Oracle® MICROS Simphony

HTML5 Extensibility Developer Reference Guide

Release 19.3
F46312-02
July 2022

Oracle Hospitality Simphony HTML5 Extensibility Developer Reference Guide Release 19.3

F46312-02

Contents

Preface

This release of Oracle MICROS Simphony includes an extensibility API for developing HTML5 dialogs that work across all platforms.

Purpose

This document is a developer reference guide for HTML5 user interface development for Oracle MICROS Simphony, including API details, reference material, code samples, and developer guidance.

Audience

This document is intended for developers who design custom dialogs for applications that extend core Oracle MICROS Simphony functionality.

Customer Support

To contact Oracle Customer Support, access My Oracle Support at the following URL:

https://support.oracle.com

When contacting Customer Support, please provide the following:

- Product version and program/module name
- Functional and technical description of the problem (include business impact)
- Detailed step-by-step instructions to re-create
- Exact error message received and any associated log files
- Screenshots of each step you take

Documentation

Oracle Hospitality product documentation is available on the Oracle Help Center at http://docs.oracle.com/en/industries/food-beverage/

Revision History

Date	Description of Change
September 2021	Initial publication.
July 2022	Updated Android information in Rendering Technology chapter.

1 Introduction

Traditionally, Simphony extensibility (SIM/C#) had very primitive user interface functionality. The SIM Window feature is limited to form entry and C# has no built-in user interface functionality.

Microsoft Windows extensibility applications can use .NET forms/WPF to build complex user interfaces, but these technologies are not available for Linux or Android.

Simphony 19.2 provides an extensibility API for displaying complex dialogs across all platforms.

The technology to achieve this is HTML5 running isolated in an iframe or a separate browser instance. The benefits of this design are:

- HTML5 is a standard and familiar technology, with an industry wide population of HTML5 capable developers.
- HTML5 renders rich animated user interface experiences.
- HTML5 is platform-independent. HTML5 dialogs render identically across Microsoft Windows, Android, and Linux.
- Isolating the HTML5 dialog gives the extensibility a safer sandbox to execute code.
- The renderer is CEF (Chromium Embedded Framework). This technology is standard for most browsers.

2 API Basics

The Custom Dialog API provides the following functionalities:

- Ext app (extensibility application) calls OpsContext. ShowExtensibilityHtmlDialog () to show the dialog window.
- The HTML5 passes in as a string or a compressed archive.
- The HTML5 app needs the JavaScript API to do anything meaningful. The API provides methods for closing the dialog, interacting with the ext app, and keyboard functionality.
- The HTML5 app optionally includes a CSS resource to ensure that the dialog style matches the POS theme.
- Images in the Content/Extension Application Content are referenced in the HTML5 document using the img element and a special access syntax for the source.
- The HTML5 app executes methods indirectly on the ext app.
- Custom dialog nesting.

3 Code Samples

Click Sample HTML5 Dialog Extension to download a sample application that demonstrates how to code a dialog with C#, SIM, and HTML5. Most of the code here (HTML5, C#, SIM) is available in complete form in the sample application.

Most of the samples demonstrate a particular aspect of the feature with a minimal amount of code. Because of this, there is less error checking and the HTML5 is minimal.

> **NOTE:**
>
> Copy and paste the code from the sample application and not from this document. The code in this document is not self-contained; only enough code is provided to illustrate the implementation.

.

4 Cross-Platform Development

The API is the same across Microsoft Windows, Linux, and Android.

When choosing a language (C# or SIM), consider the support details listed in the following table:

	Microsoft Windows	**Linux**	**Android**
C#	X	X	Not supported
SIM	X	X	X

Android does not support .NET assemblies. If your application must support all platforms, SIM is the only choice.

5 Rendering Technology

Simphony uses Chromium Embedded Framework (CEF) across all platforms for rendering the HTML5 application. CEF is the world standard for browser rendering and it allows extensibility developers to use the latest available HTML5 features. An added benefit is that dialogs render identically across platforms.

Microsoft Windows and Linux use the same CEF version (currently set to 85.0.4183.102). Android uses the version of the browser installed on the device. On all platforms, the chromium version writes to the egateway log at startup.

HTML5 dialogs on the POS have different containers based on the platform:

- Linux, Android:
  - A full-screen iframe to house the dialog HTML5.
  - The iframe background is transparent.
- Microsoft Windows:
  - A full-screen WPF dialog containing a browser control (CEF).
  - The dialog window is transparent.

Make sure the dialog is full-screen, transparent, and centered on the screen.

Android Considerations

The Custom HTML dialog feature is only supported for Chrome/CEF versions 85 and higher.

The Oracle workstation base images for Windows and Linux come with their own CEF browser control. The shipped version is currently 85, and as of this writing, the current *release* Chrome version is 103.

Android devices come with their own Chrome/CEF instance. The Service Host APK takes advantage of the built-in Android browser. As such, it is possible that the Android Chrome/CEF version is *below* 85. The extensibility writer should do the following:

1. Add Chrome/CEF version checking in the dialog javascript to validate that the version is correct.
2. Ensure that the customers' devices are at the appropriate version.

Please include this code, or some variation of it, in *all* custom dialogs.

Javascript

```
// your onload initialization code.
function onload()
{
    // call this first, before accessing the API
    validateRuntime();
}
```

```
function validateRuntime()
{
    let valid = true;

    // validate chrome/cef version
    let minCefVersion = 85;
    let cefVersion = getChromeCefVersion();
    if (cefVersion == '?' || (cefVersion < minCefVersion))
    {
        alert('Chrome/CEF version below ' + minCefVersion + ', please
upgrade.\nCalculated version: ' + cefVersion + '\nUser agent: ' +
navigator.userAgent);
        valid = false;
    }

    // validate api object valid
    if (typeof SimphonyKioskAPI == 'undefined')
    {
        alert('SimphonyKioskAPI undefined. Please verify Chrome/CEF
version is correct.');
        valid = false;
    }

    return valid;
}

function getChromeCefVersion()
{
    try
    {
        let uaParts = navigator.userAgent.split(' ');
        for (let i = 0; i < uaParts.length; i++)
        {
            if (uaParts[i].startsWith("Chrome"))
            {
                let parts = uaParts[i].split('/');
                let vers = parts[1].split('.');
                return vers[0];
            }
        }
    }
    catch (ex)
    {
        log('error getting chrome version: ' + ex);
    }
    return '?';
}
```

6 Raising a Dialog

Raise a dialog as follows:

- C#: Call the OpsContext.ShowExtensibilityHtmlDialog() method.
- SIM: Call the WaitForHtmlDialog command.

OpsContext.ShowExtensibilityHtmlDialog() is synchronous, similar to the WPF ShowDialog() method. Control returns to the extensibility application when the dialog is closed.

In SIM, the WaitForHtmlDialog returns immediately, similar to the WaitForRxMsg command.

The ext app expects to supply the HTML5/JavaScript/CSS. This is accomplished in two ways:

1. Provide a simple string: in-place html.
2. Provide a zip file in the form of a binary array: web directory.

In-Place HTML

In the follow code sample, the HTML reads from the extension application content table. It can also be read from an embedded resource in the assembly (if C#) or constructed in place.

C#

```
[ExtensibilityMethod]
public void RaiseInPlaceDialog()
{
    var parms = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHtmlDial
ogParameters();
    parms.HTML =
DataStore.ReadExtensionApplicationContentTextByNameKey( OpsContext.RvcI
D, ApplicationName, "html" );
    string result = OpsContext.ShowExtensibilityHtmlDialog( parms );
    OpsContext.ShowMessage( result );
}
```

SIM

```
event inq:1
    var parms:object = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHt
mlDialogParameters()
    parms.HTML =
@DataStore.ReadExtensionApplicationContentTextByNameKey(
@OpsContext.RvcID, @ApplicationName, "html" )
    WaitForHtmlDialog parms, "SampleDialogClosed"
endevent

event dialog:SampleDialogClosed
    @OpsContext.ShowMessage( @HtmlDialogResult )
endevent
```

Web Directory

A more versatile method of supplying the dialog HTML5 is through the “web directory dialog”. The custom dialog HTML5 passes in as a ZIP file (byte array). The ZIP file usually reads from extension application content, but it can also be an embedded resource in the ext app assembly.

