IBM FileNet Business Process Framework
Version 4.1

Developer Guide

GC31-5514-02

IBM FileNet Business Process Framework
Version 4.1

Developer Guide

GC31-5514-02

Note

Before using this information and the product it supports, read the information in "Notices" on page 87.

This edition applies to version 4.1.0 of IBM FileNet Business Process Framework (product number 5724-R75) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Revision Log

Date	Revision
8/31/2009	Removed samples for JBOSS/SQL Server from this guide and from the CODE_SAMPLES.zip file. Business Process Framework 4.1 does not support the JBoss Application Server. Modified the guide to correspond to changes in the code samples, which were revised to remove calls to unpublished core methods, and so on. Added information about using the Bp8CustomStyles.css file.
8/27/2008	Added documentation for APAR PJ34443 (BPF-4.1.0-001), describing how it is now possible to configure multiple case tabs in Business Process Framework by implementing custom code in the revised eventHandlers.onCaseDisplay method of EventHandlers.js (on BPF-4.1.0-001 and above only).
11/30/2007	Initial document posting

Introduction

This IBM® FileNet® Business Process Framework (BPF) Developer Guide is intended to be used in close conjunction with both the CODE_SAMPLES.zip file that accompanies it and with the actual files from your installed instance of Business Process Framework. To avoid having outdated code in either this document or in the CODE_SAMPLES.zip, reference will be made wherever possible to actual files from your installed instance of Business Process Framework (which you yourself will then need to pull up and examine in a text editor or the in conjunction with what is said here).

Customization areas

Business Process Framework exposes a number of customization areas for custom code extensions. Sample code for most of these is provided in the `CODE_SAMPLES.zip` that accompanies this guide. The sections of this guide correspond to these various customization areas. The description of each of them, in turn, draws examples directly from the `CODE_SAMPLES.zip` wherever possible.

In general terms, these customization areas fall under three broad categories:

- Client-side JavaScript™ EventHandlers
- Server-side JSP/Java™ plugins
- Cosmetic changes to `.CSS` and `.GIF` files

The client-side JavaScript EventHandlers are defined in three template files that are installed with the BPF Web Application:

- `<bpf_web_root>\plugins\custom\EventHandlers.js.template`
- `<bpf_web_root>\plugins\custom\ToolEventHandler.js.template`
- `<bpf_web_root>\plugins\custom\TableTabEventHandler.js.template`

The server-side JSP/Java plugins consist of such features as compiled Java classes configured to fire when a case is opened or dispatched from a designated inbasket (or with a designated action/response from a designated inbasket), custom tools and tabs (mostly using JSP), and (in most implementations) lookups (mostly using JSP) and custom preferences.

For further details, please consult the document below in conjunction with the `CODE_SAMPLES.zip`.

The `.GIF` files most likely to be subjected to customization are the files found in `<bpf_web_root>\img`.

You can further modify the appearance of the BPF Web Application by creating a new CSS file named `Bp8CustomStyles.css` and placing it in the `<web_root>\css\` directory. The BPF Web Application will automatically load and use this file if you create it with this name and deploy it in this location.

NOTE You should not make any changes to the `<web_root>\css\Bp8Styles.css` file!

Lastly, there is the option to customize the look and feel of the BPF Web Application by modifying the base XSL files used by BPF.

NOTE Please note, however, that unlike the other customization areas documented here, custom changes to these XSL files will be lost completely during any upgrade of BPF and will need to be merged back into the replaced XSL files after the upgrade.

BPF Web Application custom JavaScript EventHandlers and BusinessObjects interfaces

NOTE Please note the following points before proceeding.

- All path information provided is relative to the BPF `<web_root>` directory unless otherwise noted.
- The following descriptions and code snippets might possibly be out-of-date due to changes made in subsequent patches. For the latest and up-to-date information, always consult the comments and sample code in the `.template` files provided in the `\samples\components\bpf\custom` directory (relative to the `<install_root>` directory) of whatever build/release of BPF you happen to be working with.
- Values displayed for a given action/response in the **Action** menu may vary depending upon regional settings. Nevertheless, at present, the value of the `actionName` parameter passed in to the `onBeforeAction()` and `onAfterAction()` methods of the `EventHandlers.js` file is not localized. These values will be whatever they are configured to be in BPF Explorer.

Definition

The JavaScript `EventHandlers` interface for BPF provides the ability to implement custom code that will be run on the browser client workstation in response to a variety of events that may occur as the user interacts with BPF. This interface is defined by the `EventHandlers.js.template` file that is installed in the `\plugins\custom` directory.

As initially installed, this file is merely a blank template, whose method stubs contain nothing more than a `return true` statement. Nevertheless, each method defined there is introduced by extensive Javadoc™-style comments explaining in great detail what the method and its parameters do.

NOTE Please consult the `EventHandlers.js.SAMPLE` provided with this guide (in the CODE_SAMPLES.zip file) to learn more about how to implement this interface. This sample version of the file is probably where most developers will want to start becoming familiar with this interface and the kinds of functionality that it supports. In contrast to the `.template` file, the `.SAMPLE` file contains few explanatory comments and extensive code samples (commented out by default) and is intended to be the mirror image of the `EventHandlers.js.template` file – which contains no actual code but extensive comments. The two files are thus intended to complement one another and should be studied together.

Please consult the `EventHandlers.js.template` file in your own installed BPF environment to get an accurate description of this interface as it currently exists there. The following list (from the BPF-4.1.0-002 version of this file) is provided solely to give some idea of the kinds of events that are exposed by this interface.

```
eventHandlers.onBeforeResponseBulk = function(functionID, choiceRequired, functionName) {
	return true;
}


eventHandlers.onResponseBulkChoicesComplete = function(functionID, choiceRequired, functionName,
oChoiceResultXML) {
	return true;
}


eventHandlers.onResponseBulkCaseXmlComplete = function(functionID, choiceRequired, functionName,
oChoiceResultXML, oPropsXML) {
	return true;
}


eventHandlers.onAfterResponseBulk = function(functionID, choiceRequired, functionName,
oChoiceResultXML, oPropsXML) {
	return true;
}


eventHandlers.onPageLoad = function(sPageName) {
	return true;
}


eventHandlers.onCaseLoad = function() {
	return true;
}


eventHandlers.onCaseDisplay = function(tabContent) {
	return true;
}


eventHandlers.onBeforeHideDetail = function() {
	return true;
}


eventHandlers.onAfterHideDetail = function() {
	return true;
}


eventHandlers.onBeforeSave = function(oAuditLogItem) {
	return true;
}


eventHandlers.onAfterSave = function(oAuditLogItem) {
	return true;
}


eventHandlers.onBeforeClose = function() {
	return true;
}


eventHandlers.onAfterClose = function() {
	return true;
}
```

```
eventHandlers.onBeforeAction = function(responseName, responseId) {
        return true;
}


eventHandlers.onActionChoicesComplete = function(responseName, responseId, oChoiceResultXML) {
        return true;
}


eventHandlers.onActionCaseXMLComplete = function(responseName, responseId, oChoiceResultXML,
oPropsXML) {
        return true;
}


eventHandlers.onAfterAction = function(responseName, responseId, oChoiceResultXML, oPropsXML) {
        return true;
}


eventHandlers.onBeforeToggleExpando = function(oExpando) {
        return true;
}


eventHandlers.onAfterToggleExpando = function(oExpando) {
        return true;
}


eventHandlers.onFieldBlur = function(fieldElement) {
        return true;
}


eventHandlers.onAfterFieldBlur = function(fieldElement) {
        return true;
}


eventHandlers.onFieldChange = function(fieldElement) {
        return true;
}


eventHandlers.onFieldKeyPress = function(fieldElement) {
        return true;
}


eventHandlers.onKeyDown = function() {
        return true;
}


eventHandlers.onBeforePluginTabLoad = function (tab) {
        return true;
}


eventHandlers.onBeforeDoLookupSearch = function (lookupField, lookupButton) {
        return true;
}


eventHandlers.onBeforeProcessLookupItem = function (lookupButton, valueList) {
        return true;
}
```

```
eventHandlers.onAfterProcessLookupItem = function (lookupButton) {
        return true;
}


eventHandlers.onBeforeInbasketLoad = function() {
        return true;
}


eventHandlers.onAfterInbasketLoad = function() {
        return true;
}
```

These empty stubs can be fleshed out with custom implementation code that will be called at the appropriate times in response to various actions performed by users in the BPF Web Application.

NOTE The `EventHandlers.js.template` file provided with the BPF install package must, of course, be renamed `EventHandlers.js` before any of the code it contains will actually be executed.

Raising an Event

The methods defined in the `EventHandlers.js.template` are called through the `raiseEvent` method defined in `\js\Bp8InitMain.js`. If you examine the BPF JavaScript files, you will find calls to this method, raising the various events defined in the `EventHandlers.js.template` file at the proper times.

A typical call to this method, also taken from `\js\Bp8InitMain.js` would look like this:

```
        if (!raiseEvent( "onBeforeSave", oAuditLogItem)) {
                return;
        }
```

Please consult the JavaScript files in your installed BPF environment for more details.

NOTES

- The fact that the event handler methods are defined to allow for a boolean return value in turn allows client code to react dynamically and then to cause the operation in BPF that triggered the event either to fail or to proceed normally depending on the value returned from the event handler method by the custom code.
- It is possible to add custom events to BPF by extending the `EventHandlers.js` file with new methods and then calling them from code in `\js\Bp8InitMain.js` and the other baseline JavaScript include files. Such custom additions to BPF would not, however, be supported under normal circumstances and will be lost (overwritten) whenever subsequent BPF patches are applied.
- Any and all changes of general use and applicability to code in `\js\Bp8InitMain.js` and the other baseline JavaScript included files should be requested through the normal support channels using whatever formal change-control process happens to be currently in force. Only in this way can proper support and upgradeability be insured.

Event Sequence

The following is a short list of typical use cases, listing some of the event handlers that get called when users do various common actions in BPF and showing what source they are called from and in what order. Each of these is then followed by some brief suggestions as to what could be done at each point by inserting custom JavaScript into the body of the appropriate event handler method.

NOTES

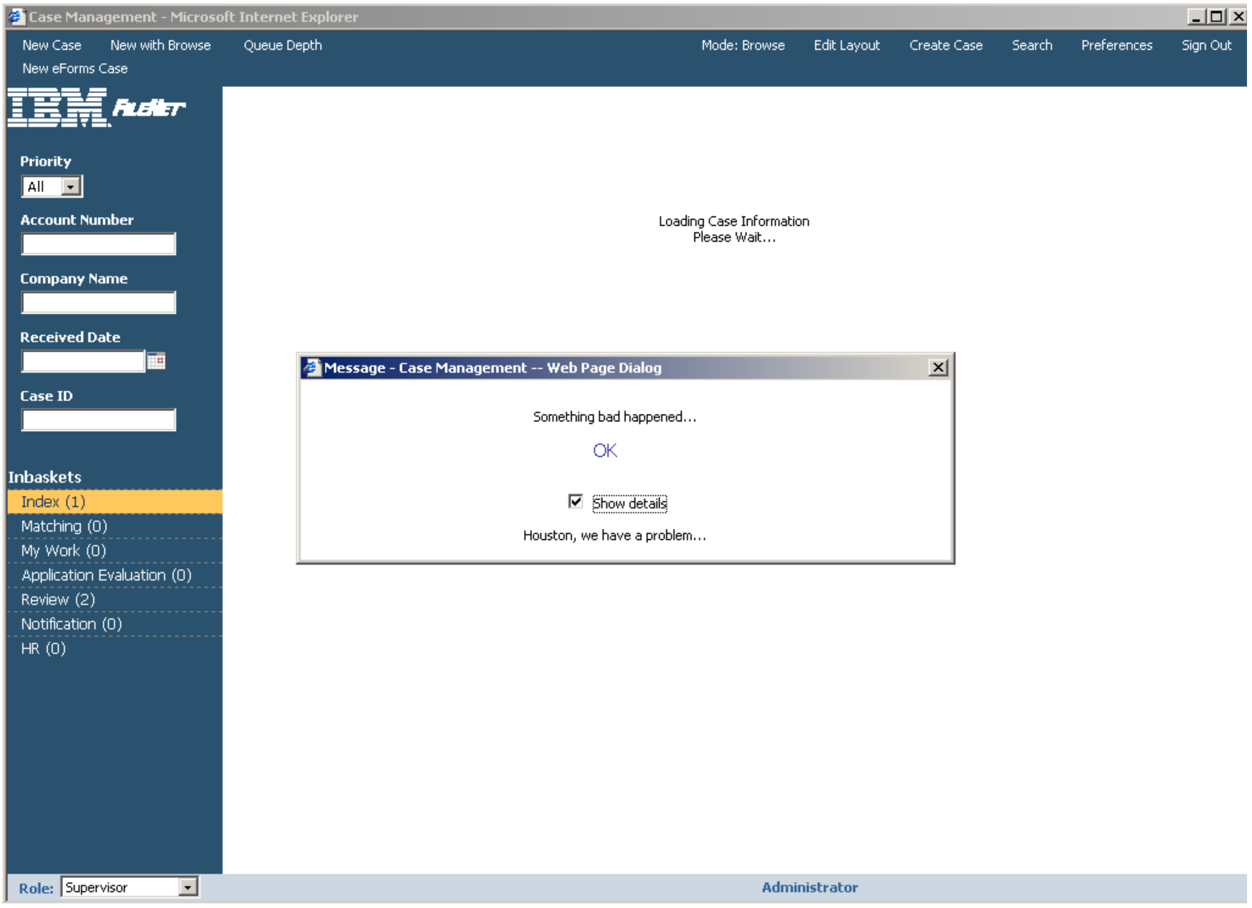
- The suggestions that follow are very rudimentary for the most part. To get a better idea of what can be done through this interface, consult the list and discussion of the JavaScript interfaces provided by the various supported business objects that follows this section.
- For displaying custom messages to the user, you are not confined to JavaScript alert calls. Consider using the `VoluntaryShowMessage` method defined in `Bp8Errors.js` to present the user with more elegantly formatted messages that you would like to display from your custom code, as in the following code snippet:

  ```
  var strDescription = "Something bad happened...";

  var strMessage = "Houston, we have a problem...";

  voluntaryShowMessage(strDescription, strMessage);
  ```

  There are also a number of other message functions defined in the same file as well if you need more specialized behavior. See the following screen capture for example results.



Loading Case Information
Please Wait...

Something bad happened...

Houston, we have a problem...

Scenario A

A user logs on to BPF and is presented with a browse list view of their default inbasket

Event raised

```
eventHandlers.onBeforeHideDetail();
```

Called from

`Bp8InitMain.js::hideDetail` (which is in turn called from `Bp8InitMain.js::openInbasket` and `Bp8InitMain.js::closeCase`)

What can you do?

`onBeforeHideDetail` is fired whenever the central area of the BPF user interface is hidden from view in the browser. This may occur whenever one inbasket's browse list is being closed so that another inbasket's browse list can be displayed – or when a case is being closed. This method is not fired when a case is being opened (for which use `onCaseLoad` and `onCaseDisplay` instead). If a case is being closed, you will still have access to both the HTML and the XML for the case that is closing here for the purposes of validation or manipulation.

Event raised

```
eventHandlers.onAfterHideDetail();
```

Called from

`Bp8InitMain.js::hideDetail` (which is in turn called from `Bp8InitMain.js::openInbasket` and `Bp8InitMain.js::closeCase`)

What can you do?

`onAfterHideDetail` is fired whenever the central area of the BPF user interface is hidden from view in the browser. This may occur whenever one inbasket's browse list is being closed so that another inbasket's browse list can be displayed – or when a case is being closed. This method is not fired when a Case is being opened (for which use `onCaseLoad` and `onCaseDisplay` instead). The `currentCase` object will be null at this point, so you will have no access to the CaseProps XML DOM for the case that has been closed. But you will still have access to the HTML DOM at this time.

Scenario B

A user clicks on a case in their inbasket browse list to open it.

Event raised:

```
eventHandlers.onCaseLoad();
```

Called from

```
Bp8InitMain.js::loadProps
```

What can you do?

Using the `currentCase.objCase` object (about which see below in the following section on supported business objects), you can do such things as:

- Auto-calculate a field value based on the value(s) of other fields
- Make a field disabled based on certain criteria
- Execute a call to an external URL and use the return values to populate fields as a kind of lookup

NOTES

- All of the above-noted things you can do here will be executed prior to the case being displayed.
- Returning `false` from this method prevents the case from loading.

Event raised

```
eventHandlers.onCaseDisplay();
```

Called from

```
Bp8InitMain.js:: loadDetail
Bp8InitMain.js:: td_tab_onclick
```

What can you do?

The case fields have already been rendered as HTML at this point, but it is possible to manipulate the HTML elements themselves.

NOTE Returning `false` from this method has no effect. It is not possible to prevent a case from displaying by returning `false` from this method.

Scenario C

A user tabs or clicks away from whatever field has focus.

Event raised

```
eventHandlers.onFieldBlur(fieldElement);
```

Called from

```
Bp8InitMain.js::fieldBlur
```

What can you do?

This is the primary method for performing edits and validation at the field level against the HTML DOM. It is possible to validate what the user has entered and, if it fails validation in some way, zero-out the field, display a message to the user, and reset the focus on the field.

NOTE It is possible to return a boolean value of `false` from this method, causing focus to be returned to the field. In this case, the `validateField` method will not be called from within `Bp8InitMain.js::fieldBlur`.

Event raised

```
eventHandlers.onAfterFieldBlur(fieldElement);
```

Called from

```
Bp8InitMain.js::fieldBlur
```

What can you do?

This is the primary method for performing XML edits and validation at the field level against the CaseProps XML DOM. At this point, the `ValidateField` method has been called and the CaseProps XML DOM has been updated. But it is possible at this point to validate what the user has entered and, if it fails validation in some way, zero-out the field, display a message to the user, and reset the focus on the field.

NOTE It is possible to return a boolean value of `false` from this method, causing focus to be returned to the field.

Scenario D

A user hits a character key to enter text into a field.

Event raised

```
eventHandlers.onKeyDown();
```

Called from

```
Bp8InitMain.js::doBodyKeyDown
```

What can you do?

This is more of a side-effect than anything else. Since this method does not take into account the HTML element in which the keystroke occurred (other than the containing `BODY` element as a whole), it is better suited to the implementation of hot keys and the like – using the interfaces supplied by the various supported business objects described below to invoke custom behavior in response to various combinations of key strokes.. For keystroke-level edits on specific fields and other HTML elements, use the `eventHandlers.onFieldKeyPress(fieldElement)` method described just below.

NOTE It is possible to return a boolean value of `false` from this method, which will cause the `Bp8InitMain.js::doBodyKeyDown` method to return immediately upon being invoked – without executing the remainder of the code it contains.

Event raised

```
eventHandlers.onFieldKeyPress(fieldElement);
```

Called from

```
Bp8InitMain.js::fieldKeyPress
```

What can you do?

This is the primary method to use for keystroke-level edits and validation on specific fields. It would be possible, for example, to use the `event.keyCode` property to simply cancel the entry of alpha characters into a numeric field, etc.

NOTE It is possible to return a boolean value of `false` from this method, causing the `fieldKeyPress` method to return immediately. In this case, the `validateField` method will not be called.

Scenario E

A user clicks the **Close** button.

Event raised

```
eventHandlers.onBeforeClose();
```

Called from

```
Bp8InitMain.js:: closeCase
```

What can you do?

At this point it is still possible to cancel the close operation altogether or even to change field values conditionally.

NOTE It is possible to return a boolean value of `false` from this method, which will cause the `Bp8InitMain.js::closeCase` method itself to return `false` immediately without executing the remainder of the code it contains.

Event raised

```
eventHandlers.onBeforeHideDetail();
```

Called from

```
Bp8InitMain.js::hideDetail
```

What can you do?

`onBeforeHideDetail` is fired before the case detail view of the currently opened case is hidden from view in the browser. You will still have access, for the purposes of validation or manipulation, to both the HTML and the XML for the case that is closing here.

Event raised

```
eventHandlers.onAfterHideDetail();
```

Called from

```
Bp8InitMain.js::hideDetail
```

What can you do?

`onAfterHideDetail` is fired after the Case Detail view of the currently opened case is hidden from view in the browser. The currentCase object will be null at this point, so no access to the CaseProps XML DOM for the case that has been closed will be available. But you will still have access to the HTML DOM at this time.

Event raised

```
eventHandlers.onAfterClose();
```

Called from

```
Bp8InitMain.js:: closeCase
```

What can you do?

At this point it is no longer possible to cancel the close operation altogether or to change field values conditionally.

NOTE It is possible to return a boolean value of `false` from this method, but this will have no real effect since the body of `Bp8InitMain.js:: closeCase` has already executed.

Scenario F

A user clicks the **Save** button.

Event raised

```
eventHandlers.onBeforeSave(oAuditLogItem);
```

Called from

```
Bp8InitMain.js::applyProps
```

What can you do?

At this point it is still possible to cancel the save operation altogether (prior to the case being saved), or to change field values conditionally. In addition, you can programmatically modify the action, reason, description and/or `eventCategory` properties of the `oAuditLogItem` parameter that will be written to the audit log using the `AuditLogItem` object interface defined in `Bp8BusinessObjects.js`.

NOTE It is possible to return a boolean value of `false` from this method, which will cause the `Bp8InitMain.js::applyProps` method to return `false` immediately without executing the remainder of the code it contains.

Event raised

```
eventHandlers.onAfterSave(oAuditLogItem);
```

Called from

```
Bp8InitMain.js:: applyProps
```

What can you do?

This method is called at the very end of the save operation. But code in the body of this method would still have access to some of the other business objects exposed by the business objects interfaces described below. In particular, changing the values of the `oAuditLogItem` parameter will have no effect on the actual audit log entry, but these values might be useful for logging or notification purposes.

NOTE It is possible to return a boolean value of `false` from this method here, but doing so will have no effect.

Scenario G

A user selects a response from the **Action** menu.

Event raised

```
eventHandlers.onBeforeAction(responseName==IndexPend, responseId==211);
```

Called from

```
Bp8InitMain.js::doFunction
```

What can you do?

This method can be used to prevent the dispatch of the current case, in spite of the fact that a user has chosen a response from the **Action** menu, if, for example, certain fields do not have values that are valid for this response. In this case, a message can be displayed to the user and the display returned to Case Details view, allowing the user either to correct the values of the required fields or to choose another action.

NOTE It is possible to return a boolean value of `false` from this method here, which will effectively cancel the response.

Scenario H

A user is presented with **Reason-** and **Comment** popup window if reasons are configured for the action chosen.

Event raised

```
eventHandlers.onActionChoicesComplete(responseName==IndexPend, responseId==211,
choiceResult==<ChoiceResults><Reason><ValueId>1</ValueId><ValueDesc>Invalid
Amount</ValueDesc><ValueInput>This amount is
invalid.</ValueInput></Reason></ChoiceResults>);
```

Called from

```
Bp8InitMain.js::doFunction
```

What can you do?

This method can be used to prevent the dispatch of the current case, in spite of the fact that a user has chosen a response from the Action menu, if, for example, certain fields do not have values that are valid for the reason and/or comment values (if any) that the user entered for this action. In this case, a message can be displayed to the user and the display returned to Case Details view, allowing the user either to correct the values of the required fields, to choose another action, or to choose the same action but with more appropriate reason and/or comment values.

NOTE It is possible to return a boolean value of `false` from this method here, which will effectively cancel the action.

Event raised

```
eventHandlers.onActionCaseXMLComplete(responseName==IndexPend, responseId==211,
choiceResult==<ChoiceResults><Reason><ValueId>1</ValueId><ValueDesc>Invalid
Amount</ValueDesc><ValueInput>This amount is
invalid.</ValueInput></Reason></ChoiceResults>, xmlProps== <CaseProps CaseId="67277"
CaseTypeID="3" GUID="{8CAFA318-606E-47C4-94AD-66CF3CEB62F9}"
WOBNum="3D256791B7B1754CB5786CE58F5C07C7" WorkobjectId="1300004.0"><Prop
Name="ContractAmount" Type="7"><Value>1</Value></Prop></CaseProps>);
```

Called from

```
Bp8InitMain.js::doFunction
```

What can you do?

This method can be used to prevent the dispatch of the current case, in spite of the fact that a user has chosen an action from the **Action** menu and the `xmlProps` parameter has been built for the fields whose values have been changed by the current user (so-called “dirty” fields), if, for example, certain fields do not have values that are valid for the reason and/or comment values (if any) that the user entered for this action. The presence of the `xmlProps` parameter also makes it very easy to see which fields the user changed and to do validation against these in isolation, rejecting the current action if any of the new values of the fields are inappropriate to the action, reason, and/or comment selected by the user. In this case, a message can be displayed to the user and the display returned to Case Details view, allowing the user either to correct the values of the required fields, to choose another action, or to choose the same action but with more appropriate reason and/or comment values.

NOTE It is possible to return a boolean value of `false` from this method here, which will effectively cancel the action.

Event raised

```
eventHandlers.onAfterAction(responseName==IndexPend, responseId==211,
choiceResult==<ChoiceResults><Reason><ValueId>1</ValueId><ValueDesc>Invalid
Amount</ValueDesc><ValueInput>This amount is
invalid.</ValueInput></Reason></ChoiceResults>, xmlProps== <CaseProps CaseId="67277"
CaseTypeID="3" GUID="{8CAFA318-606E-47C4-94AD-66CF3CEB62F9}"
WOBNum="3D256791B7B1754CB5786CE58F5C07C7" WorkobjectId="1300004.0"><Prop
Name="ContractAmount" Type="7"><Value>1</Value></Prop></CaseProps>);
```

Called from

```
Bp8InitMain.js::doFunction
```

What can you do?

Not much at this point, since the case has already been saved, closed, and dispatched. But this method might be very useful for logging purposes.

Event raised

```
eventHandlers.onBeforeHideDetail();
```

Called from

```
Bp8InitMain.js::hideDetail
```

What can you do?

`onBeforeHideDetail` is fired before the Case Detail view of the currently opened case is hidden from view in the browser. You will still have access to both the HTML and the XML for the case that is closing here, for the purposes of validation or manipulation.

Event raised

```
        eventHandlers.onAfterHideDetail();
```

Called from

```
        Bp8InitMain.js::hideDetail
```

What can you do?

`onAfterHideDetail` is fired after the Case Detail view of the currently opened case is hidden from view in the browser. The `currentCase` object will be `null` at this point, so you will have no access to the `CaseProps` XML DOM for the Case that has been closed. But you will still have access to the HTML DOM at this time.

Supported objects

JavaScript objects available for use in custom code in the JavaScript `EventHandlers.js` interface include the Case (`currentCase`) and Case object (`currentCase.objCase`), along with the Expando object (through the `onBeforeToggleExpando` and `onAfterToggleExpando` methods) and the custom plugin tab object (through the `onBeforePluginTabLoad` method). Each of these JavaScript objects/interfaces is defined in the `Bp8BusinessObjects.js` file in the `\js` directory of the BPF web root. In addition, the JavaScript `EventHandlers.js` interface also includes methods for lookup buttons (through the `onBeforeProcessLookupItem` and `onAfterProcessLookupItem` methods), though these are actual HTML objects rather than JavaScript objects defined in the `Bp8BusinessObjects.js` file. Some of these objects have their own event handler methods. But it is also possible to manipulate these objects in response to events raised by other objects (including Case, Case object, and various other BPF business objects, such as inbasket, tab, panel, tool, and user – all defined in the `Bp8BusinessObjects.js` file in the `\js` directory of the BPF web root) so as to achieve considerable complexity of behavior.

HTML Body element

The `onKeyDown` event handler defined in `EventHandlers.js` is fired through the `BODY` element containing the HTML for the entire screen. It is specific neither to a Case nor to any Case field.

Case

Probably the first thing that developers need to know when writing custom code within the `EventHandlers.js` interface is that it is possible to access the currently opened Case using the `currentCase` global variable (declared `\UI-INF\jsp\modules\apps\Bp8MainModuleJSP.jsp`).

NOTE Like the other BPF business objects described in this section, Case is a JavaScript object defined in the `\js\Bp8BusinessObjects.js` file. Although it is not the first object defined in that file, it will be handled here first and in greater detail than the others because of its relative importance.

As defined in the `\js\Bp8BusinessObjects.js` file, the Case object exposes simple `caseId`, `workobjectId`, and `workobjectNumber` properties as well as a complex `objCase` property containing the `CaseProps` XML in the form of an HTML DOM object.

```
function Case( caseId, workobjectNumber, workobjectId, caseGuid, caseNode, caseTypeId ) {
        this.caseId = caseId;
        this.workobjectId = workobjectId;
        this.workobjectNumber = workobjectNumber;
        this.caseGuid = caseGuid;
        this.objCase = caseNode;

//************************** Multiple Object Store Support ***********************************
        this.caseTypeId = caseTypeId;
//************************** Multiple Object Store Support ***********************************

        this.currentField = undefined;
        this.dirty=false;
        function Case.prototype.isOpened() {
                return (this.objCase != undefined);
        }
}
```

Likewise, calling `currentCase.isOpened()` can be used to determine whether the current case is open (has `CaseProps` XML) or not.

NOTE The BPF Case object is virtual and is exposed as an HTML DOM object through the `objCase` property or Case object described below.

The events supported on the Case level are:

- `onBeforeHideDetail`
- `onAfterHideDetail`
- `onCaseLoad`
- `onCaseDisplay`
- `onBeforeClose`
- `onAfterClose`
- `onBeforeSave`
- `onAfterSave`
- `onBeforeAction`
- `onActionChoicesComplete`
- `onActionCaseXMLComplete`
- `onAfterAction`

NOTE Please see the extensive comments in the `EventHandlers.js.template` that installs with BPF in the `<bpf_webroot>\plugins\custom` directory for up-to-date information on each of these methods.

Case object

This term is used to refer specifically to the `objCase` property of the `currentCase` object, which exposes the Case fields as HTML DOM objects.

