Customization of the IBM Case Manager 5.2.1 To-do list widget and Task property features

Type of Submission: Article

Title: Customization upon IBM Case Manager 5.2.1 - To-do list widget and Task property feature

Keywords: IBM Case Manager Client, Action Customization, To-do List widget

Author: Huo, Jia

Job Title: Senior Software Engineer, IBM Case Manager Solution Development

Email: huojia@us.ibm.com

Bio: Solution Architect

Company: IBM

Author: Zhao, Hong Dong

Job Title: Software Engineer, IBM Case Manager Solution Development

Email: zhaohd@us.ibm.com

Bio: Developer of Case Manager Solution

Company: IBM

Author: Zheng, Suo Shi

Job Title: Software Engineering Manager, Case Manager and Enterprise Content Management

Email: sszheng@us.ibm.com

Bio: Development Manager of IBM Case Manager

Company: IBM

Author: Guo, Ming Liang

Job Title: Senior Software Engineer, Case Manager and Enterprise Content Management

Email: guoml@cn.ibm.com

Bio: Architect of the IBM Case Manager Client.

Company: IBM

Author: Gao, Ying Ming

Job Title: Advisory Software developer, IBM Case Manager Client

Email: guoyingm@cn.ibm.com

Bio: Developer of the IBM Case Manager Client action framework.

Company: IBM

1. An Introduction to the To-Do List and Task Properties features

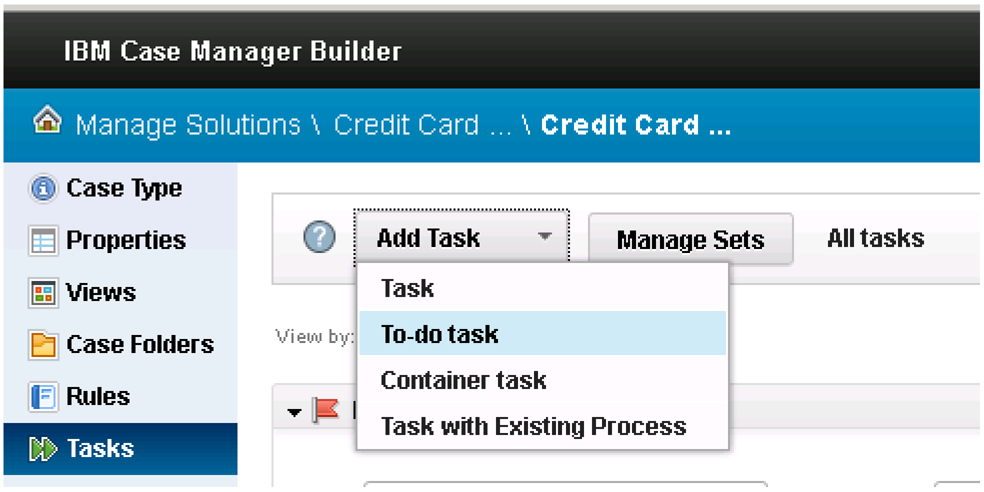
1.1 To-do Task

Tasks, and their unique relationship with cases, are a key capability of IBM Case Manager. Prior to ICM 5.2.1, a task object had one corresponding workflow and some predefined system state properties.

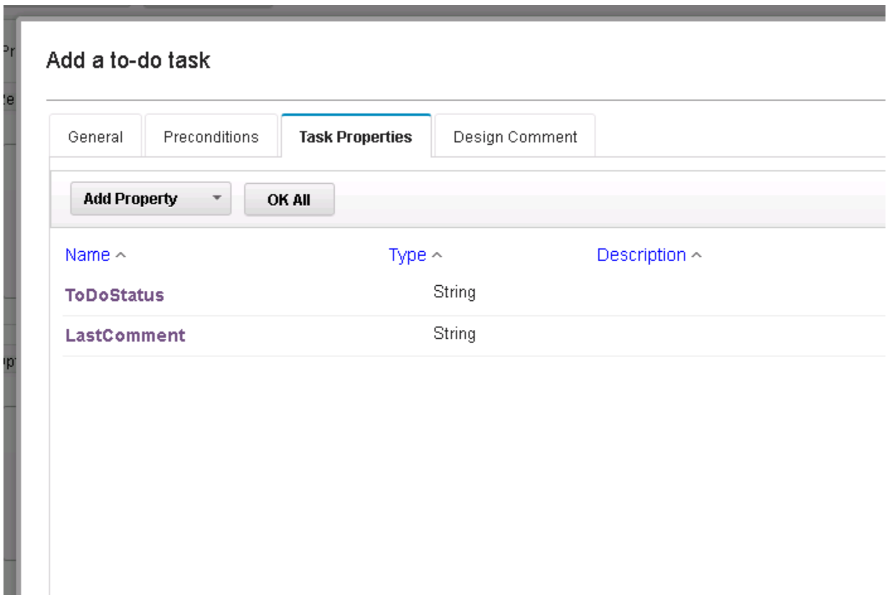
This article discusses the new type task named "To-Do" task, introduced in IBM Case Manager 5.2.1. A To-Do task is based on the existing case tasks and inherits many of its behaviors but doesn't have an associated workflow. Using the other new 5.2.1 feature for tasks, Task Properties, you define properties for a specific To-Do task as you would for a Case Type. These task properties are persisted on the task object itself, removing the need to store non case data on the case object or off in some custom location.

To-do tasks, which or without additional task properties, can be used to track the completion of simple tasks that don't need a specific process flow, or to record some extra data. For example, we can use a To-do task to track if a contract has been signed for a case, or that a call to a customer has been made. Another example is to use the To-do task view to divide, organize and track the data captured in Task or Case properties. A new To-do List widget has been introduced in ICM 5.2.1 to display the To-do tasks at runtime.

You can add a To-do task when adding Tasks in Case Builder as shown below.



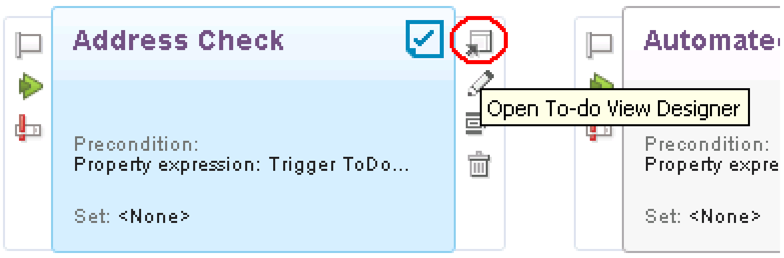
You then define the general information and the preconditions just like you do for any other task type. But now, we can also add properties to a To-do task as shown below where we added ToDoStatus and LastComment as Task Properties for our new To-do task.

