General Application Customization

- Customizing the Default Template

- Creating a New Template

- Activating an Application Map from a Menu

- Using ApplinX Repository Folders' Structure to Organize Web Pages

- Controlling the Connection Properties from the Code

- Handling Flickering of Screens

- Waiting for Screens, using Wait Conditions

- Customizing the Host Keys

- Activating the Screen Locker

- Natural UNIX: Integrating a Login Page in the Web Application

- Implementing & Controlling JavaScript Events using the gx_event Object

- Retrieving Data from Fields Outside the Modal Window Currently Displayed

- ApplinX Server Load Balancing

- Retrieving a Unique Device Name per User Name / IP Address from a Database and Applying it to the Current User Session

- Retrieving the Host Printer Device Name from a Database and Setting the ApplinX Printlet to Work with that Device

- Customize ApplinX Framework Session Error Handling

- Customize the Web Application's Error Page

Customizing the Default Template

Every ApplinX Web application contains a template file which is used by all the pages. It is possible to override parts of the default template by using placeholders. These placeholders are defined in the template and can be overridden with customized content on generated pages and generate screen groups. Using placeholders in the template provides flexibility on the overall look and feel of individual pages. For example, if the template supplies the users with links on the left side of the page and on a specific page, it is required that these links be removed, the template's default content can be overridden for that page section, with customized content for that particular page. Refer to the Composite demo, template.jsp/template.master and login.jsp/aspx.

▶ **To create a custom template for a screen/screen group (JSP):**

1. Generate a Web page for the screen/screen group (refer to Generating a Framework Page for a Screen/Screen Group).

2. For each template part that you would like to design for this screen/screen group, ensure that the area that is to be overridden is enclosed by a `gx:placeholder` tag. For example, if you would like to change the page header, implement the following in the *template.jsp* file.

```
<gx:placeholder id="<TemplatePlaceHolderID>">
     .
     .
         <your page header JSP code>
     .
     .
</gx:placeholder>
```

3. Server side functions referred by the template.jsp file will have to be placed in the GXBasicContext.java file.

4. In the jsp file that was generated for the screen/screen group (*generatedpage.jsp*), override a template placeholder by adding your content and enclosing it within a gx:content tag, setting the attribute `ContentPlaceHolderID` with the ID of the template place holder previously defined.

```
<gx:content ContentPlaceHolderID="<TemplatePlaceHolderID>">
     .
     .
                   <YOUR CONTENT HERE>
     .
     .
</gx:content>
```

▶ **To create a custom template for a screen/screen group (.NET):**

1. Generate a Web page for the screen/screen group (refer to Generating a Framework Page for a Screen/Screen Group).

2. For each template part that you would like to design for this screen/screen group, ensure that the area that is to be overridden is enclosed by a `asp:contentplaceholder` tag. For example, if you would like to change the page header, implement the following in the *template.master* file.

```
<asp:contentplaceholder id="<TemplatePlaceHolderID>" runat="server" >
     .
     .
         Links section
     .
     .
</asp:contentplaceholder>
```

3. Server side functions referred by the template.master file will have to be placed in the *template.master.cs/vb* file.

4. In the aspx file that was generated for the screen/screen group (*generatedpage.aspx*), override a template placeholder by adding your content and enclosing it within a asp:Content tag, setting the attribute `ContentPlaceHolderID` with the ID of the template place holder previously defined.

```
<asp:Content ID="<tagID>" ContentPlaceHolderID="<TemplatePlaceHolderID>" runat="server">
     .
     .
                    <YOUR CONTENT HERE>
     .
     .
</asp:Content>
```

Creating a New Template

During the process of creating a new Web application, you are requested to select a template. By default, these templates are ready to use Web site designs. You can create a template and add it to the list of available templates.

Design Tips

- Before making changes in the css files, it is highly recommended to backup these files.

- Changing colors in the css files can effect the entire look and feel of the application, therefore it is recommended to test the changes on a few different screens.

