Constellation DX components

22 October 2025

©2025 Pegasystems Inc

CONTENTS

Constellation DX components 5
Release notes 8
Getting started with Constellation DX components 23
Design requirements for Constellation DX components. 23
Impact of Pega Platform '25 update on Constellation DX components. 26
Initializing a project. 28
Configuring the tasks.config.json file. 32
Updating the version of Constellation DX Component Builder. 51
How do Constellation DX components work 55
Types of Constellation DX components 63
Layout templates. 65
Widgets. 71
Fields. 74
Choosing the type of a Constellation DX component. 75
Component examples provided by Constellation DX Component Builder. 76
Library mode 84
Developing a library mode component 89
Workflow of a library mode component. 91
Initializing library mode. 92
Creating a component library. 93
Editing a library mode component. 99
Creating Storybook mocks and unit tests for library mode components. 110
Previewing library mode components in Storybook. 113
Validating library mode components. 114
Authenticating Pega Platform Server user access for library mode components. 115
Publishing a component library. 118
Adding library mode components to an application. 122
Debugging the component library. 124

©2025 Pegasystems Inc

Localizing a library mode component. 127
Commands used in library mode. 128
Working with components in library mode. 141
Developing a standalone component 153
Disabling library mode in DXCB 25.1.10. 156
Workflow of a standalone component. 157
Creating a standalone component. 158
Editing a standalone component. . . . . .
Creating Storybook mocks for standalone components. 164
Previewing standalone components in Storybook. 167
Building standalone components. 168
Authenticating Pega Platform Server user access for standalone components. 170
Publishing standalone components. 172
Configuring a standalone component for App Studio. 176
Testing a standalone component with Pega Platform. 177
Localizing a standalone component. 179
Advanced webpack configuration. 179
Commands used for standalone components. 184
Additional standalone Constellation DX component commands. 188
Renaming Constellation DX components. 189
Listing Constellation DX components. 191
Deleting Constellation DX components. 192
Listing commands. 194
Importing a Constellation DX component. 194
Component definition 196
Properties. 203
Formats. 208
Annotations. 231
Expressions. 236
Working with PCore and PConnect public APIs 242
Working with Constellation Design System 244

PEGA ©2025 Pegasystems Inc

Tutorials 246
Field component 247
Creating a field 248
Editing the field 250
Mocking and testing the field with a Storybook 256
Publishing the field 260
Using the field in App Studio 261
Widget component 262
Creating a widget 263
Reviewing the structure of a widget 265
Editing the widget 272
Mocking and testing the widget with a Storybook 285
Styling the widget 289
Publishing the widget 303
Using the widget in App Studio 304
Layout Template component 305
Creating a layout template 306
Reviewing the structure of a layout template 309
Editing the layout template 325
Mocking and testing the layout template with a Storybook 327
Publishing the layout template 343
Using the layout template in App Studio 344
Video learning series 349
Troubleshooting Constellation DX Component Builder 351

PEGA ©2025 Pegasystems Inc

Constellation DX components

While the Constellation design system architecture has an extensive library of out-of-the-box assets, the framework provides tools for extending the Constellation user experience by creating Constellation digital experience (DX) components. This still-evolving framework helps early adopters build interfaces tailored to their specific business needs.

While using DX components in the Constellation design system, keep in mind that it is an expanding product with a narrower feature set. Consequently, the framework might not support some classic Constellation features. You might also encounter unforeseen issues.

> **NOTE:** This information is intended only for professional front-end developers in advanced use cases where out-of-the-box components do not meet the users' needs. To create Constellation DX components, you need knowledge in Constellation design system development and common development tools such as NPM and Git.
>
> The Constellation design system is designed to address most enterprise design workflow needs through out-of-the-box assets. Low-code developers, business architects, and UX designers do not create Constellation DX components in the Constellation design system. For more information on working with out-of-the-box Constellation assets, see Configuring Constellation .

To maintain a consistent, branded experience across channels, Constellation applications rely on the Constellation design system for styling. The Constellation DX component toolkit aims to extend the library of existing Constellation components and create new components that integrate seamlessly within the set of out-of-the-box components. You can use Constellation DX components to expand the Constellation

PEGA ©2025 Pegasystems Inc

design system in specific scenarios where out-of-the-box components might not meet immediate business needs.

Overview

The Constellation DX component architecture supports extending the out-of-the-box Constellation user experience with the ability to create your own components based on a composite of the existing Constellation design system components. The Constellation DX components fit seamlessly into the out-of-the-box features of Constellation. When creating your Pega Platform™ application, you will use your own Constellation DX components in the same way as you use the standard components. The style of your Constellation DX components closely matches that of the out-of-the-box Constellation design system components.

You can create, modify, and publish your own components by using the Pega *ConstellationJS* APIs and the Constellation design system UI components library. Your components are available in App Studio alongside the out-of-the-box components.

> **NOTE:** Unlike Constellation DX components, you cannot modify the out-of-the-box Constellation design system components.

Library mode components vs standalone components

With the introduction of library mode in DXCB 25.1.10, Constellation DX components can be published to a library. All components created in DXCB 25.1.10 in library mode are called library mode components, while components created in DXCB 25.1.10 after disabling library mode or created before DXCB 25.1.10 are called standalone components.

Library mode is the preferred mode of working with Constellation DX components as it has several advantages over standalone components. For more information, see Library mode.

©2025 Pegasystems Inc

If you are using DXCB 25.1.10 and want to import your standalone components to library mode, see Importing and publishing standalone components. If you are using DXCB 25.1.10 and want to use only standalone components, see Disabling library mode in DXCB 25.1.10.

©2025 Pegasystems Inc

Constellation DX Component Builder release notes

Learn what's new in Constellation DX Component Builder (DXCB) to know about the features that help you build interfaces tailored to your business requirements.

Constellation DX Component Builder 25.1.10

Constellation DX Component Builder 25.1.10 is a release of the Constellation DX Component Builder intended for Pega Platform™ '25. The version x.x.10 is specific to the Constellation DX Component Builder release for a major or minor Pega Platform release.

> **NOTE:** DXCB 25.1.10 has been tested with node version 22.15.0 and npm version 10.9.2.

Released in September 2025, DXCB 25.1.10 contains the following changes:

General changes

- Introducing library mode that enables you to publish Constellation DX Components to a library. For more information, see Library mode.

> **NOTE:** Starting from DXCB 25.1.10, library mode is the default mode of working with Constellation DX components. All components created in DXCB 25.1.10 in library mode are called library mode components, while components created in DXCB 25.1.10 after disabling library mode or created before DXCB 25.1.10 are called standalone components.

©2025 Pegasystems Inc

- Supports the typedefs package version 4.1.0.
- Constellation design system 8 is now supported for working with Pega Platform '25.
- Templates have been updated to ESLint 9. The `eslint.config.mjs` flat config file replaces the `eslintrc.json` file. This will require the existing component code to be updated.
- The following properties have been added to the `tasks.config.json` file:
  - `libraryModeCL`
  - `devServerPort`
  - `archiveRulesetName`
  - `archiveRulesetVersion`

  For more information, see Configuring the tasks.config.json file.

Component changes

- Library mode is enabled by default.
- Several commands have been introduced for library mode. For more information, see Commands used in library mode.
- Standalone components and component libraries are Rule resolved through their Ruleset or ruleset version.
- The AppStatic service is no longer supported. The component design-time information and run-time code is now obtained from the Pega Platform server. Components are Rule resolved through their ruleset version. For more information, see Impact of Pega Platform '25 update on Constellation DX components and Rule resolution for Constellation DX components.
- The `allowedApplications` attribute in the `config.json` file is no longer used. Components are available to an application only if its ruleset version is included in the application stack.
- Component names now have a limit on the number of the characters. The limit is determined based on the length of the organization, library, and version. The character limit for standalone components is higher than that of the library mode components.

PEGA ©2025 Pegasystems Inc

Constellation DX Component Builder 24.2.10

Constellation DX Component Builder 24.2.10 is a release of the Constellation DX Component Builder specifically for Pega Platform 24.2.x. The version x.x.10 is specific to the Constellation DX Component Builder release for a major/minor Pega Platform release.

> **NOTE:** DXCB 24.2.10 has been tested with node version 20 and npm version 10.

Released in November 2024, DXCB 24.2.10 contains the following changes:

General changes

- Supports the typedefs package version 3.2.1.
- Constellation design system 7 is now supported for working with Pega Platform '24.2.
- The parameter types for several functions have been changed from `any` to a specific type. If you encounter type errors, change the declaration of the parameter to the new type. You must also declare asynchronous promise responses. For more information about the type errors and how to resolve them, see Troubleshooting Constellation DX Component Builder.

  For example, if the type for the `myProp` parameter has changed from `any` to `string`, the declaration must be changed from `const myProp: any;` to `const myProp: string;`.

- The following properties have been added to the `tasks.config.json` file:
  - `showDebug`
  - `import-relative-path`
  - `export-relative-path`

  For more information, see Configuring the tasks.config.json file.

©2025 Pegasystems Inc

Component changes

- Newly created components will contain fewer `//@ts-ignore` statements in their code. If you encounter errors for your existing components, remove `//@ts-ignore` from your code.
- The following component commands have been introduced:
  - `npm run help`
  - `npm run importComponent`

  For more information, see Additional standalone Constellation DX component commands.
- The library and version details entered when creating a component are now updated in the `tasks.config.json` file and will be the default values when the next component is created.
- When initializing a component, you are no longer asked for the version number and Git repository URL details.

Constellation DX Component Builder 24.1.10

Constellation DX Component Builder 24.1.10 is a release of the Constellation DX Component Builder specifically for Pega Platform 24.1.x. The version x.x.10 is specific to the Constellation DX Component Builder release for a major/minor Pega Platform release.

> **NOTE:** DXCB 24.1.10 has been tested with node version 20 and npm version 10.

Released in May 2024, DXCB 24.1.10 contains the following changes:

General changes

- Constellation design system 6 is now supported for working with Pega Platform '24.1.

©2025 Pegasystems Inc

- More type definitions have been added as part of the typedefs package version 3.0.0.
- Webpack has been updated to handle the icons in Constellation design system 6.0.3.

NOTE:

- If you are using a local copy of the webpack, add the following code to the `webpack.config.js` file:

  Other OS

  ```
  new webpack.IgnorePlugin({
  resourceRegExp: /^\.\/.*\/.*\.icon$/,
  contextRegExp: /@pega\/cosmos-react-core\/lib\/components\/Icon$/
  }),
  ```

  For Windows

  ```
  new webpack.IgnorePlugin({
  resourceRegExp: /^\.\/.*\/.*\.icon$/,
  contextRegExp: /@pega\\cosmos-react-core\\lib\\components\\Icon$/
  }),
  ```

- If you have completely modified your current webpack, make a copy of the webpack, and run the `npm run promoteWebpack` command. You can now view the updated code of the internal `webpack.config.js` file and understand what must be changed in the local webpack.

©2025 Pegasystems Inc

Component changes

- The Icon Button URL component has been introduced as a sub type of the Field component.

> **NOTE:**
> - A URL field can be displayed in App Studio as either an Icon Button URL component or a URL component.
> - The Icon Button URL component has been introduced for educational purposes and must not be used as-is. You can customize the Icon Button URL component and use it.
> - Go through the component code for information on how to do the following:
> - Importing, registering, and using an icon.
> - Inserting an icon in a button.
> - Inserting an icon in a link.

- The Details, Details-region, Form, and Form-region components have been updated. The Details component can now be added as a read-only component to a Form component. The Form component can now be added as a read-only widget to a Details component.

> **NOTE:** As per the current design of the Constellation design system, the custom Details and Form components will work as intended with out-of-the-box components and Constellation DX components, and the out-of-the-box Details component will not work with out-of-the-box components and Constellation DX components. If you want to use a custom Details component within the context of a Form component, you must specifically create a custom Details component.

©2025 Pegasystems Inc

- The PAGE Widget component has been updated to use PCore and PConnect APIs.

Bug fixes

- Field components now render as intended in the DISPLAY_ONLY mode.
- The displayMode functionality of the Text component now functions similarly as in all other components.

Constellation DX Component Builder 23.1.12

Constellation DX Component Builder 23.1.12 is a release of the Constellation DX Component Builder specifically for Pega Platform 23.1.x. The version x.x.12 is a specific Constellation DX Component Builder release for a major/minor Pega Platform release.

Released in April 2024, DXCB 23.1.12 contains the following changes:

Component changes

- All template files now use the `.ts/.tsx` extension.
- All internal script files now have an `_internal-` prefix appended to their names to help you identify them.

> **NOTE:** Do not run internal script files as your DX components could stop working.

- Generated components are now built to use the `withConfiguration` higher order component that automatically wraps the component with a configuration object. Stylesheets have been updated to remove the configuration object that was previously used.
- Storybooks now use controls instead of knobs.
- `prop-types` have been removed due to the conversion to TypeScript. All generated components now use the TypeScript interface. Since all default

©2025 Pegasystems Inc

values that were previously a part of `prop-types` have been migrated to the deconstruction of `props` in the main function, ensure that you add the default values within `props`.

- Introduced the `usePromotedWebPack` flag to indicate the webpack file that is used for a component. For more information, see usePromotedWebPack.

Bug fixes

- If you encounter a lint error or build error while running the `buildAll` or `publishAll` commands, the process of building or publishing the components will stop until the error is resolved.
- The Details component no longer has a hard coded name when viewed in Storybook.

Constellation DX Component Builder 23.1.11

Constellation DX Component Builder 23.1.11 is a release of the Constellation DX Component Builder specifically for Pega Platform 23.1.x. The version x.x.11 is a specific Constellation DX Component Builder release for a major/minor Pega Platform release.

Released in February 2024, DXCB 23.1.11 contains the following changes:

- Component code is generated in TypeScript as compared to JavaScript in the previous release.
- IDE code completion is now possible for PCore and PConnect (ConstellationJS orchestration engine) APIs using TypeScript type definitions. For more information, see Type definitions.
- Constellation design system 4.2.1 is supported. This provides access to additional components in the Constellation design system.

> **NOTE:** If you have used the Glimpse component in your custom code before updating to Constellation DX Component Builder 23.1.11, you

©2025 Pegasystems Inc

> must change its structure. For more information, see Troubleshooting DX Component Builder.

- The ability to create Search box (AutoComplete) and RadioButtons components has been introduced.
- The rules in ESLint have been updated so that it is more aligned with the Constellation design system.
- Many components of the `package.json` file have been updated to a recent version to eliminate vulnerabilities and keep the code up to date. Storybook has been updated from version 6.x.x to 7.x.x.
- The Prettier code formatter is now supported to help keep the code style consistent across the codebase.

Constellation DX Component Builder 23.1.10

Constellation DX Component Builder 23.1.10 is a release of the Constellation DX Component Builder specifically for Pega Platform 23.1.x. The version x.x.10 is a specific Constellation DX Component Builder release for a major/minor Pega Platform release.

> **NOTE:** DXCB 23.1.10 has been tested with node version 20 and npm version 10.

Released in December 2023, DXCB 23.1.10 contains the following changes:

- Supports Constellation design system 4.0.1 that is used in Pega Platform 23.1.
- Uses ConstellationJS orchestration engine 8.23.0 that ships with Pega Platform 23.1.

Constellation DX Component Builder 8.8.20

Constellation DX Component Builder 8.8.20 is a release of the Constellation DX Component Builder specifically for Pega Platform 8.8.x. The version x.x.20 is a specific Constellation DX Component Builder release for a major/minor Pega Platform release.

PEGA ©2025 Pegasystems Inc

Released in October 2023, DXCB 8.8.20 contains minor bug fixes.

Constellation DX Component Builder 8.8.19

Constellation DX Component Builder 8.8.19 is a release of the Constellation DX Component Builder specifically for Pega Platform 8.8.x. The version x.x.19 is a specific Constellation DX Component Builder release for a major/minor Pega Platform release.

Released in September 2023, DXCB 8.8.19 contains the following key features and changes:

- The `promoteWebPack` and `revertWebPack` NPM commands related to the bundling process have been introduced.

  For more information, see Commands used for standalone Constellation DX components.

- You can now use the `npm-quick-run` utility to run NPM commands.

  For more information, see Executing commands.

- Component templates now support Pega themes by default by wrapping the component using the Constellation design system Configuration.

  For more information, see https://design.pega.com/develop/configuration/.

- As node 18 does not support legacy TLS, we have added a configuration option to manually add support to legacy TLS.

  For more information, see Troubleshooting Constellation DX Component Builder.

- In order to support externalizing the webpack build process, `"type":"module"` has been added to the `package.json` file.

PEGA ©2025 Pegasystems Inc

> **NOTE:** The version of the `webpack.config.js` file along with the components is changed to ES6 modules that use import statements for importing modules.
>
> Existing components that use require statements must comply with either of the following conditions:
>
> - The components must be updated to use import statements.
> - The `package.json` file must be demoted by deleting `"type":"module"`.

For more information, see Advanced webpack configuration.

Constellation DX Component Builder 8.8.10

Constellation DX Component Builder 8.8.10 is a release of the Constellation DX Component Builder specifically for Pega Platform 8.8.x. The version x.x.10 is a specific Constellation DX Component Builder release for a major/minor Pega Platform release.

> **NOTE:** DXCB 8.8.10 has been tested with node version 20 and npm version 8.

Released in May 2023, DXCB 8.8.10 contains the following key features and changes:

Sample code generated for Constellation DX components

Extend your out-of-the-box Constellation UX experience with Constellation DX components that enable you, as front-end developers, to jump-start your React component development. The Component Builder has a comprehensive set of example component code, component schema definitions (`config.json`) and Storybook stories for all DX Component Builder types and subtypes. Link Constellation Design System components with the ConstellationJS Orchestration Engine public APIs to create truly world class experiences for your end-users. For more information on the public APIs, see Using PCore and PConnect Public APIs.

©2025 Pegasystems Inc

You can now access detailed sample code and Storybook stories for the following components and their subtypes.

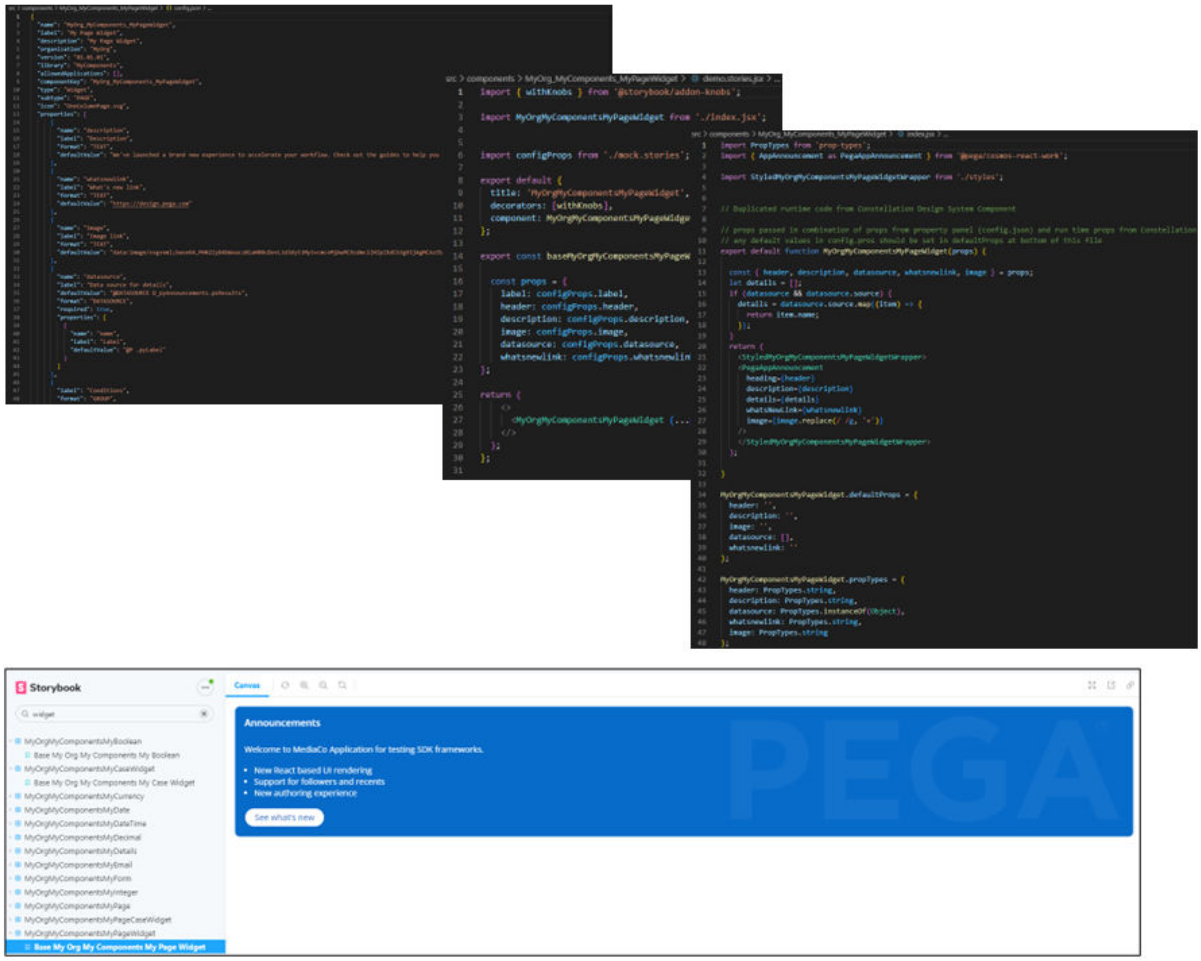
The supported components are:

Component type	Component subtype
Field	Text
Field	Text Input
Field	Paragraph
Field	Email
Field	Phone
Field	URL
Field	Integer
Field	Decimal
Field	Currency
Field	Percentage
Field	Boolean
Field	Date
Field	DateTime
Field	TimeOfDay
Field	Picklist
Widget	CASE
Widget	PAGE
Widget	PAGE & CASE
Layout template	DETAILS
Layout template	DETAILS region
Layout template	FORM

PEGA ©2025 Pegasystems Inc

Component type	Component subtype
Layout template	FORM region
Layout template	PAGE

The following image shows a preview of the sample code generated along with the Storybook story for the Widget component type.



Sample code and storybook preview - Widget component

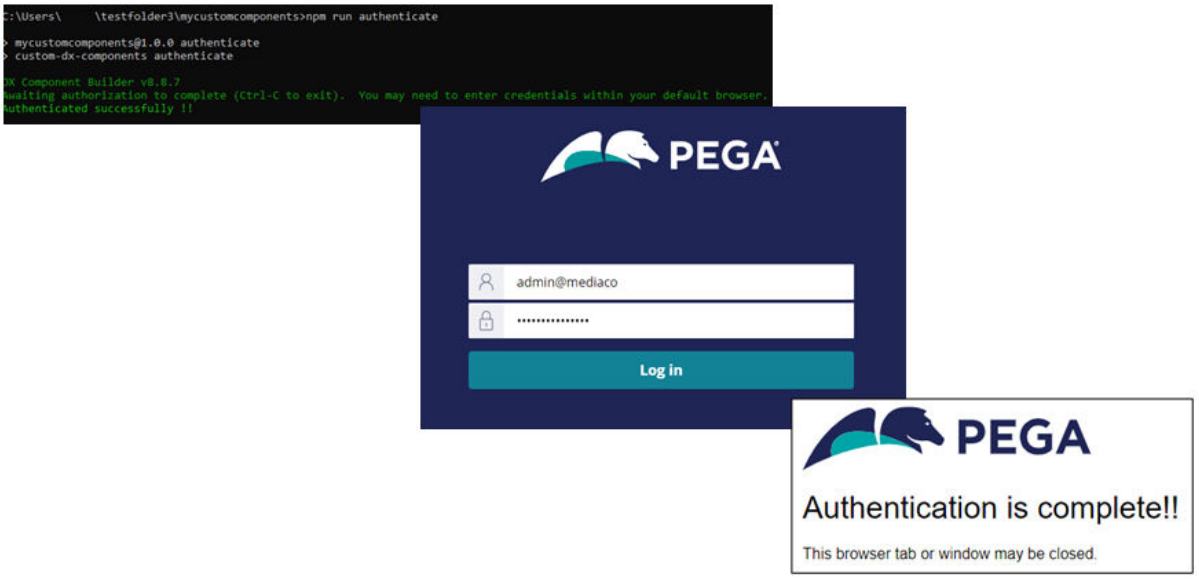
For more information on the detailed list of Constellation DX components that have sample code and are supported by Constellation DX Component Builder, see Component examples provided by Constellation DX Component Builder.

©2025 Pegasystems Inc

AuthCode support for PKCE flow for authenticating with Pega Platform

You now have authorization code with PKCE support available for authenticating the Pega Platform Server. This is available out-of-the-box and does not require you to perform any additional configuration apart from setting the Dev instance URL.

For more information on authenticating user access to the Pega Platform Server, see Authenticating Pega Platform Server user access for standalone Constellation DX components.



Authenticating user access with Pega Platform *Server*

Command line references

Several additional command line options have been introduced to enable you to perform advanced operations with the Constellation DX components using the Constellation DX Component Builder.

For more information, see Commands used for standalone Constellation DX components.

©2025 Pegasystems Inc

Node version check

Your node version is now automatically validated against the required version before you are allowed to run any commands. You no longer need to manually check if you are on the required node version.

©2025 Pegasystems Inc

Getting started with Constellation DX components

To work with Constellation DX components, you must initialize a project and update to the latest version of the Constellation DX Component Builder.

- **Design requirements for Constellation DX components**
- **Impact of Pega Platform '25 update on Constellation DX components**
- **Initializing a project**
- **Configuring the tasks.config.json file**
- **Updating the version of Constellation DX Component Builder**

Design requirements for Constellation DX components

Use the provided UX principles and guidelines to help you achieve the most optimum design for your custom Digital Experience (DX) component.

Before you begin:

Review the following checklist before you create a Constellation DX component:

- **Design considerations**

  Involve an Experience Designer in your process. This will help you determine the exact issue to be resolved. For more information on the role of an Experience Designer, see Experience Designer.

  The Experience Designer must meet the following requirements:

©2025 Pegasystems Inc

    - Must have a background in UI and UX design.
    - Must understand Pega's best practices for design. For more information, see Best practices for designing with the Constellation design system.
- **Functionality considerations**
    - Check if an existing component in App Studio contains the features you need. If not, check for existing components that contain similar features.
    - Connect with Pega Support to find out if a component containing a similar feature will be included in the next release. If yes, assess if you can wait for the next release.
- **Maintenance considerations**
    - Ensure you have the bandwidth to maintain the Constellation DX component and keep it up-to-date with every release of Pega Platform™. You must use a version of DXCB that is designed for the version of Pega Platform you are using. Because each DXCB update contains updated versions of node modules, Constellation design system, ESLint, and React, among others, the components must be rebuilt, tested, updated, and published to the corresponding version of Pega Platform. Evaluate if doing this might prevent you from updating to future releases of Pega Platform.
    - Evaluate if you are able to meet the compliance standards for security, data privacy, and accessibility for Constellation DX components. Note that not meeting these standards might lead to legal compliance issues.

Designing your Constellation DX component

You can evaluate and measure the quality of your Constellation DX component by using the RAILS guidelines. RAILS is an abbreviation for **R**esponsive, **A**ccessible, **I**n-the-model, **L**ocalizable, and **S**ecure.

©2025 Pegasystems Inc

Let us take a comprehensive look at each RAILS guideline:

- **Responsive**

  The component must be designed to be used in mobile devices, pages, and panels.

  From a design perspective, review how the component looks like at 320 pixels wide and 2500 pixels wide.

  For more information on this guideline, see Responsive UI.

- **Accessible**

  The component must provide compliant and equivalent experiences for all users.

  From a design perspective, review the following:n
  - How do you interact with this component using only a keyboard?
  - Have you consulted or user-tested this with someone who uses assistive technology daily?
  - Do you have automated accessibility testing and linting in place?

  For more information on this guideline, see What is accessibility?.

- **In-the-model**

  The component must be center-out, theme-able, and author-able.

  From a design perspective, review the following:
  - Can this component be configured in authoring?
  - Can we make this a more universal setting?
  - What does this look like with a different theme applied?
- **Localizable**

  The component must be understood by users from all cultures. It should be easy to translate any dates, currency, names, or other content into other languages.

PEGA ©2025 Pegasystems Inc

From a design perspective, review how the component works when used in different languages.

- **Secure**

  The component must protect the privacy of the business, its data, and its users.

  From a design perspective, review the following:
  - What type of data is being collected and shared with third parties?
  - Have you consulted your legal team about the data being collected by the component?
  - What happens when you insert an XSS attack value into an input or config setting within the component?

Design priorities

Pega recommends the following order of priority for designing a component:

1. Semantics of the web, such as, HTML tags, WAI-ARIA, CSS, JS, etc.
2. Industry-standard patterns
3. Unique features

Following this order will reduce both churn and load for users to consume new features.

Matching Constellation

If you want to create components that match the look and feel of Constellation, follow the prescribed visual design patterns and Constellation content patterns.

For more information, see the following resources:

- Design tokens
- Constellation content guidelines
- https://design.pega.com/

PEGA ©2025 Pegasystems Inc

Impact of Pega Platform '25 update on Constellation DX components

Learn about the effect on your existing Constellation DX components when you update to Pega Platform™ '25.

When you update to Pega Platform '25, all existing Constellation DX components that were created before Pega Platform '25 are automatically updated to use Rule resolution. For more information, see Rule resolution for Constellation DX components.

For an existing component that was published to a specific ruleset version for an application, if the ruleset version was added to the application, and if the component was published only to that ruleset version, the component will be updated without any issues.

The following components might encounter issues after being updated:

- Components published to multiple ruleset versions
- Components that used the allowedApplications attribute

Components published to multiple ruleset versions

Before Pega Platform '25, for an existing component that was published to multiple ruleset versions, only the version that was last published was served at run time. If you published multiple versions of your components to multiple ruleset versions, only the version that was last published would be updated, not the other versions.

When you update to Pega Platform '25, ensure that your application includes the correct ruleset version to which the component you want to use is published. To track your component and its versions, publish your component to a separate ruleset version and add that ruleset version to the application stack.

PEGA ©2025 Pegasystems Inc

Components that used the allowedApplications attribute

The `allowedApplications` attribute in the `config.json` file is not supported from Pega Platform '25 and has been replaced by Rule resolution. Before Pega Platform '25, if you used the `allowedApplications` attribute to choose the applications that can access the component or left the `allowedApplications` attribute blank to allow all applications to access the component, you might face issues after updating to Pega Platform '25.

After updating to Pega Platform '25, perform the following steps based on the value of the `allowedApplications` attribute:

If allowedApplications specifies applications that can access the component

If you used the `allowedApplications` attribute to specify the applications that can access the component, ensure that the specified applications include the ruleset version to which the component was published. To track your component and its versions, publish your component to a separate ruleset version and then add that ruleset version to the specified applications.

If you do not want to share a ruleset version among multiple applications, publish your component to multiple ruleset versions and add different ruleset versions to each of the specified applications. Note that this approach requires you to keep track of the different ruleset versions to which your component has been published.

For applications that are built on other applications, Rule resolution ensures that components that exist in an application are available to the built-on application without having to specifically include them.

If allowedApplications is blank

If you have left the `allowedApplications` attribute blank to allow all applications to access the component, you must follow Rule resolution. Publish your component to a ruleset version, and add that ruleset version to the applications to which you want to provide access to the component.

PEGA ©2025 Pegasystems Inc

Initializing a project

Set up an environment to work with Constellation digital experience (DX) components.

Before you begin:

Ensure that you have the following software installed:

- Pega Platform™ version 25 or later
- Git version 2.30 or later
- System node version 22.15.0 and npm version 10.9.2
- Please use a supported browser as detailed here.
- DXCB 25.1.10 uses ESLint 9, which uses the flat config file. To enable typechecking using the flat config file, configure Visual Studio Code to use flat config.
  1. In Visual Studio Code, click **View** > **Extensions** and install the ESLint extension.
  2. Click **File** > **Preferences** > **Settings**.
  3. In the **Search settings** field, enter **eslint**, and in the **Eslint: Use Flat Config** setting, click **Edit in settings.json**.
  4. In the `settings.json` file, set `eslint.useFlatConfig` to `true`.

  If you are using a terminal other than Visual Studio Code, see Plugin Migration to Flat Config.

If you have updated to Pega Platform 8.6 or later versions, update your **dev** service package to use OAuth 2.0 authentication. For more information, see Verifying and updating service packages.

1. Open your preferred terminal tool.
2. Enter `npx @pega/custom-dx-components@~25.1 init`.

©2025 Pegasystems Inc

> **NOTE:** If you encounter an error, set your npm Registry to point to the npm public registry by following the procedure below:
>
> a. Check the registry that your npm Registry currently points to by running the `npm config get registry` command.
> b. Point your npm Registry to the npm public registry (https://registry.npmjs.org) by running the `npm config set registry https://registry.npmjs.org/` command.
> c. Run the `npx @pega/custom-dx-components@~25.1 init` command again.

3. If prompted, enter `y` to install the Custom DX Components package.
4. Set up the project:
   a. Enter the name of the project.
   For example: `mycustomcomponents`

   > **NOTE:** The name of the project is used to create the root project folder.

   b. Enter the name of the organization.
   The organization name forms the first part of the component.

   For example: `customorg`

   > **NOTE:**
   > - Starting from DXCB 25.1.10, the length of the component's name is limited to a maximum number of characters that depends on the values of the `organization`, `library`, `version`, and a hash. To maximize the number of characters that can be used for the component's name, limit the lengths of the values for `organization` and `library`.

©2025 Pegasystems Inc

- Finalize the name of your organization during the initial development of your component library. If you change the name of the organization after several components have been published, you must update the component key for all the published components using the same organization. For more information, see Renaming Constellation DX components.
- This parameter governs publishing to NPM or a Git repository. It is not tied to the *Org* rule.

c. **NOTE:** Starting from DXCB 24.2.10, you are no longer asked for the version number.

   Enter the version number.

   For example: `1.0.0`

d. To describe the project, enter a description.

   For example: `This project is intended to create and publish a Constellation DX component.`

e. **NOTE:** Starting from DXCB 24.2.10, you are no longer asked for the Git repository URL.

   To publish the component to a Git repository, enter the Git repository URL.

   For example: `https://git.custom-components.io/custom-cc.git`

f. To name the project owner, enter a name.

   For example: `Michelle`

5. Check for vulnerabilities by running the `npm audit` command.
   This will let you know if any vulnerabilities are present.
6. Remove any vulnerabilities by running the `npm audit fix` command.
   This will check if new versions of Pega or dependent npms are available, which might fix some or all vulnerabilities.

PEGA ©2025 Pegasystems Inc

> **NOTE:** Since this application is built using open source components, vulnerabilities may occur. While you can regularly check for vulnerabilities by running the `npm audit` command, note that some vulnerabilities might not be fixed by running the `npm audit fix` command till the open source components are updated.

7. Navigate to your newly created project folder and update the default values of the library and version in the `tasks.config.json` file.
   For more information on configuring the `tasks.config.json` file, see Configuring the tasks.config.json file.

> **Result:**
>
> The environment is now set up to work with Constellation DX components.
>
> > **NOTE:** In case of errors, delete the newly created project folder and initialize a new project.

Configuring the tasks.config.json file

Configure the attributes in the `tasks.config.json` file to ensure that the server configuration aligns with your Constellation DX components.

The `tasks.config.json` file contains details of the Constellation DX component and the server attributes for configuring the component.

This is a sample `tasks.config.json` file:

```
{
  "components-directory-path": "src/components",
  "import-relative-path": "Downloads",
```

©2025 Pegasystems Inc

```
  "export-relative-path": "Downloads",
  "server-config": {
    "rulesetName": "CustomDXComponents",
    "rulesetVersion": "01-01-01",
    "sourceOfComponents": "Server",
    "devBuild": true,
    "devServerPort": "3030",
    "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 s
et to 'authCode')",
    "authService": "pega",
    "cert_and_key_comment": "Path to SSL certificate file and SSL key file to use when
starting local HTTPS server",
    "cert_and_key_comment2": "Needed only when using 'authCode' grantType and re
directUri begins with 'https://localhost'",
    "cert": "./keys/dxcb.crt",
    "key": "./keys/dxcb.key",
    "user_and_password_comment": "useful primarily when grantType is set to 'passw
ordCreds'",
    "user": "",
    "password": "",
    "questions_comment": "useful for designating how frequently questions should be
 asked during npm run authenticate",
    "questions_askAlways": "",
    "questions_askNever": "",
```

©2025 Pegasystems Inc

```
    "questions_askOnce": "server,authService",
    "questions_askedOnce": "",
    "archiveRulesetName": "CustomDXArch",
    "archiveRulesetVersion": "01-01-01"
  },
  "component": {
    "library": "YourLibrary",
    "type": "",
    "version": "1.0.0",
    "subtype": "",
    "description": "",
    "icon": ""
  },
  "usePromotedWebPack": false,
  "libraryModeCL": true,
  "showDebug": false
}
```

NOTE: This is only an example of a `tasks.config.json` file. The values of its attributes are specific to your application and should not be copied to your `tasks.config.json` file. This is specifically applicable to attributes, such as, `rulesetName`, `rulesetVersion`, `server`, `clientId`, `archiveRulesetName`, `archiveRulesetVersion`, `library`, and `version`.

The `tasks.config.json` file consists of the following sections:

Top level attributes

These attributes are part of the top level of the `tasks.config.json` file.

©2025 Pegasystems Inc

server-config

The attributes in this section define the settings required to connect to your server to manage Constellation DX components.

component

The attributes in this section define the default properties of the component.

Top level attributes

The following table describes the top level attributes in the `tasks.config.json` file:

Name	Type	Required	Default value	Description
`components-directory-path`	string	☐	`src/components`	The path of the created component.
`usePromotedWebPack`	boolean	☐	`false`	**NOTE:** `usePromotedWebPack` is not supported in library mode. Introduced in Constellation DX Component Builder (DXCB) 23.1.12, the `usePromotedWebPack` attribute indicates if the original webpack or promoted webpack file is used for a component. You can change the value of `usePromotedWebPack` to toggle between the promoted and original webpacks.

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				• `usePromotedWebPack` is not supported when library mode is enabled. • The default value of `usePromotedWebPack` is `false`. • If the value of `usePromotedWebPack` is `true`, it indicates that the promoted webpack is used. • If the value of `usePromotedWebPack` is `false`, it indicates that the original webpack is used. • When you promote a webpack using the `npm run promoteWebPack` command, `usePromotedWebPack` is set to `true`. When you revert to the original webpack using the `npm run revertWebPack` command, `usePromotedWebPack` is set to `false`. For more information, see

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				promoteWebpack and revertWebPack. **NOTE:** If you have a promoted webpack file and are updating from a version older than DXCB 23.1.12, perform the following steps: • If you are using the `npm update` command, complete the update, run the `npm run promoteWebPack` command, and enter `N`. This will ensure that the promoted webpack is used and will add the `usePromotedWebPack` attribute and set it to `true`.

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				• If you are performing a clean install, after copying your code and webpack file, run the `npm run promoteWebPack` command, and enter `N`. This will ensure that the promoted webpack is used and will set the `usePromotedWebPack` attribute to `true`.
`import-relative-path`	string	☐	`Downloads`	Introduced in DXCB 25.1.10, `import-relative-path` is a mandatory string attribute that contains the relative path for importing components. The default value of `import-relative-path` is `Downloads`. This value is

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				automatically updated through code, but can also be set manually. **NOTE:** The path entered is relative to the `/User/<username>` folder. For example, if you update the path to point to the `/User/JohnDoe/New Downloads` folder, the value entered for `import-relative-path` is `New Downloads`.
`export-relative-path`	string	☐	`Downloads`	Introduced in DXCB 25.1.10, `export-relative-path` is a mandatory string attribute that contains the relative path for exporting components. The default value of `export-relative-path` is `Downloads`. This value is automatically updated through

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				code, but can also be set manually. **NOTE:** The path entered is relative to the `/User/<username>` folder. For example, if you update the path to point to the `/User/JohnDoe/New Downloads` folder, the value entered for `export-relative-path` is `New Downloads`.
`libraryModeCL`	boolean	☐	`true`	Introduced in DXCB 25.1.10, the `libraryModeCL` attribute indicates if the library mode is enabled, which allows library component publishing. For more information, see Library mode. • The default value of `libraryModeCL` is `true`. This enables library component publishing.

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				• If `libraryModeCL` is set to `false`, standalone component publishing is enabled.
`showDebug`	boolean	☐	`false`	**NOTE:** This attribute is only for Pega-internal use. Introduced in Constellation DX Component Builder 24.2.10, the `showDebug` attribute indicates whether data, such as the functions called with parameters, is displayed in the terminal window. • The default value of `showDebug` is `false`. • If `showDebug` is set to `true`, the data is displayed in the terminal window.

`server-config`

The `server-config` section of the `tasks.config.json` file contains the settings required to connect to your server to manage Constellation DX components.

©2025 Pegasystems Inc

The following table describes the attributes in the `server-config` section:

Name	Type	Required	Default value	Description
`rulesetName`	string	☐	`"CustomDXComponents"`	The ruleset to which the component must be published.
`rulesetVersion`	numeric	☐	`01-01-01`	The version of the ruleset to which the component must be published.
`sourceOfComponents`	string	☐	`Server`	The default source location from which components can be listed or deleted. Use one of the following values: • `Local`: Sets the component source location to the local component library (`src/components/`). • `Server`: Sets the component source location to the server set in the `serverConfig.infinityRestServerUrl` attribute.
`devBuild`	boolean	☐	`false`	Determines if a development build of the component must be generated. Set this to `true` to enable troubleshooting the component in the browser.

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				NOTE: `devBuild` is not supported when library mode is enabled.
`devServerPort`	string	☐	3030	The port number for the local debugging server. It must match the port number in the URL of the **Constellation DXCB Dev URL** field of the operator record.
`serverType`	string	☐	`infinity`	The type of server - `infinity` or `launchpad`
`server`	string	☐	`https://localhost:1080/prweb`	The complete path to the Infinity REST server where your application is installed and running. **NOTE:** In Launchpad®, provide the base URL of the provider. For example, a sample URL follows the format below:

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				`https://cluster-xxxxxxxxxxx.pegaservice.net/`
`clientId`	string	☐	`1009518635600839 6159`	The OAuth 2.0 client ID used to authenticate access to Pega Platform. The default client ID is the DX-ComponentDev OAuth 2.0 client registration record. To view or create OAuth 2.0 client registration, log into Dev Studio and go to **Security** > **OAuth 2.0 Client Registration**.
`grantType`	string	☐	`authCode`	The grant type associated with the OAuth 2.0 client ID. Use one of the following values based on the OAuth client registration record: • `authCode` : Authorization code used with Proof Key for Code Exchange (PKCE) • `passwordCreds` : Password credentials • `clientCreds` : Client credentials

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
`clientSecret`	string	☐	`""`	The client secret associated with the OAuth 2.0 client registration referenced by the client ID. Required when the type of client is **Confidential** and should be blank when the type of client is **Public**.
`authService`	string	☐	`pega`	The authentication service alias to use. Only required when the value of `grantType` is `authCode`. Accepted values are: • `pega`: Pega "Platform Authentication" service • Any other enabled authentication service alias, such as, `Okta` or `SSO`.
`redirectUri`	string	☐	`http://localhost:4010/`	The location (application route) to send the user to after successful authentication. By default, the SDK redirects the user to the same application route where the authentication is requested. Use this to redirect the user to a different location

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				after authentication. You must configure the `cert` and `keys` attributes if using `https`.
`cert`	string	☐	`""`	The fully-qualified filesystem path to the SSL certificate file to use when starting local HTTPS server. Needed only when `grantType` is `authCode`. **NOTE:** DXCB ships with self-signed credentials out-of-the-box to deliver an immediate developer Pega Platform Authentication experience.
`key`	string	☐	`""`	The fully-qualified filesystem path to the SSL key file to use when starting local HTTPS server. Needed only when `grantType` is `authCode`.
`user`	string	☐	`""`	The Pega Platform user ID associated with your account. Needed only when `grantType`

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				is `passwordCreds` and `authService` is `pega`.
`password`	string	☐	`""`	The password associated with the Pega Platform user ID. Needed only when `grantType` is `passwordCreds` and `authService` is `pega`.
`authorize`	URL	☐	`""`	The full path to the Pega Platform server's OAuth 2.0 token authorization endpoint. For more information, see OAuth Authorization Framework.
`token`	URL	☐	`""`	The full path to the Pega Platform server's OAuth 2.0 token endpoint. For more information, see OAuth Authorization Framework.
`archiveRulesetName`	string	☐	`CustomDXArch`	The Ruleset to which an archive is exported.
`archiveRulesetVersion`	string	☐	`01-01-01`	The ruleset version to which an archive is exported.

`component`

The `component` section of the `tasks.config.json` file contains the attributes that define the default properties of the component. These attributes contain the default

©2025 Pegasystems Inc

values for the questions you encounter when you create, publish, or build a component.

The following table describes the attributes in the `component` section:

Name	Type	Required	Default value	Description
`library`	string	☐	`YourLibrary`	The default component library used to store Constellation DX components. **NOTE:** • Do not use the default value `YourLibrary`. Update it with your library's name when initializing the project or when you create a library using the `createLib` command. • Finalize the name of your library during the initial development of your component. If you change the name of the library after

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				several components have been published, you must update the component key for all the published components within the library. For more information, see Renaming Constellation DX components. • The value provided here forms the central part of the name space of a component. For example, `Org_YourComponentLib_ComponentName` • Starting from DXCB 25.1.10, the length of the component's name is limited to a maximum number of

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				characters. To maximize the number of characters that can be used for the component's name, limit the lengths of the values for `organization` and `library`.
`type`	string	☐	`""`	The default component type.
`version`	numeric	☐	`0.0.1`	The default component version. **NOTE:** • When library mode is enabled, the value entered here is the library version. • You can either use the default version or assign a version when initializing the project. Do not change the

©2025 Pegasystems Inc

Name	Type	Required	Default value	Description
				version in the `tasks.config.json` file manually after initializing a project. • To change the value of the `version` attribute, use the `createLibVersion` command to create another version of the library.
`subtype`	string	☐	`""`	The default component subtype.
`description`	string	☐	`""`	The default component description.
`icon`	string	☐	`""`	The path to the icon file used for the Constellation DX component.

Updating the version of Constellation DX Component Builder

Update to the latest version of Constellation DX Component Builder (DXCB) to benefit from its latest features.

©2025 Pegasystems Inc

1. Create a new folder.
2. Initialize the project.
   For more information, see Initializing a project.
3. Copy the contents of the `src/components` directory in the existing installation to the `src/components` directory of the new project folder.
4. In the `defaultProps` section of the `index.tsx` file, update the value of the `testId` property to a blank string.

> **NOTE:** New Constellation DX components will already contain this change in the generated template.

5. Update existing components by ensuring that all references to the `testId` property check for a string value.

> **NOTE:** Null is no longer a valid value for `testId`. All checks for null values should now check for a blank string.
>
> For example, `testId ? a : b` must be updated to `testId === "" ? a : b`.

6. Copy the values of the following properties from the existing `tasks.config.json` file and paste them in the new `tasks.config.json` file.
   Replace the values of the following properties under `server-config`:
   - `rulesetName`
   - `rulesetVersion`
   - `devBuild`
   - `server`

   Replace the values of the following properties under `component`:
   - `library`

©2025 Pegasystems Inc

- `version`

> **NOTE:**
> - When you copy properties from the existing `tasks.config.json` file and paste them in the new `tasks.config.json` file, ensure that you do not overwrite any new properties included in the latest version. Overwriting new properties will cause errors.
> - If you are publishing to Launchpad, set `libraryModeCL` to `false`.

7. Save your changes to the new `tasks.config.json` file.
8. To validate that your configuration is working, navigate to the new project folder in the command line, type `npm run authenticate` and then log in using your browser.
9. To validate that your components have been copied correctly, navigate to the new project folder in the command line, type `npm run list` and select `local`. The list of copied components is displayed.

> **NOTE:**
> - If you are updating from a version older than DXCB 23.1.11, you might need to update your components due to changes such as React 17, lint updates, Constellation design system updates, and other package updates.
>
> You might encounter the following warning message in a `.js` or `.jsx` file:
>
> ```
> Unused eslint-disable directive (no problems were reported from 'no-undef')
> ```

©2025 Pegasystems Inc

To update the components, remove the `/* eslint-disable no-undef */` line from the corresponding `.js` or `.jsx` file.

- In DXCB 24.2.10, the parameter types for several functions have been changed from `any` to a specific type. If you encounter type errors, change the declaration of the parameter to the new type. You must also declare asynchronous promise responses. For more information about the type errors and how to resolve them, see Troubleshooting Constellation DX Component Builder.

  For example, if the type for the `myProp` parameter has changed from `any` to `string`, the declaration must be changed from `const myProp: any;` to `const myProp: string;`.

Result:

You have now updated to the latest version of DXCB.

©2025 Pegasystems Inc

How do Constellation DX components work

Learn about the structure of Constellation digital experience (DX) components in an application. An organization can have several applications. Each application can use several customized components. For easier organization and maintenance, you can group components in libraries. The component libraries can be created in different rulesets, which in turn can be used for different applications.

Naming strategy for Constellation DX components

To provide namespace separation for Constellation DX components, the naming convention of the component key is `<organization>_<library>_<component>` and is derived when the component is created.

- `organization` - This value is found in the organization property of the `package.json` file in your project's root folder.
- `library` - This value is requested when the component is created and can also be set as default in the `tasks.config.json` file.
- `name` - This value is requested when the component is created.

> **NOTE:** Starting from DX Component Builder (DXCB) 25.1.10, the length of the component's name is limited to a maximum number of characters. To maximize the number of characters that can be used for `name`, limit the lengths of the values for `organization` and `library`. The character limit is because components are now served from Pega Platform™. For more information, see Rule resolution for Constellation DX components.

When you publish a component, you specify the ruleset and version where the component will be stored.

©2025 Pegasystems Inc

> **NOTE:** Because the component key aligns with the organization and library, update the component key whenever the organization and library change.

DXCB supports the following types of Constellation DX components:

Field
: Allows to configure interactive UI elements such as text boxes, buttons, and sliders.

Widget
: Improves usability through built-in logic and functionality. For example, the attachment widget allows users to upload documents.