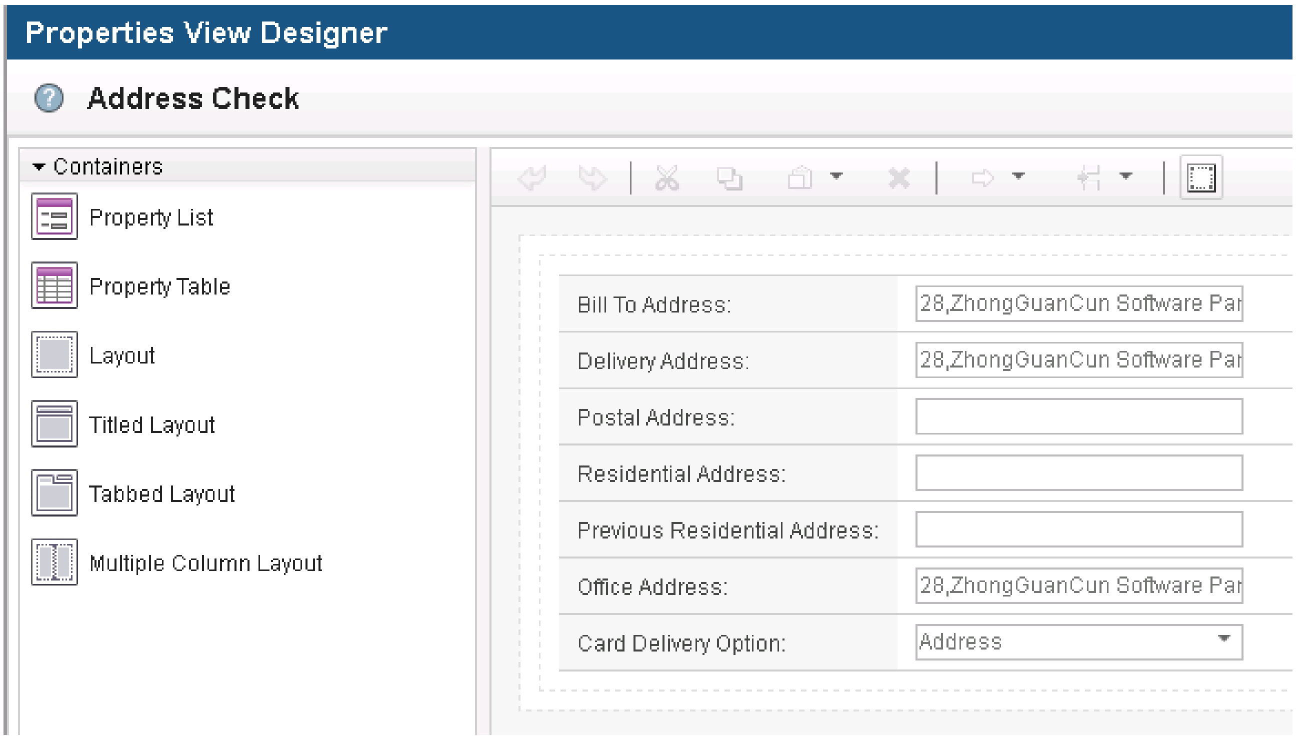


Add a to-do task

Name	Type	Description
ToDoStatus	String	
LastComment	String	

In addition to the default property view that displays a To-do's properties, each To-do task can also have a customized property view created using the View Designer. Both To-do task properties and case properties can be added to the same view in View Designer. The associated view will be displayed in To-do list widget when you open a To-do task.





Bill To Address:	28,ZhongGuanCun Software Par
Delivery Address:	28,ZhongGuanCun Software Par
Postal Address:	
Residential Address:	
Previous Residential Address:	
Office Address:	28,ZhongGuanCun Software Par
Card Delivery Option:	Address

Using the steps above, you can add one or more To-do tasks to your case type.

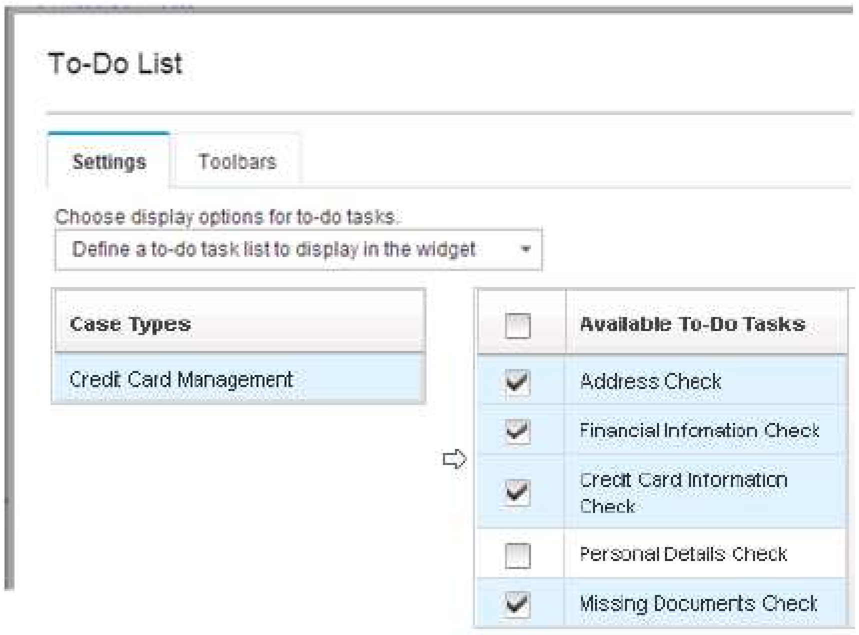
Let's now take a look at To-do List widget itself.

1.2 To-do List widget

The To-do List widget is designed to display To-do task list for a case, and allow the user to perform actions on them. You add a To-do list widget on a page in Page Designer then optionally make additional configurations.

1.2.1 Widget Configuration

By default, the To-do List widget displays all To-do tasks associated with a case. You can optionally configure what To-do tasks should be displayed for a given case type, allowing you to restrict a set of To-do tasks to a certain role or page.


To-Do List

Settings | Toolbars

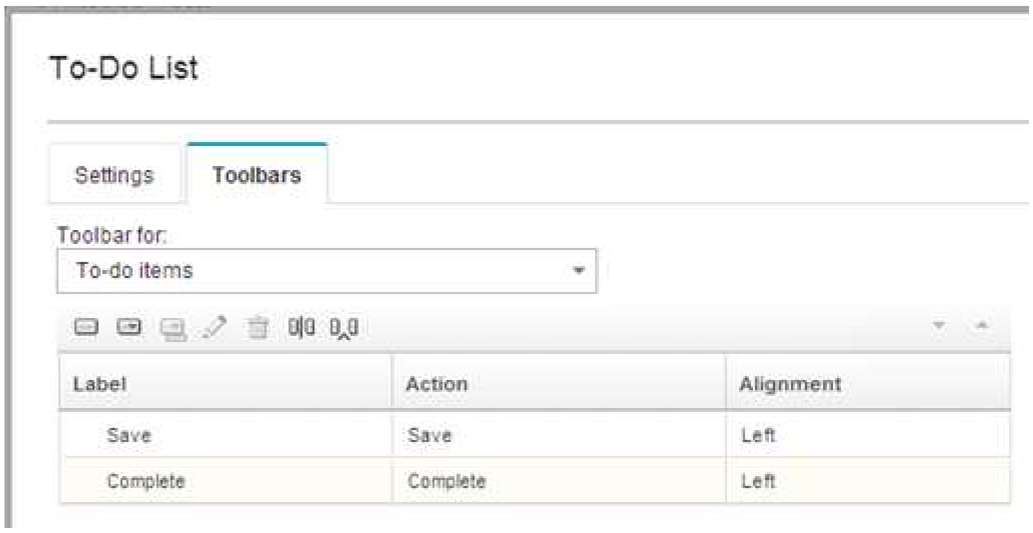
Choose display options for to-do tasks:
Define a to-do task list to display in the widget

Case Types
Credit Card Management

☐	Available To-Do Tasks
✓	Address Check
✓	Financial Infomation Check
✓	Credit Card Information Check
☐	Personal Details Check
✓	Missing Documents Check

You also can configure the toolbars for both the entire To-do list and for each To-do item. The toolbar of To-do list has two default actions: Add To-do task and Refresh. Add allows the user to add on the fly, any To-do tasks that were designed as discretionary.

There are also seven actions for each to-do task: Close, Complete, Disable, Enable, Restart, Save, and Start.
In case those out-of-the-box actions cannot meet your business requirements, you can also develop your custom actions. We'll introduce action customization at section 3.


To-Do List

Settings | Toolbars

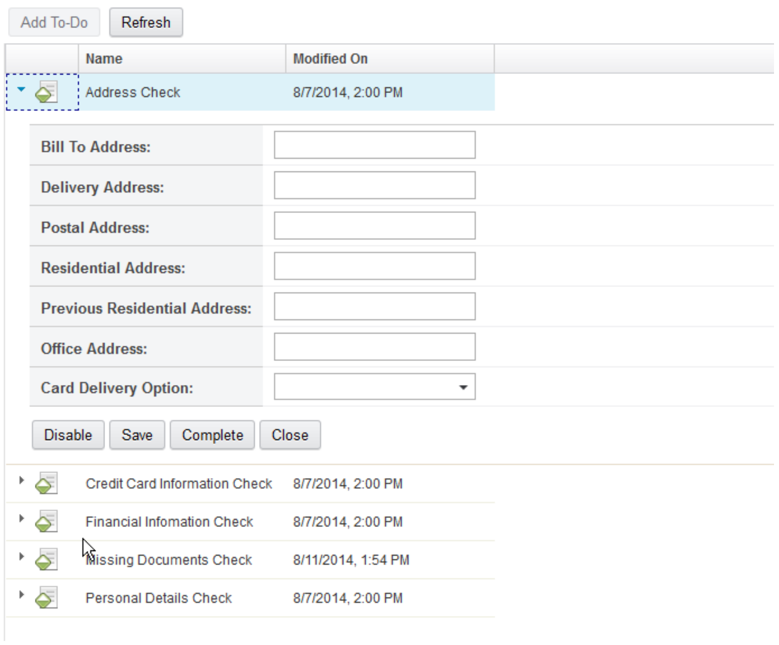
Toolbar for:
To-do items

Label	Action	Alignment
Save	Save	Left
Complete	Complete	Left

1.2.2 Widget Runtime

The To-do List widget displays the To-do tasks of a given case. By default it displays all non-hidden To-do tasks that are in a 'Working', 'Completed' or 'Failed' state. Each To-do task is displayed as row in the list.

