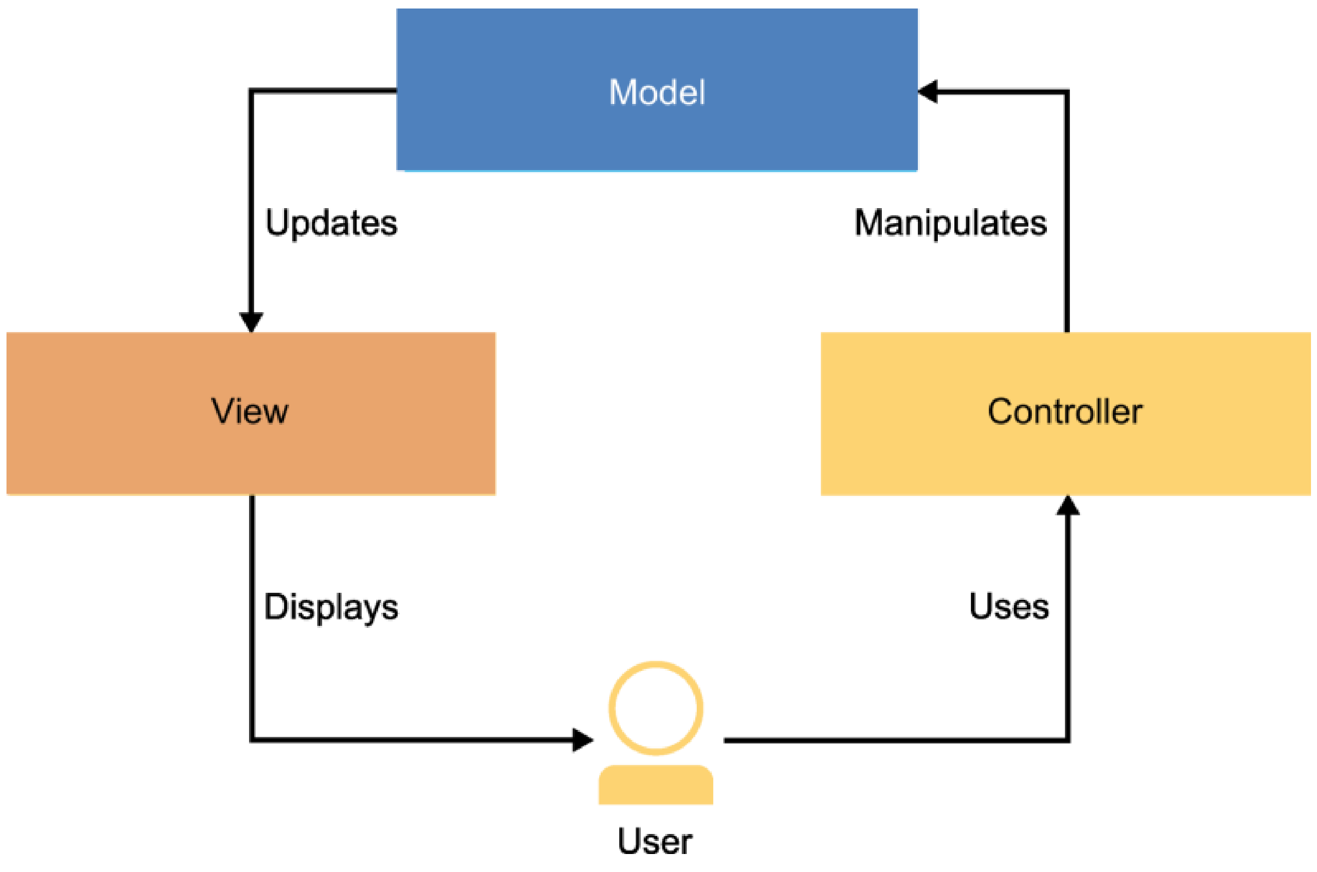
Implementing UX: Building a simple application in the IBM TRIRIGA UX framework

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.



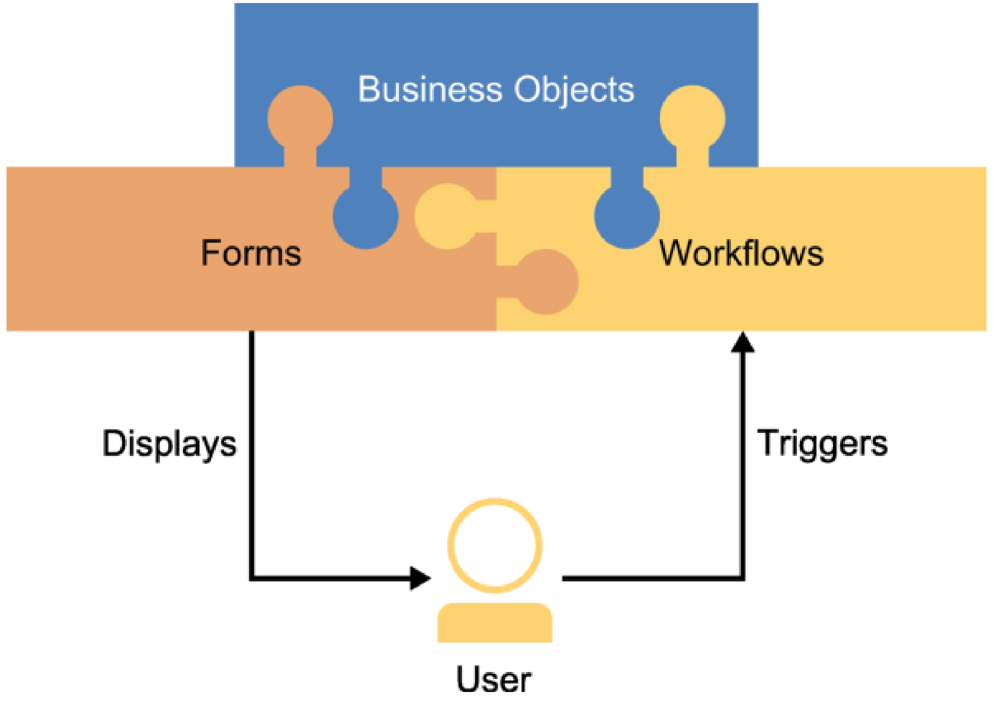
What are the key challenges?

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.

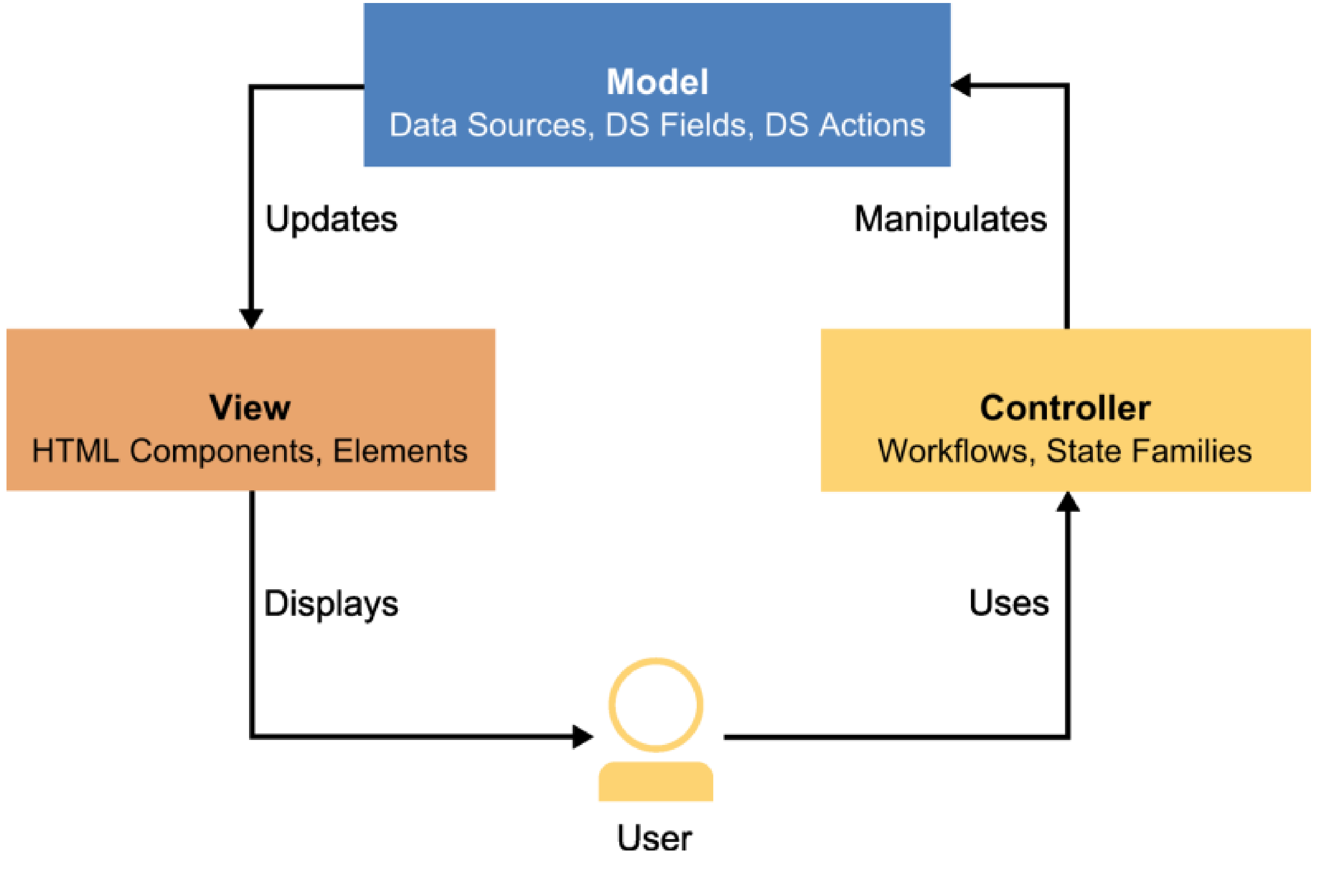
As you can see, while the UX framework tackles the key challenges in decoupling our classic framework into its separate MVC components, it also isn't meant to automatically "upgrade" our classic framework.

