EzeXtend Development Manual

Anthony Mesa

Nov 14, 2022

CONTENTS

CHAPTER

ONE

INSTALL

To work with ezeXtend, first you must install the current ezeXtend codebase in your development workspace. The current way to do this is to get the pre-setup project provided in the subversion repository for Mcube version 4.5.0.0. The subversion repository link is:

```
http://svnmcube.tcgdigital.com/svn/MCube_Implementation/Development_Artefacts/Dev-Area/4.
↪5.0.0/
```

Clone the repository to a folder of your choice, such as your user home folder.

Note: Use our Subversion guide (here) if you are unfamiliar with retrieving projects using TCG Digital's Subversion repository.

Once you have cloned the Subversion repository, the codebase for ezeXtend can be found within:

```
4.5.0.0/Code/module_designer/
```

Going forward, this `module_designer` folder will be referred to as `/EZEXTEND_ROOT`.

1.1 NVM

Node Version Manager is essential when working within a NodeJS environment as it allows us to download and use specific versions of NodeJS and Node Package Manager (NPM) given that many different projects may require different versions. The installation instructions for NPM can be found on their website

After installing, tell NVM to download the specific version we need (If you already have this version installed, you can skip the installation command):

```
$ nvm install 10.15.3
```

1.2 Nginx

Minimum Version: 14+ (v16.13.2 reccomended)

Nginx is a server software that we will be using as a simple reverse proxy for our development environment. This is required so that we can develop 'as if' we are working with a backend located outside of our development environment or on the cloud, etc. This is important for greater control in port allocation and bypassing any issues that might arise from violating Cross-Origin Resource Sharing (CORS) policy.

You will need to install the latest version of Nginx on your development environment. If you are using our Dockerized development container, then Nginx should alredy be installed.

This guide will only cover configuration for Linux based operating systems, so Windows or Mac users may need to adapt this guide to however Nginx differs on those systems (locations of config files, etc.).

Install Nginx with your package manager, e.g.:

```
$ sudo apt-get install nginx
```

CHAPTER

TWO

SETUP

2.1 NVM

Set NVM to use the NodeJs version required for this project:

```
nvm use 10.15.3
```

2.2 Nginx

ezeXtend will require an Nginx reverse proxy so that calls made to local routes will be redirected to their relevant services, whether those services are running on the local dev machine, or on a cloud-hosted VM.

To edit the Nginx configuration you are going to want to add a configuration file named `ezeXtend_proxy` inside the folder `/etc/nginx/sites-available/`. Within the file, you will need to define an Nginx server and the proxy route locations with minimal configuration, such as below:

```
server {
   location /module_designer/ {
      proxy_pass http://127.0.0.1:3001/;
   }

   location /elastic_api/ {
      proxy_pass http://127.0.0.1:9241/;
   }
}
```

The purpose of this configuration is to only register two url routes, `/module_designer/` and `/elastic_api/` that redirect calls for both the ezeXtend react page itself and the Elasticsearch database.

Now you can start Nginx with:

```
sudo nginx
```

Note: If Nginx starts successfully, you will see no errors or output, as it begins the server and runs it in the background. Because Nginx starts and runs in the background, before trying to run it, it is helpfull to use `htop` before hand to ensure that it isn't already running. If it us, kill the currently running Nginx with `sudo pkill -9 nginx`.

2.3 ezeXtend

Now we must prepare the ezeXtend project to be run. The ezeXtend project is split into two pieces, one inside of the other. The module designer project contains both the frontend for the ezeXtend page (in the `ui` folder) as well as backend code (in `server`) that helps it interface with Kibana/Elasticsearch behind the scenes.

Whenever we are extending the functionality of the front end (creating custom visualisations, etc.) we can run the front end in either a development mode which lacks an amount of full functionality, or a release mode that more closely emulates the full functionality of the page when working in tandem with Kibana/Elasticsearch. To achieve that full release functionality, you must build the `ui` react project inside `module_designer` and use the Nginx reverse proxy to provide Elasticsearch connection functionality. This may sound confusing but we will cover both aspects.

