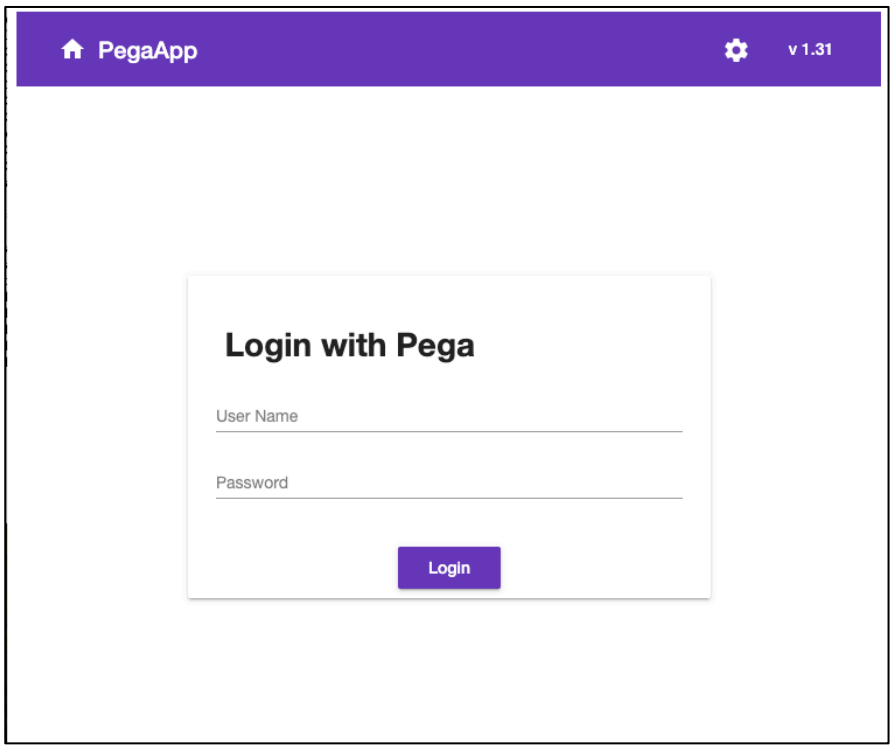
Angular Overview

This document presents an overview of how the sample Angular application works. It is accurate at the time of writing but may have been update/modified since. The sample Angular application defines a simple case worker/manager portal. It utilizes the Pega DX API along with Angular 6 and Material IO. It is meant to show examples of how to utilize the Pega DX API with an open source framework. It is not meant to provide best practices or show all use cases.

This document is created to give the developer a high level understanding of the Angular application in order understand different use cases of the Pega DX API possibly re-use some of the Angular components for other Angular applications.

login



Login screen is very basic and unsecure. We are using basic authoring with base64 encoding for authentication. This is meant as a simple example and should not be used in production.

The `login` component contains the login screen that captures the login information and uses the login service to login. If successful, the `login` component will store the login credentials in the browser local storage. As these credentials are needed for every REST call afterwards.

The `login` component will store the following in the browser local storage:

- User credentials
- User access group
- User work group
- User work baskets (queues)

`login` component will send the following messages:

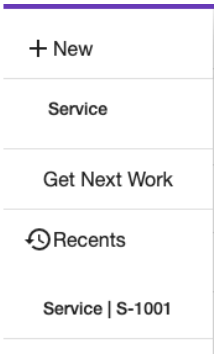
- If login is successful, send “LoggedIn” via `getloginstatus`

navigation

`navigation` component contains the main top bar which consists of:

- Drawer Icon (hamburger icon) – open/closes Drawer container
- Drawer container
    - New case type list
    - Get Next work
    - Recents (desktop recents)
- Settings icon
    - Settings dialog for turning on 8.3+ features (settings dialog component)
- Version
- User/Login/Logout (login component)
- Main tabs (maintabs component)

Drawer container



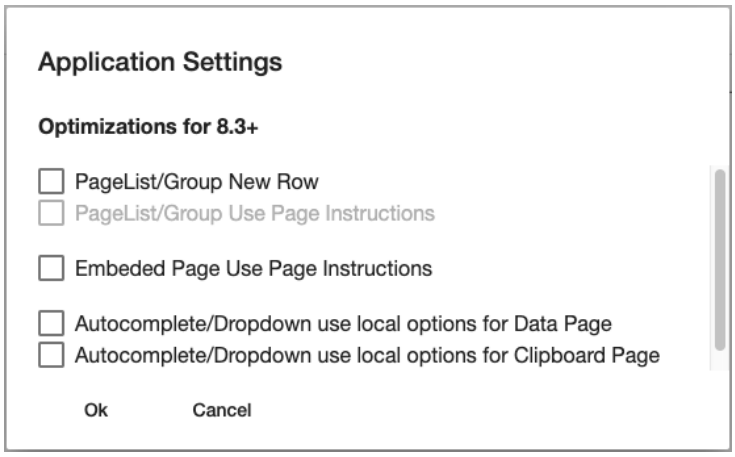
Drawer container contains:

- createcaselist – list of possible case types (see `createcaselist` component)
- Get Next Work link
- recentslist – list of recents from desktop (see `recentslist` componet)

Settings

Clicking on the settings icon, brings up the `settingsdialog` component.

settingsdialog



Settings dialog allows the user to pick 8.3+ new features and utilize them vs original api functionality. Each setting is stored in browser "local storage" so can be retrieved from various components and is kept for the current user, between sessions, provided the browser cache hasn't been refreshed.

However, the code for the settings dialog, exists within the `navigation` component, since it is the `navigation` component that launches the dialog.

createcaselist

`createcaselist` component will subscribe to the `loginStatus` message initially. Once login happens, will subscribe to the case service `getCaseTypes`. This will return a list of case types. This component will take that list and add to the "drawer" the list of case types that can be selected.