Expanding the row displays the task and any properties it might have using either the default or the configured view layout. It also displays the action buttons for that To-do as well. The following screen shows the properties view that we defined for Address Check task.



Name	Modified On
Address Check	8/7/2014, 2:00 PM

Bill To Address:	
Delivery Address:	
Postal Address:	
Residential Address:	
Previous Residential Address:	
Office Address:	
Card Delivery Option:	

Name	Modified On
Credit Card Information Check	8/7/2014, 2:00 PM
Financial Infomation Check	8/7/2014, 2:00 PM
Missing Documents Check	8/11/2014, 1:54 PM
Personal Details Check	8/7/2014, 2:00 PM

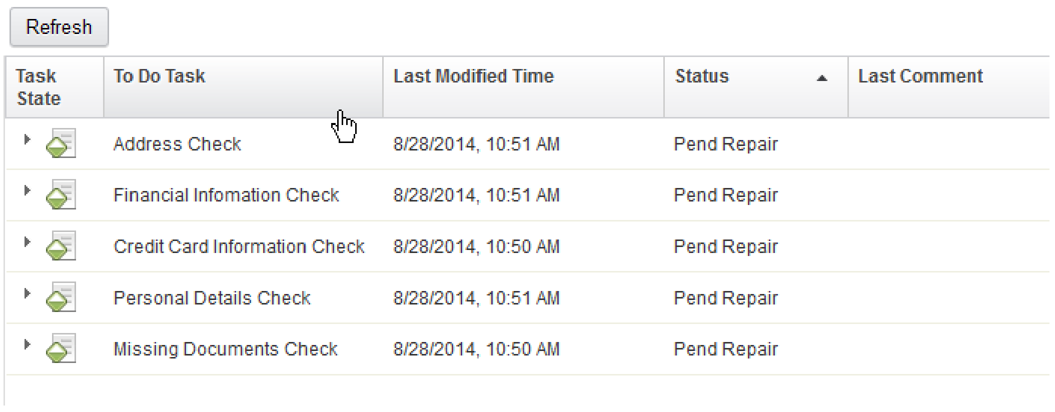
2. To-Do List Customization by using Script

You can use a script to customize the look of your To-Do list.

1. Show important task properties on To-Do list to give more business context of To-do
2. Filter To-do with criteria to just return a set of particular To-do items
3. Sort on specified properties to prioritize the To-do list based on a business requirement

We're going to create script to customize To-do list to show additional task properties including 'Last Modified Time', 'Status' and 'Last Comment'. Status is a task property, which is intended to capture the business status of the To-do. We will also define a filter on the 'Status' property to only allow a business Status of 'Pend Repair', which is a choice list item, to be shown in the list. We also want to sort on 'Task State' to always put incomplete To-do items at the top of the list.

With above customization, the to-do list would look like the following:



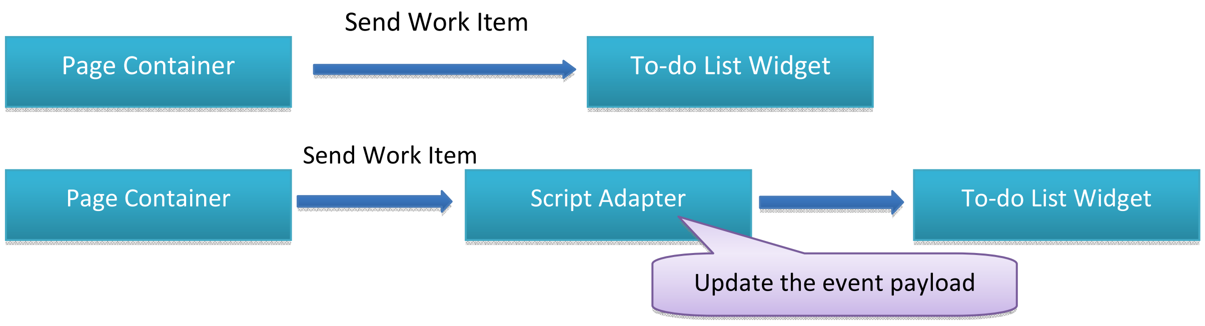
Refresh

Task State	To Do Task	Last Modified Time	Status	Last Comment
	Address Check	8/28/2014, 10:51 AM	Pend Repair	
	Financial Infomation Check	8/28/2014, 10:51 AM	Pend Repair	
	Credit Card Information Check	8/28/2014, 10:50 AM	Pend Repair	
	Personal Details Check	8/28/2014, 10:51 AM	Pend Repair	
	Missing Documents Check	8/28/2014, 10:50 AM	Pend Repair	

Figure –Custom To-do List

On the standard Work Details page, the Page Container sends a "Send Work Item" broadcast event to every widget on the page, including the new To-do List widget. For our customization, a script is used to intercept communication between Page Container and To-do List widget.

You also need disable the Page Container's broadcast event "Send Work Item" and wire the Page Container to the Script Adapter and then to the To-do widget so that the To-do only gets our modified event. Since you disabled the broadcasting of the event, you must also wire it to the other page widgets manually to ensure the event gets to them.



Let's see how you make this happen.

1. Drag a Script Adapter widget into the hidden section of a Work Details page which also contains a To-do list widget, and then you copy below script into script adapter.

```
varnewPayload = {};
newPayload.workItemEditable= payload.workItemEditable;
newPayload.coordination = payload.coordination;
newPayload.propertyFilter = "([CmAcmIsToDo] = true AND CCM_ToDoStatus ='4')";
newPayload.resultDisplay  = {sortBy: "TaskState", sortAsc: true};
newPayload.gridStructure = { "cells": [ [{
                              "width": "4.6em",
                              "sortable": true,
                              "field": "TaskState",
                                        "name":"Task State "
                           },{
```

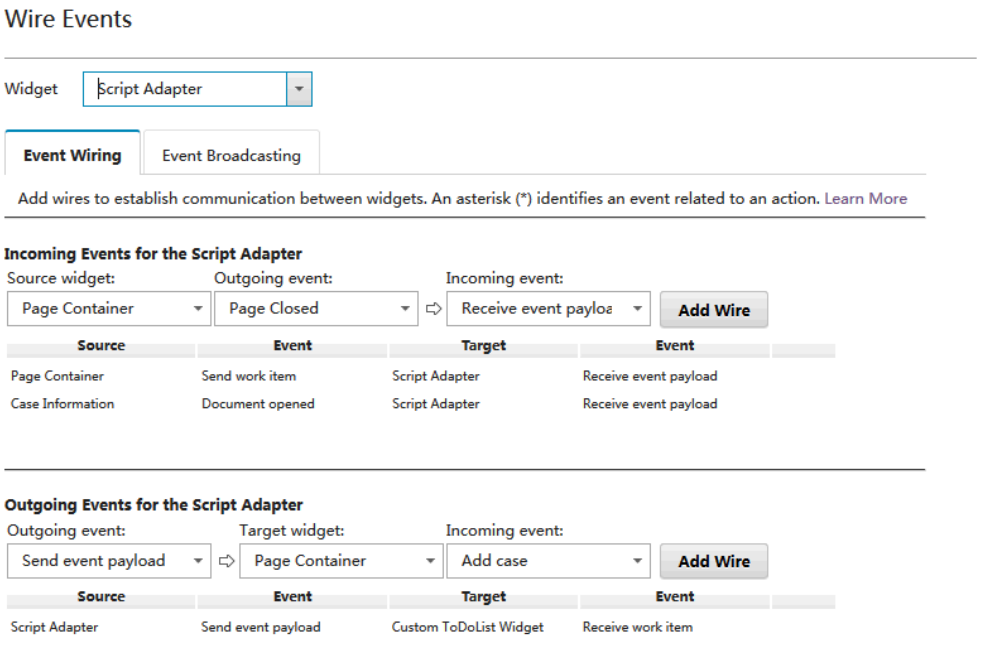
```
                        "width": "15em",
                        "sortable": true,
                        "field": "CmAcmTaskName",
                    "name":"To Do Task"
                     }, {
                        "width": "15em",
                        "sortable": true,
                        "field": "DateLastModified",
                    "name":"Last Modified Time"
                     }, {
                        "width": "15em",
                        "sortable": true,
                        "field": "CCM_ToDoStatus",
                    "name":" Status"
                     }, {
                        "width": "25em",
                        "sortable": true,
                        "field": "CCM_LastComment",
                    "name":" Last Comment"
             } ]]
               };
returnnewPayload;
```