2.3.1 Development Mode

To run ezeXtend in development mode, execute:

```
cd /EZEXTEND_ROOT/ui
npm install
npm start
```

Assuming all went well, then the react project should start up and be available at `http://localhost:3000`.

2.3.2 Release Mode

To run ezeXtend in release mode, execute:

```
cd /EZEXTEND_ROOT/
npm install
cd /EZEXTEND_ROOT/ui
npm install
npm run build
cd ..
npm start
```

Again, assuming all went well, then the release version of the React project should be available at `http://<local domain>/module_designer` where the `<local domain>` can either be localhost, or any host that you have listed in your `/etc/hosts` file that redirect to `127.0.0.1`.

CHAPTER

THREE

CREATING A CUSTOM DASHBOARD COMPONENT

This guide explains the process of creating a custom component that would be included in the ezeXtend source code and built into MCube. In the future, we hope to create a system that allows clients and users to create dashboard components without having access to the MCube or ezeXtend source code, but until then, this is the best current solution for adding custom functionality to ezeXtend.

In this guide we will be making a custom Button element.

3.1 Add Entry in Components List Sidebar

Before creating the custom Button component itself, we need to provide details about our custom element to ezeXtend so that the custom component is displayed as an option in the component sidebar on the left side of the ezeXtend window. Doing this first will allow us to be able to test our Button in the UI and catch any errors in our Button development along the way.

First, open `/EZEXTEND_ROOT/ui/src/AppConstants/WidjetsMapping.js`. Inside of this file is a constant variable WidgetsMapping that contains a JSON object of the Labels to be displayed in the sidebar panel as available elemenents to use in the dashboard. Here we add a definition for our new element `CUSTOM_BUTTON` and the string label that will be shown for it.

Note: For the sake of brevity in the code examples, elipses are used to denote code that we are not concerned with for our example and thus does not need to be displayed in this tutorial.

```
1  export const WidgetsMapping = {
2      SHAPES: {
3          ...
4      },
5      CHARTS: {
6          ...
7      },
8      INPUTS: {
9          TEXTBOX: 'Text',
10         BUTTON: 'Button',
11         CUSTOM_BUTTON: 'Custom Button',
12         RADIO: 'Radio',
13         SELECT: 'Select',
14         MULTI_SELECT: 'Multi Select',
15         LABEL: 'Label',
16         IMAGE:'Img',
```

(continues on next page)

(continued from previous page)

```
    },
    OTHERS: {
        ...
    },
};
```

Next, we need to add the data for the component entry in the sidebar, that is, we need to specify things like the icon to be displayed, etc. To do this, first open `/EZEXTEND_ROOT/ui/src/Components/Sidebar/ComponentsData.js`. This file contains a constant variable `Groups` that contains a JSON object with lists of JSON descriptions of each of the sidebar components belonging to each group in the panel. Because we add our custom button to the `Inputs` section of the `WidgetsMapping` so too do we have to add a new sidebar component to the `Inputs` list within the JSON object:

```
export const Groups = {
   Shapes: [
      ...
   ],
   Charts: [
      ...
   ],
   Inputs: [
      {
            title: WidgetsMapping.INPUTS.TEXTBOX,
            icon: <BsInputCursor size={size} color={activeColor} />,
            active: true,
      },
      {
            title: WidgetsMapping.INPUTS.BUTTON,
            icon: <GiClick size={size} color={activeColor} />,
            active: true,
      },
      {
            title: WidgetsMapping.INPUTS.CUSTOM_BUTTON,
            icon: <HiCursorClick size={size} color={activeColor} />,
            active: true,
      },
      {
            title: WidgetsMapping.INPUTS.RADIO,
            icon: <IoMdRadioButtonOn size={size} color={activeColor} />,
            active: true,
      },
      {
            title: WidgetsMapping.INPUTS.SELECT,
            icon: <BiSelectMultiple size={size} color={activeColor} />,
            active: true,
      },
      {
            title: WidgetsMapping.INPUTS.LABEL,
            icon: <MdLabel size={size} color={activeColor} />,
            active: true,
      },
      {
```