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
HTML Components, 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.*

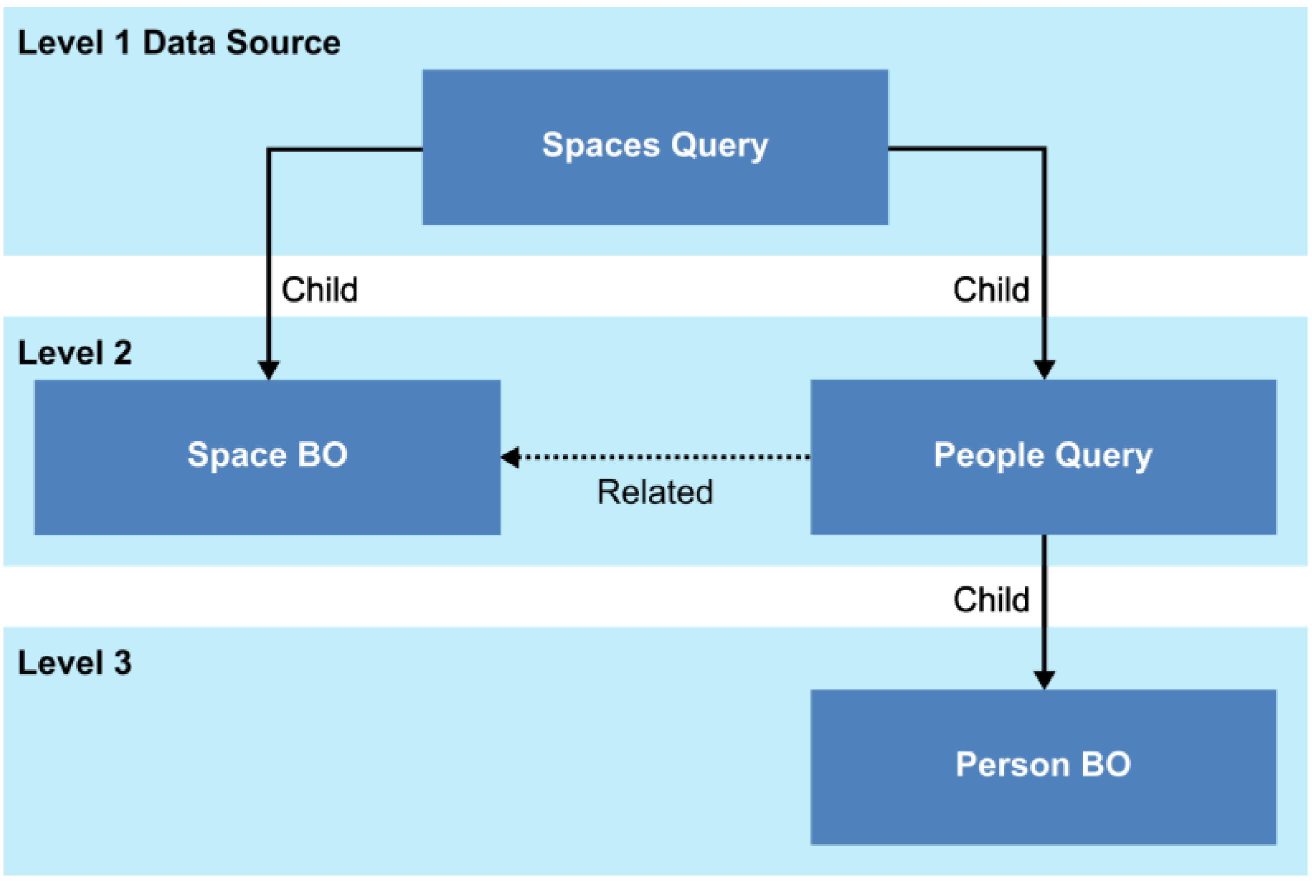
Before we look at some screenshots, here are some longer descriptions.

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. 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.
	Each data source must define at least one field. Each field corresponds to a field in the data

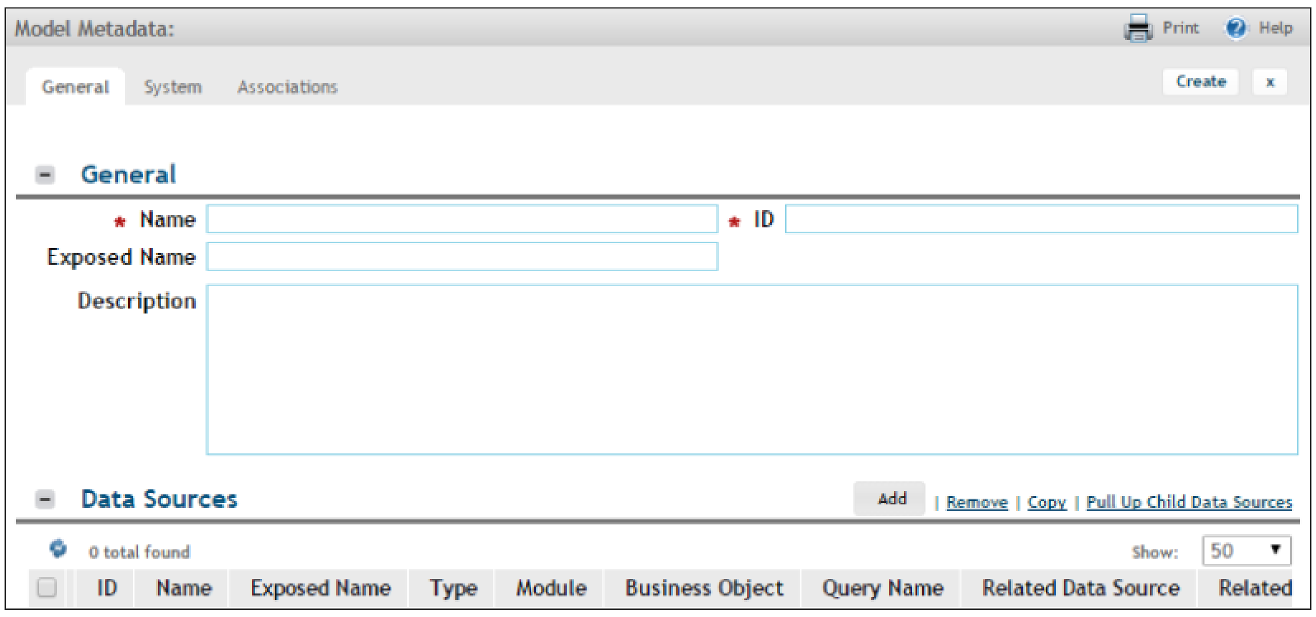
Data Source Fields	source type. To illustrate, let's say that you defined a data source with a Business Object type. Then each field in your data source references a corresponding field in the business object.
Data Source Actions	With data source actions, you define which business rules or workflow logic 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.



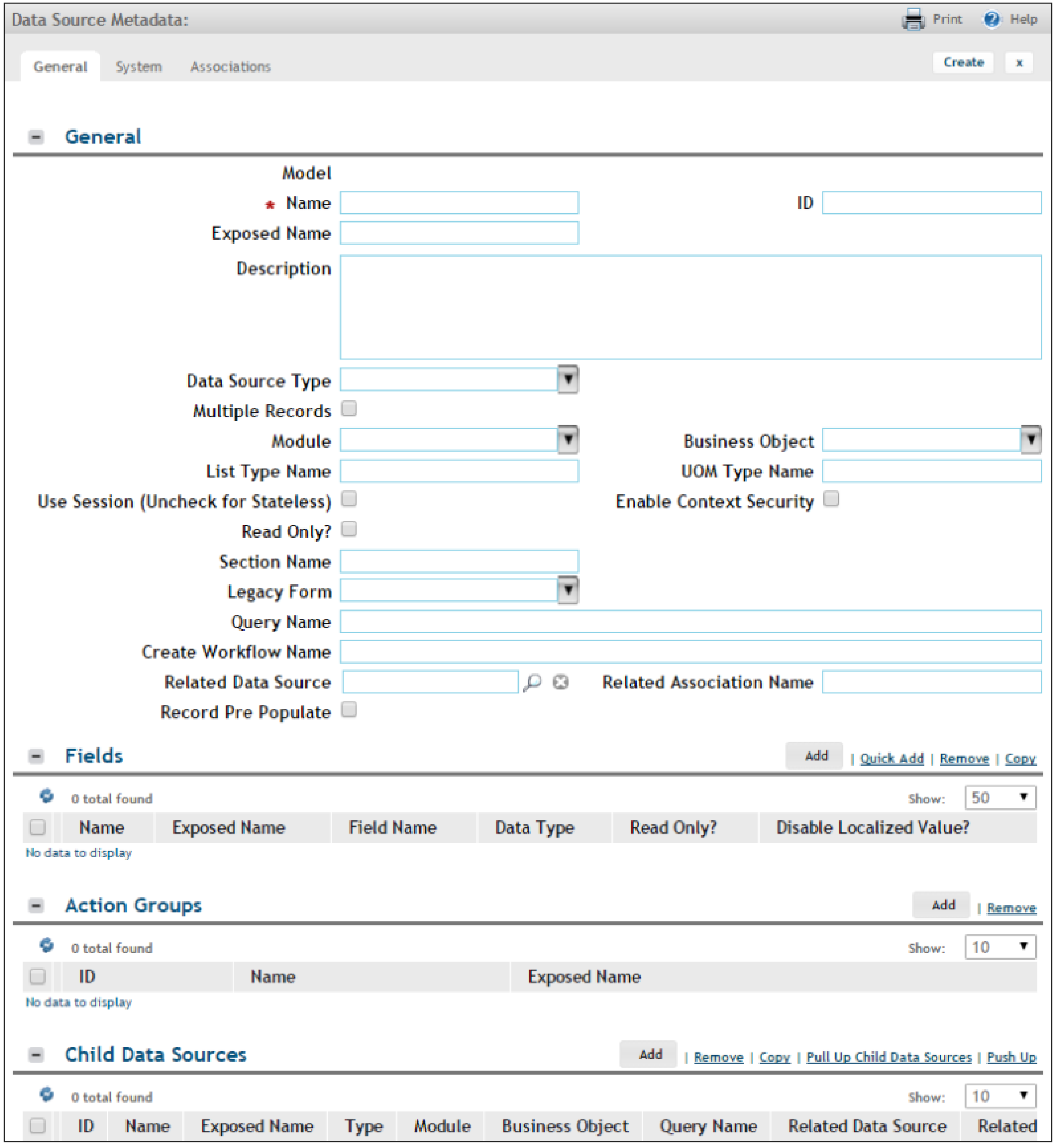
* Name		* ID	
Exposed Name			
Description			

