

Building UX Web Applications

Starting in IBM TRIRIGA Application Platform 3.8.0, the UX Framework was further enhanced to support any standard Web Application. With these enhancements, new **UX Web Applications** can be developed by using any available web framework. For example, ReactJS, VueJS, Angular, and Vanilla HTML with JS and CSS.

Contents

- [I. Introduction to UX Web Applications](#)
- [II. Changes to UX Designers](#)
- [III. Enabling CORS on the Application Server](#)
- [IV. TRIRIGA React Components Library](#)
- [V. Globalization of UX Web Applications](#)
- [VI. Deploying and Pulling the UX Web Application](#)
- [VII. Accessing the URL for the UX Web Application](#)
- [VIII. Building the UX Web Application \(Tutorial\)](#)

I. Introduction to UX Web Applications

The IBM TRIRIGA UX Framework was originally designed to support only UX apps that were implemented by using **Google Polymer**. When the UX Framework and the first UX apps were originally released, only Polymer Version 1 was available. Later, the UX Framework was enhanced to support Polymer Version 3 as well.

The UX Framework adopted a simplified architecture for developing web applications. All of the web components are provided by the TRIRIGA platform, so the developer only needs to implement web components. The developer does not need to worry about creating an **index.html** file or bootstrapping an application from scratch.

Although this approach simplified development, it also introduced some limitations:

- It supports the development of applications that use Polymer **only**.
- It does **not** support the addition of third-party dependencies that use standard package managers like **npm**.
- It does **not** support module bundlers such as Webpack, Rollup or Parcel.
- Its use of IBM Carbon components does **not** fully support Polymer.

To overcome these limitations, the UX Framework was further enhanced to support any standard Web Application. UX Web Applications are supported by TRIRIGA Platform 3.8.0 and later.

The main features of a TRIRIGA UX Web Application are:

- It is a standard web application that can be developed by using any available web framework. For example, ReactJS, VueJS, Angular, Vanilla HTML with JS and CSS, etc.
- Its application architecture is under full control by the developer. Any third-party dependency can be added by using the package manager of choice.
- It interacts with the TRIRIGA server by using the REST APIs. There is also a **tririga-react-components** library that the app can use to interact with TRIRIGA APIs.

Moving forward, new TRIRIGA-provided **Perceptive Apps** will be based on UX Web Applications that are developed with **ReactJS** and **IBM Carbon** components. However, TRIRIGA clients and partners are free to develop their own UX Web Applications with any technology of their choice.

II. Changes to UX Designers

The UX Web Application is a change to the **Web View** part of the UX Framework. All of the other UX Framework concepts like the Model, Data Source, Model-and-View, and Application remain the same.

To support UX Web Applications, the following changes were made to the **Web View Designer**:

- Added the **WEB_APP** value to the **View Type** field: This defines the view as a Web Application.
- Added the **Root Folder** field: The name of the folder that contains the build files of the Web Application. The files requested from the browser are served from this folder.
- Added the **Messages Folder** field: The name of the folder that contains all of the message-related JSON files of the Web Application. When exporting the dictionary from the Globalization Manager, TRIRIGA will look for dictionary labels in the JSON files that are inside this folder.
- Added the **Index filename** field: The name of the file that is served when the Web Application is opened. It is also used to support the HTML5 History API fallback, where the server returns the index file whenever an application requests a file path that does not exist.

General	
* Name	My First App
* ID	My First App
Exposed Name	my-first-app
View Type	WEB_APP
Description	
Custom App Configuration	
Root Folder	build
Index filename	index.html
Messages Folder	/src/utils/messages

III. Enabling CORS on the Application Server

When you run a UX Web Application locally, it will make requests to the APIs on the TRIRIGA server (e.g., fetch data from a data source or to get a floor plan). By default, the application server (Liberty or WAS) restricts calling the TRIRIGA server from a domain outside the domain from which the first resource was served. This means that an application running on the **localhost** domain will not be allowed to call an API on the TRIRIGA server.

Fortunately, a mechanism that is named **Cross-Origin Resource Sharing** (CORS) allows restricted resources to be requested from another domain. You must enable CORS on the application server that runs your development TRIRIGA server, before you can run a UX Web Application locally.

To enable CORS on a Liberty server, edit the **server.xml** file and add the following line:

```
<cors allowCredentials="true" allowedHeaders="*"
      allowedMethods="GET, POST, PUT, DELETE, HEAD" allowedOrigins="*"
      domain="/" exposeHeaders="triWebContextId"/>
```

For more information about how to configure CORS on WebSphere Application Server, see:

<https://www.ibm.com/support/pages/node/6348518>

IV. TRIRIGA React Components Library

The **tririga-react-components** library is a collection of React components and JavaScript code to help with the development of TRIRIGA UX Web Applications.

This library is intended for TRIRIGA UX Web Applications that are developed with **ReactJS**. It is automatically installed when you create your application by using the **@tririga/cra-template**. To manually install it into your project, run the following command in your terminal:

```
npm install -S @tririga/tririga-react-components
```

For more information about the **tririga-react-components** library, including documentation of all components, classes, and objects, see: <https://tririga.github.io/tririga-react-components>

V. Globalization of UX Web Applications

1. Exporting the Dictionary Labels

The **Globalization Manager** exports labels from both UX Polymer and UX Web Applications. The Globalization Manager does this by exporting the dictionary as **XLIFF** files of unique translatable text. If a specific label appears in multiple views and UX Web Applications, only one unique entry will appear in the dictionary.