If a case type is selected, this component will register a `createCase` (if skip new) service or send a `onNewCase` message if "new".

For `createCase`, upon response, this component will send an `openAssignment` message and a `refreshWorklist` message.

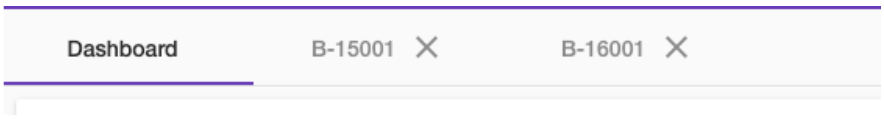
recentslist

`recentslist` component is an example of how to get recents. However, the current implementation of recents only handles work objects that were open via the desktop application and not via DX API.

`recentslist` component will subscribe to the `loginStatus` message initially. Once login happens, will subscribe to the `datapage` service `getDataPage` getting data page that has a list of recents (Declare_pxRecents). The response will update the local global `recentsLists$`, which will cause the component to redraw the lists of recents.

When a recent has been selected, an `openRecent` message will be sent.

maintabs



`maintabs` component is responsible for displaying the dashboard (as the first tab) and subsequent work items. The Dashboard tab cannot be dismissed.

Tabs will either be `worklistpanel` (dashboard) or `workitem` components.

This component will subscribe to the following messages:

- `openassignment` – when received, a new tab will be created, with the caseID, mark as *case*
- `closework` – when received, the message will contain a *workID*, will see if there is a corresponding tab, if so remove it.
- `opennewcase` – when received, a new tab will be created labeled “New”, mark as *new case*
- `renametab` – when received, find the tab with the given name and rename it
- `openrecent` – when received, create a new tab with the give case name, mark as *recent*

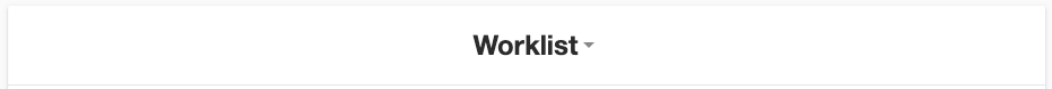
After a tab is created, `ngAfterViewChecked` will be called. Here messages will be sent:

- if marked as *case*, send `getassignment` message
- if marked as *new case*, send `getnewcase` message
- if marked as *recent*, sent `getrecent` message

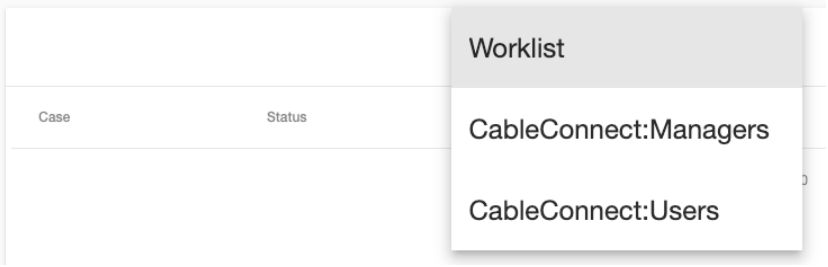
The above messages that are sent, will be received by the `workitem` component inside the tab, and will take the appropriate action.

So, the `maintab` components responsibility is to create a tab and put a `workitem` or `worklist` component in it. Then once the tab is visible, send appropriate messages to the `workitem` of why the tab was created (new case, open assignment, open recent, etc.)

worklistpanel



`worklistpanel` component is the main panel (dashboard) for the worklist. If the user has workbaskets, then instead of a label at the top of the panel, a drop down will appear, and user can select which worklist (basket) to view.



`worklistpanel` component consists of the panel to display the header/workbasket dropdown and the `worklist` component.

When the `worklistpanel` component initializes, it goes to browser local storage to retrieve the json for the user's workbaskets. If there is a list of workbaskets, it is put into an array, that is used by the html part of the component to build the drop down.

In the dropdown, if the user selects a different basket, a `refreshworklist` message is sent.

worklist

`worklist` component is a material table that has 4 columns. It is sourced from `worklist$`. It also has pagination with page sizes of 10, 20, 50, 100, 500.

Upon initialization, the `worklist` component will subscribe to the `datapage` service `getDataPage` looking for the users *D_Worklist* data page. The response will be put in `worklist$`, along with pagination and sorting data.

`worklist` component will also subscribe to `refreshworklist` message. Whenever it received this message re-subscribe to `datapage` service, as above and update `worklist$`. This will cause the worklist to re-display.

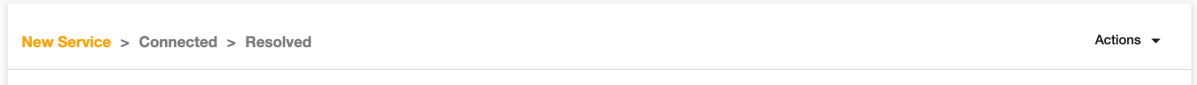
On each row of the table that is display, a click is attached. When the row is selected an `openassignment` message is sent with data from the row.

workitem

`workitem` component is the most complex component that handles lots of messages that all apply to the case that it contains.

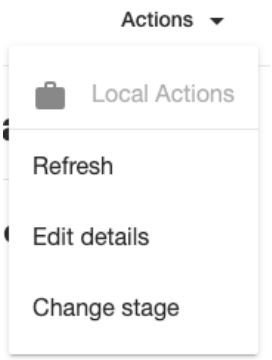
Header

breadcrumb



Uses the `breadcrumb` control

actions



Actions drop down retrieves the data from `GET /assignments/{ID}`. The action drop down has 2 lists: assignments and local actions. Going through the return *actions* from /assignments, any that have a *type* of "assignment, go in the assignments list (excluding the action we are currently on), and any that have the *type* of "case" go into the local actions list.

