

Multichannel Bank Transformation Toolkit

Extension Development Guide

8.2

Publication information

(September 2016)

Information in this publication is subject to change. Changes will be published in new editions or technical newsletters.

Documentation set

The documentation relating to this product includes:

- Multichannel Bank Transformation Toolkit Functional Developer User Guide

Copyright notice

Multichannel Bank Transformation Toolkit (the Programs and associated materials) is a proprietary product of UNICOM Systems, Inc. – a division of UNICOM Global. The Programs have been provided pursuant to License Agreement containing restrictions on their use. The programs and associated materials contain valuable trade secrets and proprietary information of UNICOM Systems, Inc. and are protected by United States Federal and non-United States copyright laws. The Programs and associated materials may not be reproduced, copied, changed, stored, disclosed to third parties, and distributed in any form or media (including but not limited to copies on magnetic media) without the express prior written permission of UNICOM Systems, Inc., UNICOM Plaza Suite 310, 15535 San Fernando Mission Blvd., Mission Hills, CA 91345 USA.

Multichannel Bank Transformation Toolkit

© Copyright 1998-2016 All Rights Reserved. UNICOM Systems, Inc. – a division of UNICOM Global.

No part of this Program may be reproduced in any form or by electronic means, including the use of information storage and retrieval systems, without the express prior written consent and authorization of UNICOM Systems, Inc.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from UNICOM Systems, Inc.

Disclaimer

We cannot guarantee freedom from, or assume any responsibility or liability for technical inaccuracies or typographical errors. The information herein is furnished for informational use only and should not be construed as a commitment by UNICOM Systems, Inc. – a division of UNICOM Global.

Trademarks

The following are trademarks or registered trademarks of UNICOM Systems, Inc. in the United States and/or other jurisdictions worldwide: Multichannel Bank Transformation Toolkit, UNICOM, UNICOM Systems.

Trademark acknowledgements

Macro 4 and Other Divisions of UNICOM Global:
Macro 4, SoftLanding, UNICOM.

IBM:

IBM[®], AIX[®], CICS[®], CICS/ESA[®], CICS TS[®] CMAC[®], DB2[®], DFSMS/MVS[®], Domino[®], ESCON[®], IMS[™], Internet Explorer[®], iSeries[®], Language Environment[®], LE[®], Lotus[®], MQSeries[®], MVS[™], MVS/ESA[®], OMEGAMON[®], OS/390[®], OS/400[®], Power[®], POWER[®], pSeries[®], RACF[®], RMF[™], S/370[®], S/390[®], SMF[®], System/390[®], System i[®], System p[®], System z[®], VisualAge[®], VM/ESA[®], VSE/ESA[®], VTAM[®], WebSphere[®], z/OS[®], z/VM[®], z/VSE[®], zSeries[®], z Systems[®] and the IBM logo are trademarks or registered trademarks of IBM Corporation in the United States or other countries or both.

Microsoft:

Active Directory, Excel, Microsoft, Notepad, PowerPoint, Visual Basic, Windows, Windows 2000, Windows NT, Windows Server 2003, Windows Server 2007, Windows Vista, Windows XP, WordPad and/or other Microsoft products referenced are either trademarks or registered trademarks of Microsoft Corporation.

Adobe Systems Incorporated:

Adobe[®] and Acrobat[®] are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Apache Software Foundation:

Apache, Apache Tomcat and Tomcat are trademarks of the Apache Software Foundation.

Apple Inc.:

AirPrint, iPad and Safari are trademarks or registered trademarks of Apple Inc. registered in the United States and other countries.

BEA Systems, Inc.:

JRockit and WebLogic are registered trademarks of BEA Systems, Inc.

BMC Software Inc.:

Boole & Babbage, Data Packer, Optimizer and Super Optimizer are trademarks or registered trademarks of BMC Software, Inc., or its affiliates or subsidiaries (collectively, "BMC Software").

BSD:

PostgreSQL is distributed under the classic BSD license. (Portions Copyright © 1996-2006, PostgreSQL Global Development Group; Portions Copyright © 1994-1996 Regents of the University of California.)

CA, Inc.:

CA ACF2, CA Datacom, CA Endevor, CA IDMS, CA InterTest, CA NetMaster, CA Optimizer, CA Panexec, CA Panvalet, CA Ramis, CA Telon and CA Top Secret are registered trademarks of CA, Inc.

Candescent SoftBase LLC:

SoftBase[®] is a registered trademark of Candescent SoftBase LLC.

Canonical Ltd:

Ubuntu is a registered trademark of Canonical Ltd.

Chicago-Soft, Ltd.:

QuickRef is a trademark of Chicago-Soft, Ltd.

Cincom Systems, Inc.:

MANTIS is a registered trademark of Cincom Systems, Inc.

Computer Sciences Corporation:

Hogan and Hogan Umbrella are trademarks or registered trademarks of Computer Sciences Corporation.

Compuware Corporation:

Abend-AID and Compuware are trademarks or registered trademarks of Compuware Corporation.

Dell Inc.:

Dell and the Dell logo are trademarks of Dell Inc.

Emtex Limited:

Emtex and VIP are trademarks of Emtex Limited.

Jean-loup Gailly and Mark Adler:

zlib is a registered trademark or trademark of Jean-loup Gailly and Mark Adler.

GNU General Public License:

Cygwin is free software released under the GNU General Public License.

Google Inc.:

Google and Google Chrome are registered trademarks of Google Inc.

Hewlett-Packard Development Company, L.P.:

HP and HP-UX are registered trademarks of Hewlett-Packard Development Company, L.P., and/or its subsidiaries.

Innovation Data Processing:

IAM is a registered trademark of Innovation Data Processing Corporation.

Kofax, Inc.:

Kofax, the Kofax logo and Kofax Capture are the trademarks or registered trademarks of Kofax, Inc., in the United States and other countries.

Linus Torvalds:

Linux is a registered trademark of Linus Torvalds.

Massachusetts Institute of Technology (MIT):

Kerberos is a trademark of the Massachusetts Institute of Technology (MIT).

Merrill Pty Ltd.:

MXG is a registered trademark of Merrill Pty Ltd.

Mozilla Foundation:

Firefox is a registered trademark of the Mozilla Foundation.

Mozilla Public License:

Expat is free software released under the Mozilla Public License.

Novell, Inc.:

openSUSE is a registered trademark of Novell, Inc.

The Open Group:
UNIX is a registered trademark of The Open Group.

Oracle Corporation:
EJB, Java, JDBC, JDK, JMX, JRE, JSP, JVM, Solaris and SunOS are trademarks or registered trademarks of Oracle Corporation and/or its affiliates. Oracle is a registered trademark, and other Oracle product names, service names, slogans or logos are trademarks or registered trademarks of Oracle Corporation.

Red Hat, Inc.:
Red Hat, Red Hat Enterprise Linux, the Shadowman logo and JBoss are registered trademarks of Red Hat, Inc. in the United States and other countries.

SAP AG:
SAP, the SAP logo, the SAP Partner logo, SAP R/3, SAP ArchiveLink, SAP NetWeaver, SAPPHIRE and Duet are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

SAS Institute Inc.:
SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.

Simon Tatham:
PuTTY is copyright Simon Tatham.

Software AG:
Adabas and Natural are registered trademarks of Software AG. Software AG and all Software AG products are either trademarks or registered trademarks of Software AG and/or Software AG USA, Inc.

SPARC International, Inc.:
SPARC is a registered trademark of SPARC International, Inc. (Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.)

Standardware Inc.:
COPE is a trademark of Standardware Inc.

Sun Microsystems, Inc.:
Sun, Sun Microsystems, the Sun logo, MySQL and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the United States and other countries.

SUSE LLC:
SUSE is a registered trademark of SUSE LLC in the United States and other countries.

Syncsort Inc.:
Syncsort is a registered trademark of Syncsort Inc.

Wireshark Foundation:
Wireshark and the "fin" logo are registered trademarks of the Wireshark Foundation.

XEROX CORPORATION:
XEROX, The Document Company and the stylized X are trademarks of XEROX CORPORATION.

X.Org Foundation:
X Window System is a trademark of the X.Org Foundation.

Additional trademarks and registered trademarks are the property of their respective owners.



Contents

About this manual	9
Release levels.	10
Conventions.	10
Chapter 1 BTT Overview	11
Chapter 2 Environment Preparation	13
Plug-in Project Setup	14
Runtime Project Setup.	16
Chapter 3 Dojo Widget Extension	17
Enable customized widget in XUI Editor.	19
Define a widget in xml file	19
Display widget in XUI Editor.	22
Import widget	23
Create widget mapping.	25
Enable customized widget in runtime.	28
Implement JSP tag handler.	28
Register JSP tag handler	29
Dijit implementation.	29
JSP template	29
Enable customized widget in preview mode	30
Register JSP taglib.	30
Modify JSP template.	31
Advanced topics	32
Customized Property Editor.	32
New Property Tab.	33
Customized Property Mapping Rule	33
ECA support	34
NLS support	35
BTT Context data binding	36

	XUI Generation Template	36
	XUI page generation from BTT context data	37
	Change default behavior of XUI generation	39
	Extend Table Column Widget	41
	How to add version control on runtime NLS files	45
Chapter 4	Data Type Extension	47
	Implement data type extension	48
	Declare new data type	48
	Implement type validator	49
	Implement type converter	50
	Implement type presentation widget	52
	Data type extension sample	53
Chapter 5	Web Services Extension	61
	Web services Tool Extension	62
	ID Mapping during self-defined operation generation	62
	Web services Runtime Extension	66
	Web services Runtime Overview	66
	Extend WS Handler and WS Connector	66
Chapter 6	Channel Policy Management and Extension	69
	Channel level policy management	71
	Implement channel policy handler	71
	Define rule provider service	72
	Configure policy for channels	73
	Exception handling	73
	Operation level policy management	75
	Implement OpStep for operation level policy	75
	Configure operation	76
	Channel policy sample	77
	How to run the sample	77
Chapter 7	Process Editor Extension	79
	Extend processor editor object	81
	Create configuration file for palette object	81
	Processor editor extension sample	93
	ClientPromptState sample	93
	AlphHtmlProcessor sample	94
Chapter 8	Global Function Extension	97
	Extend global functions	100
	Implement global functions	100
	Describe global functions in xml	101
	Register for tooling	102
	Register for runtime	104
	Global Function Extension Sample	105

Chapter 9	Generated JS File Name Extension	107
	Extend generated JS file naming rule	108
	Implement naming rule	108
	Register implementation	108
Chapter 10	Naming Conventions Extension	111
	Extend naming conventions	112
	Extend rule by registering new naming convention rule	112
	Extend by registering new naming manager class	117
Chapter 11	Multi-project Support in Extension	121
	Handle project prefix	123
Chapter 12	Pagination Extension	125
	Extend technical pagination operation	127
	Pagination parameters	129
	Register customized technical pagination operation	131
Chapter 13	Client State Extension	133
	Step 1: Extend a Client State	134
	Implement state class	134
	Register the implementation class into btt.xml	135
	Step 2: Enable the extended State in Transaction Editor	136
	Create configuration file for the extended client state	136
	Register extended client state into the palette	136
	Create configuration file for mapping rules	137
	Register mapping rules	138
	Step 3: Extend navigation engine to register command handler	140
	Extend the navigation engine to register a command handler	140
	Step 4: Add the reference of new navigation engine to template	142
Chapter 14	Reference Sample Topics	143
	How to extend a global function invoked in ECA action	144
	Define global function in XML	144
	Register global function definition as Eclipse extension	144
	Implement JavaScript for global function	145
	Enable XUI editor aware of this global function	146
	How to extend a global function to manipulate collection data	147
	Define global function in XML	147
	Register global function definition	147
	Implement the function logic to calculate the sum of account balance	148
	Register the implementation class of global function	149
	Usage Scenario of the global function in mapping editor	149



About this manual

As a multi-channel application development toolkit, the Multichannel Bank Transformation Toolkit (BTT) implements a set of common and reusable components for channel application development.

Furthermore, BTT provides tools for developers to implement channel applications more efficiently and easily. At the same time, for a channel application, there are some project specific reusable components and facilities that need to be implemented by application developers. BTT provides this capability for application developers to implement project level reusable components and integrate them with BTT framework.

Release levels

Macro 4 product release levels are of the form *n.nnn*. Minor software updates are reflected by a change in the last two digits, and do not necessarily cause the documentation to be reissued.

Conventions

The following typographic conventions are used:

boldface	Indicates a command or keyword that you should type, exactly as shown.
<i>italics</i>	Indicates a variable for which you should substitute an appropriate value.
monotype	Indicates literal input and output.
Ctrl+D	Indicates two or more keys pressed simultaneously.
[]	Brackets surround an optional value.
	Vertical bars separate alternative values from which you must make a selection.
...	Ellipsis indicates that the preceding element may be repeated.

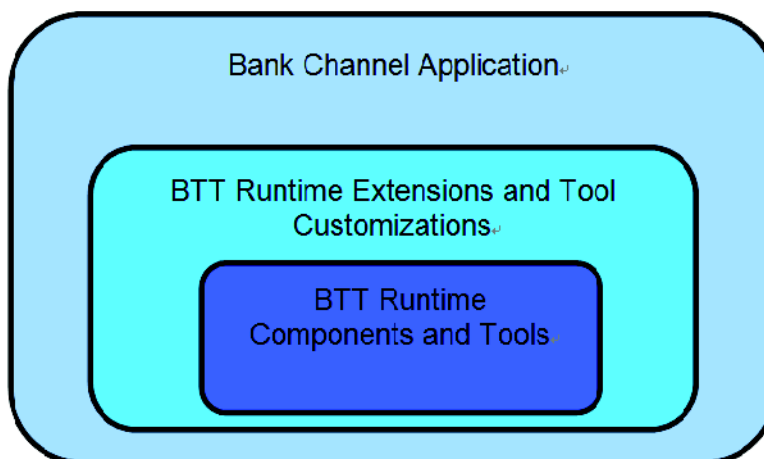


CHAPTER 1

BTT Overview

As a multi-channel application development toolkit, BTT implements a set of common and reusable components for channel application development. Furthermore, BTT provides tools for developers to implement channel applications more efficiently and easily. At the same time, for a channel application, there are some project specific reusable components and facilities that need to be implemented by application developers. BTT provides this capability for application developers to implement project level reusable components and integrate them with BTT framework.

The figure below shows the relationship of BTT framework, BTT extensions and the bank channel application.



BTT classifies BTT application developers into two types according to their roles:

- **Infrastructure developer:** Infrastructure developers have deep knowledge on BTT and related technologies such as OOP and Java EE. As Infrastructure developers, they are responsible for designing and implementing the project specific components and tool functions.

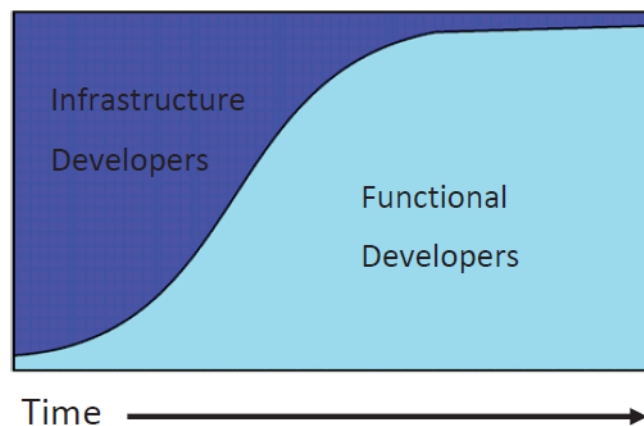
- **Functional developer:** Functional developers have a little knowledge on BTT and related technologies. As Functional developers, they implement specific transactions that include user interfaces, operation logics and transaction flows. Development productivity is one of the primary considerations for Functional developers.

The development phase should make use of the reusable components and largely improve the productivity of channel application development.

A typical BTT application project has two development phases.

- **Infrastructure development phase:** where the Infrastructure developers design and implement project specific reusable components as BTT extensions and customize BTT tools for these extensions if necessary.
- **Incremental development phase:** where the Functional developers use the tools provided by BTT and the infrastructure phase extensions to develop all the transactions.

The figure below shows the skill distribution in the infrastructure development and incremental development phases.



In the infrastructure development phase, the Infrastructure developers should consider these BTT extensions for a specific project:

- (Table Column) Widget extension
- JSP, UI and transaction template customization
- Generated JavaScript file name extension
- Processor editor extension
- Naming convention extension
- Basic data type extension
- Channel policy extension
- Web services connector extension
- Global functions extension
- Client state extension.

This document gives details on each of these BTT extensions and how to implement them.



CHAPTER 2

Environment Preparation

A typical BTT extension development extends the BTT runtime functions to meet specific project requirements. At the same time, it will also customize the BTT tool facilities to use the runtime extension effectively. When developing BTT extensions, Infrastructure developers usually create two projects:

- **BTT XUI Web Project** that includes the extensions for runtime.
- **Plug-in Project** that includes the extensions for tool customization.

Plug-in Project Setup

This is the procedure to prepare the environment for an Eclipse plug-in project to customize the BTT tool.

- 1** Create a standard Eclipse plug-in project.
 - a** Go to the [Eclipse Resources](#) page for information on how to create an Eclipse Plug-in Project.

- 2** Add Plug-in dependencies.

- a** Open the plugin.xml file for the plug-in project.

- b** Click the dependencies tab.

- c** Add these plug-ins:

org.eclipse.ui

org.eclipse.core.runtime

org.eclipse.core.resources

org.eclipse.draw2d

org.eclipse.ui.forms

org.eclipse.ui.ide

org.eclipse.ui.views.properties.tabbed

com.ibm.btt.core

com.ibm.btt.tools.xui.editor2

com.ibm.btt.tools.common

com.ibm.btt.tools.transaction

com.ibm.btt.tools.transaction.diagram

com.ibm.btt.tools.transaction.dominant

com.ibm.btt.tools.transaction.edit

com.ibm.btt.tools.transaction.editor

Required Plug-ins

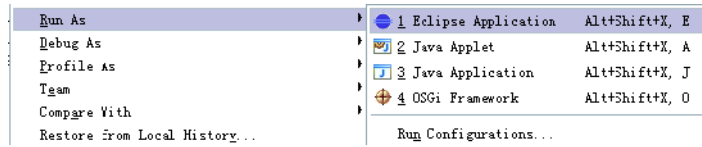
Specify the list of plug-ins required for the operation of this plug-in.

- ▶ org.eclipse.ui
- ▶ org.eclipse.core.runtime
- ▶ org.eclipse.core.resources (3.6.0)
- ▶ org.eclipse.draw2d (3.6.0)
- ▶ org.eclipse.ui.forms (3.5.0)
- ▶ org.eclipse.ui.ide (3.6.0)
- ▶ org.eclipse.ui.views.properties.tabbed (3.5.100)
- ▶ com.ibm.btt.core (7.1.0)
- ▶ com.ibm.btt.tools.common (7.1.0)
- ▶ com.ibm.btt.tools.perspective (1.0.0)
- ▶ com.ibm.btt.tools.transaction (7.1.0)
- ▶ com.ibm.btt.tools.transaction.diagram (7.1.0)
- ▶ com.ibm.btt.tools.transaction.dominant (7.1.0)
- ▶ com.ibm.btt.tools.transaction.edit (7.1.0)
- ▶ com.ibm.btt.tools.transaction.editor (7.1.0)
- ▶ com.ibm.btt.tools.xui.editor2 (7.1.0)

Runtime Project Setup

This is the procedure to establish a BTT XUI Web Project for implementing BTT runtime extensions.

- 1 Right click the plug-in project you created in ‘Plug-in Project Setup’ on page 14.
- 2 Click **Run As > Eclipse Application**.



- 3 In the Eclipse application instance, create a new BTT XUI Web Project.



CHAPTER 3

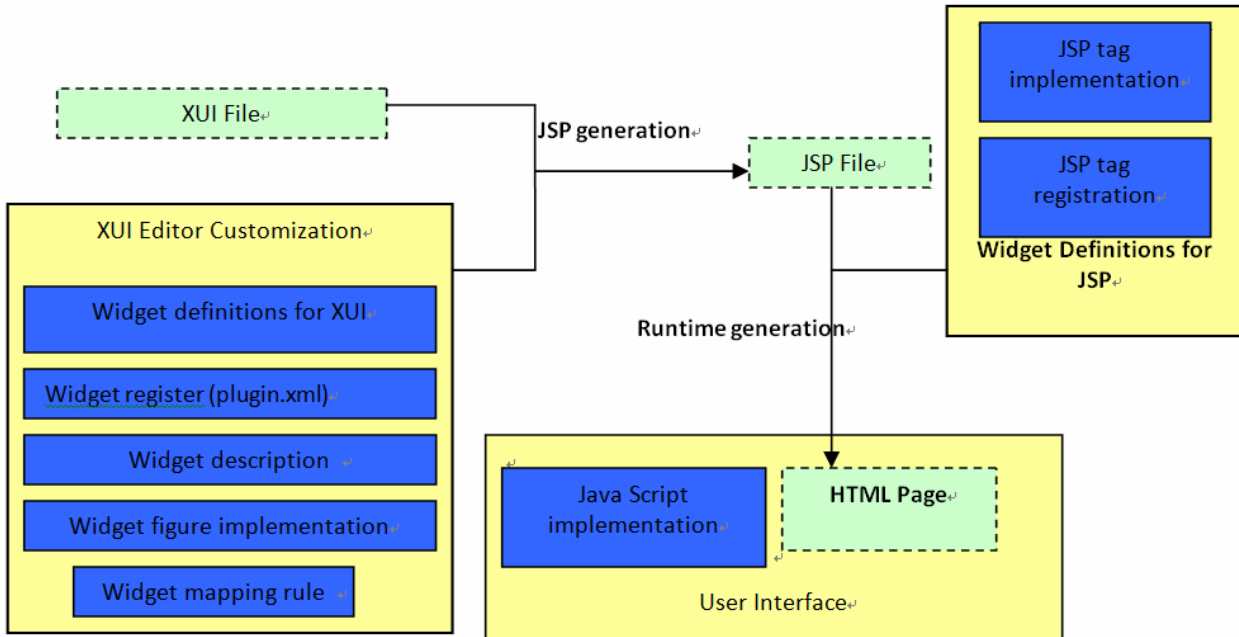
Dojo Widget Extension

BTT dojo widgets wrap the dijit widgets with BTT context binding support. BTT dojo widgets have the similar functions as the dijit widgets. Meanwhile, they can be bundled with BTT context and data automatically in BTT framework. Functional developers could use the BTT XUI editor to compose a page just by dragging and dropping BTT dojo widgets without any coding work.

The BTT framework provides the capability for Infrastructure developers to implement project- specific widgets and import them into the palette of XUI editor. Functional developers can then use these widgets in XUI editor in the same way as the original BTT widgets. Three parts are required to develop a project-specific BTT dojo widget:

- Make widgets available in XUI editor, including:
 - Define a widget with xml file
 - Register the widget as an extension of plug-in project
 - Implement Java classes to show the widget in XUI editor
 - Define mapping rules for XUI generation with xml file (Optional)
 - Register the mapping rules as an extension of plug-in project (Optional).
- Generate HTML and JavaScript code from JSP tag, including:
 - Register tags in JSP tag library file
 - Implement JSP tag handlers.
- Present a widget in browser, including
 - Implement related JavaScript for the widget.

The figure shows the relationship of the extensions.



Note Blocks with dashed lines are created by Functional developers or generated by BTT tools automatically.

Blocks with solid lines are implemented by Infrastructure developers as BTT extensions.

Enable customized widget in XUI Editor

The tasks below enable a customized widget in XUI editor:

- Define a widget in xml file
- Implement a widget figure class to show the widget in XUI editor
- Register a widget as an extension of the BTT plug-in
- Register a mapping rule for generating widgets to JSP tags.

Define a widget in xml file

BTT defines a widget in an xml file. The xml file describes:

- how the widget is shown in XUI editor
- the properties of the widget
- how to edit the properties of the widget.

The sections that follow describe the tags in the widget definition XML file.

Figure tag

It defines how to display widgets in the XUI editor by specifying the displaying the class. There are two types of figure classes for implementation, draw2d and SWT. These are described in ‘Display widget in XUI Editor’ on page 22. The table below gives the attributes for a figure tag.

Attribute	Description
type	The type of implementation the class used to show the widget in XUI editor. The available values are draw2d and SWT.
class	The implementation class used to show the widget in XUI editor.
style	Only available for the SWT type figure to provide style information.

Property tag

The table below gives the attributes for the property definition element.

Attribute	Description
name	The identifier of the property.
default	The default value of this property.
type	The type of predefined property editor for this property. The value could not be arbitrary, which should be selected from the registered type list described in ‘Figure tag’ on page 19. Default value: String
showInEditor	Determines whether to display this property in the widget property view. Default value: true

Attribute	Description
showInExpression	Determines whether to display this property in the ECA action list. Default value: false
showInAction	Determines whether to display this property in the ECA action list. Default value: false
Description	The description of this property, which is NLS aware.
Level	Reserved in current release.

Widget tag

The table below gives the predefined types for the type attribute of the widget definition.

Type	Description
String	Determines the visibility of the widget. Its available values are: visible, hidden and gone.
Visibility	Determines the visibility of the widget. Its available values are: visible, hidden and gone.
Boolean	Basic Boolean type. Values are: true and false.
ButtonType	Determines the type of a button widget. Values are: button, submit, reset, submit with no data and submit without validation.
KeyBinding	Determines the shortcut key of the widget.
DataName	The name of field, data or KeyedCollection in BTT context.
DataNameList	The name of IndexedCollection in BTT context. This type is used for widgets like Combo, SelectList and Table.
DataNameTreeContent	The name of the Tree widget data in the BTT context.
Image	The image of the widget
NLS	String with NLS
ErrorLevel	Determines the level of error info. Values are: ERROR, INFO and WARN
Integer	The integer type of the property
OperationName	The name of operation.

Event tag

The table below gives the attributes for an event definition tag.

Attribute	Description
Name	The identifier of the event
Description	The description of the event which is NLS.

Function tag

The table below gives the attributes for a function definition tag.

Attribute	Description
name	The identifier of the function
description	The description of the function which is NLS
showInAction	Determines whether to display this function in the ECA action list
showInExpression	Determines whether to display this function in the ECA expression panel
returnType	The return type of the function. Values are: String, Number and Boolean
parameter	The parameter of the function

Parameter tag

There is usually more than one parameter tag for each function element. The table below gives the attributes for each parameter tag.

Attribute	Description
name	The name of the parameter
description	The description of the parameter
type	The type of the parameter

Below is a sample of a widget definition:

```
<widget xmlns="http://btt.ibm.com/WidgetSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://btt.ibm.com/WidgetSchema
WidgetSchema.xsd ">
<figure type="draw2d" class="org.eclipse.swt.widgets.Button"
style="TOGGLE"/>
<properties>
  <!-- common properties -->
```

```

    <property name="id" type="String" />
    <property name="width" default="50" />
    <property name="height" default="28" />
    <property name="visibility" default="visible"
    type="Visibility" showInAction="true" showInExpression="true"
    description="%desc_prop_visibility" />

    <property name="icon" type="Image" showInAction="true"
    showInEditor="true" showInExpression="true"
    description="%desc_prop_icon" />

    <property name="ontext" type="String" />
    <property name="offtext" type="String" />
</properties>

<events>

    <event name="onClick" description="%desc_event_onclick" />
    <event name="onKeyDown" description="%desc_event_onkeydown" />
    <event name="onKeyPress" description="%desc_event_onkeypress"
    />
    <event name="onKeyUp" description="%desc_event_onkeyup" />
    <event name="onMouseDown"
    description="%desc_event_onmousedown" />

    <event name="onMouseUp" description="%desc_event_onmouseup" />
    <event name="onMouseEnter"
    description="%desc_event_onmouseenter" />
    <event name="onMouseLeave"
    description="%desc_event_onmouseleave" />
    <event name="onMouseMove"
    description="%desc_event_onmousemove" />

    <event name="onChange" description="%desc_event_onchange" />
</events>

<functions>

    <function name="isFocusable" showInAction="false"
    showInExpression="true" returnType="Boolean"
    description="%desc_func_isfocusable" />

    <function name="focus" showInAction="true"
    showInExpression="false" description="%desc_func_focus" />
</functions>
</widget>

```

Display widget in XUI Editor

To display the customized widget in XUI editor, Infrastructure developers need to implement a figure class to present this widget. There are two types of figure class could be used: draw2D and SWT.

draw2D shows a widget in the XUI editor with an image. This type of figure class can be easily implemented, but its look does not change when you edit the widget properties in the XUI editor. SWT type figure is a standard Eclipse SWT widget. It supports dynamic change when you edit the widget properties in XUI editor, such as changing text of widget.