The difference between UX Polymer and UX Web Applications in exporting labels is:

- **UX Polymer Applications:** The TRIRIGA platform parses all HTML and JS files from UX Polymer views by looking for static translatable labels. Conditions for translatable text can be found in the "[UX in Globalization Tool](#)" document.
- **UX Web Applications:** The TRIRIGA platform looks for all JSON files inside the **Messages Folder** of your UX Web Applications. The path of the **Messages Folder** is defined in the Web View Metadata record of your UX Web Applications. Each JSON file contains a list of key-and-value entries. The key is used by the application to reference the label. The value is the label to be translated. Only the value part of each entry is exported to the XLIFF files.

General	
* Name	My First App
* ID	My First App
Exposed Name	my-first-app
View Type	WEB_APP
Description	
Custom App Configuration	
Root Folder	build
Index filename	index.html
Messages Folder	/src/utils/messages

For example, in the **messages.json** file below, the following labels will be exported by the Globalization Manager:

- Welcome to the UX Web Application Home Page
- Current user details
- You do not have permission to access this page.
- Due to either a session timeout or unauthorized access, you do not have permission to access this page.

```
src > utils > messages > {} messages.json > ...
1   {
2     · "HOME_HEADER": "Welcome to the UX Web Application Home Page",
3     · "CURRENT_HEADER": "Current user details",
4     · "UNAUTHORIZED_TITLE": "You do not have permission to access this page.",
5     · "UNAUTHORIZED_DESCRIPTION": "Due to either a session timeout or unauthorized
6     · access, you do not have permission to access this page."
   }
```

2. Getting the Translated Labels

The difference between UX Polymer and UX Web Applications in getting translated labels is:

- **UX Polymer Applications:** The TRIRIGA Platform detects the language that is defined for the user. Then it automatically translates all static translatable labels that are defined inside the HTML and JS files that are requested by the browser. It does this by using the translations that are imported into the Globalization Manager dictionary. So, the browser receives the file with the labels already translated.
- **UX Web Applications:** The application calls the dictionary API to get the translated labels. The **tririga-react-components** library exports the **getTranslatedMessages** method to help the application to interact with the dictionary API. For more information about the **getTranslatedMessages** method, including an example, see: <https://tririga.github.io/tririga-react-components/?path=/story/javascript-library-tridictionary-gettranslatedmessages--page>

VI. Deploying and Pulling the UX Web Application

1. Deploying the UX Web Application

Before you deploy the UX Web Application, you must first build it by using the following command:

```
npm run build
```

To deploy the UX Web Application to a TRIRIGA server, use the **tri-deploy** tool:

- **Installation:** `npm install @tririga/tri-deploy -g`
- **Usage:** `tri-deploy -t http://tririga.dev:8001/dev -u myUserName -p myPassword -v my-first-app -d /my-first-app -w`
- **Documentation:** <https://www.npmjs.com/package/@tririga/tri-deploy>

2. Pulling the UX Web Application

To pull the UX Web Application from a TRIRIGA server, use the **tri-pull** tool:

- **Installation:** `npm install @tririga/tri-pull -g`
- **Usage:** `tri-pull -t http://tririga.dev:8001/dev -u myUserName -p myPassword -v my-first-app -w`
- **Documentation:** <https://www.npmjs.com/package/@tririga/tri-pull>

VII. Accessing the URL for the UX Web Application

The difference between UX Polymer and UX Web Applications in accessing the application URL is:

- **UX Polymer Applications:** `https://[hostname:port]/[context_path]/p/web/[yourApp]`
- **UX Web Applications:** `https://[hostname:port]/[context_path]/app/[yourApp]`
- Where **[hostname:port]** and **[/context_path]** are the specific values for your IBM TRIRIGA Application Platform environment, and **[yourApp]** is the exposed name of your application.

VIII. Building the UX Web Application (Tutorial)

In this tutorial, we will guide you in building a UX Web Application with **ReactJS** and **IBM Carbon** components.

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>

If you do **not** have **Node.js** or **npm** installed on your computer, then access <https://nodejs.org> to download and install Node.js (npm is installed as part of the Node.js installation).

To check if you have **Node.js** installed, run this command in your terminal:

```
node -v
```

To confirm that you have **npm** installed, you can run this command in your terminal:

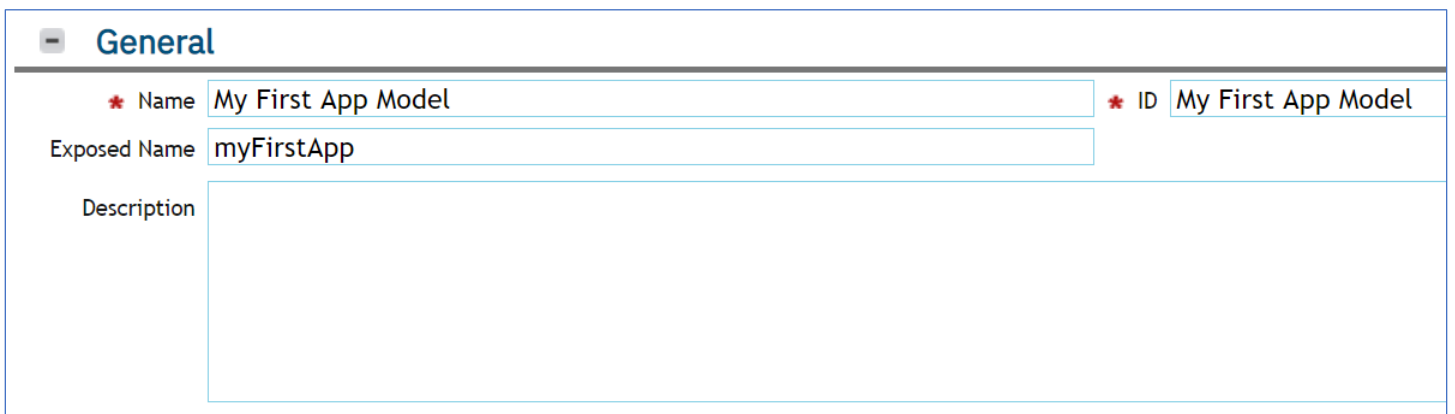
```
npm -v
```

Step 1. Add the Model

From the side navigation menu, select **Tools**. In the portal, select **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 enter the following information:

- **Name:** My First App Model
- **Exposed Name:** myFirstApp
- **ID:** My First App Model

Then click **Create**.



