

Building Custom React DX Components from Scratch

Set Up development Environment:

- Install Visual Studio code from these url <https://code.visualstudio.com/download> based on your operating system (Example: windows, Ubuntu, macOS)
Install Git from these url <https://git-scm.com/downloads> Make sure the version is 2.30 or later.
- Install Node.js from these url <https://nodejs.org/en/download> Ensure the version is 18.0.0 or later.
- Once you've downloaded and installed all these tools, open the terminal and run the following commands to verify the installations
node --version and npm --version

```
Microsoft Windows [Version 10.0.19045.5608]
(c) Microsoft Corporation. All rights reserved.

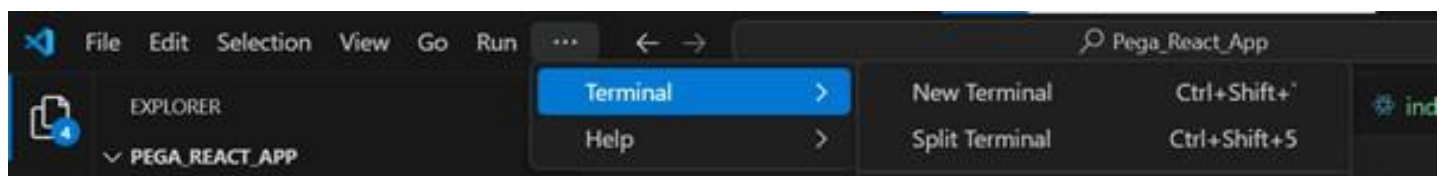
C:\Users\DineshMaddi>node --version
v20.18.1

C:\Users\DineshMaddi>npm --version
10.8.2

C:\Users\DineshMaddi>
```

Procedure of creating custom component:

In visual studio, open a new terminal.



In your terminal, enter the following command:

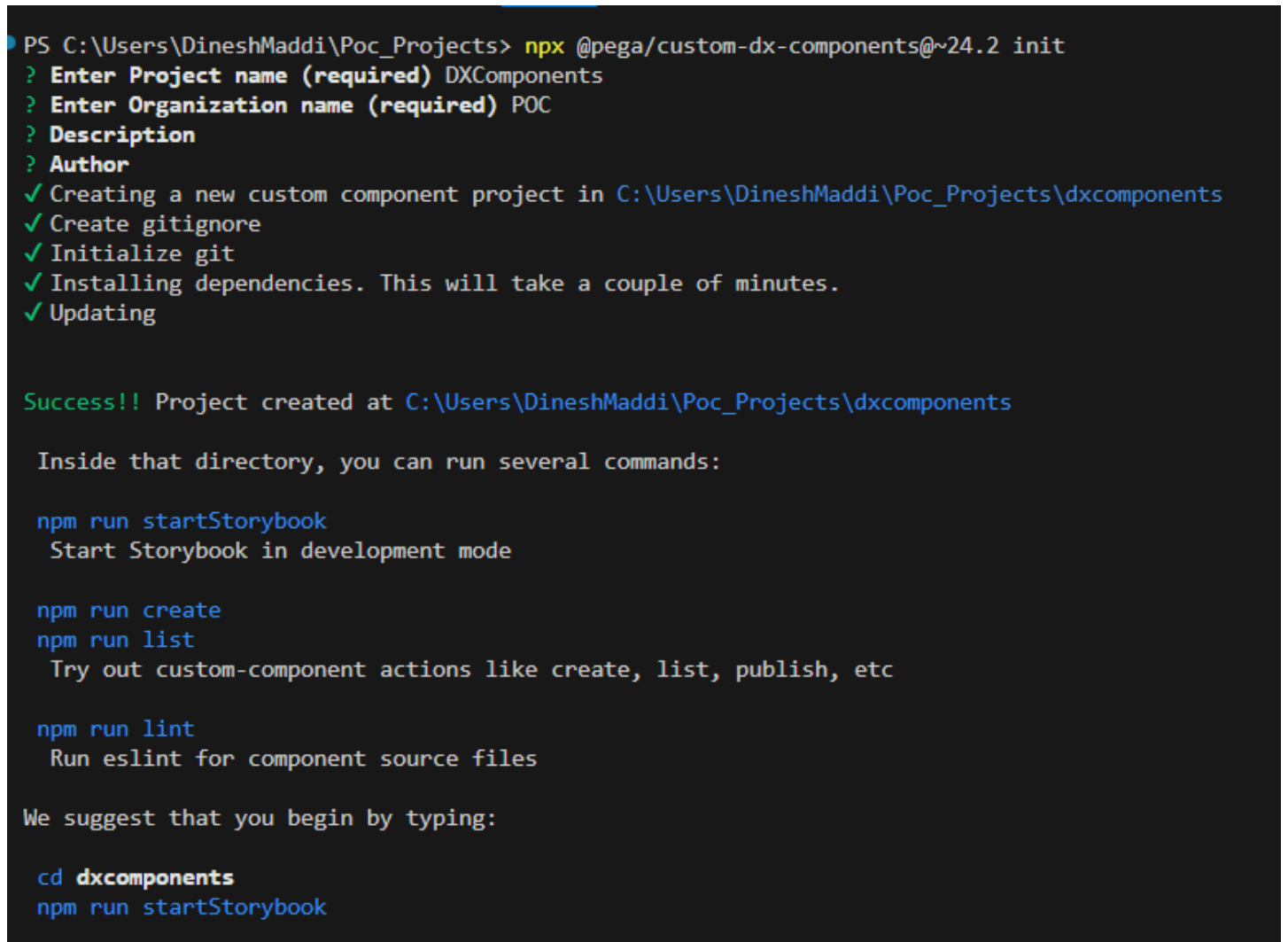
```
npx @pega/custom-dx-components@~24.2 init.
```

Complete the required information to set up your DX Component Builder project:

- a. In the Enter Project name (required) line, for example: enter Dxcomponents.
- b. In the Enter Organization name (required) line, for example: enter POC.

Press the Enter key to accept defaults for everything else.