To implement draw2D type figure, Infrastructure developers can either directly extend from the class `org.eclipse.draw2d.Shape`, or extend from `com.ibm.btt.tools.xui.editor2.figure.LabelShape`, which provides more facilities to implement. If the figure class extends from `com.ibm.btt.tools.xui.editor2.figure.LabelShape`, load an image as a label icon and refresh in its constructor method. Sample code as below:

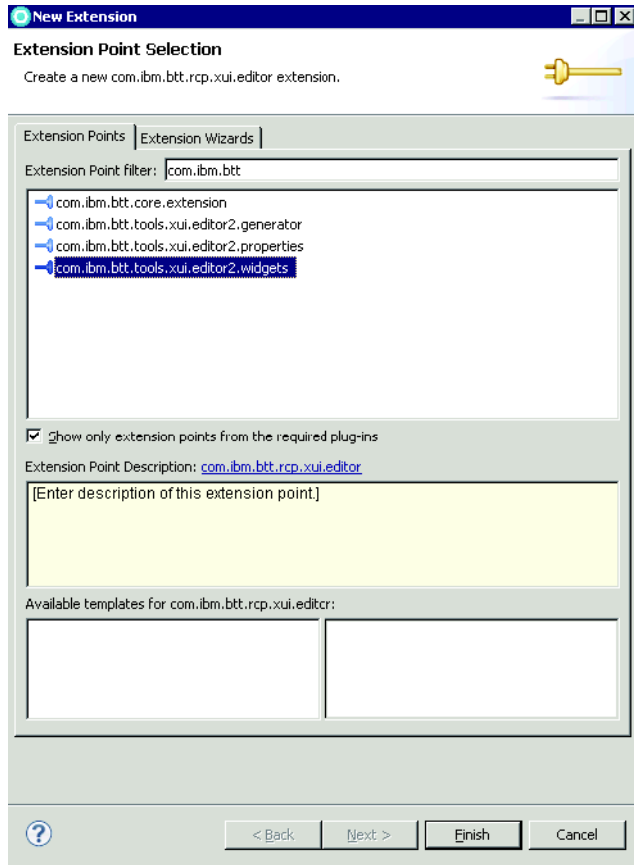
```
getLabel().setIcon(Activator.  
    getImageDescriptor("images/AccountWidget.PNG").createImage());  
refresh();
```

For the SWT type figure, Infrastructure developers should follow the [Eclipse SWT](#) specification to implement any necessary SWT widget.

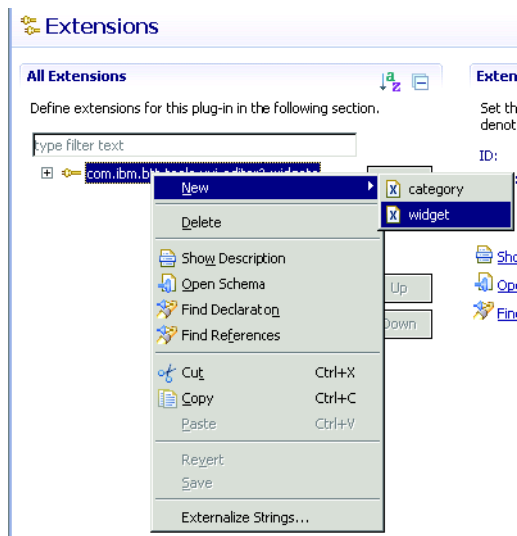
Import widget

To enable the customized widget available in XUI editor, Infrastructure developers need to add an extension for this widget in the `plugin.xml` file of the extension plugin project. To import a widget:

- 1 Open the `plugin.xml` file.
- 2 Click the **Extensions** tab.
- 3 Click **Add**.
- 4 In the **Extension Point Filter** field, input **com.ibm.btt**,
- 5 Click **com.ibm.btt.tools.xui.editor2.widgets**.
- 6 Click **Finish**.



- 7 Right click **com.ibm.btt.tools.xui.editor2.widgets** then click **New > widget**.
If you click **New > category**, a new widget category will be created to group widgets in the palette of XUI editor.



- 8 In the **Extension Element Details** dialog box, type the applicable information.
 - **name:** The value of this field serves as widget id, so it should be unique.

- **label:** The value of this field is the display name of the widget which will be shown in palette. It supports NLS.
- **icon:** This field allows users to select an image to display the widget in the palette of XUI editor. Image in 16x16 pixels is recommended as it is consistent with existing BTT widgets.
- **category:** This field allows users to input the category which the widget belongs to.
- **config:** This field allows users to select the widget definition file described in 'Define a widget in xml file' on page 19.
- **container:** This field allows users to set if the widget has the ability to contain other widgets (true) or not (false).
- **description:** This field requires the user to input a short description for a widget.

Extension Element Details

Set the properties of "widget". Required fields are denoted by "*".

name*:	<input type="text" value="ToggleButton"/>
label*:	<input type="text" value="ToggleButton"/>
icon*:	<input type="text" value="icons/togglebutton.PNG"/> <input type="button" value="Browse..."/>
category*:	<input type="text" value="AlphaSampleWidgets"/>
config*:	<input type="text" value="widgets/ToggleButton.xml"/> <input type="button" value="Browse..."/>
container*:	<input type="text" value="false"/> <input type="button" value="v"/>
description:	<input type="text"/>

Create widget mapping

After Functional developers complete composing the XUI file and when they select Generate Dojo Page function, BTT tools automatically generate the JSP file for this XUI file. In order to generate proper JSP tags for the customized widget, Infrastructure developers need to create a new widget mapping file and register it as BTT plug-in extension.

Create widget mapping file

- In a widget mapping file, there should be one mappings tag for which there is only one attribute:
 - **prefix:** the prefix text of tags when mapping widget to JSP tags. The value should be the same as the prefix attribute of tablib directive in JSP file.
- Each widget requires a widget-mapping element to describe how BTT maps this widget to a JSP tag. The widget-mapping tag has two attributes:
 - **widgetName:** the name for the given widget. It should match with the name defined in the widget extension described in 'Import widget' on page 23.
 - **tagName:** the JSP tag name for this widget.

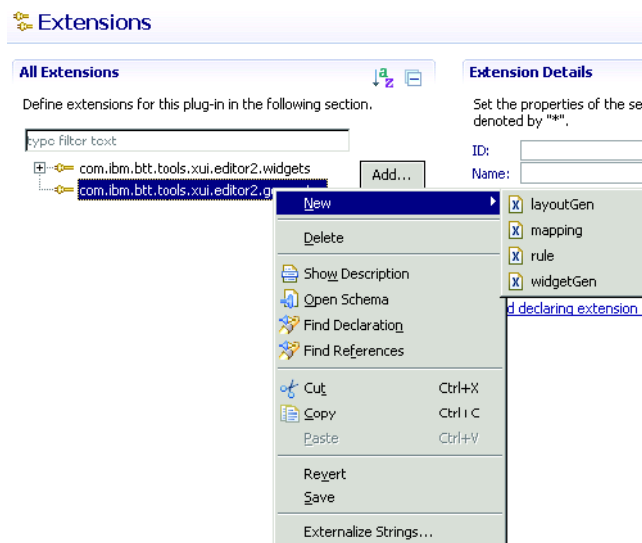
- By default, each widget property name will be mapped into the same attribute name of the JSP tag directly. To customize property mappings, Infrastructure developers can define a property-mapping element to specialize the mapping from widget property to a tag attribute. There are 3 attributes for the property-mapping tag:
 - **propName:** the name of the property which should be the same as the name defined in widget definition file described in ‘Implement JSP tag handler’ on page 28.
 - **attrName:** the name of the attribute which maps a property to a JSP tag.
 - **rule:** the property mapping rule which can handle more flexible property mapping scenarios.

To find out how to implement a property mapping rule, see ‘Customized Property Mapping Rule’ on page 33. Below is a snippet from a widget mapping file:

```
< mappings prefix="bttdojo:">
  < widget-mapping widgetName="AccountWidget" tagName="account">
    < property-mapping propName="labelColor" attrName="color" />
    < property-mapping propName="balanceColor"
      rule="RGBValueRule" />
  < /widget-mapping>
< /mappings>
```

Register widget mapping

- 1 In the **Extensions** tab of plugin.xml file.
- 2 Click **Add**.
- 3 Click **com.ibm.btt.tools.xui.editor2.generator**.
- 4 Click **Finish**.
- 5 Right click **com.ibm.btt.tools.xui.editor2.generator** then click **New > mapping**.



- 6 In **Extension Element Details** dialog box, click the mapping file defined previously into the **file** field.

Extend Default Widget Generator

A widget generator is used to map the XUI widget into a JSP tag according to the widget mapping file described previously. Infrastructure developers should implement and register a new widget generator by extending the default one to meet the project-specific scenario, such as supporting new mapping properties for extended widget.

To implement a new widget generator, Infrastructure developers should extend the class `com.ibm.btt.tools.xui.editor2.generator.WidgetGenerator` and override the method:

```
public void generate(StringBuffer buffer)
```

The method is invoked to generate JSP tag for a widget and the generated text need to be appended into the buffer object.

To register a new widget generator, Infrastructure developers should follow the procedure below.

- 1 Open the **Extensions** tab of `plugin.xml` file.
- 2 Click **Add**.
- 3 Click **com.ibm.btt.tools.xui.editor2.generator**.
- 4 Click **Finish**.
- 5 Right click **com.ibm.btt.tools.xui.editor2.generator** then click **New > widgetGen**.
- 6 In the **Extension Element Details** dialog, type the applicable information.
 - **class**: choose the new widget generator class implemented previously.
 - **target**: select the class **com.ibm.btt.tools.xui.editor2.model.impl.WidgetModel**
 - **priority**: select medium or high to override default widget generator.
 - **name**: type the name of this generator.

Extension Element Details

Set the properties of "widgetGen". Required fields are denoted by "*".

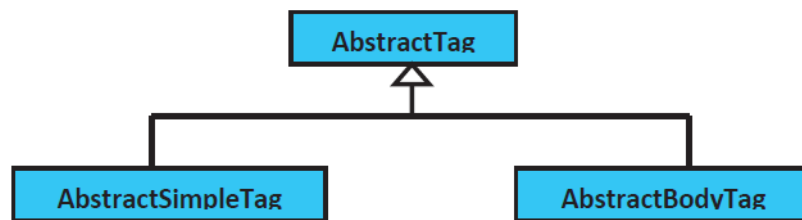
class* :	<input type="text" value="com.ibm.btt.alphasample.generator.WidgetG"/>	<input type="button" value="Browse..."/>
target* :	<input type="text" value="btt.tools.xui.editor2.model.impl.WidgetModel"/>	<input type="button" value="Browse..."/>
priority* :	<input type="text" value="medium"/>	
name :	<input type="text" value="Extended Widget Extension"/>	

Enable customized widget in runtime

Implement JSP tag handler

After the new JSP tag for a widget has been generated into a JSP file, it still requires a tag handler to generate the dynamic HTML content for this new JSP tag at runtime. This section describes how to implement a JSP tag handler and how to use it in the BTT framework.

Infrastructure developers do not need to implement a JSP tag handler from scratch. A new tag handler could be extended from BTT facility classes. BTT provides two abstract tag handlers for Infrastructure developers to extend.



The `com.ibm.btt.dojo.tag.AbstractSimpleTag` class is provided for the handler that handles the tag and does not contain sub-tags or inner content, such as a button or label tag. Two methods must be overridden when extending from `AbstractSimpleTag`:

protected void initAttributes()

The tag handler needs to put the corresponding DOJO widget type into the attributes in this method.

```

protected void initAttributes(){
    super.initAttributes();
    attributes.put("dojoType", "com.ibm.btt.dijit.Account");
}
  
```

protected String getTagName()

The tag handler needs to return the corresponding HTML tag name of the DOJO widget in this method. If it is not a DOJO widget tag, then return `null`. Additionally there are two hook methods for Infrastructure developers to implement more flexible tag handlers.

protected void beforeGenerateTag(StringBuffer buffer)

The method is used for subclass to inject other JavaScript code or generate hidden HTML fields before BTT generate dojo code.

protected void afterGenerateTag(StringBuffer buffer)

The method is used for the subclass to inject other JavaScript code or generate hidden HTML fields after BTT generates dojo code.

The `com.ibm.btt.dojo.tag.AbstractBodyTag` class is provided for a handler which handles tag contains sub-tags or inner content. For example, table tag may have nested column tags to describe each column in table.

The `AbstractBodyTag` handles the logic of generating content for sub-tags Like extending from `AbstractSimpleTag`, both `initAttributes()` and `getTagName()` methods must be overridden.

Besides `beforeGenerateTag` and `afterGenerateTag`, a new method is provided for Infrastructure developers to extend.

protected void afterGenerateStartTag(StringBuffer buffer, Map<String, String> attributes)

As indicated by name, this method is used for subclass to inject the code after generating the start tag.

Register JSP tag handler

To make sure the implemented JSP tag handler can be used by BTT at runtime. Infrastructure developers need to create a new JSP lib file to support tags created for new widgets. The file follows standard JSP tag library schema. You can access http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd for more information.

Dijit implementation

Widget JavaScript implementation presents a widget in browser. Infrastructure developers can implement JavaScript by extending some dijit widget or BTT Dojo dijit. The widget JavaScript is recommended to extend from `com.ibm.btt.dijit.AbstractWidgetMixin` provided by BTT which provides NLS and visibility functions support. Some implementation samples are provided by BTT product.

JSP template

After the JavaScript class for the widget has been implemented, the class needs to be added into JSP require declaration section by XUI generation. So Infrastructure developers need to create a new JSP template file or edit the existing BTT JSP template file to add the required declaration for the new JavaScript class. By default, JSP template files are in the `WebContent/templates` folder of XUI web project.

Enable customized widget in preview mode

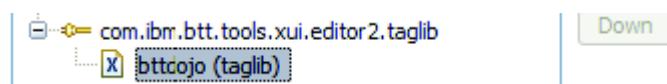
The BTT XUI editor provides the preview function that lets Functional developers preview the XUI file in a browser before the file is generated into a JSP file and deployed on application server. This function benefits Functional developers as they can see what the XUI file is like in a browser without effort of deploying.

When the XUI file is previewed, the BTT tools will dummy a JSP environment to invoke JSP tag handler so that it converts a JSP tag into an HTML tag. In order to make sure the BTT tools invoke the right JSP tag handler, Infrastructure developers need to register the JSP taglib information in their widget extension project. Meanwhile, as the preview environment is not a real JSP runtime environment, JSP code is not executed. Infrastructure developers need to ensure the template does not contain any JSP code.

Register JSP taglib

Register JSP tag handler in widget extension project

To enable the customized widget in preview mode, Infrastructure developers need to register the tag library information so that the right HTML code can be generated for preview. The extension point is `com.ibm.btt.tools.xui.editor2.taglib`.



A sample of the configuration is shown in below:

Extension Element Details

Set the properties of "taglib". Required fields are denoted by "**".

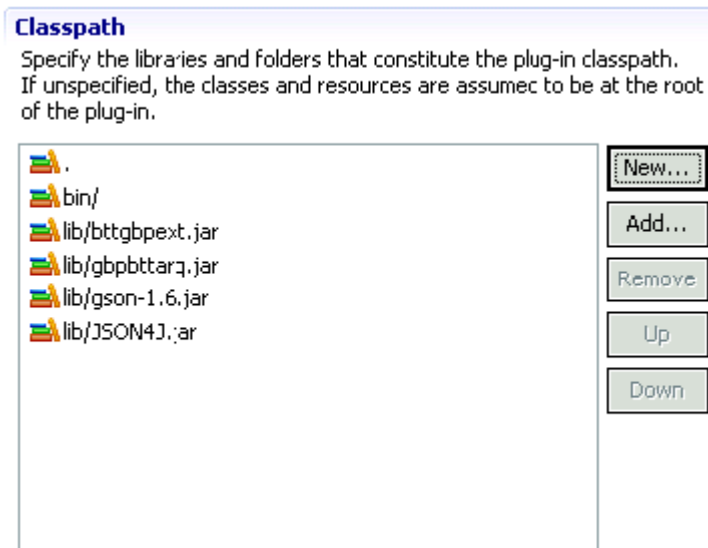
prefix*:

tld*:

- **prefix**: the prefix text for tags when mapping the widget to JSP.
- **tld**: the tag library file which follows the standard JSP tag library schema.

Adding a jar file containing JSP tag handler into classpath of widget extension project

- 1 Open `plugin.xml` and click the **Runtime** tab.
- 2 In the **Classpath** area, click **Add**.
- 3 Click the jar file that contains the customized JSP tag handlers.



Modify JSP template

As BTT supports developing applications in multi-project mode, some resources such as JavaScript files and image files are possibly not stored in the same project as the XUI file to be previewed. In order to make sure the resources in other projects could be loaded when XUI file is previewed, BTT lets Infrastructure developers add a special remote project prefix to load these resources.

The prefix is:

```
<%=JSPUtil.getRemoteProjectURL("[remote project key]")%>.
```

The parameter `remote project key` is the project key configured in `btt.xml` of the current project. For example, if `document.css` is stored in project `globalWAR` (key name) and at path `/js/dijit/themes/claro/`, Functional developers should add a snippet into the template file as following:

```
@import "<%=JSPUtil.  
    getRemoteProjectURL("globalWAR")%>  
    js/dijit/themes/claro/document.css"
```

Be aware that although the prefix is in JSP code style, it does not mean BTT supports JSP code in the template when the XUI file is previewed.

Advanced topics

In the previous sections of this chapter, we have described the primary steps of implementing a customized BTT widget. Now, we will go through some advanced topics when developing new widget.

Customized Property Editor

BTT provides several types of property editors for editing widget properties. They can be used to edit property types listed in Table X-Y. If there is other type of property such as Color, Infrastructure developers need to develop and register a new property editor.

Implement property editor

BTT follows the approach for implementing property editors of the Eclipse framework. All property editors must extend from the class `org.eclipse.ui.views.properties.PropertyDescriptor`. Infrastructure developers could either use the existing Eclipse property descriptor implementations or extend the class `PropertyDescriptor` to implement their own property editor.

Furthermore, BTT implements an abstract class `com.ibm.btt.tools.xui.editor2.properties.desc.SelectionPropertyDescriptor` for convenience of Infrastructure developers to implement property editors like Select List style. When extending the class `SelectionPropertyDescriptor`, Infrastructure developers need to override the method `protected String[] getSelections()` to return all the possible options.

Register property editor

To enable the customized property editor used in the XUI editor, Infrastructure developers need to register the property editor as an extension of the BTT plug-in. The following steps describe how to register a property editor:

- 1 Click the **Extensions** tab of `plugin.xml` file.
- 2 Click **Add**.
- 3 Click **com.ibm.btt.tools.common.properties**.
- 4 Click **Finish**.
- 5 Right click **com.ibm.btt.tools.common.properties**, then click **New > property**.
- 6 In the **Extension Element Details** dialog, type the applicable information.
 - **type**: the property type edited by this registered editor. The property type should be the same as the type attribute value defined in the widget definition xml file described in 'Define a widget in xml file' on page 19.
 - **class**: the implementation class of property editor.

Extension Element Details

Set the properties of "property". Required fields are denoted by "*".

type*:	<input type="text" value="simpleColor"/>
class*:	<input type="text" value="com.ibm.btt.alpha.sample.editor.properties.desc.SimpleColorPr"/> <input type="button" value="Browse..."/>
share:	<input type="text" value="true"/> ▼

New Property Tab

When Functional developers edit properties of the new widget, Appearance, Properties and Rules tabs will be shown by default. If the widget has action an property defined in the widget definition, the Action tab will be shown. If the widget has styleclass property in the widget definition, the Style tab will be shown.

BTT provides capabilities to add a new property tab for some specific property by following the Eclipse Tabbed Properties View implementation. If some property cannot be configured in the Properties tab due to its complexity, Infrastructure developers could add a new tab for it. For information on how to implement a property tab and register it as an extension into BTT plug-in, please refer to [Eclipse Tabbed Properties View](#).

When adding new extensions to `org.eclipse.ui.views.properties.tabbed.propertyTabs` and `org.eclipse.ui.views.properties.tabbed.propertySections` for the BTT XUI editor, the field `contributorId` needs to be set as `com.ibm.btt.rcp.xui.editor2.XUIEditor`.

If the new property tab is implemented for a property, Infrastructure developers need to set the `showInEditor` attribute of the property to be `false` in the widget definition file to ensure the property will not be edited in the Properties tab.

Customized Property Mapping Rule

As described in ‘Create widget mapping’ on page 25, if Infrastructure developers want to have more flexibility to customize property mapping, they should implement and register a property mapping rule for specific property type.

Implement property mapping rule

To implement property mapping rule, Infrastructure developers need to implement interface `com.ibm.btt.tools.xui.editor2.generator.IRule` and override the `process` method.

Register property mapping rule

To enable the implemented property mapping rule used when BTT generates a JSP file, Infrastructure developers need to register the rule as an extension of BTT plug-in. The following steps describe how to register a property mapping rule:

- 1 Click the **Extensions** tab of `plugin.xml`.
- 2 Click **Add**.
- 3 Click `com.ibm.btt.tools.xui.editor2.generator`.

- 4 Click **Finish**.
- 5 Right click **com.ibm.btt.tools.xui.editor2.generator** then click **New > property**.
- 6 In the **Extension Element Details** dialog, type the applicable information.
 - **name**: The name of the rule. It should be the same as the rule attribute value described in ‘Create widget mapping’ on page 25.
 - **class**: The implementation class of this rule.

Extension Element Details

Set the properties of "rule". Required fields are denoted by "**".

name*:

class*:

ECA support

BTT provides the ECA tool for Functional developers to handle JavaScript visually. Infrastructure developers may need to expose some functions or events of customized widget for the ECA tool.

Add Functions for widget

To add widget functions, Infrastructure developers need to define them in the widget definition file described in ‘Define a widget in xml file’ on page 19. Then these functions will be used in the ECA tool and be invoked according to ECA rule. Below is a sample definition:

```
<function name="setBalance"
    showInAction="true"
    showInExpression="true"
    returnType="Number"
    description="return balance of account" />
```

Add Events for Widget

To add widget events, Infrastructure developers need to register them into the widget definition file described in ‘Define a widget in xml file’ on page 19. Then these events will be used in the ECA tool and be triggered according to the ECA rule.

```
<event name="onClick"
    description="event when button is clicked" />
```

Monitor ECA Execution

BTT provides an ECA execution monitor to print ECA rule execution traces at runtime in browser console. For performance consideration, the monitor is disabled by default. If developers want to print rule execution traces during development, they can enable the ECA monitor by adding the following code in the template file for JSP generation:

```
engine.setMonitor(new com.ibm.btt.event.BaseMonitor());
```

The following is a sample output trace of ECA monitor in the browser console.

```

1 Monitor Event : form1.onLoaded
1 Start Rule : Object { evts=} for event form1.onLoaded
1 GetProperty Action : cbcrossbank.isChecked==false
1 Evaluated Condition : Condition = function(), Result=false
1 SetProperty Action : bank.readOnly=true
1 SetProperty Action : crossBankFee.visibility=hidden
1 End Rule : Object { evts=} for event form1.onLoaded
1 Monitor Event : form1.onLoaded
1 Start Rule : Object { evts=} for event form1.onLoaded
1 GetProperty Action : sms.isChecked==false
1 Evaluated Condition : Condition = function(), Result=false
1 SetProperty Action : cellphone.cellphone=true
1 SetProperty Action : cellphoneLabel.visibility=hidden
1 SetProperty Action : cellphone.visibility=hidden
1 End Rule : Object { evts=} for event form1.onLoaded
I18nBundle.getMessage(), key value pairs decimalPlacesMessage Object { value=2 }
1 Monitor Event : cbcrossregion.onChange
1 Start Rule : Object { evts=} for event cbcrossregion.onChange
1 GetProperty Action : cbcrossregion.isChecked==true
1 Evaluated Condition : Condition = function(), Result=true
1 SetProperty Action : region.readOnly=false
1 SetProperty Action : city.readOnly=false
POST http://localhost:8080/BPRuntimeTest/Ajax 200 OK 42ms
1 End Rule : Object { evts=} for event cbcrossregion.onChange

```

Customize Default Monitor

Infrastructure developers can customize the ECA rule monitor for their specific purpose. In this case, they need to write JavaScript code to extend the class `com.ibm.btt.event.BaseMonitor` and implement these methods:

- `monitorStartRule` : `function(event, rule)`. Invoked when begin to execute a rule.
- `monitorEndRule` : `function(event, rule)`. Invoked when complete executing a rule.
- `monitorCondition` : `function(event, rule, result)`. Invoked when complete a condition evaluation.
- `monitorCallFunctionAction` : `function(id, functionName, args, result)`. Invoked when complete calling a function.
- `monitorGetPropertyAction` : `function(id, property, value)`. Invoked when complete retrieving a property from a widget.
- `monitorSetPropertyAction` : `function(id, property, value)`. Invoked when complete setting a property to a widget.

NLS support

Eclipse already provides for NLS support and BTT follows the same way to enable a widget supporting NLS. As described above, many description and label fields support NLS. To leverage NLS provided by Eclipse and BTT, Infrastructure developers need to:

- Define Bundle-Localization path in MANIFEST.MF

Bundle-Localization entry defines which property files are loaded at runtime for NLS. This entry definition contains the name and the corresponding property file used to translate the plug-in strings that start with the prefix `%`. Below is a sample definition entry.

Bundle-Localization: plugin

- Move translatable strings into property files

When the Bundle-Localization path has been defined, Eclipse will use the file [path]_[locale].properties for specific locale. For example, if the path is set to plugin, then Eclipse will use the file plugin.properties for default locale, use file plugin_zh.properties to support a Chinese locale and use file plugin_es.properties to support a Spanish locale. Infrastructure developers need to move translatable strings into specific properties file to support the specific language.

- Use % strings for NLS support property or attribute

For example, the description attribute of event tag supports NLS. To define the description attribute, Infrastructure developers need to use %[description]. See the example below:

```
<event name="onClick" description="%desc_event_onclick" />
```

Meanwhile, Infrastructure developers need to add description messages into different properties files. For example, the plugin.properties file could contain:

```
desc_event_onclick = event triggered when the widget is clicked
```

Please access the [Internationalize your Eclipse Plug-In](#) on the Eclipse website to get more information about how Eclipse plug-ins supports NLS.

BTT Context data binding

Like BTT original widgets, customized widgets can be easily bundled with BTT Context data. Infrastructure developers need to do nothing in code. If a widget needs to be bundled with a DataField type data, Infrastructure developers need to define a dataName type property named dataName. If a widget needs to be bundled with Collection type data, Infrastructure developers need to define a dataNameList type property named dataNameForList. BTT runtime will assign value of specified Context field into this attribute.

XUI Generation Template

When the BTT XUI editor generates an XUI file into a JSP file, it uses a template file. The template file should include common content of a JSP page, such as charset, included css files and js files. By default, BTT provides two template files which are in the WebContent/templates folder (the folder can be configured in XUI Default Settings of XUI Web Project Properties Dialog) of a XUI Web project:

- **template_debug.ftl**: enables ECA debug console for debugging purpose.
- **template_ftl**: disables the ECA debug console for higher performance at runtime. Infrastructure developers can implement project-specific template file, and put it into the WebContent/templates folder. When the XUI editor generates an XUI file into a JSP file, all template files in this folder will be shown as candidate templates for user to choose. The template files should follow the FreeMarker specification and standard FreeMarker directives can be used in template files. Furthermore, the following variables are supported in template files:
 - **content**: represents all the content of a specific JSP file in text format.
 - **user**: the user who generates this JSP file

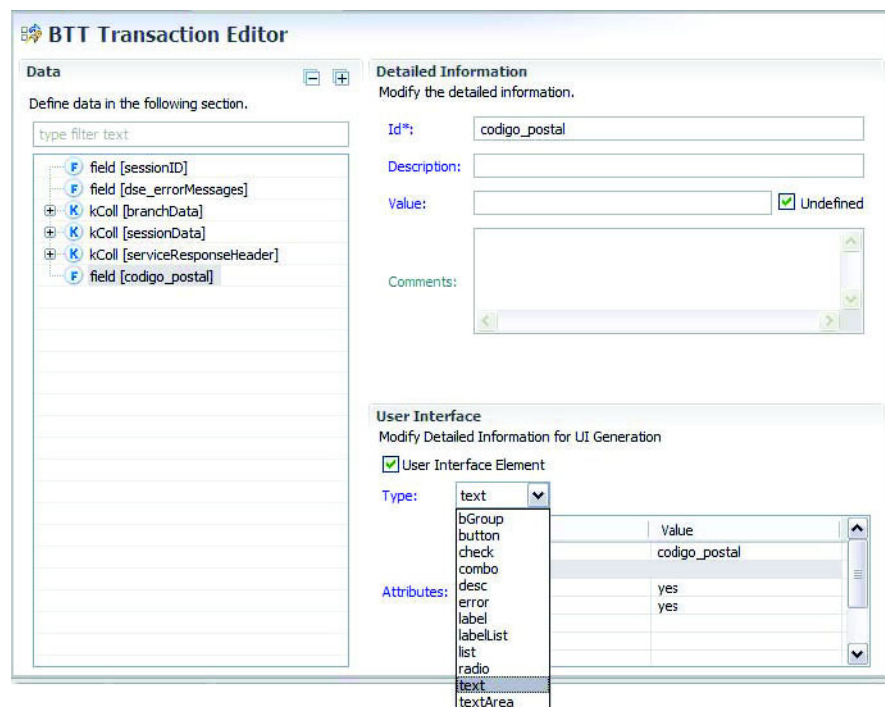
- **date:** the date when generates this JSP file.
- **cssFiles:** the list of cssFiles which should be included in this JSP file.
- **xui_file:** the name of the XUI file that generates this JSP file.
- **js_file:** the list of js files which should be included in JSP file
- **encoding:** the charset of this JSP page

Please refer to the page

http://freemarker.sourceforge.net/docs/dgui_quickstart_template.html for more information about the FreeMarker template schema.