- Rather than changing existing classes to meet the template requirements, try adding new classes of your own.

- Instant pages rely on absolute positioning defined in the styles_Instant.css. Do not change position style attributes in the css files.

- Pages generated with absolute positioning: changing position attributes in the style_generated.css file may cause them not to display properly (this can be overcome in design time).

- Remember that changes made to one css file will usually need to be implemented to all three css files, if both generated pages and instant pages are to have the same look and feel.

▶ **To create a new template:**

1. Create a new Web application using one of the built-in templates.

2. Design the template.jsp/template.master file as desired.

   Ensure that the template includes:

   JSP: <gx:placeholder id="PagePlaceHolder"/> tag.

   .NET: <asp:ContentPlaceHolder ID="GXPagePlaceHolder" runat="server"/> tag.

   This tag's position on the screen can vary and can be placed wherever you wish. Removing this tag means that screen content for generated and instant pages will not be displayed.

3. Package the template and place it in the installation directory, so that it will be displayed when using the Create new Web Application wizard:

   1. Create a new directory in the Templates directory in the ApplinX installation folder. The name of the directory will be the name of the template displayed in the New Web Application wizard.

2. Within this directory create a common directory and a java/dotnet (according to the template you created) directory. Within the common directory, create a css and images directory:
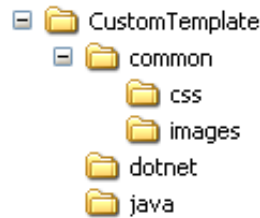
3. In the css folder place all the css files that you modified/created.

4. In the images folder place all the image files that you modified/created in the template file or in the css files.

5. In the common folder place the image that the wizard will display as a preview of this template. The image size should be 630x400pixels and the name of the image file must be thumbnail.jpg.

6. Place the template file in the relevant java/dotnet folder.

Activating an Application Map from a Menu

Navigation menus, are very common in Web applications. They provide a direct, quick navigation to any page in the application. In ApplinX Web applications, navigation menus can be used to directly access any available screen in the host application, thus eliminating the need of using the original menu screens in the host while fully preserving their functionality. This is achieved using the application map. Each menu option will be an accessible host screen, and clicking it will invoke the ApplinX application's map with the screen's name as its parameter.

Refer to the Composite demo, template.jsp/template.master.cs in which the proposals/customers links set a hidden field with the desired screen name and invoke a server side function (`beforePaneMenuBtn_Click`) that performs the navigation to the desired screen (refer to /GXBasicContext.jsp/template.master.cs).

Using ApplinX Repository Folders' Structure to Organize Web Pages

ApplinX Web application's Web pages can be hierarchically organized according to the folder structure of the screens they represent. For example, if the screen *MainMenu* resides in the folder *general\menu_screens* in the ApplinX repository, it can reside under the directory *\root folder\general\menu_screens* in the Web application.

▶ **To use ApplinX repository folder structure to organize Web pages:**

1. Refer to Configuring your Framework and access the Framework Configuration Editor.

2. Ensure that the **Use Folders** check box in the **General** node is selected. The field **Virtual directory** is displayed.

3. Enter the name of the virtual directory of the Web application. For example: CompositeDemo.

4. To generate a Web page into a specific folder:

    1. Create a directory structure under the root folder of the Web application that corresponds to the folder structure of the ApplinX repository.

    2. You can either select the directory within the generation wizard (JSP only), or you can move the generated pages to the relevant directory (JSP and .Net).

Controlling the Connection Properties from the Code

ApplinX allows passing various connection parameters from the code in order to control the configuration of the application.

For example: Controlling from the code if the application will work against a trace file.

Refer to the Composite demo (with/without a trace file).

.NET:

```
gx_appConfig.SessionConfig.addVariable(GXBaseObjectConstants.GX_VAR_REPLAY_FILE,null);
          // to cancel the replay file configured
```

JSP:

```
 getGXAppConfig().getSessionConfig().addVariable(GXBaseObjectConstants.GX_VAR_REPLAY_FILE,null);
          // to cancel the replay file configured
```

For a list of possible connection variables, see GXBaseObjectConstant class.