(continues on next page)

(continued from previous page)

```
            title: WidgetsMapping.INPUTS.IMAGE,
            icon: <FaImages size={size} color={activeColor} />,
            active: true,
        },
    ],
    Others: [
        ...
    ],
};
```

Notice that we are referencing the title via the key-value pair that we entered into the WidgetsMapping object. For the icon, we are using one of the available click-related React icons that are freely available to React developers. You can find more of these icons at react-icons. We are using the `HiCursorClick` icon for our Button, so we will need to import that icon as a dependency at the top of the file:

```
import {
    ...
} from 'react-icons/ai';

...
import { GrGraphQl } from "react-icons/gr";
import { BiSelectMultiple } from 'react-icons/bi';
import { RiCheckboxMultipleBlankLine } from 'react-icons/ri';
import { HiCursorClick } from 'react-icons/hi';
import { WidgetsMapping } from 'AppConstants';
const size = 20;
const color = 'rgb(203 203 203)';
...
```

If we run ezeXtend in Development Mode, we can see our component that has yet to be created listed as an option in the component sidebar. This is shown in Fig. 3.1.

3.2 Create Default Parameters for Component

To create the default parameters for our custom button we need to create a new file in `/EZEXTEND_ROOT/ui/src/Components/Widgets/Defaults` named `CustomButton.js`.

In this file, we need to create a single exported variable, a JSON object that contains information describing the default state of the component when it is initially dragged onto the dashboard:

```
export const CustomButtonData = {
    size: {
        width: 200,
        height: 40
    },

    data: {
        label: 'Action'
```

(continues on next page)

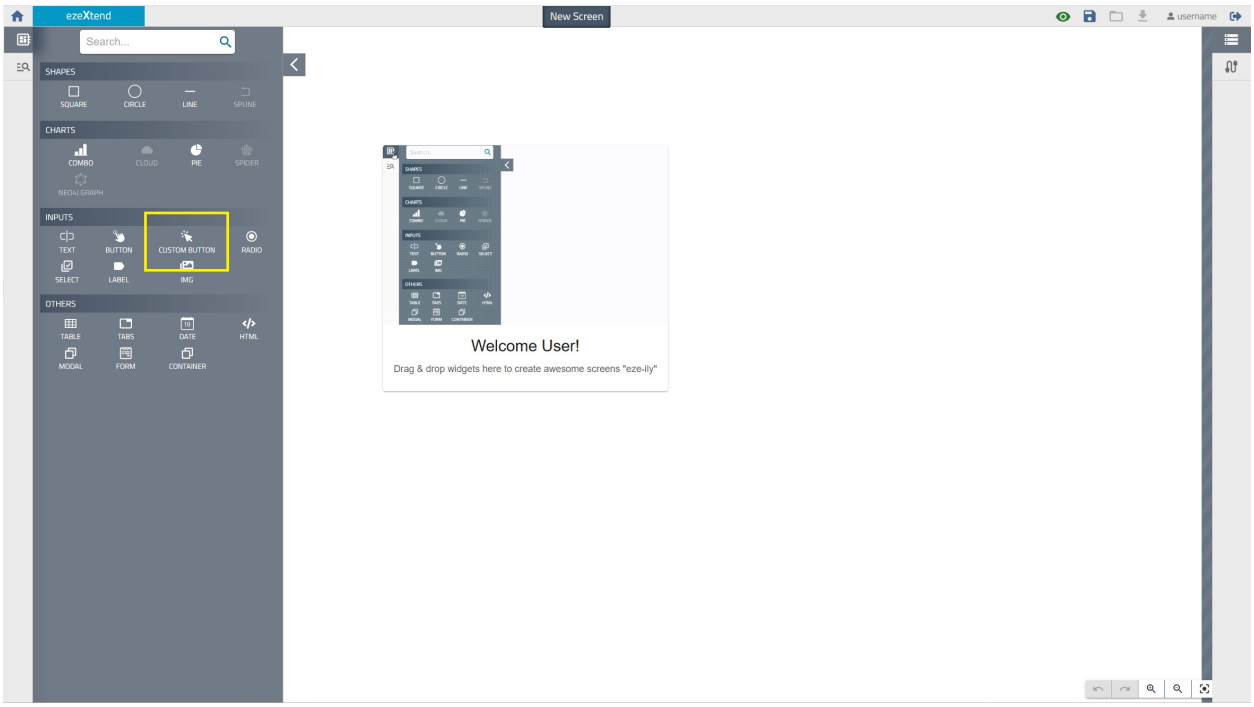

Fig. 3.1: A new custom button entry listed in the widgets sidebar.