Reading a Case field value

Because the BPF Case object is virtual and exposes its fields in the form of an HTML DOM object, the `currentCase.objCase` object/property can be accessed using standard syntax, as in the following example in which the value of the Effective Date case field is extracted from `currentCase.objCase` and loaded into a local variable.

```
var root = currentCase.objCase;
var property = root.selectSingleNode( "Prop[@Name='EffectiveDate']" );
var effectiveDate = property.selectSingleNode( "Value" ).text;
```

Setting a Case field value: Using the Dirty flag

This can be tricky because just setting the Value attribute of an HTML element will not result in the new value being persisted back to the Case object in Content Engine (CE) and/or the case work item in Process Engine (PE). Instead, the Dirty flag needs to be set on the relevant element by setting the `NewValue` attribute instead.

```
var root = currentCase.objCase;
var property = root.selectSingleNode( "Prop[@Name='ContractAmount']" );
property.setAttribute( "NewValue", 99 );
```

Appearance control

Using the syntax demonstrated above, one can use code such as the following to affect the appearance and functionality of the browser interface:

```
var oCompanyNameNoEdit = root.selectSingleNode("Prop[@Name='CompanyName']/NoEdit");

oCompanyNameNoEdit.firstChild.nodeValue = 1; // disabled, readOnly

oCompanyNameNoEdit.firstChild.nodeValue = 0; // enabled, writeable
```

Runtime field value control

Runtime field values are controlled via the following HTML DOM events: `onblur`, `onchange`, `onclick`, `onfocus`, `onkeydown`, and `onkeypress`.

The BPF-specific events currently supported on the field level by `EventHandlers.js` are:

- `onFieldBlur`
- `onAfterFieldBlur`
- `onFieldKeyPress`

Typical usage for this kind of event handler would be to provide custom inline field validation.

NOTE Please see the extensive comments in the `EventHandlers.js.template` that installs with BPF in the `<bpf_webroot>\plugins\custom` directory for up-to-date information on each of these methods.

Expando object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
    function Expando(id) {
        ...
    }
```

It exposes such properties as the Expando's ID and current state (expanded/collapsed) as well as expand and collapse methods for forcing either of these behaviors.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

AuditLogItem object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
    function AuditLogItem(action, reason, description, eventCategory) {
        ...
    }
```

It exposes such properties as the `AuditLogItem`'s `action, reason, description,` and `eventCategory` values, along with methods for serializing and deserializing such an object to and from XML.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Inbasket object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
    function Inbasket( inbasketNode, inbasketNumber ) {
        ...
    }
```

It exposes such properties as the Inbasket's `ID, name, number, type, visible, caseCount` and the like. It also exposes properties that can be used to determine whether a given Inbasket has **Save** and **Close** buttons. In addition, it exposes collections of the tabs, panels, and tools currently configured for the Inbasket, each in the form of a `nodeList` object.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Filter object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function Filter(id,name,type,value) {
        ...
}
```

It exposes such properties as the Filter's `ID, name, type, value, visible,` and the like.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Tab object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
    function Tab( tabNode, tabIndex ) {
        ...
    }
```

It exposes such properties as the tab's `ID, tabId, name, title, content, type, visible, path,` and `loaded` property.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Panel object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function Panel( panelNode ) {
    ...
}
```

It exposes such properties as the panel's `name, visible, enabled,` and `loaded` property.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Tool object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function Tool( toolNode ) {
    ...
}
```

It exposes such properties as the tool's `name, visible,` and `enabled` property, etc.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

User object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function User( userNode ) {
    ...
}
```

It exposes such properties as the current user's `ID, name, fullName, email, profile` and `preferences`.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Profile object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function Profile( userNode ) {
    ...
}
```

It exposes such properties as the profile's name`, ID, defaultTabId,` and `displayInbasketCaseCount`.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

CurrentContext object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function CurrentContext( user, profile ) {
    ...
}
```

It exposes many useful global variables pertinent to the current context as `readOnly` properties.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

ToolBarItem object

This object is defined by the following method in `Bp8BusinessObjects.js`.

```
function ToolbarItem( itemNode, itemList, parentItem ) {
    ...
}
```

It exposes properties and methods useful for the programmatic manipulation of toolbar items using custom JavaScript.

See the `Bp8BusinessObjects.js` file itself for more comments and implementation details.

Tool event handler interface

Definition

The JavaScript `ToolEventHandler` interface for BPF provides the ability for various tools (that are so written) to fire events that allow custom code to be executed during the course of executing the tool without having to modify the base code for the tool itself.

NOTE Tools need to be so written as to take advantage of this feature or it will not be available.

The interface is defined by the `ToolEventHandler.js.template` file that is installed in the `\plugins\custom` directory. As initially installed, this file is merely a blank template whose method stubs are all commented out.

NOTE A sample `ToolEventHandler.js.SAMPLE` is provided along with this guide, but since the implementation of the methods in this interface is specific to individual tools, no sample implementations have been provided beyond a bare minimum of debug statements that illustrate what information is available in each method call. As such, this sample file will provide most developers with their initial point of entry in becoming familiar with this interface.

As installed, the bare interface exposed by `ToolEventHandler.js` is as follows:

```
eventHandlers.onBeforeToolLoad = function(toolName) {
        return true;
}


eventHandlers.onAfterToolLoad = function(toolName, toolFields) {
        return true;
}


eventHandlers.onBeforeToolFieldBlur = function(toolName, fieldElement) {
        return true;
}


eventHandlers.onAfterToolFieldBlur = function(toolName, fieldElement) {
        return true;
}


eventHandlers.onBeforeToolAction = function(toolName, toolFields, toolAction) {
        return true;
}


eventHandlers.onAfterToolAction = function(toolName, toolFields, toolAction, toolObjectId) {
        return true;
}


eventHandlers.onToolKeyDown = function(toolName, keyCode) {
        return true;
}
```

These stubs can then be fleshed out with custom implementation code and called from JavaScript elsewhere in the web application by raising an event and specifying the name of the event handler method to be called.

Raising a tool event

These methods are called through the `raiseToolEvent` method defined in `\js\Tools.js`.

A sample call to this method from the `\js\CreateCase.js` file would look like this:

```
if (!raiseToolEvent("onBeforeToolLoad", toolName)) {
        return;
}
```

NOTES

- The fact that all of the tool event handler methods are defined to allow for a boolean return value in turn allows client code to react to the results of the code executed within the tool event handler method and either fail or proceed normally depending on the value returned.
- It is possible to add custom events to BPF by extending the `ToolEventHandler.js` file with new methods and then calling them from code in the source files for individual custom tools. Since the toolname is always passed in as a parameter, it would be possible to have each event handler method react differently for each tool that makes use of it. At present, this interface is used by the `CreateCase` and `AddDocument` tools only.
- Any and all changes to code in the baseline BPF tools and the other baseline JavaScript included files should be requested through the BPF Core team using whatever formal change-control process happens to be currently in force. Only in this way can proper support and upgradability be insured.

Tool event sequence

This remains fairly self-explanatory at present. The `ToolEventHandler.js.template` that installs with BPF and the `ToolEventHandler.js.SAMPLE` provided with this guide both come with extensive comments that should be consulted as the first line of documentation.

Supported tools

At present, only the `CreateCase` and `AddDocument` tools support this new `ToolEventHandler` interface, but the intention to have all BPF tools support it going forward.

Sample ToolEventHandler.js

Please consult the `CODE_SAMPLES.zip` file that accompanies this BPF Developer Guide for a copy of the current `ToolEventHandler.js.SAMPLE` file.

In order to install and test the `SAMPLE` files provided, copy them to your `<bpf_webroot>\plugins\custom` directory, remove the `.SAMPLE` extension, uncomment various blocks of code you would like to test, clear your browser cache, and try firing the corresponding events by performing the required actions in the BPF web application.

Table Tab event handler interface

Definition

The JavaScript `TableTabEventHandler` interface for BPF provides the ability for custom implementations of the BPF Table Tab interface to fire events that allow custom code to be executed during the course of working with the Table Tab without having to modify the base code for the Table Tab itself.

The interface is defined by the `TableTabEventHandler.js.template` file that is installed in the `\plugins\custom` directory. As initially installed, this file is merely a blank template whose method stubs are all commented out.

NOTE A sample `TableTabEventHandler.js.SAMPLE` is provided along with this guide, but since the implementation of the methods in this interface is specific to individual implementations of the Table Tab, no sample implementations have been provided beyond a bare minimum of debug statements that illustrate what information is available in each method call. As such, this sample file will provide most developers with their initial point of entry in becoming familiar with this interface.

As installed, the bare interface exposed by `TableTabEventHandler.js` is as follows:

```
eventHandlers.onBeforeTableRowClick = function( rowId , layoutName ) {
        return true;
};


eventHandlers.onAfterTableRowClick = function( rowId , layoutName ) {
        return true;
};


eventHandlers.onBeforeTableRowDblClick = function( rowId , layoutName ) {
        return true;
};


eventHandlers.onAfterTableRowDblClick = function( rowId , layoutName ) {
        return true;
};


eventHandlers.onTableInit = function( layoutName ) {
        return true;
}


eventHandlers.onBeforeTableRowAdd = function( rowId , layoutName ) {
        return true;
}


eventHandlers.onBeforeTableRowDelete = function( rowId , layoutName ) {
        return true;
};


eventHandlers.onAfterTableRowAdd = function( rowId , layoutName ) {
        return true;
};


eventHandlers.onAfterTableRowDelete = function( rowId , layoutName ) {
        return true;
};
```

```
eventHandlers.onTableFieldChange = function( element , rowId , layoutName ) {
       return true;
};


eventHandlers.onTableFieldKeyPress = function( element , rowId , layoutName ) {
       return true;
};


eventHandlers.onTableFieldBlur = function( element , rowId , layoutName) {
       return true;
};


eventHandlers.onTableButtonClick = function( layoutName ) {
       return true;
};
```

These stubs can then be fleshed out with custom implementation code and called from JavaScript elsewhere in the web application by raising an event and specifying the name of the event handler method to be called.

Raising a Table Tab event

These methods are called through the `raiseToolEvent` method defined in `\plugins\tabs\table.js`.

A typical call to this method from the `\plugins\tabs\grid.htc` file would look like this:

```
if(!raiseEvent( "onTableInit", getLayoutName())) {return;}
```

NOTES

- The fact that all of the Table Tab event handler methods are defined to allow for a boolean return value in turn allows client code to react to the results of the code executed within the tool event handler method and either fail or proceed normally depending on the value returned.
- It is possible to add custom events to BPF by extending the `table.js` file with new methods and then calling them from code in the source files for individual custom tools.
- Any and all changes to code in the baseline BPF tools and the other baseline JavaScript included files should be requested through the BPF Core team using whatever formal change-control process happens to be currently in force. Only in this way can proper support and upgradability be insured.

Web custom Java language Case event handler interface (Java plug-ins)

Following is a list of known problems, issues, and bugs relating to this feature.

- There is an almost complete lack of JavaDoc documentation for the Java classes involved in this interface. For now, the documentation provided here is the state-of-the-art reference for this feature.
- The strange variable name `handlerParemters` in the `CaseEventHandler.java` still needs to be corrected.
- There is currently no `getParameters` method defined on the `CaseEventHandler` interface to correspond to the `setParameters` method defined there. This means that the parameters class variable has to be accessed directly through code like the following:

```
if ((super.parameters != null) && (super.parameters.length > 0)) {
        for (int i = 0; i < super.parameters.length; i++) {
                logger.debug("super.parameters[" + i + "]==" + super.parameters[i]);
        }
}
```

The same is true of the `setCaseDOM` method on the same interface. There is no corresponding `getCaseDOM` method defined.

Definition

The Custom Java Language Case Event Handler Interface is defined by a small hierarchy of Java classes that reside in the `\com\filenet\bp8\api\ext` directory (and thus belong to the `com.filenet.bp8.api.ext` package) of the internal path of the `\<web_root>\WEB-INF\lib\bpfCM.jar` JAR file. As such, this interface is completely distinct from the custom JavaScript `EventHandlers` and `BusinessObjects` interfaces documented in the previous section.

The main difference between the two interfaces is that, while the custom JavaScript `EventHandlers` and `BusinessObjects` interfaces provide the ability to implement custom code that will be run on the browser client workstation, the custom Java Language Case Event Handler interface provides the ability to implement custom code that will be run on the web server itself.

One advantage the custom Java Language Case Event Handler interface has over the custom JavaScript `EventHandlers` and `BusinessObjects` interfaces is that the Java interface offers the full power of the Java programming language – especially in terms of strong data typing and a wide range of prepackaged classes and functionality. It is also possible to perform database lookups using JDBC and/or interact with third party applications on the web server for the purposes of validation or update from within code loaded through this interface.

By contrast, one disadvantage is that, while the custom JavaScript `EventHandlers` and `BusinessObjects` interfaces support a number of different events, the custom Java Language Case Event Handler interface only supports two: `OpenCase` and `CommitCase` (the latter of which can be configured to run either before or after the normal field validation that occurs when a case is dispatched).

A further disadvantage is that, while the custom JavaScript `EventHandlers` and `BusinessObjects` interfaces consist of ASCII text files (with a `.js` extension) that can be easily edited and redeployed, the custom Java Language Case Event Handler interface consists of compiled Java code, which means that changes to your Java source files that use this interface must be implemented by editing your files, recompiling, rebuilding your JAR file (unless you are deploying unbundled `.class` files), and then redeploying the web application after your newly recompiled code file is installed.

Class Hierarchy

All classes involved in this interface belong to the `com.filenet.bp8.api.ext package`.

- For a typical custom `OpenCaseEventHandler` class, such as the sample `OnOpen.java` provided in the code samples, the class hierarchy would be:

```
public interface CaseEventHandler {
public abstract class AbstractCaseEventHandler implements CaseEventHandler {
abstract public class OpenCaseEventHandler extends AbstractCaseEventHandler {
public class OnOpen extends OpenCaseEventHandler {
```

- For a typical custom `CommitCaseEventHandler` class, such as the sample `OnCommit.java` provided in the code samples, the class hierarchy would be:

```
public interface CaseEventHandler {
public abstract class AbstractCaseEventHandler implements CaseEventHandler {
abstract public class CommitCaseEventHandler extends AbstractCaseEventHandler {
public class OnCommit extends CommitCaseEventHandler {
```

NOTE The same Java class cannot be used for both the `OpenCase` and `CommitCase` events, since classes used for one or the other type of event should extend either `OpenCaseEventHandler` or `CommitCaseEventHandler`, respectively. It is especially important to note that `OpenCaseEventHandler` and `CommitCaseEventHandler` provide different implementations of the `getPropValue` and `setPropValue` methods defined in the `AbstractCaseEventHandler` class. *The two are not interchangeable.*

Specifically, the `getPropValue` and `setPropValue` methods defined in `OpenCaseEventHandler` differ from those defined in `CommitCaseEventHandler` in that the former simply sets the values of nodes in the `caseDOM` object with Java string values provided in the code. No validation takes place prior to the display of these values in the browser. By contrast, the methods provided by the `OpenCaseEventHandler` need to take into account two different DOM objects: the `caseDOMOriginal` and the `caseDOM` containing Dirty fields. It is also possible, depending on whether it is configured to run Pre or Post validation, that the normal validation procedures either will or will not have been run on the values involved for `CommitCaseEventHandler` classes.

NOTE This seems to be felt most acutely when using the `setPropValue` methods on DateTime fields, where the `OpenCaseEventHandler` version allows various string representations of the Date, whereas the `OpenCaseEventHandler` requires a long integer value for the same purpose.

For example, the call `setPropValue("ReceivedDate", "3/2/2009 5:26:13 PM");` will work fine in an `OpenCaseEventHandler`, but will fail in a `CommitCaseEventHandler`, which would require something like `SetPropValue("ReceivedDate", "1109721600000");` or `setPropValue("EffectiveDate", ("" + new Date().getTime()));` (both of which pass long integer values in string form). Neither form of call will work for the other class.

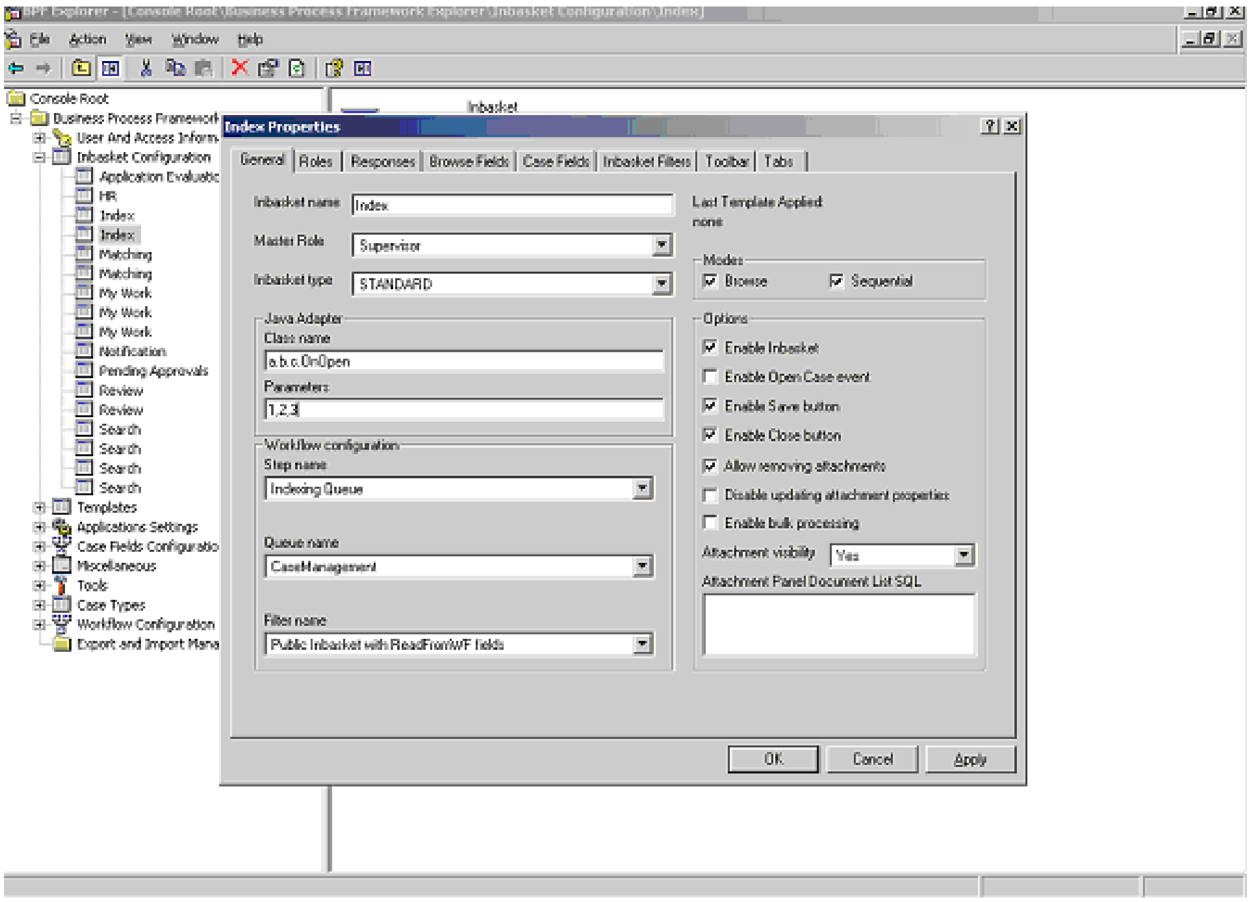
NOTES

- The date format actually used in calls to OpenCaseEventHandler.setPropValue (as seen in the example above) will depend upon the date format specified on the web server where this component is running. The example above shows a standard US formatted date, but the format required where other locales are specified may vary accordingly.
- At runtime, the event handler classes are actually loaded and run by the code defined in the `com.filenet.bp8.api.impl.Bp8CaseManagerImpl` class (`\com\filenet\bp8\api\impl\Bp8CaseManagerImpl.java`) which has access to the configuration information entered into the BPF Explorer related to Java event handlers as described in the following section. The `com.filenet.bp8.api.impl.Bp8CaseManagerImpl` class in turn makes use of the functionality defined and implemented in the `com.filenet.bp8.api.ext.CaseEventHandlerManager` class (`\com\filenet\bp8\api\ext\CaseEventHandlerManager.java`) in determining how and when to run the Java plugins configured.

OpenCaseEventHandler configuration

`OpenCaseEventHandler` classes are configured at the Inbasket level through two input boxes on the General tab of the inbasket configuration dialog. Only one such class can be configured for each inbasket, and the fully qualified Java class name must be provided (as in the example provided below) so that the application server in which BPF is running can find and load your code.

The following screenshot provides an example of how the sample `onOpen` Java class might be configured on an inbasket named Index.



Inbasket name: Index
Master Role: Supervisor
Inbasket type: STANDARD

Java Adapter
Class name: a.b.c.OnOpen
Parameters: 1,2,3

Workflow configuration
Step name: Indexing Queue
Queue name: CaseManagement
Filter name: Public Inbasket with ReadFromWF fields

Last Template Applied: none

Modes
- [✓] Browse
- [✓] Sequential

Options
- [✓] Enable Inbasket
- [ ] Enable Open Case event
- [✓] Enable Save button
- [✓] Enable Close button
- [✓] Allow removing attachments
- [ ] Disable updating attachment properties
- [ ] Enable bulk processing

Attachment visibility: Yes
Attachment Panel Document List SQL:

Parameters are an important feature of this interface. It is possible to pass a comma-delimited string of parameters to both `OpenCaseEventHandler` and `CommitCaseEventHandler` classes, but in each case this parameter string is configured differently. For `OpenCaseEventHandler` classes, a separate input field is provided for the parameter string. The sample provided here ("72,Index,Supervisor") represents the Inbasket ID, the Inbasket Name, and the Security Profile Name, which the custom code in `OnOpen.java` could be configured to recognize on the basis of their positions in the resulting `String[]` array and then load into local variables for use in decision logic and for other purposes. Alternatively, this information could be passed in as key=value pairs.

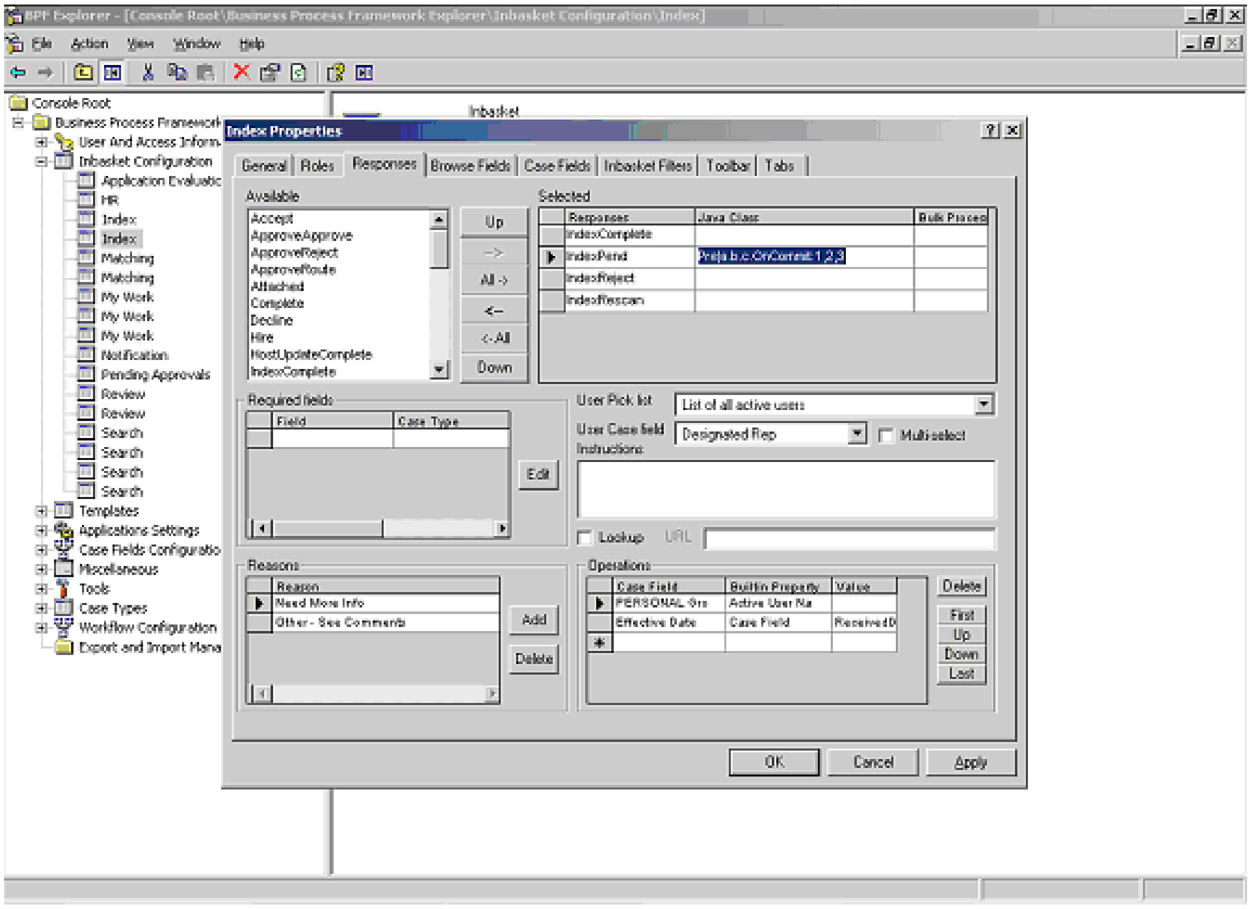
NOTE The fields in the Bp8Metastore database (COM_NAME and COM_PARAMS on the INBASKET_CONFIG table) that contain the Class name and parameters fields here are both defined as varchar(724), which should provide enough space.

Once these configuration settings are cached, the `OnOpen` Java class will be run each time a Case is opened from this Inbasket.

CommitCaseEventHandler configuration

`CommitCaseEventHandler` classes are configured at the inbasket-response level, using the Java Class field of the selected grid display on the Responses tab of the inbasket configuration dialog. Only one such class can be configured for each response, and the fully qualified Java class name must be provided (as in the example provided below for the `IndexPend` response).

The following screenshot provides an example of how the sample `OnCommit` Java class might be configured for the `IndexPend` response on an Inbasket named Index.



Available: Accept, ApproveApprove, ApproveReject, ApproveRoute, Attached, Complete, Decline, Hire, HostUpdateComplete, IndexComplete

Selected:

Responses	Java Class	Bulk Proces
IndexComplete		
IndexPend	a.b.c.OnCommit 1,2,3	
IndexReject		
IndexRescan		

Required fields:

Field	Case Type

User Pick list: List of all active users
User Case field: Designated Rep ☐ Multi-select
Instructions:
☐ Lookup URL

Reasons:

Reason
Need More Info
Other - See Comments

Operations:

Case Field	Builtin Property	Value
PERSONAL On	Active User Na	
Effective Date	Case Field	ReceivedD

Make sure you understand the syntax of the complex delimited string used to configure the `CommitCaseEventHandler` for each response:

```
        "Pre|a.b.c.OnCommit:72,index"
```

In particular, note that the fully qualified class name is preceded by the prefix `Pre`, followed by the delimiter (`|`). The other possible value for this substring prefix is `Post`. The default here is "`Pre`" if neither is specified.

NOTE This prefix is evaluated in a case-sensitive manner. Values like "`pre`", "`PRE`", "`post`", and "`POST`" will simply not work.

The significance of specifying `Pre` or `Post` is to determine whether the Java class is run before or after the normal field validation that occurs when a case is dispatched after a user selects an Action/Response from the Action menu. It does not refer to the actual dispatch of the case, which always occurs *afterwards* regardless of whether `Pre` or `Post` is specified.

NOTE This means that it is possible to prevent the actual dispatch by raising an exception within your custom Java class regardless of whether `Pre` or `Post` is specified.

NOTE A comma-delimited parameters list, preceded by its own (:) delimiter, can optionally follow the fully qualified name of the `CommitCaseEventHandler` class. The sample parameter string provided here ("`72,Index`") represents the Inbasket ID and the Inbasket name, which the custom code in `OnCommit.java` could be configured to recognize on the basis of their position in the resulting `String[]` array and then load into local variables for use in decision logic and for other purposes. Alternatively, this information could be passed in as key=value.

NOTE The field in the Bp8Metastore database (COM_NAME on the INBASKET_FUNCTIONS table) that contains the class name and parameters fields here is defined as varchar(724), which should provide enough space.

To summarize the preceding, the format of the single complex delimited string used to configure a `CommitCaseEventHandler` class for a response is as follows:

```
    "{Pre | Post}"  "|" "com.filenet.bp8.api.ext.MyCommitCaseEventHandler" [ ":"
    "param_1[,param_2[,...param_n]]]"
```

Sample OnOpen.java

The code in this file is designed to demonstrate some of the available hooks and show how it is possible, in the context of such a class, to do such things as obtain a string representation of any case field value using the `getPropValue` method, and then perform conditional logic or custom validation either directly against the resulting strings or after having converted them to their appropriate Java language data types.

The sample code also demonstrates some of the formatting issues involved in updating the values of case fields from this context.

Specifically, it is possible to:

- Reset fields that currently contain inappropriate or outdated values (as the result of an external database or web service call, for example).
- Reset fields to new values based on the values of other fields.
- Raise either a warning or an exception if certain conditions are found. **NOTE R**aising a warning allows a custom message to be displayed to the user, who can then either continue or cancel, whereas raising an exception prevents the case from being opened by the user at all.

- Reuse the same Java class on many different inbaskets, configuring different parameters on each inbasket and then checking for these values in the Java code so that the same class can act differently on different inbaskets.

All of this logic will all be executed before the user sees the case loaded in the Case Details view.

A simple use case for this kind of functionality would be a situation in which a case field for ExpirationDate is to be updated with a value from a database lookup and then compared with the current system date. If the ExpirationDate still lies in the future, the case can be displayed normally after the field is updated. If it lies in the past, a warning can be displayed before the case is displayed. If the database lookup fails because the database server is currently down, an exception can be raised, displaying a message to that effect, and the user will not be able to open the case until the database server is up and running again.