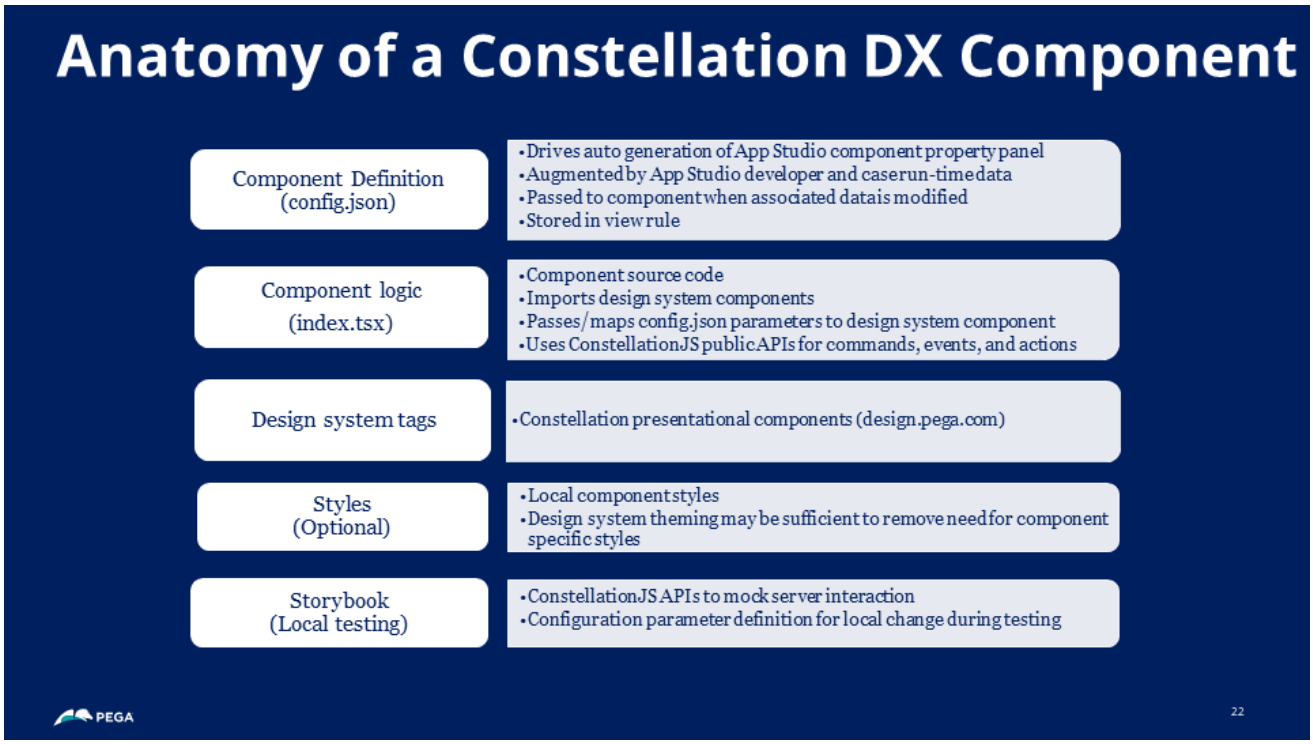
Layout template
: Provides layouts for the fields and widgets on a form.

For more information about the types of Constellation DX components, see Types of Constellation DX components.

For more information about the out-of-the-box components that are available in the Constellation design system, see Constellation UI.

Anatomy of a Constellation DX component

©2025 Pegasystems Inc



Anatomy of a Constellation DX component

Anatomy of a DXCB project

Your project is a fully functioning React development environment that uses Node.js to provide command line driven features. It is configured to use the correct dependencies that ensure your Constellation DX Components will integrate into the version of Pega Platform™ you are using. It is important to know which parts of your project are specific to the version of Pega Platform.

DXCB uses the Constellation Design System and the ConstellationJS library. The Constellation Design System includes the React presentational components that are packaged as multiple modules built on `cosmos-react-core`.The version of `cosmos-react-core` and the other modules in the `cosmos-react-*` family need to be the same version as used by Constellation in Pega Platform. Since this version mapping is done for you in the `package.json` file, it is important that you use the correct version of `cosmos-react-core` for your environment.

©2025 Pegasystems Inc

Semantic Versioning is used to determine the correct version of the `@pega/custom-dx-components` package for the version of Pega Platform. For more information, see Semantic Versioning.

The highest patch version of the `@pega/custom-dx-components` package is matched to the highest patch version of the same major/minor version of Pega Platform.

For example:

- `@pega/custom-dx-components@25.1.x` for any patch version of Pega Platform '25
- `@pega/custom-dx-components@24.2.x` for any patch version of Pega Platform '24.2
- `@pega/custom-dx-components@24.1.12` for any patch version of Pega Platform 24.1
- `@pega/custom-dx-components@23.1.15` for any minor version of Pega Platform '23

The Node Package Manager (npm) also uses Semantic Versioning and the DXCB project is configured to use the latest patch version of DXCB for a specific major/minor version of Pega Platform.

Rule resolution for Constellation DX components

From DXCB 25.1.10 onwards, both standalone and library mode components are picked through Rule resolution.

> **NOTE:** This section is applicable only if you are using DXCB 25.1.10 or later versions.

Standalone components and libraries are now published to ruleset versions. For a component to be available to an application, the ruleset version to which the component is published must be included in the application.

©2025 Pegasystems Inc

Rule resolution for standalone components

Standalone components can be published only to an unlocked Ruleset that is present in the application to which you are being authenticated. Standalone components cannot be published to a Branch Ruleset.

While multiple versions of a standalone component can be published to multiple ruleset versions, the correct version of a standalone component is picked through Rule resolution. Standalone components in your application stack are also Rule resolved.

When you publish a standalone component to Pega Platform™, the following Rules are created:

- *Rule-UI-Component*
- *Rule-File-Text*

Rule-UI-Component

The *Rule-UI-Component* is created for each standalone component published from DXCB. The standalone component is published to the unlocked ruleset version in the application to which you are authenticated. *Rule-UI-Component* provides the configuration information for App Studio.

Rule-File-Text

When a *Rule-UI-Component* is created for a standalone component, a corresponding *Rule-File-Text* is created along with the compiled JavaScript code. The *Rule-File-Text* is Rule resolved at run time and is used by Constellation.

The *Rule-File-Text* is large in size because it contains the component code along with the associated code required through imports. The code in the *Rule-File-Text* is also not shared, so it is replicated in components that have the same utilities and modules.

The *Rule-File-Text* cannot be edited, deleted, saved, branched, checked out, or checked in because it is a Rule managed by the corresponding *Rule-UI-Component*. The *Rule-File-Text* is deleted only when the corresponding *Rule-UI-Component* is deleted through Pega Platform or DXCB. The *Rule-File-Text* is updated only when the corresponding *Rule-UI-Component* is updated through DXCB.

PEGA ©2025 Pegasystems Inc

To update the code, you must update it in DXCB and publish it again. If the ruleset version is locked, you must publish it to a higher unlocked ruleset version.

Rule resolution for library mode components

For library mode components, the entire library is Rule resolved. Through Rule resolution, a library can have multiple versions with different components, but only one version of the library and its components is used at run time. This happens because Rule resolution resolves to the `Org_Lib` library name.

You can create versions of a library that are independent of its ruleset version. However, you can publish only one library version to a ruleset version. When you publish an incremental library version to the same ruleset version, it overwrites the existing library version. The overwritten library versions are archived in the local `store` folder.

For applications that are built on other applications, Rule resolution ensures that components that exist in an application are available to the built-on application without having to include them.

Because the `allowedApplications` attribute in the `config.json` file is no longer used, ensure that the components of the library are added to the application so that they are available at run time. For more information, see Adding library mode Constellation DX components to an application.

When you publish a library to Pega Platform™, the following Rules are created:

- *Rule-UI-ComponentLibrary*
- *Rule-UI-Component*
- *Rule-File-Text*

Rule-UI-ComponentLibrary

The *Rule-UI-ComponentLibrary* contains information about the published library. The corresponding Rule form displays the following information:

PEGA ©2025 Pegasystems Inc

- library name
- library version
- Constellation design system version
- build date
- server information
- components
- JavaScript files, such as, chunks and entry points
- component map
- component library as a zipped folder

The Rule form displays the components that are available to the application. View the components in a library by performing the following steps:

1. In the navigation pane of Dev Studio, click **Records**.
2. In the **Records Explorer**, click **User Interface** > **Component Library**.
3. On the **Instances of Component Library** tab, click the name of your library.

   The **Components** tab displays the list of components in the selected library.

Rule-UI-Component

Rule-UI-Component is created for each component in a library and contains the following information:

- type
- sub type
- `config.json`
- localizations
- library name
- library version

Rule-UI-Component is managed by the corresponding *Rule-UI-ComponentLibrary*.

©2025 Pegasystems Inc

Rule-File-Text

The webpack creates several *Rule-File-Text* Rules to dynamically load code and represent common code and component entry points, and contains the following information:

- library map
- component entry points containing the first JavaScript code to load for a component
- hash named files containing common and dynamically loaded code

The *Rule-File-Text* Rules exist as a group and cannot be separated or edited.

A copy of the *Rule-UI-ComponentLibrary*, *Rule-UI-Component*, and *Rule-File-Text* Rules is stored in a corresponding *Rule-File-Binary* Rule.

PEGA ©2025 Pegasystems Inc

Types of Constellation DX components

You can use the Constellation DX Component Builder (DXCB) to create various types of Constellation DX components. DXCB supports selected App Studio components. Choose the type and sub-type of the component such that it is most similar to the behavior of the desired final component.

DXCB contains a comprehensive set of sample code that can be used to create your own Constellation DX components. There are three types of component samples and each component type has several sub-types.

You can create the following types of Constellation DX components:

- Layout templates
- Widgets
- Fields

NOTE:

- When creating a component, choose the component type and sub-type that is most similar to the behavior of the desired final component. For more information, see Choosing the type of a Constellation DX component.
- DXCB does not support all App Studio components. For more information on the list of supported components, see Component examples provided by Constellation DX Component Builder.

Layout templates

Layout template components can be used in form, details, and page contexts.

©2025 Pegasystems Inc

The following video shows an example of the edit view of a form template and the various form templates available. In the **Workflow** tab of a case type, a step is selected and **Configure view** is clicked. In the **Edit View: Determine Category** window, **Edit** is clicked to view the types of form layout templates available. The **Edit** button is visible only when additional custom form layout templates have been published.

For more information, see Layout templates.

Widgets

Widget components can be used to add widgets to the case utilities panel that provides additional case-related functionality, such as adding stakeholders, followers, or attachments to a case. Page widgets can be added to landing pages where the context is a landing page context and is not specific to a case. An example of a page widget is the ToDo widget.

The following video shows an example of a Stakeholders widget being added to the Utilities section. In this video, the **UX** tab of a case type is open. In the **Full Page View** tab, **+ Add** is clicked within the **Utilities** section. From the list of available widgets, the **Stakeholders** widget is selected and **Add** is clicked. The Preview now refreshes and displays the **Stakeholders** widget added to the **Utilities** section.

For more information, see Widgets.

Fields

Field components can be used to replace out-of-the-box field components with your own implementation.

The following video shows an example of a Terms and Conditions URL field component. In this video, the **Edit view** window of the **Contact Info** form is open. Under the **Fields**

©2025 Pegasystems Inc

section, + Add is clicked. From the list of available fields, the Terms and Conditions URL field is selected and Add is clicked. The Preview now refreshes and displays the Terms and Conditions URL field.

The **URL Component** is an extension DX component created with DX Component Builder. The video proceeds to show how to configure the Terms and Conditions URL field as a URL Component. Under the **Fields** section, the Configure icon for the Terms and Conditions URL field is clicked. In the **Configure field: Terms and Conditions URL** window, under the Display as list, URL Component is selected and Save is clicked. The Terms and Conditions URL field is now configured as a URL Component.

For more information, see Fields.

- **Layout templates**
- **Widgets**
- **Fields**
- **Choosing the type of a Constellation DX component**
- **Component examples provided by Constellation DX Component Builder**

Layout templates

Use a layout template to create layouts that cannot be achieved using the out-of-the-box layout templates provided by App Studio.

Layout template components provide layouts for Constellation views.

In standard web development, layouts are created using CSS primitives, such as, CSS Grid and CSS Flexbox. In the Constellation Design System these CSS primitives have a

©2025 Pegasystems Inc

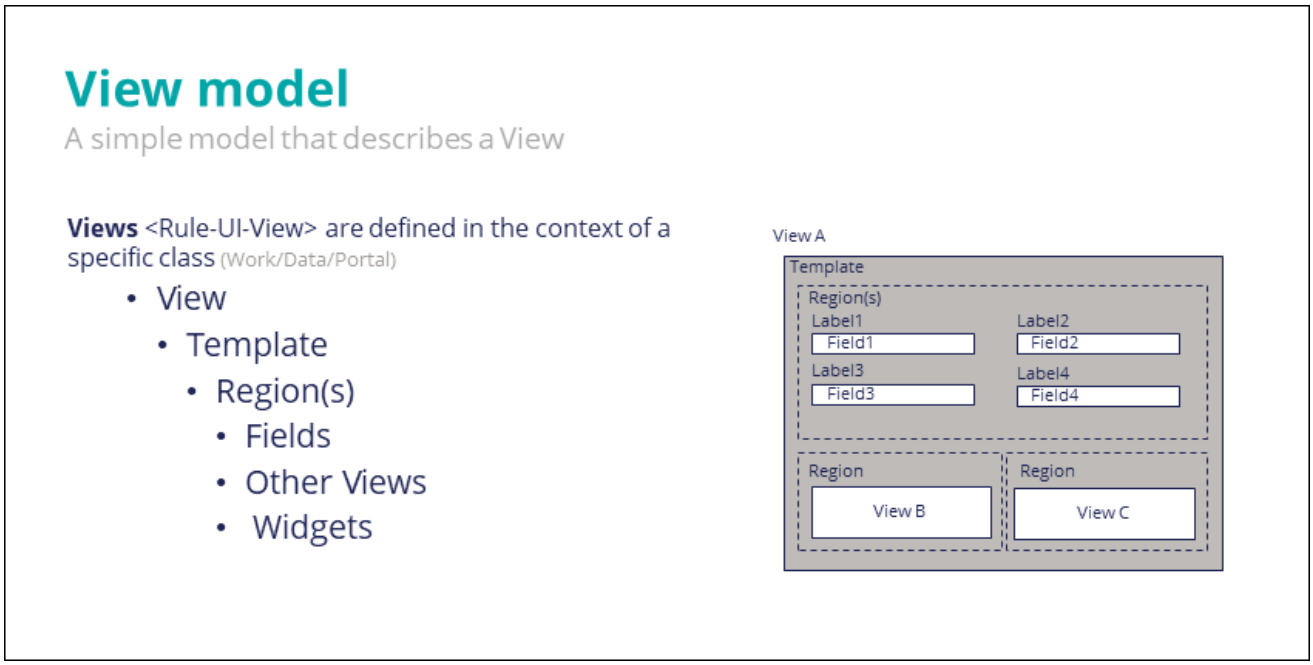
React wrapper applied and the components are called Grid and Flex respectively. These Constellation Design System primitives are the basis of all layout templates.

The Grid organizes the layout of an application in a grid format containing columns and rows. The intersection between a column and row creates a container and each container includes an item. The relationship between the containers and items defines the structure of the layout.

The Flex component organizes the layout of an application in a uni-directional format of either columns or rows. A container will contain either a row or a column of items. A container can wrap items, and items can also be containers. The relationship between the containers and items builds the structure of the layout.

View model

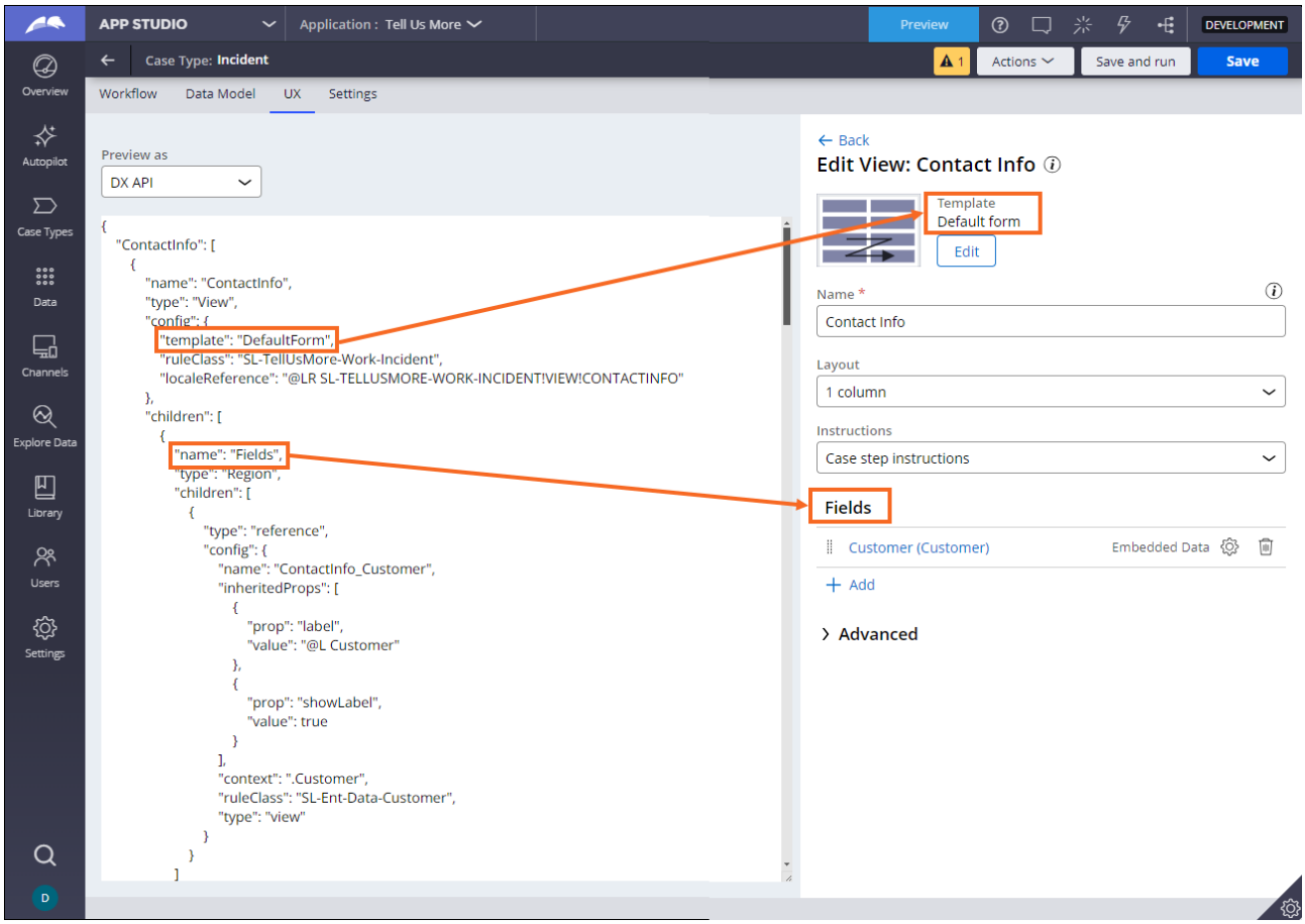
Using App Studio, views are configured with layout templates that have been either predefined or selected by the author. The view model is a JSON data structure that describes the view metadata and is organized as shown in the following image.



View model

©2025 Pegasystems Inc

All Constellation DX component layout templates provide slots called Regions. When configuring views at design time, App Studio authors add components, such as, fields, field groups, views, insights, and widgets into Regions. The layout template code determines how the components slotted into Regions are displayed at run time. Most layouts have been designed and are available through the Constellation design system, such as, the Case Page layout, which cannot be extended. Some of the Constellation design system templates are available for use in your Constellation DX component layout templates.



DX API preview

When presentational templates are used in your Constellation DX component templates, the layout and responsiveness of a View will be optimized to provide a

©2025 Pegasystems Inc

seamless user experience. Depending on the type and location of your View, the available Constellation DX component templates and their behavior might vary.

The behavior of the templates, including breakpoints, white space management, and wrapping, is designed to facilitate compliance with the latest accessibility standards and requirements.

You can create three types of Constellation DX component layout templates using the Constellation DX Component Builder:

- Details
- Form
- Page

Note: The three types of Constellation DX component layout templates use the Constellation Design System layout resources, either directly or indirectly.

Details layout template

The Details layout template controls the layout of the Details tab, which is available in both in the Case View and the Data type. Applying the Details template renders the child components referenced in the Regions as read-only.

The following video shows an example of a Details layout template in the Case View. In this video, the **Edit view** window of a Details template is opened. A customized Details layout template called Extension Details Layout Template has been created. To change the type of the Details template to the customized template, Edit is clicked, Extension Details Layout Template is selected, and Submit is clicked. The Preview now refreshes and displays the new Details layout template. The video now proceeds to show how highlighted fields can be enabled and added to the details layout template. The Enable highlighted fields checkbox is selected and within the **Highlighted fields** section, + Add is clicked. The Commit DateTime field is selected and Add is clicked. The Commit DateTime field is displayed in the **Highlighted fields** section and the preview refreshes

©2025 Pegasystems Inc

to display the **Commit DateTime** field at the top and the remaining fields at their regular positions.

Partial Views

Partial Views use Details templates to cluster UI components into a convenient module that you can reuse in different contexts throughout your application. For more information on partial Views, see Configuring partial Views.

You can use partial Views in read-only scenarios, for example, as tabs on the full View page. Partial Views have prescriptive layouts to improve accessibility and readability when Views using the template are displayed to the user. For more information on the prescriptive layouts applied to partial Views, see Details template.

You can create your own Details layout templates using the following template samples:

- **Details**: provides highlighted data and a single region example
- **Details Region**: provides highlighted data with two regions example

These templates can be used as the basis for your own templates that can contain more than two regions, or remove the ability to set highlighted data.

Details templates are added to partial Views. This will render the content read-only by default. Partial Views can be included in Form Views, which enables creating read-only areas on a form, where the read-only data is not submitted when the form is submitted.

Form layout template

Forms are designed to efficiently collect data from the users of your application. All forms can be edited, and the default form template is designed to ensure optimal responsiveness, consistency, and accessibility.

For more information on form layout templates, see Default form template.

PEGA ©2025 Pegasystems Inc

The following video shows an example of a form layout template in the Case View. In this video, the **Edit view** window of a form template is opened. A customized form layout template called Extension Form Layout Template has been created. To change the type of the form template to the customized template, **Edit** is clicked, **Extension Form Layout Template** is selected, and **Submit** is clicked. The Preview now refreshes and displays the new form layout template.

You can create your own form templates using the following template samples:

- **Form** – single column form layout
- **Form Region** – two column form layout

Similar to the Details template, you can change form templates to have more columns or a completely different layout. By default, the components rendered in form templates are not read-only (apart from embedded partial Views) and any fields including those in embedded form views will be submitted when the form is submitted. Also, by default, form templates cannot contain widgets.

Page layout template

Page layout templates control the layout of landing pages. For more information on landing pages, see Landing pages.

The following video shows an example of a page layout template in the Portal View. In this video, the **Edit landing page** window of a page template is opened. A customized page layout template called Extension Page Layout Template has been created. To change the type of the page template to the customized template, **Edit** is clicked, **Extension Page Layout Template** is selected, and **Submit** is clicked. The Preview now refreshes and displays the new page layout template.

©2025 Pegasystems Inc

You can build your own page layout templates using the **Page** template sample. This template lets you provide App Studio authors with a configurable page (portal) layout template. This template also demonstrates how to enable the Get Next Work functionality and the ability to use an inline layout for insights.

Widgets

Use a widget to display interactive self contained information.

Widget components improve usability through built-in logic and functionality. They are typically a collection of multiple out-of-the-box Constellation UI components combined in a unique way.

FileInput is an example of a Widget component. The *FileInput* component lets a user upload files or attachments using drag-and-drop or by opening the user's file explorer when clicked.

Constellation provides a wide range of widgets that you can add to your application to improve usability by gathering helpful functions in one place. Widgets are supported by Portal landing pages and full object Views. In the full view scenario, widgets populate the expandable **Utilities** panel. Some widgets are customizable and require additional configuration to fully work, while others are ready to use by default. For example, you can help users maintain clarity in attachments by defining custom categories for the attachments that they upload to cases.

You can use the DX Component Builder to create three types of widgets:

- **PAGE** (Portal widgets) - This widget can be used only in landing pages.
- **CASE** (Utilities pane widgets) - This widget can be used only in case views.
- **PAGE & CASE** (Both) - This widget can be used in portal landing pages, and the Utilities pane for both case and data views.

PEGA ©2025 Pegasystems Inc

PAGE widgets

Page widgets can be added to Portal landing pages and can be used for any purpose. They should operate in the Portal context, where they should be operating on cases and data in bulk as opposed to at the instance level.

Examples of out-of-the-box page widgets include the ToDo list and App Announcements.

The following video shows an example of a page widget being added to a landing page in the Portal View. In this video, the **Edit landing page** window of a page template is opened. Within the **Region A** section, + Add is clicked and under **Widgets**, the Quick create checkbox is selected and Add is clicked. The **Quick create** widget is displayed in the **Region A** section and the preview refreshes to display the widget.

CASE widgets

Case widgets are added to the Utilities pane in App Studio. Case widgets interact with a case instance, where they provide data in the instance context related to a specific case.

Examples of out-of-the-box Utilities pane widgets include the Stakeholders widget, the Followers widget, and the Related cases widget.

Use the example widget provided in DX Component Builder as the starting point for your own widget. DX component builder examples are relatively basic for this type of widget and do not provide starting points for common Utilities widgets. The Widget challenge in this mission provides a comprehensive example of how to build everything mentioned above.

The following video shows an example of a case widget being added to a case. In this video, the UX tab of a case type is opened. In the Full Page View tab, + Add is clicked within the **Utilities** section. From the list of available widgets, the **Stakeholders** widget

©2025 Pegasystems Inc

is selected and Add is clicked. The Preview now refreshes and displays the **Stakeholders** widget added to the **Utilities** section.

PAGE & CASE widgets

Page & Case widgets can be used in the Case, Data, and Portal contexts. The only out-of-the-box example of Page & Case widgets is the Pulse (Feed) widget.

Use the example widget provided in DX Component Builder as the starting point for your own widget.

The following video shows an example of a page and case widget being added to a data object. In this video, the UX tab of a case type is opened. In the Full Page View tab, + Add is clicked within the **Utilities** section. The **Pulse** widget is selected and Add is clicked. The **Pulse** widget is displayed in the **Utilities** section.

The following video shows an example of a page and case widget being added to a landing page. In this video, the **Edit landing page** window of a page template is opened. Within the **Region A** section, + Add is clicked and under **Widgets**, the Pulse checkbox is selected and Add is clicked. The **Pulse** widget is displayed in the **Region A** section and the preview refreshes to display the **Pulse** widget.

The following video shows an example of a page and case widget being added to a case. In this video, the UX tab of a case type is opened. In the Full Page View tab, + Add is clicked within the **Utilities** section. From the list of available widgets, the **Pulse** widget is selected and Add is clicked. The **Pulse** widget is displayed in the **Utilities** section.

PEGA ©2025 Pegasystems Inc

To provide a similar user experience to the Pega provided widgets, all Utilities pane widgets in the case and data contexts must adhere to the following design system guidelines:

- Use the **SummaryList** design system component for displaying lists of data with an item count.
  - Use the **View all** functionality for lists that contain more than three items.
  - Use the **Actions** functionality for actions to be performed at the instance level as well as for creating new items.
- Provide the **Icon** design system component in the upper-left corner of the widget. For more information on the **Icon** design system component, see Icon.
- Use Popovers or Modal dialogs to add, edit, or delete entries invoked by actions performed at the instance level. For more information, see Popover and Modal dialog.
- The following guidelines are implemented in most out-of-the-box Utilities pane widgets:
  - If the Utilities pane is in collapsed mode, provide a widget icon and current item count.

    > **NOTE:** To display the widget icon when the Utilities pane is in collapsed mode, you must add the `iconName` property to your `config.json` file, and then import and register the icon. For more information, see Import and register Pega icons.

  - If the item count must reflect the current number of items in the SummaryList for the current browser session, use the ConstellationJS publish/subscribe (PubSub) methods.
  - If the item count must reflect the current number of items in the SummaryList for all browser sessions across all users viewing this case or data instance, use the ConstellationJS notifications manager (Websockets API).

©2025 Pegasystems Inc

Fields

Use a field component to display a field in a manner that cannot be achieved with any other out-of-the-box component.

Field components such as the TextArea, Button, and Slider allow developers to configure interactive UI elements such as text boxes, buttons, and sliders.

Use the example code provided by DX Component Builder as a starting point for building your own field components. The example code is not intended to replace Pega's out-of-the-box components. Ensure that you do not create customized field components for minor formatting changes, since the components you create need to be maintained, tested, and possibly updated whenever you update to a new version of Pega Platform™.

The following video shows an example of a Terms and Conditions URL field component. In this video, the **Configure field: Terms and Conditions URL** window is opened. The **Extension URL Component** is an extension DX component created with DX Component Builder. Under the **Display as** list, **Extension URL Component** is selected. The **Terms and Conditions URL** field is now configured as an Extension URL Component.

Choosing the type of a Constellation DX component

Learn how to choose the type and sub-type of a Constellation DX component.

When creating a Constellation DX component, choose the component type and sub-type that is most similar to the behavior of the desired final component. By doing so, the Constellation DX Component Builder will generate a sample based on the chosen type and sub-type.

PEGA ©2025 Pegasystems Inc

> **NOTE:** The sample code serves as a foundation to expedite the development of your component and does not provide the complete functionality of the out-of-the-box Constellation DX components. The Constellation DX Component Builder contains templated examples for each supported sub-type of Constellation DX components. For the list of supported sub-types of Constellation DX components, see Component examples provided by Constellation DX Component Builder.

When creating your component, choosing the correct sub-type for your component will help App Studio users in locating your component during application authoring. If you choose the incorrect type and sub-type, your component will not be available for selection in App Studio authoring. The component definition or the `config.json` file defines the type and sub-type that will be used to display the component to the App Studio author in the correct context. After the component has been created, you can change the type and sub-type by editing the associated `config.json` file in the component source directory, as long as the type and sub-type match the component definition schema.

For example, if you want to create a widget that will be used in the Utilities pane of a case type then the type would be Widget and sub-type would be CASE.

> **NOTE:** When you change the type and sub-type of a component by editing the associated `config.json` file in the component source directory, the sample code will not be regenerated.

Component examples provided by Constellation DX Component Builder

Learn about the various App Studio components supported by Constellation DX component builder.

©2025 Pegasystems Inc

The following table lists the components that can be configured in App Studio along with the override support and examples provided by the Constellation DX Component Builder. For more information about the component definition, see Component definition.

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
Address	Field	Address	No	No
AppAnnouncement	Widget	PAGE	Yes	Yes
Attachment	Field	DATA_CAPTURE	No	No
Search box (AutoComplete)	Field	Picklist	Yes	Yes
BannerPage	Template	PAGE	No	No
Boolean	Field	Boolean	Yes	Yes
CaseHistory	Widget	CASE	No	No
CaseOperator	Widget	CASE	No	No
CasePreview	Template	CASEPREVIEW	No	No
CaseSummary	Template	SUMMARY	No	No
CaseView	Template	CASEVIEW	No	No
Checkbox	Field	Boolean	Yes	Yes
Confirmation	Template	CASE	No	No

PEGA ©2025 Pegasystems Inc

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
Currency	Field	Decimal-Currency	Yes	Yes
DashboardFilter	Field		No	No
DashboardPage	Template	PAGE	No	No
DashboardTab	Template	TAB	No	No
DataReference	Template	DATAVIEW	No	No
Date	Field	Date	Yes	Yes
DateTime	Field	DateTime	Yes	Yes
Decimal	Field	Decimal	Yes	Yes
DefaultForm	Template	FORM	No	No
Details	Template	DETAILS	Yes	Yes
DetailsSubTabs	Template	DETAILS	No	No
DetailsThreeColumn	Template	DETAILS	No	Yes
DetailsTwoColumn	Template	DETAILS	Yes	Yes
Dropdown	Field	Picklist	Yes	Yes
DuplicateCases	Template	FORM	No	No
DynamicTabs	Template	DETAILS	No	No

©2025 Pegasystems Inc

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
Email	Field	Text-Email	Yes	Yes
EmbeddedData	Field	DATA_CAPTURE	No	No
ExploreDataPage	Template	PAGE	No	No
FileUtility	Widget	CASE	No	No
Followers	Widget	CASE	No	No
InlineDashboardPage	Template	PAGE	No	No
InlineDashboardTab	Template	TAB	No	No
Integer	Field	Integer	Yes	Yes
ListField	Field	DATA_CAPTURE	No	No
ListPage	Template	PAGE	No	No
ListView	Template	LIST	No	No
Location	Field	DATA_CAPTURE	No	No
NarrowWideDashboard	Template	DASHBOARD	No	No
NarrowWideDetails	Template	DETAILS	No	No

©2025 Pegasystems Inc

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
NarrowWideForm	Template	FORM	No	No
NarrowWidePage	Template	PAGE	No	Yes
NarrowWideTab	Template	TAB	No	No
ObjectPage	Template	OBJECTVIEW	No	No
OneColumn	Template	FORM	Yes	Yes
OneColumnPage	Template	PAGE	Yes	Yes
OneColumnTab	Template	TAB	No	No
Percentage	Field	Decimal-Percentage	Yes	Yes
Phone	Field	Text-Phone	Yes	Yes
Picklist	Field	Picklist	Yes	Yes
Predictions	Widget	CASE	No	No
PromotedFilter	Field		No	No
Pulse	Widget	PAGE & CASE	No	No
Query	Field	Query	No	No
QuickCreate	Widget	PAGE	No	No
RadioButtons	Field	Picklist	Yes	Yes

©2025 Pegasystems Inc

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
RelatedCases	Widget	CASE	No	No
RichText	Field	Text-Paragraph	No	Yes
ScalarList	Field		No	No
SemanticLink	Field	ACTION	No	No
SimpleTable	Template	DATAVIEW	No	No
Stakeholders	Widget	CASE	No	No
SubTabs	Template	TAB	No	No
TabbedPage	Template	PAGE	No	No
Tags	Widget	CASE	No	No
TextArea	Field	Text-Paragraph	Yes	Yes
TextContent	Field	DATA_CAPTURE	No	No
TextInput	Field	Text	Yes **NOTE:** There are two examples: `Tex`	Yes

©2025 Pegasystems Inc

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
			`t` and `TextInput`.	
ThreeColumnDashboard	Template	DASHBOARD	No	No
ThreeColumnPage	Template	PAGE	Yes	Yes
ThreeColumnTab	Template	TAB	No	No
ThreeUpOneDownDashboard	Template	DASHBOARD	No	No
Time	Field	TimeOfDay	Yes	Yes
Todo	Widget	PAGE	No	No
TwoColumn	Template	FORM	Yes	Yes
TwoColumnDashboard	Template	DASHBOARD	No	No
TwoColumnPage	Template	PAGE	Yes	Yes

©2025 Pegasystems Inc

App Studio component	Component definition type	Component definition sub type	Example provided by Constellation DX Component Builder	Supported by Constellation DX Component Builder
TwoColumnTab	Template	TAB	No	No
TwoUpOneDownDashboard	Template	DASHBOARD	No	No
URL	Field	Text-URL	Yes	Yes
UserDataReference	Field	DATA_CAPTURE	No	No
UserReference	Field	DATA_CAPTURE	No	No
WideNarrowDashboard	Template	DASHBOARD	No	No
WideNarrowDetails	Template	DETAILS	No	Yes
WideNarrowForm	Template	FORM	No	No
WideNarrowPage	Template	PAGE	No	Yes
WideNarrowTab	Template	TAB	No	No

©2025 Pegasystems Inc

Library mode

Library mode enables you to publish Constellation DX components to a library and maintain multiple libraries, giving you more flexibility when working with components.

Library mode publishes Constellation DX components to a single versioned library. Components created in library mode override standalone components with the same name.

Introduced in DXCB 25.1.10, library mode is the default mode of working with Constellation DX components. This is indicated by the value of the `libraryModeCL` attribute set to `true` in the `tasks.config.json` file.

NOTE:

- To work only with standalone components, disable library mode by opening the `tasks.config.json` file and setting the `libraryModeCL` attribute to `false`. For more information, see Disabling library mode in DXCB 25.1.10.
- To convert your existing standalone components and work with them in library mode, see Importing and publishing standalone components.
- Library mode is not supported in Launchpad®. If you are working in Launchpad, set `libraryModeCL` to `false`.

With library mode enabled, DXCB can create and locally maintain multiple libraries and library versions, with their configuration options. With DXCB, you can import and export library versions for code sharing and development purposes. You can also import existing components into a library. Code development, validation, and publishing of components is currently limited to a single library or library version at a time.

©2025 Pegasystems Inc

- Library mode is only compatible with Pega Platform™ '25.
- The components in the library are smaller in size than standard Constellation DX components because components in the library share common code.
- You can publish several libraries and library versions. Only one library version is available for a ruleset version. An application can only access one library version at a time.
- After selecting the library through the application record, App Studio automatically locates components in the library that have the same naming convention (`Org_Lib_ComponentName`) as previous single components.
- Published libraries can be deleted.
- A dynamic system setting in Pega Platform enables the service to provide component-specific libraries that are published to that Pega Platform Server.
- Libraries are Rule resolved based on the application and application stack.

Advantages of library mode

Publishing components in a library provides the following advantages:

- Multiple versions of a library can be created. Applications can select a specific version of a library that is available for a ruleset version.
- Multiple libraries can exist through Rule resolution.
- Multiple libraries and multiple versions of a library can be managed locally.
- Libraries can be imported and exported locally.
- Libraries can only be published to an unlocked ruleset version.
- Libraries can be archived to Pega Platform.
- Components in a library share common code through a shared directory.
- Components can be imported to a library.

Limitations of library mode

Publishing components in a library has the following limitations:

- Only one library version is available for a ruleset version.

©2025 Pegasystems Inc

- An application can use multiple libraries, but only one version of each library. With multiple versions of the same library in multiple ruleset versions, the application will be Rule resolved to one library version.
- Multiple libraries and multiple versions of a library can only be managed locally. Since the published library data cannot be retrieved from services, you must maintain code externally from Pega Platform.
- Any actions performed on a specific library or version of a library must be performed in the `src/components` directory. If you switch between libraries or versions of a library, the code of the existing library is replaced by the code of the selected library in the `src/components` directory. The code of the existing library is moved to the `store` directory as an archive that cannot be edited from this directory.

> **NOTE:** Ensure that you regularly back up the code stored in your local directory.

- To view the archived code, switch between libraries or versions of a library using the `switchLib` or `switchLibVersion` commands.
- Multiple libraries and multiple versions of a library cannot be edited simultaneously.
- When you switch between libraries or versions of a library, unsaved changes are not archived. Ensure that you save your changes before switching between libraries or library versions.
- Updating the organization in the `package.json` file causes issues. If you want to change the organization of a component, create a new project with the organization that you want to use, export the libraries from your previous project, and import them into your new project. Your components will be updated to reflect the updated organization.

©2025 Pegasystems Inc

> **NOTE:** Use the `exportLibVersion` command to export the libraries and the `importLibVersion` command to import the libraries.

- A library can only be published to an open ruleset version.
- A library cannot be published to a branch Ruleset.
- The *Rule-UI-ComponentLibrary* and the components that it contains are Rule resolved together. Rule resolution cannot happen individually.
- Components that are generated by publishing through the *Rule-UI-ComponentLibrary*, such as, *Rule-UI-Components*, *Rule-File-Text*, and *Rule-File-Binary*, are completely managed by the *Rule-UI-ComponentLibrary* and cannot be modified in Pega Platform. The *Rule-UI-ComponentLibrary* and its components cannot be saved, saved as, deleted, branched, checked in, or checked out. To modify these components, publish the updated components to the *Rule-UI-ComponentLibrary* through the Constellation DX Component Builder (DXCB).
- Deleting the *Rule-UI-ComponentLibrary* through DXCB or Pega Platform will delete all managed rules. A deleted *Rule-UI-ComponentLibrary* cannot be restored in Pega Platform, but it can be restored from DXCB by republishing changed components to the *Rule-UI-ComponentLibrary*.
- Non-library components with the same `Org_Lib` prefix cannot be published if a *Rule-UI-ComponentLibrary* with the same name is in the same Ruleset.
- After you initialize a project, update the `component` object in the `tasks.config.json` file, and create a library. After the library is created, do not manually update the `component` object in the `tasks.config.json` file. This is because the `component` object is updated when you create libraries and switch between them. To manually update the `component` object, use the `createLib` and `switchLib` commands.
- The `promoteWebPack`, `publishAll`, `revertWebPack`, `buildComponent`, and `buildAllComponents` commands are not supported.

©2025 Pegasystems Inc

Best practices

- Create a new library version whenever you publish a library to a new release of Pega Platform. Because the versions of the server and Constellation design system are different, separate versions of the `package.json` and `tasks.config.json` files are created. This makes it easier to go back to an older version.
- Publish your library to a separate Ruleset included in your application stack instead of your application's default Ruleset. This lets you keep track of component code and keeps it separate from your application's Ruleset. Because a large number of Rules can be created with library mode, having a separate Ruleset allows better maintenance and sharing of components.
- Publish archives to a dedicated Ruleset that is different from your component Ruleset. This ensures your archives are retained if you delete your component Ruleset.
- After the application development process is completed, lock your Ruleset. This prevents further changes to your application.
- Depending on your use case, a project can contain multiple library versions or each project can contain a library. You can also have multiple projects for a release.

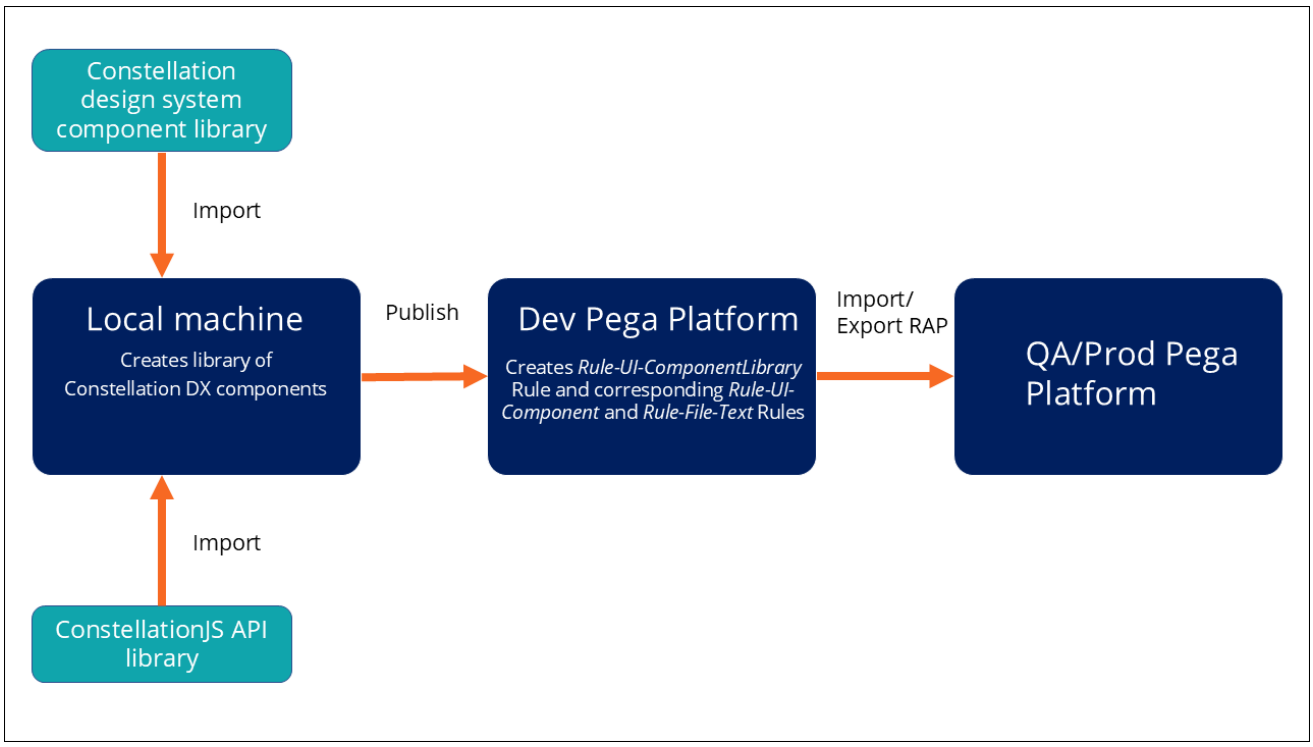
PEGA ©2025 Pegasystems Inc

Developing a library mode Constellation DX component

Learn to create and use a library mode Constellation DX component in your application. Understanding all steps of the process helps you successfully build and deploy library mode Constellation DX components to adapt and extend the UI of your application.

The three stages in developing a library mode Constellation DX component are:

- Create the component library.
- Publish the component library.
- Deploy the library mode components.



Developing a library mode Constellation DX component

©2025 Pegasystems Inc

Create the component library

Build a component library locally by using web development tools, such as npm, storybook, and writing JavaScript inside a code editor. Create the code of the component using the Constellation design system component library and *ConstellationJS* API library.

Import these libraries into your application to use the styling and theming of the Constellation design system component library and the data from the *ConstellationJS* API library for creating your component library.

After you test the library mode component locally, you can publish the library to the Pega developer platform.

Publish the component library

Publish the component library to a ruleset version in the Pega developer platform.

The *Rule-UI-ComponentLibrary* Rule is created containing information about the published library. The *Rule-UI-Component* Rule is created for each library mode component. Several *Rule-File-Text* Rules are created to dynamically load code and represent common code and component entry points.

For more information about these Rules, see Rule resolution for library mode components.

Deploy the library mode component

Use the published library mode component in your application by using App Studio.

Import or export the library mode component as a RAP file by using the Deployment Manager to a Pega QA or production platform.

For detailed information on each stage of developing a library mode Constellation DX component, see the following topics:

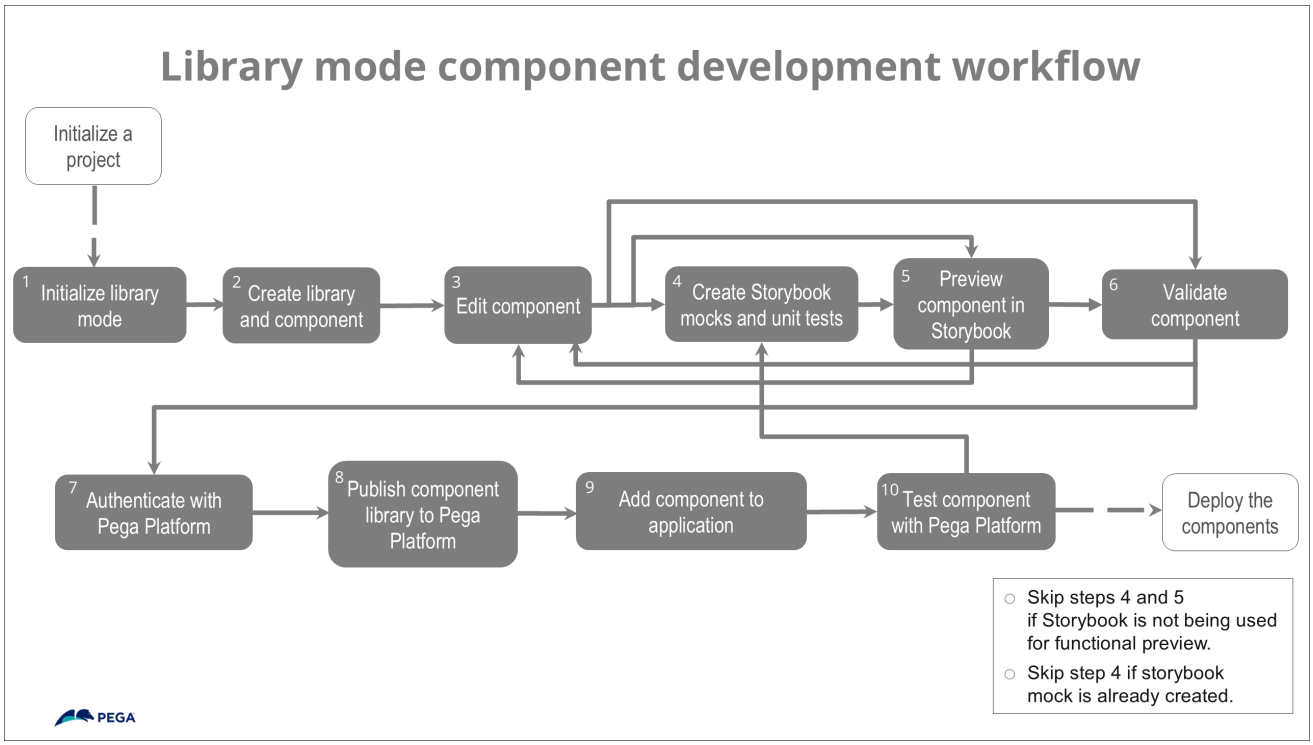
PEGA ©2025 Pegasystems Inc

- Workflow of a library mode Constellation DX component
- Initializing library mode
- Creating a component library
- Editing a library mode Constellation DX component
- Creating Storybook mocks and unit tests for library mode Constellation DX components
- Previewing library mode Constellation DX components in Storybook
- Validating library mode Constellation DX components
- Authenticating Pega Platform Server user access for library mode Constellation DX components
- Publishing a component library
- Adding library mode Constellation DX components to an application
- Debugging the component library
- Localizing a library mode Constellation DX Component
- Commands used in library mode
- Working with components in library mode

Workflow of a library mode Constellation DX component

The developer workflow illustrates the steps involved in creating, publishing, and deploying a library mode Constellation DX component.

©2025 Pegasystems Inc



Developer workflow of a library mode Constellation *DX component*

Initializing library mode

Configure the `tasks.config.json` file to initialize library mode.

> **Before you begin:**
>
> Ensure that you have performed the following tasks:
>
> - Install Pega Platform™ '25.
> - Initialize a project to work with Constellation DX components. For more information, see Initializing a project.

1. Open the `tasks.config.json` file and ensure that `libraryModeCL` is set to `true`.
   This setting enables you to use the library mode commands.