C#

```
[ExtensibilityMethod]
public void RaiseWebDirDialog()
{
    var parms = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityWebDirectoryDia
logParameters();
    parms.WebDirectoryZip =
DataStore.ReadExtensionApplicationContentDataByNameKey( OpsContext.RvcI
D, ApplicationName, "webdir" );
    string result = OpsContext.ShowExtensibilityHtmlDialog( parms );
    OpsContext.ShowMessage( "Dialog returned: " + result );
}
```

SIM

```
event inq:2
    var parms:object = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityWebDirect
oryDialogParameters()
    parms.WebDirectoryZip =
@DataStore.ReadExtensionApplicationContentDataByNameKey(
@OpsContext.RvcID, @ApplicationName, "webdir" )
    parms.StartResource = "index.html"
    WaitForHtmlDialog parms, "SampleDialogClosed"
endevent

event dialog:SampleDialogClosed
    @OpsContext.ShowMessage( @HtmlDialogResult )
endevent
```

7 Example Dialog HTML5

Simple Centered Dialog

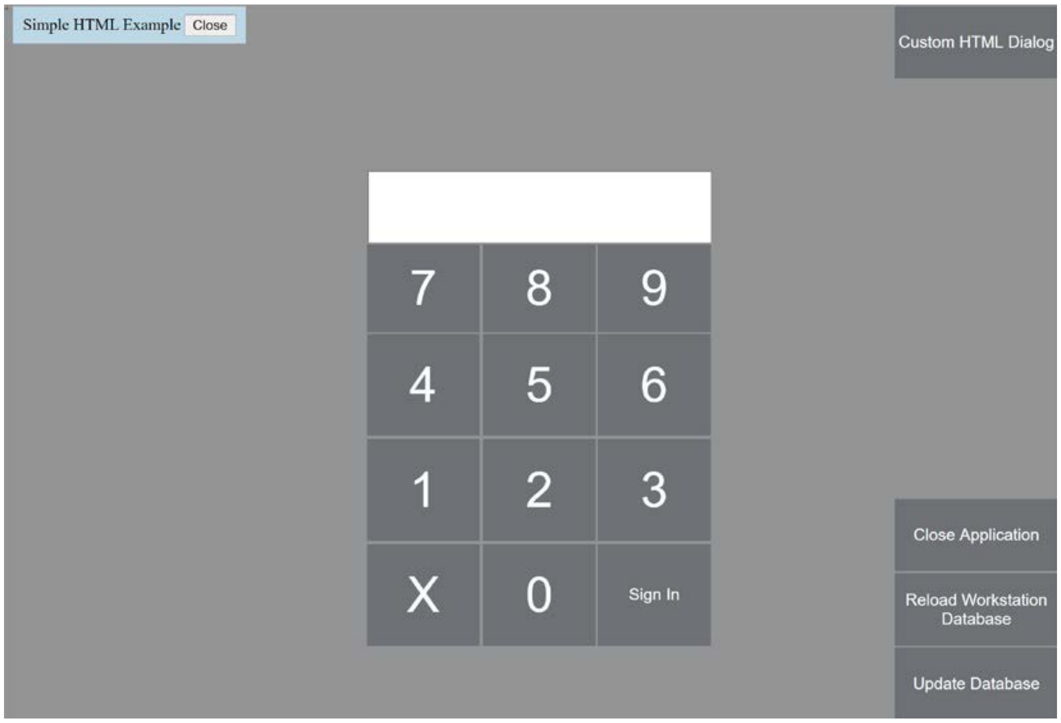
This is an example of a simple HTML5 dialog:

- The <script> tag is used to include the Simphony POS API.
- The close button calls the API to close the dialog.

This dialog appears at the top left of the full screen, and the container is transparent.

HTML (SampleMinimal.html)

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <script src="Simphony/POSDialogAPI.js" charset="UTF-8"></script>
</head>
<body>
   <span style="background-color:lightblue; padding: 10px; width: auto;">
      Simple HTML Example
      <button onclick="SimphonyPOSAPI.closeDialog()">Close</button>
   </span>
</body>
</html>
```



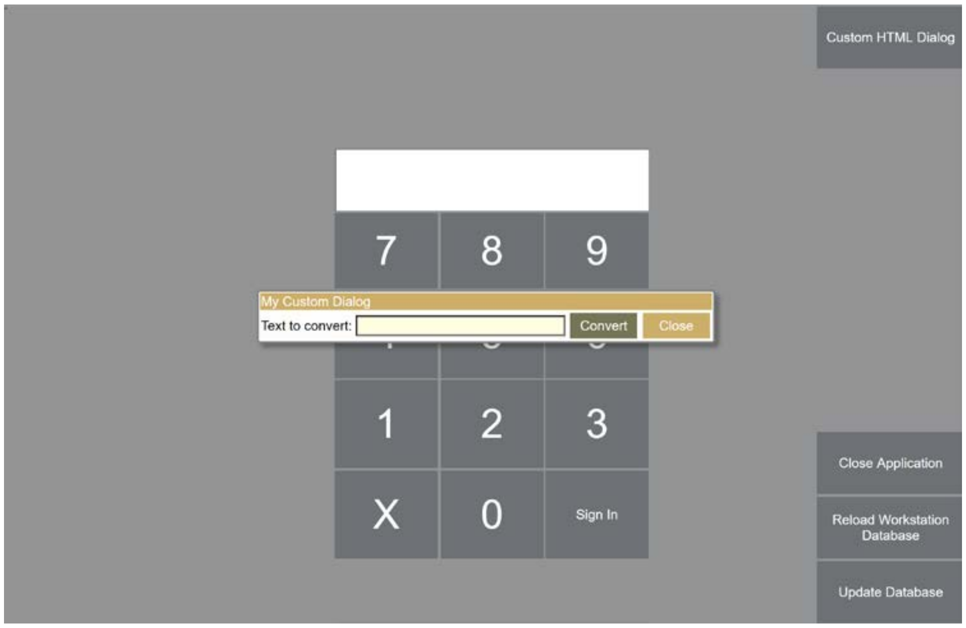
Dialog Styling

The purpose of styling is to convert text in an input box to upper case, as well as return the contents of the input box to the extensibility application. This section explains how to:

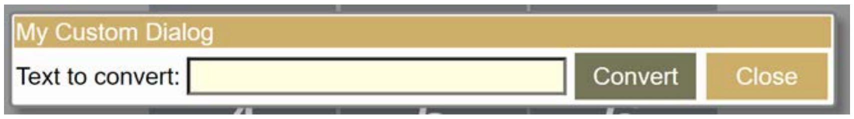
- Style dialog elements to match the Simphony style.
- Center the dialog.
- Return a result to the extensibility app.

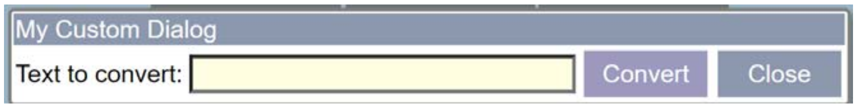
HTML (SampleStyling.html)

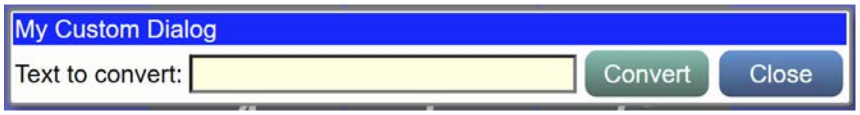
```
<html>
<head>
    <link href="Simphony/POSTheme.css" rel="stylesheet" type="text/css"
/>
    <script src="Simphony/POSDialogAPI.js"></script>

    <style>
        button {
            min-height: 30px;
            min-width: 80px;
        }
    </style>

    <SCRIPT LANGUAGE='JavaScript'>
        function closeClick()
        {
            let elem = document.getElementById('myTextInput');
            SimphonyPOSAPI.closeDialog(elem.value);
        }
        function convert()
        {
            let elem = document.getElementById('myTextInput');
            elem.value = elem.value.toUpperCase();
        }
    </SCRIPT>
</head>

<body onload="onload()">
    <div class="SimphonyPOSDialogWindow SimphonyPOSCenterWindow">
        <table border="0">
            <tr>
                <td class="SimphonyPOSDialogTitleBar">My Custom
Dialog</td>
            </tr>
            <tr>
                <td>
                    Text to convert:
                    <input type="text" id="myTextInput"
style="width:250px; background-color: lightyellow;" />
                    <button class="SimphonyPOSDialogFunctionButton"
onclick='convert()'>Convert</button>
                    <button class="SimphonyPOSDialogControlButton"
onclick='closeClick()'>Close</button>
                </td>
            </tr>
        </table>
    </div>
</body>
</html>
```



This dialog will be styled based on the current theme:







Notes

1. The application can call the API closeDialog() function with optional data. This data is returned to the ext app from the RaiseBrowserDialog() method. The data must be a string. If you pass back a complicated object, use JSON.stringify().
2. The styling elements are:
   a. SimphonyPOSDialogWindow: styles the window foreground and background colors, the dialog ‘shading’, and the border.
   b. SimphonyPOSCenterWindow: styles the window to center within the screen.
   c. SimphonyPOSDialogTitleBar: styles a title bar based on the current theme.
   d. SimphonyPOSDialogControlButton, SimphonyPOSDialogFunctionButton: two buttons styles to match the POS dialog theme. The control button style is usually for buttons that close the dialog, the function buttons are for dialog functionality.

Image References

Images can be static or dynamically retrieved.

A web directory can contain an img folder with all relevant images. The <img> tag src attribute provides the reference as follows:

```
<img src='img/myimage.jpg'/>
```

You can use one of the following methods to dynamically reference an image:

- From Extension Application Content.
- From Content.
- From the extensibility application.

To reference database images, the Simphony/Database virtual directory pulls the image directly from the database without any extra coding. Ext app images require the application to first subscribe to the Resource Not Found event and then monitor the resources requests.

The resource not found event is useful for these reasons:

- Supplies a dynamically generated image from code.
- Intercepts unknown resource requests for debugging. For example, if a resource reference is misspelled, it is intercepted.

HTML (SampleImages.html)