Consult the `\src\Java CaseEventHandlers Samples` folder in the code samples for more details.

Sample OnCommit.java

The code in this file is designed to demonstrate some of the available hooks and show how it is possible, in the context of such a class, to do such things as obtain a string representation of any case field value using the `getPropValue` method, and then perform conditional logic or custom validation either directly against the resulting strings or after having converted them to their appropriate Java language data types.

The sample code also demonstrates some of the formatting issues involved in updating the values of case fields from this context.

Specifically, it is possible to:

- Reset fields that currently contain inappropriate or outdated values (as the result of an external database or web service call, for example).
- Reset fields to new values based on the values of other fields.
- Raise either a warning or an exception if certain conditions are found. **NOTE** Raising a warning allows a custom message to be displayed to the user, who can then either continue or cancel, raising an exception prevents the case from being dispatched by the user.
- Reuse the same Java class on many different responses on many different inbaskets, configuring different parameters on each response and then checking for these values in the Java code so that the same class can act differently on different responses on different inbaskets.

All of this logic will all be executed after the user chooses a response, for which this class is configured, from the Action menu. Depending on whether `Pre` or `Post` is specified, the logic will be executed either before or after the normal case field validation is performed.

A simple use case for this kind of functionality would be a situation in which a certain response requires certain case fields to be populated with specific values. A Java class of this type could be used to ensure that the state of all of the values of all of the relevant case fields was correct for this type of response, displaying either a warning (which would optionally allow the user to continue) or raising an exception (which would prevent the user from completing the case altogether).

Consult the `\src\Java CaseEventHandlers Samples` folder in the code samples for more details.

Web custom tool development procedure

General introduction

BPF comes equipped with a number of tools designed for such things as creating a new case, attaching or adding documents to an existing case, checking queue depth (i.e. how many cases are in an inbasket) and the like. In addition to the tools provided, it is also possible to create custom tools. Custom tools use the same interface as the one used by the core tools provided with the BPF installation package. This means that any of the core tools can be used as a sample and a starting point for custom development.

In general, the BPF tools interface allows configuring and exposing a set of actions that extend BPF case functionality or provide additional features and functionality not provided with the core application.

Once developed and deployed (see below), a custom tool is configured in the BPF Explorer in two places:

1. A new tool node needs to be added and configured as described below under the Register Tool step.
2. Once the tool is configured at an application-wide level in this manner, it can then be configured on individual inbaskets in just the same manner as is done for core tools (described below under the Expose Tool step).

NOTE The new custom tool will only appear on those inbaskets for which it has been configured/exposed.

Developers can build their own tools and expose them on the Web Application UI to extend case functionality or provide interfaces to external systems. BPF allows configuring a URL that is executed whenever a user clicks the tool's link and provides management for the window displayed.

There are several tools provided with the source code that can be used as examples:

```
/samples/components/bpf/tools/add_attach
/samples/components/bpf/tools/add_document
/samples/components/bpf/tools/create_case
```

In addition, the sample code for the `my_tool` tool featured in the following screenshots is also provided in the code samples that accompany this guide.

The following sections are presented as a series of steps in order to make the logical process easier to follow. But in fact developing a custom tool is an iterative process that will require a number of passes through most or all of these steps before the tool is ready to deliver to the customer. To begin with, it is probably best to make a copy of the folder containing one of the existing core tools (or the `my_tool` sample) change the file, folder and other names to match those you would like to use with the tool you will be developing, bundle it, register it, and expose it on an Inbasket so that you can test and adjust your code from that inbasket within the web application as the development process progresses.

Step 1: Create code

The initial Handler URL for your tool can be either a servlet or a JSP page. JSP is usually a better choice here, because the user interface elements involved will probably require extensive adjustment and it is much easier to redeploy a JSP than a compiled Java Servlet.

Tools that simply display information (such as `queue_depth`) may consist of just a single source file. Other tools that allow the user to enter some information before performing some action will typically consist of an initial JSP page that then posts to a servlet when the user clicks the tool's **OK** button. The sample `my_tool` tool provided with this guide follows the first pattern.

As noted above, you will probably want to begin by picking the existing tool that most closely matches the one you would like to develop, make a copy of the relevant folders and files, and then at least get this copy installed, configured, and working from an Inbasket in the web application before you begin making changes to implement the custom functionality you are trying to achieve.

As you develop your code, remember to keep in mind the following useful points.

- The tool interface offers access to the following three sources of information.
  - As the screenshot of the `my_tool` user interface shown below demonstrates, tools are initially passed four querystring parameters:
    - Inbasket
    - ToolName
    - caseID
    - workobjectNumber
  - In addition, it is possible to use the reference to the main/parent BPF application window passed in through the `dialogArguments` variable as a way of obtaining access to all of the global JavaScript variables that are available to the rest of the web application.

The sample `my_tool` tool provided with this guide is designed to illustrate these features.

- Building a custom tool involves two main tasks:
  - The first is to make sure that you can obtain all of the information you need to perform whatever operations the tool will attempt to perform.
  - The second is to design and implement the user interface you would like your tool to have and to present to the user for display and/or interaction.

Once you have your code developed to a point where you would like to test, you will need to deploy and configure it. Please consult the `\src\Tool Samples` folder in the code samples for more details.

NOTE As delivered with the code samples, the sample `my_tool` tool does not actually do anything. It is written for demonstration purposes only.

Nevertheless, using the sample code provided, it should be possible to modify and add to the code contained in the sample `my_tool` tool so as to do virtually anything that might be required through this interface.

Step 2: Deploy code

Consult the `\src\Tool Samples` folder in the code samples for more details on how to deploy a simple JSP tool in BPF. If your tool uses a Java servlet as well, you will need to compile the source code and deploy it in a JAR or as a single `.class` file within the classpath for the deployed BPF Web Application so that the application server can find and load your code.

Step 3: Register tool

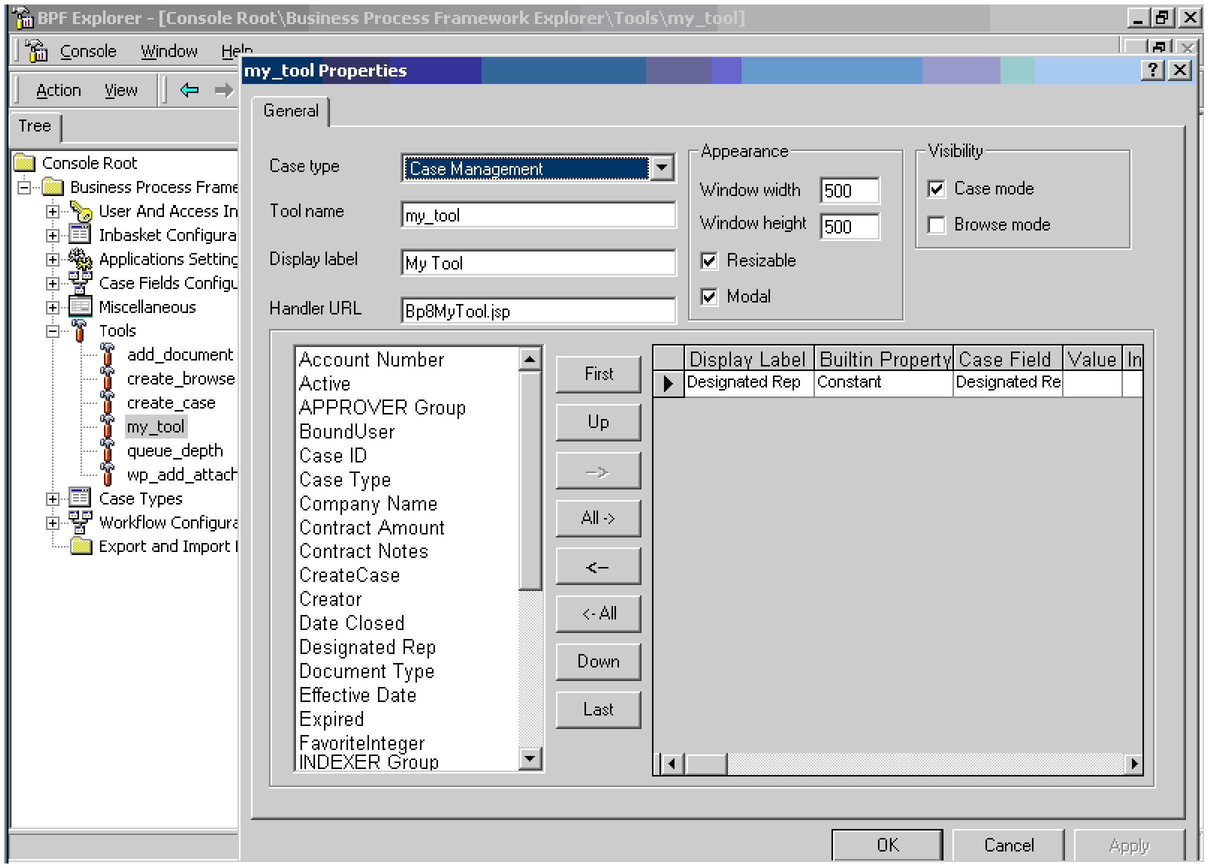
Register the tool in BPF Explorer using **Tools** > **Right-Click** > **New**.

Item	Description
Case type	This is applicable only if you will be retrieving case field configuration exposed for this tool and allows having different sets of case fields for each case type.
Tool name	Name should match the name of the folder under `/plugins/tools` in which you deployed your tool.
Display label	Tool name label that you would like to display on the Web user interface.

Handler URL	Fully qualified or relative path to the tool JSP page.
	
	For example, if the JSP is external to BPF (deployed in another application), the URL would look something like this:
	
	`http://<someotherserver>:<port>/app/PageName.jsp`
	
	If it is deployed in the root folder of the BPF Web Application, it would look like this:
	
	`/PageName.jsp`
	
	And if it is deployed in the standard deployment folder (in a <Tool name> folder under `/plugins/tools`), it would look like this:
	
	`PageName.jsp`

Window width	The desired/required window width for your tool.
Resizable	Determines whether the tool window can be resized.
Modal	Determines whether the tool window is locked and prevents access to other computer elements while the tool window is open.
Visibility	Determines whether the tool is visible in Case mode, Browse mode, or both.

See the following screenshot for an example of how this would look using the sample `my_tool` tool provided with this guide.



Case type: Case Management
Tool name: my_tool
Display label: My Tool
Handler URL: Bp8MyTool.jsp

Appearance
Window width: 500
Window height: 500
☑ Resizable
☑ Modal

Visibility
☑ Case mode
☐ Browse mode

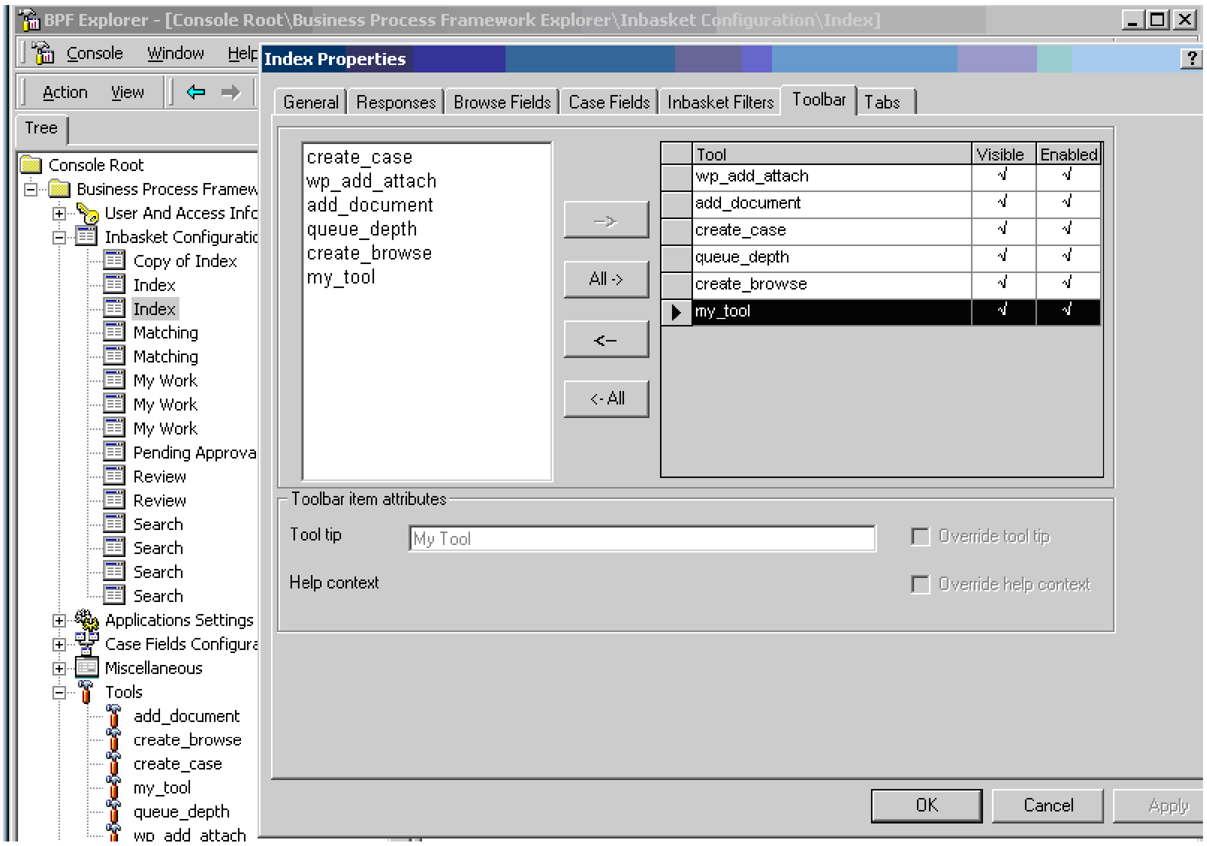
Account Number
Active
APPROVER Group
BoundUser
Case ID
Case Type
Company Name
Contract Amount
Contract Notes
CreateCase
Creator
Date Closed
Designated Rep
Document Type
Effective Date
Expired
FavoriteInteger
INDEXER Group

Display Label	Builtin Property	Case Field	Value	In
Designated Rep	Constant	Designated Re		

Step 4: Expose tool

Expose your custom tool in BPF Explorer, using the Inbasket configuration Toolbar tab to expose the tool and configure its properties for this Inbasket.

See the following screenshot for an example of how this would look using the sample `my_tool` tool provided with this guide.



create_case
wp_add_attach
add_document
queue_depth
create_browse
my_tool

Tool	Visible	Enabled
wp_add_attach	√	√
add_document	√	√
create_case	√	√
queue_depth	√	√
create_browse	√	√
my_tool	√	√

Toolbar item attributes

Tool tip: My Tool

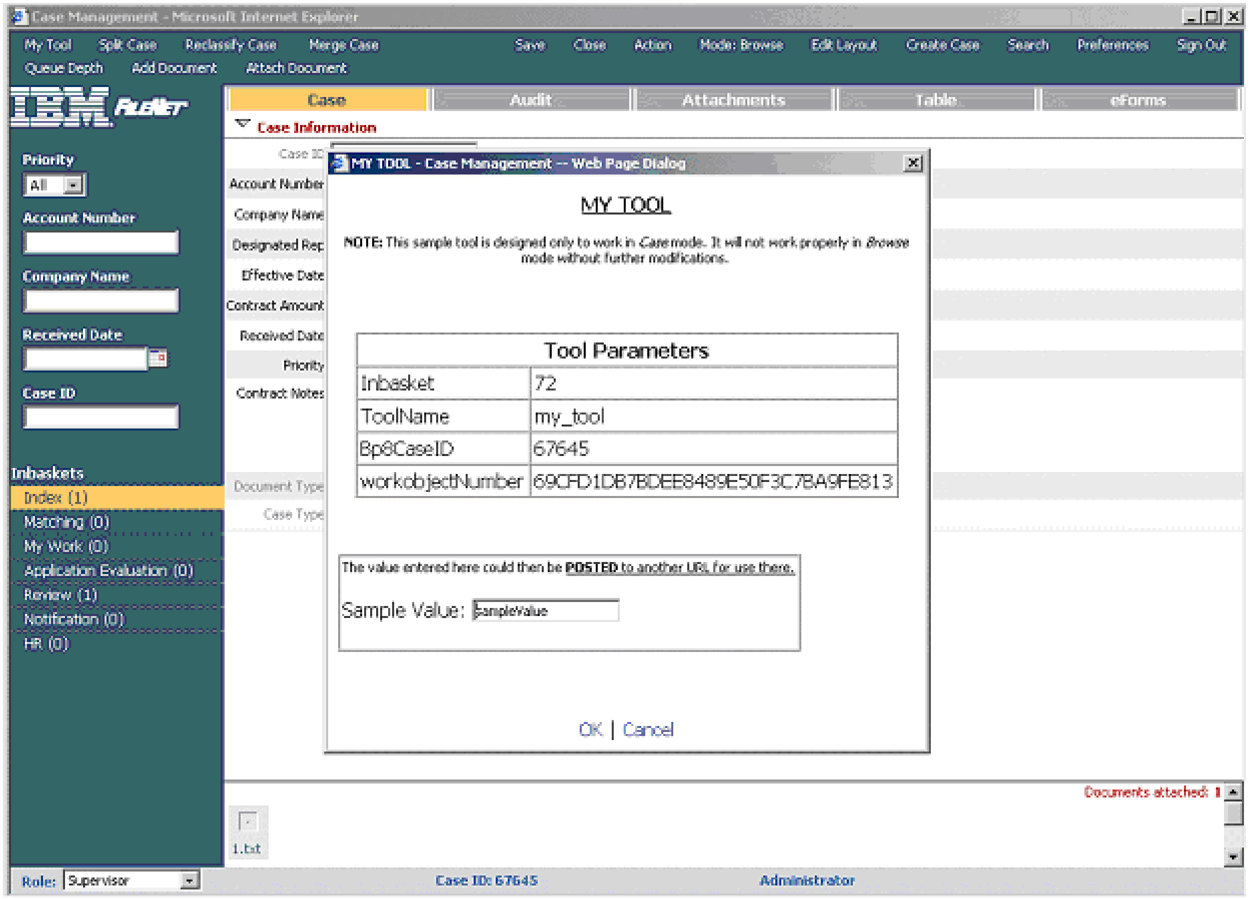
☐ Override tool tip

Help context

☐ Override help context

The result of these configuration settings can be seen in the following screenshot. As this screenshot demonstrates, tools are displayed as HTML links in the toolbar and are configured to be displayed or hidden on the individual Inbasket-level. Tools can be configured to display in either the Case or Browse mode or both.

Once these steps have been completed, you should be able to launch the tool from the Inbasket on which you configured/exposed it, as seen in the following screenshot.



MY TOOL

NOTE: This sample tool is designed only to work in *Case* mode. It will not work properly in *Browse* mode without further modifications.

Tool Parameters	
Inbasket	72
ToolName	my_tool
Bp8CaseID	67645
workobjectNumber	69CFD1DB7BDEE8489E50F3C7BA9FE813

The value entered here could then be POSTED to another URL for use there.

Sample Value: SampleValue

Sample my_tool code

Consult the `\src\ Tool Samples` folder in the code samples for more details on how to implement the custom tool sample.

Web custom tab development procedure

Introduction

BPF case data is presented to users in a tabbed interface. By default, the BPF Web Application presents each case in the form of two tabs: Case and Audit. Like all tabs, these two default tabs are configured for each inbasket using the BPF Explorer. But, unlike the other (custom) tabs discussed here, the Case and Audit tabs are integral to BPF and do not implement the custom tab interface discussed in this document.

In contrast to these default tabs, the ability to extend the BPF interface with custom tabs is exemplified by the Attachments and Table tabs that are also provided with the install package. The Attachments tab is used to display Case attachments in the form of an HTML table and the Table tab is designed to provide an easily customizable framework for presenting data in a tabular form within the BPF UI.

In addition, this guide comes with the sample code for a very simple, generic tab called My Tab, which is used as an example in what follows here.

NOTES

- Unlike the Attachments tab, whose JSP pages make use of compiled Java classes developed especially for BPF, the My Tab sample provided here consists of nothing more than a simple custom JSP page (see the `\src\Tab Samples` directory in the code samples) that displays a simple message and loads some JavaScript (commented out by default) for demonstrating just what information is available to client-side JavaScript on a custom tab.
- It is possible to have a tab point to an external URL by entering **the fully qualified absolute URL** into the TAB_CONTENT column of the TABS table (see below under Step 3: Register Tab). However, security limitations with IFrame will result in Access Denied errors if the external URL belongs to a different domain.
- In order to facilitate the HotKeys functionality associated with BPF Actions during the time the user has your custom tab displayed, it will be necessary for you to include the Bp8InitMain.js file using code like the following

  ```
  <script type="text/javascript" src="js/Bp8InitMain.js"></script>
  ```

  to the UI JSP (or equivalent) of your custom tab and then include an

  ```
  onkeydown="doBodyKeyDown()"
  ```

  event in the `BODY` tag of the UI JSP (or equivalent) for your custom tab as well. Refer to the UI TableTab.jsp for an example of this. Failure to do so will result in an absence of HotKeys functionality for BPF Actions while your custom tab is displayed.

Step 1: Create code

Create your JSP page, Java Servlet, hard-coded HTML page, external website or other source of HTML for display in your custom tab.

Step 2: Bundle code

Assuming your tab is a simple JSP page along the lines of the `Bp8MyTab.jsp` sample, create a new subdirectory under `<bpf_web_root>\plugins\tabs`, such as, for example, `\my_tab`. Note that, when you register your tab in the `TABS` table in the `Bp8Metastore` database, the name of this new subdirectory for your tab **must** be entered in the `TAB_NAME` column. The `TAB_NAME` column value for your tool **is** the name of this subdirectory under `<bpf_web_root>\plugins\tabs` where your JSP page (whose name and querystring are defined in the `TAB_CONTENT` column) will reside. At runtime, BPF will concatenate the URL as follows and look for your tab in the following location:

```
<bpf_web_root>/plugins/tabs/ + TAB_NAME + / + TAB_CONTENT
```

As is noted again below under Step 3, failure to appreciate this fact will lead to HTTP 404 errors in the browser when BPF cannot find the page for your custom tab.

Step 3: Register Tab

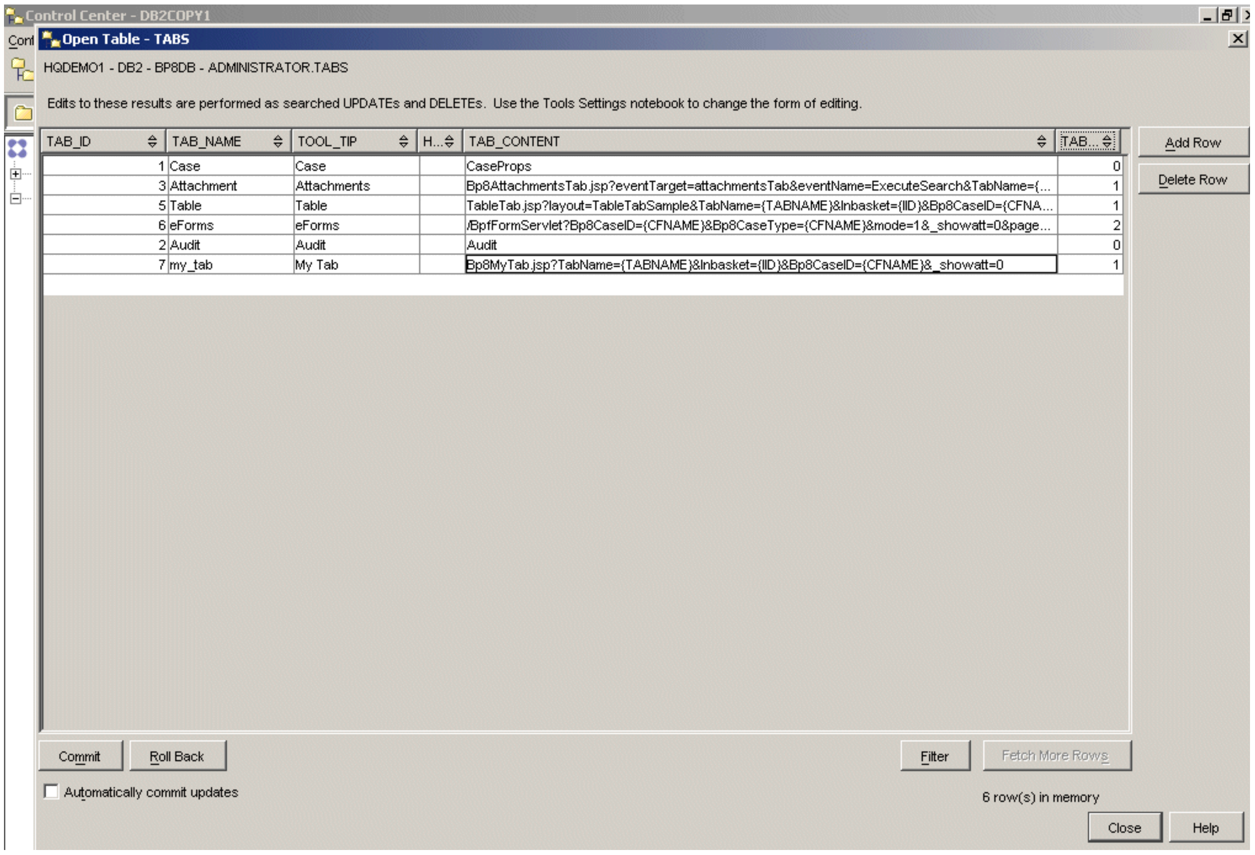
Register tab in the TABS table in Bp8Metastore:

TAB_ID	<next available>
TAB_NAME	The name of the subdirectory under `<bpf_web_root>\plugins\tabs` where you have deployed your JSP or the equivalent.
TOOL_TIP	Tab name label that you would like to display on the Web user interface.
HELP_CTX	<NULL>
TAB_CONTENT	<jsp_page_name>?<querystring parameters and values>

NOTE Refer to the documentation for the `<bpf_help_root>\bpfwa_help\integration\tab_interface.htm` and `<bpf_help_root>\bpfwa_help\plug-ins\tab_table.htm` files for examples and information.

When you are done, the row for your new tab in the TABS table should look something like the following example.

*



HQDEMO1 - DB2 - BP8DB - ADMINISTRATOR.TABS

Edits to these results are performed as searched UPDATEs and DELETEs. Use the Tools Settings notebook to change the form of editing.

TAB_ID	TAB_NAME	TOOL_TIP	H...	TAB_CONTENT	TAB...
1	Case	Case		CaseProps	0
3	Attachment	Attachments		Bp8AttachmentsTab.jsp?eventTarget=attachmentsTab&eventName=ExecuteSearch&TabName={...	1
5	Table	Table		TableTab.jsp?layout=TableTabSample&TabName={TABNAME}&Inbasket={IID}&Bp8CaseID={CFNA...	1
6	eForms	eForms		/BpfFormServlet?Bp8CaseID={CFNAME}&Bp8CaseType={CFNAME}&mode=1&_showatt=0&page...	2
2	Audit	Audit		Audit	0
7	my_tab	My Tab		Bp8MyTab.jsp?TabName={TABNAME}&Inbasket={IID}&Bp8CaseID={CFNAME}&_showatt=0	1

☐ Automatically commit updates

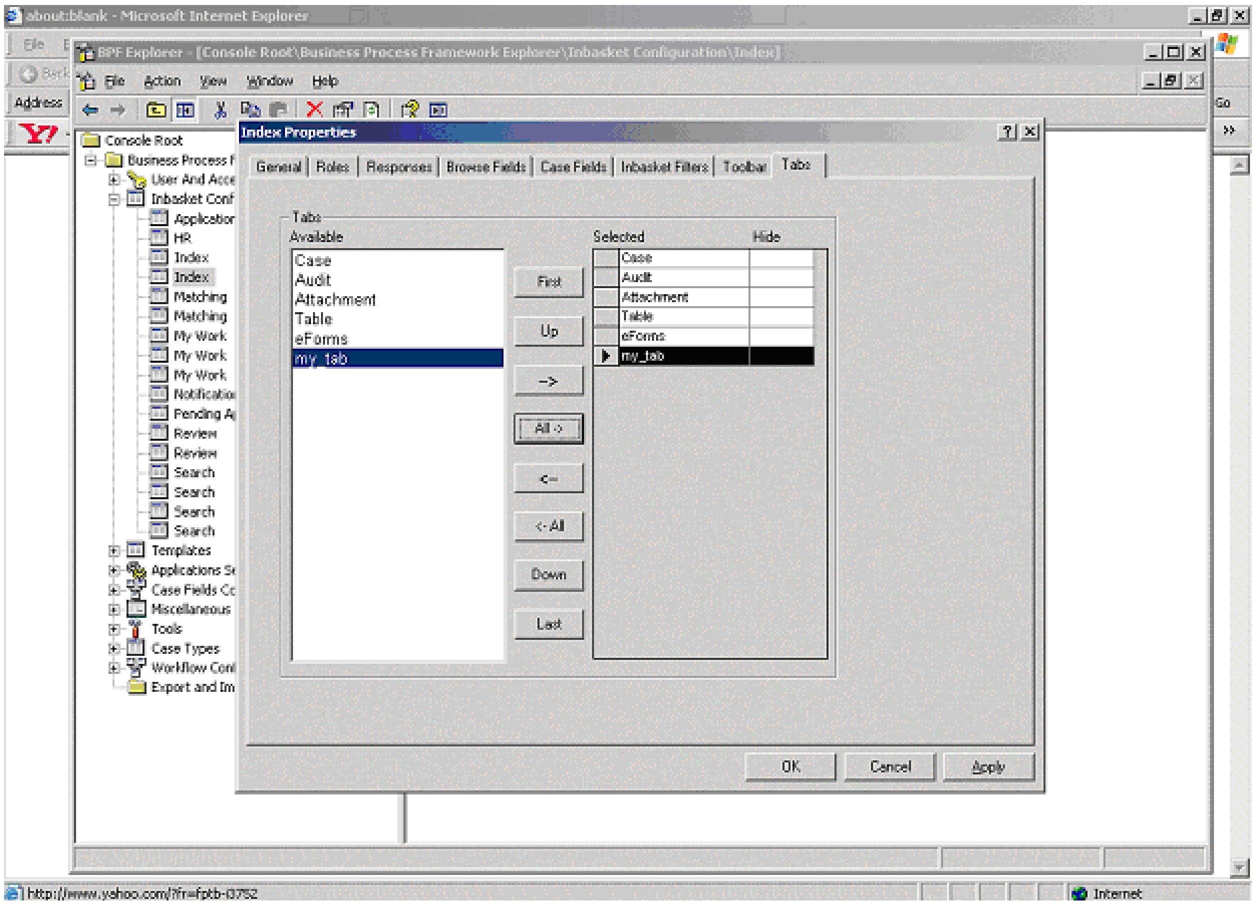
6 row(s) in memory

NOTES

- The TAB_NAME value is actually the name of the physical directory inside the `<web_root>\plugins\tabs` directory where the JSP files for this tab reside. There is no other way to configure this path. If the names do not match, opening the new tab will result in a 404 error in the browser.
- The TOOL_TIP value is in fact the string that will appear in the Web Application user interface for the link that opens this tab.