For code sample, see Composite Demo Application: "Login" path procedure - ApplinX repository Login.aspx / Login.aspx.cs - .NET Login.jsp / WEB-INF/classes/contexts/Login.java – JSP

Handling Flickering of Screens

It is necessary to use the Flickering of Host Sessions feature when one of the following happens:

- In the browser, a blank screen (empty screen) is displayed when navigating between two host screens.

- In the browser you are required to submit the [ENTER] key (or any other key) twice in order to navigate to the next host screen.

The initial need for Flicker arises when specific host screens are received 'split' between several buffers of data. Thus ApplinX Server needs to be informed to wait an additional amount of time for the complete screen to arrive. This additional amount of time is defined (in milliseconds) in the Flicker parameter in the Application Configuration dialog box.

The flicker setting applies to the entire ApplinX application, meaning that if the flicker is set to 500ms, after each host transaction the flicker time will be added to the communication time. In other words, the entire application will be 'slowed down' by the flicker time. Therefore this value should only be set for

the entire application according to the guidelines detailed in Handling Flickering of Screens. Also refer to Waiting for Screens using Wait Conditions and to the **Perform background check for host screen changes** parameter in the Configuration Editor for further details.

Waiting for Screens, using Wait Conditions

Wait conditions are used to handle specific 'problematic' screens that require waiting an additional amount of time for the complete screen to arrive. Refer to Handling Flickering of Screens for further details regarding using the wait conditions.

To add a wait condition that waits for a specific screen in the application:

JSP

In *GXBasicContext.java*, in the method `gx_postSendKey`, add the following code:

```
GXGetScreenRequest sr = new GXGetScreenRequest();
//An example of adding a wait condition in a page that waits for specific screens according to the key sent.
if(gx_context.getGXForm().get_HostKeys().equalsIgnoreCase("[PF3]")
&& gx_context.getGXSession().getScreen().getName().equalsIgnoreCase("CustomerDetails"))
{
    sr.addWaitCondition(new GXWaitForScreen("MainMenu",50000,0));
    gx_context.getGXSession().getScreen(sr);
}
```

.NET

In *GXBasicWebForm.cs*, in the method `user_postSendKeys`, add the following code:

```
GXGetScreenRequest sr = new GXGetScreenRequest();
//An example of adding a wait condition in a page that waits for specific screens according to the key sent.
if(gx_form.HostKeys.Equals("[PF3]")
&&  gx_session.getScreen().getName()=="CustomerDetails")
{
    sr.addWaitCondition(new GXWaitForScreen("MainMenu",50000,0));
    gx_session.getScreen(sr);
}
```

To apply a flicker setting to a certain action:

JSP

In *GXBasicContext.java*, in the method `gx_postSendKey`, add the following code:

```
GXGetScreenRequest sr = new GXGetScreenRequest();
if(gx_context.getGXForm().get_HostKeys().equalsIgnoreCase("[enter]")
&& gx_context.getGXSession().getScreen().getName().equalsIgnoreCase("CustomerDetails"))
{
    long lTimeout = 50000;//millisecond
    long lFlicker = 750;//millisecond
    sr.addWaitCondition(new GXWaitHostQuiet(lTimeout,lFlicker));
    gx_context.getGXSession().getScreen(sr);
}
```

.NET

In *GXBasicWebForm.cs*, in the method `user_postSendKeys`, add the following code:

```
GXGetScreenRequest sr = new GXGetScreenRequest();
if(gx_form.HostKeys.Equals("[enter]")
&&  gx_session.getScreen().getName()=="CustomerDetails1")
{
    long lTimeout = 50000;//millisecond
    long lFlicker = 750;//millisecond
    sr.addWaitCondition(new GXWaitHostQuiet(lTimeout,lFlicker));
    gx_session.getScreen(sr);
}
```

Customizing the Host Keys

It is possible to display the host keys for the current host screen (analyzed by ApplinX Server according to the defined host keys patterns) as Web buttons, hyperlinks or in other advanced formats (images, etc.). In addition, it is possible to fully customize the host keys using the GXIHostKeysTagUserExit interface (adding buttons, removing links, changing captions and more). To make complex changes to the control, such as adding a row/column, implement the interface GXIHostKeysTagUserExit as necessary.