XUI page generation from BTT context data

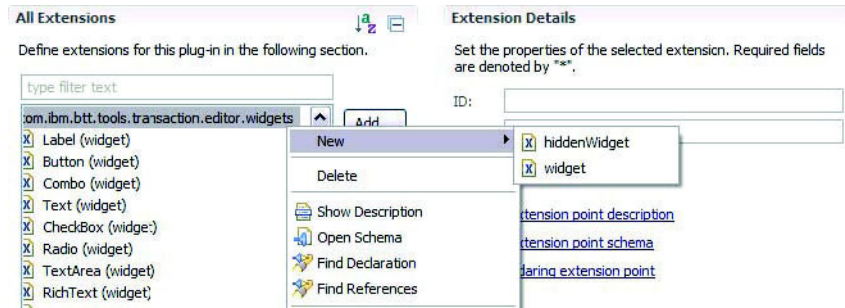
BTT tools provide the function to generate an XUI page skeleton from transaction context data. With this feature, Functional developers do not need to create the XUI page from scratch. This greatly improves the productivity of application development. When a project-specific widget is created, Infrastructure developers can register this new widget into the candidate widget list of the transaction editor shown as below:



The following steps describe how to register a widget for XUI page skeleton generation:

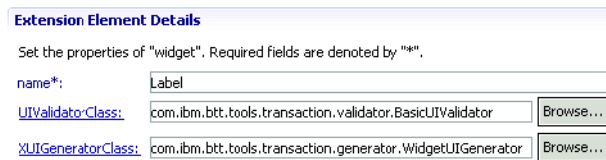
Register new extension point

- 1 Right click `com.ibm.btt.tools.transaction.editor.widgets` then click **New > widget**.



2 In the **Extension Element Detail** dialog, type the applicable information.

- **name:** the name of the widget. This name should be the same as the one registered in the extension of `com.ibm.btt.tools.xui.editor2.widgets` described in ‘Import widget’ on page 23.
- **UIValidatorClass:** the implementation class that decides if the widget will be shown in the candidate widget list when a specific data element is selected. The class should implement the interface `com.ibm.btt.tools.transaction.validator.WidgetValidator`.
- **XUIGeneratorClass:** The implementation class that generates the widget to XUI file. Infrastructure developers can either use BTT default XUI widget generation class `com.ibm.btt.tools.transaction.generator.WidgetUIGenerator` or implement specific widget generation class by implementing the interface `com.ibm.btt.tools.transaction.generator.WidgetGenerator`.



Implement required classes

- **UIValidatorClass:** The implementation class must implement the `validate` method of the interface `WidgetValidator`. The following is the description of this method:

```
/**
 * Returns true or false depending if the widget makes sense for the data passed as parameter or not.
 * @param data the data to be validated
 * @return true if the widget makes sense for data passed as parameter. False otherwise.
 */
public boolean validate(MetaData data);
```

- **XUIGeneratorClass:** Please refer to ‘Create widget mapping’ on page 25 for more details about how to implement a widget specific generation class.

Change default behavior of XUI generation

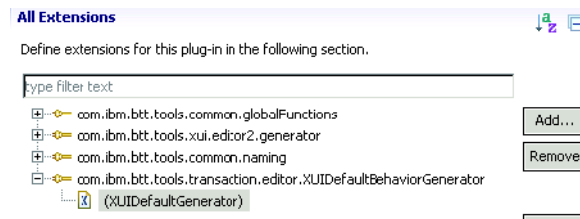
BTT implements the default behavior of generating the XUI page skeleton from transaction context data. For example, a Text widget will be used for field type data and Table widget will be used for iColl type data. It is also possible for Infrastructure developers to override this default BTT behavior such as using a customized widget for iColl type data.

There are two ways for Infrastructure developers to change the default behavior of XUI generation:

- **Simple way:** Infrastructure developers could give some controls on BTT default generation behavior without the effort of implementing a new XUI generation class. They can change the default behavior by modifying the configuration file of default BTT XUI generator. This way fits for changing or creating the mapping between data type and widget.
- **Full control way:** Infrastructure developers can implement and register a new XUI generator to have the full control on XUI generation. The following steps described how to change default BTT behavior of generating the XUI page skeleton.

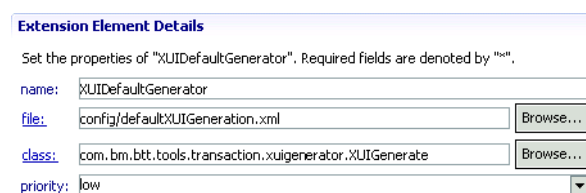
Register an extension

- 1 Add a new XUIDefaultGenerator element to the extension **com.ibm.btt.tools.transaction.editor.XUIDefaultBehaviourGenerator**.



- 2 In the **Extension Element Details** dialog, type the applicable information.

- **name:** The name of the generator which should be unique.
- **file:** the configuration file for the generator.
- **class:** the implementation class of the generator. Infrastructure developers can either use the default BTT class `com.ibm.btt.tools.transaction.xuigenerator.XUIGenerate` or a new class which implements the interface `com.ibm.btt.tools.transaction.xuigenerator.IXUIGenerate`.
- **priority:** the priority of this extended generator. If the extension point is registered by multiple plug-ins, the highest priority extension will be invoked by BTT. BTT default generator is registered as low priority. So the customized generator should be registered as medium or high priority.



Modify default configuration file

The default configuration file of XUI generation maintains the mapping information between XUI widget and context data type. The following is the content in BTT default configuration file. Infrastructure developers can create a new file to create a new mapping entry or modify the existing mapping entries.

```
<defaultXUIGeneration>
  <widgetMappings>
    <widgetMapping dataElement="field" widgetName="Text"/>
    <widgetMapping dataElement="iColl" widgetName="Table"/>
    <widgetMapping dataElement="kColl" widgetName="Combo"/>
    <widgetMapping dataElement="data" widgetName="Text"/>
  </widgetMappings>
</defaultXUIGeneration>
```

Note The dataElement value must match with the one defined in btt.xml file (data > classTable section). And the widgetName value must be the widget name that defined in the extension of com.ibm.btt.tools.xui.editor2.

Implement new XUI generation class

BTT provides the capability for Infrastructure developers to implement and register a new XUI generation class to have full control when generating an XUI page skeleton from transaction context data. The class should implement the interface com.ibm.btt.tools.transaction.xuigenerator.IXUIGenerate. The following are the API descriptions of IXUIGenerate interface:

- public void generateXUIFile(String folder, String fileName, String XUITemplateName, List <MetaData> dataToBeGenerated, String contextName)

The method performs XUI generation for data passed as a parameter named dataToBeGenerated. This data is in the context hierarchy specified as argument. In case of using a template (XUITemplateName parameter), the generated content is located in the first form of the template. The generation is performed in the file with the fileName and in the folder specified as parameter folder.

- public void generateXUIFile(String folder, String fileName, String XUITemplateName, List <MetaData> dataToBeGenerated, String contextName, Hashtable <String, String> defaultMapping)

The method performs XUI generation for data passed as a parameter named dataToBeGenerated. This data is in the context hierarchy specified as argument named contextName. For data included in defaultMapping, the widget generated is the one specified in this hashtable. So this hashtable must contain keys as the following values: IXUIGenerate.FIELD, IXUIGenerate.KEYED_COLLECTION, IXUIGenerate.INDEXED_COLLECTION and IXUIGenerate.DATA and corresponding values must a widget name.

In case of using a template (XUITemplateName parameter), the generated content is located in the first form of the template. The generation is performed in a file with the fileName and in the folder specified as parameter folder.

- public IRootModel generateXUIFile(String XUITemplateName, List <MetaData> dataToBeGenerated, String contextName)

The method performs XUI generation for data passed as a parameter named `dataToBeGenerated`. This data is in the context hierarchy specified as argument named `contextName`. In case of using a template (`XUITemplateName` parameter), the generation content is located in the first form of the template. An `IRootModel` object is returned containing the generation result.

- `public IRootModel generateXUIFile(String XUITemplateName, List<MetaData> dataToBeGenerated, String contextName, Hashtable<String, String> defaultMapping)`

The method performs XUI generation for data passed as a parameter named `dataToBeGenerated`. This data is in the context hierarchy specified as argument named `contextName`. For data included in `defaultMapping`, the widget generated is the one specified in the hashtable. This hashtable must contain keys as the following values: `IXUIGenerate.FIELD`, `IXUIGenerate.KEYED_COLLECTION`, `IXUIGenerate.INDEXED_COLLECTION` and `IXUIGenerate.DATA`.

In case of using a template (`XUITemplateName` parameter), the generated content is located in the first form of the template. An `IRootModel` object is returned containing the generation result.

Extend Table Column Widget

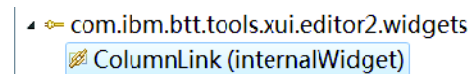
BTT has provided several default definitions for table column widgets in the XUI Editor. Besides, the widgets displayed in a table column, their properties and the rules for JSP generation could also be extended through Eclipse extension points and some XML definitions.

Define Extension for Table Column Widget

If Infrastructure developers need to customize table column widgets in his self-defined plug-in project, the prerequisite is to add the plug-in `com.ibm.btt.tools.xui.editor2` as the plug-in dependency.

- 1 Right click `com.ibm.btt.tools.xui.editor2.widgets` then click **New > widget**.

If	Then
Infrastructure developers need to define a common XUI widget which could also be used as a table column widget.	Click widget
Infrastructure developers need to define a widget that is only for use as a table column widget.	Click internalWidget



- 2 In the **Extension Element Details** dialog, type the applicable information. for **name**, **label**, **icon** and **config**. For the attribute **config**, you should choose or specify a relative path of the widget definition file. Later, Infrastructure developers will take steps to create an XML file to describe this widget.

name*:	<input type="text" value="ColumnLink"/>
label*:	<input type="text" value="ColumnLink"/>
icon*:	<input type="text" value="icons/ColumnLink.gif"/> <input type="button" value="Browse..."/>
config*:	<input type="text" value="widgets/ColumnLink.xml"/> <input type="button" value="Browse..."/>
description:	<input type="text"/>

Configure Detailed Definition for Table Column Widget

In XUI Editor, Infrastructure developers could follow the same style to describe a column widget just like the common widget definition. For facility in this extension sample, we have copied the xml definition file of the link widget (Link.xml) in the BTT product and made some modifications on it with below steps:

- 1 Rename the **Link.xml** to **ColumnLink.xml**.
- 2 Move to the folder **widgets**.
- 3 Delete the **functions** and **events** sections from the file
- 4 Add a tag named **columnWidget** to indicate this widget will be used for table column `<columnWidget editable="false"/>`

The details of available attributes for tag **columnWidget** are listed in below table.

Attribute	Description
name	The attribute to identify the column widget. It is optional and the widget name registered in extended plug-in will be used if this attribute is not defined.
editable	The available values are 'true' and 'false' which indicate whether the widget could be chosen as an editable one or not for a table. Besides, it is optional and the widget could be chosen from both the editable and read-only widget list in table properties view.
addTypeInfo	This attribute is optional and asks for a Boolean value. It will be used if the widget needs to be bound with a BTT typed data. If it is set to true, the generated jsp tag of the widget will contain the attribute named 'type'.

The sample code snippet of this column widget definition is listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://btt.ibm.com/WidgetSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/WidgetSchema WidgetSchema.xsd ">
  <columnWidget editable="false" />
</widget>
```

- 5 Add or remove **property** tag in the properties block to indicate the properties of the widget.

```
<property name="visibility" default="visible"
  type="Visibility" showInColumn="true"/>
```

The details of the available attributes for tag property are listed in below table.

Attribute	Description
name	The attribute is used to define the property name. It is required; otherwise this property will be ignored during generation.
type	The attribute is used to indicate which property editor will be used for this property. The value of it should equals with one of the registered property editor ID in existing BTT toolings. It is required; otherwise the property editor will be disabled in editor.
default	The attribute is used to indicate the default value of this property. The generated JSP tag will contain this default value attribute if it is defined. It is an optional one.
showInColumn	The attribute asks for a Boolean value which defines whether this property is available when this widget is used for a table column.

The sample code snippet of this column widget definition is listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://btt.ibm.com/WidgetSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/WidgetSchema WidgetSchema.xsd ">
  <columnWidget name="ColumnLink" editable="false"/>
  <properties>
    <property name="id" type="String" showInColumn="true"/>
    <property name="text" type="NLS" showInColumn="true"/>
    <property name="target" default="_parent" type="Target" showInColumn="true"/>
    <property name="styleClass" type="String" showInColumn="true"/>
  </properties>
</widget>
```

Configure Mapping Rules for Table Column Widget

There are more details about this topic in ‘Customized Property Mapping Rule’ on page 33.

If there is some special logic put in during the JSP generation, such as extra conversion for the names and values of JSP tag attributes, you need to define a mapping rule for the column widget.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings prefix="bttdojo:">
  <widget-mapping widgetName="ColumnLink" tagName="a">
    <property-mapping propName="text" rule="remoteNLSPathRule" />
    <property-mapping propName="styleClass" attrName="class" />
  </widget-mapping>
</mappings>
```

Taking the above snippet as an example, the tagged block `widget-mapping` matches with the generation rules for the sample column widget `ColumnLink`. The attributes for the tag `widget-mapping` are listed in below table.

Attribute	Description
widgetName	The attribute is used to link the generation rules with the registered column widget. It is required otherwise the rules could not be assigned correctly. So its value should equals with the column widget identifier.
tagName	The attribute is required and used to indicate the JSP tag name for the column widget generation.

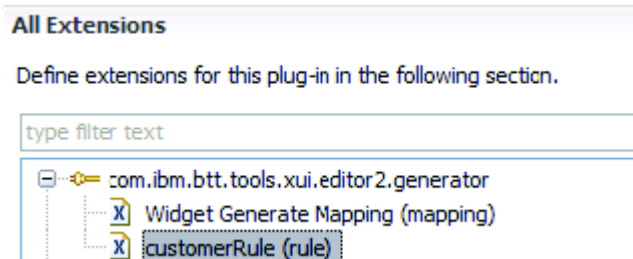
In the above sample snippet, the tag `property-mapping` matches with the detailed mapping policy for each widget property during generation. The attributes for tag `property-mapping` are listed in below table.

Attribute	Description
propName	The attribute is used to indicate the identifier of the property which needs extra generation logic.
attrName	The attribute is used to indicate that the property name will be replaced with the value of <code>attrName</code> during JSP generation. It is used for conversion of JSP attribute name.
rule	The attribute is used to indicate the identifier of a registered mapping rule. It is used for conversion of JSP attribute value.

Define Extension for Mapping Rule of Table Column Widget

There are more details about this topic in ‘Customized Property Mapping Rule’ on page 33.

Finally, if Infrastructure developers decides to use the extra rule to control the value generation of JSP attributes, they also need register the rule into BTT tooling extension to indicate the path of the configuration file which contains the mapping rules. To do that, they could define an extension for the extension point `com.ibm.btt.tools.xui.editor2.generator`. Then as in the graphic below, add a child option of type `rule`.



As the graphic below shows, the required attributes `name` and `class` should be configured. Especially for the attribute `class`, Infrastructure developers need to indicate the full path to a class which implements the interface `com.ibm.btt.tools.xui.editor2.generator.IRule`. During the JSP generation, the class would be instantiated and executed for the configured mapping rules.

Extension Element Details

Set the properties of "rule". Required fields are denoted by "**".

name*:

class*:

Infrastructure developers need to add code for the attribute value conversion as the graphic shows.

```
@Override
public void process(Map<String, String> attributes, String property,
    PropInfo info, IRootModel root, ICommonModel widgetModel) {

    System.out.println("execute in customer rule.....");
    String value = widgetModel.getPropertiesValues().get(property);
    if(info!=null){
        if(info.attrName!=null)
            property=info.attrName;
    }
    if(value!=null){
        value=":"+value+":";
        attributes.put(property, value);
    }
}
```

How to add version control on runtime NLS files

BTT provides the basic infrastructure to support project-specific version control of runtime NLS files. In this chapter, Infrastructure developers will be guided to implement their version control logic which governs runtime NLS files by timestamp.

Customize tooling behavior of NLS file generation

For the tooling side, BTT provides the default framework for runtime NLS control.



CHAPTER 4

Data Type Extension

BTT typed data elements represent business objects such as Date, ProductNumber and Money. Compared with non-typed data element, a typed data element identifies how BTT displays the business object and what validation must occur when BTT changes a data value. BTT types can be a simple type or a compound type. A simple type only has a single property while a compound type has multiple properties. By default, BTT provides four basic types, such as String, Number, Date and Currency. BTT also provides the capability for Infrastructure developers to implement project-specific data types.

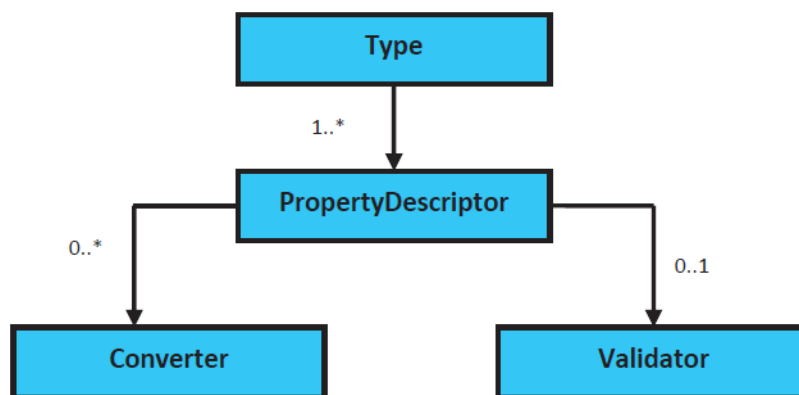
Implement data type extension

Declare new data type

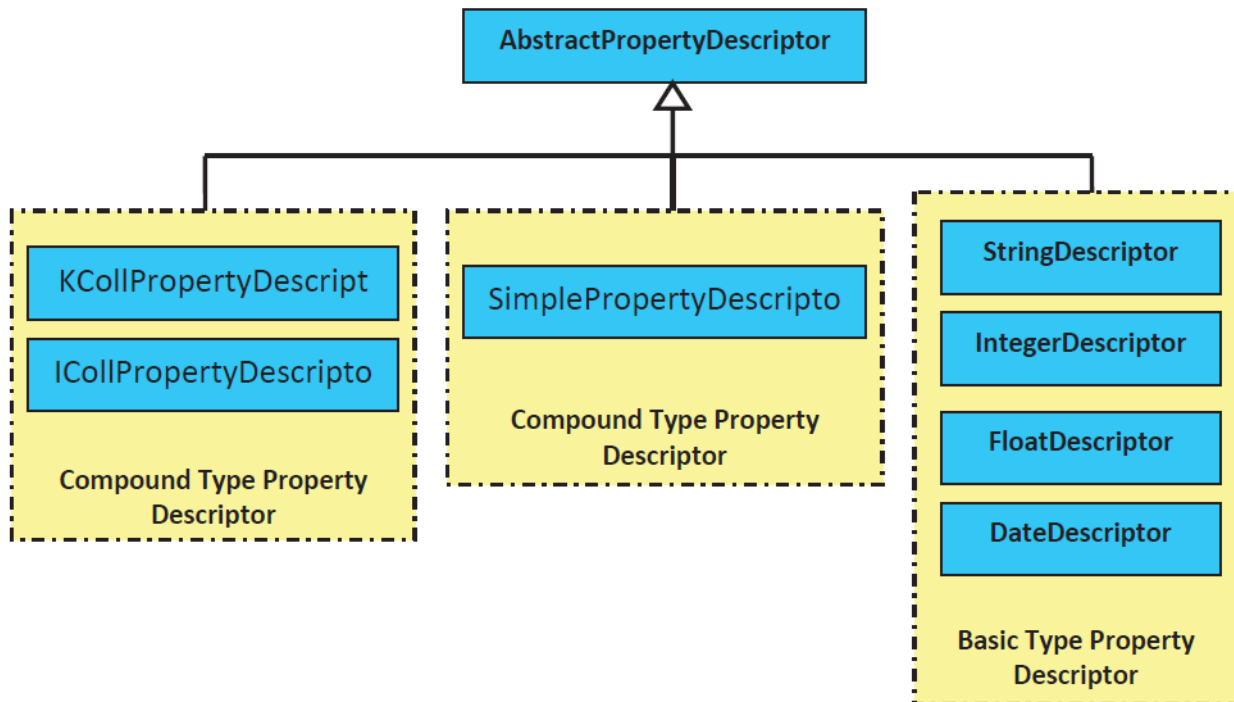
To create a project-specific data type, firstly Infrastructure developers need to declare this new type in type.xml file. The declaration includes:

- **Type id**
It is the name of the type and should be unique.
- **Implementation class**
It defines the implementation class of the type. For simple type which has only one property, the implementation class is `com.ibm.btt.base.DataField`. For compound type which has multiple properties, the implementation class is `com.ibm.dse.base.KeyedCollection` or `com.ibm.btt.base.IndexedCollection`.
- **Property Descriptor**
Property descriptor specifies the default business rules and behaviors for this data type. A type can have one or more property descriptors. For a simple type, it has only one property descriptor. For a compound type, it has multiple property descriptors. A property descriptor can have only one validator which is used to check the data instance, and one or multiple converters which are used to transform the data instance into a specific format.

The below figure shows the relationship of type, property descriptor, converter and validator.



In most cases, Infrastructure developers do not need to implement their own property descriptor class. BTT provides `SimplePropertyDescriptor` for simple types and provides `KCollPropertyDescriptor` and `ICollPropertyDescriptor` for compound types and provides `StringDescriptor`, `IntegerDescriptor`, `FloatDescriptor`, and `DateDescriptor` for the basic types. Infrastructure developers can also implement their own property descriptor for some special cases by extending `AbstractPropertyDescriptor` which is the parent for all property descriptors. The graphic below shows hierarchy of property descriptor classes.



Below is an example of a type definition:

```

<type id="Currency" implClass="com.ibm.btt.base.DataField">
  <Descriptor id="typeDefault"
    implClass="com.ibm.btt.base.types.impl.SimplePropertyDescriptor">
    <Converter convTypes="default"
      implClass="com.ibm.btt.base.types.impl.CurrencyConverter">
    </Converter>
  </Descriptor>
</type>

```

Implement type validator

A validator ensures that the typed data element conforms to the business rules of its binding type. The property descriptor definition of the type specifies the validator. If the property descriptor does not specify a validator, all values for the data element are valid.

A validator can have validation parameters. For example, a validator for a date type checks whether the value to be validated lies within limits defined by parameters such as `lowerLimit` and `upperLimit`.

Infrastructure developers do not need to implement the validator from scratch. BTT provides `com.ibm.btt.base.types.impl.BaseValidator` as super class of all validator implementations. Infrastructure developers can extend the class and override `validate` method.

Below is sample code of `validate` method implementation:

```

public void validate(TimeZone convertedValue, TimeZoneValidationParamBean params)
throws DSETypeException {

    // TODO Auto-generated method stub

    if(convertedValue != null && params.maxoffset != null &&
        !params.maxoffset.equals("")){

        if (TimeZone.getTimeZone(params.maxoffset).getRawOffset() >
            convertedValue.getRawOffset()){

            String msg = "validation failed in " +
                this.getClass().getName() +
                ".The offset of TimeZone '" +
                convertedValue.getID() +
                "' should be smaller than '" +
                params.maxoffset + "'";

            throw new DSETypeException(DSETypeException.harmless, "", msg);

        }

    }

    if(convertedValue != null && params.minoffset != null &&
        !params.minoffset.equals("")){

        if (TimeZone.getTimeZone(params.minoffset).getRawOffset() <
            convertedValue.getRawOffset()){

            String msg = "validation failed in " +
                this.getClass().getName() +
                ". The offset of TimeZone '" +
                convertedValue.getID() +
                "' should be larger than '" +
                params.maxoffset + "'";

            throw new DSETypeException(DSETypeException.harmless, "", msg);

        }

    }

}

```

Implement type converter

Converters transform business objects to Strings (formatting) and Strings to business objects (unformatting).

A converter can have conversion parameters when convert the type to or from String. For example, a converter for Date type can have pattern parameter, which defines the format of a date String in 'YYYY-MM-DD' or 'MM-DD-YYYY' or other formats.

Infrastructure developers need not to implement converter from scratch. BTT provides `com.ibm.btt.base.types.impl.BaseConverter` as super class of all converter implementations. Infrastructure developers can extend the class and override `format` and `unformat` methods.

```

public abstract String format(K value, T params, String convType, Locale locale)
throws DSETypeException;

```

```
public abstract K unformat(String value, T params, String convType,  
                           Locale locale) throws DSETypeException;
```

Below is simple code of implementing the two methods:

```
public String format(TimeZone value,  
com.ibm.btt.base.types.impl.BaseConverter.FormatParamBeam params, String  
convType, Locale locale) throws DSETypeException {  
  
    // TODO Auto-generated method stub  
    //in the formate of: GMT Sign TwoDigitHours : Minutes  
    return value.getID();  
  
}  
  
@Override  
  
public TimeZone unformat(String value,  
com.ibm.btt.base.types.impl.BaseConverter.FormatParamBeam params, String  
convType, Locale locale) throws DSETypeException {  
  
    // TODO Auto-generated method stub  
    //in the formate of: GMT Sign TwoDigitHours : Minutes  
    return TimeZone.getTimeZone(value);  
  
}
```

Implement type presentation widget

A simple type can be integrated with the BTT XUI Editor. That means, when a field in specific type is selected as `dataName` of `TextBox` widget, XUI Editor will generate specific widget for this type automatically. For example, if a `Date` type field is chosen, a `DateTextBox` will be generated as presentation widget automatically. Infrastructure developers need to implement a presentation widget for a simple type. Infrastructure developers need three steps to register and implement presentation widget for a new simple data type.

- Extending BTT JSP tag handler for `TextBox` widget
- Modifying `bttdojo.tld`
- JavaScript implementation.

Extend BTT JSP tag handler for `TextBox` widget

To extend the BTT JSP tag handler for a `TextBox` widget, Infrastructure developers need to extend `com.ibm.btt.dojo.tag.DojoTextBoxTag` class and override `getWidgetType` method. The `DOJO` widget class for the new type can be returned if `dataName` of `TextBox` in this type.

Below is the implementation code:

```
protected String getWidgetType(String type) {
    if ("TimeZone".equalsIgnoreCase(type)) {
        return "com.ibm.btt.digit.TimeZoneTextBox";
    }
    else{
        return super.getWidgetType(type);
    }
}
```

Modify `bttdojo.tld`

After extending the JSP tag handler for the `TextBox` widget, Infrastructure developers need to modify `bttdojo.tld` file to use the new tag handler for the `TextBox` widget. Infrastructure developers can search tag with name `textbox` and change the tag-class to be the class implemented previously

JavaScript implementation

To implement JavaScript for the new data type as a presentation widget, Infrastructure developers need to extend the BTT base class `com.ibm.btt.digit.ValidationTextBox` and override `validator` method.

Bean Property Converter

For web applications, most data transferred through http is in text format, such as user information, session ID, and even some complex data types are always kept in text format. The bean property converter provides typed data element to handle the conversion between plain text and Java object.

Infrastructure developers can use the steps that follow to implement the bean property converter:

- 1 Implement bean property converter class.
- 2 Register bean property converter.

Data type extension sample

A data type extension sample is provided to demonstrate how to implement data type extension in the BTT framework. In the sample, we will implement a simple `TimeZone` data type to demonstrate all the tasks necessary to extend a data type described previously.

`TimeZone` is a common data type in business. In this sample, we require a `TimeZone` type in the format of 'GMT+(-)DD' where DD is two digits with scope from 0 to 12. The `TimeZone` type can have two validation parameters: `minoffset` and `maxoffset`, which can limit the valid scope of `TimeZone` data for a specific purpose. The procedure that follows describes the primary steps of implementing this data type.

- 1 Define the data type.
 - a Add the lines below into `types.xml`.

```
<type id="TimeZone" implClass="com.ibm.btt.base.DataField">
  <Descriptor id="typeDefault"
    implClass="com.ibm.btt.base.types.impl.SimplePropertyDescriptor">

    <Converter convTypes="default"
      implClass="com.ibm.btt.alphatest.types.impl.TimeZoneConverter">
    </Converter>

    <Validator
      implClass="com.ibm.btt.alphatest.types.impl.TimeZoneValidator"/>
  </Descriptor>
</type>
```

- 2 Implement the `TimeZone` validator.
 - a Create new class
`com.ibm.btt.alphatest.types.impl.TimeZoneValidator` that extends `com.ibm.btt.base.types.impl.BaseValidator`
 - b Override the `validate` method.
 - c Create the class `TimeZoneValidationParamBean` in `TimeZoneValidator`, which extends from `BaseValidator.ValidationParamBean`.
 - d Add the validation parameter `maxoffset` as a public variable.
 - e Add the validation parameter `minoffset` as a public variable.

```
public static class TimeZoneValidationParamBean extends
    BaseValidator.ValidationParamBean{

    //in format:   GMT Sign TwoDigitHours : Minutes

    public String maxoffset;
    public String minoffset;
}
```

- 3 Create the `TimeZoneConverter` for the `TimeZone`.
 - a Create a new class `com.ibm.btt.alphatest.types.impl.TimeZoneConverter` that extends `com.ibm.btt.base.types.impl.BaseConverter`.
 - b Override the `format` method.
 - c Override the `unformat` method.
- 4 Create the presentation widget.
 - a Create a new dojo widget for `TimeZone` with the name `TimeZoneTextBox`.
 - b Create a validate method validator: `function(/*anything*/value, /*dijit.form.ValidationTextBox.____ Constraints*/constraints)`

Note This validates that the input text is in the expected format of `TimeZone`.

- c Create a new class `com.ibm.btt.alphatest.dojo.tag.AlphaTextBoxTag` that extends class `DojoTextBoxTag`.

Note After implementing dojo widget for `TimeZone` type, we need to ensure the `TimeZoneTextBox` can be generated as an input text box for `TimeZone` type data when BTT generates HTML from JSP. By default, BTT runtime generates JSP file to HTML file, it uses `com.ibm.btt.dojo.tag.DojoTextBoxTag` to generate tag for input data. `DojoTextBoxTag` can only handle original BTT data types such as `String`, `Number`, `Currency`, etc. It can not generate tags for new data types.
- d Override the `getWidgetType` method to generate new widget tag for `TimeZone` data type.