```
<html>
<head>
    <link href="Simphony/POSTheme.css" rel="stylesheet" type="text/css"
/>
    <script src="Simphony/POSDialogAPI.js"></script>
    <style>
        button {
            min-height: 30px;
            min-width: 80px;
        }

        img {
            width: 100px;
            height: 50px;
        }
    </style>
</head>

<body>
    <div class="SimphonyPOSDialogWindow SimphonyPOSCenterWindow">
        <table>
            <tr>
                <td class="SimphonyPOSDialogTitleBar">My Custom
Dialog</td>
            </tr>
            <tr>
                <td>
                    <!-- image supplied by ext app -->
                    <img src="a/b/flag-a.jpg"/>
                    <!-- image read from ext app content -->
                    <img
src="Simphony/Database/ExtensionAppContent/flag-b.jpg"/>
                    <!-- image read from content -->
                    <img src="Simphony/Database/Content/flag-c.jpg"/>
                </td>
            </tr>
            <tr>
                <td align="right">
                    <button class="SimphonyPOSDialogControlButton"
onclick='SimphonyPOSAPI.closeDialog()'>Close</button>
                </td>
            </tr>
        </table>
    </div>
</body>
</html>
```

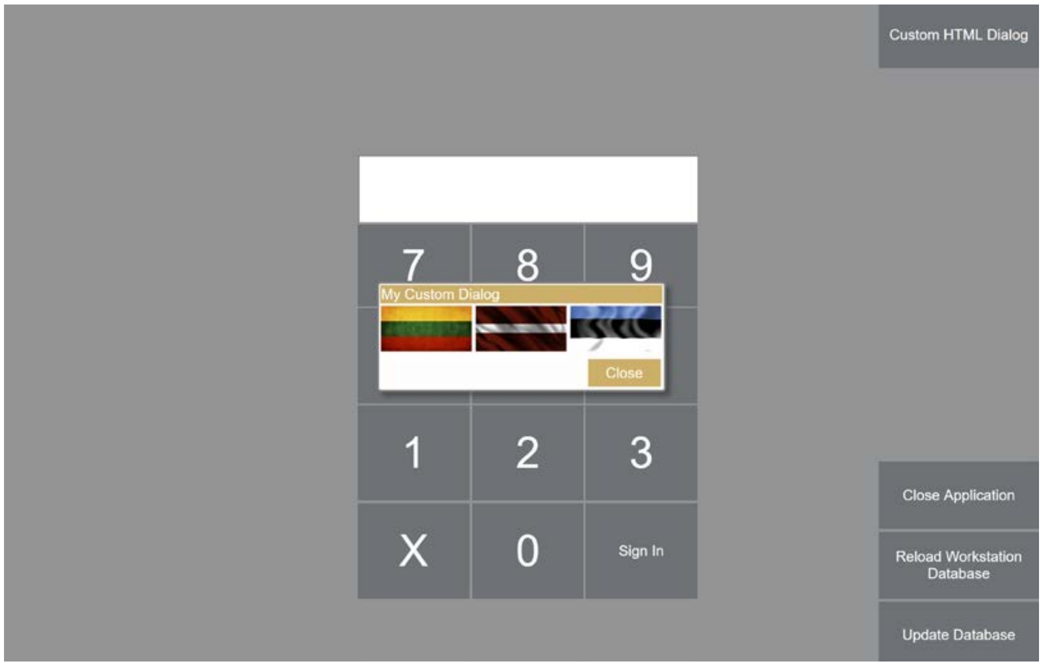
C#

```
public CustomHTMLDialogApp( IExecutionContext context )
    : base( context )
{
    OpsResourceNotFoundEvent +=
CustomHTMLDialogApp_OpsResourceNotFoundEvent;
}

private EventProcessingInstruction
CustomHTMLDialogApp_OpsResourceNotFoundEvent( object sender,
OpsResourceNotFoundEventArgs args )
{
    if ( args.URL == "a/b/ext-flag-a.jpg" )
    {
        args.ResourceSupplied = true;
        args.ResourceExtension = "jpg";
        args.BinaryResource =
DataStore.ReadExtensionApplicationContentDataByNameKey( OpsContext.RvcI
D, ApplicationName, "flag-a.jpg" );
    }
    return EventProcessingInstruction.Continue;
}
```

SIM

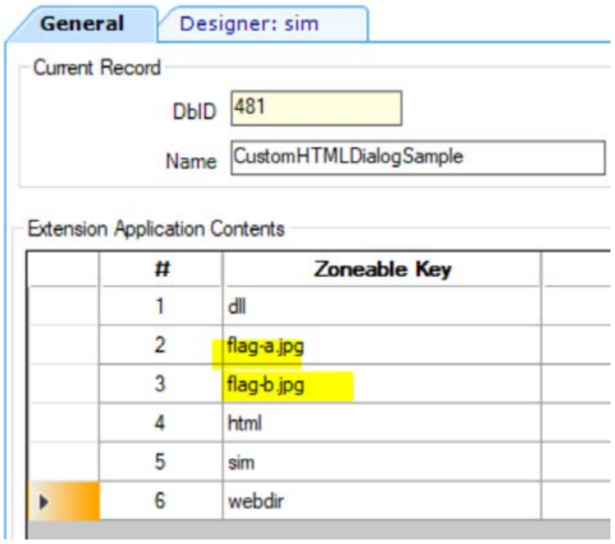
```
event init
    SubscribeToEvent "OpsResourceNotFoundEvent",
"ResourceNotFoundEvent"
endevent

event ResourceNotFoundEvent( var sender:object, var args:object)
    if args.URL = "a/b/flag-a.jpg"
        args.ResourceSupplied = 1
        args.ResourceExtension = "jpg"
        args.BinaryResource =
@DataStore.ReadExtensionApplicationContentDataByNameKey(
@OpsContext.RvcID, @ApplicationName, "flag-a.jpg" )
    endif
```

```
endevent
```



Notes

- The API reserves the Simphony root directory for API-specific resources. This includes JavaScript, CSS, and image references. Do not create a Simphony directory in your application because the API ignores it.
- There are two images stored in this database, one in the Extension application module and the other in the Content module. The image name (flag-a.png) must match the content name. Include the extension in the name.
    - The Extension Application module.



Current Record

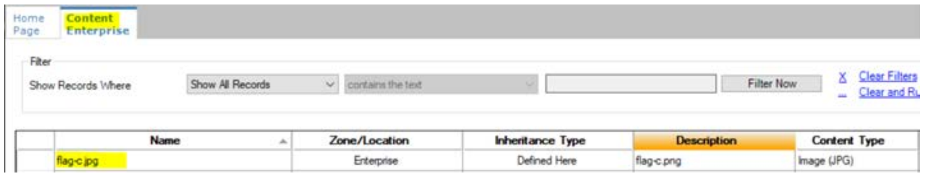
DbID: 481

Name: CustomHTMLDialogSample

Extension Application Contents

	#	Zoneable Key
	1	dll
	2	flag-a.jpg
	3	flag-b.jpg
	4	html
	5	sim
▶	6	webdir

    - Content module



	Name	Zone/Location	Inheritance Type	Description	Content Type
	flag-c.jpg	Enterprise	Defined Here	flag-c.png	Image (JPG)

- Any type of resource can be stored in content: images, CSS, JavaScript.
- This sample reads the image from the extension app content but it can read from an embedded resource.
- If you do not subscribe to this event, unknown resources are ignored.

8 Extensibility Callbacks

This feature has the ability for the dialog JavaScript to call into the C# extensibility app while the dialog is running.

High-level overview:

- The following options are available for calling C# code from JavaScript:
  - Extensibility event
  - Extensibility method
  - Context method.
- String data passes to the C# app and string data returns to the JavaScript app.

The result types from a call are success, error, and timeout. All types are functionally equivalent. The differences between the types are stylistic and provide different levels of abstraction for the extensibility application.

Extensibility Event

Use the following JavaScript to raise a custom dialog event in the extensibility application:

JavaScript

```javascript
function convertThroughExtEvent()
{
    let parms = new SimphonyPOSAPI_ExtensibilityEventParameters();
    parms.sender = 'MainDialog';
    parms.command = 'ToUpper';
    parms.argument = document.getElementById('textInput').value;

    SimphonyPOSAPI.runExtensibilityEvent(parms,
        (parmsOut) => document.getElementById('textInput').value =
parmsOut.response);
}
```

- ExtensibilityEventParameters must be allocated; do not create a simple Object.
- SimphonyPOSAPI.runExtensibilityEvent() is asynchronous and must be passed as a callback to process the response. If no processing is required, null can be passed in.
- The callback receives an object, which then indicates success, error, or timeout.
- The callback object response contains the string value returned from the extensibility application.

The code to process this event is:

C#

```
public CustomHTMLDialogApp( IExecutionContext context )
    : base( context )
{
    OpsCustomDialogEvent += CustomHTMLDialogApp_OpsCustomDialogEvent;
}

private EventProcessingInstruction
CustomHTMLDialogApp_OpsCustomDialogEvent( object sender,
OpsCustomDialogEventArgs args )
{
    switch ( args.Command )
    {
        case "ToUpper":
            args.Response = args.Argument.ToUpper() + " from ext
event";
            break;
    }
    return EventProcessingInstruction.Continue;
}
```

SIM