(continued from previous page)

```
    }
}
```

Here we have defined the initial size of the element and the string text to be displayed on the button when it is first created. Now, we can navigate to the file `/EZEXTEND_ROOT/ui/src/Components/Widgets/Defaults/index.js` to import and apply the default data we have just created. First we will import the JSON object we just created:

```
...
import { ChartData } from './Chart';
import { ButtonData } from './Button';
import { CustomButtonData } from './CustomButton';
import { MarkdownData } from './Markdown';
...
```

Currently as you can see, all of the custom data is spread across multiple files. This index file will serve as a one-stop-shop to access all of these default data objects. To achieve this, all of the defaults that are imported are promptly exported from `index.js` so that they can be called elsewhere by importing this index file. We will include our imported custom button data as an available export:

> **Attention:** More info needs to be added here as why this is necessary given that this list of exports is not used elsewhere in the code.

```
export {
  ChartData,
```

(continues on next page)

(continued from previous page)

```
  ButtonData,
  CustomButtonData,
  MarkdwonData,
  ...
};
```

Lastly, we need to add some logic to the default function `WidgetDefaultsProvider(type)` so that when it is called with the argument `WidgetsMapping.INPUTS.CUSTOM_BUTTON` it returns the appropriate data:

```
export default function WidgetDefaultsProvider(type) {
   switch (type) {
      case WidgetsMapping.CHARTS.COMBO:
         return ChartData;
      case WidgetsMapping.INPUTS.BUTTON:
         return ButtonData;
      case WidgetsMapping.INPUTS.CUSTOM_BUTTON:
         return CustomButtonData;
      case WidgetsMapping.INPUTS.RADIO:
         return RadioData;
      ...
   }
}
```

3.3 Create a Custom React Component

Now, we can actually start creating our custom buton! To do so, we will first create a folder for our React module code called `CustomButton` in `/EZEXTEND_ROOT/ui/src/Components/Widgets/`.

Create and open a new file `CustomButton.js`. EzeXtend uses Redux to manage complicated state in the application, so we will need to import the state selector from Redux:

Note: If this is your first time hearing about "state" in regards to React, Redux, or both, then you should take a moment to look into both of these softwares and the concept of state and how they handle them in greater depth, as this is an integral part of working with React and can be confusing if you are out of the loop.

```
import { useSelector } from 'react-redux';
import { Button } from '@mui/material';
```

We are taking the easy route with this button, and rather than recreating a button from scratch, we are instead using a pre-made React button provided in the Material library and we are wrapping it with our custom specifications. We need to define the functional component and provide it as an exportable default:

```
...
import { Button } from '@mui/material';

function CustomButton({ id }) {

}
```

(continues on next page)

(continued from previous page)

```

export default CustomButton;
```

Next, we will make our custom button return the pre-made button provided by the Material library. We will also provide some properties to the JSX element so that we can customize the button slightly:

```
function CustomButton({ id }) {
   return (
      <Button sx={{height: '100%', width: '100%'}} variant='outlined'>

      </Button>
   )
}
```

Here we are passing two props, `sx` and `variant`. The former allows us to define a JSON object containing a subset of CSS parameters that will be used to override the default CSS styling that comes with Material UI buttons. The latter is a property defined by Material, the button type or variant. In this case we are using the 'outlined' button variant.