The screenshot shows a form titled "General" with the following fields:

- Name:** My First App Model
- ID:** My First App Model
- Exposed Name:** myFirstApp
- Description:** (empty text area)

Step 2. Add the Data Source and Fields for “Current User”

Next, in the Data Sources section of your model, click **Add**. Enter the following information:

- **Name:** Current User
- **Exposed Name:** currentUser
- **Data Source Type:** CURRENT_USER

Next, in the Fields section of your data source, click **Quick Add** for two fields. Enter the following information:

- **Name:** (1) triFirstNameTX, (2) triLastNameTX
- **Exposed Name:** (1) firstName, (2) lastName
- **Field Name:** (1) triFirstNameTX, (2) triLastNameTX
- **Data Type:** (1) STRING, (2) STRING

Then click **Create**. Finally, **Save & Close** your model.

General

Model: My First App 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

Export 2 total found Apply Filters Clear Filters

	Name	Exposed Name	Field Name	Data Type
<input type="checkbox"/>	Contains	Contains	Contains	Contains
<input type="checkbox"/>	triFirstNameTX	firstName	triFirstNameTX	STRING ▼
<input type="checkbox"/>	triLastNameTX	lastName	triLastNameTX	STRING ▼

Step 3. Add the View

From the side navigation menu, select **Tools**. In the portal, select **Web View Designer**. Click **Add**. Enter the name, exposed name, and ID of your view. For our example, we'll enter the following information:

- **Name:** My First App
- **Exposed Name:** my-first-app
- **ID:** My First App
- **View Type:** WEB_APP
- **Root Folder:** build
- **Messages Folder:** /src/utills/messages
- **Index filename:** index.html

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

General	
* Name	My First App
* ID	My First App
Exposed Name	my-first-app
View Type	WEB_APP
Description	
Custom App Configuration	
Root Folder	build
Index filename	index.html
Messages Folder	/src/utills/messages

Step 4. Add the Model-and-View

From the side navigation menu, select **Tools**. In the portal, select **Model and View Designer**. Click **Add**. Enter the name, exposed name, and ID of your model-and-view. For our example, we'll enter the following information:

- **Name:** My First App Model and View
- **Exposed Name:** myFirstApp
- **ID:** My First App Model and View
- **Model Name:** Select "My First App Model" that you created earlier.
- **View Name:** Select "My First App" that you created earlier.
- **View Type:** Select WEB_VIEW.

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

General	
* Name	My First App Model and View
* ID	My First App Model and View
Exposed Name	myFirstApp
Description	
Model Name	My First App Model
View Name	My First App
View Type	WEB_VIEW

Step 5. Add the Application

From the side navigation menu, select **Tools**. In the portal, select **Application Designer**. Click **Add**. Enter the name, exposed name, and ID of your application. For our example, we'll enter the following information:

- **Name:** My First App
- **Exposed Name:** myFirstApp
- **ID:** My First App
- **Label:** My first app
- **App Type:** Select WEB_MODEL_AND_VIEW.
- **App Name:** Select "My First App Model and View" that you created earlier.
- **Instance ID:** -1

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

General

Name	My First App	ID	My First App
Exposed Name	myFirstApp		
Label	My first app		
Description			
App Type	WEB_MODEL_AND_VIEW	App Name	My First App Model and View
Instance ID	-1		

Step 6. Create the Skeleton Application from a Template

A TRIRIGA template can be used with the NPM-based **create-react-app** tool to build a "skeleton" UX Web Application. This is the best way to start building a UX Web Application with **ReactJS** and **IBM Carbon** components.

To create a new UX Web Application from this template, run this command in your terminal:

```
npm create-react-app my-first-app --template @tririga/cra-template --use-npm
```


Step 7. Run the Application Locally

Now let's run the UX Web Application that you created. In the **my-first-app** folder that was created, make a copy of the **.env.development.local.template** file and rename the copied file to **.env.development.local**. Open the **.env.development.local** file and set the values for the following variables:

- **REACT_APP_INSTANCE_ID**: Instance ID from the application metadata
- **REACT_APP_TRIRIGA_URL**: URL of the TRIRIGA server
- **REACT_APP_CONTEXT_PATH**: TRIRIGA context path
- **REACT_APP_MODEL_AND_VIEW**: Exposed name of the model-and-view
- **REACT_APP_BASE_PATH**: Context path when running the app on the local development server
- **REACT_APP_EXPOSED_NAME**: Exposed name of the application
- **REACT_APP_SSO**: If SSO is enabled on the server, then true; otherwise, false

For our example, we'll enter the following information, where **REACT_APP_TRIRIGA_URL** and **REACT_APP_CONTEXT_PATH** are the specific values for your TRIRIGA environment:

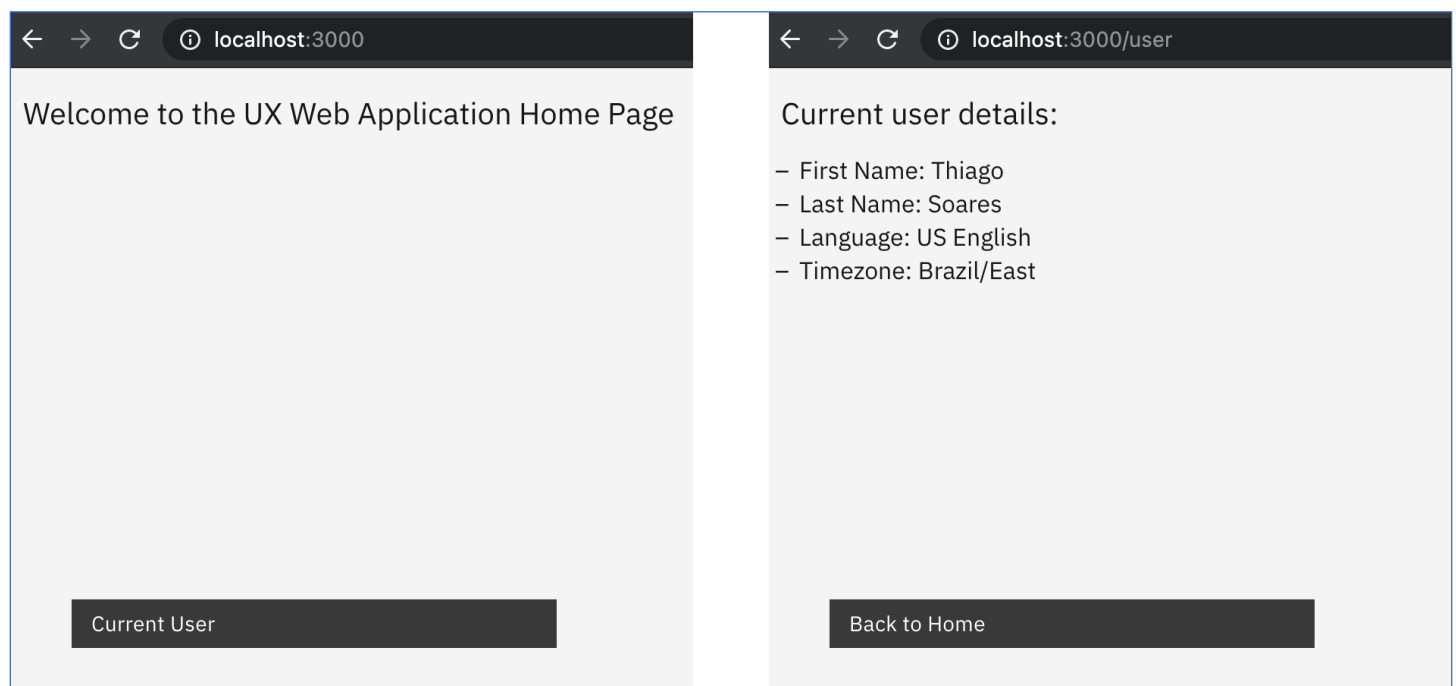
```
REACT_APP_INSTANCE_ID=-1
REACT_APP_TRIRIGA_URL=https://www.tririga-dev.com/dev
REACT_APP_CONTEXT_PATH=/dev
REACT_APP_MODEL_AND_VIEW=myFirstApp
REACT_APP_BASE_PATH=/
REACT_APP_EXPOSED_NAME=myFirstApp
REACT_APP_SSO=false
```

After you've entered the information, run this command in your terminal:

```
cd my-first-app
npm start
```

This command automatically opens your app home page in a new tab of your default browser. By default, the URL of an app that runs locally is: <https://localhost:3000>. To check if your app displays some information about your user, click the **Current User** button.

Congratulations! You've built your first TRIRIGA UX Web Application with **ReactJS** components. Now, let's create a new page to display all of the buildings.



Step 8. Add the Data Source and Fields for “All Buildings”

First, let’s add a data source to query all of the buildings in TRIRIGA.

From the side navigation menu, select **Tools**. In the portal, select **Model Designer**. Select the **My First App Model**.

Next, in the Data Sources section of your model, click **Add**. Enter the following information:

- **Name:** All Buildings
- **Exposed Name:** allBuildings
- **Data Source Type:** QUERY
- **Multiple Records:** Yes
- **Module:** Location
- **Business Object:** Building (triBuilding)
- **Query Name:** triBuilding - Building Details

Next, in the Fields section of your data source, click **Quick Add** for two fields. Enter the following information:

- **Name:** (1) Building, (2) Parent Property
- **Exposed Name:** (1) building, (2) parentProperty
- **Field Name:** (1) Building, (2) Parent Property
- **Data Type:** (1) STRING, (2) STRING

Then click **Create**. Finally, **Save & Close** your model.