```
event init
    SubscribeToEvent "OpsCustomDialogEvent", "CustomDialogEvent"
endevent

event CustomDialogEvent( var sender:object, var args:object )
    if args.Command = "ToUpper"
        args.Response = args.Argument.ToUpper() && " from sim dlg
event"
    endif
endevent
```

This is the classic extensibility pattern for shuttling all events into one method. The disadvantage of this pattern is that the switch() statement will eventually become very complicated.

Extensibility Method

Use the following code to call an individually named extensibility method in both SIM and C#.

JavaScript

```
function convertThroughExtMethod()
{
    let parms = new SimphonyPOSAPI_ExtensibilityMethodParameters();
    parms.dllName = "CustomHTMLDialogSample.dll";
    parms.method = 'ToUpper';
    parms.argument = document.getElementById('textInput').value;
    SimphonyPOSAPI.runExtensibilityMethod(parms,
          (parmsOut) => document.getElementById('textInput').value =
parmsOut.response );
}
```

This code is similar to calling an extensibility event. In SIM, the 'DllName' property is ignored.

C#

```
[ExtensibilityMethod]
public object ToUpper( object arg )
{
    var text = arg as string;
    return text.ToUpper() + " from ext method";
}
```

SIM

```
event extensibility_method : ToUpper
    var emd:object = @ExtensibilityMethodArgs
    emd.Response = emd.Argument.ToUpper() && " from sim ext
method"
endevent
```

This pattern is more useful for isolating code.

Context Method

The context method provides a more useful pattern when the ext app contains multiple dialogs with many states to initialize. It is similar to the extensibility method but instead calls the method on a specific user-supplied .NET object rather than the extension app.

JavaScript

```
function convertThroughContextMethod()
{
    let parms = new SimphonyPOSAPI_ContextMethodParameters();
    parms.Method = 'ToUpper';
    parms.Argument = document.getElementById('textInput').value;
    SimphonyPOSAPI.runContextMethod(parms,
         (parmsOut) => document.getElementById('textInput').value =
parmsOut.response );
}
```

C#

```
[ExtensibilityMethod]
public void RaiseInPlaceDialog()
{
    var parms = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHtml
DialogParameters();
    parms.HTML =
DataStore.ReadExtensionApplicationContentTextByNameKey( OpsContext.
RvcID, ApplicationName, "html" );
    parms.Sender = "CustomHTMLDialogSample-SampleApp";
    parms.Context = new DialogContext();
    string result =
OpsContext.ShowExtensibilityHtmlDialog( parms );
}

public class DialogContext
{
    public object ToUpper( object arg )
    {
        var text = arg as string;
        return text.ToUpper() + " from content method";
    }
}
```

SIM

SIM does not support the Context method

Keep the following in mind:

- The context for the dialog is set in the parameters. It can be any object.
- The [ExtensibilityMethod] attribute is not required.

Consider an ext app with multiple dialogs, each having multiple callbacks. The code would look like:

```
Dialog1_FuncA {}
Dialog1_FuncB {}
Dialog1_FuncC {}

Dialog2_FuncA {}
Dialog2_FuncB {}

Dialog3_FuncA {}
Dialog3_FuncB {}
Dialog3_FuncC {}
Dialog3_FuncD {}
```

The code for all three dialogs is mixed in with each other and initialization can be easy to overlook.

If each dialog has its own context object, the code can be more easily partitioned, isolated, and initialized:

```
class Dialog1
{
    A() {}
    B() {}
    C() {}
}

class Dialog2
{
    A() {}
    B() {}
}

class Dialog3
{
    A() {}
    B() {}
    C() {}
    D() {}
}
```

In a sense, this pattern mirrors a WPF dialog, which has XAML and code-behind. In this case, we use HTML5 and the code-behind is its own object.

Which to Use?

Use either the extensibility methods or context methods. If an ext app has one dialog and that dialog does not have many callbacks, the extensibility methods are the most useful.

If the ext app raises multiple dialogs, each with its own state and methods, the context method is preferable.

Results Status

Each method call type returns a method response object. This object contains the status of the call (success, timeout, error) and the returned string data.

Status	Usage	Response value
SUCCESS	`if (parmsOut.isSuccess())`	The string value from the method call
ERROR	`if (parmsOut.isError())`	Error text generate by the system indicating the error.
TIMEOUT	`if (parmsOut.isTimeout())`	None

The following method will process a response from the API. All three conditions are tested (success, error, timeout) and error handling is up to the user.

JavaScript

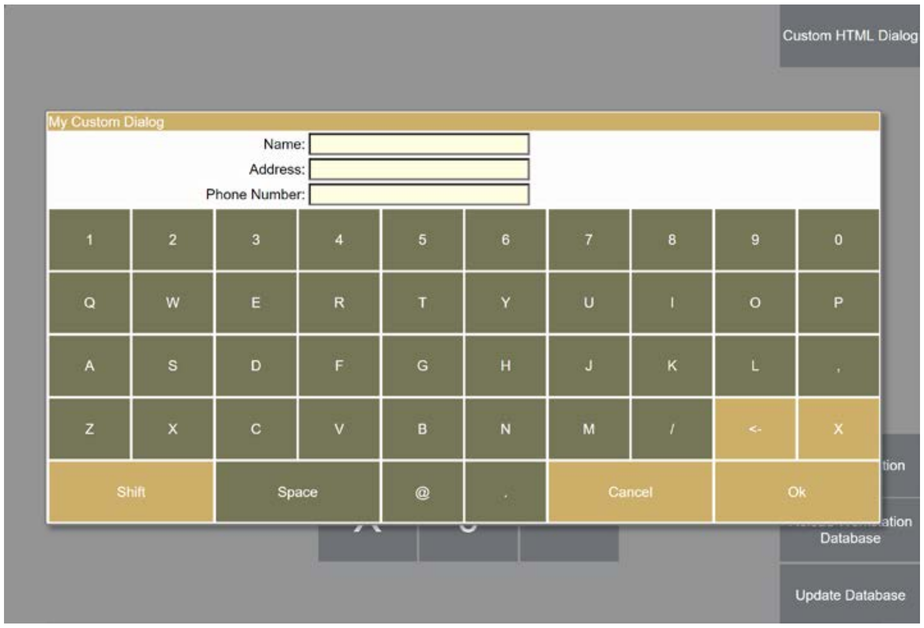
```
function processResponse(parmsOut, callback)
{
    if (parmsOut.isSuccess())
    {
        if (callback)
            callback(parmsOut.Response);
    }
    else if (parmsOut.isError())
        alert('Error: ' + parmsOut.Response);
    else if (parmsOut.isTimeout())
        alert('Timeout');
}
```

9

System Keyboard

The API lets you insert a keyboard directly into the HTML5 document. If a form requires a text input, the dialog keyboard eliminates the need for a physical keyboard.

The following is an example dialog for entering custom information using the API keyboard:



Dialog elements:

- Three input fields, each with its own input requirements
- API-supplied keyboard with Ok and Cancel buttons

Displaying the Keyboard

Displaying a keyboard is very straightforward. The minimum requirements to display the keyboard are:

- Creating a DOM element to ‘contain’ the keyboard. For example, a simple <div>.
- Calling SimphonyPOSAPI.setKeyboard() to initialize the keyboard.
- Indicate if the OK/Cancel buttons should be displayed with the keyboard

HTML

```
<div id='dlgKeyboard'></div>
```

JavaScript

```javascript
function onload()
{
    let kbParms = new SimphonyPOSAPI_KeyboardParameters();
    kbParms.container = document.getElementById('dlgKeyboard');
    kbParms.showFunctionButtons = false;
    SimphonyPOSAPI.setKeyboard(kbParms);
}
```

The user application is responsible for sizing the keyboard appropriately. This sample code will size the keyboard to take 90 percent of the width and 50 percent of the height of the display.

JavaScript

```javascript
let kb = document.getElementById('dlgKeyboard');
kb.style.height = '' + (window.innerHeight * .50) + 'px';
kb.style.width = '' + (window.innerWidth * .90) + 'px';
```

Reacting to Keyboard Events

Reacting to keyboard events is a complicated process. There are several uses for the extensibility developer:

- The ext app JavaScript handles all key presses.
- The API handles all the key presses.
- The API and ext app work together.

Ext App Handles All Key Presses

In this mode, the ext app initializes the API keyboard with a callback. When a button is pressed, the callback performs any processing.

JavaScript

```javascript
function onload()
{
    let kbParms = SimphonyPOSAPI.allocateKeyboardParameters();
    kbParms.container = document.getElementById('dlgKeyboard');
    kbParms.callback = kbCallback;
    SimphonyPOSAPI.setKeyboard(kbParms);
}

function kbCallback(args)
{
    switch (args.keyboardEvent)
    {
        case SimphonyPOSAPI_KeyboardEventType.OK:
            break;
        case SimphonyPOSAPI_KeyboardEventType.CANCEL:
            break;
        case SimphonyPOSAPI_KeyboardEventType.TEXT:
            break;
    }
}
```

The ext app is responsible for populating any <input> elements.

API Handles All Key Presses

Place the API in the OSK mode. Any <input> elements present in the HTML5 document will be fed keyboard data based on the currently focused element. OSK mode is useful when there are multiple text boxes on the dialog.

JavaScript