Refer to Host Keys for configuring a host key pattern.

▶ **To display and customize the host keys:**

1. By default, after defining a host key pattern, the analyzed keys are displayed as hyperlinks in the Web application.

2. To display the keys as buttons, refer to Configuring your Framework and access the Framework Configuration Editor. Select Buttons in the Host keys field in the Instant node.

3. Customization of the host keys' appearance (colors, fonts etc.) is done in the style sheets (*css\styles_instant.css* and *css\styles_generated.css*), in the gx_hky class.

4. For advanced customization, it is possible to use the Host Keys' control. By using the template option of the control, it is possible to write a custom template for displaying the host keys. The following code is usually placed in one of the template jsp/aspx files and displays a different image for each host key; the image name is determined according to its caption. In addition, the keys are positioned vertically:

   JSP

   ```
   <gx:hostKeys vertical="true" border="0" keyType="template">
           <a href="#" onclick="gx_SubmitKey('$(ACTION)')" title="$(ACTION)-$(CAPTION)">
           <img src="images/keybuttons/btn$(CAPTION).gif" border="0"/>
           </a>
   </gx:hostKeys>
   ```

   .NET

   ```
   <gx:GXHostKeysControl runat="server" vertical="true" border="0" KeyType="Template">
                   <a href="#" onclick="gx_SubmitKey('$(ACTION)')" title="$(ACTION)-$(CAPTION)">
                   <img src="images/keybuttons/btn$(CAPTION).gif" border="0"/>
                   </a>
   </gx:GXHostKeysControl>
   ```

In addition, add the following import statement to the file containing the control: `<%@ Register TagPrefix="gx" Namespace="com.sabratec.dotnet.framework.web.controls" Assembly="GXDotnet" %>`

▶ **To fully customize the host keys control:**

1. **JSP**

   . In the *GXBasicContent.java* file, the `gx_onInit` method, add the following line:

   ```
   getGXAppConfig().setHostKeysTagUserExit(new transforms.UserHostKeysTagTransform());
   ```

   .NET

   : In the *GXBasicWebForm* file, the `OnInit` method, add the following line:

   ```
   gx_appConfig.setHostKeysTagUserExit(new transforms.UserHostKeysTagTransform());
   ```

2. In your Web application, transforms directory, open the *userHostKeysTagTransform* file (when upgrading your application, copy the file from the relevant framework type new application) and implement the `onHostKeysComplete` interface.

For example JSP:

```
public void onHostKeysComplete(GXITableTag hostKeysTableTag, GXIScreen screen) {

        super.onHostKeysComplete(hostKeysTableTag, screen);

        /*
        // add link
        addLink("Host help","gx_SubmitKey('[pf1]')");

        //get all tags
        GXITag[] tags = getHostKeysTags();

        for (int index=0; index<tags.length; index++){
              //get tag
              GXITag tag = tags[index];

              if (tag instanceof GXILinkTag){ // or GXIButtonTag or GXHtmlString (for type="template")
                   GXILinkTag link = (GXILinkTag) tag;

                   // change tag text
                   if (link.getText().indexOf("Prev") >= 0){
                        link.setText("Previous");
                   }

                   //remove tag
                   if (link.getText().indexOf("Pagedn") >= 0){
                        removeTag(link);
                   }
              }
        }
        */

    }
```