©2025 Pegasystems Inc

2. The values for `library` and `version` are updated when you create a library. Do not change these values after creating the library.
3. Ensure that the default value of `components-directory-path` is `src/components`.
4. In `import-relative-path` and `export-relative-path`, enter the paths to which the libraries are imported and exported.

   The path is relative to the `User/<username>` folder. For example, to update the path to point to the `User/JohnDoe/New Downloads` folder, enter `New Downloads` as the value for `import-relative-path` or `export-relative-path`.

   For more information about the `import-relative-path` and `export-relative-path` attributes, see Top level attributes.

5. Enter the `rulesetName` and `rulesetVersion` to which the component libraries must be published.

   > **NOTE:** For better maintenance and sharing of components, enter a separate Ruleset for each library.

> **Result:**
>
> Library mode is initialized.

Creating a component library

To create a library mode component, you must first create a library.

©2025 Pegasystems Inc

Before you begin:

Initialize library mode. For more information, see Initializing library mode.

NOTE:

- For information about the types of Constellation DX components that can be created by the Constellation DX Component Builder (DXCB), see Types of Constellation DX components.
- DXCB does not support all App Studio components. For more information on the list of supported components, see Component examples provided by Constellation DX Component Builder.

1. Open your preferred terminal tool.
2. Browse to your Constellation DX Component Builder project folder.
3. Enter `npm run createLib`.
   For more information, see createLib.
4. Enter the details of the library:
   a. Enter the name of the library.
   b. Enter the version of the library.
   c. If asked, choose if you want to retain the current changes to your config files or go back to the original config files.

   Result:

   Your library has been created.

5. Enter `npm run create`.

©2025 Pegasystems Inc

When creating a component, choose the component type and sub-type that is most similar to the behavior of the desired final component. For more information, see Choosing the type of a Constellation DX component.

Locate each DX component type in App Studio as follows:

- To view the Details template for a Case Type, click the **Case types** pane, select an existing Case Type, click the UX tab, click the Other Views tab, navigate to the **Partial** section, and then click Details. In the **Edit View: Details** pane, click Add to add new Page widgets and click Edit to change the Details template.
- To add CASE or PAGE & CASE widgets, click the **Case types** pane, select an existing Case Type, click the UX tab, click the Full case page tab, navigate to the **Utilities** section, and click Add.
- To view fields and the FORM template, click the **Case types** pane, select an existing Case Type, select a Step, and click Configure view. In the **Edit view: <view name>** pane, navigate to the **Fields** section, click Add to add new fields, and then click Edit to change the FORM template.
- To view Page widgets and the PAGE layout template, click the **Channels** pane, select an existing channel interface, click Landing pages, and then click Home. In the **Edit landing page: Home** pane, click Add to add new Page widgets and click Edit to change the PAGE layout template.

For information on each type of Constellation DX component, see Types of Constellation DX components.

6. Enter the details of the component:
   a. Enter the type of the component.
      For example: `Field`.
   b. Enter the sub-type of the component, for example `Text`.

©2025 Pegasystems Inc

> **NOTE:** Component authoring teams can use sub-type metadata. The template code does not use the metadata. You can change it at any time in the component definition file.

c. Enter the name of the component.

   For example: `customcomp0`.

d. Enter the component label.

   For example: `custom component`.

e. Describe the component.

   For example: `This is a customized searchbox that filters data.`

Result:

The library mode component is created in the following path:

```
..../<project name>/src/components/
<Org_Lib_ComponentName>
```

> **NOTE:**
> - The name of the component now uses the `<Org_Lib_ComponentName>` format.
> - The length of the component's name is limited to a maximum number of characters. The maximum number of characters varies depending on the string lengths of the organization, library, version, and other constants, and is different for standalone and library mode components.
> - Do not manually change the name of the `<Org_Lib_ComponentName>` folder.
> - Do not manually change the names of the classes or artifacts in the generated templated code in the `<Org_Lib_ComponentName>` folder.

©2025 Pegasystems Inc

- To change the name of the component, use the `rename` command. For more information, see Renaming Constellation DX components.

A `src/components` folder is created along with the files required to build the component. Apart from the core files shown in the code block below, which are initialized for every component, some components also generate supporting files that contain utility libraries.

```
src
  components
    <Org_Lib_ComponentName>
      index.tsx
      demo.stories.tsx
      config.json
      styles.ts
    shared
      create-nonce.ts
      PConnProps.d.ts
```

The `index.tsx` file contains the actual component rendering logic.

The `demo.stories.tsx` file contains the demo that can be viewed in the storybook.

The `config.json` component definition file contains the mapping of properties for configuring the auto-generated property pane. The property pane is displayed in App Studio when you author an application to use the Constellation DX component.

The `create-nonce.ts` file contains the webpack nonce for security. For more information, see https://webpack.js.org/guides/csp/.

©2025 Pegasystems Inc

> The `PConnProps.d.ts` file contains the TypeScript definitions for PCore and PConnect APIs.
>
> The properties in the context of Constellation DX components are definitions of component parameters that are used during design and at run time. Design settings are constant. Run-time parameters that contain field references related to the Pega data model are resolved from the actual data at run time.

7. Run the following command to start the Storybook to view the demo:
   `npm run startStorybook`
8. Modify the source files to build your own component.
   You create Constellation DX components by using generally available tools. You need advanced knowledge to create reliable Constellation DX components using the Constellation design system. For more information, see the following resources:
   - Configuring Constellation
   - Constellation Design System
   - Using PCore and PConnect Public APIs

   You can also install and import third-party libraries, if required, for your component.

What to do next:

> **NOTE:** You can use the `createAll` command to create all supported Constellation DX components.
>
> The purpose of the `createAll` command is to create working examples of each supported Constellation DX component. After using the `createAll` command, delete the components that you do not need. As a best practice, always use the `create` command to create

©2025 Pegasystems Inc

> the components you need, to minimize the library size and reduce ongoing library maintenance cost.

Related concepts

- Choosing the type of a Constellation DX component

Related reference

- Component examples provided by Constellation DX Component Builder

Editing a library mode Constellation DX component

Learn about the structure of a library mode Constellation DX component, and how to modify it to match your specific business need.

Updating the component definition

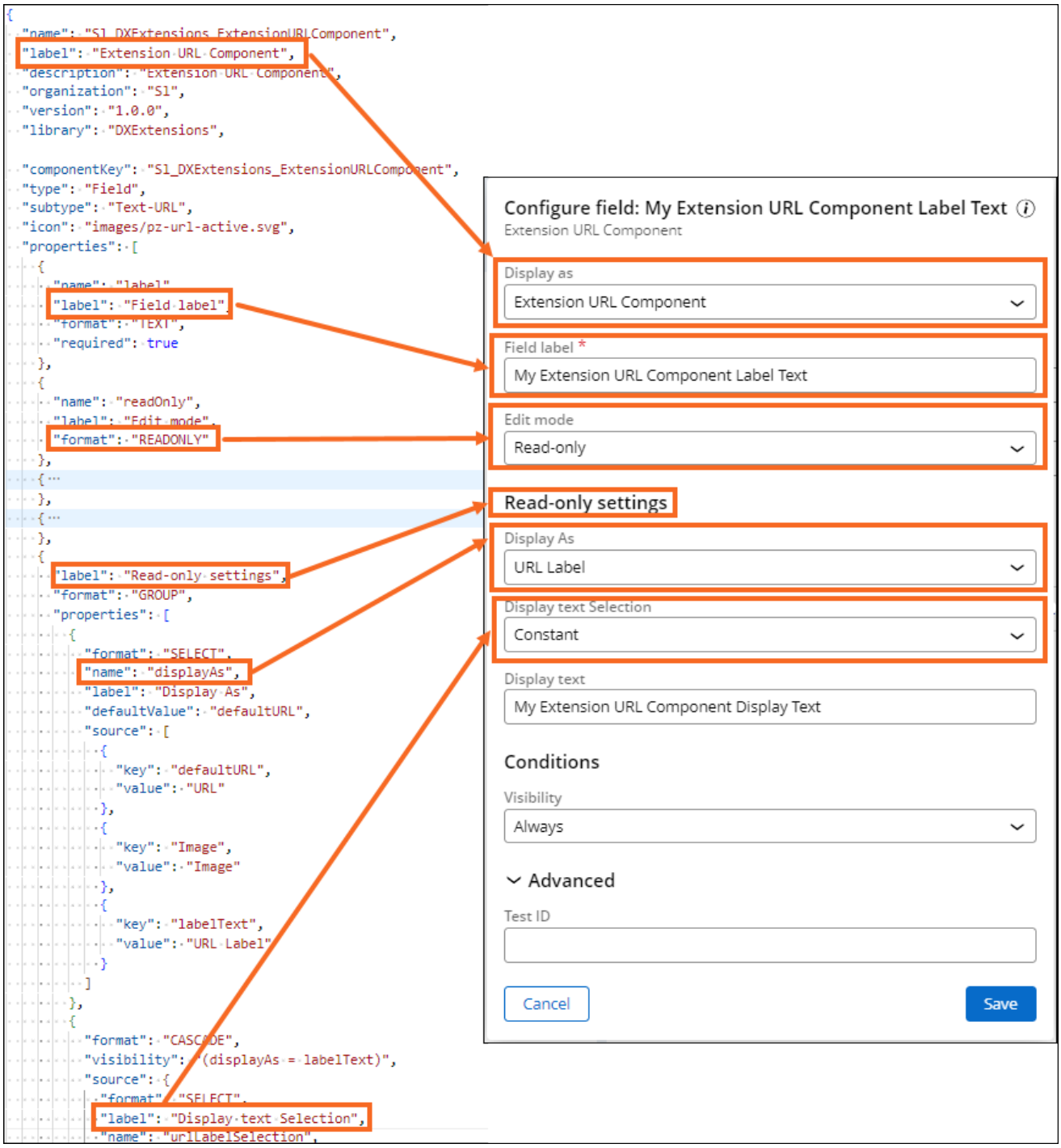
When you create a component, the Constellation DX Component Builder generates a `config.json` file, or component definition file. Every Constellation DX component created with Constellation DX Component Builder requires a component definition file to publish the component to Pega Platform™.

The component definition drives the UI authoring experience when you configure the component for use in Pega Platform, meaning that the property pane of the component in Pega Platform is defined by this file.

The UI view authoring relies on the information in the component definition to provide a streamlined experience when you edit the view metadata that is consumed at run time.

©2025 Pegasystems Inc

The component definition consists of header attributes and an array of property definitions. You can modify these attributes and add other attributes when you design your component.



©2025 Pegasystems Inc

Component definition mapping to property panel

This is an evolving schema that is expanding with each release. For more information on the component definition, see Component definition.

The headers of the component definition file will be generated during component creation, along with sample properties relating to the component type and sub-type selected. Update the properties section of the component definition file to ensure that the App Studio author configuring your component has the best possible authoring experience. This is especially important if you are building a library of reusable Constellation DX components that may need to be configured for use across different business domains or different data model contexts.

Execution contexts of your component

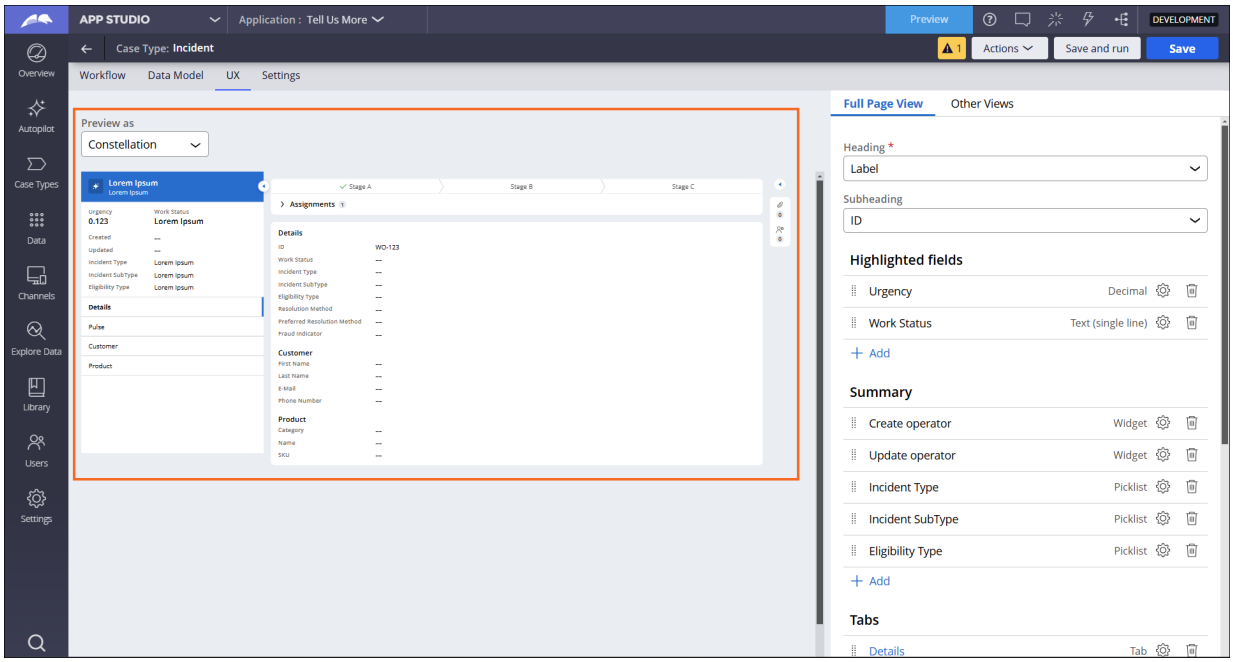
There are two types of component run-time modes:

- UI Authoring Preview mode
- Portal run-time mode

UI Authoring Preview mode

When using UI Authoring in App Studio, your component will be executed by the UI Authoring Preview if you are previewing as Constellation. However, if you are previewing as DX API, your component will not be executed.

©2025 Pegasystems Inc



Preview as Constellation window

Your component is passed mock data from the UI Authoring preview component and this data might not be in an expected format. To prevent your Constellation DX component from displaying an error during preview, ensure that your component handles unexpected data being passed in to standard props as well as mock results returned from the PCore data APIs.

If you need to verify whether the component is being rendered in Authoring mode, you can check for the existence of a redux container called `root/authoring-preview` using the following PCore API:

`PCore.getContainerUtils().areContainerItemsPresent('root/authoring-preview')`

This API will return `true` if the UI Authoring preview is displayed, and `false` if the UI Authoring preview is not displayed. Please note that the API also returns `false` if the **Preview as DX API** option is selected in UI Authoring.

Ensure that your component is rendered without any errors in both contexts. Test the component in the UI Authoring preview mode to ensure that it does not display any rendering errors.

©2025 Pegasystems Inc

Understanding your code

All Constellation DX components are functional React components with hooks that accept a set of props defined in the component props interface.

The current version of React used in Pega Platform '25 is 18. Constellation DX components are specifically designed to be used within the Constellation front-end architecture, which is a React single-page application. Please note that components created with other React frameworks and versions, including those that perform Server-side rendering (SSR), are not supported.

All Constellation DX components are created in the directory specified in the `tasks.config.json` file. Most Constellation DX component Builder projects typically use the default directory of `src/components` located in the Constellation DX component Builder project folder, which is created when you run the `init` command through `npx`.

All components have the following common characteristics:

Main React component code

The `index.tsx` file contains the entry point for the component and contains the majority of the component code.

Import declarations

- This contains the React imports used by your component.
- Several components are imported from the Constellation design system libraries. All Constellation DX components will have at least one import from the `@pega/cosmos-react-core` library. The version of the `@pega/cosmos-react-core` library and other Constellation design system libraries are predefined in your `package.json` file to align with the major version used by the Constellation front-end version that aligns with the version of Pega Platform you are using. You can use any components from the Constellation design system published on npm, but only a subset of these are documented with code examples on design.pega.com. Avoid changing

©2025 Pegasystems Inc

these package.json versions as doing so may cause your component to stop rendering at runtime in a specific version of Pega Platform. For more information about the @pega/cosmos-react-core library, see @pega/cosmos-react-core.

- The `PConnFieldProps` property contains a standard set of types of Constellation DX components.
- A Styled Component wrapper component is automatically generated by Constellation DX Component Builder during component creation and added to your import declarations. A separate file is generated containing boilerplate code for styled components. For more information, see Styling a component using Styled Components.

Interface definition

This is the standard interface definition that describes the properties of your Constellation DX component. It extends the `PConnFieldProps` property, which defines a standard set of properties that are common to most Constellation DX components. The `PConnFieldProps` property is imported from the PConnProps.ts file, which defines the `getPConnect` API and imports the `PConnect` object from the @pega/pcore-pconnect-typedefs package that is automatically included in the package.json file of your project. For more information on the @pega/pcore-pconnect-typedefs package, see @pega/pcore-pconnect-typedefs.

> **NOTE:** Do not modify the PConnProps.ts file, as it might be moved to a shared directory in the future, along with other common files.

The @pega/pcore-pconnect-typedefs package is the TypeScript type module that defines the types of the PCore and PConnect APIs. This package is directly referenced in the tsconfig.json file of the project and provides auto-complete and type-checking features for the PCore and PConnect APIs used in your project.

©2025 Pegasystems Inc

> **NOTE:** The versions of the `@pega/pcore-pconnect-typedefs` package are aligned with the Pega Platform releases. Avoid modifying the version specified in the `package.json` file, as this may result in TypeScript generating type errors in the existing code and in any new components based on the Constellation DX component samples.

Component Render function

The component render function is the main body of the functional component. All sample components need to be tested to ensure that their performance is acceptable for production. We create Effects to retrieve data using the PCore data APIs and also to subscribe and unsubscribe to the PCore PubSub and MessageServiceManager subsystems external to React. For more information about Effects, see You Might Not Need an Effect.

The component developer must perform most of the optimizations as the samples prioritize simplicity over performance.

The Constellation DX component is wrapped in a *withConfiguration* higher order component in the default export. This ensures that styles are properly scoped at run time for UI authoring preview and web embed.

Component definition

For a component to be published, it must have a valid config.json file that is validated against the component schema during the build and publication process. For more information on the component definition, see Updating the component definition.

Storybook

For more information on Storybook stories and the associated mock data, see Creating Storybook mocks for standalone Constellation DX components.

Migration of code to TypeScript

Over time, the samples of Constellation DX components have changed as Constellation DX Component Builder has transitioned to TypeScript and moved away from using PropTypes in the samples. Although TypeScript and PropTypes do not completely overlap, most React components do not use both. TypeScript is

©2025 Pegasystems Inc

used for type-checking during development, while PropTypes can be used to enforce basic run-time type-checking of props passed into React components. If you are migrating from an older version of Constellation DX Component Builder that uses PropTypes and JavaScript, consider migrating your components from PropTypes to TypeScript.

Styling a component using Styled Components

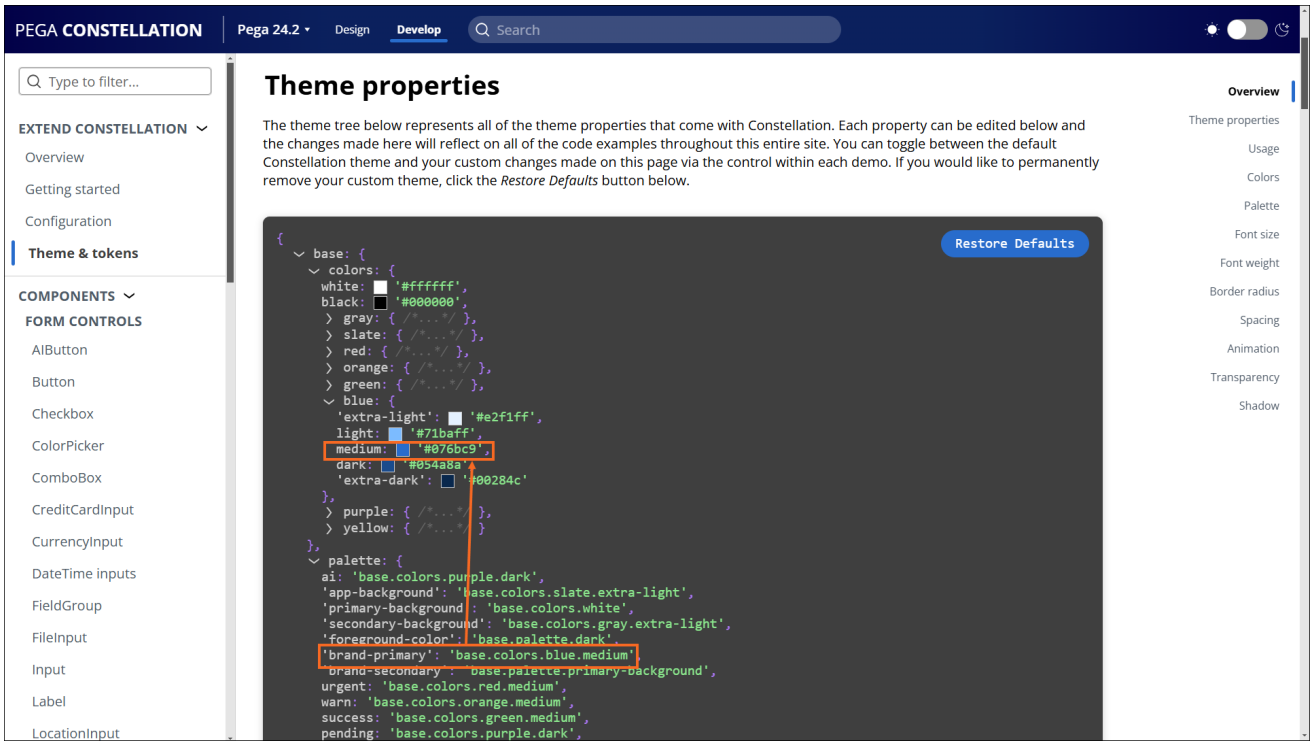
The Constellation design system uses styled components to encapsulate CSS styles for a Constellation DX component in a JavaScript object scoped to your component. This is commonly referred to as CSS-in-JS. For more information on styled components, see Styled Components.

While you will primarily use styles from the Constellation design system, you may need to create your own styled components for advanced component composition or when using a third-party library.

> **NOTE:** Although styled components is the most popular CSS-in-JS solution in the React ecosystem, some third-party components may not offer integration with styled components. In this case, it is necessary to evaluate the use of third-party libraries based on their support for styled components. It is still possible to use third-party components that do not explicitly support styled components, but it is your responsibility to extract the CSS from a stylesheet and wrap it in a styled component, as well as to maintain alignment of these styles with changes in the third-party component.

The Constellation design system provides a wide range of design tokens in the Theme component. For more information, see Theme.

PEGA ©2025 Pegasystems Inc



Theme properties

In the above figure, the highlighted `base.palette['brand-primary']` design token points to the `base.colors.blue.medium` design token, which is set to the static RGB value `#076bc9`.

These tokens can be easily applied to your styled components using the theme object at run time, as shown in the figure below:

©2025 Pegasystems Inc

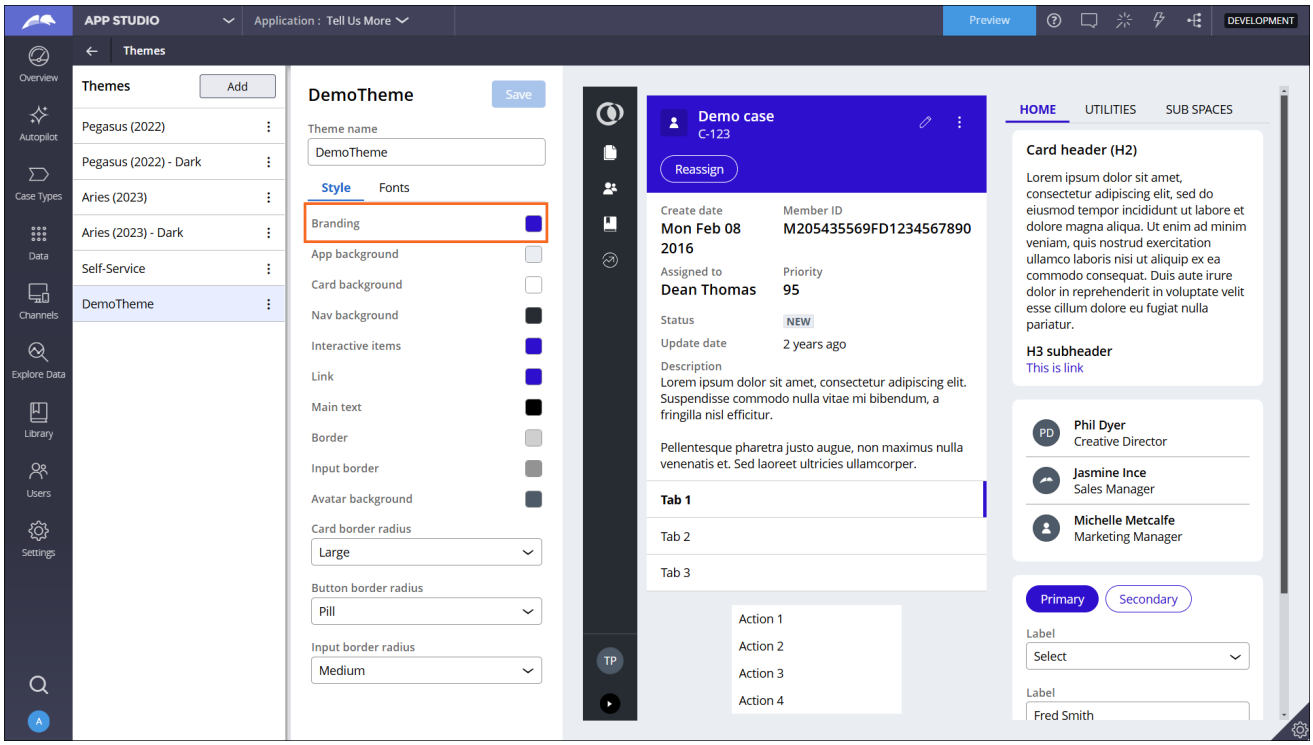
```
src > components > SI_DXExtensions_ExtensionCaseWidget > TS styles.ts > default
import { defaultThemeProp } from '@pega/cosmos-react-core';
import styled, { css } from 'styled-components';

export default styled.div(({ theme }) => {
  return css`
    background: ${theme.base.palette['brand-primary']};
  `;
}).defaultProps = defaultThemeProp;

```

Token for styled components using theme object

In this example, the background color of the component uses the theme design token that represents the brand-primary color provided by the `theme.base.palette['brand-primary']` design token.

Using Constellation design system's design tokens through the theme object ensures that your Constellation DX component will always reflect the portal theme applied in App Studio. This will result in a polished and integrated experience for end users and App Studio authors.

©2025 Pegasystems Inc



Demo theme

In the above figure, we have changed the Branding style that maps to the `theme.base.palette['brand-primary']` design token.

There may be instances where you need to override the applied out-of-the-box theme. In such cases, base your overrides on the original design tokens so that any changes to the token values will also be reflected in your overrides.

Any changes made in App Studio will reflect in the underlying *Rule-UI-Theme* instance and also for advanced theming you can edit the *Rule-UI-Theme* directly in DevStudio as long as you adhere to the Theme properties schema. For more information, see Theme properties.

The Constellation design system and Constellation front-end must have control over where styles are applied in portals and views. To ensure the integrity of Constellation design system styles, Constellation DX components have a special loader that prevents the styles in your component from impacting the global styles. To ensure that your

PEGA ©2025 Pegasystems Inc

Constellation DX component always previews well in UI Authoring and has your styles applied when being rendered through a Web Embed, use the Configuration component provided by the Constellation design system.

All Constellation DX component samples from Pega Platform 23.x onwards use the *withConfiguration* higher order component wrapper provided by the `@pega/cosmos-react-core` library. Therefore, you do not need to perform any additional actions as this wrapper ensures that the Configuration component is properly applied to your component at run time, and that your component styles will be applied in all execution contexts. For more information about the `@pega/cosmos-react-core` library, see @pega/cosmos-react-core.

To handle cases where you want to use the capabilities of the Configuration object directly, remove the *withConfiguration* wrapper in the default export, wrap your component render in a Configuration object, and set the required properties. For more information about the Configuration component, see Configuration.

> **NOTE:** All examples of Constellation DX components are generated with a boilerplate styled components wrapper for you to add your styles. If you do not need to override any of the component styles that you are using, you can remove this from your code along with the associated imports and component reference from the render function.

Working with external libraries

When working with external libraries, check if they are compatible with React 18 and Styled Components v5.

Creating Storybook mocks and unit tests for library mode Constellation DX components

Provide mock data that is representative of the data that your component will display or manage at run time.

©2025 Pegasystems Inc

If you want to use the storybook to preview your component, you must create a PCore and PConnect mock function along with any required data for every function call made from the main component node.

When you create your extension DX component using Constellation DX Component Builder, the mock data relevant to the sample type and sub type will be available for modification in the `mock.ts` or `mock.tsx` file.

The process for updating this mock data for your own component use case depends upon the purpose of your component and the data it needs to function. To develop your component in the storybook, you must provide mock data that is representative of the data that will be provided to your component at run time through the PCore and PConnect APIs.

After creating storybook mocks for each created component, enter `npm run test` to run a unit test on each storybook mock.

The following example is of a `mock.tsx` file used to provide mock data to the PCore.getDataPageUtils().getDataAsync() API.

```
import type { DataAsyncResponse } from '@pega/pcore-pconnect-typedefs/data-view
/types';

export const newRating = {
  CustomerRating: 0,
  NumberOfStars: 5,
  CaseID: 'SL-TELLUSMORE-WORK Z-12345',
  CaseClassName: 'SL-TellUsMore-Work-Incident',
  CustomerID: 'Q1234',
  pxUpdateDateTime: '2020-06-29T11:06:24.329Z'
};

const ratingData: Omit<DataAsyncResponse, 'config'> = {
  data: [
```

©2025 Pegasystems Inc

```
    {
      CustomerRating: 3,
      NumberOfStars: 5,
      CaseID: 'SL-TELLUSMORE-WORK Z-1234',
      CaseClassName: 'SL-TellUsMore-Work-Incident',
      CustomerID: 'Q1234',
      pyGUID: '6876786876876868123',
      pxUpdateDateTime: '2020-06-29T11:06:24.329Z'
    },
    {
      CustomerRating: 3,
      NumberOfStars: 5,
      CaseID: 'SL-TELLUSMORE-WORK Z-123',
      CaseClassName: 'SL-TellUsMore-Work-Incident',
      CustomerID: 'Q1234',
      pyGUID: '68767868768768681',
      pxUpdateDateTime: '2020-06-29T11:06:24.329Z'
    }
  ],
  status: 200
};

export default ratingData;
```

This example mock data above is used in the `demo.stories.tsx` file as shown below to provide a mock of the PCore.getDataPageUtils().getDataAsync() function.

```
import mockRatingData from './mock';
import type DataPageUtils from '@pega/pcore-pconnect-typedefs/datapage/index';
import type { DataAsyncResponse } from '@pega/pcore-pconnect-typedefs/data-view
/types';
```

©2025 Pegasystems Inc

```
const mockGetDataAsync = (
  ...args: any[]
): Promise<Partial<DataAsyncResponse>> => {
  const filter = args[4]?.filter as Filter;
  const queryCustomerID = filter?.filterConditions.F1.rhs.value;
  let { data } = mockRatingData;
  if (queryCustomerID?.length)
    data = data.filter(rating => rating.CustomerID === queryCustomerID);

  return Promise.resolve({ data, status: 200 });
};

const mockDataPageUtils = (): Partial<typeof DataPageUtils> => {
  return {
    getDataAsync: mockGetDataAsync as () => Promise<DataAsyncResponse>
  };
};


...
```

Previewing library mode Constellation DX components in Storybook

View how the existing Constellation DX components are displayed at run time by opening the components in Storybook. Open the project and then start Storybook in development mode so that you can view each component separately.

> **NOTE:** If the components have calls to Constellation orchestration layer APIs, disable the calls or create an API mock to view these components in Storybook.

1. Open your preferred terminal tool.

©2025 Pegasystems Inc

2. Enter `cd <name of the project>`.
3. Enter `npm run startStorybook`.

> **NOTE:** To test the different behaviors of your component, go to the `index.tsx` file and change the default values in your <component name>. `defaultProps` while the Storybook is running.

> **Result:**
>
> You can now view each existing Constellation DX component in Storybook. To close Storybook, use `ctrl + c` and then close the browser tab.
>
> For more information on Storybook, see https://storybook.js.org/.

Validating library mode Constellation DX components

Test schema, lint, and compile issues to ensure that your library mode components build and publish without errors. Confirm that the components in a project meet quality standards before deployment.

You use the `validate` or `validateAll` commands to validate library mode components.

The `validate` command validates a specific component. It validates the component schema, performs linting, and bundles the component. These actions also occur during publishing. However, validate the component locally to ensure that it compiles without errors before publishing it to the server.

The `validateAll` command validates all components in a project. It runs schema and lint checks for each component, copies the current version of the library to the

©2025 Pegasystems Inc

store, and then builds a temporary library to ensure that the components build with each other successfully in the library. You can find the bundled components in the `dist` folder of your project's root folder.

NOTE:

- Pega recommends that you use the `validateAll` command to validate your components because it validates the components in the context of a library.
- If changes made to the `config.json` file are not validated, the validation of the component schema might fail during the building and publishing process. To prevent this, ensure that you validate any changes made to the `config.json` file. For more information about the structure and attributes of the `config.json` file, see Component definition.
- Warnings might be displayed when linting is being performed. These warnings do not prevent publishing.

1. Open a terminal at the Constellation DX component directory and enter `npm run validate`.
2. In the terminal, select the component that you want to validate.

Result:

The Constellation DX Component Builder tests schema, lint, and compile issues, and then validates the code for the selected library mode component.

Related information

- validate
- validateAll

©2025 Pegasystems Inc

Authenticating Pega Platform Server user access for library mode Constellation DX components

To publish, list, or delete a library mode Constellation DX component, you require credentials to access the resources on the Pega Platform™ Server.

Before you begin:

- Ensure that the credentials you enter for authentication has access to your application and the unlocked Rulesets to which you want to publish your component library.
- Ensure that the `server`, `clientID`, `ruleset`, and `rulesetVersion` are set up in the `tasks.config.json` file before authentication.
- Ensure that the `Allow rule check out` access setting under the **Security** tab has been added to the operator record used for authentication and publication.
- Ensure you have a valid operator ID in your Pega Platform development instance that you can use to log in to your Pega applications.
- Ensure you are using Pega Platform '25 or a later version.

NOTE:

- After completing the authentication, you must reauthenticate yourself after the default timeout of 15 minutes. You can change the timeout period by editing the *REST* API Auth 2.0 Client registration record. The record contains a `clientID` that matches the configuration contained in the `tasks.config.json` file of each component.
- Constellation DX components require you to use the OAuth 2.0 authentication protocol.

©2025 Pegasystems Inc

- If you do not want to use the provided client credentials, you can create your own client credentials. If you use a different OAuth 2.0 authentication flow in the client credentials, you must update your `tasks.config.json` file accordingly. For more information on the various configuration options for the `tasks.config.json` file, see server-config. The relevant attributes in the `server-config` section are `clientId`, `clientSecret`, `grantType`, and `authService` (if you need to integrate Constellation DX Component Builder with your SSO setup). For more information on setting up alternative authentication schemes that can be used with Constellation DX Component Builder, see OAuth 2.0 client registrations.

1. In Dev Studio, click **Records** > **Security** > **OAuth 2.0 Client Registration**. If the **DX-ComponentDev** record is visible in the list of instances, you already have a valid client credential. Go to step 7 to revalidate it.
   If the **DX-ComponentDev** record does not exist, import it as directed in the following steps.
2. In Dev Studio, click **Configure** > **Application** > **Distribution** > **Import**.
3. On the **Application: Import wizard** tab, click **Local file**, and then click **Choose file**.
4. Navigate to the DX component project's `keys` folder, select the `DX-ComponentDev.zip` file, click **Open**, and click **Next**.
5. Select the **Enable advanced mode to provide more granular control over the import process** checkbox and click **Next**.
6. On the **Import Wizard File : DX-ComponentDev.zip** tab, verify that the DX-COMPONENTDEV instance is selected, and click **Next**.
   The import process is complete. The **DX-ComponentDev** record is now present in the list of instances.
7. In your terminal, authenticate user access to the Pega Platform Server by running the `npm run authenticate` command.
   This command sets up a token internally and retrieves the information necessary for authentication.

©2025 Pegasystems Inc

NOTE:

- After authentication, the JWT access token is retrieved. You do not need to authenticate again as long as the JWT access token does not time out.
- If you update the Ruleset list of your application, you must authenticate access to the Pega Platform Server again.

8. Enter your operator ID and password, and click **Log in**.
   If the credentials are valid, an authentication success message is displayed.

Result:

You now have authentication to publish, list, or delete library mode Constellation DX components.

Publishing a component library

Publish your component library to the Pega Platform™ server so that you can discover the components in the component palette and use them to build the UI of your application.

Before you begin:

Before you publish the component library, confirm that your environment meets the following requirements:

- Authenticate to access the resources in the Pega Platform server. For more information, see Authenticating Pega Platform Server user access for library mode Constellation DX components.
- Map your default Access Group to the application where you want to publish the component library. If you use a different Access Group, enter the complete application URL alias.

©2025 Pegasystems Inc

- Access the Ruleset to which you want to publish your component library.
- Publish the library to a Ruleset that is not a Branch Ruleset. Publishing the library to a Branch Ruleset results in an error.
- Publish the library to an existing unlocked ruleset version. Publishing the library to a locked ruleset version results in an error.
- Verify that the Pega Platform application record to which you are authenticated includes the ruleset version.

1. Open your terminal tool, enter `npm run publish`, and then run the command.

> **NOTE:** If the version of the library already exists on the server, the system overwrites it. If the version does not already exist, create a new version by running the `npm run createLibVersion` command. Copy the components into the new version, and then publish it.

2. Complete the parameter values:
   a. Select the library that you want to publish.
   b. In the `rulesetName` parameter, enter the name of the Ruleset.
   c. In the `rulesetVersion` parameter, enter the version of the Ruleset.

> **NOTE:**
> - Publish the library to a separate Ruleset from the Ruleset of your application. Include the ruleset version of the published library in your application's Ruleset stack.
> - The system automatically stores the entered ruleset version in the `tasks.config.json` file.

Result:

You have published your library to the server.

©2025 Pegasystems Inc

Add the components of the library to the application, so that the components are available at run time. For more information, see Adding library mode Constellation DX components to an application.

NOTE:

- The system creates the local library after the validation of the components.
- After the publication of the library, Pega Platform creates the corresponding *Rule-UI-ComponentLibrary* Rule, associated *Rule-UI-Components* Rule, JS files as *Rule-File-Text*, and the component library as *Rule-File-Binary*. For more information, see Rule resolution for library mode components.
- After publishing a library, if you update its components and then publish the library to another Ruleset or ruleset version, the updated components might not reflect in the run-time portal. This happens because the browser cache contains the older components and not the updated components. With browser cache enabled, the components are fetched from the browser cache instead of the server. To resolve this, save the application Rule after re-publishing the updated library. The browser cache is updated and the updated components reflect in the run-time portal.
- If you cannot find the component in App Studio after publication, refresh the browser.
- In some cases, the system might not display the component preview in App Studio. However, you can still configure the component.
- When you view a published component in App Studio or any Constellation Portal, you view the latest published version of the component for that environment based on Rule resolution. With Rule resolution, if you publish the component library to a lower

©2025 Pegasystems Inc

> ruleset version, you might not view the latest published version of the component.

What to do next after publishing the component library

You can perform any of the following actions:

Publish **a new version of the library with updated code**

1. Create a new library version using the `createLibVersion` command.

   For more information on creating a new library version, see createLibVersion.

2. Update your code and publish the new library version to the same ruleset version.

Reuse existing components

You can reuse an existing component by making a copy of it and modifying it. For more information, see Reusing existing library mode components.

Prepare for production release

1. Lock your ruleset version.
2. Archive your library version. For more information, see exportLibVersion.
3. Create a new ruleset version.
4. Create a new library version, update your code, and publish the new library version to the new ruleset version.

Create new components that are not related to your existing components

1. Create a new library. For more information, see Creating a component library.
2. Create new components.

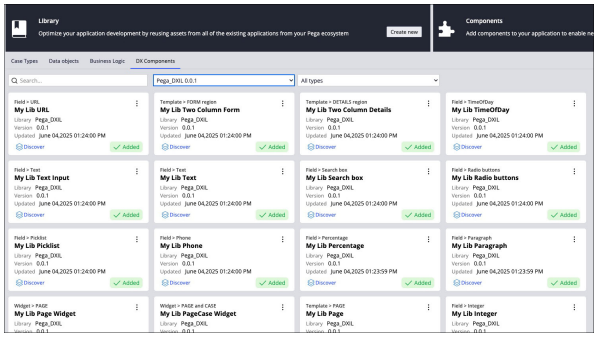
©2025 Pegasystems Inc

3. Create a new ruleset version.
4. Publish the library to the new ruleset version.

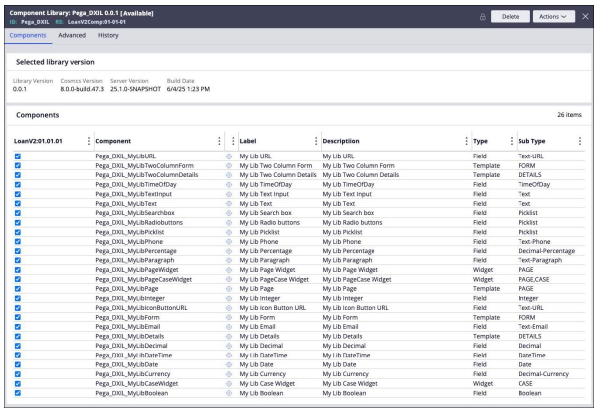
Adding library mode Constellation DX components to an application

After publishing a component library, add the components of the library to the application so that the components are available at run time.

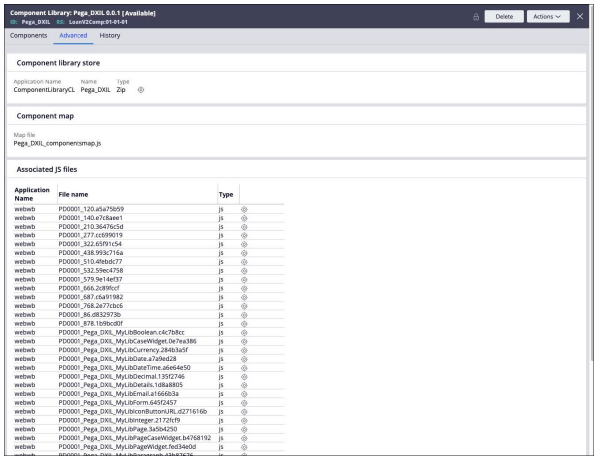
Add the components of the library to the application using either App Studio or Dev Studio:

Choices	Actions
App Studio	a. In the navigation pane of App Studio, click **Library**. b. Click the **DX Components** tab. c. In the **All libraries** list, select your library. The system displays the components in your library. d. Add a component to the application by clicking **More vertical** > **Add to application**. 

©2025 Pegasystems Inc

Choices	Actions
	Adding components to an application in App Studio
Dev Studio	a. In the navigation pane of Dev Studio, click Records. b. In the **Records Explorer**, click User Interface > Component Library. c. On the Instances of Component Library tab, click the name of your library. The **Components** tab displays the list of components in the selected library. d. Add a component to the application by selecting its checkbox.  *Adding components to an application in App Studio* **NOTE:** If your library is published to a Ruleset that is used by several applications, the Ruleform displays

©2025 Pegasystems Inc

Choices	Actions
	only the components available to your current application. The **Advanced** tab displays the published component library and its associated JavaScript files as *Rule-File-Text* that will be used at run time. The JavaScript files are read-only because they are managed by the *Rule-UI-Component*.  *Viewing published component library*

Result:

The components added to the application are available at run time.

Debugging the component library

Test library mode components in the Pega Platform™ environment with a local server.

©2025 Pegasystems Inc

Before you begin:

Ensure that the library that you want to debug is built and deployed, and that you are authenticated to the server.

To debug components, you require complete access to the code in development mode. Because library mode does not have a development mode, you can use a local server to serve component JS files and map files with which to perform debugging.

1. In the navigation pane of Dev Studio, click **Records**.
2. Expand the **Decision** category, and then click **When**.
3. In the list of When Rules, open *pxEnableC11nDev*.
4. Click **Private edit**, and then in the **Private Edit** dialog box, click **Continue with private edit**.

> **NOTE:** Perform a private edit only if you are an administrator and want the changes to the *pxEnableC11nDev* Rule to be visible only to you. If you want to make the changes to the *pxEnableC11nDev* Rule visible to users, check out the Rule to a shared branch.

5. On the **Advanced** tab, in the first condition row, enter `true`, and then click **Save**.
   Evaluating the condition to true enables the ability to point to a local server.
6. In the lower-left corner of Dev Studio, click the user profile icon, and then click **Operator**.
7. Edit the operator record with which you want to debug the component library.
8. On the **Security** tab, in the **Constellation DXCB Dev URL** field, enter `http://localhost:3030/`, and then click **Save**.
   This URL represents the local server.
9. Open a terminal, and ensure that you are in the library version that you want to debug.

©2025 Pegasystems Inc

10. Open your `tasks.config.json` file, and verify that the `devServerPort` property is set to `3030`.

> **NOTE:** To change the value of the `devServerPort` property, update the value in Dev Studio as well.

11. In the terminal, enter `npm run startDevServer` to start the Dev server. This command rebuilds your library locally and starts a server that serves the components.
12. In Dev Studio, in the **Launch Portal** list, select the application to run with the previously selected operator.

> **Result:**
>
> You can now debug the library components in the browser.

What to do next

Perform the following debugging actions through the browser:

- Set break points.
- Add debugger statements in your code.
- If you started the local server from the Visual Studio Code terminal, you can edit your component code live, and save it. After the code is saved, the Dev server rebuilds your library and serves it, which causes the browser to refresh.
- Stop the Dev server by pressing Ctrl+C.

> **NOTE:**
>
> - You can debug only one library at a time. If you have more than one library in your application or application stack, the remaining libraries

©2025 Pegasystems Inc

and their components will not be visible in your application during the debugging process.
- If you stop the local server, your components will not be visible in your application.
- After debugging is complete, go to your operator record in Dev Studio and delete the value of the **Constellation DXCB Dev URL** field. This must also be performed before creating a RAP file of your application to avoid causing issues with your components.
- To apply your code changes, you must publish your code to Pega Platform because your changes were made only to the local server.

Localizing a library mode Constellation DX Component

Provide multi-language support for your library mode component by integrating it with out-of-the-box Constellation localization features.

To ensure that your library mode components are properly localized,

- integrate your component with the appropriate localization APIs
- externalize your static string data into a component-specific `localizations.json` file

For more information on the localization APIs, see

- getLocalizedValue(rawString, localePath, localeRuleKey)
- APIs in the LocaleUtils class
- APIs in the LocalizationService class

Your `localizations.json` file must be added to your component's root folder to ensure that it is packaged with your component when the component is published.

The following code contains an example of a `localizations.json` file:

©2025 Pegasystems Inc

```
{
  "fields": {
    "Edit": "Edit",
    "Navigate to step": "Navigate to step"
  }
}
```

Some of the template components have examples including Icon Button Url, Picklist, Search box, Radio buttons, Case Widget, and Page Widget.

For more information about localizing your Constellation application, see Internationalization and localization.

Commands used in library mode

In library mode, some original commands remain unchanged, some have been modified, some are disabled, and several new commands have been introduced.

The following table illustrates the changes:

Name	Description
`authenticate`	Authenticates the Pega Platform™ Server, user credentials, and OAuth 2.0 credentials based on the configuration of the `tasks.config.json` file. For more information, see Authenticating Pega Platform Server user access for library mode Constellation DX components.
`create`	Creates a component. In library mode, the length of the component's name is limited to a maximum number of characters. The maximum number of characters varies

©2025 Pegasystems Inc

Name	Description
	depending on the string lengths of the organization, library, version, and other constants. For more information, see Creating a component library.
`createAll`	Creates a version of each type of library mode component with a label prefixed to the name of each component. The length of the prefix is limited to a maximum number of characters. The maximum number of characters varies depending on the string lengths of the organization, library, version, and other constants. For more information, see this note about using the `createAll` command.
`createLib`	Creates a library for your components. Enter the following details: • Name of the library. Must be alphanumeric without any spaces. • Version of the library. The code base of the created library is located in the `src/components` directory. When you run the `createLib` command to create a new library, the current library version is archived in the store. The new library is created and its code base is located in the same `src/components` folder. Because the newly created library has no components, the `src/components` folder is empty.

©2025 Pegasystems Inc

Name	Description
	The `tasks.config.json` file is automatically updated to point to the created library. For more information, see Creating a component library. **NOTE:** • In library mode, only the `createLib` and `importLibVersion` commands function until a library is created using either of these commands. • You can import components into the newly created library. If you want to import standalone components into library mode, use the `importComponent` command. If you want to import components created in library mode, use the `move` command. To copy or move the components, you must switch to the library containing the components and then copy or move the components from it. You can switch between libraries using the `switchLib` command.
`createLibVersion`	Creates an additional version of an existing library. By default, the existing components are copied into the new library version. All existing code will be copied into a store along with the `package.json`, `tasks.config.json`, and other config files.

©2025 Pegasystems Inc

Name	Description
	If you choose to not copy the existing components into the new library version, an empty `src/components` directory is created. **NOTE:** While `createLibVersion` automatically increments the current version of your library, you can modify the version manually.
`delete`	Deletes the specified local library mode component from the `src/components` directory.
`deleteAll`	Deletes all local library mode components from the `src/components` directory.
`deleteLib`	Deletes a component library or a version of a library. • If you opt to delete a library or its version that you are currently working on, you receive a warning that all your components will be deleted. • If you proceed with deleting the library or its version, any existing older version of the library is restored as the current library. If there are no older versions of the library, the first library in your store is restored as the current library. • If there are no libraries remaining, you must create a library by using the `createLib` command or import an archived library using the `importLibVersion` command before running other commands.
`deletePublishedLib`	Deletes a component library from your local project or server.