```javascript
let kbParms = SimphonyPOSAPI.allocateKeyboardParameters();
kbParms.enableOSKMode = true;
kbParms.container = document.getElementById('dlgKeyboard');
SimphonyPOSAPI.setKeyboard(kbParms);
```

When OSK mode is selected, the API assumes that every text box is selected for OSK mode. Alternatively, the ext app can supply the text boxes of interest.

JavaScript

```javascript
let kbParms = SimphonyPOSAPI.allocateKeyboardParameters();
kbParms.enableOSKMode = true;
kbParms.domElementsToUse = [document.getElementById('txtbox1'),
document.getElementById('txtbox2')];
kbParms.container = document.getElementById('dlgKeyboard');
SimphonyPOSAPI.setKeyboard(kbParms);
```

API and Ext App Both Handle Key Presses

When the ext app sets the callback property in the keyboard parameters, the API will always call the callback if the keyboard event args. The callback can do the following:

1. Inspect the command and text elements (alpha, ok, cancel)
2. Inspect the currently focused input element (‘target’ property). Note that this can be null.
3. Set the ‘ignore’ property to ignore the keyboard event.
4. Set the ‘text’ property to change the alpha text.

Consider the use case of the sample dialog, which has the following three text boxes: name, address, and phone number. The ext app may have these constraints:

- The name must always be in uppercase.
- The phone number can only contain digits.

In OSK mode, the text boxes are fed the alpha characters. By specifying a callback, the ext app can implement the two constraints.

JavaScript (SampleKb.html)

```
function kbCallback(args)
{
    if (args.keyboardEvent == SimphonyPOSAPI_KeyboardEventType.OK)
    {
        // package form elements and pass back to app
        SimphonyPOSAPI.closeDialog();
    }
    else if (args.keyboardEvent ==
SimphonyPOSAPI_KeyboardEventType.CANCEL)
    {
        SimphonyPOSAPI.closeDialog();
    }
    else if (args.keyboardEvent ==
SimphonyPOSAPI_KeyboardEventType.TEXT && args.target && args.target.id)
    {
        // if phone number ignore non-digits
        if (args.target.id == 'textInputPhoneNumber')
        {
            if (!("0123456789".includes(args.text)))
                args.ignore = true;
        }
        // if name convert to upper case
        else if (args.target.id == 'textInputName')
        {
            args.text = args.text.toUpperCase();
        }
    }
}
```

HTML/JavaScript

```
<html>
<head>
    <link href="Simphony/POSTheme.css" rel="stylesheet" type="text/css" />
    <script src="Simphony/POSDialogAPI.js"></script>

    <style>
        input {
            background-color: lightyellow;
        }
    </style>

    <SCRIPT LANGUAGE='JavaScript'>
        function onload()
        {
            // size the keyboard area
            let kb = document.getElementById('dlgKeyboard');
            kb.style.height = '' + (window.innerHeight * .50) + 'px';
            kb.style.width = '' + (window.innerWidth * .90) + 'px';

            // initialize on-screen keyboard
            let kbParms = new SimphonyPOSAPI_KeyboardParameters();
            kbParms.enableOSKMode = true;
            kbParms.container = document.getElementById('dlgKeyboard');
            kbParms.callback = kbCallback;
            kbParms.showFunctionButtons = true;
            SimphonyPOSAPI.setKeyboard(kbParms);
        }

        function kbCallback(args)
        {
            if (args.keyboardEvent == SimphonyPOSAPI_KeyboardEventType.OK)
            {
                // package form elements and pass back to app
                SimphonyPOSAPI.closeDialog();
            }
            else if (args.keyboardEvent ==
SimphonyPOSAPI_KeyboardEventType.CANCEL)
            {
                SimphonyPOSAPI.closeDialog();
            }
            else if (args.keyboardEvent == SimphonyPOSAPI_KeyboardEventType.TEXT
&& args.target && args.target.id)
            {
                // if phone number ignore non-digits
                if (args.target.id == 'textInputPhoneNumber')
                {
                    if (!("0123456789".includes(args.text)))
                        args.ignore = true;
                }
                // if name convert to upper case
                else if (args.target.id == 'textInputName')
                {
                    args.text = args.text.toUpperCase();
                }
            }
        }
    </SCRIPT>
</head>

<body onload="onload()">
    <div class="SimphonyPOSDialogWindow SimphonyPOSCenterWindow">
        <table>
```

```
            <tr>
                <td class="SimphonyPOSDialogTitleBar" colspan="2">My Custom
Dialog</td>
            </tr>
            <tr>
                <td align="right">Name:</td>
                <td><input type="text" id="textInputName" style="width:250px;"
/></td>
            </tr>
            <tr>
                <td align="right">Address:</td>
                <td><input type="text" id="textInputAddress"
style="width:250px;" /></td>
            </tr>
            <tr>
                <td align="right">Phone Number:</td>
                <td><input type="text" id="textInputPhoneNumber"
style="width:250px;" /></td>
            </tr>
            <tr>
                <td colspan="2">
                    <div id='dlgKeyboard'></div>
                </td>
            </tr>
        </table>
    </div>
</body>
</html>
```

10 JSON Serialization

There are two methods in the extensibility namespace for serializing and deserializing JSON:

C#