- General

Model My First App 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

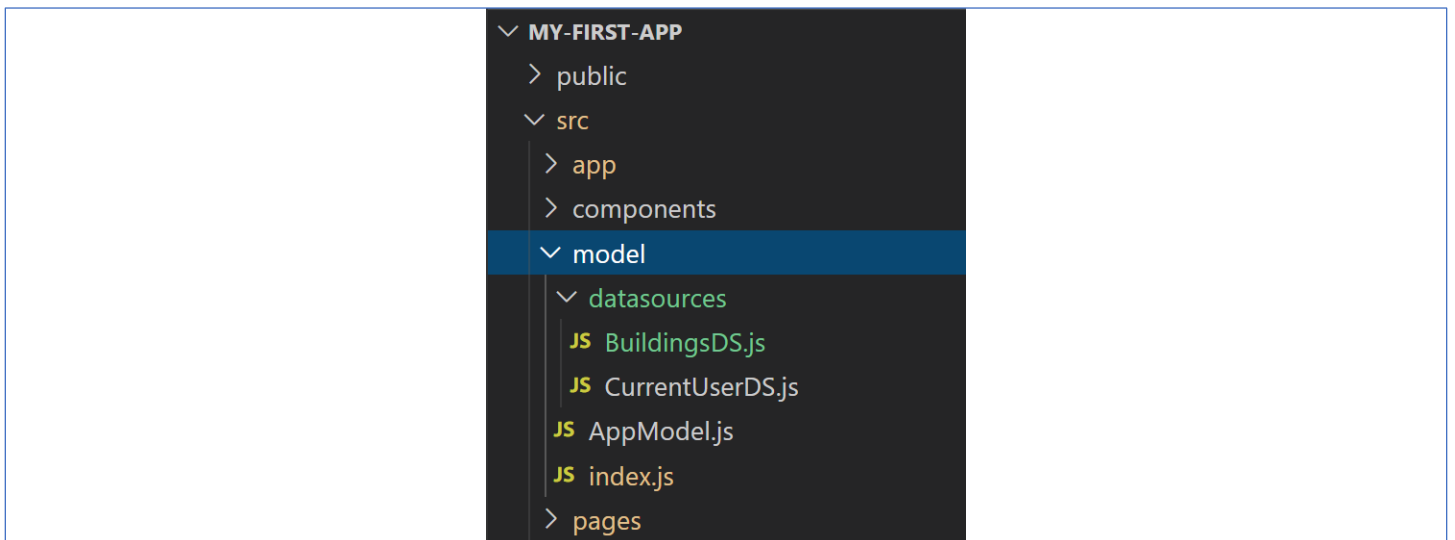
- Fields

🔄 Export 2 total found Apply Filters Clear Filters

	Name	Exposed Name	Field Name	Data Type
<input type="checkbox"/>	<input type="text" value="Contains"/>	<input type="text" value="Contains"/>	<input type="text" value="Contains"/>	<input type="text" value="Contains"/>
<input type="checkbox"/>	<input type="text" value="Building"/>	<input type="text" value="building"/>	<input type="text" value="Building"/>	<input type="text" value="STRING"/> ▼
<input type="checkbox"/>	<input type="text" value="Parent Property"/>	<input type="text" value="parentProperty"/>	<input type="text" value="Parent Property"/>	<input type="text" value="STRING"/> ▼

Step 9. Add the Method to Interact with the Data Source

Open the **my-first-app** folder by using the integrated development environment (IDE) of your choice. For our example, the following screenshots are taken from Microsoft® Visual Studio Code. The code that interacts with the data sources are placed under: `/src/model/datasources`. Under that folder, create a new file that is named **BuildingsDS.js**.



Open the **BuildingsDS.js** file and add the following code:

```
import { getAppModel } from "../AppModel";
import { DatasourceNames } from "../../utils";

export async function getAllBuildings() {
  const response = await getAppModel().getRecord(
    DatasourceNames.BUILDINGS_DS_NAME
  );
  return response.data;
}
```

Basically, this code gets the model object and then calls the **getRecord** method to get the data from the **BUILDINGS_DS_NAME** data source.

Next, let's create the **BUILDINGS_DS_NAME** constant and set its value to **allBuildings**, the exposed name of the "All Buildings" data source. Under the `/src/Utils/constants` folder, open the **DatasourceNames.js** file and add the new **const** (constant) as follows:

```
export const CURRENT_USER_DS_NAME = "currentUser";
export const BUILDINGS_DS_NAME = "allBuildings";
```

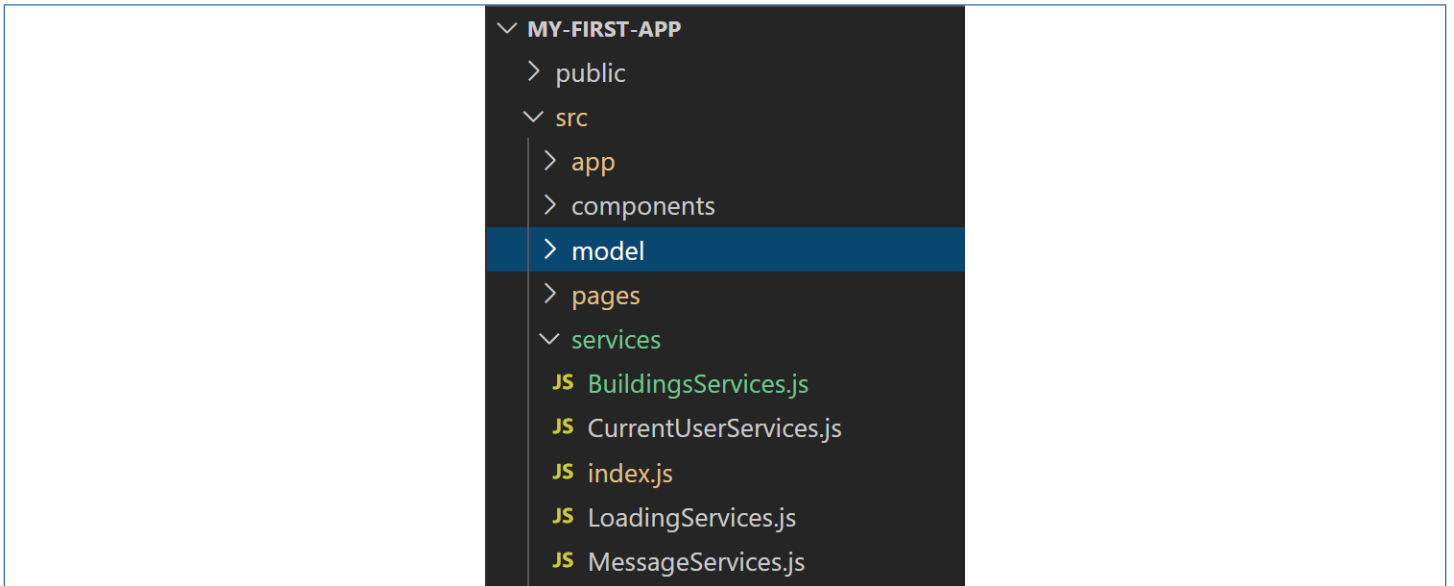
Now, let's export the **BuildingsDS** from the model **index.js** file. Under the `/src/model` folder, open the **index.js** file and add the export of the **BuildingsDS** as follows:

```
import { createAppModel, getAppModel } from "./AppModel";
import * as CurrentUserDS from "./datasources/CurrentUserDS";
import * as BuildingsDS from "./datasources/BuildingsDS";

export { createAppModel, getAppModel, CurrentUserDS, BuildingsDS };
```