The action dropdown consists of the local actions and assignments.

Body

topview/toppage

`workitem` component contains the `topview` or `toppage` component, described in the subcomponents section.

casedetails

In addition, the `workitem` component partially mimics a perform harness by containing the `casedetails` component.

Footer (actions)

Cancel

If a local action, just refresh the view, otherwise send `closework` message.

Save

Register `case` service `updateCase`, with response:

- clear page instructions
- get assignment
- send `refreshcase` message

Submit

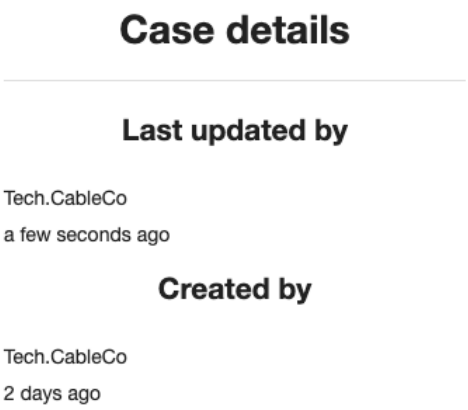
If a local action, update local action state and refresh the view, otherwise:

- register for `assignment` service `performActionOnAssignment`, with response:
    - if assignment
        - get next assignment
        - send `refreshcase` message
    - if page (Confirm, etc.)
        - get case
        - send `refreshcase` message

Close

If a local action, refresh view with restored state, otherwise, send `closework` message.

casedetails



Because we don't provide an API to show *Perform*, this is an example of getting some of the parts that can be found in the out of the box *Perform*. Case details shows created and last updated data. This data changes upon submit of the case, so it is a good example of getting non flow action case data and displaying/updating with a case.

`casedetails` component contains a header and `group` component.

Upon creation, `casedetails` component registers for messages from the `case` service. Whenever the case is updated, this component will update.

```
<app-casedetails [CaseID]></app-casedetails>
```

Services

There are 2 kinds of services that we are using in this Angular app. The standard service which makes a REST call to a server. The other service is a pub/sub service, which we are calling “messages” in this context.

_services

Services are http requests to the server using REST api. Services will return a response from the server. Each service can have a number of methods which translate to different REST endpoints and can include GET, POST and PUT.

assignment

`assignment` service has the following methods:

- `getAssignment`
- `getFieldsForAssignment`
- `performRefreshOnAssignment`
- `performActionOnAssignment`

case

`case` service has the following methods:

- `getCase`
- `getCaseTypes`
- `getCaseCreationPage`
- `createCase`
- `updateCase`
- `refreshCase`
- `getPage`
- `getView`

datapage

`datapage` service has the following methods:

- `getDataPage`

user (login)

`user` service has the following methods:

- `login`

_messages

Messages are pub/sub service. This is the main way that components communicate between one another in the Angular app.

Since different components subscribe to the same message, it is important that those components know if the message is received pertains to that component. Most messages contain `sCaseID`, such that if the component is tied to a case, it should only utilize the message if the *caseID* matches.

Most components are noted here if there are main subscriber (they are component you are requesting them to do something.)

closework

Publish: when want `maintabs` component to close a work item
Subscribe: when you want to be notified to close your work item

getactions

Publish: when you want the `workitem` component to handle an action (all actions are handled via the `workitem` component)
Subscribe: if you want to know if some other component is requesting a handle action

NOTE: all field components that support “actions” will Publish this message

getassignment

Publish: when you want the `workitem` component to retrieve an assignment
Subscribe: if you want to know if some component is requesting an assignment

getcase

Publish: when you want the `workitem` component to retrieve a case
Subscribe: if you want to know if some component is requesting a case

NOTE: `dropdown` and `autocomplete` subscribe to the `getcase` message

getchanges

Publish: when you want the `workitem` component get update state based upon given changes
Subscribe: when you want to know if some component should update state

NOTE: most field components Publish this message as they are changed.

getloginstatus

Publish: when you want to get request the login status
Subscribe: when you to get the login status

getnewcase

Publish: when you want the `workitem` component to get a new case
Subscribe: when you want to know if some component is requesting a new case

getpage

Publish: when you want the `workitem` component to get a page (like **Review** or **Confirm**)
Subscribe: when you want to know if some component is requesting a page

getrecent

Publish: if you want the `workitem` component to open a recent assignment (like open assignment)
Subscribe: if you want to know if some component is requesting an open recent

getview

Publish: when you want to update `topview` component with view data
Subscribe: when you want to know if some component sent view data

openassignment

Publish: when you want the `maintabs` component to open an assignment
Subscribe: when you to know if some component has requested an open assignment

opennewcase

Publish: when you want `maintabs` component to open a new case
Subscribe: when you want to know if some component has requested an open new case

openrecent

Publish: when you want `maintabs` component to open a recent tab (just the tab)
Subscribe: when you want to know if some component has requested an open recent

refreshassignment

Publish: when you want the `workitem` component to refresh the assignment (given data)

Subscribe: when you want to know if some component is requesting a refresh assignment

refreshcase

Publish: when you want the `workitem` component to refresh the case (given data)
Subscribe: when you want to know if some component is requesting a refresh of the case

refreshworklist

Publish: when you want the `worklist` component to refresh the list (calls server)
Subscribe: when you want to know if some component requested a refresh of the worklist

renametab

Publish: when you want `maintabs` component to rename an existing tab
Subscribe: when you want to know if some component requested a tab to be renamed

Components

Components are created that utilize Material IO components as their base and then provide custom functionality. *_subcomponents* are top level components and *_fieldcomponents* are components that are utilized by the `field` component.