```
namespace Micros.PosCore.Extensibility.Networking
{
    static public class JsonSerialization
    {
        static public string SerializeObjectToJSON( object thing, bool
formatted );
        static public object DeserializeObjectFromJSON( string json, Type
type );
    }
}
```

Although these methods are available, for SIM it is unclear of their utility, as SIM cannot define its own classes.

11 What It Does Not Do

The custom dialog feature does not provide the following functionalities for the HTML5 dialog application:

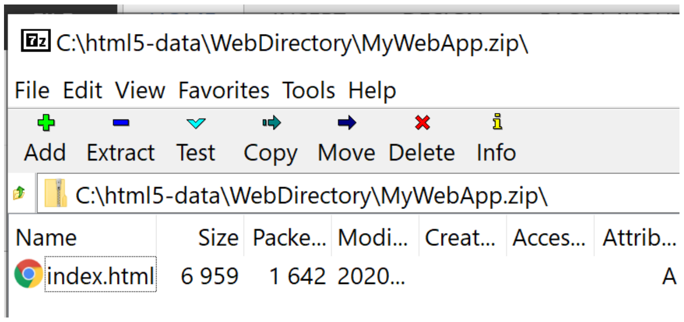
- Access to OpsContext and DataStore: The code-behind supplies this data to the dialog.
- Third-party frameworks: Libraries, such as jQuery, are not provided. The HTML5 app provides the framework in the web directory.
- XmlHttpRequest communications to extensibility application: The API must be used for executing code-behind.
- EGateway logging: The extensibility application should provide its own logging based on its needs.
- Asynchronous message from code-behind to the dialog: The dialog should poll for data periodically if needed.

12 Web Directory Dialog Format

Some HTML5 dialogs are too complex for a simple HTML5 document. There may be many images, different modules, and third-party frameworks. While it is possible to use content/ext app content to solve some of these problems, it can be difficult to configure and maintain.

The alternate method is to pass a ZIP file to OpsContext. ShowExtensibilityHtmlDialog (). The ZIP file can be stored in content, ext app content, or embedded in a .NET assembly. RaiseBrowserDialog() is passed a byte array with the zip file contents.

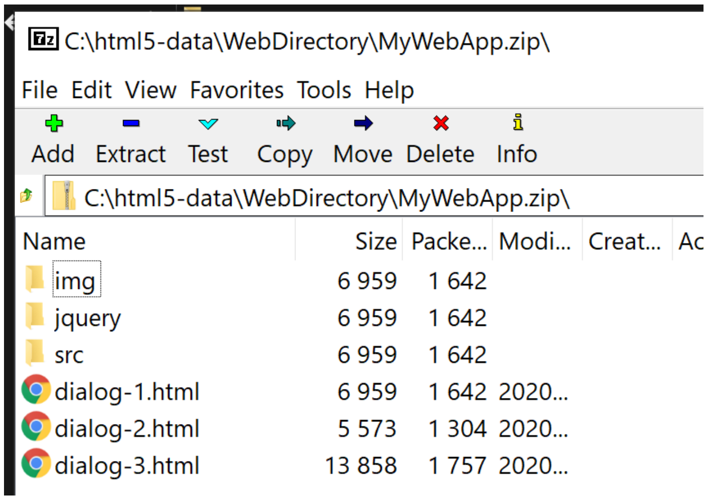
The simplest web directory is a ZIP file that contains a single index.html file:



C:\html5-data\WebDirectory\MyWebApp.zip\
File Edit View Favorites Tools Help
Add Extract Test Copy Move Delete Info
C:\html5-data\WebDirectory\MyWebApp.zip\

Name	Size	Packe...	Modi...	Creat...	Acces...	Attrib...
index.html	6 959	1 642	2020...			A

At run-time, the client searches for index.html as the start page. Override the start page resource in the dialog parameters. This allows multiple dialogs to coexist inside a web directory file. A more complex web directory can contain much more.



C:\html5-data\WebDirectory\MyWebApp.zip\
File Edit View Favorites Tools Help
Add Extract Test Copy Move Delete Info
C:\html5-data\WebDirectory\MyWebApp.zip\

Name	Size	Packe...	Modi...	Creat...	Ac
img	6 959	1 642			
jquery	6 959	1 642			
src	6 959	1 642			
dialog-1.html	6 959	1 642	2020...		
dialog-2.html	5 573	1 304	2020...		
dialog-3.html	13 858	1 757	2020...		

13 Debugging and Troubleshooting

Debugging a standard web application is a simple process, so long as you have direct access to the browser and its built-in debugger. Debugging on Simphony is more complex, given the closed nature of the container as well as operating system differences.

The custom dialog feature provides several facilities for aiding in writing and troubleshooting applications. Always disable these facilities on production code.

Close Button

When creating a custom HTML5 dialog, the JavaScript cannot parse, throw exceptions, and other issues. When this occurs, your dialog is considered ‘dead’ and there is no way to close it, as your close button may not even be displayed.

When instantiating a dialog, the C#/SIM can set a flag that automatically adds a ‘close’ button to the dialog.

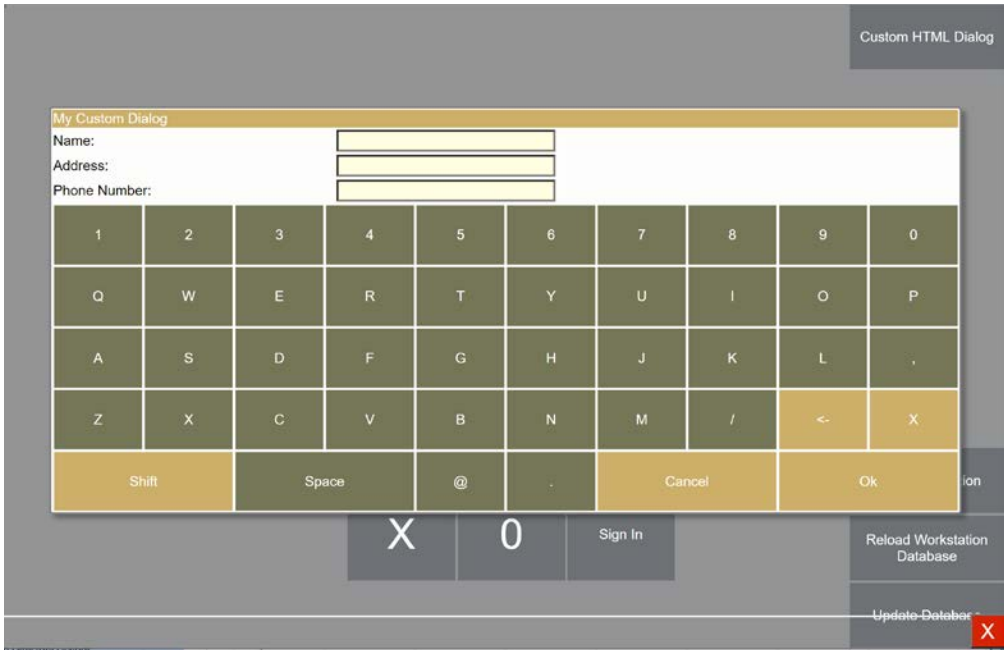
C#

```
var parms = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHtmlDial
ogParameters();
parms.HTML =
DataStore.ReadExtensionApplicationContentTextByNameKey( OpsContext.RvcI
D, ApplicationName, "html" );
parms.ShowCloseButton = false;
string result = OpsContext.ShowExtensibilityHtmlDialog( parms );
```

SIM

```
event inq:1
    var parms:object = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHt
mlDialogParameters()
    parms.HTML =
@DataStore.ReadExtensionApplicationContentTextByNameKey(
@OpsContext.RvcID, @ApplicationName, "html" )
    parms.ShowCloseButton = 1
    WaitForHtmlDialog parms, "SampleDialogClosed"
endevent
```

When the dialog displays, the close button appears at the bottom right.



Pressing the close button is a hard close, meaning it dismisses the dialog without returning any data to the extensibility application. The size of the dialog is slightly smaller to accommodate the close button.

Debug Mode

None of the platforms allow for direct access to the browser or a built-in HTML5 debugger. The container on all platforms has no built-in debugger module.

When debug mode is enabled for a dialog, the extensibility application can open up the browser of its choice with a supplied URL. The POS will not open the browser; instead, it will only provide a URL that displays in a browser. In practical terms, debug mode only applies to Microsoft Windows. Theoretically, this can work on Linux and Android, however, it is a difficult process.

The following is an example of the supplied URL:

```
http://localhost:8080/CustomContent/ExtensibilityDialog/dbgyilfiwzidxwto
zfno/1/$CustomHTMLDialogSample-inplace-html-1$
```

The requirements for the extensibility application and environment are:

- POS web.config.txt debug mode is enabled for the extensibility application.
- When instantiating the dialog set the DebugMode property to true in the dialog parameters.
- Subscribe to the OpsCustomDialogDebugModeReadyEvent.
- When the OpsCustomDialogDebugModeReadyEvent fires, the ext app should open a browser with the supplied URL.

Both the web.config.txt debug mode setting and the program set DebugMode have to be true to enable the mode.

Once the browser is opened, you can use the built-in browser debugger.

web.config.txt

```
<add key="Debug.CustomHTMLDialogSample.CustomDialogDebugMode.Enabled"
value="1" />
```

C#

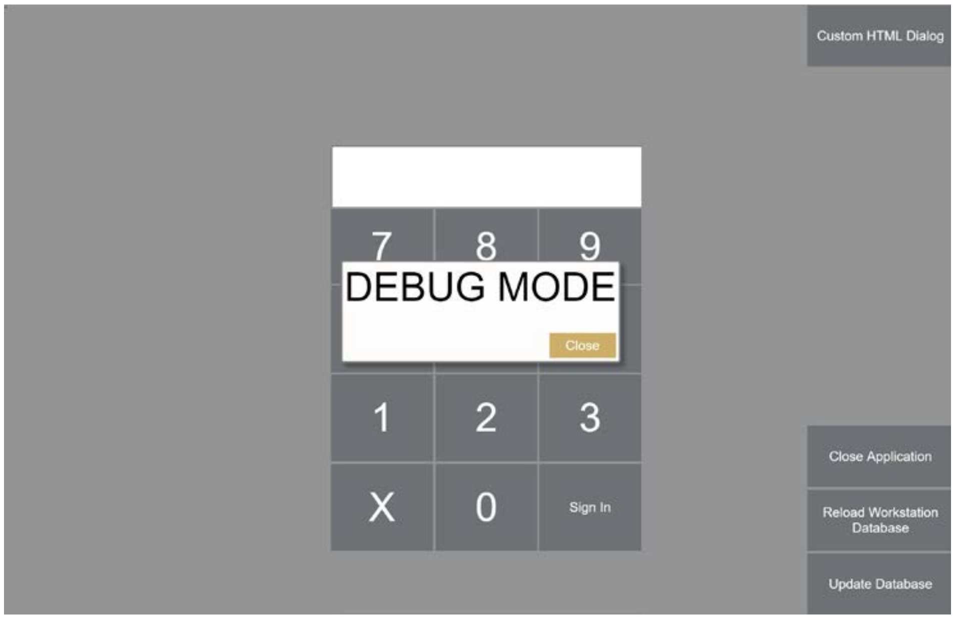
```
public CustomHTMLDialogApp( IExecutionContext context )
    : base( context )
{
    OpsCustomDialogDebugModeReadyEvent +=
CustomHTMLDialogApp_OpsCustomDialogDebugModeReadyEvent;
}

[ExtensibilityMethod]
public void RaiseInPlaceDialog()
{
    var parms = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHtmlDial
ogParameters();
    parms.HTML =
DataStore.ReadExtensionApplicationContentTextByNameKey( OpsContext.RvcI
D, ApplicationName, "html" );
    parms.DebugMode = true;
    string result = OpsContext.ShowExtensibilityHtmlDialog( parms );
}

private EventProcessingInstruction
CustomHTMLDialogApp_OpsCustomDialogDebugModeReadyEvent( object
sender, OpsCustomDialogDebugModeReadyArgs args )
{
    var browserProcess = new System.Diagnostics.Process();
    browserProcess.StartInfo.FileName = @"C:\Program
Files\Chromium\chrome.exe";
    browserProcess.StartInfo.Arguments = args.URL;
    browserProcess.Start();
    return EventProcessingInstruction.Continue;
}
```

SIM