In the script, we create a new event payload to pass on to the To-do widget

- "propertyFilter" to filter the To-do tasks, such as "([CmAcmIsToDo] = true AND CCM_ToDoStatus ='4')"
- " resultDisplay" to sort the To-do task list, for example, resultDisplay = {sortBy: "TaskState", sortAsc: true} to sort on TaskState
- "gridStructure" to display important task properties. You can also specify column width, sortable and change the name of property. You need use the symbolic name of task property for "field". (NOTE – make sure when you copy from an existing solution to use new solution prefix, the property symbolic name would be changed. Using the getPrefix() function can help make it more generic)

2. Now you can wire the "Send Work Item" event between "Page Container" and the "Receive event payload" on the "Script Adapter" as the incoming Events for Script Adapter, then wire "Receive work item" event on the "To-do List" Widget to the outgoing events from the Script Adapter.

develop


Wire Events

Widget: Script Adapter

Event Wiring | Event Broadcasting

Add wires to establish communication between widgets. An asterisk (*) identifies an event related to an action. Learn More

Incoming Events for the Script Adapter

Source widget: Page Container; Outgoing event: Page Closed; Incoming event: Receive event payloa; Add Wire

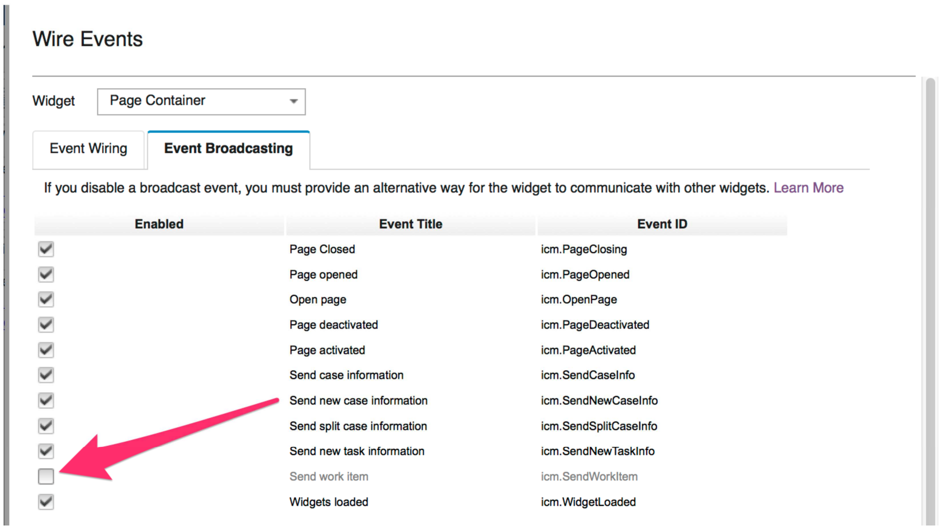
Source	Event	Target	Event
Page Container	Send work item	Script Adapter	Receive event payload
Case Information	Document opened	Script Adapter	Receive event payload

Outgoing Events for the Script Adapter

Outgoing event: Send event payload; Target widget: Page Container; Incoming event: Add case; Add Wire

Source	Event	Target	Event
Script Adapter	Send event payload	Custom ToDoList Widget	Receive work item

3. Since page container broadcasts the “send work item” event, you need disable the broadcast event “send work item” on this page to prevent our To-do from also receiving the event that way.


Wire Events

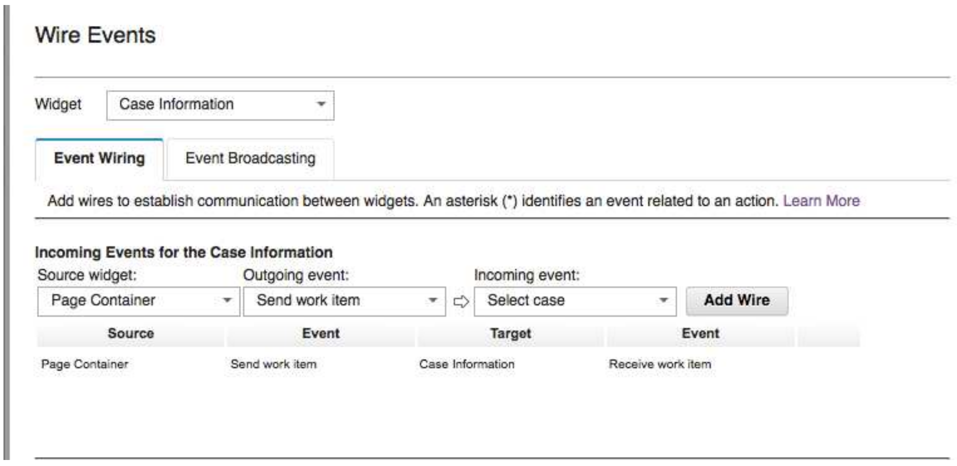
Widget: Page Container

Event Wiring | Event Broadcasting

If you disable a broadcast event, you must provide an alternative way for the widget to communicate with other widgets. Learn More

Enabled	Event Title	Event ID
✓	Page Closed	icm.PageClosing
✓	Page opened	icm.PageOpened
✓	Open page	icm.OpenPage
✓	Page deactivated	icm.PageDeactivated
✓	Page activated	icm.PageActivated
✓	Send case information	icm.SendCaseInfo
✓	Send new case information	icm.SendNewCaseInfo
✓	Send split case information	icm.SendSplitCaseInfo
✓	Send new task information	icm.SendNewTaskInfo
☐	Send work item	icm.SendWorkItem
✓	Widgets loaded	icm.WidgetLoaded

4. You then need to wire other page widgets, which need to receive “send work item” event, with the Page Container. For example, you can wire Page Container and Case Information like below screenshot.


Wire Events

Widget: Case Information

Event Wiring | Event Broadcasting

Add wires to establish communication between widgets. An asterisk (*) identifies an event related to an action. Learn More

Incoming Events for the Case Information

Source widget: Page Container; Outgoing event: Send work item; Incoming event: Select case; Add Wire

Source	Event	Target	Event
Page Container	Send work item	Case Information	Receive work item

3.To-Do Action Customization

By default, the To-Do List Widget provides a set of actions, such as complete, disable, start, save. In a real project, you may want to customize the actions to have you own logic to implement a specifc reuirement. We're going to show you how to customize an action to meet the following business requirement, using the To-do task created in the previous section of this article.

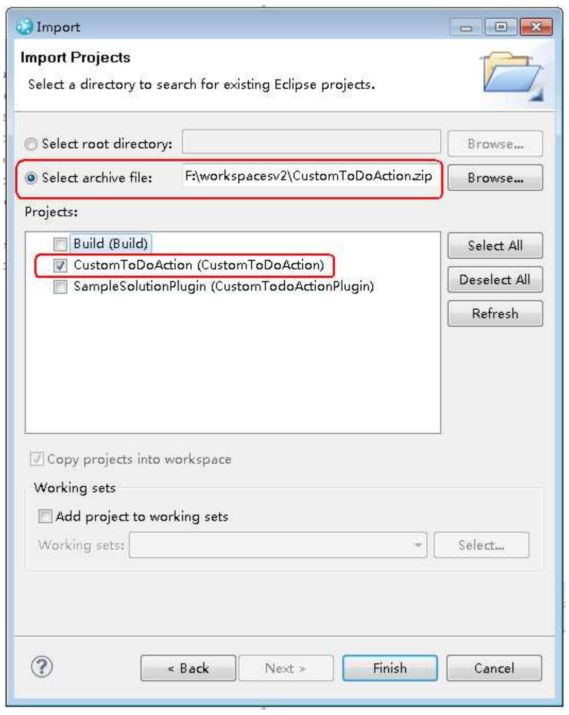
1. Custom Complete To-do Action
   a) In the event of a data change on the task view, it will pop up a dialog to require a mandatory case comment to explain the reason for change.
   b) Change To-do business status (CCM_ToDoStatus) to either 'Pend Review' or 'Reviewed' based on current status.
   c) Change To-do task state (TaskState) to completed.
2. Custom Follow Up To-do Action
   a) Always pop up a dialog to require a mandatory case comment to explain the reason for follow up.
   b) Change To-do business status (CCM_ToDoStatus) to 'Follow Up' regardless current status.
   c) Change To-do task state(TaskState) to completed.