_subcomponents

breadcrumb

`breadcrumb` component displays the list of stages along with the current stage for a given case. This information is passed in as the `CurrentCase` input

`breadcrumb` component takes the following inputs:

- `CurrentCase` – JSON object with case information

Definition

```
<app-breadcrumb [CurrentCase] ></app-breadcrumb>
```

caption

The `caption` component represents the LABEL control. LABELs are found a control that can be dropped or the labels of the columns for a grid.

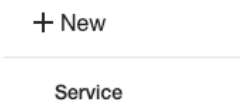
`caption` component takes the following inputs:

- `captionComp` – JSON for a component
- `formGroup` – Angular object for validation

Definition

```
<app-caption [captionComp] [formGroup]></app-caption>
```

createcaselist



`createcaselist` component creates a list of cases that can be created based upon the user (retrieved this info form browser local storage).

`createcaselist` component is a list of buttons. Each button represents a displayable case type. This component subscribes to `getloginstatus` service. When logged in, we then subscribe to `case` service `getCaseTypes`. The response will be a list of case types. We search for those that are displayable (*CanCreate* = *true*). We create an array of these.

The above array is used in the component html to display the buttons.

When a button is clicked, are going to create a new case:

- We check if `requiresFieldToCreate` is *false*. If so, we will skip "New" and call the `case` service `createCase`. This will call the server to create a case.
    - With the response of the created case we:
        - Send an `openassignment` message
        - Send a `refreshworklist` message
- If `requiresFieldToCreate` is *true*, then we don't skip New, so we send an `opennewcase` message

Selecting a case, creates a new case by sending to create a case (skip new) or open new case.

`createcaselist` component registers for `GetLoginStatus` service.

`createcaselist` component sends the following messages:

- `openassignment`
- `refreshworklist`

`createcaselist` component takes the following inputs:

- none

Definition

```
<app-createcaselist></app-createcaselist>
```

field

`field` component is the top level component for all fields, which then, in turn selects the correct field sub component if exists or `nosupport` component if it does not.

`field` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-field [fieldComp] [formGroup] [CaseID] [RefTypes$]></app-
field>
```

group

`group` component is an array of components. This component iterates through the list and calls each corresponding component. A group component can contain the following components:

- `layout`
- `field`
- `caption`
- `view`
- `pararaph`

`group` component takes the following inputs:

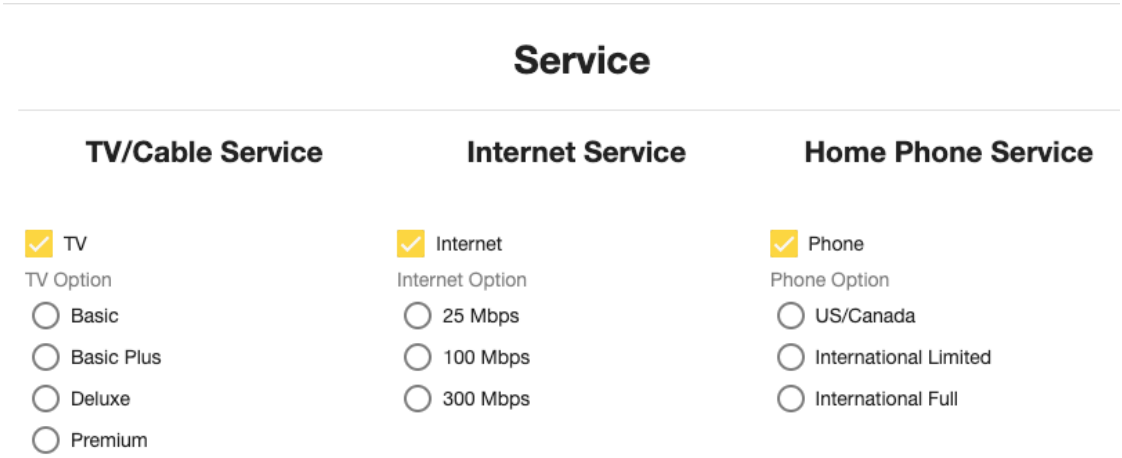
- `group` – JSON for a component
- `formGroup` – Angular object for validation

- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-group [group] [formGroup] [CaseID] [RefTypes$]></app-group>
```

layout



`layout` component determines what kind of layout to apply to the sub components.

`layout` component currently only supports a *WARNINGS* container format. This will make the layout have a muted orange background. This is an example of how a layout can support container formats.

`layout` component supports a header.

`layout` component supports the following layouts:

- Inline grid double
- Inline grid 70 30
- Inline grid 30 70
- Inline grid triple
- Stacked

- Action area
- Mimic a sentence
- Simple list
- Inline middle

Repeating

- REPEATINGROW
- REPEATINGGRID

Screen layouts

- SCREENLAYOUT
    - header
    - header_footer

Screen layout sections

- TOP
- BOTTOM
- CENTER

`layout` component takes the following inputs:

- `layoutComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-layout [layoutComp] [formGroup] [CaseID] [RefTypes$]></app-layout>
```

page

`page` component is used for *Review* and *Confirm*. It contains just a `group` component.

`page` component supports a header.

`page` component registers for `getPage` from `case` service.

`page` component takes the following inputs:

- `pageID` – ID of the page (Confirm, Review, New, etc.)
- `caseID` – case ID, unque for a case

Definition

```
<app-page [pageID] [caseID]></app-page>
```

paragraph

`paragraph` component represents the paragraph control. It uses a pipe component to push HTML into Angular (which normally not allowed).

`paragraph` component takes the following inputs:

- `paragraphComp` – JSON for a component

Definition

```
<app-paragraph [paragraphComp]></app-paragraph>
```

recentlist

Recent list is just an example of how to get a recent list. However, the recent list is form “desktop” usage and not usage via REST apis.

`recentlist` component registers for `GetLoginStatus` service.

`recentlist` component sends the following messages:

- `openrecent`

`recentlist` component takes the following inputs:

- none

Definition