0 total found   Show: 50

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

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.



Data Source Metadata:

Print | Help

General | System | Associations

Create | x

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

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 HTML files. In turn, each HTML file can be made up of the following components:

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

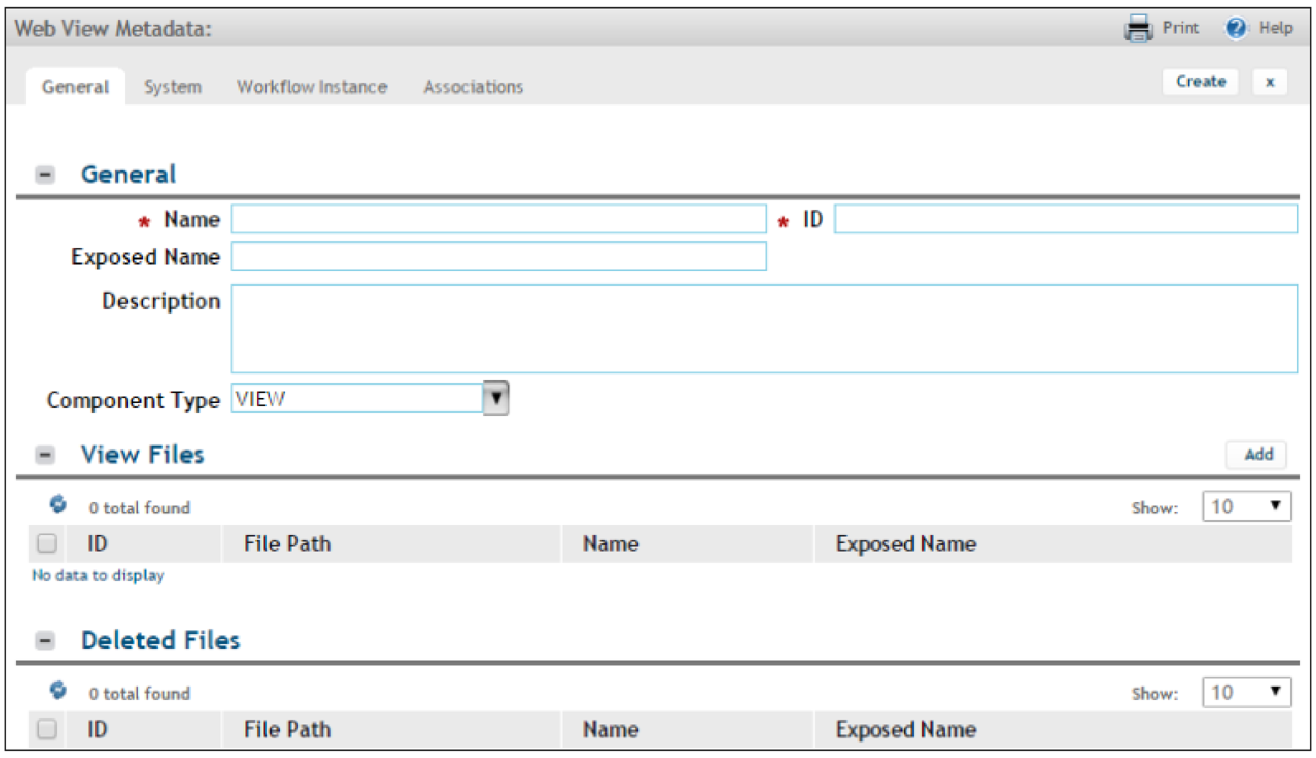
Before we peek at a few screenshots, here are some deeper descriptions.

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 *Gold*, *Neon*, and *Platinum* elements represent ecommerce, animation, offline, push, 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 HTML files. Later, we'll learn to add HTML files.

Web View Metadata.



* Name	
* ID	
Exposed Name	
Description	
Component Type	VIEW

View Files

0 total found

ID	File Path	Name	Exposed Name

No data to display

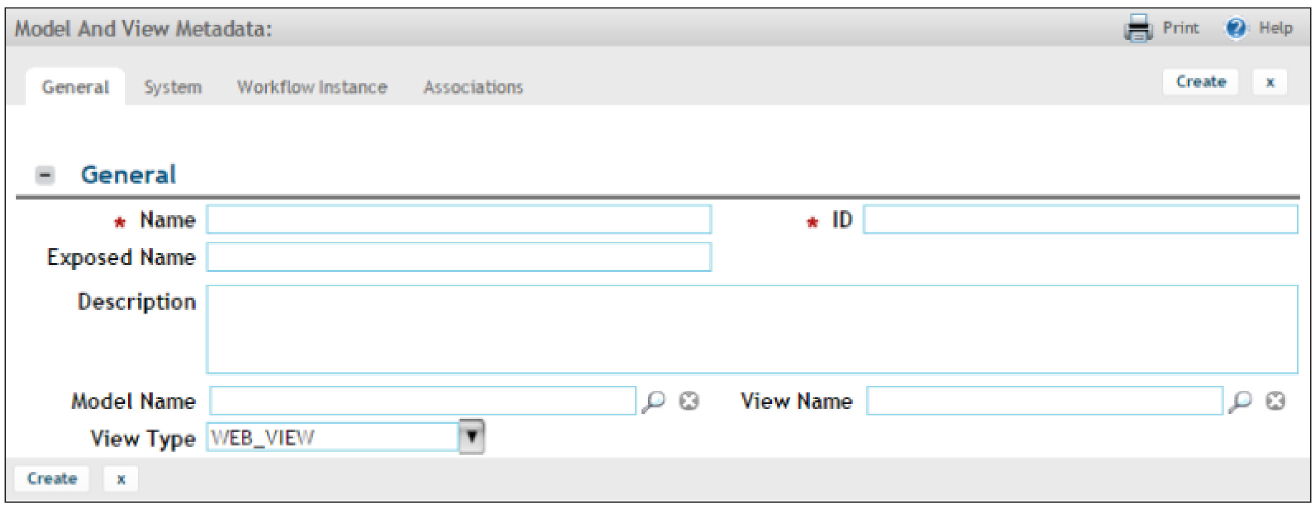
Deleted Files

0 total found

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.

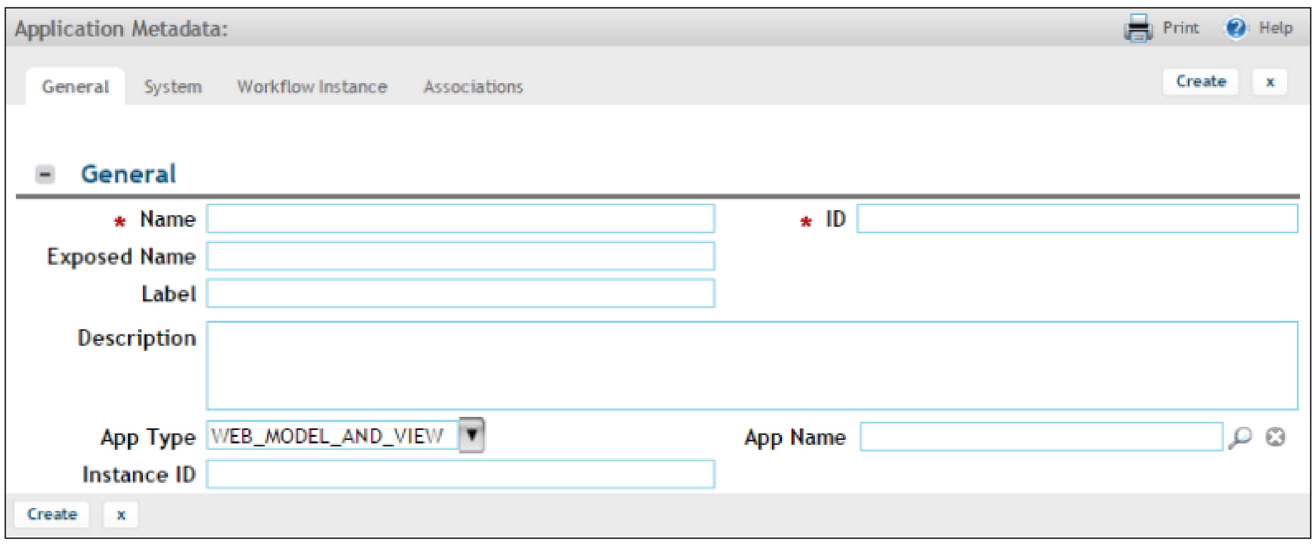
Model and View Metadata.