```
NetImport from "System.Diagnostics.Process"
RetainGlobalVar
SetLocalScoping 1

event init
    SubscribeToEvent "OpsCustomDialogDebugModeReadyEvent",
"CustomDialogDebugModeReadyEvent"
endevent

event inq:1
    var parms:object = new
Micros.PosCore.Extensibility.UserInterface.ExtensibilityInPlaceHt
mlDialogParameters()
```

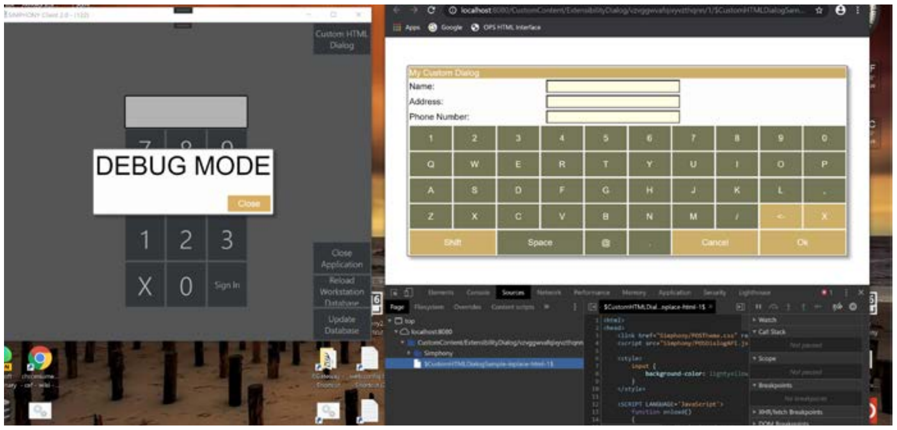
```
    parms.HTML =
@DataStore.ReadExtensionApplicationContentTextByNameKey(
@OpsContext.RvcID, @ApplicationName, "html" )
    parms.DebugMode = 1
    WaitForHtmlDialog parms, "SampleDialogClosed"
endevent


event CustomDialogDebugModeReadyEvent( var sender:object, var
args:object)
    var browserProcess:object = new System.Diagnostics.Process()
    browserProcess.StartInfo.FileName = "C:\Program
Files\Chromium\chrome.exe"
    browserProcess.StartInfo.Arguments = args.URL
    browserProcess.Start()
endevent
```

When the dialog is raised, the POS instead shows a debug window on the POS.



The extensibility application debug event creates a browser instance, which can be used to troubleshoot your dialog.

The following image shows the POS application (on the left side) in debug mode, and a browser (on the right side) with debugging tools opened.



When the API closeDialog() method is called, the POS dismisses the debug window and you must close the browser window.

Automatically Close Browser Window

To make development easier, the SimphonyPOSAPI.DebugMode property determines if you are running in debug mode. If in debug mode, the browser dialog can call ‘window.close()’ to close the browser instance or the current tab. This code snippet remains in production, as debug mode should never be enabled in production code.

Javascript

```
function closeClick()
{
    // if we are in debug mode then close the current tab/window
    // there are timing issues with any javascript executing after the
closeDialog()
    // method call. Close the window soon after closing the dialog.
    if (SimphonyPOSAPI.debugMode)
        setTimeout(function () { window.close(); }, 250);

    // send text back to ext app
    let result = getElem('textResult');
    SimphonyPOSAPI.closeDialog(result);
}
```

Checking for API Existence

We recommend checking for the existence of the API before calling any methods. One way to accomplish this is:

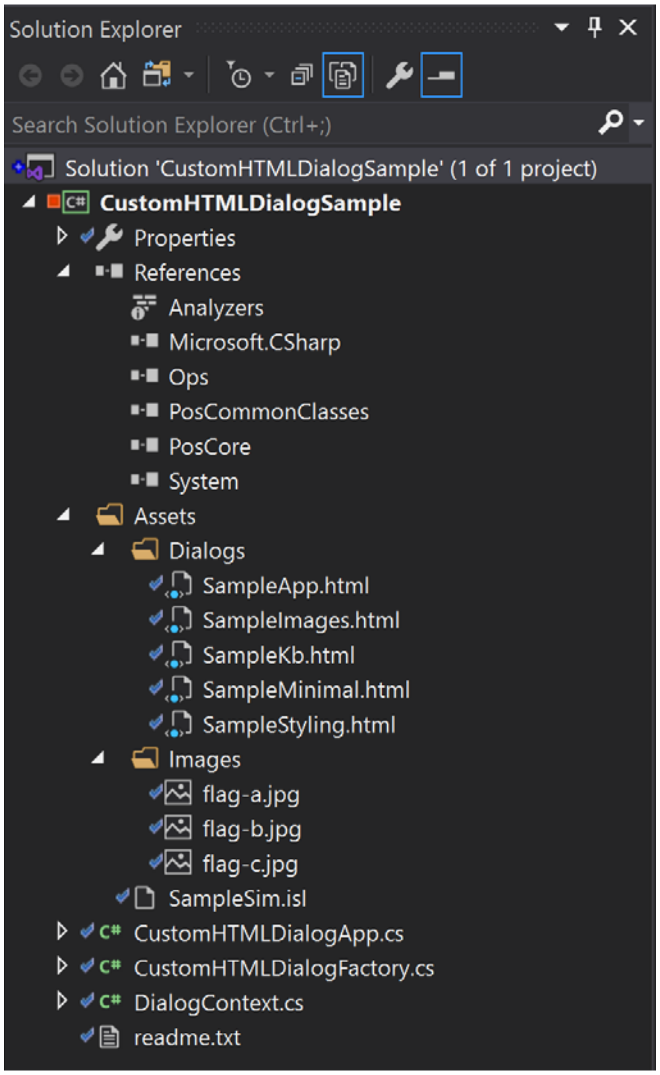
Javascript

```
function onload()
{
    // older or incompatible browsers can sometimes cause
Simphony/POSDialogAPI.js to fail to load.
    // If this happens the api object SimphonyPOSAPI may not exist
causing very subtle failures.
    // Show an error if the SimphonyPOSAPI is undefined.
    if (!(("SimphonyPOSAPI" in window) && SimphonyPOSAPI))
    {
        alert('SimphonyPOSAPI is undefined, browser is possibly not up
to date');
        return;
    }
}
```

14 Sample Custom HTML5 Dialog Application

The following sample application contains documentation on how to configure the Simphony system. It is a standard Visual Studio .csproj.

All of the code in this document is available in complete form in this sample application. The project contains HTML5, images, sim, and C# files.