3.1 Creating Case Manager Custom Widget Project

IBM Case Manager Client is built on top of the IBM Content Navigator framework. You should be familiar with the concepts and programing models of IBM Content Navigator and Case Manager Client. To customize the To-do action, you need to create a custom widget project. We provide a sample project for you to start quickly.

3.1.1 Import Custom Todo Action project

You can create a Java project in Eclipse, then import achive file CustomTodoAction.zip, but you just choose CustomTodoAction in the Projects list.



Import

Import Projects

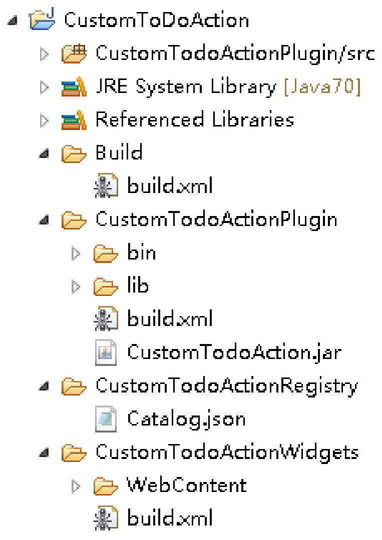
Select a directory to search for existing Eclipse projects.

Select root directory: [✓ Select archive file:] F:\workspacesv2\CustomToDoAction.zip

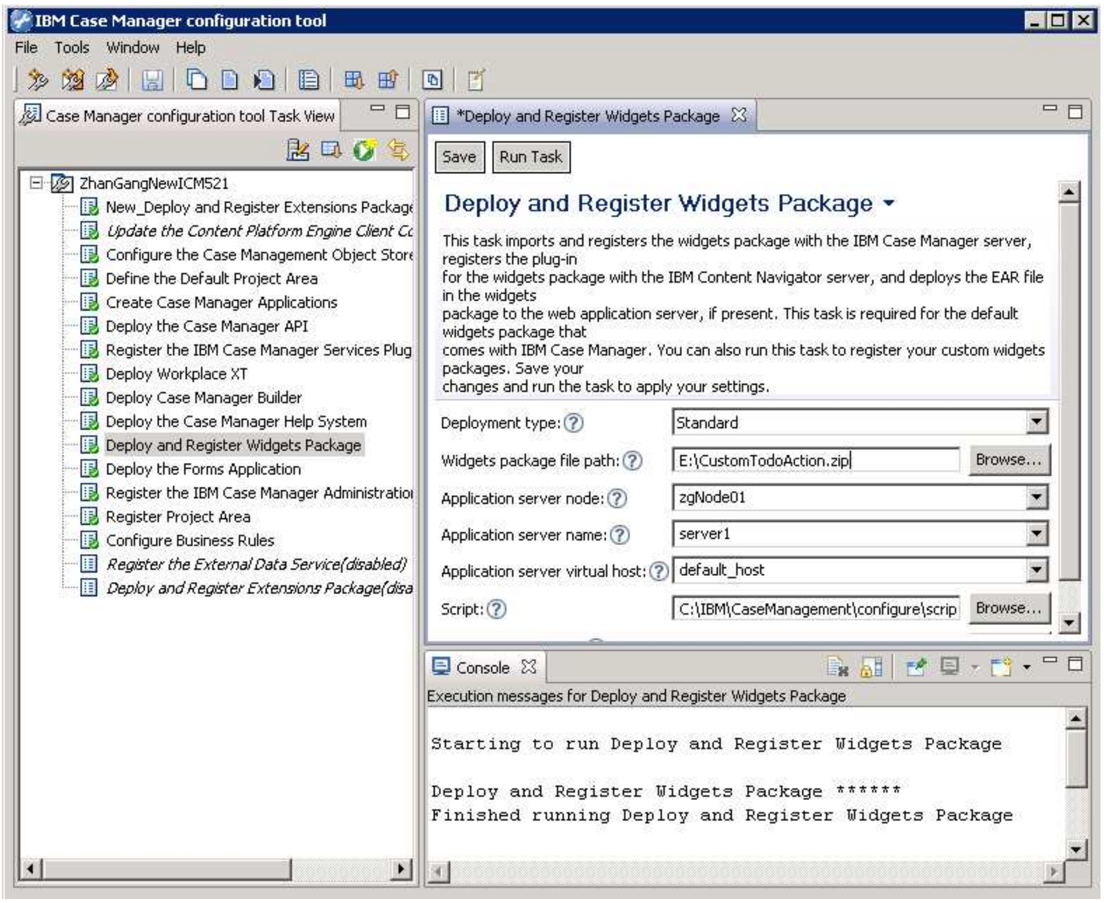
Projects:

- [ ] Build (Build)
- [x] CustomToDoAction (CustomToDoAction)
- [ ] SampleSolutionPlugin (CustomTodoActionPlugin)

Once you import the source code, you can find it contains IBM Content Navigator Plugin, Custom Widgets, and Custom Widgets Registration. The diagram below shows the files structure of sample project.



```
CustomToDoAction
    CustomTodoActionPlugin/src
    JRE System Library [Java70]
    Referenced Libraries
    Build
        build.xml
    CustomTodoActionPlugin
        bin
        lib
        build.xml
        CustomTodoAction.jar
    CustomTodoActionRegistry
        Catalog.json
    CustomTodoActionWidgets
        WebContent
        build.xml
```


IBM Case Manager configuration tool
File Tools Window Help
Case Manager configuration tool Task View
ZhanGangNewICM521
New_Deploy and Register Extensions Package
Update the Content Platform Engine Client Co
Configure the Case Management Object Store
Define the Default Project Area
Create Case Manager Applications
Deploy the Case Manager API
Register the IBM Case Manager Services Plug
Deploy Workplace XT
Deploy Case Manager Builder
Deploy the Case Manager Help System
Deploy and Register Widgets Package
Deploy the Forms Application
Register the IBM Case Manager Administratio
Register Project Area
Configure Business Rules
Register the External Data Service(disabled)
Deploy and Register Extensions Package(disa
*Deploy and Register Widgets Package
Save
Run Task
Browse...
Console

Deploy and Register Widgets Package

This task imports and registers the widgets package with the IBM Case Manager server, registers the plug-in
for the widgets package with the IBM Content Navigator server, and deploys the EAR file in the widgets
package to the web application server, if present. This task is required for the default widgets package that
comes with IBM Case Manager. You can also run this task to register your custom widgets packages. Save your
changes and run the task to apply your settings.

Deployment type: Standard
Widgets package file path: E:\CustomTodoAction.zip
Application server node: zgNode01
Application server name: server1
Application server virtual host: default_host
Script: C:\IBM\CaseManagement\configure\scrip

Execution messages for Deploy and Register Widgets Package

```
Starting to run Deploy and Register Widgets Package

Deploy and Register Widgets Package ******
Finished running Deploy and Register Widgets Package
```

3.1.2 Package and Deploy Custom To-do Action

You can execute CustomToDoAction/Build/build.xml , it will build widget package CustomTodoAction.zip. Then you can deploy the package into IBM Case Manager through IBM Case Manager configuration tool. You should run the "Deploy and Register Widgets Package" task by specifying the "Widgets package file path". You can find custom actions in Case Builder once the task is executed successfully.

3.2 IBM Content Navigator Plug-in of Custom Case Manager Widget

To build custom IBM Case Manager widgets, you need to create an IBM Content Navigator plug-in. As this article focuses on action customization, we just modify CustomTodoActionPlugin.java file to provide the initial JavaScript file name in the method getScript() and custom action list in the method getActions(). Then we add Java class CustomCompleteToDoTaskAction for custom Complete action and CustomFollowUpToDoTaskAction for custom Follow Up action respectively. We’ll introduce action classes in detail later.

```
@Override
publicPluginAction[] getActions() {
```

```
return new PluginAction[] {
newCustomCompleteToDoTaskAction(),
newCustomFollowUpToDoTaskAction()
};
}

@Override
public String getScript() {
return "CustomTodoActionPlugin.js";
}
```

Besides Plug-in class (CustomTodoActionPlugin) and custom action classes (CustomCompleteToDoTaskAction and CustomFollowUpToDoTaskAction), there are a few files also good know.