```
<app-recentlist></app-recentlist>
```

repeatinggrid

`repeatinggrid` component represents a table layout (grid). It creates a table with headers for the columns. Headers are `caption` components. It has a default *addRow* and *deleteRow* buttons at the bottom and supports adding *insert/delete* buttons on the rows.

`repeatinggrid` component supports PageLists and PageGroups. For PageGroup, an alert dialog will request the name of the row upon insert/append.

In 8.3, the `repeatinggrid` component supports the *newRow* functionality, along with page instructions (using the `pageinstruction` service.)

`repeatinggrid` component registers for `getactions` service.

`repeatinggrid` component sends the following messages:

- `refreshassignment` – removeRow, addRow
- `pageinstructions` – append, insert, add, delete
- `getchanges` - all

`repeatinggrid` component takes the following inputs:

- `layoutComp` – JSON for a component
- `formGroup` – Angular object for validation
- `CaseID` – case ID (unique for each case)

Definition

```
<app-repeatinggrid [layoutComp] [formGroup] [CaseID]></app-
repeatinggrid>
```

repeatinglayout

`repeatinglayout` component represents a dynamic repeating layout, which repeats a section. Each repeating section is a `view` component. It supports add, insert and delete.

`repeatinglayout` component supports PageLists and PageGroups. For PageGroup, an alert dialog will request the name of the row upon insert/append.

In 8.3, the `repeatinglayout` component supports the *newRow* functionality, along with page instructions (using the `pageinstruction` service.)

`repeatinglayout` component sends the following messages:

- `refreshassignment` – removeRow, addRow

`repeatinglayout` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `CaseID` – case ID (unique for each case)

Definition

```
<app-repeatinglayout [layoutComp] [formGroup] [CaseID]></app-
repeatinglayout>
```

toppage

When there is a full page (Confirm, Review, New, etc.) verses just a flow action, then the `toppage` component is used. `toppage` component takes *CaseID* as input and will pass this to the `group` component.

Upon creation, `toppage` will register for `GetPage` service.

`toppage` component takes the following inputs:

- `CaseID` – case ID (unique for each case)

Definition

```
<app-toppage [CaseID]></app-topppage>
```

topview

`topview` component is where flow action will be displayed. It contains a header and form. The form contains `group` component. `topview` does not have any inputs, instead when created, it registers for messages to receive information/data.

`topview` component registers for `GetView` service.

`topview` component takes the following inputs:

- none

Definition

```
<app-topview></app-topview>
```

view

`view` component represents a section rule.

`view` component takes the following inputs:

- `viewComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label

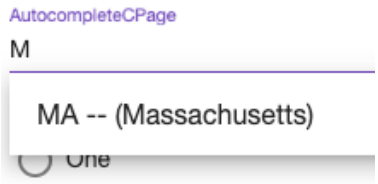
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-view [viewComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-view>
```

_fieldcomponents

autocomplete



`autocomplete` component represents *pxAutoComplete*.

`autocomplete` component uses the following validators:

- required

`autocomplete` component registers for:

- `getDataPage` if using a data page
- `getCase` if clipboard page

`autocomplete` component sends the following messages:

- `getChange`

All actions are handled through `actionsHandler`.

`autocomplete` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-autocomplete [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-autocomplete>
```

button

`button` component represents *pxButton*.

All actions are handled through `actionsHandler`.

`button` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)

Definition

```
<app-button [fieldComp] [formGroup] [noLabel] [CaseID]></app-
button>
```

checkbox

`checkbox` component represents *pxCheckbox*.

`checkbox` component sends the following messages:

- `getChange`

All actions are handled through `actionsHandler`.

`checkbox` component takes the following inputs:

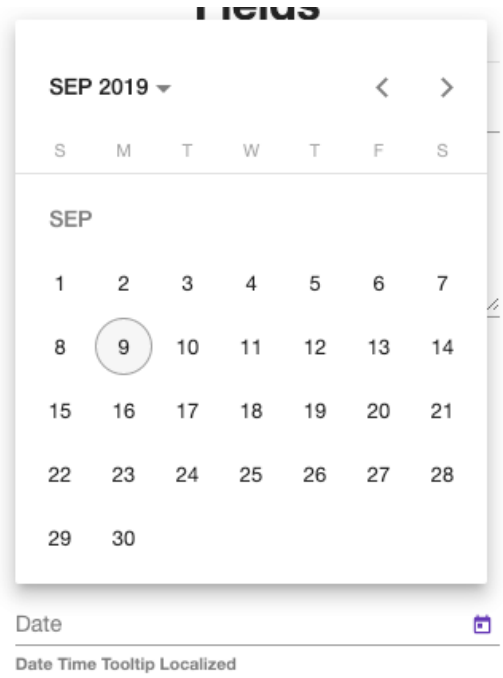
- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label

- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-checkbox [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-checkbox>
```

date



`date` component represents *pxDateTime*.

`date` component uses moment for date formats.