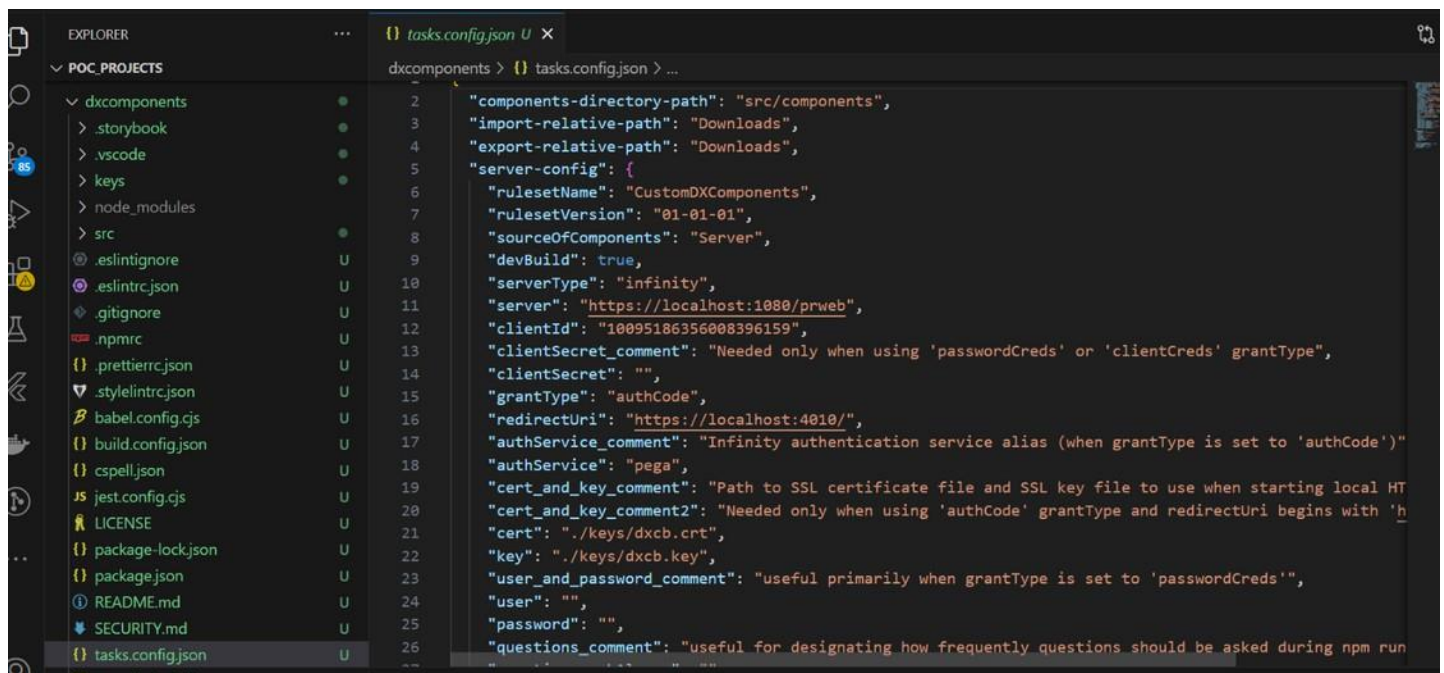
After successful installation, the following information is displayed similar to the following image.

A terminal window with a dark background and light blue text. The prompt is 'PS C:\Users\DineshMaddi\Poc_Projects>'. The command entered is 'npx @pega/custom-dx-components@~24.2 init'. The output shows a series of prompts: 'Enter Project name (required)' with 'DXComponents' entered, 'Enter Organization name (required)' with 'POC' entered, and 'Description', 'Author', and 'Description' prompts which are skipped. Then, several green checkmarks indicate successful steps: 'Creating a new custom component project in C:\Users\DineshMaddi\Poc_Projects\dxcomponents', 'Create gitignore', 'Initialize git', 'Installing dependencies. This will take a couple of minutes.', and 'Updating'. A green 'Success!!' message follows, stating 'Project created at C:\Users\DineshMaddi\Poc_Projects\dxcomponents'. Below this, it says 'Inside that directory, you can run several commands:' and lists several npm commands with their descriptions: 'npm run startStorybook' (Start Storybook in development mode), 'npm run create' (Try out custom-component actions like create, list, publish, etc), 'npm run list' (Try out custom-component actions like create, list, publish, etc), 'npm run lint' (Run eslint for component source files). Finally, it suggests 'We suggest that you begin by typing:' and shows 'cd dxcomponents' and 'npm run startStorybook'.

Set your path in the terminal with the project name (for example: cd dxcomponents)

Update your tasks.config.json file:

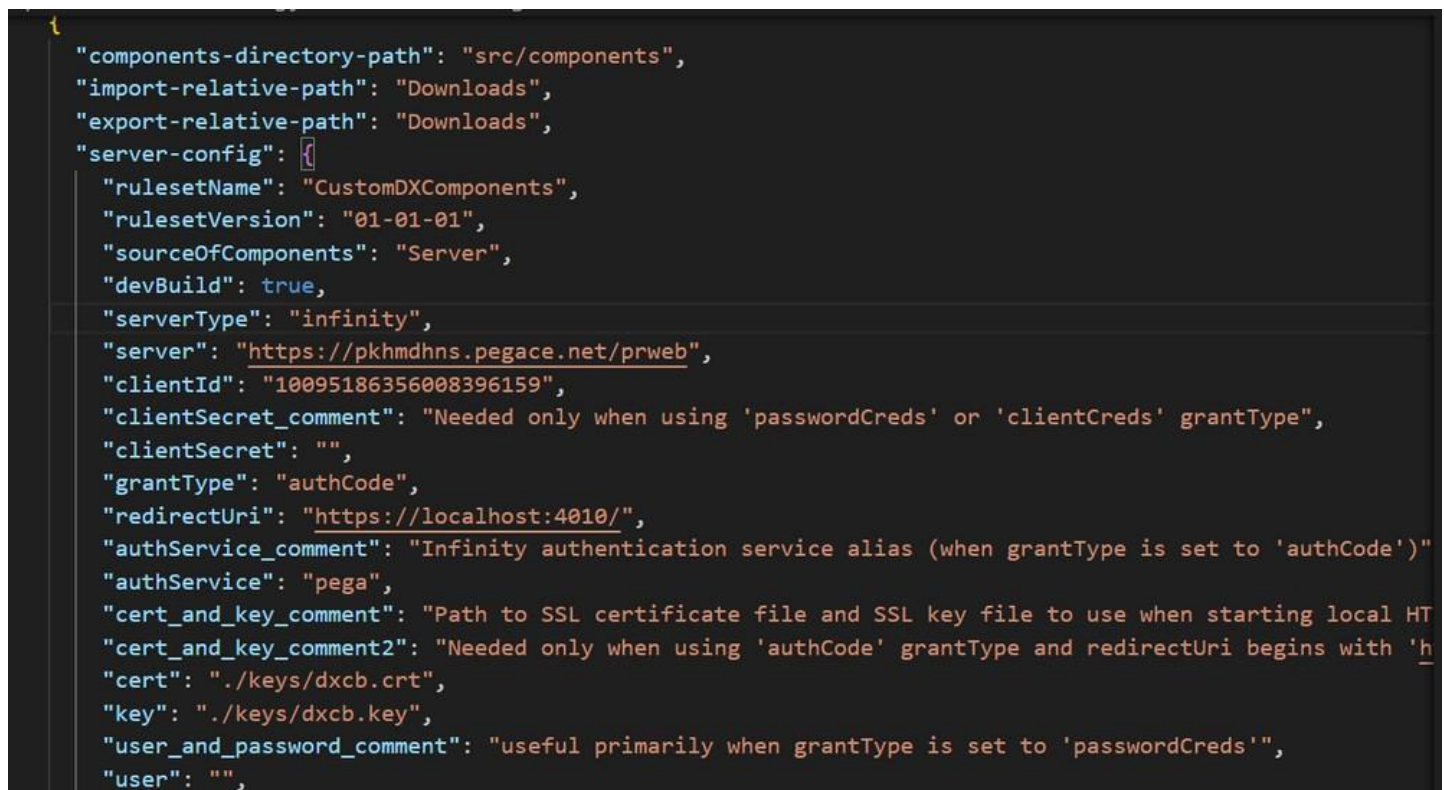
Open your task.config.json file in your created custom component project. It will look like as given image below