- **com.ibm.icm.extension.sample.NLSResource.java** is a utility class to support national language support for the plug-in name or other names used in the custom plug-in.
- **CustomTodoActionPlugin.properties** contains the message resources.
- **CustomTodoActionPlugin.js** is the initial JavaScript file for the CustomTo-DoAction plug-in. This initial JavaScript file performs the following two steps to bootstrap the Custom To-Do Action widget at runtime: firstly it load the CSS files used by the custom Action or Page Widgets, and then register the Dojo module path **customTodoAction** for the CustomTo-DoAction code.
- **MANIFEST.MF** is used to specify the plug-in class of CustomTo-DoAction plug-in. You can also include any of the standard properties for a JAR file in theMANIFEST.MF file.

3.3 Custom To-Do ActionRegistry File

The Catalog.json file provides a description of your custom widget and lists the widgets that the package contains. We just simply define the information below so that the custom action can be registered through Case Manager Administrative Console.

```
{
  "Name":"CustomTodo Action",
  "Description":"CustomTodo Action",
  "Locale":"",
  "Version":"5.2"
}
```

3.4 Custom To-Do Action

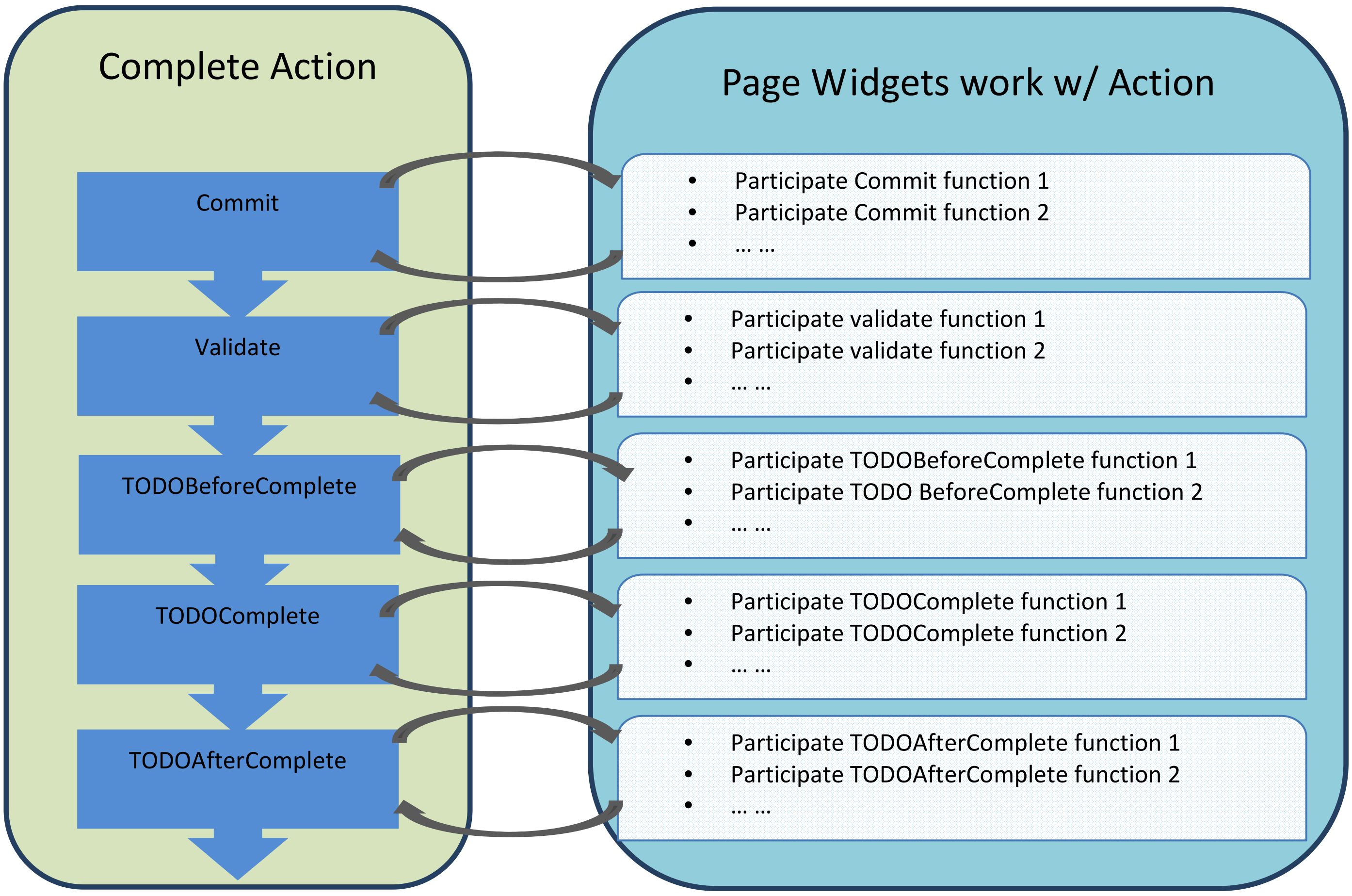
3.4.1 How the actions work with Page Widgets

The actions use the coordination mechanism to work with Page widgets. Taking CompleteTo-doAction as an example, the action will issue the following sequential coordination topics including Commit, Validate, TODOBeforeSave, TODOBeforeComplete, TODOComplete, ToDOAfterSave and TODOAfterComplete. When a coordination topic is issued, the participating function will be executed. You can register a participated function for a coordination topic using coordination.participate(coordination_Topic_Name, participated_function).

An action uses the following steps to execute the coordination steps:

1. Execute the first coordination step
   a) Start the coordination topic
   b) Execute the participated functions
   c) Execute the step callback or errorback function
   d) goto next topic or skip next topic
2. If it has the next coordination step, then execute the next coordination step as the first step, otherwise finish the coordination execution.

The following diagram shows how the CompleteToDoTask works together with the Page Widgets using coordination.


Complete Action
Commit
Validate
TODOBeforeComplete
TODOComplete
TODOAfterComplete
Page Widgets work w/ Action
Participate Commit function 1
Participate Commit function 2
… …
Participate validate function 1
Participate validate function 2
… …
Participate TODOBeforeComplete function 1
Participate TODO BeforeComplete function 2
… …
Participate TODOComplete function 1
Participate TODOComplete function 2
… …
Participate TODOAfterComplete function 1
Participate TODOAfterComplete function 2
… …

- Commit: The topic for a coordination step for page widgets to accept user inputs and processing.
- Validate: The topic for a coordination step for page widgets to do data validation.
- TODOBefore Complete: The topic for a coordination step for page widgets to do preprocessing before completing a to-do task
- TODOComplete: The topic for a coordination step for page widgets to do processing when completing a to-do task
- TODO After Complete: The topic for a coordination step for page widgets to do post-processing after completing a to-do task

3.4.2 Customization points of Action

When the out-of-the-box (OOTB) actions cannot meet your business requirements, you can consider customizing the existing OOTB actions or creating your own action to work with the page widgets. A custom action can implement one or more customization points to fulfill your business requirements.
The following customization points can be considered:

- execute: You can customize the code logic in execute function.
  - Coordination Topics: you can create a custom coordination topic for your custom action to work with

Page Widgets, of course you have to enable your custom page widgets to participate the custom topics. For example, you can define a new coordination topic named "Reject", and then let your custom Page widgets participate in the new topic. As well, you can define the coordination topics sequence for your custom actions.

- Coordination Step Callback function: You can define your custom logic in a coordination step callback function, which will be executed after all the participated functions are executed. For example, you need to update some data into another system when saving a case or To-do task, you can add that logic into a "Save" step's callback function.
- Coordination Step Errorback function: You can define your custom error processing function in case some errors occur when executing the participating functions. For example, you can design a custom warning message dialog when data validation fails.

- isVisible: You can add your custom logic to determine when we should display the action button or menu item.
- isEnable: You can add your custom logic to determine when we should enable the action button or menu item.

3.4.3 Introduction of Coordination step API

Before we introduce the customization code, let us take a look at the coordination step API. The step function is designed to add a coordination step by specifying the coordination topic, callback function, and abortbackfunction.

```
icm.util.Coordination.step(/*string*/ Coordination_Topic,
/*function*/ callback ( /*array*/results,
                                  /*function*/ next,
                                  /*function*/ skip,
                                        coordinationContext){
                                        // callback code...
                                        },
/*function*/ abortback( /*array*/errors,
                                   /*function*/ next,
                                        /*function*/ skip,
                                        coordinationContext){
// errorback code...
})
```

{string} Coordination_Topic - The topic of coordination step. Defined in icm/base/Constants.CoordTopic.