`date` component uses the following validators:

- required

`date` component sends the following messages:

- `getChange`

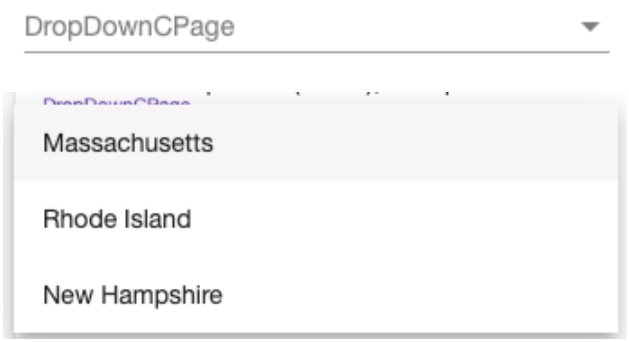
All actions are handled through `actionsHandler`.
`date` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-date [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-date>
```

dropdown



`dropdown` component represents *pxDropdown*.

`dropdown` component uses the following validators:
- required

`dropdown` component registers for:
- `getDataPage` if using a data page
- `getCase` if clipboard page

`dropdown` component sends the following messages:
- `getChange`

All actions are handled through `actionsHandler`.

`dropdown` component takes the following inputs:
- `fieldComp` – JSON for a component

- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-dropdown [fieldComp] [formGroup] [CaseID] [noLabel]
[RefTypes$]></app-dropdown>
```

email

`email` component represents *pxEmail*.

`email` component uses the following validators:
- required
- email

`email` component sends the following messages:
- `getChange`

All actions are handled through `actionsHandler`.

`email` component takes the following inputs:
- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-email [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-email>
```

icon

`icon` component represents *pxIcon*.

All actions are handled through `actionsHandler`.

`icon` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)

Definition

```
<app-icon [fieldComp] [formGroup] [noLabel] [CaseID]]></app-
icon>
```

link

`link` component represents *pxLink*.

All actions are handled through `actionsHandler`.

`link` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)

Definition

```
<app-link [fieldComp] [formGroup] [noLabel] [CaseID]></app-link>
```

nosupport

nosupport component is used when `field` component doesn't have a corresponding component for the given field control.

`nosupport` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)

Definition

```
<app-nosupport [fieldComp] [formGroup] [noLabel] [CaseID]></app-nosupport>
```

number

`number` component represents *pxInteger* and *pxCurrency*.

`number` component uses the following validators:

- required

`number` component sends the following messages:

- `getChange`

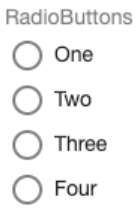
All actions are handled through `actionsHandler`.
`number` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-number [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-number>
```

radio



`radio` component represents *pxRadioButtons*.

`radio` component uses the following validators:

- required

`radio` component registers for:

- `getDataPage` if using a data page
- `getCase` if clipboard page

`radio` component sends the following messages:

- `getChange`

All actions are handled through `actionsHandler`.
`radio` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-radio [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-radio>
```

text

FMT_None
none

FMT_Date
5/4/18

FMT_DateTime
5/4/18 1:48 PM

FMT_Number
555.00

FMT_Text
a quick brown fox

FMT_TrueFalse
False

FMT_Email
a@b.com

FMT_Phone
617-555-1212

FMT_Url
www.pega.com

`text` component represents a blank control or *pxDisplayText.*

`text` component utilizes a lot of functions to generate formatted text. Functions are example of possible formats and are not complete.

`text` component takes the following inputs:

- `fieldComp` – JSON for a component
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-text [fieldComp] [noLabel] [CaseID] [RefTypes$]></app-text>
```

textarea

`textarea` component represents *pxTextArea*.

`textarea` component uses the following validators:

- required

`textarea` component sends the following messages:

- `getChange`

All actions are handled through `actionsHandler`.
`textarea` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-textarea [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-textarea>
```

textinput

`textinput` component represents *pxTextInput, pxPhone* and *pxURL*.

`textinput` component uses the following validators:

- required

`textinput` component sends the following messages:

- `getChange`

All actions are handled through `actionsHandler`.
`textinput` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-textinput [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-textinput>
```

unitdays

`unitdays` component represents *UnitDays*.

`unitdays` component uses the following validators:

- required

`unitdays` component sends the following messages:

- `getChange`

All actions are handled through `actionsHandler`.

`unitdays` component takes the following inputs:

- `fieldComp` – JSON for a component
- `formGroup` – Angular object for validation
- `noLabel` – true, don't show label
- `CaseID` – case ID (unique for each case)
- `RefTypes$` - reference type

Definition

```
<app-unitdays [fieldComp] [formGroup] [noLabel] [CaseID]
[RefTypes$]></app-unitdays>
```

Helpers

_helpers

reference-helpers

Helper functions that are public

getPostContent

Translate a flat list of fields (with a full path) and values to a nested object:

```
{"pyWorkPage.Address.Street": "1 Rogers St" }
```

Translates to:

```
{"Address":
     {"Street":"1 Rogers St"}
}
```

getRepeatFromReference

Returns a PageGroup *object* or PageList *array*, given a property reference.

getBlankRowForRepeat

Returns a blank row (*object* for PageGroup, *array* for PageList), given the PageGroup/List.

getInitialValuesFromView

Get initial state with flat references (page.page.property, etc.)

customUpdateJSON

Update a given JSON object with a key/value pair.

getControlNameFromReference

Given a reference and JSON object, find the control that has this reference and returns its control name (i.e. *pxButton*, etc.)

findObjects

Given a JSON object, name of a property and its value, find all the instances and put in an array called *finalResults*.

removeDataPages

Find all data pages in JSON object and remove them. This is so if you post back state data, there won't be any data pages in the state object.

htmlDecode

Take some HTML and put it in a document element and retrieve the html. The html sent can have html encoded data (i.e. *<, >* etc.) which will be decoded.

updateViewWithLocaleState

Update a given view with state data.

getUniqueControlID

Get a unique control id. The id will be of the form "control-XX", where XX is an incrementing number. All *FormGroups* need a unique id for referencing.

pageinstructions

clearPageInstructions

Clear out *pageInstructions* object

getPageIntructions

Get *pageInstructions* object

addAListInstruction

Adds a *listInstruction* (insert, move, delete, append) with data to the *pageInstructions* object.

getLastInstruction

Retrieve the last instruction added to the *pageInstructions* object.

isLastListInstruction

Determine if given instruction is the last instruction in the *pageInstructions* object.

getLastInstructionContent

Get the data of the last instruction of the *pageInstructions* object.

updateLastInstructionContent

Update the data of the last instruction of the *pageInstructions* object.

addGroupInstruction

Adds a group instruction to the *pageInstructions* object.

isLastGroupInstruction

Detemrine if given instructions is the last instruction in the *pageInstructions* object.

addAnUpdatePageInstruction

Add an “update” instruction to the *pageInstructions* object.

getEmbeddedPageInstruction

Get embedded page from *pageInstructions* object.