The screenshot shows a Visual Studio Code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'POC_PROJECTS' with a sub-project 'dxcomponents'. The code editor displays the 'tasks.config.json' file with the following content:

```
{
  "components-directory-path": "src/components",
  "import-relative-path": "Downloads",
  "export-relative-path": "Downloads",
  "server-config": {
    "rulesetName": "CustomDXComponents",
    "rulesetVersion": "01-01-01",
    "sourceOfComponents": "Server",
    "devBuild": true,
    "serverType": "infinity",
    "server": "https://localhost:1080/prweb",
    "clientId": "10095186356008396159",
    "clientSecret_comment": "Needed only when using 'passwordCreds' or 'clientCreds' grantType",
    "clientSecret": "",
    "grantType": "authCode",
    "redirectUri": "https://localhost:4010/",
    "authService_comment": "Infinity authentication service alias (when grantType is set to 'authCode')",
    "authService": "pega",
    "cert_and_key_comment": "Path to SSL certificate file and SSL key file to use when starting local HT",
    "cert_and_key_comment2": "Needed only when using 'authCode' grantType and redirectUri begins with 'h",
    "cert": "./keys/dxcb.crt",
    "key": "./keys/dxcb.key",
    "user_and_password_comment": "useful primarily when grantType is set to 'passwordCreds'",
    "user": "",
    "password": "",
    "questions_comment": "useful for designating how frequently questions should be asked during npm run"
  }
}
```

Open your pega Infinity and copy the instance url (Ex: <https://pkhmdhns.pegace.net/prweb>) these url paste into server in the tasks.config.json.



This is a close-up view of the 'server-config' object in the 'tasks.config.json' file. The configuration is as follows:

```
{
  "rulesetName": "CustomDXComponents",
  "rulesetVersion": "01-01-01",
  "sourceOfComponents": "Server",
  "devBuild": true,
  "serverType": "infinity",
  "server": "https://pkhmdhns.pegace.net/prweb",
  "clientId": "10095186356008396159",
  "clientSecret_comment": "Needed only when using 'passwordCreds' or 'clientCreds' grantType",
  "clientSecret": "",
  "grantType": "authCode",
  "redirectUri": "https://localhost:4010/",
  "authService_comment": "Infinity authentication service alias (when grantType is set to 'authCode')",
  "authService": "pega",
  "cert_and_key_comment": "Path to SSL certificate file and SSL key file to use when starting local HT",
  "cert_and_key_comment2": "Needed only when using 'authCode' grantType and redirectUri begins with 'h",
  "cert": "./keys/dxcb.crt",
  "key": "./keys/dxcb.key",
  "user_and_password_comment": "useful primarily when grantType is set to 'passwordCreds'",
  "user": "",
  "password": ""
}
```

Go to Pega Infinity for creating client credentials in Records>Security> OAuth2.0 Client Registration. Create a new one client registration and it will generate “ClientID” and “Client Secret” ,download the creditanls before saving.

edit OAuth 2.0 Client Registration: CustomDXComponents

D: CustomDXComponents RS: No associated ruleset [Edit]

Actions Save

Client Credentials

Type of client

☒ Confidential

☐ Public

Client ID

56239932768162319480

Client secret

.....

Authorization endpoint

https://pkhmdhns.pegace.net/prweb/PRRestService/oauth2/v1/authorize

Access token endpoint

https://pkhmdhns.pegace.net/prweb/PRRestService/oauth2/v1/token

Token revocation endpoint

https://pkhmdhns.pegace.net/prweb/PRRestService/oauth2/v1/revoke

JWKS endpoint

https://pkhmdhns.pegace.net/prweb/PRRestService/oauth2/v1/token/keys

Token introspection endpoint

https://pkhmdhns.pegace.net/prweb/PRRestService/oauth2/v1/token/introspect

A secure TLS access token endpoint is required at runtime.

Regenerate client secret Revoke access and refresh token View & download

In the Granttypes uncheck the clientcreditanls , check the checkbox of Password Creditanls select the enable refresh token and select the pyDefaultIdentityMappingForPasswordGrant

Supported grant types

☐ Authorization code

☐ Client credentials

☒ Password credentials

Identity mapping*

pyDefaultIdentityMapping

☒ Enable refresh token

☐ SAML bearer

☐ JWT bearer

Open your visual studio code with the project created custom dx component. Go to file again task.config.json change the ClientID and ClientSecret with previously created in pega.

Change the grantType as passwordCreds, redirect uri is also your pega infinity url. Give the pega operator id username and password in the task.config.json file.