We want the label for our button to be displayed within the button itself, so to do this we will need to get the label from our `CustomButtonData` from earlier. The crux here is that we will not be importing the button data directly, instead EzeXtend will load that custom data at runtime and make it available throughout the application via Redux. Because of this, we will have to load the data in from Redux and apply it to our label within the CustomButton module like so:

Note: The array `evaluatedWidgetProperties` at index `id` contains the JSON object value corresponding to the `data` key in the `CustomButtonData` object we defined.

```
function CustomButton({ id }) {

   const props = useSelector(
      (store) => store.dashboard.evaluation.evaluatedWidgetProperties[id]
   );

   return (
      <Button sx={{height: '100%', width: '100%'}} variant='outlined'>
         { props.label }
      </Button>
   )
}
```

Were we are providing `useSelector()` an anonymous function as an argument, knowing that that anonymous function will be provided the store that contains our state data.

Finally, we need to tell EzeXtend that our component exists. To do this we will modify `/EZEXTEND_ROOT/ui/src/Components/Panel.js`. First import the new component, and then in the function `ComponentProvider()`, we need to return the JSX for our new custom button in the case that it is selected from within the panel:

```
...
import Button from 'Components/Widgets/Button/Button';
import CustomButton from 'Components/Widgets/CustomButton/CustomButton';
```

(continues on next page)

(continued from previous page)

```
import Radio from 'Components/Widgets/Radio/Radio';
...
function ComponentProvider(type, id) {
   switch(type) {
      case WidgetsMapping.CHARTS.COMBO:
         return <Chart id={id} />;
      case WidgetsMapping.INPUTS.BUTTON:
         return <Button id={id} />;
      case WidgetsMapping.INPUTS.CUSTOM_BUTTON:
         return <CustomButon id={id} />;
      case WidgetsMapping.INPUTS.RADIO:
         return <Radio id={id} />;
      ...
   }
}
```

Now, we can run the project in development mode once more. We should be able to click and drag the custom button from the sidebar entry onto the dashboard. This is shown in Fig. 3.2.