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

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.



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

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, and (3) defining and designing a **view** with a single HTML 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: Set up the view sync.

- 8: Add the HTML file for your view.
- 9: Access the application.

- Design your view.
  - 10: Start the view sync.
  - 11: Add a paragraph element to your HTML file.
  - 12: Add a few field elements to your HTML file.
  - 13: Add a few button elements to your HTML file.

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**

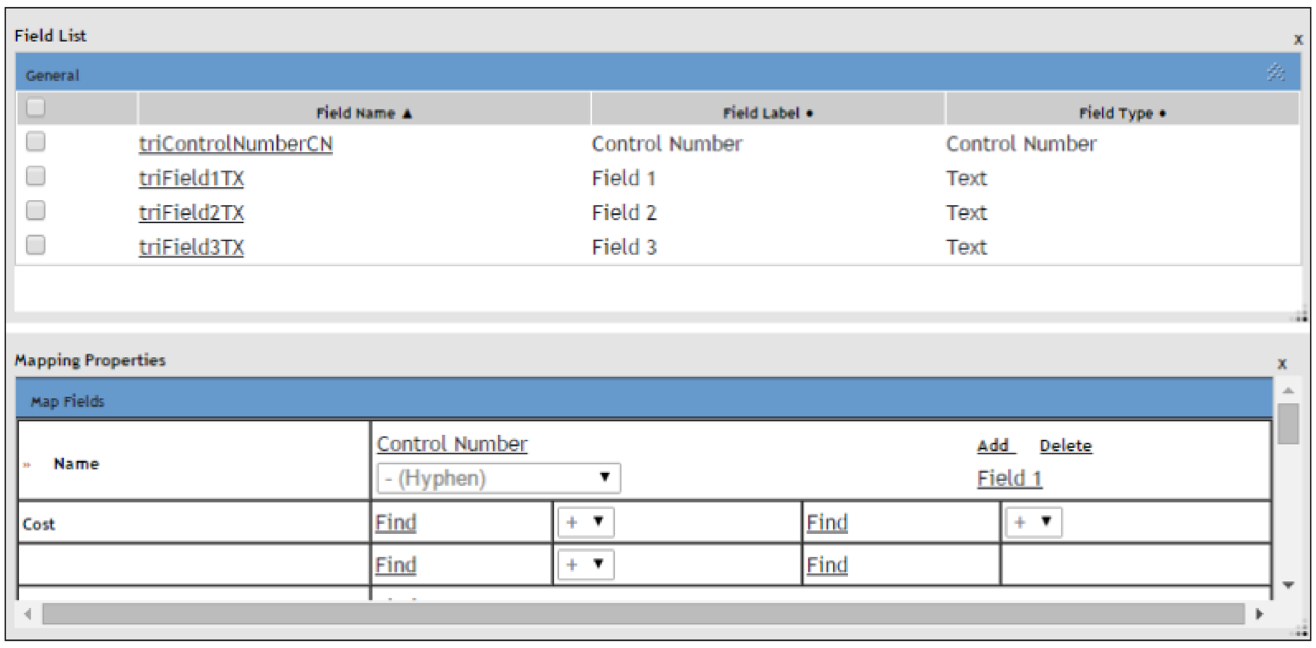
In **Step 7**, I'll ask you to contact your IBM TRIRIGA representative for the download location of the **WebViewSync** tool. So be prepared for that.

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

	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

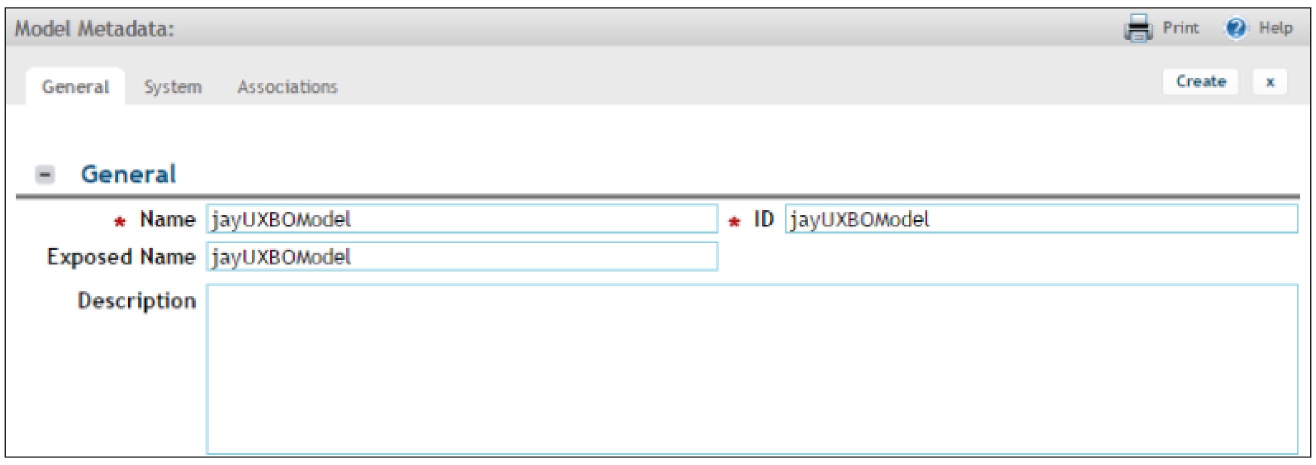
Map Fields

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



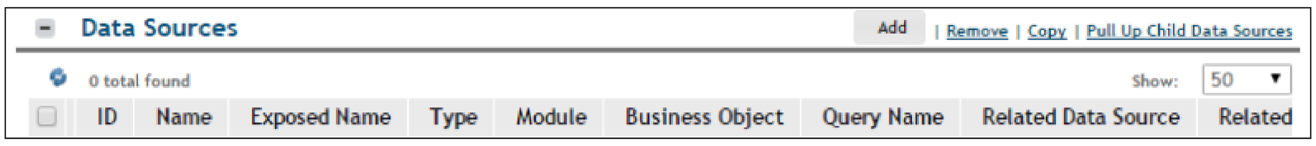
General

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.

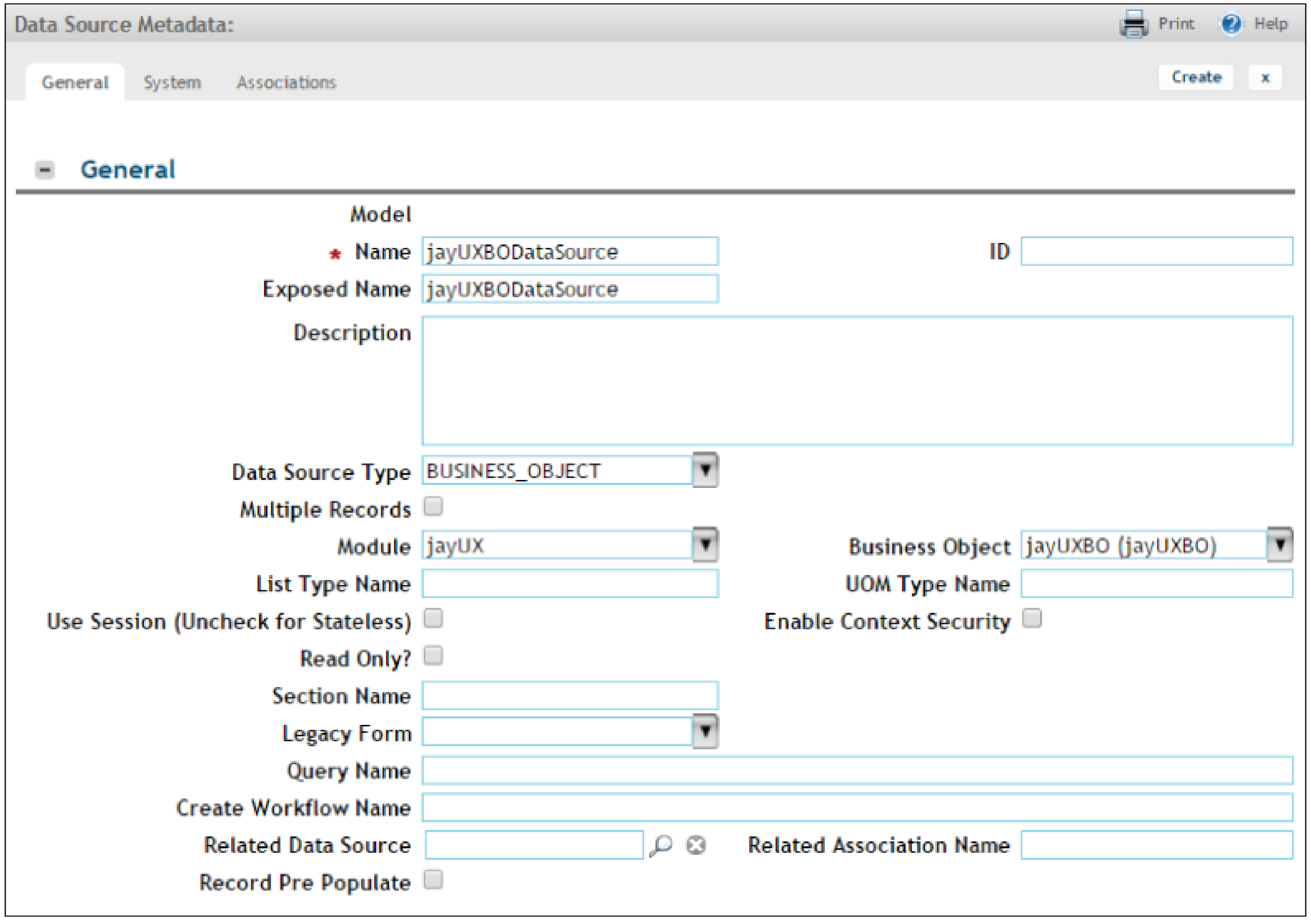


0 total found   Show: 50

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

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.