_pipe

SafeHtmlPipe

Pipe allows us to insert HTML into an angular component. This is used by the paragraph component. This function can be dangerous, so guard against adding <script> and other bad html to the paragraph.

Angular App functionality

Here we will go through some of the basic flows from a click on an element, through the messages, REST calls, responses and final handling of the events.

Understanding how the components and sub components update themselves

The JSON that is returned from REST calls for `assignments/actions` or `cases/actions` return layout information along with data. The JSON basically of the format:

(assignment)

```
view: {
      groups: [{
            layout: {
                  groups: [{
                        field: {},
                        field: {}
                  }}
            }
      }]
}
```

(new cases)

```
creation_page: {
      groups: [{
            layout: {
                  groups: [{
                        field: {},
                        field: {}
                  }}
            }
      }]

}
```

Given the above JSON, the corresponding sub components (page, view, group, layout, field) match the JSON. Each component uses the current JSON node and then the sub nodes get passed to the appropriate sub component.

So, for example if we start with a view (following the above JSON)

1. A `view` can contain a group (both JSON and view.html show this)
2. The view.ts takes view JSON and gets the groups array and passes this to the group component as `groups$`.
3. The `group` sub component can contain `layout`, `field`, `caption`, `view` and `paragraph` sub components. This is show in group.html and matches the JSON possibilities.
4. The `group` JSON element is an array. So, the group.html iterates over the array, each element becoming `groupComp`.
5. `groupComp` is passed to the corresponding sub components that exist in the `group`. In the above example, `groupComp` will be passed to `field` sub component.
6. `field` sub component then determines which type of field component will be used and passes that data as `fieldComp`.

Navigation

Subscription

Navigation subscribes to following messages:

- `getloginstatus` – will do the following:
    - if "LoggedIn", store the user/password in local storage
    - else, clear local storage

Create Case List

Contains a list of case types to be created

Subscription

`createcaselist` subscribes to following messages:

- `getloginstatus` – will do the following:
    - if logged in, call `case` service method `getCaseTypes`, response will:
        - display case types that "can be created"

In the list, those that have *CanCreate* as *true* can be used to create new cases. The field `requiresFieldToCreate` determines whether we use New (*true*) or Skip New (*false*).

See “New Case” for when a case above has been selected.

WorkList Panel

Contains the worklist and workbasket drop down (if needed)

Workbasket

Workbasket dropdown, when exists and is changes will do the following:

- Publish `refreshworklist` message with selected workbasket

Work List

Retrieving a work list, the application makes a REST call to data page `D_Worklist`. The response is sent to the Material Table `MatTableDataSource` called `worklists$`.

Subscription

Worklist subscribes to the following messages:

- `refreshworklist` – will do the following:
    - if *work* || *worklist*
        - Calls `datapage` service method `getDataPage`, retrieving “D_Worklist”, response will:
            - Push the results to `worklist$` (will redraw the Material table)

- Else
  - Calls `datapage` service method `getDataPage`, retrieving "D_WorkBasket" with a parameter of which workbasket, response will:
    - Push the results to `worklist$` (will redraw the Material table)

Opening a work item

1. Attached to the `<TR>` of the worklist rows, we have a `click` call to `openAssignment`.
2. This in turn will Publish an `openassignment` message.
3. `maintabs` component Subscribes to `openassignment` message, and receives this message, and creates a new tab and selects the tab.
4. With the new tab, there is an associated `workitem` component that is created, and it Subscribes to `getassignment` message.
5. Once the tab is created, it will Publish `getassignment` message.
6. `workitem` will receive the `getassignment` message and call the `assignment` service method `getAssignment`, which does a REST call to get the assignment. The return data will tell us the next assignment action and will do another `assignment` service method `getFieldsForAssignment`. The response for this will be view layout with embedded view data.
7. Unsubscribe to `getAssignment`, so won't receive any more messages.
8. `workitem` will take the view data and Publish a `getview` message.
9. `workitem` will also get the embedded view data and update local state, as the `workitem` is in charge of local state that will be POSTed to the server later on.
10. Any validation messages will be pushed to the Material `snackBar` display.
11. `workitem` contains `topview` component.
12. `topview` Subcribes to `getview` message and receives the message and populates local data with view and group information. `topview` contains a `group` component.
13. This `group` component takes the given information and proceeds to traverse the message tree for subcomponents. Each subcomponent does the same, down to `field` components and then the display is finished.

New Case

New

From the navigation component, you can create a new case. If the case does NOT skip “new”, will follow this path.

1. From `navigation` component, selecting a “+New” case, this is in the component `createcaselist`. This is a list of cases types for this application. Selecting a *casetype* Publishes `opennewcase` message.
2. `maintabs` component Subscribes to `opennewcase` message, and receives this message, and creates a new tab and selects the tab.
3. With the new tab, there is an associated `workitem` component that is created and Subscribes to `getnewcase` message.
4. With the new tab create, it Publishes a `getnewcase` message.
5. `workitem` will receive the `getnewcase` message and call the `case` service method `getCaseCreationPage`, which does a REST call to get the case. The return data is the case data, which will store the data as local state. Then Publishes `getpage` message.
6. Unsubscribe to `getnewcase`, so won’t receive this message again.
7. `workitem` contains the `page` component.
8. `page` component Subscribes to `getpage` message, when it receives that message will load the page component with the data. Page component contains a `group` component. The `page` data will update the `group` component.
9. The `group` component takes the given information and proceeds to traverse the message tree for subcomponents. Each subcomponent does the same, down to `field` components and then the display is finished.