For example .NET

```
public override void onHostKeysComplete(GXITableTag hostKeysTableTag, GXIScreen screen)
    {
        /*
        base.onHostKeysComplete(hostKeysTableTag, screen);

        //add link
```

```
        addLink("Host help", "gx_SubmitKey('[pf1]')");

        //get all tags
        GXITag[] tags = getHostKeysTags();

        //loop on controls in hostKeys array
        foreach (GXITag tag in tags)
        {
            if (tag is GXILinkTag)  // or GXIButtonTag or GXHtmlString (for type="template")
            {
                //get link
                GXILinkTag link = (GXILinkTag)tag;

                // change control text
                if (link.getText().IndexOf("Prev") >= 0)
                {
                    link.setText("Previous");
                }

                //remove control
                if (link.getText().IndexOf("Pagedn") >= 0)
                {
                    removeTag(link);
                }
            }
        }
        */
    }
```

For complete description of the Host Keys control properties, see Host Keys Component.

Sample code can be found in the Instant Demo application. *Template.jsp , template.master (.NET) –* contains an example for using the Host Keys control.

Activating the Screen Locker

The ApplinX Framework contains a built-in feature of a screen locker. The purpose of a screen locker is to indicate to the user by means of a message, that the application is processing his request, and blocks him from interfering with the current process by repressing a button/link or keyboard PF/ENTER.

▶ **To activate the screen locker:**

1. Refer to Configuring your Web Application and access the Framework Configuration Editor. In the **General** node, select **Use screen locker**.

2. Use the width/height percentages in the file *template/screenLocker.htm* to control the location of the message.

3. Replace the text "Please wait" with an alternative text/image as required.

Note:
When adding a link in the page that performs a JavaScript action it is highly recommended to use onclick event and not the href attribute. Using the href attribute will cause the screen locker to be activated.

It is also possible to control the screen locker on the Web page level (refer to Handling the Screen Locker on the Page Level).

Natural UNIX: Integrating a Login Page in the Web Application

When the Natural UNIX host requires separate connection for each individual user together with support of password changing/expiring, the ApplinX Framework provides a suitable Login Page to be used in your Web application. The Login page should be displayed before attempting to connect to the host. In this page the user is required to enter the user name and password. Only after the user name and password are authenticated by the host, the connection to the host is made. The look and feel of the Login page can be edited and changed to suit your Web application. The login page logic is implemented within the ApplinX Framework.

Note:
The hostLogin page is provided with your Web application but can also be created using the Base Object. Refer to the Start>Programs>Software AG ApplinX>Documentation>ApplinX Development API for further information.

▶ **To use the Login page:**

- Edit the `index.jsp/index.aspx` file and instead of `gxfirstpage.jsp/gxfirstpage.aspx`, enter `hostLogin.jsp/hostLogin.aspx`, or refer your end users to this page.

Implementing & Controlling JavaScript Events using the gx_event Object

The ApplinX JavaScript engine provides a cross browser event object that enables creating dynamic web pages. The gx_event object is created within the Javascript engine. Whenever an event is triggered it is passed to the user Exit functions (/js/userExits.js) and from there to the page-level function (if one exists).

gx_event enables handling browser events such as OnKeyDown. The following example will cancel the `OnKeyDown` event whenever the [Enter] key is pressed on a certain text area ("myTextArea"), prevent the page from being submitted and manually add a newline character to the text area value:

Assume your JSP/ASPX page has the following input:

```
<textarea row="5" id="myTextArea" ></textarea>
```

Add the following `pageOnKeyDown` function to your generated page

```
function pageOnKeyDown(gx_event){
        ...
        var win = gx_event.window;
        if (gx_event.keyCode==13 && gx_event.element.id=="myTextArea"){
                gx_event.cancel();
                GXBrowserUtil.getElement("myTextArea").value += "\r\n"
        }
...
}
```

Refer to the API:

- GXEvent_Object

- GXEvent.keyCode

- GXEvent.additionalKey

- GXEvent.element

Retrieving Data from Fields Outside the Modal Window Currently Displayed

Relevant data, such as host messages or errors, may sometimes appear in the area outside the currently displayed modal window frame. To retrieve these messages/errors it is necessary to retrieve the entire screen content rather than just the modal window content. This can be achieved by writing the following method:

Java

```java
private String getMessgeOutsideWindow(int MessageRow, int MessageCol){
    try {
            // Create a Screen Request the would ignore modal window definition
            GXGetScreenRequest gsr = new GXGetScreenRequest();
            gsr.addVariable(GXBaseObjectConstants.GX_VAR_HOST_WINDOW_ENABLED,"false");

            // Get the current screen from ApplinX server using the Screen request
            GXIScreen screen = getGXSession().getScreen(gsr);

            // Return Field content according to the specified position
            GXPosition pos = new GXPosition(MessageRow, MessageCol);
            return screen.getFields().getFieldByPosition(pos).getContent();

    } catch (GXGeneralException e) {
            return null;
    }
}
```

.NET

```csharp
private string getMessgeOutsideWindow(int MessageRow, int MessageCol)
{

    // Create a Screen Request the would ignore modal window definition
    GXGetScreenRequest gsr = new GXGetScreenRequest();
    gsr.addVariable( GXBaseObjectConstants.GX_VAR_HOST_WINDOW_ENABLED,  "false");

    // Get the current screen from ApplinX server using the Screen request
    GXIScreen screen = gx_session.getScreen(gsr);

    // Return Field content according to the specified position
    GXPosition pos = new GXPosition(MessageRow, MessageCol);
    return screen.getFields().getFieldByPosition(pos).getContent();
}
```

Note:
This method can be added to `GXBasicContext` or to a generated page's code class.

ApplinX Server Load Balancing

When you have several ApplinX servers and want new sessions to always be created on the least busiest server, determine this using the GXLoadBalancer class.

▶

1. From within the relevant application in the new JSP directory, open
   WEB-INF\classes\contexts\GXBasicContext.

2. Add the necessary statements:

   JSP

   ```
   import com.sabratec.ci.GXLoadBalancer
   ```

   .NET

   ```
   using com.sabratec.dotnet.ci;
   ```

3. Within the file, locate the `gx_initSessionConfig` method and place the following code:

   ```
   /// ApplinX Server Load Balancing   //////////////////////////////////
           GXWebAppConfig gx_appConfig = getGXAppConfig();

           GXLoadBalancer lb = new GXLoadBalancer();
           // Add server URLs. Example:
           // lb.addServerURL("applinx://localhost:2323");
           lb.addServerURL("applinx://<ApplinXServer1>:<portNumber>");
           lb.addServerURL("applinx://<ApplinXServer2>:<portNumber>");
           lb.addServerURL("applinx://<ApplinXServer3>:<portNumber>");
           lb.addServerURL("applinx://<ApplinXServer4>:<portNumber>");

   // Replace ApplinXServerX with the server name
   // Replace portNumber with the appropriate ApplinX server port number.

           String availableServer = lb.getAvailableServer();
           gx_appConfig.getSessionConfig().setServerURL(availableServer);
   ```

Retrieving a Unique Device Name per User Name / IP Address from a Database and Applying it to the Current User Session

Some hosts require that every user connection must supply a device name in order to establish a connection. These device names may be unique per user or per IP address. In such cases, you will need to retrieve the device name from the place where your organization stores it (database, LDAP, XML etc.).

The following example will demonstrate how to retrieve the device name from the database (MDB) using the user's IP address and how to set the retrieved value to the ApplinX session, prior to connecting to the host.