Data Source Metadata: Print Help

General System Associations Create x

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:

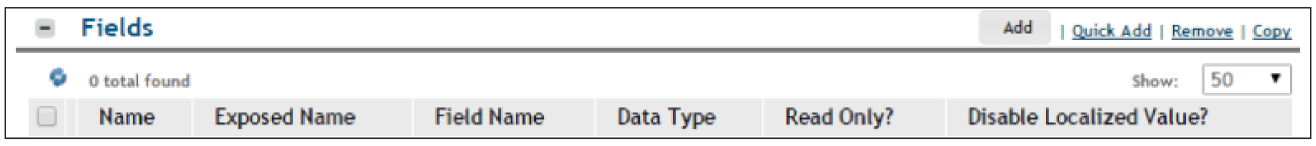
Related Data Source: — Related Association Name:

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.


Fields Add | Quick Add | Remove | Copy

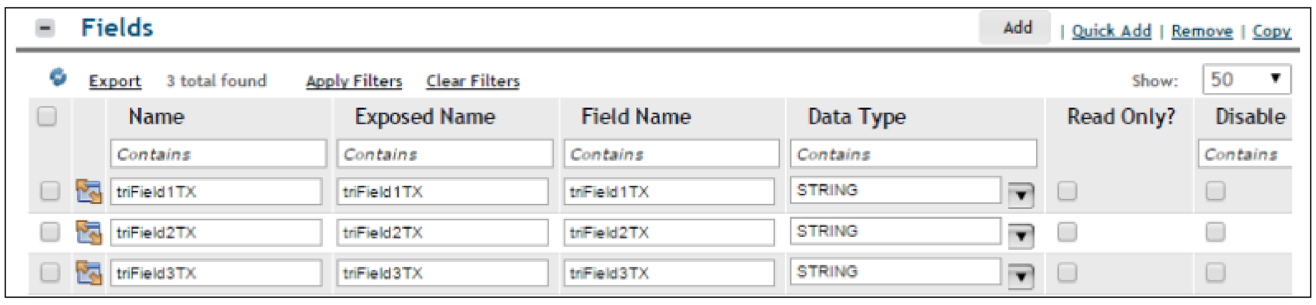
0 total found — Show: 50

	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.


Fields Add | Quick Add | Remove | Copy

Export 3 total found Apply Filters Clear Filters — Show: 50

		Name	Exposed Name	Field Name	Data Type	Read Only?	Disable
☐		Contains	Contains	Contains	Contains		Contains
☐		triField1TX	triField1TX	triField1TX	STRING	☐	☐
☐		triField2TX	triField2TX	triField2TX	STRING	☐	☐
☐		triField3TX	triField3TX	triField3TX	STRING	☐	☐

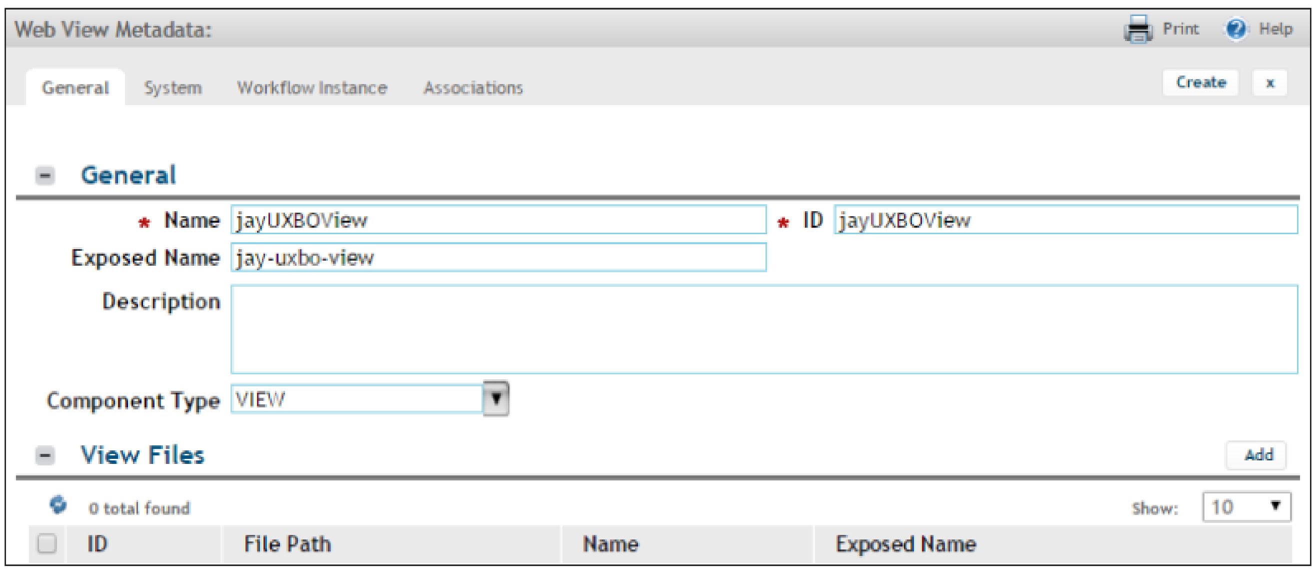
Finally, **Save & Close** your model components -- data source and model. Guess what? We're done with the first part. *You've defined your first model!* Ready to move on to the next part?

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. Click **Create**. Then **Save & Close**.

Web View Metadata.


Web View Metadata: Print Help

General System Workflow Instance Associations Create x

General

* Name: jayUXBOView — * ID: jayUXBOView

Exposed Name: jay-uxbo-view

Description:

Component Type: VIEW

View Files Add

0 total found — Show: 10

	ID	File Path	Name	Exposed Name

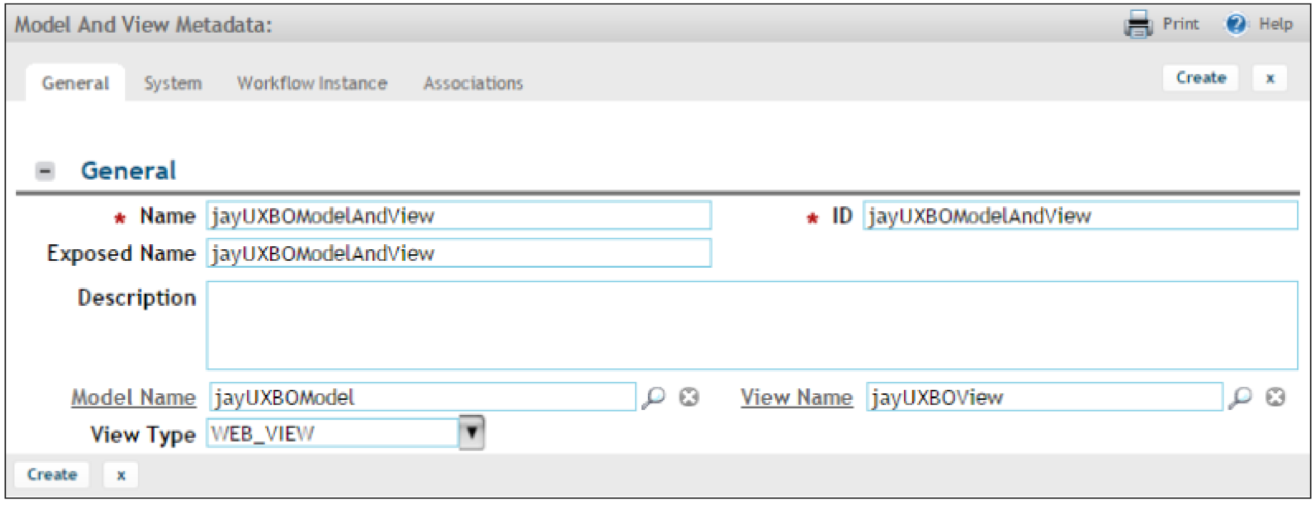
Why do we need a dash in the exposed name? *In Polymer, custom element names must always contain a dash (-).* This distinguishes custom elements from regular elements but also ensures forward compatibility when new tags are added to HTML. So, later in our example, when you design your HTML view, your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


Model And View Metadata:

General

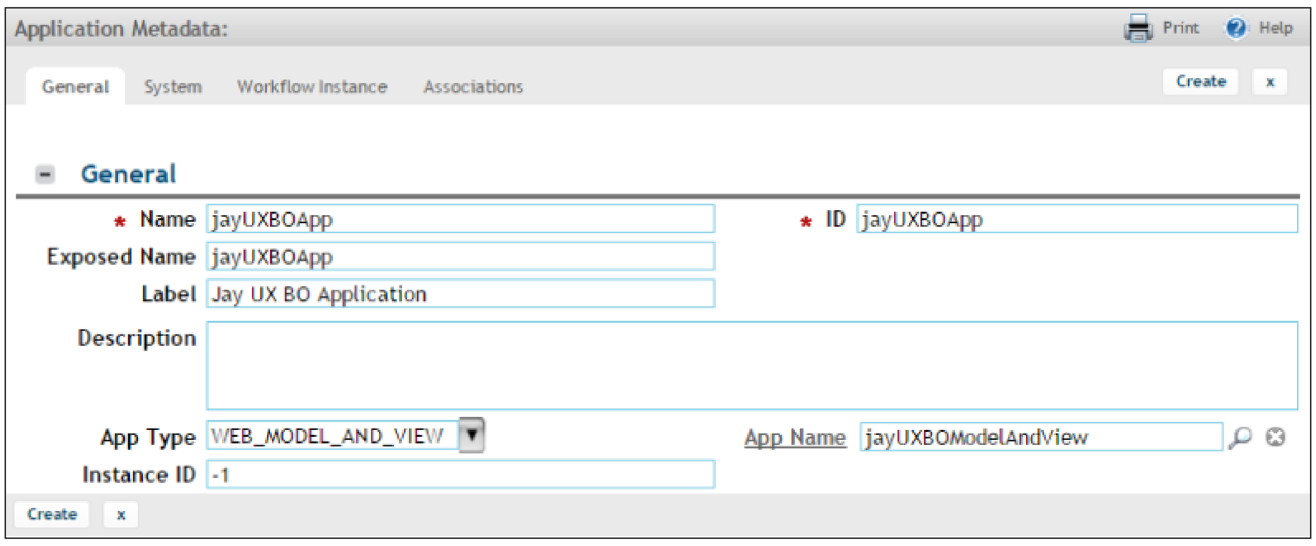
* Name	jayUXBOModelAndView	* ID	jayUXBOModelAndView
Exposed Name	jayUXBOModelAndView		
Description			
Model Name	jayUXBOModel	View Name	jayUXBOView
View Type	WEB_VIEW		

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.


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: Set up the view sync.

Contact your IBM TRIRIGA representative or business partner for the download location of the WebViewSync tool. This tool is used to populate the HTML files in your view metadata, and to automatically sync your HTML changes with the HTML files in your TRIRIGA environment.

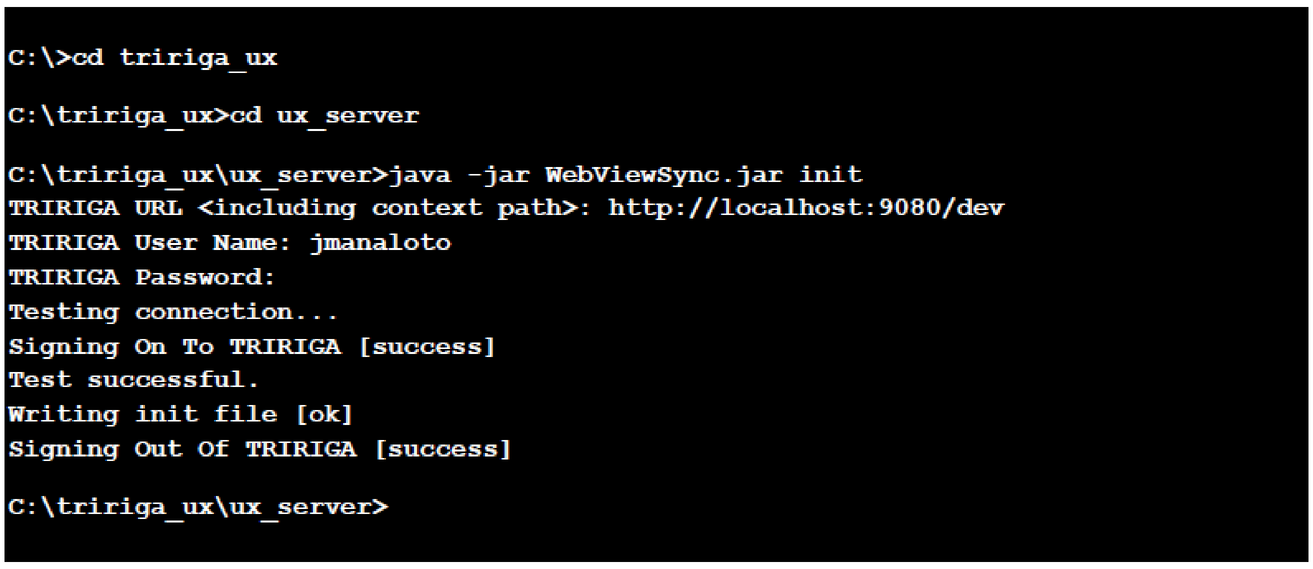
Download the file: **WebViewSync_[v].jar**, where **[v]** is the specific version of your TRIRIGA platform. For example, **WebViewSync_3.5.2.jar**. Make sure to identify your version in the command prompt. But for simplicity in our examples, I'll refer to the file as **WebViewSync.jar**.

Then save the file in a new or existing folder that's easy to identify. For our example, we'll save the file in this folder: **C:\tririga_ux\ux_server**.

Next, open the command prompt in the folder that you've selected. Run the following **init** command: **java -jar WebViewSync.jar init** and enter the URL, user name, and password for your TRIRIGA environment.

Your URL should include a valid FQDN or valid IP address. Be aware that typing your password will be hidden, so your cursor won't move. You need to run **init** only once, unless you change your environment details.

WebViewSync > Init.

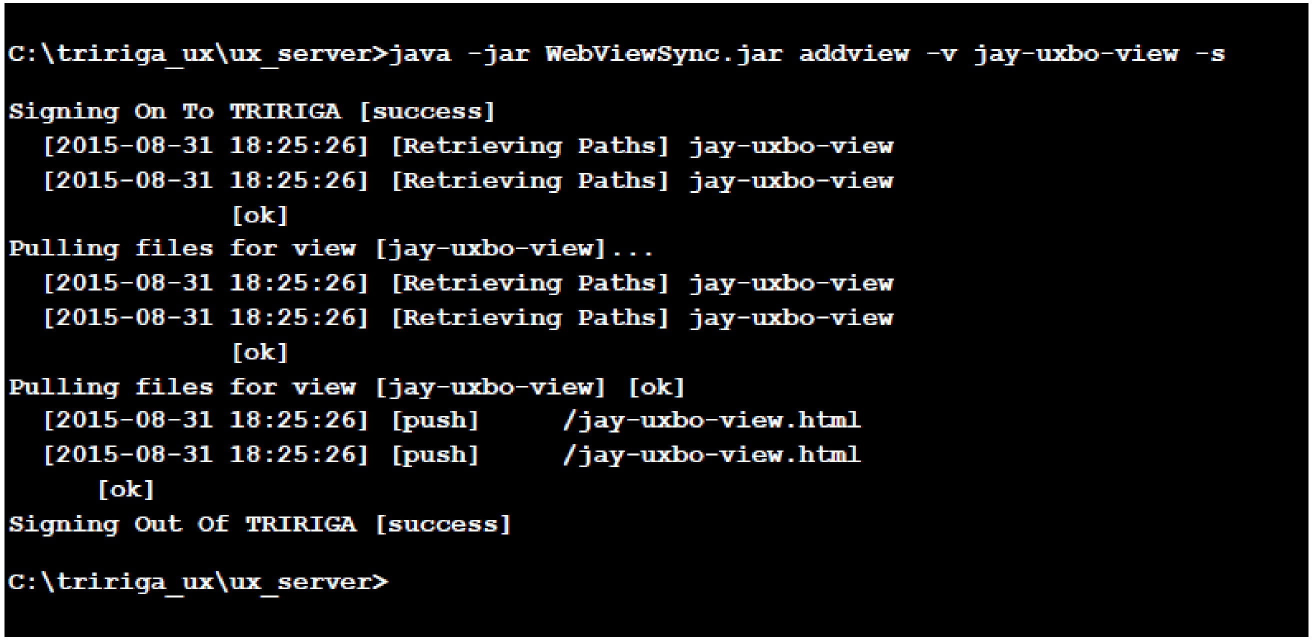

```
C:\>cd tririga_ux

C:\tririga_ux>cd ux_server

C:\tririga_ux\ux_server>java -jar WebViewSync.jar init
TRIRIGA URL <including context path>: http://localhost:9080/dev
TRIRIGA User Name: jmanaloto
TRIRIGA Password:
Testing connection...
Signing On To TRIRIGA [success]
Test successful.
Writing init file [ok]
Signing Out Of TRIRIGA [success]

C:\tririga_ux\ux_server>
```

Step 8: Add the HTML file for your view.

Next, it's time to add your HTML view file. If you remember, we skipped the View Files section of your view metadata. Now we'll auto-populate it.

To go ahead and add your HTML view file, run the following **addview** command: **java -jar WebViewSync.jar addview -v view-exposed-name -s** where **-s** generates the starter view file and **view-exposed-name** is the exposed name of your view (with the dash). We'll use **jay-uxbo-view**.

WebViewSync > Add View.