Note that this is page, so the buttons at the bottom of the display will be **Create** and **Cancel** found in `workitem`.

Skip new

From the navigation component, you can create a new case. If the case DOES skip "new", will follow this path.

1. From `navigation` component, selecting a "+New" case, this is in the component `createcaselist`. This is a list of cases types for this application. Selecting a casetype calls the `case` service method `createCase`. The REST response will be used to Publish to `openassignment` message and to `refreshworklist` message.
2. Once the `openassignment` has been published, the subscriber will be same as **Step 3** in "opening a work item" (above) and there forward.
3. In addition, because we created a new work item, the `worklist` component will receive the `refreshworklist` message and redraw, getting a new work item

Main Tabs

Subscription

`maintabs` component subscribes to the following messages:

- `openassignment` – will do the following:
  - add a new tab if needed and select it
- `closework` – will do the following:
  - remove the indicated tab, change selection
- `opennewcase` – will do the following:
  - add a new tab of name "New"
- `renametab` – will do the following:
  - find given tab and rename it
- `openrecent` – will do the following:
  - add a new tab if needed and select it

Work Item

Subscriptions

`workitem` subscribes to the following messages:

- `getassignment` – will do the following:
    - Calls the `assignment` service method `getAssignment`, the response will get the next assignment.
        - If assignment
            - Call `assignment` service method `getAssignment`, the response will:
                - Update some local variables
                - Check for errors, show `snackBar` if there are some
                - Call `assignment` service method `getFieldsForAssignment`, the response will:
                    - Update some local variables
                    - Update state
                    - Turn off progress indicator
                    - Publish `getview` message (with view data)
                    - Update the action drop down
                    - Call `case` service method `getCase`, the response will:
                        - Captures etag
                        - Update local varialbes
                        - Publish `getcase` message (with case data)
            - Publish message `refreshcase` (this is to update case details)
        - If page (confirm/review)
            - Call `case` service method `getCase`, the response will:
                - Captures etag
                - Update local variables
                - Publish `refreshworklist` message
                - Publish `getcase` message (with page data)
            - Publish `refreshcase` message (this is to update case details)
        - Else error
    - Unsubscribe from `getAssignment` message, so won't receive any more

- `getchanges` – will do the following:

    - if current caseID matches, update state with provided data
- `getactions` – will do the following:
    - handles actions (see Actions)
- `refreshassignment` – will do the following:
    - handles add and remove rows for Repeating Grids
- `getnewcase` – will do the following:
    - Calls case `service` method `getCaseCreationPage`, the response will:
        - Update local variables
        - Turn off progress indicator
        - Publish `getpage` message
        - Update local state with initial values
    - Unsubscribe from `getnewcase` message, so won't receive this message again
- `getrecent` – will do the following:
    - Calls `case` service method `getCase`, the response will:
        - If have assignments
            - See "getassignment", above
        - If not, then Review
            - Set up local variables, will trigger drawing of Page view
    - Unsubscribe from `getrecent` message, so won't receive this message again

Create

Create button is available when this is a new page (new case). Create button will do the following:

1. Calls the `case` service method `createCase`. The response will Publish 3 messages:
    a. Message `renametab`, sending "New"
    b. Message `openassignment` (getting an assignment from `createCase` response)
    c. Message `refreshworklist`, creates a temp entry called Work.
2. This in turn will update the worklist, a new tab will created. Inside a workitem will be created, that will receive the `getassigment` message (and continues with `workitem` - `getassignment` above.)

Submit

Submit button is available when an assignment can be moved forward in the flow. Submit button will do the following:

1. Check if form is valid
2. Turn progress indicator on
3. Clean up state
4. Call the `assignment` service method `performActionOnAssignment`, posting information and state. The response will:
    a. See `workitem` – `gettassignment` (above)

Cancel

Currently cancel closes the work item (in the future it should go to the **Review** harness)

Cancel will do the following:

1. Publishes message `closework`.

Save

Save button is available when the "Submit" button is available. Save button will do the following:

1. Check if form is valid
2. Call the `case` service method `updateCase`, posting information and state. The response will:
    a. Send a `refreshcase` message.

TopView

Main element in the `workitem`, when the `workitem` is an assignment. It contains the left `view` and the right `casedetails` sub components. The analogy of this element is a hard coded **Perform** harness.

Subscriptions

`topview` subscribes to the following messages:

- `getview` – will do the following:
    - sets up some local variables
    - searches the message for the "view" data object from the message. Finds the "groups" array and sets it to `groups$`. This will begin the cascade of redrawing the display, from **Step 2** in "Understanding how components and sub components update themselves".

TopPage

Main element in the `workitem`, when the `workitem` is a page (Confirm/Review/New). This will represent a harness.

Subscriptions

`toppage` subscribes to the following messages:

- `getpage` – will do the following:
    - update some local variables
    - searches "page" data object from the message. Finds the "groups" array and sets it to `groups$`. This will begin the cascade of redrawing the display, from **Step 2** in "Understanding how components and sub components update themselves".

Case Details

On **Perform** harness (which is not supported in PegaAPI DX) for case worker, we see a case details section. In the AngularComponentApp we are showing this as an example of how to put sections (views) together to mimic the **Perform** harness.

Subscriptions

`casedetails` component subscribes to the following messages:

- `refreshcase` – will do the following:
    - calls `case` service method `getView`, the response will:
        - updated the local `groups$` which will update the case details display