{function} callback - The callback function that is to be executed after the coordination step completes (it means that all the participating functions are executed). The callback (result, next, skip, context) function has the following parameters:

- result - An array that contains the result of each participating callback. For each callback, the array contains two elements: [[true,'result1'],[false,'error']]. The first element is a Boolean value that indicates whether the participating callback completed (true) or was aborted (false). The second element provides either the complete result of the participating callback or the reason the callback was aborted, which can be any object type.
- {function} next - A function that determines that the current step is complete and that the coordination is to proceed to the next step.
- {function} skip - A function used determines that the current step is complete and that the coordination is to skip the next steps. The skip function accept a topic as optional input parameter, it tells the coordination about skipping to which remaining topic. If the parameter is not be given, the coordination will skip all remaining topics.
- context - The coordination context.

{function} abortback- the error back will be executed after coordination aborted. The errorback (error, next, skip, context) function has the following parameters:

- error - an array of two elements array which is the result of each participating callback - [[true,'result1'],[false,'error']]. The first element is a Boolean, indicate if the participator complete(true) or abort(false), the second element indicates the either the complete result or abort reason which could be any object type.
- next - A function that determines that the current step is complete and that the coordination is to proceed to the next step.
- skip - A function used determines that the current step is complete and that the coordination is to skip the next step.
- context - The coordination context.

3.4.4 Build the Custom CompleteToDo Task Action

To implement a custom action, we have the two following steps to do:

- Create an Action java class by extending the PluginAction, and provide the custom action in getActions() method of custom plug-in
- Create the JavaScript class by extending icm.action.Action to implement the custom action.

3.4.4.1 Add CustomCompleteToDoTaskAction class and update plug-in Class

Firstly, we extend com.ibm.ecm.extension.PluginAction to implement CustomCompleteToDoTaskAction as the following code shows. We override the following methods:

- getId():Returns an alphanumeric identifier that is used to describe this action.
- getName():Returns a descriptive label for this action that is displayed on pop-up menus and as hover help for the icon on the toolbar.
- getIcon():Returns the name of the icon that is displayed on the toolbar for this action.
- getPrivilege():Returns a String that contains the list of privilege names that the user must have on the selected documents for this action to be enabled.
- isMultiDoc():Returns a Boolean value that indicates whether this action is enabled when multiple documents are selected.
- getServerTypes ():Returns the server types that this action is valid on.
- getActionModelClass():Provides the name of the icm.action.Action subclass to use when this plug-in action is invoked. Taking CustomCompleteToDoTaskAction as an example, it returns "customTodoAction.action.todo.Complete".
- getActionFunction():Provides the name of the JavaScript function that is invoked for this action. It must return "performAction" as the action function, which is defined in "icm.action.Action".
- getAdditionalConfiguration():Returns additional JSON that will appear on the ecm.model.Action JavaScript object for this action.The following table describes the properties that are supported for an action definition file:

Property	Required or Optional	Type	Description
ICM_ACTION_COMPATIBLE	Required	Boolean	A Boolean value that is set to true if the action can be used in the IBM Case Manager action framework. This framework extends the IBM Content Navigator action framework to provide case-related functions. This property should always be set to true for IBM Case Manager.
type	Optional	String	A string that indicates special processing for the action. The following values are valid for the type property: ● iterator: Specify this value if the action is defined using a method such as getIterator(). The method returns a series of items which are rendered as buttons or menu items. ● checkbox: The action is rendered as a checkbox in the toolbar or pop-up menu. If you sent the type property to this value, you must also set the fieldname property to checkbox.
fieldname	Required	String	If the type property is set to checkbox, you must set this property to the identifier of a property that is defined in the properties array.

			If type property is not set to checkbox, omit the fieldname property.
description	Required	String	A string that provides a brief description of the action.
context	Required	Array	An array that indicates the contexts in which the action can be used. The array elements can take the following formats: ● [["Context 1", "Context 2"]] :The action requires both Context 1 and Context 2 to run. ● ["Context 1", "Context 2"] :The action requires either Context 1 or Context 2 to run. ● [[“Context 1”, “Context 2”],[“Context 1”, “Context 3”], “Context 4”] : The action requires Context 1 and Context 2 or Context 1 and Context 2 or Context 4 to run. ● [] :The action does not require a context to run. E.g. we defined a context as [["ToDoEditable","Coordination"]]
name	Required	String	The name that is displayed in the user interface for the action.
properties	Required	Array	An array that defines the properties that a user can configure an action in a toolbar or menu for a page widget or that is used internally by the action at run time.
events	Required	Array	An array that defines the events that are published by the action. This array can be empty if the event does not publish any events.

CustomCompleteToDoTaskAction.java :

```
packagecom.ibm.icm.extension.sample.actions.todo;
importjava.io.IOException;
importjava.util.Locale;
importcom.ibm.ecm.extension.PluginAction;
importcom.ibm.json.java.JSONObject;
public class CustomCompleteToDoTaskAction extends PluginAction
{
    private final static String CONST_ACTION_ICON = "ToDoComplete.gif";
    private final static String CONST_SVR_TYPE = "p8";
    private final static String CONST_FUNC_NAME = "performAction";
    private final static String CONST_Action_NAME = "Custom Complete ToDo";
    private final static String CONST_ID = "customTodoAction.action.todo.Complete";
    private final static String CONST_ACTION_MODEL_CLASS = "customTodoAction.action.todo.Complete";
    @Override
    public String getId() {
        return CONST_ID;
    }
    @Override
    public String getIcon() {
        return CONST_ACTION_ICON;
    }
    @Override
    public String getPrivilege() {
        return "";
    }
```

```
@Override
public boolean isMultiDoc() {
        return false;
}
public boolean isGlobal() {
        return false;
}
@Override
public String getActionModelClass() {
        return CONST_ACTION_MODEL_CLASS;
}
@Override
publicJSONObjectgetAdditionalConfiguration(Locale locale) {
        String jsonString = "{\r\n" +
                        "             \"ICM_ACTION_COMPATIBLE\": true,\r\n" +
                        "             \"context\": [[\"ToDoEditable\",\"Coordination\"]],\r\n" +
                        "        \"name\": \"Custom Complete ToDo\",\r\n" +
                        "             \"description\": \"An action to complete a ToDo\",\r\n" +
                        "        \"properties\": [\r\n" +
                        "            {\r\n" +
                        "              \"id\": \"label\",\r\n" +
                        "              \"title\": \"Custom Complete\",\r\n" +
                        "              \"defaultValue\": \"Complete\",\r\n" +
                        "              \"type\": \"string\",\r\n" +
                        "              \"isLocalized\":false\r\n" +
                        "            }\r\n" +
                        "         ]\r\n" +
                        "       }";
        try {
                returnJSONObject.parse(jsonString);
        } catch (IOException e) {
                e.printStackTrace();
        }
        return null;
}
@Override
public String getActionFunction() {
        return CONST_FUNC_NAME;
}
@Override
public String getName(Locale arg0) {
        returnCONST_Action_NAME;
}
@Override
public String getServerTypes() {
        return CONST_SVR_TYPE;
}
```

```
}
```

Secondly we need to provide the new custom action in the plug-in by updating the getActions() method of ICN Plug-in class CustomTodoActionPlugin.java, as the following code:

```
    @Override
    public Plugin Action[] getActions() {
        return new PluginAction[] {
                    newCustomCompleteToDoTaskAction()
                    };
    }
```

3.4.4.2 Implement Custom CompleteTodo Action Java Script

We now need to implement the custom Complete action by extending ICM Java Script classicm.action.Action. To address the fore-mentioned requirements, we need customize the coordination step callbacks of both TODO_BEFORECOMPLETE topic and TODO_COMPLETE topic. In TODO_BEFORECOMPLETE step callback function, we will check if the data of the To-do task view is modified or not, and popup a dialog to require the user to input change reason in case of any update.