Step 4: Expose tab

Expose your tab using BPF Explorer. Use the inbasket configuration Tabs tab to expose the tab and configure the order in which it and the other tabs configured for this inbasket will appear.

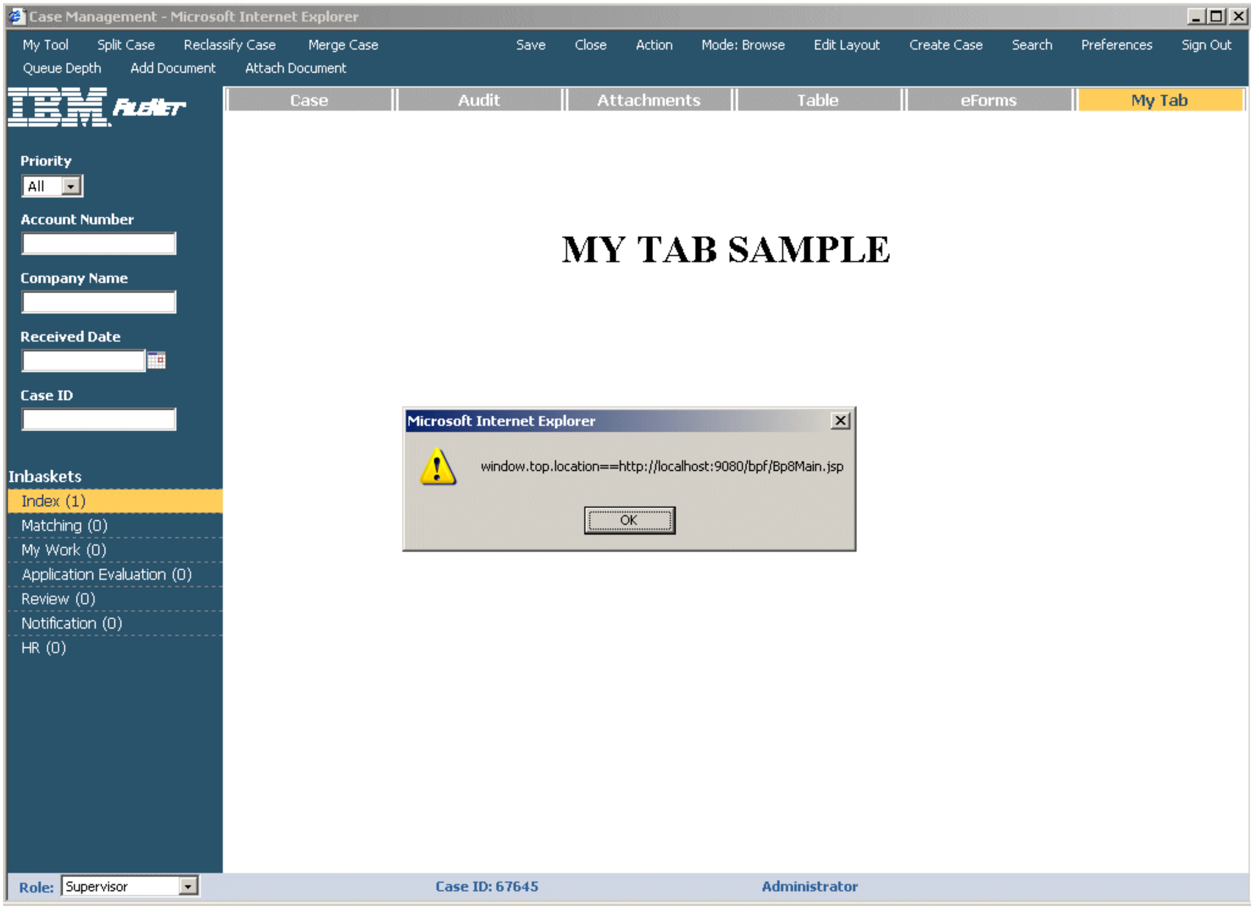


Tabs

Available:
- Case
- Audit
- Attachment
- Table
- eForms
- my_tab

Selected	Hide
Case	
Audit	
Attachment	
Table	
eForms	
my_tab	

The end result of installing and configuring the My Tab sample tab can be seen in the following screenshot of the tab when it is opened:



MY TAB SAMPLE

Microsoft Internet Explorer

window.top.location==http://localhost:9080/bpf/Bp8Main.jsp

Role: Supervisor

Case ID: 67645

Administrator

Sample my_tab code

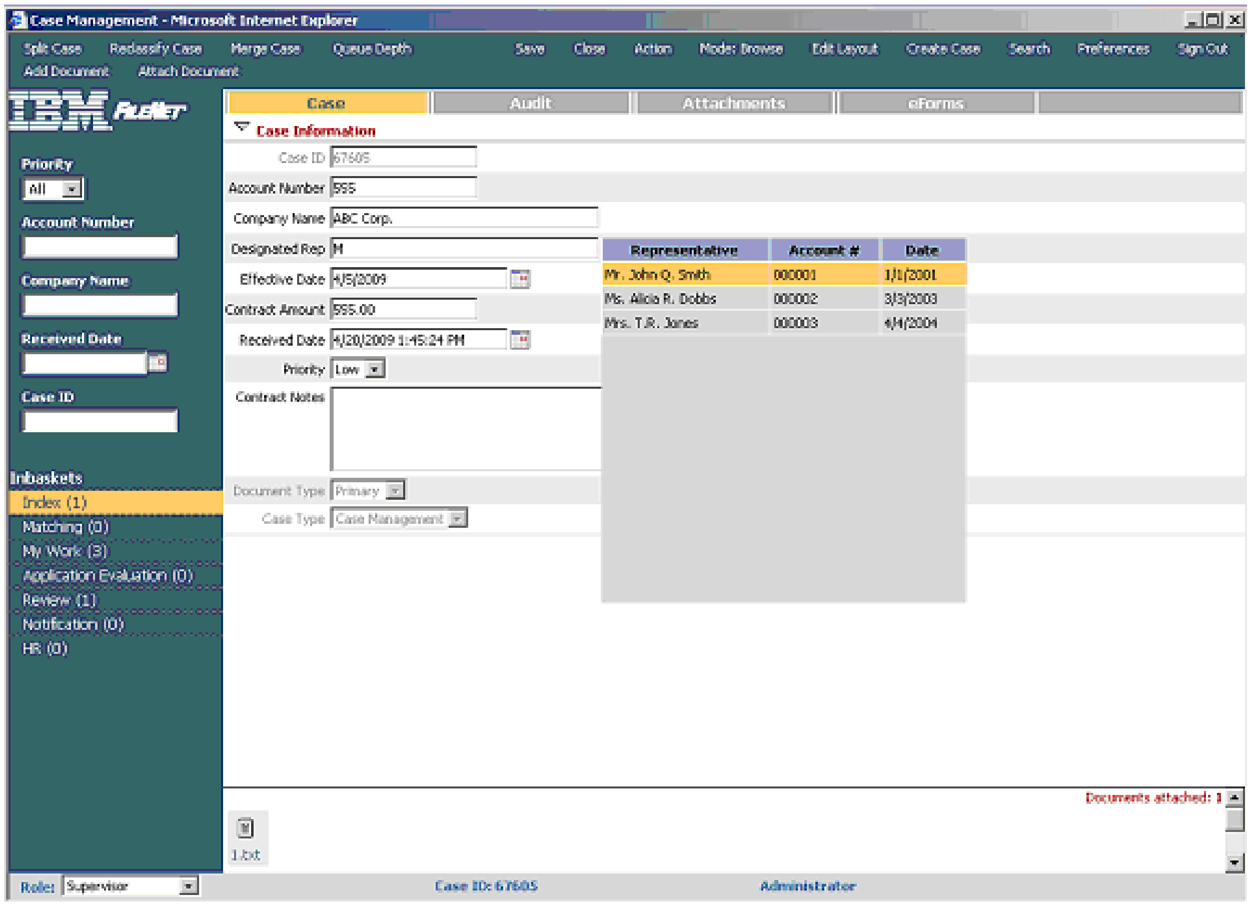
Consult the `\src\ Tab Samples` folder in the code samples for more details on how to implement the custom tab sample.

Lookup extensions interface

Introduction

BPF provides a Lookup Extensions interface that allows custom lookup capability to be provided for individual Case fields in such a way that one of three possible results will always occur for a correctly configured lookup.

- If more than one matching record is found, a popup will be displayed showing the fields and values for each record in a tabular form. When the user clicks on one of the rows in this popup table, the configured case field(s) will be updated with the selected value(s).



Priority: All

Case ID	67605
Account Number	555
Company Name	ABC Corp.
Designated Rep	M
Effective Date	4/5/2009
Contract Amount	555.00
Received Date	4/20/2009 1:45:24 PM
Priority	Low
Contract Notes	
Document Type	Primary
Case Type	Case Management

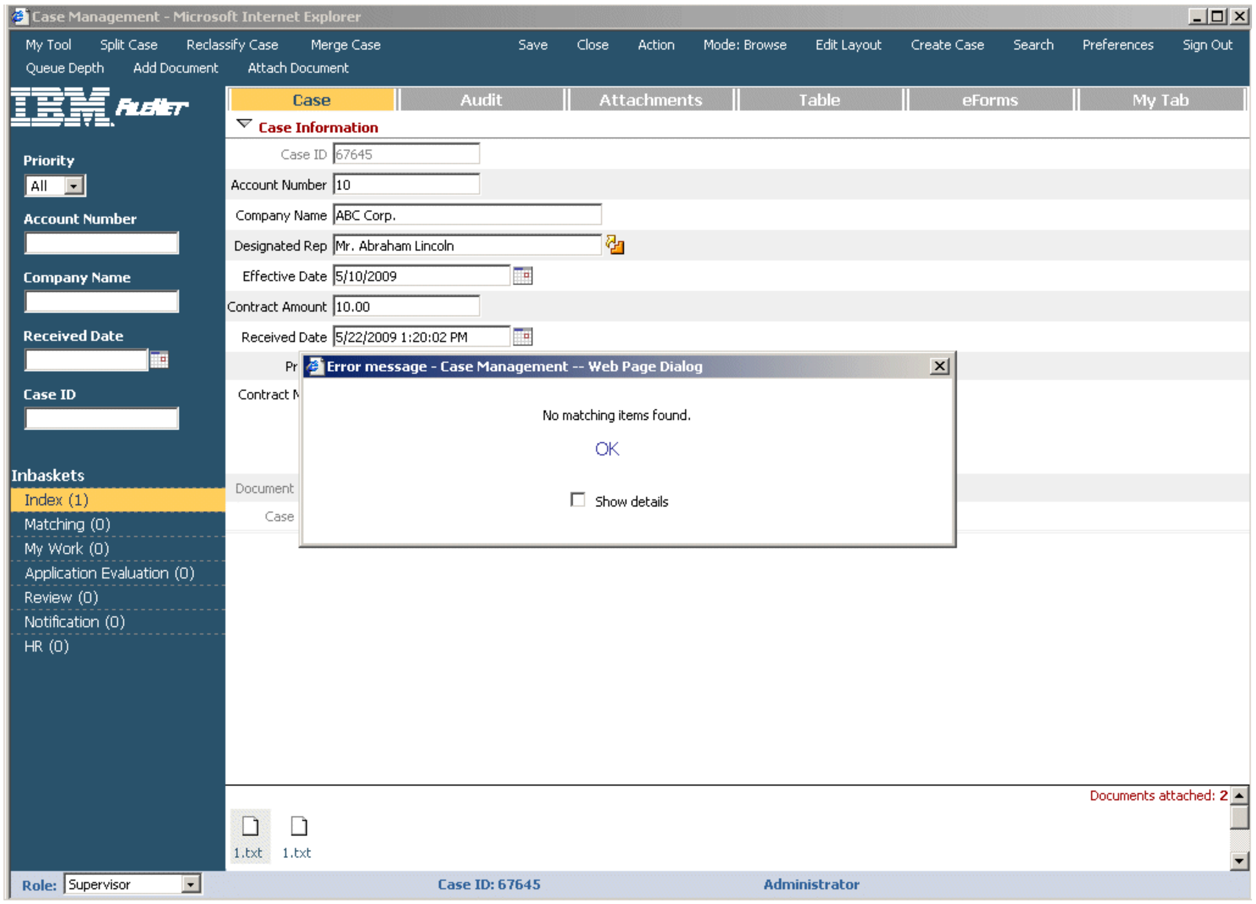
Representative	Account #	Date
Mr. John Q. Smith	000001	1/1/2001
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004

Documents attached: 1

Role: Supervisor Case ID: 67605 Administrator

If only one matching record is found, the configured Case field(s) will automatically be updated with the value(s) returned and no list of options will be presented as for multiple matches.

- If no matching records are found, an alert will be presented indicating this fact.



Case Information

Case ID	67645
Account Number	10
Company Name	ABC Corp.
Designated Rep	Mr. Abraham Lincoln
Effective Date	5/10/2009
Contract Amount	10.00
Received Date	5/22/2009 1:20:02 PM

Priority: All

Error message - Case Management -- Web Page Dialog

No matching items found.

Role: Supervisor   Case ID: 67645   Administrator

This functionality is typically (but not necessarily) provided by a custom JSP page that needs to be written and configured as the Lookup Service URL for individual Case fields. Other options might include a servlet or a specially written web service. Such JSP pages (for example) must return XML, rather than HTML, and specifically they must build XML according to the following schema (with the following example using some of the same values as those found in the `DesignatedRepLookup.jsp.SIMPLE` file contained in the sample code for this feature):

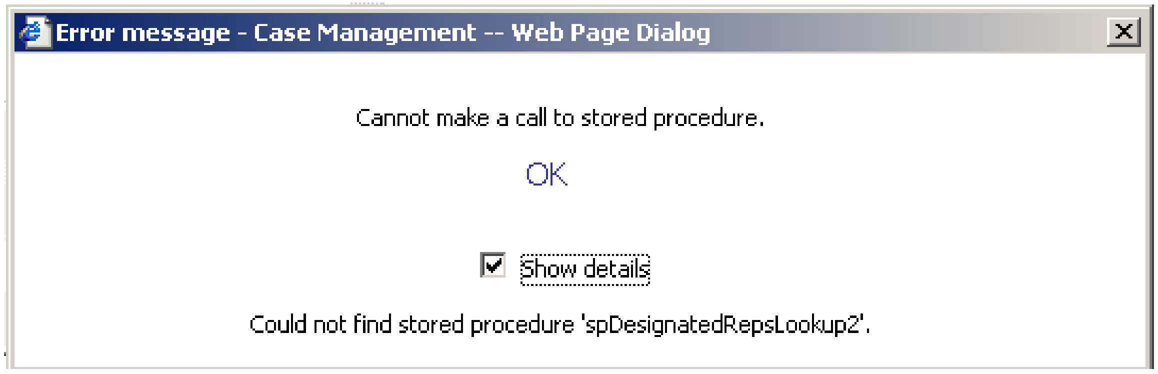
```
<Response>
 <Items LimitExceeded="0">
  <Header>
   <Value>Representative</Value>
   <Value>Account #</Value>
   <Value>Date</Value>
  </Header>
  <Item>
   <Value Field="DesignatedRep" Name="Mr. A.B. Rhett">Mr. A.B. Rhett</Value>
   <Value Field="AccountNumber" Name="000001">000001</Value>
   <Value Field="EffectiveDate" Name="2/2/2002">2/2/2002</Value>
  </Item>
 <Item>
  <Value Field="DesignatedRep" Name="Ms. Sally Brown">Ms. Sally Brown</Value>
  <Value Field="AccountNumber" Name="000002">000002</Value>
  <Value Field="EffectiveDate" Name="000002">5/5/2005</Value>
  </Item>
 </Items>
</Response>
```

The XML from the sample code above illustrates a normal, error-free result containing two possible matches based on the submitted.

It is also possible to return error or exception information instead of items by passing a `Response` element with an `Exception` node instead of an `Items` node, as in the following example (from `DesignatedRepLookup.jsp.SIMPLE`:

```
<Response>
 <Exception>
  <Header key="lookupException" mode="info">
   <Value>No matching items found.</Value>
  </Header>
  <Description>
   <Value>
To get any matching items returned with this code sample, enter only 1 character (which
must be 'M') while the CompanyName value is either 'ABC Corp.' or 'XYZ Corp.'.
Otherwise, no matching items will be returned.  NOTE: This is just a sample and not a
fully working lookup.
   </Value>
  </Description>
 </Exception>
</Response>
```

For example, if the custom code attempts to make a call to a stored procedure and fails due to the fact that the name of the store procedure has been misspelled in the custom code or the stored procedure does not exist in the database, it is possible to return this information to the client by using the Exception node to produce the following display:


Error message - Case Management -- Web Page Dialog

Cannot make a call to stored procedure.

OK

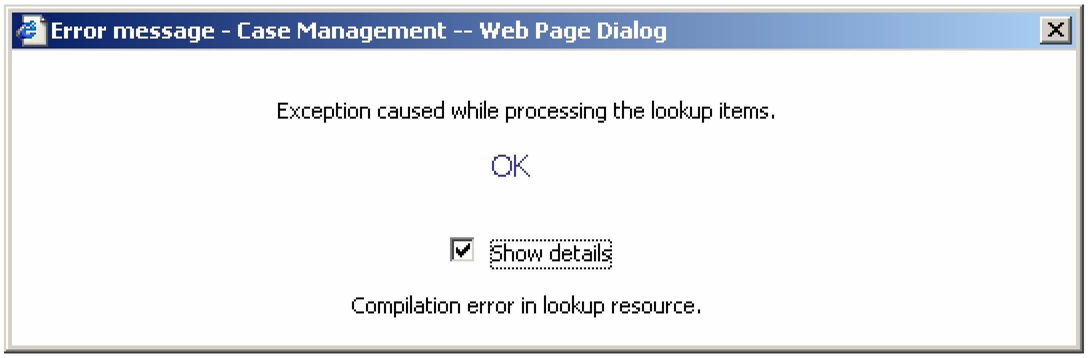
Show details

Could not find stored procedure 'spDesignatedRepsLookup2'.

Assuming the custom code for the lookup is in Java, this would be done by building the Exception node inside a catch block and putting all of the relevant code the handles the normal processing of the lookup inside a corresponding try block (see below under **Writing the Code**).

Note that BPF itself will use this feature to display XML parsing error information if the XML Response from the custom code contains invalid XML. This is illustrated by the following examples.

- If there is any compilation error in the custom page, the BPF Web Application displays a compilation error message:


Error message - Case Management -- Web Page Dialog

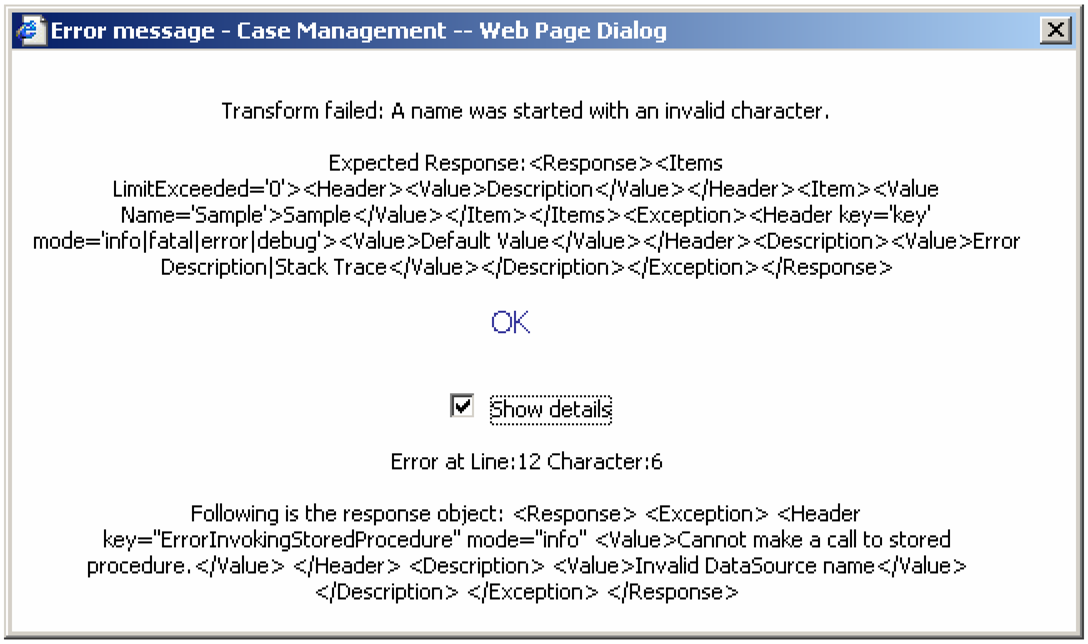
Exception caused while processing the lookup items.

OK

Show details

Compilation error in lookup resource.

- When the Response object that is returned by the custom code is not well-formed, the BPF Web Application displays an appropriate message:



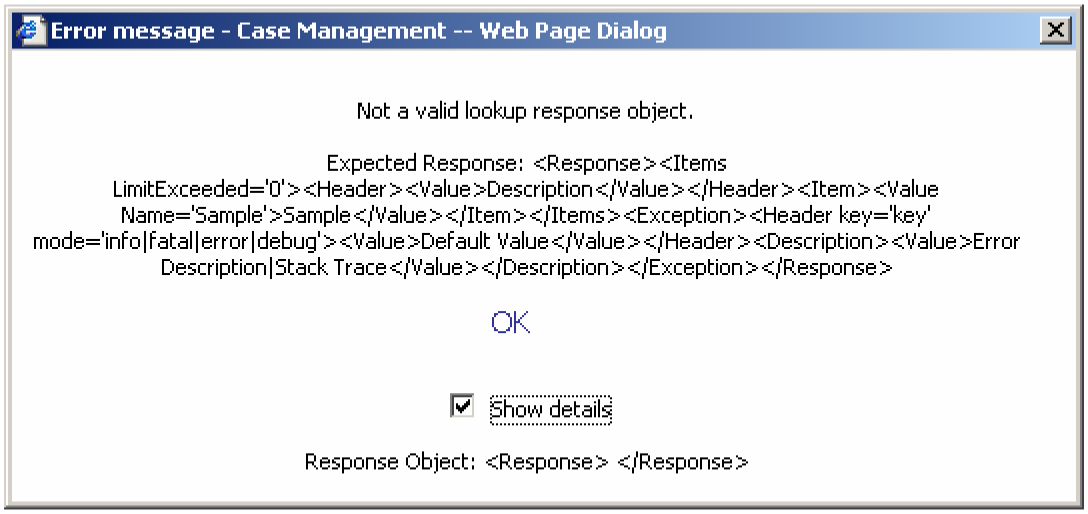
Transform failed: A name was started with an invalid character.

Expected Response: <Response><Items LimitExceeded='0'><Header><Value>Description</Value></Header><Item><Value Name='Sample'>Sample</Value></Item></Items><Exception><Header key='key' mode='info|fatal|error|debug'><Value>Default Value</Value></Header><Description><Value>Error Description|Stack Trace</Value></Description></Exception></Response>

Error at Line:12 Character:6

Following is the response object: <Response> <Exception> <Header key="ErrorInvokingStoredProcedure" mode="info" <Value>Cannot make a call to stored procedure.</Value> </Header> <Description> <Value>Invalid DataSource name</Value> </Description> </Exception> </Response>

- When the Response object that is returned is constructed with no Items **and** no Exception node in the below format (in other words, just and empty "<Response></Response>"), the BPF Web Application displays the following message:

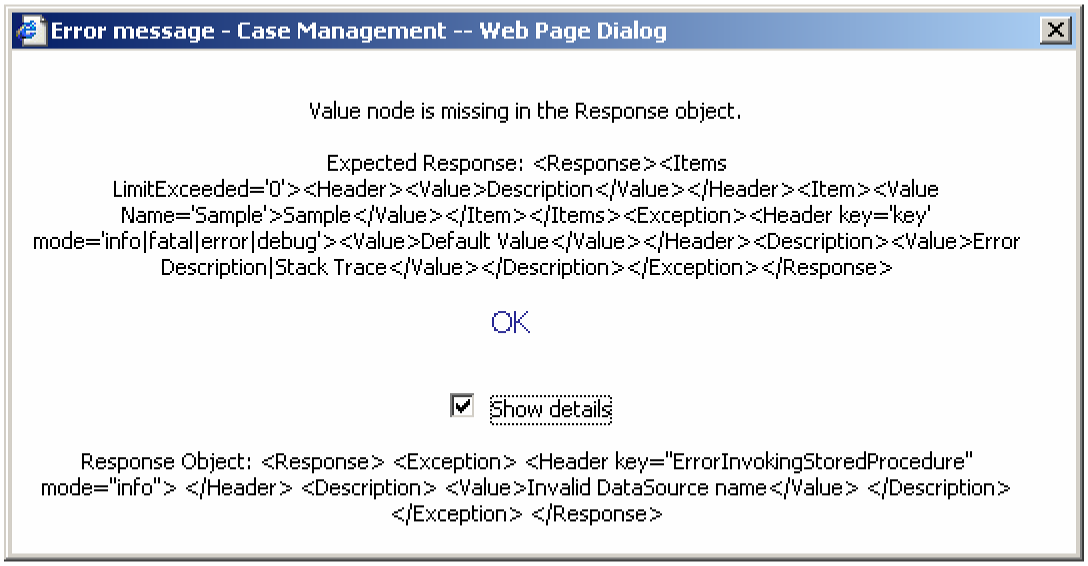


Not a valid lookup response object.

Expected Response: <Response><Items LimitExceeded='0'><Header><Value>Description</Value></Header><Item><Value Name='Sample'>Sample</Value></Item></Items><Exception><Header key='key' mode='info|fatal|error|debug'><Value>Default Value</Value></Header><Description><Value>Error Description|Stack Trace</Value></Description></Exception></Response>

Response Object: <Response> </Response>

- If the Value subnode is missing inside the Header node of the Exception tag, the BPF Web Application displays the following message:

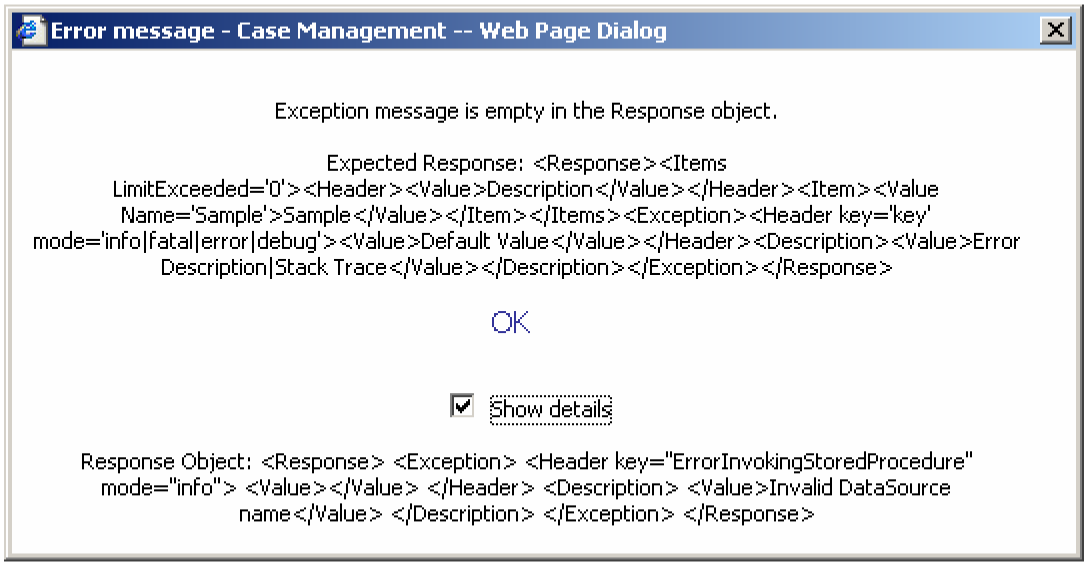


Value node is missing in the Response object.