©2025 Pegasystems Inc

Name	Description
	NOTE: Ensure that the library you want to delete has been published before running the `deletePublishedLib` command. Select the location from which you want to delete the component library: • `Local` for component libraries in your local project. You can select a library to delete or delete all libraries. • `Server` for component libraries that are already published. A list of libraries for your application is displayed. Select the library to delete. **NOTE:** The local stored library can differ from the component library published locally or to the server. To match the libraries, you must publish a library.
`exportLibVersion`	Exports a library in the form of a ZIP file to a specified directory or archives the library to Pega Platform. Select the location to which the library must be exported: `Local` Select the library version to be exported and enter the directory to which the library must be exported. The default directory is `User/<username>/Downloads`. If the default path is changed, the updated path will be saved as the default path for the next export. The updated path is stored in the `tasks.config.json` file.

©2025 Pegasystems Inc

Name	Description
	`Server` Select the library version to be exported and enter the Ruleset name and Ruleset version to which the library must be published in Pega Platform. • Ensure that server authentication has been performed before publishing to server. • Publish the library to a separate Ruleset version. • The entered Ruleset name and Ruleset version are stored in the `tasks.config.json` file as `archiveRulesetName` and `archiveRulesetVersion`.
`help`	Shows commands related to library mode components. For more information, see Listing commands. **NOTE:** The list of commands that is displayed varies depending on whether library mode is active.
`importComponent`	Imports a standalone component from a specified directory into your library. • Ensure that the specified directory is located in `User/<username>`. • While the name of the component remains unchanged, the values of org and library update to their corresponding values in the destination library. • The value of `version` in the `config.json` file is updated to match the version of the destination library.

©2025 Pegasystems Inc

Name	Description
	• If you change the default path, it is saved for the next import. The updated path is stored in the `tasks.config.json` file. • Validate the component to be imported because it might require updates to dependencies in the `package.json` file before publication. For more information, see Importing and publishing standalone components.
`importLibVersion`	Imports an archived library in the form of a ZIP file from a specified directory or server to the store. Select the location from which the archived library must be imported: `Local` Select the library version to be imported and enter the directory from which the library must be imported. The default directory is `User/<username>/Downloads`. If you change the default path, the updated path is saved as the default path for the next import. The updated path is stored in the `tasks.config.json` file. **Server** Select the library version to be imported from the Pega Platform server. Ensure that server authentication occurs before importing from the server. • The organization of the imported library updates to match the organization of the local library. This value is updated in the `componentKeys`, the `config.json` file, and the code.

©2025 Pegasystems Inc

Name	Description
	• If you import a library version that is the same as your current working library version, it will be added to the store. To update your current working library version with the imported code, run the `restoreLibVersion` command. • If you have an existing library, the imported library is imported to the store. You must switch to the imported library to make it your active library. • If you do not have any existing libraries, the imported library becomes the active library. • The imported library might contain an older `package.json` file. After switching to the imported library, the `dependencies` and `devDependencies` attributes update to match those used in Constellation DX Component Builder. Verify the custom and additional dependencies and update them, if required. After updating the `package.json` file, run the `npm update` command so that the node_modules contain the correct packages to ensure error-free publishing.
`list`	Lists the local library mode components located in the `src/components` directory.
`listLib`	Lists the components located in a specific library in the store. If you select your current library version, you can opt to display either the components that are currently in the `src/components` directory or those in the store.

©2025 Pegasystems Inc

Name	Description

`listPublishedLib`	**NOTE:** Ensure that the libraries you want to list are published before running the `listPublishedLib` command.
	
	Lists the components located in a specific library in your local project or server.
	Select the location from which you want to list the components:
	
	`Local`
	For component libraries in your local project.
	`Server`
	For component libraries that are already published.
	
	The local stored library can differ from the component library published locally or to the server. To match the libraries, you must publish a library.

`move`	Moves or copies components from your current library to another library. • Use the `move` command to move or copy components instead of the file system to prevent any instability in the library components. This command updates the code base and the corresponding files. • The data of the components that have moved is updated in accordance with the updated library version. • You can create a copy of components in the same library version. You must provide a name and a label to the copied components, while the code is automatically

©2025 Pegasystems Inc

Name	Description
	updated. The description in the `config.json` file can be manually updated, if required.
`publish`	Creates a local component library and then publishes the library to Pega Platform. • A component library cannot be published to a Branch Ruleset. • The local library is created after the validation of the components. • After the creation of the local library, Pega Platform creates the corresponding *Rule-UI-ComponentLibrary*, associated *Rule-UI-Components*, JS files as *Rule-File-Text*, and the component library as *Rule-File-Binary*. • Ensure that authentication occurs before running the `publish` command. • Ensure that you publish the library to an existing ruleset version that is unlocked. Publishing the library to a locked ruleset version results in an error. • Ensure that the Pega Platform application record to which you are authenticated includes the ruleset version. • Ensure that you publish the library to a separate Ruleset from the Ruleset of your application. Include the ruleset version of the published library in your application's Ruleset stack. For more information, see Publishing a component library.
`rename`	Renames the selected library mode component. • Use the `rename` command to rename components instead of the file system to prevent any instability in the

©2025 Pegasystems Inc

Name	Description
	library mode components. This command updates the code base and the corresponding files. • The library and organization of the component do not update. • To change the library of a component, create a new library, switch to the old library, and then move the component to the new library. • To change the organization of a component, create a new project and choose a different organization. If you want your component to exist in the new project, export the library from your old project, create a new library in your new project, and then import the library to your new project. The imported library updates to the new organization and library.
`restoreLibVersion`	Replaces the current version of the library with the version of the current library archived in the store. Use this command when you want to revert the changes made to a library. `restoreLibVersion` restores the last saved library version from the store. The library versions are automatically stored when you run the `exportLibVersion`, `storeLibVersion`, `validateAll`, `publish`, `switchLib`, or `switchLibVersion` commands or when you import the same library version from an archive by using the `importLibVersion` command. To ensure that you have a library version that can be restored, run the `storeLibVersion` command regularly.

©2025 Pegasystems Inc

Name	Description
`showStatus`	Displays the status of the current library, such as its organization, library, version, component count, and the expected ruleset version to which the library can be published.
`startDevServer`	Builds the current library and starts a local server to serve the library components. You can use this local server to test library mode components in a Pega Platform environment after setting up an operator. For more information, see Debugging the component library.
`startStorybook`	Starts the storybook to externally test or mock your components. For more information, see Previewing library mode Constellation DX components in Storybook.
`storeLibVersion`	Copies the current version of the library to the store. This happens automatically when you export, switch, publish a library, or validate all components.
`switchLib`	Switches from your current library to another library in the store. • Your `tasks.config.json` file is restored and the `dependencies` and `devDependencies` from the `package.json` file of your previous library will be copied to the `package.json` file of your current library. • If there are differences in the `dependencies` or `devDependencies`, the system asks you to run the `npm update` command to sync your node modules. • Save your code before you switch libraries to avoid losing any unsaved work.
`switchLibVersion`	Switches between versions of your current library in the store.

©2025 Pegasystems Inc

Name	Description
	• The existing code in the `src/components` folder is archived and is replaced with the code of the library you are switching to. • Your `tasks.config.json` file is restored and the `dependencies` and `devDependencies` from the `package.json` file of your previous library version are copied to the `package.json` file of your current library version. This ensures that any additional `devDependencies` of your current library version are retained. • If there are differences in the `dependencies` or `devDependencies`, the system asks you to run the `npm update` command to sync your node modules. • Save your code before you switch library versions to avoid losing any unsaved work.
`test`	Runs unit tests on each component created. Unit tests are run against mocks in storybook. For more information, see Creating Storybook mocks and unit tests for library mode Constellation DX components.
`validate`	Tests schema, lint, and compile issues and validates the code for a component. For more information, see Validating library mode Constellation DX components.
`validateAll`	Tests schema, lint, and compile issues and validates the code for all components. • Use the `validateAll` command to validate your components because it validates the components in the context of a library.

©2025 Pegasystems Inc

Name	Description
	• Running the `validateAll` command also copies the current version of the library to the store. For more information, see this note about using the `validateAll` command.

Working with components in library mode

Learn about the additional actions you can perform using components in library mode.

You can perform the following actions on library mode components:

- Working with component libraries
- Importing and publishing an existing library
- Importing and publishing standalone components
- Sharing files with library mode components
- Reusing existing library mode components
- Deleting a component from a published library in the server
- Important considerations

Working with component libraries

Learn about the additional considerations when working with component libraries.

Creating a library

When you run the `createLib` command to create a new library, the current library version is archived in the store. The system creates the new library and places its code base in the same `src/components` folder. Because the newly created library has no components, the `src/components` folder is empty. For more information, see createLib.

©2025 Pegasystems Inc

You can copy, move, or import components into the library. To copy or move the components, you must switch to the other library and then copy or move the components from it. You can switch between libraries using the `switchLib` command. For more information, see switchLib.

Creating a library version

When you run the `createLibVersion` command to create a new version of the library, the system archives the current library version to the store and you can choose to copy the existing components into the new library version. The new library version is created and its code base is located in the same `src/components` folder. For more information, see createLibVersion.

The `store` folder cannot be recovered if it is deleted. To create a backup of this folder, you can export each library version using the `exportLibVersion` command, or copy the `store` folder to another location, or use Git to store the project.

To switch between library versions, use the `switchLibVersion` command. For more information, see switchLibVersion.

Archiving libraries

Use the `exportLibVersion` command to archive a library in the form of a ZIP file to a specified local directory or to the Pega Platform Server.

The archived file has the folllowing naming convention: `<org>_<lib>_<version>.arch.zip`. For example, `Acme_Lib1_0.0.1.arch.zip`. You can rename the file, but you must retain the `arch` suffix. To view the archived files on the server, run the `importLibVersion` command and select **Server**. Alternatively, perform the following actions:

1. In the navigation pane of Dev Studio, click **Records**.
2. In the **Records Explorer**, click **Technical** > **Binary File**.

©2025 Pegasystems Inc

3. On the **Binary File** tab, filter the **Application Name (Directory)*** column by `ComponentLibraryArchive`.

For more information, see exportLibVersion.

Restoring libraries

Use the `importLibVersion` command to restore an archived library from a local directory or from the Pega Platform Server.

The imported library is placed in the `store` directory. If you import a library when the project does not have any existing libraries, the `store` directory is created and the imported library becomes the working library.

For more information, see importLibVersion.

Backing up and restoring the store

Back up and restore the `store` folder for storage and archival purposes.

You can copy the `store` folder to another location for storage or archival purposes. If you need to restore a copy that you archive, replace the current `store` folder with the backed-up `store` folder.

NOTE:

- Before restoring an archived copy, ensure that the values of the `libraryName` and `version` attributes in the `tasks.config.json` file match the corresponding values of the library in the archive. Also, ensure that the value of the `organization` attribute in the `package.json` file matches the corresponding value of the library in the archive.
- After restoring the archive, run the `restoreLibVersion` command to restore the matching library as the active library. You can then switch libraries or versions.

©2025 Pegasystems Inc

- If you run other commands before running `restoreLibVersion`, you might lose data in the `store` folder. The store should be synced with DXCB.

Importing and publishing an existing library

Learn how to import and publish an existing library of components.

1. Open a terminal window and enter `npm run importLibVersion`.
2. Enter the path to the library that you want to import, and then select the ZIP file. The terminal may prompt you to run the `npm update` command if the `package.json` file differs from the previous library. Go through the `package.json` file and then run the `npm update` command. This installs any dependencies required by the imported library's `package.json` file.
3. Enter `npm run showStatus` and verify your ruleset version and server.
4. Enter `npm run validateAll`.
5. Enter `npm run publish`.

Result:

The imported library is published.

Importing and publishing standalone components

Learn how to import standalone components into a library and publish the library.

In the `package.json` file, check whether the components are from the same org and library, and have the same organization and then perform one of the following actions:

©2025 Pegasystems Inc

Choices	Actions
The components are from the same org and library, and have the same organization in the `package.json` **file**	a. Create a new project and enter the organization that matches the org of your components. b. Copy your component folders into the new project's `src/components` directory. c. Open the `tasks.config.json` file and set `libraryModeCL` to `true`. d. Enter `npm run createLib` and ensure that your library name matches the library name in the `tasks.config.json` file. The `config.json` file is updated to match the version and creates an **about** section. e. Decide whether you want to update your code base for ESLint changes: enter `Y` to update your code base for ESLint changes (even though the code base is updated, you are responsible for any manual updates), or enter `N` to retain your code base. f. Enter `npm run authenticate` to authenticate the components, if required. g. Enter `npm run publish` to publish the library.

©2025 Pegasystems Inc

Choices	Actions
The components are not from the same org and library	a. Create a new project and enter the organization that matches the org of your components or enter the organization that will be your new org. b. Create a temp folder and move the components to that folder. c. Open the `tasks.config.json` file and set `libraryModeCL` to `true`. d. Enter `npm run createLib` to create a library. e. Enter `npm run importComponent` to import each component into the newly created library. All the imported components now have the same org and library. f. Update your `package.json` file. For more information, see Updating the package.json file. g. Enter `npm run validateAll` to validate all the components and copy the current version of the library to the store. h. Open the `tasks.config.json` file and verify your ruleset version and server URL. i. Enter `npm run authenticate` to authenticate the components, if required.

©2025 Pegasystems Inc

Choices	Actions
	j. Enter `npm run publish` to publish the library.

If multiple standalone components have different library names

If the standalone components have different library names, you can perform either of the following actions:

- Import the components into a specific library. The components will be renamed. The applications must be updated separately.
- Create separate libraries and import the components to each library. Include the library Rulesets in the applications using the components.

If multiple standalone components have the same library name

If the standalone components have the same library name, you can perform either of the following actions:

- Let the components remain in the same library. Include the library Ruleset in the applications using the components.
- Import the components into separate libraries of the same name, but different rulesets because components were possibly created with different projects. This could however, lead to issues with applications missing components because rule resolution resolves to only one library version of the same name.

Ensure that you create libraries that represent the nature of your components.

Advantages and disadvantages of having components in one library

Advantage - Common code like react is only loaded once.

Disadvantage - The large code base requires maintenance.

Advantages and disadvantages of having components across multiple libraries

Advantage - Less bloat, code is compartmentalized, and code is represented in a cleaner way.

Disadvantage - Common code like react is loaded more than once.

©2025 Pegasystems Inc

Updating the package.json file

After you import standalone components into a component library, update the values of `dependencies` and `devDependencies` in your `package.json` file with the current code.

> **Before you begin:**
>
> Open your `package.json` file in Visual Studio Code.

1. Copy the values of the `dependencies` from your previous `package.json` file to the `package.json` file in your new project.
   Visual Studio Code displays a list of duplicate dependencies.
2. Delete the older copies of the duplicate dependencies.
   The `package.json` file in your new project now contains both your previous and latest dependencies.
3. Ensure that all the included cosmos packages have the same version.
4. Repeat steps 1 and 2 for `devDependencies`.
5. Save the updated `package.json` file and enter `npm run update`.

> **Result:**
>
> The `package.json` file is updated with the current code.

Sharing files with library mode components

When you publish a component library, you can share code between its components.

When you create components in library mode, the system creates a `shared` folder in the `src/components` folder. Components that you create using the `create` or `createAll` commands share a few files by default. You can add files to the `shared` folder and use the following import statement in the code:

©2025 Pegasystems Inc

```
import <file_name> from '../shared/<file_name>';
```

NOTE:

- While it is recommended to use the `shared` folder for common code, do not create any other folders in the `src/components` folder yourself. This is because the Constellation DX Component Builder will not recognize these folders when you try to move or copy the shared components in these folders, or when the existing components are copied into the new library version through the `createLibVersion` command.
- You can create subfolders inside the `shared` folder itself.
- The `shared` folder is packaged when the library is built.
- At run time, the shared files are loaded as required.

Reusing existing library mode components

Reuse an existing library mode component by making a copy of it and modifying it.

1. Open a terminal window and enter `npm run move`, select `Copy`, and then select the same library and version.
2. Enter a name and label for the component.

NOTE: If necessary, update the description in the `config.json` file manually.

Result:

A new component is created with the code of the existing component and the new name that you entered.

©2025 Pegasystems Inc

Deleting a component from a published library in the server

Learn how to delete a component from a published library in the server.

Before you begin:

- Before you delete a component, remove all instances of its use in App Studio. This action prevents any errors from occurring after the deletion of the component.
- If your application uses this component, remove the availability of the component from the application. You must either remove the component from the *R-UI-ComponentLibrary* Ruleform in Dev Studio or from the DX Components landing page in App Studio.
- Ensure that your current library is the published library that contains the component you want to delete. If your current library is a different library, switch to the published library by running the `switchLib` or `switchLibVersion` commands.

Verify whether the component that you want to delete is the only component in the library, and then perform the deletion depending on the status of your component:

Choices	Actions
The component is the only component in the library	In your terminal, enter `npm run deletePublishedLib`, and then select `Server`.
The component is one of several components in the library and your Ruleset is unlocked	a. In your terminal, enter `npm run delete`, and then select the component to delete. b. Enter `npm run publish` to publish the library without the component.

©2025 Pegasystems Inc

Choices	Actions
The component is one of several components in the library and your Ruleset is locked	a. In Dev Studio, create a new ruleset version and unlock it. b. Create a new library version, and then copy the components into the library. c. In your terminal, enter `npm run delete`, and then select the component to delete. d. In your terminal, publish the library without the component to the new ruleset version by entering `npm run publish`. e. Ensure that your application can view the new ruleset version.

Important considerations

Learn about important limitations when working with components in library mode.

- Do not change the version in the `config.json` file. Create a new library version by using the `createLibVersion` command and move the components from the previous library to it.
- Do not change the version or library name in the `tasks.config.json` file after creating an initial library. Change the version or library name using the `createLib`, `createLibVersion`, `switchLib`, `switchLibVersion`, `deleteLib`, and `deleteLibVersion` commands.
- Do not change the organization in the `package.json` file after creating libraries. Instead, initialize a new project with a new organization. For more information, see Initializing a project.

©2025 Pegasystems Inc

Import components into the new library using the `importComponent` command. This operation updates the organization and component namespaces.

- Do not manually rename components. Use the `rename` command, as this ensures that the `config.json` file, components, and classes are all updated.
- You can publish only one version of a library at a time. To publish other versions, you must switch between versions and publish each version separately. When you switch versions, the components and the `config.json` file are updated with the new version number, which becomes the default version.
- The `store` folder contains the local archive of all libraries and its versions. It cannot be recovered if it is deleted. To create a backup of this folder, you can export each library version using the `exportLibVersion` command, or copy the `store` folder to another location, or use Git to store the project.

©2025 Pegasystems Inc

Developing a standalone Constellation DX component

Learn to create and use a standalone Constellation DX component in your application. Understanding all steps of the process helps you successfully build and deploy standalone Constellation DX components to adapt and extend the UI of your application.

Starting from DXCB 25.1.10, Pega recommends developing library mode components as it has several advantages over standalone components. For more information, see Advantages of library mode.

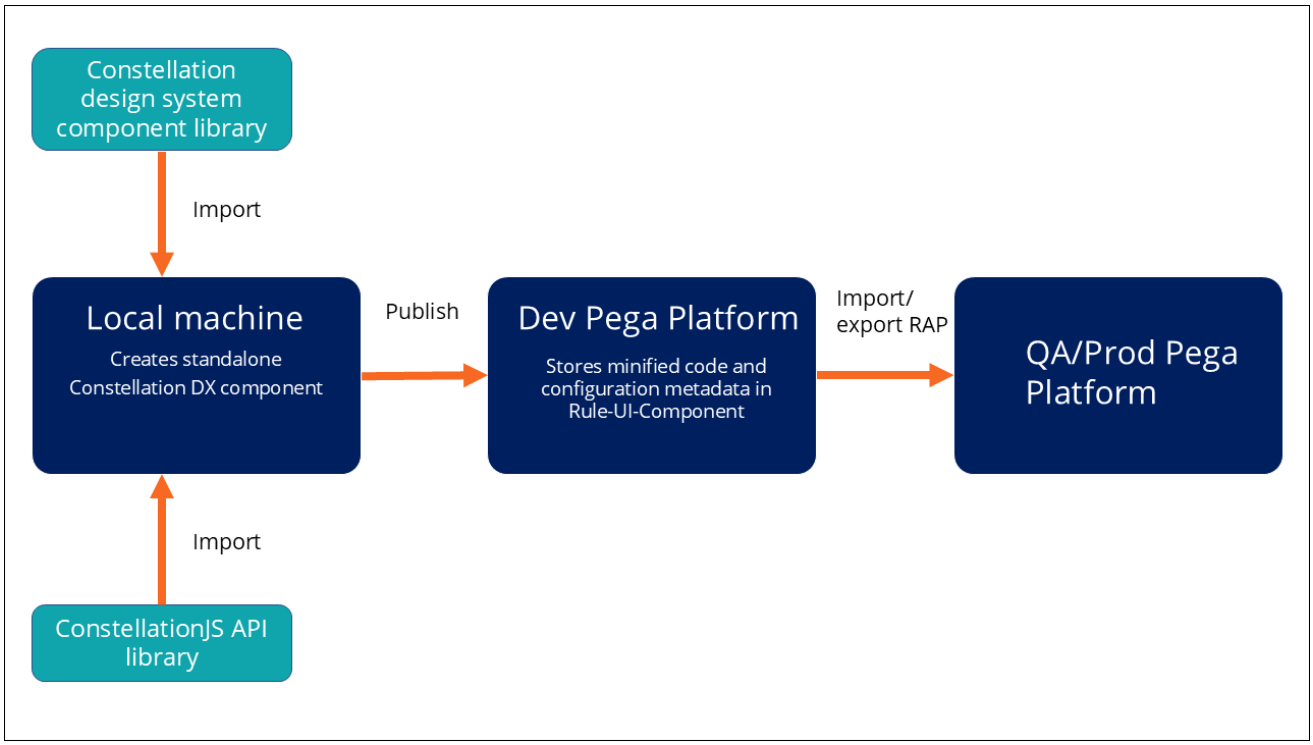
If you have existing standalone components, you can work with them in library mode by importing them into a library. For more information, see Importing and publishing standalone components.

If you do not want to use library mode and want to use only standalone components for specific use cases, you can still do that as DXCB 25.1.10 supports developing and working with standalone components.

The three stages in developing a standalone Constellation DX component are:

- Create the component locally.
- Publish the component.
- Deploy the component.

PEGA ©2025 Pegasystems Inc



Developing a standalone Constellation DX component

Create the standalone component locally

Build a standalone component locally by using web development tools, such as npm, storybook, and writing JavaScript inside a code editor. Create the code of the standalone component using the Constellation design system component library and *ConstellationJS* API library.

Import these libraries into your application to use the styling and theming of the Constellation design system component library and the data from the *ConstellationJS* API library for creating your standalone component.

After you test the standalone component locally, you can publish it to the Pega developer platform.

©2025 Pegasystems Inc

Publish the standalone component

Publish the created standalone component to a ruleset version in the Pega developer platform.

A *Rule-UI-Component* contains the minified code and the configuration metadata of the standalone component. The configuration metadata stored in a component definition file is used to render the parameters of the standalone component such as a constant, property, or picklist.

Deploy the standalone component

Use the published standalone component in your application by using App Studio.

Import or export the standalone component as a RAP file by using the Deployment Manager to a Pega QA or production platform.

For detailed information on each stage of developing a standalone Constellation DX component, see the following topics:

- **Disabling library mode in DXCB 25.1.10**
- **Workflow of a standalone Constellation DX component**
- **Creating a standalone Constellation DX component**
- **Editing a standalone component**
- **Creating Storybook mocks for standalone Constellation DX components**
- **Previewing standalone Constellation DX components in Storybook**
- **Building standalone Constellation DX components**
- **Authenticating Pega Platform Server user access for standalone Constellation DX components**

PEGA ©2025 Pegasystems Inc

- **Publishing standalone Constellation DX components**
- **Configuring a standalone Constellation DX component for App Studio**
- **Testing a standalone Constellation DX component with Pega Platform**
- **Localizing a standalone Constellation DX component**
- **Advanced webpack configuration**
- **Commands used for standalone Constellation DX components**
- **Additional standalone Constellation DX component commands**

Disabling library mode in DXCB 25.1.10

To work with standalone components in DXCB 25.1.10, you must disable library mode.

Starting from DXCB 25.1.10, library mode is the default mode of working with Constellation DX components. If you are using DXCB 25.1.10 and want to use only standalone components, disable library mode.

> **NOTE:** This section does not apply if you are using a version of DXCB that is older than 25.1.10.

1. Install a new version of Constellation DX Component Builder.
2. Open the `tasks.config.json` file and set the `libraryModeCL` attribute to `false`.
3. Copy all the components to the `src/components` folder.
4. Move the code in the `shared` folder to the folder of each component that uses them.
   You must update the component code to import the moved code from the component, and not from the `shared` folder.

©2025 Pegasystems Inc

5. Run the `buildAllComponents` command to verify that the components compile correctly.

Result:

This disables all the commands that are specific to library mode. Any components that you now publish can be viewed along with the components published in libraries, in library mode.

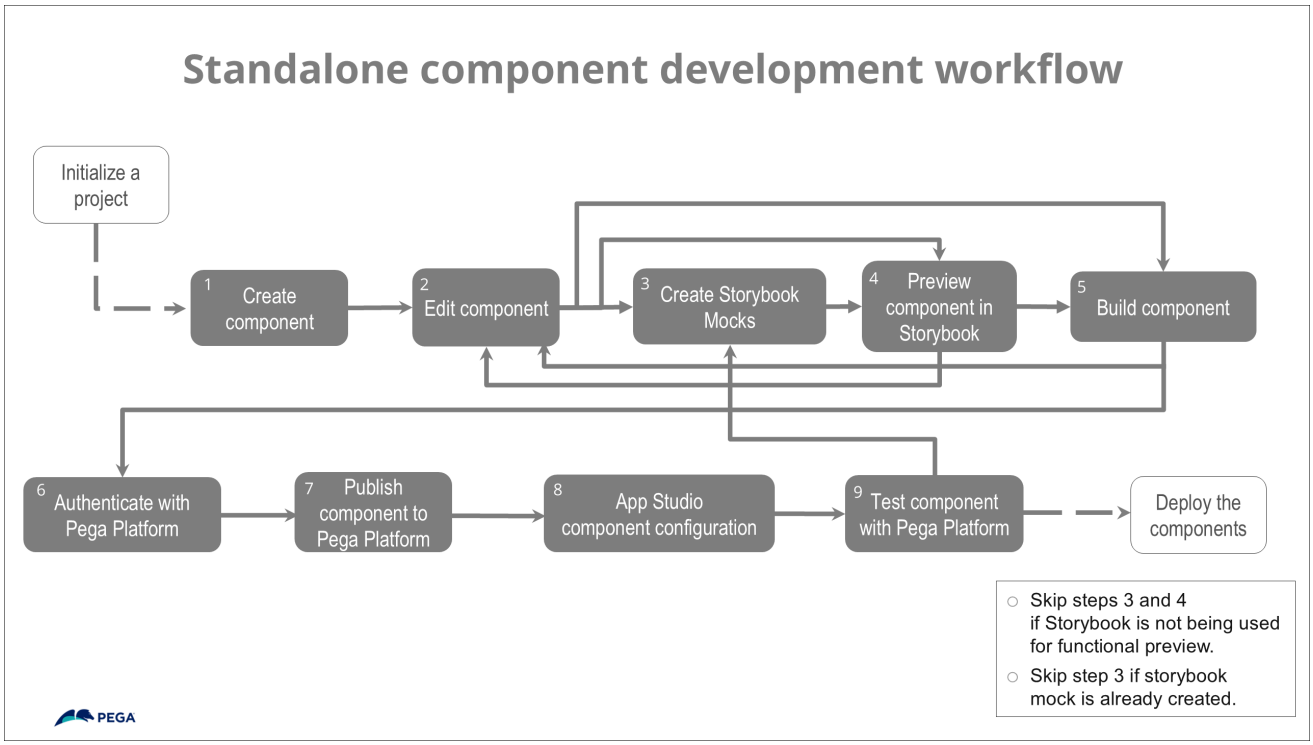
What to do next:

- Use separate projects for component libraries created in library mode and for standalone components created after disabling library mode. This is because all component code is located in the `src/components` folder, and the components in libraries created in library mode have their `config.json` file configured differently, compared to standalone components created after disabling library mode.
- After disabling library mode, if your Ruleset contains a component library, you cannot publish standalone components with the same `org_lib` prefix.

Workflow of a standalone Constellation DX component

The developer workflow illustrates the steps involved in creating, publishing, and deploying a standalone Constellation DX component.

©2025 Pegasystems Inc



Developer workflow of a standalone Constellation *DX component*

Creating a standalone Constellation DX component

Use the `create` command to create a standalone Constellation DX component.

Before you begin:

Ensure that you have initialized your project before creating a standalone Constellation DX component. For more information, see Initializing a project.

NOTE:

- For information about the types of Constellation DX components that can be created by the Constellation DX Component Builder (DXCB), see Types of Constellation DX components.

©2025 Pegasystems Inc

> - DXCB does not support all App Studio components. For more information on the list of supported components, see Component examples provided by Constellation DX Component Builder.

1. Open your preferred terminal tool.
2. Go to your Constellation DX Component Builder project folder.
3. Enter `npm run create`.

   When creating a component, choose the component type and sub-type that is most similar to the behavior of the desired final component. For more information, see Choosing the type of a Constellation DX component.

   Locate each DX component type in App Studio as mentioned below:
   - To view the Details template for a case type, click the **Case types** pane, select an existing case type, click the UX tab, click the Other Views tab, navigate to the **Partial** section, and click Details. In the **Edit View: Details** panel, click Add to add new Page widgets and click Edit to change the Details template.
   - To add CASE or PAGE & CASE widgets, click the **Case types** pane, select an existing case type, click the UX tab, click the Full case page tab, navigate to the **Utilities** section, and click Add.
   - To view Fields and the FORM template, click the **Case types** pane, select an existing case type, select a step, and click Configure view. In the **Edit view: <view name>** panel, navigate to the **Fields** section, click Add to add new Fields and click Edit to change the FORM template.
   - To view Page widgets and the PAGE layout template, click the **Channels** pane, select an existing channel interface, click Landing pages, and click Home. In the **Edit landing page: Home** panel, click Add to add new Page widgets and click Edit to change the PAGE layout template.

©2025 Pegasystems Inc

For information on each type of Constellation DX component, see Types of Constellation DX components.

4. Enter the details of the component:
   a. Enter the type of the component.

      For example: `Field`
   b. Enter the sub-type of the component.

      > **NOTE:** Component authoring teams can use sub-type metadata. The template code does not use the metadata. You can change it at any time in the component definition file.

      For example: `Text`

   c. Enter the name of the component.

      For example: `customcomp0`
   d. Enter the component label.

      For example: `custom component`
   e. Enter the version number of the component.

      For example: `0.0.1`

      > **NOTE:** The version number of the component has no impact during run time or design time.

   f. Enter the name of the library.

      The name of the library should match the name of an existing company library.

      For example: `MyLib`

©2025 Pegasystems Inc

g. **NOTE:** This step is applicable only if you are using Pega Platform™ '24.2 or earlier versions. From Pega Platform '25 onwards, for a component to be available to an application, the ruleset version to which the component is published must be included in the application.

Provide a list of supported applications by entering the names of the supported applications, with each name separated by a comma.
For example: `uPlusApp, ABCInsuranceClaims, uBankingTool`

NOTE: If you do not provide any supported applications, the component will be available to all applications by default.

h. Describe the component.
For example:

This is a customized searchbox that filters data.

```
                \testfolder\test> npm run create

> test@1.0.0 create
> custom-dx-components create

DX Component Builder v8.8.1
? Enter type of component Field
? Enter subtype of component Text
? Enter component name (required) TextComp
? Enter component label for display (required) TextComp
? Enter component version 0.0.1
? Enter library name (required) MyLib
? Please enter the application names to be supported (comma-separated). Keep empty for all applications
? Enter description for the component Text component
Pega_MyLib_TextComp component is created in              \testfolder\test\src\components\Pega_MyLib_TextComp
```

Command line interface - create

©2025 Pegasystems Inc

Result:

The standalone component is created in the following path:

```
..../<project name>/src/components/
<Org_Lib_ComponentName>
```

NOTE:

- The name of the component now uses the `<Org_Lib_ComponentName>` format.
- Starting from DXCB 25.1.10, the length of the component's name is limited to a maximum number of characters. The maximum number of characters varies depending on the string lengths of the organization, library, version, and other constants, and is different for standalone and library mode components.
- Do not manually change the name of the `<Org_Lib_ComponentName>` folder.
- Do not manually change the names of the classes or artifacts in the generated templated code in the `<Org_Lib_ComponentName>` folder.
- To change the name of the component, use the `rename` command. For more information, see Renaming Constellation DX components.

A `src/components` folder is created along with the necessary files required to build the component. Apart from the core files shown below that are initialized for every component, some components also generate supporting files that contain utility libraries.

```
└── src
    └── components
        └── <Org_Lib_ComponentName>
```

©2025 Pegasystems Inc

```
                    ├── index.tsx
                    ├── demo.stories.tsx
                    ├── config.json
                    ├── styles.ts
                    ├── create-nonce.ts
                    ├── PConnProps.d.ts
```

The `index.tsx` file contains the actual component rendering logic.

The `demo.stories.tsx` contains the demo that can be viewed in the storybook.

The `config.json` component definition file contains the mapping of properties for configuring the auto-generated property pane. This property pane is displayed in App Studio when an application is being authored to use the Constellation DX component.

The `create-nonce.ts` file contains the webpack nonce for security. For more information, see https://webpack.js.org/guides/csp/.

The `PConnProps.d.ts` file contains the TypeScript definitions for PCore and PConnect APIs.

The properties in the context of Constellation DX Components are definitions of component parameters used during design and at run time. Design settings are constant. Run-time parameters that contain field references related to the Pega data model are resolved from the actual data at run time.

5. Run the following command to start the Storybook to view the demo:
   `npm run startStorybook`
6. Modify the source files to build your own component.

©2025 Pegasystems Inc

You create Constellation DX components by using generally available tools. You need advanced knowledge to create reliable Constellation DX components using the Constellation design system. For more information, see the following resources:

- Configuring Constellation
- Constellation Design System
- Using PCore and PConnect Public APIs

You can also install and import third-party libraries if required for your component.

> **NOTE:** You can use the `createAll` command to create all supported Constellation DX components.
>
> The purpose of creating all supported Constellation DX components is only to validate that the Constellation DX Component Builder produces the templates as expected. After using the `createAll` command, delete the components that you do not need. As a best practice, always use the `create` command to create the components you need in order to minimize library size and reduce ongoing library maintenance cost.

Creating Storybook mocks for standalone Constellation DX components

Provide mock data that is representative of the data that your standalone component will display or manage at run time.

If you want to use the storybook to preview your standalone component, you must create a PCore and PConnect mock function along with any required data for every function call made from the main component node.

PEGA ©2025 Pegasystems Inc

When you create your extension DX component using Constellation DX Component Builder, the mock data relevant to the sample type and sub type will be available for modification in the `mock.ts` or `mock.tsx` file.

The process for updating this mock data for your own component use case depends upon the purpose of your component and the data it needs to function. To develop your component in the storybook, you must provide mock data that is representative of the data that will be provided to your component at run time through the PCore and PConnect APIs.

The following example is of a `mock.tsx` file used to provide mock data to the PCore.getDataPageUtils().getDataAsync() API.

```
import type { DataAsyncResponse } from '@pega/pcore-pconnect-typedefs/data-view
/types';

export const newRating = {
  CustomerRating: 0,
  NumberOfStars: 5,
  CaseID: 'SL-TELLUSMORE-WORK Z-12345',
  CaseClassName: 'SL-TellUsMore-Work-Incident',
  CustomerID: 'Q1234',
  pxUpdateDateTime: '2020-06-29T11:06:24.329Z'
};

const ratingData: Omit<DataAsyncResponse, 'config'> = {
  data: [
    {
      CustomerRating: 3,
      NumberOfStars: 5,
      CaseID: 'SL-TELLUSMORE-WORK Z-1234',
      CaseClassName: 'SL-TellUsMore-Work-Incident',
      CustomerID: 'Q1234',
```

©2025 Pegasystems Inc

```
      pyGUID: '6876786876876868123',
      pxUpdateDateTime: '2020-06-29T11:06:24.329Z'
    },
    {
      CustomerRating: 3,
      NumberOfStars: 5,
      CaseID: 'SL-TELLUSMORE-WORK Z-123',
      CaseClassName: 'SL-TellUsMore-Work-Incident',
      CustomerID: 'Q1234',
      pyGUID: '68767868768768681',
      pxUpdateDateTime: '2020-06-29T11:06:24.329Z'
    }
  ],
  status: 200
};

export default ratingData;
```

This example mock data above is used in the `demo.stories.tsx` file as shown below to provide a mock of the PCore.getDataPageUtils().getDataAsync() function.

```
import mockRatingData from './mock';
import type DataPageUtils from '@pega/pcore-pconnect-typedefs/datapage/index';
import type { DataAsyncResponse } from '@pega/pcore-pconnect-typedefs/data-view
/types';



const mockGetDataAsync = (
  ...args: any[]
): Promise<Partial<DataAsyncResponse>> => {
  const filter = args[4]?.filter as Filter;
  const queryCustomerID = filter?.filterConditions.F1.rhs.value;
  let { data } = mockRatingData;
```

©2025 Pegasystems Inc

```
  if (queryCustomerID?.length)
    data = data.filter(rating => rating.CustomerID === queryCustomerID);

  return Promise.resolve({ data, status: 200 });
};

const mockDataPageUtils = (): Partial<typeof DataPageUtils> => {
  return {
    getDataAsync: mockGetDataAsync as () => Promise<DataAsyncResponse>
  };
};


...
```

Previewing standalone Constellation DX components in Storybook

View how the existing standalone Constellation DX components are displayed at run time by opening the components in Storybook. Open the project and then start Storybook in development mode so that you can view each standalone component separately.

> **NOTE:** If the standalone components have calls to Constellation orchestration layer APIs, disable the calls or create an API mock to view these components in Storybook.

1. Open your preferred terminal tool.
2. Enter `cd <name of the project>`.
3. Enter `npm run startStorybook`.

©2025 Pegasystems Inc

> **NOTE:** To test the different behaviors of your component, go to the `index.tsx` file and change the default values in your <component name>. `defaultProps` while the Storybook is running.

> **Result:**
>
> You can now view each existing standalone Constellation DX component in Storybook. To close Storybook, use `ctrl + c` and then close the browser tab.
>
> For more information on Storybook, see https://storybook.js.org/

Building standalone Constellation DX components

Create a local build of a standalone component, or all standalone components in a project, to validate component configuration, and to check for coding errors.

Use the following commands to build standalone components:

```
npm run buildComponent (builds a specific standalone component)
```

```
npm run buildAllComponents (builds all standalone components in the source folder )
```

The build command validates the component schema, performs linting, and bundles the component. These actions are also performed when publishing a component; however, build the component locally to ensure that it compiles without errors before publishing it to the server.

> **NOTE:**

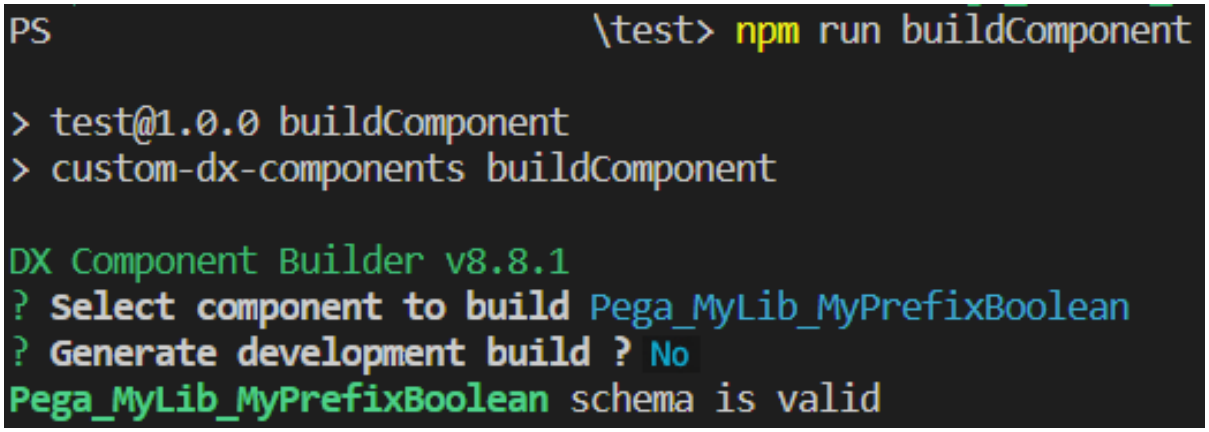
©2025 Pegasystems Inc

- If changes made to the `config.json` file are not validated, the validation of the component schema could fail during the building and publishing process. To prevent this, ensure that you validate any changes made to the `config.json` file. For detailed information on the structure and attributes of the `config.json` file, see Component definition.
- You might see warnings when linting is being performed. This will not prevent the process of publishing a component.

1. Open a terminal at the DX component directory and run the following command:

   ```
   npm run buildComponent
   ```

2. Select the standalone component to be built.
3. Specify whether to generate a development build.
   A development build should be generated only during component development as it enables run-time debugging in the browser. Do not select a development build for deploying to production as the development build does not compress and obfuscate the component code .



Command line interface - buildComponent

©2025 Pegasystems Inc

Result:

The Constellation DX Component Builder compiles the standalone component and creates a local build.

> **NOTE:** Building all standalone components (`npm run buildAllComponents`) automatically executes the `npm run buildComponent` command for all standalone components in your source folder. You can find the bundled standalone components in the `dist` directory of your project's root folder.

Authenticating Pega Platform Server user access for standalone Constellation DX components

To publish, list, or delete a standalone Constellation DX component, you require credentials to access the resources on the Pega Platform™ Server.

Before you begin:

- Ensure that the `Allow rule check out` access setting under the **Security** tab has been added to the operator record used for authentication and publication.
- Ensure you have a valid operator ID in your Pega Platform development instance that you can use to log in to your Pega applications.
- Ensure you are using Pega Platform 8.7 or a later version.

> **NOTE:**

©2025 Pegasystems Inc

- After completing the authentication, you must reauthenticate yourself after the default timeout of 15 minutes. You can change the timeout period by editing the *REST* API Auth 2.0 Client registration record. The record contains a `clientID` that matches the configuration contained in the `tasks.config.json` file of each component.
- Constellation DX components require you to use the OAuth 2.0 authentication protocol.
- If you do not want to use the provided client credentials, you can create your own client credentials. If you use a different OAuth 2.0 authentication flow in the client credentials, you must update your `tasks.config.json` file accordingly. For more information on the various configuration options for the `tasks.config.json` file, see server-config. The relevant attributes in the `server-config` section are `clientId`, `clientSecret`, `grantType`, and `authService` (if you need to integrate Constellation DX Component Builder with your SSO setup). For more information on setting up alternative authentication schemes that can be used with Constellation DX Component Builder, see OAuth 2.0 client registrations.

1. In Dev Studio, click **Records** > **Security** > **OAuth 2.0 Client Registration**. If the **DX-ComponentDev** record is visible in the list of instances, you already have a valid client credential. Go to step 7 to revalidate it.
   If the **DX-ComponentDev** record does not exist, import it as directed in the following steps.
2. In Dev Studio, click **Configure** > **Application** > **Distribution** > **Import**.
3. On the **Application: Import wizard** tab, click **Local file**, and then click **Choose file**.
4. Navigate to the DX component project's `keys` folder, select the `DX-ComponentDev.zip` file, click **Open**, and click **Next**.
5. Select the **Enable advanced mode to provide more granular control over the import process** checkbox and click **Next**.

©2025 Pegasystems Inc

6. On the **Import Wizard File : DX-ComponentDev.zip** tab, verify that the DX-COMPONENTDEV instance is selected, and click **Next**.
   The import process is complete. The **DX-ComponentDev** record is now present in the list of instances.
7. Run the following command to initiate authentication:
   `npm run authenticate`

```
\mycomycomponents>npm run authenticate

> mycomycomponents@1.0.0 authenticate
> custom-dx-components authenticate

? Enter pega server url
? Enter user id admin@mediaco
? Enter password [hidden]
? Enter clientId 12782580023870311851
? Enter clientSecret [hidden]
Authenticated successfully !!
```

Authenticating user access

8. Enter your operator ID and password, and click **Log in**.
   If the credentials are valid, an authentication success message is displayed.

Result:

You now have authentication to publish, list, or delete standalone Constellation DX components.

Publishing standalone Constellation DX components

Publish your standalone Constellation DX components to the Pega Platform™ Server so that the component is discoverable as you author in the component palette. As a result, you can use the components while building the UI of your application.

©2025 Pegasystems Inc

Before you begin:

- Ensure that you have authentication to access the resources in the Pega Platform Server to list, publish, or delete components. For more information, see Authenticating Pega Platform Server user access for standalone Constellation DX components.
- Ensure that your default access group is mapped to the application where you want to publish the component. If you do not use your default access group, use the full application URL alias.
- Ensure that you can access the Ruleset to which you want to publish your component.
- If you are using DXCB 25.1.10, ensure that you publish the component to a Ruleset that is not a Branch Ruleset. Publishing the component to a Branch Ruleset results in an error.

1. Open your preferred terminal tool.
2. Enter `npm run publish`.
3. Complete the parameter values:
   a. Select the component that you want to publish.
   b. For the `rulesetName`, enter the name of the ruleset.
   c. For the `rulesetVersion`, enter the version of the ruleset.

NOTE:

- Make sure you unlock the relevant ruleset version before you publish the component.
- If you are using a version earlier than Pega Platform '25, ensure that the component is added to the highest ruleset version of the application.

©2025 Pegasystems Inc

d. For the `devBuild`, enter `Y` to allow debugging or `n` to compress and obfuscate the code.

> **NOTE:** Setting `devBuild` to `Y` allows you to use your browser's developer tools to set breakpoints in your component code. `devBuild` also generates sourceMaps for this purpose. During development and debugging of your component, set `devBuild` to `Y` to debug and troubleshoot your code. When you are ready to publish your component for deployment to another environment, set `devBuild` to `N` as it will decrease the size of your bundled component.

```
> mycomycomponents@1.0.0 publish
> custom-dx-components publish

? Select component to publish Pega_MyComponents_MyLocalActionLauncher
? Enter ruleset name MyCompon
? Enter ruleset version 01-01-01
? Generate development build ? No
Pega_MyComponents_MyLocalActionLauncher schema is valid
  √ Validate config schema
  √ Bundle Component
  √ Validate config schema
  √ Bundle Component
  √ Zip Component
  √ Publish Component
Success : Component successfully created in ruleset MyCompon:01-01-01
```

Publishing the component

> **NOTE:** You can also preconfigure the values in a `tasks.config.json` file.

©2025 Pegasystems Inc

For example:

```
{
  "components-server-config": {
    "server": "http://127.0.0.1:8080/prweb",
    "user": "customcomponentoperator",
    "password": "pwd"
    "rulesetName": "TestPega-C11nGall",
    "rulesetVersion": "01-01-03",
    "sourceOfComponents": "Server",
    "devBuild": false,
  }
}
```

4. **Optional:** To list your Constellation DX components and ensure that your component is displayed in the list, run the `npm run list` command.

Result:

The component publication is complete and application developers can now use the component in the authoring view of App Studio. The component uses the following naming convention:

`<organization>_<library>_<component name>`

NOTE:

- Starting from Pega Platform '25, after you publish the component, Pega Platform creates the corresponding *Rule-UI-Component* and *Rule-File-Text* Rules. For more information, see Rule resolution for standalone components.

©2025 Pegasystems Inc

- After publishing a standalone component, if it is not visible for selection in App Studio, please refresh the browser.
- In some cases, the preview of the standalone component might not appear in App Studio. However, you can still configure the standalone component in App Studio.
- Publishing all standalone components (`npm run publishAll`) follows the same procedure, except for selecting individual components. The Constellation DX Component Builder automatically validates and publishes all standalone components sequentially in the selected component category.

  Using arguments with the `publishAll` command is not recommended. You are responsible for troubleshooting any issues that may arise as a result of using arguments.

- If you are using Pega Platform '24.2 or earlier versions, when you view a published standalone component in App Studio or any Constellation portal, you will be viewing the latest published version of the component for that environment regardless of rule resolution.
- Starting from Pega Platform '25, when you view a published standalone component in App Studio or any Constellation portal, you will be viewing the latest published version of the component for that environment based on rule resolution. With rule resolution, if you publish the component to a lower ruleset version, you may not be viewing the latest published version of the component.