```
protected String getWidgetType(String type) {
    if ("TimeZone".equalsIgnoreCase(type)) {
        return "com.ibm.btt.dijit.TimeZoneTextBox";
    }
    else{
        return super.getWidgetType(type);
    }
}
```

- 5 Update `bttdojo.tld` to replace `DojoTextBoxTag` by `AlphaTextBoxTag`.
 - a Open `bttdojo.tld` and identify the `DojoTextBoxTag` class.

```
<tag>
  <name>textbox</name>
  <tag-class>com.ibm.btt.dojo.tag.DojoTextBoxTag</tag-class>
```

- b Change the `DojoTextBoxTag` to the `AlphaTextBoxTag` class.

```
<tag>
  <name>textbox</name>
  <tag-class>com.ibm.btt.alphatest.dojo.tag.AlphaTextBoxTag</tag-class>
```

- 6 Export and copy the data type extension files.
 - a Run the new Eclipse Application described in ‘Runtime Project Setup’ on page 16.
 - b Export the AlphaWidget project as a plug-in and copy it to the RAD plug-in folder.
 - c Export the classes below to a jar file named `alphatype.jar`.


```
com.ibm.btt.alphatest.types.impl.TimeZoneConverter
com.ibm.btt.alphatest.types.impl.TimeZoneValidator
com.ibm.btt.alphatest.dojo.tag.AlphaTextBoxTag
```
 - d Copy `types.xml` and `bttdojo.xml` from the AlphaWidget project to the location of the `BTTExtensionWeb` project.
 - e Copy `alphatype.jar` to `WebContent\WEB-INF\lib` folder of `BTTExtensionWeb` project.

- 7 Create the test.
 - a Create a new operation definition file named `datatypeExtensionOp.xml`.
 - b Create an operation named `datatypeExtensionOp` with an the `implClass` attribute of `com.ibm.btt.sample.operation.DataTypeExtensionOperation`.
 - c Create a context with an `id` attribute of `datatypeExtensionCtx` and a `type` attribute of `oper`.
 - d Create that applicable reference elements for the context. See example below.

```
<datatypeExtensionOp.xml>

<!-- This operation gets from the context a field containing the page
      wanted to be shown to the user and places it in the right place to
      make Composer understand that this page must be shown. -->
<!-- Operation definition -->
<operation context="datatypeExtensionCtx" id="datatypeExtensionOp"
      implClass="com.ibm.btt.sample.operation.DataTypeExtensionOperation">
</operation>

<context id="datatypeExtensionCtx" type="oper">
  <refKColl refId="datatypeExtensionData" />
</context>

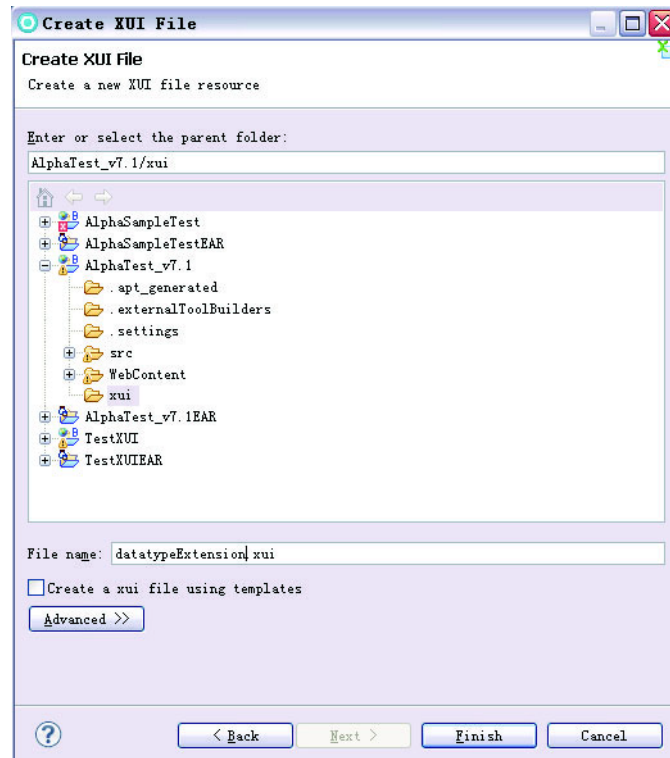
<kColl id="datatypeExtensionData">
  <refData id="preferTimeZone" />
  <field id="timezone" refType="TimeZone"/>
  <kColl id="timezoneList">
    <field id="GMT+1" value="GMT+1"/>
    <field id="GMT+2" value="GMT+2"/>
    <field id="GMT+3" value="GMT+3"/>
    <field id="GMT+4" value="GMT+4"/>
    <field id="GMT+5" value="GMT+5"/>
    <field id="GMT+6" value="GMT+6"/>
    <field id="GMT+7" value="GMT+7"/>
    <field id="GMT+8" value="GMT+8"/>
    <field id="GMT+9" value="GMT+9"/>
  </kColl>
</kColl>
<data id="preferTimeZone" refType="TimeZone">
  <param value="true" id="isMandatory"/>
  <param value="GMT+8" id="maxoffset"/>
  <param value="GMT+6" id="minoffset"/>
</data>
</datatypeExtensionOp.xml>
```

- 8** Create a class `com.ibm.btt.sample.operation.DataTypeExtensionOperation` that extends the class `BTTServerOperation` and override the `execute` method with the code below.

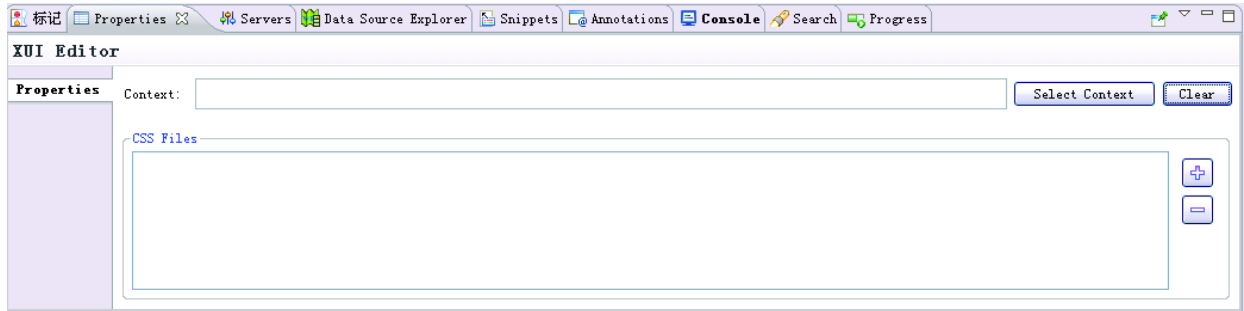
```
System.out.println("DataTypeExtensionOperation");
setValueAt(HtmlConstants.REPLYPAGE, "datatypeExtension.jsp");
```

Note As this is a very simple sample, the class just extends `BTTServerOperation` and overrides the `execute` method and no business logic is implemented.

- 9** Create Test XUI file.
- a** Expand the **Alpha_Testv7.1** project.
 - b** Right click the `xui` folder.
 - c** Click **New > Other > New XUI File**.
 - d** In **File name** field, type `datatypeExtension.xui`.
 - e** Click **Finish**.

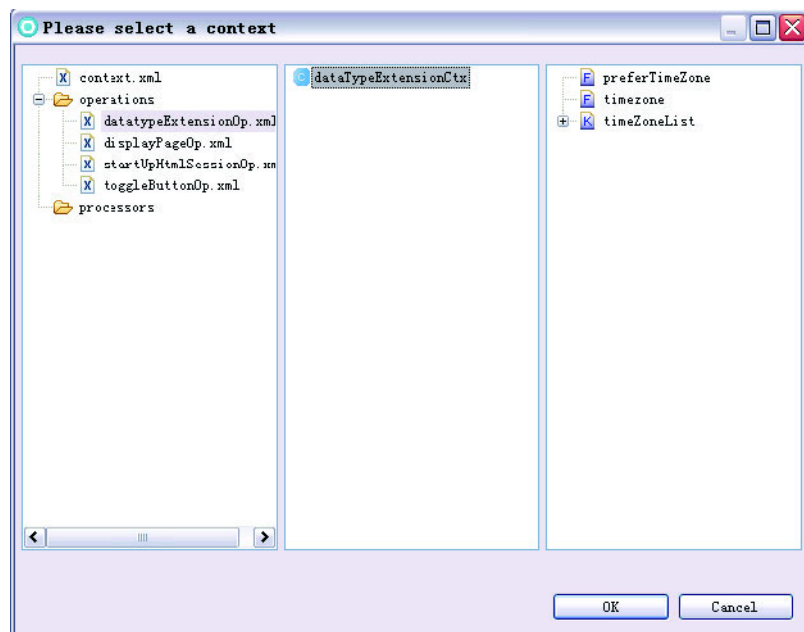


- 10** Open `datatypeExentsion.xui` with XUI editor.
- 11** Click the grey area and open the **Properties** tab.
- 12** Click **Select Context**.



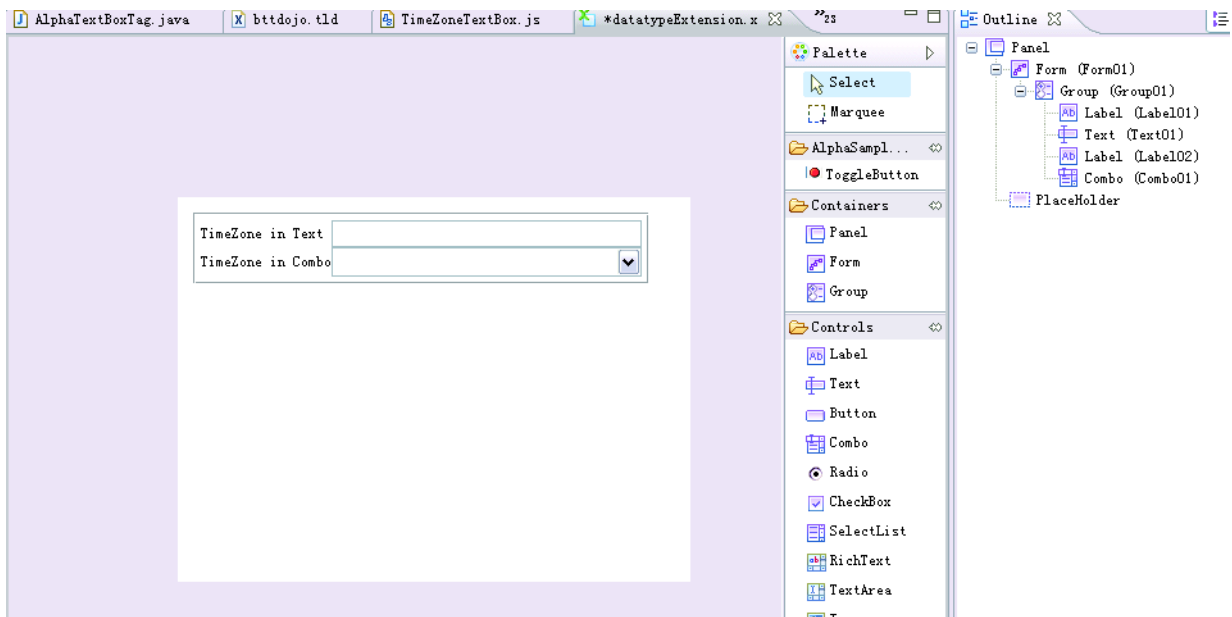
13 In popup dialog, select `datatypeExtensionOp.xml` > `dataTypeExtensionCtx` you created earlier.

14 Click **OK**.

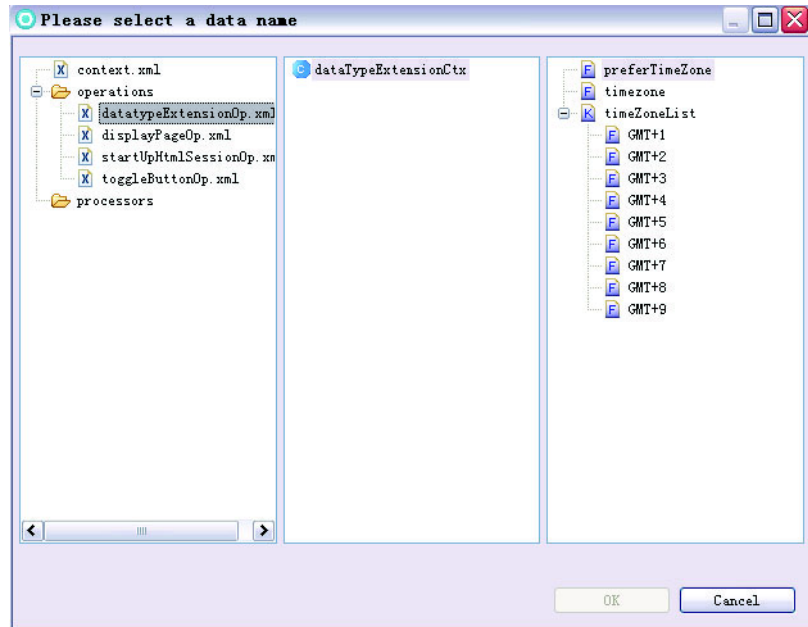


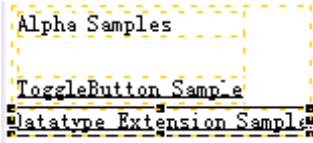
15 Create the UI.

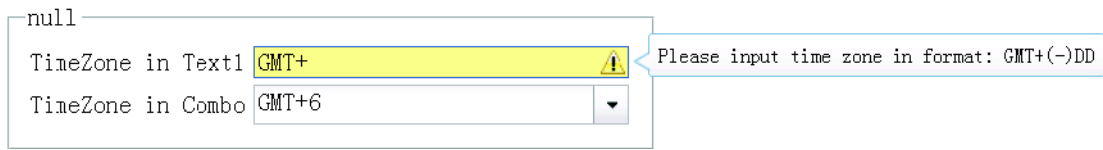
- a** In the XUI editor, compose the page as shown below.



- b** Click **Text Widget**.
- c** Open the **Properties** tab
- d** Edit the **dataName** property.
- e** Click **datatypeExtension > dataTypeExtensionCtx > preferTimeZone**.

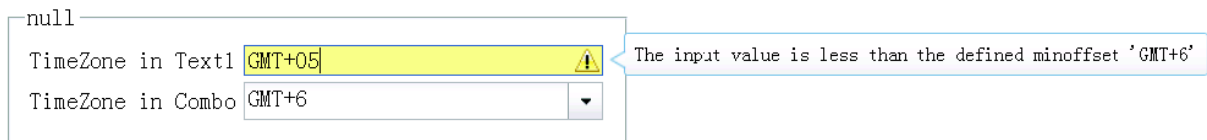


- f Click **Combo** widget.
 - g Open the **Properties** tab.
 - h Click **datatypeExtension** > **datatypeExtensionCtx** > **timezone** field for the **dataName** property.
 - i Click **datatypeExtension** > **datatypeExtensionCtx** > **timeZoneList** for the **dataNameForList** property.
 - j Click **Save**.
- 16** Right click **datatypeExentsion.xui** file then click **Generate dojo** to generate JSP.
- 17** Open **index.xui** file with XUI editor.
- 18** Add a **Link** widget,
- 
- 19** Open the **Properties** > **Action** tab.
 - 20** Click **Launch Operation** for **Action Type**.
 - 21** Click **datatypeExtensionOp** for **operationId**.
 - 22** Click **Save**.
 - 23** Right click **index.xui** file and click **Generate dojo** to generate JSP.
 - 24** Copy **TimeZoneTextBox.js** from **AlphaWidget** project to **BTTExtensionWeb** project.
 - 25** Deploy the test application to WAS and run.
 - 26** Open the **datatypeExtension.jsp** page (see below).



27 Do a check to make sure that the user input is correctly validated.

Note The text box validates the user input String in the format 'GMT+(-)DD'. Furthermore, even user input right TimeZone format, it will also check if the value is in the right range (from GMT+6 to GMT+8).

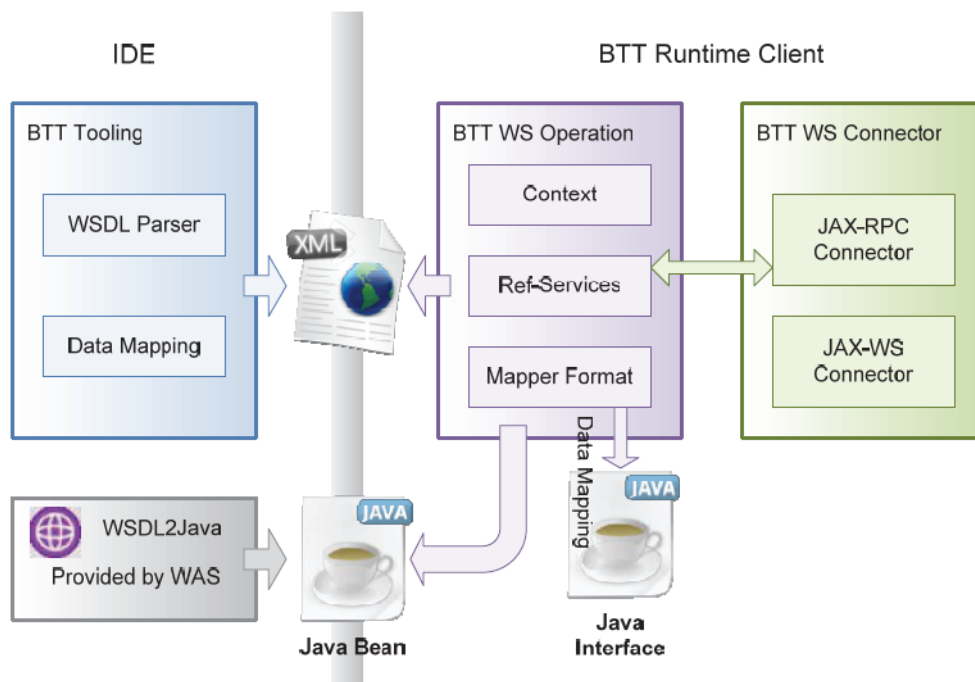




CHAPTER 5

Web Services Extension

For Web services invocation, BTT provides both tool and runtime support for technical Functional developers to use. The following diagram shows the high-level structure of BTT Web services component which could be separated into tool and runtime parts. There are also extension capabilities on each part for Infrastructure developers to customize project specific behavior during Web services integration.



Web services Tool Extension

The BTT Web services tool provides the one-stop facility to enable the Web services integration for a customer project. It consumes WSDL/XSD files as input, and then generates the three artifacts required for Web services invocation in BTT project:

- Web services connector
- Web services self-defined operation
- Web services client stub classes.

Referring to the usage guide of Web services tool, please find the related content from the *Multichannel Bank Transformation Toolkit Functional Developer User Guide*.

ID Mapping during self-defined operation generation

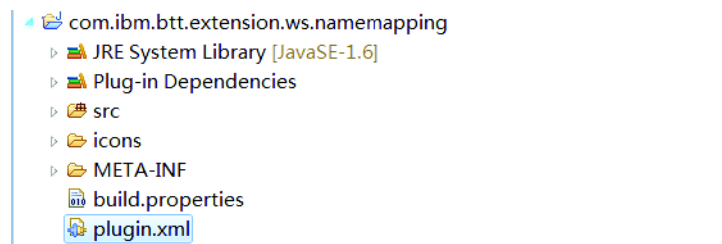
For Web services integration in a BTT project, technical Functional developers needs to produce a self-defined operation to invoke the related server operations. As a self-defined operation artifact, the associated data definition is derived from the input/output messages of the chosen Web services operations automatically.

All the data in the generated self-defined operation has local accessibility and follows the default naming convention. But sometimes for the purpose of high re-usage rate of global data definition in a BTT application, the automatic data generation logic could be in intervened by project extension. Infrastructure developers could extend the pre-defined extension point to indicate the data link reference between self-defined operation and global data dictionary.

Prepare tool extension environment

For tool extension, Infrastructure developers need to prepare the working environment by the following steps. For more details, please refer to the 'Plug-in Project Setup' on page 14.

- 1 Create an Eclipse Plug-in project
- 2 Add the required plug-ins as project dependency

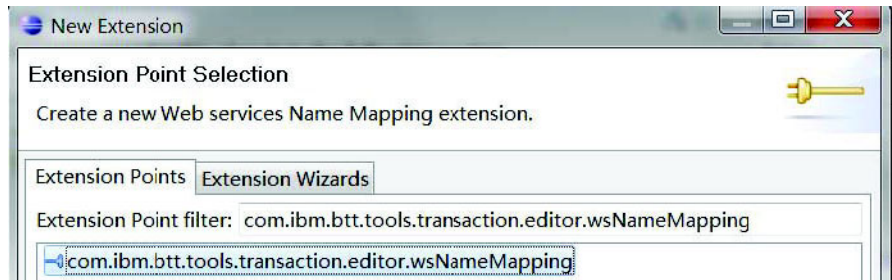


Add extension definition for target extension point

In this step, Infrastructure developers needs to provide specific extension definition to the extension point pre-defined in BTT project:

- 1 Open the file plugin.xml of the working project
- 2 Click the **Extension** tab.
- 3 Click **Add**.

- 4 Browse to the extension
com.ibm.btt.tools.transaction.editor.wsNameMapping.



- 5 Click **Finish**.

Configure extension details

In the details configuration panel of newly added extension definition, Infrastructure developers need to provide a implementation class of the interface `com.ibm.btt.tools.transaction.ws.generator.mapping.name.INameMapper`. The behavior of this class is to provide a Java map data to describe the relation between names of self-defined operation and global data dictionary.

Extension Element Details

Set the properties of "nameMapper". Required fields are denoted by "**".

class*

Implement name mapper interface

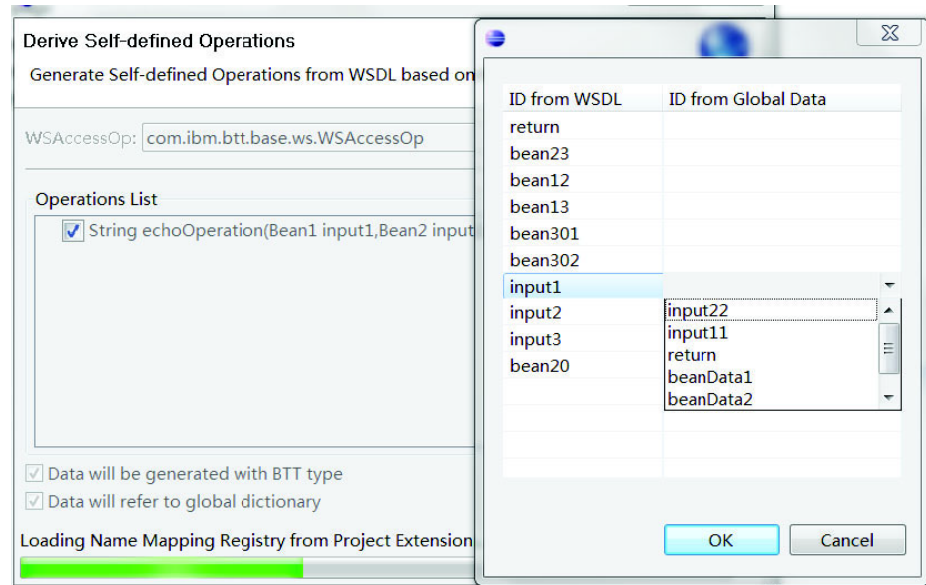
During the self-defined operation generation, the names defined in WSDL will be changed into valid Java names which will be used for the self-defined operation data by default. In the name mapper interface, there is an input parameter with the type of String set that contains all the changed Java names of selected operations. And then, it should give back a map data in which the keys are the Java names and the values are the mapped data names of global data dictionary. There is another parameter with the type of List that contains all the global data in the application.

```
public interface INameMapper {
    Map<String, String>
        map (Set<String> javaNames, List<DataElement> globalData);
}
```

The implementation class of the mapper interface should be configured in the extension. The Web services tool will then prepare the required parameters and launch it. After execution of the extension class, the returned map data will be used during the self-defined operation data generation.

Samples overview

In the sample, the extension plug-in will pop-up a dialog for the end user to manipulate the data mapping relationship between self-defined operation and global data dictionary. As the screen shot below shows, the left column lists the names from self-defined operation. The right column provides combo editors for each cell from which the global data could be selected.



For the details of the sample, please refer to the extension project `MappingName.zip`.

The screen below shows the related classes of the extension.

- `com.ibm.btt.extension.ws.namemapping.mapper.controller`
 - `NameMapperExt.java`
- `com.ibm.btt.extension.ws.namemapping.mapper.model`
 - `NameMappingModel.java`
- `com.ibm.btt.extension.ws.namemapping.mapper.ui`
 - `MapperDialog.java`
 - `NameMappingColumnLabelProvider.java`
 - `NameMappingContentProvider.java`
 - `NameMappingEditingSupport.java`
 - `NameMappingLabelProvider.java`

- `NameMapperExt`. This is the implementation class of the interface registered for the extension point. It performs the model creation and launches the dialog for data manipulation.

```
final Display display = Display.getDefault();
display.syncExec(new Runnable() {
    @Override
    public void run() {
        Shell shell = new Shell(display.getActiveShell());

        Dialog dialog = new MapperDialog(shell, model);
        dialog.open();
    }
});
```

- `NameMappingModel`. This is the data model of the extension application, which contains the global data and the names from Web services operation. The manipulation result is also stored in this model class.
- `MapperDialog`. This is the main UI part of the extension application which contains a table to show names and accept manipulations. This dialog box is a popup during the extension execution.

- `NameMappingContentProvider` and `NameMappingLabelProvider`. These two classes are used for the table shown in dialog box to interact with the data model. The main purpose for them is to show the data model in the table.

```
tableViewer.setContentProvider(new NameMappingContentProvider());  
tableViewer.setLabelProvider(new NameMappingLabelProvider());
```

```
tableViewer.setInput(model.getNameMappings());
```

```
valueColumnViewer.setEditingSupport(new NameMappingEditingSupport(tableViewer, model.getGlobalData()));  
valueColumnViewer.setLabelProvider(new NameMappingColumnLabelProvider());
```

- `NameMappingColumnLabelProvider` and `NameMappingEditingSupport`. These two classes are used for the right column to provide the combo cell editor and related editing support.

Web services Runtime Extension

BTT Web services runtime follows the principles of the BTT programming model, which makes use of BTT core concepts (operation, format, service) to resolve the Web services invocation. The learning curve is, therefore, shorter for Infrastructure developers.

Web services Runtime Overview

At runtime part, BTT invokes Web services by means of the work of these three primary runtime components. For project-specific purpose, Infrastructure developers could extend their own runtime components used for Web services runtime.

Web services Connector

Web services connector communicates with the Web services provider. It covers the implementation details such as creating a service delegate object, reflecting the requested operation, configuring the invocation properties. In the BTT product, there are two different sets of code to support the JAX-RPC and the JAX-WS specifications. The latter is recommended by default because it is compatible with the former one.

The Web services connector has implemented a BTT service interface. Developers could use it like a standard BTT service object. The Web services connector instance could be configured by XML and instantiated for use standalone by code like `readObject(String serviceName)`. The Web services connector could be created and edited by the BTT Transaction Editor.

Web services Access Operation

In a BTT application, developers should use Web services Access Operation to invoke a Web services. Web services Access Operation extends the standard BTT Operation. In its execution logic, it communicates to Web services with the help of Web services connector and then processes the input and output message data by Web services mapper.

Web services Mapper

The data transferred between Java Web services and client is quite different from the BTT Context and Data. To bridge the gap between the two different data systems, Web services mapper is used closely with Web services Access Operation to change between BTT XML based data and Web services Java Beans.

Extend WS Handler and WS Connector

Developers could use handlers to inject additional logic into the message flow during Web services invocation, for a variety of purposes such as capturing and logging information and adding security or other information to a message.

The handlers registered to application level are supported by both JAX-RPC and JAX-WS. But for JAX-WS, handlers could also be plugged into runtime by annotation or code. The handlers registered into runtime will be used by a handler resolver that is provided by BTT Web services runtime.

The extension for WS handlers is always implemented together with WS connector extension. So this chapter uses a sample to cover the extension guide for both of these two components.

Samples Overview

In the sample, there is guide about how to print the SOAP messages transferred during the invocation. Actually, Infrastructure developers could follow similar steps to manipulate the inbound and outbound messages, such as adding SOAP header or encrypting messages.



CHAPTER 6

Channel Policy Management and Extension

Channel policy management provides bank customers with unified business data integration across different channels, along with a unified user experience cross channel. With channel policy management, developers can rapidly develop multi-channel business integration solution, and rapidly change and deploy business rules in production. Furthermore, the channel policy implementation is loosely coupled with specific transaction logic.

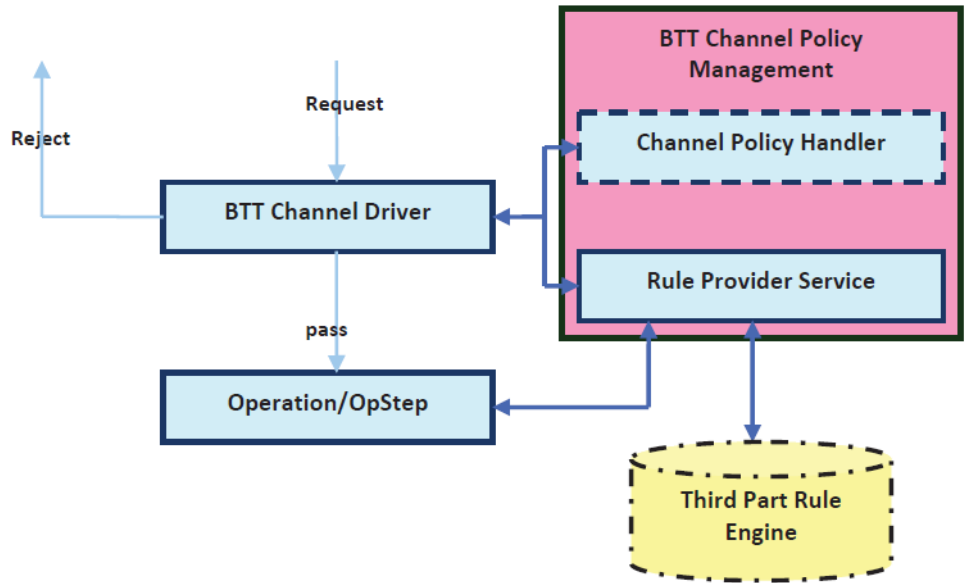
BTT provides the embedded channel policy management mechanism in the BTT Channel layer. The channel policy can be used to handle in two levels:

- **Channel level:** In this level, all requests can be validated by channel policy. For example, to check if a user has the privilege to use Internet banking channel or if the user can use a specific operation in the Internet banking channel.
- **Operation level:** In this level, when a specific transaction or operation is invoked by a user, channel policy can be used to check the authorization limits to this user. For example, when a user is using the Internet banking channel to transfer money in account, the channel policy can be used to check the maximum amount money the user can transfer on the Internet banking channel.

There are two primary components in BTT channel policy management framework:

- **Channel Policy Handler:** is used to extract policy input parameters from the channel context and parse the policy check result from the rule provider service. As parameters and policy result are project specific, there is no default implementation. Infrastructure developers need to implement their own handler for a specific project.
- **Rule Provider Service:** provides a common interface to access the rule engine. Two pre-built rule providers are implemented as BTT services: ILog Connector Service and Java Code Rule Provider Service. Infrastructure developers can extend rule provider service to support any other third part rule engine.

The figure below shows the logic view of the BTT channel policy component.



Channel level policy management

Implement channel policy handler

BTT provides an abstract channel policy handler `com.ibm.btt.channel.AbstractChannelPolicy` to handle policy in the channel level. The `doCheck` method in `AbstractChannelPolicy` provides a default implementation to use with the defined rule service to do channel level policy management. Infrastructure developers can override the method as necessary.

Infrastructure developers should override the following two methods in their handler implementation:

```
protected Map<String, Object> getInputParameter(ChannelContext ctx)
```

The method is used to construct the input parameters for the `doCheck` method using the data from channel context.