Expected Response: <Response><Items LimitExceeded='0'><Header><Value>Description</Value></Header><Item><Value Name='Sample'>Sample</Value></Item></Items><Exception><Header key='key' mode='info|fatal|error|debug'><Value>Default Value</Value></Header><Description><Value>Error Description|Stack Trace</Value></Description></Exception></Response>

Response Object: <Response> <Exception> <Header key="ErrorInvokingStoredProcedure" mode="info"> </Header> <Description> <Value>Invalid DataSource name</Value> </Description> </Exception> </Response>

- If the Value node value is empty inside the Header node of the Exception tag, the BPF Web Application displays the following message:

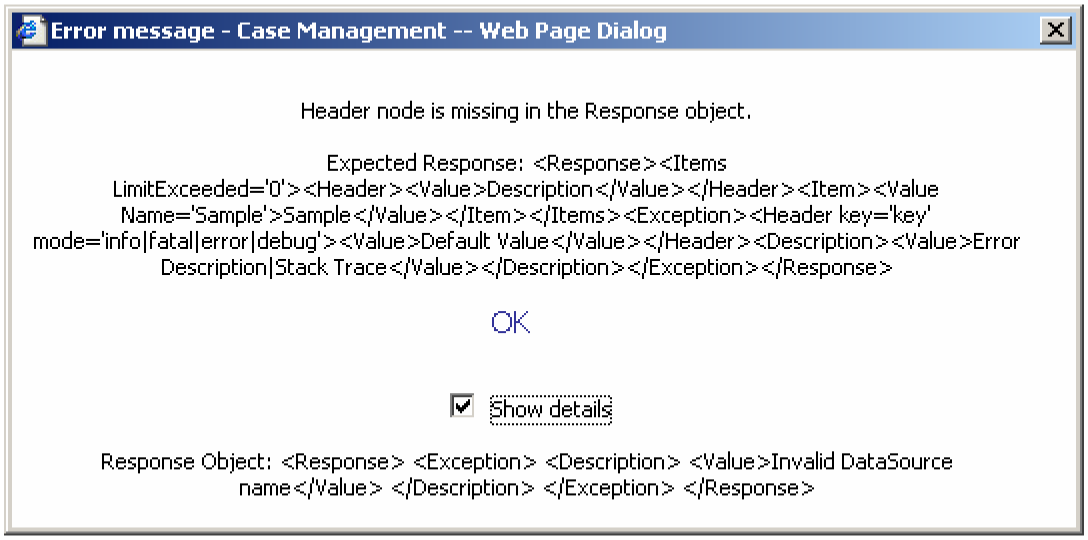


Exception message is empty in the Response object.

Expected Response: <Response><Items LimitExceeded='0'><Header><Value>Description</Value></Header><Item><Value Name='Sample'>Sample</Value></Item></Items><Exception><Header key='key' mode='info|fatal|error|debug'><Value>Default Value</Value></Header><Description><Value>Error Description|Stack Trace</Value></Description></Exception></Response>

Response Object: <Response> <Exception> <Header key="ErrorInvokingStoredProcedure" mode="info"> <Value></Value> </Header> <Description> <Value>Invalid DataSource name</Value> </Description> </Exception> </Response>

- If there is no Header node inside the Exception tag, the BPF Web Application displays the following message:



Error message - Case Management -- Web Page Dialog

Header node is missing in the Response object.

Expected Response: <Response><Items LimitExceeded='0'><Header><Value>Description</Value></Header><Item><Value Name='Sample'>Sample</Value></Item></Items><Exception><Header key='key' mode='info|fatal|error|debug'><Value>Default Value</Value></Header><Description><Value>Error Description|Stack Trace</Value></Description></Exception></Response>

OK

☑ Show details

Response Object: <Response> <Exception> <Description> <Value>Invalid DataSource name</Value> </Description> </Exception> </Response>

In short, every valid XML `Response` element must therefore contain **either** (1) an `Items` node with at least one `Header` and a corresponding Item **or** (2) an `Exception` node with a `Header` and `Description`, according to the following patterns: In other words, the `Response` XML should return either

```
<Response>
 <Items LimitExceeded="0">
  <Header>
   <Value>Description</Value>
  </Header>
  <Item>
   <Value Field="CaseField" Name="Sample">Sample</Value>
  </Item>
 </Items>
</Response>
```

or

```
<Response>
 <Exception>
  <Header key="localization key">
   <Value>Default Value</Value>
  </Header>
  <Description>
   <Value>Error Description | Stack Trace</Value>
  </Description>
 </Exception>
</Response>
```

But it should not return both.

As long as the JSP page configured as the Lookup Service URL for a given case field returns an XML stream matching this schema, it can draw the information itself from virtually any source, whether by redirecting to a web service, calling a servlet, or making a direct database query using JDBC.

The values of the `Value` children of the `Header` tag will appear as column headers in the results popup returned by the lookup (assuming more than one match is found).

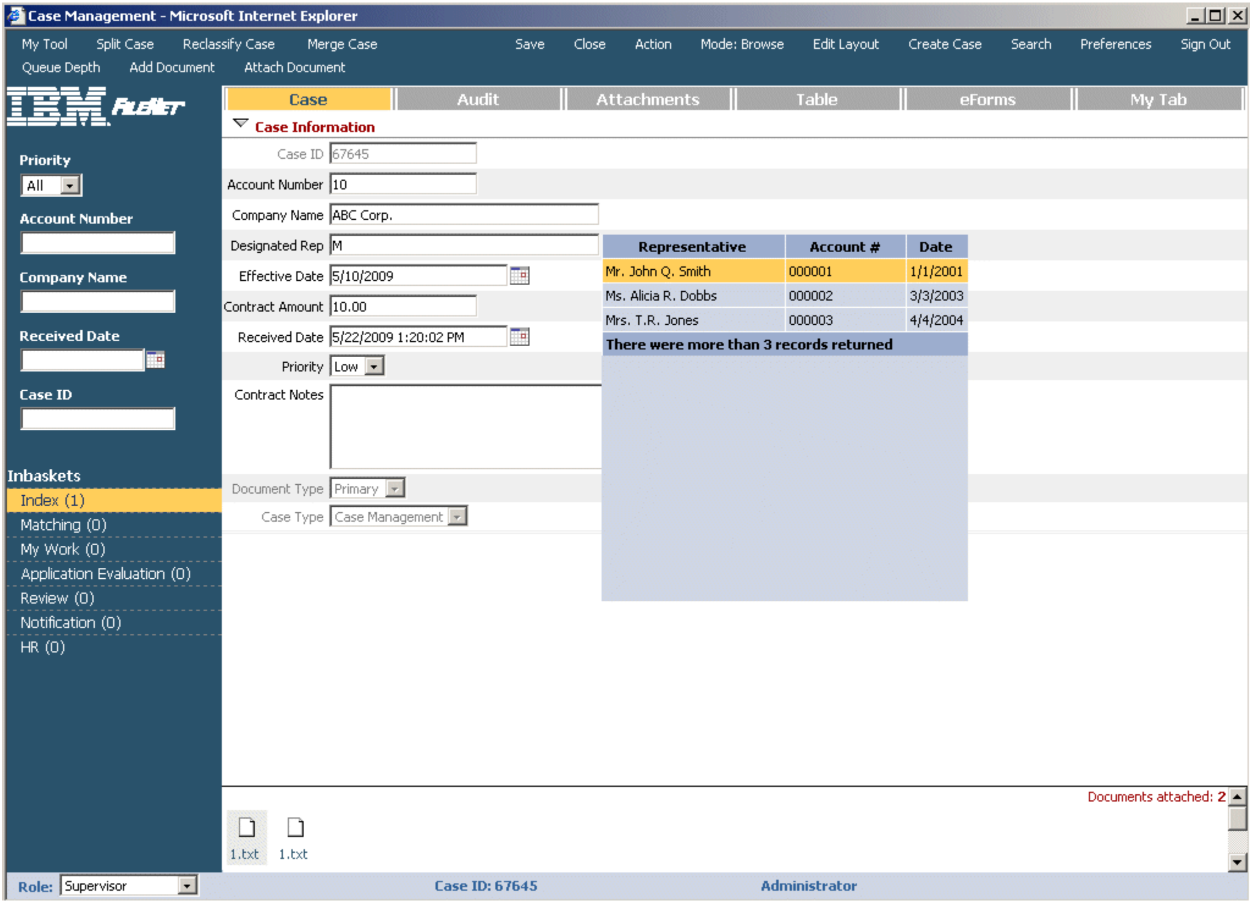
The values of the `Value` children of the `Item` tags are used to indicate the values returned by the lookup for each field, the name of the CaseField to be updated with the Item's value being indicated by the `Field` attribute for each record (`Item`). In this way, a single lookup based on a single case field can update multiple fields in the BPF user interface.

By the same token, it is possible to configure the Lookup Service URL for a single case field's lookup in such a way that the values of multiple BPF case fields are passed into the lookup search as dynamic parameters. The notation used for this is simply the name of the BPF Case field surrounded by percentage (%) signs, as in the following example:

```
plugins/custom/lookups/DesignatedRepLookup.jsp?designatedRep=%DesignatedRep%&companyName=%Com
panyName%
```

This configuration string will automatically pass the current values of the DesignatedRep and CompanyName case fields as JSP parameters named designatedRep and companyName, respectively.

In implementing your custom lookup JSP page it may be useful to limit the number of records that can be presented to the browser client (for performance reasons) while at the same time notifying the user that more records were returned by the actual lookup query. This can be done by returning only a fixed number of records and setting the <Items LimitExceeded=0> tag's LimitExceeded value to 1. The immediate effect of this (using one of the sample `DesignatedRepLookup.jsp` pages provided) is seen in the following screenshot, where the warning, "There were more than 3 records returned", is automatically displayed to the user.



Priority: All
Account Number:
Company Name:
Received Date:
Case ID:

Case Information

Case ID	67645
Account Number	10
Company Name	ABC Corp.
Designated Rep	M
Effective Date	5/10/2009
Contract Amount	10.00
Received Date	5/22/2009 1:20:02 PM
Priority	Low
Contract Notes	
Document Type	Primary
Case Type	Case Management

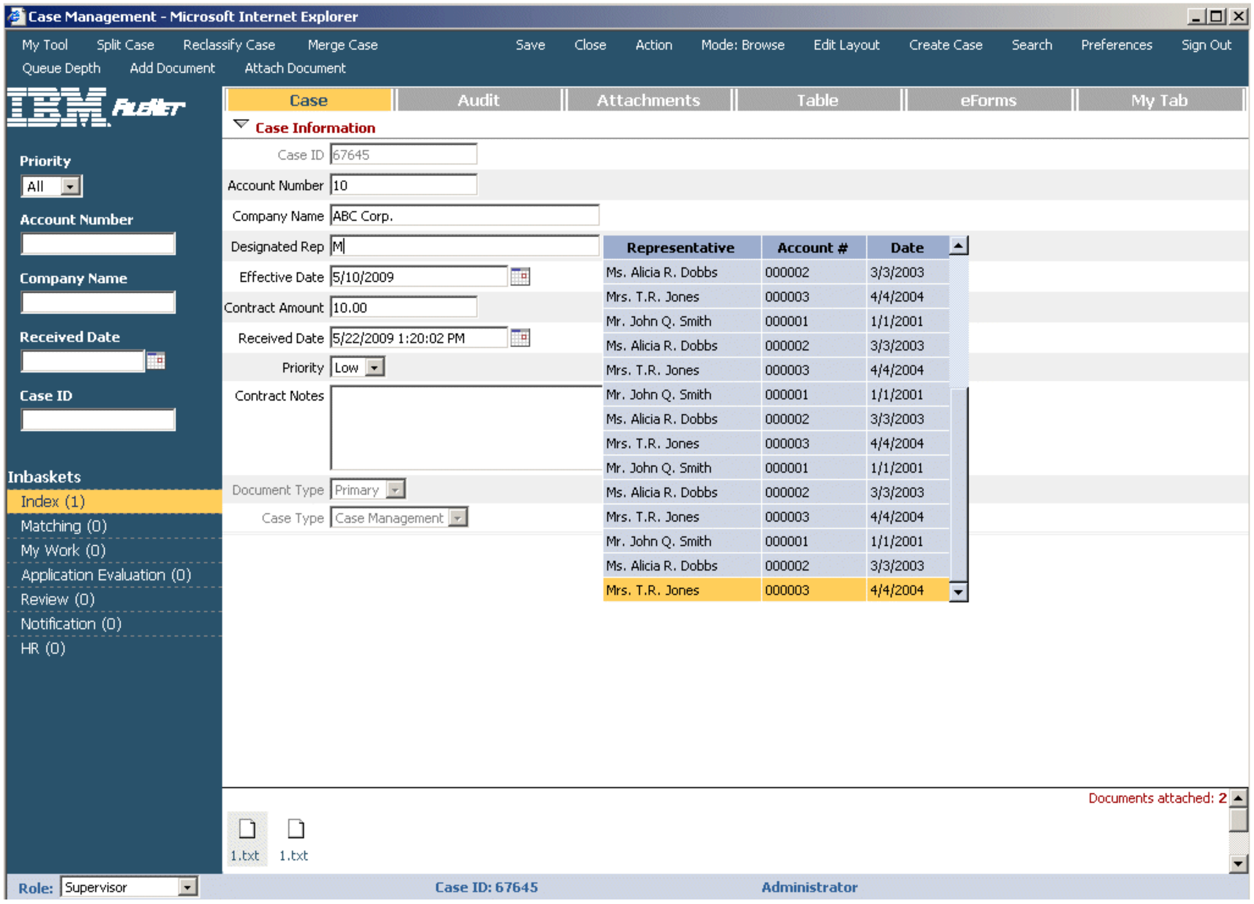
Representative	Account #	Date
Mr. John Q. Smith	000001	1/1/2001
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004

There were more than 3 records returned

Documents attached: 2

Role: Supervisor   Case ID: 67645   Administrator

Lastly, if your lookup query returns more records or columns than can be presented in the popup window seen in the screenshots above, vertical and/or horizontal scrollbars are automatically provided - as in the screenshot below:



Priority: All
Account Number:
Company Name:
Received Date:
Case ID:

Case Information

Case ID	67645
Account Number	10
Company Name	ABC Corp.
Designated Rep	M
Effective Date	5/10/2009
Contract Amount	10.00
Received Date	5/22/2009 1:20:02 PM
Priority	Low
Contract Notes	
Document Type	Primary
Case Type	Case Management

Representative	Account #	Date
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004
Mr. John Q. Smith	000001	1/1/2001
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004
Mr. John Q. Smith	000001	1/1/2001
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004
Mr. John Q. Smith	000001	1/1/2001
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004
Mr. John Q. Smith	000001	1/1/2001
Ms. Alicia R. Dobbs	000002	3/3/2003
Mrs. T.R. Jones	000003	4/4/2004

Documents attached: 2

Role: Supervisor   Case ID: 67645   Administrator

Writing the Code

Since the only real requirement here is for whatever code pointed to by the Lookup Service URL to produce an XML stream that conforms to the schema specified above, it is hard to give any hard and fast guidelines for how this code should be written other than to point to code samples. So please consult the `\src\Lookup Samples` folder in the code samples for more details on how to implement custom lookups.

Points to Keep in Mind

Nevertheless, especially in regard to making sure that problems your code encounters can be accurately displayed by the BPF Web Application, there are some specific points to be remembered while building, for example, a custom JSP page to handle the lookup functionality exposed by BPF as documented above.

- All validation needs to be done within a try-catch block, so that the custom code can build an Exception node in the catch block and send a valid response to the BPF web application indicating what went wrong.
- If JSP is used, the Page directive "errorPage" should not be defined in the custom page. This is because better diagnostic information can be obtained when an “errorPage” directive is not used. When a JSP fails, almost all JSP engines will display the error and the complete stack trace. This identifies the JSP methods that were called. Some configurations also identify the line where the

code failed. This is very useful for debugging purposes. Note that the function of the `Description` subnode of the Exception node can vary from containing a stack trace of the Exception object (for debugging) to a detailed and localized message for end users once development is complete. It was primarily designed for the latter purpose, so as to allow BPF to present a user friendly error message and description when the custom page fails for any reason.

- Page comments need to be proprietary to the language used. For instance, if the custom page is developed in JSP, then JSP page comments need to be used.
- The JSP try-catch statement allows you to test a block of code for errors. The try block contains the code to be run, and the catch block contains the code to be executed if an error occurs. These blocks can be used to capture any runtime exception in the custom page. When an Exception is thrown, the reason for the error can be sent as a message to the catch block for constructing Exception sub node in the `Response` DOM object.
- All code validations need to be present inside the JSP try-catch block if you are using JSP. For example, any exceptions thrown by the custom code in response to unexpected results need to be thrown within the try block.
- Assuming the custom code is a JSP page, the BPF Web Application will display an appropriate message if there are any compilation errors.

Localization & Globalization

The BPF Web Application will localize the Exception labels based on the key attributes provided. However, if a `Value` node is present in the Exception sub node, the BPF Web Application will show the message being defined in the `Value` node. But the 'key' attribute value would take the precedence in displaying the localized message, if the key=value pair is present in the resource bundle.

The following format needs to be used when defining keys in the resource bundle for lookup module:

```
bp8.client."Key"=localized value
```

Deploying the Code

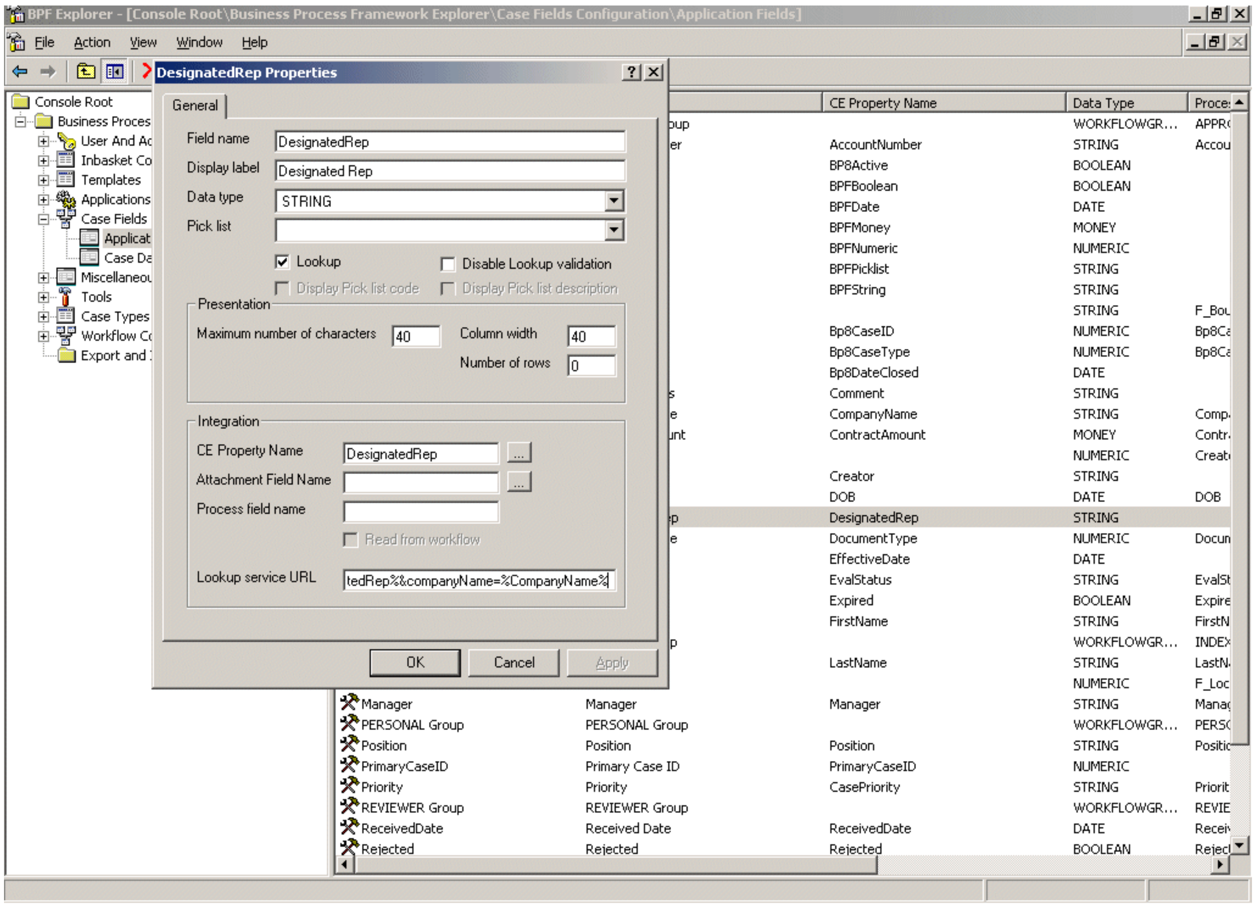
Since the JSP page (or servlet, web service, or other data source) that will be configured as a Lookup Service URL might well be located on another Web server entirely in some remote location, there are no hard and fast rules as to how this code should be deployed except that it should be accessible to the BPF Web Application via HTTP.

Configuration

As mentioned in the Introduction to this section, little more is required here than specifying a valid URL that points to your lookup JSP page (or Servlet, etc.), passing whatever parameters are required, as well as remembering that the current values of case fields can be passed dynamically by using the `%caseFieldName%` syntax mentioned earlier, as in the following example (repeated here from the sample code):

```
plugins/custom/lookups/DesignatedRepLookup.jsp?designatedRep=%DesignatedRep%&companyName=%Com
panyName%
```

The following screen shot shows a partial view (the end of the URL string) of how the configuration screen actually looks within BPF Explorer.



Field name: DesignatedRep
Display label: Designated Rep
Data type: STRING
Pick list:
[✓] Lookup
[ ] Disable Lookup validation
[ ] Display Pick list code
[ ] Display Pick list description

Presentation
Maximum number of characters: 40
Column width: 40
Number of rows: 0

Integration
CE Property Name: DesignatedRep
Attachment Field Name:
Process field name:
[ ] Read from workflow
Lookup service URL: tedRep%&companyName=%CompanyName%

		CE Property Name	Data Type	Proce
	oup		WORKFLOWGR...	APPR
	er	AccountNumber	STRING	Accou
		BP8Active	BOOLEAN	
		BPFBoolean	BOOLEAN	
		BPFDate	DATE	
		BPFMoney	MONEY	
		BPFNumeric	NUMERIC	
		BPFPicklist	STRING	
		BPFString	STRING	
			STRING	F_Bou
		Bp8CaseID	NUMERIC	Bp8Ca
		Bp8CaseType	NUMERIC	Bp8Ca
		Bp8DateClosed	DATE	
	s	Comment	STRING	
	e	CompanyName	STRING	Comp
	unt	ContractAmount	MONEY	Contr
			NUMERIC	Creat
		Creator	STRING	
		DOB	DATE	DOB
	p	DesignatedRep	STRING	
	e	DocumentType	NUMERIC	Docun
		EffectiveDate	DATE	
		EvalStatus	STRING	EvalSt
		Expired	BOOLEAN	Expire
		FirstName	STRING	FirstN
	p		WORKFLOWGR...	INDEX
		LastName	STRING	LastN
			NUMERIC	F_Loc
Manager	Manager	Manager	STRING	Manag
PERSONAL Group	PERSONAL Group		WORKFLOWGR...	PERS
Position	Position	Position	STRING	Positic
PrimaryCaseID	Primary Case ID	PrimaryCaseID	NUMERIC	
Priority	Priority	CasePriority	STRING	Priorit
REVIEWER Group	REVIEWER Group		WORKFLOWGR...	REVIE
ReceivedDate	Received Date	ReceivedDate	DATE	Recei
Rejected	Rejected	Rejected	BOOLEAN	Rejec

Sample code

Consult the `\src\Lookup Samples` folder in the code samples for more details on how to implement custom lookups.

Customizing BPF Date/Time handling capabilities

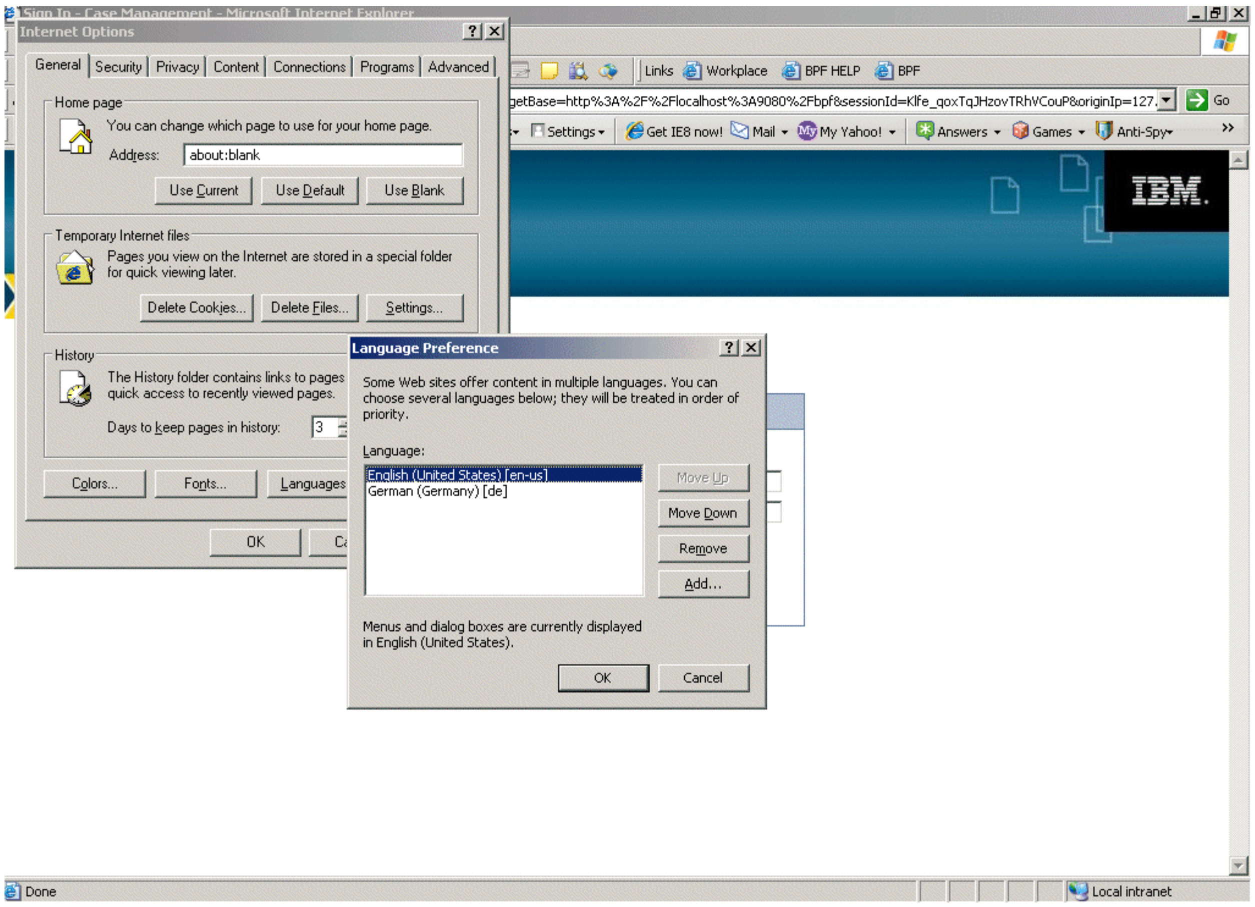
Introduction

Overview

NOTE The changes discussed here involve the modification of BPF core files like `Bp8Main.jsp` or `Bp8Util.vbs`. Any changes you make to BPF core files in the course of implementing a custom solution like this will have to be backed up and reapplied each time you upgrade BPF or apply a new patch, since they may be overwritten otherwise.

The display and formatting of Date/Time values in the BPF Web Application is determined for each individual by two distinct settings.

- The first of these is the user's individual default language preference setting for Internet Explorer as determined through **Tools** > **Internet Options...** > **Languages...** (as in the screenshot below):



getBase=http%3A%2F%2Flocalhost%3A9080%2Fbpf&sessionId=Klfe_qoxTqJHzovTRhVCouP&originIp=127.

You can change which page to use for your home page.
Address: about:blank

Pages you view on the Internet are stored in a special folder for quick viewing later.

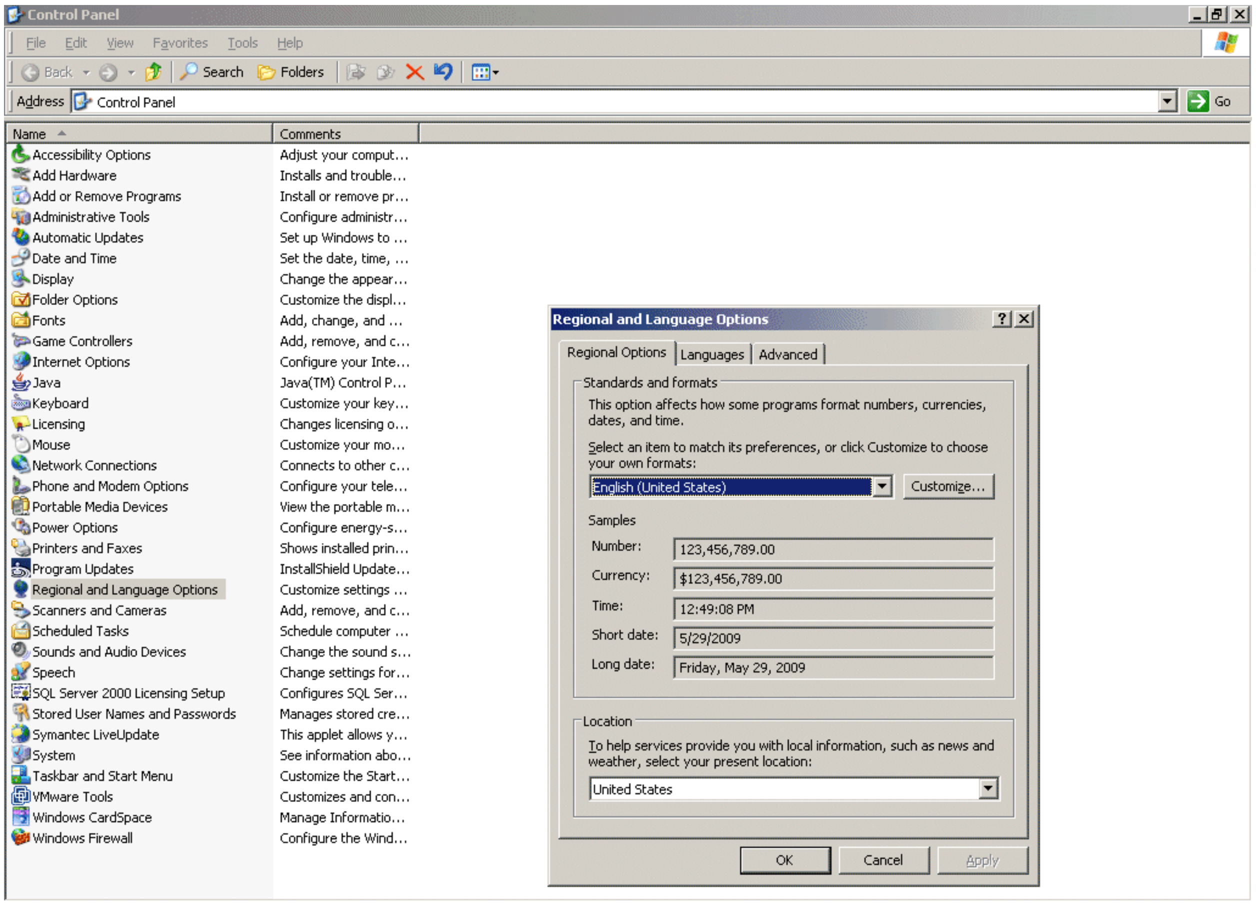
The History folder contains links to pages quick access to recently viewed pages.
Days to keep pages in history: 3

Some Web sites offer content in multiple languages. You can choose several languages below; they will be treated in order of priority.