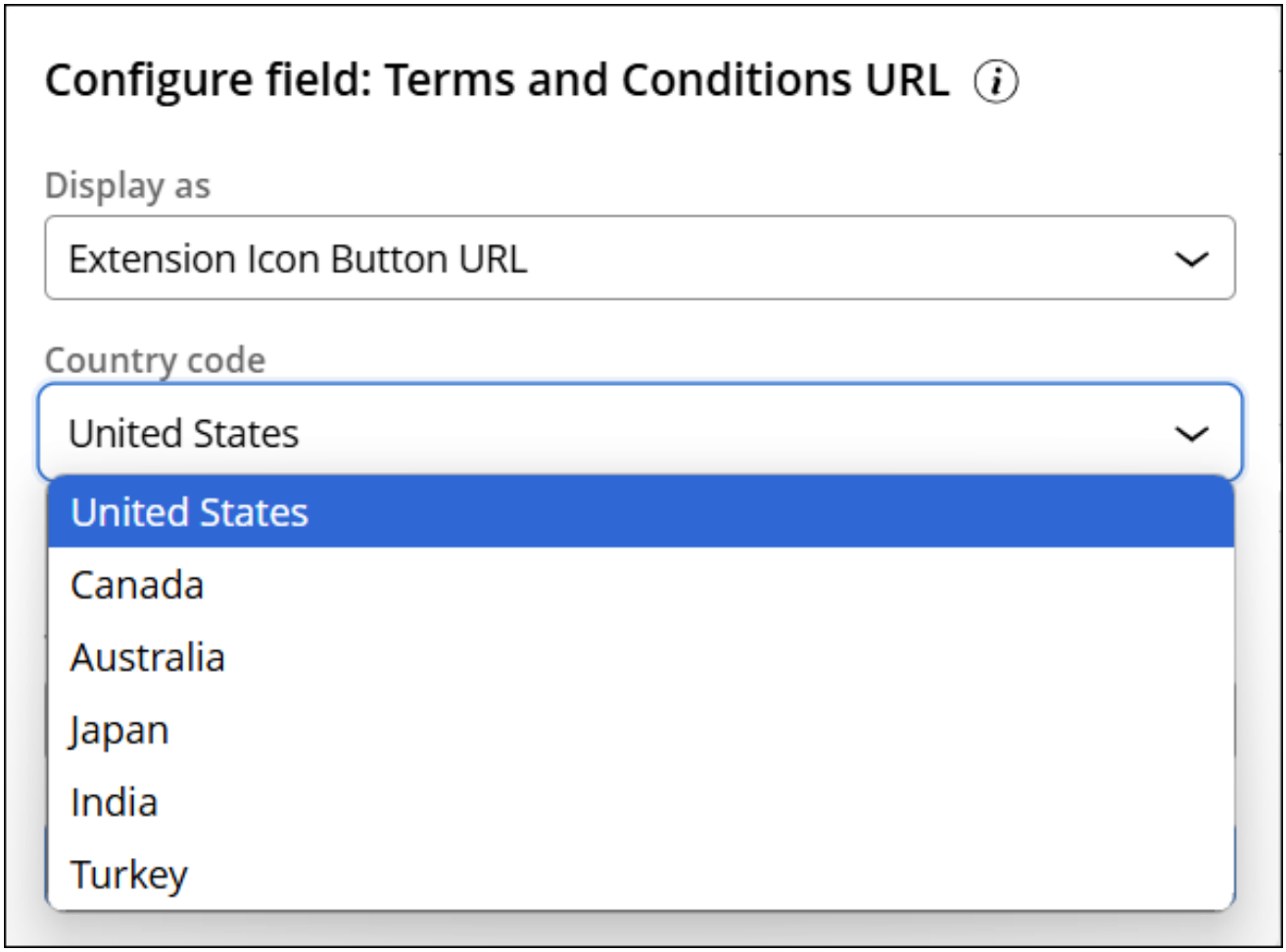
Configuring a standalone Constellation DX component for App Studio

Test the component definition and configure your standalone component.

©2025 Pegasystems Inc

After publishing your component to App Studio, to test your component definition or `config.json` file you must add your standalone component to a View. You can test if the component definition is correct for the required authoring context, such as, a form layout template. If you need to modify your component definition, update the component definition for your component in your local development environment and republish your component.

The following figure shows an example of the UI authoring property panel for an extension DX component.



Example of UI authoring property panel for an extension DX component

©2025 Pegasystems Inc

Testing a standalone Constellation DX component with Pega Platform

Perform an integration test on your component definition and component code using App Studio in your Pega Platform™ development environment.

Your standalone component can be tested using two execution modes:

- **UI Authoring Preview**

  When your component is added to a view, the UI Authoring Preview in App Studio will render your component using a standard set of mock data. This data is unlikely to be relevant for your specific component, so it is important to check that the data being passed to your component in UI Authoring mode is in the format you expect. If the data is not in the expected format, you must not render your component with that data.

- **Run-time Portal**

  After your component has been configured in UI Authoring, you will preview your component in App Studio or a standalone portal. The props passed to your component will now be the actual data.

> **NOTE:** During component run-time, you can determine the execution mode by checking if the name of the current PConnect context contains the term "authoring".

For example, the following code detects if your component is in authoring mode.

```
const [inAuthoringMode, setInAuthoringMode] = useState(false);
if (getPConnect().getContextName()?.includes('authoring')) {
  setInAuthoringMode(true);
}
```

©2025 Pegasystems Inc

```
if(inAuthoringMode){
  ...
}
```

Localizing a standalone Constellation DX component

Provide multi-language support for your standalone Constellation DX component by integrating it with out-of-the-box Constellation localization features.

To ensure that your standalone Constellation DX components are properly localized,

- integrate your component with the appropriate localization APIs
- externalize your static string data into a component-specific `localizations.json` file

Your `localizations.json` file must be added to your component's root folder to ensure that it is packaged with your component when the component is published.

The following code contains an example of a `localizations.json` file:

```
{
  "fields": {
    "Edit": "Edit",
    "Navigate to step": "Navigate to step"
  }
}
```

For more information about localizing your Constellation application, see Internationalization and localization.

©2025 Pegasystems Inc

Advanced webpack configuration

Learn about advanced topics that require modifying the out-of-the-box behavior of Constellation DX Component Builder.

promoteWebpack

When you are building an advanced web component, such as a component based on third party libraries, the bundling process shipped with the Constellation DX component builder may not provide sufficient control for your component to be packaged correctly. This can be resolved by running the `promoteWebpack` command. For more information, see promoteWebpack.

Running this command creates a copy of the internal `webpack.config.js` file in the root folder of your project and configures the Constellation DX Component Builder to use the copied `webpack.config.js` file in the bundling stage of the publishing process. You can now bundle your component as desired.

Restrictions during bundling

Adhere to the following recommendations when you modify the bundling process:

- Modify the plugins and loaders only if required.
- Do not use `webpack-cli` or `npx webpack` to invoke the bundling process. Use the provided commands, such as `publish`, `publishAll`, `buildComponent`, or `buildComponentAll`.
- Do not modify the output section of the `webpack.config.js` file, since it is important for the Zip and Publish Component tasks.
- Do not modify the usage of any of the input parameters, such as `inputFile`, `outputFile`, `componentKey`, `externals`, `globals`, `sourceMap`, and `devBuild` as they are preset by Constellation DX component builder.
- Do not modify the maxChunks and minChunksize.

©2025 Pegasystems Inc

> **NOTE:** If you modify the internal `webpack.config.js` file, you are responsible for troubleshooting any build issues and integrating your changes into new versions of Constellation DX Component Builder.

Example

This example shows `ts-loader` being replaced with `babel-loader` for transpiling.

> **NOTE:** You must have a strong understanding of webpack configuration to attempt any changes.

```
{
  test: /\.(js|mjs|jsx|ts|tsx)$/,
  exclude: {
    and: [/node_modules/]
  },
  loader: "babel-loader",
  options: {
    presets: [
      [
        "@babel/preset-env",
        {
          targets: {
            node: "current",
          },
        },
      ],
      [
        "@babel/preset-react",
        {
```

©2025 Pegasystems Inc

```
            runtime: "automatic",
          },
        ],
        '@babel/preset-typescript'
      ],
      env: {
        production: {
          plugins: [
            ["babel-plugin-styled-components"],
            [
              "transform-react-remove-prop-types",
              {
                mode: "remove",
                removeImport: true,
              },
            ],
            ["@babel/plugin-syntax-top-level-await"],
          ],
        },
        development: {
          plugins: [
            ["babel-plugin-styled-components"],
            ["@babel/plugin-syntax-top-level-await"],
          ],
        },
      },
    },
  },

// Process application JS with TS-Loader.
// The preset includes JSX, Flow, TypeScript, and some ESnext features.
// {
```

©2025 Pegasystems Inc

```
//   test: /\.(js|mjs|jsx)$/,
//   include: appSrc,
//   loader: 'ts-loader',
//   options: {
//     configFile: 'tsconfig.build.json'
//   },
//   exclude: /node_modules/
// },
//
// {
//   test: /\.(ts|tsx)$/,
//   loader: 'ts-loader',
//   options: {
//     configFile: 'tsconfig.build.json'
//   },
//   exclude: /node_modules/
// },
```

For more configuration options, see the documentation for Babel. Some advanced options may conflict with the bundling process, so ensure that you test any changes to make sure that they still work with component publishing.

revertWebPack

If you want to revert to using the original internal `webpack.config.js` file, run the `revertWebPack` command. For more information, see revertWebPack.

Running this command will rename the existing `webpack.config.js` file in the root folder of your project to `webpack.config.js.notused`. Constellation DX component builder will then use the internal webpack configuration for bundling.

If you have already run `npm run promoteWebPack` and want to demote the `package.json` file, you must demote the `webpack.config.js` file. To do this, call `npm`

©2025 Pegasystems Inc

`run revertWebPack` and then call `npm run promoteWebPack`. This will create a `webpack.config.js` file that uses require statements and removes `"type":"module"` from the `package.json` file. If you want to update back to `"type":"module"`, call the above sequence of commands again and a `webpack.config.js` file will be created that uses import statements and adds `"type":"module"` to the `package.json` file.

Extension points

You can use extension points to drive different behaviour in the internal bundling process.

You can add additional externals or globals in the `build.config.js` file located in the root folder of your project.

```
{
  globals: {
    myGlobal: 'MyGlobal'
  },
  externals: {
    myExternal: 'MyExternal'
  }
}
```

These will be merged into the globals and externals objects that are preset by Constellation DX Component Builder during bundling. This can also be used in conjunction with a promoted `webpack.config.js` file. Changes made here will be applicable to all components.

PEGA ©2025 Pegasystems Inc

Commands used for standalone Constellation DX components

Learn about the various commands for standalone Constellation DX components that can be executed through the command line interface of the Constellation DX Component Builder.

Executing commands

You can execute commands provided by the Constellation DX Component Builder in the `npm run <enter command name>` format.

List of executable commands

Name	Description
`authenticate`	Authenticates the Pega Platform™ Server, user credentials, and OAuth 2.0 credentials based on the configuration of the `tasks.config.json` file. For more information, see Authenticating Pega Platform Server user access for standalone Constellation DX components.
`buildAllComponents`	Generates a build of all Constellation DX components in your local library. You can also optionally generate a development build for all components using this command. For more information, see Building standalone Constellation DX components.
`buildComponent`	Generates a build of a specified Constellation DX component in your local library. You can also optionally generate a development build for the specified component using this command.

©2025 Pegasystems Inc

Name	Description
	For more information, see Building standalone Constellation DX components.
`create`	Creates a Constellation DX component based on your inputs. For more information, see Creating a standalone Constellation DX component.
`createAll`	Creates all supported Constellation DX components. For more information, see this note on using the `createAll` command.
`delete`	Deletes the specified Constellation DX component from the server or local library based on your inputs. For more information, see Deleting Constellation DX components.
`deleteAll`	Deletes all Constellation DX components from the server or local library based on your inputs. For more information, see Deleting Constellation DX components.
`help`	Displays the list of commands that can be performed related to Constellation DX components. For more information, see Listing commands.
`importComponent`	Imports an existing component into your components.

PEGA ©2025 Pegasystems Inc

Name	Description
	For more information, see Importing a Constellation DX component.
`list`	Lists the Constellation DX components in your local machine or server based on your inputs. For more information, see Listing Constellation DX components.
`promoteWebPack`	Creates a copy of the internal `webpack.config.js` file that is used for building or publishing, promotes it to the top level of your project folder, and then redirects the building or publishing process to use the copied file. **NOTE:** Depending on the type of the `package.json` module, the version of the `webpack.config.js` file will either be an ES6 module using import statements or an older version using require statements. For more information, see Advanced webpack configuration.
`publish`	Publishes the specified Constellation DX component to Pega Platform. For more information, see Publishing standalone Constellation DX components.
`publishAll`	Publishes all local components sequentially to Pega Platform. Additionally, this command overwrites or updates components published previously. Ensure that the details of the Pega

©2025 Pegasystems Inc

Name	Description
	Platform Server are configured in the `tasks.config.json` file before running this command. For more information, see Publishing standalone Constellation DX components.
`rename`	Renames the selected Constellation DX component, its directory, and code references. For more information, see Renaming Constellation DX components.
revertWebPack	Marks the existing `webpack.config.js` file in the top level of your project folder as `webpack.config.js.notused` and redirects the building or publishing process to use the original internal `webpack.config.js` file. For more information, see Advanced webpack configuration.
`startStorybook`	Starts a local instance of Storybook in the web browser. Use this command to verify Storybook installation or preview component mocks. For more information, see Previewing standalone Constellation DX components in Storybook.

Additional standalone Constellation DX component commands

Perform additional actions on standalone Constellation DX components using these commands.

©2025 Pegasystems Inc

Apart from the commands used as part of the workflow of a standalone Constellation DX component, you can run commands to perform the following additional actions:

- Rename a Constellation DX component. For more information, see Renaming Constellation DX components.
- View the available Constellation DX components. For more information, see Listing Constellation DX components.
- Delete Constellation DX components. For more information, see Deleting Constellation DX components.
- Display the list of available commands that can be performed on Constellation DX components. For more information, see Listing commands.
- Import a Constellation DX component. For more information, see Importing a Constellation DX component.

- **Renaming Constellation DX components**
- **Listing Constellation DX components**
- **Deleting Constellation DX components**
- **Listing commands**
- **Importing a Constellation DX component**

Renaming Constellation DX components

Use the `rename` command to rename Constellation DX components in the local library.

The `rename` command automatically updates the component directory, storybook, and the component definition file (`config.json`).

> **NOTE:**

©2025 Pegasystems Inc

- The `rename` command does not impact the published version of the Constellation DX component. If you need to rename the component after it is published, use the rename command as mentioned in the procedure below, republish the component, replace the component in all views using the component, and then delete the original component from the server.

1. Open your preferred terminal tool.
2. Enter `npm run rename`.
3. Select the component that you want to rename.
4. Enter the new name of the component.
5. Enter the new label of the component.
6. Enter the new organization of the component.
7. Enter the new version of the component.
8. Enter the new name of the library.
9. Provide a list of supported applications by entering the names of the supported applications, with each name separated by a comma.

NOTE: If you do not provide any supported applications, the component will be available to all applications by default.

10. Provide a new description of the component.

```
PS                \testfolder\test> npm run rename

> test@1.0.0 rename
> custom-dx-components rename

DX Component Builder v8.8.1
? Select component to rename Pega_MyLib_MyPrefixURL
? Enter new component name (required, no spaces) RenamedComponent
? Enter new component label for display (required) NewLabel
? Enter new component organization Pega
? Enter new component version 0.0.1
? Enter new library name (required) NewLib
? Please enter the application names to be supported (comma-separated). Keep empty for all applications
? Enter new description for the component (defaults to new label) Sample description
Component Pega_MyLib_MyPrefixURL renamed to Pega_NewLib_RenamedComponent
```

©2025 Pegasystems Inc

Command line interface - rename

Result:

The component, config, and styles are renamed along with the code references as per the specified details. Revalidate that the code works and make the necessary adjustments.

Listing Constellation DX components

View the available components on your local machine or the Pega Platform™ Server. You can list the components to view what is already available to the authenticated operator and to verify if the publish or delete process completed successfully.

Before you begin:

Ensure that you have authentication to access the resources in the Pega Platform Server to list, publish, or delete components. For more information, see Authenticating Pega Platform Server user access for standalone Constellation DX components.

1. Open your preferred terminal tool.
2. Enter `npm run list`.
3. Select the location of the components to be listed:
   - `Server` for components that are already published.
   - `Local` for components in your local project.

©2025 Pegasystems Inc

```
\mycomycomponents>npm run list

> mycomycomponents@1.0.0 list
> custom-dx-components list

? List components from Server or Local ?
  1) Server
  2) Local
```

Options for listing Constellation DX components

Result:

The Constellation DX components are displayed in the terminal.

```
? List components from Server or Local ? Local
List of custom components in Local
```

(index)	Rule name
0	'Pega_MyComponents_MyLocalActionLauncher'
1	'Pega_MyComponents_MyPageWidget'

List of Constellation DX components

Deleting Constellation DX components

Free up space and make navigation easier by deleting unused or obsolete Constellation DX components either locally or on the Pega Platform™ Server. You can delete both published and unpublished components.

Before you begin:

- Ensure that you have authentication to access the resources in the Pega Platform Server to list, publish, or delete components. For more

©2025 Pegasystems Inc

information, see Authenticating Pega Platform Server user access for standalone Constellation DX components.

- Ensure that the associated ruleset is open. If the associated ruleset is not open, the components in the ruleset will not be deleted and will be displayed in App Studio.

1. Open your preferred terminal tool.
2. Enter `npm run delete`.
3. Select the location from which you want to delete the components:
   - `Local` for components in your local project.
   - `Server` for components that are already published.

```
\mycomycomponents>npm run delete

> mycomycomponents@1.0.0 delete
> custom-dx-components delete

? Delete components from Server or Local ?
  1) Server
  2) Local
```

Options for deleting Constellation DX components

4. Choose the component you want to delete.
5. Enter `yes`.

Result:

Components deleted from the local machine are not deleted from the server. Components deleted from the server are removed from the ruleset and are no longer available in App Studio.

©2025 Pegasystems Inc

```
? Delete components from Server or Local ? Local
? Select component to delete Pega_MyComponents_MyLocalActionLauncher
? Are you sure ? Yes
Pega_MyComponents_MyLocalActionLauncher is deleted from Local
```

Deleted Constellation DX component

NOTE: Deleting all components (`npm run deleteAll`) follows the same procedure except that instead of selecting the components to be deleted, it deletes all components from the server or local machine.

Listing commands

Display the list of available commands that can be performed on Constellation DX components.

Introduced in Constellation DX Component Builder 24.2.10, use the `npm run help` command to display the list of commands that can be performed related to Constellation DX components.

NOTE: When in library mode, running this command will display the library mode commands. For more information, see Commands used in library mode.

Importing a Constellation DX component

The `importComponent` command is used to import an existing standalone component into your project. In library mode, the `importComponent` command is used to import an existing standalone component into a library.

©2025 Pegasystems Inc

Before you begin:

In a specific location, create a copy of the folder of the component that must be imported. You must enter this location when importing the component to the project.

Introduced in Constellation DX Component Builder 24.2.10, you use the `npm run importComponent` command to import an existing standalone component into your project. The org and library names of the imported component are automatically renamed if they do not match the corresponding details of your components. The value of the `version` property is updated in the `config.json` file to match the version of the current library.

1. In your terminal, navigate to your project and enter `npm run importComponent`.
2. Enter the location containing the folder of the component that must be imported.
3. Select the component to be imported.
   - If the imported component already exists in your project, choose whether you want to abort the import, overwrite the existing component, or rename the existing component.
   - If the imported component does not exist in your project, choose whether you want to validate the imported component.

Result:

The imported component is located in the `src/components` directory.

©2025 Pegasystems Inc

Component definition

The Constellation DX Component Builder generates a `config.json` file, or component definition file, when you create a component. This file drives the UI authoring experience when you configure the component for use in Pega Platform™, meaning that the property pane of the component in Pega Platform is defined by this file.

UI View authoring relies on the information in the component definition to provide a streamlined experience when you edit the View metadata that is consumed at run time.

The component definition consists of header attributes and an array of property definitions. You can modify these attributes and add other attributes when you design your component.

The following example code shows the typical structure of the component definition file of a sample standalone Integer component:

```
{
  "name": "MyCo_MyComponents_MyInteger",
  "label": "My Integer",
  "description": "My Integer",
  "organization": "MyCo",
  "version": "01.01.01",
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyInteger",
  "type": "Field",
  "subtype": "Integer",
  "properties": [
    {
      "name": "label",
      "label": "Field label",
```

©2025 Pegasystems Inc

```
    "format": "TEXT",
    "required": true
  },
  {
    "name": "showGroupSeparators",
    "label": "Show thousands separator",
    "format": "BOOLEAN",
    "visibility": "(!displayAs = slider)"
  },
  {
    "label": "Input settings",
    "format": "GROUP",
    "visibility": "(!readOnly = true)",
    "properties": [
      {
        "name": "displayAs",
        "label": "Display as",
        "format": "SELECT",
        "defaultValue": "input",
        "source": [
          {
            "key": "input",
            "value": "Input"
          },
          {
            "key": "stepper",
            "value": "Stepper"
          },
          {
            "key": "slider",
            "value": "Slider"
          }
```

©2025 Pegasystems Inc

```
      ]
 },
{
   "format": "SELECT",
   "name": "negative",
   "label": "Negative number display",
   "visibility": "(formatter = Currency)",
   "defaultValue": "minus-sign",
   "source": [
     {
       "key": "minus-sign",
       "value": "Standard"
     },
     {
       "key": "parentheses",
       "value": "Accounting"
     }
   ]
 },
 {
   "label": "Advanced",
   "format": "GROUP",
   "collapsible": true,
   "properties": [
     {
       "name": "testId",
       "label": "Test ID",
       "format": "TEXT",
       "ignorePattern": "[^-_\\p{N}\\p{L}]",
       "includeAnnotations": false
     }
   ]
```

©2025 Pegasystems Inc

```
    }
  ],
  "defaultConfig": {
    "label": "@L $this.label"
  }
}
```

The following example code shows the typical structure of the component definition file of a sample library mode Text component in DXCB 25.1.10:

```
{
    "name": "MyCo_MyComp_TextComp",
    "label": "Text component",
    "description": "",
    "organization": "MyCo",
    "version": "0.0.1",
    "library": "MyComp",
    "allowedApplications": [],
    "componentKey": "MyCo_MyComp_TextComp",
    "type": "Field",
    "subtype": "Text",
    "subtypeLabel": "Text",
    "properties": [
        {
            "name": "label",
            "label": "Label value",
            "format": "TEXT"
        },
        {
            "name": "placeholder",
            "label": "Placeholder",
            "format": "TEXT"
```

©2025 Pegasystems Inc

```
    },
    {
      "name": "visibility",
      "label": "Visible",
      "format": "VISIBILITY"
    },
    {
      "name": "disabled",
      "label": "Disabled",
      "format": "DISABLED"
    },
    {
      "name": "required",
      "label": "Required",
      "format": "REQUIRED"
    },
    {
      "label": "Advanced",
      "format": "GROUP",
      "collapsible": true,
      "properties": [
        {
          "name": "testId",
          "label": "Test ID",
          "format": "TEXT",
          "ignorePattern": "[^-_\\p{N}\\p{L}]",
          "includeAnnotations": false
        }
      ]
    },
    {
      "format": "LABEL",
```

©2025 Pegasystems Inc

```
            "label": "About",
            "name": "MyCoAboutLabel",
            "variant": "h3"
        },
        {
            "format": "LABEL",
            "label": "MyCo_MyComp 0.0.1, Wed Aug 20 2025, 25.1.0-17",
            "name": "MyCoAboutData",
            "variant": "primary"
        }
    ],
    "defaultConfig": {
        "detailFVLItem": true,
        "isFormWidth": false,
        "isContainerWidth": false
    },
    "buildDate": "2025-08-20T14:55:11.640Z",
    "infinityVersion": "25.1.0-17",
    "packageCosmosVersion": "8.0.0-build.51.0"
}
```

The following list provides a brief description of the typical headings in the component definition file:

`name`
: The name of the component. Used in the view metadata produced by App Studio to identify the component name to be rendered.

`label`
: The name of the component in the UI.

`description`
: A description of the component's intended function and use.

©2025 Pegasystems Inc

`allowedApplications`

The applications which can use the components. Leave blank to allow all applications to use the component.

> **NOTE:** To use a Constellation DX Component in an application, you must add the Ruleset in which the Constellation DX Component exists to each application specified in `allowedApplications` and save the application definition.

`componentKey`

The identifier associated with the component. This is the same as `name` and should not be changed unless the name of the component is changed. This is for future use.

`type`

The category of the component, such as Field, Template, or Widget.

`subtype`

The subcategory of the selected component type. For example, Form, Page, and Details are the available subtypes of the Template component. Subtypes are also used in UI view authoring to control the templates that are displayed in the template picker. Matching subtypes are grouped together as list items.
In Field components, subtypes are checked against the Constellation DX components' subtype, which is created with the Constellation DX Component Builder. UI view authoring includes the optional **Display as** property pane option, which you use to override the run-time display with that of the Constellation DX component.

`properties`

An array of property definitions (objects) that represent the properties that can be configured for the component. You define the characteristics of a Constellation DX component in this array, such as `name`, `label`, `format`, and `visibility`. For more information, see Properties.

©2025 Pegasystems Inc

`defaultConfig`
: An array of objects representing the default state of the component.

`buildDate`, `infinityVersion`, **and** `packageCosmosVersion`
: These properties are specific to library mode and provide information about the date that the component was validated and published, the version of the Pega Platform server to which the component was authenticated, and the Constellation design system version that was imported.

Recommended reading

See the following articles to learn more about the component definition file's attributes:

- **Properties**
- **Formats**
- **Annotations**
- **Expressions**

Properties

The `properties` attribute in the component definition is an array of objects, each representing a property definition. The property definitions represent the component's configuration through a variety of formats. When authoring UI views, you use these formats to display the configurable options, while also storing the values in the view metadata.

Available properties

The following table shows the available properties, their descriptions, and some examples:

©2025 Pegasystems Inc

Name	Description	Example

`format`	The object type to render in the property panel. Every object in the `properties` attribute must have a `format` attribute.
	
	For more information about the `format` attribute, see Formats.
		`TEXT`
		
		`BOOLEAN`
		
		`VIEWPICKER`

`name`	The name of the object. This attribute might not be required for certain formats, such as `CASCADE` and `GROUP`. The name corresponds to the attribute key value as stored in the configJSON of the view metadata.	`viewPicker`
		
		`placeholder`

`label`	The label associated with the object. This is the label that appears in the UI.	Registered email address Full name of the applicant
`required`	A Boolean value that determines whether the property is required. Appears as an asterisk (*) in the UI. Applicable only to the Field and Widget component types.	`true`
`saveToReference`	A Boolean value that controls where the property value is stored in a view.	`true`

©2025 Pegasystems Inc

Name	Description	Example
	If the user drills into an embedded view, `saveToReference` (if `true`) ensures that the property of this child view reflects on the parent (outer) view. **NOTE:** This only works when you are drilled into the view that you are authoring. Applicable only to the Template component type.	
`visibility`	Determines whether to dynamically show or hide a format based on an expression. For more information, see Expressions.	`"$this.displayAs != 'table' && $this.mode!= 'readonly'"`
`isFormWidth`	A Boolean value that indicates if the field component will span the entire width of the container of the form. • The default value of `isFormWidth` is `false`. • If `isFormWidth` is set to `true`, the field component will span the entire width of the container of the form.	`true`

©2025 Pegasystems Inc

Name	Description	Example
	For more information, see the **Fields in the default form template** section in Default form template.	
`displayMode`	A string value that determines how the renderer returns the component. The `displayMode` property passes to the component as an inherited property even if it is not defined. • If `displayMode` is undefined, the component is expected to be rendered in editable state. • If `displayMode` is set to `DISPLAY_ONLY`, the renderer is expected to return only the component's value for display. You can render the component in a summary panel or in a Details view. **NOTE:** To correctly handle the rendering in a Details view, you must configure the `detailFVLItem` property in the `config.json` file if the component can be	Empty `DISPLAY_ONLY`

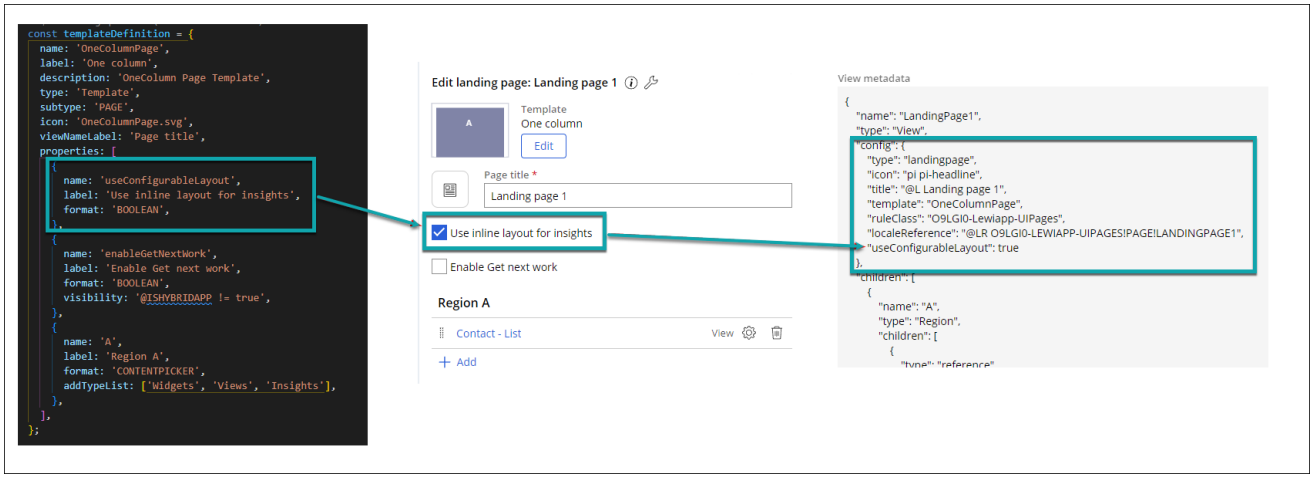
©2025 Pegasystems Inc

Name	Description	Example
	displayed as a single value in a details list.	
`detailFVLItem`	A Boolean value that indicates if the field value list item of the field component can be displayed as plain text along with its label in a details template. • The default value of `detailFVLItem` is `false`. • If `detailFVLItem` is set to `true`, the field component is rendered along with its label in a Details list. **NOTE:** If `detailFVLItem` is set to `true`, the component must return a single read-only value when the inherited property `displayMode` is set to `DISPLAY_ONLY`. For more information, see the **Fields in the details template** section in Details template.	`true`

©2025 Pegasystems Inc

Special considerations

Properties in the Template definition are stored as attributes in the `viewMetadata.config` JSON file. Field and Widget definitions, on the other hand, update the corresponding children `config` JSON entry within the Template definition region entry of `viewMetadata.children`. This is shown in the following figure:



Property attributes for template definition in the viewMetadata.config file

Configuration formats, such as `LABEL`, `GROUP`, and `CASCADE`, must always have nested property definitions, meaning that an inner properties array must be defined in the JSON file. For more information, see Formats.

Related reference

- Formats
- Expressions

Formats

Configuration formats determine the type of element defined in the component definition file, and how a field is displayed visually in the property pane in App Studio.

©2025 Pegasystems Inc

Boolean

Presents the user with a checkbox, as shown in the following figure:

`BOOLEAN` *format property pane*

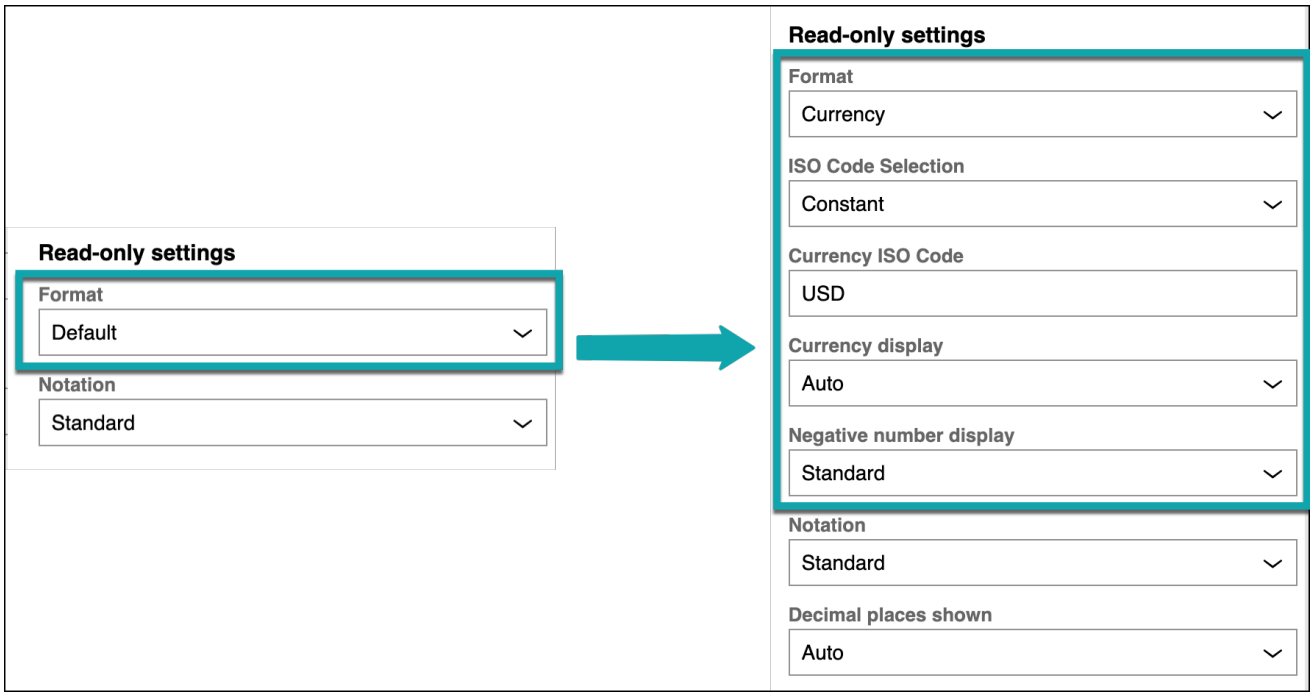
The following example code snippet shows the configuration of the `BOOLEAN` format in the component definition file:

```
{
    "name": "showGroupSeparators",
    "label": "Show thousands separator",
    "format": "BOOLEAN",
    "visibility": "(!$this.displayAs = slider)"
}
```

Cascade

Shows or hides fields that are nested under a format attribute based on the values that are specified in the `source` object. You specify the cascading elements in the `cascadeElements` array and use the `match` attribute to specify the value to show when displaying the elements. The following figure shows an example of the `CASCADE` format with the fields (`cascadeElements`) displayed based on the selection in the **Format** field:

©2025 Pegasystems Inc



`CASCADE` *format property pane with the cascading fields*

The following example demonstrates an implementation of `CASCADE` in the `SELECT` format:

```
{
  "format": "CASCADE",
  "visibility": "(formatter = Currency)",
  "source": {
    "format": "SELECT",
    "label": "ISO Code Selection",
    "name": "isoCodeSelection",
    "defaultValue": "constant",
    "source": [
      {
        "key": "constant",
        "value": "Constant"
      },
      {
```

©2025 Pegasystems Inc

```
        "key": "propertyRef",
        "value": "Property Reference"
      }
     ]
    },
   "cascadeElements": [
      {
        "key": "currencyISOCode",
        "format": "TEXT",
        "name": "currencyISOCode",
        "label": "Currency ISO Code",
        "defaultValue": "USD",
        "match": "constant"
      },
      {
        "key": "currencyISOCode",
        "format": "PROPERTY",
        "name": "currencyISOCode",
        "label": "Currency ISO Code",
        "match": "propertyRef"
      }
    ]
}
```

The following list shows the additional attributes that are specific to this format:

`cascadeElements`
: An array of the possible or allowed keys to match with the field values that are specified in the `source` object.

`cascadeElements.match`
: The value in the parent format field to match with the key in the `cascadeElements` array.

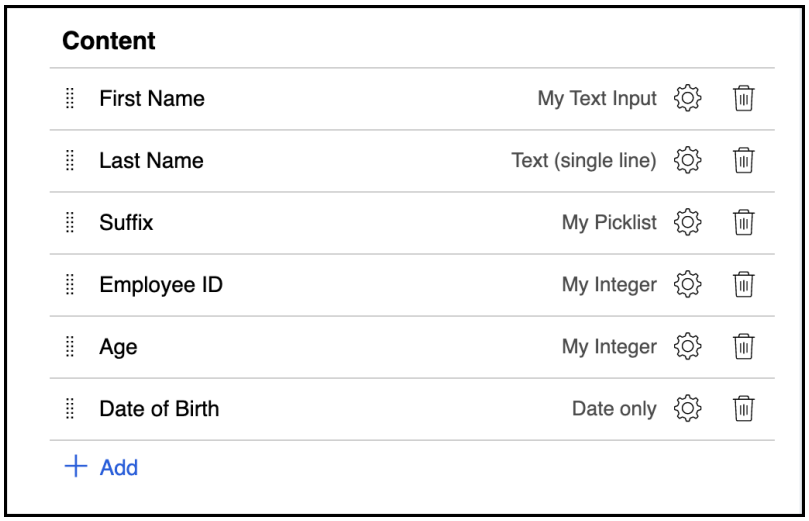
©2025 Pegasystems Inc

`source`

An object that specifies whether to show or hide the `format` field.

Contentpicker

Presents the user with a content picker, in which users can modify the children in the view metadata, as shown in the following figure:



`CONTENTPICKER` *format property pane*

> **NOTE:** The `CONTENTPICKER` format should only be used in the component definition for templates. This is because nested property panels are not supported in view authoring.

The following example code shows the configuration of the `CONTENTPICKER` format in the component definition file:

```
{
    "name": "Fields",
    "label": "Fields",
    "format": "CONTENTPICKER",
    "itemTypes": ["Fields"],
```

©2025 Pegasystems Inc

```
    "allowCreatingGroup": true
}
```

The following list shows additional attributes that are specific to this format:

`allowCreatingGroup`
: A Boolean value that determines whether to create groups in the view metadata with the `CONTENTPICKER`. Defaults to `true`, if not specified.

`itemTypes`
: An array that specifies the type of elements in the region list. The default value is `["Fields", "Views"]`. You can also use `Widgets` and `Insights`. This attribute is a renamed version of `addTypeList`.

You can configure the content picker to use the relevant record structure that the system returns when you edit a `View`. You can also configure the content picker to retrieve addable items from a data page list. If you do that, include a `source` attribute and set the `saveToConfig` attribute to `true`, as shown in the following example:

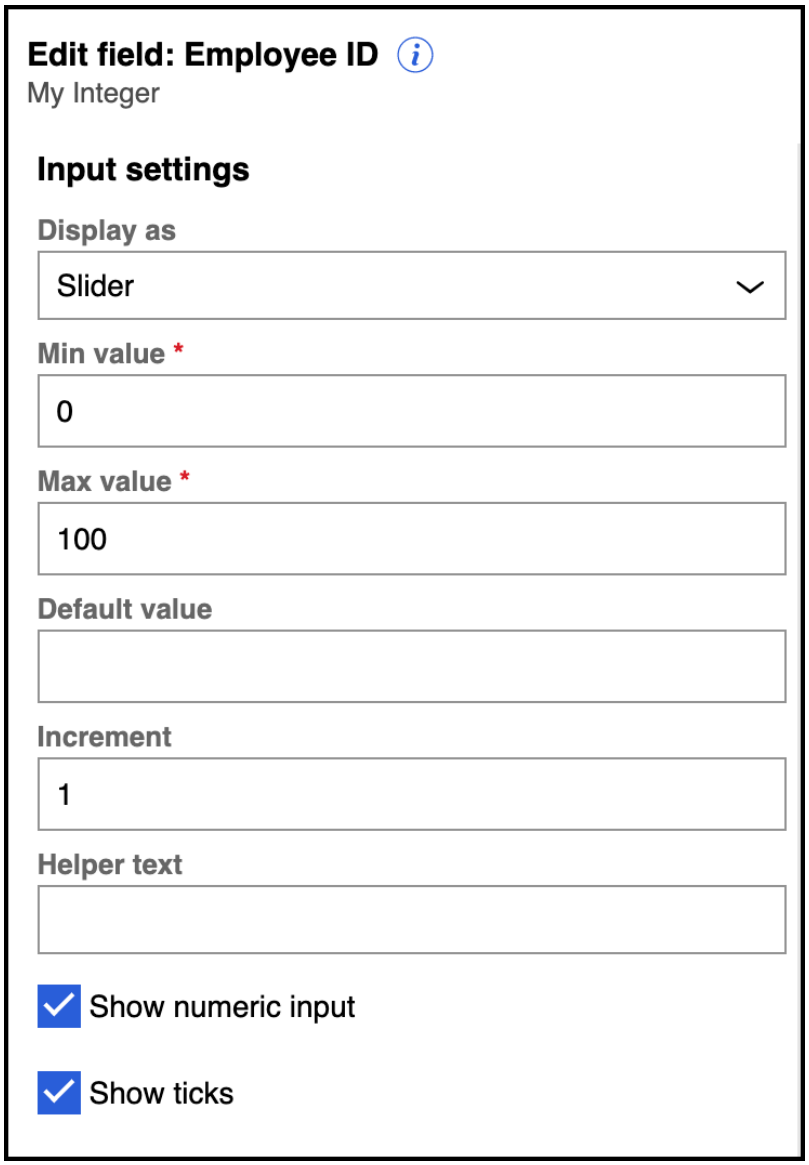
```
{
  "format": "CONTENTPICKER",
  "name": "sourceList",
  "label": "Objects",
  "saveToConfig": true,
  "source": {
    "name": "D_ObjectsToExploreList",
    "parameters": {
      "pyObjectType": "Data",
    },
    "displayProp": "pyCaseName",
    "valueProp": "pyID"
  }
}
```

©2025 Pegasystems Inc

Parameters for the data page list are optional. The system saves the data to a `view` record in the `config` section as an array of values that are based on the `valueProp` setting.

Group

Groups together fields or elements and presents the grouped fields as a single set. Use the `GROUP` format to collect similar controls together. The following figure shows an example of the `GROUP` format:



©2025 Pegasystems Inc

`GROUP` *format property pane*

The following example code shows the configuration of the `GROUP` format in the component definition file:

```
{
  "label": "Input settings",
  "format": "GROUP",
  "visibility": "(!readOnly = true)",
  "properties": [
   {
     "name": "displayAs",
     "label": "Display as",
     ...
     ...
   },
   {
     "name": "min",
     "label": "Min value",
     ...
     ...
   },
   {
     "name": "max",
     "label": "Max value",
     ...
     ...
   },
   {
     "name": "defaultValue",
     "label": "Default value",
     ...
     ...
```

©2025 Pegasystems Inc

```
    },
    {
      "name": "step",
      "label": "Increment",
      ...
      ...
    },
    {
      "name": "helperText",
      "label": "Helper text",
      ...
      ...
    },
    {
      "name": "showInput",
      "label": "Show numeric input",
      ...
      ...
    },
    {
      "name": "showTicks",
      "label": "Show ticks",
      ...
      ...
     }
  ]
}
```

The following list shows the additional attributes that are specific to this format:

`collapsible`
: Determines whether to display the grouped fields as a collapsible item. The default value is `false`.

©2025 Pegasystems Inc

Inline

Displays text in line with a field value, as shown in the following figure:

`INLINE` *format property pane*

The following example code shows the configuration of the `INLINE` format in the component definition file:

```
{
  "format": "INLINE",
  "elements": [
    {
      "key": "next",
      "format": "LABEL",
      "label": "Next",
      "variant": "secondary"
    },
    {
      "key": "nextYear",
      "format": "TEXT",
      "name": "nextYears",
      "label": "Next Year",
      "required": false
    },
    {
      "key": "year",
      "format": "LABEL",
      "label": "Years",
```

©2025 Pegasystems Inc

```
    "variant": "secondary"
   }
  ]
}
```

The following list shows the additional attributes that are specific to this format:

`elements`
: Array elements to display inline with the text.

`elements.variant`
: Determines how the element is displayed. Use this attribute in elements where you want to display the field label in a different style than the default text. Allowed values are `primary`, `secondary`, `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`.

Label

Determines the display format for a field label, as shown in the following figure:

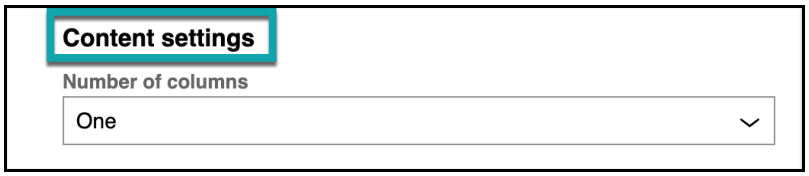
`LABEL` *format property pane*

The following example code shows the configuration of the `LABEL` format in the component definition file:

```
{
  "format": "LABEL",
  "label": "Year range",
  "variant": "secondary"
}
```

The following figure and code snippet of the `LABEL` format with the `h1` `variant` value:

©2025 Pegasystems Inc



`LABEL` *format property pane with h1 style*

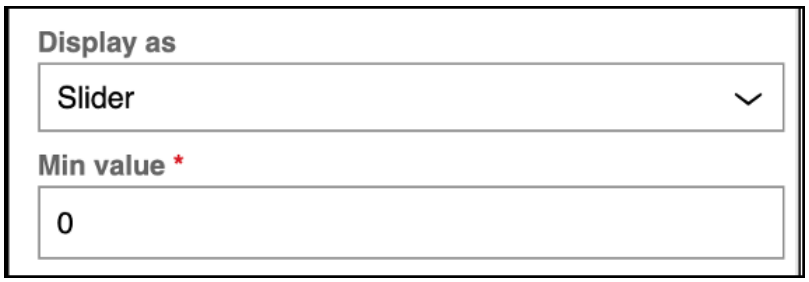
```
{
  "format": "LABEL",
  "label": "Content settings",
  "name": "Content settings",
  "variant": "h2"
}
```

The following list shows the additional attributes that are specific to this format:

`variant`
: Determines how the element is displayed. Use this attribute to display the field label in a different style than the default text. Allowed values are `primary`, `secondary`, `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`.

Number

Presents the user with a text box in which they can input numbers, as shown in the following figure:



NUMBER format property pane

©2025 Pegasystems Inc

The following example code shows the configuration of the `NUMBER` format in the component definition file:

```
{
    "name": "min",
    "label": "Min value",
    "format": "NUMBER",
    "defaultValue": "0",
    "required": true,
    "visibility": "($this.displayAs = slider)"
}
```

The following list shows additional attributes that are specific to this format:

`min`
: The minimum value that the system allows in the input. The default is 0.

`max`
: The maximum value that the system allows in the input.

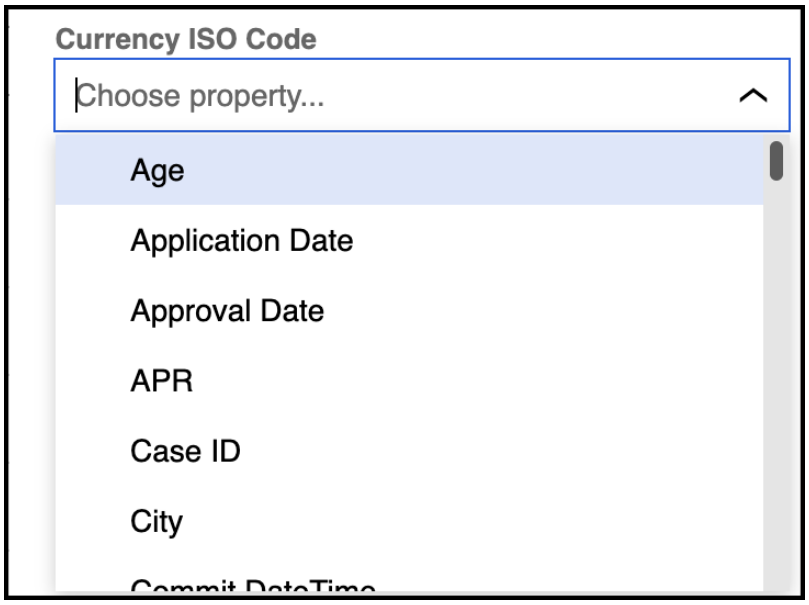
`numberOfDecimals`
: The maximum number of decimal places. The system rounds the value to the number of places.

`showDecimal`
: BOOLEAN value to determine whether to show decimal places. The system rounds the values if this attribute is set to `true`.

Property

Presents the user with a combo box to select a field record from the data model of the case type, as shown in the following figure. The properties that the system retrieves are based on the class of the view.

©2025 Pegasystems Inc



`PROPERTY` *format property pane*

The following example code shows the configuration of the `PROPERTY` format in the component definition file:

```
{
    "key": "currencyISOCode",
    "format": "PROPERTY",
    "name": "currencyISOCode",
    "label": "Currency ISO Code",
    "match": "propertyRef"
}
```

Select

Presents the user with a drop-down list, as shown in the following figure. You can generate the list options based on a data page, or use hard-coded data.

©2025 Pegasystems Inc

`SELECT` *format property pane*

The following example code shows the configuration of the `SELECT` format in the component definition file:

```
{
  "name": "displayAs",
  "label": "Display as",
  "format": "SELECT",
  "defaultValue": "input",
  "source": [
    {
      "key": "input",
      "value": "Input"
    },
    {
      "key": "stepper",
      "value": "Stepper"
    },
    {
      "key": "slider",
      "value": "Slider"
    }
  ]
}
```

The following list shows additional attributes specific to this implementation of the `SELECT` format:

©2025 Pegasystems Inc

`defaultValue`
: Determines the default option from the list to display.

`placeholder`
: Determines the informational text to display in the field.

`source`
: An array of key-value pairs for the list.

The following example code shows the implementation of the SELECT format with options loaded from a data page:

```
{
  "format": "SELECT",
  "name": "displayAs",
  "label": "Display as",
  "source": {
    "name": "D_listOfInputs",
    "parameters": {
      "DataPageParam1": "hardcodedparamvalue",
      "DataPageParam2": "$this.id" // uses value of `id` attribute in the metadata confi
g
    },
    "displayProp": "pyLabel", // default is "label"
    "valueProp": "pyValue" // default is "name"
  }
}
```

NOTE: You can also configure a field using the `SELECT` format to filter field records by using the `source.filter` attribute. For example, in the following sample code, the system filters records based on two fields: one field is empty (NULL), and the other field begins with a given string ("`TEST-FIELDS-SCALAR`") and does not contain a specific character ("`:`").

©2025 Pegasystems Inc

```
{
  "name": "displayfield",
  "format": "SELECT",
  "label": "Display field",
  "visibility": "$this.displayAs != 'table' && $this.mode != 'readonly'",
  "source": {
    "filter":
      "$this.isSystemField IS_NULL && $this.typeKey STARTS_WITH 'Test-Fields-Scalar' && $this.typeKey NOT_CONTAINS ':'"
  }
}
```

The following list shows additional attributes specific to this implementation of the `SELECT` format:

`defaultValue`
: Determines the default option from the list to display.

`placeholder`
: Determines the informational text to display in the field.

`source`
: An object to configure the source of the options. Defaults to use the field records from the view context class, if not specified.

`source.displayProp`
: Determines the item in the source to use as the label. Defaults to `label`, if not specified.

`source.filter`
: The expression for filtering records from the specified source.

`source.name`
: The name of the data page to call. Defaults to use the field records from the view context class, if not specified.

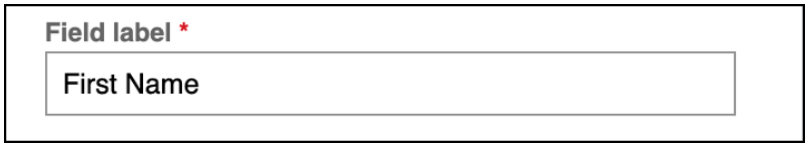
©2025 Pegasystems Inc

`source.parameters`
: An object from the data page parameters.