Create a ruleset in pega with the given ruleset in task.config.json file and as well as same version as below give in the image.

```
server-config : {
  "sourceOrComponents": "Server",
  "devBuild": true,
  "serverType": "infinity",
  "server": "https://pkhmdhns.pegace.net/prweb",
  "clientId": "96660693548746559302",
  "clientSecret_comment": "Needed only when using 'passwordCreds' or 'clientCreds' grantType",
  "clientSecret": "3AC78D867004AF9E12123D5E3767C686",
  "grantType": "passwordCreds",
  "redirectUri": "https://pkhmdhns.pegace.net/prweb",
  "authService_comment": "Infinity authentication service alias (when grantType is set to 'authCod",
  "cert_and_key_comment": "Path to SSL certificate file and SSL key file to use when starting loca",
  "cert_and_key_comment2": "Needed only when using 'authCode' grantType and redirectUri begins wit",
  "cert": "./keys/dxcb.crt",
  "key": "./keys/dxcb.key",
  "user_and_password_comment": "useful primarily when grantType is set to 'passwordCreds'",
  "user": "dinesh@pega.com",
  "password": "rules@1234",
  "questions_comment": "useful for designating how frequently questions should be asked during npm",
  "questions_askAlways": "",
  "questions_askNever": "",
  "questions_askOnce": "server,authService",
  "questions_askedOnce": "server"
},
```

Go to terminal for authentication, run the command of “npm run authenticate” then it navigates to pega to check authentication as below shown in the image.

```
PS C:\Users\DineshMaddi\Poc_Projects\dxcomponents> npm run authenticate
DX Component Builder v24.2.12
Authenticated successfully !!
PS C:\Users\DineshMaddi\Poc_Projects\dxcomponents> 
```

- To Create a Component run this script in the Terminal:
>npm run create

This will prompt you to select the type of your component, such as Field, Layout Template or Widget.

After selecting the component type, it will display the relevant subtypes based on your selection. Then, you will be prompted to enter the required fields: Name and Label. You can skip all other fields by simply pressing the Enter key.

- For the related component these files can create

config.json	A JSON file that is used by the App Studio authoring experience to build the property pane used by authors to configure your component at design time.
demo.stories.tsx	An example Storybook story is provided to get you started.
index.tsx	The entry point for your component that contains your primary Constellation DX component code.
mock.ts	An example of mock data used by Storybook.
PConnProps.d.ts	A type definition file that provides standard types for the common Constellation DX component props.
styles.ts	Styled Components wrapper component example generated automatically.
create-nonce.ts	Webpack nonce is generated to comply with secure content security policies (CSPs).

- config.json – The properties of the component should be declared in config.json because they will be bound in JSON format and rendered in the UI.

Example: {

```
"name": "placeholder",  
"label": "Placeholder",  
"format": "TEXT"  
}
```

- Demo.stories.tsx – This file is used to test the UI of the component in Storybook. By default, it is linked to the main index.tsx file. In the return statement, the main function/component from index.tsx is called, allowing the component's rendered UI to be tested and verified.
- index.tsx – This file contains the main functionality of the component.

Import the required packages for the component.(@pega/cosmos-react-core)

If your component needs custom styles or external libraries, you can install them via npm and import them as needed in this file.

Next, you need to initialize the props of your component—these are the variables used within the component. All props should be defined inside an interface that extends the default PConnFieldProps.

```
interface PocExtensionsBarCodeProps extends PConnFieldProps {  
  
  // your custom props here  
  
}
```

Note: Defining and using props in the component is mandatory. If the props are not used correctly, ESLint (used by Pega for code validation) will throw an error, and the component will not be allowed to be published.

After that, if you want the component to perform an action—such as storing data in an array or triggering a specific behavior—you should declare a useState hook with an appropriate initial value (e.g., false, an empty string, or an empty array).

```
const [data, setData] = useState([]);
```

```
const [isVisible, setIsVisible] = useState(false);
```

Next, for the function you have created like `setData`, you need to refer to Pega's `pCore` and `pConnect` methods. In Pega, to set a property with a given value for storage, you can use the `pConnect` method `pConnect.getStateProps()`. This method allows you to access the props where data is stored.

For example:

```
const pConn = getPConnect();

const stateProps = pConn.getStateProps();

const propValue = stateProps?.value;
```

Here, `value` refers to the property defined in the interface's props.

Next, implement your logic using the `useEffect()` hook. It will trigger the action inside your function and also initialize or update the prop value accordingly. You should follow this syntax:

```
useEffect(() => {

  // Initialization or data-fetching logic here

  // You can also call your setData function here using pConnect

}, []);
```

Then, return the rendered UI of your component. This defines how the component should appear in Pega, using your props and custom styles. You should extend your component with a CSS file such as `Style.ts`, and use it in the main component file as shown in the example below:

```
return (

  <StyledMyNewComponentWrapper>

    <FieldValueList

      variant={hideLabel ? 'stacked' : variant}

      data-testid={testId}

      fields={[{ id: 'field1', name: hideLabel ? '' : label, value: value ?? '--' }]}

    />
```



```
</StyledMyNewComponentWrapper>
```

```
);
```

In this code:

StyledMyNewComponentWrapper comes from your Style.ts file for applying custom styles.

FieldValueList is used to display a label-value pair UI.

variant, label, value, and hideLabel are passed in via props

Example Use Case with Building Custom React DX Components from Scratch:

This use case demonstrates how to build a radio button group component for publishing in Pega using easy steps with the creation of a DX component.

1. Import Required Packages and Properties

Initially, import the required packages and necessary properties into the main file. This includes the Cosmos React Core components, Pega property types, and your custom styles.

```
1  import {
2    RadioButtonGroup ,
3    RadioButton,
4    withConfiguration
5  } from '@pega/cosmos-react-core';
6  import './create-nonce';
7  import type { PConnFieldProps } from './PConnProps';
8  import StyledPocExtensionsTestWrapper from './styles';
9
```

2. Define Props and State

In this step, Define Props and StateDefine the props interface, extending Pega's PConnFieldProps.

```

11 interface PocExtensionsTestProps extends PConnFieldProps {
12     // Additional props specific to this component
13 }
14
15 // interface for StateProps object
16 interface StateProps {
17     value: string;
18     hasSuggestions: boolean;
19 }

```

3. Initialize Pega Methods and State

Inside your component, initialize the Pega connection and actions API. Extract the property name and any runtime values from Pega's state.

```

function PocExtensionsTest(props: PocExtensionsTestProps) {
    const { getPConnect, value, disabled, readOnly, required, hideLabel, testId } = props;

    const pConn = getPConnect();
    const actions = pConn.getActionsApi();
    const stateProps = pConn.getStateProps() as StateProps;
    const propName: string = stateProps.value;

    const handleOnChange = (event: any) => {
        const { value: updatedValue } = event.target;
        actions.updateFieldValue(propName, updatedValue);
    };
}

```

4. Render the Component

Render the radio button group, passing all necessary props and handlers. This ensures the component is interactive and accessible.

```

return (
  <StyledPocExtensionsTestWrapper>
    <RadioButtonGroup name="radioGroup">
      <RadioButton
        value="SignUp"
        label="SignUp"
        labelHidden={hideLabel}
        disabled={disabled}
        readOnly={readOnly}
        required={required}
        onChange={handleOnChange}
        checked={value === "SignUp"}
        testId={` ${testId}-SignUp`}
      />
      <RadioButton
        value="Login"
        label="Login"
        labelHidden={hideLabel}
        disabled={disabled}
        readOnly={readOnly}
        required={required}
        onChange={handleOnChange}
        checked={value === "Login"}
        testId={` ${testId}-Login`}
      />
    </RadioButtonGroup>
  </StyledPocExtensionsTestWrapper>
);
};

```

6. Export the Component

Export the component using Pega's withConfiguration HOC for integration.

```

export default withConfiguration(PocExtensionsTest);

```

After developing the component, publish it to Pega and start using it within the application. Once in use, check how it renders in the UI, ensuring it works as expected and interacts correctly with user inputs.