```
protected PolicyResult processResult(Map<String, Object> result)
```

The method is used to parse the result returned from the rule provider service.

The `com.ibm.btt.channel.PolicyResult` is the return type of `processResult` method. It includes the execution result of policy check. If the check passes, the BTT channel provider will continue the request processing. Otherwise, the BTT channel driver will throw the `com.ibm.btt.channel.ChannelPolicyException`. The messages in `PolicyResult` are used to keep the messages generated during rule check.

The following is sample policy handler implementation:

```
public class UserSecurityChannelPolicyHandler extends AbstractChannelPolicy {
    @Override
    protected Map<String, Object> getInputParameter(ChannelContext channelCtx) {
        Map input = new HashMap();
        String operationName=null;
        String userID="user01";
        try {
            operationName=(String) channelCtx.getRequestData().getValueAt("data.dse_operationName");
        } catch (Exception e) {
        }
    }

    ChannelPolicyInputData data=new ChannelPolicyInputData();
    data.setOperationName(operationName);
    data.setUserID(userID);
    PolicyResult result=new PolicyResult();
    input.put("result", result);
    input.put("channelData", data);
    return input ;
}

@Override
protected PolicyResult processResult(Map<String, Object> arg0) {
    return (PolicyResult) arg0.get("result");
}
}
```

Define rule provider service

The rule provider is implemented as a BTT service. As with other BTT services, it needs to be defined in service.xml file. BTT provides two pre-built rule provider services:

- ILOG connector service

IBM ILOG can be used as rule engine to store channel management policy. The service is used to connect ILOG to the access channel policy defined in it. The BTT ILOG connector service supports the following attributes:

Name	Description
id	The unique id of this service
ruleID	The rule id defined in ILOG accessed by service.
mode	The mode of accessing ILOG. There are two possible values: J2EE, J2SE and WebService. The default value is J2EE
WSClientBeanName	The name of the class to access ILOG when mode is WebService. The class is generated by WSClientBeanName.

The following is a sample service definition in service.xml:

```
<com.ibm.btt.channel.ruleprovider.ilog.ILogRuleProviderService
  id="checkTransferAmountILogRule"
  ruleID="/checkChannelRuleApp/BTTChannelRules" />
```

- Java Code Rule Provider Service

BTT provides

`com.ibm.btt.channel.ruleprovider.java.JavaCodeRuleProviderService` as base the class for Infrastructure developers to implement a rule provider service in Java Code. Infrastructure developers need to extend the `JavaCodeRuleProviderService` and override `checkRule` method.

```
public Map<String, Object> checkRule(Map<String, Object> params)
```

The method is used to do a check of business policy rules and decides if the request can be accepted.

The following is a sample implementation of `JavaCodeRuleProviderService`. In it, any transfer amount more than 10000 will be rejected.

```
public class TransferLimitRuleService extends JavaCodeRuleProviderService {
    public Map<String, Object> checkRule(Map<String, Object> arg0) {
        Double amount= (Double) arg0.get("transfer_amount");
        if (amount.doubleValue() > 10000.0)
            arg0.put("accept", Boolean.FALSE);
        else
            arg0.put("accept", Boolean.TRUE);
        return arg0;
    }
}
```


Furthermore, Infrastructure developers can implement their own policy provider service such as supporting other third party rule engines. To implement a policy provider service, Infrastructure developers need to extend the abstract class `com.ibm.btt.base.Service` and implement the `com.ibm.btt.channel.ruleprovider.IBTTRuleProvider` interface.

Configure policy for channels

After implementing the channel policy handler and defining rule provider service, Infrastructure developers need to configure the handler and service in `btt.xml` for the specific channel. The following is a sample configuration for an html channel:

```
<kColl id="html">
  <field id="encoding" value="UTF-8" />
  <field id="cookies" value="true" />
  <field id="runInSession" value="true" />
  <field id="requestHandler"
    value="com.ibm.btt.cs.html.AjaxHtmlRequestHandler" />
  <field id="presentationHandler"
    value="com.ibm.btt.cs.html.AjaxHtmlPresentationHandler" />
  <field id="channelPolicyHandler"
    value="com.ibm.btt.sample.channelpolicy.UserSecurityChannelPolicyHandler"/>
  <field id="ruleService" value="UserSecurityProfileRuleService" />
</kColl>
```

Exception handling

When the channel policy check rejects (`PolicyResult.accept==false`) the request, the BTT Channel driver throws `com.ibm.btt.channel.ChannelPolicyException`. By default, the BTT presentation handler returns the exception to the client end. But the application may want to return a more user friendly error message to the client end. In this case, Infrastructure developers need to extend the channel presentation handler to handle the exception.

For example, in the case of the html channel, the default presentation handler is

```
com.ibm.btt.cs.html.AjaxHtmlPresentationHandler
```

or

```
com.ibm.btt.cs.html.HtmlPresentationHandler.
```

To handle exception with application needs, Infrastructure developers need to extend either of the presentation handlers above and override `handleException` method.

```
public void handleException(ChannelContext channelContext,
                           Exception e)
```

In the method, the presentation handler handles any exception thrown while processing the request and navigates the user to the correct error page with the applicable message. The following is an example of `handleException` method implementation that demonstrates how to handle a `ChannelPolicyException`.

```

public class MyHtmlPresentationHandler extends HtmlPresentationHandler {

    @Override
    public void handleException(ChannelContext channelContext, Exception e) {
        if (e instanceof ChannelPolicyException) {
            // See if we have a response to use.
            HttpServletResponse res = getOrgResponse(channelContext);
            if (res != null) {
                try {
                    PrintWriter htmlOut = res.getWriter();
                    htmlOut.println("Policy Check result:");
                    htmlOut.println(e.getMessage());
                } catch (IOException exc) {
                    // do nothing
                }
            }
        } else {
            super.handleException(channelContext, e);
        }
    }
}

```

To apply the new presentation handler, Infrastructure developers need to configure this presentation handler in `btt.xml` for the channel.

Operation level policy management

Implement OpStep for operation level policy

To support operation level policy management, BTT provides `com.ibm.btt.channel.AbstractPolicyOperationStep` as a base class for Infrastructure developers to extend. The `execute` method in `AbstractPolicyOperationStep` provides a default implementation which uses the rule service to check the operation level policy. Infrastructure developers can extend it based on the application needs. Meanwhile, Infrastructure developers should implement the following two abstract methods:

```
protected abstract Map<String, Object> getInputParameter()
```

The method is used to construct the input parameters to execute the method using the data from the operation context.

```
protected abstract int processResult(Map<String, Object> result)
```

The method is used to parse the result returned from the rule provider service. The return value of the `processResult` method is used to control the state transition between `opSteps`.

The following is a sample `OpStep` implementation class that demonstrates how to implement operation level policy.

```
public class CheckLimitOpStep extends AbstractPolicyOperationStep {

    @Override
    protected Map<String, Object> getInputParameter() {

        Map input = new HashMap();
        try {
            Currency amount=(Currency) this.getContext().getValueAt("transfer.amount");
            System.out.println("##### :"+amount.getClass());

            input.put("transfer_amount", amount.getValue().doubleValue());
            input.put("CurrencyType", amount.getCurrencyType());
        } catch (DSEObjectNotFoundException e) {

        }

        return input;
    }

    @Override
    protected int processResult(Map<String, Object> arg0) {
        Boolean accept= (Boolean) arg0.get("accept");

        try {
            getContext().setValueAt("transfer.transferLimit&accept",accept);
        } catch (Exception e) {

            e.printStackTrace();
        }

        return RC_OK;
    }
}
```

Configure operation

After extending the `AbstractPolicyOperationStep`, Infrastructure developers need to configure the related operation to use the `opStep`. In the operation definition, Infrastructure developers need to configure an `opStep`. The `implClass` should be the class that extends from `AbstractPolicyOperationStep`. The `refRuleService` should be the rule provider service defined in `service.xml`. The service will be used by `opStep` as the rule provider service. The following demonstrates how to configure an operation to use `opStep` for operation level policy management.

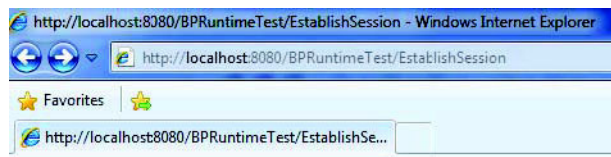
```
<operation id="checkLimitOp" implClass=
    "com.ibm.btt.sample.transfer.operation.CheckTransferLimitOperation">
    <opStep id="initTransferOpStep" refRuleService="checkLimitRuleService"
        implClass="com.ibm.btt.sample.transfer.operation.CheckLimitOpStep"
        onOdo="return"/>
</operation>
```

Channel policy sample

The BPRuntimeTest.war file is provided as a channel policy sample. The sample implements the real case that demonstrates how to use BTT Channel Policy. The guide shows the code and configuration related to the channel policy and the extension point.

How to run the sample

- 1 Deploy the WAR file to Tomcat or WAS.
- 2 Type the URL **http://localhost:8080/BPRuntimeTest/EstablishSession** to start the sample.

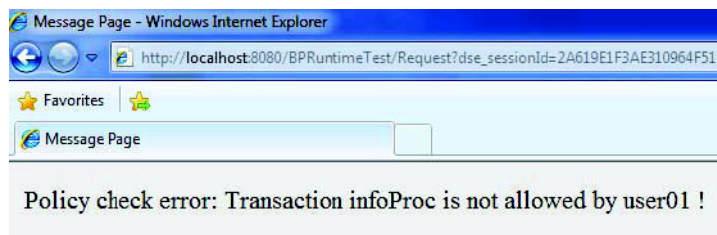


Sample:

[Start Information Submission](#)

[Start Account Transfer](#)

- 3 If you click **Start Information Submission**, the channel policy will not let you run this transaction. The page below is displayed.



- 4 If you click Start Account Transfer, the channel policy lets you run this transaction. The sample implements a simple rule to the check transaction limit. If the amount is over 10000, it displays an error page.

Transfer Page 1
 Please input account transfer info:

From Account Please select an account 0.00

Cross Region **Country** Spain **City**

Cross Bank **Bank** Moon Bank

Name	Account	Bank	Comments
Messi	22223333444455	Bank of Mars	Wage Account
C Ronaldo	89898989121212	Moon Bank	Bonus Account

Name C Ronaldo Hidden Account Tab

To Account 89898989121212

To Account Confirm 89898989121212


Amount \$29,998.50

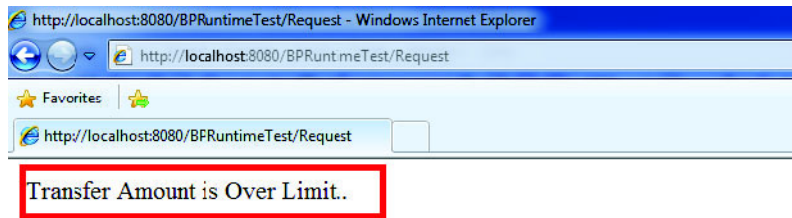
Comments

Save to Account List

SMS Notification

Password ●●●●●●

Verification code 0000 

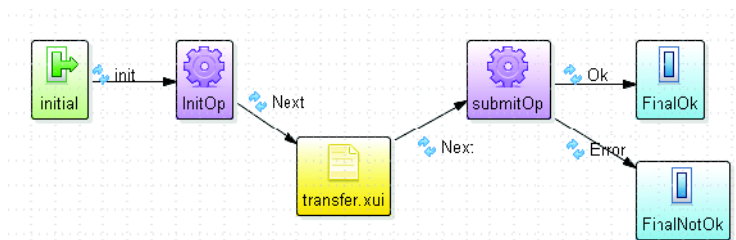




CHAPTER 7

Process Editor Extension

BTT provides a Processor Editor for developers to visually construct a BTT flow. The Processor Editor is shown in the Processor tab of BTT Transaction Editor. Developers can compose a BTT flow by dragging and dropping different kinds of states from the palette and connecting them with transitions. The following figure shows a flow composed with the Processor Editor.



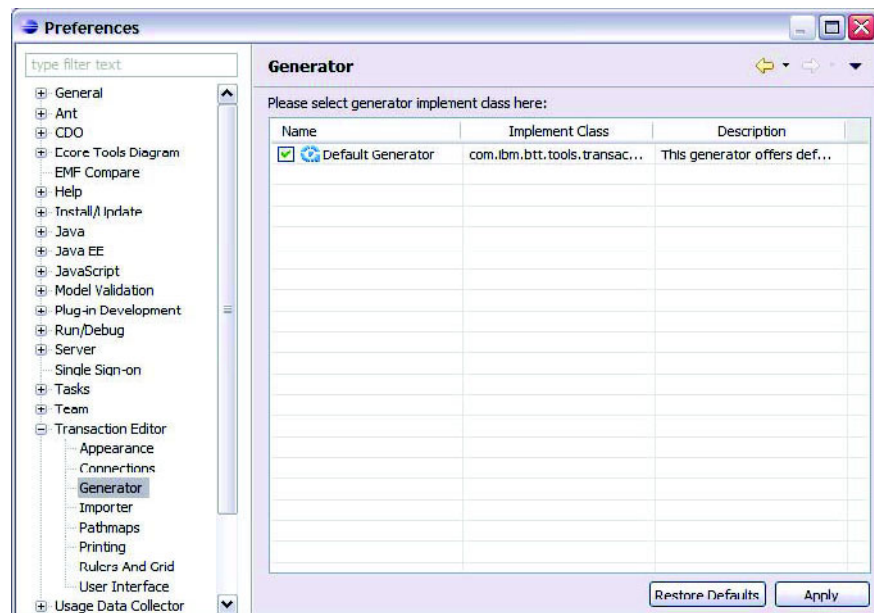
Furthermore, the BTT Processor Editor supports two kinds of editing modes. According to the project requirement, Infrastructure developers could customize the default mode of BTT Processor Editor.

- **Compatibility Mode:** In this mode, developers can operate on all the possible properties of a state or a transition.
- **Default Mode:** In this mode, the property editor of a state or transition is simplified. A developer with less BTT knowledge could also use the Processor Editor to compose the BTT flow.

BTT Transaction Editor stores the BTT flow in a file with the extension of transaction, which describes a generic, channel-independent flow. To run this flow in the runtime environment, the transaction file needs to be generated into a BTT XML file (Select the option **Transaction Editor** from the context menu of transaction file, then choose the option **Generate BTT Transaction XML**).

This transaction file is a channel-neutral flow definition, which will be generated based on different channel transaction generators. Actually, each BTT channel has its own generator which is defined via the extension point `com.ibm.btt.tools.transaction.editor.generator`. For example, a BTT

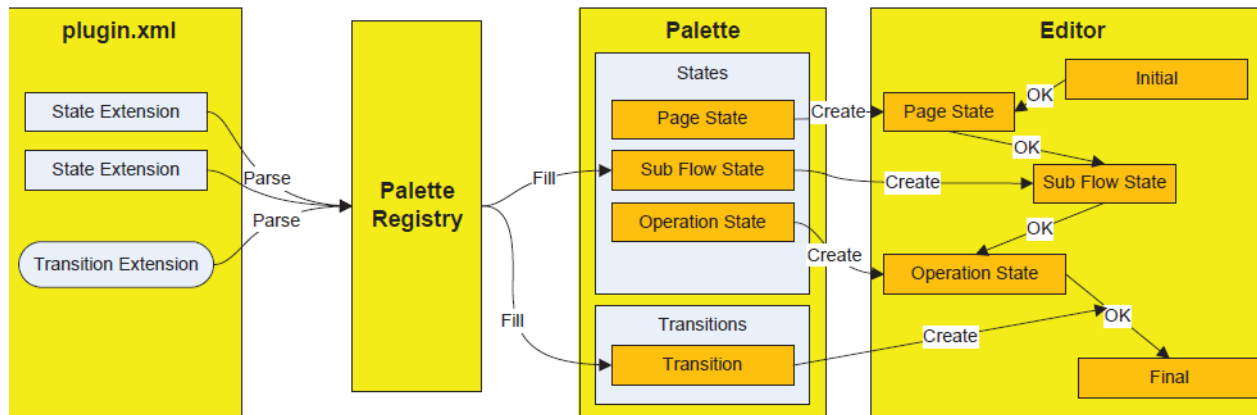
HTML channel has its own generator named `com.ibm.btt.tools.transaction.generator.xml.GeneratorFactory`. In case several generators are defined, developers could choose the preferred one by choosing **Window > Preferences > Transaction Editor > Generator** option.



A generator can work with one or more mapping definition files, which contain the tag mappings from PageState to htmlState, property mappings from page to typeIdInfo and the logic to be injected into the generation process. In case there are several mapping definition files, Infrastructure developers should make sure they have not defined overlapped tags set. Otherwise, there is no guarantee about which mapping rules will be taken into account.

Extend processor editor object

To create a new Processor Editor object, the Infrastructure developers need to define it in an xml definition file and then register this definition into the Palette Registry as a plug-in extension. After that, the object will be shown in the palette of the Processor Editor, and can be dragged and dropped into the canvas of Processor Editor. In the following sections, we will describe how to create a new processor editor object in more details.



Create configuration file for palette object

BTT defines a palette object by an xml definition file which describes how this object will be shown in the palette and the canvas of the BTT Processor Editor, which properties of the object can be edited, which property editor will be used for the properties. The following sections will describe the tags of an object configuration file in more details.

There are two types of palette objects: state object and transition object. For a state object, there are three kinds of tags used in its configuration file:

Appearance

This tag defines how to display the extended state object in the palette. The table below shows the available attributes for an Appearance tag.

Attribute Table	Description
gradient	true or false
fontColor	This attribute is used to defined the font color in RGB, such as: fontColor="0,0,0"
backgroundColor	This attribute is used to defined the background color in RGB, such as: backgroundColor="255,216,1"
font	This attribute is used to indicate the font style of the state shown in palette, such as: font="Arial-regular-10"

Properties tag of state

This tag lists all properties of a state object. It needs to contain one or more Property tags.

Property tag of state

The tag describes how the Processor Editor is to display and edit a property. The table below shows the attributes for a Property tag.

Attribute	Description
name	The property name identifier.
defaultValue	It corresponds to the property value by default.
hidden	It specifies if the property must or not be displayed. Possible values are: true or false. If it is true, the attributes described next don't apply. The default value is false.
editRule	It is the property editor that will be used by the user to enter the property value. It should be the same as any rule id in table 7-3. If it is not specified, a default editor is assigned
description	It contains the text to be used as tooltip.
required	It indicates if the property is mandatory (user must enter a value) or not. Possible values are: true or false. Default value is false.

Below are predefined property edit rules:

Rule ID	Description
Boolean	For boolean chosen rule
XValidate	For XValidate property editor rule
Context	For Context chosen rule
ConditionAdjust	For Condition adjust rule
OpStepAction	For OpStep action chosen rule
OpStepCondition	For OpStep condition rule
Operation	For Operation chosen rule
EventId	For Event Id chosen rule
PageSelection	For Page chosen rule

Below is an example of state object configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<state xmlns="http://btt.ibm.com/StateSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/StateSchema StateSchema.xsd ">

  <appearance backgroundColor="254,46,154" font="Arial-italic-20"
    fontColor="0,0,0" gradient="true"/>

</properties>
```

```

    <property name="Page" displayName="Page Name"
      defaultValue="" hidden="false"
      required="true" description="Page file path"
      editRule="PageSelection" />
    <property name="Back Allowed" defaultValue="False" hidden="false"
      required="false"
      description="Specifies the possibility to use the back button from
the navigator" editRule="Boolean" />
    <property name="id" defaultValue="" hidden="true" required="false"
      description="id" editRule="" />
  </properties>
</state>

```

For a transition object, there are three tags in its configuration file:

Appearance tag of transition

It defines how to display a transition object in palette and canvas. The table below shows the attributes for an Appearance tag.

Attribute	Description
lineColor	Color in RGB. For example, 255,255,255. Default value is 0,0,0.
lineWidth	Line width in pixel. Default value is 1.
lineStyle	Possible values are: Solid, Dash, DashDotDot, DashDot, Dot, Double. Default value is Solid.
arrowTypeStyle	Possible values are: None, OpenArrow and SolidArrow. Default value is SolidArrow.
font	The font.
fontColor	Color in RGB. For example, 255,255,255.

Transition object configuration file should include `Properties` and `Property` tags which are the same as state object described previously.

Below is an example of transition configuration:

```

<transition xmlns="http://btt.ibm.com/TransitionSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/TransitionSchema TransitionSchema.xsd ">

  <appearance lineColor="0,0,0" lineWidth="1" lineStyle="Solid"
    arrowTypeStyle="SolidArrow" font="Arial-regular-10"
    fontColor="0,0,0"/>

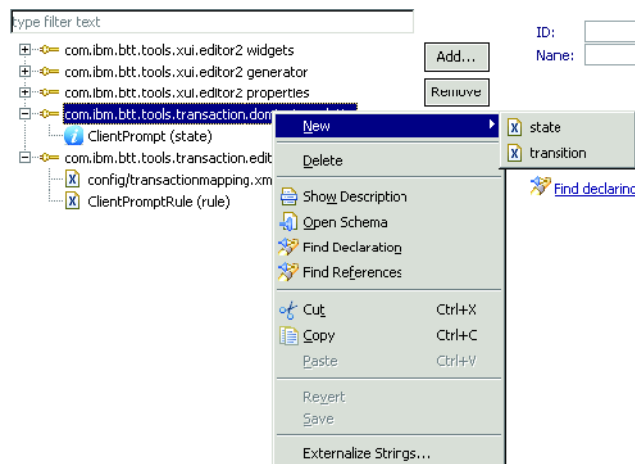
  <properties>
    <property name="Event" hidden="false" required="true"
      defaultValue="" editRule="EventTransitionBeta"/>
    <property name="Input Data Format" required="false"
      description="%InputFormat" editRule="MapperFormat"/>
    <property name="Output Data Format" required="false"
      description="%OutputFormat" editRule="MapperFormat"/>
    <property name="Skip Validation" defaultValue="false"
      description="%TransValidated" editRule="Boolean"/>
    <property name="id" defaultValue="" hidden="true" required="false" />
  </properties>
</transition>

```

Register palette object

To register customized palette object to the Processor Editor, Infrastructure developers need to add an extension point for this object in the plugin.xml file of his extension plug-in. To register the object, do the steps that follow.

- 1 Open plugin.xml file.
- 2 Click the **Extensions** tab.
- 3 Click **Add**.
- 4 In **Extension Point Filter** field, type **com.ibm.btt**.
- 5 Click **com.ibm.btt.tools.transaction.dominated.palette**.
- 6 Click **Finish**.
- 7 Right click **com.ibm.btt.tools.transaction.dominated.palette** then click **New > state** or **New > transition** to create the applicable object.



- 8 In **Extension Element Details** dialog box, type the applicable information.
 - **name** field requires inputting the name of the object. It serves as ID, so should be unique.
 - **label** field requires inputting the display name of the object, the label will be shown in palette as object name. It supports NLS.
 - **smallIcon** field requires selecting an image to display the object in palette of Processor Editor. Image in 16x16 pixels is recommended as it is consistent with existing BTT objects.
 - **largeIcon** field requires selecting an image to display the object in the canvas of the Processor Editor. Image in 32x32 pixels is recommended as it is consistent with existing BTT objects.
 - **config** field requires inputting file name of object configuration xml file described in 'Create configuration file for palette object' on page 81.
 - **description** field describes the function of the state, which will be shown in palette when cursor moves over the icon in palette.
 - **stateParser** field indicates the class that is used to parse events of the state automatically.

Below is an example:

Extension Element Details

Set the properties of "state". Required fields are denoted by "*".

name*:

label*:

smallIcon*:

largeIcon*:

config*:

description:

stateProvider:

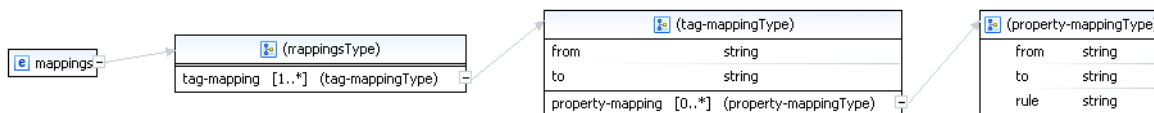
Create and register object mapping

After Functional developers complete composing the XUI file, they select **Transaction Editor->Generate BTT Transaction XML** function and the BTT tool will automatically generate BTT transaction xml file for this transaction file. In order to generate proper xml tag for the customized processor object, Infrastructure developers need to create a new object mapping file and register it as plug-in extension.

Create object mapping file

In a processor object mapping file, it should include one mappings tag. The mappings tag should contain one or more tag-mapping elements. Each tag-mapping tag can contain property-mapping element.

The figure below demonstrates the relationship:



Attributes of tag-mapping tag:

Attribute	Description
-----------	-------------

from	The tag name in .transaction file.
to	The tag name in .xml file

Attributes of property-mapping tag:

Attribute	Description
-----------	-------------

from	The tag name in .transaction file. Required
to	The tag name in .xml file. Required
Rule	The id of property conversion rule to convert the tag. Predefined rule is described in the table HTMLFinalIdRule.

Predefine rules: HTMLFinalIdRule

Rule ID	Description
HTMLFinalIdRule	The rule of converting final state id from neutral flow file to html channel flow file
HTMLOperationIdRule	The rule of converting operation state id from neutral flow file to html channel flow file
HTMLPageIdRule	The rule of converting page state id from neutral flow file to html channel flow file
HTMLSubflowIdRule	The rule of converting subflow state id from neutral flow file to html channel flow file
HTMLProcessorIdRule	The rule of converting processor id from neutral flow file to html channel flow file
TransitionIdRule	The rule of converting transition id from neutral flow file to html channel flow file
TransitionTargetRule	The rule of converting transition target from neutral flow file to html channel flow file

Below is a sample of object mapping file:

```
<mappings xmlns="http://btt.ibm.com/MappingsSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/MappingsSchema MappingsSchema.xsd ">

  <tag-mapping from="ClientPromptState" to="htmlPromptState">
    <property-mapping from="id" to="id" rule="ClientPromptRule" />
  </tag-mapping>
</mappings>
```

Register object mapping

- 1 Click the Extensions tab of plug-in file.
- 2 Click **Add**.
The **New Extension** dialog appears.
- 3 Click **com.ibm.btt.tools.transaction.editor.generator**.
- 4 Click **Finish**.
- 5 Right click **com.ibm.btt.tools.transaction.editor.generator** then click **New > generator**.
- 6 In the **Extension Element Details** dialog box, type the applicable information.
 - **file:** put the mapping file defined previously.
 - **target:** type **Default Generator**.

Create and register property generation rule

When generating a flow xml file, the default rule is to use to String to replace from String simply. But there are more complex conversion rules when generate flow xml file.

BTT implements several pre-defined rules to convert existing processor objects. When Infrastructure developers create new object, it is very possible for them to create new conversion rule to generate xml lines for the object. To create a new generation rule, Infrastructure developers need to create a Java class to implement the rule, and then register the rule as plug-in extension.

To create a generation rule class, Infrastructure developers need to extend the BTT abstract class `com.ibm.btt.tools.transaction.extend.generator.Rule` and override the `process` method. In the `process` method, Infrastructure developers can manipulate the target tag object as their needs such as adding new attribute and changing attribute an name or value.

```
public void process(Taggable object, Map<String, String> attributes,
                  PropMapping mapping, String value)
```

After creating a new rule class, Infrastructure developers need to register this rule in plug-in.