Step 10. Add the Service to Get "All Buildings"

A **service** encapsulates the logic to run the application actions. These actions are usually triggered by a user or event. The services are placed under: `/src/services`. Under that folder, create a new file that is named **BuildingsServices.js**.



Open the **BuildingsServices.js** file and add the following code:

```
import { LoadingServices } from ".";
import { BuildingsDS } from "../model";

export async function getAllBuildings() {
  let buildings = [];
  try {
    LoadingServices.setLoading("getAllBuildings", true);
    buildings = await BuildingsDS.getAllBuildings();
  } finally {
    LoadingServices.setLoading("getAllBuildings", false);
  }
  return buildings;
}
```

This code is a simple action that toggles the loading while calling the **BuildingsDS** to get all buildings.

Now, let's export the **BuildingsServices** from the services **index.js** file. Under the `/src/services` folder, open the **index.js** file and add the export of the **BuildingsServices** as follows:

```
export * as LoadingServices from "./LoadingServices";
export * as CurrentUserServices from "./CurrentUserServices";
export * as MessageServices from "./MessageServices";
export * as BuildingsServices from "./BuildingsServices";
```

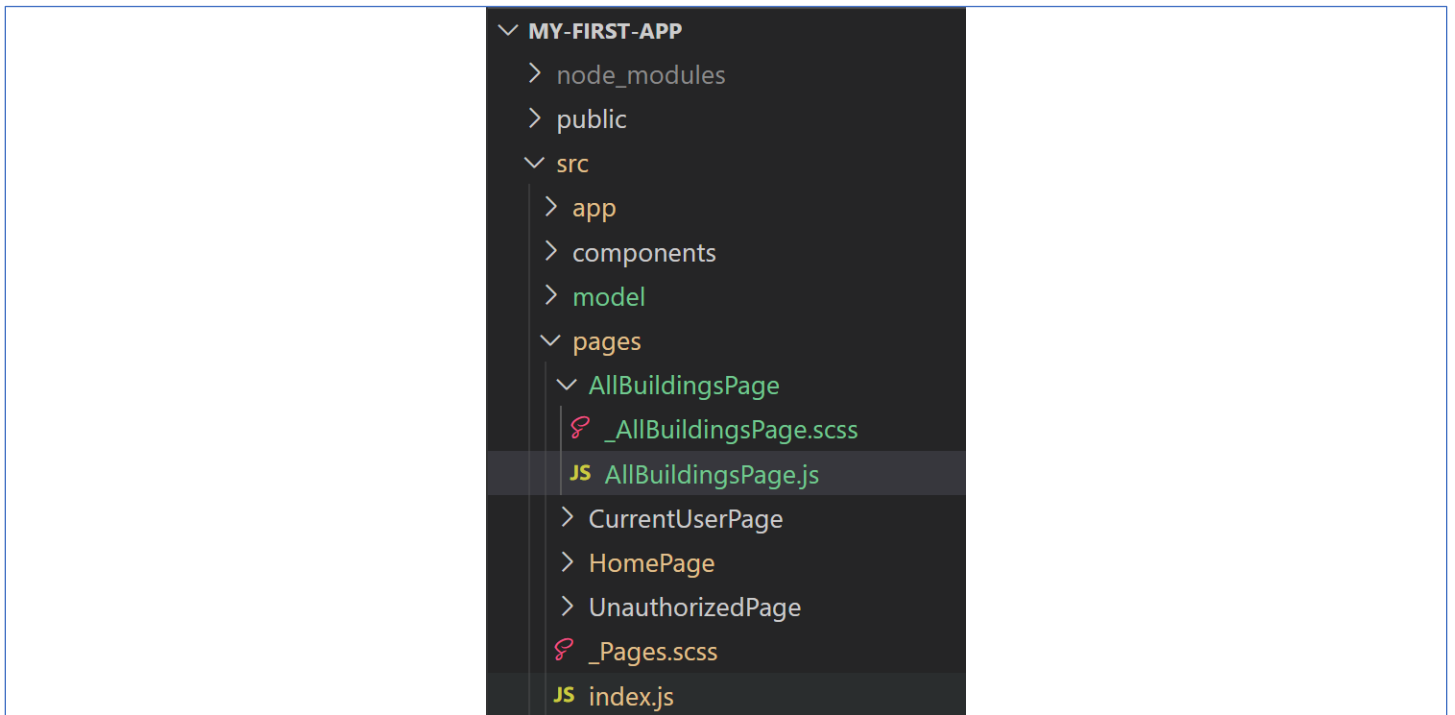
Step 11. Add the Page to Display “All Buildings”