Language:
English (United States) [en-us]
German (Germany) [de]

Menus and dialog boxes are currently displayed in English (United States).

- The second of these is the user's individual Regional Options settings (on their individual PC where Internet Explorer is running) for the formatting of Time and Date for the specific language setting that is at the top of the list of Languages in Internet Explorer as described above under item 1. For example, the following screenshot shows that the format for Date has been set a certain way specifically for English (United States).



Name	Comments
Accessibility Options	Adjust your comput…
Add Hardware	Installs and trouble…
Add or Remove Programs	Install or remove pr…
Administrative Tools	Configure administr…
Automatic Updates	Set up Windows to …
Date and Time	Set the date, time, …
Display	Change the appear…
Folder Options	Customize the displ…
Fonts	Add, change, and …
Game Controllers	Add, remove, and c…
Internet Options	Configure your Inte…
Java	Java(TM) Control P…
Keyboard	Customize your key…
Licensing	Changes licensing o…
Mouse	Customize your mo…
Network Connections	Connects to other c…
Phone and Modem Options	Configure your tele…
Portable Media Devices	View the portable m…
Power Options	Configure energy-s…
Printers and Faxes	Shows installed prin…
Program Updates	InstallShield Update…
Regional and Language Options	Customize settings …
Scanners and Cameras	Add, remove, and c…
Scheduled Tasks	Schedule computer …
Sounds and Audio Devices	Change the sound s…
Speech	Change settings for…
SQL Server 2000 Licensing Setup	Configures SQL Ser…
Stored User Names and Passwords	Manages stored cre…
Symantec LiveUpdate	This applet allows y…
System	See information abo…
Taskbar and Start Menu	Customize the Start…
VMware Tools	Customizes and con…
Windows CardSpace	Manage Informatio…
Windows Firewall	Configure the Wind…

Regional and Language Options

Regional Options | Languages | Advanced

Standards and formats

This option affects how some programs format numbers, currencies, dates, and time.

Select an item to match its preferences, or click Customize to choose your own formats:

English (United States) — Customize...

Samples

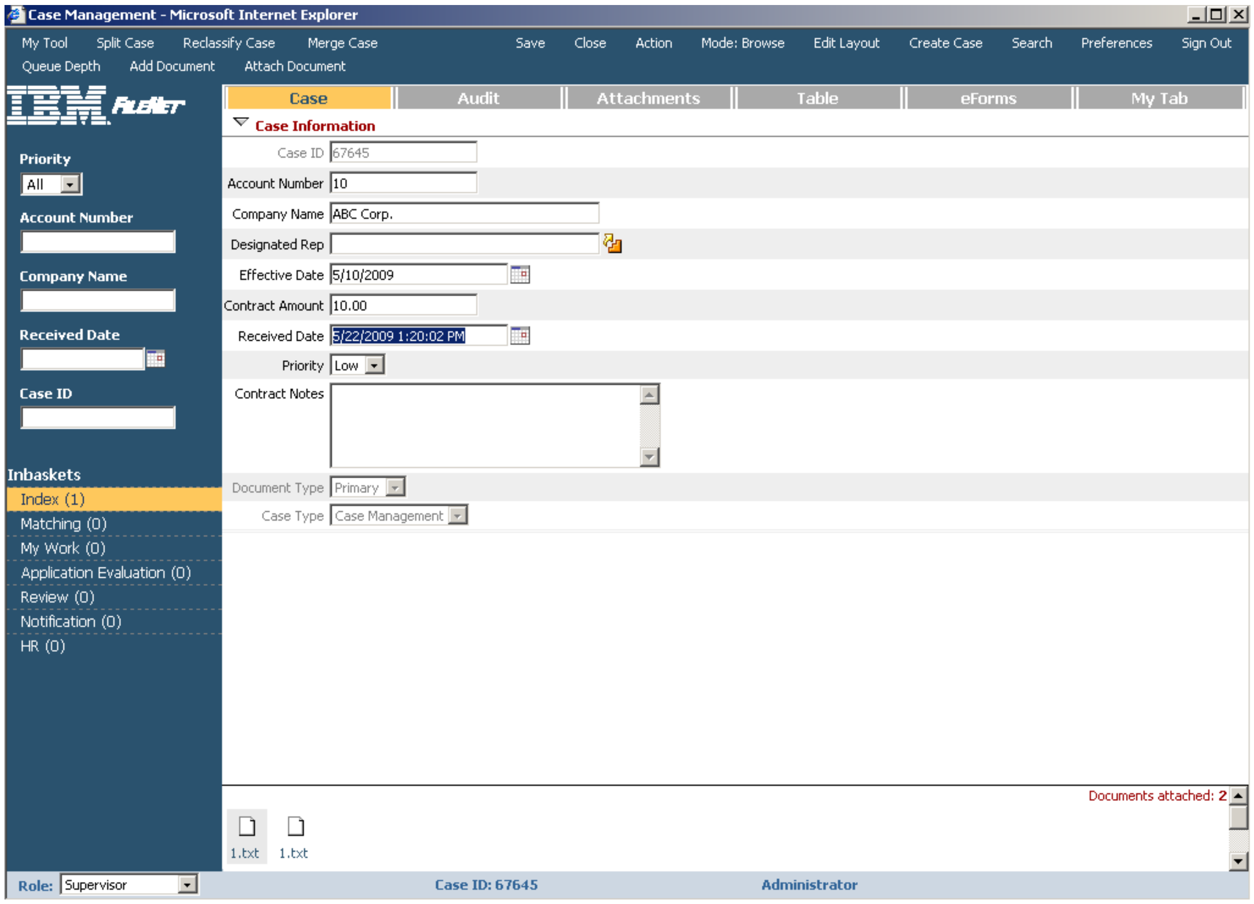
Number:	123,456,789.00
Currency:	$123,456,789.00
Time:	12:49:08 PM
Short date:	5/29/2009
Long date:	Friday, May 29, 2009

Location

To help services provide you with local information, such as news and weather, select your present location:

United States

As long as the user also has their Internet Explorer default Language set to English (United States) as well, this will be the format of the Date values that they will see in the BPF Web Application.



Case ID	67645
Account Number	10
Company Name	ABC Corp.
Designated Rep	
Effective Date	5/10/2009
Contract Amount	10.00
Received Date	5/22/2009 1:20:02 PM
Priority	Low
Contract Notes	
Document Type	Primary
Case Type	Case Management

Role: Supervisor Case ID: 67645 Administrator

Implementation

The basic Date/Time handling capabilities of BPF are implemented in the `Bp8Util.vbs` file that resides in the `\vbs` directory of BPF `<web_root>` directory. The reason to use VBScript (rather than JavaScript) to implement this functionality has to do with the superior versatility of some core VBScript functions when running in Microsoft® environments such as those that BPF supports. Ultimately, this functionality lies almost entirely with a combination of the Microsoft VBScript `CDate`, `FormatDateTime`, and `DatePart` functions:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vsfctcdate.asp

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vsfctformatdatetime.asp

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vsfctdatepart.asp

These functions automatically reference and take into account the Regional Settings language, time zone, and Date/Time formatting options on the Windows machine where they are executed (in this case, the end user's browser client PC), and handle Date/Time values accordingly.

Customization

Nevertheless, it may well happen that individual implementations will wish to make use of these VBScript functions for customization purposes in ways that the methods exposed by `Bp8Util.vbs` do not address.

For example, it might be necessary to implement script from `EventHandlers.js` to parse Date values entered by the user (prior to the normal BPF validation routines) into their correct date parts and perform custom validation on them (by passing them to a JavaScript function that references calendar information to determine whether the date entered falls on a weekend, for example), but in a portable fashion that will automatically take the user's individual language and formatting options into account.

To achieve this, the recommended approach would be as follows.

1. Add to the end of `Bp8Util.vbs`, or place in another, `custom .vbs` include file in the `\vbs` directory, a new VBScript function patterned after the following sample code. If you create a `new .vbs` include file, make sure to include it by adding it to the `\UI-INF\jsp\ui\Bp8Main.jsp` file under the entry for the existing Bp8Util.vbs:

   ```
    <script type="text/vbscript" src="vbs/Bp8Util.vbs"></script>
   <script type="text/vbscript" src="vbs/MY_NEW_FILE.vbs"></script>
   ```

   NOTE Again, you will need to modify the code here, at the end of this function, so as to have it build the correctly formatted string for your own custom Weekend checker JavaScript function (whatever that happens to be). The "d.m.yyyy" format is just used as an example.

```
'================================================================================
'VBScript

Function CustomParseDate(psValue)

 Const LOCATION_HERE = "CustomParseDate"
 Dim ldtDate 'As Date
 Dim liDayDatePart 'As Integer
 Dim liMonthDatePart 'As Integer
 Dim liYearDatePart 'As Integer
 Dim lsStringToPassToMyWeekendCheckerFunction

On Error Resume Next

 ldtDate = CDate(psValue)

 If (Err.Number <> 0) Then
 CustomParseDate = "ERROR:[" & LOCATION_HERE & "]" & Err.Number & " " & Err.Description
 Exit Function
 End If

 liDayDatePart = DatePart("d", ldtDate)
 liMonthDatePart = DatePart("m", ldtDate)
 liYearDatePart = DatePart("yyyy", ldtDate)

 lsStringToPassToMyWeekendCheckerFunction = liDayDatePart & "." & liMonthDatePart & "." &
liYearDatePart

 CustomParseDate = lsStringToPassToMyWeekendCheckerFunction

End Function

'================================================================================
```

2. Add the following code to the `onFieldBlur` method of your `EventHandlers.js` file.

```
eventHandlers.onFieldBlur = function(fieldElement) {

//===============================================================================

 //var FIELDTYPE_DATETIME = '3'; // Already declared in Bp8InitHead.jsp.
 //alert("fieldElement.datatype==" + fieldElement.datatype);
 //alert("fieldElement.tagName==" + fieldElement.tagName);

 // Only do this for date fields.
 if ((fieldElement.tagName == 'INPUT') && (fieldElement.datatype== FIELDTYPE_DATETIME)) {
 //alert("fieldElement.value==" + fieldElement.value);
 //alert("fieldElement.name==" + fieldElement.name);
 var sFieldElementValue = fieldElement.value;
 var oRootXmlDOM = currentCase.objCase;
 var sFieldNameParameter = "Prop[@Name='" + fieldElement.name + "']/Value";
 var sOriginalValue = oRootXmlDOM.selectSingleNode(sFieldNameParameter).nodeTypedValue;
 //alert("sOriginalValue==" + sOriginalValue);

 var sCustomParseDate = CustomParseDate(sFieldElementValue);
 if (sCustomParseDate.indexOf("ERROR") >= 0) {
  //alert("\""sFieldElementValue + "\" is an invalid date.")
  voluntaryShowMessage("Invalid Date!", "\"" + sFieldElementValue + "\" is an invalid date.");
  fieldElement.value = sOriginalValue;
  fieldElement.focus();
  return false;
 }
 else {
  alert("sCustomParseDate==" + sCustomParseDate);
  // Call your custom WeekendChecker Function here with the value returned above.
 }
 }

 //===============================================================================

 return true;
}
```

This simplistic example should at least provide some indication of the approach to take.

NOTE It is perfectly possible to call VBScript from JavaScript and vice versa.

NOTE Any changes you make to BPF core files like `Bp8Main.jsp` or `Bp8Util.vbs` in the course of implementing a custom solution like this will have to be backed up and reapplied each time you upgrade BPF or apply a new patch, since they may be overwritten otherwise.

User Preferences

Introduction

Overview

The BPF user preferences interface allows the BPF system administrator to define a preferences structure and have BPF store its values for each user. By default, this structure is defined in the `user-preferences.xml` file, which resides stand-alone, in the `<web_root>\WEB-INF` directory. The actual preference values for each user are stored in XML files that reside in the CE Object Store, with document title values like the following:

```
User preferences for Administrator(83) on Case Management
```

The current version of User Preferences supports following attribute types: String, Integer, Boolean, and Option.

Preferences access

There are two ways to access and modify preferences data through custom code – either directly (without caching) or by using the `UserPreferencesManager` class (with caching). The first of these effectively bypasses the default functionality described above in the Overview. The second of these exposes the default functionality described above in the Overview for customization purposes.

In the first case you have to initialize your own instance of the `Preferences` class by performing the following operations manually in your custom Java code:

```
String configFileId = "user-preferences.xml";

InputStream stream =
Thread.currentThread().getContextClassLoader().getResourceAsStream(configFileId);

PreferencesTree preferencesTree = new PreferencesTreeImpl(stream);

Preferences preferences = new PreferencesImpl(<some stream with stored user preferences>,
preferencesTree);
```

After preferences are initialized you can access and modify them:

```
preferences.getIntValue('id of attribute');
preferences.setIntValue('id of attribute', 123);
```

Finally you have to save all your changes:

```
preferences.save(<some output stream>);
```

In the second case you can use `UserPreferencesManager` helper-class to simplify preferences access. It initializes itself from the predefined configuration file `user-preferences.xml` and can load and store preferences for the specific user in CE. This implementation uses the `PreferencesDocument` CE document class with version support to store user preferences data.

```
UserPreferencesManager preferencesManager = UserPreferencesManager.getInstance();

PreferencesTree preferencesTree = preferencesManager.getPreferencesTree();

Preferences preferences = preferencesManager.getPreferences(<Bp8User reference>,
<Bp8TxDataStore reference>);
```

Access and modification of the values are handled in the same manner as for the previous case:

```
preferences.getIntValue('id of attribute');

preferences.setIntValue('id of attribute', 123);
```

Finally you can save changes for the specific user to CE:

```
preferencesManager.save(<Bp8User reference>, <Bp8TxDataStore reference>);
```

In both cases you can use an alternate approach for preferences access:

```
preferences.getAttributeValue(<Attribute reference>);

preferences.setAttributeValue(<Attribute reference>, <PersitentValue reference>);
```

It is recommended that you use this approach for unified access to attributes of different types. In addition, you can define new attribute type (multi option value for example) and handle it in this manner. No interface modification is required in this case.

Configuration file format description

BPF reads a `PreferencesTree` structure from the XML configuration file to create the user preferences list. Here you will find description of all of the important elements of this file.

See the current `<web_root>\WEB-INF\preferences.xsd` file in your deployed BPF Web Application for further information.

User Preferences Sample Code

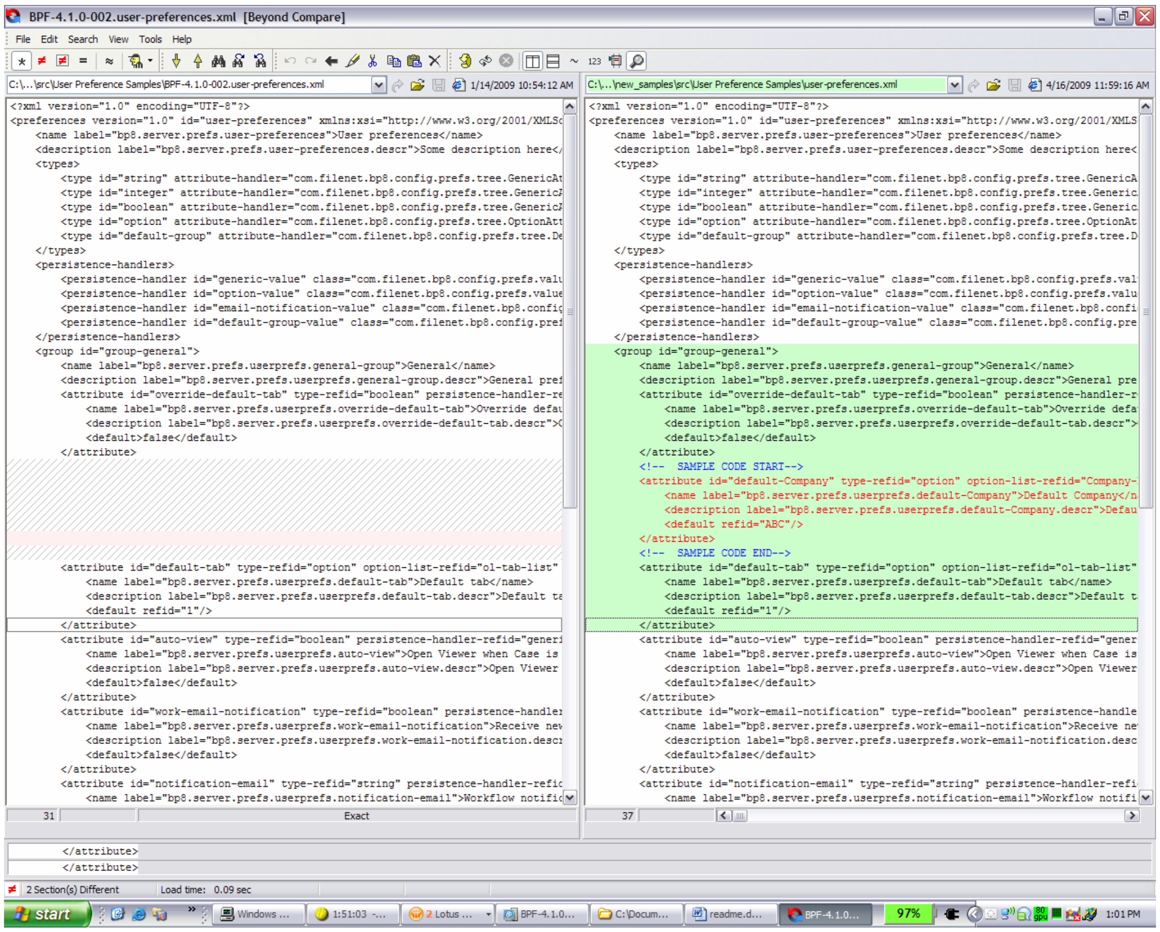
The following instructions apply to the sample code in the `\src\User Preference Samples` folder in the BPF code samples that accompany this guide. For ease of reference, the following steps assume your code samples have been installed to (unzipped in) the following folder: “C:¥bpf_code_samples¥CODE_SAMPLES“

1. Back up your

   `<bpf>\WEB-INF\user-preferences.xml`

   file by making a copy of it and adding an “`.org`” extension to the copy.

2. Open your `<bpf>\WEB-INF\user-preferences.xml` file and modify it by pasting the first of the two sections in your `C:\bpf_code_samples\CODE_SAMPLES\src\User Preference Samples\paste_me_into_user-preferences.xml.txt` into the location seen in red in the following screenshot:



BPF-4.1.0-002.user-preferences.xml [Beyond Compare]

C:\...\src\User Preference Samples\BPF-4.1.0-002.user-preferences.xml 1/14/2009 10:54:12 AM

```
<?xml version="1.0" encoding="UTF-8"?>
<preferences version="1.0" id="user-preferences" xmlns:xsi="http://www.w3.org/2001/XMLSc
    <name label="bp8.server.prefs.user-preferences">User preferences</name>
    <description label="bp8.server.prefs.user-preferences.descr">Some description here</
    <types>
        <type id="string" attribute-handler="com.filenet.bp8.config.prefs.tree.GenericAt
        <type id="integer" attribute-handler="com.filenet.bp8.config.prefs.tree.GenericA
        <type id="boolean" attribute-handler="com.filenet.bp8.config.prefs.tree.GenericA
        <type id="option" attribute-handler="com.filenet.bp8.config.prefs.tree.OptionAtt
        <type id="default-group" attribute-handler="com.filenet.bp8.config.prefs.tree.De
    </types>
    <persistence-handlers>
        <persistence-handler id="generic-value" class="com.filenet.bp8.config.prefs.valu
        <persistence-handler id="option-value" class="com.filenet.bp8.config.prefs.value
        <persistence-handler id="email-notification-value" class="com.filenet.bp8.config
        <persistence-handler id="default-group-value" class="com.filenet.bp8.config.pref
    </persistence-handlers>
    <group id="group-general">
        <name label="bp8.server.prefs.userprefs.general-group">General</name>
        <description label="bp8.server.prefs.userprefs.general-group.descr">General pref
        <attribute id="override-default-tab" type-refid="boolean" persistence-handler-re
            <name label="bp8.server.prefs.userprefs.override-default-tab">Override defau
            <description label="bp8.server.prefs.userprefs.override-default-tab.descr">C
            <default>false</default>
        </attribute>






        <attribute id="default-tab" type-refid="option" option-list-refid="ol-tab-list"
            <name label="bp8.server.prefs.userprefs.default-tab">Default tab</name>
            <description label="bp8.server.prefs.userprefs.default-tab.descr">Default ta
            <default refid="1"/>
        </attribute>
        <attribute id="auto-view" type-refid="boolean" persistence-handler-refid="generi
            <name label="bp8.server.prefs.userprefs.auto-view">Open Viewer when Case is
            <description label="bp8.server.prefs.userprefs.auto-view.descr">Open Viewer
            <default>false</default>
        </attribute>
        <attribute id="work-email-notification" type-refid="boolean" persistence-handler
            <name label="bp8.server.prefs.userprefs.work-email-notification">Receive new
            <description label="bp8.server.prefs.userprefs.work-email-notification.descr
            <default>false</default>
        </attribute>
        <attribute id="notification-email" type-refid="string" persistence-handler-refic
            <name label="bp8.server.prefs.userprefs.notification-email">Workflow notific
```

31 Exact

C:\...\new_samples\src\User Preference Samples\user-preferences.xml 4/16/2009 11:59:16 AM

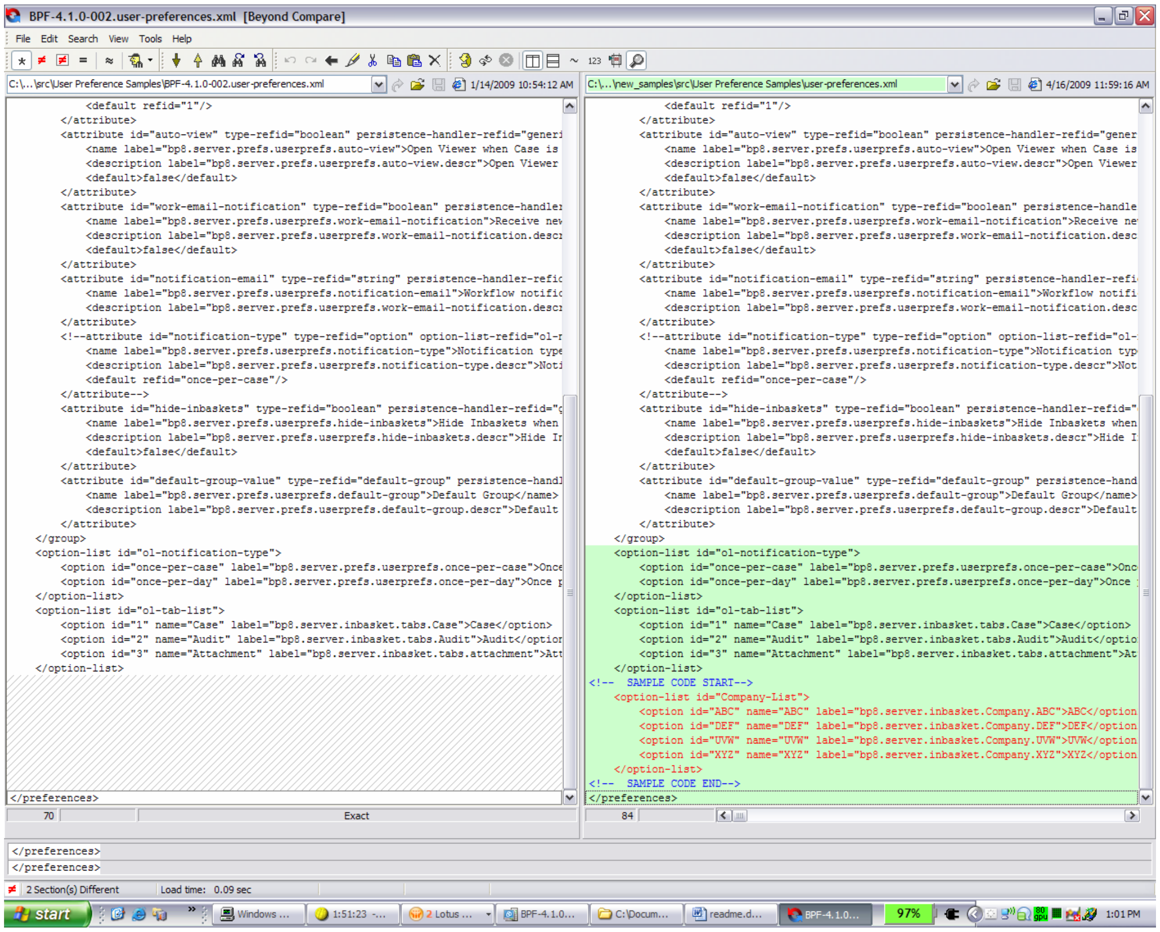
```
<?xml version="1.0" encoding="UTF-8"?>
<preferences version="1.0" id="user-preferences" xmlns:xsi="http://www.w3.org/2001/XMLS
    <name label="bp8.server.prefs.user-preferences">User preferences</name>
    <description label="bp8.server.prefs.user-preferences.descr">Some description here<
    <types>
        <type id="string" attribute-handler="com.filenet.bp8.config.prefs.tree.GenericA
        <type id="integer" attribute-handler="com.filenet.bp8.config.prefs.tree.Generic
        <type id="boolean" attribute-handler="com.filenet.bp8.config.prefs.tree.Generic
        <type id="option" attribute-handler="com.filenet.bp8.config.prefs.tree.OptionAt
        <type id="default-group" attribute-handler="com.filenet.bp8.config.prefs.tree.D
    </types>
    <persistence-handlers>
        <persistence-handler id="generic-value" class="com.filenet.bp8.config.prefs.val
        <persistence-handler id="option-value" class="com.filenet.bp8.config.prefs.valu
        <persistence-handler id="email-notification-value" class="com.filenet.bp8.confi
        <persistence-handler id="default-group-value" class="com.filenet.bp8.config.pre
    </persistence-handlers>
    <group id="group-general">
        <name label="bp8.server.prefs.userprefs.general-group">General</name>
        <description label="bp8.server.prefs.userprefs.general-group.descr">General pre
        <attribute id="override-default-tab" type-refid="boolean" persistence-handler-r
            <name label="bp8.server.prefs.userprefs.override-default-tab">Override defa
            <description label="bp8.server.prefs.userprefs.override-default-tab.descr">
            <default>false</default>
        </attribute>
        <!--  SAMPLE CODE START-->
        <attribute id="default-Company" type-refid="option" option-list-refid="Company-
            <name label="bp8.server.prefs.userprefs.default-Company">Default Company</n
            <description label="bp8.server.prefs.userprefs.default-Company.descr">Defau
            <default refid="ABC"/>
        </attribute>
        <!--  SAMPLE CODE END-->
        <attribute id="default-tab" type-refid="option" option-list-refid="ol-tab-list"
            <name label="bp8.server.prefs.userprefs.default-tab">Default tab</name>
            <description label="bp8.server.prefs.userprefs.default-tab.descr">Default t
            <default refid="1"/>
        </attribute>
        <attribute id="auto-view" type-refid="boolean" persistence-handler-refid="gener
            <name label="bp8.server.prefs.userprefs.auto-view">Open Viewer when Case is
            <description label="bp8.server.prefs.userprefs.auto-view.descr">Open Viewer
            <default>false</default>
        </attribute>
        <attribute id="work-email-notification" type-refid="boolean" persistence-handle
            <name label="bp8.server.prefs.userprefs.work-email-notification">Receive ne
            <description label="bp8.server.prefs.userprefs.work-email-notification.desc
            <default>false</default>
        </attribute>
        <attribute id="notification-email" type-refid="string" persistence-handler-refi
            <name label="bp8.server.prefs.userprefs.notification-email">Workflow notifi
```

37

```
        </attribute>
        </attribute>
```

2 Section(s) Different Load time: 0.09 sec

Then further modify your `<bpf>\WEB-INF\user-preferences.xml` file and by pasting the second of the two sections in your `C:\bpf_code_samples\CODE_SAMPLES\src\User Preference Samples\paste_me_into_user-preferences.xml.txt` into the location seen in red in the following screenshot:


BPF-4.1.0-002.user-preferences.xml [Beyond Compare]
File Edit Search View Tools Help
C:\...\src\User Preference Samples\BPF-4.1.0-002.user-preferences.xml
1/14/2009 10:54:12 AM
C:\...\new_samples\src\User Preference Samples\user-preferences.xml
4/16/2009 11:59:16 AM
70
Exact
84
2 Section(s) Different
Load time: 0.09 sec
start
1:01 PM

```
            <default refid="1"/>
        </attribute>
        <attribute id="auto-view" type-refid="boolean" persistence-handler-refid="generi
            <name label="bp8.server.prefs.userprefs.auto-view">Open Viewer when Case is
            <description label="bp8.server.prefs.userprefs.auto-view.descr">Open Viewer
            <default>false</default>
        </attribute>
        <attribute id="work-email-notification" type-refid="boolean" persistence-handler
            <name label="bp8.server.prefs.userprefs.work-email-notification">Receive new
            <description label="bp8.server.prefs.userprefs.work-email-notification.descr
            <default>false</default>
        </attribute>
        <attribute id="notification-email" type-refid="string" persistence-handler-refic
            <name label="bp8.server.prefs.userprefs.notification-email">Workflow notific
            <description label="bp8.server.prefs.userprefs.work-email-notification.descr
        </attribute>
        <!--attribute id="notification-type" type-refid="option" option-list-refid="ol-r
            <name label="bp8.server.prefs.userprefs.notification-type">Notification type
            <description label="bp8.server.prefs.userprefs.notification-type.descr">Noti
            <default refid="once-per-case"/>
        </attribute-->
        <attribute id="hide-inbaskets" type-refid="boolean" persistence-handler-refid="g
            <name label="bp8.server.prefs.userprefs.hide-inbaskets">Hide Inbaskets when
            <description label="bp8.server.prefs.userprefs.hide-inbaskets.descr">Hide In
            <default>false</default>
        </attribute>
        <attribute id="default-group-value" type-refid="default-group" persistence-handl
            <name label="bp8.server.prefs.userprefs.default-group">Default Group</name>
            <description label="bp8.server.prefs.userprefs.default-group.descr">Default
        </attribute>
    </group>
    <option-list id="ol-notification-type">
        <option id="once-per-case" label="bp8.server.prefs.userprefs.once-per-case">Once
        <option id="once-per-day" label="bp8.server.prefs.userprefs.once-per-day">Once p
    </option-list>
    <option-list id="ol-tab-list">
        <option id="1" name="Case" label="bp8.server.inbasket.tabs.Case">Case</option>
        <option id="2" name="Audit" label="bp8.server.inbasket.tabs.Audit">Audit</option
        <option id="3" name="Attachment" label="bp8.server.inbasket.tabs.attachment">Att
    </option-list>
</preferences>
```

```
            <default refid="1"/>
        </attribute>
        <attribute id="auto-view" type-refid="boolean" persistence-handler-refid="gener
            <name label="bp8.server.prefs.userprefs.auto-view">Open Viewer when Case is
            <description label="bp8.server.prefs.userprefs.auto-view.descr">Open Viewer
            <default>false</default>
        </attribute>
        <attribute id="work-email-notification" type-refid="boolean" persistence-handle
            <name label="bp8.server.prefs.userprefs.work-email-notification">Receive ne
            <description label="bp8.server.prefs.userprefs.work-email-notification.desc
            <default>false</default>
        </attribute>
        <attribute id="notification-email" type-refid="string" persistence-handler-refi
            <name label="bp8.server.prefs.userprefs.notification-email">Workflow notifi
            <description label="bp8.server.prefs.userprefs.work-email-notification.desc
        </attribute>
        <!--attribute id="notification-type" type-refid="option" option-list-refid="ol-
            <name label="bp8.server.prefs.userprefs.notification-type">Notification typ
            <description label="bp8.server.prefs.userprefs.notification-type.descr">Not
            <default refid="once-per-case"/>
        </attribute-->
        <attribute id="hide-inbaskets" type-refid="boolean" persistence-handler-refid="
            <name label="bp8.server.prefs.userprefs.hide-inbaskets">Hide Inbaskets when
            <description label="bp8.server.prefs.userprefs.hide-inbaskets.descr">Hide I
            <default>false</default>
        </attribute>
        <attribute id="default-group-value" type-refid="default-group" persistence-hand
            <name label="bp8.server.prefs.userprefs.default-group">Default Group</name>
            <description label="bp8.server.prefs.userprefs.default-group.descr">Default
        </attribute>
    </group>
    <option-list id="ol-notification-type">
        <option id="once-per-case" label="bp8.server.prefs.userprefs.once-per-case">Onc
        <option id="once-per-day" label="bp8.server.prefs.userprefs.once-per-day">Once
    </option-list>
    <option-list id="ol-tab-list">
        <option id="1" name="Case" label="bp8.server.inbasket.tabs.Case">Case</option>
        <option id="2" name="Audit" label="bp8.server.inbasket.tabs.Audit">Audit</optio
        <option id="3" name="Attachment" label="bp8.server.inbasket.tabs.attachment">At
    </option-list>
<!--  SAMPLE CODE START-->
    <option-list id="Company-List">
        <option id="ABC" name="ABC" label="bp8.server.inbasket.Company.ABC">ABC</option
        <option id="DEF" name="DEF" label="bp8.server.inbasket.Company.DEF">DEF</option
        <option id="UVW" name="UVW" label="bp8.server.inbasket.Company.UVW">UVW</option
        <option id="XYZ" name="XYZ" label="bp8.server.inbasket.Company.XYZ">XYZ</option
    </option-list>
<!--  SAMPLE CODE END-->
</preferences>
```

```
</preferences>
</preferences>
```

3. Examine your modified

   `<bpf>\WEB-INF\user-preferences.xml`

   - especially the two sections bracketed by "`<!--  SAMPLE CODE START-->`" and "`<!-- SAMPLE CODE END-->`" comments. The additions enclosed by these brackets will show up as a new "**Default Company**" user preference in the BPF Web Application after you redeploy it.

4. Restart your application server to see your changes to take effect.

5. Use the **New** button (create_case tool) to create a new case, and be sure to specify "ABC" as the value for the CompanyName field. If you like, you can also create other cases with `CompanyName` values of "CDE", "UVW", and "XYZ" at this time.

6. Open your

   `C:\bpf_code_samples\CODE_SAMPLES\src\User Preference`

`Samples\paste_me_into_EventHandlers.js.txt`

and examine it before pasting its contents into your

`<bpf>\plugins\custom\ EventHandlers.js`

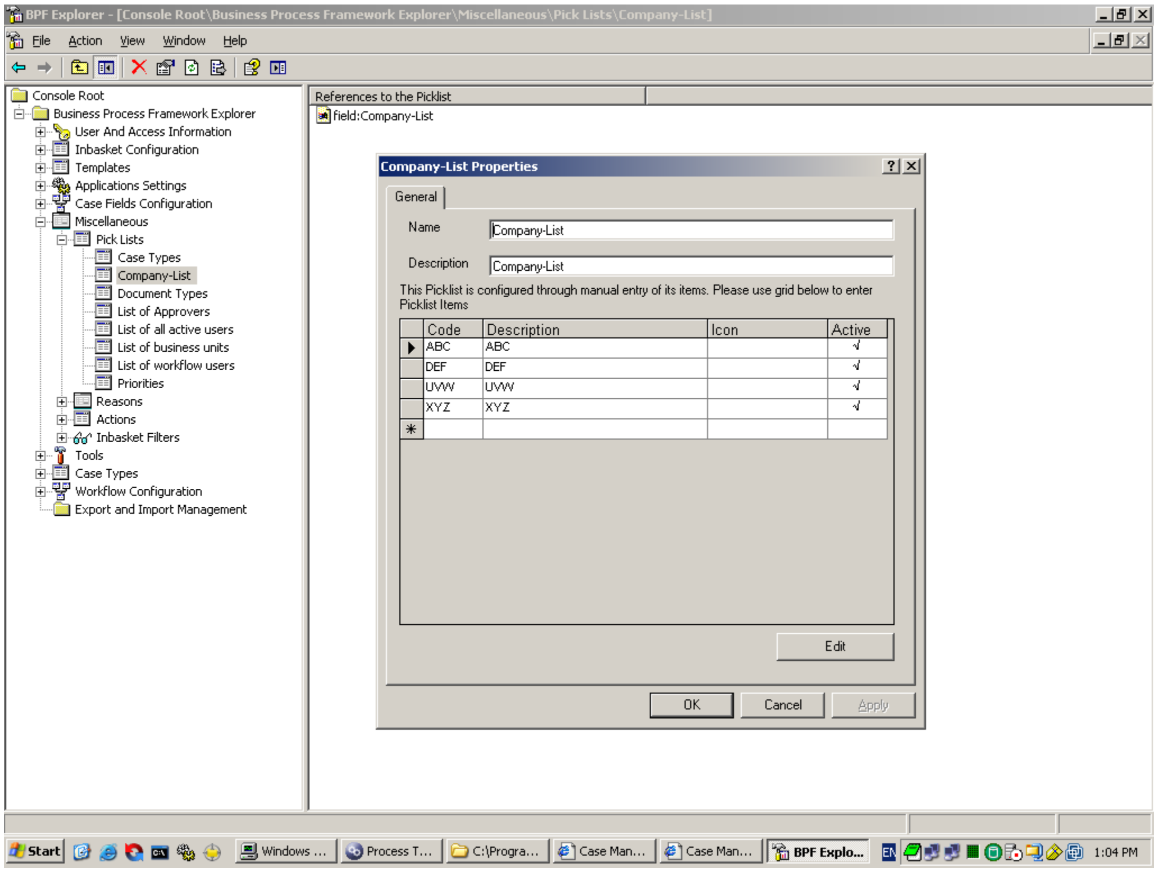
file, making sure to comment out or delete the existing `eventHandlers.onBeforeInbasketLoad` method there.

NOTE You will need to copy the `EventHandlers.js.template` file in this directory and rename it as `EventHandlers.js` if you do not already have a file by this name.

7. Now open BPF Explorer and, under

   Miscellaneous > Pick Lists

   configure a new pick list called "Company-List" as detailed in the following screenshot:



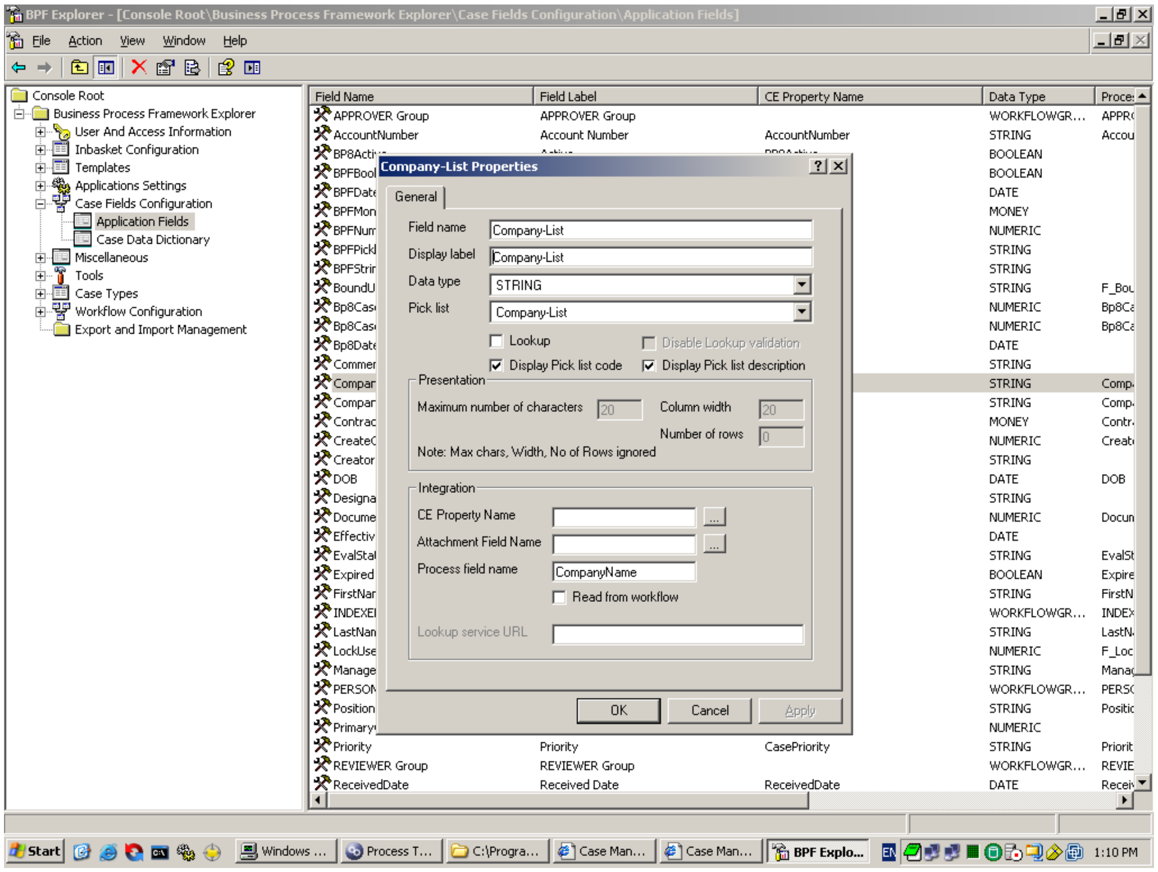
Name: Company-List

Description: Company-List

This Picklist is configured through manual entry of its items. Please use grid below to enter Picklist Items

Code	Description	Icon	Active
ABC	ABC		√
DEF	DEF		√
UVW	UVW		√
XYZ	XYZ		√

8. Now, configure a new case field called “Company-List” with a PE field mapping associating it with the pre-existing `CompanyName` PE field, as detailed in the following screenshot:



BPF Explorer - [Console Root\Business Process Framework Explorer\Case Fields Configuration\Application Fields]

- Console Root
  - Business Process Framework Explorer
    - User And Access Information
    - Inbasket Configuration
    - Templates
    - Applications Settings
    - Case Fields Configuration
      - Application Fields
      - Case Data Dictionary
    - Miscellaneous
    - Tools
    - Case Types
    - Workflow Configuration
    - Export and Import Management

Field Name	Field Label	CE Property Name	Data Type	Proce
APPROVER Group	APPROVER Group		WORKFLOWGR...	APPR(
AccountNumber	Account Number	AccountNumber	STRING	Accou
BP8Acti			BOOLEAN	
BPFBool			BOOLEAN	
BPFDate			DATE	
BPFMon			MONEY	
BPFNum			NUMERIC	
BPFPickl			STRING	
BPFStrir			STRING	
BoundU			STRING	F_Bou
Bp8Casi			NUMERIC	Bp8Ca
Bp8Casi			NUMERIC	Bp8Ca
Bp8Date			DATE	
Commer			STRING	
Compar			STRING	Comp.
Compar			STRING	Comp.
Contrac			MONEY	Contr.
CreateC			NUMERIC	Creat
Creator			STRING	
DOB			DATE	DOB
Designa			STRING	
Docume			NUMERIC	Docun
Effectiv			DATE	
EvalStal			STRING	EvalSt
Expired			BOOLEAN	Expire
FirstNar			STRING	FirstN
INDEXEI			WORKFLOWGR...	INDEX
LastNan			STRING	LastN.
LockUse			NUMERIC	F_Loc
Manage			STRING	Manag
PERSON			WORKFLOWGR...	PERS(
Position			STRING	Positic
Primary			NUMERIC	
Priority	Priority	CasePriority	STRING	Priorit
REVIEWER Group	REVIEWER Group		WORKFLOWGR...	REVIE
ReceivedDate	Received Date	ReceivedDate	DATE	Recei

Company-List Properties

General

Field name	Company-List
Display label	Company-List
Data type	STRING
Pick list	Company-List

☐ Lookup ☐ Disable Lookup validation
☑ Display Pick list code ☑ Display Pick list description

Presentation

Maximum number of characters	20	Column width	20
		Number of rows	0

Note: Max chars, Width, No of Rows ignored

Integration

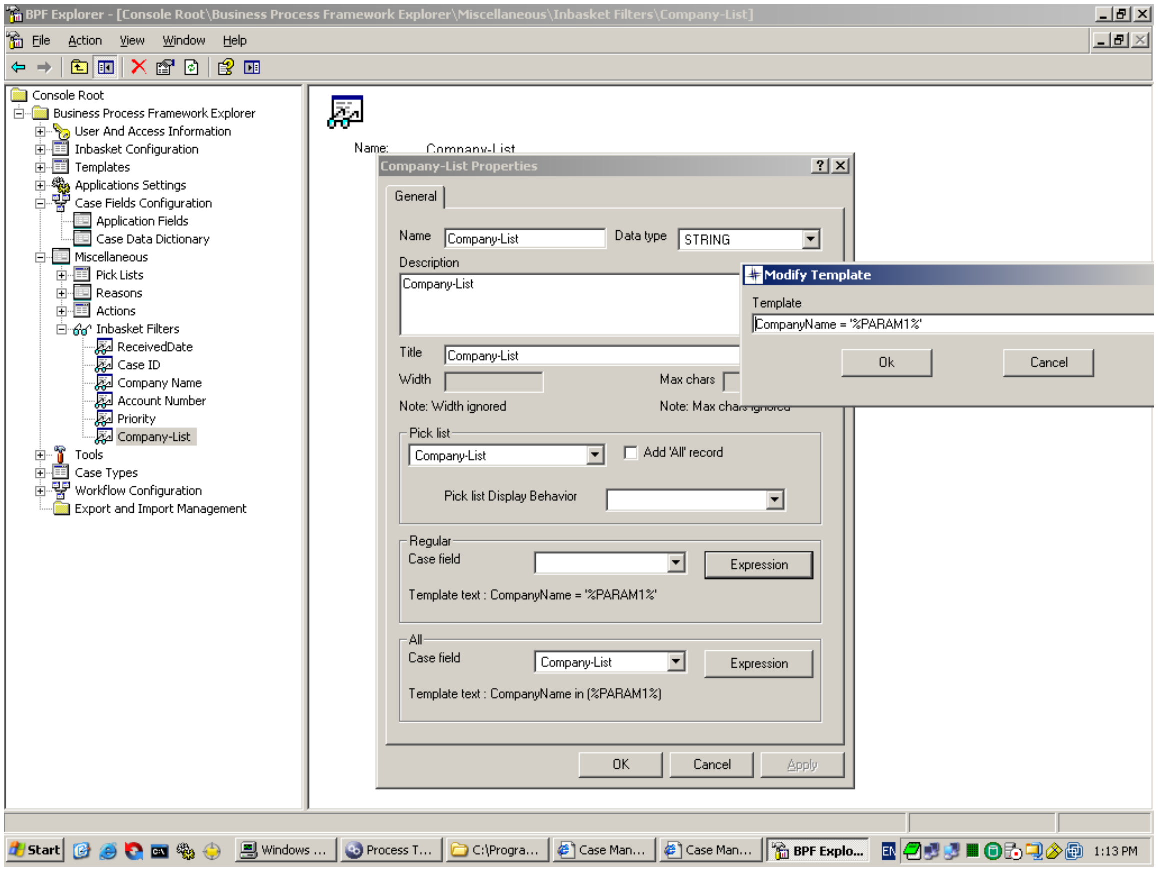
CE Property Name	
Attachment Field Name	
Process field name	CompanyName
☐ Read from workflow	
Lookup service URL	

OK Cancel Apply

9. Now , under

   Miscellaneous > Inbasket Filters

   create a new inbasket filter for this field and mapped to use the pick list you created earlier, as detailed in the following screenshot (<u>and be sure to put single-quotes around the `%PARAM1%` variable in the expression, since this is a String field</u>):



Name: Company-List

Name: Company-List
Data type: STRING
Description: Company-List
Title: Company-List
Width:
Max chars:
Note: Width ignored
Note: Max chars ignored

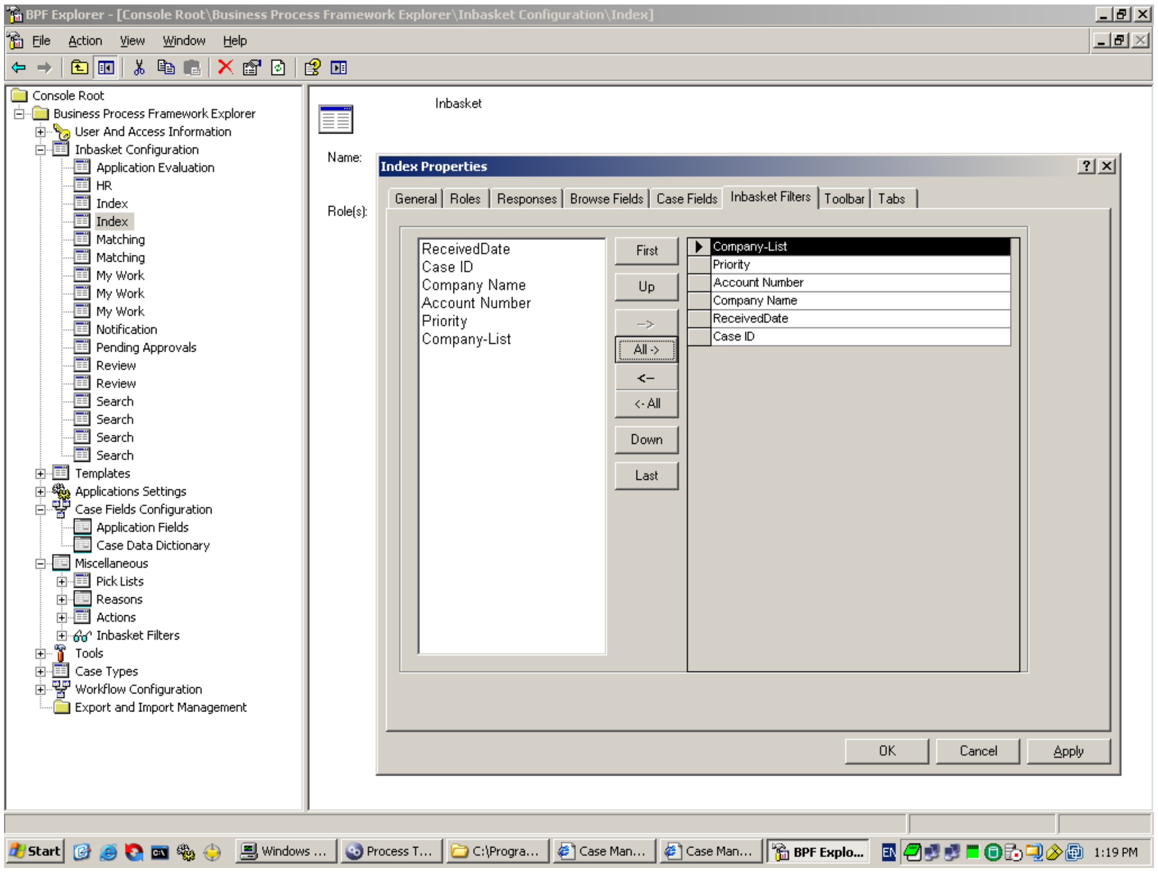
Pick list: Company-List
Add 'All' record: ☐
Pick list Display Behavior:

Regular
Case field:
Template text : CompanyName = '%PARAM1%'

All
Case field: Company-List
Template text : CompanyName in (%PARAM1%)

Template: CompanyName = '%PARAM1%'

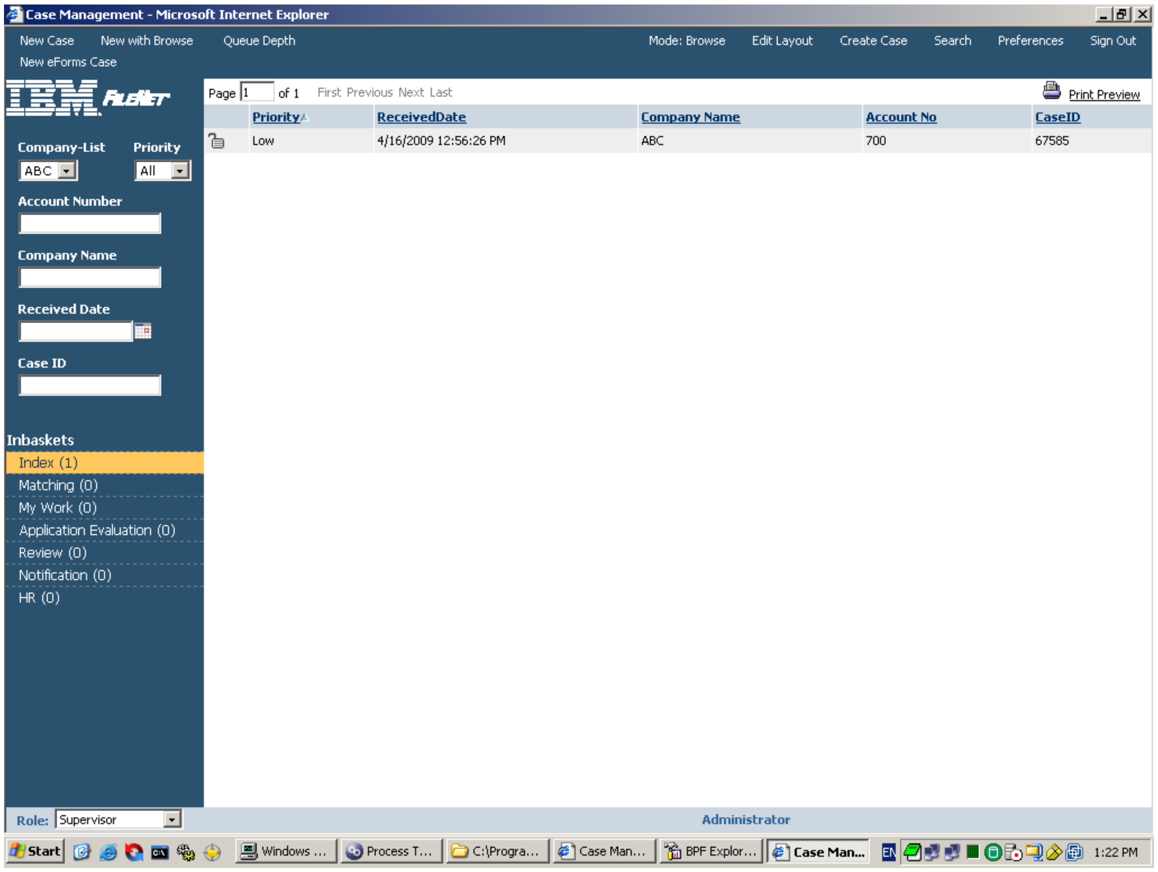
10. Now configure your Index (Supervisor) inbasket to use this new inbasket filter, as detailed in the following screenshot:



ReceivedDate
Case ID
Company Name
Account Number
Priority
Company-List

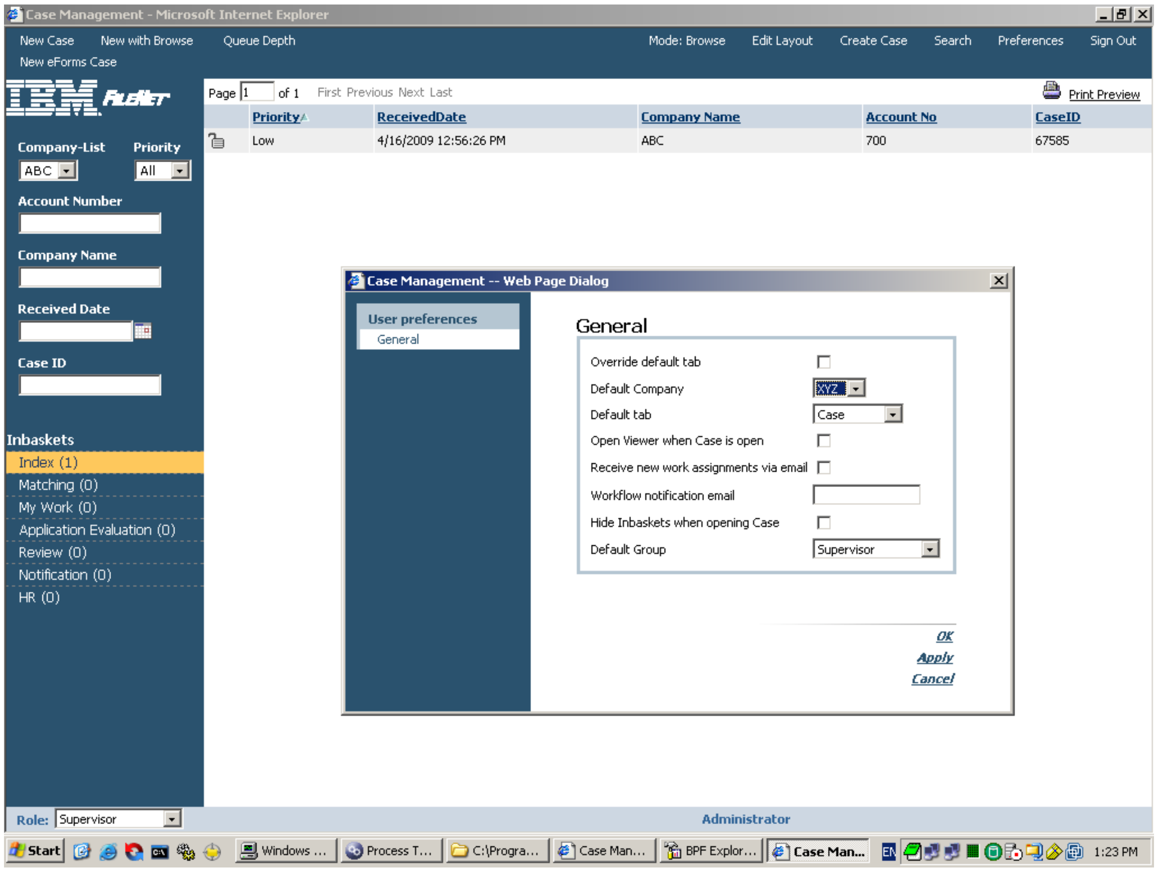
Company-List
Priority
Account Number
Company Name
ReceivedDate
Case ID

11. Now log on as an Administrator and you should see something like the following:



Priority	ReceivedDate	Company Name	Account No	CaseID
Low	4/16/2009 12:56:26 PM	ABC	700	67585

12. Now click on the **Preferences** button and change the value of the **Default Company** user preference from “ABC” to “XYZ” and click OK as detailed in the following screenshot:

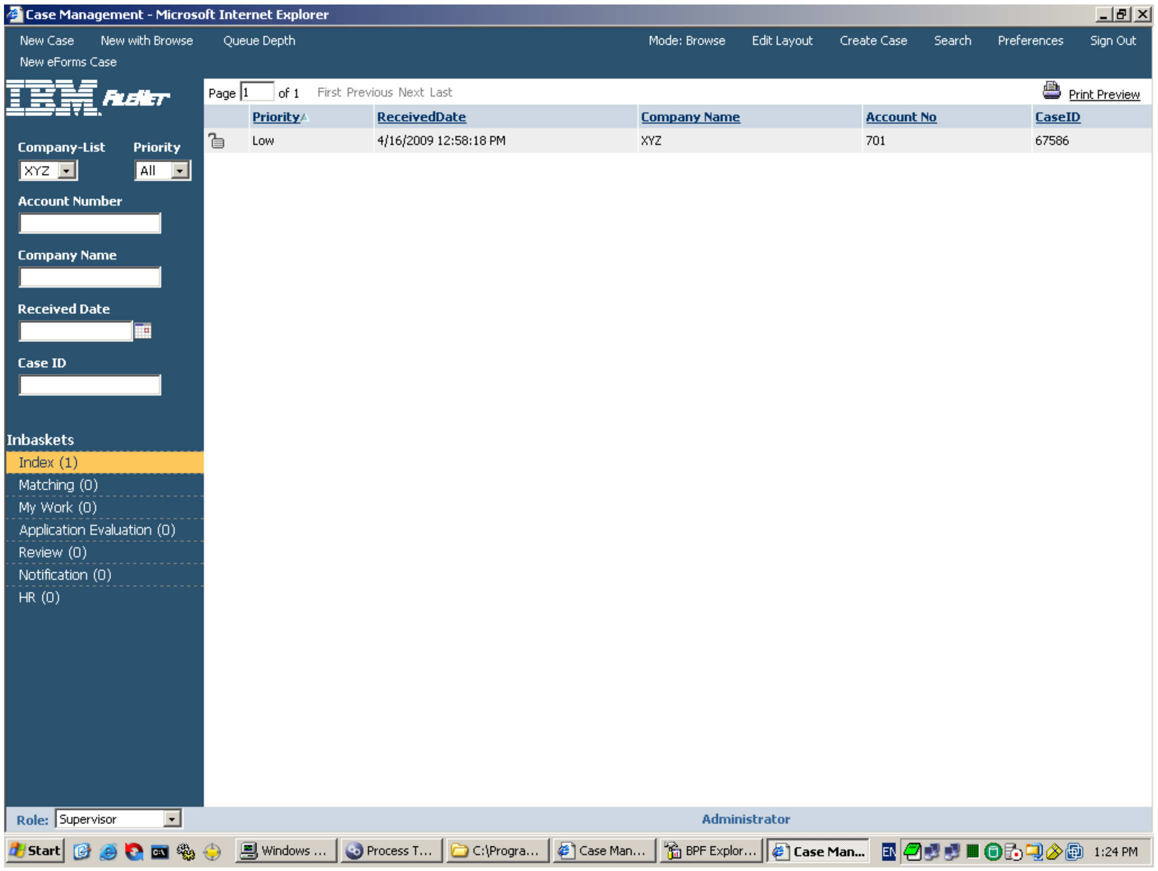


Priority	ReceivedDate	Company Name	Account No	CaseID
Low	4/16/2009 12:56:26 PM	ABC	700	67585

General

Override default tab	☐
Default Company	XYZ
Default tab	Case
Open Viewer when Case is open	☐
Receive new work assignments via email	☐
Workflow notification email	
Hide Inbaskets when opening Case	☐
Default Group	Supervisor

13. Sign out and log back in to see your new preference take effect, as follows:



Priority	ReceivedDate	Company Name	Account No	CaseID
Low	4/16/2009 12:58:18 PM	XYZ	701	67586

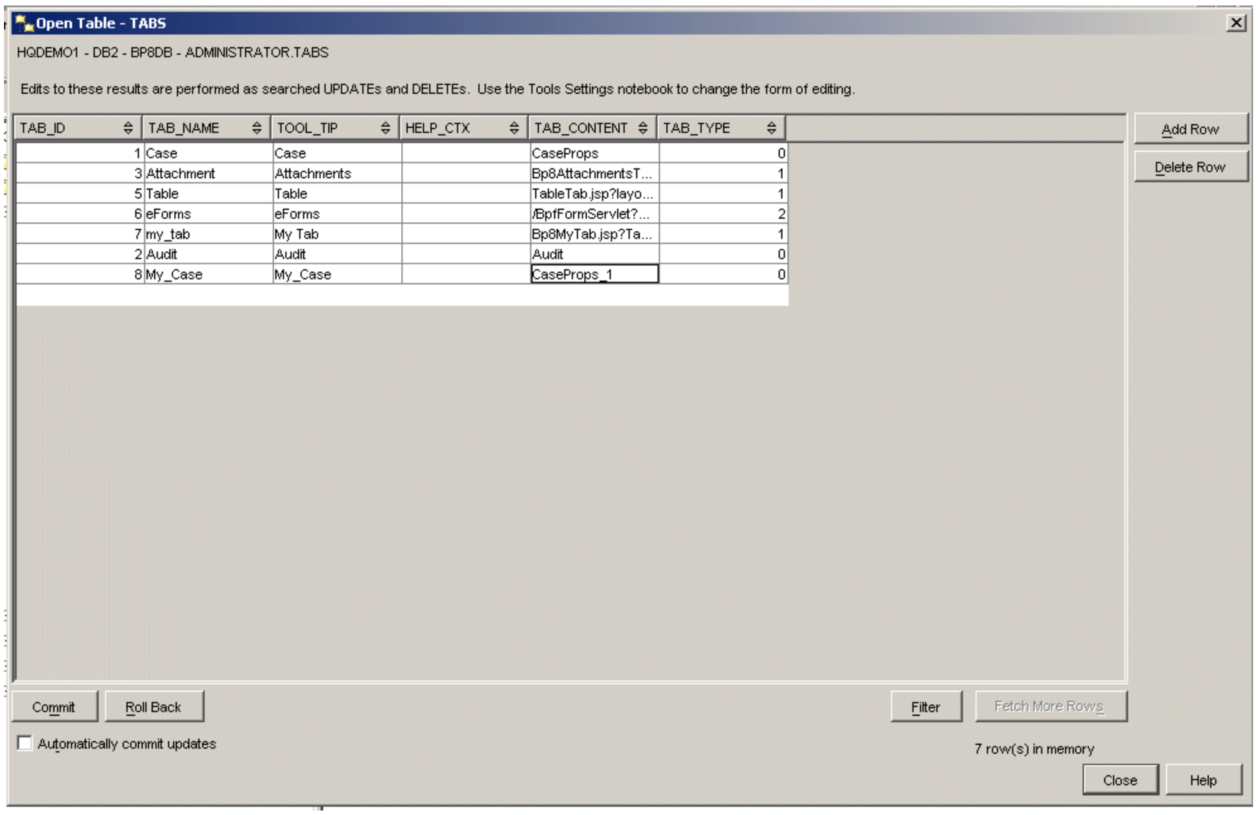
Consult the `\src\User Preference Samples` folder in the code samples for more details.

Implement multiple Case Tabs

With BPF-4.1.0-001 and above, it is possible to configure multiple Case Tabs and spread your inbasket case fields among them. Nevertheless, some configuration and custom coding (in the `EventHandlers.js::onCaseDisplay` method) is required.

The steps required are as follows.

1. Add a new row (e.g. "CaseProps_1") into the TABS table of the Bp8Metastore database (and be sure to set TAB_TYPE to 0) per the screenshot below:


Open Table - TABS

HQDEMO1 - DB2 - BP8DB - ADMINISTRATOR.TABS

Edits to these results are performed as searched UPDATEs and DELETEs. Use the Tools Settings notebook to change the form of editing.

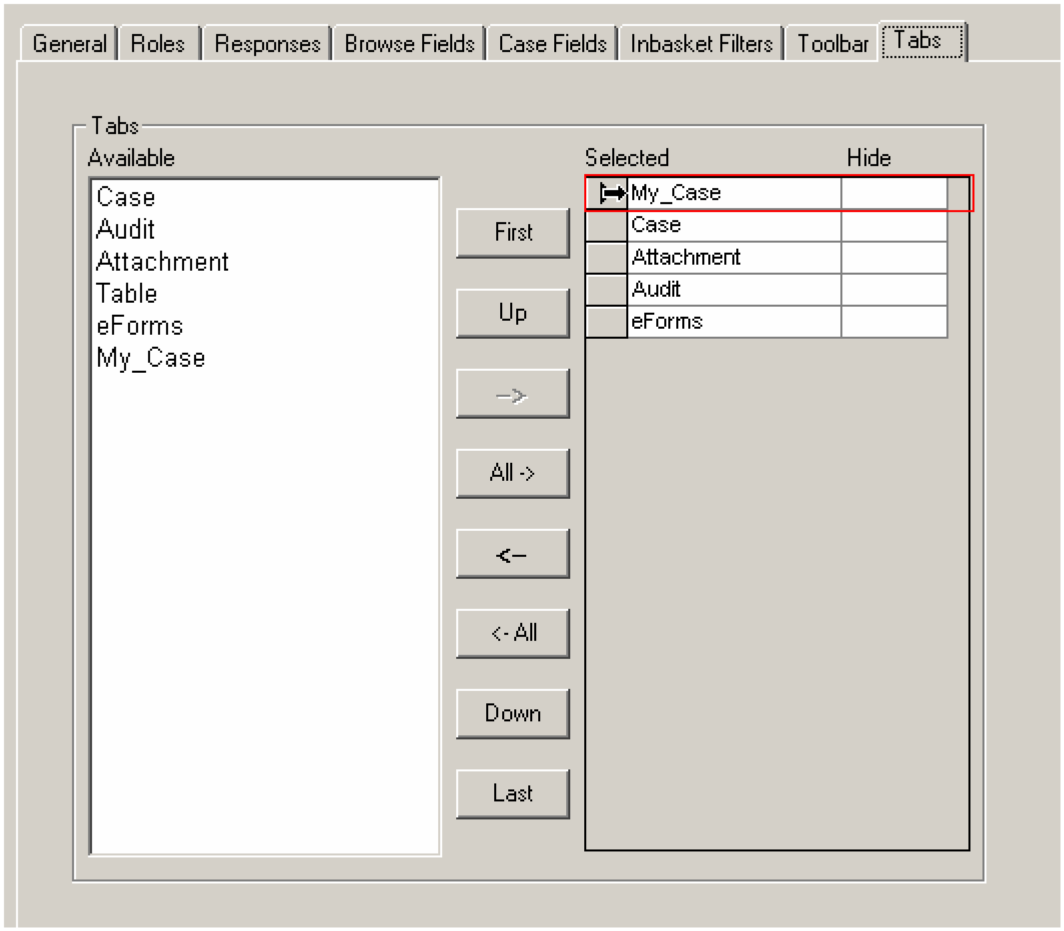
TAB_ID	TAB_NAME	TOOL_TIP	HELP_CTX	TAB_CONTENT	TAB_TYPE
1	Case	Case		CaseProps	0
3	Attachment	Attachments		Bp8AttachmentsT...	1
5	Table	Table		TableTab.jsp?layo...	1
6	eForms	eForms		/BpfFormServlet?...	2
7	my_tab	My Tab		Bp8MyTab.jsp?Ta...	1
2	Audit	Audit		Audit	0
8	My_Case	My_Case		CaseProps_1	0

Automatically commit updates

7 row(s) in memory

Adding a row to the TABS table in the Bp8Metastore is one of the few configuration options that cannot be done through the BPF Explorer, so you will have to do this using your database administration utility (such as DB2 Control Center, Oracle Enterprise Manager, or MSSQL Enterprise Manager), just as you would do when adding a custom tab.

2. Switching now to BPF Explorer, add your new Tab to the Tabs tab of the Index (Supervisor) Inbasket (or whatever inbasket you want to try this with) as follows:



3. Now edit your `EventHandlers.js` file (located in the

```
\plugins\custom
```

directory of your deployed BPF web application folder or, if you have not yet done so, rename a copy of the EventHandlers.js.template file that you will find in that location to “EventHandlers.js”. Then edit your EventHandlers.js file by adding code along the lines of the following sample:

```
eventHandlers.onCaseDisplay = function(tabContent) {

        /*-------------------------------------------------------------------*/
        /*-------------------------------------------------------------------
        If you have configured a second case tab (say "CaseProps_1") in the TABS table
        of your Bp8Metastore and added it to the Tabs tab of one or more of your
        inbaskets in BPF Explorer, then, by default, ALL case fields will appear on
        BOTH case tabs.

        So code needs to be added here to hide/show individual fields selectively
        depending on which tab is being displayed.

        For example...
          -------------------------------------------------------------------*/

        if(tabContent == 'CaseProps') {
                // If this is the original Case Tab, hide the AccountNumber field
                //      and display the CompanyName field.
                var accElement = document.getElementsByName("AccountNumber")[0];
                accElement.parentElement.parentElement.style.display = "none";

                var comElement = document.getElementsByName("CompanyName")[0];
                comElement.parentElement.parentElement.style.display = "";
        }
        else if (tabContent == 'CaseProps_1'){
                // If this is the new case tab (CaseProps_1), then show the
                //      AccountNumber field and hide the CompanyName field.
                var comElement = document.getElementsByName("CompanyName")[0];
                comElement.parentElement.parentElement.style.display = "none";

                var accElement = document.getElementsByName("AccountNumber")[0];
                accElement.parentElement.parentElement.style.display = "";
        }
        /*---------------------------------------------------------------------*/

        return true;
}
```

4. Clear your IE browser cache (so that the revised `EventHandlers.js` file will be picked up), restart the app server (so that your configuration changes will be picked up), and try opening a case from your Index (Supervisor) Inbasket (or whatever inbasket you used) to see if the new “CaseProps_1” tab appears and is functional. Verify that the fields whose show/hide behavior you modified with your custom Jscript behave as expected on your various case tabs (e.g. if you used the example above, make sure that your CompanyName field appears only on the Case Tab and the AccountNumber field appears only on the new CaseProps_1/My_Case tab).

General customizations

One of the most common customizations to the BPF Web Application is the replacement of the default GIF files found in the `<web_root>\img` directory with customer-specific GIF files of the same exact dimensions (and having the same exact names as the default BPF files). The customer-specific GIF files are then loaded by the BPF Web Application without any code changes being made whatsoever.

XSL Stylesheet modifications

There is also the option to customize the look and feel of the BPF UI by modifying the base XSL files in the

```
<bpf_web_root>\xsl
```

and

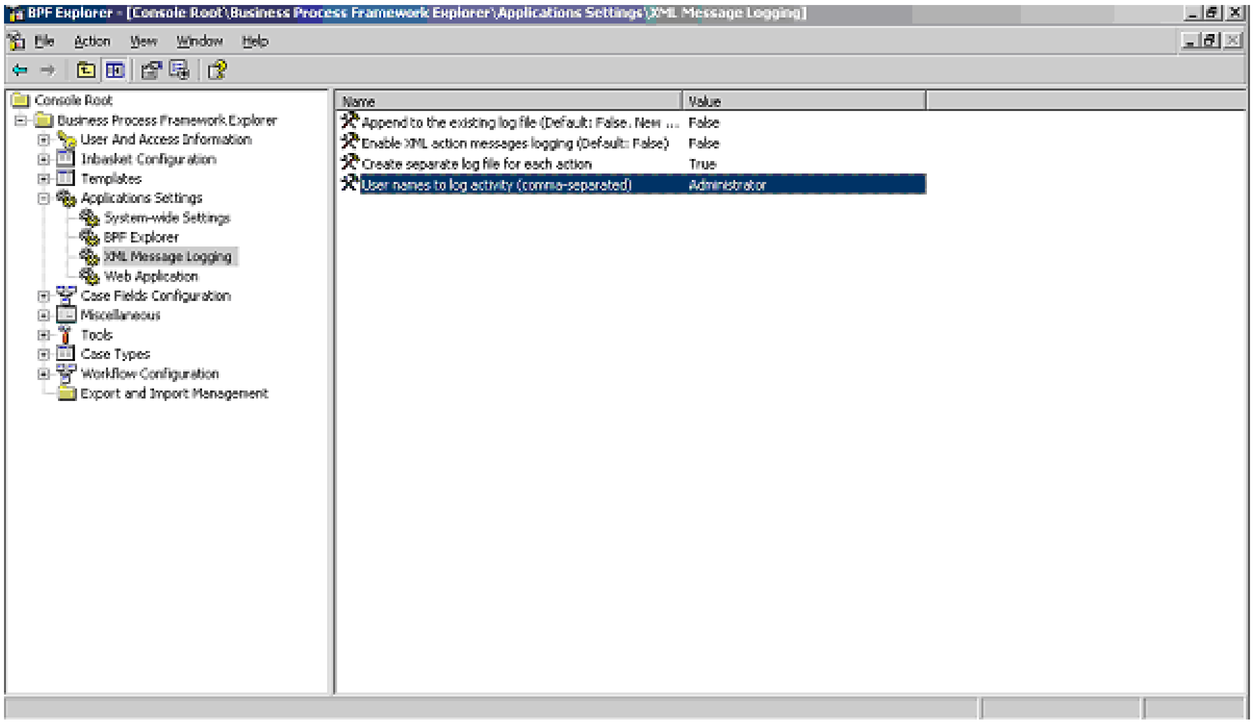
```
<bpf_web_root>WEB-INF\xsl
```

directories.

Please note, however, that unlike most of the other customization areas documented here, custom changes to these XSL files will be lost completely during any upgrade of BPF and will need to be merged back into the replaced XSL files after the upgrade.

With this warning in mind, it is possible to make extensive changes to the HTML displayed in the BPF UI by following the procedures illustrated in the following example. They, show how to modify the background and text color of the cells in the Inbasket Browse list HTML table based on the text values of the cells (Browse List fields) themselves.

The HTML displayed by the BPF UI is the result of XSLT transformations performed upon XML messages, passed from the server to the client, using XSL stylesheet files located on the web server. Because of this, you will want to turn on XML Message Logging (temporarily) using the following Application Settings values in your BPF Explorer and then restart the web server:


BPF Explorer - [Console Root\Business Process Framework Explorer\Applications Settings\XML Message Logging]
File
Action
View
Window
Help
Console Root
Business Process Framework Explorer
User And Access Information
Inbasket Configuration
Templates
Applications Settings
System-wide Settings
BPF Explorer
XML Message Logging
Web Application
Case Fields Configuration
Miscellaneous
Tools
Case Types
Workflow Configuration
Export and Import Management

Name	Value
Append to the existing log file (Default: False. New ...	False
Enable XML action messages logging (Default: False)	False
Create separate log file for each action	True
User names to log activity (comma-separated)	Administrator

(Please consult the documentation in the *IBM FileNet Business Process Framework Explorer Handbook* for more information on this feature.)

Turning on XML Message Logging will result in the creation of XML files in your

```
        <bpf_web_root>WEB-INF\logs
```

directory containing the actual XML that is being returned by the web server in response to requests from the client.

Depending on what screen within BPF you are trying to modify, you will need to look at different XML logs to see the XML that the XSL file you modify will be working with. In general, it is a good practice to navigate to the screen you want to modify and stop working, then look at the most recent XML logs created.

As with determining which XSL file to modify, some trial and error may be required in determining which XML you are dealing with on the screen you want to change. Adding some extra characters to likely candidates among the XSL files in places where they will be displayed in the UI (such as adding ZZZ immediately after a `TD` in a likely XSL file) and then bringing up the screen in BPF again to see if your change appears will tell you for certain if you are in the right place or not.

NOTE Changes to the XSL files take effect immediately and do not require restarting anything. Examining the XSL files themselves along with the JavaScript files from which the XSLT transform calls are made should also assist you in determining which XSL file you will want to change.

As an example, if we want to modify the color scheme used on the Inbasket Browse list, we will look for an XML log file named "*_`GetCases_Response_*`" (for example "`Administrator_GetCases_Response_1157575259671.xml`" or <user_name> + <action_name> + "Request" or "Response" + <timestamp> + ".xml).

Determining the XML you are working with will also help you determine which XSL file you need to modify because the XSL will typically reference the same tag names as occur in the XML that it is working with.

In the case of our example, we can, using the methods described above, determine that the XML in our

```
        <bpf_web_root>\WEB-INF\logs\Administrator_GetCases_Response_1157575259671.xml
```

file is being transformed using the

```
        <bpf_web_root>\xsl\Bp8Cases.xsl
```

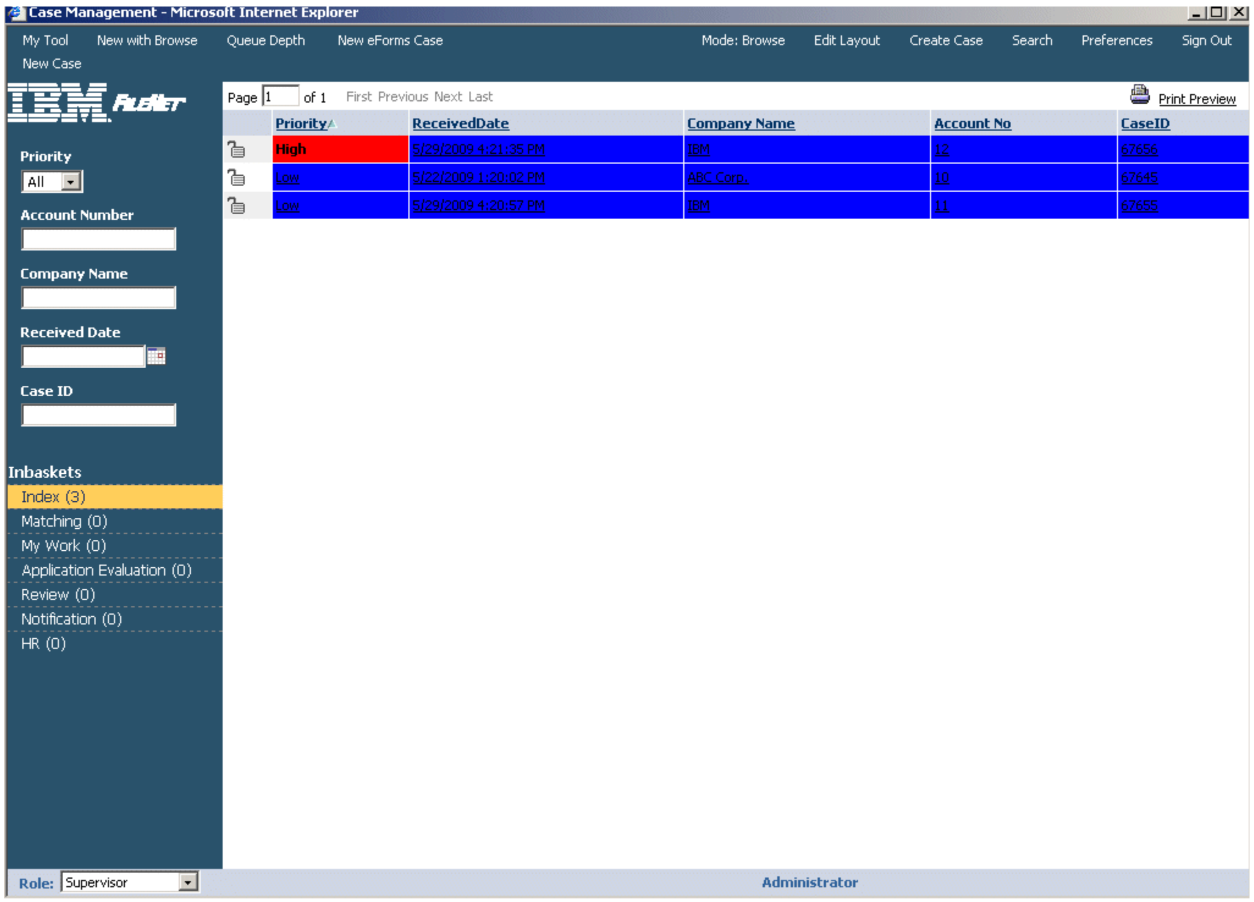
file. Opening this XSL file, you will see that it contains XSL to create both the Headers row of the Inbasket Browse HTML table and the subsequent data rows for individual Cases. Since we want to try changing the color scheme of the cells in the data rows containing individual Cases, we will focus on the following section of Bp8Cases.xsl, changing it from

```
<xsl:for-each select="Value[@Visible='1']"<td class="tblData" valign="top" align="left"
nowrap="true">
```

to

```
<xsl:for-each select="Value[@Visible='1']">
        <td class="tblData" valign="top" align="left" nowrap="true">
                <xsl:choose>
                        <xsl:when test=".[.='High']">
                                <xsl:attribute name="bgcolor">red</xsl:attribute>
                                <xsl:attribute name="style">font-weight:bold</xsl:attribute>
                        </xsl:when>
                        <xsl:otherwise>
                                <xsl:attribute name="bgcolor">blue</xsl:attribute>
                                <xsl:attribute name="style">text-
decoration:underline</xsl:attribute>
                        </xsl:otherwise>
                </xsl:choose>
```

to achieve the following effect:



	Priority	ReceivedDate	Company Name	Account No	CaseID
	High	5/29/2009 4:21:35 PM	IBM	12	67656
	Low	5/22/2009 1:20:02 PM	ABC Corp.	10	67645
	Low	5/29/2009 4:20:57 PM	IBM	11	67655

or to

```
<xsl:for-each select="Value[@Visible='1']">
        <td valign="top" align="left" nowrap="true">
                <xsl:choose>
                        <xsl:when test=".[.='High']">
                                <xsl:attribute name="class">my_custom_class1</xsl:attribute>
                        </xsl:when>
                        <xsl:otherwise>
                                <xsl:attribute name="class">my_custom_class2</xsl:attribute>
                        </xsl:otherwise>
                        </xsl:choose>
```

to reference custom CSS style classes that would/could be defined in `Bp8CustomStyles.css` as follows

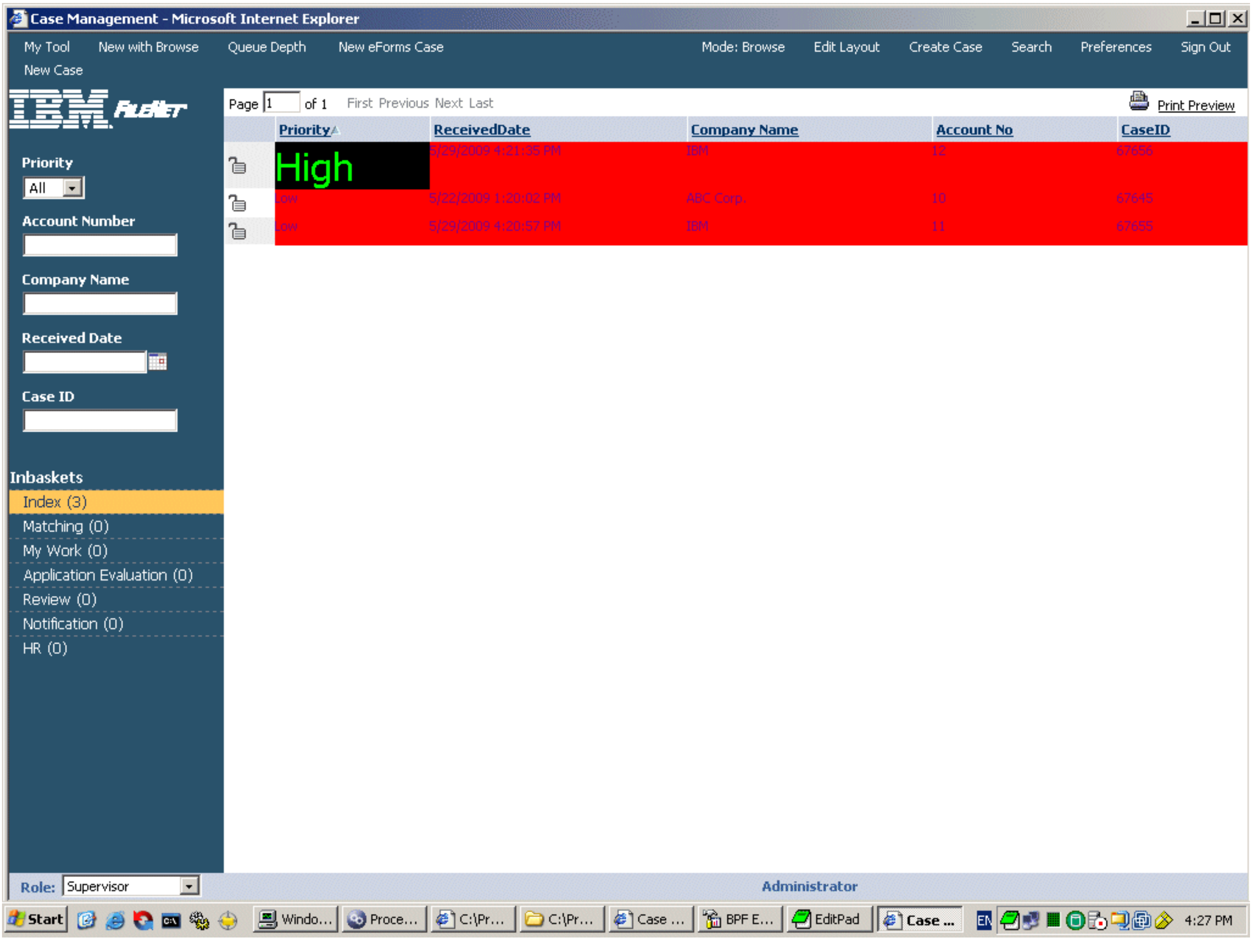
```
<style type="text/css">

body {
        /*---*/
}

.my_custom_class1 {
    font-size:        24pt;
    color:            lime;
    background-color: black;
}

.my_custom_class2 {
    font-size:        8pt;
    color:            purple;
    background-color: red;
}
</style>
```

to produce the following effect:



Priority	ReceivedDate	Company Name	Account No	CaseID
High	5/29/2009 4:21:35 PM	IBM	12	67656
Low	5/22/2009 1:20:02 PM	ABC Corp.	10	67645
Low	5/29/2009 4:20:57 PM	IBM	11	67655

Of course, you may run into problems where it is not possible to achieve what you have in mind by changing the XSL alone without also changing the XML itself. This cannot be done through customization.

Notices

Trademarks

Program Number: 5724-R75

Printed in USA

GC31-5514-02