- 1 Click the **Extensions** tab of plug-in file
- 2 Right click **com.ibm.btt.tools.transaction.editor.generator** then click **New > rule**.
- 3 In the **Extension Element Details** dialog box, type the applicable information
 - **name:** type the name of rule as unique identifier
 - **class:** put the implementing class.

Extension Element Details

Set the properties of "rule". Required fields are denoted by "*".

name*:	<input type="text" value="ClientPromptRule"/>
class*:	<input type="text" value="com.ibm.btt.alphasample.transaction.generator.rule.ClientPro"/> <input type="button" value="Browse..."/>
description:	<input type="text"/>

Implement state in runtime

During the flow execution process at runtime, each state composed in this flow will be represented as a Java object. When the flow enters a state, the Java object corresponding to this state will be initiated and activated by processor.

After creating the palette object and its mapping rules for the Processor Editor, Infrastructure developers need to implement a Java class corresponding to this object. The class should extend class `DSEState` or its derived classes such as `DSEHtmlState`, and override the two methods that follow to implement state specific logic.

```
public Object initializeFrom(Tag aTag)
    throws java.io.IOException, com.ibm.btt.base.DSEException
```

This method is invoked when the processor initializes the state from tags. As the state may have several properties, Infrastructure developers need to initiate these properties from tag attributes when the state is initiated. The following code snippet is the method implementation sample:

```

public Object initializeFrom(Tag aTag) throws java.io.IOException,
    com.ibm.ktt.base.DSEException {
    super.initializeFrom(aTag);

    String name = null, value = null;
    TagAttribute attribute = null;

    for (java.util.Enumeration e = aTag.getAttrList().elements(); e.hasMoreElements();) {
        attribute = (TagAttribute) e.nextElement();

        name = attribute.getName();
        value = (String) attribute.getValue();
        if (name.equals(HTMLPromptState.MESSAGE)) {
            setPromptMessage(value);
        }
    }
    return this;
}

```

```
public void activate() throws DSEInvalidArgumentException, DSEProcessorException
```

This method is invoked when the processor activates this state. Infrastructure developers need to override this method to implement state specific logic. The following code snippet is the method implementation sample:

```

public void activate() throws DSEInvalidArgumentException, DSEProcessorException {
    try {
        if (getPromptMessage() != null) {
            this.getProcessor().getContext().setValueAt(HTMLPromptState.MESSAGE_FIELD, getPromptMessage());
        }
    } catch (DSEObjectNotFoundException e) {
        if (.Log.doError()) {
            String error = "HTMLPromptState: element '" + HTMLPromptState.MESSAGE_FIELD + "' is not defined in Context";
            log.error(error, e);
        }
        e.printStackTrace();
    } catch (DSEInvalidRequestException e) {
        if (.Log.doError()) {
            String error = "HTMLPromptState: error in activate()";
            log.error(error, e);
        }
    }
    super.activate();
}

```

Create and register global function

Global function is used in the Condition state of the Processor Editor to define expression for different conditions. BTT implements several functions by default. Infrastructure developers may have a requirement to extend BTT pre-defined functions to implement project specific functions. An plug-in extension point is provided for Infrastructure developers to implement project specific functions.

Infrastructure developers need first create a function definition file to declare the functions that will be implemented. The following is an example of function definition file.

```

<functions>
  <function name="concat" returnType="String" description="" >
    <parameters>
      <parameter name="string1" description="%concatString1" type="String" />
      <parameter name="string2" description="%concatString2" type="String" />
    </parameters>
  </function>
  <function name="length" returnType="Integer" description="" >
    <parameters>
      <parameter name="string1" description="%descString1" type="String" />
    </parameters>
  </function>
</functions>

```


Infrastructure developers then need to create a Java class to implement the declared functions. The class contains the set of implemented static methods corresponding to the declared global functions.

Lastly, the Infrastructure developers need to register the definition file and Java class as an plug-in extension point called `com.ibm.btt.tools.transaction.dominated.palette.globalFunctions`. In the Extension Element Details dialog, type the following information:

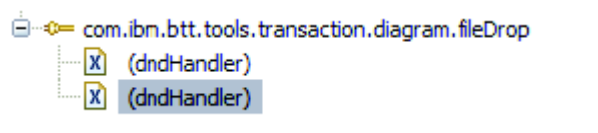
class: This is the global functions declaration class.

config: This is the global functions definition file.

Create editor object by dragging items

The Processor editor supports a drag-and-drop operation to the XUI page of flow and operation and it will create Page state for XUI page dropping, SubFlow state for flow and Operation state for operation. Meanwhile, you can provide your own drag-and-drop code to extend the Processor editor to accept another type drop as well override the default drag and drop support on page, flow and operation drop.

The extension point is `com.ibm.btt.tools.transaction.diagram.fileDrop`.



Attributes for the drag-and-drop handler are below:

class: The implementations of the interface

`com.ibm.btt.tools.transaction.diagram.file.drop.IDiagramEditorDropFileHandler`. It defines two methods: `accepts` and `parseDroppedFile` - the former one is used to check whether the handler can accept the dropped item and the latter defines the detail action on drop. Recommended to inherit from `com.ibm.btt.tools.transaction.diagram.file.drop.DefaultDiagramEditorDropFileHandler`.

priority: The priority of the handler. On dropping an item, the Processor editor will find the handler with highest priority to treat the current drop action. The valid values are: `low` | `medium` | `high`.

If you want to override the default actions on page/flow/operation drop action, add a drag-and-drop handler with medium or high priority since the priority of the default handler is low.

Samples:

`FlowEditorJSFileDragAndDropHandler` and `FlowEditorXUIFileDragAndDropHandler` are two customized drag-and-drop handlers registered to the Processor editor which lets the javascript file(.js) create a new customized JSSate state in the processor and override the default XUI drop by adding a comment to the page state created.

The implementation stuffs of this sample are in `Drop4Extension` project. The table below has a list of stuffs for this sample:

Stuff Name	Description	Reference Materials	Location
JSState.xml	State configuration file	'Create configuration file for palette object' on page 81	palette folder in Drop4Extension project
Info16.PNG and Info32.PNG	Icons used to show State in Processor Palette and Canvas	'Register palette object' on page 83	icons folder in Drop4Extension project
JSStateMapping.xml	Palette object mapping file	'Create and register object mapping' on page 85	config folder in Drop4Extension project
JSStateRule.java	Property generation rule for JSState	'Create and register object mapping' on page 85	src/com/ibm/btt/tools/drop4extension./ransaction./generator/ule folder in Drop4Extension project
FlowEditorXUIFileDragAndDropHandler.java	Drag-and-drop implementation class in runtime		src/com/ibm/btt/tools/drop4extension/transaction/presentation folder in Drop4Extension project
FlowEditorJSFileDragAndDropHandler.java	Drag-and-drop implementation class in runtime		src/com/ibm/btt/tools/drop4extension/transaction/presentation folder in Drop4Extension project

NLS support

National Language Support (NLS) is provided by the Processor Extension Editor. The attributes below are supported for NLS:

- label (in State and Transition palette extensions)
- description (in State and Transition palette extensions)
- displayName (in state and transition configuration files)
- description (in state and transition configuration files).

The definition of a NLS string must have the % prefix. At the same time, the pair <string key=string value> must be defined in the plugin.properties file.

The following is an example of NLS String definition.

.....

```
<property name="Page" displayName="%PageName" defaultValue=""
  hidden="false" required="true"
  description="%PagePath" editRule="PageSelection" />
```

....

Extend runtime processor

After a processor flow is composed by the Processor Editor, the flow can be executed at runtime. Infrastructure developers can also change the processor runtime behavior according the project requirement. For example, by default, the BTT processor does not process errors or exceptions that occur in a state at runtime. When there is an error in a state during processor execution, an error page

will be returned to end user. Application developers may want to provide more friendly interactions with end user (such as to prompt user decide continue or cancel the flow). They need to define a state to handle a specific error. Meanwhile, they need to define error event and transition. In this case, application developers have to spend some effort to define error handling states, events and transitions for each processor. If application developers want to implement implicit error handling at runtime for all processors, they can extend the default BTT processor at runtime.

As the BTT processor at runtime is channel specific. The following section gives information on how to extend the BTT html processor at runtime. For other channels, they can be extended in the same way.

How a flow processor works

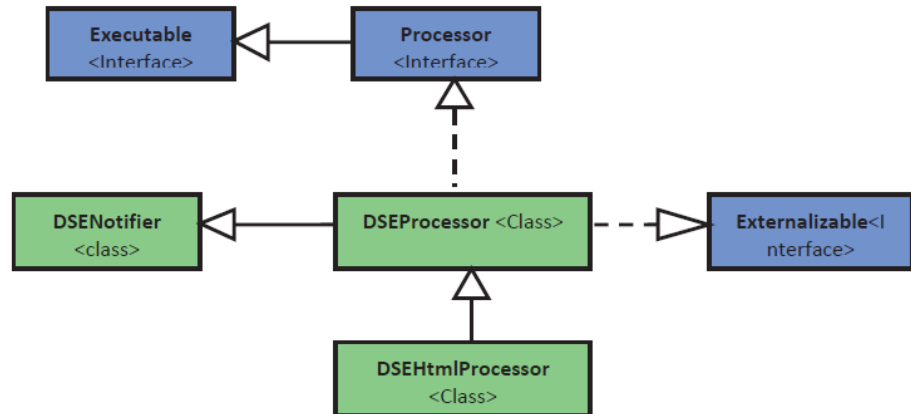
The following process describes how a flow processor handles a flow through states according to the definitions until the process reaches a final state:

- 1** The processor externalizer instantiates a given instance of the processor from its external definition. The behavior of the processor externalizer is the same as other BTT externalizers to create an object.
- 2** The externalizer searches for the name of the flow processor.
- 3** It obtains the flow processor class from the configuration file.
- 4** It sends the `initializeFrom(Tag aTag)` method to the flow processor instance to initialize it according to the definitions embedded in the tag.
- 5** When the toolkit initializes an instance of a flow processor, it caches in memory all of the possible states along with their actions, transitions, and data without actually instantiating them. Objects from these definitions are only created at runtime when they are required during the life cycle of the process.
- 6** The processor externalizer implements an object cache to significantly improve performance.
- 7** The toolkit executes the processor instance. The process handled by the flow processor starts in its initial state and follows a defined path until it enters one of its final states.
- 8** When the processor enters a state, the state registers with notifiers as being interested in any events specified in the state's transitions. The notifiers can be any notifier available in the context or any of the actions being executed.
- 9** The processor synchronously executes the entry actions of the state in the order in which they appear in the external definition of the state. If an entry action causes an event to fire and the event belongs to a transition defined for the state, the processor places the event in an internal queue to synchronize the actual handling of events.
- 10** After executing the entry actions, the flow processor checks the event queue and executes any events it finds there. If there are no events in the queue, the processor waits in the state for a triggering event. The use of the event queue to synchronize events does not prevent actions and guard conditions from having the opportunity to handle an event fired by a notifier while the processor is executing entry actions. The processor provides the event to them without losing the original event data.

- 11 To execute an event, the flow processor executes the actions for the event's transition after evaluating the guard conditions for each action. Depending on the results of evaluating the guard conditions and applying flow modifiers, the flow processor performs the exit actions defined for the state and then enters the defined target state to advance the process.

Extend flow processor

The DSEProcessor class extends the DSENotifier class and implements the Processor and Externalizable interfaces. The class diagram is shown as below:



Infrastructure developers need to extend `com.ibm.btt.automaton.htmlDSEHtmlProcessor` class to customize flow processor behavior for the HTML channel or extend `com.ibm.btt.automatonDSEProcessor` class for other channels.

After extending the BTT default processor implementation, Infrastructure developers need to register the new implementation in the class table of `btt.xml`. For example, for the HTML channel processor, Infrastructure developers need to replace the original implementation class with the new one. The following is an example:

Change definition from:

```
<field id="htmlProcessor" value="com.ibm.btt.automaton.html.DSEHtmlProcessor" />
```

to

```
<field id="htmlProcessor" value="com.ibm.btt.alpha.sample.automation.html.AlphaHtmlProcessor" />
```

After that BTT will use the extended process class to handle the flow in runtime.

Processor editor extension sample

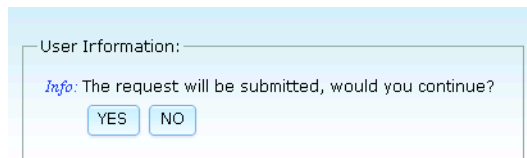
Two samples are provided to demonstrate how to implement Processor Editor extension in BTT framework.

- **ClientPromptState sample:** The sample demonstrates how to implement a Processor Editor palette state object and how to integrate it into the Processor Editor.
- **AlphaHtmlProcessor sample:** The sample demonstrates how to customize the BTT html processor behavior in runtime.

ClientPromptState sample

The ClientPromptState is a state object of the Processor Editor. It is designed to show prompt information in a web page to the end user during execution of the flow processor. It gives the end user options to continue or cancel the flow.

The example of a ClientPromptState in runtime is shown as below:



In this sample, primary Processor Editor object extension tasks are covered to help Infrastructure developers have overall understanding of implementation quickly. Below are list of tasks covered in this sample:

- Creating State object configuration file.
- Registering state as a plug-in extension point.
- Creating State object mapping file.
- Registering State object mapping as a plug-in extension point.
- Implementing property generating rule
- Registering property generating rule as a plug-in extension point.

The implementation stuffs of this sample are in two projects: AlphaSampleWidget project and BTTExtensionWeb project. 'Environment Preparation' on page 13 describes how to create the two projects. Below are list of stuffs for this sample:

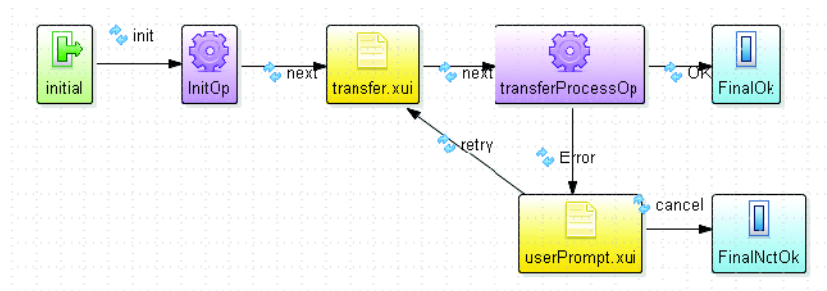
Stuff Name	Description	Reference Material	Location
ClientPromptState.xml	State configuration file	'Create configuration file for palette object' on page 81	palette folder in AlphaSampleWidget project
Info16.PNG and Info32.PNG	Icons used to show State in Processor Palette and Canvas	'Appearance tag of transition' on page 83	icons folder in AlphaSampleWidget project
transactionmapping.xml	Palette object mapping file	'Create and register object mapping' on page 85	config folder in AlphaSampleWidget project

Stuff Name	Description	Reference Material	Location
ClientPromptRule.java	Property generation rule for ClientPromptState	‘Register palette object’ on page 83	src/com/ibm/btt/alphasample/transaction/generator/rule folder in AlphaSampleWidget project
HtmlPromptStat.java	State implementation class in runtime	‘Create object mapping file’ on page 85	src/com/ibm/btt/alphasample/automation/htmlfolder in BTTExtensionWeb project

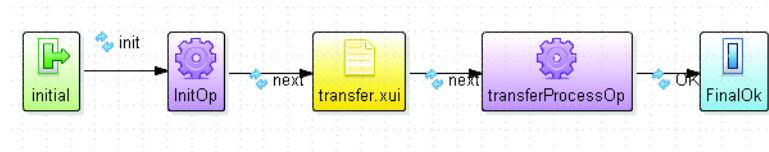
AlphaHtmlProcessor sample

The AlphaHtmlProcessor extends the BTT HTML channel processor to demonstrate how to handle implicit events. With this processor, developers need not to define any state or transition in processor to handle user prompt event. The processor will handle the user prompt event implicitly.

For example, the following figure shows a simple traditional account transfer flow. If an error occurs (such as the amount is more than daily transferring limit) during the transferProcessOp process account transfer request, a page will be returned to user ask for input again or cancel the process.



But with AlphaHtmlProcessor, developers do not need to userPrompt and FinalNotOk state. The processor will handle implicitly. The following figure shows flow using AlphaHtmlProcessor.



In this sample, primary runtime processor extension tasks will be covered.

The implementation stuffs of this sample are in two projects: AlphaSampleWidget project and BTTExtensionWeb project. ‘Environment Preparation’ on page 13 described how to create the two projects. Below are list of stuffs for this sample:

The implementation stuffs of this sample are in BTTExtensionWeb project. Below are list of stuffs for this sample:

Stuff Name	Description	Reference Material	Location
AlphaHtmlProcessor.java	Java class for runtime processor extension	'AlphaHtmlProcessor sample' on page 94	src\com\ibm\btt\alphasample\automation\htmlfolder in BTTExtensionWeb project
flowForProcessorExtension.transaction	Sample flow of using AlphaHtmlProcessor		src\definitions\processors folder in BTTExtensionWeb project



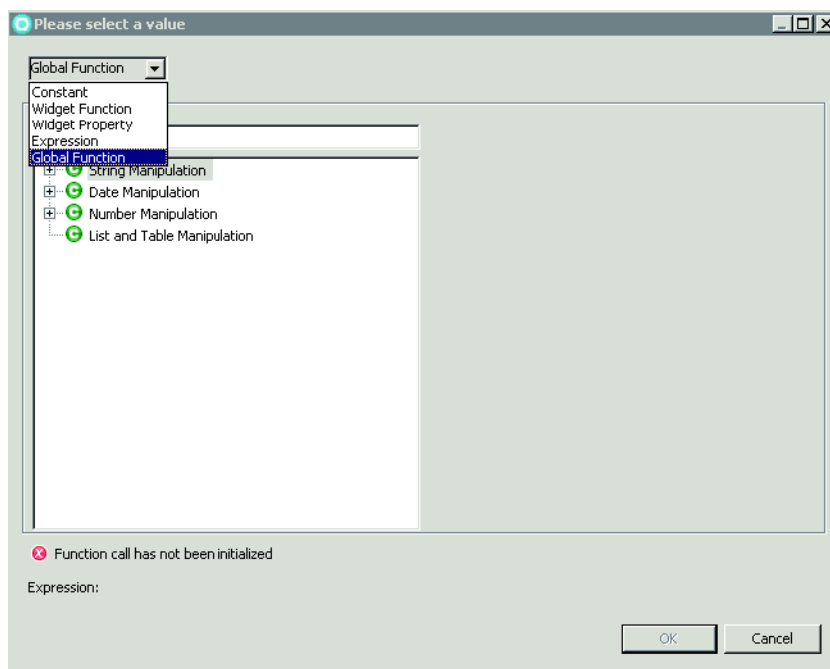
CHAPTER 8

Global Function Extension

BTT global functions provide common utility functions to manipulate data for BTT visual editors. BTT global functions are available in three BTT visual editors:

- XUI ECA editor
- Flow condition editor
- Flow mapping editor

The following screen shot is the usage scenario of the BTT global functions in the XUI ECA editor.



The following table lists the pre-defined global functions provided by the BTT product.

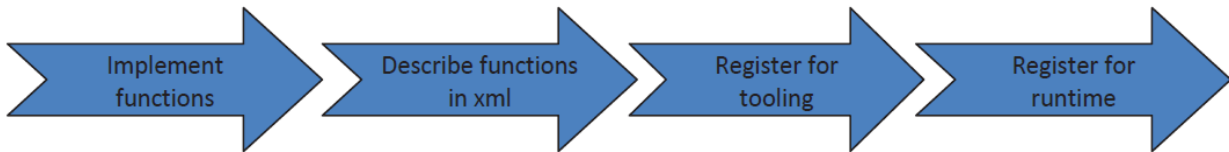
Category	Function Name	Description	Is Server Side	Is Client side
String Function	concat	Concatenates two strings	Y	Y
	length	Returns the length of a string	Y	Y
	contains	Returns whether a string is part of another string	Y	Y
	subString	Returns a portion of a string	Y	Y
	indexOf	Returns the position of a substring	Y	Y
	lastIndexOf	Returns the position of a substring starting from the end	Y	Y
	replace	Replaces all occurrences of a substring in a string with a new value	Y	Y
	trim	Removes the leading and trailing blanks	Y	Y
	upperCase	Converts a string to its upper case	Y	Y
	lowerCase	Converts a string to its lower case	Y	Y
	compare	Compares two strings lexicographically. Returns 0 if string1 is the same as string2, returns 1 if string1 is after string2, returns -1 if string1 is before string2 in dictionary order	Y	Y
	compareIgnoreCase	Compares two strings lexicographically, ignoring case differences. Returns 0 if string1 is the same as string2, returns 1 if string1 is after string2, returns -1 if string1 is before string2 in dictionary order	Y	Y
Number Function	round	Returns the closest long or integer value to a number	Y	Y
	truncate	Returns the truncated value of a number	Y	Y
	absolute	Returns the absolute value of a number	Y	Y
	numberToString	Returns value in string format	Y	N
	parseNumber	Returns a number parsing from a string	Y	N

Date Function	today	Returns the current date	Y	Y
	dayOfWeek	Returns the day of the week in number	Y	Y
	year	returns the year of the day in number	Y	Y
	month	Returns the month of the day in number	Y	Y
	day	Returns the date of the day in number	Y	Y
	after	Returns the date which is after than a given date with specified (days, months, years) period	Y	Y
	Before	Returns the date which is before than a given date with specified (days, months, years) period	Y	Y
	daysBetween	Returns the days between the two dates in decimal as time in day is taken into account	Y	Y
	natureDaysBetween	Returns the nature days between the two dates ignoring time difference	Y	Y
	parseDate	Returns date in BTT Date type, 'pattern' argument defines the format (such as 'dd/MM/yyyy') of the date parameter	Y	Y
Collection Function	tableSize	Returns the number of elements in an IndexedCollection	Y	N
	tableAdd	Adds the (numeric) values of a given column in an IndexedCollection	Y	N
	getRowByIndex	Returns the element of the given IndexedCollection according to the index	Y	N

Infrastructure developers could extend the BTT global functions to implement application specific utilities, such as function of converting a String to its lower case or upper case.

Extend global functions

The following figure shows the steps for extending BTT global functions.



Implement global functions

If a function is expected to run on the client side, the function should be implemented using the JavaScript language. If a function is expected run on the server side, the function should be implemented using the Java language. At the same time, the function should be declared as public and static method.

The following code sample demonstrates how to implement a global function for server side.

```

package com.ibm.btt.alpha.sample.drop3.globalfunction;

public class ExtendedFunctions {

    /**
     * Converts the characters in this String to lowercase.
     *
     * @return a new String containing the lowercase characters equivalent to the
     *         characters in this String
     */
    public static String toLowerCase(String value){
        return value == null ? value : value.toLowerCase();
    }

    /**
     * Converts the characters in this String to upper case.
     *
     * @return a new String containing the upper case characters equivalent to the
     *         characters in this String
     */
    public static String toUpperCase(String value){
        return value == null ? value : value.toUpperCase();
    }
}
  
```

The following code sample demonstrates how to implement a global function for the client side.

```

dojo.provide("com.ibm.btt.globalfunctions.FunctionExtension");

(function() {

    var bttExtension = {
        toLowerCase : function(obj) {
            if(obj != null)
                return obj.toLowerCase();
            else
                return obj;
        },

        isNumber : function(obj) {
            if ((obj.constructor === Number && !isFinite(obj) && !isNaN(obj))
                || /^[+-]?\d+${|^[+-]?\d+\.\d+$/ .test(obj)) {
                return true;
            } else {
                return false;
            }
        },

        isNull : function(obj) {
            return typeof (obj) == "undefined" || obj == null ? true : false;
        }
    };

    window.ExtensionFunctions = bttExtension;

})();

```

Describe global functions in xml

A global function should be described by an xml file before the BTT tools can show it in the visual editors.

functions tag

It contains one or more function tag.

function tag

The tag describes a function signature. It may contain parameter tags. The following is list of the attributes of a function tag.

Attribute Name	Description
name	The name of the function. It should be unique in a file.
returnType	The return type of the function such as String, Date, Number and Boolean.
Description	The description of the function. The description will be shown in visual editor tool when mouse hovers on the function. The attribute value supports NLS when it starts with '%'
isServer	Indicates whether the function can be used on server side or not. When it is false, the function will not be shown in transaction editors. Default value: true
isClient	Indicates whether the function can be used on client or not. When it is false, the function will not be shown in XUI editor Default value: true

parameter tag

The tag describes a parameter of a function. The following is the list of the attributes of a parameter tag.

Attribute Name	Description
name	The name of the parameter which will be shown in visual editors.
description	The description of the parameter. It supports NLS when it starts with %.
type	The data type of the parameter such as String, Date, Number and Boolean

The following is an example of global function definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<functionslist xmlns="http://btt.ibm.com/FunctionslistSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/FunctionslistSchema BTTFunctions2.xsd" >
  <functions>
    <!-- ***** String Function Extension ***** -->
    <function name="toLowerCase" returnType="String" description="%toLowerCaseDescription">
      <parameters>
        <parameter name="value" description="%toLowerCase_param" type="String" />
      </parameters>
    </function>
    <!-- Validate isClient parameter -->
    <function name="toUpperCase" returnType="String" description="%toUpperCaseDescription" isClient="false">
      <parameters>
        <parameter name="value" description="%toUpperCase_param" type="String" />
      </parameters>
    </function>
    <!-- Validate isServer parameter -->
    <function name="isNull" returnType="Boolean" description="%isNullDescription" isServer="false">
      <parameters>
        <parameter name="value" description="%isNull_param" type="String" />
      </parameters>
    </function>

    <function name="isNumber" returnType="Boolean" description="%isNumberDescription" isServer="false">
      <parameters>
        <parameter name="value" description="%isNumber_param" type="String" />
      </parameters>
    </function>
  </functions>
</functionslist>
```

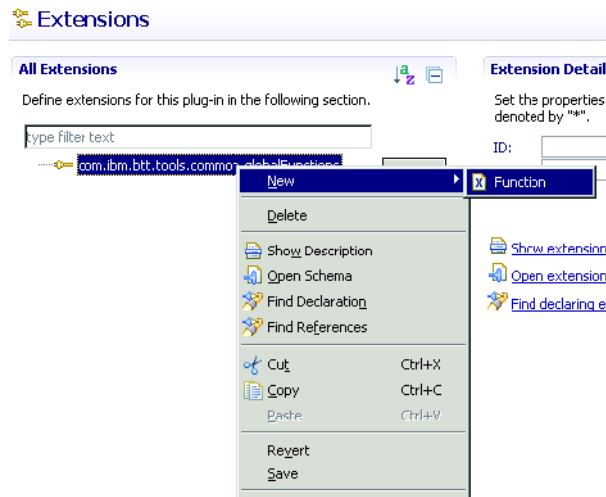
Register for tooling

To enable the extended global functions being shown in the BTT visual editors, Infrastructure developers need to register the functions as a BTT plug-in extension:

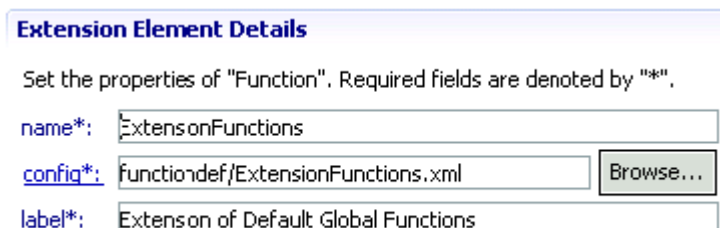
- 1 Open plugin.xml file
- 2 Click the **Extensions** tab.
- 3 Click **Add**
- 4 In the field **Extension Point Filter**, type **com.ibm.btt**.
- 5 Click **com.ibm.btt.tools.common.globalFunctions**.
- 6 Click **Finish**.



- 7 Right click **com.ibm.btt.tools.common.globalFunctions** then click **New > function**.



- In the **Extension Element Details** dialog box, type the applicable information.
 - **name** field requires users to input the name of the functions group.
 - **config** field requires users to provide the definition file described in ‘Describe global functions in xml’ on page 101.
 - **label** field requires users to input a display name for the functions group which will be displayed in BTT visual editors.



Register for runtime

For the global functions of the server side, the implementation class should be registered into the BTT global definition file (btt.xml). Then the BTT tools know how to invoke a defined global function at runtime.

Search for `globalFunctions` in `btt.xml`, and then add a new entry for the extended global function. The attribute `id` should be equal to the name registered in the BTT plug-in extension, and the value should be the function implementation class. The following snippet in bold is an example of function registering:

```
<kColl id="globalFunctions">  
  <field id="BTTStringFunctions" value="com.ibm.btt.utils.GlobalFunctions" />  
  <field id="BTTDateFunctions" value="com.ibm.btt.utils.GlobalFunctions" />  
  <field id="BTTNumberFunctions" value="com.ibm.btt.utils.GlobalFunctions" />  
  <field id="BTTCollectionFunctions"  
    value="com.ibm.btt.utils.GlobalFunctions" />  
  <field id="ExtensionFunctions"  
    value="com.ibm.btt.alpha.sample.drop3.globalfunction.ExtendFunctions" />  
</kColl>
```

For the client side, global functions are mapped into the related JavaScript object automatically. The object name of the JavaScript implementation should be equal to the name registered in the BTT plug-in extension.

Global Function Extension Sample

An extension sample of the global function is provided to demonstrate how to extend the BTT global functions. In this sample, two functions for the server side and three functions for the client side are provided. The following is the list of the extended functions:

- Server side:
 - `toLowerCase`: convert a `String` into lowercase.
 - `toUpperCase`: convert a `String` into uppercase
- Client side:
 - `toLowerCase`: convert a `String` into lowercase
 - `isNull`: check if a object has a value
 - `isNumber`: check if a `String` can be converted into number

A runnable sample is also provided to show the result of applying the extended global functions on both the client and server side.



CHAPTER 9

Generated JS File Name Extension

When selecting an XUI file to generate a Dojo page, if the XUI file contains ECA rule definitions, BTT will generate a Javascript file. The file is the Javascript implementation for the defined ECA rule. By default, BTT will create the file with the same name of the XUI file, just change suffix from .xuito .js. For example, if the XUI file name is index.xui, the generated JavaScript file name will be index.js. BTT provides an extendible point for convenience of Infrastructure developers to change the default naming behavior. For example, Infrastructure developers can add version information into the generated js file name.

Extend generated JS file naming rule

Implement naming rule

BTT provides three APIs for extending the default naming rule. Infrastructure developers need to implement `com.ibm.btt.tools.xui.editor2.generatorGenerateUIHandler` interface or extend the BTT default implementation class to implement the expected JS file naming behavior. The following is description of the three APIs.