JSP

Add the following method to *GXBasicContext.java*:

```java
// This method retrieves the device name from an MDB database.
// You can change this code to match your type of database.
public String getDevice(String userIP) {
  String Result = "";
  try {
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     // set this to a MS Access DB you have on your machine
     String filename = "C:\\users.mdb";
     String database = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=";

     database+= filename.trim() + ";DriverID=22;READONLY=true}";    // add on to the end
     // now we can get the connection from the DriverManager
     System.out.println(database);
     Connection con = DriverManager.getConnection(database , "", "");
     Statement stmt = con.createStatement();
     ResultSet rs;

     // Retrive Device name according to user's IP address
     rs = stmt.executeQuery("Select Device from userDevices where userIPAddr='" + userIP + "'");

     while ( rs.next() ) {
         Result = rs.getString("Device");
     }
     con.close();
  } catch (Exception e) {
        System.err.println(e.getMessage());
  }
  return Result;
}
```

In the this file, edit the `gx_initSessionConfig` method as follows:

```java
public void gx_initSessionConfig(){
   // for device name
   GXWebAppConfig gx_appConfig = getGXAppConfig();
   String ipAddr = getRequest().getRemoteAddr();
   gx_appConfig.getSessionConfig().addVariable(
        GXBaseObjectConstants.GX_VAR_DEVICE_NAME, getDevice(ipAddr));
}
```

.NET

Add the following method to *GXBasicWebForm.cs*:

```csharp
private string getDevice(string userIP)
{
   OdbcConnection con = new OdbcConnection();
   con.ConnectionString = "Driver={Microsoft Access Driver (*.mdb)};Dbq=c:\\users.mdb;Uid=sa;Pwd=sa;";
   con.Open();
   string result="";
   try
   {
      // Retrive Device name according to user's IP address
      OdbcCommand cmd = new OdbcCommand("Select Device from userDevices where IP='" + userIP + "'", con); // creating query command
      OdbcDataReader reader = cmd.ExecuteReader(); // executes query
      while (reader.Read()) // if can read row from database
      {
         result = reader["Device"].ToString();
      }
   }
   catch (Exception ex)
   {
      // shows exception message in console if any errors occured
      Console.WriteLine("Ex: " + ex.Message);
   }
   finally
```

```
   {
      con.Close();  // finally closes connection
   }
   return result;
}
```

In the this file, edit the `gx_initSessionConfig` method as follows:

```
public void gx_initSessionConfig(){
   // for device name
   String ipAddr = Request.UserHostName;
   gx_appConfig.getSessionConfig().addVariable(
         GXBaseObjectConstants.GX_VAR_DEVICE_NAME, getDevice(ipAddr));
}
```

Retrieving the Host Printer Device Name from a Database and Setting the ApplinX Printlet to Work with that Device

Connecting to a host printer queue requires users to supply a device name. The device provides the host information as to which printer queue to associate each user. Usually, each user or IP address has a pre-defined device name. This data can be stored in a database and retrieved before creating the printer session in the ApplinX web application.

The following example will demonstrate how to retrieve the device name from the database and set it to the appropriate printlet parameter.

JSP

Replace the `getDevice` method in *RunPrintlet.java*:

```
// This method retrieves the device name from an MDB database.
// You can change this code to match your type of database.
public String getDevice(String userIP) {
   String Result = "";
   try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      // set this to a MS Access DB you have on your machine
      String filename = "C:\\users.mdb";
      String database = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=";

      database+= filename.trim() + ";DriverID=22;READONLY=true}";    // add on to the end
      // now we can get the connection from the DriverManager
      System.out.println(database);
      Connection con = DriverManager.getConnection(database , "", "");
      // String url = "jdbc:odbc:users.mdb";
      // Connection conn = DriverManager.getConnection(url,"","");
      Statement stmt = con.createStatement();
      ResultSet rs;

      rs = stmt.executeQuery("Select Device from userDevices where userIPAddr='" + userIP + "'");

      while ( rs.next() ) {
         Result = rs.getString("Device");
         System.err.println(Result);
      }
      con.close();
   } catch (Exception e) {
         System.err.println(e.getMessage());
   }
   return Result;
}
```