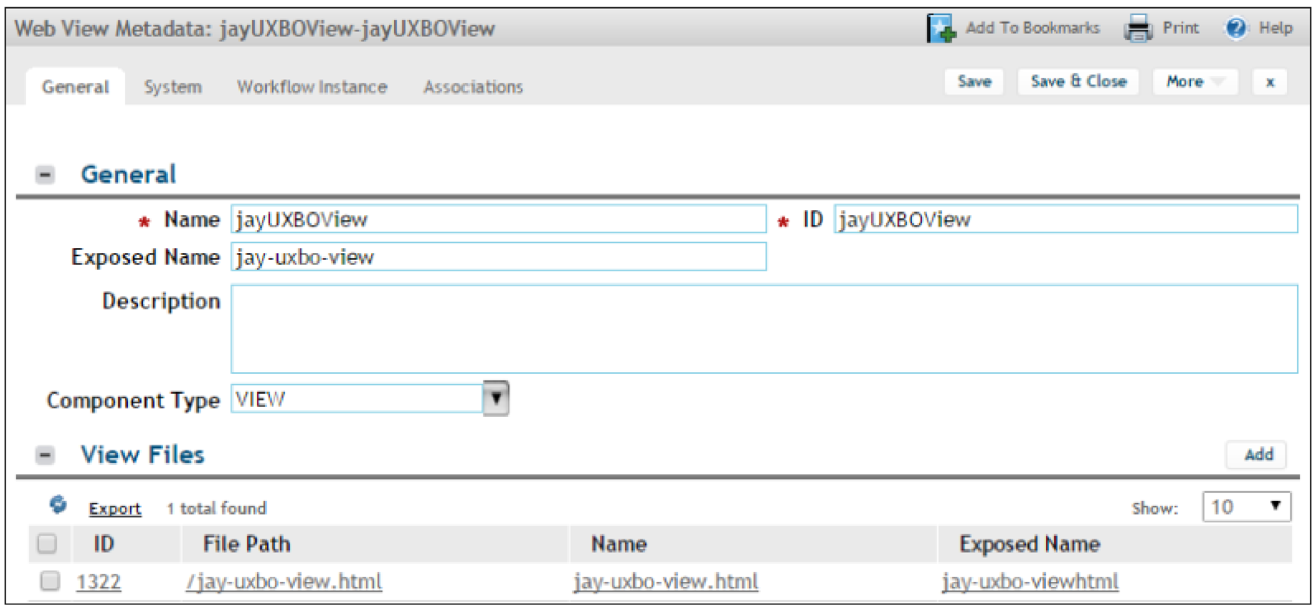


```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar addview -v jay-uxbo-view -s

Signing On To TRIRIGA [success]
  [2015-08-31 18:25:26] [Retrieving Paths] jay-uxbo-view
  [2015-08-31 18:25:26] [Retrieving Paths] jay-uxbo-view
             [ok]
Pulling files for view [jay-uxbo-view]...
  [2015-08-31 18:25:26] [Retrieving Paths] jay-uxbo-view
  [2015-08-31 18:25:26] [Retrieving Paths] jay-uxbo-view
             [ok]
Pulling files for view [jay-uxbo-view] [ok]
  [2015-08-31 18:25:26] [push]     /jay-uxbo-view.html
  [2015-08-31 18:25:26] [push]     /jay-uxbo-view.html
      [ok]
Signing Out Of TRIRIGA [success]

C:\tririga_ux\ux_server>
```

After the HTML file is added, you'll see that a new folder is created in the same folder where you saved the **WebViewSync.jar** file. In our example, the **C:\tririga_ux\ux_server** folder now contains the **jay-uxbo-view** folder, which now contains the **jay-uxbo-view.html** starter file that you added.

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.html** view file metadata.

Web View Metadata > View Files.



Web View Metadata: jayUXBOView-jayUXBOView

General

* Name: jayUXBOView
* ID: jayUXBOView
Exposed Name: jay-uxbo-view
Description:
Component Type: VIEW

View Files

1 total found   Show: 10

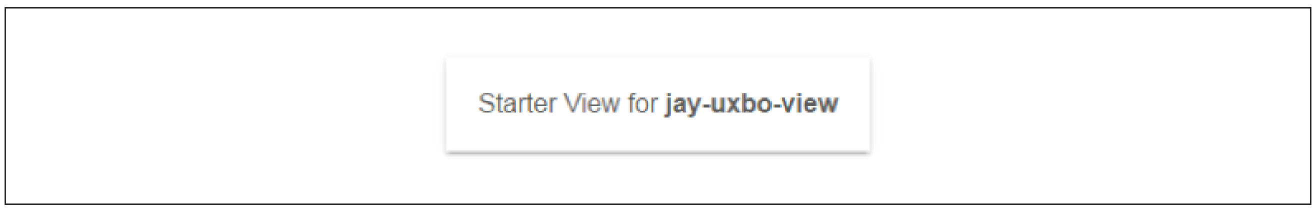
ID	File Path	Name	Exposed Name
1322	/jay-uxbo-view.html	jay-uxbo-view.html	jay-uxbo-viewhtml

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://localhost:9080/dev/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**

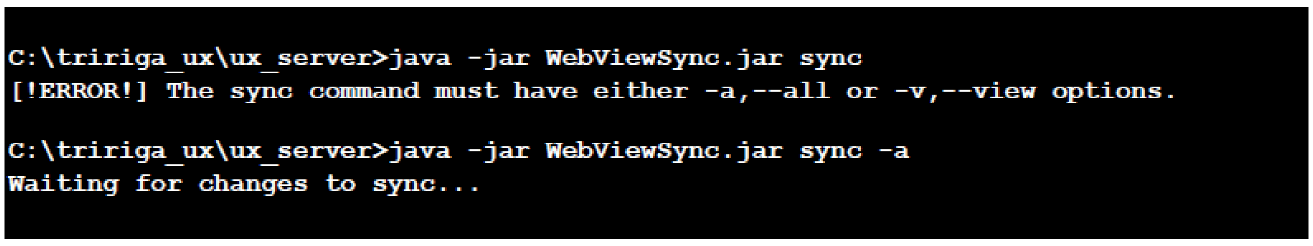
Design your view.

Step 10: Start the view sync.

To start the "listening" process so your HTML changes are pushed automatically into your TRIRIGA environment, return to the command prompt in the same folder as before. Run the following **sync** command: **java -jar WebViewSync.jar sync -a** where **-a** listens to all views.

When you see the message "Waiting for changes to sync...", it's time to design your view! But be careful not to close the command prompt.

WebViewSync > Sync.



```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar sync
[!ERROR!] The sync command must have either -a,--all or -v,--view options.

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

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

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

First, add the **<link>** tag at the top to import the TRIRIGA **triplat-ds** (data source) component: **<link rel="import" href="../triplat-ds/triplat-ds.html">**

HTML File > Import triplat-ds.

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

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

Next, add the **<triplat-ds>** tag to declare the TRIRIGA **triplat-ds** (data source) component: **<triplat-ds id="model" 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>**

HTML File > Declare triplat-ds.

```
        <template>

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

                <div class="layout horizontal center-justified">
                        <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>
        </template>
```

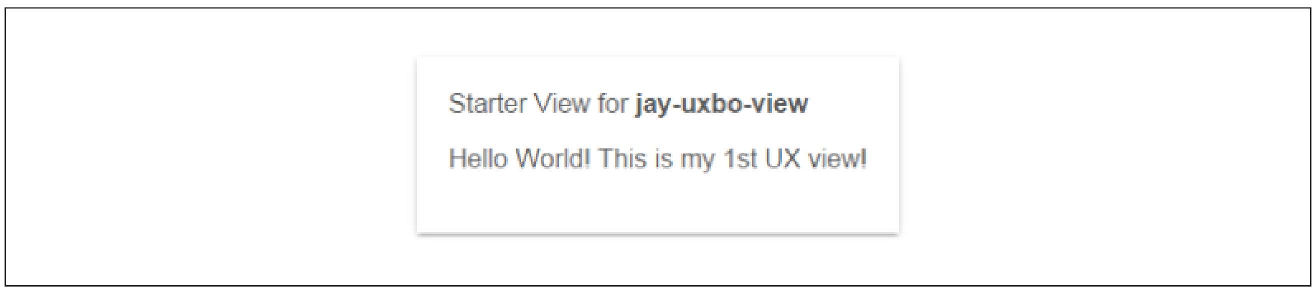
When you save the file, return to the command prompt. You'll see that the changed **jay-uxbo-view.html** is "pushed" into your environment.

WebViewSync > Push.