`source.valueProp`
: Determines the item in the source to use as the value. Defaults to `value`, if not specified.

Text

Presents the user with a text input, as shown in the following figure:



`TEXT` *format property pane*

The following example code shows the configuration of the `TEXT` format in the component definition file:

```
{
    "name": "label",
    "label": "Field label",
    "format": "TEXT",
    "ignorePattern": '[0-9]',
    "required": true
}
```

The following list shows the additional attributes that are specific to this format:

`ignorePattern`
: Sets a regex pattern of characters to ignore in the field input. In the example code, this attribute is configured to ignore numbers.

`includeAnnotations`
: A Boolean value that determines whether to include annotations in the view meta data. Defaults to `true`, if not specified.

©2025 Pegasystems Inc

ValueInput

Presents users with either a text input or a Rule selector when **Custom** is the active selection in the **Field label** list, as shown in the following figure:

VALUEINPUT format property pane

> **NOTE:** The `VALUEINPUT` format was introduced in Pega Platform™ '24.2.

The following example code shows the configuration of the `VALUEINPUT` format in the component definition file:

```
{
    "name": "fieldMessage",
    "label": "Message",
    "format": "VALUEINPUT",
    "ruleType": "Paragraph"
}
```

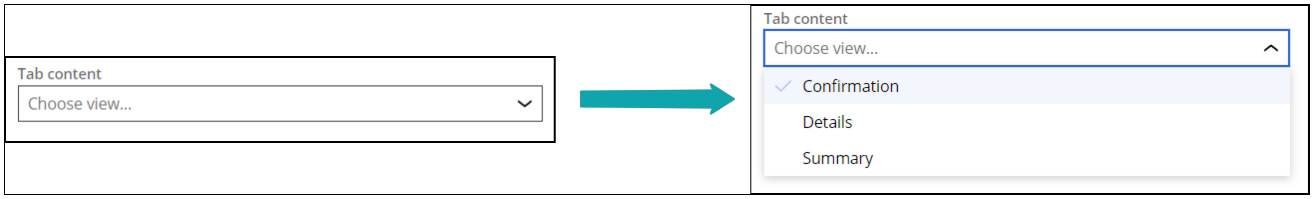
The following additional attribute is specific to this format:

`ruleType`
: Defines the Rule that users select from the system. In the example code, this attribute is configured to select a paragraph Rule.

©2025 Pegasystems Inc

Viewpicker

Presents the user with a combo box to select a view record, as shown in the following figure:



`VIEWPICKER` *format property pane*

The following example code shows the configuration of the `VIEWPICKER` format in the component definition file:

```
{
  "format": "VIEWPICKER",
  "name": "tabContent",
  "label": "Tab content"
}
```

Condition-based properties

Condition-based properties are applicable in all formats. They present the user with a drop-down menu of hard-coded options. Available condition properties are:

- `DISABLED`
- `READONLY`
- `REQUIRED`
- `VISIBILITY`

The following figure shows an example of the condition-based properties:

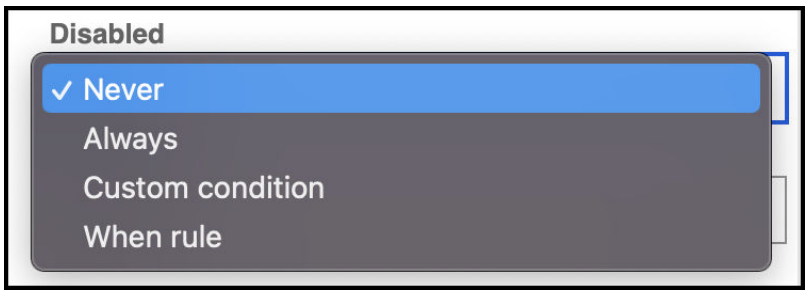
©2025 Pegasystems Inc

Conditions property pane

> **NOTE:** The `VISIBILITY` format is not the same as the `visibility` parameter. The parameter determines when to display a field or field group in the property pane, whereas, the format determines when to show a field during application execution.

The following list shows an example figure and configuration of the properties in the component definition file:

- DISABLED



`DISABLED` *format property pane*

```
{
 "format": "DISABLED",
```

©2025 Pegasystems Inc

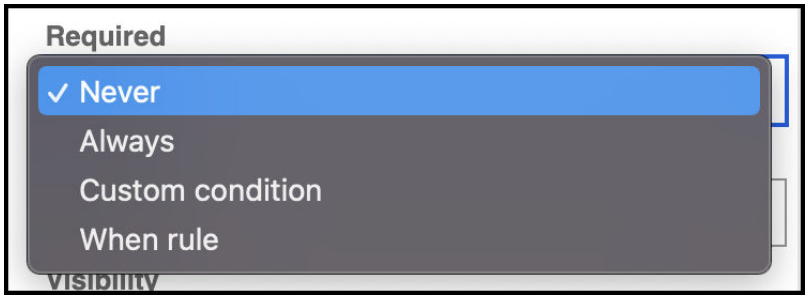
```
  "name": "disabled",
  "label": "Disabled",
  "visibility": "(!$this.readOnly = true)"
}
```

- READONLY

`READONLY` *format property pane*

```
{
  "format": "READONLY",
  "name": "readOnly",
  "label": "Edit mode"
}
```

- REQUIRED

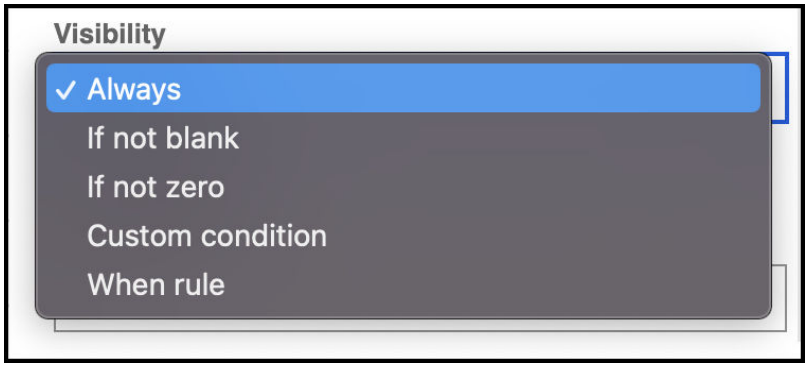


`REQUIRED` *format property pane*

©2025 Pegasystems Inc

```
{
  "format": "REQUIRED",
  "name": "required",
  "label": "Required",
  "visibility": "(!$this.readOnly = true)"
}
```

- VISIBILITY



`VISIBILITY` *format property pane*

```
{
 "format": "VISIBILITY",
 "name": "visibility",
 "label": "Visibility"
}
```

NOTE:

- The **When rule** option displays an additional combo box with a list of When rules that you can select, based on your requirements.
- The **Custom** condition option presents a condition builder in a dialog box in which you define conditions, based on your requirements.

©2025 Pegasystems Inc

Annotations

You use annotations in component metadata to perform specific operations, such as retrieving case information, running When rules, and retrieving environment information, during UI rendering at application runtime.

Supported annotations

The following table shows the annotations that Pega Platform™ supports:

Annotation	Annotation name	Description
@ASSOCIATED	Associated	Resolves the meta data of associated properties and returns the value to the client. Typically used in the picklist component sub-type. If used in a data page, the result is a part of the response. For example: `{ datasource: "@ASSOCIATED .Genders" }`
@CASE	Case	Returns the ID of the current case. For example: `{ caseId: "@CASE" }`

©2025 Pegasystems Inc

Annotation	Annotation name	Description
@DATASOURCE	Data source	Downloads the content in a data page to the client. The client then resolves the right records. Combine this annotation with `@P` to denote the list of properties to resolve, as shown in the example. For example: `{ "datasource": { "source": "@DATASOURCE D_pyAnnouncements .pxResults", "fields": { "name": "@P .pyLabel" } } }`
@E	Expression	Evaluates an expression in the client and returns a Boolean value. All associated property values are also downloaded to the client, to evaluate the expression at the client. For example: `{ visible: "@E .Amount > 0" }`

©2025 Pegasystems Inc

Annotation	Annotation name	Description
@ENV	Environment information	Resolves the environment information such as application name, application description, and access group. The values of `@ENV` are: • `APPLICATION_NAME` - Application name • `ACCESS_GROUP` - Operator access group • `USER_NAME` - Operator name • `USER_ID` - Operator ID • `APPLICATION_DESC` - Application description label • `USER_IMAGE` - Operator image inskey • `LOCALE` - Operator locale • `TIMEZONE` - Operator timezone • `WORK_GROUP` - Operator work group For example: `{ "title": "@ENV APPLICATION_DESC", }`
@FIELD_META	Field meta data	Returns the specified field's metadata as an object. For example: `{ "fieldMetadata": "@FIELD_META .FirstName", }`

©2025 Pegasystems Inc

Annotation	Annotation name	Description
@L	Localize string	Translates a string into a target locale value. The annotated values become part of the locale bundle. For example: `{ label: "@L Enter the first name" }`
@LR	Localization rule	References to a *Rule-UI-Localization* rule. For example: `{ "localeReference": "@LR OYH8EG-C11NAPP-WORK-USECASES!VIEW!CASEWIDELOCALACTIONS3", }`
@P	Property	Downloads the property information from the server and resolves the value from redux at the client. For example: `{ Value: "@P .FirstName" }`
@ROUTING_INFO	Routing information	Returns the information about the container where the component is rendered. For example: `{ "containerInfo": "@ROUTING_INFO", }`

©2025 Pegasystems Inc

Annotation	Annotation name	Description

@USER	User data	Returns the user's object name and ID.
		For example:
		`{`
		`"loggedInAs": "@USER john",`
		`}`

@VIEWAUTHORINGCONTEXT	View authoring context	Returns the type of view being authored.
		The values of `@VIEWAUTHORINGCONTEXT` are:
		• `page`
		• `case`
		For example:
		`{`
		`"visibility": "@VIEWAUTHORINGCONTEXT = 'page'"`
		`}`

@VIEWTYPE	View type	Dynamically changes the behavior of the property panel based on the parent view information.
		The values of `@VIEWTYPE` are:
		• `case`
		• `data`
		• `landingpage`
		• `MultiRecordDisplayAsTable`
		For example:
		`{`
		`"visibility": "@VIEWTYPE == 'MultiRecordDisplayAsT`

©2025 Pegasystems Inc

Annotation	Annotation name	Description
		able'", }

@W	When rule	Executes a When rule on the server and downloads the result to the client. The client then reads the value and passes it to the component.
		For example:
		{
		visible: "@W IsGreater"
		}

Expressions

You use expressions in the component definition file to control field visibility dynamically in the property pane in App Studio™ based on the configuration of another field in the same component. Depending upon the complexity of the expression, you can also build an expression with multiple operators and comparators.

Categories of expressions

The following categories of expressions are available:

Boolean expressions
: Compares two values or variables and returns a Boolean result based on the comparison.

Unary operator
: Evaluates the expression by using a single operator and returns a Boolean result.

Binary operator
: Evaluates the expression by using two operators, such as `!=` or `==`, and returns a Boolean result.

©2025 Pegasystems Inc

String-based expressions include:

CONTAINS and NOT_CONTAINS
: Evaluates the expression based on whether a string contains, or does not contain, the specified characters.

ENDS_WITH and NOT_ENDS_WITH
: Evaluates the expression based on whether a string ends with, or does not end with, the specified characters.

IS_IN_LIST and IS_NOT_IN_LIST
: Evaluates the expression based on whether the string is present in a specified list.

IS_NULL and IS_NOT_NULL
: Evaluates the expression based on whether the variable has a value, or is blank.

STARTS_WITH and NOT_STARTS_WITH
: Evaluates the expression based on whether the string begins with, or does not begin with, the specified characters.

Supported operators

You can use the following operators in expressions:

AND (&&)
: All conditions must be true for the expression to evaluate as true.

Equal to (==)
: Variables on both sides of the operator must match.

Greater than (>)
: Value must be greater.

Greater than or equal to (>=)
: Value must be greater than, or equal to.

Less than (<)
: Value must be lower.

Less than or equal to (<=)
: Value must be lower than, or equal to.

Not equal to (!=)
: Value must not be equal to.

PEGA ©2025 Pegasystems Inc

OR (||)

Any condition must be true for the expression to evaluate as true.

Referencing other properties in expressions

You can build expressions that reference other properties in the component definition. Use `$this` in the expressions in such cases. The following example shows a visibility-based expression that references the `columnWidth` attribute:

For example:

```
{
   "name": "width",
   "label": "Width (px)",
   "format": "NUMBER",
   "visibility": "$this.columnWidth == 'custom'"
}
```

Composite expressions

You can also combine multiple expressions into a single expression by enclosing the individual expressions within parenthesis.

For example:

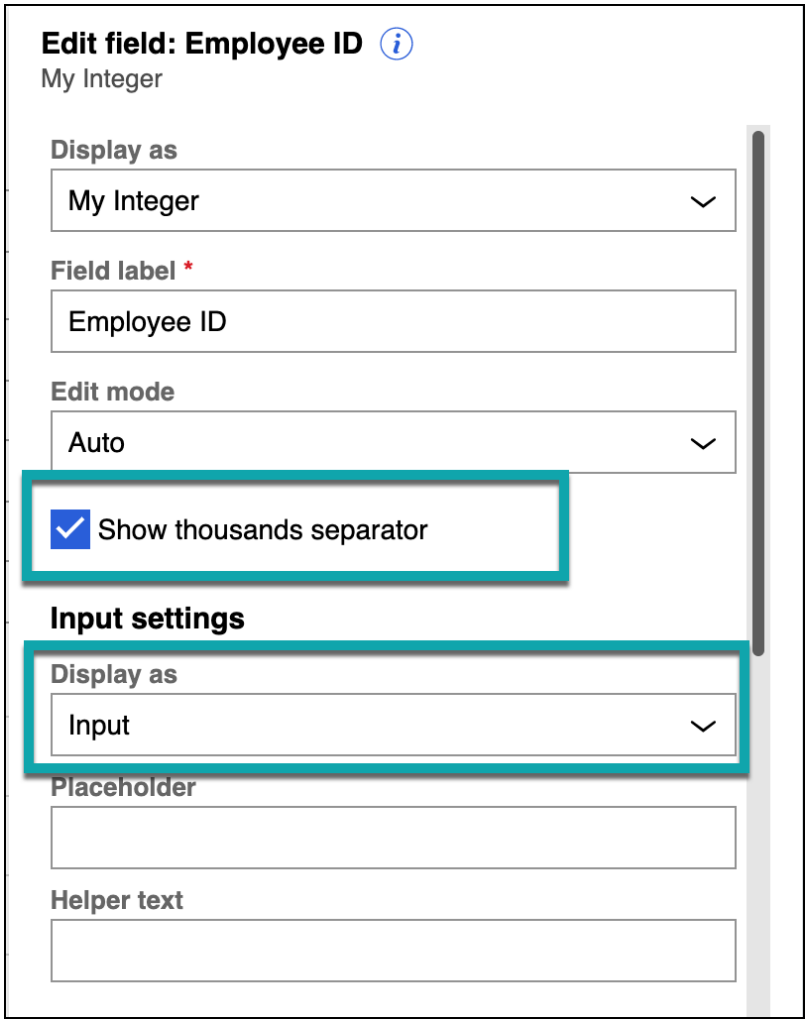
`"($this.displayAs = 'input' || $this.age > 55) && $this.minor IS_NULL"`

Using expressions to control field visibility

In the following example, the **Employee ID** field is based on an Integer Constellation DX component. The input in one of the fields uses an expression that controls the visibility

PEGA ©2025 Pegasystems Inc

of a checkbox in the component. The following figure shows the field property pane with the checkbox and the field setting, on which its visibility depends:



Example of how the field visibility in a component is controlled using expressions

The system requires the **Show thousands separator** checkbox to be visible when you set the **Display as** field to either **Input** or **Stepper**.

The following example code snippet shows the configuration of the **Display as** field in the component definition file:

```
{
  "name": "displayAs",
```

©2025 Pegasystems Inc

```
  "label": "Display as",
  "format": “SELECT",
  "defaultValue": "input",
  "source": [
    {
     "key":   "input",
     "value":    Input"
    },
    {
     "key": "stepper",
     "value": "Stepper"
    },
    {
     "key": "slider",
     "value": "Slider”
    }
   ]
}
```

You can use the following example of an expression in the `visibility` attribute of the checkbox to evaluate this condition:

```
{
   "name": "showGroupSeparators",
   "label": "Show thousands separator",
   "format": "BOOLEAN",
   "visibility": "$this.displayAs !=  'slider'"
}
```

Other valid configurations include the following examples:

```
"$this.displayAs == 'input' || $this.displayAs == 'stepper'"
```

©2025 Pegasystems Inc

```
"$this.displayAs IS_IN_LIST 'input,stepper'"
```

```
"($this.displayAs = 'input' || $this.thing1 > 5) && $this.thing2 IS_NULL"
```

©2025 Pegasystems Inc

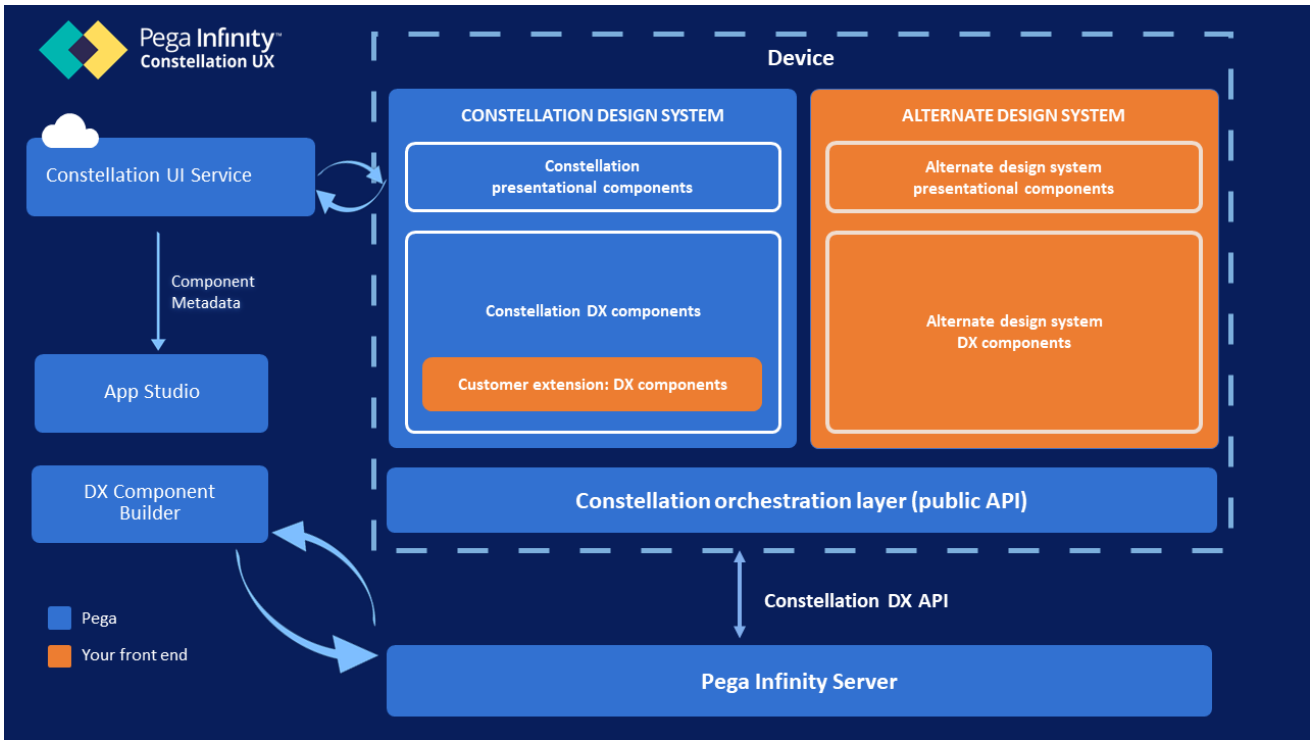
Working with PCore and PConnect public APIs

In Constellation applications, the Constellation engine relies on PCore and PConnect public APIs to deliver the application's UI. This approach delivers access to state management, actions, events, and a higher level of abstraction to the DX APIs.

> **NOTE:** For the complete PCore and PConnect documentation, see Using PCore and PConnect Public APIs.

The Constellation orchestration layer is the backbone of the Constellation architecture and contains the PCore and PConnect APIs. The Constellation DX components that are part of the Constellation design system use the PCore and PConnect APIs to connect the Constellation presentational components to the Pega Platform™ Server.

©2025 Pegasystems Inc



Constellation Client Architecture

Type definitions

Using TypeScript type definitions, the IDE helps you with code completion and provides the functional definition, functional parameters, and return data.

For more information on type definitions, see Using type definitions for PCore and PConnect APIs.

©2025 Pegasystems Inc

Working with Constellation Design System

Front-end developers can construct a feature-rich UI by using various components of the Constellation design system library ranging from basic building blocks to complex layouts.

> **NOTE:** For detailed information on the components of the Constellation design system library, see https://design.pega.com/extend/.

Import and register Pega icons

A Pega icon is a component provided by the Constellation design system. Pega icons are used to depict an idea, object, or action. For more information, see Icon.

To use Pega icons with Constellation DX components, you must import and register them.

1. Open the `package.json` file, scroll to the `dependencies` section, and check whether the `@pega/cosmos-react-core` package is present. If not, you must add the `@pega/cosmos-react-core` package.
2. Ensure that the icon you want to import is present in the library at https://design.pega.com/design/icon/.
3. Open the `index.tsx` file for your component, and then add the code for importing and registering the icon at the top.

   Example

   If you want to import and register the **align-center** icon, add the following code at the top of the `index.tsx` file:

©2025 Pegasystems Inc

```
import {
  registerIcon,
} from '@pega/cosmos-react-core';
// Import the icon
// Icons are available under @pega/cosmos-react-core/lib/components/Icon/icons and follow kebab case
import * as alignCenterIcon from '@pega/cosmos-react-core/lib/components/Icon/icons/align-center.icon';
// Register the icon after importing it
registerIcon(alignCenterIcon);
```

If you want to import and register the **align-center** and **times** icons, add the following code at the top of the `index.tsx` file:

```
import {
  registerIcon,
} from '@pega/cosmos-react-core';
// Import the icons
// Icons are available under @pega/cosmos-react-core/lib/components/Icon/icons and follow kebab case
import * as alignCenterIcon from '@pega/cosmos-react-core/lib/components/Icon/icons/align-center.icon';
import * as timesIcon from '@pega/cosmos-react-core/lib/components/Icon/icons/times.icon';
// Register the icons after importing them
registerIcon(alignCenterIcon, timesIcon);
```

Pega icons are now imported and registered, and can be used by Constellation DX components.

©2025 Pegasystems Inc

Constellation DX component tutorials

Use Constellation DX components to extend Constellation applications to augment your out-of-the-box experience.

The Constellation DX Component Builder is used to build and publish Constellation DX components. The Constellation developer experience allows you as a developer to use familiar tools and development workflows to develop your own components.

For a more detailed view of the Constellation DX components, see the Constellation DX Components UI Gallery, which contains a collection of ready-to-use and customizable components. You can use the gallery to explore the components and view their source code.

To work with Constellation DX components, you must have an understanding of the following concepts:

- React functional component patterns, including React Hooks
- Fundamentals of Constellation JavaScript public APIs, including the PCore and PConnect interfaces

The Constellation DX component tutorial will help you achieve the following:

- Identify the type of Constellation DX component you need for your business requirement.
- Adapt the Constellation DX Component Builder's out-of-the-box templates to meet your business requirement.
- Know the various ConstellationJS APIs available and use them to accomplish your business needs.
- Understand the Constellation Design System and connect its components to the ConstellationJS APIs to create a new Constellation DX component.

PEGA ©2025 Pegasystems Inc

- Ensure the Constellation DX components you create adhere to the Application Theme configured in Pega App Studio.
- Use tools such as JavaScript/Typescript, VS Code, Git, Webpack, NodeJS, NPM, Storybook, Jest, etc to build and test your component locally.

This tutorial contains the following topics:

- **Field component**
- **Widget component**
- **Layout Template component**

Field component

Use a field component to display a field in a manner that cannot be achieved with any other out-of-the-box component.

In this tutorial, we will build a field component that allows you to launch a local action using a button in a modal overlay. This will demonstrate how an advanced component can be created for rendering a button on a view.

For more information on the Button component, see Button.

After completing this tutorial, you will be able to:

- Extend a field component for advanced use cases that are not supported by Constellation.
- Use the APIs in the ActionsApi class through your field's PConnect object.
- Retrieve data using the getValue API and use the PCore constants.
- Use the openLocalAction API to launch a local action from a button.
- Use the openLocalAction callback functionality for hooking into the modal submit or cancel actions.

Learn about the following aspects of a field:

©2025 Pegasystems Inc

- **Creating a field**
- **Editing the field**
- **Mocking and testing the field with a Storybook**
- **Publishing the field**
- **Using the field in App Studio**

Creating a field

Learn how to create a field.

Creating a component

For the purposes of this tutorial, we will model our button as a URL field.

To create a new URL field, follow the procedure below:

1. Open your preferred terminal tool, navigate to your project, and enter `npm run create`.

> **NOTE:** Ensure your project is initialized before running this command. For more information on initializing a project, see Initializing a project.

©2025 Pegasystems Inc

2. Create the URL field with the following details:
   - Component name: *MyLocalActionLauncher*
   - Component label: *My Local Action Launcher*
   - Component version: *01.01.01*
   - Library name: *MyComponents*
   - Supported applications: Do not provide any supported applications, so that the component is available for all applications by default.
   - Component type: *Field*

```
? Enter type of component
  1) Field
  2) Layout template
  3) Widget
  Answer:
```

Component type

   - Component sub-type: *URL*

```
? Enter subtype of component
  3) Paragraph
  4) Email
  5) Phone
  6) URL
  7) Integer
  8) Decimal
  9) Currency
```

Component sub-type

   - Component description: *My Local Action Launcher*

©2025 Pegasystems Inc

```
DX Component Builder v2.0.16
? Enter component name (required) MyLocalActionLauncher
? Enter component label for display (required) My Local Action Launcher
? Enter component version 01.01.01
? Enter library name (required) MyComponents
? Please enter the application names to be supported (comma-separated). Keep empty for all applications
? Enter type of component Field
? Enter subtype of component URL
? Enter description for the component My Local Action Launcher
```

Creating URL Field

The `MyCo_MyComponents_MyLocalActionLauncher` component is created in the `<your project folder>/src/components/MyCo_MyComponents_MyLocalActionLauncher` path as seen below:

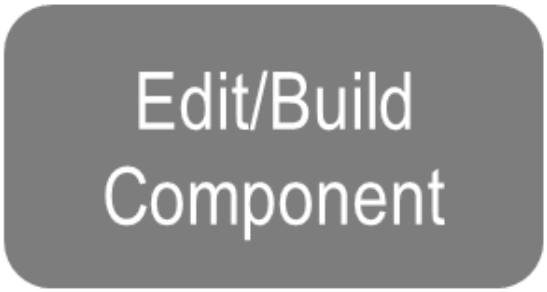
```
src
    components
        MyCo_MyComponents_MyLocalActionLauncher
            config.json
            demo.stories.tsx
            event-utils.ts
            index.tsx
            styles.ts
            mock.stories.ts
            suggestions-handler.ts
            text-url.ts
```

> **NOTE:** The list of files above is indicative. You may see a slightly different list of files depending on the version of the Constellation DX Component Builder you are running.

Editing the field

Customize the field by updating its auto-generated code.

©2025 Pegasystems Inc



Editing or building a component

In this tutorial, we will customize the field to display text on a button and launch a local action when the button is clicked.

1. Navigate to the `MyCo_MyComponents_MyLocalActionLauncher` folder and delete all files except the following:
   - `config.json`
   - `demo.stories.tsx`
   - `index.tsx`
2. Rename the `demo.stories.tsx` file to `mymodallauncher.stories.tsx`. The `MyCo_MyComponents_MyLocalActionLauncher` folder now contains the following files:

```
□—— src
    □—— components
        □—— MyCo_MyComponents_MyLocalActionLauncher
            □—— config.json
            □—— mymodallauncher.stories.tsx
            □—— index.tsx
```

3. Replace the contents of the `config.json` component definition file with the following code:

```
{
  "name": "MyCo_MyComponents_MyLocalActionLauncher",
```

©2025 Pegasystems Inc

```
  "label": "My Local Action Launcher",
  "description": "My Local Action Launcher",
  "organization": "MyCo",
  "version": "01.01.01",
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyLocalActionLauncher",
  "type": "Field",
  "subtype": "Text-URL",
  "properties": [
    {
      "name": "buttonText",
      "label": "Button Text",
      "format": "TEXT",
      "required": true
    },
    {
      "name": "actionId",
      "label": "Action ID",
      "format": "TEXT",
      "required": true
    },
    {
      "name": "testId",
      "label": "Test ID",
      "format": "TEXT"
    }
  ],
  "defaultConfig": {
  }
}
```

4. Replace the contents of the `index.tsx` file with the following code:

©2025 Pegasystems Inc

```
/* global PCore */
/* eslint no-undef: 0 */
import PropTypes from "prop-types";
import { Button } from "@pega/cosmos-react-core";

export function launchLocalAction(
    getPConnect,
    assignmentID,
    action,
    cb) {
  const actionsAPI = getPConnect().getActionsApi(); // □
  const openLocalAction = actionsAPI
    && actionsAPI.openLocalAction.bind(actionsAPI);
  let caseID;
  if (assignmentID.indexOf('!') === -1) {
    caseID = assignmentID;
  } else {
    // "ASSIGN-WORKLIST ON8TTL-C11NGALL-WORK LAT-3!ENTERINFO_FLOW_1"
use case
    const [assignKey] = assignmentID.split('!');
    const [, className, workId] = assignKey.split(' ');
    // eslint-disable-next-line prefer-template
    caseID = className + ' ' + workId;
  }
  openLocalAction(action.ID, { // □
    caseID,
    type: 'express',
    containerName: 'modal',
    name: action.name,
    callbacks: { // □
      submit: (response) => {
        cb(response);
```

©2025 Pegasystems Inc

```
      },
      cancel: () => {
        cb();
      }
    }
  });
}

const callback = payload => { // □
  // eslint-disable-next-line no-console
  console.log("Callback payload:", payload);
};

const MyCoMyComponentsMyLocalActionLauncher = props => {
  const {
    getPConnect,
    buttonText,
    testId,
    actionId
  } = props;

  const availableActions =
    getPConnect()
    .getValue(PCore.getConstants().CASE_INFO.AVAILABLEACTIONS); // □

  const action =
    availableActions
    .filter((act) => act.ID === actionId); // □

  const assignmentId =
    getPConnect()
    .getValue(PCore.getConstants().CASE_INFO.ASSIGNMENT_ID);
```

©2025 Pegasystems Inc

```
  const handleClick = () => {
    launchLocalAction( // 
      getPConnect,
      assignmentId,
      action[0],
      callback);
  };

  return (
    <>
      {action && assignmentId && (
        <Button
          variant="simple"
          onClick={handleClick}
          data-test-id={testId}
        >
          {buttonText}
        </Button>
      )}
    </>
  );
};
MyCoMyComponentsMyLocalActionLauncher.propTypes = {
  buttonText: PropTypes.string,
  getPConnect: PropTypes.func,
  testId: PropTypes.string,
  actionId: PropTypes.string
};
export default MyCoMyComponentsMyLocalActionLauncher;
```

The code is described below:

©2025 Pegasystems Inc

- The getActionsApi API is used to obtain access to the openLocalAction API. For more information on these APIs, see getActionsApi() and openLocalAction(actionID, options).

- A bind operation is performed on the ActionsApi object to make it accessible from within the context of the promise callback.

- These are the success and cancel callbacks that can be invoked once the promise resolves.

- This is the actual callback function that is called. When the callback is successful, the payload is passed as output, while if the callback fails, nothing is passed. This callback function is optional.

- The getValue API obtains the list of available actions for this case instance. The getValue API is accessed through the getPConnect API passed in to all components through `props`. For more information on the getValue API, see getValue(prop, pageReference).

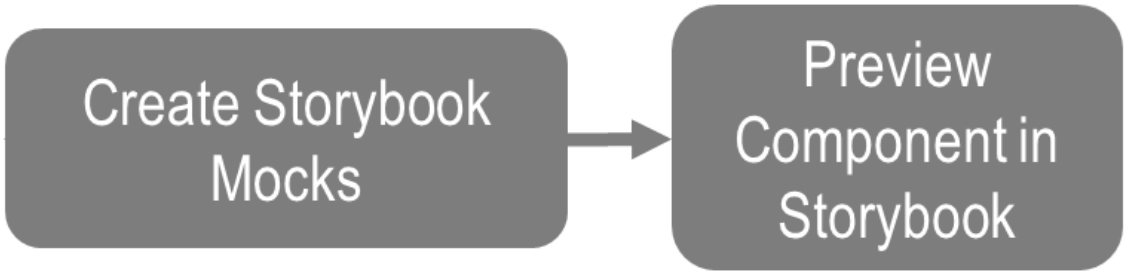
- The list of available actions is filtered to return the action we're interested in. This also validates at run-time that the `actionId` passed as a prop exists in the list of available actions.

- The local action is launched through the `handleClick` event handler. The first result of the `action` filtered array is passed, along with a reference to the callback function.

Mocking and testing the field with a Storybook

Use a Storybook to preview the field and perform unit testing on it.

©2025 Pegasystems Inc



Creating Storybook mocks and previewing component in Storybook

In this tutorial, we will only test that the button is displayed with the correct text and that the input parameters are parsed.

Replace the contents of the `mymodallauncher.stories.tsx` file with the following code:

```
import MyCoMyComponentsMyLocalActionLauncher from './index.tsx';

export default {
  title: 'MyCoMyComponentsMyLocalActionLauncher',
  component: MyCoMyComponentsMyLocalActionLauncher
};

if(!window.PCore) { //
  window.PCore= {};
}

window.PCore.getConstants = () => { //
  return {
    CASE_INFO: {
      CASE_INFO: 'caseInfo',
      ASSIGNMENT_ID: 'caseInfo.assignments[0].ID',
      AVAILABLEACTIONS: 'caseInfo.availableActions'
    }
```

©2025 Pegasystems Inc

```
  }
}

export const baseMyCoMyComponentsMyLocalActionLauncher = () => { //
  const props = {
    buttonText: 'Test',
    testId: '123456',
    actionId: 'pyUpdateCaseDetails',
    getPConnect: () => { //
      return {
        getActionsApi: () => {
          return {
            openLocalAction: () => {
              return {};
            }
          };
        },
        getValue: (parameter) => { //
          switch (parameter) {
            case window.PCore.getConstants().CASE_INFO.ASSIGNMENT_ID :
              return "ASSIGN-WORKLIST ON8TTL-C11NGALL-WORK LAT-3!ENTERINFO_FLO
W_1";
            case window.PCore.getConstants().CASE_INFO.AVAILABLEACTIONS :
              return [
                {
                  name: 'Edit details',
                  links: {
                    open: {
                      rel: 'self',
                      href: '/cases/ON8TTL-C11NGALL-WORK L-3/actions/pyUpdateCaseDetail
s',
                      type: 'GET',
```

©2025 Pegasystems Inc

```
                title: 'Get case action details'
              }
            },
            ID: 'pyUpdateCaseDetails',
            type: 'Case'
          },
          {
            name: 'Change stage',
            links: {
              open: {
                rel: 'self',
                href: '/cases/ON8TTL-C11NGALL-WORK L-3/actions/pyChangeStage',
                type: 'GET',
                title: 'Get case action details'
              }
            },
            ID: 'pyChangeStage',
            type: 'Case'
          }
        ];
        default:
          return {};
      }
    }
  };
}
};

return (
  <>
    <MyCoMyComponentsMyLocalActionLauncher {...props} />
  </>
```

©2025 Pegasystems Inc

```
  );
};
```

The code is described below:

- This sets up the global PCore mock. If you are executing multiple Stories in your Storybook, it is a best practice to move your global mock data into a single file for your project.

- This is the primary (base) Story, where the Story-specific mock data is set up.

- The calls to the openLocalAction API are stubs since we are not launching a modal dialog in the Story.

- The getValue Constellation run-time API returns the mock data. For more information on building mock data, see Structure of Constellation Core configuration data.

Publishing the field

Perform integration testing on the field by publishing it to Pega Infinity™.

Authenticating and publishing the component

To use a component in App Studio Authoring, you must publish it. Publishing the component bundles all the assets together and uses Webpack to create a gzipped package. The Constellation DX Component Builder then connects with the Pega Infinity service to create or update an instance of Rule-UI-Component. A declare trigger on

©2025 Pegasystems Inc

Rule-UI-Component performs the onward publish to the Constellation Service. For more information about the Constellation Service, please see Constellation UI Service.

Authenticate your access to the Pega Infinity Server and publish the component.

For more information, see:

- Authenticating Pega Platform Server user access for standalone Constellation DX components
- Publishing standalone Constellation DX components

> **NOTE:** Your authentication might expire while publishing the component. Run the `npm run authenticate` command again to get a new token.

The field is now ready to be tested in App Studio.

Using the field in App Studio

Learn how to use the field in App Studio.

Testing the component

After the field is published successfully, follow the below procedure to use it in App Studio:

©2025 Pegasystems Inc

1. In the **Navigation** pane of App Studio, click **Case types**, and click an existing case type.
2. Click the **User interface** tab, click the **Views** tab, and click an existing form view.
3. Click the **Configure** icon for an existing URL field.

> **NOTE:** If no URL field exists for the selected form view, create a new URL field.

4. In the **Edit field** dialog box, in the **Display as** list, select **My Local Action Launcher**.
5. In the **Button Text** box, enter the text that must be displayed on the button.
6. In the **Action ID** box, enter **pyUpdateCaseDetails**.
7. In the **Test ID** box, enter a random number, and click **Save** to save the field.
8. Click **Save** to save the case type.
9. Click **Preview** to preview the case type.

Widget component

Use a widget to display interactive self contained information.

You can use the Constellation DX Component Builder to create three types of widgets:

- **PAGE** - This widget can be used only in landing pages.
- **CASE** - This widget can be used only in case views.
- **PAGE & CASE** - This widget can be used in both landing pages and case views.

In this tutorial, we will go through a custom AppAnnouncement widget that is created by Constellation DX Component Builder. For more information on the AppAnnouncement widget, see AppAnnouncement.

After completing this tutorial, you will be able to:

©2025 Pegasystems Inc

- Create your own version of the AppAnnouncement widget using a different data source and styling.
- Use the getDataApiUtils() API to directly access data provided by data pages and pass parameters.
- Process the results and change the styling of your component while being consistent with the portal theme that wraps your widget at run time.

Learn about the following aspects of a widget:

- **Creating a widget**
- **Reviewing the structure of a widget**
- **Editing the widget**
- **Mocking and testing the widget with a Storybook**
- **Styling the widget**
- **Publishing the widget**
- **Using the widget in App Studio**

Creating a widget

Learn how to create a widget.

©2025 Pegasystems Inc

Creating a component

To create a PAGE widget, follow the procedure below:

1. Open your preferred terminal tool, navigate to your project, and enter `npm run create`.

> **NOTE:** Ensure your project is initialized before running this command. For more information on initializing a project, see Initializing a project.

2. Create the PAGE widget with the following details:
   - Component name: *MyPageWidget*
   - Component label: *My Page Widget*
   - Component version: *01.01.01*
   - Library name: *MyComponents*
   - Supported applications: Do not provide any supported applications, so that the component is available for all applications by default.
   - Component type: *Widget*

```
? Enter type of component
  1) Field
  2) Layout template
  3) Widget
```

Component type

   - Component sub-type: *PAGE*

```
? Enter subtype of component
  1) CASE
  2) PAGE
  3) PAGE & CASE
```

©2025 Pegasystems Inc

Component sub-type

- Component description: *My Page Widget*

```
> widgetcomponent@1.0.0 create
> custom-dx-components create

DX Component Builder v2.0.16
? Enter component name (required) MyPageWidget
? Enter component label for display (required) My Page Widget
? Enter component version 01.01.01
? Enter library name (required) MyComponents
? Please enter the application names to be supported (comma-separated). Keep empty for all applications
? Enter type of component Widget
? Enter subtype of component PAGE
? Enter description for the component My Page Widget
```

Creating Page Widget

The `MyCo_MyComponents_MyPageWidget` component is created in the `<your project folder>/src/components/MyCo_MyComponents_MyPageWidget` path as seen below:

```
src
    components
        MyCo_MyComponents_MyPageWidget
            index.tsx
            config.json
            styles.ts
            demo.stories.tsx
            mock.stories.ts
            OneColumnPage.svg
```

> **NOTE:** The list of files above is indicative. You may see a slightly different list of files depending on the version of the Constellation DX Component Builder you are running.

©2025 Pegasystems Inc

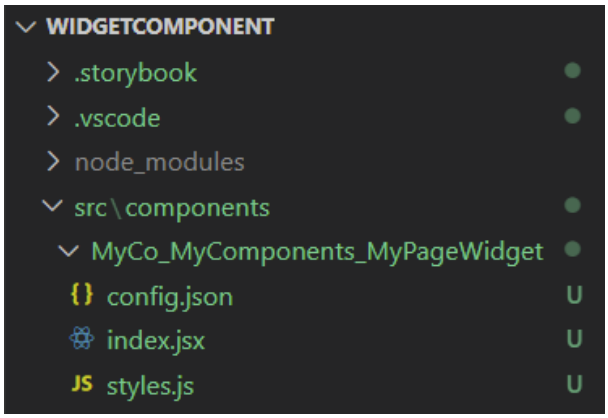
Reviewing the structure of a widget

Learn about the auto-generated files that are part of a widget.

After creating the widget, navigate to its source directory and you will see several auto-generated files within the component.

Navigate to the `MyCo_MyComponents_MyPageWidget` folder and delete all files except the following:

- `config.json`
- `index.tsx`
- `styles.ts`



Structure of a widget

The widget contains the following important files:

- `config.json`
- `index.tsx`
- `styles.ts`

`config.json`

The `config.json` component definition file drives the App Studio experience for the component. It shows how the component is displayed in App Studio. As a component

©2025 Pegasystems Inc

developer, you can control the configuration data captured when the component is used by App Studio. The data captured is passed into your component as `props` at run time. For more information on the component definition schema, see Component definition.

The contents of the component definition file is as shown below.

```
{
  "name": "MyCo_MyComponents_MyPageWidget",
  "label": "My Page Widget",
  "description": "My Page Widget",
  "organization": "MyCo",
  "version": "01.01.01",
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyPageWidget",
  "type": "Widget",
  "subtype": "PAGE",
  "properties": [
    {
      "name": "description",
      "label": "Description",
      "format": "TEXT",
      "defaultValue": "We've launched a brand new experience to accelerate your wor
kflow. Check out the guides to help you get the most of Constellation."
    },
    {
      "name": "whatsnewlink",
      "label": "What's new link",
      "format": "TEXT",
      "defaultValue": "https://design.pega.com"
    },
```

©2025 Pegasystems Inc

```
    {
      "name": "image",
      "label": "Image link",
      "format": "TEXT",
      "defaultValue": <... removed for brevity ...>
    },
    {
      "name": "datasource",
      "label": "Data source for details",
      "defaultValue": "@DATASOURCE D_pyAnnouncements.pxResults",
      "format": "DATASOURCE",
      "required": true,
      "properties": [
        {
          "name": "name",
          "label": "Label",
          "defaultValue": "@P .pyLabel"
        }
      ]
    },
    {
      "label": "Conditions",
      "format": "GROUP",
      "properties": [
        {
          "name": "visibility",
          "label": "Visibility",
          "format": "VISIBILITY"
        }
      ]
    }
  ],
```

©2025 Pegasystems Inc

```
  "defaultConfig": {}
}
```

The code is described below:

☐ - The use of the PAGE widget is restricted within landing pages in App Studio.

☐ - This config handles a list based data page that does not have parameters. The `datasource` property called `name` is mapped to the `.pyLabel` property of the `.pxResults` page list property. `pxResults` is an array of objects exposed by the run-time instance of the `D_pyAnnouncements` data page.

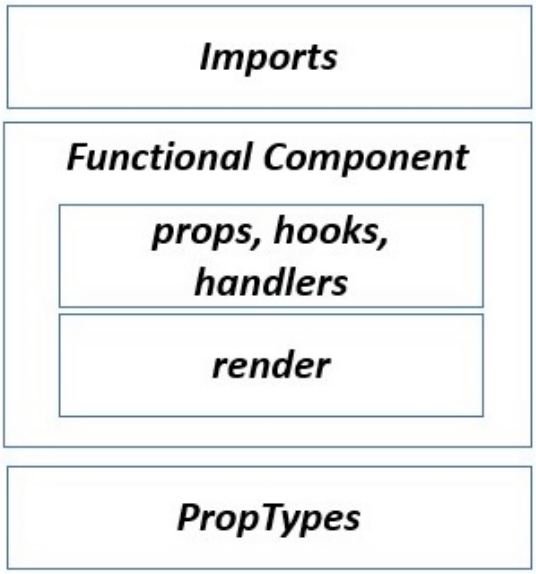
> **NOTE:** Please remove the numbered notations from the above code if you are copying it.

For more information on the `config.json` file, see Component definition.

`index.tsx`

Typically, a functional DX component is structured as shown below.

©2025 Pegasystems Inc



Structure of an index.tsx file

The contents of the `index.tsx` file is as shown below.

```
import PropTypes from 'prop-types';
import { AppAnnouncement as PegaAppAnnouncement } from '@pega/cosmos-react
-work'; //

import StyledMyCoMyComponentsMyPageWidgetWrapper from './styles';

export default function MyCoMyComponentsMyPageWidget(props) {

  const { header, description, datasource, whatsnewlink, image } = props;

  let details = [];

  if (datasource && datasource.source) { //
    details = datasource.source.map((item) => {
      return item.name;
```

©2025 Pegasystems Inc

```
    });
  }

  return (
    <StyledMyCoMyComponentsMyPageWidgetWrapper>
    <PegaAppAnnouncement
      heading={header}
      description={description}
      details={details}
      whatsNewLink={whatsnewlink}
      image={image.replace(/ /g, '+')}
    />
    </StyledMyCoMyComponentsMyPageWidgetWrapper>
  );
}

MyCoMyComponentsMyPageWidget.defaultProps = {
  header: '',
  description: '',
  image: '',
  datasource: [],
  whatsnewlink: ''
};

MyCoMyComponentsMyPageWidget.propTypes = {
  header: PropTypes.string,
  description: PropTypes.string,
  datasource: PropTypes.instanceOf(Object),
  whatsnewlink: PropTypes.string,
  image: PropTypes.string
};
```

©2025 Pegasystems Inc

The code is described below:

□ - The generated code uses the Constellation Design System component.

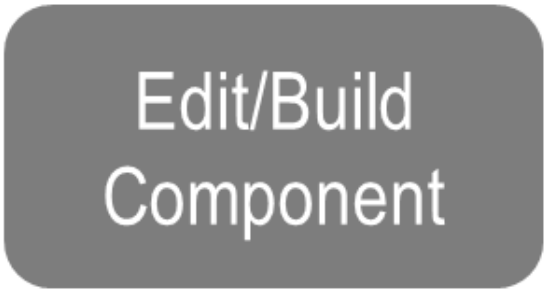
□ - This shows the use of a `datasource` configured in the component defintion using the `@DATASOURCE` annotation. The `datasource` points to `D_pyAnnouncements` and this code assumes that `datasource` resolves to a list based data page where each page has a `name` property. You can change the construction of the `details` array by modifying the transformation of the datasource data.

styles.ts

For information on the `styles.ts` file, see styles.ts.

Editing the widget

Customize the widget by updating its auto-generated code.



Editing or building a component

In this tutorial, we will build a widget that accesses data at run time. We will also use this widget to perform advanced styling. To do this and avoid installing anything on the Pega Infinity™ platform, let us reuse two of the existing data pages deployed with Pega Platform™: `D_pyCountryList` and `D_pyStateList`.

©2025 Pegasystems Inc

`D_pyCountryList` is a data page without parameters that allows you to use the Constellation cache to preload the country data, while `D_pyStateList` is a dynamic data page that requires a country parameter as a filter. Since the data is hierarchical, it is displayed using the Tree control provided by the Constellation Design System. Using the Card container, the Tree is wrapped in a header and footer that can be styled later.

NOTE:

- For more information on the Tree control, see Tree.
- For more information on the Card container, see Card.

When the associated node of the Tree is expanded, a list of countries and their states is displayed. In this tutorial, the states are displayed only for Australia, Canada, India, Japan, Turkey, and USA.

Update component definition

Update the widget's component definition based on the Card container's API and the widget's data requirements. This decides the configuration options provided to the application builders who use the widget in App Studio.

To allow only custom header text, replace the `properties` array in the `config.json` file with the following code:

```
"properties": [
 {
  "name": "headerText",
  "label": "Header text",
  "format": "TEXT",
  "defaultValue": "@L Country and State Navigator"
 }
]
```

©2025 Pegasystems Inc

Since we do not want the DATASOURCE to be configured by the application builder, add the configuration for the `D_pyCountryList` datasource to the `defaultConfig` so that it is available to the widget at run time and the data is pre-cached by Constellation. Replace the `defaultConfig` object in the `config.json` file with the following code:

```
"defaultConfig": {
  "target": "primary",
  "countryList": {
    "source": "@DATASOURCE D_pyCountryList.pxResults",
    "fields": {
      "id": "@P .id",
      "name": "@P .name",
      "value": "@P .value",
      "objectClass": "@P .objectClass",
    }
  },
}
```

The `@L` notation defines text that should be localized from the default English text to the supported languages. To support localized values, create a new file called `localizations.json` in your component's root directory as shown below.

```
{
   "fields" : {
      "Country and State Navigator": "Country and State Navigator"
   }
}
```

The `localizations.json` file will be picked up by the Constellation DX Component Builder when the widget is bundled and packaged for publication.

The contents of the updated component definition file is as shown below.