In *run_Printlet.jsp* set the device name parameter as follows:

```
<param name="device_name" value="<%=new RunPrintlet().getDevice(request.getRemoteAddr()) %>">
```

.NET

Replace the `getDevice` method in *run_printlet.aspx.cs*:

```
private string getDevice(string userIP)
{
    OdbcConnection con = new OdbcConnection();
    con.ConnectionString = "Driver={Microsoft Access Driver (*.mdb)};Dbq=c:\\users.mdb;Uid=sa;Pwd=sa;";
    con.Open();
    string result="";
    try
    {
        // Retrive Device name according to user's IP address
        OdbcCommand cmd = new OdbcCommand("Select Device from userDevices where IP='" + userIP + "'", con); // creating query command
        OdbcDataReader reader = cmd.ExecuteReader(); // executes query
        while (reader.Read()) // if can read row from database
        {
            result = reader["Device"].ToString();
        }
    }
    catch (Exception ex)
    {
        // shows exception message in console if any errors occured
        Console.WriteLine("Ex: " + ex.Message);
    }
    finally
    {
        con.Close();  // finally closes connection
    }
    return result;
}
```

In *run_printlet.apx.cs* edit the `Page_Load` method and add the following assignment:

```
// Retrieve Device name By User's IP
GXPrinterControl1.deviceName = getDevice(Request.UserHostName);
```

Notes:

1. You can implement the same methodology for of the printlet parameters.
2. If ApplinX server and the web application's server do not reside on the same machine, refer to Configuring the Printlet Redirector

Customize ApplinX Framework Session Error Handling

The default behavior of the ApplinX Framework when it encounters an ApplinX session error is to redirect the user to the *error.jsp* page. This page will specify the error code and will also provide a description of the error. For example: If, for some reason, the ApplinX session has been disconnected, pressing a PF key in the web application will result in the following error:

ApplinX Error in Application *instantdemo*

Error Number: **5001**

Error Message: **User is not connected**

- Click <u>Here</u> to try again.

- Click <u>Here</u> to disconnect.

- If the problem persists check your ApplinX Server status and configuration.

However, in some cases you may want to capture these errors and perform a different action. The following example will demonstrate how the default ApplinX method can be overridden with new functionality.

For example, upon receiving a 5001 error, you may want the web application to automatically try to re-establish a connection to the host.

JSP

To do this, add the following code to the GXBasicContext class:

```
// Override the gx_handleSessionError with additional functionality
public void gx_handleSessionError(GXGeneralException ex){
        try{
                // Is the error code 5001?
                if (ex.getErrorCode()== 5001){
                        // Try to re-establish a connection to the host
                        getResponse().sendRedirect("gxfirstpage.jsp");
                }
                else{   // Otherwise, perform the default behavior
                        super.gx_handleSessionError(ex);
                }
        }
        catch (IOException e){
                // TODO : Handle Failed page redirection
        }
}
```

.NET

To do this, add the following code to the GXBasicContext class:

```
public override void gx_handleSessionError(com.sabratec.applinx.baseobject.GXGeneralException err)
{
   if (err.getErrorCode() == 5001)
   {
      // Try to re-establish a connection to the host
      Response.Redirect("gxfirstpage.aspx");
   }
   else
   {    // Otherwise, perform the default behavior
      base.gx_handleSessionError(err);
   }
}
```

Customize the Web Application's Error Page

One may want to customize the Web application's error page for any number of reasons. These reasons may include providing the error page in non-English languages, or to change the look and feel of the page or add additional messages.

The look and feel of the page can easily be customized by changing the layout of the error.jsp page. Changing the page's dynamic content can be achieved by editing the *error.java* file.

The *error.java/Error.apsx.cs/vb* file contains an example for translating a specific error message to another language, according to the user's locale setting. In order to activate this functionality un-comment the CustomizeErrorHandling method and un-comment the call to CustomizeErrorHandling in gx_onLoad in the *error.java* file (JSP) or Page_Load in the error.aspx.cs file (.Net).