Next, let’s create a new page where we will use an **IBM Carbon** component that is named **DataTable** to display all buildings. For more information about the **DataTable** component, including an example, see:

<https://react.carbondesignsystem.com/?path=/story/datatable--usage>

The pages are placed under: `/src/pages`. Under that folder, create a new folder that is named **AllBuildingsPage**. Under this new **AllBuildingsPage** folder, create two new files as follows:

- **_AllBuildingsPage.scss**: Partial Sass file that contains the styles that are used by the page
- **AllBuildingsPage.js**: JavaScript file that contains the component that renders the page



Open the **_AllBuildingsPage.scss** file and add the following code:

```
.allBuildingsPage {
  @include page;

  &__content {
    display: flex;
    flex-direction: column;
  }

  & .#{$prefix}--data-table-container {
    height: 100%;
    display: flex;
    flex-direction: column;
  }
}
```

Open the **AllBuildingsPage.js** file and add the following code:

```
import React from "react";
import {
  DataTable,
  TableContainer,
  Table,
  TableHead,
  TableRow,
  TableHeader,
  TableBody,
  TableCell,
} from "carbon-components-react";
import { Routes, AppMsg } from "../../utils";
import { FooterButtons } from "../../components";
import { BuildingsServices } from "../../services";

const cssBase = "allBuildingsPage";
```

```

export default class AllBuildingsPage extends React.PureComponent {
  state = {
    buildings: [],
  };

  async loadBuildings() {
    const buildings = await BuildingsServices.getAllBuildings();
    this.setState({ buildings });
  }

  componentDidMount() {
    this.loadBuildings();
  }

  render() {
    const { buildings } = this.state;
    buildings.forEach((item) => {
      item.id = item._id;
    });
    const headers = [
      {
        key: "building",
        header: AppMsg.getMessage(AppMsg.MESSAGES.NAME),
      },
      {
        key: "parentProperty",
        header: AppMsg.getMessage(AppMsg.MESSAGES.PARENT_PROPERTY),
      },
    ];
    return (
      <div className={cssBase}>
        <div className={` ${cssBase}__content`}>
          <DataTable rows={buildings} headers={headers}>
            ({
              rows,
              headers,
              getHeaderProps,
              getRowProps,
              getTableProps,
              getTableContainerProps,
            }) => (
              <TableContainer
                title={AppMsg.getMessage(AppMsg.MESSAGES.BUILDINGS)}
                description={AppMsg.getMessage(
                  AppMsg.ALL_BUILDINGS_DESCRIPTION
                )}
                {...getTableContainerProps()}
              >
                <Table {...getTableProps()} isSortable>
                  <TableHead>
                    <TableRow>
                      {headers.map((header) => (
                        <TableHeader
                          key={header.key}
                          {...getHeaderProps({ header })}
                          isSortable
                        >
                          {header.header}
                        </TableHeader>
                      ))}
                    </TableRow>
                  </TableHead>
                  <TableBody>
                    {rows.map((row) => (
                      <TableRow key={row.id} {...getRowProps({ row })}>
                        {row.cells.map((cell) => (
                          <TableCell key={cell.id}>{cell.value}</TableCell>
                        ))}
                      </TableRow>
                    ))}
                  </TableBody>
                </Table>
              </TableContainer>
            )}
          </DataTable>
        </div>
        <FooterButtons
          secondaryLabel={AppMsg.getMessage(AppMsg.BUTTONS.HOME)}
          secondaryRoute={Routes.HOME}
        />
      </div>
    );
  }
}

```

Basically, this code loads all buildings by calling the **BuildingsServices.getAllBuildings** method when the component is mounted, and then saves the list of buildings to the component state. The **render** method uses the IBM Carbon **DataTable** component to render a table with all buildings. Meanwhile, all of the labels that are displayed by the page are retrieved by using the **AppMsg.getMessage** method.

Now, let's import the **_AllBuildingsPage.scss** file into the **_Pages.scss** file. Under the `/src/pages` folder, open the **_Pages.scss** file and add the import of the **_AllBuildingsPage.scss** file as follows:

```
@import "../HomePage/HomePage";
@import "../CurrentUserPage/CurrentUserPage";
@import "../UnauthorizedPage/AuthorizedPage";
@import "../AllBuildingsPage/AllBuildingsPage"; // add this line
```

Next, let's export the **AllBuildingsPage** from the pages **index.js** file. Under the `/src/pages` folder, open the **index.js** file and add the export of the **AllBuildingsPage** as follows:

```
import HomePage from "../HomePage/HomePage";
import CurrentUserPage from "../CurrentUserPage/CurrentUserPage";
import UnauthorizedPage from "../UnauthorizedPage/UnauthorizedPage";
import AllBuildingsPage from "../AllBuildingsPage/AllBuildingsPage";

export { HomePage, CurrentUserPage, UnauthorizedPage, AllBuildingsPage };
```

When you use an **IBM Carbon** component, you must also import the styles that are used by the component. Under the `/src` folder, open the **index.scss** file and add the import of the **data-table.scss** file as follows:

```
//Carbon components styles
@import "carbon-components/scss/components/data-table/data-table";
@import "carbon-components/scss/components/list/list";
@import "carbon-components/scss/components/loading/loading";
@import "carbon-components/scss/components/modal/modal";
@import "carbon-components/scss/components/notification/toast-notification";
```