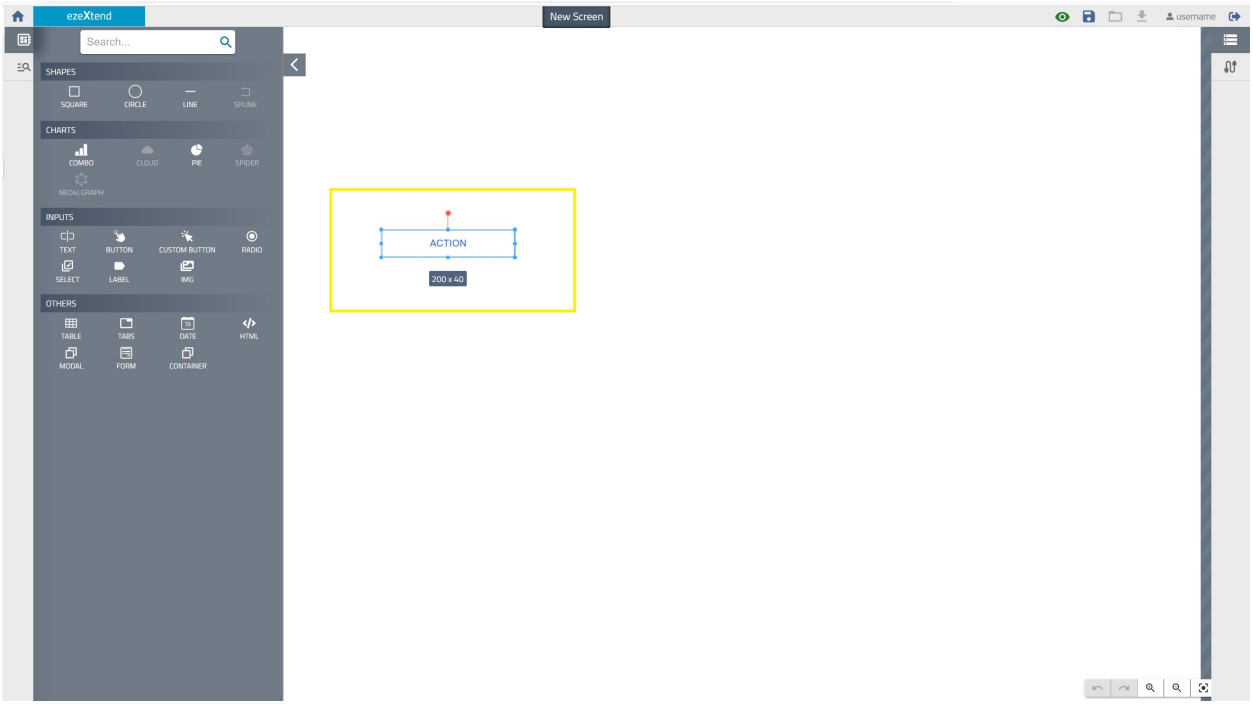

Fig. 3.2: A custom button widget displayed in the dashboard.

3.4 Create Component Formatting Options

The last step for integrating a custom element into the EzeXtend UI is to add formatting options, that is, options that can be provided to the element while it is on the dashboard to customize its appearance or action.

We will begin with a new file `/EZEXTEND_ROOT/ui/src/Components/Widgets/CustomButton/CustomButtonOptions.js`.

In this new file we need to import some important things to help us out. First we need to import the `PropertiesGroup`. This is an element provided by EzeXtend that we can modify and will then be injected into the Properties panel providing our desired functionality. We can create multiple groups if we would like, it just depends on how you are wanting to structure your options. Next we are importing `TextField` from the material UI library, this is the text field we will display in our property group that will update our button. For state-related functionality (such as changing the text displayed on the button) we will need to get two functions from redux, `useSelector` and `useDispatch`. The former allows us to retrieve data from Redux, and the latter allows us register changes we would like to make to the state (and Redux will handle the changes accordingly). `UpdateWidgetProperty` is a Reducer we designed to work with Redux to make this state-changing functionality easier to work with, so we will also import that as well.

```
import PropertiesGroup from "CommonComponents/PropertiesGroup";
import { TextField } from '@mui/material';
import { useSelector, useDispatch } from 'react-redux';
import UpdateWidgetProperty from "Store/Reducers/Actions/UpdateWidgetProperty";
```

Like before, we will define our module, its return statement and that it is the default export of this file:

```
function CustomButtonOptions() {
   return (

   )
}

export default CustomButtonOptions;
```

We need to display the label text of our button saved in `CustomButtonData.js`, so we will need to grab the label from the state data store managed by Redux. First we retrieve the id of the widget by getting the id value stored by EzeXtend as the currently 'active panel'. Then we use that ID to retrieve the option data from Redux. We will also need to create a Redux dispatch object to use later when we want to update the label value.

```
function CustomButtonOptions() {
   const id = useSelector((store) => store.dashboard.appState.activePanelID);
   const options = useSelector((store) => store.dashboard.widgets[id]);
   const dispatch = useDispatch();

   return (

   )
}
```

In the return statement, we define our property group, giving it a title, and inside that group, we create a text field element provided by Material UI. The value displayed on the label is the 'label' value of the options object that we retrieved earlier. The onChange property we are providing a function name `handleChange` that we haven't yet defined.

```
function CustomButtonOptions() {
   const id = useSelector((store) => store.dashboard.appState.activePanelID);
   const options = useSelector((store) => store.dashboard.widgets[id]);
   const dispatch = useDispatch();

   return (
      <PropertiesGroup title="Format">
         <TextField label="Label" value={options.label} onChange={handleChange}/>
      </PropertiesGroup>
   )
}
```

To wrap it all up, we create our last variable, a function that takes in an event `e`. The function runs Redux's dispatch function which evaluates whatever is returned by `UpdateWidgetProperty()`. We provide `UpdateWidgetProperty` with the id of the widget we are looking to modify, the path of the options value we want to modify (in this case, the path is equal to the 'key's traversed in `CustomButtonData.data` to get to the value). The value is set to be the value of the text box whose updating triggered the event:

```
function CustomButtonOptions() {
   const id = useSelector((store) => store.dashboard.appState.activePanelID);
   const options = useSelector((store) => store.dashboard.widgets[id]);
   const dispatch = useDispatch();

   const handleChange = function(e) {
      dispatch(
            UpdateWidgetProperty({
               id,
               update: {
                  path: 'label',
                  value: e.target.value
               }
            })
      )
   }

   return (
      <PropertiesGroup title="Format">
         <TextField label="Label" value={options.label} onChange={handleChange}/>
      </PropertiesGroup>
   )
}
```