Configure field: Test ⓘ

Display as

Test



Label value

.Test

Placeholder

Visible

Always



Disabled

Never



Required

Never



Output Screen:

- ☒ SignUp
- ☐ Login

Integrating NPM Packages to Build and Deploy Custom DX Components in Pega

For the project setup of a custom DX component, refer to the “**Building Custom React DX Components from Scratch**” document up to the updates of the **task.config.json** file for authorization with Pega.

If any other UI components are used from an external source via npm, you need to install that package in your node_modules. For example, if you want to use the “**react-open-weather**” package to check the output preview or any prerequisite props, you should visit npmjs.com. There, you'll find explanations on how to use the package, along with a preview of the output and code implementation.

I am currently using the **react-open-weather** package. Below, I will provide the required props and a preview of the output UI component.

This package supports three different weather data providers:

- **OpenWeather** (useOpenWeather)
- **WeatherBit** (useWeatherBit)
- **Visual Crossing** (useVisualCrossing)

In this case, I am using the **WeatherBit** provider through the useWeatherBit function provided by the **react-open-weather** package.

To use this provider, visit the [WeatherBit website](#) to obtain an API key.

The output preview is shown in the image below.



To use the **WeatherBit** provider from the react-open-weather package, first import the required modules and then call useWeatherBit with the appropriate props.

```
import ReactWeather, { useWeatherBit } from 'react-open-weather';

const { data, isLoading, errorMessage } = useWeatherBit({
  key: 'YOUR-API-KEY',
  lat: '48.137154',
  lon: '11.576124',
  lang: 'en',
  unit: 'M', // values are (M,S,I)
});
```

To use the **react-open-weather** package in your project, you need to install it first. Run the following script in your terminal:

```
>npm install react-open-weather
```

After the installation is complete, you can check that the package has been successfully added by looking in your package.json file. You should see the package listed with its version under the dependencies section, like this:


```
"react-open-weather": "^1.3.8",
```

When creating the component, choose the following options:

1. **Component Type:** Select **Widget**.
2. **Subtype:** Select **Page** because this is a weather component that will be used across the entire application to display weather reports.

```
PS C:\Users\DineshMaddi\Poc_Projects\dxcomponents\dxextensions> npm run create
DX Component Builder v24.2.13
? Enter type of component Widget
? Enter subtype of component
  1) CASE
  2) PAGE
  3) PAGE & CASE
Answer: █
```

After that, add your **Name** and the **Label** for the component. These are required fields. For all other fields, simply press the **Enter** key to skip them.

Next, go to the config.json file and initialize the properties for **latitude** and **longitude**. This allows you to reuse these values throughout the entire component.

To retrieve the current latitude and longitude, you need to pass these values to the react-open-weather package, which will also use them to fetch city data. These values are essential for passing to the weather API.

Add the following properties in your config.json file:

```
"properties": [  
  {  
    "name": "defaultLatitude",  
    "label": "Default Latitude",  
    "format": "TEXT"  
  },  
  {
```

```
"name": "defaultLongitude",

"label": "Default Longitude",

"format": "TEXT"

}

]
```

Next, declare the module for react-open-weather to define the required props. Create a file named react-open-weather.ts, where you can initialize all the required props, as shown in the image below.

```
1  declare module 'react-open-weather' {
2    export interface WeatherData {
3      current: {
4        temperature: number;
5        windSpeed: number;
6        humidity: number;
7        description: string;
8        icon: string;
9      };
10     forecast?: Array<{
11       date: string;
12       temperature: number;
13       description: string;
14       icon: string;
15     }>;
16   }
17   export interface WeatherBitOptions {
18     key: string;
19     lat: string;
20     lon: string;
21     lang?: string;
22     unit?: 'M' | 'S' | 'I';
23   }
```

```

24   export interface ReactWeatherProps {
25       data: WeatherData;
26       lang?: string;
27       locationLabel: string;
28       unitsLabels: {
29           temperature: string;
30           windSpeed: string;
31       };
32       showForecast?: boolean;
33       isLoading?: boolean;
34       errorMessage?: string;
35   }
36   export function useWeatherBit(options: WeatherBitOptions): {
37       data: WeatherData | null;
38       isLoading: boolean;
39       errorMessage: string | null;
40   };
41
42   export default function ReactWeather(props: ReactWeatherProps): JSX.Element;
43   }

```

After creating the react-open-weather.ts file and defining the props, the next step is to import this module into your main code file, index.tsx.

```

import { useEffect, useState } from 'react';
import { withConfiguration, Card, CardContent } from '@pega/cosmos-react-core';
import ReactWeather, { useWeatherBit } from 'react-open-weather';
import type { PConnFieldProps } from './PConnProps';
import StyledPocExtensionsWeatherwidgetWrapper from './styles';

```

You need to import the function from react-open-weather.ts and use the props defined in that file. You can extend the interface class in the react-open-weather.ts file to handle the required props.

Next, initialize the necessary props for useWeatherBit, such as **latitude**, **longitude**, and **location label**. These values should not be hardcoded. Instead, you will fetch the user's current location (latitude and longitude) and use them to retrieve the city information based on the coordinates. These props will then be passed to the useWeatherBit function to fetch the weather data.

```
function PocExtensionsWeatherwidget(props: PocExtensionsWeatherwidgetProps) {
  const { defaultLatitude = '17.4065', defaultLongitude = '78.4760' } = props;
  const [latitude, setLatitude] = useState<string>(defaultLatitude);
  const [longitude, setLongitude] = useState<string>(defaultLongitude);
  const [locationLabel, setLocationLabel] = useState<string>('Loading location...');
```

To get the **latitude** and **longitude**, you can use the built-in browser function `navigator.geolocation.getCurrentPosition`. This function retrieves the user's current geographic position.