PEGA ©2025 Pegasystems Inc

```
{
  "name": "MyCo_MyComponents_MyPageWidget",
  "label": "My Page Widget",
  "description": "My Page Widget",
  "organization": "MyCo",
  "version": "01.01.01",
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyPageWidget",
  "type": "Widget",
  "subtype": "PAGE",
  "properties": [
    {
      "name": "headerText",
      "label": "Header text",
      "format": "TEXT",
      "defaultValue": "@L Country and State Navigator"
    }
  ],
  "defaultConfig": {
    "target": "primary",
    "countryList": {
      "source": "@DATASOURCE D_pyCountryList.pxResults",
      "fields": {
        "name": "@P .pyFieldValue",
        "value": "@P .pyLocalizedValue"
      }
    },
  }
}
```

©2025 Pegasystems Inc

Change the component

Update the widget component's code to use the following:

- The Card container and Tree control for styling and displaying the country data.
- The PCore getDataApiUtils() API to dynamically fetch the state data for a selected country.

Use the `useState` React hook by adding the following code to the beginning of the `index.tsx` file:

```
import { useState } from 'react';
```

Replace `import { AppAnnouncement as PegaAppAnnouncement } from '@pega/cosmos-react-work';` in the `index.tsx` file with the following code:

```
import {
  Card,
  CardHeader,
  CardContent,
  Text,
  StandardTree,
  treeHelpers
} from '@pega/cosmos-react-core'; // ◻
```

The code is described below:

◻ - The Card, CardHeader, CardContent, Text, and StandardTree components and the treeHelpers utility library are imported. For more information, see Card, Text, StandardTree, and treeHelpers.

The StandardTree component and the treeHelpers utility library are used to populate the tree with the data fetched from the data source.

©2025 Pegasystems Inc

The Tree is populated with the country data and then dynamically loads the state data for the selected country at run time.

The `datasource` property defined in the component definition lets us access the country data by pre-fetching it and caching it in the local Redux Store. To use this, we must iterate the `D_pyCountryList.pxResults` @DATASOURCE, adding a new node to the Tree for each country.

The component definition also defines the `countryList` property. The country data is pre-populated into the `source` array when the widget is instantiated. For this tutorial, since we have picked a default data source shipped with Pega, the data structure of each country is already defined. `value` denotes the long name for a country, while `name` denotes the country code.

Add the following code before the `export default function MyCoMyComponentsMyPageWidget(props) {….}` section in the `index.tsx` file.

```
const createCountryNodes = (countryList) => {
  const isParent = true;
  let nodes = [];
  nodes = countryList.source?.map((country) => {
    return {
      id: country.name,
      label: country.value,
      icon: isParent ? 'folder-solid' : 'document-solid',
      nodes: isParent ? [] : undefined
    }
  });
  return nodes;
};
```

The getStates function demonstrates the use of the PCore.getDataApiUtils() API to dynamically populate the child nodes for each country tree node when the tree node is

PEGA ©2025 Pegasystems Inc

selected. Fetching the state data can be handled by the `onNodeClick` handler as seen in the code for the Live Demo of the StandardTree. For more information on the getDataApiUtils() API, see getDataApiUtils().

The getStates function fetches the state data for a country. Add the following code into the `export default function MyCoMyComponentsMyPageWidget(props) {….}` section in the `index.tsx` file.

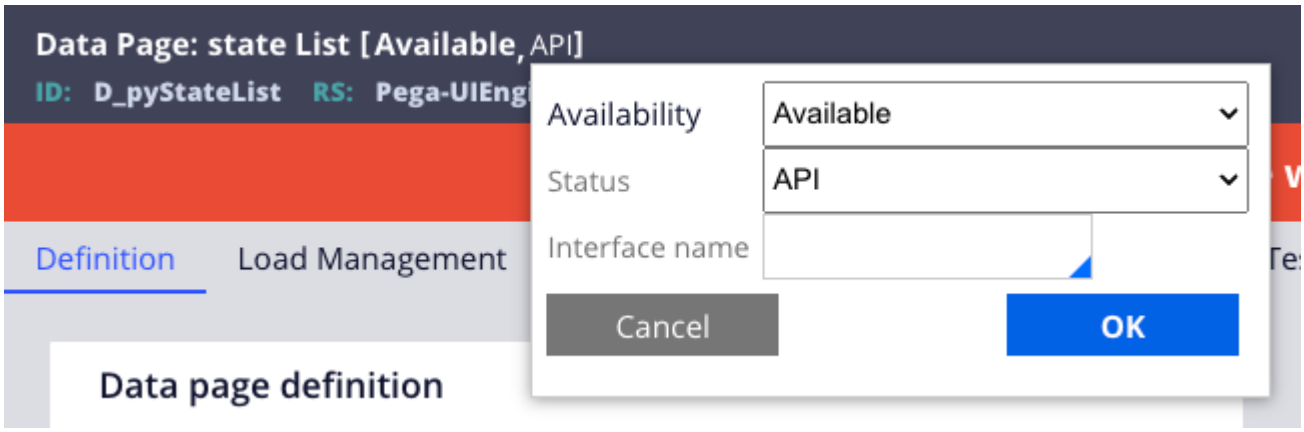
```
const {
   headerText,
   countryList
 } = props;

 const [currentNodeId, setCurrentNodeId] = useState();
 const [allNodes, setAllNodes] = useState(createCountryNodes(countryList));

 const getStates = (id) => {
   PCore
     .getDataApiUtils()
     .getData(
       'D_pyStateList',
       { // ◻
       dataViewParameters: { pyCountry: id }
     })
     .then((response) => {
       const nodes = response.data?.data.map((state) => { // ◻
         const isParent = false;
         return {
           id: state.pyStateCode,
           label: state.pyLabel,
           icon: isParent ? 'folder-solid' : 'document-solid',
           nodes: isParent ? [] : undefined
         }
```

©2025 Pegasystems Inc

```
      });
      setAllNodes(tree => { // 
        return treeHelpers.mapNode(tree, id, node => {
          return {
            ...node,
            loading: false,
            nodes: [...(node.nodes ?? []), ...nodes]
          };
        });
      });
    })
    .catch((e) => { // 
      // Handle error
    })
  };
```

The code is described below:

 - getData is an asynchronous API that returns the data page response as a promise. The `dataViewParameters` object corresponds to the name of the parameter of the data page. To access a data page using ConstellationJS APIs, set the **Status** of the data page to **API** as shown below.



©2025 Pegasystems Inc

Status of a data page

□ - The data structure is pre-defined by the data pages and can return both list based data structures as a Javascript array and non-list based data structures as a standard object. Here, the response contains a data object that contains a data array. When not returning an array, the data is returned directly within the first data object (`response.data.pyOperatorInfo`). The new `StandardTreeNode` object mapping from the response data is returned into the `StandardTreeNode` API.

□ - After the `StandardTreeNode` array is created, the `treeHelpers` module populates the `StandardTree` component.

□ - Handles data fetching errors.

Replace the return function of the `index.tsx` file with the following code:

```
return (
  <StyledMyCoMyComponentsMyPageWidgetWrapper>
    <Card>
      <CardHeader>
        <Text variant='h2'>{headerText}</Text>
      </CardHeader>
      <CardContent>
        <StandardTree
          lined={false}
          currentNodeId={currentNodeId}
          nodes={allNodes}
          onNodeClick={id => { // □
            const clickedNode = treeHelpers.getNode(allNodes, id);
            setCurrentNodeId(id);
            if (!clickedNode?.nodes) return;
            setAllNodes(tree =>
              treeHelpers.mapNode(tree, id, node => {
```

©2025 Pegasystems Inc

```
            return {
              ...node,
              expanded: !node.expanded,
              loading: node.nodes?.length === 0
            };
          })
        );
        if (clickedNode?.nodes?.length > 0) return;
        getStates(id);
      }}
    />
  </CardContent>
 </Card>
</StyledMyCoMyComponentsMyPageWidgetWrapper>
);
```

- The `onNodeClick` handler is mapped directly to the `treeHelpers.getNode` function and this populates the states in the tree.

> **NOTE:**
>
> - If the node of the tree is collapsed, the state data does not need to be fetched again since the country and state data is static.
> - This component has not been production tested or hardened and is intended only for the purpose of this tutorial.

The contents of the updated `index.tsx` file is as shown below.

```
import { useState } from 'react';
import PropTypes from 'prop-types';
import {
  Configuration,
```

©2025 Pegasystems Inc

```
  Text,
  Card,
  CardHeader,
  CardContent,
  StandardTree,
  treeHelpers
} from '@pega/cosmos-react-core';

const createCountryNodes = (countryList) => {
  const isParent = true;
  let nodes = [];
  nodes = countryList.source?.map((country) => {
    return {
      id: country.name,
      label: country.value,
      icon: isParent ? 'folder-solid' : 'document-solid',
      nodes: isParent ? [] : undefined
    }
  });
  return nodes;
};

export default function MyCoMyComponentsMyPageWidget(props) {
  const {
    headerText,
    countryList
  } = props;

  const [currentNodeId, setCurrentNodeId] = useState();
  const [allNodes, setAllNodes] = useState(createCountryNodes(countryList));

  const getStates = (id) => {
```

©2025 Pegasystems Inc

```
if (id === undefined || id === '')
  return;

// eslint-disable-next-line no-undef
PCore.getDataApiUtils().getData('D_pyStateList', {
  dataViewParameters: { pyCountry: id }
})
.then((response) => {
  const nodes = response.data.data!==null ?
    response.data?.data.map((state) => {
      const isParent = false;
      return {
        id: state.pyStateCode,
        label: state.pyLabel,
        icon: isParent ? 'folder-solid' : 'document-solid',
        nodes: isParent ? [] : undefined
      }
    })
  :[];
  setAllNodes(tree => {
    return treeHelpers.mapNode(tree, id, node => {
      return {
        ...node,
        loading: false,
        nodes: [...(node.nodes ?? []), ...nodes]
      };
    });
  });
})
.catch((e) => {
  // Simply log the error in this example
  // eslint-disable-next-line no-console
```

©2025 Pegasystems Inc

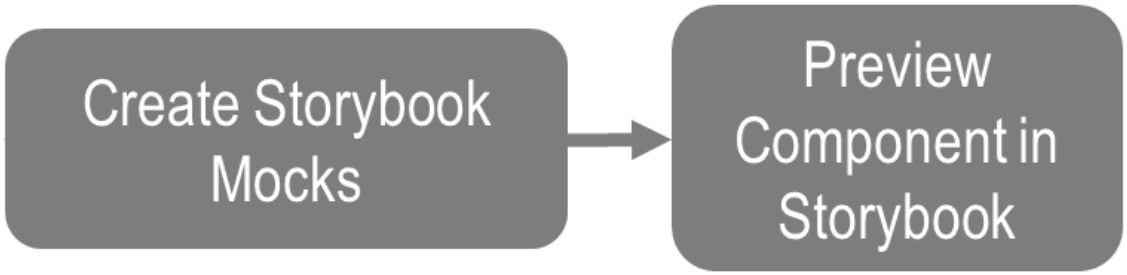
```
    console.error(e);
  })
};

return (
  <Card>
    <CardHeader>
      <Text variant='h2'>{headerText}</Text>
    </CardHeader>
    <CardContent>
      <StandardTree
        lined={false}
        currentNodeId={currentNodeId}
        nodes={allNodes}
        onNodeClick={id => {
          const clickedNode = treeHelpers.getNode(allNodes, id);
          setCurrentNodeId(id);
          if (!clickedNode?.nodes) return;
          setAllNodes(tree =>
            treeHelpers.mapNode(tree, id, node => {
              return {
                ...node,
                expanded: !node.expanded,
                loading: node.nodes?.length === 0
              };
            })
          );
          if (clickedNode?.nodes?.length > 0) return;
          getStates(id);
        }}
      />
    </CardContent>
```

©2025 Pegasystems Inc

```
    </Card>
  );
}

MyCoMyComponentsMyPageWidget.defaultProps = {
  headerText: '',
  countryList: {}
};

MyCoMyComponentsMyPageWidget.propTypes = {
  headerText: PropTypes.string,
  countryList: PropTypes.instanceOf(Object),
};
```

Mocking and testing the widget with a Storybook

Use a Storybook to preview the widget and perform unit testing on it by creating mock country data.



Creating Storybook mocks and previewing component in Storybook

To create a Story in the Storybook, you must write test mocks for the ConstellationJS PCore and PConnect APIs. You must also update the widget to add support for multiple Regions.

©2025 Pegasystems Inc

When you bootstrap a new project using the Constellation DX Component Builder, the root of the new project's workspace folder contains a `.storybook` folder. This provides you with a Constellation-specific initialization of the Storybook for the project.

Storybook folder

The Story you create is automatically decorated by the `preview.tsx` file and a Pega theme is applied to the Storybook.

1. Create a new file called `pagewidget.stories.tsx` in your component root directory as shown below.

```
import MyCoMyComponentsMyPageWidget from './index.tsx';

export default {
  title: 'MyCoMyComponentsMyPageWidget',
  component: MyCoMyComponentsMyPageWidget
};

export const baseMyCoMyComponentsMyPageWidget = () => {

  const props = {
  };

  return (
    <>
      <MyCoMyComponentsMyPageWidget {...props} />
    </>
```

©2025 Pegasystems Inc

```
  );
};
```

> **NOTE:** If you created your widget with a different name, ensure that you update the name of the component in the `pagewidget.stories.tsx` file.

2. Set up the mock properties for the properties that are passed as seen in the component definition. The `headerText` property from `properties[]` and the `countryList` object from `defaultConfig` are passed into the widget as `props`. In the `pagewidget.stories.tsx` file, replace `const props = { };` with the following code:

```
  const props = {
    countryList: {
      "source": [
        {
          "objectClass": "Rule-Obj-FieldValue",
          "value": "United Kingdom",
          "id": "RULE-OBJ-FIELDVALUE @BASECLASS PYCOUNTRYCODE!GBR #20
180713T132223.211 GMT",
          "name": "GBR"
        },
        {
          "objectClass": "Rule-Obj-FieldValue",
          "value": "United States",
          "id": "RULE-OBJ-FIELDVALUE @BASECLASS PYCOUNTRYCODE!USA #201
80713T132225.795 GMT",
          "name": "USA"
        },
      ],
```

©2025 Pegasystems Inc

```
  },
  headerText: 'Country and State Navigator'
};
```

This establishes the country data.

3. Add support for fetching the list of states based on the country selected. For this mock data, let us consider only two states. In the `pagewidget.stories.tsx` file, insert the following code before `export const baseMyCoMyComponentsMyPageWidget = () => {...`.

```
if (!window.PCore) {
  window.PCore = {};
}

window.PCore.getDataApiUtils = (() => {
  return {
    getData: (dataView, parameters, options) => {
      return new Promise((resolve) => {
        if (dataView === 'D_pyStateList' && parameters.dataViewParameters.pyCou
ntry === 'USA') {
          resolve({
            data: {
              "data": [
                {
                  "pyLabel": "Alabama",
                  "pyStateCode": "AL"
                },
                {
                  "pyLabel": "Alaska",
                  "pyStateCode": "AK"
                }
              ]
```

©2025 Pegasystems Inc

```
          }
        })
      } else {
        resolve({
          data: {
            "data": []
          }
        })
      }
    })
  }
}
});
```

Result:

The mock data and test (`baseMyCoMyComponentsMyPageWidget`) are now ready. You can now run the Story in the Storybook to view the fully mocked widget component.

Styling the widget

Override the out-of-the-box theme applied to the widget by styling it using properties from the Constellation Design System Theme.

Using Styled Components

The `styles.ts` file created with the widget contains the following structure:

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

export default styled.div(() => {
  return css`
    margin: 8px 0;
```

©2025 Pegasystems Inc

```
  `;
});
```

Replace the contents of the `styles.ts` file with the following code:

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

export const StyledCard = styled.article(() => {
  return css`
    width: 25rem;
  `;
});
```

Override the width of the Card component by performing the following steps:

1. Import the StyledCard from the `styles.ts` file.

   ```
   import { StyledCard } from './styles'
   ```

2. Apply the new styling to the Card.

   ```
   ...
      <Card as={StyledCard}>
   ...
   ```

Using the Constellation Design System Theme

You can override specific parts of the Theme prop with CSS overrides. You must pass the themes override object in a Configuration component, with the overrides applying to any children that are wrapped by the Configuration.

NOTE:

- For more information on the Theme property, see Theme.

©2025 Pegasystems Inc

- For more information on the Configuration component, see Configuration.

Let us wrap our component with a new Configuration that overrides the parts of the Portal Theme where our widget is being hosted. This allows us to perform style overrides at the component level as required.

NOTE: Typically, we will not need to override component styles for presentation components from the Constellation Design System since Constellation Theming and App Studio are tightly coupled. As a best practice, do not override the default behaviour when building components based on the Constellation Design System so that the component can be reused across different App Studio authored applications while dynamically adopting the styles of the Portal it is displayed in.

Update the `index.tsx` file with the following code:

```
// ◻
const theme = {
  components: {
    card: {
      'border-radius': '0.5rem',
      background: '#71baff'
    }
  }
};

return (
  // ◻
  <Configuration theme={ theme }>
    <Card as={StyledCard}>
```

©2025 Pegasystems Inc

```
      <CardHeader>
        <Text variant='h2'>{headerText}</Text>
      </CardHeader>
      <CardContent>
        <StandardTree
          lined={false}
          currentNodeId={currentNodeId}
          nodes={allNodes}
          onNodeClick={id => {
            const clickedNode = treeHelpers.getNode(allNodes, id);
            setCurrentNodeId(id);
            if (!clickedNode?.nodes) return;
            setAllNodes(tree =>
              treeHelpers.mapNode(tree, id, node => {
                return {
                  ...node,
                  expanded: !node.expanded,
                  loading: node.nodes?.length === 0
                };
              })
            );
            if (clickedNode?.nodes?.length > 0) return;
            getStates(id);
          }}
        />
      </CardContent>
    </Card>
  </Configuration>
);
```

The code is described below:

©2025 Pegasystems Inc

- A theme object is created that overrides the specific theme properties. It contains several properties that can be referenced directly.

©2025 Pegasystems Inc

```
{
  ⌄ base: {
    > colors: { /*...*/ },
    > palette: { /*...*/ },
    'font-family': '"Open Sans", sans-serif',
    'font-size': '0.875rem',
    'font-scale': 'linear',
    > 'font-weight': { /*...*/ },
    'line-height': 'normal',
    scale: 1,
    'border-radius': '0.5rem',
    spacing: '0.5rem',
    > 'hit-area': { /*...*/ },
    'custom-scrollbar': true,
    > animation: { /*...*/ },
    > transparency: { /*...*/ },
    'disabled-opacity': 'base.transparency.transparent-4',
    > shadow: { /*...*/ },
    > 'z-index': { /*...*/ },
    > breakpoints: { /*...*/ },
    > 'content-width': { /*...*/ }
  },
  ⌄ components: {
    > 'app-shell': { /*...*/ },
    > avatar: { /*...*/ },
    > badges: { /*...*/ },
    > banner: { /*...*/ },
    > button: { /*...*/ },
    > card: { /*...*/ },
    > chat: { /*...*/ },
    > checkbox: { /*...*/ },
    > 'form-control': { /*...*/ },
    > 'form-field': { /*...*/ },
    > input: { /*...*/ },
    > 'interaction-timer': { /*...*/ },
    > label: { /*...*/ },
    > link: { /*...*/ },
    > mark: { /*...*/ },
    > progress: { /*...*/ },
    > 'radio-check': { /*...*/ },
    > 'radio-button': { /*...*/ },
    > rating: { /*...*/ },
    > 'search-input': { /*...*/ },
    > select: { /*...*/ },
    > sentiment: { /*...*/ },
    > switch: { /*...*/ },
    > table: { /*...*/ },
    > tabs: { /*...*/ },
    > 'task-manager': { /*...*/ },
```

©2025 Pegasystems Inc

Theme properties

- A new Configuration object wrapping the component is created to override specific properties of the Portal theme for the widget. The theme prop overrides specific theme properties for each component. All components that use the Constellation Design System are wrapped by a Configuration object at the top of the object tree and can be selectively overridden by nested Configuration objects that will then apply those overrides to all of its children.

The contents of the updated files are as shown below.

`config.json`

```
{
  "name": "MyCo_MyComponents_MyPageWidget",
  "label": "My Page Widget",
  "description": "My Page Widget",
  "organization": "MyCo",
  "version": "01.01.01",
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyPageWidget",
  "type": "Widget",
  "subtype": "PAGE",
  "properties": [
    {
      "name": "headerText",
      "label": "Header text",
      "format": "TEXT",
      "defaultValue": "@L Country and State Navigator"
    }
  ],
  "defaultConfig": {
```

©2025 Pegasystems Inc

```
    "target": "primary",
    "countryList": {
      "source": "@DATASOURCE D_pyCountryList.pxResults",
      "fields": {
        "id": "@P .pzInsKey",
        "name": "@P .pyFieldValue",
        "value": "@P .pyLocalizedValue",
        "objectClass": "@P .pxObjClass"
      }
    }
  }
}
```

styles.ts

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

export const StyledCard = styled.article(() => {
  return css`
    width: 25rem;
  `;
});
```

index.tsx

```
import { useState } from 'react';
import PropTypes from 'prop-types';
import {
  Configuration,
  Text,
  Card,
  CardHeader,
  CardContent,
  StandardTree,
```

©2025 Pegasystems Inc

```
  treeHelpers
} from '@pega/cosmos-react-core';

import StyledCard from './styles';

const createCountryNodes = (countryList) => {
  const isParent = true;
  let nodes = [];
  nodes = countryList.source?.map((country) => {
    return {
      id: country.name,
      label: country.value,
      icon: isParent ? 'folder-solid' : 'document-solid',
      nodes: isParent ? [] : undefined
    }
  });
  return nodes;
};

export default function MyCoMyComponentsMyPageWidget(props) {
  const {
    headerText,
    countryList
  } = props;

  const [currentNodeId, setCurrentNodeId] = useState();
  const [allNodes, setAllNodes] = useState(createCountryNodes(countryList));

  const getStates = (id) => {
    if (id === undefined || id === '')
      return;
```

©2025 Pegasystems Inc

```
  // eslint-disable-next-line no-undef
  PCore.getDataApiUtils().getData('D_pyStateList', {
    dataViewParameters: { pyCountry: id }
  })
  .then((response) => {
    const nodes = response.data.data!==null ?
      response.data?.data.map((state) => {
        const isParent = false;
        return {
          id: state.pyStateCode,
          label: state.pyLabel,
          icon: isParent ? 'folder-solid' : 'document-solid',
          nodes: isParent ? [] : undefined
        }
      })
    :[];
    setAllNodes(tree => {
      return treeHelpers.mapNode(tree, id, node => {
        return {
          ...node,
          loading: false,
          nodes: [...(node.nodes ?? []), ...nodes]
        };
      });
    });
  })
  .catch((e) => {
    // Simply log the error in this example
    // eslint-disable-next-line no-console
    console.error(e);
  })
};
```

©2025 Pegasystems Inc

```
const theme = {
  components: {
    card: {
      'border-radius': '0.5rem',
      background: '#71baff'
    }
  }
};

return (
  <Configuration theme={theme}>
    <Card as={StyledCard}>
      <CardHeader>
        <Text variant='h2'>{headerText}</Text>
      </CardHeader>
      <CardContent>
        <StandardTree
          lined={false}
          currentNodeId={currentNodeId}
          nodes={allNodes}
          onNodeClick={id => {
            const clickedNode = treeHelpers.getNode(allNodes, id);
            setCurrentNodeId(id);
            if (!clickedNode?.nodes) return;
            setAllNodes(tree =>
              treeHelpers.mapNode(tree, id, node => {
                return {
                  ...node,
                  expanded: !node.expanded,
                  loading: node.nodes?.length === 0
                };
```

©2025 Pegasystems Inc

```
            })
           );
           if (clickedNode?.nodes?.length > 0) return;
           getStates(id);
          }}
         />
       </CardContent>
     </Card>
   </Configuration>
  );
}

MyCoMyComponentsMyPageWidget.defaultProps = {
  headerText: '',
  countryList: {}
};

MyCoMyComponentsMyPageWidget.propTypes = {
  headerText: PropTypes.string,
  countryList: PropTypes.instanceOf(Object),
};
```

`pagewidget.stories.tsx`

```
import MyCoMyComponentsMyPageWidget from './index.tsx';

export default {
  title: 'MyCoMyComponentsMyPageWidget',
  component: MyCoMyComponentsMyPageWidget
};

if (!window.PCore) {
  window.PCore = {};
```

©2025 Pegasystems Inc

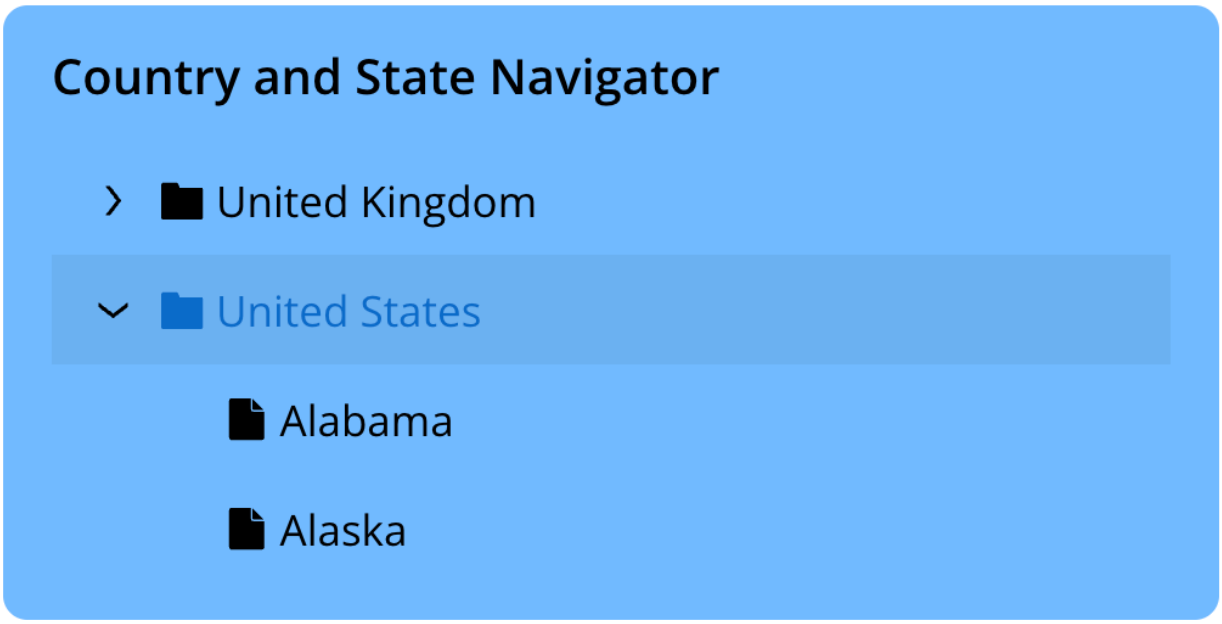
```
}

window.PCore.getDataApiUtils = (() => {
  return {
    getData: (dataView, parameters, options) => {
      return new Promise((resolve) => {
        if (dataView === 'D_pyStateList' && parameters.dataViewParameters.pyCountry
=== 'USA') {
          resolve(
            {
              data: {
                "data": [
                  {
                    "pyLabel": "Alabama",
                    "pyStateCode": "AL"
                  },
                  {
                    "pyLabel": "Alaska",
                    "pyStateCode": "AK"
                  }
                ]
              }
            })
        } else {
          resolve(
            {
              data: {
                "data": []
              }
            })
        }
      })
```

©2025 Pegasystems Inc

```
    }
  }
});
export const baseMyCoMyComponentsMyPageWidget = () => {

  const props = {
    countryList: {
      "source": [
        {
          "objectClass": "Rule-Obj-FieldValue",
          "value": "United Kingdom",
          "id": "RULE-OBJ-FIELDVALUE @BASECLASS PYCOUNTRYCODE!GBR #20180713T1
32223.211 GMT",
          "name": "GBR"
        },
        {
          "objectClass": "Rule-Obj-FieldValue",
          "value": "United States",
          "id": "RULE-OBJ-FIELDVALUE @BASECLASS PYCOUNTRYCODE!USA #20180713T1
32225.795 GMT",
          "name": "USA"
        },
      ],
    },
    headerText: 'Country and State Navigator'
  };
  return (
    <>
      <MyCoMyComponentsMyPageWidget {...props} />
    </>
  );
};
```

©2025 Pegasystems Inc

The Storybook story for the widget is displayed as shown below.



Widget Story

Publishing the widget

Perform integration testing on the widget by publishing it to Pega Infinity.

Authenticating and publishing the component

To use a component in App Studio Authoring, you must publish it. Publishing the component bundles all the assets together and uses Webpack to create a gzipped

©2025 Pegasystems Inc

package. The Constellation DX Component Builder then connects with Pega Infinity and then uses a Pega service to create or update an instance of Rule-UI-Component. A declare trigger on Rule-UI-Component performs the onward publish to the Constellation Service. For more information about the Constellation Service, please see Constellation UI Service.

Authenticate your access to the Pega Infinity Server and publish the component.

For more information, see:

- Authenticating Pega Platform Server user access for standalone Constellation DX components
- Publishing standalone Constellation DX components

> **NOTE:** Your authentication might expire while publishing the component. Run the `npm run authenticate` command again to get a new token.

The widget is now ready to be tested in App Studio.

Using the widget in App Studio

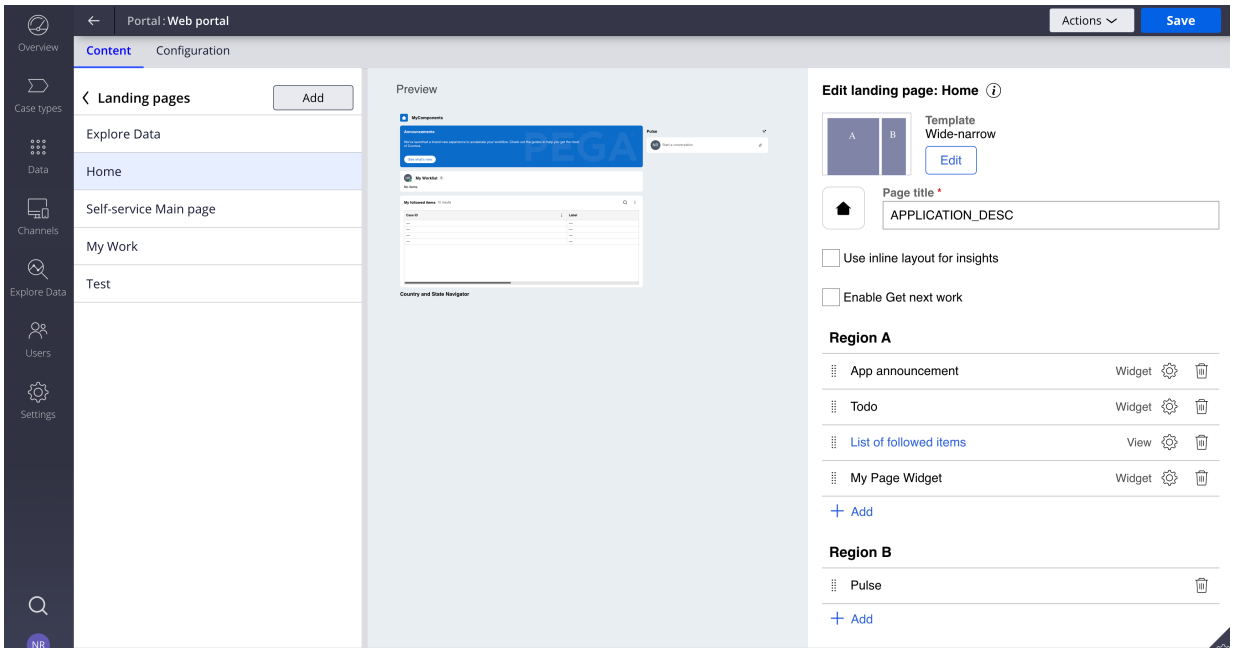
Learn how to use the widget in App Studio.

Testing the component

©2025 Pegasystems Inc

After the widget is published successfully, add the widget to the Home landing page that exists in all Pega portals.

1. In App Studio, click **Channels** > **Web portal** > **Landing pages** > **Home**.



Adding a widget to the Home landing page

2. In **Region A**, click + **Add**, click **Widget**, and select the **My Page Widget** checkbox.

> **Result:**
>
> The widget is ready to be previewed.

Layout Template component

Use a layout template to create layouts that cannot be achieved using the layout templates provided by App Studio.

The top-level component of every view is a layout template. Every template component has one or more Regions. Regions are containers for other components and are used

©2025 Pegasystems Inc

to organize groups of components together. Application Builders use the UI Authoring tool to add Fields, Views, and other components to Regions at design time.

You can create three types of layout templates using the Constellation DX Component Builder:

- Details Layout Template - This template controls the layout of the Details tab in the Case View. This is read-only by design.
- Form Layout Template - This template controls the layout of Views that will be used for Assignment processing.
- Page Layout Template - This template controls the layout of Landing Pages.

Learn about the following aspects of a layout template:

- **Creating a layout template**
- **Reviewing the structure of a layout template**
- **Editing the layout template**
- **Mocking and testing the layout template with a Storybook**
- **Publishing the layout template**
- **Using the layout template in App Studio**

Creating a layout template

Learn how to create a layout template.

©2025 Pegasystems Inc

Creating a component

To create a new Details layout template, follow the procedure below:

1. Open your preferred terminal tool, navigate to your project, and enter `npm run create`.

   > **NOTE:** Ensure your project is initialized before running this command. For more information on initializing a project, see Initializing a project.

2. Create the Details layout template with the following details:
   - Component name: *MyDetailsTemplate*
   - Component label: *My Details Template*
   - Component version: *01.01.01*
   - Library name: *MyComponents*
   - Supported applications: Do not provide any supported applications, so that the component is available for all applications by default.
   - Component type: *Layout template*

©2025 Pegasystems Inc

```
? Enter type of component
  1) Field
  2) Layout template
  3) Widget
  Answer:
```

Component type

- Component sub-type: *DETAILS*

```
? Enter subtype of component
  1) DETAILS
  2) DETAILS-region
  3) FORM
  4) FORM-region
  5) PAGE
  Answer:
```

Component sub-type

- Component description: *My Details Template*

```
> mycomponents@1.0.0 create
> custom-dx-components create

DX Component Builder v2.0.9
? Enter component name (required) MyDetailsTemplate
? Enter component label for display (required) My Details Template
? Enter component version 01.01.01
? Enter library name (required) MyComponents
? Please enter the application names to be supported (comma-separa
ted). Keep empty for all
applications
? Enter type of component Layout template
? Enter subtype of component DETAILS
? Enter description for the component My Details Template
```

Creating Details layout template

The `MyCo_MyComponents_MyDetailsTemplate` component is created in the `<your project folder>/src/components/MyCo_MyComponents_MyDetailsTemplate` path as seen below:

©2025 Pegasystems Inc

```
src
    components
        MyCo_MyComponents_MyDetailsTemplate
            index.tsx
            utils.ts
            config.json
            styles.ts
            demo.stories.tsx
            mock.stories.ts
            OneColumnDetails.svg
            StatusWork.tsx
```

> **NOTE:** The list of files above is indicative. You may see a slightly different list of files depending on the version of the Constellation DX Component Builder you are running.

Reviewing the structure of a layout template

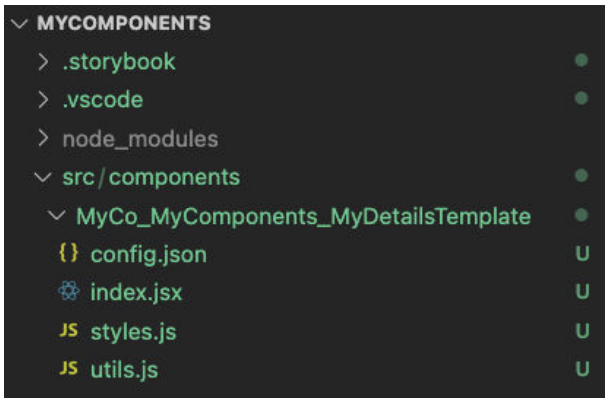
Learn about the auto-generated files that are part of a layout template.

After creating the layout template, navigate to its source and you will see several auto-generated files within the component.

Navigate to the `MyCo_MyComponents_MyDetailsTemplate` folder and delete all files except the following:

- `config.json`
- `styles.ts`
- `utils.ts`
- `index.tsx`

©2025 Pegasystems Inc



Structure of a layout template

The layout template contains the following files:

- `config.json`
- `styles.ts`
- `utils.ts`
- `index.tsx`

`config.json`

The `config.json` component definition file drives the App Studio experience for the component. It shows how the component is displayed in App Studio. As a component developer, you can control the configuration data captured when the component is used by App Studio. The data captured is passed into your component as `props` at run time. For more information on the component definition schema, see Component definition.

The contents of the file is as shown below:

```
{
  "name": "MyCo_MyComponents_MyDetailsTemplate",
  "label": "My Details Template",
  "description": "My Details Template",
  "organization": "MyCo",
  "version": "01.01.01",
```

©2025 Pegasystems Inc

```
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyDetailsTemplate",
  "type": "Template",
  "subtype": "DETAILS",
  "icon": "OneColumnDetails.svg",
  "properties": [
    {
      "name": "showHighlightedData", //
      "label": "Enable highlighted fields",
      "format": "BOOLEAN"
    },
    {
      "name": "highlightedData", //
      "label": "Highlighted fields",
      "format": "CONTENTPICKER",
      "addTypeList": ["Fields"],
      "visibility": "$this.showHighlightedData = true",
      "saveToConfig": true
    },
    {
      "name": "A", //
      "label": "Region A",
      "format": "CONTENTPICKER",
      "addTypeList": ["Fields", "Views"],
      "allowCreatingGroup": true
    }
  ]
}
```

The file contains a `properties` array with three properties - `showHighlightedData`, `highlightedData`, and `A`.

©2025 Pegasystems Inc

☐ - `showHighlightedData` is a Boolean field that is used to determine if the highlighted fields data should be displayed.

☐ - The `highlightedData` field contains an array of fields that an App Studio user can add as children. This section allows the template to format these fields differently as compared to normal Regions.

☐ - `A` is an example of a Region that allows the App Studio author to add both fields and views.

The `type` and `sub-type` are also used by App Studio to present a list of components relevant to your current authoring context.

For more information on the `config.json` file, see Component definition.

`styles.ts`

The `styles.ts` file uses the styled components library to manage CSS in JavaScript. The wrapper function is already created and imported. The `styles.ts` file provides extensive capabilities to extend and override the styles of the Constellation application Portal that hosts your component at run time.

For more information on the styled components library, see styled components.

The contents of the `styles.ts` file is as shown below:

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

export default styled.div(( theme ) => {
  return css`
    margin: 8px 0;
  `;
});
```

©2025 Pegasystems Inc

You can adopt the theme of the Constellation application Portal configured in App Studio by using the theme object available to you through a Configuration wrapper component. For more information on using themes for the Constellation application Portal, see Branding your application.

`utils.ts`

This utility library provides several useful functions that can be used in Constellation DX components. The getAllFields function shown below iterates through all our Region's children components and resolves the config metadata into the actual data passed in as props. Any side effects that are required to apply changes in field configuration are captured in the `utils.ts` file.

The contents of the `utils.ts` file is as shown below:

```
/**
 * Given the PConnect object of a Template component, retrieve the children
 * metadata of all regions.
 * @param {Function} pConnect PConnect of a Template component.
 */
export function getAllFields(pConnect) {
  const metadata = pConnect().getRawMetadata(); //
  if (!metadata.children) {
    return [];
  }

  let allFields = [];

  const makeField = (f) => ({ //
    ...pConnect().resolveConfigProps(f.config),
    type: f.type
  });
```

©2025 Pegasystems Inc

```
  const hasRegions = !!metadata.children[0]?.children;
  if (hasRegions) {
    allFields = metadata.children.map((region) =>
      region.children.map((field) => {
        if (field.config?.value === '@P .pyStatusWork') { //
          field.type = 'TextInput';
          field.config.displayAsStatus = true;
          return pConnect().createComponent(field); //
        }
        return makeField(field);
      })
    );
  } else {
    allFields = metadata.children.map(makeField);
  }
  return allFields;
}
```

The code is described below:

- The getRawMetadata() function is used to access the raw metadata object of the `children` array.

- To ensure that our Region's components are rendered as read-only and with labels on the left as defined in the Constellation Design System, we must resolve the metadata into an object that has the real data. The makeField() function resolves the metadata for all `children`.

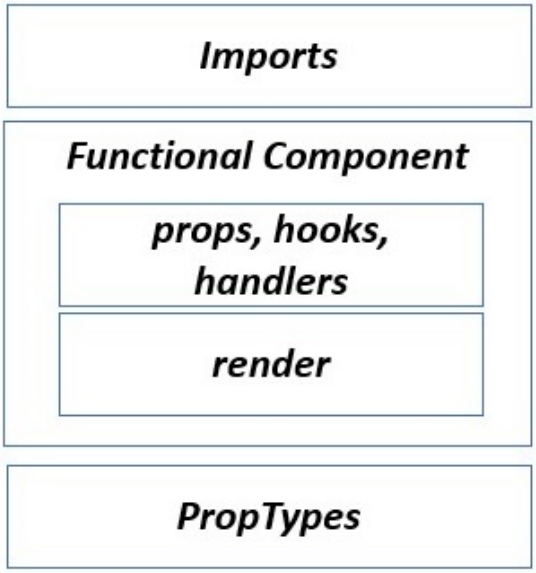
- `pyStatusWork` requires special handling as it is displayed as it is displayed as a Status Badge for applications based on the Constellation Design System. The `displayAsStatus` property can be used by any Constellation JavaScript based renderer to control the display of Work Status. For more information on a Status Badge, see Status Badge.

©2025 Pegasystems Inc

□ - Since the `TextInput` field type with the `displayAsStatus` property set is always handled directly in the `TextInput` component, you can create the component directly using the registered React component creator function.

index.tsx

Typically, a functional DX component is structured as shown below.



Structure of an index.tsx file

For the Details Layout Template, the `index.tsx` file containing React JSX code with the component's name is generated from a template.

Let's look at each code block within the `index.tsx` file.

Imports

```
import { Fragment } from 'react'; //□
import PropTypes from 'prop-types'; //□
```

©2025 Pegasystems Inc

```
import { Grid, Flex, FieldGroup } from '@pega/cosmos-react-core'; //
import StyledMyCoMyComponentsMyDetailsTemplateWrapper from './styles'; //

import { getAllFields } from './utils'; //
```

The code is described below:

 - Our dependencies are imported from the React ecosystem. This convention is used because React uses `prop-types` for run-time type checking. For more information on `prop-types`, see prop-types.

 - The Constellation Design System is used to render Constellation DX components. For this layout template, we use the Grid, Flex, and FieldGroup components. For more information on the Constellation Design System, see Pega Constellation.

 - This is the automatically generated `styles.ts` file as seen above.

 - The getAllFields function is imported from the `utils.ts` file as seen above.

Functional Component - props, hooks, and handlers

```
export default function MyCoMyComponentsMyDetailsTemplate(props) { //
  const { getPConnect, children, showLabel, label, showHighlightedData } = props; //
  const propsToUse = { showLabel, label, ...getPConnect().getInheritedProps() }; //

  getAllFields(getPConnect); //

  const numRegions = children.length;
```

The code is described below:

 - As seen in modern React component development, we use a "function with hooks" development pattern, not the more traditional classed-based design pattern.

 - The `props` are destructured into local variables.

©2025 Pegasystems Inc

□ - In order to drive rendering later, we will merge any inherited props that parents may have set using the setInheritedProp(propName, value) API with our local configuration override for `label` and `showLabel`.

□ - The getAllFields function rebuilds the `children` array into resolved children.

handling Highlighted Fields

```
 // Set up highlighted data to pass in return if is set to show, need raw metadata to p
ass to createComponent
 let highlightedDataArr = []; //□

 if (showHighlightedData) { //□
  const { highlightedData = [] } = getPConnect().getRawMetadata().config;
  highlightedDataArr = highlightedData.map((field) => {
   field.config.displayMode = 'STACKED_LARGE_VAL';

   // Mark as status display when using pyStatusWork
   if (field.config.value === '@P .pyStatusWork') { //□
    field.type = 'TextInput';
    field.config.displayAsStatus = true;
   }

   return getPConnect().createComponent(field); //□
  });
 }
```

The code is described below:

□ - The display is fixed for each configured Field in the `highlightedData` array. This ensures that the `displayMode` emphasizes the Field with a large bold font and stacks the label on top. The highlighted data fields are selected in App Studio based on our

©2025 Pegasystems Inc

definition in the component definition file. For more information on the component definition schema, see config.json.

☐ - If the `showHighlightedData` config is selected, the `highlightedData` array is obtained from the raw metadata config object.

☐ - We have followed the format used for highlighted data in the standard Details Template, setting the `displayMode` and formatting `pyStatusWork` to display as a Status Badge.

☐ - The components are created using the component creator function with the updated field metadata. Since the renderer is React, new instances of the React DX components are created that map to each field type.

For more information on the structure of the Constellation Core configuration data partially illustrated above in the field data model, see Structure of Constellation Core configuration data.

Functional Component - render

```
return (
  <StyledMyCoMyComponentsMyDetailsTemplateWrapper>
    <FieldGroup name={propsToUse.showLabel ? propsToUse.label : ''}>
      {showHighlightedData && highlightedDataArr.length > 0 && (
        <Grid container={{ cols:'repeat(${numRegions}, 1fr)', gap: 2, alignItems: 'start',  p
ad: [0, 0, 2, 0] }}
          data-testid={`highlighted-column-count-${numRegions}`}>
          {highlightedDataArr.map((child, i) => (
            <Fragment key={`hf-${i + 1}`}>{child}</Fragment>
          ))}
        </Grid>
      )}
      <Grid container={{ cols:'repeat(${numRegions}, 1fr)', alignItems: 'start', colGap: 1 }
} data-testid={`column-count-${numRegions}`} >
```

©2025 Pegasystems Inc

```
      {children.map((child, i) => (
       <Flex container={{ direction: 'column'}} key={`r-${i + 1}`}>
       {child}
       </Flex>
      ))}
     </Grid>
    </FieldGroup>
  </StyledMyCoMyComponentsMyDetailsTemplateWrapper>
);
```

The code is described below:

The render function uses the presentation components of the Constellation Design System to display the processed props data mapped to appropriate component attributes. The render function displays the React components of the highlighted field (`highlightedDataArr`) first and then the Region component that renders its children using the new Field config.

PropTypes

```
MyCoMyComponentsMyDetailsTemplate.defaultProps = {
  label: undefined,
  showLabel: true,
  showHighlightedData: false
};

MyCoMyComponentsMyDetailsTemplate.propTypes = {
  showLabel: PropTypes.bool,
  label: PropTypes.string,
  getPConnect: PropTypes.func.isRequired,
  template: PropTypes.string.isRequired,
  showHighlightedData: PropTypes.bool
};
```

©2025 Pegasystems Inc

The code is described below:

The default prop values are declared in the PropTypes set-up. If you want the default values to be used by the component, align the default prop values with the defaultConfig {} in the component definition file.

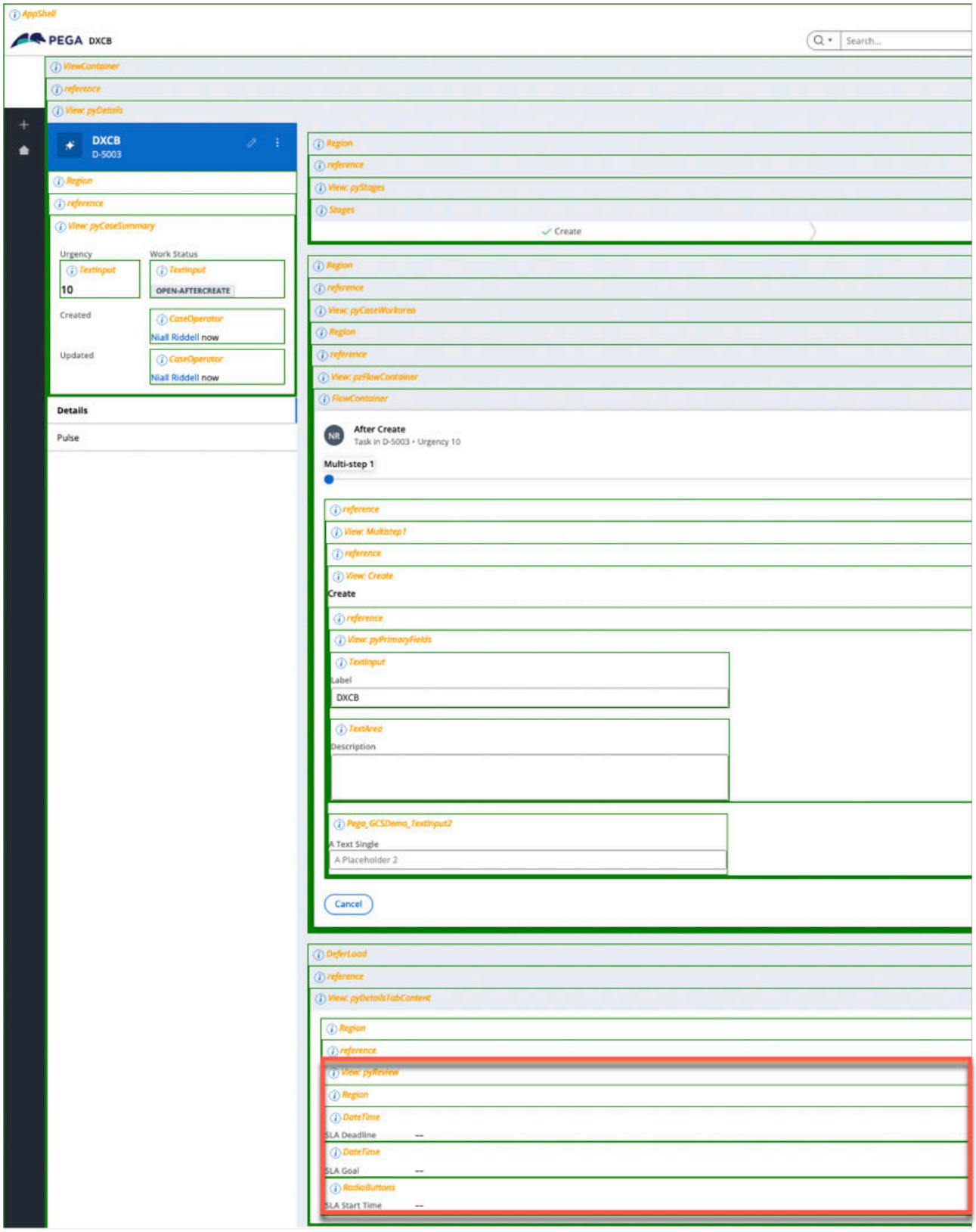
View the component by publishing it. For more information on publishing the component, see Publishing standalone Constellation DX components.