Running the development server once more, if we drag the CustomButton element onto the dashboard and open the buton's properties panel by clicking the gear icon, then we will see the Format properties group displayed. Using the new 'Label' text field we can change the name displayed on the button to anything we would like. We can see this change reflected in Fig. 3.3.

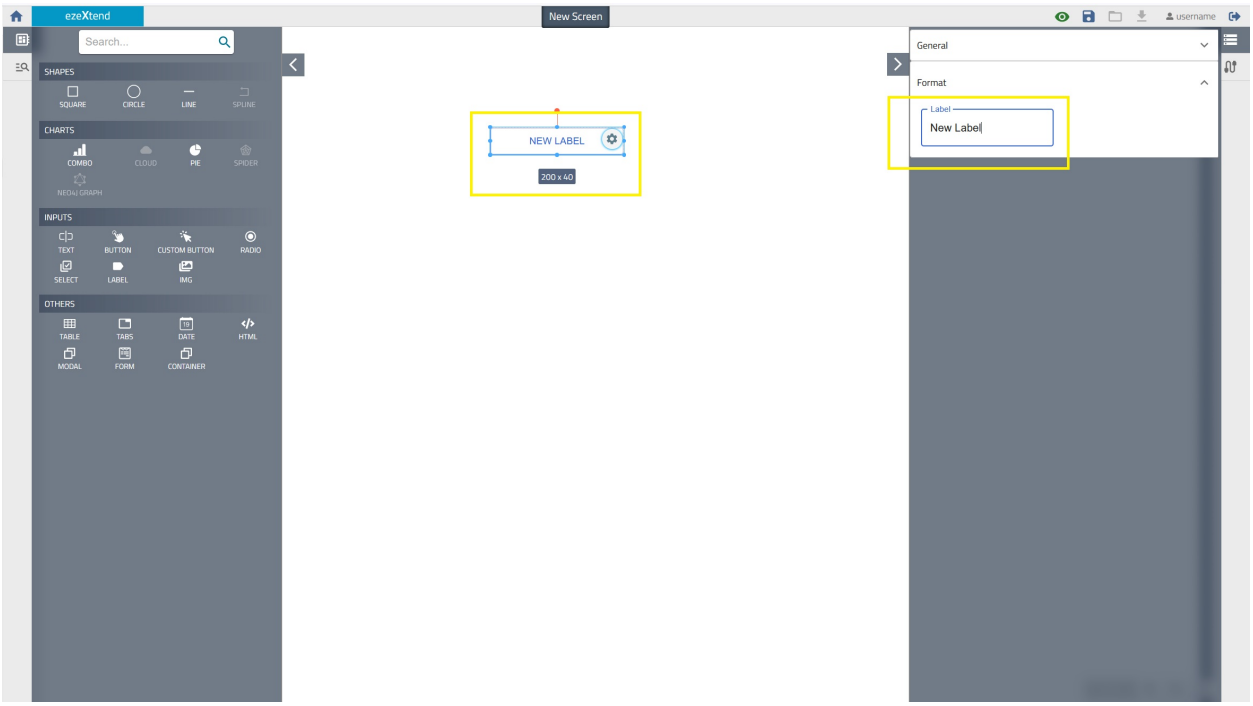

Fig. 3.3: A custom button and its properties panel. The panel has been used to update the button's label text.

CHAPTER

FOUR

WRAPPING UP

If you encounter an issue during this process, open an issue on the Github repository for this documentation Here and we will do the best we can to help you. As of this time the EzeXtend extension process requires access to the EzeXtend source code that is only available to TCG Digital developers and partners. If you would like to become a part of EzeXtend community development in the future, please reach out to us at () for more info.