Step 12. Add the Labels for the Page

All of the labels that are used by the application are defined in the JSON files that are inside the **Messages Folder**. Under the `/src/utlis/messages` folder, open the **messages.json** file and add the following labels:

- **BUILDINGS:** Buildings
- **ALL_BUILDINGS_DESCRIPTION:** A list of all buildings in TRIRIGA
- **NAME:** Name
- **PARENT_PROPERTY:** Parent Property

```
{
  "HOME_HEADER": "Welcome to the UX Web Application Home Page",
  "CURRENT_HEADER": "Current user details",
  "UNAUTHORIZED_TITLE": "You do not have permission to access this page.",
  "UNAUTHORIZED_DESCRIPTION": "Due to either a session timeout or unauthorized access, yo
u do not have permission to access this page.",
  "BUILDINGS": "Buildings",
  "ALL_BUILDINGS_DESCRIPTION": "A list of all buildings in TRIRIGA",
  "NAME": "Name",
  "PARENT_PROPERTY": "Parent Property"
}
```

Next, let's create the label constants. Under the `/src/utlis/messages` folder, open the **ApplicationMessages.js** file and add the following label constants:

- BUILDINGS
- ALL_BUILDINGS_DESCRIPTION
- NAME
- PARENT_PROPERTY

```
export const MESSAGES = {
  HOME_HEADER: "HOME_HEADER",
  CURRENT_HEADER: "CURRENT_HEADER",
  UNAUTHORIZED_TITLE: "UNAUTHORIZED_TITLE",
  UNAUTHORIZED_DESCRIPTION: "UNAUTHORIZED_DESCRIPTION",
  BUILDINGS: "BUILDINGS",
  ALL_BUILDINGS_DESCRIPTION: "ALL_BUILDINGS_DESCRIPTION",
  NAME: "NAME",
  PARENT_PROPERTY: "PARENT_PROPERTY",
};
```


Step 13. Add the Route to the Page

Now, let's add a route to the new page that we created. Under the `/src/utills/constants` folder, open the `Routes.js` file and add the path to the **BUILDINGS** route as follows:

```
export const HOME = "/";
export const CURRENT_USER = "/user";
export const BUILDINGS = "/buildings";
```

Next, add the **AllBuildingsPage** to the main application component. Under the `/src/app` folder, open the `TririgaUXWebApp.js` file, add the import of the **AllBuildingsPage**, and declare the **AllBuildingsPage** component inside a **Route** element as follows:

```
import { HomePage, CurrentUserPage, AllBuildingsPage } from "../pages"; // update this line
...
render() {
  const { loading, message } = this.state;
  return (
    <div className={cssBase}>
      <Switch>
        <Route path={Routes.CURRENT_USER}>
          <CurrentUserPage />
        </Route>
        <Route path={Routes.BUILDINGS}> // add this line
          <AllBuildingsPage /> // add this line
        </Route> // add this line
        <Route path={Routes.HOME}>
          <HomePage />
        </Route>
      </Switch>
      <ShowAppMessages message={message} clearMessage={this.clearMessage} />
      {createPortal(<Loading active={loading} withOverlay />, document.body)}
    </div>
  );
}
```

Next, we must add a new **Buildings** button on the home page that allows the user to navigate to the **AllBuildingsPage**. Under the `/src/pages/HomePage` folder, open the `HomePage.js` file and configure the primary button of the **FooterButtons** component as follows:

```
export default class HomePage extends React.PureComponent {
  render() {
    return (
      <div className={cssBase}>
        <div className={` ${cssBase}__header`}>
          {AppMsg.getMessage(AppMsg.MESSAGES.HOME_HEADER)}
        </div>
        <div className={` ${cssBase}__content`} />
        <FooterButtons
          secondaryLabel={AppMsg.getMessage(AppMsg.BUTTONS.CURRENT_USER)}
          secondaryRoute={Routes.CURRENT_USER}
          primaryLabel={AppMsg.getMessage(AppMsg.MESSAGES.BUILDINGS)} // add this line
          primaryRoute={Routes.BUILDINGS} // add this line
        />
      </div>
    );
  }
}
```

Finally, return to your browser to check the new page that you created.

- If you closed the browser tab with your page, open a new tab with your local URL: <http://localhost:3000>.
- If your application is not running, verify that your local server is running (Step 7).
- If necessary, check your browser console for errors.

Congratulations! You've built your first TRIRIGA UX Web Application with **ReactJS** and **IBM Carbon** components.

The screenshot displays a web application interface with a light gray background. On the left, a vertical sidebar contains the text "Welcome to the UX Web Application Home Page" at the top. Below this, there are two buttons: "Current User" (a dark gray button) and "Buildings" (a blue button). On the right side of the page, there is a table titled "Buildings". The table has two columns: "Name" and "Parent Property". The "Parent Property" column has an upward-pointing arrow icon to its right. The table contains eight rows of data. Below the table, there is a dark gray button labeled "Back to Home".

Name	Parent Property	↑
BBFC Denver - Corporate	Denver Property	
BBFC Denver - Shipping	Denver Property	
Las Vegas - Building Eight	Headquarters Campus	
Las Vegas - Building Eleven	Headquarters Campus	
Las Vegas - Building Fifteen	Headquarters Campus	
Las Vegas - Building Five	Headquarters Campus	
Las Vegas - Building Four	Headquarters Campus	
Las Vegas - Building Fourteen	Headquarters Campus	