```
/**
 * Indicates if it need to generate new JS file or not.
 * @param model XUI Root Model which contains isRuleDirty flag.
 * @return If return false, then ECA rule generation logic will not be invoked.
 */
public boolean generateRuleFile(IRootModel model);

/**
 * Get new generated rule file name. If rule file should contain version post-fix, then override this method.
 * @param model XUI Root Model which contains isRuleDirty flag.
 * @return New generated rule file name.
 */
public String getRuleFileName(IRootModel model);

/**
 * Clean old rule files if necessary. This method will be invoked before rule generation.
 * @param folder Target folder which will contain generated rule files in workspace.
 * @param model XUI Root Model which contains isRuleDirty flag.
 */
public void cleanRuleFilesIfNecessary(IFolder folder, IRootModel model);
```

The following is a sample implementation.

```
public class ExtendGenerateUIHandler extends GenerateUIHandler {

    public boolean generateRuleFile(IRootModel model){
        if(model.isRuleDirty()){
            return true;
        }
        else
            return false;
    }

    public String getRuleFileName(IRootModel model) {
        SimpleDateFormat fmt = new SimpleDateFormat("yyMMddHHmm");
        return getXUIFileNameWithoutExt(model) | "_" | fmt.format(new Date()) | ".js";
    }

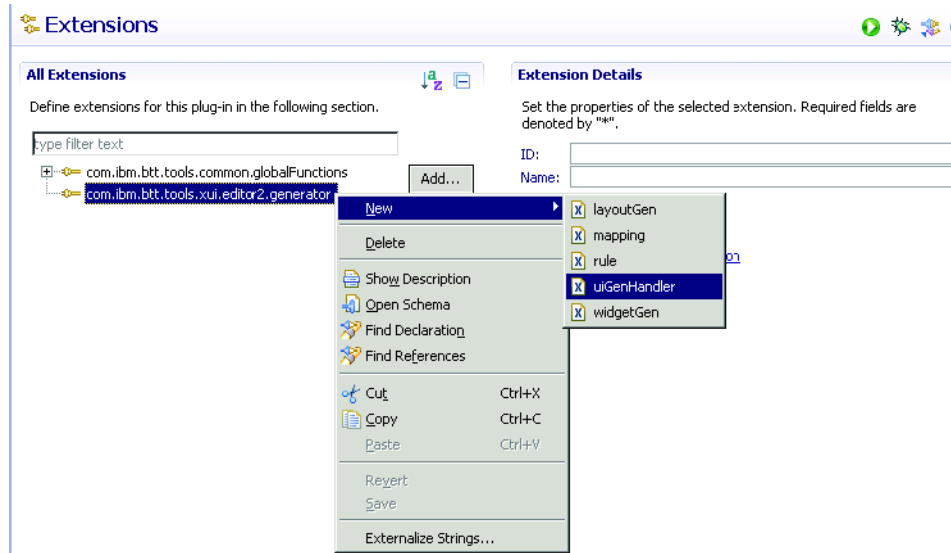
    public void cleanRuleFilesIfNecessary(IFolder folder, IRootModel model,
        IProgressMonitor monitor) throws CoreException{
        String xuiName = getXUIFileNameWithoutExt(model);
        IFile file = folder.getFile(xuiName+".js");
        if(file != null){
            file.delete(false, monitor);
        }
    }
}
```

Register implementation

The following describes how Infrastructure developers register the JS file naming rule.

- 1 Open `plugin.xml` file of BTT extension plug-in project.
- 2 Click the **Extensions** tab.
- 3 Click **Add**.
- 4 In the field **Extension Point Filter**, type `com.ibm.btt`.

- 5 Click **com.ibm.btt.tools.xui.editor2.generator**.
- 6 Click **Finish**.
- 7 Right click **com.ibm.btt.tools.xui.editor2.generator** then click **New > uiGenHandler**.



- 8 In **Extension Element Details** dialog box, type the applicable information.
 - **name** field requires the user to input name of the naming rule.
 - **class** field requires the user to input implementation class of extended naming rule described in ‘Implement naming rule’ on page 108.
 - **priority** field requires the user input the priority of this extension. If the extension point is registered by multiple plug-ins, the higher priority extension will be invoked by BTT. BTT default naming rule is registered as low priority. So the customized JS file naming rule should be registered as medium or high priority.

Extension Element Details

Set the properties of "uiGenHandler". Required fields are denoted by "*".

name*:

class*:

priority*:



CHAPTER 10

Naming Conventions Extension

When creating an element with the BTT visual tooling, the element name or ID starts with a default prefix. For example, when dragging a label widget onto canvas using the BTT XUI editor, the ID of new created label will start with label by default. Infrastructure developers can customize default BTT naming conventions to comply with their project naming specification, such as operations start with op, formatters start with fmt. Furthermore, naming validators can also be customized to validate if the created elements comply with the naming conventions.

Customized naming conventions can be applied to the following BTT elements:

Transaction Editor

- name of Transaction file
- ID of Processor
- ID of Operation
- ID of Operation step
- ID of Context
- ID of data, field, iColl, kColl, bColl
- ID of formatter
- ID of service

XUI Editor

- Name of XUI file
- ID of widget
- ID of ECA rule

Customized naming conventions can be applied in two levels:

- Project level: The naming conventions are applied to specific project.
- System level: The naming conventions are applied to all projects in workspace.

Extend naming conventions

There are two ways of customizing BTT name conventions:

- Extend by registering new naming convention rule.
- Extend by registering new naming manager class.

Extend rule by registering new naming convention rule

Overview

BTT provides a naming convention rule for Infrastructure developers to customize simple naming conventions without coding.

A rule is in the format of `elementRuleID=[prefix]{variable}[suffix]`.

- `elementRuleID`: is the rule ID of element supporting customized naming conventions.
- `prefix`: can be 0 or n number of characters.
- `variable`: can be 0 or n number of predefined variables can be referenced by naming convention rule.
- `suffix`: can be 0 or n number of characters.

Below is an example of naming convention rule:

```
FILE_TRANSACTION=newTransaction{file_count}.transaction
```

The following tables list BTT elements, and their naming convention rule IDs.

Transaction Editor elements

Elements	ID
Transaction file name	FILE_TRANSACTION
Processor ID	Processor
Operation ID	operation
Operation step ID	opStep
Context ID	context
Data ID	data
Field ID	field
Indexed Collection ID	iColl
Keyed Collection ID	kColl
Bean Collection ID	bColl
Formatter ID	fmtDef
Service ID	service

XUI Editor elements

Elements	ID
XUI file name	FILE_XUI
Widget ID	ELEMENT_WIDGET
ECA rule ID	ELEMENT_ECA_RULE

Predefined variables

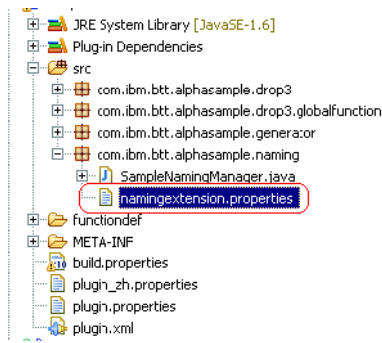
The following table lists predefined variables that can be referred to by the naming convention rule:

Variable	Description
file_count	Returns count of files applied the rule in current project.
file_name	Returns the file name
widget_type	Returns widget type string of the widget in XUI editor.
widget_count	Returns count of widgets applied the rule in current XUI file.
rule_count	Returns the count of ECA rule applied in current XUI file.
impl_class	Returns the implementation class name of a transaction element. For example, an implement class name of an Operation object.
oper_count	Returns the count of operations applied the rule in current transaction file.
op_step_count	Returns the count of operation steps applied the rule in current transaction file
ctx_count	Returns the count of Context in current transaction file.
item_count	Returns the count of elements with the same type in Data section of current transaction file.
fmt_count	Returns the count of formatter in current transaction file.
service_name	Return the service name of a service object. The object can be normal service or web service object. For web service, will return port name For normal service, will return implement class name
service_count	Return the count of service applied the rule in current transaction file.

Create naming conversion rule

A rule can be applied either in system level or project level.

- 1 Customize naming convention rule at system level
 - a Creating new bundler file with extension **properties** in plug-in project, for example: namingextension.properties.



b Edit the naming convention rule in the file. The following is a example of naming convention rule file:

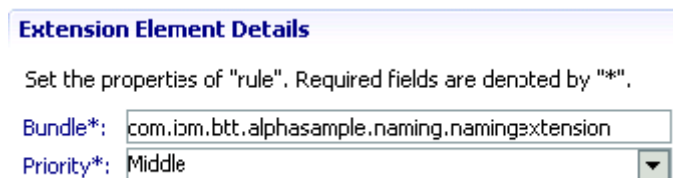
```

namingextension.properties
FILE_TRANSACTION=extTrans(file_count).transaction
Processor={file_name}
operation=ext(op_count)Op
opStep=ext(op_step_count)OpStep
    
```

- c** Open plugin.xml.
- d** Click the **Extensions** tab.
- e** Click **Add**.
- f** Click **com.ibm.btt.tools.common.naming**.
- g** Click **Finish**.
- h** Right-click **com.ibm.btt.tools.common.naming** then click **New > rule**.
- i** In the **Extension Element Details** dialog box, type the applicable information.
 - **Bundle** field requires the user to input the path of bundle file created in step 1.

Note Do not include the extension name properties in the bundle attribute

- **Priority** field requires the user input the priority of this extension. If the extension point is registered by multiple plug-ins, the higher priority extension will be invoked by BTT. The BTT default naming convention rule is registered as low priority. The customized naming convention rule should be registered as medium or high priority.



- 2** Customize naming convention rule at project level
 - a** Add a properties file under the project root path with the file name **naming_convention.properties**.
 - b** The file content is similar as the bundle file described in ‘Customize naming convention rule at system level’ on page 113.

Add new Variable

Infrastructure developers can create new variable and use it in a naming convention rule. The following steps describe how to create a new variable.

1 Implement the

`com.ibm.btt.tools.common.naming.variable.IVariableGenerator` interface. The following is description of `IVariableGenerator`:

```
public interface IVariableGenerator {
    /**
     * Character case policy
     */
    public enum CharCase {
        /**
         * Do not need to change first character's case
         */
        NoChange,
        /**
         * Change first character's case to upper case
         */
        UpperCase,
        /**
         * Change first character's case to lower case
         */
        LowerCase;
    }

    /**
     * Set generator's ID, default naming manager will call an IVariableGenerator
     * if this generator's ID is same as the variable id used in rule.<br>
     * By default, default naming manager will set this value base on the ID attribute of
     * extension point "com.ibm.btt.tools.common.naming -> variable_generator"
     * @param id the ID of this generator
     */
    public void setID(String id);
    /**
     * Get current generator's ID, default naming manager call this method to check if this generator can be used of a variable.
     * @return the ID of this generator
     */
    public String getID();
}
```

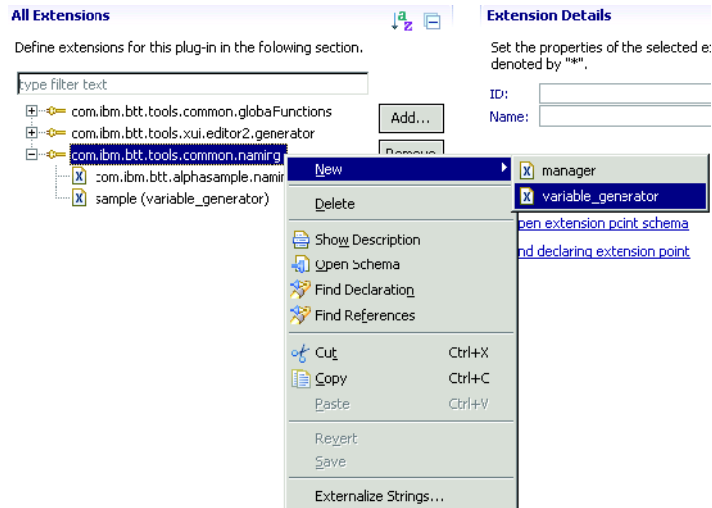
```

/**
 * Set first char case policy<br>
 * <B>NOTES:</B> default naming manager will apply this policy JUST when this variable generator
 * appear at first position of a rule
 * @param firstCharCase @see {@link CharCase}:
 * <ol>
 * <li><b>NoChange</b>: means do not need to change first char's case</li>
 * <li><b>UpperCase</b>: change first char to upper case</li>
 * <li><b>LowerCase</b>: change first char to lower case</li>
 * </ol>
 */
public void setFirstCharCase(String firstCharCase);
/**
 * Get first char case policy<br>
 * <B>NOTES:</B> default naming manager will apply this policy JUST when this variable generator
 * appear at first position of a rule
 * @return
 * <ol>
 * <li>NoChange: means do not need to change first char's case</li>
 * <li>UpperCase: change first char to upper case</li>
 * <li>LowerCase: change first char to lower case</li>
 * </ol>
 */
public CharCase getFirstCharCase();
/**
 * Default naming manager will call this method to convert a variable that defined in rule.
 * @param context {@link INamingContext}
 * @return
 * @throws ExpectedInfoNotFoundException
 */
public String getValue(INamingContext context) throws ExpectedInfoNotFoundException;
/**
 * Default naming manager will call the getValue method on each IVariableGenerator, and than
 * call this method on each IVariableGenerator.<br>
 * Can use this method to generator variable that can just be generated only all other variable
 * has been generated (for example: the file_count variable)
 * @param context {@link INamingContext}
 * @param rule rule string, each variable in this rule has been converted with relevant
 * IVariableGenerator's getValue method.
 * @return new whole rule name.
 * @throws ExpectedInfoNotFoundException
 */
public String postGenerate(INamingContext context, String rule) throws ExpectedInfoNotFoundException;

```

2 Register the variable generator

- a Right click extension point **com.ibm.btt.tools.common.naming** then click **New > variable_generator**.



- b** In the **Extension Element Details**, type the applicable information.
- i ID:** The **ID** will be used in naming convention rule. For example, the new variable's ID is **file_count**.
 - ii Generator:** The implementation class of variable generator.
 - iii FirstChar:** Indicates if need to change first character and how to change the first character.

Extension Element Details

Set the properties of "variable_generator". Required fields are denoted by "*".

ID*:

Generator*:

FirstChar:

Extend by registering new naming manager class

Infrastructure developers can have the full capability of naming BTT elements at the time they are created. On the other hand, Infrastructure developers have to implement from scratch and cannot leverage existing BTT implementation in naming. The following steps describes how to implement and register a new naming manger class.

Implement naming manager

BTT provides APIs for extending BTT naming convention from scratch. Infrastructure developers need to implement `com.ibm.btt.tools.common.naming.INamingManager` interface. The following is description of `INamingManager`:

```

public interface INamingManager {
    /**
     * Check point ID used when creating new XUI file
     */
    public final static String KEY_XUI_FILE = "FILE_XUI";
    /**
     * Check point ID used when creating new Transaction file
     */
    public final static String KEY_TRANSACTION_FILE = "FILE_TRANSACTION";
    /**
     * Check point ID used when creating new widget in a XUI file
     */
    public final static String KEY_WIDGET = "ELEMENT_WIDGET";
    /**
     * Check point ID used when creating new ECA rule in a XUI file
     */
    public final static String KEY_ECA_RULE = "ELEMENT_ECA_RULE";

    /**
     * get INamingManager instance
     * @param project which project this INamingManager will working for
     * @return
     * <li>if has project level naming convention rule, return naming manager with project level naming convention rule</li>
     * <li>else if just find customized system level naming convention rule, return naming manager for this level</li>
     * <li>otherwise, return default naming convention manager</li>
     */
    public INamingManager getInstance(IProject project);

    /**
     * Invoke from file creation wizard in order to get predefined file name.
     * @param type Wizard type for file creation, such as transaction wizard or UI wizard.
     * @param context runtime context @see {@link INamingContext}
     * @return file name after convention, return null if no naming convention applied.
     */
    public String getConventionId(String type, INamingContext context) throws HandlerNotFoundException, ExpectedInfoNotFoundException;

    /**
     * Validate element id to see if it follows naming convention or not.
     * @param type Wizard type for file creation, such as transaction wizard or UI wizard.
     * @param conventionId the convention ID need to be validated
     * @param context runtime context @see {@link INamingContext}
     * @return @see {@link NamingValidateResult}
     * @throws HandlerNotFoundException
     * @throws ExpectedInfoNotFoundException
     */
    public NamingValidateResult validateConventionId(String type, String conventionId, INamingContext context) throws HandlerNotFoundException,
}

```

Register naming manager

- 1 Open plugin.xml file of BTT extension plug-in project
- 2 Click the **Extensions** tab.
- 3 Click **Add**.
- 4 In the field **Extension Point Filter**, type **com.ibm.btt**.
- 5 Click **com.ibm.btt.tools.common.naming**.
- 6 Click **Finish**.
- 7 Right click **com.ibm.btt.tools.common.naming** then click **New > manager**.
- 8 In the **Extension Element Details** dialog box, type the applicable information.
 - **Class** field requires the user to input implementation class of naming manager described previously.
 - **Priority** field requires the user input the priority of this extension. If the extension point is registered by multiple plug-ins, the higher priority extension will be invoked by BTT. BTT default naming manager is registered as low priority. So the customized naming manager should be registered as medium or high priority.

Extension Element Details

Set the properties of "manager". Required fields are denoted by "*".

Class*:

Priority*:



CHAPTER 11

Multi-project Support in Extension

For a banking customer, it possibly has multiple channel applications, such as teller banking, internet private banking, corporate banking and etc. These applications may be required to share some common business logic or resources such as operations and flows, NLS files or image files. At development time, the applications are organized as multiple projects in RAD. BTT tooling supports referring to resources in a project from other projects. For example, developers can choose NLS definition from one project for a widget of another project. The BTT multi-project feature improves the maintainability of the application code, flexibility of project management and avoids code redundancy. Also it improves the runtime flexibility by hot deployment capability. For example, business logic in shared sub-flows or web resources like images or the CSS has been changed, administrators only need to re-deploy the shared EAR, the base business-specific application EAR does not need to be restarted.

The types of multiple projects:

- **Global Web Project:** the project contains web resources which are shared hierarchically with other global, common and local projects. In Global web projects, there are the following web resources types
 - NLS
 - CSS
 - Static Lists (for combos and selection)
 - Images
 - Dojo JS(include Dojo base, BTT Dojo and application extended widgets)
- **Global Java Project:** the project contains components which are shared hierarchically with other global, common and local projects. In Global java projects, there are the following web resources types
 - BTT Global Definitions XML (btt.xml, context, data, type, service etc.)
- **Common Java project:** the project contains self-defined operations generated by WS import wizard. These operations are then reused by other common and local projects. Typical resources contained in a common project are:
 - Self-defined operation

- Self-defined flow processor without views
- **Common Web Project:** the project contains sub page flows which are shared by local projects. The common web project are self-contained that means it has self-defined flows and needs to initialize btt.xml in the BTT startup servlet. In Global java projects, there are the following web resources types
 - Self-defined sub-flow with views
- **Local project:** the local project is application web project and does not share any component with other projects. It refers shared components from global and common projects. Local project has business-specific logic implemented by page flow and operations that cannot be shared. As best practice, in local projects, the data/type is not defined, it uses the data defined in the global java project.

Handle project prefix

In the BTT tooling, the following resources can be chosen from another project:

- 1 NLS definition
- 2 Image file,
- 3 Context definition
- 4 Data name in a context,
- 5 CSS file
- 6 List file
- 7 Flow
- 8 Operation

At time of generating JSP from XUI, the value of a widget attribute needs to be generated into runtime value properly.

The following mapping rules handle multi-project for widget attributes specially:

- `remoteNLSPathRule`: which handles NLS String for multi-project.
- `changePathRule`: which handles List file for multi-project.
- `remotePathRule`: which handles Operation for multi-project
- `checkDataNameRule`: which handles Data name for multi-project
- `changePathRule`: which handles Image file for multi-project

When a resource is selected from another project, BTT will add a project prefix to the resource reference information. In this case, resource reference information will be in the format of `[project prefix]:[resource reference identity]`. In order to avoid large modification when a common or global project is renamed, BTT uses a project map key instead of a project name as a project prefix.

The common or global projects information is defined in the BTT configuration file as `remoteProjectURL kColl` of the local project. For each entry, the `id` field is the project name, the `description` field is the project map key and the `value` field is the access information at runtime. The following is a sample of the map key definition:

```
<kColl id="remoteProjectURL">
  <field id="commonProject"
    value="http://localhost:8080/CommonWebPrject/"
    description="CommonWebProject" />
  <field id="globalProject"
    value="http://localhost:8080/GlobalWebPrject/"
    description="GlokalProject" />
</kColl>
```

For an extended widget, if its attribute refers to a resource in another project, the widget implementation needs to handle the project prefix to locate the resource. BTT provides two APIs for Technical developers to handle the project prefix.

- `IProject FilePathUtil.getProjectByResourceID(String resourceID, IProject activeProject)`

The API returns the project in which the resource is located according to the resourceID. If resourceID does not contain a project prefix, the activeProject will be returned. If the project map key is not defined in the BTT configuration file, an IllegalArgumentException exception will be thrown.

- String FilePathUtil.getValueByResourceID(String resourceID)

The API returns the resource without project prefix.

The following sample code demonstrates how to retrieve an icon file from a global project.

```
try{
    IProject imageProject =
    FilePathUtil.getProjectByResourceID(imageLocation,
        EditorUtils.getActiveProject());
    String imageRelativeLocation = FilePathUtil.getValueByResourceID(imageLocation);
    IFile imageFile = imageProject.getFile(imageRelativeLocation);
}
catch(IllegalArgumentException ie){
    // handle map key is not defined exception
}
```

The following sample code demonstrates how to retrieve a NLS definition from a global project.

```
try{
    IProject selectedProject = FilePathUtil.getProjectByResourceID(nlsLocation,
        EditorUtils.getActiveProject());
    String nlsTextField = getValueByResourceID(nlsLocation);
    String nlsValue = NLSUtils.getPropertiesValue(nlsTextField,selectedProject);
}
catch(IllegalArgumentException ie){
    // handle map key is not defined exception
}
```



CHAPTER 12

Pagination Extension

If a grid contains massive data, it will be an expensive operation to retrieve all data at once. Because the operation will consume a large amount of memory and network bandwidth to store and transfer the data. BTT provides pagination support on both the browser side and the server side. Large amounts of data can be separated into multiple parts, and each part can be retrieved at different time.

If the `isPageable` attribute of a grid widget is selected as true, the BTT pagination function will be enabled, and grid attributes for pagination are required for definition.

At runtime, when one of pagination widgets is selected in a browser, an Ajax pagination request will be sent to the server side. And then the BTT server operation will retrieve the data according to the pagination parameter and send it back to the browser.

On the server side, BTT invokes two server operations to handle a pagination request:

- **Technical pagination operation**

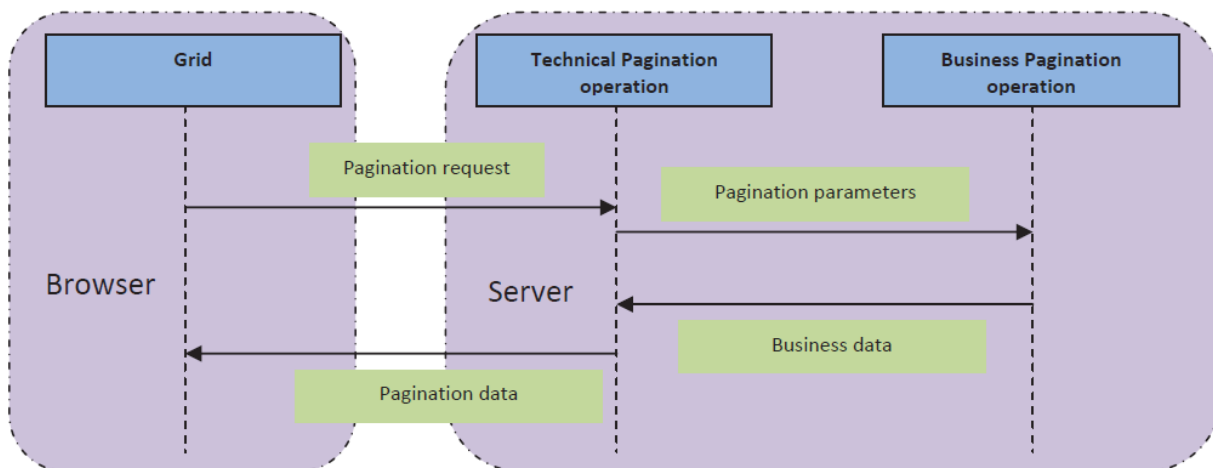
Technical pagination operation is implemented by the BTT product or Technical application developers. The operation is responsible for:

- Retrieving and handling pagination parameters from request.
- Chaining business operation context to processor context.
- Invoking business pagination operation
- Mapping data between business operation context and processor context
- Handling exception

- **Business pagination operation**

Business pagination operation is implemented by Functional application developers. The operation retrieves business data according to the pagination parameters.

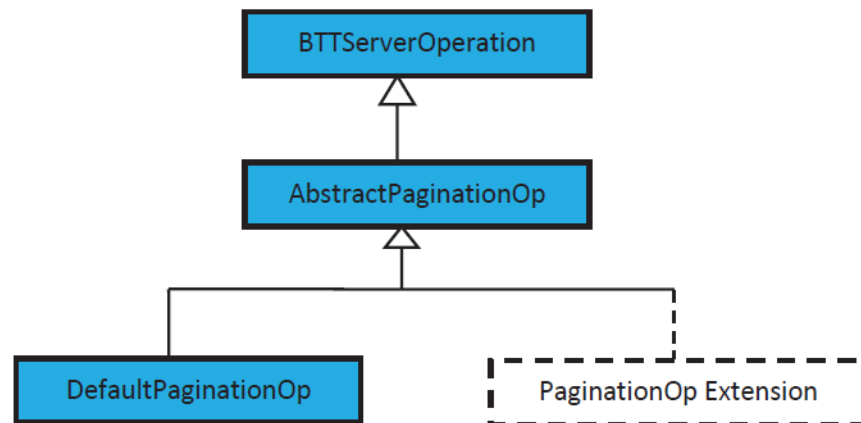
The following diagram demonstrates the relationship between the technical pagination operation and business pagination operation.



Extend technical pagination operation

The technical pagination operation is responsible for handling pagination parameters, such as rows per page and total page number. BTT provides a default technical pagination operation to handle the parameters. The application may have a requirement to handle application specific pagination behavior. Technical developers possibly need to implement a customized technical pagination operation.

The customized technical pagination operation needs to extend the class `com.ibm.btt.cs.ajax.AbstractPaginationOp`. The following diagram shows class hierarchy of technical pagination operation.



The customized technical pagination operation needs to override the following methods:

1 beforeExecutionBizOp

```

/**
 * The logic will be implemented on the project-level to get some required parameters
 * from the request data and put the parameter into the biz operation context
 * and chain the biz operation context into parent(processor) context
 *
 * this is occurred before the biz operation instanced.
 *
 * @throws DSEEException
 */
abstract void beforeExecuteBizOp() throws DSEEException;

```

2 handleBizOpInputMapping

```

/**
 * the alpha developer who create a new Technical operation can extends this method
 * for adding some data mapping from parent/processor context to the biz operation
 * context.
 * @throws DSEEException
 */
abstract void handleBizOpInputMapping() throws DSEEException;

```

3 handleBizOpOutputMapping

```

/**
 * the alpha developer who create a new Technical operation can extends this method
 * for adding some data mapping from biz operation context into parent/processor context.
 *
 * this method is called after the biz operation instanced.
 *
 * @throws DSEException
 */
abstract void handleBizOpOutputMapping() throws DSEException;

```

4 afterExecuteBizOp

```

/**
 * This logic will be implemented on the project level to handle the result
 * from the business operation and put the necessary data into the operation
 * context if necessary.
 *
 * @throws DSEException
 */
abstract void afterExecuteBizOp() throws DSEException;

```

5 handleException

```

/**
 * When an exception occurred in the operation executing process, how
 * to handle the exception. maybe update the errMsg field or just throw
 * the exception out
 *
 * @param e
 * @throws Exception
 */
abstract void handleException(Exception e) throws Exception ;

```


Pagination parameters

On the server side, when the BTT request handler receives a pagination request from the browser, it will parse the pagination parameters from the request and store the parameters in the operation context of the technical pagination operation. The technical pagination operation can use the parameters to do pagination correctly. Technical pagination operation context has three data parts:

- `tableProperties`

Contains the pagination attributes of the pagination table.

Field name	Description
<code>tableId</code>	The id of table defined in XUI
<code>dataName</code>	The <code>dataName</code> attribute defined in table
<code>dataNameForList</code>	<code>dataNameForList</code> attribute defined in table
<code>rowsPerPage</code>	<code>rowsPerPage</code> attribute defined in table
<code>operationNameForPagination</code>	<code>operationNameForPagination</code> attribute defined in table
<code>directPagination</code>	<code>directPagination</code> attribute defined in table
<code>tableColumnIDs</code>	<code>dataName</code> list of each column in table. Each <code>dataName</code> is separated by comma, for example {name1, name2, name3}

- `pageRequest`

Contains the control parameters of pagination

Field name	Description
<code>pageEvent</code>	The event that triggers pagination request. The possible value can be 'initial', 'next', 'prev', 'page'.
<code>pageNumber</code>	Page number to be requested
<code>sortData</code>	
<code>customData</code>	The reserved field for extension usage

- `pageReply`

Contains the data and control parameters after pagination request is processed.

Field name	Description
<code>totalRowNumber</code>	The total row number of data
<code>enableNext</code>	Will Next icon in client be enabled
<code>enablePrevious</code>	Will Previous icon in client be enabled
<code>errMsg</code>	Error message to be shown on client when exception occurs

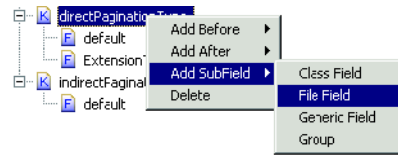
Field name	Description
items	Business data is retrieved
customData	The reserved field for extension usage

Register customized technical pagination operation

After a customized technical pagination operation is implemented, technical developers need to register the operation so Functional developers can choose the operation in the XUI editor.

The following steps describe how to add a new technical pagination operation:

- 1 Open `btt.xml`, and click the **Settings** tab.
- 2 Right click `directPaginationType` or `indirectPaginatortype` then click **Add SubField > Field Field**



- 3 In the **Detail Information** area type:
Id: the unique Id of the operation value: the class of the operation.
description: the description of the operation.
- 4 Click **Save**.



CHAPTER 13

Client State Extension

In a banking application project, there are some common usage scenarios to execute the client side operation from flow. Especially, for a web channel application, BTT supports the capability to interact with the UI feedback from the thin client (browser) in AJAX style.

For example in a BTT web channel application, sometimes we need:

- To pop-up a window dialog for implicit flow requirement to:
 - Close flow
 - Retry or close flow
 - Display a warning message and a button to continue
 - Go to the previous page
 - Go to the previous page or close flow
- Super-user approval
- Devices control
- Call client side applications such as TP16 or TP32 existing transaction.

For these requirements, BTT has provided a kind of abstract state called client state. Meanwhile, there is a default client state implementation for the model pop-up dialog in the BTT product. A more detailed introduction about the client state can be found in the product document.

For customer specific scenarios, alpha developers could implement their customized client state such as device control or override. To implement a customized client state, Infrastructure developers can follow the four steps described below.

Step 1: Extend a Client State

Client state is an abstract state class which defines the general behaviors for client interaction. In the BTT product, there is a default implementation of the client state called `PopupPageClientState`. Alpha developers always need to customize the client state to fulfill the usage scenarios.

Implement state class

The BTT product has provided an abstract class named `com.ibm.btt.automaton.ext.ClientState` to facilitate the alpha extension. This class is extended from the class of page state `com.ibm.btt.automaton.html.DSEHtmlState`. Alpha developers can focus more on the logics of client interaction and pay less attention on common state methods like `initializeFrom`.

In the extension usage scenario, there are four methods may be commonly extended by alpha developers in the application:

- `protected abstract String getCommand():`
This method returns the command of behavior which will be brought to the client side. For instance, an extended Client State used for printing forms may return a command like `fromPrint`. The JavaScript handlers registered for command `fromPrint` will be invoked. It is the required method to be implemented.