Structure of Constellation Core configuration data

View where the Details layout template is used and the `props` that need to be processed by enabling the debug mode within the Developer tools section of your browser.

The PCore.getDebugger().toggleXRay("true") function will open the debug mode.

©2025 Pegasystems Inc



©2025 Pegasystems Inc

Toggling xRay Debugger

Click ☐ in the *View pyReview* section to get the json view of the unresolved raw config metadata. This view contains the props that will be passed to the Details layout template.

View Metadata

```
{
  "name": "pyReview",  -- name of the metadata component instance
  "type": "View", -- The Constellation Component that will process this metadata
  "config": {
    "template": "Details",  -- The name of the View Template
    "ruleClass": "OE4R5B-DXCB-Work-DXCB",
    "showLabel": true, -- drives label display (see config.json)
    "label": "@L Details", -- @L is locale annotation
    "localeReference": "@LR OE4R5B-DXCB-WORK-DXCB!VIEW!PYREVIEW", -- locale dat
a
    "showHighlightedData": true, -- drives highlighted data display (see config.json)
    "highlightedData": [ -- array of Fields configured for highlighting
      {
        "type": "TextInput",
        "config": {
          "value": "@P .pyID", -- @P annotation indicates that this props value will be use
d
          "label": "@L Case ID",
          "displayMode": "STACKED_LARGE_VAL" -- drives display
        }
      },
      {
        "type": "TextInput",
        "config": {
```

©2025 Pegasystems Inc

```
        "value": "@P .pyStatusWork",
        "label": "@L Work Status",
        "displayMode": "STACKED_LARGE_VAL",
        "displayAsStatus": true -- drives Status Badge display
      }
    }
  ],
  "inheritedProps": [ -- props passed down by parent.
    {
      "prop": "label",
      "value": "@L Details"
    },
    {
      "prop": "showLabel",
      "value": true
    }
  ]
},
"children": [ -- this View has 0-many Regions as children
  {
    "name": "A",
    "type": "Region",
    "children": [  -- each Region has 0 to many children (can be Fields, Views)
      {
        "type": "DateTime",
        "config": {
          "value": "@P .pySLADeadline",
          "label": "@L SLA Deadline"
        }
      },
      {
        "type": "DateTime",
```

©2025 Pegasystems Inc

```
        "config": {
          "value": "@P .pySLAGoal",
          "label": "@L SLA Goal"
        }
      },
      {
        "type": "RadioButtons",
        "config": {
          "value": "@P .pySLAStartTime",
          "label": "@L SLA Start Time",
          "listType": "associated",
          "datasource": "@ASSOCIATED .pySLAStartTime" -- @ASSOCIATED annotation
        }
      }
    ]
  }
],
"classID": "OE4R5B-DXCB-Work-DXCB"
}
```

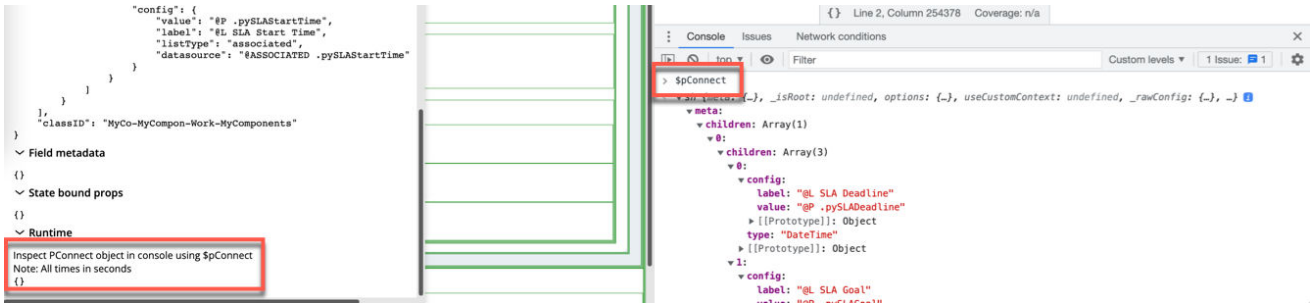
This metadata is derived from the configuration of a component in the Pega App Studio UI. It is used to generate the props data at run time, which is then used to render the component.

A new Constellation DX component gets its props from:

- The Constellation DX components wrapper components which vary depending upon the type and sub-type of your component and the context in which they are used.
- The App Studio configuration of the component by the app builder at design time as defined in the component's component definition file.

©2025 Pegasystems Inc

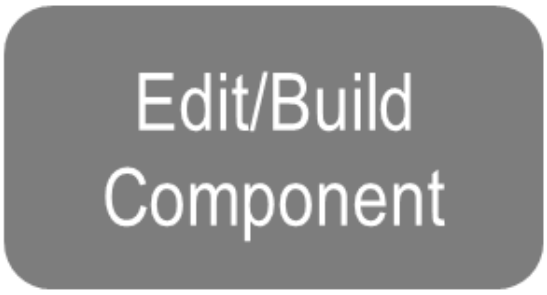
Run-time instantiation of your React component uses ConstellationJS to resolve this metadata into the props in your component. It also uses the children metadata to render the React children components that are passed into the children prop. You can view the run-time ConstellationJS PConnect object generated from this raw metadata by entering `$pConnect` into the console in the toggleXRay debug mode. This object is passed as the getPConnect prop into your component.



Run-time ConstellationJS PConnect object

Editing the layout template

Customize the layout template by updating its auto-generated code.



Editing or building a component

Add another Region in the Details layout template by appending the following code to the `config.json` component definition file:

©2025 Pegasystems Inc

```
  {
    "name": "B",
    "label": "Region B",
    "format": "CONTENTPICKER",
    "addTypeList": ["Fields", "Views"],
    "allowCreatingGroup": true
  }
```

Another Region is now created in the Details layout template and the contents of the updated component definition file is as shown below:

```
{
  "name": "MyCo_MyComponents_MyDetailsTemplate",
  "label": "My Details Template",
  "description": "My Details Template",
  "organization": "MyCo",
  "version": "01.01.01",
  "library": "MyComponents",
  "allowedApplications": [],
  "componentKey": "MyCo_MyComponents_MyDetailsTemplate",
  "type": "Template",
  "subtype": "DETAILS",
  "icon": "OneColumnDetails.svg",
  "properties": [
    {
      "name": "showHighlightedData",
      "label": "Enable highlighted fields",
      "format": "BOOLEAN"
    },
    {
      "name": "highlightedData",
      "label": "Highlighted fields",
      "format": "CONTENTPICKER",
```

©2025 Pegasystems Inc

```
      "addTypeList": ["Fields"],
      "visibility": "$this.showHighlightedData = true",
      "saveToConfig": true
    },
    {
      "name": "A",
      "label": "Region A",
      "format": "CONTENTPICKER",
      "addTypeList": ["Fields", "Views"],
      "allowCreatingGroup": true
    },
    {
      "name": "B",
      "label": "Region B",
      "format": "CONTENTPICKER",
      "addTypeList": ["Fields", "Views"],
      "allowCreatingGroup": true
    }
  ]
}
```

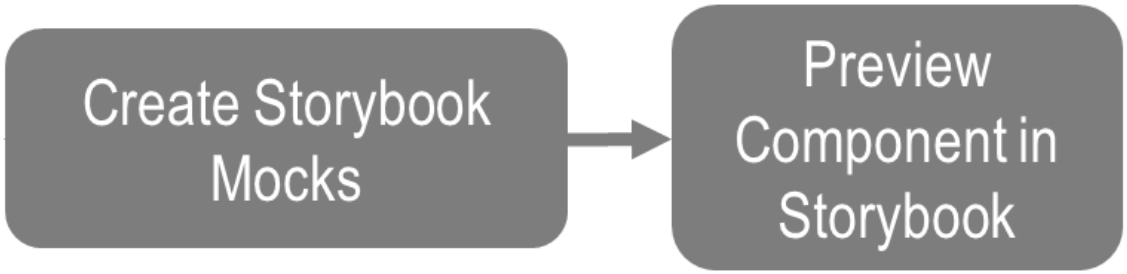
NOTE: The `index.tsx` file is already coded to handle multiple Regions and does not need to be updated.

You must update the mock data for this new Region in a Storybook.

Mocking and testing the layout template with a Storybook

Use a Storybook to preview the layout template and perform unit testing on it.

©2025 Pegasystems Inc



Creating Storybook mocks and previewing component in Storybook

To create a Story in the Storybook, you must write test mocks for the ConstellationJS PCore and PConnect APIs. You must also update the Details layout template to add support for multiple Regions.

When you bootstrap a new project using the Constellation DX Component Builder, the root of the new project's workspace folder contains a `.storybook` folder. This provides you with a Constellation-specific initialization of the Storybook for the project.

Storybook folder

The Story you create will be decorated automatically by the `preview.tsx` file and a Pega theme is applied to the Storybook.

1. Create or update your Story by creating a file called `details.stories.tsx` in your new component folder and pasting the following code into it:

```
import MyCoMyComponentsMyDetailsTemplate from './index'; //

export default { //
```

©2025 Pegasystems Inc

```
  title: 'Tutorial Storybook for Details Layout Template',
  component: MyCoMyComponentsMyDetailsTemplate,
};

const Template = (args) => (<MyCoMyComponentsMyDetailsTemplate {...args}/>
); //

export const HighlightedData = Template.bind({}); //

HighlightedData.storyName = 'Highlighted Fields Story'; //
HighlightedData.parameters = { controls: { hideNoControlsWarning: true } };
HighlightedData.args = { //
  showLabel = true,
  label = 'Details Layout Label'
  template = 'MyCoMyComponentsMyDetailsTemplate',
  getPConnect: () => {
    }
};
```

The code is described below:

- Import the component under test.

- Set-up the Story referencing the component under test.

- The render function.

- The function that binds the test data to the render function.

- The properties of the Story that drive the display of the Storybook.

- The arguments that will be passed in as props to the component through the render function.

©2025 Pegasystems Inc

> **NOTE:** To start the Storybook, the getPConnect function must be mocked for this Story.

2. Create or update your mocks by creating a mock data file called `mock.stories.ts` in your component directory and pasting the following code into it:

```
export const pyReviewRawMetadata =
{
  "name": "pyReview",
  "type": "View",
  "config": {
    "template": "Details",
    "ruleClass": "MyCo-MyCompon-Work-MyComponents",
    "showLabel": true,
    "label": "@L Details",
    "localeReference": "@LR MYCO-MYCOMPON-WORK-MYCOMPONENTS!VIEW!P
YREVIEW",
    "showHighlightedData": true,
    "highlightedData": [
      {
        "type": "TextInput",
        "config": {
          "value": "@P .pyStatusWork",
          "label": "@L Work Status",
          "displayMode": "STACKED_LARGE_VAL",
          "displayAsStatus": true
        }
      },
      {
        "type": "TextInput",
        "config": {
```

©2025 Pegasystems Inc

```
      "value": "@P .pyID",
      "label": "@L Case ID",
        "displayMode": "STACKED_LARGE_VAL"
        }
      },
    {
      "type": "DateTime",
      "config": {
        "value": "@P .pxCreateDateTime",
        "label": "@L Create date/time",
        "displayMode": "STACKED_LARGE_VAL"
      }
    },
    {
      "type": "UserReference",
      "config": {
        "label": "@L Create Operator",
        "value": "@USER .pxCreateOperator",
        "placeholder": "Select...",
        "displayMode": "STACKED_LARGE_VAL"
      }
    }
  ],
  "inheritedProps": [
    {
      "prop": "label",
      "value": "@L Details"
    },
    {
      "prop": "showLabel",
      "value": true
    }
```

©2025 Pegasystems Inc

```
  ]
},
"children": [
  {
    "name": "A",
    "type": "Region",
    getPConnect: () => {
      return {
        getRawMetadata: () => {
          return pyReviewRawMetadata.children[0];
        }

      }
    },
    "children": [
      {
        "type": "DateTime",
        "config": {
          "value": "@P .pySLADeadline",
          "label": "@L SLA Deadline"
        }
      },
      {
        "type": "DateTime",
        "config": {
          "value": "@P .pySLAGoal",
          "label": "@L SLA Goal"
        }
      },
      {
        "type": "RadioButtons",
        "config": {
```

©2025 Pegasystems Inc

```
          "value": "@P .pySLAStartTime",
          "label": "@L SLA Start Time",
          "listType": "associated",
          "datasource": "@ASSOCIATED .pySLAStartTime"
        }
      }
    ]
  }
 ],
 "classID": "MyCo-MyCompon-Work-MyComponents"
}

export const pyReviewResolved =
{
 "readOnly": true,
 "template": "Details",
 "ruleClass": "MyCo-MyCompon-Work-MyComponents",
 "showLabel": true,
 "label": "Details",
 "localeReference": "MYCO-MYCOMPON-WORK-MYCOMPONENTS!VIEW!PYREVI
EW",
 "showHighlightedData": true,
 "highlightedData": [
  {
    "type": "TextInput",
    "config": {
      "value": "New",
      "label": "Work Status",
      "displayMode": "STACKED_LARGE_VAL",
      "displayAsStatus": true
    }
  },
```

©2025 Pegasystems Inc

```
    {
      "type": "TextInput",
      "config": {
        "value": "M-1002",
        "label": "Case ID",
        "displayMode": "STACKED_LARGE_VAL"
      }
    },
    {
      "type": "DateTime",
      "config": {
        "value": "2022-12-11T20:06:27.232Z",
        "label": "Create date/time",
        "displayMode": "STACKED_LARGE_VAL"
      }
    },
    {
      "type": "UserReference",
      "config": {
        "label": "Create Operator",
        "value": {
          "userId": "riddn",
          "userName": "Niall Riddell"
        },
        "placeholder": "Select...",
        "displayMode": "STACKED_LARGE_VAL"
      }
    }
  ],
  "displayMode": "LABELS_LEFT"
}
```

©2025 Pegasystems Inc

```
export const regionChildrenResolved =
[
  {
    "readOnly": true,
    "value": "",
    "label": "SLA Deadline",
    "displayMode": "LABELS_LEFT"
  },
  {
    "readOnly": true,
    "value": "",
    "label": "SLA Goal",
    "displayMode": "LABELS_LEFT"
  },
  {
    "readOnly": true,
    "value": "",
    "label": "SLA Start Time",
    "listType": "associated",
    "datasource": [
      {
        "key": "AssignmentCreation",
        "value": "The creation time of the Assignment"
      },
      {
        "key": "CurrentTime",
        "value": "Current Date Time"
      }
    ],
    "displayMode": "LABELS_LEFT"
  }
]
```

©2025 Pegasystems Inc

ConstellationJS is a public API layer that requires mocking. As seen in the `index.tsx` file, the following PConnect functions need mocking:

- getChildren
- getRawMetadata
- getInheritedProps
- createComponent
- setInheritedProps
- resolveConfigProps

3. Add the mock data into the Story by opening the `details.stories.tsx` file, importing the mock data, and updating the empty getPConnect function as per the following code:

```
import {
  pyReviewRawMetadata,
  regionResolvedData,
  highlightedDataResolvedData
} from './mock.stories.ts';

...
HighlightedData.args = {
  ...
  getPConnect: ()=> {
    return {
      getChildren: () => {
        return pyReviewRawMetadata.children; //
      },
      getRawMetadata: () => {
        return pyReviewRawMetadata; //
      },
      setInheritedProps: () => {}, //
      getInheritedProps: () => { //
        return pyReviewRawMetadata.inheritedProps;
      },
```

©2025 Pegasystems Inc

```
      resolveConfigProps: (configProps) => {
        return resolvedFields[configProps.config.value];
      },
      createComponent: (config) => { //
        return createComponent(config);
      }
    }
  },
}
```

The code is described below:

 - Return the `children` array for the mock.

 - Return the complete mock object.

 - Obtain the array and perform no operation on this set as this is a mock.

 - Pull the resolved data from the mock `resolvedFields` array.

 - Call the createComponent mock function that will produce a test React component based on the config.

4. Build the mock component creator and prop resolver. The prop resolver should convert the raw config into resolved JSON data. In ConstellationJS, resolving props is the process of applying the model to the data to produce real objects that can be used in rendering. Since there is no mock ConstellationJS available, apply the model to the field data in the `pyReviewRawMetadata` object.
   Create a renderField function that converts raw field data into a mock object that can be used as the basis for rendering the component. Paste the following renderField function at the end of the `details.stories.tsx` file:

```
const renderField = (resolvedProps) => { //
```

©2025 Pegasystems Inc

```
  const { displayAsStatus = false, displayMode, value = "", label = "", theme = use
Theme()} = resolvedProps;

  const variant = displayMode === 'LABELS_LEFT' ? 'inline' : 'stacked'; //

  let val = value!="" ? (
    <Text variant='h1' as='span'>
      {value}
    </Text>
  ):("");

  if(label === 'Create date/time')
    val =
    (
      <DateTimeDisplay
        value={value}
        variant='datetime'
        format='long'
        clockFormat={null}
      />
    )

  if(displayAsStatus === true)
     val = (StatusWorkRenderer({value})); //

  if(label === 'Create Operator') //
    val = (
      <Button
        variant='link'
        style={label !== null ? { width: 'max-content', height: theme.components.in
put.height } : undefined}
      >
```

©2025 Pegasystems Inc

```
      {value.userName}
    </Button>
  );

  if(variant === 'inline'){
    val = value || <span aria-hidden='true'>&ndash;&ndash;</span>;
  }else{
    val = (
      <Text variant='h1' as='span'>
        {val}
      </Text>
    )
  }

  return ( //
    <FieldValueList
      variant={variant}
      fields={[{ name: label, value: val }]}
    />
  );
};
```

The code is described below:

 - Passes the resolved props gathered using getDebugger.

 - The FieldValueList component can handle both the scenarios - `LABELS_LEFT` and `STACKED_LARGE_VAL`. For more information, see FieldValueList.

 - The StatusWorkRenderer component added to the utils library ensures that pyStatusWork is displayed in the correct format.

©2025 Pegasystems Inc

□ - The operator record is rendered using a Link Button component. This displays the user name in the same way as during run-time rendering. It does not provide any preview functionality. For more information on Link Buttons, see Link Buttons.

□ - The render function.

5. Replace the Template and Primary functions in the `details.stories.tsx` file with the following code:

```
const Template = (args) => {

  const regionAChildren = pyReviewRawMetadata.children[0].children.map((chil
d) => {
    return Primary.args.getPConnect().createComponent(child)
  }); //□

  const regionBChildren = pyReviewRawMetadata.children[0].children.map((chil
d) => {
    return Primary.args.getPConnect().createComponent(child)
  }); //□

  return (
    <MyCoMyComponentsMyDetailsTemplate {...args}>
      <Region>
        {regionAChildren} //□
      </Region>
      <Region>
        {regionBChildren} //□
      </Region>
    </MyCoMyComponentsMyDetailsTemplate>
  )
};
```

©2025 Pegasystems Inc

```
export const Primary = Template.bind({}); //

Primary.args = {
  template: 'MyCo_MyComponents_MyDetailsTemplate',
  showHighlightedData: true,
  label: "Test Details",
  showLabel: "true",
  getPConnect: () => { //
    return {
      getChildren: () => {
        return pyReviewRawMetadata.children;
      },
      getRawMetadata: () => {
        return pyReviewRawMetadata;
      },
      getInheritedProps: () => {
        return pyReviewRawMetadata.config.inheritedProps;
      },
      createComponent: (config) => { //
        switch (config.config.value){
          case '@P .pyStatusWork':
            return renderField(pyReviewResolved.highlightedData[0].config);
          case '@P .pyID':
            return renderField(pyReviewResolved.highlightedData[1].config);
          case '@P .pxCreateDateTime':
            return renderField(pyReviewResolved.highlightedData[2].config);
          case '@USER .pxCreateOperator':
            return renderField(pyReviewResolved.highlightedData[3].config);
          case '@P .pySLADeadline':
            return renderField(regionChildrenResolved[0]);
          case '@P .pySLAGoal':
            return renderField(regionChildrenResolved[1]);
```

©2025 Pegasystems Inc

```
        case '@P .pySLAStartTime':
          return renderField(regionChildrenResolved[2]);
      }
    },
    setInheritedProp: () => {},
    resolveConfigProps: () => {},
  };
 }
};
```

The code is described below:

☐ - Create a `children` array to simulate the single Region A. The children are only Regions that contain the rendered components.

☐ - Pass this array as children that will be accessible from the component under test through props.children.

☐ - Standard Storybook set-up for binding data to Stories.

☐ - Mock all PConnect functions used in the component under test.

☐ - Pass the mock data into the component creator that returns a React component using the resolved data from the `mock.stories.ts` file.

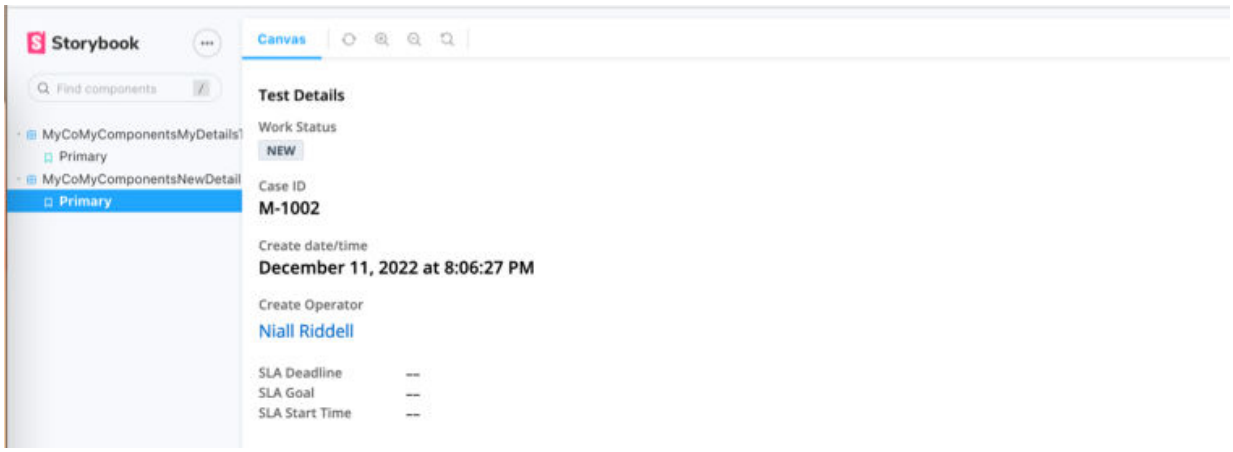
6. Replace the header section of the `details.stories.tsx` file with the following code:

```
import { Text, FieldValueList, Button, DateTimeDisplay, useTheme } from '@peg
a/cosmos-react-core';
import MyCoMyComponentsMyDetailsTemplate from './index';
import { pyReviewRawMetadata, pyReviewResolved,  regionChildrenResolved } f
rom './mock.stories'
import { Region, StatusWorkRenderer } from './utils'
```

©2025 Pegasystems Inc

You can now run the Storybook in the project directory.

7. Run the Storybook by entering the following command: `npm start`
   You will see an additional Story in your Storybook as seen in the following image:



Storybook component view

8. (Optional) Go to Step 2 above. Repeating the steps will ensure that the component is completely tested and also means that you can view your changes in the Storybook without having to wait for the component to be published to Pega Infinity.

Publishing the layout template

Perform integration testing on the layout template by publishing it to Pega Infinity.

Authenticating and publishing the component

©2025 Pegasystems Inc

To use a component in App Studio Authoring, you must publish it. Publishing the component bundles all the assets together and uses Webpack to create a gzipped package. The Constellation DX Component Builder then connects with Infinity and then uses a Pega service to create or update an instance of Rule-UI-Component. A declare trigger on Rule-UI-Component performs the onward publish to the Constellation Service. For more information about the Constellation Service, please see Constellation UI Service.

Authenticate your access to the Pega Infinity Server and publish the component.

For more information, see:

- Authenticating Pega Platform Server user access for standalone Constellation DX components
- Publishing standalone Constellation DX components

> **NOTE:** Your authentication might expire while publishing the component. Run the `npm run authenticate` command again to get a new token.

The layout template is now ready to be tested in App Studio.

Using the layout template in App Studio

Learn how to use the layout template in App Studio.

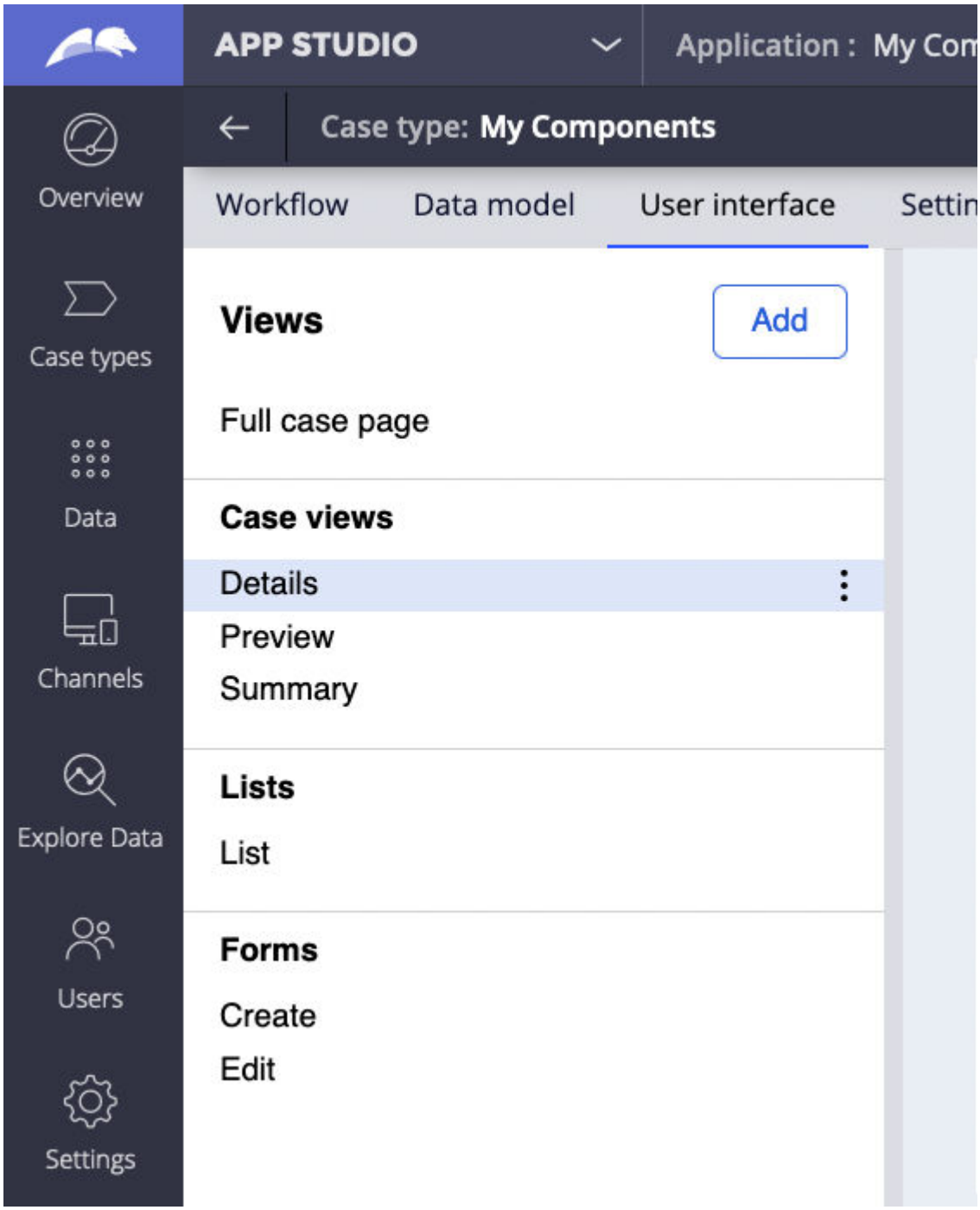
©2025 Pegasystems Inc

Testing the component

After the Details layout template is published successfully, follow the below procedure to use it in App Studio:

1. In App Studio, create a new Case type or reuse an existing Case type.
2. In the Case type, navigate to the **User interface** tab and select **Details**.

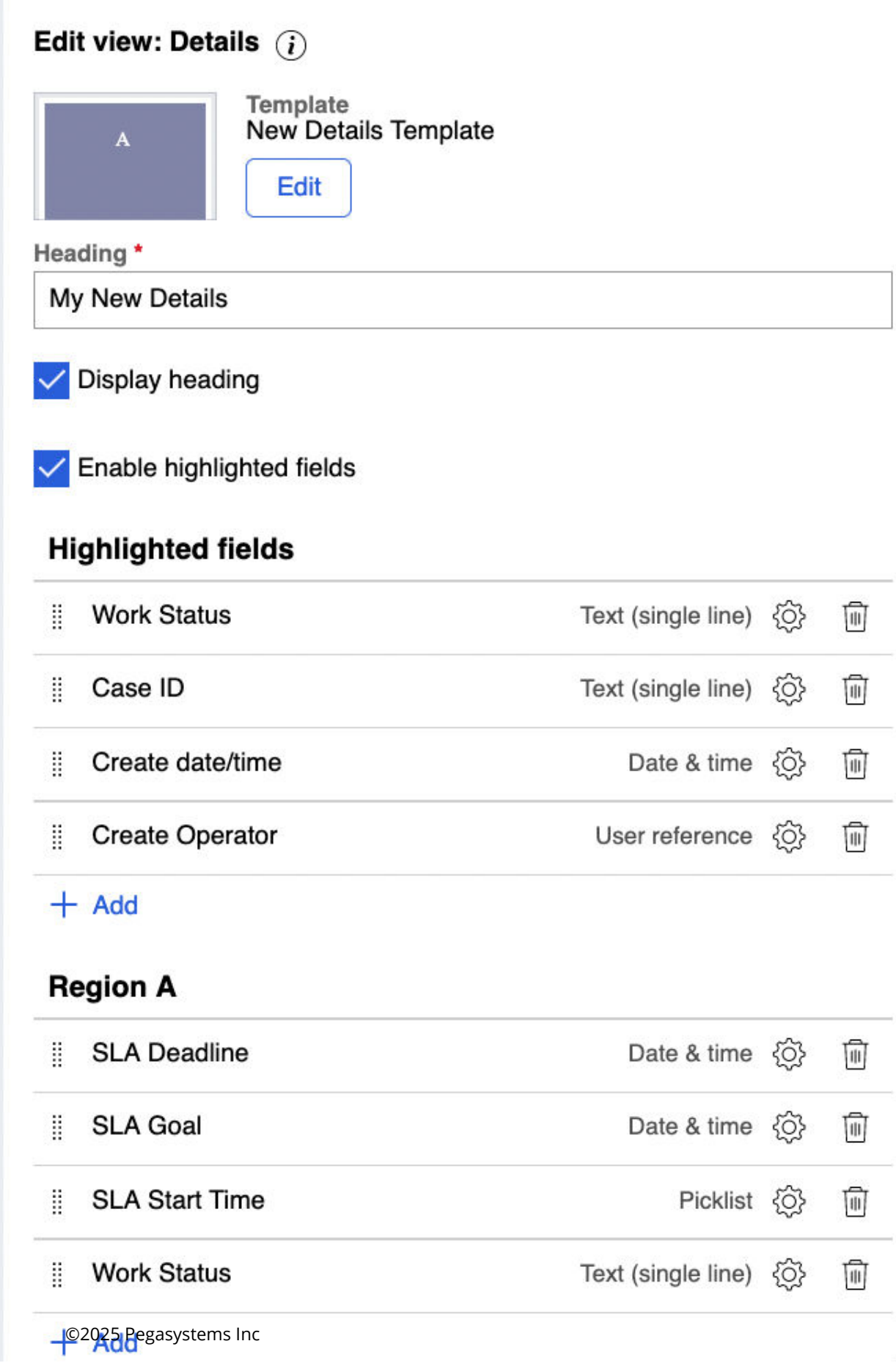
©2025 Pegasystems Inc



Case type User interface details

3. In the **Edit view: Details** panel, select your template.

©2025 Pegasystems Inc

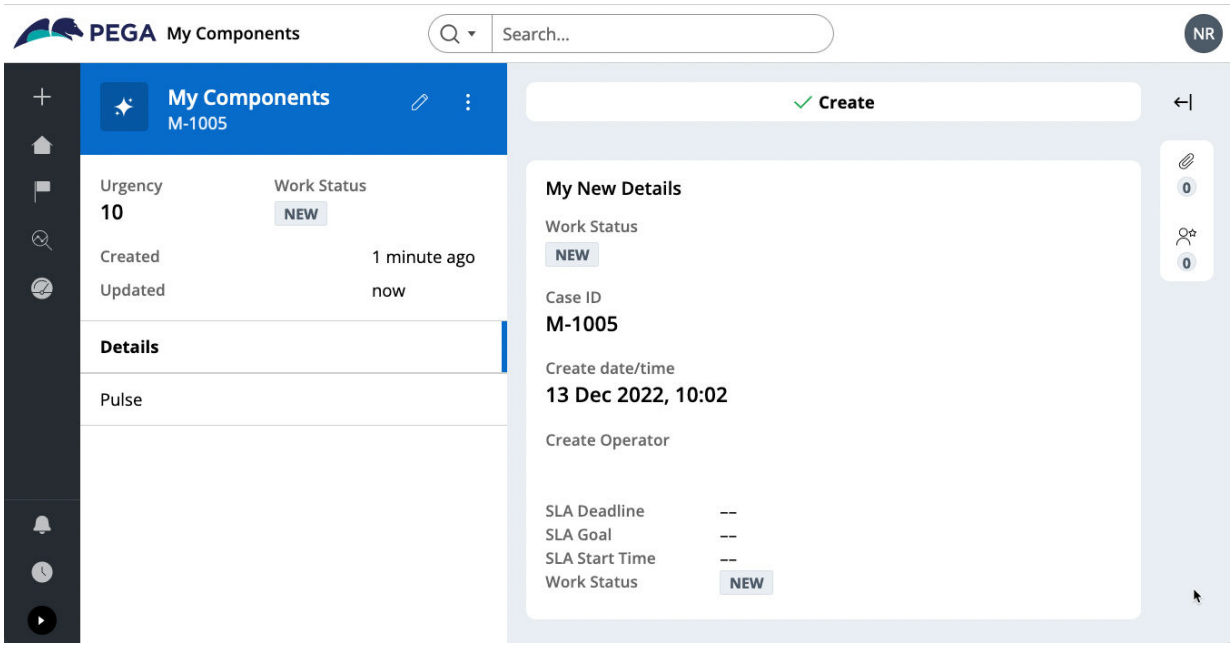


©2025 Pegasystems Inc

Edit view: Details panel

You can now add fields and views to determine if your template renders as expected at run time.

4. Click **Preview**, and create a new case from the Case type that the template was added to.
   You can now view your new template.



Details template view

Result:

To efficiently build the layout template, perform the editing, mocking, testing, and publishing steps frequently till the desired result is achieved. Mocking and testing the layout template should especially be performed instead of frequently publishing it to Pega Infinity. The publishing process is aimed at integration testing the component.

©2025 Pegasystems Inc

Video learning series

Watch these videos to help you gain a better understanding of working with Constellation DX components.

List of videos

The video learning series comprises the following videos:

- Build Constellation DX components
- Learn about the DataPageUtils class
- Configure a data page for Constellation
- Paging using the getDataAsync API
- Querying using the getDataAsync API
- Using Themes in Constellation DX components
- Reading and writing data using the getValue and setValue APIs

Videos

Video	Description
Build Constellation DX components	This video provides a detailed overview of the Constellation DX components.
Learn about the DataPageUtils class	This video provides a detailed overview on how to retrieve data using the APIs within the DataPageUtils class.
Configure a data page for Constellation	This video provides information on configuring

©2025 Pegasystems Inc

Video	Description
	a data page, making it searchable or queryable, and managing access to the data page.
Paging using the getDataAsync API	This video provides information on paging and its modes along with examples that show how paging can be used to retrieve data from a data page.
Querying using the getDataAsync API	This video provides information on the query element and its properties along with an example that shows how it can be used to perform advanced data retrieval operations on a data page.
Using Themes in Constellation DX components	This video shows you how to use themes in Constellation DX components.
Reading and writing data using the getValue and setValue APIs	This video provides a detailed overview on how to read and write data using the getValue and setValue APIs.

PEGA ©2025 Pegasystems Inc

Troubleshooting Constellation DX Component Builder

The following article covers the errors you might encounter when using the DX Component Builder, the reason for the error, and the resolution to implement.

For more information on troubleshooting Pega Platform-related errors, see Troubleshooting your system.

Error code or message	Reason	Resolution
custom-dx-components: command not found	• The node is not installed or the wrong version of the node is installed. • Your system might not have the necessary files where the command is set up. • The Node Version Manager (nvm) is unable to locate the command.	• If nvm is installed, enter `nvm list` to display all the node versions installed on your system. If the correct node version is not installed, switch to it by entering `nvm use <version>`. • If nvm is not installed, install it. For more information, see Installing nvm. • Enter `npm install -g @pega/custom-dx-components@<version>`.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
The terminal or command prompt window freezes when running npm commands	This error usually occurs when you are using Artifactory. Artifactory might be missing the path information.	Update the `.npmrc` file and add their path to Artifactory, along with the `auth` and `email` attributes. For example: @pega:registry=https://bin.pega.io/artifactory/api/npm/npmjs-devbin.pega.io/artifactory/api/npm/npmjs-dev:_auth = xxxxxxx always-auth = true email = youremail@test.com
Publishing a component displays a success message in command line, however the component is not published.	Operator used to authenticate with and publish to Pega Platform does not have the correct configuration.	Add the `Allow rule check out` access setting to the operator record under the **Security** tab.
An authentication failure occurs along with an EPROTO error when you run the `npm run authenticate` command.	This error occurs because node 18 does not support legacy versions of TLS.	Update the TLS version for the Pega Platform server and any other intermediate servers. **NOTE:** If you are unable to update the TLS version ,

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
		set the `legacyTLS` attribute to `true`. This will reinstate support to legacy versions of TLS.
The published component does not match the look and feel of the Constellation design system.	This happens when the component does not use the Pega theme.	Use the Pega theme when creating your component. From Constellation DX Component Builder 8.8.19 onwards, component templates support Pega themes by default. For more information about using themes in components, see Using Themes in custom DX components.
The following error message appears: `[_internal-build-dev-only] "building..." [_internal-build-dev-only] Browserslist: caniuse-lite is outdated. Please run:`	This is a Microsoft Windows-specific issue.	Run the following command: `npx browserslist@latest –update-db`

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
[_internal-build-dev-only] npx browserslist@latest --update-db [_internal-build-dev-only] Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating Terminate batch job (Y/N)? y **NOTE:** This error occurs only in machines running Microsoft Windows.		
The local server does not start when you run the `npm run start-dev` or `npm run start-dev-https` commands. **NOTE:** This occurs only in machines running	This is a Microsoft Windows-specific issue.	Delete the `dist/constellation` directory before you run the server.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
Microsoft Windows.		
A dialog box appears prompting you to install rimraf. **NOTE:** This occurs only in machines running Microsoft Windows.	This is a Microsoft Windows-specific issue.	Install rimraf by running the following command: `npm install --save-dev rimraf`
With the adoption of cosmos 4.0.1, there has been an increase in the bundle size of the component.	This is because the size of cosmos 4.0.1 has increased.	The bundle size of the component will be reduced in the new packaging that will be released in early 2024.
The following warning message appears in a .js or .jsx file: `Unused eslint-disable directive (no problems were reported from 'no-undef')`	This warning message appears because PCore was added to globals in ESLint rules.	Open the corresponding .js or .jsx file and remove the following code: `/* eslint-disable no-undef */`

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
The `NET::ERR_CERT_AUTHORITY_INVALID` warning message appears in your Google Chrome browser when you run the `npm run authenticate` or `npm run startStorybook` commands.	This issue occurs when using the Google Chrome browser and the local front-end server is configured with HTTP. Due to Google Chrome's domain security policies, the browser automatically redirects to the HTTPS protocol.	Update Google Chrome's domain security policies to ignore localhost: 1. Enter the following text in the address bar of the browser and press the Enter key: `chrome://net-internals/#hsts` The **Domain Security Policy** page is displayed. 2. In the **Delete domain security policies** section, enter `localhost` in the Domain text box and click Delete. 3. Reload the browser and accept localhost as unsafe.
The Glimpse component does not work.	This happens because the structure of the Glimpse component has changed in cosmos 4.2.1.	Update the code of your existing Glimpse component as mentioned below: 1. Move the `primary`, `secondary`, and

©2025 Pegasystems Inc

Error code or message	Reason	Resolution

		`visual` parameters to the `heading` object.
		2. Add the mandatory `target` parameter that provides information on the component that launched the Glimpse component.
		3. Add the `onDismiss` parameter that is a callback function that is triggered when the Glimpse component is dismissed.
		
		Example
		
		The following code is an example of the updated Glimpse component in cosmos 4.2.1:
		< Glimpse
		heading = {
		{
		primary: data.pyOperatorInfo.pyUserName,
		secondary: data.pyOperatorInfo.pyEmailAddress,

©2025 Pegasystems Inc

Error code or message	Reason	Resolution

		``
		visual: opAvatar
		}
		}
		fields = {
		fields
		}
		target = {
		pTarget
		}
		onDismiss = {
		() => {
		pTarget = null;
		}
		}
		``

Components do not get published to the specified rule set version.	This happens because the operator used for authentication does not have its default access group set to the application containing the ruleset and ruleset version specified during the publishing process.	Ensure that the operator used for authentication has its default access group set to the application containing the ruleset and ruleset version that the component will be published to.
Custom component styles are not displayed correctly in the App Studio preview or web embed.	All Constellation DX Components need to be wrapped by a Configuration component inside the render. This was not done in	Wrap your render method with a Configuration component. Example:

©2025 Pegasystems Inc

Error code or message	Reason	Resolution

	Component code created in DXCB 8.8.10 or earlier versions.	`return (`
		`  <Configuration>`
		`  ...your existing code`
		`  </Configuration>`
		`);`

Icons used for Constellation DX components are not visible.	This happens when the icons are not imported and registered correctly.	For an icon to be visible, you must import the icon file, register it in Camel Case, and then use it in Kebab Case.
		
		1. **Import the icon file**
		Import each icon file from `@pega/cosmos-react-core`.
		`import * as listNumber from '@pega/cosmos-react-core/lib/components/Icon/icons/list-number.icon';`
		`import * as arrowRight from '@pega/cosmos-react-core/lib/components/Icon/icons/arrow-right.icon';`
		`import * as globe from '@pega/cosmos`

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
		-react-core/lib/components/Icon/icons/globe.icon'; 2. **Register the icons** Register the icons in Camel Case. registerIcon(listNumber, arrowRight, globe); 3. **Use the icons** Use the icons in Kebab Case. <Icon style={{ height: '1.5rem', width: '1.5rem' }} name='arrow-right' role='img' aria-label='arrow right' /> For more information on the icons available in the Constellation design system, see Icon.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
Constellation DX components are visible as *Rule-UI-Component* rules in Pega Platform, but are not visible in App Studio.	This happens when UI assets do not synchronize between Pega Platform and Constellation.	**NOTE:** This resolution applies to only Pega Platform '24.2 and earlier releases. Trigger the synchronization of the UI assets between Pega Platform and Constellation by resetting the ConstellationSvcURL Dynamic System Settings (DSS) as mentioned in the following procedure: 1. Open the ConstellationSvcURL DSS and verify that it is set to a value. 2. Delete the value and click **Save**. 3. Enter the deleted value and click **Save**. Constellation DX components should now be visible in App Studio. If not, check the Pega log for HTTP errors.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
		NOTE: For customers of Pega Cloud, the rule form DSS will appear empty. This is expected. For customers of Pega Cloud, reset the ConstellationSvcURL DSS as mentioned in the following procedure: 1. Open the Constellation SvcURL DSS and verify that it not set to any value. 2. Enter any random value, such as 'abc', and click **Save**. 3. Delete the value and click **Save**.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
The expected version of a Constellation DX component is not displayed at run time.	There are two reasons due to which this issue might occur: • In an environment where multiple applications use different ruleset versions of a specific component, saving the Dynamic System Settings (DSS) causes the non-deterministic version of the component to be selected. This might not be the expected version of the component. • When the current sync mechanism does not have the current application context, performing an application sync will pick the highest ruleset version of the application. This might not be ruleset version of the configured application.	**NOTE:** This resolution applies to only Pega Platform '24.2 and earlier releases. To ensure that the expected version of a Constellation DX component is displayed at run time, ensure that the following conditions are met: • All applications use the same ruleset version that holds the expected component. • The ruleset version of the configured application is the highest ruleset version.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
The `ERR_MODULE_NOT_FOUND` error message is displayed when you try to initialize a project.	This occurs because of deprecated code in the code base.	Initialize the project by running one the following commands depending on the Pega Platform version installed. • Pega Platform '25 - `npx @pega/custom-dx-components@25.1.13 init` • Pega Platform '24.2 - `npx @pega/custom-dx-components@24.2.10 init` • Pega Platform '24.1 - `npx @pega/custom-dx-components@24.1.11 init` • Pega Platform '23 - `npx @pega/custom-dx-components@23.1.14 init`
The following error message appears when running the `PCore.getDataApiUtils().getData` API:	This occurs when an array of objects is passed as the `payload`. With the typedefs package version	Pass an object as the `payload` instead of an array of objects. For example:

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
TS2345: Argument of type '(displayResponse: { data: Array<never>;}) => void' is not assignable to parameter of type '(value: unknown) => void | PromiseLike<void>'. Types of parameters 'displayResponse' and 'value' are incompatible. Type 'unknown' is not assignable to type '{ data: never[]; }'.	3.2.1 supported in DXCB 24.2.10, you can only pass an object.	{ dataViewParameters: { CaseInstanceKey: caseID } }
The following error message appears: TS2322: Type 'string | undefined' is not assignable to type 'string'. Type 'undefined' is not assignable to type 'string'.	This occurs when you have defined a variable as a string. With the typedefs package version 3.2.1 supported in DXCB 24.2.10, several types were migrated from `any` to `string | undefined`.	Redefine your property as `string | undefined`. For example: const myProp: string | undefined; If you cannot redefine your property as `string | undefined` because the property is also used in a function of type `string`, handle the undefined type as mentioned in the following scenarios:

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
		• If your `string` uses `string` functions, you will need to use a `?` to handle the undefined type that may be assigned to it. For example, fix errors when matching a `string` to `string | undefined` as shown below: `myString?.toLowerCase()` • If you have a `string` defined and it is populated with a function that passes `string | undefined`, you will need to handle the undefined type. For example, handle the undefined type by changing it to a blank string as shown below: `myString = PCore.getEnvironm`

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
		`entInfo().getOperatorName() ?? "";`
The following error message appears: TS1484: 'ReactElement' is a type and must be imported using a type-only import when 'verbatimModuleSyntax' is enabled.	This occurs when `ReactElement` is not defined as a type.	Define `ReactElement` as a type by changing `import { Fragment, ReactElement } from 'react';` to `import { Fragment, type ReactElement } from 'react';`.
The following error message appears: TS18046: 'displayResponse' is of type 'unknown'.	This occurs when the response of the promise needs to be casted.	Cast the response of the promise from `.then(displayResponse => {` to `.then((displayResponse: any) => {`.
The following error message appears: TS2322: Type '{ value: any; formattingOptions: { fractionDigits: number; groupSeparators: boolean; notation: "standard" | "compact"; }; unit: string | undefined; unitPlacement: string | undefined; }' is not assignable to type 'IntrinsicAttributes & Omit<NumberDisplayProps, "ref	This occurs when the `unitPlacement` attribute is used. This attribute is not supported from DXCB 24.2.10.	Delete the `unitPlacement` attribute.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
">& RefAttributes<HTMLSpanElement>'. Property 'unitPlacement' does not exist on type 'IntrinsicAttributes & Omit<NumberDisplayProps, "ref">& RefAttributes<HTMLSpanElement>'.		
The following error message appears when you run the command to create a Constellation DX component: npm : File C:\Program Files\nodejs\npm.ps1 cannot be loaded because running scripts is disabled on this system. **NOTE:** This occurs only in machines running Microsoft Windows.	This error occurs when script execution is not enabled.	Enable script execution: 1. Open Windows PowerShell as Administrator. 2. Run the following command to allow script execution: Set-ExecutionPolicy RemoteSigned
The following error message appears in DXCB 25.1.10 when initializing a project:	This error occurs when the node version installed is lower than 22.15.0.	Update your system node version to 22.15.0 or later before initializing a project.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
SyntaxError: The requested module 'node:events' does not provide an export named 'addAbortListener'		
Components in your application are not available after you update to DXCB 25.1.10.	This happens when the ruleset version to which the component was published is not included in the application.	Add the component's ruleset version to the application.
The following error message appears when you deploy a Constellation DX component to production in a version earlier than Pega Infinity '25: Error fetching/processing component definition	This happens when you are using a version earlier than Pega Infinity '25 and the component is not added to the highest ruleset version of the application.	Update to Pega Infinity '25 or add the component to the highest ruleset version of the application.
Linting config issues occur in Visual Studio Code after you update to DXCB 25.1.10.	This happens because DXCB 25.1.10 uses ESLint 9, which uses the flat config file and Visual Studio Code is not configured to use flat config.	To enable typechecking using the flat config file, configure Visual Studio Code to use flat config. 1. In Visual Studio Code, click **View** > **Extensions** and install the ESLint extension. 2. Click **File** > **Preferences** > **Settings**.

©2025 Pegasystems Inc

Error code or message	Reason	Resolution
		3. In the **Search settings** field, enter **eslint**, and in the **Eslint: Use Flat Config** setting, click **Edit in settings.json**. 4. In the `settings.json` file, set `eslint.useFlatConfig` to `true`. If you are using a terminal other than Visual Studio Code, see Plugin Migration to Flat Config.
After publishing a library, if you update its components and then publish the library to another Ruleset or ruleset version, the updated components do not reflect in the run-time portal.	This happens because the browser cache contains the older components and not the updated components. With browser cache enabled, the components are fetched from the browser cache instead of the server.	Save the application Rule after re-publishing the updated library. The browser cache is updated and the updated components reflect in the run-time portal.

©2025 Pegasystems Inc