1. Firstly we implement inputComment and isViewModified functions. We generate a Case comment after a comment is input, and also store the comment into the To-do task propertyCCM_LastComment. In isViewModified function, we check if the data in the To-do task's view is changed or not.

```
inputComment: function(todoTask, results, next, skip){
    var title = null;
    varshowDialog = false;
    var dialog = null;
    varbuttonsCollection = {};
    varOKButtonObj = {};
    OKButtonObj.buttonLabel = "OK";
    OKButtonObj.onExecute = lang.hitch(this, function() {
            //Add Case Comment
            varcurrentTime = dojoLocale.format(new Date(), {datePattern: 'EEE, dd MMM yyyy'});
            varcommentText = "ToDoTask Name: "
            + todoTask.getTaskName()
            + "; Action: " + "Complete"
            + "; Action Time: " + currentTime
            + "; Comment: " + dialog.reason.get("value");
            this.caseEditable.getCase().addCaseComment(ICMConstants.CommentContext.CASE,
                commentText, null);
            console.log("Added Case Comment: " + commentText);
            this.lastComment = dialog.reason.get("value") || "";
            next();
```

```
          });
          buttonsCollection.OK = OKButtonObj;
          text="Please specify reason for change.";
                    dialog = new ToDoDialog({
                                        title: "Complete", // DEV: to be localized
                                        text: text,
                                        buttonsCollection: buttonsCollection,
                                        onCancel: lang.hitch(this, function() {
                                                  this.executing = false;
                                                  this.setEnabled(true);
                                                  skip();
                              })
          });
          dialog.show();
},
```

isViewModified function is just using property view function to determine if the data on current is modified or not.

```
isViewModified:function (){
          if(this.widget.viewNode){
                              if(this.widget._view&& !this.widget._view.get("readOnly")
                              &&this.widget._view.someProperty({
                              callback: lang.hitch(this, function(prop){
                              //only set dirty for the changes on the properties which are in view.
                                        returnprop.controller.isModified(this.widget._widgetLoadTime || new Date());
                              })})){
                                                            return true;
                              }else {
                                        return false;
                              }
                    }else {
                              return false;
          }
},
```

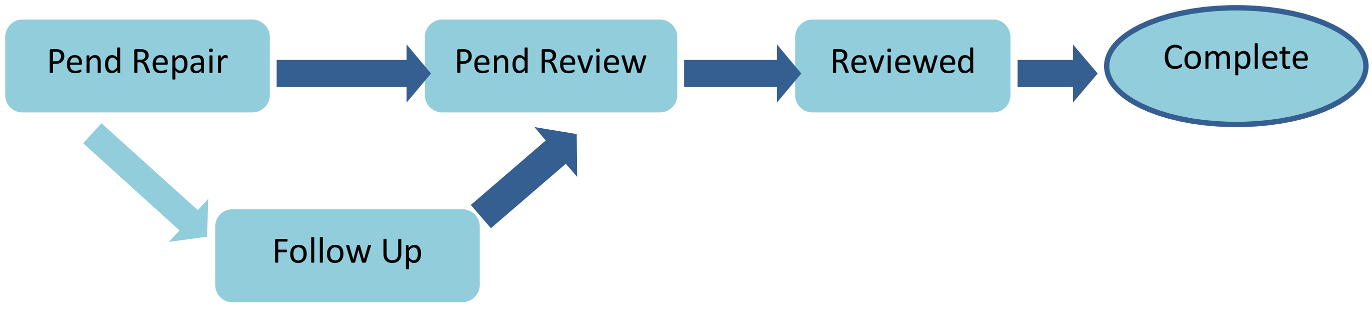
2. Then we customize the TODO_BEFORECOMPLETE step callback function. We will check if its data is changed or not, if it is changed, then popup the comment dialog, otherwise we go to the next coordination step.

```
step(this.icmBaseConst.TODO_BEFORECOMPLETE,
                              lang.hitch(this, function(results, next, skip){
                                        this.logInfo("execute", "in beforeCOMPLETE step callback, results");
                                        this.logInfo("execute", results);
                                                            if(this.isViewModified())   {
                                                                      this.inputComment(todoTask[0], results, next, skip);
                                                            }else   {
                                                                      next();
                                                            }
```

```
        }),
            lang.hitch(this, function(errors, next, skip){
                this.logInfo("execute", "in beforeCOMPLETE step errback, errors");
                this.logInfo("execute", errors);
                        this.showErrDialog("actionExecutedErr", errors);
                        // enable action if failed
                        this.executing = false;
                        this.setEnabled(true);
            skip();
            })
        )
```

3. In TODO_COMPLETE topic step callback function, we manage the ToDoStatus before the data is saved server side. It handles any changes of the following status:

- Pend Repair -> Pend Review
- Follow Up -> Pend Review
- Pend Review -> Reviewed



We use the following code to implement that status changes logic and store the comment to LastComment property.

```
vartodoStatusProperty = todoTask[0].getProperty("F_CaseTask",this.widget.solution.getPrefix()+"_ToDoStatus");
varcurrentStatus = todoStatusProperty.getValue();
// if the current status is Follow Up or Pend Repair
    // then set its status to Pend Review
if(currentStatus == "4" || currentStatus == "1"){
    todoStatusProperty.setValue("3");
}
// if the current status is Pend Review
// then set its status to Reviewed and Completed
if(currentStatus == "3"){
    todoTask[0].setCompleted();
    todoStatusProperty.setValue("0");
}
varlastCommentProperty = todoTask[0].getProperty("F_CaseTask",this.widget.solution.getPrefix()+"_LastComment");
//to set last comment
lastCommentProperty.setValue(this.lastComment);
```

3.4.5 Build Custom Follow Up To-do Task Action

3.4.5.1 Add CustomFollowUpToDoTaskAction class and update plug-in class

Similar to what we have done for custom Complete action, we need to complete the following tasks

1. To add action class CustomCompleteToDOTaskAction
2. To update plug-in class getActions() to include custom Follow Up Action

You can refer to CustomFollowUpToDoTask.java and SampelSolutionPlugin.java for more detail.

3.4.5.2 CustomTodoAction.action.todo.FollowUp

We also need to create a "customTodoAction.action.todo.FollowUp" JavaScript class to implement the custom action.

1. Firstly, we customize the TODO_BEFOREFOLLOWUP step callback function to request user to input a comment as a reason. Please see more details in the following code. Please be aware inputComment() function is similar to the on we created for Custom Complete action.

```
step(this.icmBaseConst.TODO_BEFORECOMPLETE,
                        lang.hitch(this, function(results, next, skip){
                            this.logInfo("execute", "in beforeSave step callback, results");
                            this.logInfo("execute", results);
                            this.inputComment(todoTask, results, next, skip);
                        }),
                        lang.hitch(this, function(errors, next, skip){
                            this.logInfo("execute", "in beforeSave step errback, errors");
                            this.logInfo("execute", errors);
                            // enable action if failed
                            this.executing = false;
                            this.setEnabled(true);
                            skip();
                        })
                    )
```

2. Secondly, we add the code to set the ToDoStatus to "Follow Up" in COMPLETE step callback function. We set the ToDoStatus property to "Follow Up", and store the comment into LastComment property.

```
vartodoStatusProperty = todoTask[0].getProperty("F_CaseTask",this.widget.solution.getPrefix() + "_ToDoStatus");
//to set FollowUp status
todoStatusProperty.setValue("4");
varlastCommentProperty = todoTask[0].getProperty("F_CaseTask",this.widget.solution.getPrefix() + "_LastComment");
//to set last comment
lastCommentProperty.setValue(this.lastComment);
```

4. Summary

We introduced you to the To-Do List widget and Task property features, which came out in the IBM Case Manager 5.2.1 release. We also showed you how to customize To-Do List widget by using the Script Adapter and also a way to build your own action against To-Do tasks. Now, you can develop your own custom actions or page widgets based on the sample project.

5. References

- IBM Case Manager Information Center: http://pic.dhe.ibm.com/infocenter/casemgmt/v5r2m0/index.jsp?topic=%2Fcom.ibm.casemgmttoc.doc%2Fic-homepage.html
- Defining registry files for custom actions, properties, page widgets, and events in IBM Case Manager V5.2:http://www-01.ibm.com/support/docview.wss?uid=swg21648524
- IBM Content Navigator 2.0.2 documentation: http://www-01.ibm.com/support/knowledgecenter/SSEUEX_2.0.2/contentnavigator_2.0.2.htm
- Customizing and Extending IBM Content Navigator: http://www.redbooks.ibm.com/redbooks/pdfs/sg248055.pdf
- Creating custom widgets with the IBM Case Manager JavaScript API: https://www.ibm.com/developerworks/mydeveloperworks/blogs/e8206aad-10e2-4c49-b00c-fee572815374/resource/ACM_LP/ICM52CustomWidgets.pdf