You can then pass these latitude and longitude values to an API to fetch the address corresponding to these coordinates. Below is an example of how to achieve this:

```
useEffect(() => {
  navigator.geolocation.getCurrentPosition(
    async (position) => {
      const lat = position.coords.latitude.toString();
      const lon = position.coords.longitude.toString();
      setLatitude(lat);
      setLongitude(lon);

      try {
        const response = await fetch(
          `https://nominatim.openstreetmap.org/reverse?lat=${lat}&lon=${lon}&format=json`
        );
        const data = await response.json();
        setLocationLabel(data.address?.city || data.address?.town || 'Unknown Location');
      } catch (error) {
        console.error('Error fetching location name:', error);
        setLocationLabel('Unknown Location');
      }
    },
    (error) => {
      console.error('Geolocation error:', error);
      setLocationLabel('Location not available');
    }
  );
}, []);
```

Now that we have all the required props for the usage of **WeatherBit**, we can assign the necessary values to those props. Here's how you can pass all the values to the `useWeatherBit` function:


```
const { data, isLoading, errorMessage } = useWeatherBit({
  key: 'xxxxxxxxxxxxxxxxxxxx',
  lat: latitude || '',
  lon: longitude || '',
  lang: 'en',
  unit: 'M'
});
```

To render the weather data on the UI screen, you need to call all the functions within the return statement. This will ensure the weather information is displayed correctly on the Pega UI.

You can wrap the weather information inside a `<card></card>` to present it in a structured format. If you want to add any styling, you can include it in a `styles.ts` file, as shown in the images below.

```
return (
  <StyledPocExtensionsWeatherwidgetWrapper>
    <Card>
      <CardContent>
        <ReactWeather
          isLoading={isLoading}
          errorMessage={errorMessage || undefined}
          data={data}
          lang="en"
          locationLabel={locationLabel}
          unitsLabels={{ temperature: 'C', windSpeed: 'Km/h' }}
          showForecast
        />
      </CardContent>
    </Card>
  </StyledPocExtensionsWeatherwidgetWrapper>
);
```

```
import styled, { css } from 'styled-components';

export default styled.div(() => {
  return css`
    margin: 0px 0;

    .rw-container {
      width: 100%;
      min-height: 400px;
    }

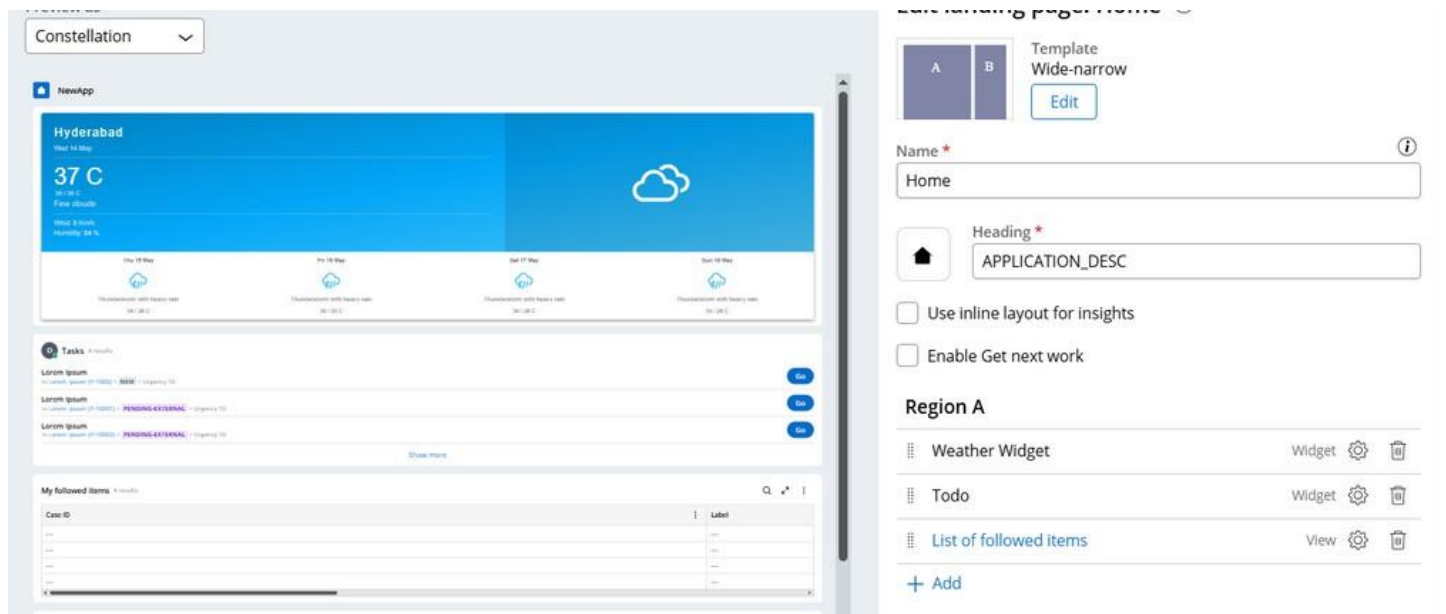
    .card-content {
      padding: 1rem;
    }
  `;
});
```

Next do for publishing your component

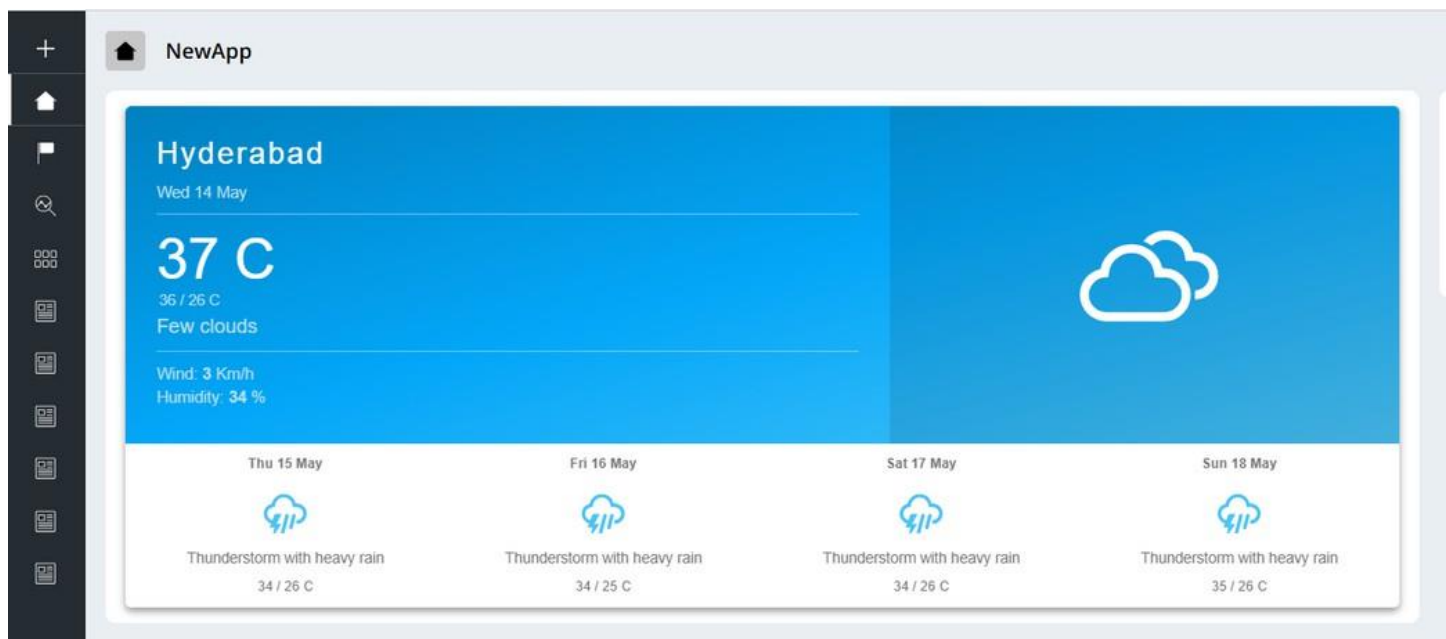
It will first validate and build your component, and then publish it. This ensures the component is properly packaged and ready for use.