```
C:\tririga_ux\ux_server>java -jar WebViewSync.jar sync -a
Waiting for changes to sync...
Signing On To TRIRIGA [success]
  [2015-08-31 18:56:28] [push]     /jay-uxbo-view.html
  [2015-08-31 18:56:28] [push]     /jay-uxbo-view.html
     [ok]
```

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

UX App > Refresh 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 HTML 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 **sync** command is running in the command prompt.

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

HTML File > Import paper-input.

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

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

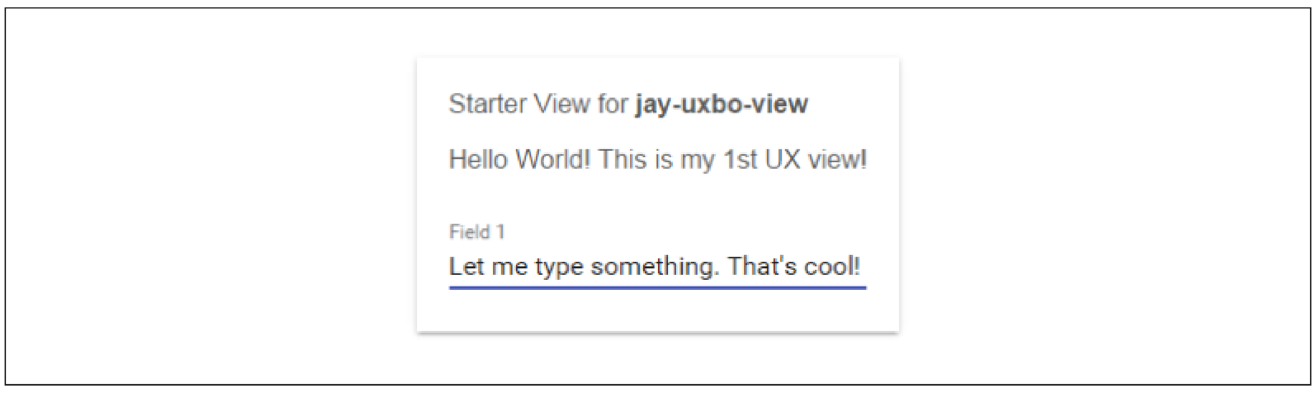
Next, add the **<paper-input>** tag to declare the element: **<paper-input label="Field 1" floating-label value="{{data.triField1TX}}"></paper-input>**

HTML File > Declare paper-input.

```
    <template>

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

        <div class="layout horizontal center-justified">
            <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>
    </template>
```

Save the file and refresh 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 > Refresh Starter View.


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

Hello World! This is my 1st UX view!

Field 1
Let me type something. That's cool!

Step 13: Add a few button elements to your HTML 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 **sync** command is running in the command prompt.

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

HTML File > Import paper-button.

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

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

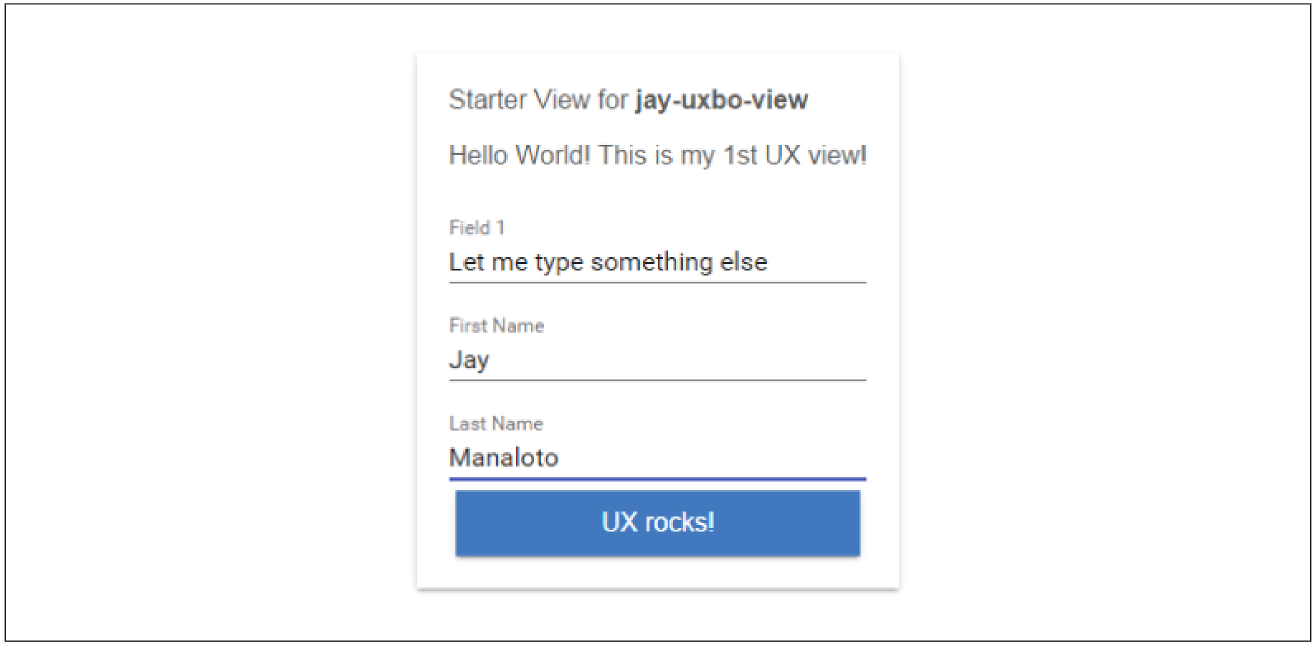
Next, add the **<paper-button>** tag to declare the Polymer element: **<paper-button raised>UX rocks!</paper-button>** where **raised** adds a shadow.

HTML File > Declare paper-button.

```
            <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>
```

Save the file and refresh 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 > Refresh 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
Manaloto

UX rocks!

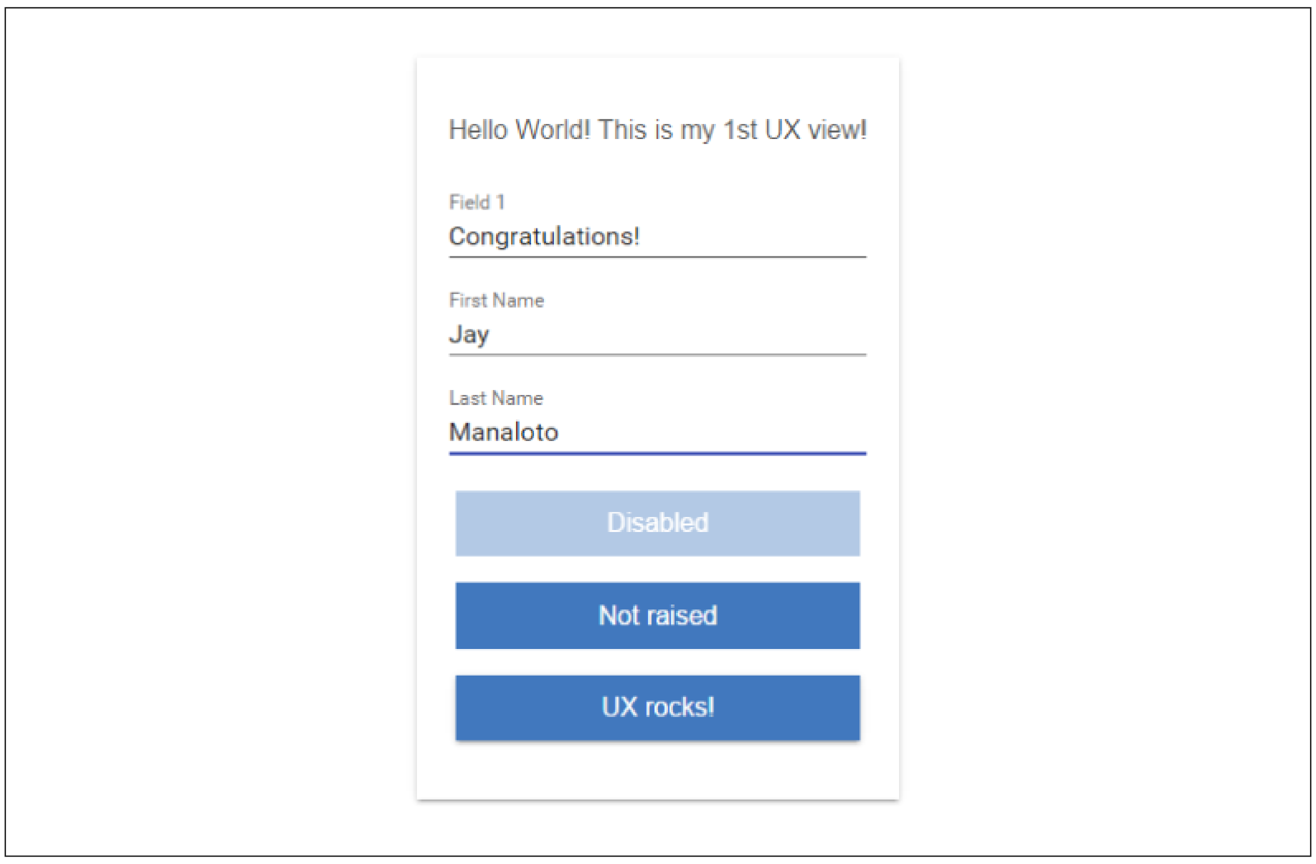
With some creativity, you can try other Polymer **<paper-button>** attributes.

HTML File > Declare paper-button.

```
                    <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>

                        <p><paper-button disabled>Disabled</paper-button>
                        <p><paper-button>Not raised</paper-button>
                        <p><paper-button raised>UX rocks!</paper-button>

                    </paper-material>
```

Guess what? We're done. As planned, we built a simple 3-field 3-button application. *Congratulations! You've built your first UX application!*

UX App > Refresh Starter View.


Hello World! This is my 1st UX view!

Field 1
Congratulations!

First Name
Jay

Last Name
Manaloto

Still want more?

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

Background Q & A.

Question	Answer
What will the new MVC model look like?	You might be wondering if it's enough to reuse business objects from Data Modeler as the model. Unfortunately, business objects are not flexible and would require complex data models to render a rich view. Instead, the model could contain many data sources such as business objects, queries, and even integration data. In addition, the model could also contain "actions" that the user can perform via the controller. These actions could be imported from business objects and queries.
What will the new MVC view look like?	The view could support many types of views such as a form view, mobile form view, and maybe even an API view and custom view. The form view could be rendered from a flexible hierarchy of form-layout components that are sourced by the model. For example, a single field type could be included in several

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

Next >