```
Solution 'CustomHTMLDialogSample' (1 of 1 project)
  CustomHTMLDialogSample
    Properties
    References
      Analyzers
      Microsoft.CSharp
      Ops
      PosCommonClasses
      PosCore
      System
    Assets
      Dialogs
        SampleApp.html
        SampleImages.html
        SampleKb.html
        SampleMinimal.html
        SampleStyling.html
      Images
        flag-a.jpg
        flag-b.jpg
        flag-c.jpg
      SampleSim.isl
    CustomHTMLDialogApp.cs
    CustomHTMLDialogFactory.cs
    DialogContext.cs
    readme.txt
```

15 C# and SIM Differences

Overall, the same features are provided to SIM and C#. The notable differences are:

- SIM should not call the OpsContext. ShowExtensibilityHtmlDialog() method. The callbacks back into SIM will not work. Always use WaitForHtmlDialog instead.
- SIM does not allow for context methods, only extensibility event and extensibility method are allowed. In general, the context facility requires a more robust programming environment than SIM can offer. However, all three methods are functionally equivalent.

16 Feature Reference

POSDialogAPI.js

- SimphonyPOSAPI
  - argument
    - property, read-only string
    - The value of the Argument property set in the parameters in the call to OpsContext.ShowExtensibilityHtmlDialog()
  - debugMode
    - property, read-only boolean
    - Set to true if the dialog was launched in debug mode
  - closeDialog(data)
    - method, called to close the current executing dialog
    - data parameter must be a string, null, or undefined
  - setKeyboard(parms)
    - method, called to initialize on-screen keyboard
    - parms must be of type SimphonyPOSAPI_KeyboardParameters()
    - this method cannot be called twice in one dialog session. That is, it's not possible to have 2 or more keyboards in a single dialog.
  - runExtensibilityEvent(parms, callback)
    - method, called to execute code in extensibility application using extensibility event mechanism
    - parms must be of type SimphonyPOSAPI_ExtensibilityEventParameters
    - this method is asynchronous and returns control immediately
    - callback is the method to execute when a response is received.
    - The callback method is called with a SimphonyPOSAPI_ApplicationResponse object.
    - Callback can be null
  - runExtensibilityMethod(parms, callback)
    - method, called to execute code in extensibility application using extensibility method mechanism
    - parms must be of type SimphonyPOSAPI_ExtensibilityMethodParameters
    - see runExtensibilityEvent() for description of callback
  - runContextMethod(parms, callback)
    - method, called to execute code in extensibility application using context method mechanism
    - parms must be of type SimphonyPOSAPI_ContextMethodParameters
    - see runExtensibilityEvent() for description of callback
- SimphonyPOSAPI_ApplicationResponseType
  - Enum, values indicate status of an extensibility call: SUCCESS, ERROR, TIMEOUT

- SimphonyPOSAPI_ApplicationResponse
  - Object passed to callback method for runExtensibilityEvent, runExtensibilityMethod, runContextMethod
  - status
    - read-only property, SimphonyPOSAPI_ApplicationResponseType
  - response
    - read-only property, string
    - if a successful call then this property is the result set by the extensibility application back to the dialog
    - if an error occurred then this property will contain the error message
  - isSuccess(), isError(), isTimeout()
    - shorthand methods to test status
- SimphonyPOSAPI_ExtensibilityEventParameters
  - Object passed to the runExtensibilityEvent() method
  - command
    - string property set by caller
    - this string will be passed to the extensibility application OpsCustomDialogEventArgs.Message property
  - argument
    - string property optionally set by caller
    - this string will be passed to the extensibility application OpsCustomDialogEventArgs.Argument property
    - This property must be a string and not an object. Use JSON.stringify() to pass complex data.
- SimphonyPOSAPI_ExtensibilityMethodParameters
  - Object passed to the runExtensibilityMethod () method
  - dllName
    - the name of the assembly the extensibility method resides in. This parameter is the same that is used when configured a button in Page Designer.
    - This is null for SIM applications
  - method
    - String property set by caller
    - This string represents the name of the method in the extensibility application
  - argument
    - string property optionally set by caller
    - this string will be passed to the extensibility method as the first argument
    - This property must be a string and not an object. Use JSON.stringify() to pass complex data.
- SimphonyPOSAPI_ContextMethodParameters
  - Object passed to the runContextMethod () method
  - method
    - string property set by caller
    - Name of method in the dialog context object
  - argument
    - this string will be passed to the method as the first argument
- SimphonyPOSAPI_KeyboardParameters
  - Object passed to the setKeyboard method()

  - container
    - DOM element whose innerHTML will be set to the on-screen keyboard
    - Cannot be null
  - enableOSKMode
    - boolean property
    - if true then the API will ‘feed’ characters into the currently focused textbox.
    - If false then the dialog must process all key presses.
    - The types of input elements that are searched for are 'email', 'number', 'password', 'search', 'tel', 'text', 'url', 'date'
  - callback
    - property that can be optionally set with a callback when a key is pressed.
    - The callback will be passed a SimphonyPOSAPI_KeyboardEventArgs object
  - domElementsToUse
    - optional array property, each element of the array is an <input> dom element
    - This configuration element allows finer control for choosing which textboxes are fed key presses.
  - showFunctionButtons
    - Boolean property set by user
    - If true then the ok/cancel buttons will be displayed. If false the buttons will be hidden.
- SimphonyPOSAPI_KeyboardEventType
  - Enum of keyboard event types: OK, CANCEL, TEXT, BACKSPACE, CLEAR, OTHER
- SimphonyPOSAPI_KeyboardEventArgs
  - The object passed to the callback method defined in SimphonyPOSAPI_KeyboardParameters
  - keyboardEvent
    - read-only property of type SimphonyPOSAPI_KeyboardEventType
    - If TEXT then the ‘text’ property will contain the key press.
  - text
    - String property containing the key press value if the keyboardEvent is TEXT. The value is undefined if the keyboardEvent is not TEXT.
    - The dialog can change this value, for example, set it to its upper case value.
  - target
    - Readonly property, DOM element.
    - if enableOSKMode is enabled the API keeps track of the currently focused input element. When a keyboard event target is set to the currently focused input element.

POSTheme.css

- SimphonyPOSDialogWindow
  - Style of dialog window.
- SimphonyPOSCenterWindow
  - Will center selected item in the window, both vertically and horizontally.
- SimphonyPOSDialogTitleBar
  - Will format the selected item with the theme-specific foreground and background colors.
- SimphonyPOSDialogControlButton
  - Will format the selected item as a control button.
  - Control buttons should be buttons that dismiss the dialog: ok, cancel, close.
- SimphonyPOSDialogFunctionButton
  - Will format the selected item as a function button
  - Function buttons perform actions within the dialog.

OpsContext

- ShowExtensibilityHtmlDialog(ExtensibilityHtmlDialogParameters parameters)
  - Parameters must be one of these types:
    - ExtensibilityInPlaceHtmlDialogParameters
    - ExtensibilityWebDirectoryDialogParameters
  - Returns string result from dialog
  - Control is only passed back when the dialog exist.
  - Can be nested.

Support Classes

- ExtensibilityHtmlDialogParameters
  - Namespace: Micros.PosCore.Extensibility.UserInterface
  - Base class for parameters for showing a dialog. Contains common properties.
  - Sender
    - String property
    - The Sender value will be sent to the extensibility application in the OpsCustomDialogEventArgs.Sender field
    - If an extensibility application can display two or more dialogs, this property allows the extensibility application to determine which dialog was raised. For example, the value could be “MyDialog1”, “MyDialog2”, …
  - Argument
    - String property
    - The Argument is available to the dialog in POSDialogAPI.argument property
  - Context
    - Object property

        - When POSDialogAPI.runContextMethod() is called, the specified method is invoked on this object.
    - ShowCloseButton
        - Boolean property
        - When set to true the POS will display a close button alongside the dialog.
        - This property is useful during development when the dialog code can be unstable.
    - DebugMode
        - Boolean property
        - When set to true the dialog will be raised in debug mode.
        - The corresponding web.config.txt setting must also be present to enable debug mode. The format of the web.config.txt entry is the following. Note that the ExtAppName must correspond to the extensibility application name .
            - <add key="Debug.*ExtAppName*.CustomDialogDebugMode.Enabled" value="1" />
- ExtensibilityInPlaceHtmlDialogParameters
    - Namespace: Micros.PosCore.Extensibility.UserInterface
    - Parameters object for raising in-place html dialog
    - HTML
        - String property
        - Set to the HTML to be displayed in the dialog.
- ExtensibilityWebDirectoryDialogParameters
    - Namespace: Micros.PosCore.Extensibility.UserInterface
    - Parameters object for raising web directory dialogs
    - WebDirectoryZip
        - Byte array property containing zipped image of web directory.
    - StartResource
        - String property indicating the start resource to display. If none is specified the default is “index.html”.
        - If an extensibility application contains multiple dialogs sharing the same resources, this property allows multiple dialogs to be contained in one web directory.
    - ZipPassword
        - String property set to the password if the zip file is password protected
- OpsCustomDialogEventArgs
    - Namespace: Micros.PosCore.Extensibility.Ops
    - Object passed to extensibility when the POSDialogAPI.runExtensibilityEvent() is called
    - Sender
        - String property set to the value of the parameters Sender property when the dialog is raised.
    - Command
        - String property set to the SimphonyPOSAPI_ExtensibilityEventParameters.command property
    - Argument

        - String property set to the SimphonyPOSAPI_ExtensibilityEventParameters.argument property
    - Response
        - String property set by extensibility application and available to dialog in the SimphonyPOSAPI_ApplicationResponse.response property.
- OpsResourceNotFoundEventArgs
    - Namespace: Micros.PosCore.Extensibility.Ops
    - Object passed to the extensibility application in the OpsResourceNotFoundEvent
    - URL
        - Read-only string property containing the URL of the invalid resource
    - Sender
        - Read-only string property referring to the Sender property when raising the dialog
    - HttpMethod
        - Read-only string indicating the method of the call (POST, GET, …)
    - ResourceSupplied
        - Boolean property set by extensibility application if it chooses to supply the resource
    - ResourceExtension
        - String property set by the extensibility application to indicate the resource extension (jpg, png, …)
    - BinaryResource
        - Byte array property set by the extensibility application containing the binary resource if the resource is binary.
    - TextResource
        - String property set by the extensibility application containing the text resource if the resource is text.
- OpsCustomDialogDebugModeReadyArgs
    - Namespace: Micros.PosCore.Extensibility.Ops
    - Object passed to the extensibility application in the OpsCustomDialogDebugModeReady event
    - URL
        - Read-only string property containing the fully qualified URL to be rendered in a browser for that dialog instance.
        - The URL is randomized, it cannot be used across sessions.

SIM

- WaitForHtmlDialog
  - SIM command to show the dialog
    - Parameter 1: Either a ExtensibilityInPlaceHtmlDialogParameters or ExtensibilityWebDirectoryDialogParameters object
    - Parameter 2: String value indicating the name of the dialog event to raise when the dialog closes. This parameter must contain only alphanumeric characters or an underscore. It cannot contain spaces or punctuation.
  - This command returns control immediately to the SIM application.
- event dialog : *eventname*
  - event raised when a SIM dialog closes.
  - The *eventname* is the parameter in the WaitForHtmlDialog command
  - @HtmlDialogResult is valid in this event.
- event extensibility_method : *methodname*
  - This event is raised when the POSDialogAPI.runExtensibilityMethod() method is called
  - The *methodname* is the method name specified in the dialog API call.
  - @ExtensibilityMethodArgs is valid in this event.
- @HtmlDialogResult
  - SIM string system variable containing the POSDialogAPI.closeDialog() data parameter.
  - Valid only in the “event dialog” event.
- @ExtensibilityMethodArgs
  - SIM object system variable containing extensibility method data
  - Argument
    - Read-only string property containing the argument from the dialog.
  - Response
    - String property containing the result passed back to the dialog