>npm run publish

Now, you can call your widget on the Pega landing page by referencing the component name you created. To do this, simply include the widget in the appropriate section of your Pega landing page using the component name.



After successfully calling the widget component on the Pega landing page, you can go to the **Pega portal** to preview the **weather report widget**. This will allow you to see how the component is rendered and verify that the weather data is displayed correctly.



Constellation-UI-Gallery Component

To import the **Constellation-UI-Gallery**, we need to clone the Pega repository for this component using the following command in the terminal:

```
> git clone https://github.com/pegasystems/constellation-ui-gallery.git
```

After cloning the repository, navigate to the project directory and install the necessary packages listed in the package.json file by running:

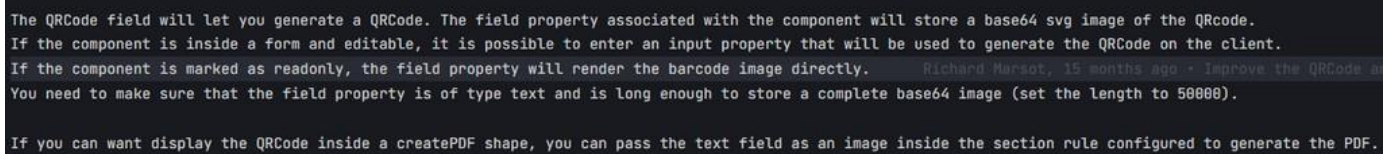
```
> npm install
```

Open the task.config.json file from the cloned repository, then update it with your client credentials, instance URL, and the ruleset. These updates are required to authorize the connection with Pega.

I have selected the **Pega_Extensions_QRCode** to publish in Pega. Here are the steps to follow:

Pega_Extensions_QRCode is one of the reusable components created by Pega for real-life scenarios. The implementation of **Pega_Extension_QRCode** can be seen below.

Requirements of using this component check in the DOC.mdx it will provide the component prerequisites to do in pega and how it will work in pega you can check in that.



The QRCode field will let you generate a QRCode. The field property associated with the component will store a base64 svg image of the QRcode. If the component is inside a form and editable, it is possible to enter an input property that will be used to generate the QRCode on the client. If the component is marked as readonly, the field property will render the barcode image directly. Richard Marsot, 15 months ago · Improve the QRCode a You need to make sure that the field property is of type text and is long enough to store a complete base64 image (set the length to 50000). If you can want display the QRCode inside a createPDF shape, you can pass the text field as an image inside the section rule configured to generate the PDF.

On the Pega side, for this component, you need to specify which property the QR code should be rendered in. This means you need to update the property to change the maximum length from 256 to 50,000 to accommodate the base64 value in that text property. Once this is done, you can call the property wherever needed within Pega, and it will display as a QR code.

The **config.json** file contains the Pega-side properties that should be used in the code. These properties need to be defined in this file. For example, in this component, the **Input Property** is declared on the Pega side, which holds the user input. This input will then be rendered as a QR code. When the QR code is scanned, it will retrieve the value of the input property.

```
{  
  "key": "inputProperty",  
  "format": "PROPERTY",  
  "name": "inputProperty",  
  "required": true,  
  "label": "Input Property"  
}
```

The demo.stories.tsx file contains a sample QR code showing the UI with hard-coded values for the InputProperty, and by extending PconnProps, the image generation code is also available in the index.tsx file. The functionality is invoked through props, which generates the image based on the provided text.

The index.tsx file first imports the CSS style function and the required Pega packages to use the Pcore and Pconnect methods in our components.

Next, the necessary props are initialized in the code to store and use the required values.

```

type QRCodeCompProps = {
  label: string;
  value: string;
  inputProperty: string;
  helperText?: string;
  validatemessage?: string;
  hideLabel: boolean;
  readOnly?: boolean;
  testId?: string;
  displayMode?: string;
  getPConnect: any;
};

```

To perform actions, the functions are initialized as follows:

```

const pConn = getPConnect();

const [outputValue, setOutputValue] = useState(value);

const [info, setInfo] = useState(validatemessage || helperText);

const [status, setStatus] = useState<'success' | 'warning' | 'error' | 'pending' | undefined>(
  undefined
);

const actions = pConn.getActionsApi();

const propName = pConn.getStateProps().value;

```

For the above function to trigger any action, we need to implement the logic inside the useEffect hook, as shown below:

```

useEffect(() => {
  if (!readOnly) {

```

```
if (validatemessage !== "") {  
  
  setStatus('error');  
  
}  
  
if (status !== 'success') {  
  
  setStatus(validatemessage !== "" ? 'error' : undefined);  
  
}  
  
setInfo(validatemessage || helperText);  
  
}  
  
}, [inputProperty, validatemessage, helperText, readOnly, status]);
```

This `useEffect` will monitor changes in `inputProperty`, `validatemessage`, `helperText`, `readOnly`, and `status`. If `validatemessage` is not empty, it sets the status to 'error'. It also updates the status and info based on the conditions defined inside the hook.