```
@Override
protected String getCommand() {
    Object obj;
    String command = "";
    try {
        obj = getProcessor().getContext().getValueAt("clientoperation");
        if (obj != null) {
            command = obj.toString();
        } else {
            if (logger.doError())
                logger.error("Client command is null.");
        }
    } catch (DSEObjectNotFoundException e) {
        if (logger.doError())
            logger.error("Error when reading client ccommand : ", e);
    }
    return command;
}
```

- `protected void afterFirstExecute():`
Client State will be executed twice during the flow execution. Once is when it is activated and gives a response to a client request; the second time when it handles a client response after the execution of the client logic and moves to the next state of the flow. Alpha developers could add extra logic for the server side in this method.

```
@Override
protected void afterFirstExecute() throws DSEProcessorException,
    DSEInvalidArgumentException {
    super.afterFirstExecute();
}
```

- protected void addRequiredDataToContext(Context context):
This method can be overridden to add or remove data which may be brought to client side. If you need to render or pop-up a page, you should set the value for the reply page as shown in the code snippet below.

```
@Override
protected void addRequiredDataToContext(Context context) {
    try {
        getProcessor().getContext().setValueAt(HtmlConstants.REPLYPAGE, getTypeIdInfo());
    } catch (Exception e) {
        if (logger.doError())
            logger.error("Error when setting replyPage in popup page state: ", e);
    }
}
```

Otherwise, if you are going to implement your own logic instead of page rendering, you can just leverage the implementation of the super class like the code snippet below.

```
@Override
protected void addRequiredDataToContext(Context context) {
    super.addRequiredDataToContext(context);
}
```

- public String generateClientResponse():
This method is used to convert the server data to JSON message and send them back to the client side. In the implementation of the class `com.ibm.btt.automaton.ext.ClientState`, all the flow context data will be formatted into JSON as response data. Alpha developers could override this method to filter the response data.

```
@Override
public String generateClientResponse() throws DSEInvalidRequestException {
    String res = super.generateClientResponse();
    try {
        JSONObject jo = JSONObject.parse(res);
        jo.put("XYZ", "xyz");
        res = jo.toString();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return res;
}
```

Register the implementation class into btt.xml

To make the extended state class available during runtime, alpha developers should register it into the `btt.xml` of target application. You need to open the `btt.xml` in your application project, and move to the processor tab. Then, you can register your extended state class in the class table of processors. See the following screen shot.

Detailed Information

id	<input type="text" value="testClientState"/>	
value	<input type="text" value="com.ibm.btt.test.cs.TestClientState"/>	<input type="button" value="Browse..."/>
description	<input type="text"/>	

Step 2: Enable the extended State in Transaction Editor

For more information about how to enable an extended state available in the Transaction Editor, see ‘Process Editor Extension’ on page 79. Also, this chapter explores how to enable the extended client state in the Transaction Editor step by step.

Create configuration file for the extended client state

Just like the extension steps for a state, alpha developers should define it with an xml file. To define a state for the palette, there are three kinds of tags will be used:

- `appearance`. This tag is used to indicate the appearance of the state shown in the palette. For example, we can control the font style with the attribute `font`.
- `properties`. This tag is used to group the property tags.
- `property`. This tag is used to describe the property for Transaction Editor about how to display, what is the default value or extra generation rule.

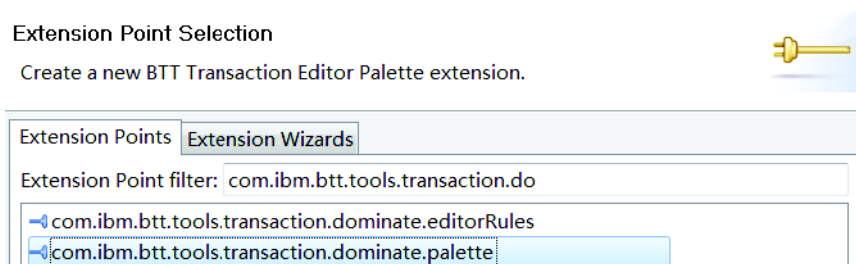
The code snippet below shows a sample of the configuration file for extended client state:

```
<?xml version="1.0" encoding="UTF-8"?>
<state xmlns="http://btt.ibm.com/StateSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/StateSchema StateSchema.xsd">
  <appearance backgroundColor="154,136,222" font="Arial-regular-10" fontColor="0,0,0" gradient="true" />
  <properties>
    <property name="id" defaultValue="" hidden="true" required="false" />
    <property name="Page" defaultValue="" hidden="false" displayName="Page" required="true"
      description="Page file path" editRule="PageSelectionBeta" />
  </properties>
</state>
```

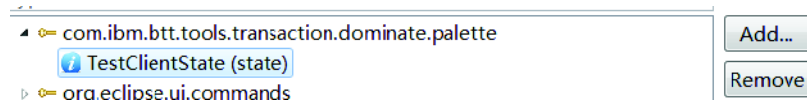
Register extended client state into the palette

To make sure the extended state could be chosen from the palette and dragged into flow canvas, alpha developers also need to register it in the Eclipse extension point of palette. For that, you can follow the steps below.

- 1 Open the `plugin.xml` in your plug-in project.
- 2 Click the **Extensions** tab.
- 3 Click **Add**.
- 4 In **Extension Point Filter** field, type `com.ibm.btt`.
- 5 Click `com.ibm.btt.tools.transaction.dominante.palette`.



- 6 Click **Finish**.
- 7 Right click **com.ibm.btt.tools.transaction.dominante.palette** then click **New > state** to create the applicable object.



- 8 Click **Finish**.
- 9 In **Extension Element Details** dialog box, type the applicable information.

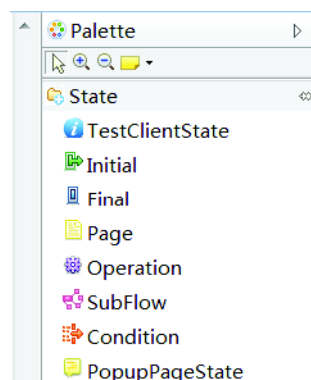
Note For the attribute **config**, alpha developers should browse the workspace to choose the definition file we defined already. For the attribute **stateParser**, you can just choose **PageStateParser** provided by BTT product for your extended Client State.

Extension Element Details

Set the properties of "state". Required fields are denoted by "**".

name*:	<input type="text" value="TestClientState"/>
label*:	<input type="text" value="TestClientState"/>
smallIcon*:	<input type="text" value="icons/info16.PNG"/> <input type="button" value="Browse..."/>
largeIcon*:	<input type="text" value="icons/info32.PNG"/> <input type="button" value="Browse..."/>
config*:	<input type="text" value="palette/TestClientState.xml"/> <input type="button" value="Browse..."/>
description:	<input type="text"/>
stateParser:	<input type="text" value="com.ibm.btt.tools.transaction.extend.parser.PageStateParser"/> <input type="button" value="Browse..."/>

You can now use the extended state in the Transaction Editor. The screen shot below shows the result view. You can find more detailed introduction about this part in the 'Process Editor Extension' on page 79.



Create configuration file for mapping rules

In most scenarios, alpha developers need to inject some extra generation rules into the Transaction Editor. To fulfill this requirement, alpha developers need to create a xml file in the plug-in project to define the mapping rules.

There are two kinds of tags commonly used in rules definition:

- tag-mapping. This tag is used to indicate the conversion rules between the tags used in transaction file (the value of attribute from) and the ones in the generated xml file (the value of attribute to).
- property-mapping. This tag is used to indicate the conversion rule between the attributes in transaction file (the value of attribute from) and the ones in the generated xml file (the value of attribute to).

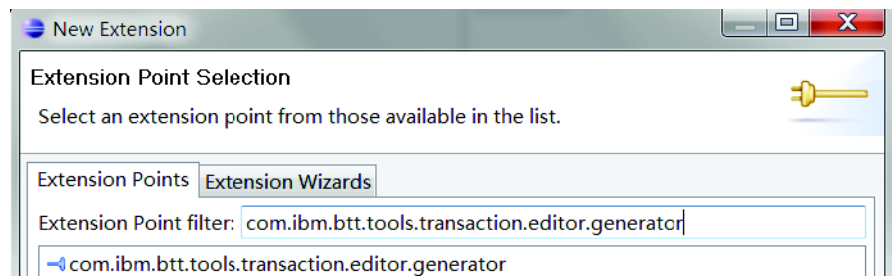
```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://btt.ibm.com/MappingsSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://btt.ibm.com/MappingsSchema MappingsSchema.xsd ">
  <tag-mapping from="TestClientState" to="testClientState">
    <property-mapping from="id" to="id" rule="PageIdRule" />
    <property-mapping from="Page" to="typeIdInfo" rule="PageRule"/>
  </tag-mapping>
</mappings>
```

Alpha developers could assign a more complex property generation rule with the attribute rule, which references to some program logic. The BTT product has provided several rule implementations and developers could also implement and register their own property generation rules. You can find more detailed information about his topic in ‘Process Editor Extension’ on page 79

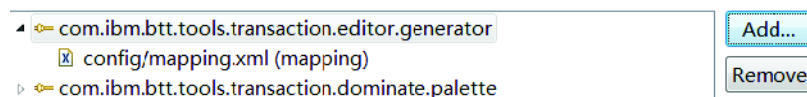
Register mapping rules

To make the mapping rules work in the Transaction Editor, alpha developers should register it with an Eclipse plug-in extension following the steps below:

- 1 Open the plugin.xml of your plug-in project.
- 2 Click the **Extension** tab
- 3 Click **Add**.
- 4 In **Extension Point Filter** field, type **com.ibm.btt**.
- 5 Click **com.ibm.btt.tools.transaction.editor.generator**.



- 6 Click **Finish**.
- 7 Right click **com.ibm.btt.tools.transaction.editor.generator** then click **New > mapping**.



- 8 In **Extension Element Details** dialog, type the applicable information.
 - **file**: browse to locate the xml file defined above.

- **target:** the name of currently used transaction generator should be input as its value.

Extension Element Details

Set the properties of "mapping". Required fields are denoted by "**".

file*:	<input type="text" value="config/mapping.xml"/>	<input type="button" value="Browse..."/>
target*:	<input type="text" value="Default Generator"/>	
description:	<input type="text"/>	

Step 3: Extend navigation engine to register command handler

After the client state is activated in the server side, a command will be added into the response data and sent back to the client side. Then the navigation engine will invoke different target command handlers to handle the related command in the reply data.

In order to correctly respond to the command and invoke the associated client operation logic, alpha developers should extend the navigation engine of the BTT product and register their own command handlers.

Extend the navigation engine to register a command handler

The navigation engine provided by BTT product is located in the file `NavigationEngine.js`. In this JS library file there is a function named `postCreate`, which will be invoked after the engine is created and all the widgets loaded for the first time. The extended command should be registered in this method as the code snippet below.

```
postCreate : function() {
    this.registerCommand("render_page", dojo.hitch(this, this.renderPageHandler));
    this.registerCommand("popup_page", dojo.hitch(this, this.popupPageHandler));
    this.registerCommand("redirect", dojo.hitch(this, this.redirectPage));
},
```

Alpha developers should extend the `NavigationEngine` of the product code, and then override the function `postCreate`

```
dojo.provide("test.ExtendedNavigationEngine");

dojo.require("com.ibm.btt.event.NavigationEngine");

dojo.declare("test.ExtendedNavigationEngine", [ com.ibm.btt.event.NavigationEngine ], {
    postCreate : function() {
        this.inherited(arguments);
        // register command handlers here
    }
});
```

Alpha developers should then define their command handlers which will be invoked by the navigation engine. The following code shows a sample of device control, which will invoke the printer to print current form. You can find more sample handlers in the sample project.

```
printFormHandler : function(resp) {
    console.log("response form", resp);
    this.clientStateResp = dojo.fromJson(resp.data);
    // todo
    window.print();

    var oldpars = this._getBTTHiddenData();
    var params = this.clientStateResp;
    console.log("new dse", params);
    params.dse_nextEventName = "ok";
    params.dse_pageId = oldpars.dse_pageId;
    this._submitData(params);
},
```

Finally, alpha developers should register the handlers defined above in the extended navigation engine. This is the same as the code style of the navigation engine of the BTT product.

```
postCreate : function() {  
    this.inherited(arguments);  
    this.registerCommand("cmd_print_form", dojo.hitch(this, this.printFormHandler));  
    console.log("ExtendedNavigationEngine created.");  
},
```

Step 4: Add the reference of new navigation engine to template

Alpha developers should modify the JS template of their BTT project. The following code snippet should be added to the end of the script section. After that, the XUI files should be generated again with the new template to make sure the extended engine and command handler would take effect at runtime.

```
dojo.require("dojox.data.QueryReadStore");
dojo.require("test.ExtendedNavigationEngine");
var BTT_ECA_MONITOR = new com.ibm.btt.event.BaseMonitor();
if(!window.engine){
    window.engine = new test.ExtendedNavigationEngine();
    engine.setMonitor(new com.ibm.btt.event.BaseMonitor());
}
```



CHAPTER 14

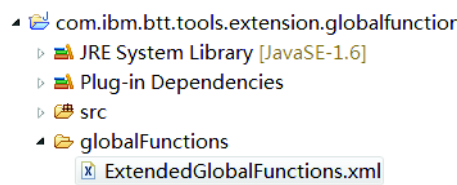
Reference Sample Topics

How to extend a global function invoked in ECA action

For the XUI editor, global functions could be invoked both in the Expression Panel and the Action Panel. In this reference sample, Infrastructure developers will be guided to extend a global function used in the ECA Action. The purpose of this function is to disable a widget in a browser.

Define global function in XML

- 1 Create a new plug-in project or use the already existing one (see 'Plug-in Project Setup' on page 14).
- 2 Create a new folder under the project; for example, name it as globalFunctions.
- 3 Create an xml file under the folder; for example, name it as ExtendedGlobalFunction.xml.



- 4 Edit the file to define a global function. Notice that, the attribute showInAction should be set to true to indicate this function will be available in the Action panel of XUI ECA editor.

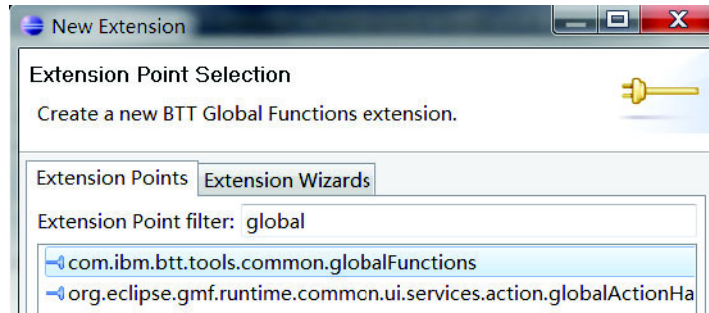
```

ExtendedGlobalFunctions.xml
<?xml version="1.0" encoding="UTF-8"?>
<functionslist xmlns="http://btt.ibm.com/FunctionslistSchema"
  xml:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://btt.ibm.com/FunctionslistS
  <functions>
    <function name="disableWidget" returnType="" description="Disable widget" showInAction="true">
      <parameters>
        <parameter name="widgetId" description="Widget ID" type="String" />
        <parameter name="flag" description="Flag to indicate disable widget or not" type="Boolean" />
      </parameters>
    </function>
  </functions>
</functionslist>

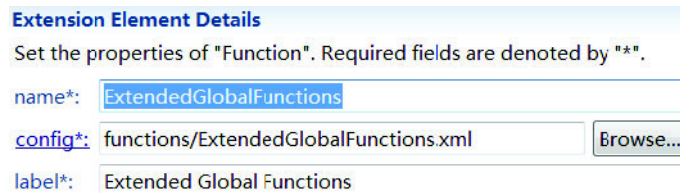
```

Register global function definition as Eclipse extension

- 1 Open the plugin.xml.
- 2 Click the **Extension** tab.
- 3 Click **Add**.
- 4 In **Extension Point Filter** field, type **global**.



- 5 Click **Finish**.
- 6 In **Extension Element Details** dialog, type the applicable information.
 - **name**. This value is the identifier of this extension.
 - **config**. Click **Browse** to find the configuration file defined in previous steps.
 - **label**. This value will be shown as the name of global functions group.



Implement JavaScript for global function

For global functions running in the browser side, Infrastructure developers should prepare a piece of JavaScript snippet. In this sample, the implementation code is used to disable a widget in a dojo page.

```

ExtendedGlobalFunctions.js
dojo.provide("com.ibm.btt.extension.globalfunction.ExtendedGlobalFunctions");

(function() {
    var egf = {};

    egf.disableWidget = function(widgetId, flag) {

        if(flag == undefined){
            flag = true;
        }

        var widget = dijit.byId(widgetId);

        if (widget.getDescendants != undefined) {
            var widgets = widget.getDescendants();
            for ( var i = 0; i < widgets.length; i++) {
                if (widgets[i].disabled != undefined) {
                    if (widgets[i].invalid != undefined) {
                        widgets[i].set("invalid", flag);
                    } else {
                        widgets[i].set("disabled", flag);
                    }
                }
            }
        }
        else if (widget.disabled != undefined) {
            if (widget.invalid != undefined) {
                widget.set("invalid", flag);
            } else {
                widget.set("disabled", flag);
            }
        }
        else {
            console.error("Widget "
                + widgetId
                + " neither is a container widget nor have a disabled property, can not disabe this widget.");
        }
    };

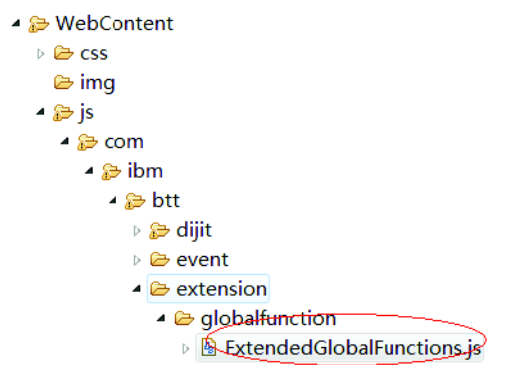
    window.ExtendedGlobalFunctions = egf;
})();

```

Enable XUI editor aware of this global function

Here are two steps for Infrastructure developers to make the XUI editor aware of this global function:

- 1 Put the JavaScript code under the WebContent folder of the runtime project. For this sample, the folder structure is like the following screen shot.



- 2 Modify the temple of XUI editor to make sure the implementation code of global function will be referenced by the generated JSP page.

```

dojo.require("dojox.data.QueryReadStore");

dojo.require("com.ibm.btt.extension.globalfunction.ExtendedGlobalFunctions");
var BTT_ECA_MONITOR = new com.ibm.btt.event.ConsoleMonitor();

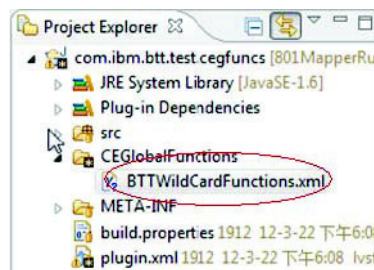
```

How to extend a global function to manipulate collection data

When composing a flow, sometimes it is necessary to manipulate the collection data contained in an indexed collection. For example, customers may have several accounts and each account has a balance. The account summary view need to present the sum of the balances of all accounts. In this topic, Infrastructure developers will be guided to fulfill this job by extending a global function.

Define global function in XML

- 1 Create a new plug-in project or use the already existing one.
- 2 Create a new folder under the project; for example, name it as CEGlobalFunctions.
- 3 Create an xml file under the folder; for example, name it as BTTWildCardFunctions.xml.



- 4 Define the signature of global function.

Note The type of the input parameter should be set as `Array` to enable the property editor used for collection data manipulation.

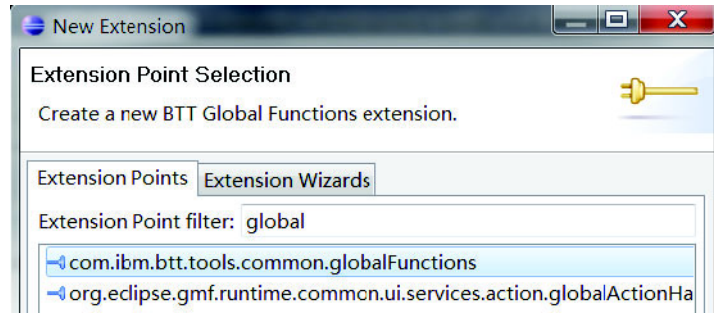
```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <functionlist xmlns="http://btt.ibm.com/FunctionslistSchema"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://btt.ibm.com/FunctionslistSchema BTTFunctions2.xsd">
5   <functions>
6     <function name="calcSummation" returnType="Number"
7       description="Summary the account balance" isClient="false">
8       <parameters>
9         <parameter name="balances" description="A list of account balances"
10          type="Array" />
11       </parameters>
12     </function>
13   </functions>
14 </functionlist>

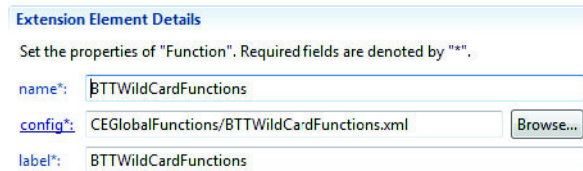
```

Register global function definition

- 1 Open plugin.xml.
- 2 Click the **Extension** tab.
- 3 Click **Add**.
- 4 In **Extension Point Filter** field, type **global**.



- 5 Click **Finish**.
- 6 In **Extension Element Details** dialog, type the applicable information.
 - **name**. This value is the identifier of this extension.
 - **config**. Click **Browse** to find the configuration file defined in previous steps.
 - **label**. This value will be shown as the name of global functions group.



Implement the function logic to calculate the sum of account balance

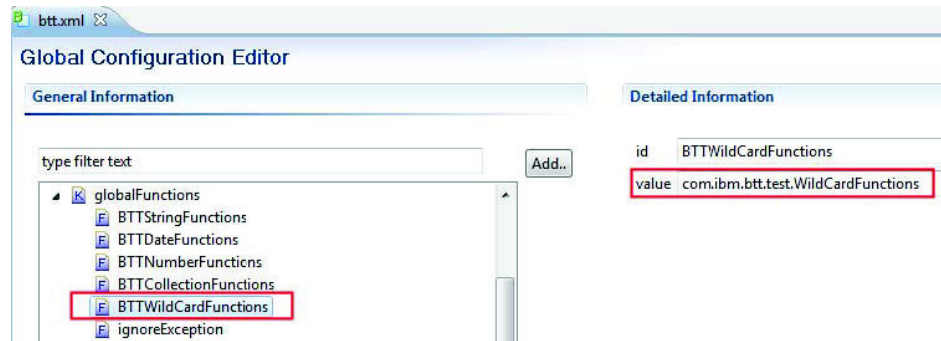
To calculate the sum of the account balances, the global function takes a value array of account balances as its parameter, and returns the calculated sum.

```
public static BigDecimal calcSummation(String[] strBalances) {
    try {
        BigDecimal sumBal = BigDecimal.ZERO;
        if ((null == strBalances) || 0 == strBalances.length)
            return sumBal;
        for (String strAcctBal : strBalances)
            sumBal = sumBal.add(new BigDecimal(strAcctBal));
        return sumBal;
    } catch (RuntimeException re) {
        if (!isIgnoreException()) {
            throw re;
        }
        return null; // return null if exception encountered
    }
}
```

Note that, the global function does not take `IColl` data as its parameter. BTT will automatically transform the specified `IColl.*.Balance` to an array of `String` values by using the instance `toString()` method. It is necessary to make sure the instances stored in `IColl.*.Balance` can be transformed into valid `String` values with the `toString()` method. The values can be transformed back to their original instance type.

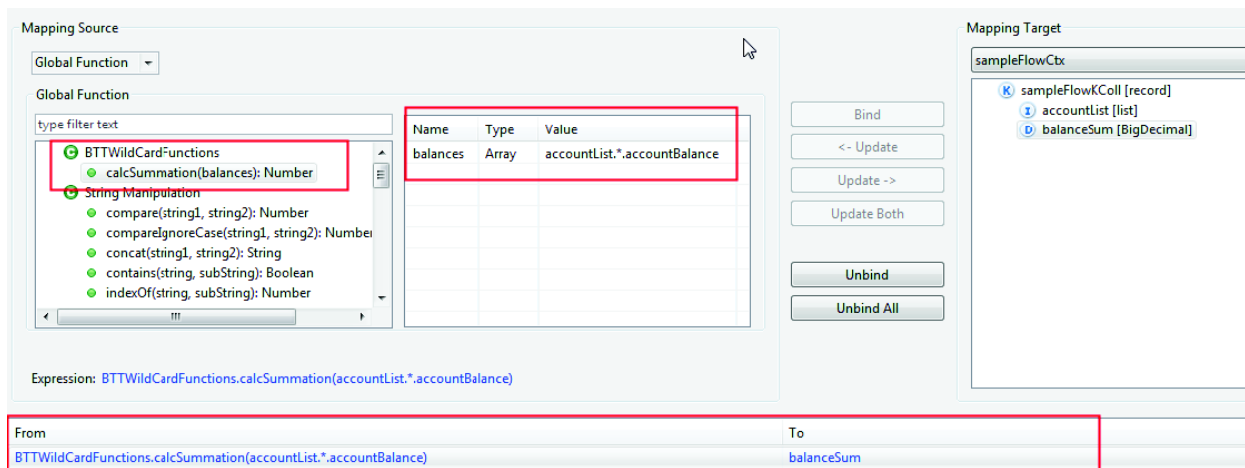
Register the implementation class of global function

In this sample, it is supposed that the extended global function is located in the class `com.ibm.btt.test.WildCardFunctions`. For the server-side global functions, Infrastructure developers should register the implementation class in `btt.xml`.



Usage Scenario of the global function in mapping editor

This section shows how to use the extended global function in the mapping editor. Note that, the parameter type is Array and the selected value for the parameter will be like `accountList.*.accountBalance`.



And finally, here is the generated XML snippet:

```
<fmtDef id="sampleOp.OK_InputFmt">
  <mapperConverterExpression>
    <map fromExpression="funcs_BTTWildCardFunctions.calcSummation(accountList$*$accountBalance)"
      to="balanceSum"/>
  </mapperConverterExpression>
</fmtDef>
```




www.unicomsi.com

We welcome feedback on our documentation. Please email us at:
tech.authors@unicomsi.com

www.unicomglobal.com