Then after display QRCode with the output value base64 taken as source of image to display then we pass our props to any reference having for rendering then we pass in the return function like this

```

return (
  <StyledWrapper>
    <Flex container={{ direction: 'column', justify: 'center', alignItems: 'center' }}>
      <FormField
        label={label}
        labelHidden={hideLabel}
        info={info}
        status={status}
        testId={testId}
      >
        <FormControl ariaLabel={label}>
          {readOnly ? (
            <img alt='QR Code' src={outputValue} />
          ) : (
            <QRCode
              value={inputProperty}
              label={label}
              onLoad={(event: SyntheticEvent<HTMLImageElement, Event>) => {
                const blob = (event.currentTarget as HTMLImageElement)?.src;
                if (blob && propName) {
                  actions.updateFieldValue(propName, blob);
                  setOutputValue(blob);
                }
              }}
            />
          )}
        </FormControl>
      </FormField>
    </Flex>
  </StyledWrapper>
);
};

```

To publish the UI Gallery component in Pega, follow these steps. In the terminal, run the following script: `>npm run publish`

This will perform all necessary validations and build the component for successful publishing in Pega.

```

PS C:\Users\DineshMaddi\Poc_Projects\constellation-ui-gallery> npm run publish

> pega-constellation-ui-gallery@2.0.0 publish
> custom-dx-components publish

DX Component Builder v24.2.13
? Select component to publish
 30) Pega_Extensions_NetworkDiagram
 31) Pega_Extensions_OAuthConnect
 32) Pega_Extensions_PasswordInput
 33) Pega_Extensions_QRCode
 34) Pega_Extensions_RangeSlider
 35) Pega_Extensions_RatingLayout
 36) Pega_Extensions_Scheduler
(Move up and down to reveal more choices)
Answer: █

```

After successfully publishing the component in the correspondence ruleset specified in Pega, the component will be available only within that ruleset as defined in the task.config.json file.


```

✓ Validate config schema
✓ Lint component
✓ Bundle Component (dev build)
✓ Zip Component
✓ Publish Component

Pega_Extensions_QRCode schema is valid
  Linting C:\Users\DineshMaddi\Poc_Projects\constellation-ui-gallery\src\components\Pega_Extensions_QRCode
=====

WARNING: You are currently running a version of TypeScript which is not officially supported by
@typescript-eslint/typescript-estree.

You may find that it works just fine, or you may not.

SUPPORTED TYPESCRIPT VERSIONS: >=4.7.4 <5.6.0

YOUR TYPESCRIPT VERSION: 5.8.3

Please only submit bug reports when using the officially supported version.

=====
  Linting complete...
Creating a development build...
stats:223 assets
orphan modules 4.18 MiB [orphan] 570 modules
runtime modules 1.14 KiB 6 modules
built modules 759 KiB [built]
  modules by path ./node_modules/ 756 KiB 234 modules
  modules by path ./src/components/ 2.41 KiB
    ./src/components/Pega_Extensions_QRCode/index.tsx 2.07 KiB [built] [code generated]
    ./src/components/Pega_Extensions_QRCode/styles.ts 175 bytes [built] [code generated]
    ./src/components/create-nonce.ts 173 bytes [built] [code generated]
  external "React" 42 bytes [built] [code generated]
  external "ReactDOM" 42 bytes [built] [code generated]
webpack 5.74.0 compiled successfully in 74755 ms
Compiled successfully.

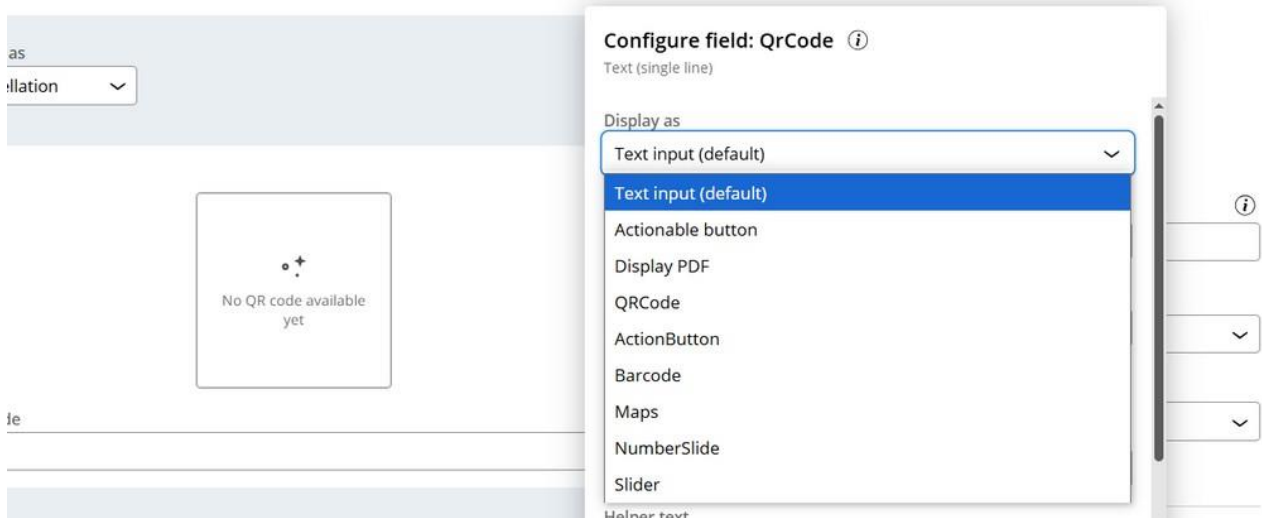
File sizes after gzip:
562.13 kB  components\Pega_Extensions_QRCode\Pega_Extensions_QRCode.js
component zipped with size: 557 KB
Success : Component successfully created in ruleset ConstellationUIGallery:01-01-01

```

In pega side go to your ruleset and check the component is published for using the component in pega.

ConstellationUIGallery 01-01-01							Refresh	Export	×
Rule type	Rule name	Applies to	Available	Updated by	Last updated	Circumstance			
Component	Pega_Extensions_ActionableButton		Yes	dinesh@pega.com	18 days ago				
Component	Pega_Extensions_Calendar		Yes	dinesh@pega.com	21 days ago				
Component	Pega_Extensions_CheckboxTrigger		Yes	dinesh@pega.com	21 days ago				
Component	Pega_Extensions_DisplayPDF		Yes	dinesh@pega.com	20 days ago				
Component	Pega_Extensions_Map		Yes	dinesh@pega.com	19 days ago				
Component	Pega_Extensions_QRCode		Yes	dinesh@pega.com	5 minutes ago				

So create a field property to show QR Code for the create a Text Property to hold the QR Code with the maximum value 50,000 and for this give a Input Property which that given value to when scan the input property value will show.



Then change you display as with your component QrCode and select the Input Property to hold the value.

Input settings

Helper text

Input Property *

I am giving the value as given image below

Enter Your Mail Address

You can test the QrCode below of the image and above Email Address:



This document is prepared by **Dinesh Maddi**

Associate Software Engineer- PEGA Consultant

Truvio Systems

